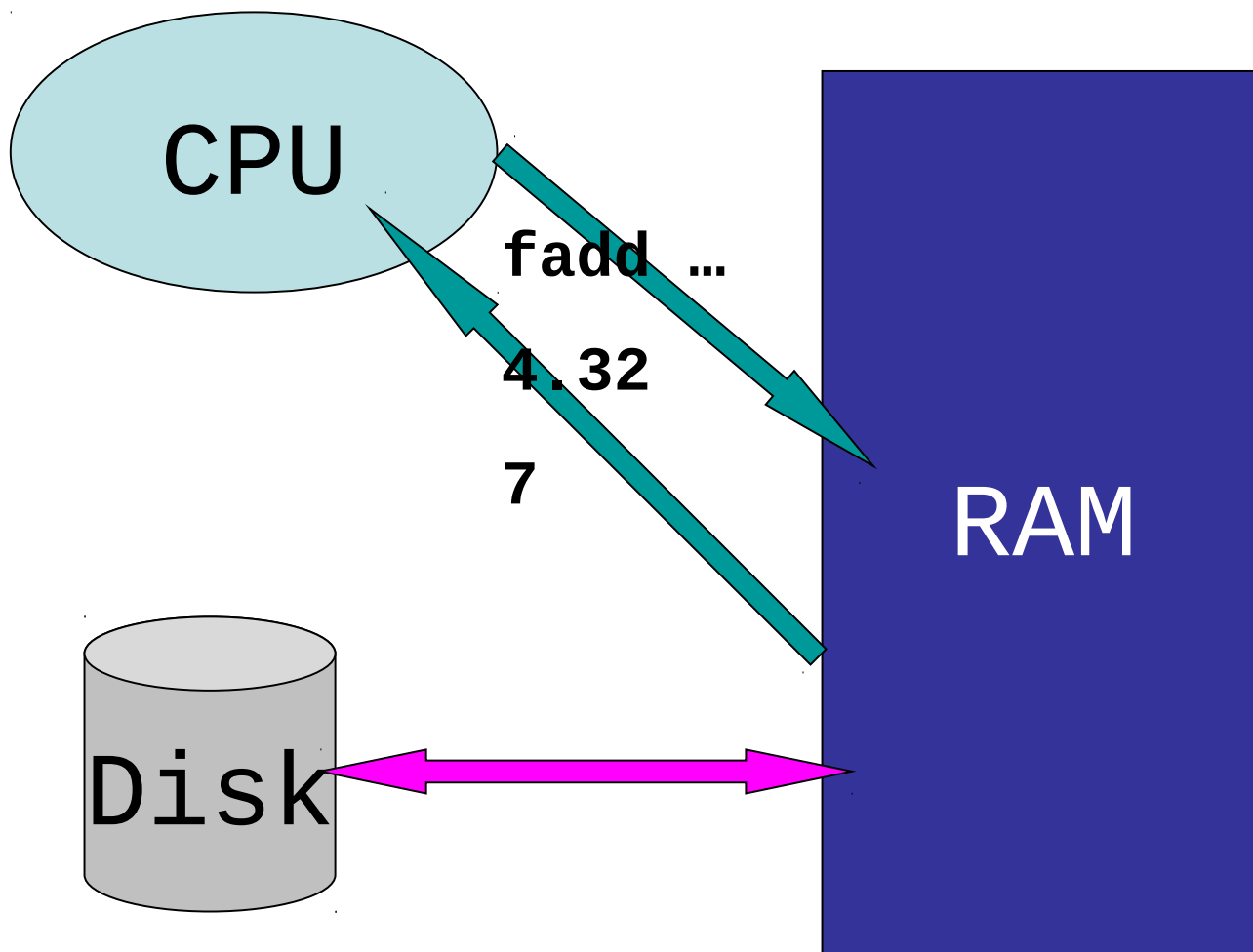
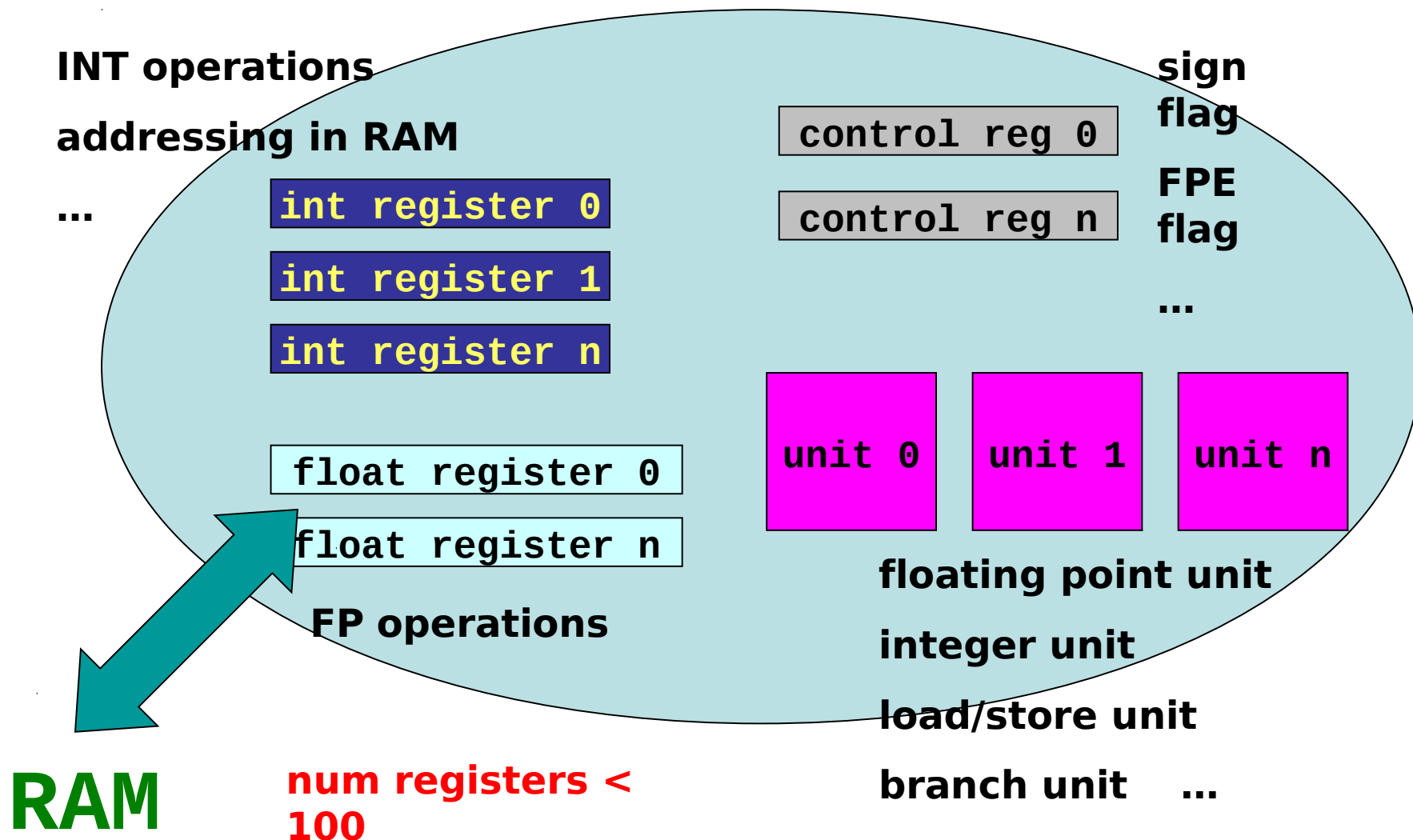


