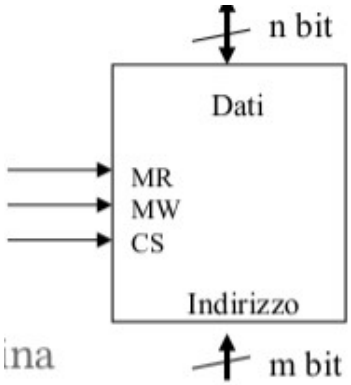
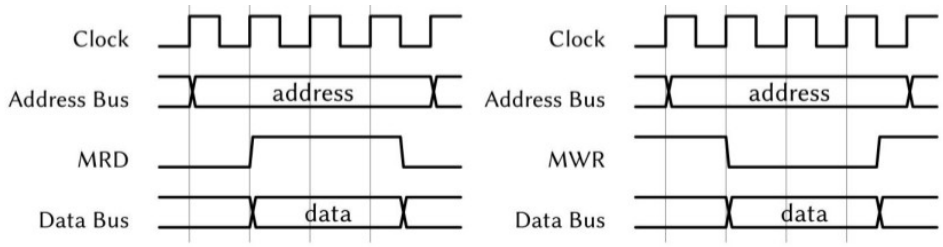
Lezione 36 Gerarchia di memoria parte 2

mercoledì 6 marzo 2024 16:59

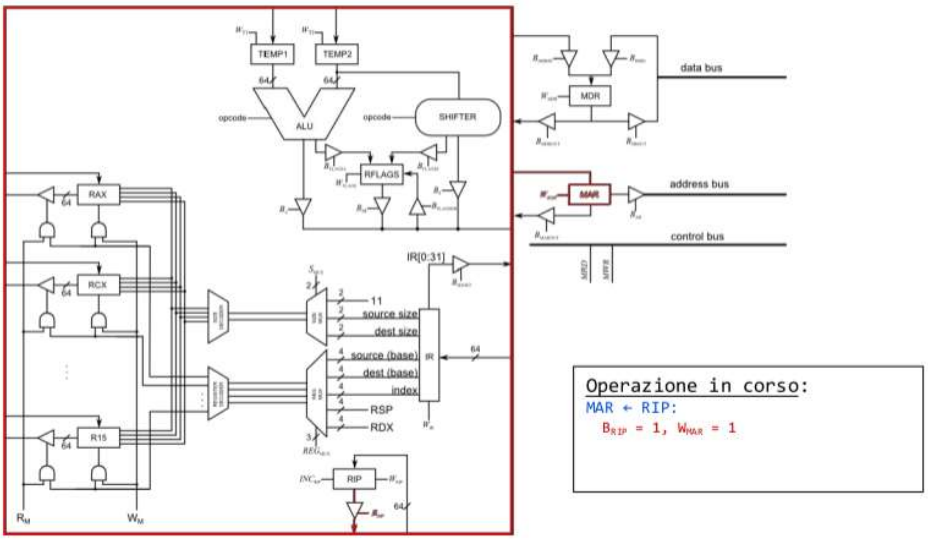
Andiamo ora a vedere come è fatta la memoria : è fatta come blocchetti :



Il quale è caratterizzato da indirizzo della parola da leggere/scrivere, MR (memory read) che se affermato permette di leggere, WR (write to register) che se affermato permette di scrivere nel registro, CS (chip select) che permette di abilitare uno o più moduli e l'insieme dei dati da processare. Analogamente alla rom, andiamo a vedere la latenza di accesso e/o interazione con il dispositivo :

