Gerarchie di memoria

Salvatore Orlando

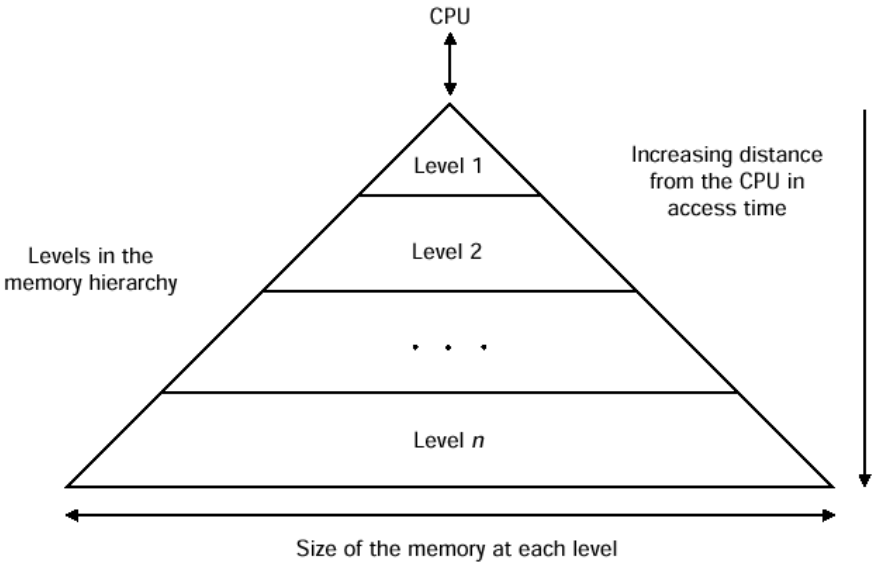
Gerarchie di memoria

- I programmatori, per memorizzare i loro programmi e dati, necessiterebbero di memorie molto veloci e capienti
- La tecnologia permette solo di costruire
  - memorie grandi e lente, ma poco costose
  - memorie piccole e veloci, ma molto costose

- Conflitto tra
  - esigenze programmatori
  - vincoli tecnologici

- Soluzione: gerarchie di memoria
  - piazziamo memorie veloci vicino alla CPU
    - per non rallentare la dinamica di accesso alla memoria tramite fetch istr. e load/store
  - man mano che ci allontaniamo dalla CPU
    - memorie sempre più lente e capienti
  - soluzione compatibile con i costi …..
  - meccanismo dinamico per spostare i dati tra i livelli della gerarchia

Gerarchie di memoria



- Al livello 1 poniamo la memoria più veloce (piccola e costosa)
- Al livello n poniamo la memoria più lenta (grande ed economica)

- Scopo gerarchia e delle politiche di gestione delle memorie
  - dare l'illusione di avere a disposizione una memoria
    - *grande* (come al livello n) e *veloce* (come al livello 1)

Costi e capacità delle memorie

- Dati 1997

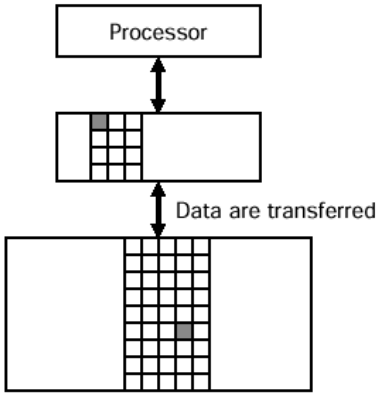
  - SRAM
    - latenze di accesso di 2-25 ns
    - costo da $100 a $250 per MB
    - tecnologia usata per i livelli più vicini all CPU (cache)

  - DRAM
    - latenze di accessi di 60-120 ns
    - costo da $5 a $10 per MB
    - tecnologia usata per la cosiddetta memoria principale

  - Dischi
    - latenze di accesso di 10-20 milioni di ns (10-20 ms)
    - costo da $.10 a $.20 per MB
    - memoria stabile usata per memorizzare file
    - memoria usata anche per contenere l'immagine (text/data) dei programmi in esecuzione => memoria (principale) virtuale

Illusione = memoria grande e veloce !?

- All'inizio i nostri dati e i nostri programmi sono memorizzati nel livello n (più capiente e lenta)
- I blocchi di memoria man mano *riferiti* vengono fatti fluire verso
  - i livelli più alti (memorie più piccole e veloci), più vicini alla CPU



- Problema:
  - *Cosa succede se un blocco riferito è già presente nel livello 1 (più alto) ?*
  - La CPU può accedervi direttamente, ma abbiamo bisogno di un meccanismo per trovare il blocco all'interno del livello 1 !
- Problema:
  - *Cosa succede se i livelli più alti sono pieni ?*
  - Dobbiamo implementare una politica di rimpiazzo dei blocchi !

Terminologia

- Anche se i trasferimenti tra i livelli avvengono sempre in blocchi, questi hanno dimensione diversa, e (per ragioni storiche) nomi diversi
  - abbiamo blocchi più piccoli ai livelli più alti (più vicini alla CPU)
  - es. di nomi: blocco di cache e pagina
- Hit (Successo)
  - quando il blocco cercato a livello i è stato individuato
- Miss (Fallimento)
  - quando il blocco cercato non è presente al livello i
- Hit rate (%)
  - frequenza di Hit rispetto ai tentativi fatti per accedere blocchi al livello i
- Miss rate (%)
  - frequenza di Miss rispetto ai tentativi fatti per accedere blocchi al livello i
- Hit Time
  - latenza di accesso di un blocco al livello i in caso di Hit
- Miss Penalty
  - tempo per copiare il blocco dal livello inferiore

Località

- L'illusione offerto dalla gerarchia di memoria è possibile in base al:

  Principio di località

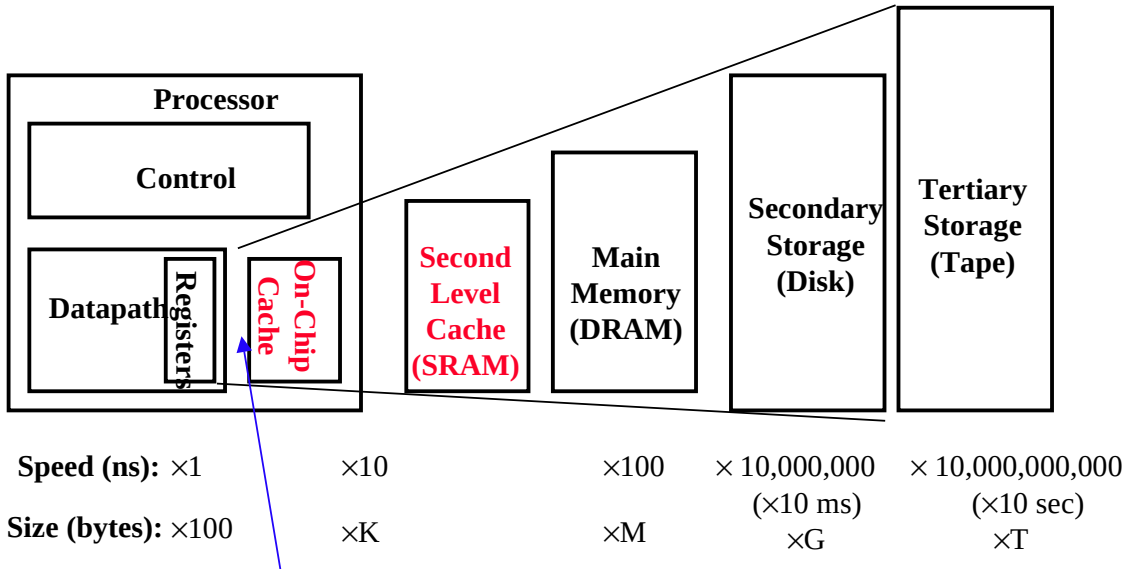
- Se un elemento (es. word di memoria) è riferito dal programma
  - esso tenderà ad essere riferito ancora, e presto ⇐ *Località temporale*
  - gli elementi ad esse vicini tenderanno ad essere riferiti presto ⇐ *Località spaziale*

- In altri termini, in un dato intervallo di tempo, i programmi accedono una relativamente piccola porzione dello spazio di indirizzamento totale

- La località permette il funzionamento ottimale delle gerarchie di memoria
  - aumenta la probabilità di *riusare* i blocchi, precedentemente spostati ai livelli superiori, riducendo il *miss rate*

Cache

- E' il livello di memoria (SRAM) più vicino alla CPU (oltre ai Registri)