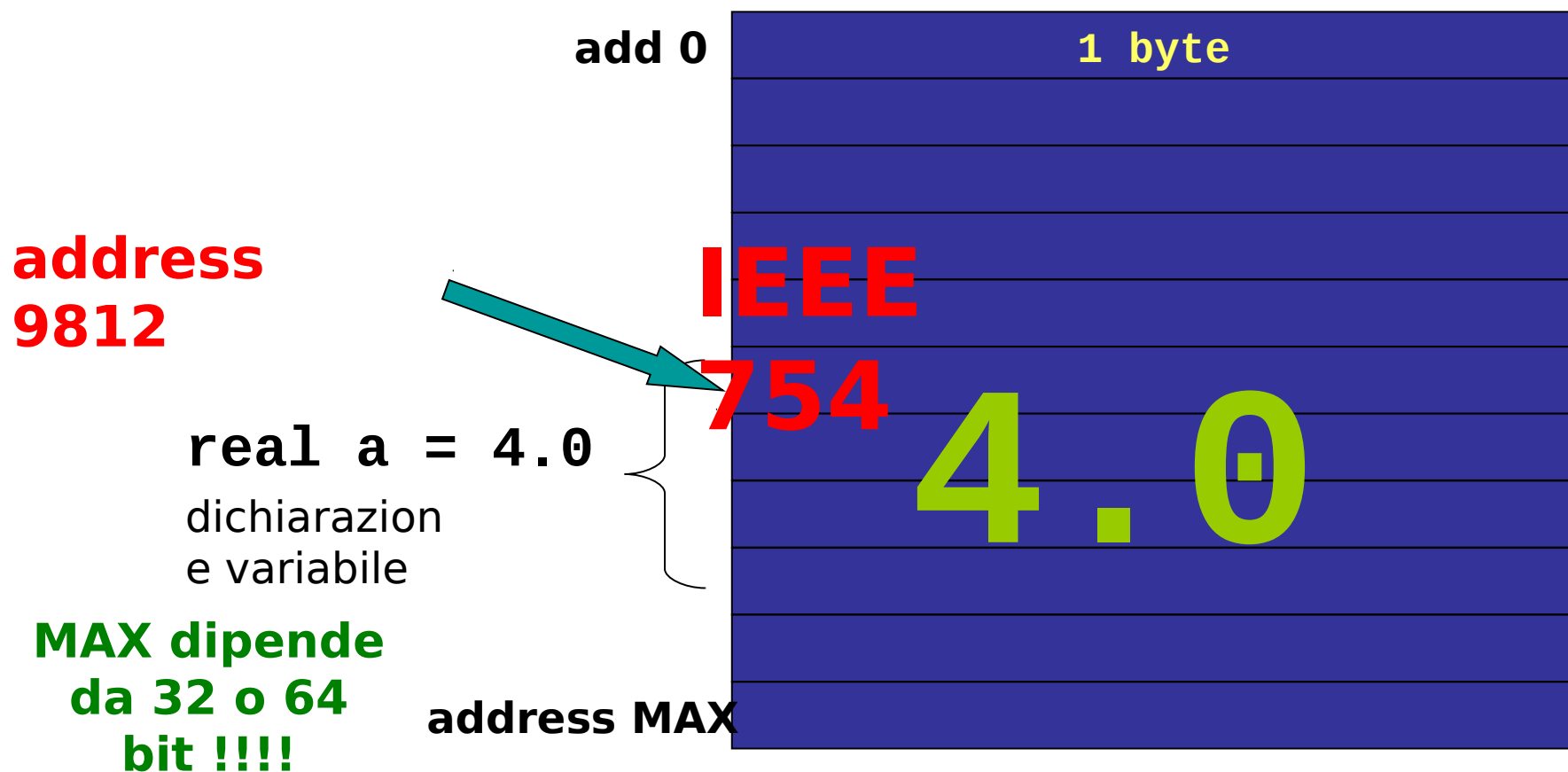
Architettura dei calcolatori

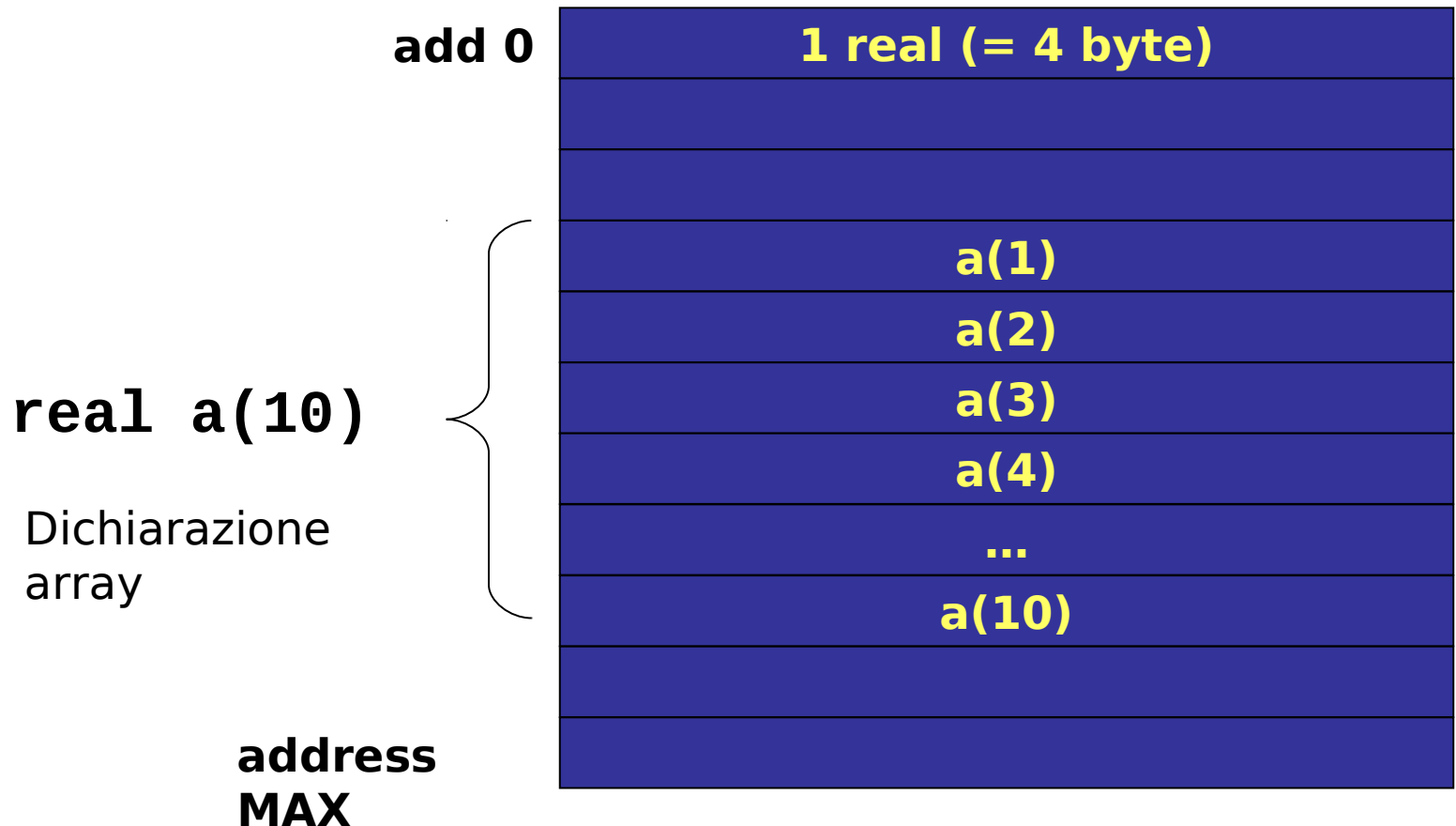


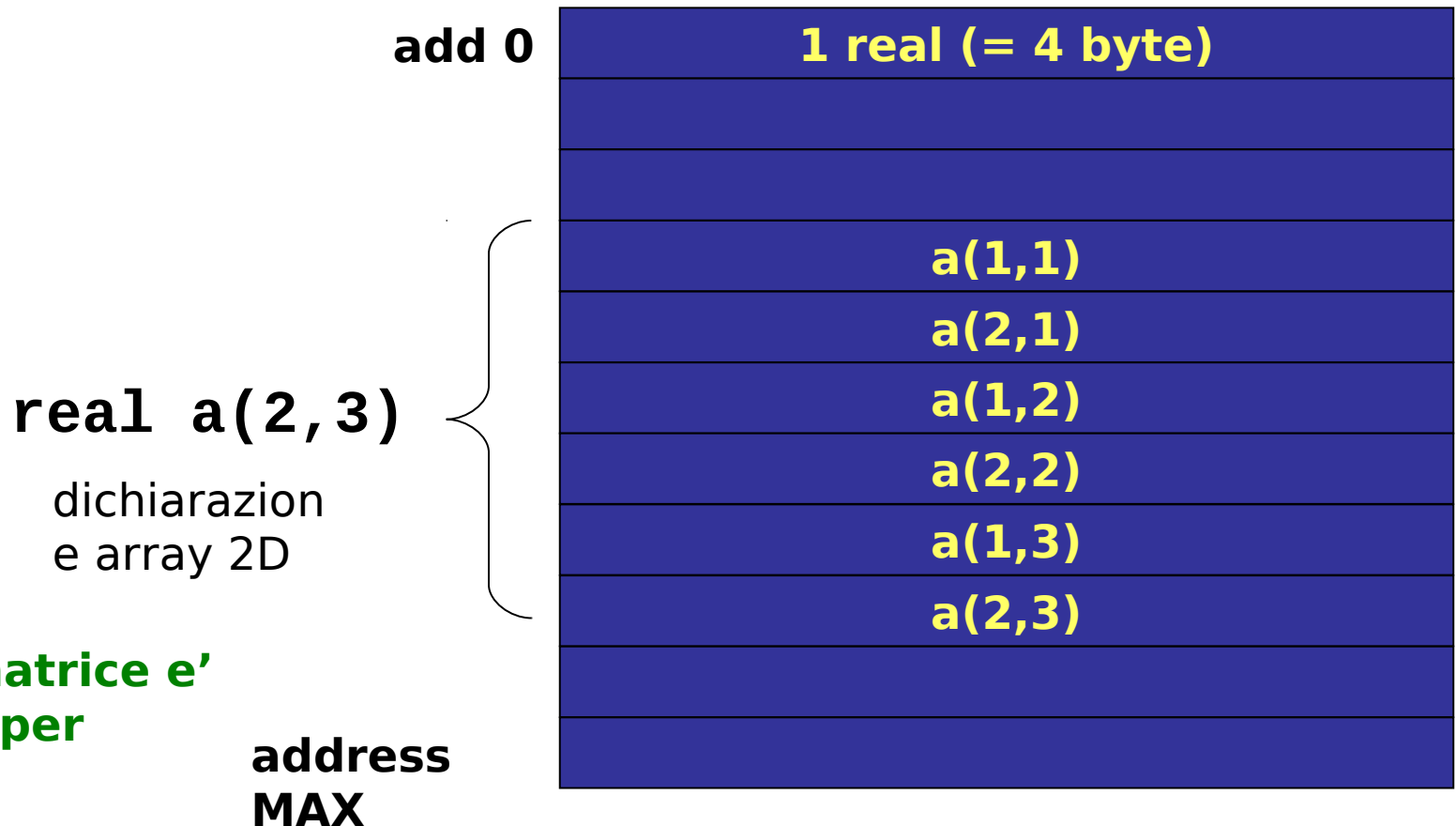
Gabriele Fatigati - [g.fatigati@Cineca.it](mailto:g.fatigati@ Cineca.it)
Gruppo Supercalcolo - Dipartimento Sistemi e Tecnologie











