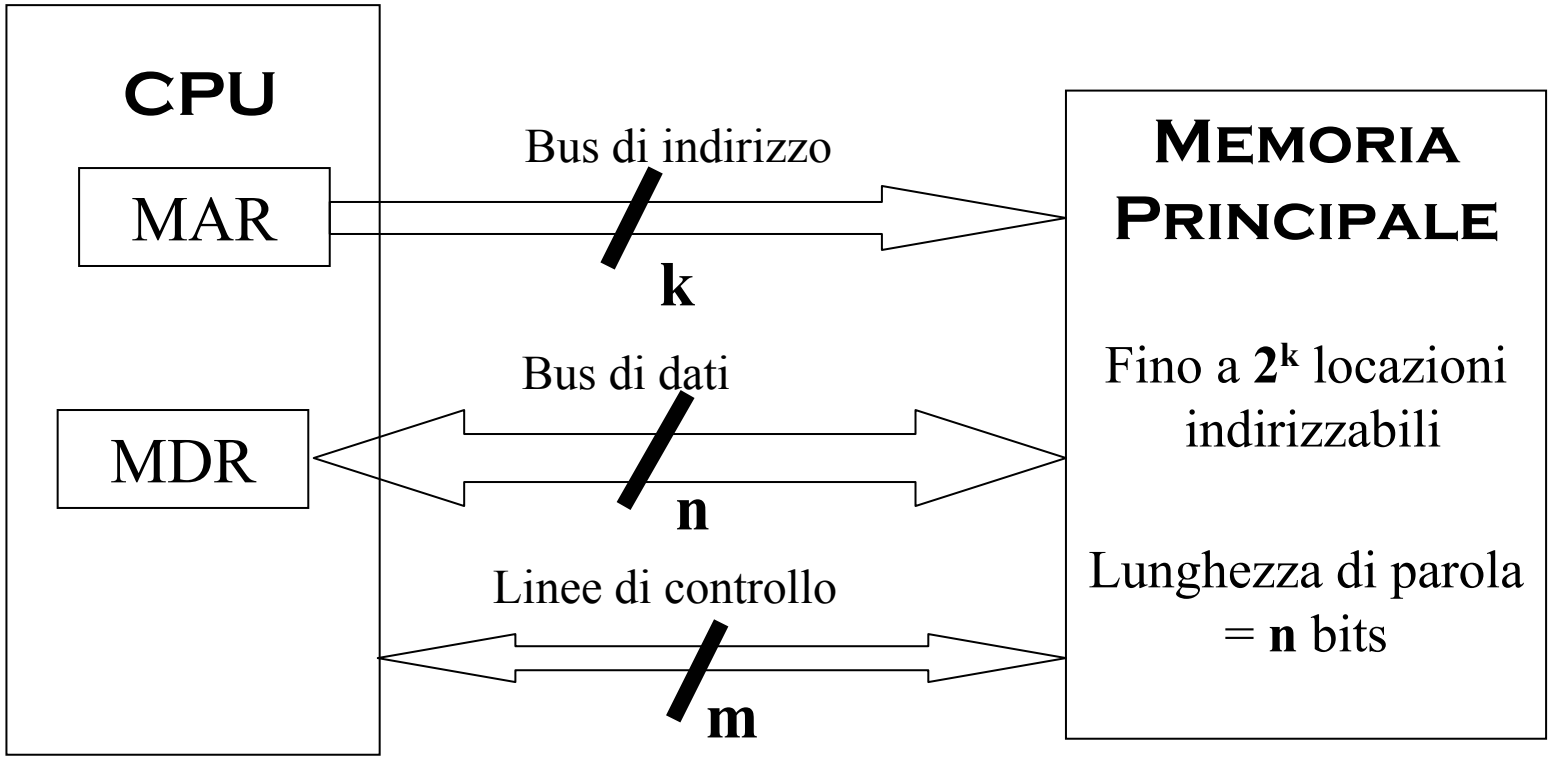
Calcolatori Elettronici B
a.a. 2004/2005

GERARCHIA DI MEMORIE
MEMORIA VIRTUALE

Massimiliano Giacomin

Accedere alla memoria

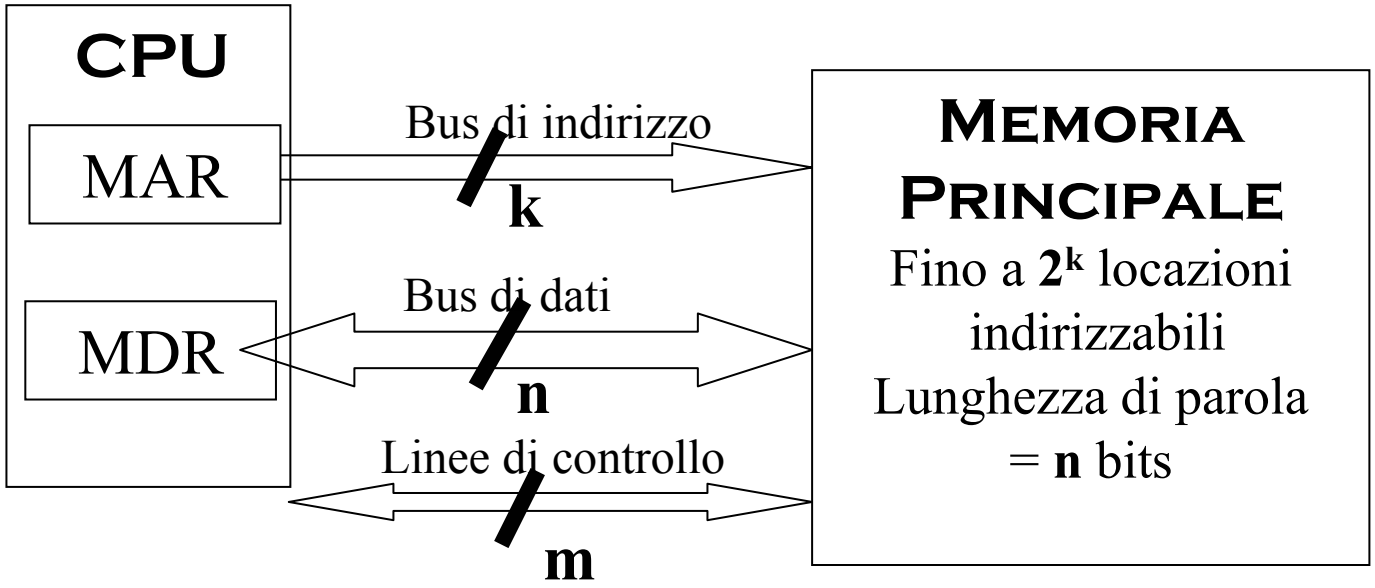


Linee di controllo: Read,Write, MFC (Memory Function Completed)

Operazione di memoria [a grandi linee]:

CPU: Metti indirizzo in MAR [dato in MDR se scrittura]
Asserisci Write (o Read)
MEMORIA: (Trova parola) e trasferisci n bit
ad operazione completata asserisci MFC

Misure relative alla memoria



Importanza di **k** (spazio di indirizzamento)

Tempo di accesso alla memoria (ta) :impiegato per eseguire una operazione di memoria
(p.es. per operazione di lettura: tempo tra l'affermazione di Read e l'arrivo di MFC)

Tempo di ciclo (tc): tempo che deve intercorrere tra due operazioni di memoria
[per memorie con refresh o altri problemi di gestione che richiedono Δ t dopo l'operazione **tc= ta+Δt**]

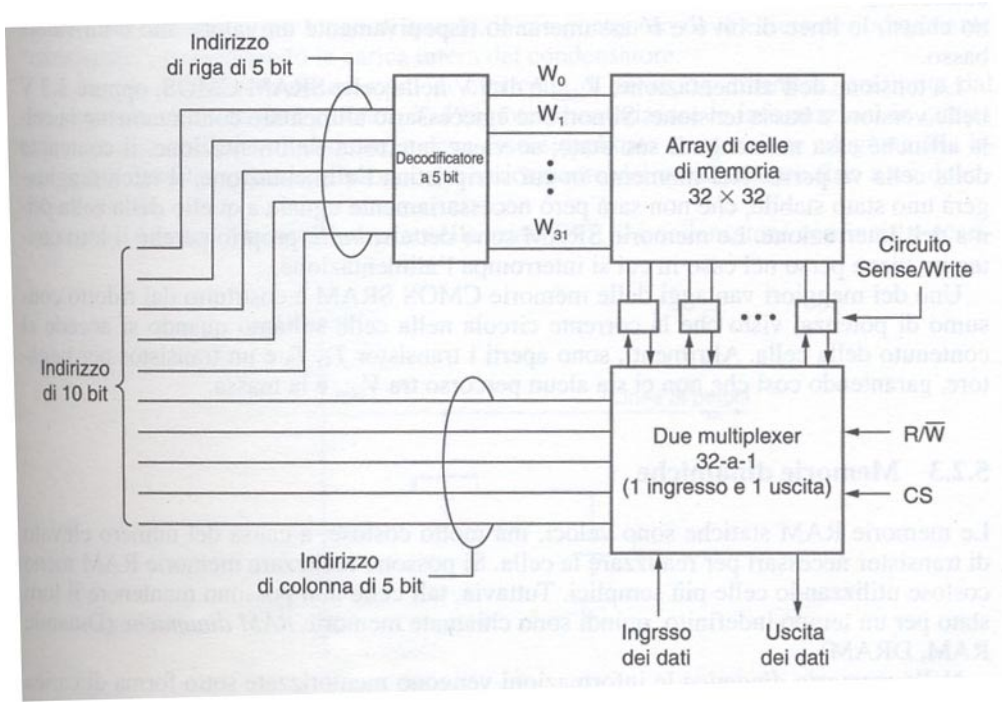
Velocità di trasferimento: velocità con cui i dati possono essere inviati o ricevuti dall'unità di memoria. Per memorie ad accesso casuale: 1/*tc*

Memorie RAM

Chip di memoria

- Per limitare il numero di pin, spesso i chip hanno un numero limitato di linee dati:
  P.es. 1M*1 bit 4M*4 bit ecc. ecc.
- All'interno di un chip, struttura di celle a matrice [decoder decodifica indirizzi]:
  - linea di parola: pilota le celle che fanno parte di una "linea"
  - linee di bit: una linea in ingresso a ciascuna cella, una linea in uscita

Esempio di organizzazione 1K*1:



NB:
- R/$\overline{W}$: lettura o scrittura
- CS: seleziona il chip

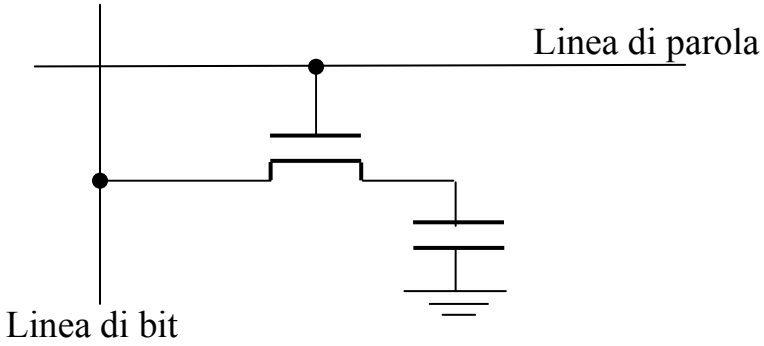
RAM statiche (SRAM):

La cella è costituita da 1 Flip-Flop (due invertitori in retroazione):

- diversi transistor ⇒ aumenta costo e dimensioni
- ciascuna cella capace di "memorizzare" 1 bit
- ridotto consumo di potenza (idealmente la corrente fluisce solo in transizione)

RAM dinamiche (DRAM):

La cella di memoria è molto più semplice:

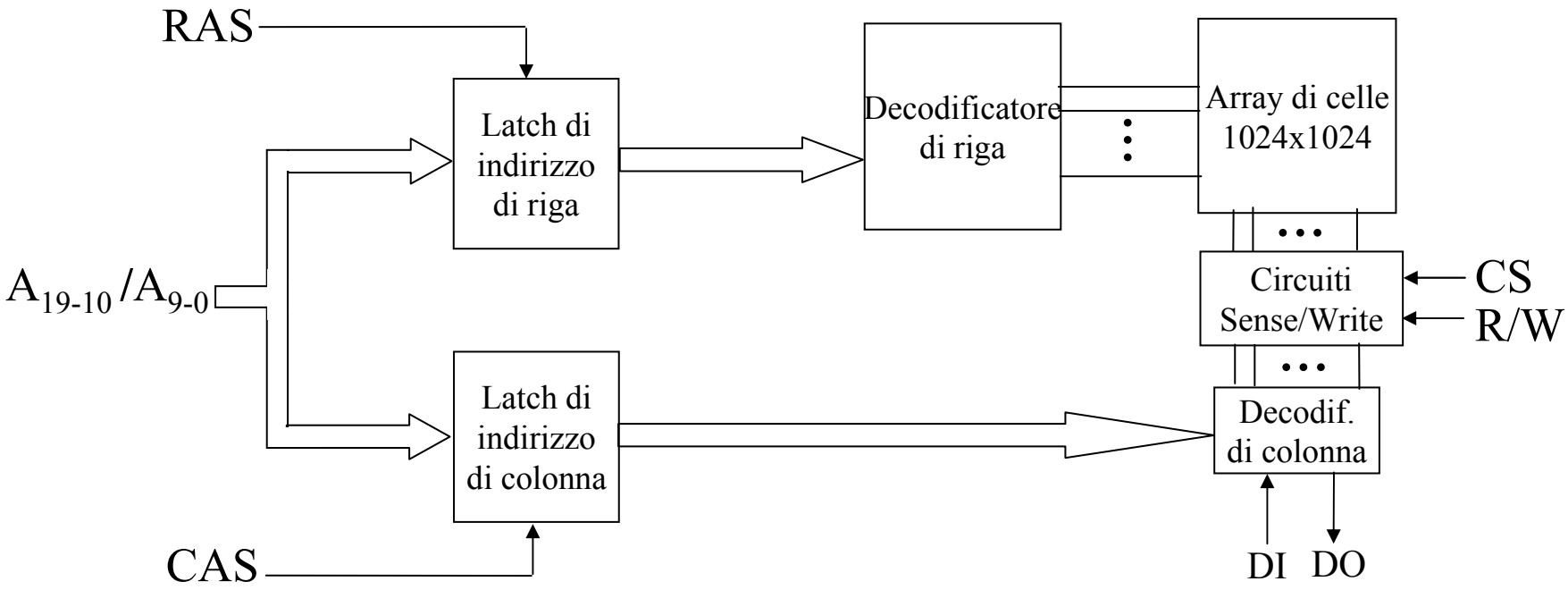


- Condensatore memorizza Bit [carico-scarico]
- Attivando la linea di parola si può leggere la linea di bit [o scrivere]

⇒ Minori dimensioni (aumenta la densità di integrazione)
e quindi aumenta la capacità in bytes
Minor costo: sono le memorie usate per memoria principale del calcolatore

… ma il condensatore si scarica: necessità di un **circuito di refresh**
⇒ minori prestazioni

Organizzazione interna di un tipico chip DRAM: 1M x 1



Operazione di lettura o scrittura

- Si applica l'indirizzo di riga
- Si attiva il segnale "Row Address Strobe" [RAS]
- Si applica l'indirizzo di colonna
- Si attiva il segnale CAS
- A questo punto, lettura o scrittura a seconda di R/W

Ogni volta che RAS=1, vengono rinfrescate tutte le celle della riga corrispondente!