Registri: livello di memoria più vicino alla CPU

Movimenti tra Cache ↔ Registri gestiti a sw dal compilatore / programmatore assembler

Cache e Trend tecnologici delle memorie

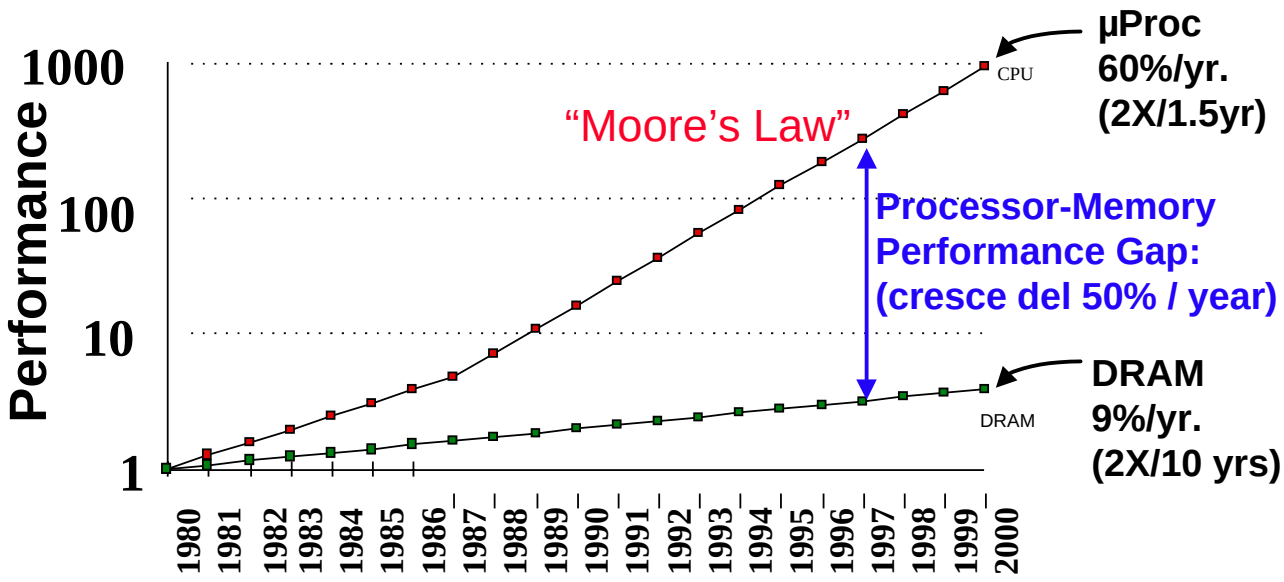
	Capacità	Velocità (riduz. latenza)
Logica digitale:	2x in 3 anni	2x in 3 anni
DRAM:	4x in 3 anni	2x in 10 anni
Dischi:	4x in 3 anni	2x in 10 anni

Anno	DRAM Dimensione	Latenza accesso
1980	64 Kb	250 ns
1983	256 Kb	220 ns
1986	1 Mb	190 ns
1989	4 Mb	165 ns
1992	16 Mb	145 ns
1995	64 Mb	120 ns
1998	256 Mb	100 ns

1000:1! 2:1!

Accesso alla memoria = Von Neumann bottleneck

Processor-DRAM Memory: Performance Gap



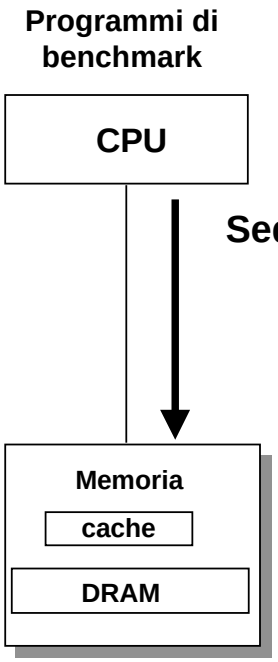
- Il gap di prestazioni cresce sempre di più (50% all'anno)
  - l'accesso alla memoria è oggi il *collo di bottiglia* del modello computazionale di Von Neumann, su cui sono basati i computer moderni

Cache

- L'uso di cache grandi e multivello è necessario per
  - tentare di risolvere il *von Neumann bottleneck,* il problema costituito dalle memorie DRAM
    - sempre più capienti
    - ma sempre meno veloci, rispetto agli incrementi di prestazione delle CPU (micoprocessori)

- Gestione movimenti di dati tra livello cache e livelli sottostanti (Main memory)
  - realizzata dall'hardware

Progetto di un sistema di cache



Sequenza di riferimenti alla memoria:
<op,addr>, <op,addr>,<op,addr>,<op,addr>, . . .

op: i-fetch, read (load), write (store)

Obiettivo del progettista:

Ottimizzare l'organizzazione della memoria in modo da minimizzare il tempo medio di accesso alla memoria per carichi tipici (per sequenze di accesso tipiche)

Ovvero, aumentare il cache hit rate

Problemi di progetto di una cache

- **Dimensionamenti**
  - **size del blocco e numero di blocchi nella cache**
- **Necessario fissare una funzione di mapping tra**
  - **Indirizzo Memoria → Identificatore blocco**
- **Sulla base del mapping, è possibile rispondere a domande quali:**
  - **Come faccio a sapere se un blocco è presente in cache ?**
  - **Se un blocco è presente, come faccio a individuarlo ?**
  - **Se un blocco non è presente, e devo recuperarlo dalla memoria a livello inferiore, dove lo scrivo in cache?**
- **Un problema frequente da affrontare è il seguente**
  - **se il blocco da portare in cache deve essere scritto (sulla base della funzione di mapping) sopra un altro blocco già presente in cache (conflitto), cosa faccio del vecchio blocco ?**

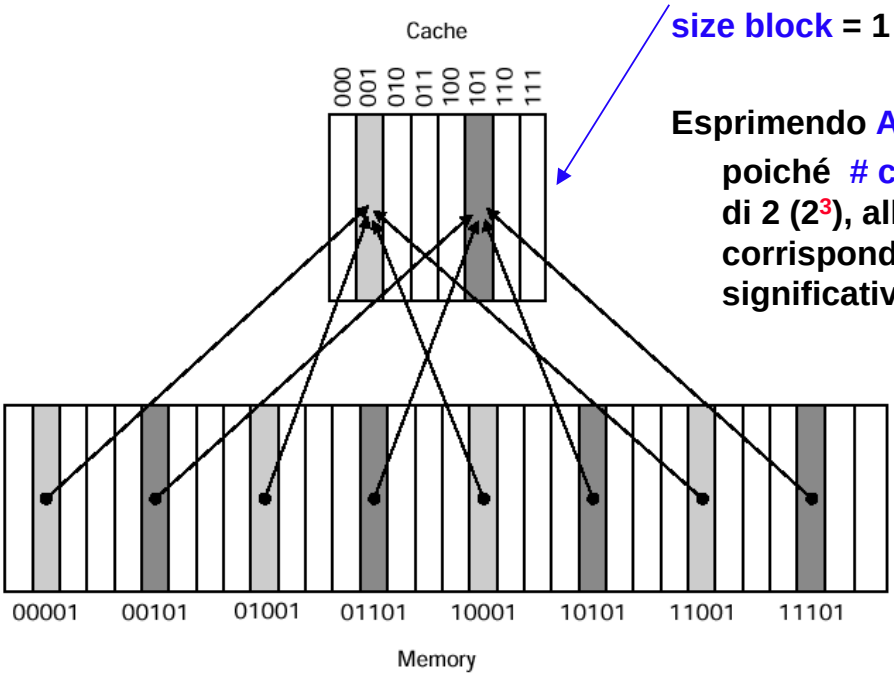
- **Come faccio a mantenere la coerenza tra i livelli ?**
  - **Write through (scrivo sia in cache che in memoria)**
  - **Write back (scrivo in memoria solo quando il blocco in cache deve essere rimpiazzato)**

Caso semplice: cache ad accesso diretto

- **Mapping: Cache block Index = Address mod # cache blocks**

 # cache blocks = 8 = 2^3

size block = 1 B



Esprimendo Address in binario:

poiché # cache blocks è una potenza di 2 (2^3), allora Cache block Index corrisponde ai 3 bit meno significativi di Address