**In C la matrice e'
allocata per
righe !**

esecuzione
istruzioni

```
do i=1, n
    a(i) = b(i) + c(i)
enddo
```

Human

generalmente
fra registri

store 1, i

load i, ir0

load n, ir1

START: load b(i), fr0

load c(i), fr1

fadd fr0, fr1 → fr2

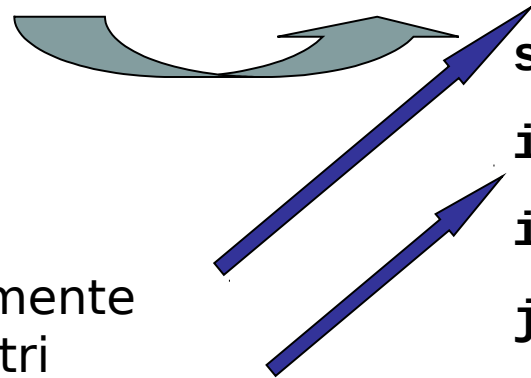
store fr2, a(i)

iadd 1, ir0 → ir0

icmp ir0, ir1 → cr0

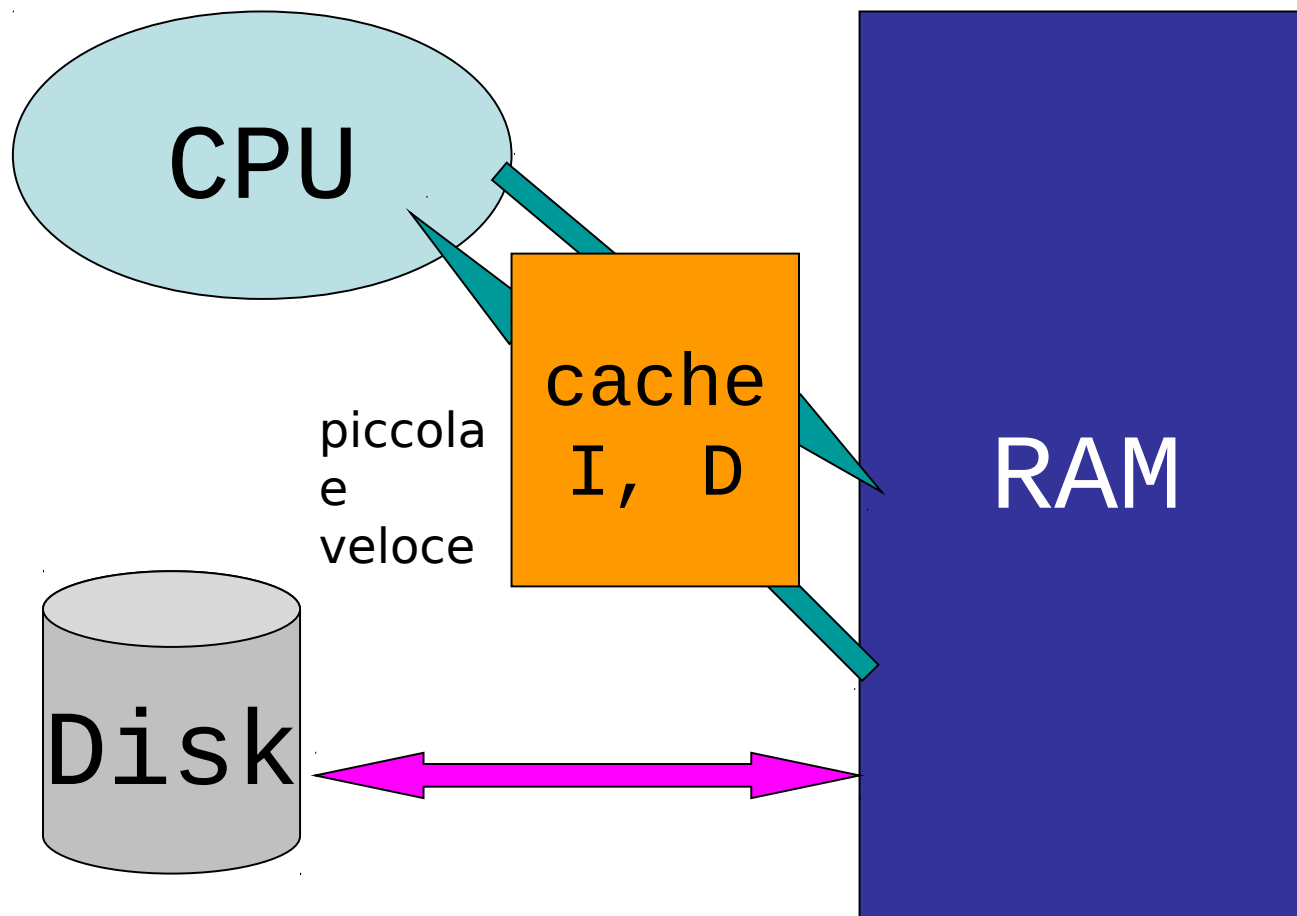
jmp cr0 → **START**

CPU



Per una buona esecuzione

- veloce accesso ai dati in memoria
 - problema: tempi accesso dati in RAM molto grandi rispetto clock CPU
- veloce esecuzione delle istruzioni
 - scelta istruzioni light
 - buon scheduling istruzioni



Organizzate per livelli e per istruzioni e/o dati

Registro → L1 → L2 → L3 → RAM

Size aumenta L1 → → L_n

Velocità diminuisce L1 → → L_n

Quando la CPU ha bisogno di un dato cerca questo in L1: se qui è presente lo carica in un registro (L1 cache hit), altrimenti (L1 cache miss) lo cerca in L2...

Cache miss: penalità (in cicli di clock) nel caricare il dato nei registri (può causare uno stallo se il processore non ha niente altro da eseguire)

