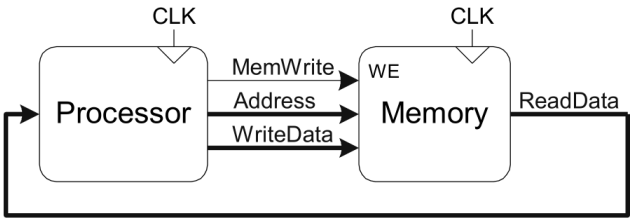
7. MEMORIE

CPU vs Memorie

Nell'architettura Von Neuman la comunicazione tra la CPU (processore) e la memoria è il punto critico (collo di bottiglia) del sistema, ossia uno dei punti in cui il sistema perde più tempo durante l'esecuzione del programma. Nel tempo sono state realizzate CPU sempre più veloci e memorie sempre più grandi ma sostanzialmente le memorie hanno avuto sempre una velocità di accesso inferiore rispetto alla velocità della CPU.I due componenti viaggiano a velocità diverse.
Il processore ha di per sè una memoria interna, ossia i 16 registri contenuti nel file register, che però costituiscono solo una memoria di supporto per l'esecuzione delle singole istruzioni una dopo l'altra e dunque non è sufficiente avere registri di 32 bit; c'è bisogno di una memoria esterna, le cui operazioni principali sono operazioni di LOAD e STORE.

Il processore richiede di caricare una certa locazione di memoria o scrivere su una certa locazione di memoria fornendo un certo indirizzo. In particolare il processore comunica con la memoria attraverso un'interfaccia inviando un indirizzo alla memoria attraverso il bus indirizzi (*Address*). In caso di lettura il segnale *MemWrite* vale 0 e la memoria restituisce il dato sul bus di lettura *ReadData*. In caso di scrittura il segnale *MemWrite* vale 1 e il processore invia il dato alla memoria sul bus di scrittura *WriteData*.
Nell'architettura di Von Neumann questo processo è uguale sia per i dati che per le istruzioni dato che entrambi risiedono in memoria principale. Gli indirizzi passati dal processore attraverso il bus *Address* o sono contenuti nel program counter e si riferiscono alla prossima istruzione da eseguire o sono indirizzi che riguardano il *fetch* di dati dalla memoria.

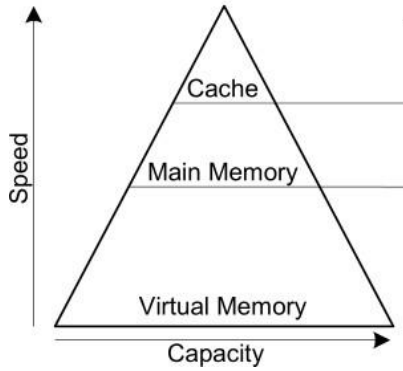


Storicamente quindi le CPU sono sempre state più veloci delle memorie.
Al giorno d'oggi siamo in grado di produrre delle memorie veloci quanto una CPU moderna, il reale problema è dato dal fatto che:
- queste memorie hanno un costo elevatissimo: ad esempio avere una memoria realizzata solo attraverso SRAM avrebbe un costo elevatissimo. Sarebbe veloce ma antieconomica.
- le memorie dovrebbero essere piazzate in gran parte sugli stessi chip delle CPU, il che non è possibile, in quanto di capacità grandi.

Gerarchia delle memorie

Si è pensato di introdurre quindi una "**gerarchia di memoria**": cioè la memoria è organizzata in livelli in cui abbiamo:
- una quantità molto piccola di memoria (qualche Mb) estremamente veloce, la **cache (SRAM)**
- una quantità più grande (Gb) di memoria più lenta, la **main memory (DRAM)**
- una quantità di memoria molto grande (Tb) ma estremamente lenta, la **virtual memory (HDD-SSD)**