Cache diretta e blocchi più grandi

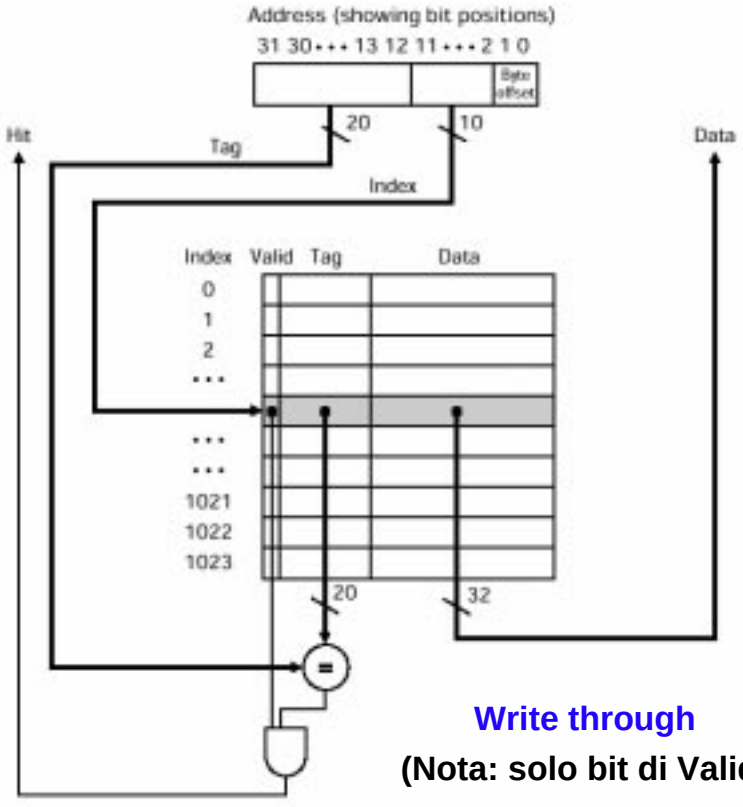
- Ricordiamo che abbiamo un indirizzamento al Byte, per cui, per blocchi più grandi di 1 B:
  - Address diversi possono corrispondere allo stesso Cache block
- Considera che le dimensioni dei blocchi sono potenze di 2
  - Block size = 4, 8, 16, o 32 B
  - I bit *meno significativi* dell'Address diventano *byte offset del blocco*

- Block Address : indirizzamento al blocco (invece che al Byte)
  - Block Address = Address / Block size
  - In binario, si ottiene shiftando a destra Address per un numero *n* di bit, dove $n = log_2(Block\ size)$
  - Questi n bit costituiscono il *byte offset*

- Nuova funzione di Mapping :
  Block Address = Address / Block size
  Cache block Index = Block Address mod # cache blocks

Cache ad accesso diretto (vecchio MIPS)

- *Byte offset*
  - $n = log_2(Block\ size) = 2$
- *INDEX*
  - *corrisponde ai* $log_2(1024) = 10$ bit meno significativi del Block Address
  - Block Address ottenuto da Address rimuovendo gli n bit del byte offset
- TAG
  - parte alta dell'Address, da memorizzare in cache assieme al blocco
  - serve a determinare l'Address originale del blocco memorizzato
- Valid
  - il blocco è significativo



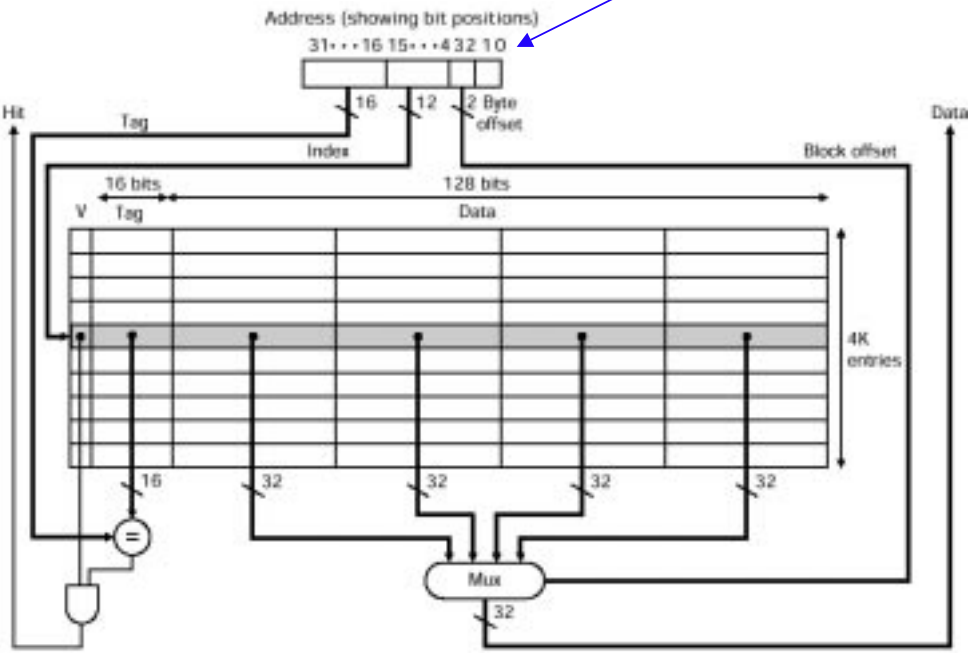
Write through

(Nota: solo bit di Valid)

Blocco più grande di una word

- Approccio vantaggioso per ridurre il Miss rate se abbiamo
  - località spaziale
- Infatti, se si verifica un Miss
  - si carica un blocco grosso
  - se sono probabili accessi spazialmente vicini, questi cadono nello stesso blocco
    ⇒ Hit

Offset (n=$\log_2$ 16=4) suddiviso in due parti
Block offset (2 b) : seleziona word
Byte offset (2 b) : seleziona byte



Hits vs. Miss

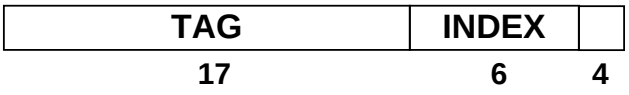
- Read Hit
  - accesso alla memoria con il massimo della velocità
- Read Miss
  - il controllo della CPU deve mettere in stallo la CPU (cicli di attesa, con registri interni immutati), lettura del blocco dalla memoria in cache, ripresa dell'istruzione con load del blocco
- Write Hit
  - write through: scrive sulla cache e in memoria
  - write back: scrive solo sulla cache
- Write Miss
  - con politica *write*-back, stallo della CPU (cicli di attesa), lettura del blocco dalla memoria in cache, ripresa dell'istruzione e store nel blocco letto
  - con politica *write-through,* non è necessario ricopiare il blocco in cache prima di effettuare la scrittura

Ottimizzazioni

- Le scritture possono porre problemi, soprattutto per politica *write through*
  - *Write buffer* come memoria tampone tra cache e memoria
  - Se la memoria non è pronta a ricevere i blocchi scritti, blocchi sono scritti temporaneamente nel *write buffer*, in attesa della scrittura asincrona in memoria
- Write miss
  - Se il blocco da scrivere (store) è grande quanto la word, in caso di write miss non è necessario ricopiare il blocco dalla memoria in cache, anche se la politica è write-back
    - scriviamo direttamente il nuovo blocco nella cache, e non paghiamo il costo del *miss penalty*

Esempio

- Cache con 64 blocchi
- Blocchi di 16B
- Se l'indirizzo è di 27 bit, com'è composto?
  - INDEX deve essere in grado di indirizzare 64 blocchi: $SIZE_{INDEX} = \log_2(64)=6$
  - BLOCK OFFSET (non distinguiamo tra byte e block offset) deve essere in grado di indirizzare 16B: $SIZE_{BLOCK\ OFFSET} = \log_2(16)=4$
  - TAG corrisponde ai rimanenti bit (alti) dell'indirizzo: $SIZE_{TAG} = 27 - SIZE_{INDEX} - SIZE_{OFFSET} = 17$