Cache hit: il dato è presente in cache, nessuna penalità

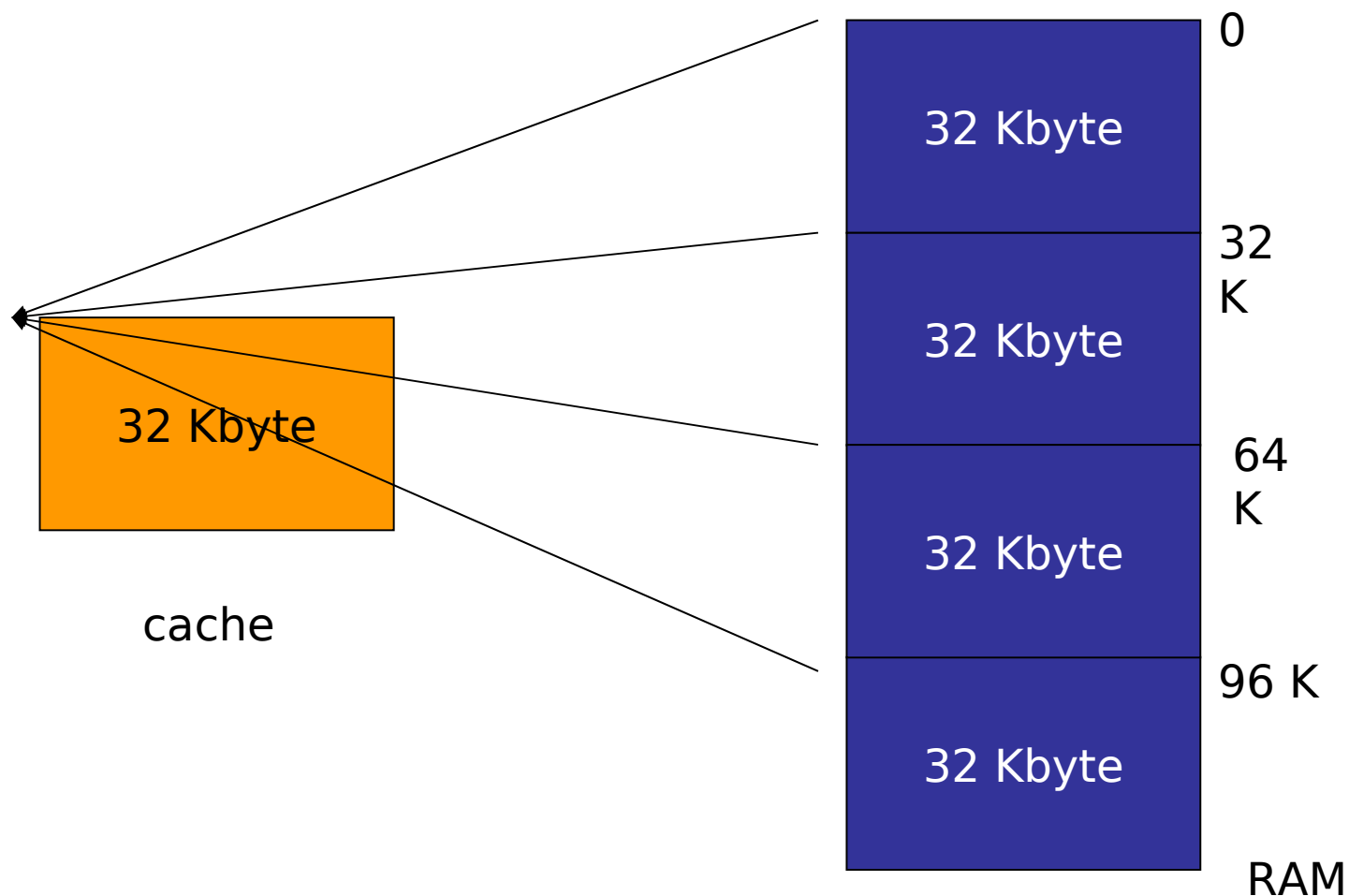
LOAD a RAM \rightarrow register

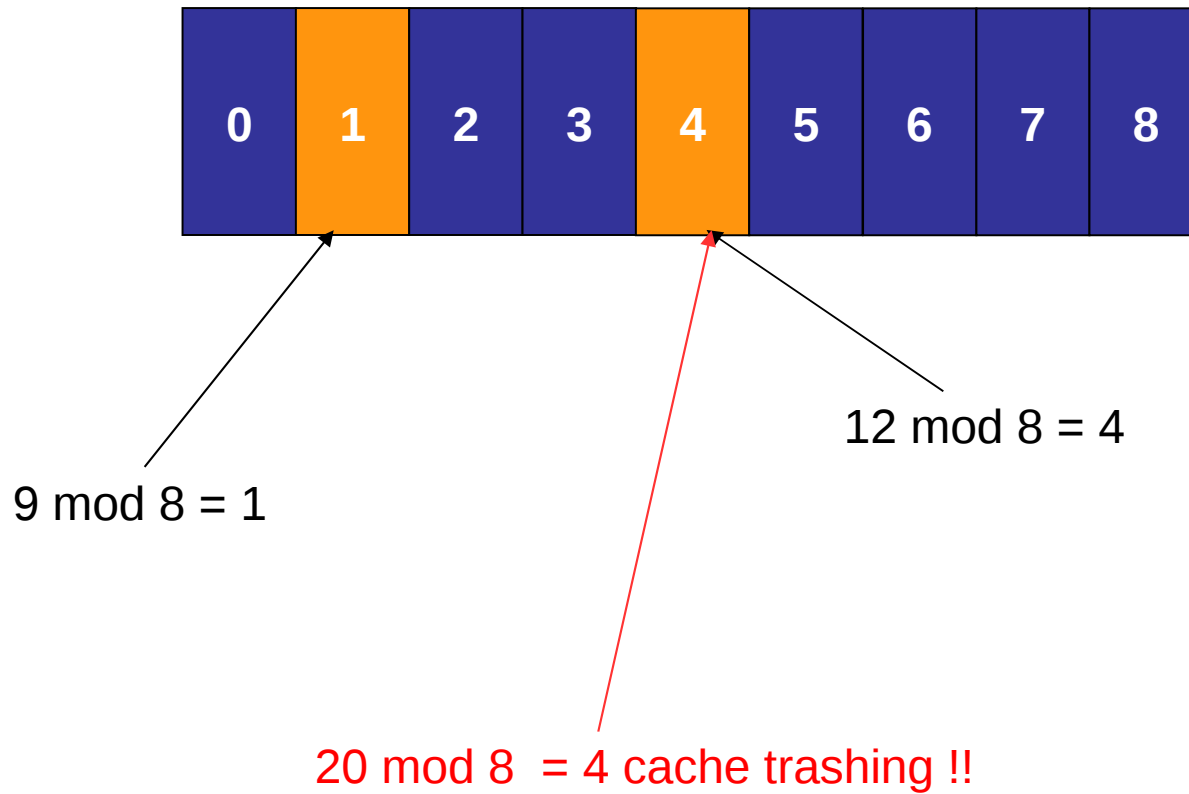
- Viene caricata la linea di cache L3 che contiene a da RAM \rightarrow L3 \rightarrow **linea di cache L3 “dato atomico” in lettura**
- Viene caricata la linea di cache L2 che contiene a da L2 \rightarrow L1
- Viene caricata la linea di cache L1 che contiene a da L1 \rightarrow L2
- Viene caricato il dato a da L1 in un registro

STORE a register \rightarrow RAM (es un solo livello di cache L1)

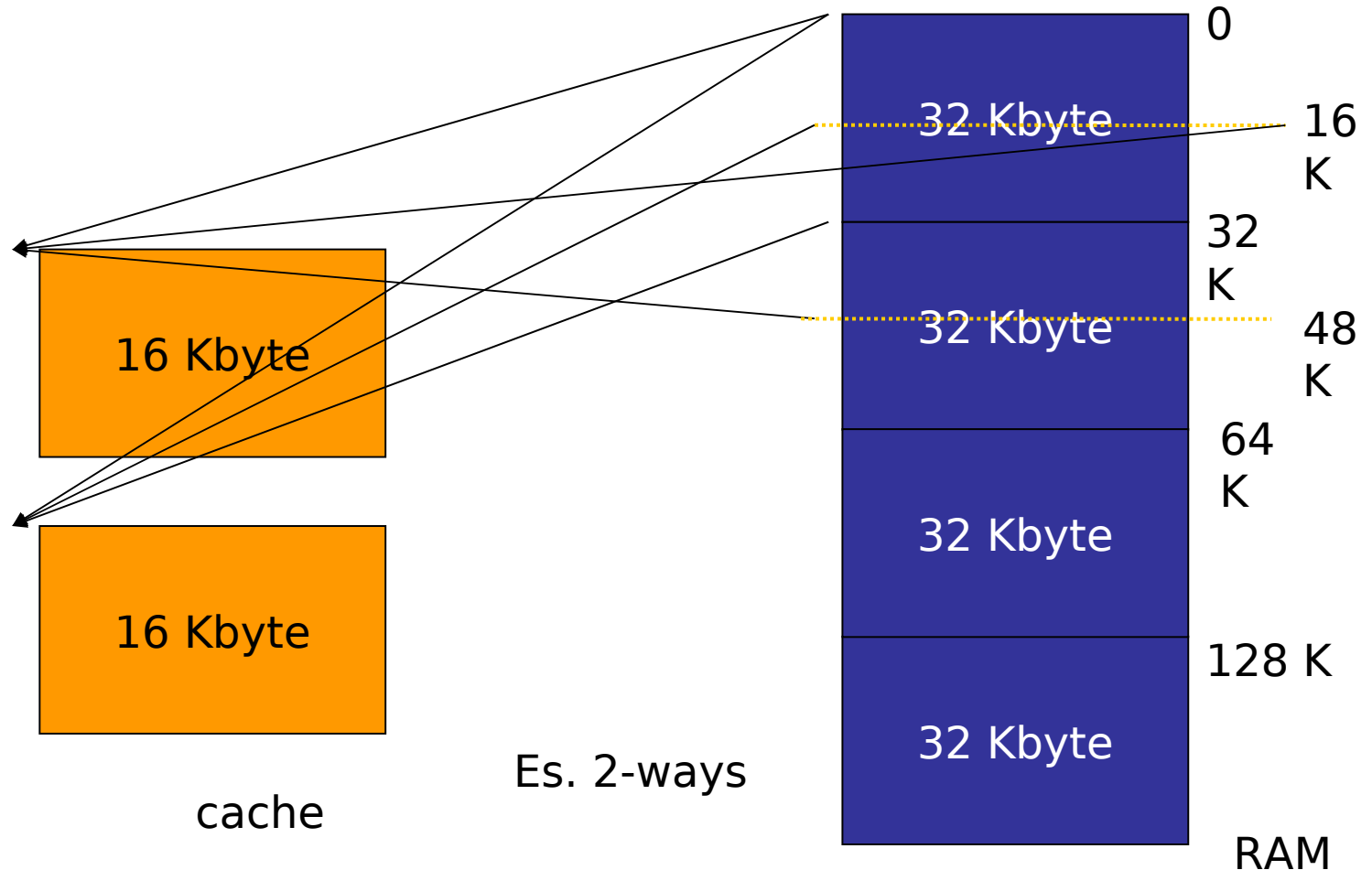
- registro \rightarrow linea L1 e // [registro o linea L1] \rightarrow RAM (cache **write through**) oppure
- registro \rightarrow linea L1 (cache **write back**); il dato verra' scritto in RAM solo quando necessario (quando la linea di cache dovra' essere rimpiazzata)
 \rightarrow meno transazioni (store) cache \rightarrow RAM