L’organizzazione è di tipo piramidale. Si sfrutta al meglio questa gerarchia di memorie per accedere rapidamente ai dati più usati offrendo comunque la capacità di immagazzinare grandi quantità di dati. Infatti idealmente una memoria dovrebbe essere veloce, grande ed economica: siccome non esiste una memoria del genere, combinando queste tipologie di memorie riusciamo ad ottimizzare tutti i parametri.
Si combina una memoria veloce, piccola ed economica con una memoria lenta, grande ed economica. La memoria veloce memorizza le istruzioni e i dati usati più di frequente, sfruttando la velocità di accesso e la capacità ridotta. La memoria grande il resto delle istruzioni e dei dati, avendo una grande capacità.
Nella gerarchia al crescere della capacità e della bandwidth, la velocità e quindi il tempo di accesso decresce.
Access time/Tempo di accesso: è la latenza, il tempo che intercorre da quando si fornisce un indirizzo da cui andare a leggere dei byte fino a quando viene letto in output il primo byte di memoria richiesto.
Bandwidth: Gigabyte di memoria letti e trasferiti al secondo dalla memoria appena inizia a leggere.



Technology	Price / GB	Access Time (ns)	Bandwidth (GB/s)
SRAM	$10,000	1	25+
DRAM	$10	10 - 50	10
SSD	$1	100,000	0.5
HDD	$0.1	10,000,000	0.1

Località dei riferimenti
La cache deve essere sfruttata al meglio, essendo essa piccola ma estremamente veloce. Sappiamo che la maggior parte del tempo di esecuzione di una CPU è, di solito, impegnato da procedure in cui vengono eseguite ripetutamente le stesse istruzioni.
Questo concetto è noto come **località dei riferimenti**, cioè il fatto che alcune istruzioni in aree ben localizzate di un programma vengono eseguite ripetutamente in un determinato periodo di tempo, e si accede al resto del programma relativamente di rado.
Si manifesta in due modi: **località temporale** e **località spaziale.**

- La **località temporale** indica il fatto che un riferimento in memoria che è stato letto di recente ha un’alta probabilità di essere riletto nel prossimo immediato (futuro).
Significa che il processore ha un’elevata probabilità di accedere nuovamente nel prossimo futuro a un dato se lo ha utilizzato da poco. Oppure dopo aver inizializzato una variabile successivamente dopo poco la si utilizza in un’espressione. Oppure istruzioni in cicli.

- La **località spaziale** indica il fatto che se si richiede accesso ad un certo riferimento di memoria in lettura o scrittura, locato ad un certo indirizzo, molto probabilmente si richiederà di lì a poco un accesso in memoria ad un indirizzo vicino all'indirizzo richiesto nell’immediato.

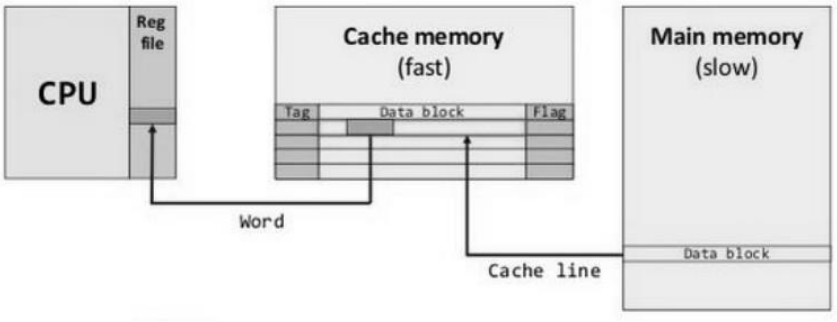
Significa che quando il processore accede ad un certo dato ha un'elevata probabilità di accedere nel prossimo futuro ad altri dati in locazioni di memoria vicine al dato in questione. Esempio: array che è costituito da locazioni di memoria contigue (4 in 4), i riferimenti sono vicini. Anche per le istruzioni c'è una forte località spaziale, nonostante i branch che di solito fanno saltare poche istruzioni e sono abbastanza pochi.