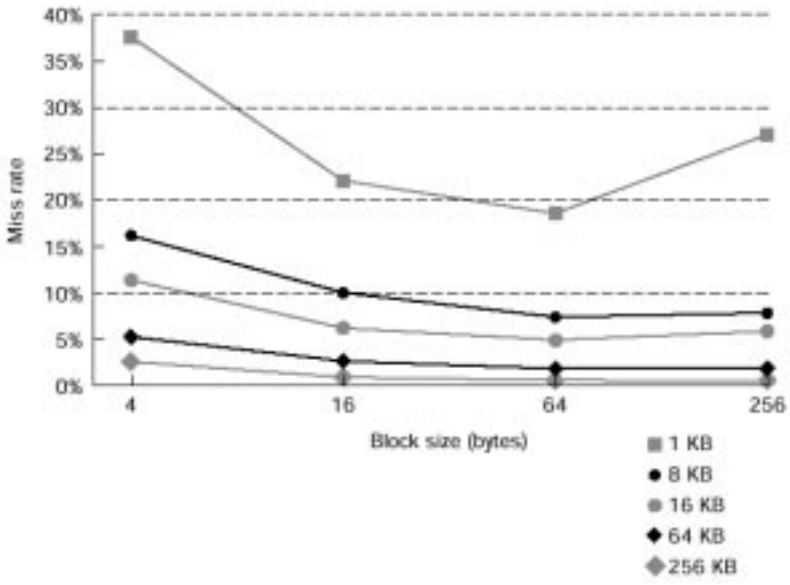


- Qual è il blocco (INDEX) che contiene il byte all'indirizzo 1201 ?
  - Trasformo l'indirizzo al Byte nell'indirizzo al blocco: 1201/16 = 75
  - L'offset all'interno del blocco è: 1201 % 16 = 1
  - L'index viene determinato con l'operazione di modulo: 75 % 64 = 11
    ⇒ il blocco è il 12º (INDEX=11), il byte del blocco è il 2º (OFFSET=1)

Costo dei miss

- Aumentare la dimensione dei blocchi
  - può diminuire il *miss* rate, in presenza di località spaziale
  - aumenta il *miss penalty*
- Quanto costa il miss ?
  - è un costo che dipende (parzialmente) dalla dimensione del blocco:
    - Costo miss = Costante + Costo proporzionale al block size
    - La Costante modella i cicli spesi per inviare l'indirizzo e attivare la DRAM
    - Ci sono varie organizzazioni della memoria per diminuire il costo di trasferimento delle varie parole (tra cui l'uso di SRAM)
- In conclusione
  - costo dei miss per blocchi grandi non è molto maggiore del costo dei miss per blocchi piccoli
- Allora, perché non si usano comunque blocchi grandi invece che piccoli ?
  - esiste un tradeoff !!

Aumento del block size



- Frequenza di miss in genere diminuisce all'aumentare della dimensione del blocco ⇐ vantaggio dovuto alla località spaziale !!
- Se il blocco diventa troppo grande, la località spaziale diminuisce, e per cache piccole aumenta la frequenza di miss a causa di conflitti (blocchi diversi caratterizzati dallo stesso INDEX)

  ⇐ aumenta la competizione nell'uso della cache !!

Aumento del block size

Program	Block size in words	Instruction miss rate	Data miss rate	Effective combined miss rate
gcc	1	6.1%	2.1%	5.4%
	4	2.0%	1.7%	1.9%
spice	1	1.2%	1.3%	1.2%
	4	0.3%	0.6%	0.4%

- Nota che aumentando la dimensione del blocco, la riduzione più marcata, soprattutto per gcc, si ha per l'Instruction Miss Rate
  - la località spaziale è maggiore per la lettura delle istruzioni
- Per blocchi di una sola parola
  - write miss non conteggiati

Prestazioni

- Modello semplificato:

  CPU time = (execution cycles + stall cycles) $\times$ cycle time

  stall cycles = IC $\times$ miss ratio $\times$ miss penalty

- Il miss ratio (ed anche gli stall cycles) possono essere distinti in
  - instruction miss ratio (lettura istruzioni)
  - write miss ratio (store)
  - read miss ratio (load)

  considerati assieme: data miss ratio
- Per il miss penalty possiamo semplificare, considerando un penalty unico per scriture/letture

- Per migliorare le prestazioni, dobbiamo
  - diminuire il miss ratio e/o il miss penalty

- Cosa succede se aumentiamo il block size?

  diminuisce (per cache abastanza grandi) il miss rate, ma aumenta (di poco) il miss penalty

Esempio (1)

- Conoscendo
  miss penalty, instr. miss ratio, data miss ratio,
  CPI ideale (senza considerare l'effetto della cache)
  è possibile calcolare di quanto rallentiamo rispetto al caso ideale (memoria ideale)

- In altri termini, è possibile riuscire a conoscere il CPI reale:
  - $CPI_{actual} = CPI_{ideale}$ + cycle/istr dovuti agli stalli

- Programma gcc:
  - instr. miss ratio = 2%
  - data miss ratio = 4%
  - numero lw/sw = 36% IC
  - $CPI_{ideal} = 2$
  - miss penalty = 40 cicli

Esempio (2)

- Cicli di stallo dovuti alle *instruction miss*
  - instr. miss ratio $\times$ IC $\times$ miss penalty = 0.02 $\times$ IC $\times$ 40 = 0.8 $\times$ IC
- Cicli di stallo dovuti ai *data miss*
  - data miss ratio $\times$ num. lw/sw $\times$ miss penalty = 0.04 $\times$ 0.36 $\times$ IC $\times$ 40 = 0.58 $\times$ IC
- Cicli di stallo totali dovuti ai miss = 1.38 $\times$ IC
- Numero di cicli totali
  - $CPI_{ideal} \times$ IC + Cicli di stallo totali = 2 $\times$ IC + 1.38 $\times$ IC = 3.38 $\times$ IC
- CPI_{actual} = Numero di cicli totali / IC = (3.38 $\times$ IC) / IC = 3.38

- Per calcolare lo speedup basta confrontare i CPI, poiché IC e Frequenza del clock sono uguali:
  - Speedup = CPI_{actual} / CPI_{ideal} = 3.38 / 2 = 1.69

Considerazioni

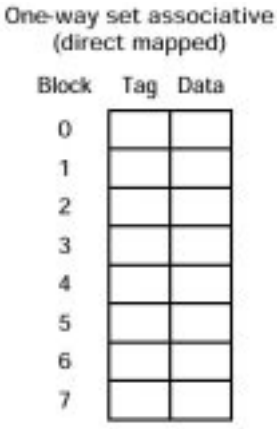
- **Cosa succede se velocizzo la CPU e lascio immutato il sottosistema di memoria?**
  - **il tempo per risolvere i miss è lo stesso**
  - **se ragioniamo in percentuale rispetto al tempo di CPU ideale, questo tempo aumenta !!**

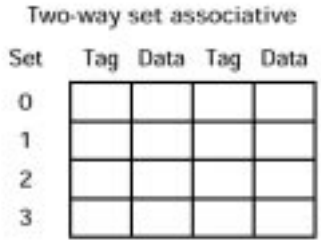
- **Posso velocizzare la CPU in 2 modi:**
  - **cambio l'organizzazione interna**
  - **aumento la frequenza di clock**

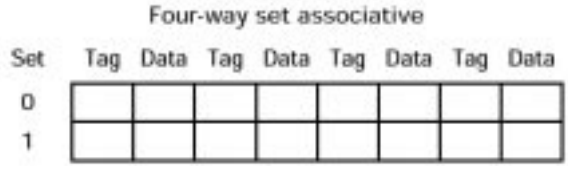
- **Se cambio l'organizzazione interna, diminuisco il CPI_{ideal}**
  - **purtroppo miss rate e miss penalty rimangono invariati, per cui rimangono invariati i cicli di stallo totali dovuti ai miss**
- **Se aumento la frequenza**
  - **il CPI_{ideal} rimane invariato, ma aumentano i cicli di stallo totali dovuti ai miss**
  - **infatti, il tempo per risolvere i miss è lo stesso, ma il numero di cicli risulta maggiore perché i cicli sono più corti !!**

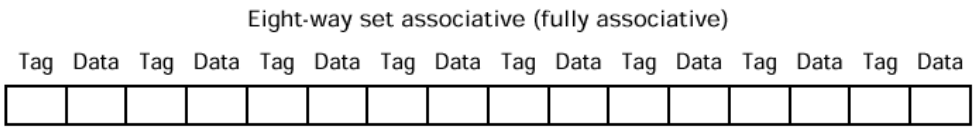
Diminuiamo i miss con l'associatività

- ***Diretta***
  - **ogni blocco di memoria *associato* con un solo possibile blocco della cache**
  - **accesso sulla base dall'indirizzo**
- ***Completamente associativa***
  - **ogni blocco di memoria *associato* con un qualsiasi blocco della cache**
  - **accesso non dipende dall'indirizzo (bisogna cercare in ogni blocco)**
- ***Associativa su insiemi***
  - **compromesso**