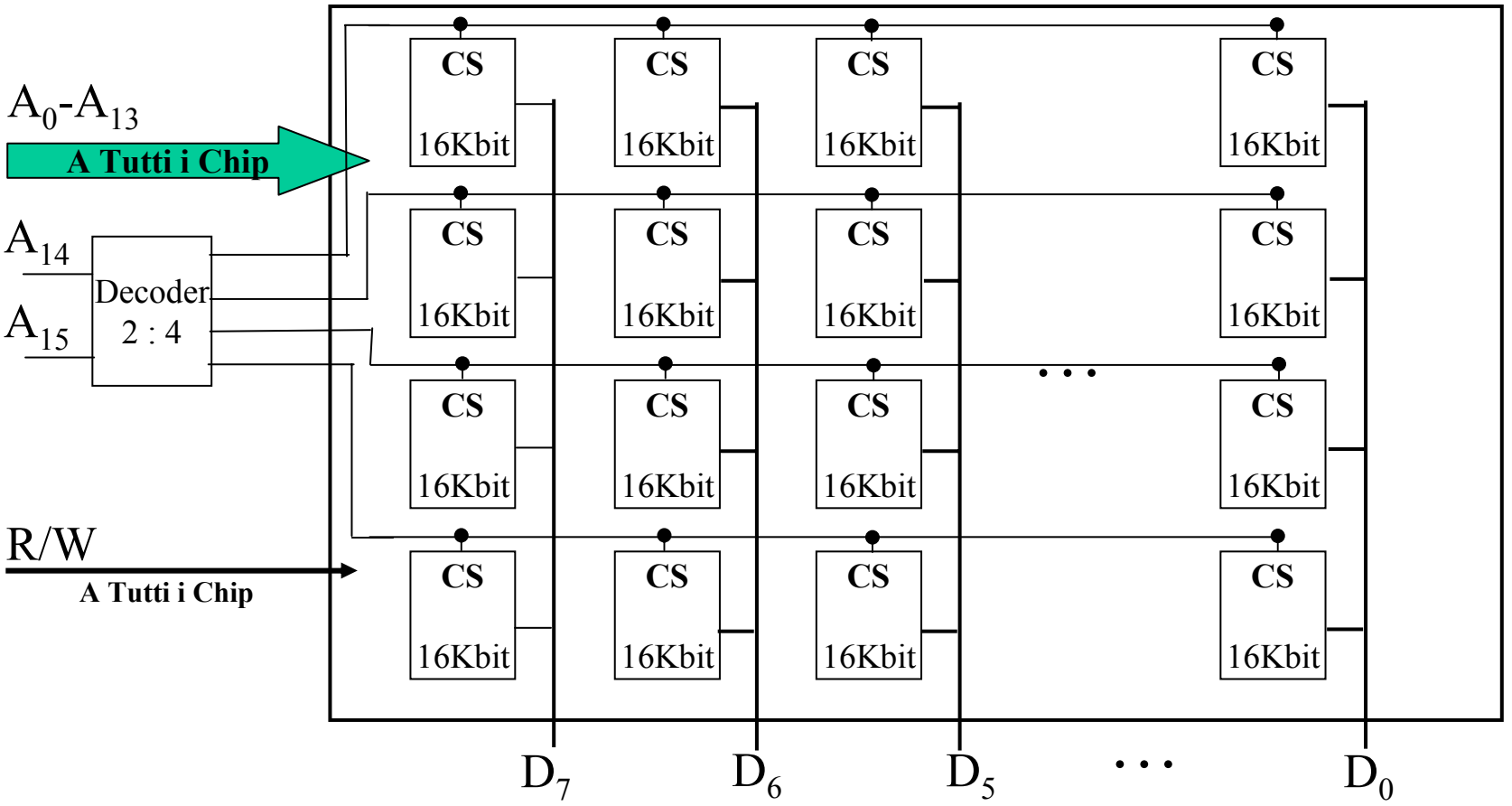
➡ Il circuito di refresh si occupa di rinfrescare periodicamente tutte le righe

Banchi di memoria (SRAM):

Utilizzando i segnali chip-select, più chip possono essere collegati (tramite decodificatori) per formare un banco di memoria

Esempio: memoria 64Kbytes via chips 16K * 1:

- righe da 8 chips X
- 4 righe [ciascuna 16 K bytes]

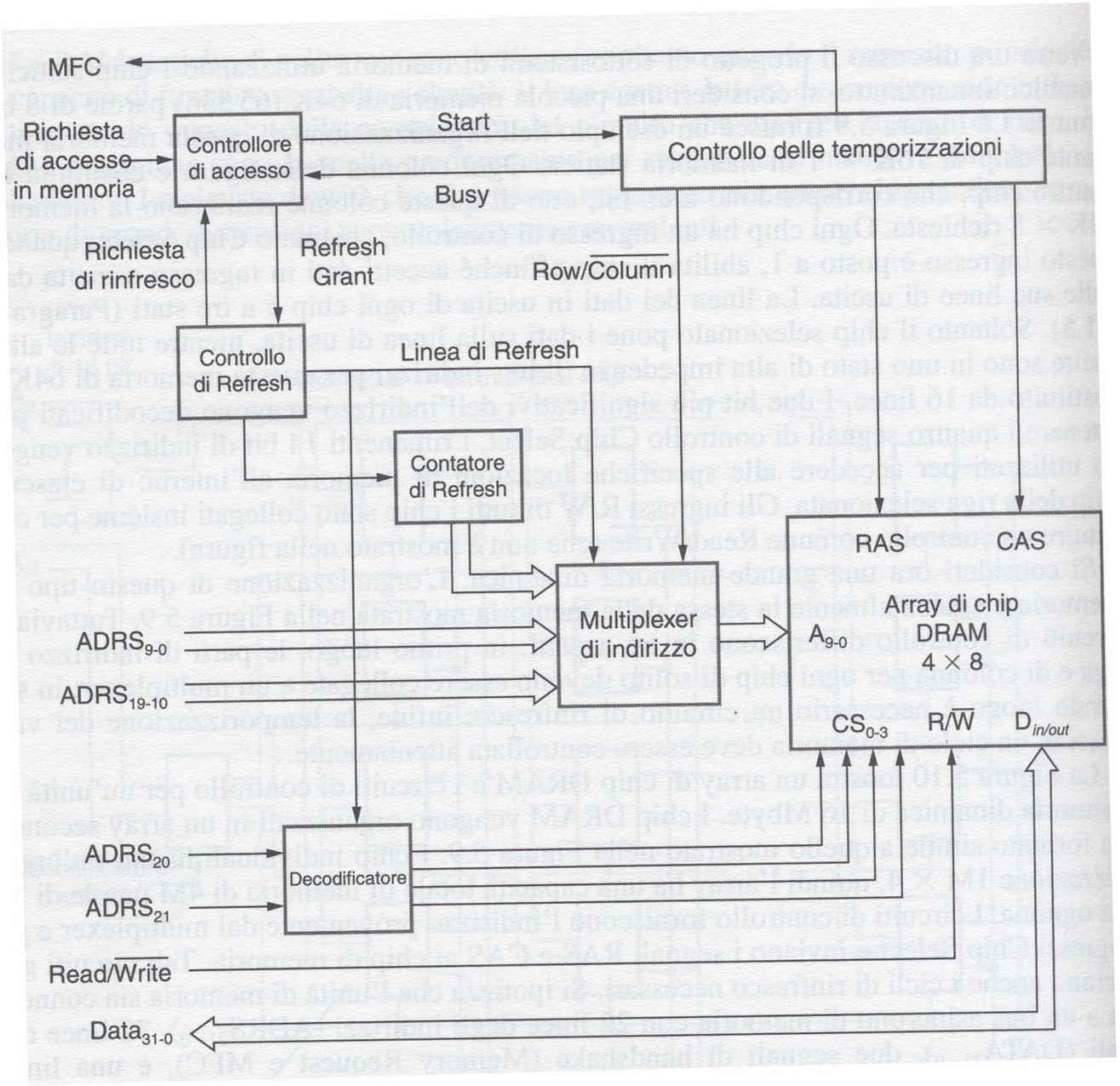


Esercizi:

"Allenarsi" a progettare banchi di memoria, p.es.

- Memoria 4M x 8 con chips da 256K x 1

Banchi di memoria (DRAM) Dobbiamo provvedere a circuito di refresh



Chip singolo: 1M*4

Memoria:
4M*32

Note al lucido precedente

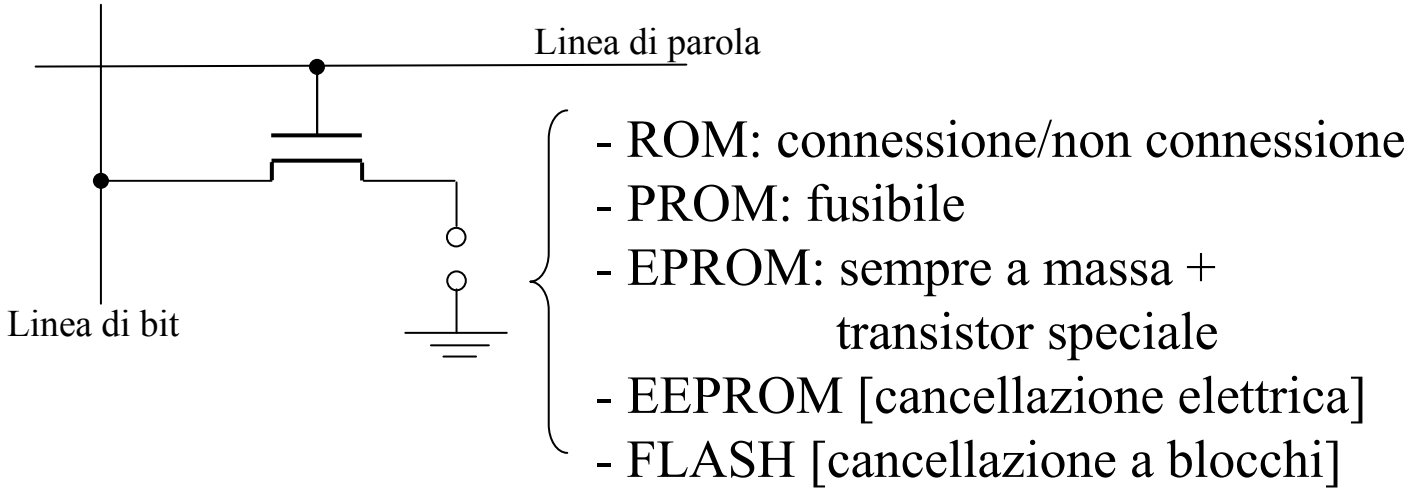
- Differenze con il banco SRAM:
  - Gestione indirizzi di riga e colonna via Multiplexer di Indirizzo
  - Circuito di rinfresco

  $\Rightarrow$ Necessità di temporizzare i vari passi [Controllo della temporizzazione]
- Accesso in memoria:
  - CPU attiva R/W + indirizzo [e dato per la scrittura] + Richiesta di accesso
  - Controllore di accesso accorda accesso e attiva Start
  - Controllo temporizzazioni risponde con Busy=1 e poi:
    - ° Row=1 e RAS [indirizzo di riga]
    - ° Row=0 e CAS [indirizzo di colonna]
  - La riga dei chip di memoria attivati da CS leggono o scrivono i dati [R/W]
  - Controllo temporizzazioni porta Busy = 0 e MFC=1
- Controllo di refresh rinfresca una intera riga periodicamente:
  - Chiede accesso con Richiesta di rinfresco
  - Controllore di accesso accorda accesso con Refresh Grant
  - Ciclo di memoria usuale, ma MUX è forzato [Linea di Refresh] ad usare l'indirizzo di riga nel contatore di refresh + CS tutti disabilitati
- Circuito di rinfresco ha priorità su normale accesso [cfr. Controllore di Accesso]:
  - Modalità di "rinfresco completo", oppure:
  - Modalità di rinfresco interallacciata (interleaved)