Questi aspetti in qualche modo giustificano l'utilizzo di una memoria piccola ma estremamente veloce. Si inseriscono nella memoria cache quei dati/istruzioni che hanno una maggiore probabilità di essere richiamate a breve termine.

Cache memory
Per la CPU l'organizzazione della memoria è come una black-box cioè la CPU ha una comunicazione con la memoria attraverso le operazioni di load e di store che prescinde da come è organizzata la memoria. Inoltre sappiamo che la velocità con cui la memoria risponde alle richieste di istruzioni e dati della CPU ha un peso determinante sulle prestazioni di un sistema.

La **memoria cache**, detta anche "memoria tampone" è una memoria che fa da filtro fra la CPU e la main memory che serve a velocizzare il tempo di accesso per prelevare o scrivere i dati o le istruzioni, in quanto contiene parti di programma e di dati che, volta per volta, interessano l'elaborazione. Essa è una memoria molto veloce e piccola, da qualche kb a qualche mb, ed è costituita generalmente da SRAM; è situata a bordo dello stesso chip del processore, eliminando così i ritardi dovuti alla propagazione dei segnali elettrici tra chip diversi. La cache può memorizzare sia dati che istruzioni e riduce il tempo totale di esecuzione in modo significativo ed impedisce alla CPU di "vedere" i tempi di risposta reali della memoria.

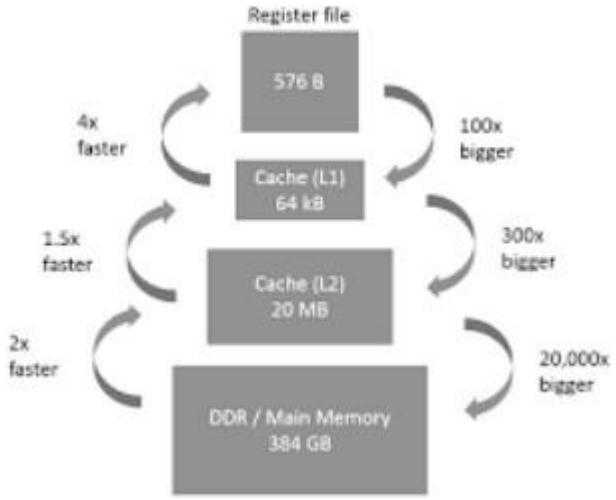
Principi di funzionamento cache



I circuiti di controllo della memoria cache sono progettati per avvantaggiarsi della proprietà della località dei riferimenti.
Abbiamo la cpu col suo file register, essa mediante le operazioni di load e store richiede dei dati o delle istruzioni (program counter) dalla memoria. Nel mezzo abbiamo la memoria cache che fa da filtro fra la CPU e la main memory. Inizialmente se il processore richiede un dato che è direttamente presente nella cache memory, tale dato viene reso immediatamente disponibile al processore rapidamente, risparmiando il tempo di accesso alla main memory. In questo caso abbiamo un **cache hit.** Se il dato o l'istruzione richiesta dal processore non è direttamente presente nella cache memory, il processore lo recupera dalla main memory. In questo caso abbiamo un **cache miss.**

In questo caso, dopo aver prelevato il dato o l'istruzione dalla main memory per la prima volta, il **principio di località temporale** secondo cui questo dato potrà essere richiesto nel prossimo futuro, ci suggerisce di copiare il riferimento di memoria nella cache in modo tale che rimanga a disposizione nel caso di una nuova richiesta, minimizzando il tempo di accesso alla main memory e facendo verificare un hit al prossimo accesso. Inoltre per il principio di **località spaziale** se il processore ha richiesto un dato o istruzione in memoria locato ad un certo indirizzo, è molto probabile che richieda nel prossimo futuro altri riferimenti di memoria (dati o istruzioni) presenti in locazioni di memoria contigue (vicine) al dato/istruzione in questione.
Dunque la cache quando preleva una parola dalla main memory preleva anche altre parole adiacenti: questo gruppo di parole è denominato blocco. Quindi **un blocco o linea di cache** è un insieme di indirizzi contigui in memoria di una qualche dimensione.