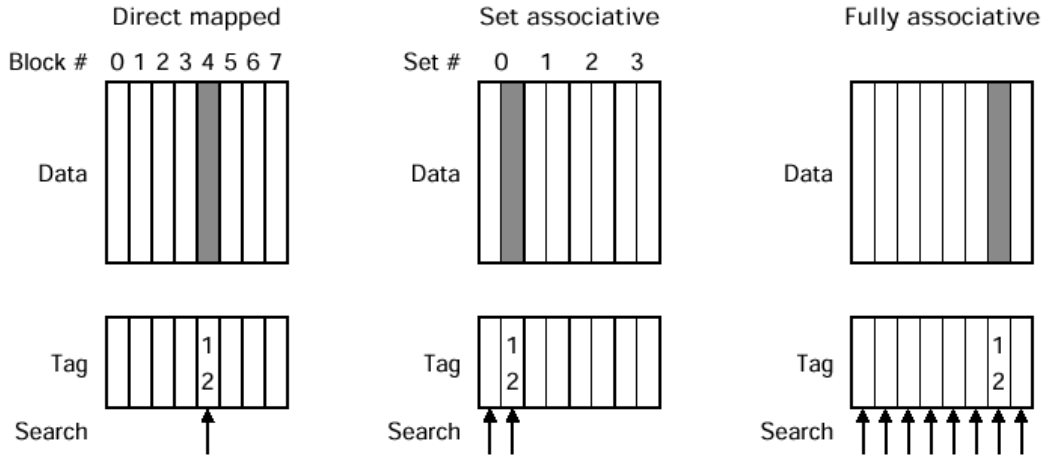






Set associative

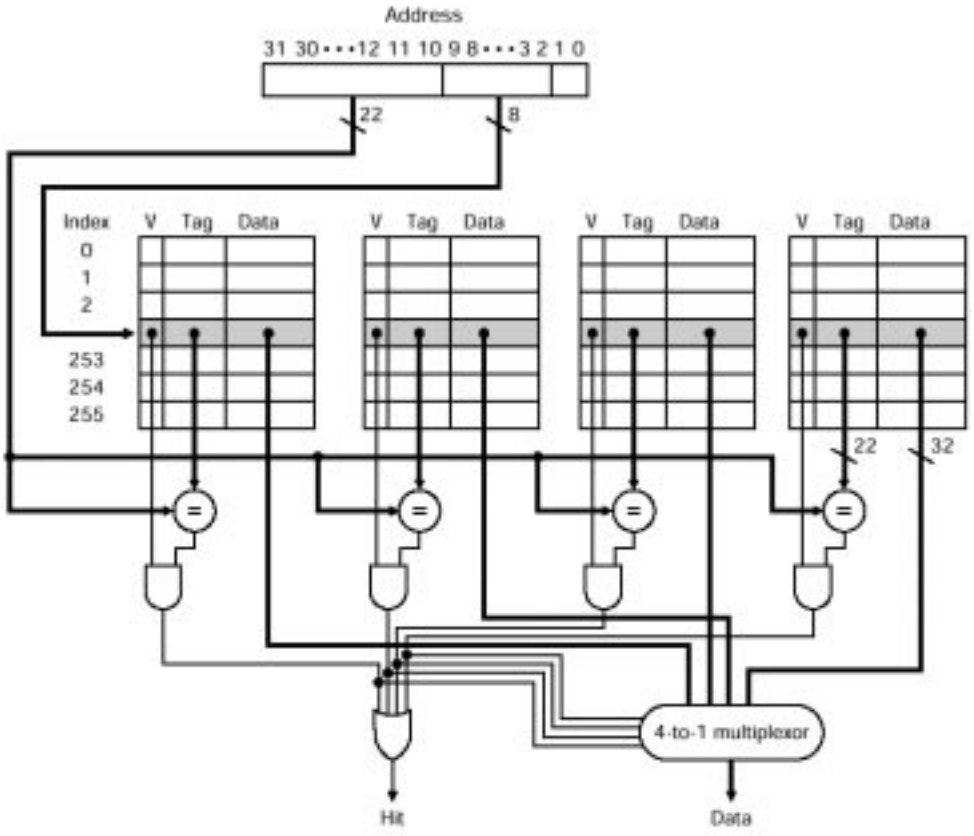
- **Abbiamo insiemi di 2/4/8/16 ... blocchi ⇒ cache associative a 2/4/8/16 vie ...**
- **Cache diretta ≡ Cache associativa a 1 via**
- **Nuova funzione di mapping**
  Block Address = Address / Block size
  Cache block INDEX = Block Address mod **# set**
- **L'INDEX viene usato per determinare l'insieme.**
  Dobbiamo controllare tutti i TAG associati ai vari blocchi per individuare il blocco



Scelta del blocco da sostituire

- **In caso di miss, possiamo trovarci nella condizione di dover sostituire un blocco**
- **Cache diretta**
  - **se il blocco corrispondente ad un certo INDEX è occupato (bit V=1), allora dobbiamo rimpiazzare il blocco**
  - **se il vecchio blocco è stato modificato e abbiamo usato politica di write-back, dobbiamo aggiornare la memoria**
- **Cache associativa**
  - **INDEX individua un insieme di blocchi**
  - **se nell'insieme c'è un blocco, non c'è problema**
  - **se tutti i blocchi sono occupati, dobbiamo scegliere il blocco da sostituire**
    - **sono possibili diverse politiche per il rimpiazzamento**
      - **LRU (Least Recently Used) - necessari bit aggiuntivi per considerare quale blocco è stato usato recentemente**
      - **Casuale**

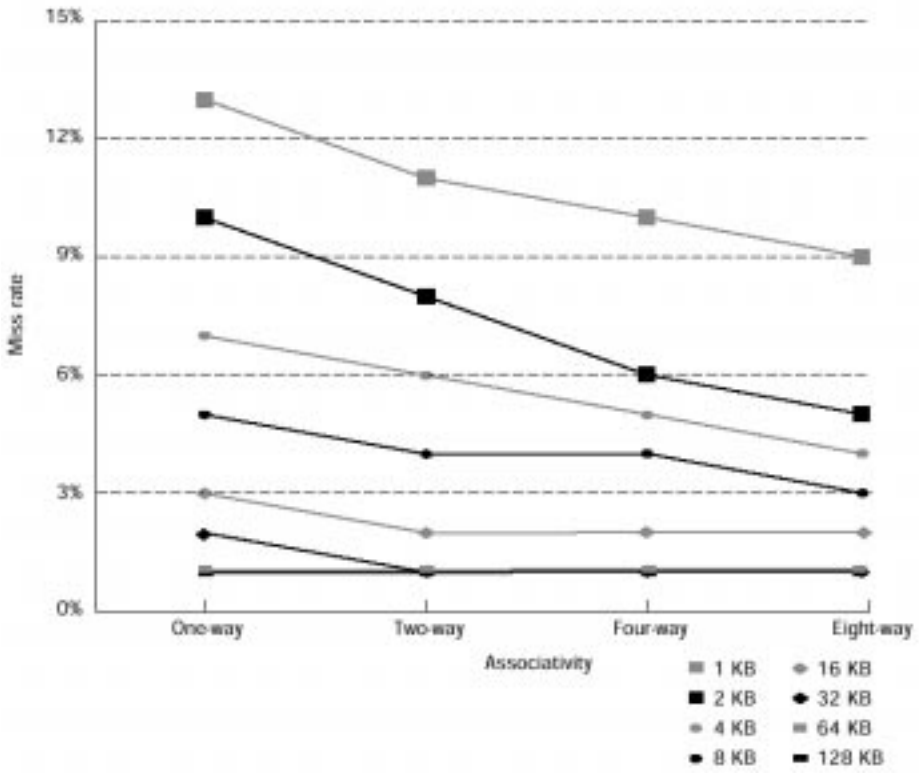
Un’implementazione (4-way set-associative)



- Nota
  - i comparatori
  - il multiplexer
- Vantaggio
  - minore miss rate
- Svantaggio
  - aumenta tempo di hit

Associatività e miss rate

- Le curve a lato si riferiscono a
  - blocchi di 32 B
  - benchmark Spec92 interi
- Benefici maggiori per cache piccole
  - perché si partiva da un miss rate molto alto nel caso diretto



Cache a più livelli

- **CPU con**
  - **cache di 1^ livello (L1), di solito sullo stesso chip del processore**
  - **cache di 2^ livello (L2), esterno al processore, implementato con SRAM**
  - **cache L2 serve a ridurre il miss penalty per la cache L1**
    - **solo se il dato è presente in cache L2**