Cache direct mapped

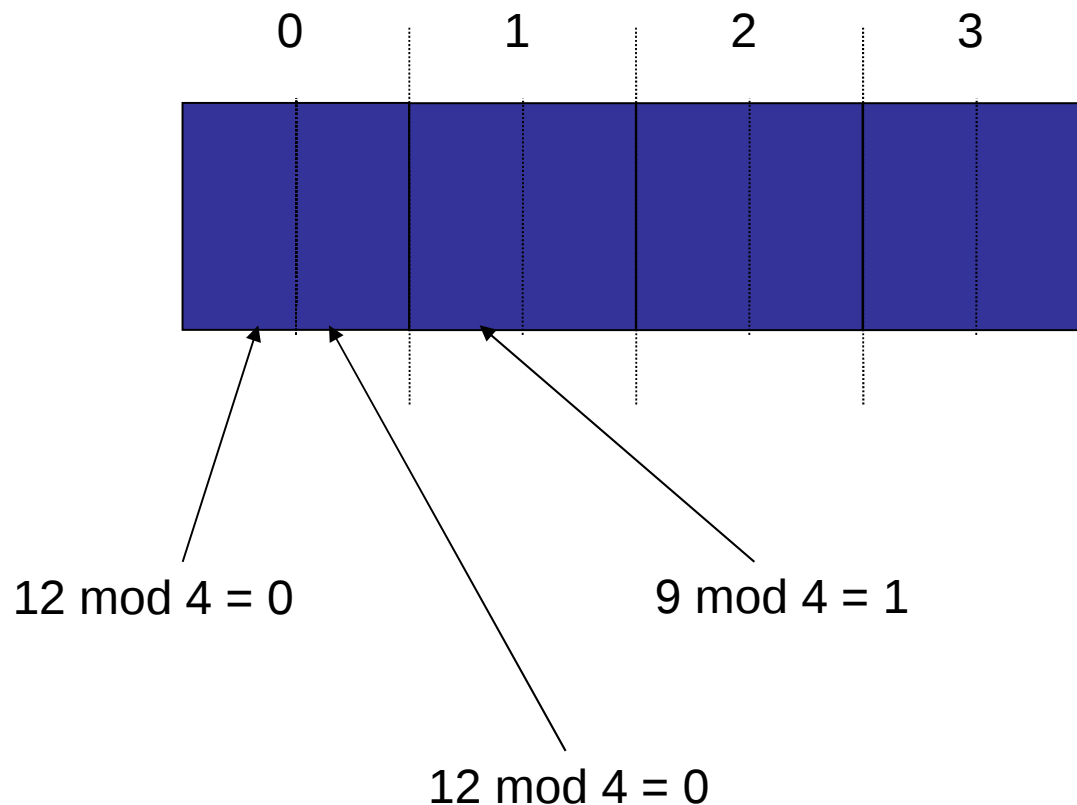




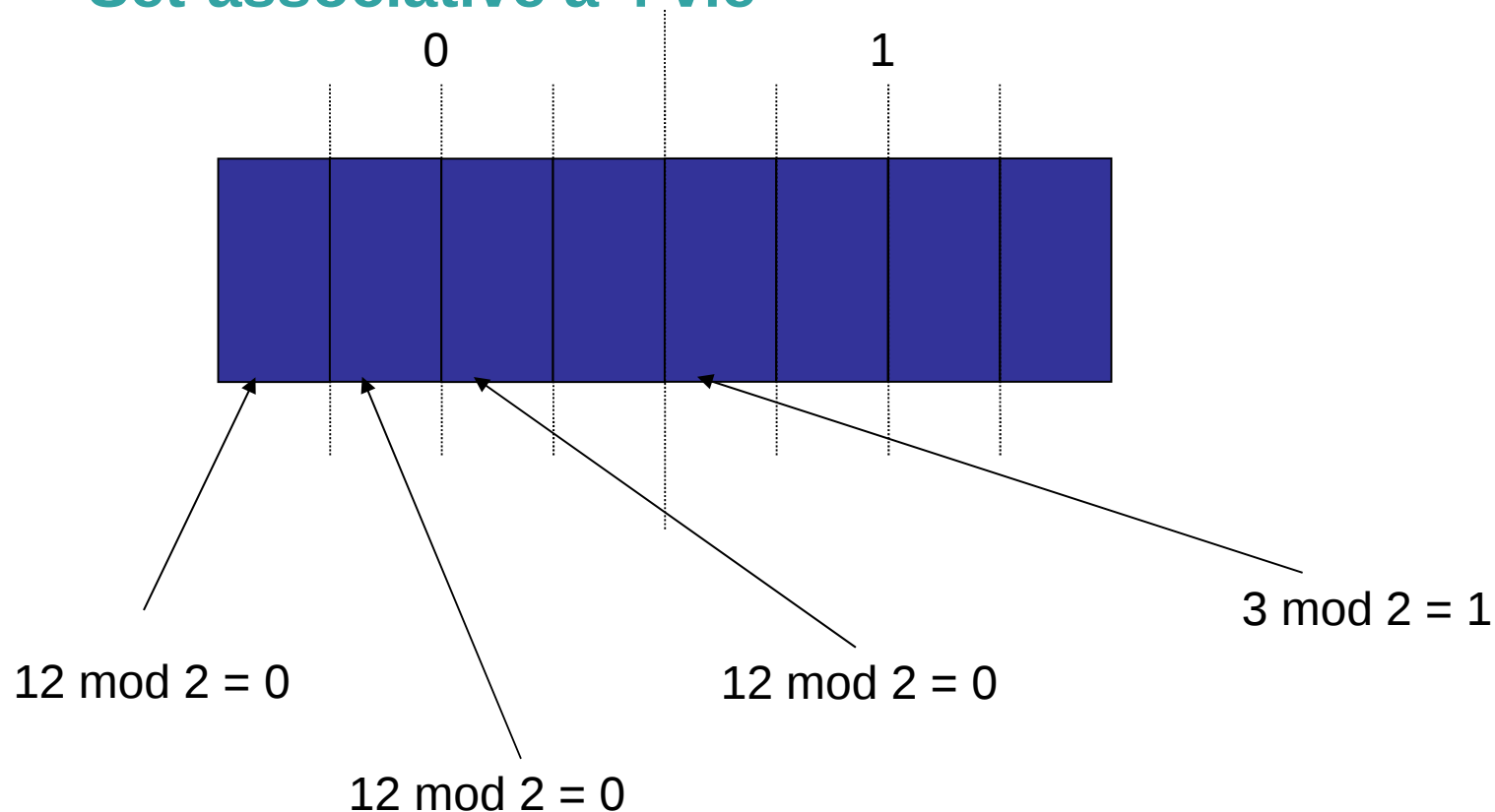
Cache set-associative



Set-associative a 2 vie



Set-associative a 4 vie



Politiche di rimpiazzamento dei dati

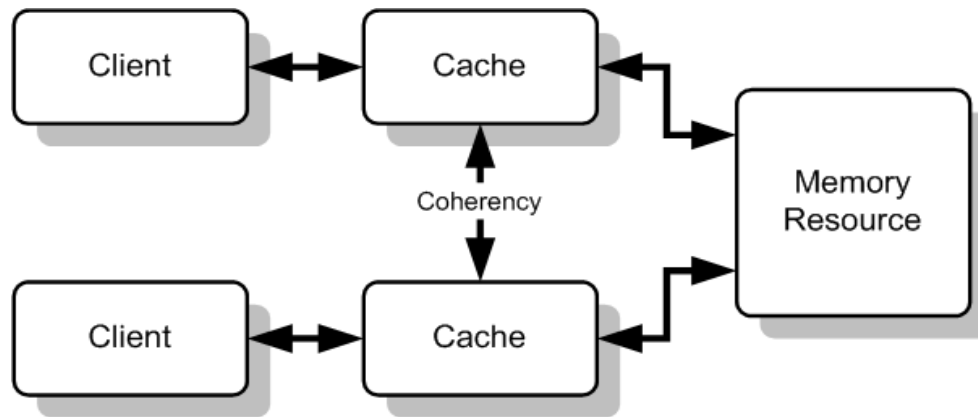
- **Least recently used:** viene rimpiazzato l'elemento utilizzato meno di recente. Utilizzo di “age-bits”, per stabilire il meno recente.
- **Round robin:** rimpiazzamento mediante ordine circolare
- **Random:** Il più semplice, non richiede informazioni aggiuntive

Politiche di scrittura

Quando un dato viene scritto, il valore in cache non è allineato con il valore in RAM. Prima di rileggere tale valore, è necessario aggiornarlo, per evitare di leggere un valore vecchio.

Write-through: Ogni scrittura in cache comporta una scrittura sincrona con la RAM. Dati sempre aggiornati ma molte scritture effettuate

Write-back: Ogni scrittura è posticipata, contrassegnandola con un “dirty-bit”. Meno scritture, ma problema di cache coherency.



Cosa succede se due client scrivono la stessa variabile?

La coerenza definisce il comportamento di lettori, scrittori nella stessa locazione di memoria. Si raggiunge se:

Una lettura fatta da P su X che segue una scrittura di P su X, senza altre scritture intermedie da altri attori, X deve ritornare sempre il valore scritto da P

Una lettura fatta da P1 su X che segue una scrittura di P2 su X, deve ritornare sempre la scrittura fatta da P2, se non vi sono altri scrittori tra i due accessi. Se un processore può leggere lo stesso vecchio valore dopo la scrittura di P2, la memoria è **incoerente**

Una scrittura nella stessa locazione deve essere sequenziale. Se X riceve due differenti valori A e B, in questo ordine da due processori, il processore non può mai leggere X come B e quindi come A. X deve essere visto come A e B in quest'ordine.

Si suppone che le letture e scritture siano istantanee. In realtà non è così, in generale una scrittura è più lenta di una lettura, senza considerare le varie latenze. Un modello di coerenza deve tenere conto di questo aspetto.

SVANTAGGIO CACHE

- Se si elaborano sempre dati diversi (ogni dato e' utilizzato solo una volta) → cache inserisce solo overhead

VANTAGGIO CACHE

- Elaborazione ripetuta sugli stessi dati (localita' temporale)
- Elaborazione di dati vicini in memoria (localita' spaziale)

**Loop
Fusion**

```
do i=1, n
  a(i) = b(i) + 1.0
enddo
do i=2, n
  c(i) = sqrt(a(i-1))
enddo
```

Se n grande o se faccio del lavoro fra i due loop butto via a che devo poi ricaricare in cache

```
a(1) = b(1) + 1.0
do i=2, n
  a(i) = b(i) + 1.0
  c(i) = sqrt(a(i-1))
enddo
```

Appena carico $a(i)$ in cache lo utilizzo il + possibile di volte

n.b. Attenti a $\text{sqrt}(a(i+1))$ e ai bounds degli array

Loop Interchange

```
do i=1, n
  do j=1, n
    a(i,j) = b(i,j) + 1.0
  enddo
enddo
```

```
do j=1, n
  do i=1, n
    a(i,j) = b(i,j) + 1.0
  enddo
enddo
```

**Minimizzare
sempre lo stride
!!!!**

Una volta caricata una linea di cache non utilizzo altri elementi di a che probabilmente dovrò poi ricaricare perché buttati via

Utilizzo subito tutti gli elementi delle linee di cache

n.b. In C accedere alle matrici per righe


```
integer x(n) = ...  
do i=1, n  
    a(x(i)) = b(x(i)) + 1.0  
enddo
```

**Removing
indirect
addressing**

■ ■ ■

Una volta caricata una linea di cache rischio di non utilizzare gli altri elementi di a che probabilmente dovrò poi ricaricare perché buttati via

Se possibile conviene ristrutturare l'algoritmo, l'indirizzamento indiretto è la morte delle performance

```
real, dimension (1024) :: a,b  
COMMON /my_com/ a, b  
do i=1, 1024  
    a(i) = b(i) + 1.0  
enddo
```

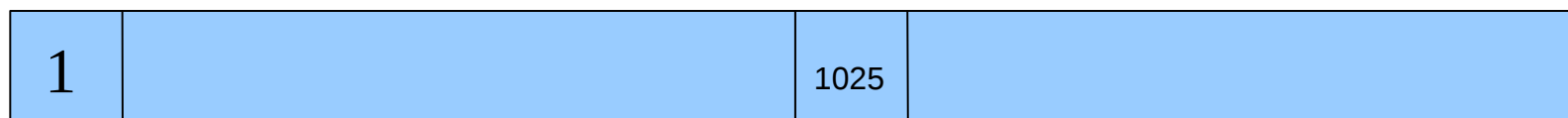
Se size cache = 4×1024 , direct mapped, a,b contigue (es):
cache thrashing (carico e scarico ogni volta le stesse linee in cache)

array di size = multiplo cache size -> possono originare cache thrashing

Set Associative: riduzione problema

a(1)