Dimensioni cache

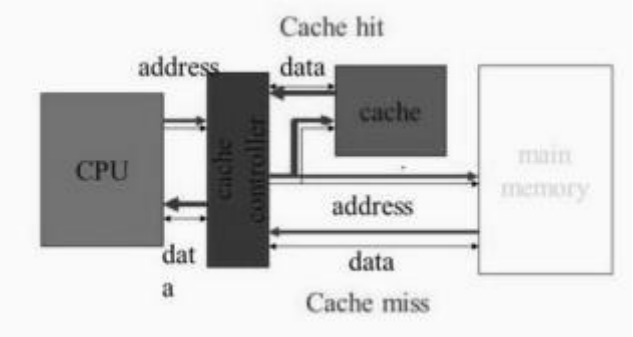


La cache memory è a sua volta suddivisa in due livelli L1 e L2. Il livello L1 è formato da una cache piccolissima e velocissima di 64kb, vicina alla velocità dei registri della CPU. Poi c'è una cache L2 che è più lenta ed è grande una decina di Mb. Poi c'è la main memory.
Dopo aver caricato un blocco di parole di memoria in cache, ogniqualvolta il processore fa riferimento a una di queste locazioni del blocco, i valori desiderati vengono letti direttamente dalla cache. Affinché la cache svolga efficacemente il suo compito deve avere tempi di accesso molto più brevi di quelli della memoria principale e ciò impone che sia piccola.
Se è piccola non potrà che contenere una frazione ridotta delle istruzioni ed i dati della memoria principale cioè solo i dati e le istruzioni di uso frequente.

Tecniche di gestione cache
La memoria cache ha un certo insieme di indirizzi; la main memory un altro insieme di indirizzi. La mian memory è molto più grande della memoria cache quindi ha molti più indirizzi di essa. Dunque deve esserci un **"mapping" cioè una funzione / algoritmo di posizionamento**, che dato un blocco della main memory stabilisce dove esso va a posizionarsi nella memoria cache. Deve crearsi una corrispondenza tra gli indirizzi della memoria cache e gli indirizzi della memoria virtuale.
Inoltre poichè la memoria cache è più piccola della main memory, durante l'esecuzione del programma essa sicuramente diventerà **satura** (piena) quindi quando vogliamo aggiungere un nuovo blocco prelevato dalla main memory e non presente in cache si deve fare spazio

cancellando un altro blocco riscrivendolo in memoria. Questa gestione è affidata ad un **algoritmo di sostituzione** che individua quale blocco sostituire quando la cache è satura.

Tutte queste gestioni sono affidate ad un **controller della cache**.
La CPU, infatti, nell'esecuzione del programma effettua le richieste di lettura e scrittura utilizzando gli indirizzi delle locazioni nella memoria principale. E' la logica di controllo della cache (**controller**) che si fa carico di determinare se la parola richiesta è presente o meno nella cache.
Il controller ricerca la parola nella cache; se essa è presente viene effettuata l'operazione di lettura o scrittura della locazione di memoria appropriata. In questo caso si dice che **l'accesso in lettura o in scrittura ha avuto successo** (**read hit o write hit**) a seconda dell'istruzione. Se la parola non viene trovata in cache abbiamo un **read miss o un write miss**.



Gestione di un CACHE HIT
Se l'accesso alla cache ha avuto successo e quindi abbiamo una situazione di cache hit allora bisogna distinguere due tipi di situazioni:

Operazione di lettura: la memoria principale non viene coinvolta. Non c'è un coinvolgimento della main memory perchè la CPU richiede un indirizzo, il controller della cache si accorge che il dato a quell'indirizzo è già presente in cache e lo preleva da essa velocemente senza interpellare la main memory.