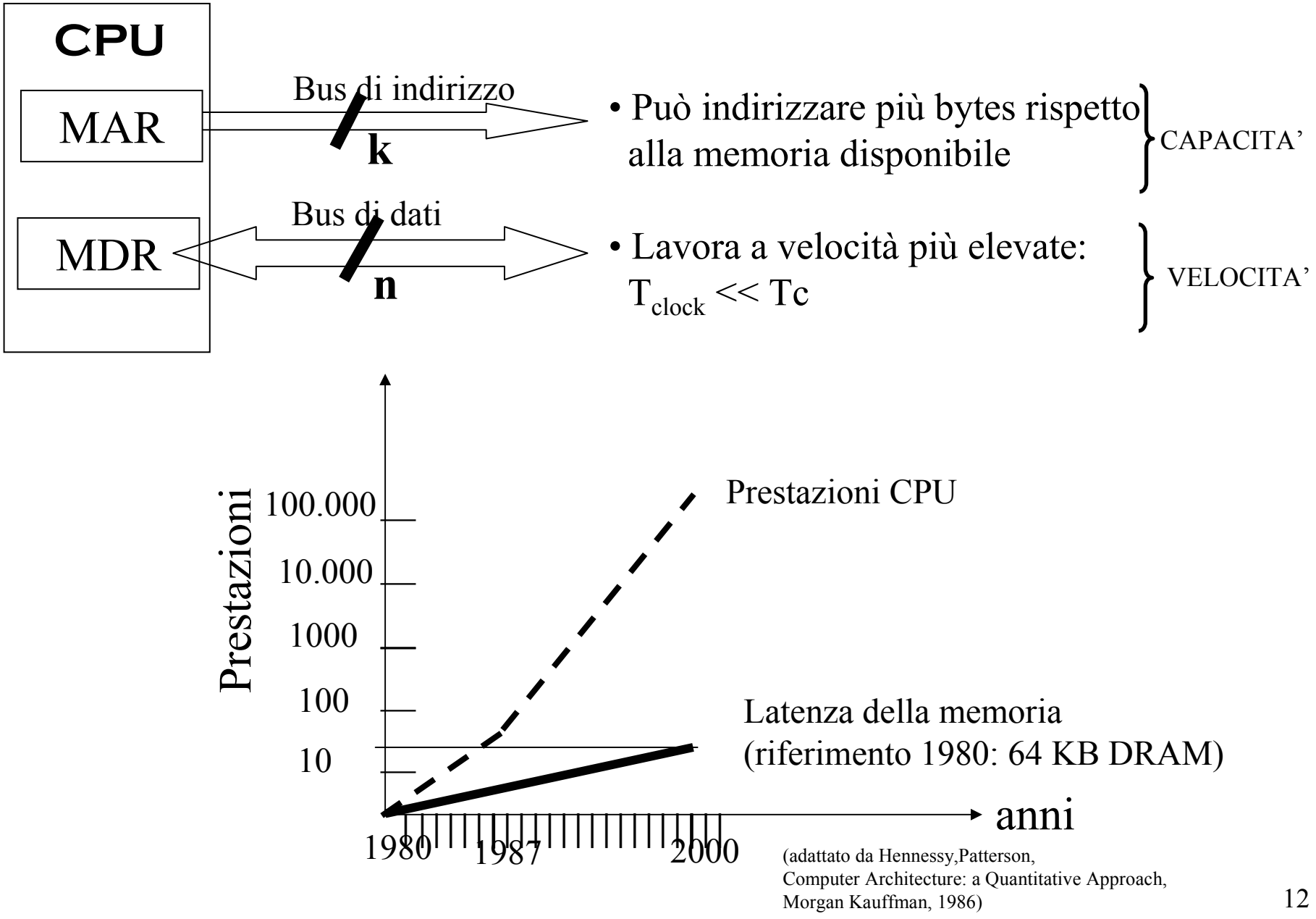
Memorie a sola lettura

- Nel calcolatore, contengono il BIOS (Basic Input/Output System): operazioni di base per la gestione dell'hardware (p.es. caricamento da disco) necessarie nel momento dell'accensione del calcolatore

- Cella ROM:



Problema: il collo di bottiglia



(adattato da Hennessy,Patterson,
Computer Architecture: a Quantitative Approach,
Morgan Kauffman, 1986)

Le richieste dei programmatori:

- Memoria grande — idealmente: memoria di massa (dischi)
- ” ” veloce — idealmente: CPU
- ” ” economica — idealmente: il meno possibile nel complesso!

Problema:

- SRAM più veloce (no refresh) ma più costosa e di dimensioni maggiori
- Velocità adeguata solo su CHIP del processore, ma dimensioni limitate

Osservazione: struttura iterativa dei programmi favorisce il ri-uso degli stessi blocchi di registri di memoria

Principi di località

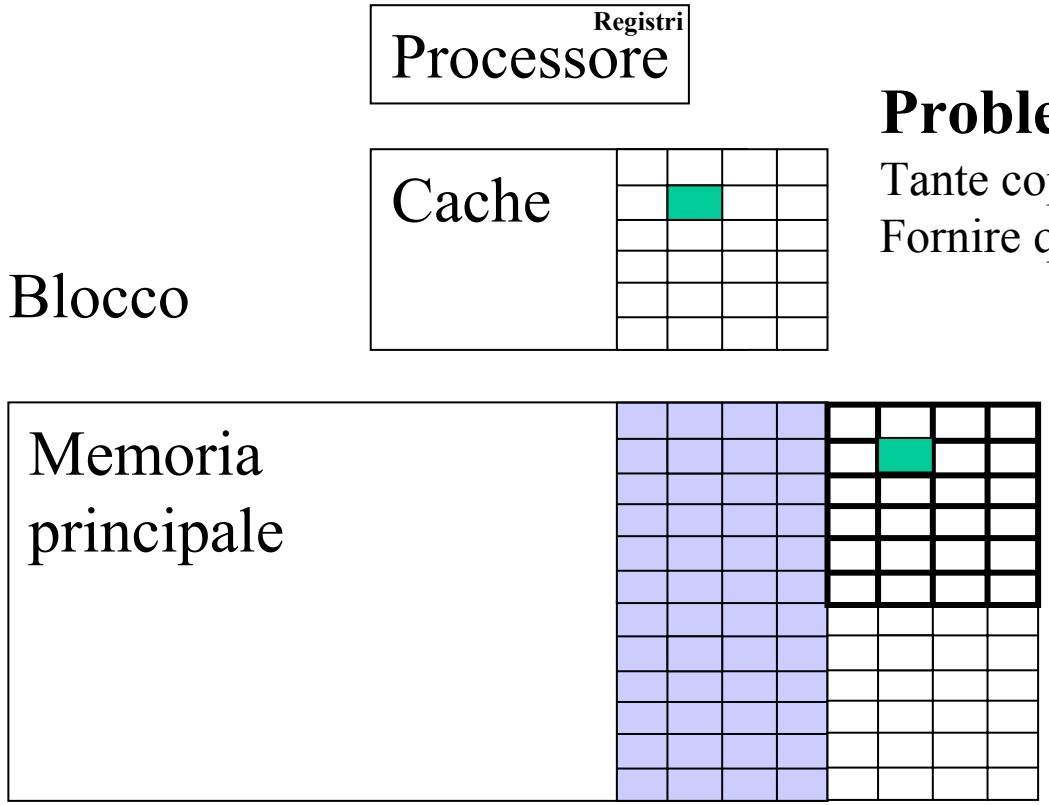
temporale: una locazione di memoria acceduta è plausibile sia ri-acceduta in breve tempo

spaziale: è plausibile che indirizzi vicini ad un indirizzo acceduto siano acceduti in tempi brevi

⇒ **Gerarchia di memorie**

Gerarchia di memorie

Tecnica per far apparire la memoria grande ma veloce : portare vicino alla CPU blocchi di parole che prevedibilmente saranno usate in base ai principi di località



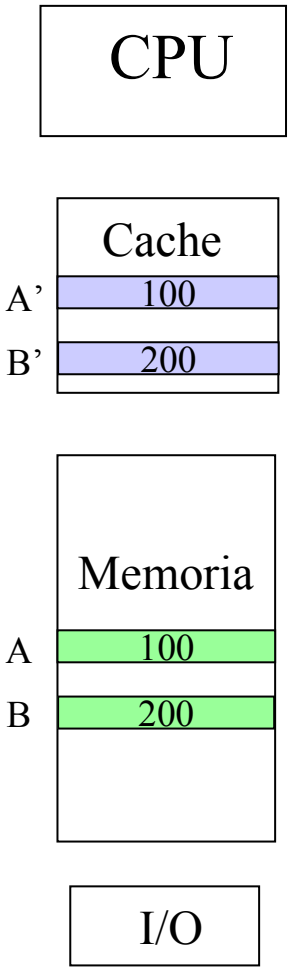
Problema della coerenza dei dati:
Tante copie ripetute dello stesso dato.
Fornire quella attuale

Problema della gestione del traffico tra i diversi livelli: quando debbo spostare i dati dal livello inferiore al superiore, dove copio il blocco? E quale tra i blocchi presenti sostituisco?

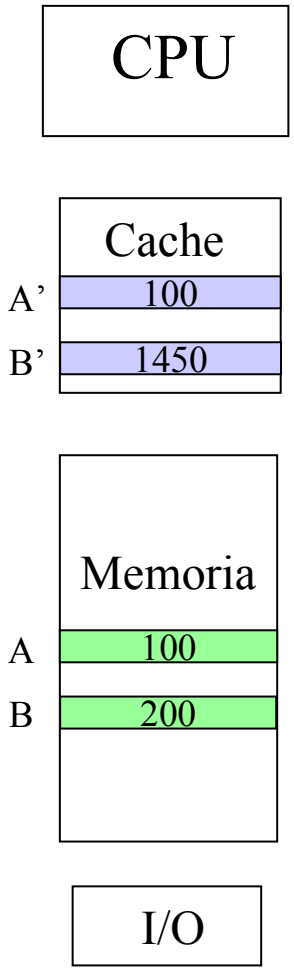
Note al lucido precedente:

- le cache in realtà sono due:
  - cache interna al processore (solo così si ottengono velocità comparabili, ma ciò limita notevolmente le dimensioni)
  - cache di livello 2 (esterna al processore)
- per i livelli inferiori aumentano dimensioni e diminuisce il costo unitario, ma diminuisce anche la velocità
- dalla cache in giù, i trasferimenti sono sempre "a blocchi": occorre progettare i vari elementi della gerarchia tenendo conto di questa caratteristica
- DRAM consentono trasferimenti "a blocchi" di frequenza doppia rispetto a parole singole: RAS solo una volta + CAS ripetuti per parole consecutive
  $\Rightarrow$ ok per "trasferimenti a blocchi" con la cache
- I dischi hanno tempi di accesso molto alti, ma velocità di trasferimento discrete
  $\Rightarrow$ ok per "trasferimenti a blocchi" via DMA…

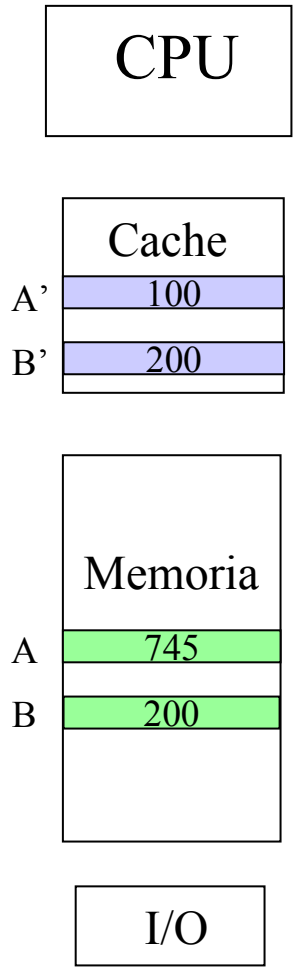
Il problema della coerenza dei dati



Cache e Memoria
Coerenti
[A]=[A’]& [B]=[B’]



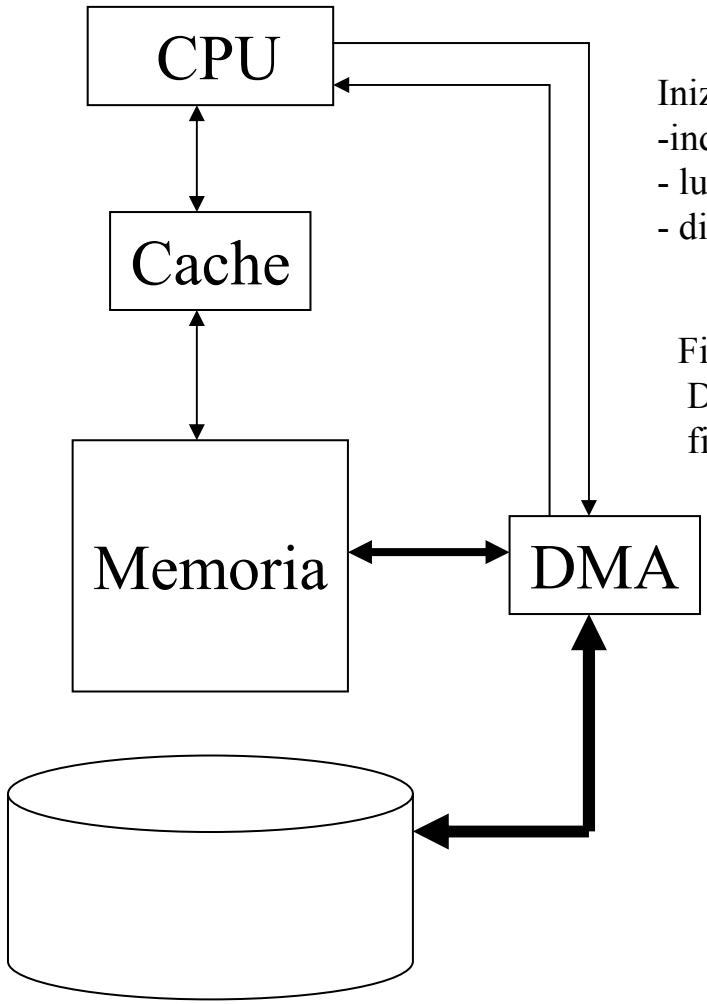
Cache e Memoria
Incoerenti per operazione CPU
[B]≠[B’], [B] vecchio



Cache e Memoria
Incoerenti per operazione di I/O
[A] ≠[A’]. [A’] vecchio

Problema della gestione del traffico

DMA (Direct Memory Access)



Inizio CPU invia
-indirizzo di partenza
- lunghezza del blocco da trassferire
- direzione (I/O)

Fine:
DMA invia interrupt per informare fine operazione

Le memorie cache

NB: quasi tutti i concetti sono validi a tutti i livelli della gerarchia!

La CPU ignora la gerarchia: invia alla 'memoria ' una richiesta di lettura o scrittura.

La circuiteria di gestione (nel controllo della CPU o dei singoli livelli di memoria) della memoria verifica se la parola è nella cache, nella memoria o nel disco e provvederà a fornire il dato corretto alla CPU.