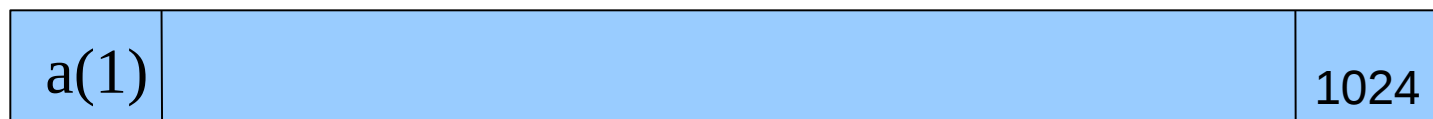
b(1)



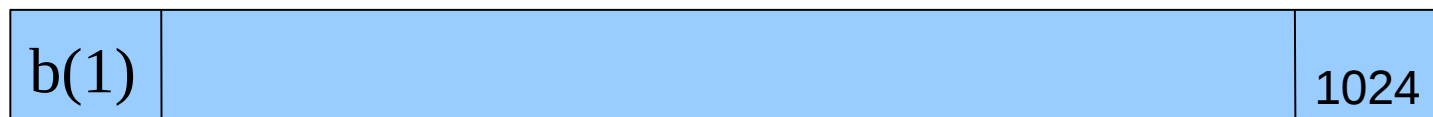
$1 \bmod 1024 = 1$

$1025 \bmod 1024 = 1$

Nella cache:



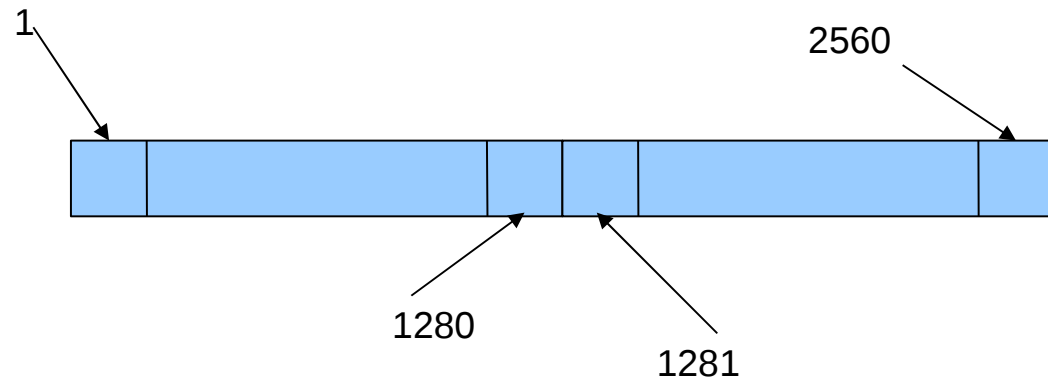
trashing



```
integer offset =
(linea_cache)/SIZE(REAL)
real, dimension
(1024+offset) :: a,b
COMMON /my_com/ a, b
do i=1, 1024
    a(i) = b(i) + 1.0
enddo
```

offset → sfalzo le matrici
in cache → no more
problems

→ **NON** usare multipli di
2 per dimensioni matrici



$$1 \bmod 1024 = 1$$

$$1281 \bmod 1024 = 257$$

**tmp arrays
elimination**

```
do i=1, n
    squareb(i) = b(i)*b(i)
enddo

...

do i=2, n-1
    a(i-1) = squareb(i) + c(i+1)
            - d(j)
enddo
```

Se utilizzo array per mantenere solo valori intermedi questi generano un traffico inutile fra il processore e la memoria. Eventuali invarianti in inner loop conviene memorizzarli esplicitamente in scalari (prob così verranno allocati in un registro)

```
dd = d(j)

...

do i=1, n-2
    a(i) = b(i+1)*b(i+1) +
          c(i+2)
          - dd
enddo
```

Minor traffico verso la memoria ma peggiore leggibilità

Alcuni processori permettono il prefetch dei dati

prefetch → caricamento in cache dei dati dalla memoria che verranno utilizzati nel seguito → in questo modo quando questi dati serviranno sono già pronti e non si ha la penalità dei cache misses

– es.

```
real*4 a
```

```
do i=1, n
```

```
    a(i) = a(i) + 1.0
```

```
enddo
```



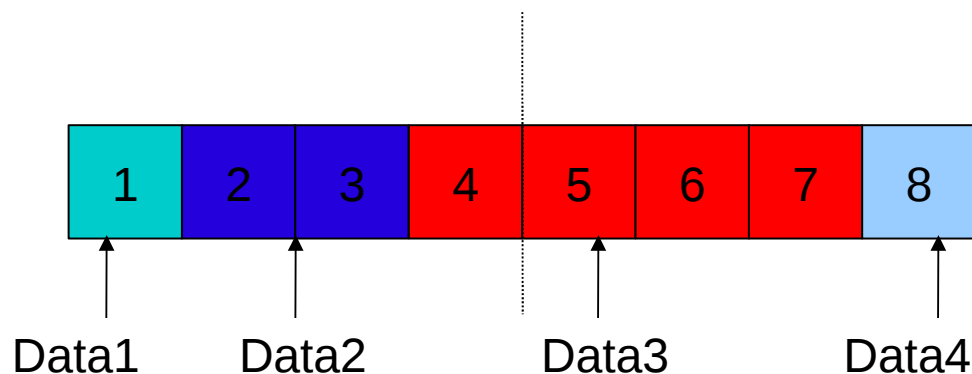
prefetch a(i+linea_cache/4)

prefetch: automatico (hardware) o scatenato da istruzioni (software)

Quando si parla di cache, è fondamentale che i dati siano allineati in memoria.

Si supponga di leggere a colpi di 4 bytes (1 WORD) (sistemi a 32 bit). Gli indirizzi in memoria per essere allineati devono essere multipli di 4.

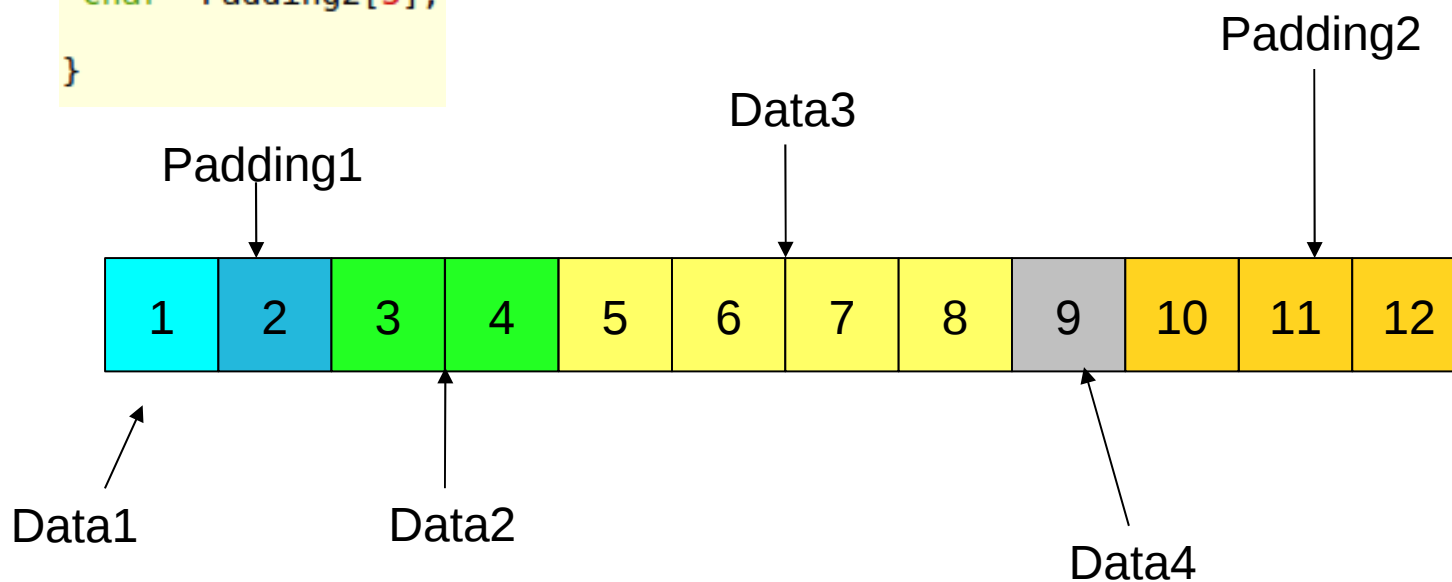
```
struct MixedData{  
    char  Data1;  
    short Data2;  
    int   Data3;  
    char  Data4;  
}
```



Per leggere Data3 servono due letture dalla memoria

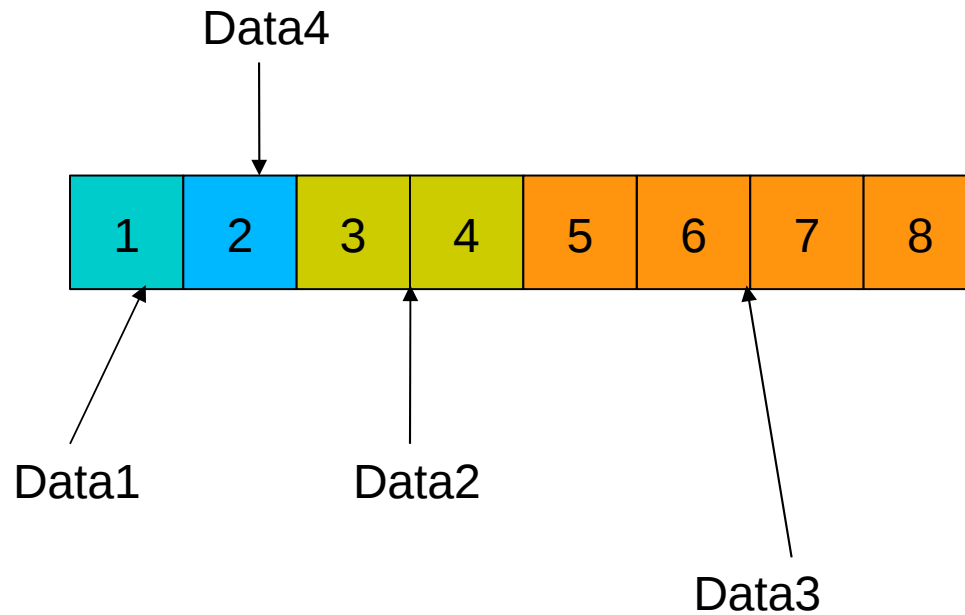
Con allineamento:

```
struct MixedData{
    char  Data1;
    char  Padding1[1];
    short Data2;
    int   Data3;
    char  Data4;
    char  Padding2[3];
}
```