Operazione di scrittura: si può procedere con 2 strategie diverse: **write-through o write-back**

1) write-through: il dato da modificare in una certa locazione viene scritto simultaneamente sia nella memoria cache che nella main memory. Si mantiene una coerenza fra quello che c'è scritto nella memoria cache e quello che c'è scritto nella main memory. In realtà non viene riscritto solo un dato, poiché il passaggio fra cache e memoria avviene sempre in blocchi, quindi si aggiorna tutto il blocco in cache in cui è presente il dato da aggiornare e così si aggiorna anche il corrispondente blocco in memoria principale. Questa strategia è molto semplice e conservativa (safe) ma non è sempre efficace perchè se si va ad aggiornare lo stesso dato in qualche istruzione successiva, si deve di nuovo riaggiornare tutto il blocco in cache e in memoria: questa comunicazione fra le due memorie ha una certa latenza e costo e ciò inoltre richiede molti accessi alla memoria principale (molto lenta), aumentando i tempi. Quindi ci sono inutili operazioni di scrittura nella memoria principale, se una parola viene aggiornata più volte durante il periodo in cui risiede nella cache.

2) write-back: in questa strategia si aggiorna soltanto il blocco della memoria cache e ad ogni blocco è associato un **bit di modifica o dirty** che vale 1 se il blocco è stato modificato da almeno una scrittura, altrimenti vale 0 se non ha subito alcuna modifica in scrittura. Il blocco corrispondente in main memory viene aggiornato in seguito, quando il blocco modificato della memoria cache deve essere rimosso per far posto a un nuovo blocco. Quindi quando quel blocco della cache deve essere sostituito, se il suo bit di modifica è pari a 1 effettuo il trasferimento di quel blocco nella memoria principale.
Anche il protocollo write-back può causare inutili scritture nella memoria principale, visto che, quando si procede con la scrittura nella memoria principale di un blocco, tutte le parole del blocco vengono scritte, anche se solo una delle parole del blocco della cache è stata modificata

Gestione di un READ MISS
Quando una parola richiesta in lettura dalla CPU non è presente nella cache si dice che l'accesso in lettura è fallito (**read miss**). Il blocco contenente la parola richiesta deve essere copiato dalla memoria principale nella memoria cache.
Abbiamo due modalità diverse per caricare il blocco in cache:
1- Si prende il blocco contenente la parola richiesta dalla memoria principale, lo si trasferisce in cache e poi lo si legge direttamente dalla memoria cache. Quindi dopo aver caricato l'intero blocco nella memoria cache, la parola richiesta viene inviata alla CPU.
2- **Load-through o early restart:** appena si trova la parola ricercata nel blocco della main memory si invia contemporaneamente la parola richiesta alla cpu e poi si sostituisce il blocco contenente la parola nella memoria cache. Quindi si invia contemporaneamente la parola alla CPU appena viene trovata in main memory. Ciò riduce in qualche modo il tempo di attesa della CPU, a discapito di una maggiore complessità del circuito di controllo della cache.

Gestione di un WRITE MISS
Durante un'operazione di scrittura, se la parola indirizzata non è nella cache si dice che l'accesso in scrittura è fallito (**write miss**). Non si trova un dato che devo scrivere in cache.
Due alternative:
1- **se si utilizza un protocollo write-through**, le informazioni vengono scritte direttamente nella memoria principale, poi si trasferisce il blocco in cache.
2- **se si utilizza un protocollo write-back** il blocco contenente la parola indirizzata in main memory viene prima caricato nella cache, poi viene sovrascritto sempre nella cache con le nuove informazioni.

Prestazioni: hit rate, miss rate, amat
Una metrica per l'analisi delle prestazioni di una memoria sono **i tassi (o percentuali) di hit/miss (hit rate e miss rate)** e il **tempo medio di accesso in memoria (amat)**.