Parametri di giudizio

1- frequenza di successo (hit) h

2- frequenza di mancamento (miss) $1-h$

3- tempo di successo C

4- penalità di fallimento M

$\Rightarrow$ Tempo medio di accesso: $hC + (1-h)M$

Elementi importanti:

•Funzione di traduzione (mapping):

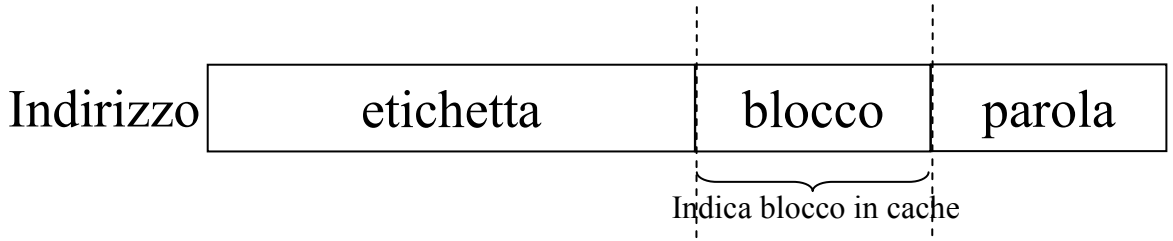
posizione blocco livello inferiore (memoria) $\rightarrow$ posizione livello superiore [cache]

• Algoritmo di sostituzione:

interviene per decidere quale blocco sostituire quando la cache è piena

Funzioni di traduzione

• Indirizzamento diretto: blocco memoria in posizione fissa



• Indirizzamento completamente associativo

Più flessibile, ma costo della *ricerca associativa*

• Indirizzamento set-associativo

Coerenza dei dati: caso cache - memoria principale

Successo (hit) la parola cercata è in cache

in lettura: OK

in scrittura: problema della coerenza con i dati in memoria; due alternative:

tecniche di **write through**

[aggiornamento contemporaneo memoria]

tecniche di **write back** (dirty bit)

[aggiornamento memoria in seguito, quando il blocco deve essere rimosso dalla cache]

Insuccesso (miss):

in lettura: spostamento di un blocco in memoria cache, due alternative:

spostamento blocco e poi accesso in lettura, oppure

tecniche di **load-through** [si invia subito parola alla cache]

in scrittura: due alternative

write through:

scrittura in memoria RAM senza trasferire blocco in cache

write back:

trasferimento blocco in cache e scrittura nuove informazioni

Coerenza dei dati: caso cache - memoria principale (continua)

NB: serve anche un **bit di validità** per ogni blocco, che indica se il blocco è valido:
HIT solo se bit di validità a 1, altrimenti il blocco in cache non è valido

- all'inizio tutti i bit di validità a 0 [cache non caricata]
- bit di validità di un blocco posto a 1 quando un blocco da memoria principale è trasferito nella cache [blocco caricato e aggiornato]
- ogni volta che DMA trasferisce dati da dispositivo a memoria RAM [blocco]:
  se il blocco è presente in cache bit di validità a 0 [dati incoerenti con RAM]
- ogni volta che DMA trasferisce dati da memoria RAM [blocco] a dispositivo:
  se la cache utilizza write-back [write in blocchi cache senza aggiornare memoria]
  la memoria può non riflettere la cache: una possibilità è lo svuotamento (flush) della cache: i blocchi con dirty bit a 1 sono trasferiti in memoria

Algoritmi di sostituzione

- Ovviamente, servono solo per cache associative o set-associative!
- Algoritmo LRU:

  in base a località spaziale e temporale eliminare i blocchi usati meno di recente

  Problemi possibili:

  - implementazione costosa

    [per ogni blocco, contatore memorizza posizione – è necessario aggiornarli ogni volta che c'è accesso con hit – miss e cache vuota – miss e cache piena]

    $\Rightarrow$ talvolta approssimazioni di LRU [es. blocco "più vecchio"]

  - in alcuni casi [p.es. accesso a vettore che non sta in cache] è inefficiente:

    preferibile algoritmo casuale o che comunque introduce casualità nella scelta

NB: vedremo un caso in cui LRU è decisamente inefficiente nel contesto della memoria virtuale…

Miglioramento delle prestazioni

Tempo medio di accesso: hC + (1-h)M

h: frequenza di hit

C: tempo per accedere ai dati in cache

M: penalità di fallimento = tempo per trasferire il blocco richiesto in cache

⇒ Occorre agire su tutti i parametri…

C: nel caso di hit, con cache sul chip della CPU idealmente C = 1 ciclo di clock; per approssimare questa condizione, uso di una coda di prefetch

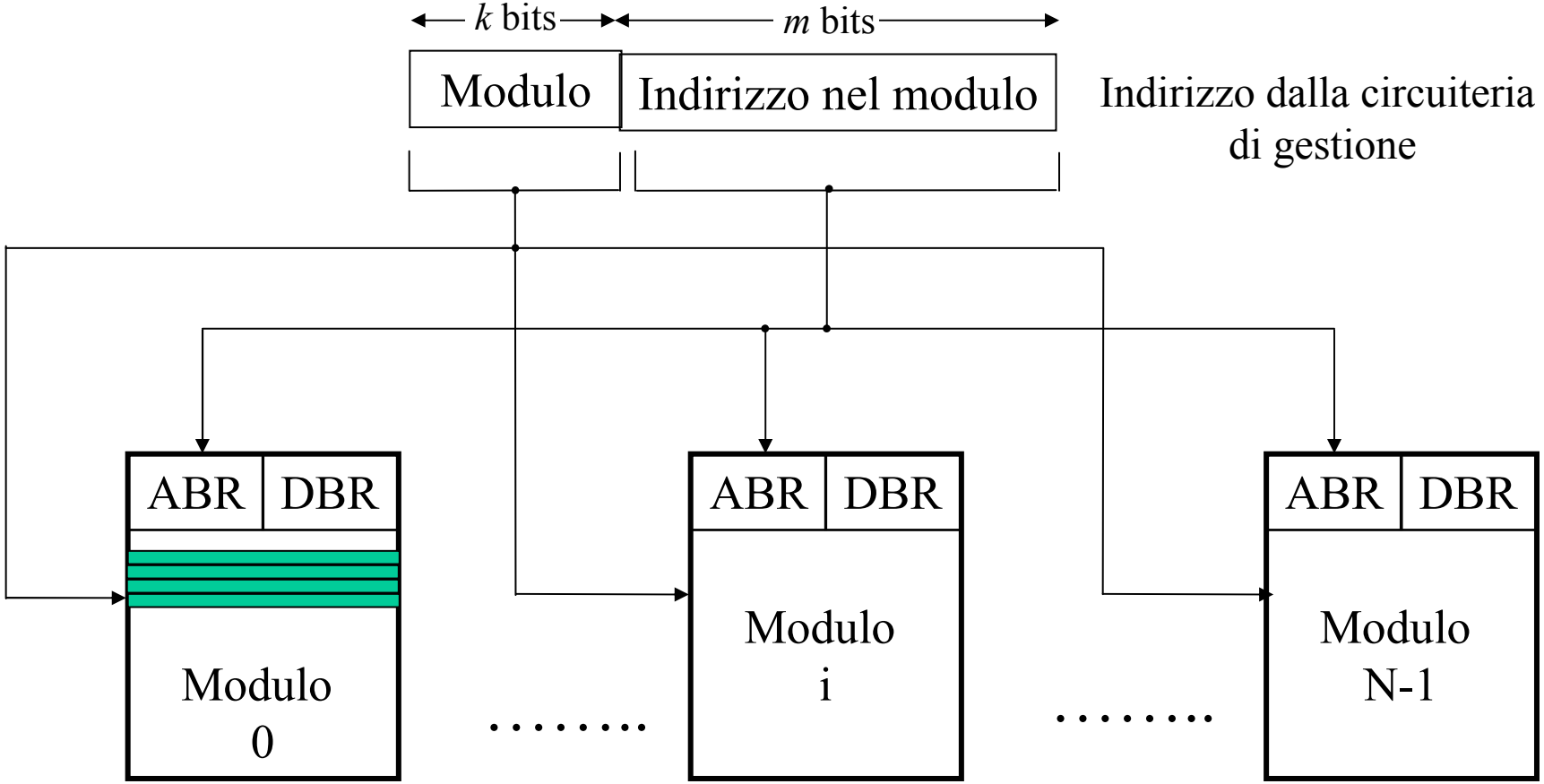
M: occorre aumentare la velocità di trasferimento di un blocco da RAM a Cache

⇒ Tecnica dell'**interallacciamento**:

- Tra i livelli cache e memoria, utilizzare in parallelo più chip di memoria (unità più lenta) per sopperire alla minor velocità
- Occorre distribuire gli indirizzi attraverso i moduli in modo particolare…

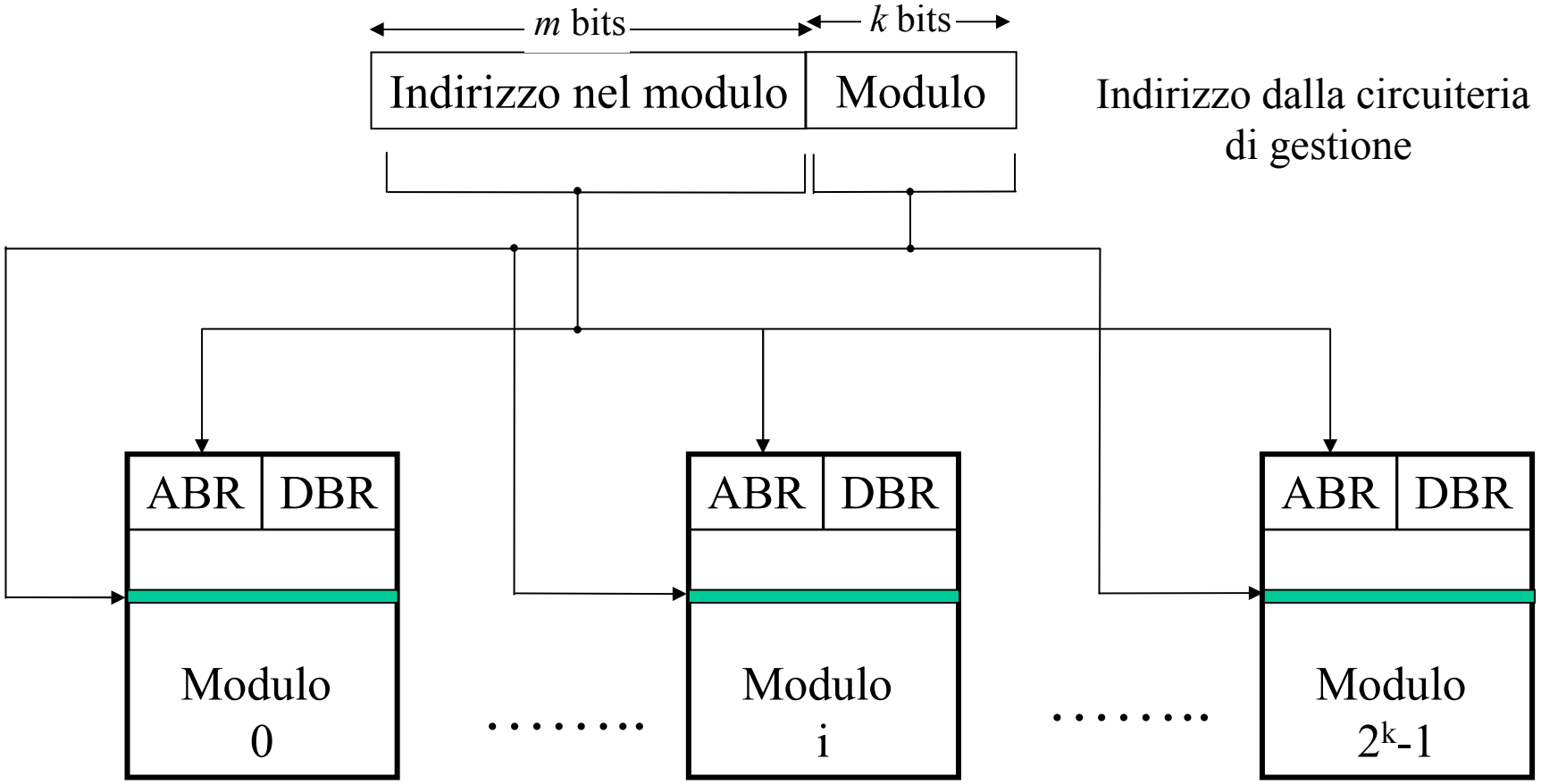
NB: si ricorda che la memoria è organizzata a moduli…

Senza interallacciamento: le parole di un blocco sono memorizzate consecutivamente in un modulo