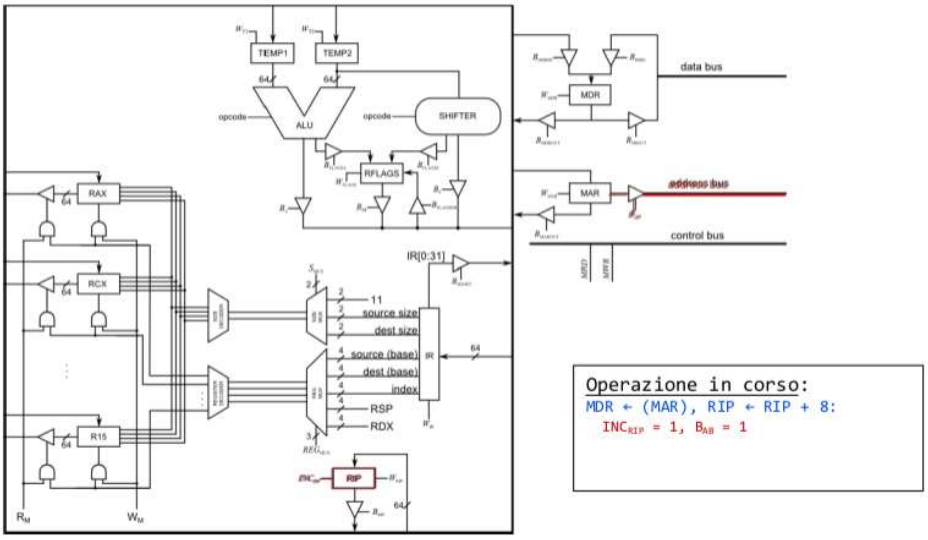


Quindi facciamo un esempio : voglio leggere/scrivere un indirizzo , quindi il dato contenente l'indirizzo viene mantenuto in memoria(buffer) con clock altro per tutto il tempo di interazione , per poi comunicare con il colpo clock successivo se sto leggendo e/o scrivendo. Quindi ora dopo la stabilizzazione , si riesce a leggere/scrivere da/verso memoria. Quindi vediamo ora cosa succede durante la latenza di accesso , ovvero la fase di fetch : <u>ricordiamoci che per fare il fetch di una qualunque istruzione servono 3 cicli di clock</u> . Vediamolo in dettaglio :

1. Fase 1
   a. 

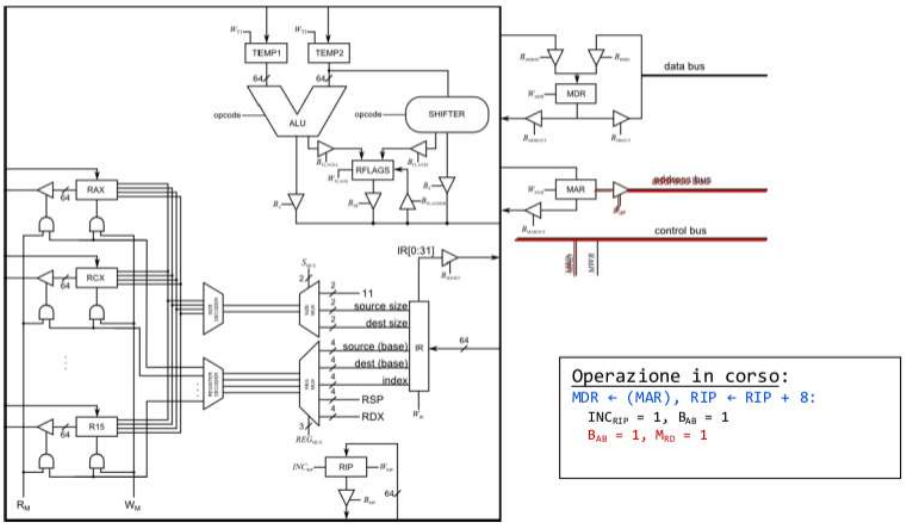
2. Fase 2:

a.



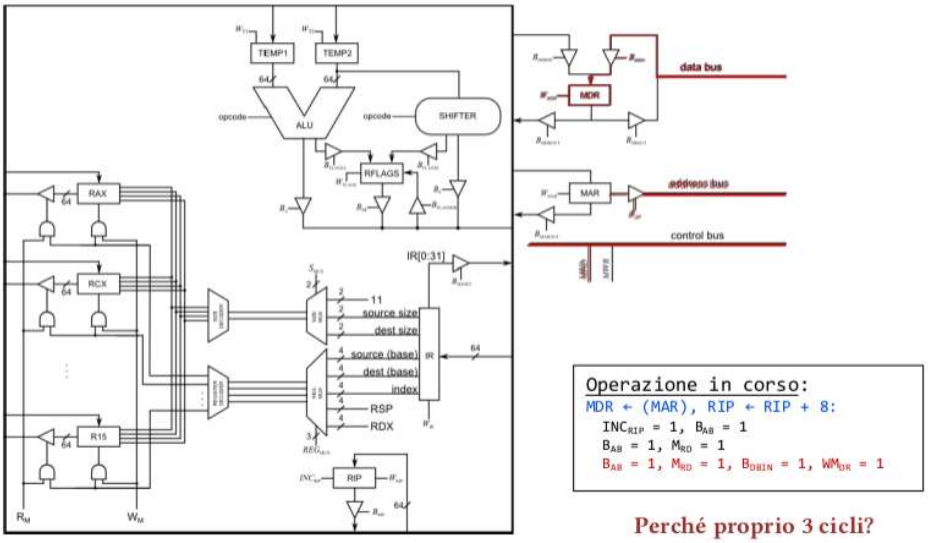
3. Fase 3

a.