La struttura allineata risultante occupa più spazio: 12 bytes anziché gli 8 di quella originale. E' possibile allineare i partenza scambiando l'ordine dei campi:

```
struct MixedData{  
    char  Data1;  
    char  Data4;  
    short Data2;  
    int   Data3;  
}
```



In generale:

$\text{padding} = \text{align} - (\text{offset} \bmod \text{align})$

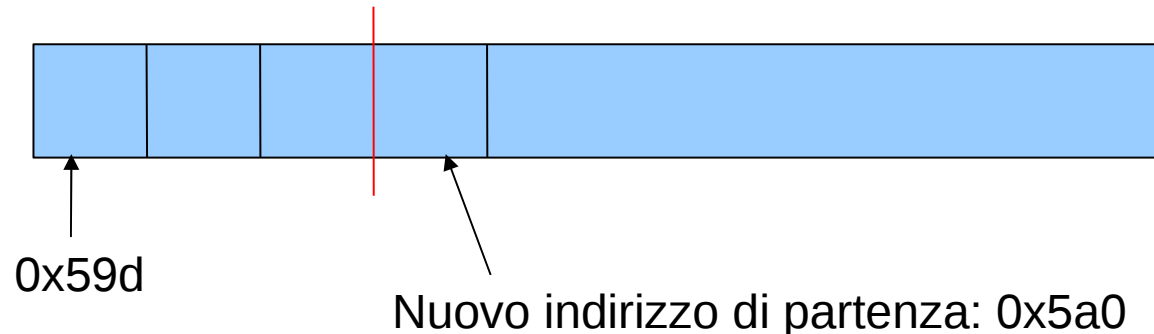
$\text{newoffset} = \text{offset} + \text{padding} = \text{offset} + \text{align} - (\text{offset} \bmod \text{align})$

Esempio:

$\text{offset} = 0x59d$, $\text{align} = 4$

$\text{padding} = 4 - (0x59d \% 4) = 4 - 1 = 3$

$\text{newoffset} = 0x59d + 3 = 0x59a0$



I moderni sistemi di calcolo implementano un sistema di **memoria virtuale**, che permette di indirizzare una quantita' di memoria maggiore di quella fisica realmente disponibile

Gli indirizzi generati in un programma sono sempre indirizzi logici (virtuali), NON direttamente indirizzi fisici

Ogni dato e' contraddistinto da un indirizzo virtuale

Per poter recuperare il dato l'indirizzo virtuale deve essere tradotto in un indirizzo reale di memoria

La memoria virtuale e' gestita a pagine di dati

La traduzione indirizzo virtuale → indirizzo fisico avviene determinando in quale pagina (identificata da un numero) il dato si trova

Translation Lookaside Buffer (TLB) e' una tabella che contiene i riferimenti (traduzioni indirizzo virtuale -indirizzo fisico della pagina) alle pagine *ultimamente refereziate, presenti effettivamente nella RAM*

Quando la CPU richiede un dato, se la pagina che contiene il dato richiesto:

- ✓ e' presente nel TLB, il dato viene caricato dalla RAM nelle caches e poi nei registri, senza penalita'
- ✓ non e' presente nel TLB (**TLB miss**) → nuova traduzione virtual-physical e ricerca nel **Page Frame Table (PFT)**, una tabella piu' grande del TLB. Questa traduzione e ricerca e' costosa in termini di cicli di attesa da parte del processore

Se la pagina e'

- ✓ presente nel PFT → dato recuperato, la pagina viene memorizzata nel TLB rimpiazzando una TLB entry persistente
- ✓ non e' presente nel PFT → il sistema operativo cerca nella RAM una pagina di memoria disponibile (al limite deallocando pagine persistenti), carica da disco (*swap space*) la pagina richiesta, update PFT e TLB. Questa procedura (**page fault**) e' molto costosa.

Tramite la memoria virtuale si riesce quindi ad indirizzare uno spazio di indirizzi maggiore dello dimensione fisica della RAM

Uno spazio contiguo di indirizzi logici del programma puo' non essere contiguo fisicamente in RAM (dipende dalla locazione delle pagine). e' pero' fisicamente contiguo all'interno di una pagina di memoria virtuale

Se il programma pero' accede ad uno spazio dati > memoria fisica il sistema operativo carica/scarica in continuazione pagine di memoria dalla RAM \leftrightarrow disco

→ **paging** del programma

→ morte delle performance

Uno stride elevato durante l'elaborazione di una matrice

- accesso consecutivo ad elementi situati in pagine diverse e scarso utilizzo degli elementi una volta caricata la pagina
- aumento di TLB misses, page faults
- big overhead !

es. pagine di 4 Kbyte

```
real a(1024, 1024), b(1024,1024)
```

```
do j=1, 1024
```

```
  do i=1, 1024
```

```
    a(i,j) = b(j,i)
```

```
  enddo
```

```
enddo
```



stride 1024 →
 $4 \times 1024 = 4 \text{ Kbyte}$ → ogni
accesso puo' causare un
TLB miss

- Tecnica utilizzata per aumentare il throughput di istruzioni (numero di istruzioni eseguite in una unità di tempo)
- L'idea di base è dividere il processamento di una istruzione in stadi indipendenti, in modo da avere un processing rate pari allo step più lento, che è comunque più veloce dell'intero ciclo di step.
- Concetto simile alle catene di montaggio delle automobili.
- La pipeline classica è composta da 5 stadi.

ALU instructions: Prendono il valore contenuto in due registri, effettuano l'operazione richiesta e memorizzano il risultato in un terzo registro.

(fadd fr0, fr1 → fr2)

Load/store instructions: Utilizzate per trasferire dati dalla RAM ai registri o viceversa:

store 1, i

load i, ir0

Branches and jump: Trasferimenti condizionali di controllo (es: cicli for, do)

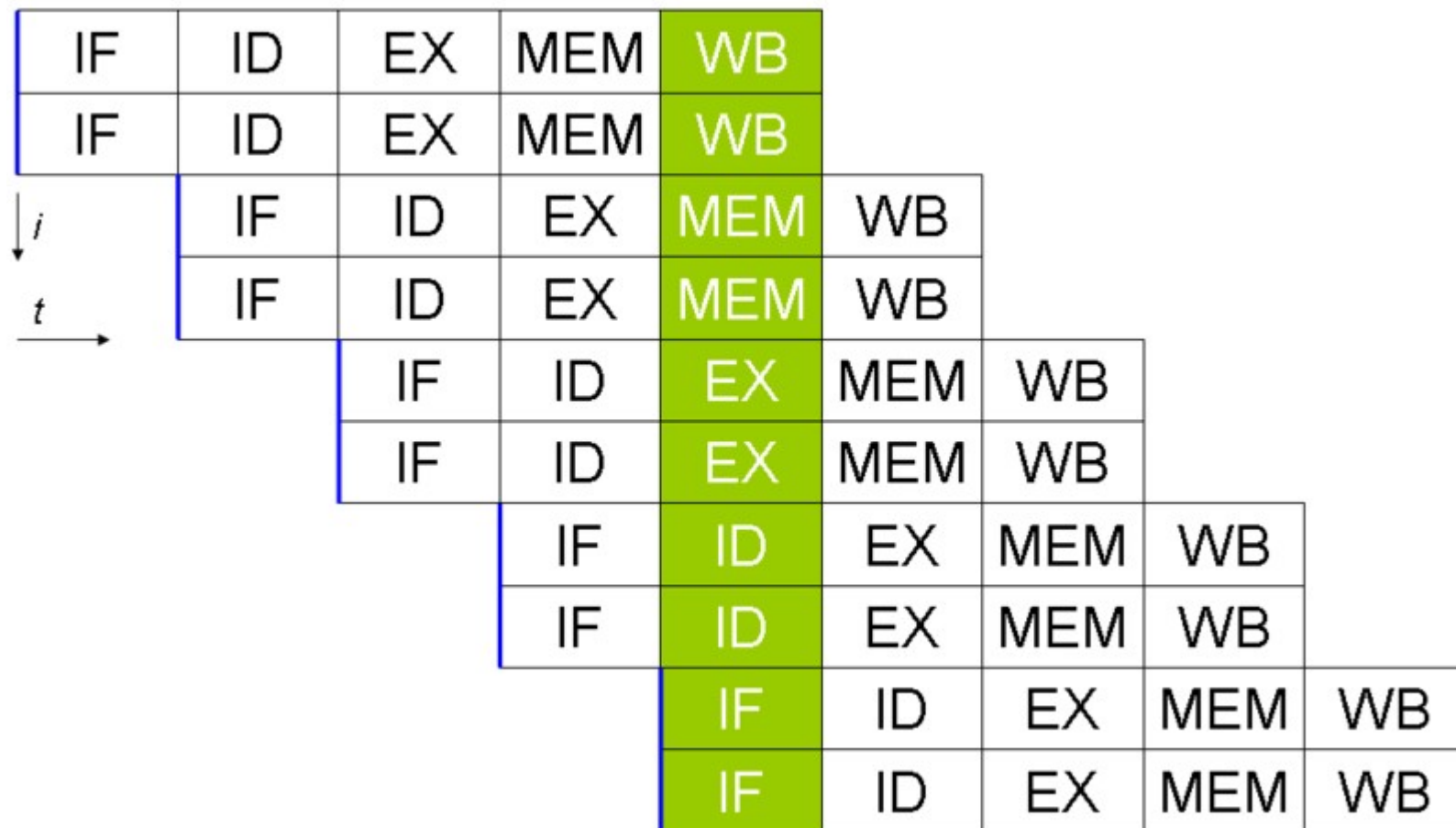
Instruction fetch (IF): Legge il PC(Program Counter) e preleva l'istruzione corrente. Aggiorna il PC aggiungendo 4 (se l'istruzione è 4 bytes)

Instruction decode (ID): Decodifica l'istruzione e legge i registri sorgente.