- **Esempio**
  - **CPI=1 su un processore a 500 MHz con cache unica (L1), con un miss rate del 5%, e un tempo di accesso alla DRAM di 200 ns (100 cicli)**
    - **CPI_{L1} = CPI + 5% 100 = 1 + 5 = 6**
  - **Aggiungendo una cache L2 con tempo di accesso di 20 ns (10 cicli), il miss rate della cache L1 rispetto alla DRAM viene ridotto al 2%**
    - **il miss penalty in questo caso aumenta (200 ns + 20 ns, ovvero 110 cicli)**
    - **il restante 3% viene risolto dalla cache L2**
      - il miss penalty in questo caso è solo di 20 ns
    - **CPI_{L1+L2} = CPI + 3% 10 + 2% 110 = 1 + 0.3 + 2.2 = 3.5**
  - **Speedup = CPI_{L1} / CPI_{L1+L2} = 6 / 3.5 = 1.7**

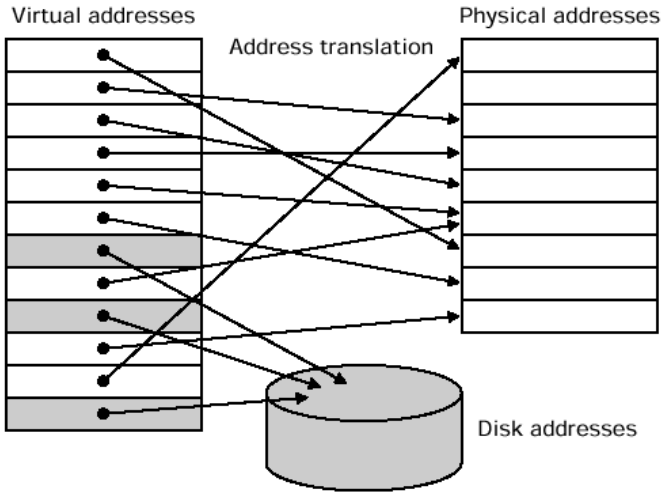
Cache a 2 livelli

- **Cache L1: piccola e con blocchi piccoli, di solito con maggior grado di associatività, il cui scopo è**
  - **ottimizzare l'hit time per diminuire il periodo del ciclo di clock**

- **Cache L2: grande e con blocchi più grandi, con minor grado di associatività, il cui scopo è**
  - **ridurre il miss rate (rispetto ai miss che devono accedere la DRAM)**
  - **la maggior parte dei miss sono risolti dalla cache L2**

Memoria Virtuale

- **Uso della memoria principale come una cache della memoria secondaria (disco)**



- **I programmi sono compilati rispetto ad uno spazio di indirizzamento virtuale, diverso da quello fisico**
- **I meccanismi di memoria virtuale effettuano la traduzione**
  - **indirizzo virtuale → indirizzo fisico**

Vantaggi della memoria virtuale

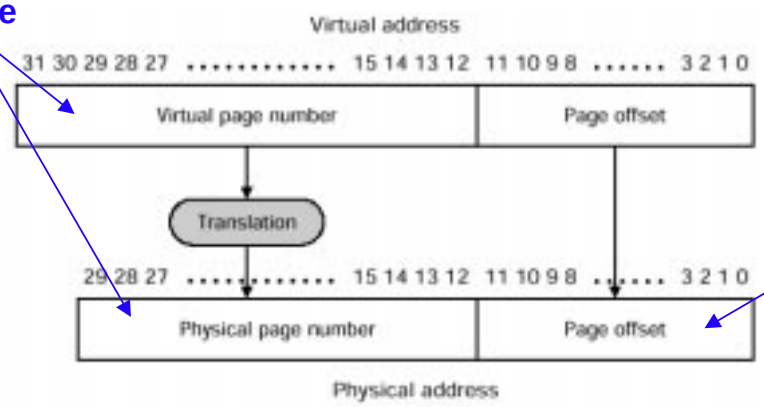
- **Illusione di avere più memoria fisica**
  - **solo le parti attive dei programmi sono presenti in memoria**
  - **più programmi, con codici e dati maggiori della memoria fisica, possono essere eseguiti**
- **Rilocazione dei codici**
  - **i programmi, compilati rispetto a uno spazio virtuale, sono caricati in memoria fisica *on demand***
  - **tutti i riferimenti alla memoria sono *virtuali* (fetch istruzioni, load/store), e sono tradotti dinamicamente nei corrispondenti indirizzi fisici**
- **Protezione**
  - **il meccanismo di traduzione garantisce la protezione**
  - **c'è la garanzia che gli spazi di indirizzamento virtuali di programmi diversi sono effettivamente mappati su indirizzi fisici distinti**

Pagine: sono i blocchi della memoria virtuale

- ***Page fault*: la pagina non è in memoria, e deve essere letta dal disco**
- ***Miss penalty* grande (msec), per cui è utile che i blocchi (pagine) siano grandi (es.: 4KB)**
  - **le letture da disco hanno un costo iniziale alto, dovuto a movimenti meccanici dei dispositivi**
- **Ridurre i *page fault* è molto importante**
  - **mapping dei blocchi (pagine) completamente associativo**
  - **politica LRU, per evitare di eliminare dalla memoria pagine da riusare, a causa della località degli accessi**
- ***Miss (page fault)* gestiti a software tramite l'intervento del SO**
  - **algoritmi di mapping e rimpiazzamento più sofisticati**
- **Solo politica *write-back* (perché scrivere sul disco è costoso)**

Paginazione vs Segmentazione

Numero di pagine virtuali possono essere maggiori di quelle fisiche

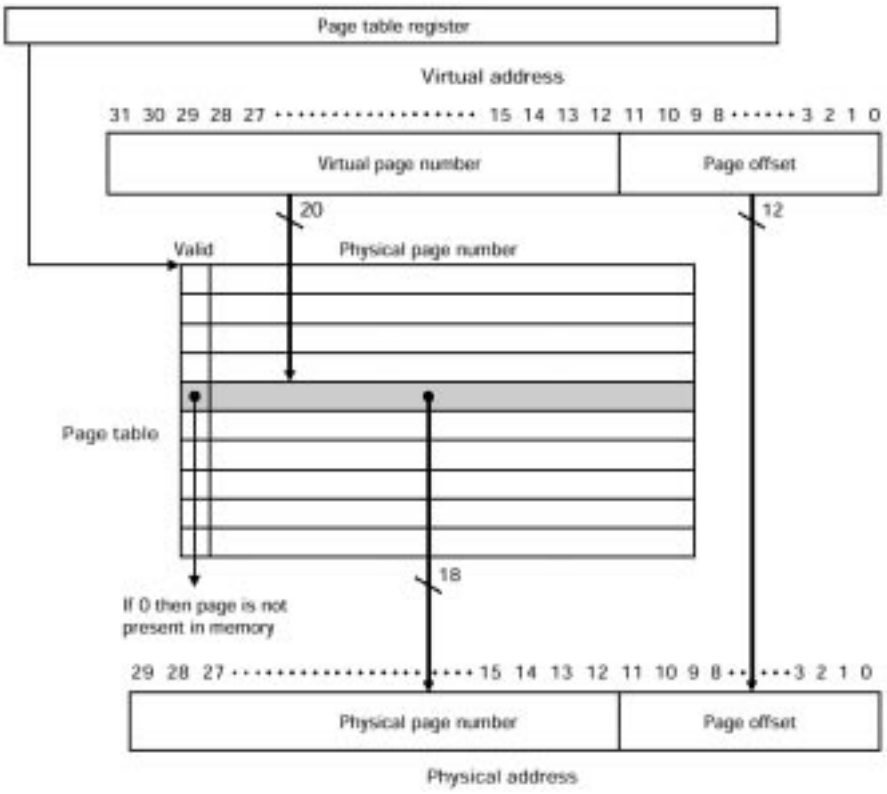


Dimensione dell'*Offset* dipende dal *page size*: log_2(*Page size*)