4. Fase 4:

a.



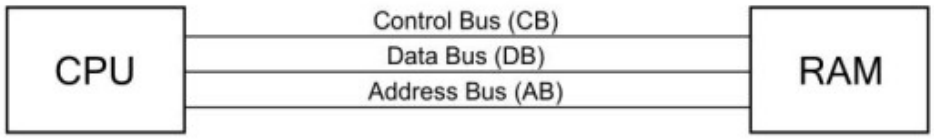
Perché proprio 3 cicli?

Quindi ora vista l'incompatibilità di alcune memorie , con alcuni processori , andiamo a vedere i parametri utili per vedere o meno questa compatibilità :

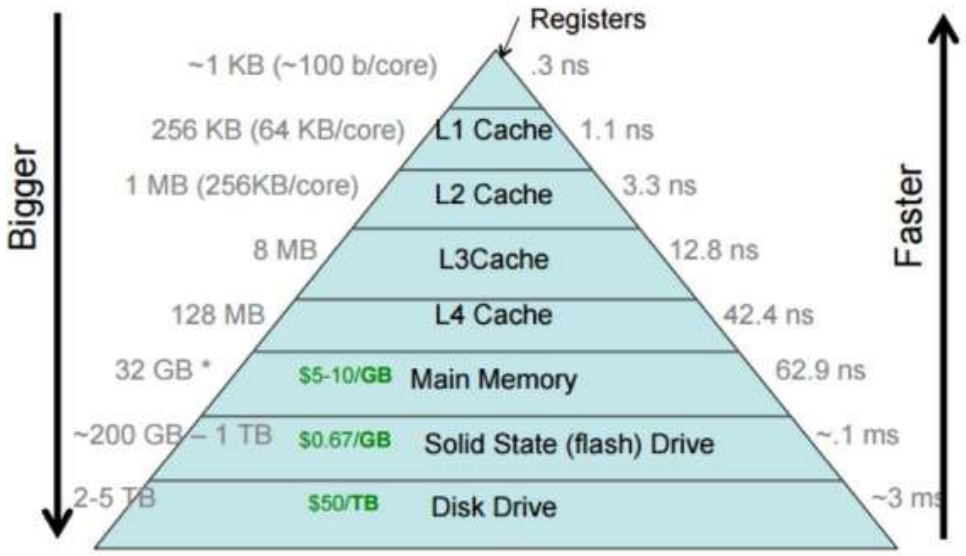
1. CAS (column address strobe): ritardo (colpi di clock) tra istruzione/comando di read ed il momento in cui sono disponibili
   a. Espresso in ciclo di clock per dram sincrone
   b. Espresso in nanosecondi per dram asincrone

2. Ritardo tra indirizzo riga ed indirizzo di colonna : numero cicli di clock per accedere ad una riga (copiato nel buffer tampone) e ne legge le colonne. Se questo tempo viene sommato al cas si ha il tempo necessario a leggere il primo bit dal bus dati
3. Tempo di precaricamento di riga : tempo necessario per risolvere un conflitto se si è nella fase di caricamento di una riga
4. Tempo di attivazione di riga : cattura il tempo di refresh.

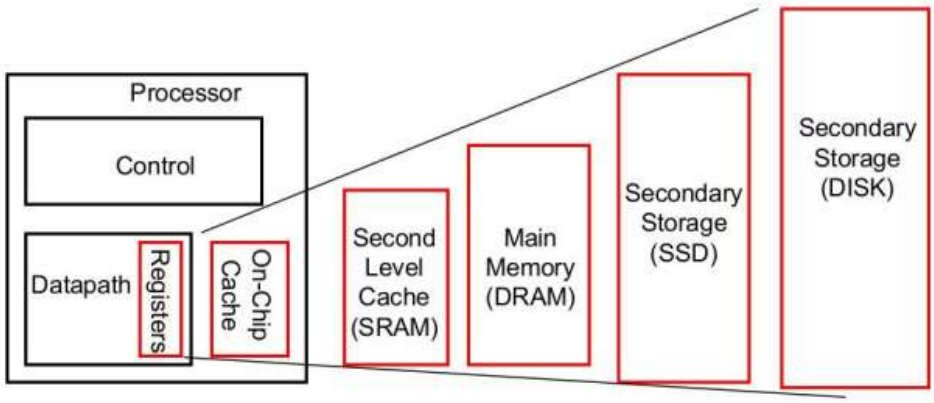
Ricordiamo che secondo il modello di Von-Neumann la CPU (processore) è direttamente collegata alla memoria (Ram) , ma avendo due velocità **completamente differenti**, si crea il famoso collo di bottiglia (la velocità di trasferimento è limitata dal dispositivo più lento) :