Execution (EX): Esegue l'istruzione definita. (Nel caso di load/store, viene calcolato l'indirizzo effettivo)

Memory access (MEM): se l'istruzione è una load, viene effettuata una lettura utilizzando l'indirizzo effettivo calcolato nel ciclo precedente. Se è una store, viene scritto in memoria il dato.

Write-back (WB): Scrive il risultato nel register file.



Il flusso di una pipeline può andare in stallo a causa di criticità di vario tipo:

Structural hazards: conflitti su risorse, quando l'hardware non ha a disposizione più di tot unità funzionali per i vari stati (es: con 5 unità per IF, posso caricare al più 5 istruzioni nello stesso tempo.

Data hazards: Quando una istruzione dipende dal risultato di precedenti istruzioni

Control hazards: istruzioni di branches o altro tipo che possono modificare il PC

In particolare, Data Hazards:

Read After Write (RAW):

R2 = R1 + R3

R4 = **R2** + R3

Write after Read (WAR):

R1 = R2 + **R3**

R3 = R4 + R5

Write After Write (WAW):

R2 = R1 + R3

R2 = R4 + R7

Read After Read (RAR):

R2 = R1 + **R3**

R4 = R1 + **R3**

La pipeline non diminuisce il tempo di completamento di una singola istruzione.

La profondità della pipeline è cruciale per le prestazioni.
Fino a 40 stadi nel Pentium IV !

Più è lunga la pipeline, meno dipendenze vi sono tra i vari stadi, ma aumenta la complessità hardware e nel caso di branch misprediction il tempo speso per l'esecuzione delle istruzioni nella pipeline in quel momento è maggiore.

In-order execution:

- 1) Instruction fetch
- 2) Se l'operando di input è disponibile, l'istruzione è demandata all'unità funzionale appropriata. Se uno degli operandi non è disponibile, il processore attende fino a che non diventa disponibile.
- 3) L'istruzione è eseguita
- 4) Scrittura del risultato

Se due istruzioni hanno hazards, la pipeline va in stallo, anche se vi sono istruzioni successive senza dipendenze.

- 1) Instruction fetch
- 2) Dispatch in una instruction queue (reservation station)
- 3) L'istruzione attende nella coda fino a che gli operandi di input non sono disponibili. L'istruzione può lasciare la coda nel caso di attesa lunga.
- 4) L'istruzione è eseguita dalla unità funzionale appropriata
- 5) Il risultato è accodato
- 6) Solo dopo che tutte le istruzioni precedenti hanno scritto i risultati nei registri, l'istruzione corrente scrive il suo risultato.

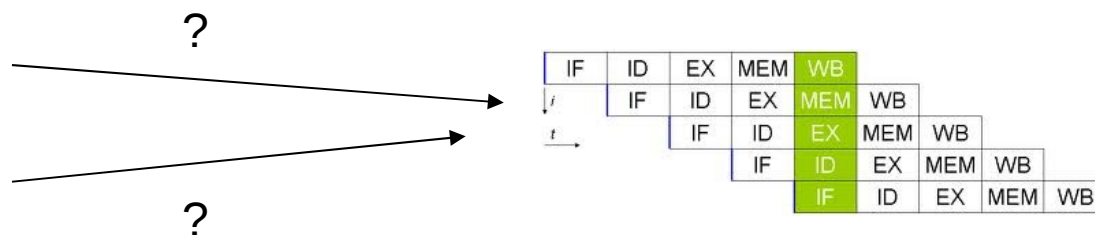
Permette di eliminare lo stallo nel punto 2 della in-order execution. Nella OoO execution gli stalli sono riempiti con altre istruzioni, quindi vengono riordinati i risultati. Program order, secondo il programmatore e data order, secondo il processore.

Richiede un'analisi delle dipendenze (scoreboarding), mediante la quale vengono eseguite fuori ordine le istruzioni.

In presenza di percorsi condizionati (branch), la CPU dovrebbe attendere l'esito del salto (stadio EX) per poter caricare le istruzioni successive, in quanto non conosce a priori quali siano.

In architetture con pipeline, soprattutto con molti stadi, l'attesa può essere molto onerosa, in quanto blocca il flusso di esecuzione.

```
if( a > 0)
    x = x + 1
else
    x = x - 1
```



Una predizione del salto (esecuzione speculativa), se corretta, permette di eliminare lo stallo dato dalla risoluzione del salto. Istruzioni subito nella pipeline!

Se la predizione è corretta, il salto non costa nulla, se errata, il salto costa quanto lo svuotamento della pipeline, ed il caricamento delle istruzioni corrette

```
if( a > 0)
    x = x + 1
else
    x = x - 1
```

Speculazione



```
a = 100;  
for( int i=0; i < 100; i++) {  
    a = a - 1;  
    if( a == 2)  
        x = 3;  
}
```

Per il 98% delle volte la condizione nell'if è falsa!

Una esecuzione speculativa, mi permette di selezionare correttamente 97 salti su 100, senza attendere inutilmente

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(){
    double a;
    srand ( time(NULL) );

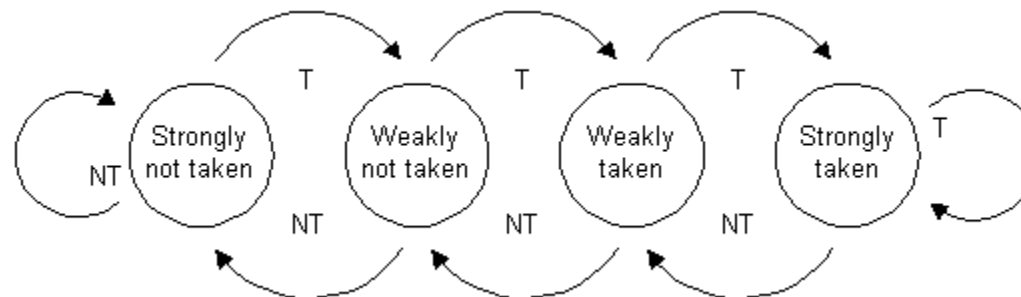
    for ( int i = 0; i < 100; i++){
        a = ((double)rand())/RAND_MAX;
        if( a > 0.8)
        {
            printf(" a %f \n", a);
            printf(" i %d \n", i);
            break;
        }
    }
}
```

La condizione del salto è indefinita ad ogni iterazione!

La predizione è molto complicata in questo caso.

Predizione statica: Salti verso istruzioni precedenti sempre accettati (es: ciclo). Verso istruzioni in "avanti" non accettate (es: uscita da un ciclo). Nei cicli fallisce solo alla fine del ciclo.

Predizione bimodale (o contatore a saturazione): utilizza un contatore a 2 bit:



Quando il branch è valutato, viene aggiornato lo stato della macchina a stati.

I branches valutati come non presi decrementano il loro stato fino a “strongly not taken”, mentre quelli valutati come presi incrementano lo stato fino a “strongly taken”.

Prestazioni: fino al 93,5% di predizioni corrette.

- Esistono piu' unita' funzionali (di diverso o dello stesso tipo) che possono operare parallelamente fra loro
- il discorso precedente e' svolto da tutte le unita' funzionali parallelamente
- un numero alto di istruzioni eseguito contemporaneamente ad un certo istante
- per ogni ciclo di clock possono essere completate piu' istruzioni

Loop Unrolling

```
do j=1, n
  do i=1, (n-1)
    a(i,j)= b(i,j)+b(i+1,j)+1.0
  enddo
enddo
```

**Attenzione quando
n non multiplo del
FACT_UNROLLING !**

```
do j=1, n
  do i=1, (n-1), 2
    a(i,j) = b(i,j) +b(i+1,j)+1.0
    a(i+1,j) = b(i+1,j)+b(i+2,j)+1.0
  enddo
enddo
```

Loop equivalenti come risultati. Il loop unrollato (oltre alla diminuzione di jump e di controlli iniziali) permette la riduzione/eliminazione di eventuali dipendenze, un miglior controllo del flusso dati e crea blocchi piu' "succosi" per un miglior scheduling e sfruttamento dei diversi stage e unita' funzionale del processore

3 tipi fondamentali di processori

- RISC (Reduced Instruction Set Computer)
- CISC (Complex Instruction Set Computer)
- Vector

Molti registri

Poche istruzioni Assembler

Solo load/store operano in memoria: tutte le altre istruzioni operano sui registri

Pipelining efficiente

Molte istruzioni assembler

Le istruzioni possono operare sui registri e anche su operandi direttamente in memoria

Ogni istruzione assembler → diverse μ ops (microcode) effettivamente eseguito dalla CPU

elevato numero di istruzioni, microcode → maggior difficoltà' pipelining

Registri vettoriali (piu' di un dato scalare)

Load/Store caricano in un colpo solo piu' dati scalari in un registro

Le operazioni fra registri vettoriali operano contemporaneamente su tutti i dati scalari

□ il codice puo' essere vettorizzato (stesso tipo di operazione su dati localmente vicini in memoria senza dipendenze fra le operazioni)