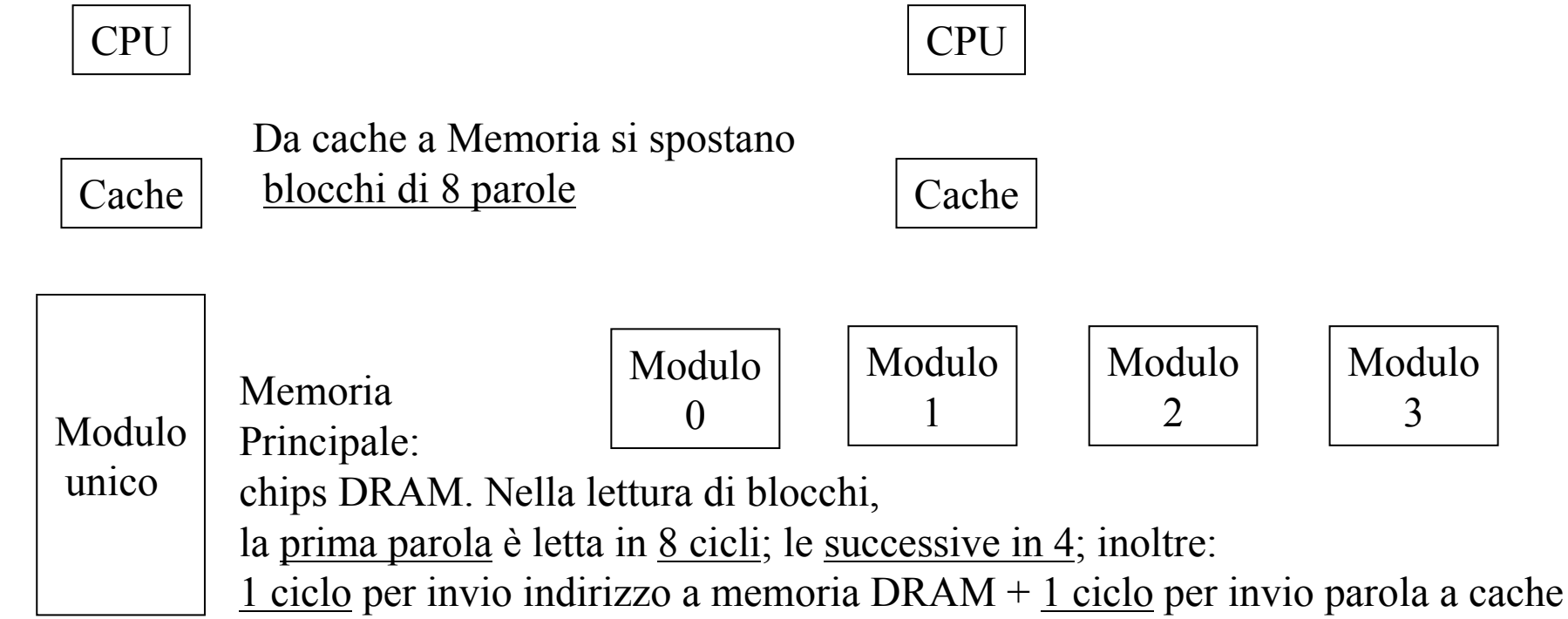
NB: ha comunque il vantaggio che, intanto, DMA può accedere a moduli che non si trovano nel blocco acceduto correntemente

Con interallacciamento: Le parole di un blocco sono sparpagliate
e memorizzate ognuna in un modulo diverso

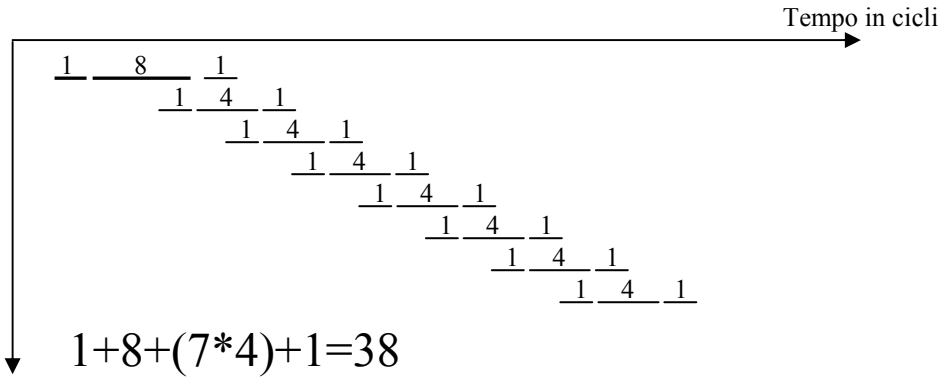


Consente di attivare in parallelo i chips DRAM, risparmiando cicli

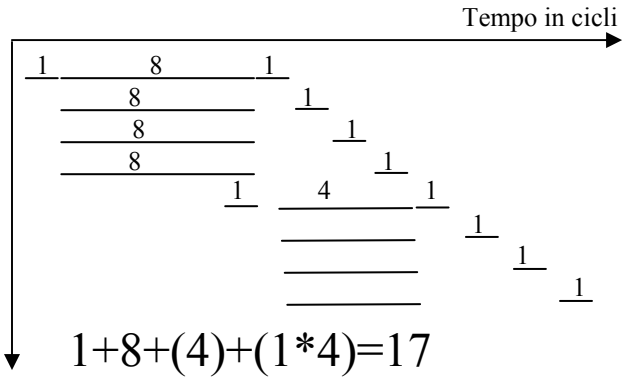
Esempio (può essere esercizio)



Memoria Principale:
chips DRAM. Nella lettura di blocchi,
la prima parola è letta in 8 cicli; le successive in 4; inoltre:
1 ciclo per invio indirizzo a memoria DRAM + 1 ciclo per invio parola a cache



1+8+(7*4)+1=38



1+8+(4)+(1*4)=17

La penalità M passa da 39 a 17 cicli !!!

Continuazione dell'esercizio

Si supponga di avere:

- memoria DRAM come descritto (1+8+1 cicli | 1+4+1 per l'accesso)
- blocchi di 8 parole e memoria interallacciata
- 30% delle istruzioni fanno accesso a memoria dati
- Cache istruzioni e cache dati: h = 95% e 90% rispettivamente

⇒ Calcolare l'incremento di prestazioni [su tempo di accesso] grazie a cache

Nel lucido precedente abbiamo ottenuto M=17

$T_{\text{no cache}} = 1*10 + 0.3*10 = 13$ cicli/istruzione

$T_{\text{cache}} = (0.95*1+0.05*17) + 0.3*(0.9*1+0.1*17) = 2.58$ cicli/istruzione

Quindi: Speedup = 13/2.58 = 5.39

M: diminuita anche utilizzando la tecnica del load-through

h: frequenza di hit può essere migliorata da:

- memoria cache più grande [ma aumentano costi e dimensioni]
- blocchi più grandi (località spaziale),

ma blocchi più grandi aumentano anche penalità di fallimento

[trasferimento più lungo!]

$\Rightarrow$ Cache istruzioni e dati: una sola consente flessibilità maggiore e blocchi grandi,
ma due cache aumentano parallelismo [no criticità strutturale con pipeline]
e tendono a far diminuire tempi d'accesso [dimensioni inferiori]

Uso di due livelli di cache:

Consente di diminuire M (penalità di fallimento): miss in cache primaria
comporta accesso a cache secondaria: importante più h_2 che velocità!

$$T = h_1C_1 + (1-h_1)\underbrace{[h_2C_2 + (1-h_2)*M]}$$

M: tempo di accesso RAM

Si vede che coeff. di M è $(1-h_1)(1-h_2)$

Altri miglioramenti alle prestazioni

- **Buffer di scrittura**:
  - memorizza temporaneamente le richieste di scrittura nella memoria principale
  - permette alla CPU di non dover attendere la scrittura della memoria ogni volta che effettua un'operazione di scrittura: la CPU non dipende immediatamente da un'operazione di scrittura
  - i dati vengono scritti in memoria quando questa non è impegnata in lettura

CASO WRITE-THROUGH

- Per fare operazione di scrittura, la CPU posiziona richiesta bel buffer e non attende l'accesso alla memoria per proseguire
- Ad ogni richiesta di lettura dalla memoria principale, l'indirizzo del dato richiesto viene confrontato con tutti gli indirizzi dei dati presenti nel buffer: se presente, la lettura buffer avviene direttamente dal buffer stesso

CASO WRITE-BACK

- Quando un blocco deve essere eliminato dal sopraggiungere di un altro blocco (che contiene un dato da leggere) va scritto in memoria
- Al posto di aspettare l'operazione di scrittura, il blocco viene posto nel buffer di scrittura evitando l'attesa e servendo direttamente la "lettura"

- **Prelievo anticipato (prefetching)**:
  - Istruzioni di prelievo anticipato (poste dal programmatore/compilatore) forzano il prelievo di un blocco prima che ce ne sia bisogno, evitando un successivo miss di cache. Possibile anche la soluzione hardware sulla base del profilo dei riferimenti alla memoria
  - Svantaggi: allungamento dei programmi [inserimento delle istruzioni di prefetch]
    se il blocco viene eliminato prima di essere usato (per altri fallimenti in lettura) il lavoro è costoso e inutile
  - In ogni caso, richiede cache non bloccanti [mentre viene caricato il blocco, la cache deve permettere accessi in lettura!]
- **Cache non bloccanti (Lockup-free):**
  - Permettono alla CPU di accedere alla cache anche mentre si sta rispondendo ad un fallimento di accesso
  - Per gestire due o più fallimenti, devono contenere registri per tener traccia dei fallimenti in sospeso
  - Utili per favorire il prelievo anticipato + pipeline [superscalari e dinamiche]

Memoria Virtuale

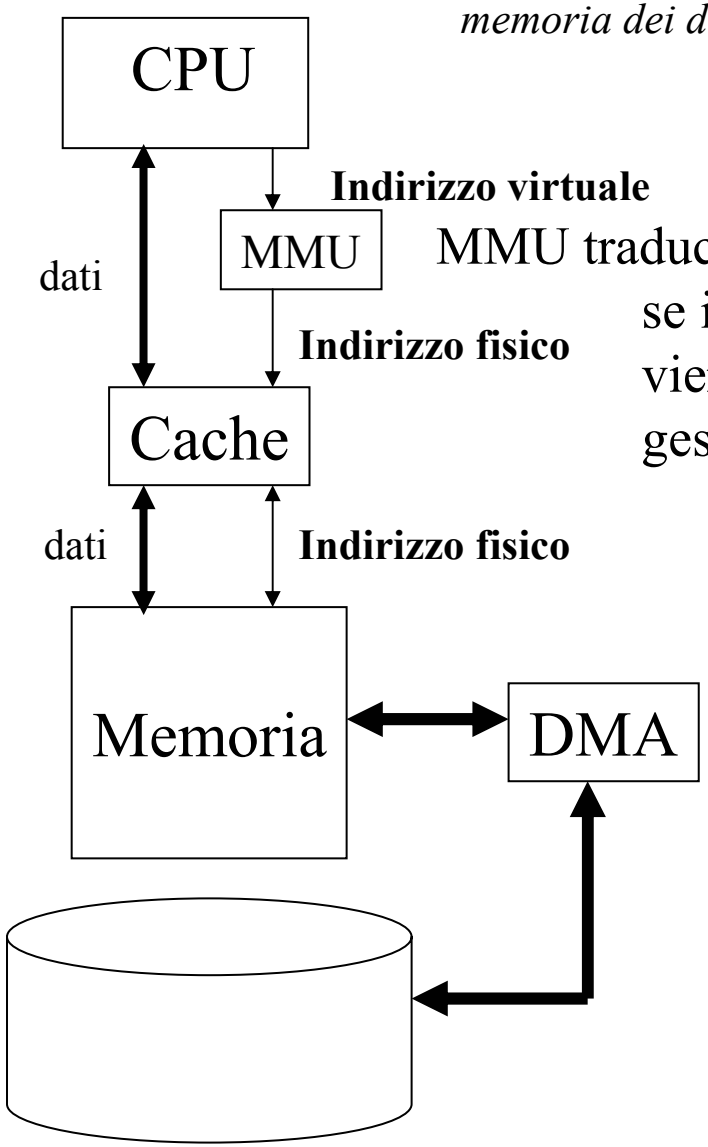
- Tecnica di gestione della memoria usando la memoria principale come 'cache' del disco, sopperendo alle differenze di:
  - velocità [tempo di accesso RAM molto minore]: CPU accede a RAM
  - capacità [capacità RAM molto minore rispetto a disco e a capacità di indirizzamento della CPU]: necessità di "spostare le pagine"
- Mette a disposizione una memoria indirizzabile più grande della memoria principale reale in modo trasparente alla CPU, evitando al programmatore di gestire l'overlay
- Altra ragione (che vedremo in seguito): permettere la gestione automatica di più programmi 'contemporaneamente ' senza interferenze, garantendo la separazione degli spazi di indirizzamento…

Definizione

Indica il *sistema* di circuiti e programmi che gestisce automaticamente la memoria RAM (cache+memoria principale) + memoria secondaria come un unico *sistema* di memoria.

Sistemi a Memoria Virtuale

(*hardware e SO collaborano per gestire uno spazio degli indirizzi più grande di quello di memoria una velocità di trasferimento vicina a quella della cache e per garantire la non invasione di memoria dei differenti processi)*

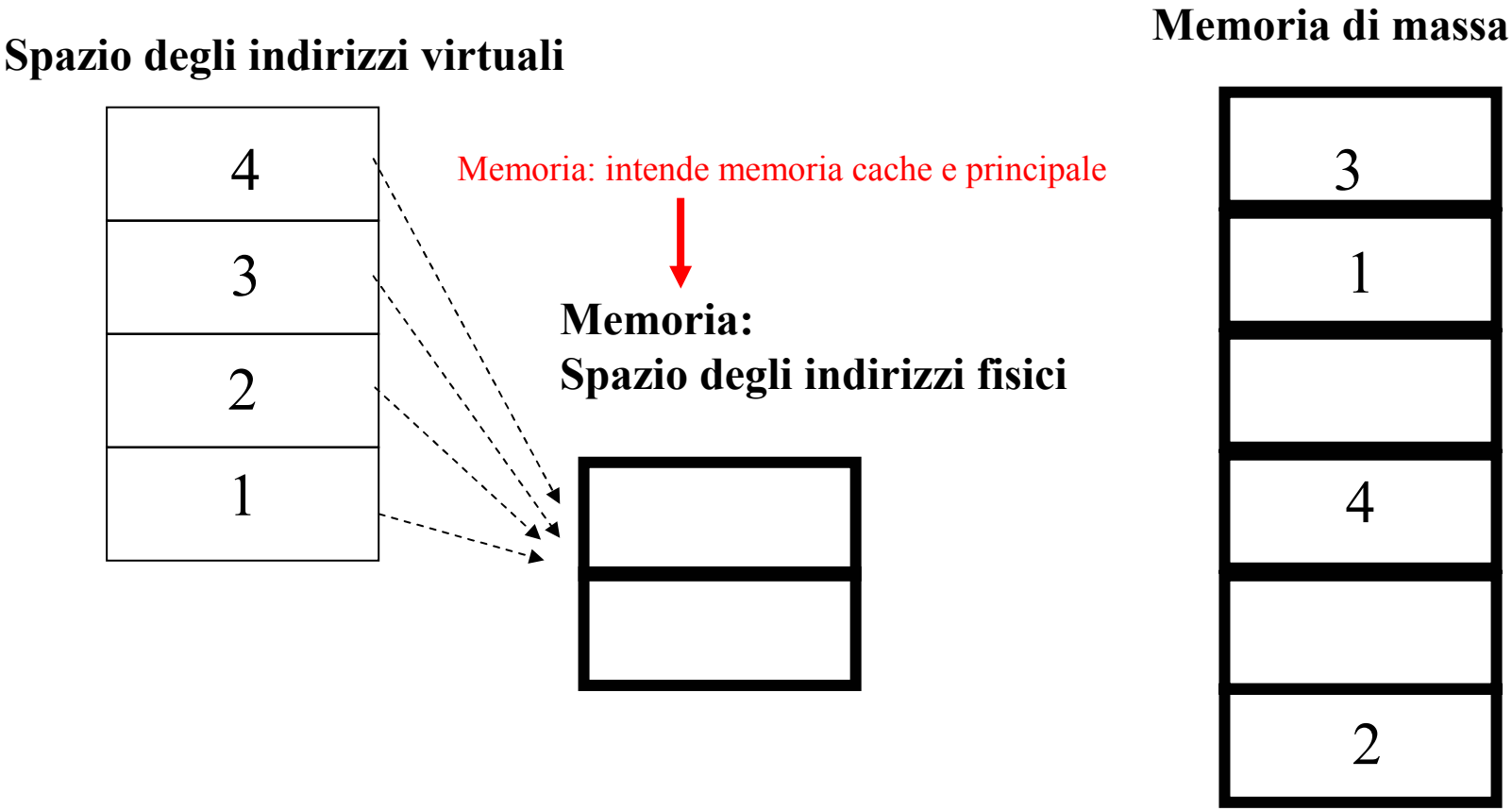


MMU traduce indirizzi virtuali in reali:
se i dati sono in memoria, ok; se sono su disco, viene invocato il Sistema Operativo per gestire il trasferimento dal disco in Memoria

DMA gestisce il trasferimento di pagine: riceve comando e alla fine segnala fine transazione.