Che nonostante la *legge di moore* non si riesce a trovare un compromesso buono . In virtù di quanto detto finora , è possibile avere una memoria a costo minimo e capacità massima e latenza accesso minima : si parla di **gerarchia di memoria** : ovvero di più livelli di memoria invisibili al programmatore :



Da notare che sopra la memoria principale, ci sono delle memorie cache (registri tra memoria e processore) , aumentando cosi' le dimensioni e smorzando il trade off. Vista la gerarchia di memoria, i dati vengono passati solo tra livelli adiacenti . Per ogni cache (in particolare per la cache L1) , si hanno due ulteriori suddivisioni : cache dati e cache istruzioni: **Architettura Harvard.** Quindi in generale (o meglio realmente) la gerarchia di memoria è realizzata nel seguente modo :

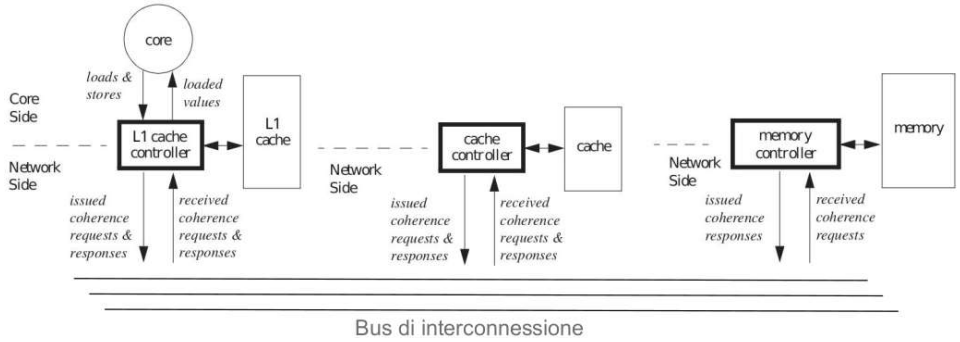


Basandosi quindi su prove empiriche , gli ingegneri usando dei pattern , sono arrivati a vedere il **principio di località** : ovvero uso della memoria con mascheramento della latenza in base al modo in cui è stato scritto il programma . Vediamolo in dettaglio : se in un qualunque istante , un programma tende ad accedere ad una porzione di dati relativamente piccola , sia per quanto riguarda i dati che le istruzioni : esecuzione locale del programma . **Si divide nel principio di**

località temporale e spaziale: il primo dice che se accedo ad un elemento nella memoria, quello stesso elemento potrà essere acceduto entro breve termine (nascondendo la latenza) , mentre la seconda dice se accedo ad un elemento, entro breve tempo accederò ad elemento "vicino" . Per sfruttare al massimo il principio nel primo caso si tengono i blocchi acceduti più frequentemente vicino al processore, mentre nel secondo sposto i blocchi contigui tra livelli della gerarchia. Andiamo ora a vedere delle definizioni molto importanti :

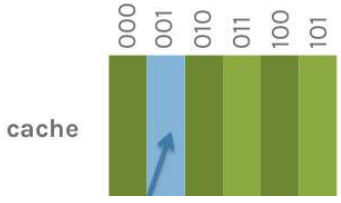
1. Hit rate : da massimizzare
    a. Dati trovati in un livello vicino al processore , quindi programma più veloce e diminuisce la latenza
2. Miss rate : opposto dell'hit rate
    a. Frequenza fallimento
    b. (1-hit rate) : se scende l'hit rate sale
    c. Se questo parametro è basso , di conseguenza altro è altro, si ha un utilizzo massimale del processore.
3. Miss penalty
    a. Tempo necessario per trasferimento del blocco dal livello inferiore
4. Miss time : tempo fallimento
    a. Tempo per ottenere il dato
    b. Miss time= miss penalty + hit rate

Quindi si ha una situazione di questo tipo:

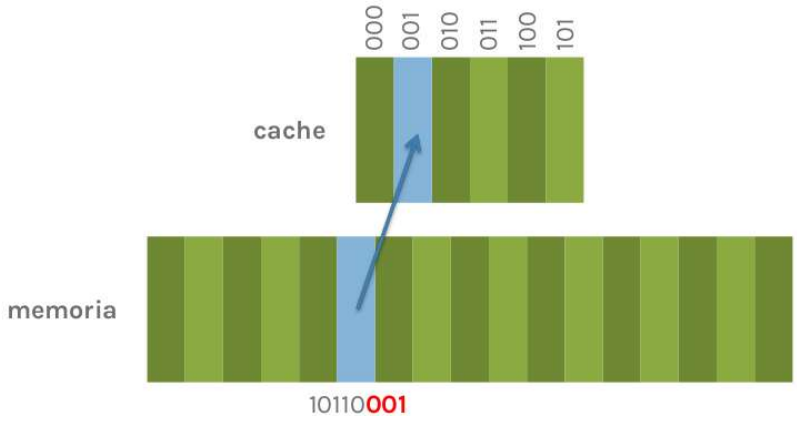


Dove il core (processore) parla solo con le cache di livello 1 , passando per il controllore della cache di primo livello , il quale va a vedere se ci sono i dati nella cache di livello 1: se si sono ok, altrimenti li deve andare a "pescare" dal bus di interconnessione (collegato agli altri controllori). Andiamo ora a vedere in dettaglio come funziona questo collegamento : ogni cache è strutturata in linee , ognuna delle quali è formata da un blocco di 64 byte , ovvero la minima unità di dati che vengono passati tra le cache . Quindi per quanto detto prima , **il processore parla con il controllore della cache di primo livello.** Quindi de il dato è presente nella cache L1 si ha un hit ! (finisce qui) , altrimenti il controllore chiede agli altri controllori (chiede info al controllore di L2) e se non c'è neanche qui , si procede al livello 3 (controllore L3). Analogamente si fa quando si ha accesso in RAM. Tornando alle cache : sono di due tipi : **inclusive e non inclusive** : quella di livello superiore (più vicina al processore) mantiene informazioni riguardo i livelli inferiori , i quali memorizzano le info richieste. Quindi nel nostro caso solo le L1 vengono accedute direttamente dal processore. Mentre per le seconde : piò capitare che non tutti i livelli siano inclusivi , come per esempio se miss in L1 , si accede diretto in memoria , scrivendo cosi' solo in L1 , e se non vi fosse spazio, questo blocco scende in L2. In generale in una qualunque riga della cache, oltre ai dati veri e proprio vi si scrive anche altri dati : le informazioni di controllo : come per esempio la directory. In virtù di queste informazioni aggiuntive , la dimensione della linea di cache è maggiore di 64 byte . Con queste ulteriori informazioni , si hanno politiche di accesso alle cache più efficienti. Vediamo ora le strategie di posizionamento/utilizzo cache :

1. Accesso diretto : max hit rate non sapendo working set a priori
    a. Uso la funzione hash per determinare il numero del blocco dove scrivere i dati
    b. Funziona per via dell'indirizzamento a blocchi di 2^n

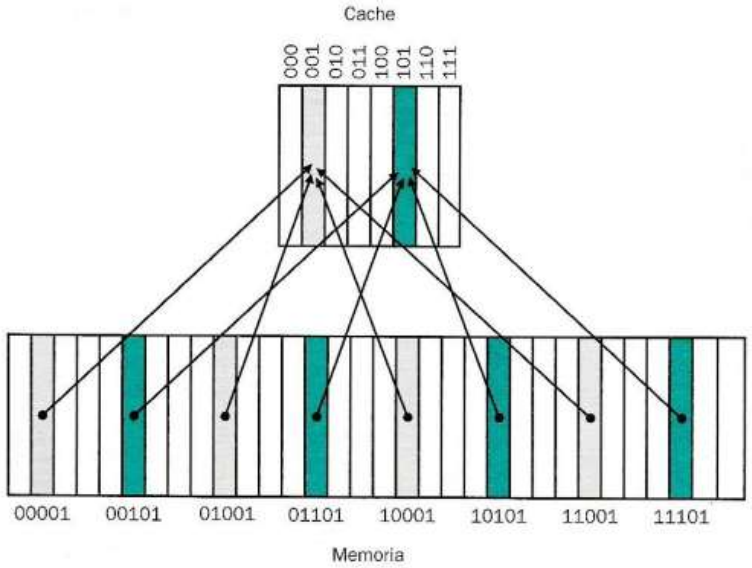


c.