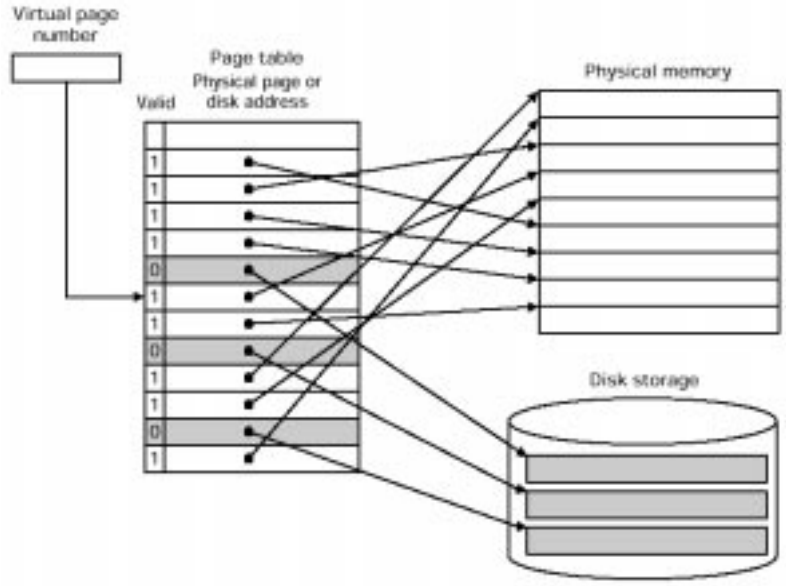
- **Oltre la paginazione, storicamente la memoria virtuale è stata anche implementata tramite *segmentazione***
  - **blocco variabile**
  - **Registri Relocation and Limit**
  - **enfasi su protezione e condivisione**
- **Svantaggio: esplicita suddivisione dell'indirizzo virtuale in *segment number + segment offset***

Page table, traduzione indirizzi, e associatività

- **Page Table (PT) mantiene la corrispondenza tra pagine virtuale e fisica**
- **La PT di un programma in esecuzione (processo) sta in memoria:**
  - **la PT è memorizzata ad un certo indirizzo fisico, determinato dal *page table register***
- **Ogni pagina virtuale può corrispondere a qualsiasi pagina fisica (completa associatività)**



Page fault



- **Al loading del processo, viene creato su disco l'immagine delle varie pagine del programma e dei dati**
- **Page table (o *struttura corrispondente*) usata anche per registare gli indirizzi su disco delle pagine**
  - **indirizzi su disco utilizzati dal SO per gestire il page fault, e il rimpiazzo delle pagine**

Approfondimenti

- Spesso, oltre al valid bit, sono aggiunti altri bit associati alla pagine
  - dirty bit: serve a sapere se una pagina è stata modificata. Grazie a questo bit è possibile sapere se la pagina deve essere ricopiata sul livello di memoria inferiore (disco). Il bit è necessario in quanto usiamo una politica write-back
  - reference bit: serve a sapere se, in un certo lasso di tempo, una certa pagina è stata riferita. Tali bit sono azzerati periodicamente, e settati ogni volta che una pagina è riferita. I reference bit sono usati per implementare una politica di rimpiazzo delle pagine di tipo LRU (Least Recently Used)

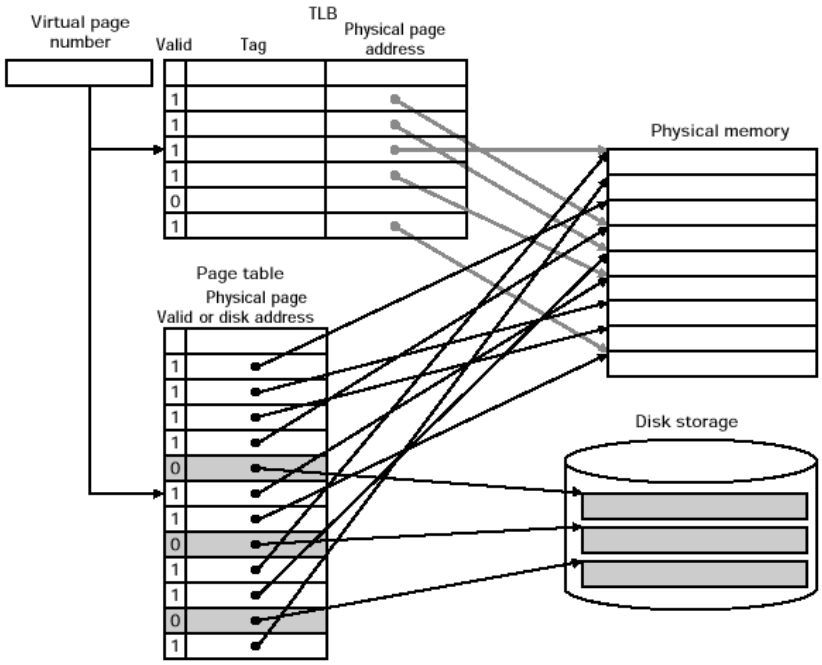
- La *page table*, per indirizzi virtuali grandi, diventa enorme
  - supponiamo ad esempio di avere un ind. virtuale di 32 b, e pagine di 4 KB. Allora il *numero di pagina virtuale* è di 20 b. La *page table* ha quindi 2^{20} entry. Se ogni entry fosse di 4 B, la dimensione totale sarebbe: 2^{22} B = 4 MB
  - se ci fossero molti programmi in esecuzione, una gran quantità di memoria sarebbe necessaria per memorizzare *soltanto* le varie *page table*
- Esistono diversi metodi per ridurre la memoria per memorizzare la PT
  - i metodi fanno affidamento sull'osservazione che i programmi (piccoli) usano solo una piccola parte della page table a loro assegnata, e che c'è anche una certa località nell'uso delle page table (es.: page table paginate, pagine a due livelli, ecc.…)

TLB: traduzione veloce degli indirizzi

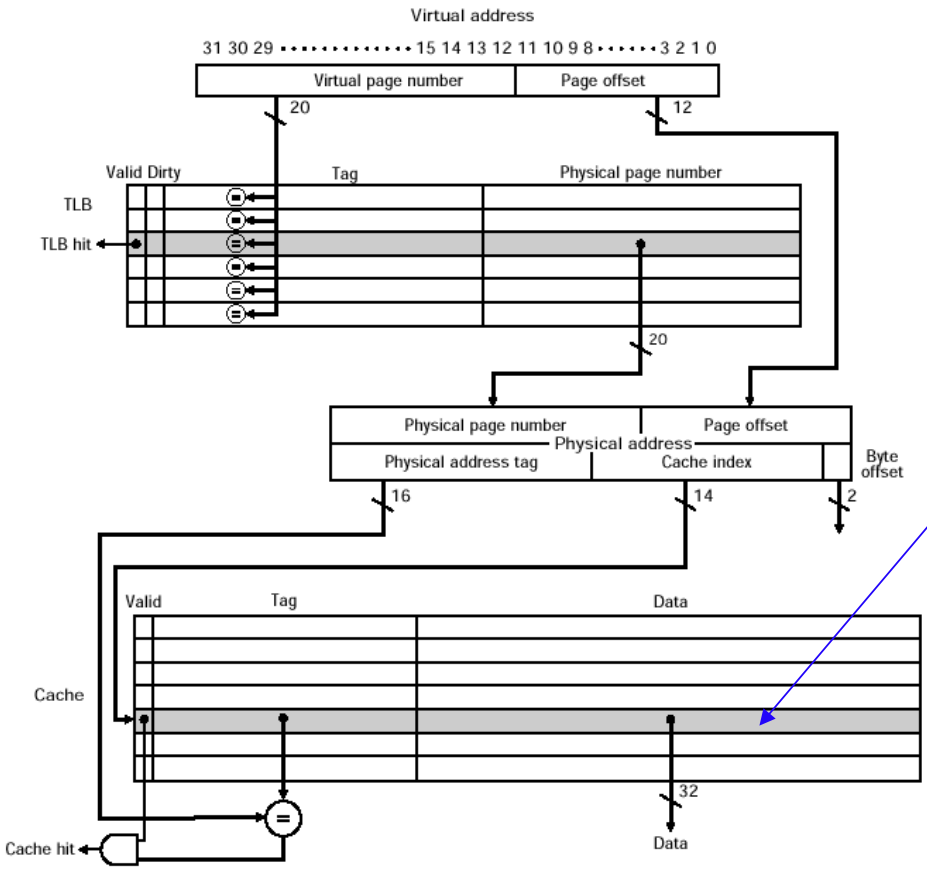
- La traduzione degli indirizzi fatta a software, accedendo ogni volta alla *page table* in memoria, è improponibile
  - troppo costoso
- La traduzione degli indirizzi viene solitamente fatta in hardware, da un componente denominato MMU (Memory Management Unit), tramite l'uso di una piccola memoria veloce, denominata:
  - TLB : translation-lookaside buffer
- La TLB è in pratica una cache della page table, e quindi conterrà solo alcune entry della page table (le ultime riferite)
  - a causa della località, i riferimenti ripetuti alla stessa pagina sono molto frequenti
  - il primo riferimento alla pagina (page hit) avrà bisogno di leggere la page table. Le informazioni per la traduzione verranno memorizzate nella TLB
  - i riferimenti successivi alla stessa pagina potranno essere risolti velocemente in hardware, usando solo la TLB