Spazio degli indirizzi e Locazioni di memoria

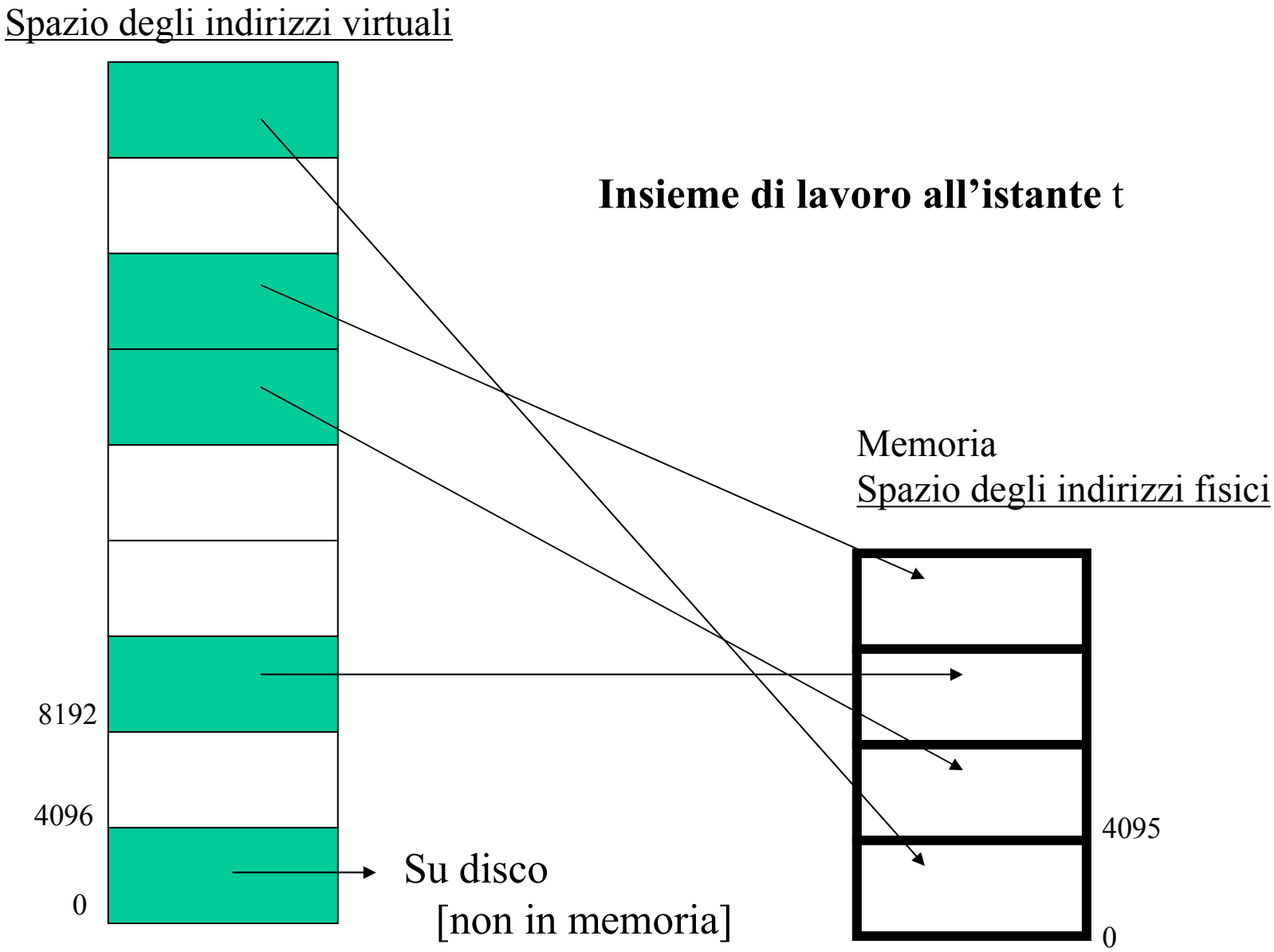
NB: inizialmente consideriamo per semplicità un solo programma in esecuzione



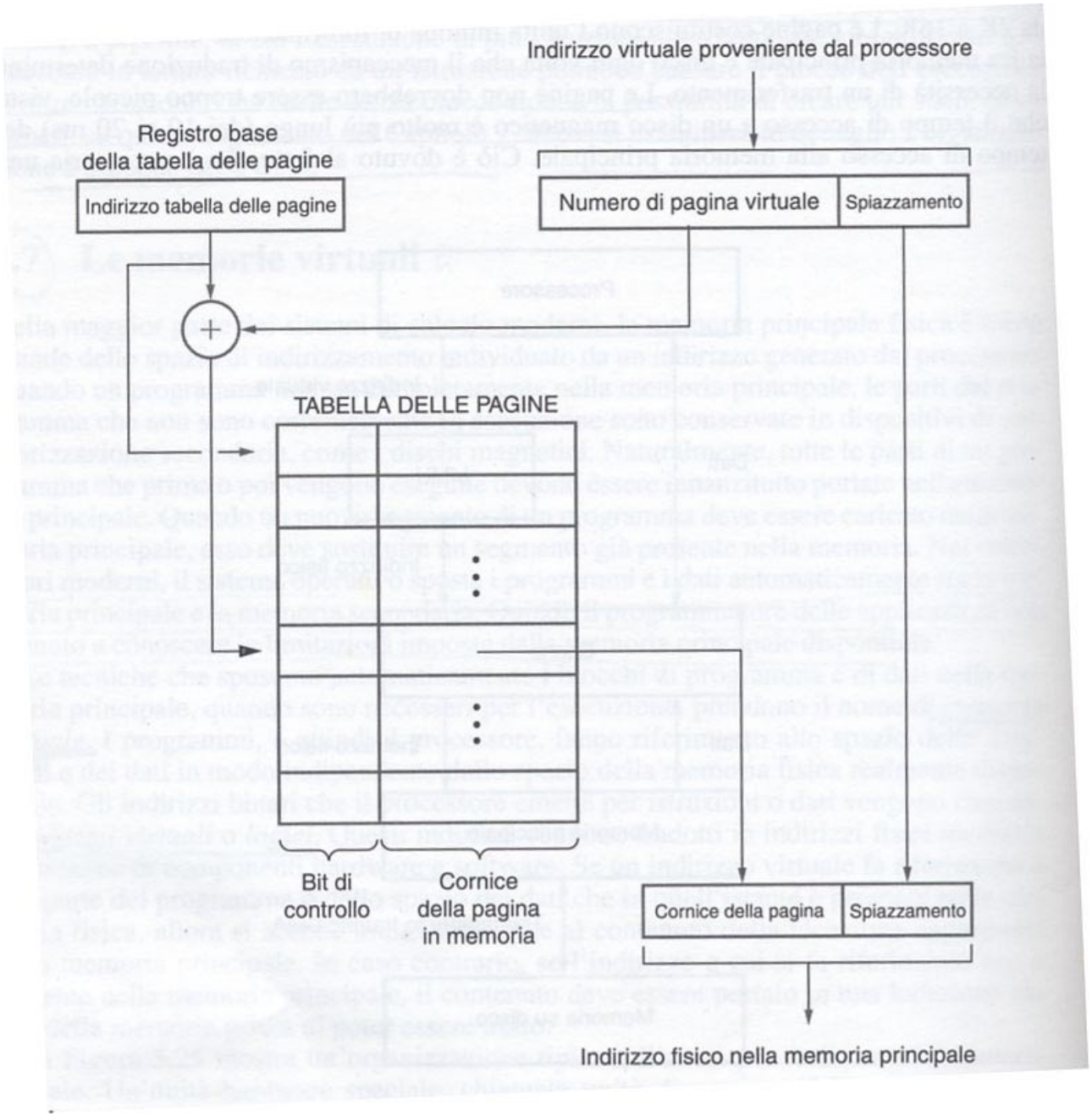
Spazio degli indirizzi: insieme dei possibili indirizzi ad un certo livello.
Dipende dal numero di bit nel campo indirizzi (Virtuale: indirizzi CPU; fisico: indirizzi RAM)

Locazioni di memoria:
l'insieme dei registri effettivamente indirizzabili

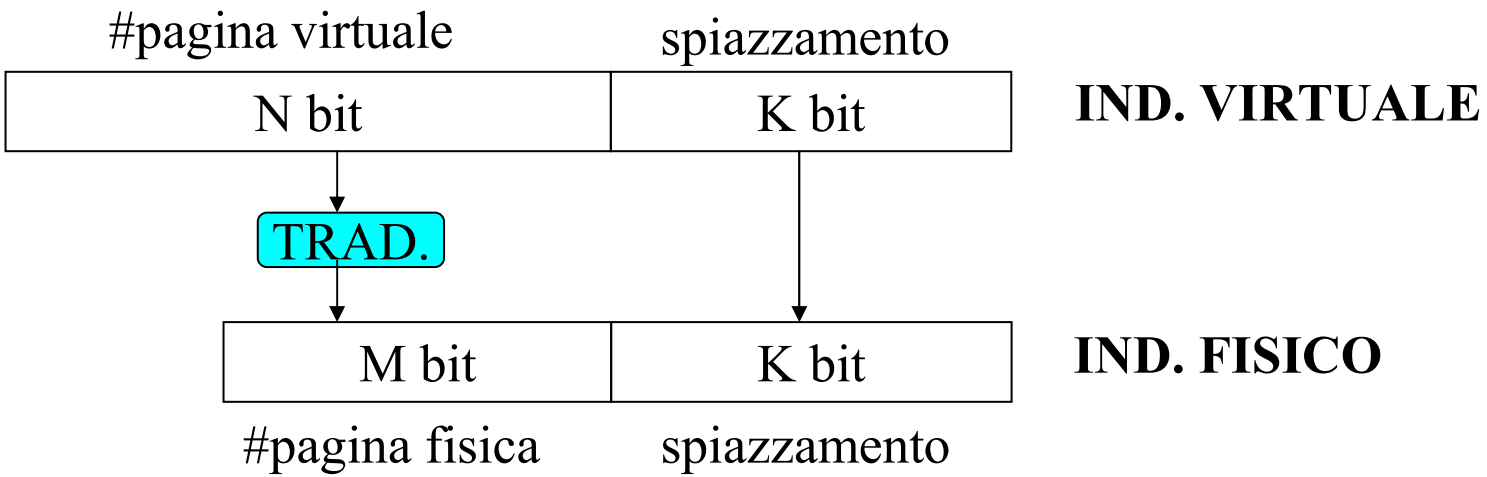
IDEA: programmi e dati composti da "**pagine**" [locazioni consecutive in memoria]:
pagina= unità minima trasferita fra memoria e disco



La traduzione degli indirizzi via tabella delle pagine



• La tabella delle pagine crea l'associazione pagina virtuale → pagina fisica



Spazio virtuale: 2^{N+K} [2^N pagine ciascuna di 2^K bytes]

Spazio fisico: 2^{M+K} [2^M pagine ciascuna di 2^K bytes]

• Ogni volta che c'è un accesso in memoria, consultazione tabella delle pagine:

- l'indirizzo virtuale è presente: traduzione nell'indirizzo fisico
- l'indirizzo virtuale è assente (bit validità a 0): la pagina no in memoria, ma su disco

⇒ MMU solleva eccezione invocando il sistema operativo, che:

- trasferisce una pagina in memoria, eventualmente eliminandone una
- aggiorna di conseguenza la tabella delle pagine
- restituisce il controllo al processo interrotto

NB: se ora si considerano più processi, si vede che il sistema operativo può passare il controllo ad un altro processo mentre DMA provvede a trasferire la pagina!