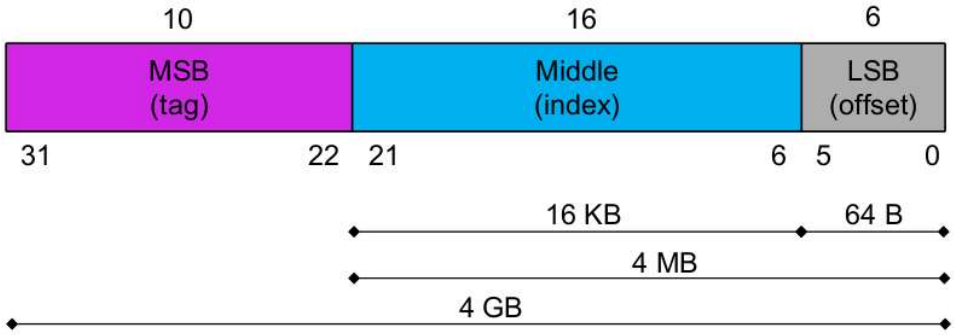
d. Ma attenzione : ad una stessa posizione in cache , ci possono essere più parole (sinonimi) :
e. Si deve usare un tag re risolvere queste collisioni :

f.



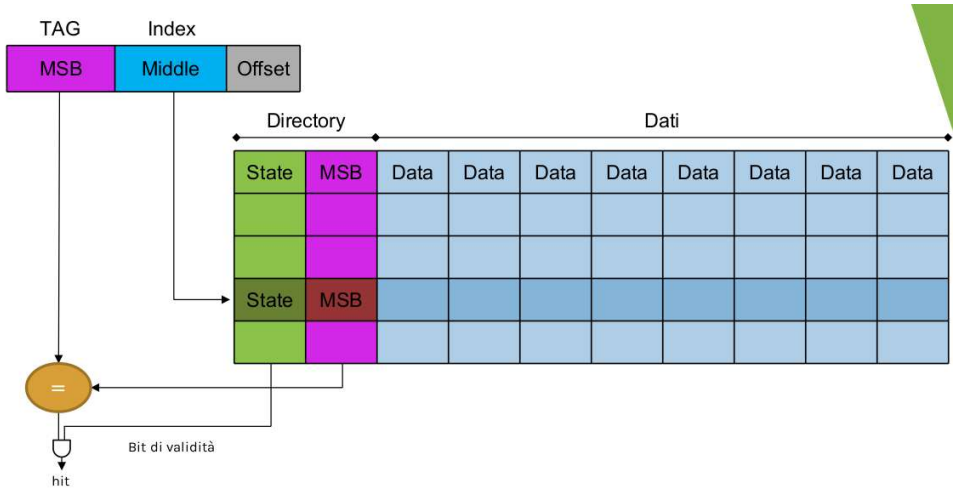
2. Accesso diretto : indirizzo memoria logico viene scomposto in 3 pezzi

a.



b. Tag serve per discriminare sinonimi
c. Index per selezionare una linea di cache
d. Offset seleziona la parola corrispondente all'interno del blocco

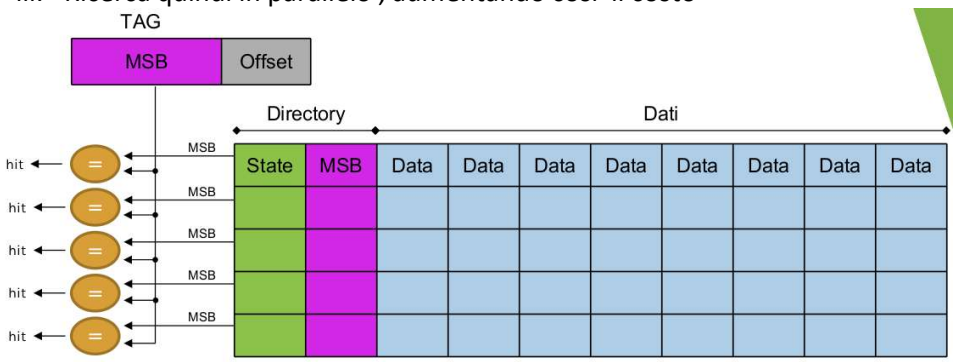
e.