TLB

- **Esempio di TLB *completamente associativa***
  - **in questo caso il TAG della TLB è proprio il numero di pagina virtuale da tradurre**
  - **per ritrovare il numero di pagina fisica, bisogna confrontare il numero di pagina virtuale con tutti i TAG della TLB**
- **Nota che la TLB contiene solo entry che risultano anche Valid nella page table**
  - **perché ?**
- **La TLB, come la cache, può essere implementata con vari livelli di set-associativity**

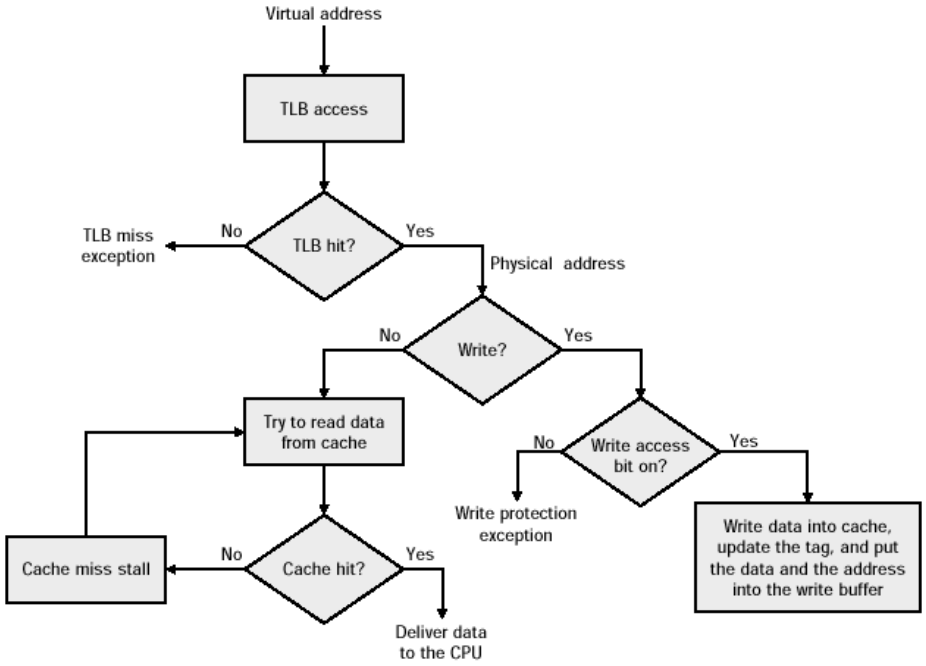


TLB e cache (vecchio processore MIPS)



- **TLB completamente associativa**
- **Cache diretta, acceduta con l'indirizzo fisico**
- **Esempio ottimale nell'accesso ad un dato**
  - **TLB hit**
  - **cache hit**

TLB e cache (vecchio processore MIPS)



- Gestione delle lettura/scrittura
  - TLB hit e miss
  - cache hit e miss
- Nota i cicli di stallo in caso di cache miss
- Nota la cache con politica write-through (solo bit di *Valid*, e Read Miss)
- Nota le eccezioni
  - TLB miss
  - write protection (usato come trucco per aggiornare i bit di dirty sulla page table)

Modello di classificazione dei *miss*

- Nelle varie gerarchie di memoria, le miss si possono verificare per cause diverse
  - modello delle tre C per classificare i miss
- Ci riferiremo al livello cache, anche se il modello si applica anche agli altri livelli della gerarchia
- Tipi di miss
  - Miss Certi (Compulsory)
    - miss di partenza a freddo, che si verifica quando il blocco deve essere portato nella cache per la prima volta
  - Miss per Capacità
    - la cache non è in grado di contenere tutti i blocchi necessari all'esecuzione del programma
  - Miss per Conflitti
    - anche se la cache non è tutta piena, più blocchi sono in conflitto per una certa posizione
    - questo tipo di miss non si verifica se abbiamo una cache completamente associativa

SO e gestione della memoria

- Per quanto riguarda la memoria virtuale, il SO viene invocato per gestire due tipi di eccezioni
  - TLB miss (anche se la TLB miss può essere gestita in hardware)
  - page fault
- In risposta ad un'eccezione/interruzione, il processore salta alla routine di gestione del SO, ed effettua anche un passaggio di modalità di esecuzione
  user mode $\rightarrow$ *kernel (supervisor) mode*
- Operazioni importanti dal punto di vista della protezione NON possono essere effettuate in *user mode*
  - non possiamo cambiare il PT register
  - non possiamo modificare le entry della TLB
  - non possiamo settare direttamente il bit che fissa l'execution mode
  - esistono istruzioni speciali, eseguibili SOLO in *kernel mode*, per effettuare le operazioni di cui sopra
- Nota che un processo che sta eseguendo in *user mode* può passare volontariamente in *kernel mode* SOLO invocando una `syscall`
  - le routine corrispondenti alle varie syscall (chiamate di sistema) sono prefissate, e fanno parte del SO (l'utente non può crearsi da solo una sua syscall e invocarla)

SO e gestione della memoria

- solo TLB miss
  - la pagina è presente in memoria
  - l'eccezione può essere risolta tramite la page table
  - l'istruzione che ha provocato l'eccezione deve essere rieseguita
- TLB miss e page fault
  - la pagina non è presente in memoria
  - la pagina deve essere portata in memoria dal disco
    - operazione di I/O dell'ordine di ms
    - è impensabile che la CPU rimanga in stallo, attendendo che il page fault venga risolto
  - *context switch*
    - salvataggio dello stato (contesto) del programma (processo) in esecuzione
      - fanno ad esempio parte dello stato i registri generali, e quelli speciali come il registro della page table
      - processo che ha provocato il fault diventa bloccato
    - ripristino dello stato di un altro processo pronto per essere eseguito
    - restart del nuovo processo
  - completamento page fault
    - processo bloccato diventa pronto, ed eventualmente riprende l'esecuzione

SO e gestione della memoria

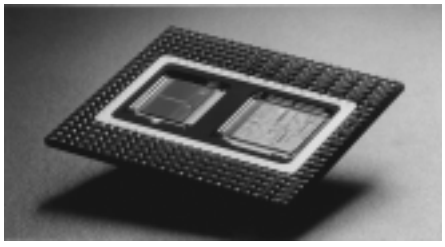
- **Page fault e rimpiazzamento di una pagina**
  - **se la memoria fisica è tutta piena, bisogna rimpiazzare una pagina (es. usando una politica LRU)**
  - **la pagina deve anche essere scritta in memoria secondaria se *dirty* (write-back)**
  - **poiché in questo caso rimpiazziamo una entry della page table, se questa entry**
  - **è anche cached nella TLB, bisogna anche ripulire la TLB**

- **Protezione**
  - **il meccanismo della memoria virtuale impedisce a ciascun processo di accedere porzioni di memoria fisica allocata a processi diversi**
  - **la TLB e la PT NON possono essere modificate da un processo in esecuzione in modalità utente**
    - **possono essere modificate solo se il processore è in *stato kernel***
    - **ovvero solo dal SO**

Casi di studio

- **Gerachie di memoria: Intel Pentium Pro e IBM PowerPC 604**

Characteristic	Intel Pentium Pro	PowerPC 604
Virtual address	32 bits	52 bits
Physical address	32 bits	32 bits
Page size	4 KB, 4 MB	4 KB, selectable, and 256 MB
TLB organization	A TLB for instructions and a TLB for data Both four-way set associative Pseudo-LRU replacement Instruction TLB: 32 entries Data TLB: 64 entries TLB misses handled in hardware	A TLB for instructions and a TLB for data Both two-way set associative LRU replacement Instruction TLB: 128 entries Data TLB: 128 entries TLB misses handled in hardware



Characteristic	Intel Pentium Pro	PowerPC 604
Cache organization	Split instruction and data caches	Split intruction and data caches
Cache size	8 KB each for instructions/data	16 KB each for instructions/data
Cache associativity	Four-way set associative	Four-way set associative
Replacement	Approximated LRU replacement	LRU replacement
Block size	32 bytes	32 bytes
Write policy	Write-back	Write-back or write-through