• RAM è cache rispetto a memoria di massa. Notare però che:

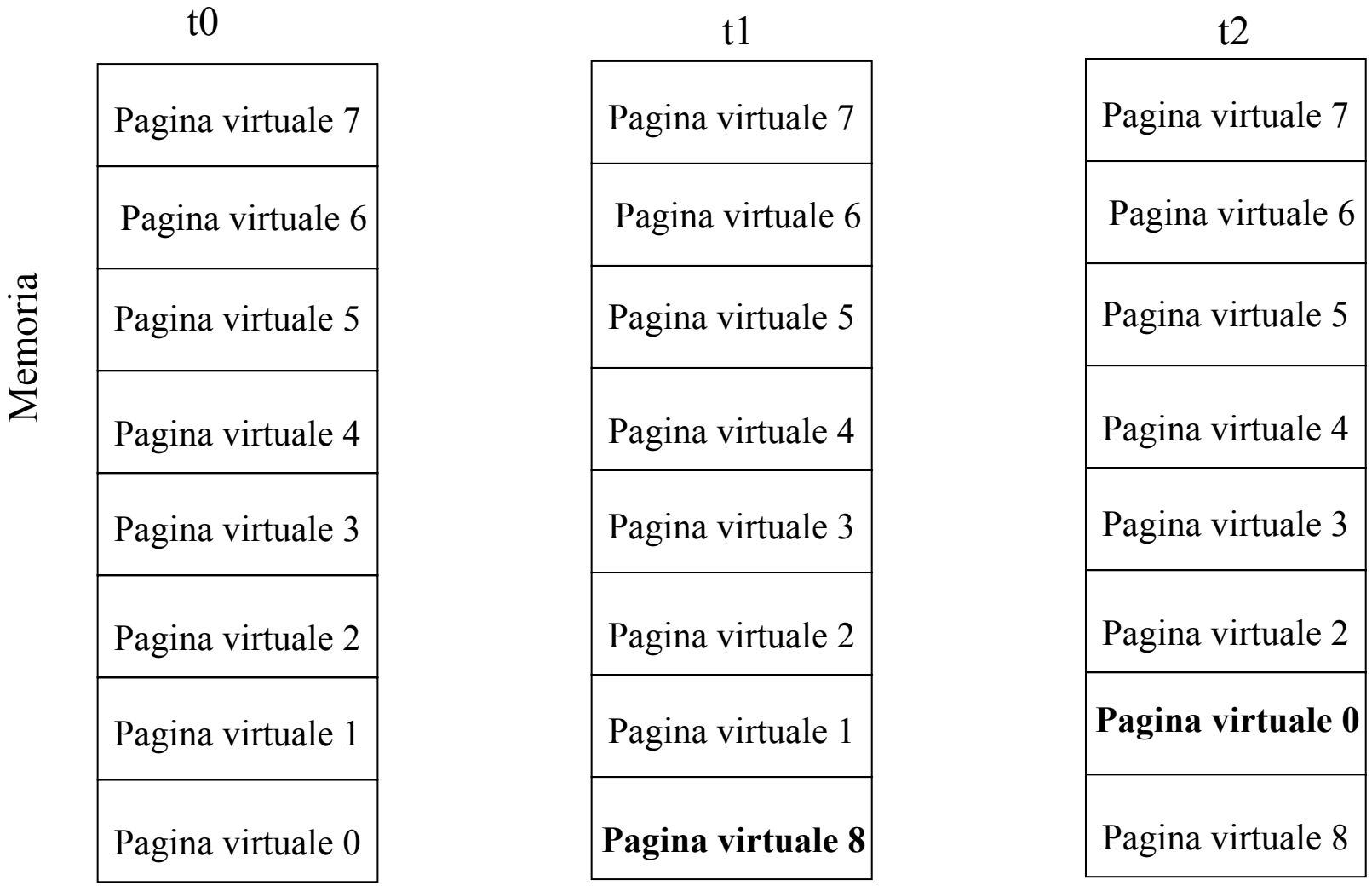
- l'indirizzamento diretto non è praticabile: page fault si pagano troppo cari; quindi la RAM è gestita come cache completamente associativa
- gestione page fault via SW: overhead SW trascurabile rispetto a miss + flessibilità
- è necessaria la tabella per mantenere i riferimenti: sarebbe troppo oneroso confrontare i tag in memoria centrale…
- per minimizzare il numero di page-fault, la sostituzione avviene con politica LRU, di solito approssimata [si hanno molti blocchi in memoria]:
  es. un bit per ogni blocco posto a 1 ad ogni accesso, periodicamente il S.O. li pone tutti a 0
- write-through non è praticabile: tempo di accesso al disco è molto alto
  $\Rightarrow$ un bit di modifica per ogni blocco

In particolare, ogni pagina ha nella tabella i seguenti bit:

- bit di validità [se è 0 la pagina non è in memoria; S.O. può invalidare una pagina senza eliminarla fisicamente dalla memoria]
- bit di modifica
- gruppo di bit che indicano il permesso [in lettura/scrittura]

Quando l'algoritmo LRU è costoso

Un programma esegue un loop che coinvolge 9 pagine
[0-1-…-8 0-1-…-8 0-1-…8 ecc.]



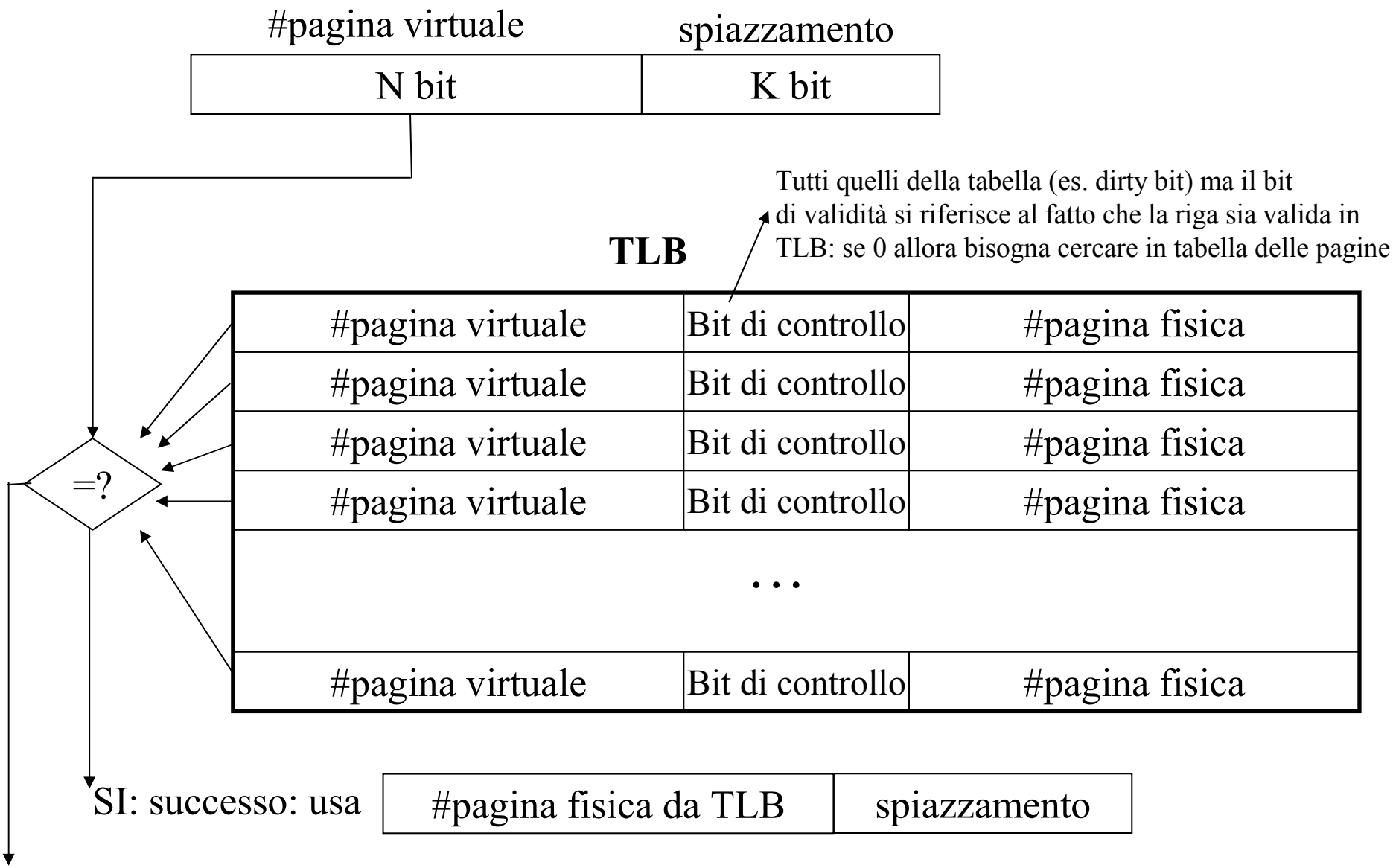
NB: una dose di casualità [come in LRU approssimato] può aiutare…

PROBLEMA

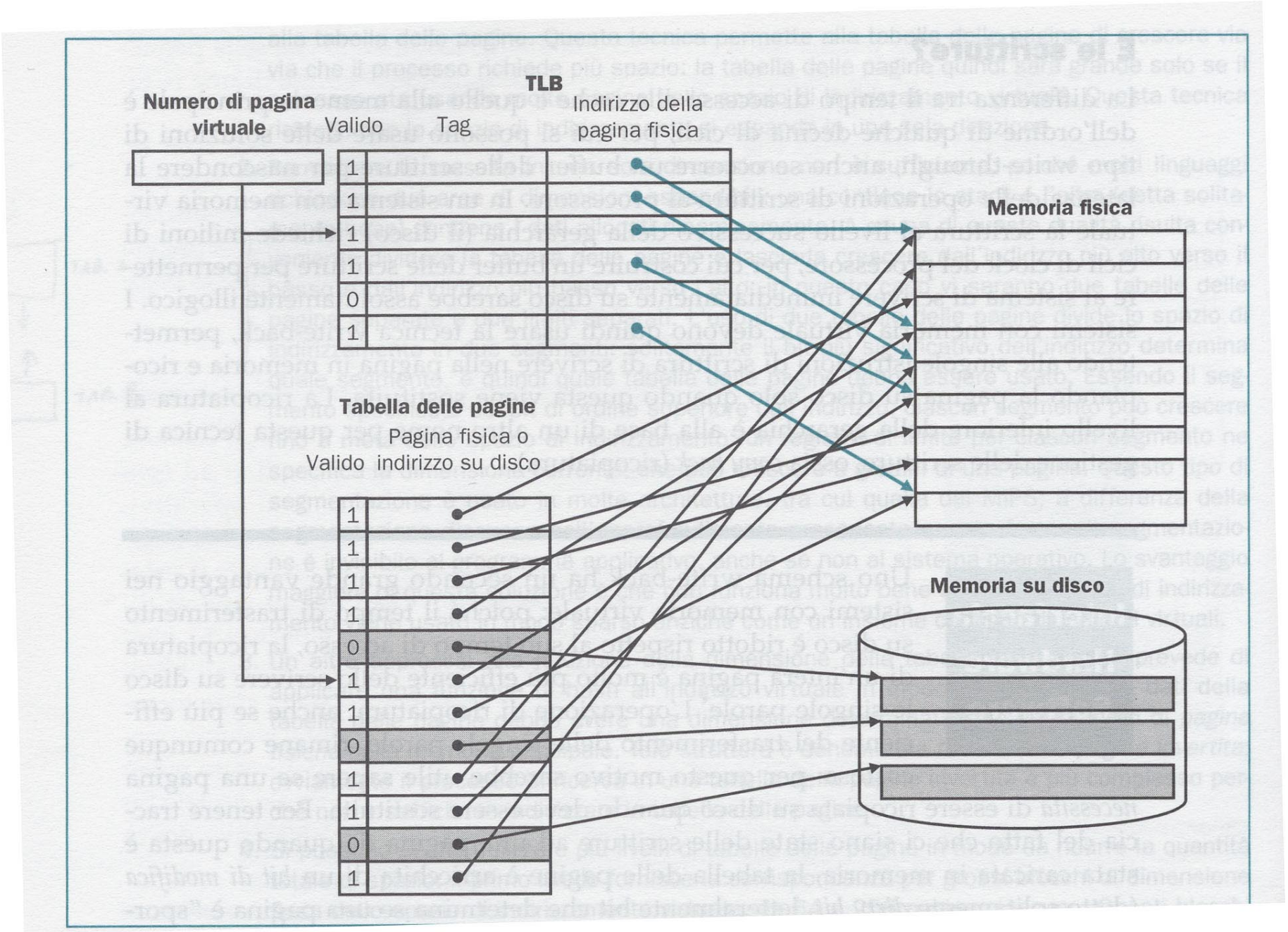
La tabella delle pagine deve essere acceduta ogni volta che si usa un indirizzo virtuale [fetch istruzioni + accesso a memoria dati!]

⇒ Per gli stessi motivi che impongono l'uso di cache, essa deve stare in MMU all'interno del processore…
… ma le dimensioni notevoli non lo permettono… essa è in RAM…

⇒ Uso una cache della tabella delle pagine detta **Translation Lookaside Buffer [TLB]** =
Buffer per la traduzione degli indirizzi

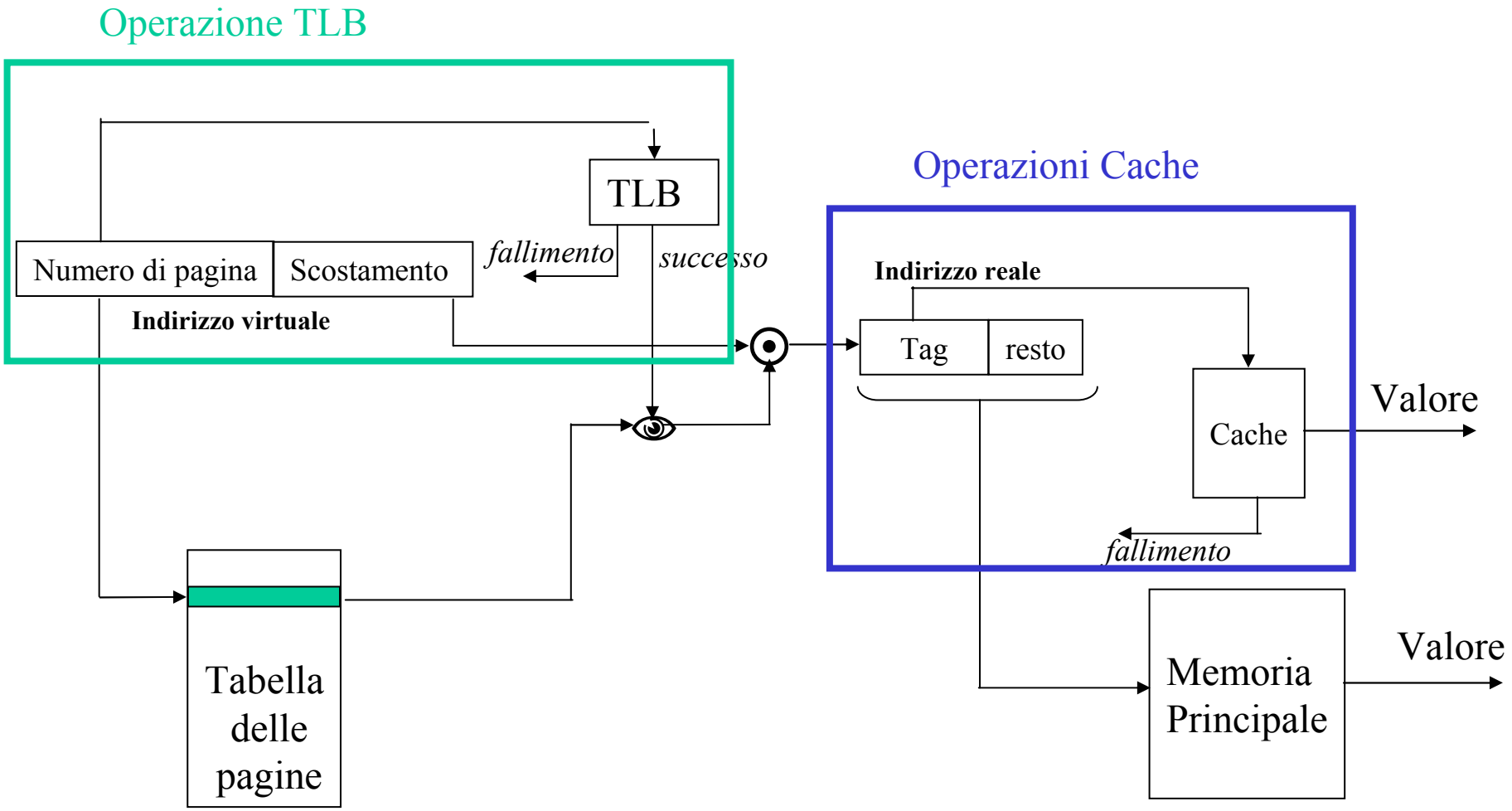


NB: TLB mantiene solo riferimenti che si trovano in memoria fisica, non su disco!