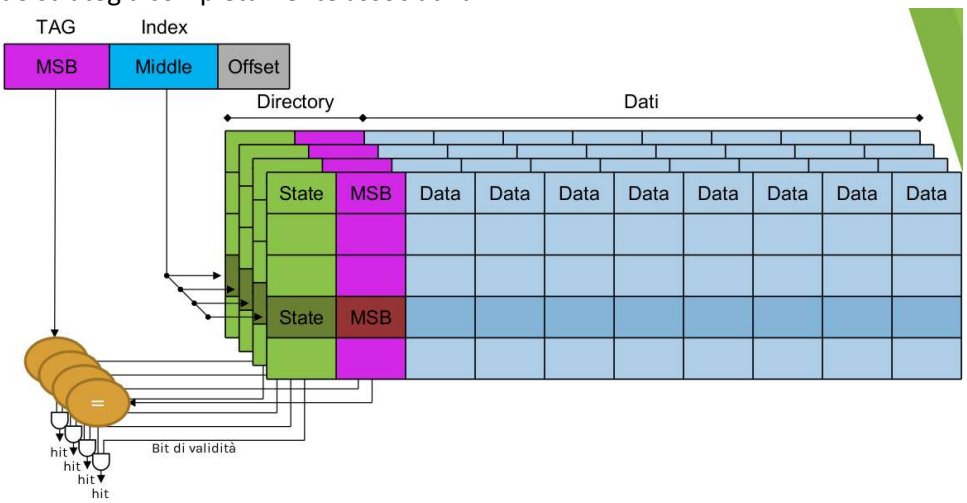
f. Problema : miss sempre su stesso blocco, sotto utilizzo delle linee di cache

3. Cache completamente associativa : ogni blocco può essere inserito in una qualunque riga della cache , se in caso di miss
   a. Quindi devo leggere il tag di tutte le righe
   b. Utilizzo massimo cache
   c. Riduzione conflitti
   d. Contro:
      i. Per ricerca devo scorrere tutta la cache
      ii. Ricerca sequenziale troppo lunga : aumenta latenza
      iii. Ricerca quindi in parallelo , aumentando cosi' il costo

e.



4. Cache set associativa a n-vie : cache che identifica e lavora su un insieme di linee , utilizzato secondo strategia completamente associativa



   a. Utilizzo migliore ma aumenta costo
   b. Ogni blocco può essere memorizzato in un insieme (>=2) di linee
   c. Ricerca su massimo di n tag
   d. Quindi il costo è accettabile
   e. Ed è attualmente la tecnologia utilizzata dai processori

Soffermiamoci sulle cache completamente associative e set associative a n-vie : si ha bisogno di scegliere una vittima per "rimpiazzare il blocco" , ma come viene scelta?

1. Least recently used : si ha una sorta di contatore dell'età
    a. Il contatore mantiene il tempo finche' non si ha il caricamento del blocco che viene azzerato , altrimenti se viene acceduto per vedere se libero si incrementa di uno
2. Fifo (first in first out) : Come LRU ma incremento avviene solo in caso di rimpiazzo
3. Least frequently used : difficile da implementare
4. Round robin (carosello): usato nelle completamente associative
    a. Mantiene un puntatore alla linea da rimpiazzare , ma viene aggiornato ad ogni rimpiazzo
5. Random : come round robin , ma il puntatore viene aggiornato ad ogni accesso

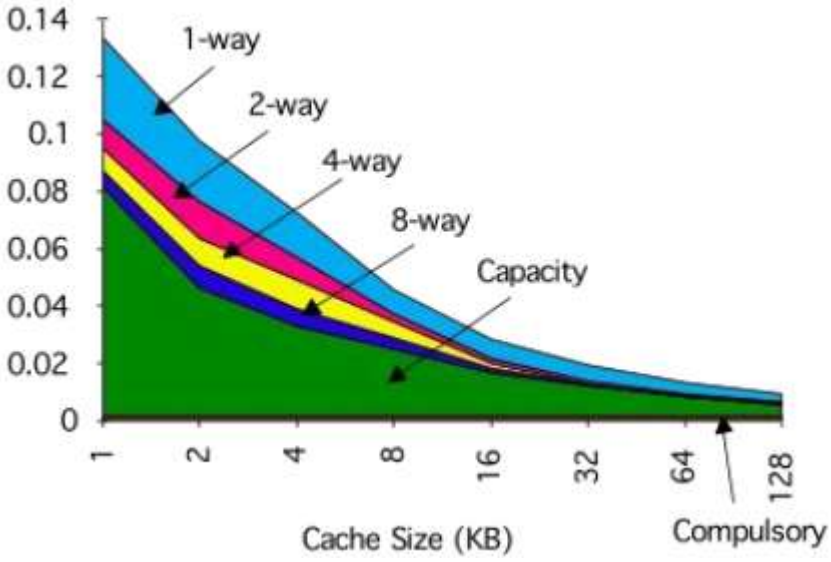
Quanto detto finora , riguarda solo la lettura , ma andiamo ora a vedere la scrittura : ricordiamoci che il processore comunica solo ed esclusivamente con la cache di primo livello (L1) , la quale mantiene una copia dei dati . Quindi in virtù delle copie , se si è in presenza si aggiornamenti, si deve mantenere la consistenza con le copie, implicando prima il caricamento del dato e poi la copia dello stesso. Queta scrittura avviene secondo due modi : **Write through e Write back** : nella prima (scrivi attraverso) , se si scrive in L1 automaticamente si aggiornano le copie , quindi molto facile come implementazione ma prestazioni minori; mentre la seconda opta per aggiornamento delle copie (livelli sottostanti) se e solo c'è un rimpiazzo: quindi più difficile da implementare , ma prestazioni nettamente superiori. Andiamo ora a vedere le prestazioni invece : in generale una qualunque esecuzione di un programma impiega :

$$T_{exec} = n_{istruzioni} \cdot CPI \cdot \frac{sec}{ciclo}$$

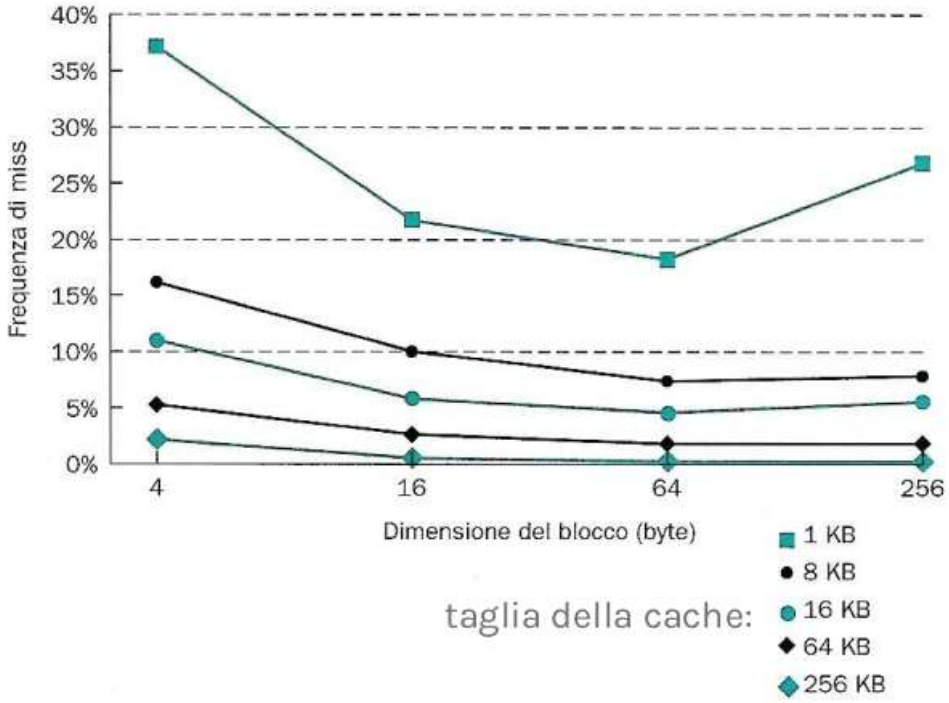
Il numero di *cicli di clock per istruzione* (CPI) dipende dal Tempo Medio di Accesso in Memoria (TMAM):

$$TMAM = Hit\ Time\ +\ (Miss\ Rate \cdot Miss\ Penalty)$$

Il quale se si riesce a ridurre, implica un aumento delle prestazioni. Torniamo ai miss, o meglio alle cause del miss : possibile **miss per conflitto, capacità ed obbligatori**. Vediamo il primo : si è nel caso si cache set associative a n-vie con blocchi di m byte , ma dato che n<m si ha il conflitto. Come risolvere ? O aumentiamo il numero della dimensione della linea ,oppure aumento la dimensione del blocco. In dettaglio :



Vediamo ora quelli obbligatori : la prima volta che si accede a quel dato non sarà disponibile (per esempio accendo il pc) , quindi si adotta la tecnica del cache prefetching : prevedo/indovino i blocchi che verranno richiesti in futuro , mentre nell'ultima si ha che la linea di cache , non può mantenere tutti i blocchi richiesti , quindi in seguito ad aumento della dimensione della cache aumentano i costi. Quindi in generale si ha una sorta di andamento a "vasca da bagno" :


40%
35%
30%
25%
20%
15%
10%
5%
0%
Frequenza di miss
4
16
64
256
Dimensione del blocco (byte)
taglia della cache:
1 KB
8 KB
16 KB
64 KB
256 KB