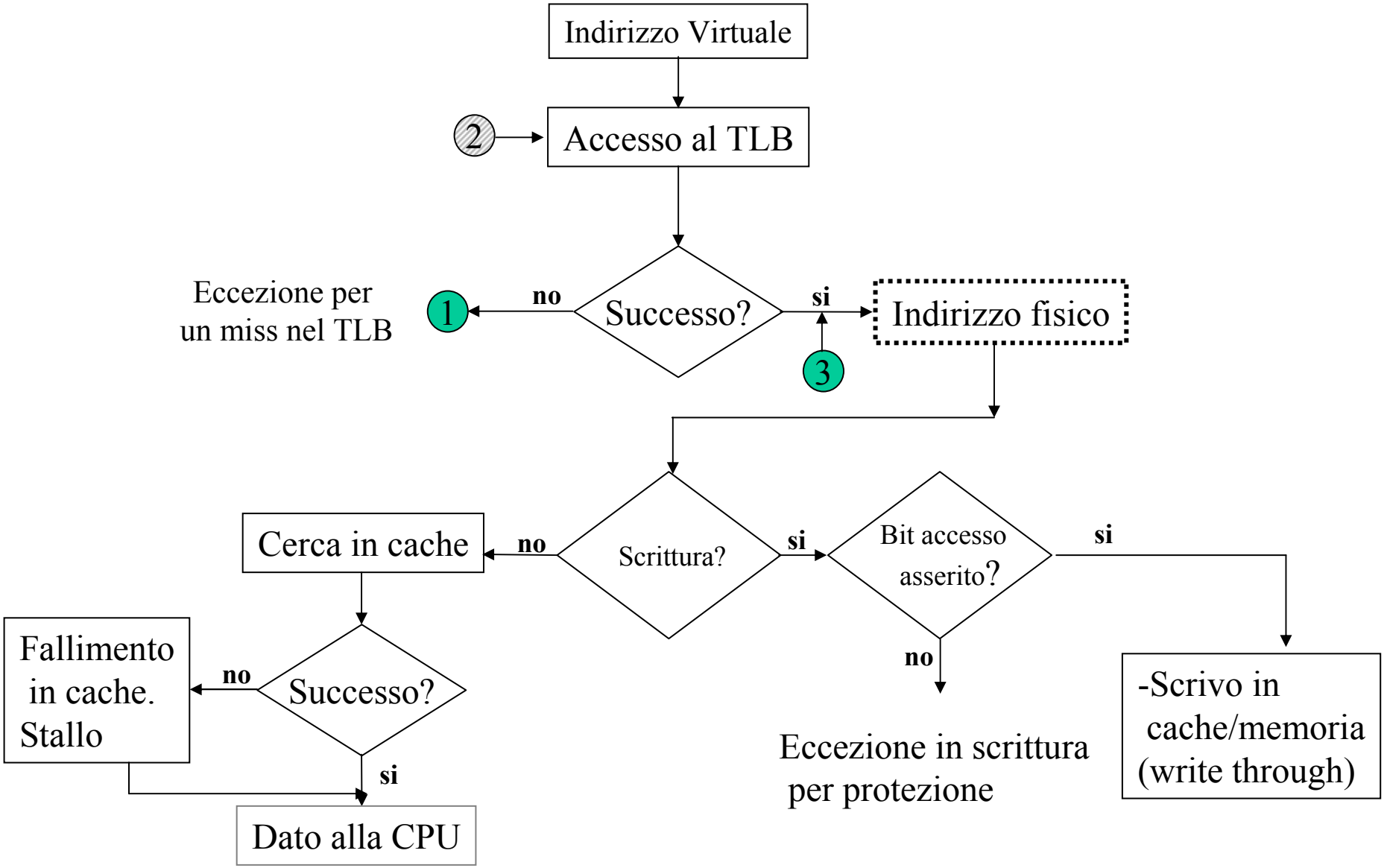


NB: [Valido in TLB] =1 indica valido e "in memoria fisica", 0: miss TLB

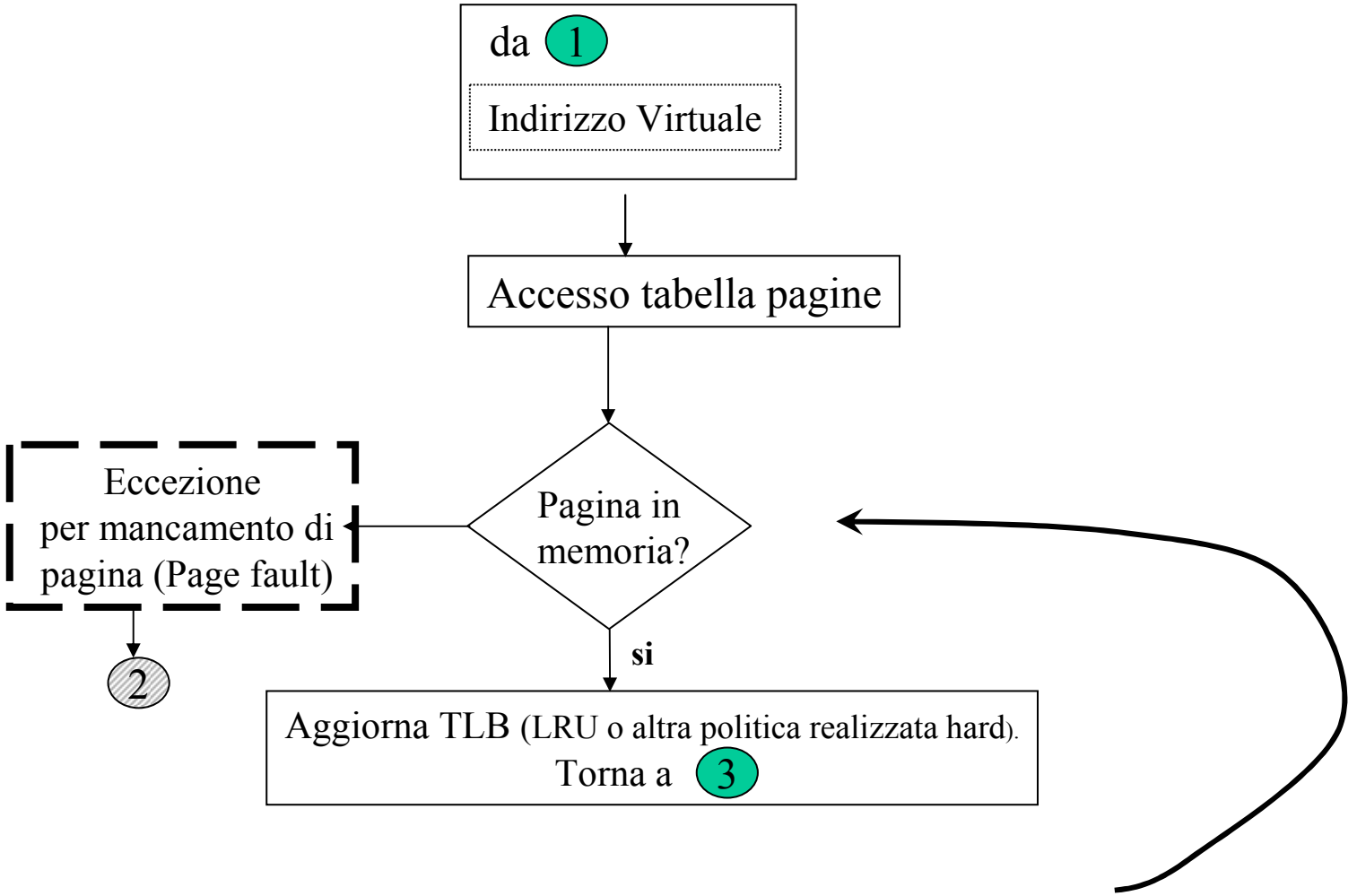
Visione di insieme: Buffer di traduzione e operazioni in cache



Accesso ad una locazione di memoria (ispirato MIPSR2000)



Gestione miss TLB (ispirato MIPSR2000)



NB: un'alternativa è quella di copiare elemento della tabella delle pagine in TLB, senza controllare page fault; se non è valido [pagina è su disco] page fault avverrà dopo, accedendo a TLB. Privilegia miss TLB - hit di pagina

NB: mentre la memoria è sempre gestita come cache completamente associativa, per TLB si usano vari gradi di associatività [nei lucidi precedenti si era implicitamente supposto TLB completamente associativo]

NB2: miss TLB può essere gestito via SW [con eccezione] o via HW; page fault invece è gestito sempre da SW [Sistema Operativo]

Scelta della Dimensione delle Pagine in memoria centrale

A favore di pagina grande:

- minor numero pagine, quindi tabella delle pagine piccola
- trasferimento più economico vs. miss per tempo di accesso del disco: privilegiare la riduzione dei page fault
- TLB più piccolo (numero di pagina virtuale più piccolo)
- riduzione miss TLB

A favore di pagina piccola:

- meno memoria sprecata [evita parti di pagina mai utilizzati]
- per processi piccoli: accelera start-up

Nella pratica, le pagine devono essere abbastanza grandi [32-64 KB] per l'elevato costo di miss dovuto all'alto tempo di accesso al disco: meglio ridurre i page fault e sopportare un tempo di trasferimento più lungo [comunque molto minore del tempo di accesso]

Memoria virtuale e Sistema Operativo

- Sono presenti in realtà più processi
- La memoria fisica è condivisa tra:
  - sistema operativo
  - processo utente 1
  - processo utente 2
  - ecc. ecc.
- PROBLEMI:
  - al momento della compilazione non si conoscono i processi che condivideranno memoria e quindi neppure dove il processo sarà collocato
  - le dimensioni di memoria di tutti i processi [e anche di uno solo] superano quelle della memoria; necessità di condividere la memoria con le pagine "attive" dei processi

SOLUZIONE:

- Ciascun processo [utente o sistema operativo] ha il suo spazio di indirizzamento virtuale, cui si fa riferimento in compilazione, che parte dalla locazione 0 ed è virtualmente illimitato;
- la traduzione da spazio virtuale a fisico è poi realizzata "run-time" con una tabella delle pagine distinta per ogni processo $\Rightarrow$ rilocazione semplificata.

- Lo spazio virtuale del sistema operativo – non quello dei processi- comprende tabelle delle pagine dei processi: il S.O. può intervenire [es. nei page fault] per gestirle.
- Ogni volta che il S.O. rilascia il controllo ad un processo, gli assegna la tabella delle pagine cambiando opportunamente il registro base della tabella delle pagine. L'indirizzo base [che identifica la tabella] fa parte dello "stato" del processo salvato dal S.O. e ripristinato quando si cede il controllo al processo
- Ogni processo utente ha a disposizione uno spazio virtuale illimitato [da 0 in poi] e non deve preoccuparsi della collocazione fisica degli altri processi
- Protezione:
  - il Sistema Operativo assegna pagine fisiche distinte ai processi
  - pagine condivise tra processi ma con bit di protezione
  - i processi non "vedono" le pagine fisiche destinate al sistema operativo
  - in "stato di utente" i programmi non possono eseguire istruzioni privilegiate; in particolare non possono modificare il registro base della tabella pagine
  - passaggio a "stato supervisore" solo via eccezione per invocare S.Operativo

NB: tutte queste cose verranno affrontate nel corso di Sistemi Operativi…

Un problema

- Ogni volta che c'è il cambio di contesto, S.O. cambia la tabella delle pagine (via registro base) ritornando alla tabella del processo cui si cede controllo;
- TLB deve essere "ripulito" [invalidando bit di validità] per evitare che il processo usi ancora le corrispondenze "vecchie" [interferendo con lo spazio fisico del processo precedente]
- Ciò comporta continui miss nel TLB finchè questo non è caricato
- Se il nuovo processo esegue poche istruzioni e magari il controllo ritorna al precedente, TLB è di nuovo "ripulito" ed il processo precedente deve di nuovo ripassare attraverso la tabella delle pagine (miss TLB)
- Soluzione più efficiente:
  - tag del TLB contiene un campo addizionale: *identificatore di processo*
  - un registro (aggiornato da sistema operativo) indica il processo corrente
  - TLB non ripulito, ma hit si ha solo se coincidono processo corrente e identificatore di processo