L'**hit rate** è il rapporto tra il numero hit (cioè il numero di volte in cui data una richiesta in memoria ho trovato l'istruzione o il dato nella cache) e il numero complessivo di accessi in memoria. E' uguale anche a *1-MISS RATE*

Il **miss rate** è il rapporto tra il numero di miss e il numero complessivo di accessi in memoria. E' uguale anche a *1-HIT RATE*

L'**AMAT** è l'**Average Memory Access Time** cioè il tempo medio di accesso in memoria, ossia il tempo medio di attesa da parte del processore per completare un'istruzione di lettura o scrittura dalla memoria. Influiscono su di esso hit rate e miss rate. Ipotizzando che vi sia anche la memoria virtuale e che il dato possa anche non trovarsi in main memory, l'amat si calcola come:

$$\mathbf{AMAT} = t_{\text{cache}} + MR_{\text{cache}}[t_{MM} + MR_{MM}(t_{VM})]$$

Nel caso in cui non vi fosse memoria virtuale abbiamo:
AMAT = t_cache + MR_cache * t_MM

t_cache: tempo di accesso alla cache
MR_cache: miss rate della cache
t_MM: tempo di accesso alla main memory
MR_MM: miss rate della main memory
t_VM: tempo di accesso alla memoria virtuale

Terminologia cache
Capacity (C): è il numero di parole che la cache può contenere (numero di byte)
Blocco: è un gruppo di parole adiacenti della cache
Block size (b): è il numero di parole contenute in un singolo blocco di cache, ossia il numero di bytes che definisce la grandezza del blocco della cache.
Numero di blocchi della cache (B): B=C/b è il rapporto fra la grandezza della cache e la dimensione del blocco di cache

Set della cache
Ogni cache è organizzata in S insiemi o **set** , ciascuno dei quali contiene uno o più blocchi *N* di parole. Quindi un **set** è un insieme della cache in cui sono contenuti uno o più blocchi *N* di parole della memoria.
Il **mapping** è una relazione tra l'indirizzo di un dato in memoria principale e la locazione di tale dato in cache. Ogni indirizzo di memoria è mappato in un set della cache. Questo serve al controller della cache per verificare dove andare a cercare il dato in cache.
Il **numero di blocchi in un set** è definito *N (degree of associativity).*
Il **numero di set in cache S=B/N** è dato dal rapporto tra il numero di blocchi totali della cache e il numero di blocchi in un set.

Tipologie di memorie cache
Abbiamo diverse categorie di memoria cache che sono organizzate in base al numero di blocchi che un set contiene:
▪ **Direct mapped: Ogni set è formato da 1 blocco di parole** e dunque un qualsiasi indirizzo di main memory è mappato in un solo blocco della cache. Il numero di set è uguale al numero di blocchi della cache. Il numero di righe è 1 poichè ogni set ha 1 solo blocco.
▪ **N-way set associative: Un set è formato da N blocchi di parole.**. L'indirizzo di main memory è mappato in un set della cache, ma il dato corrispondente a tale indirizzo può finire in uno qualsiasi degli N blocchi di quel set. Il numero di vie è compreso fra 1 < N < B
Il numero di set è da B/N ossia numero di blocchi totali della cache fratto il numero di blocchi in un set
▪ **Fully associative: C'è solo 1 set che contiene tutti gli B blocchi di parole della cache,** quindi il dato può andare in uno qualsiasi dei blocchi dell'unico set presente

Direct mapped cache

Una cache a mappatura diretta ha un solo blocco in ogni set, quindi è organizzata in tanti set quanti sono il numero di blocchi nella cache. Ogni indirizzo della memoria principale può essere mappato in un unico set. Immaginando che la main memory sia suddivisa anch'essa in blocchi di parole come la cache, possiamo dire che un indirizzo nel blocco 0 della main memory viene mappato nel set 0 della cache, un indirizzo del blocco 1 della main memory viene mappato nel set 1 della cache e così via fino all'ultimo set della cache.

Nella mappatura l'**indirizzo della main memory** viene logicamente suddiviso in 3 tre parti:

1- il **byte offset**, cioè i due bit meno significativi dell'indirizzo che sono sempre 00 perchè si procede di parola in parola. Servono ad indicare un byte dentro la parola (1° byte, 2° byte ecc.).

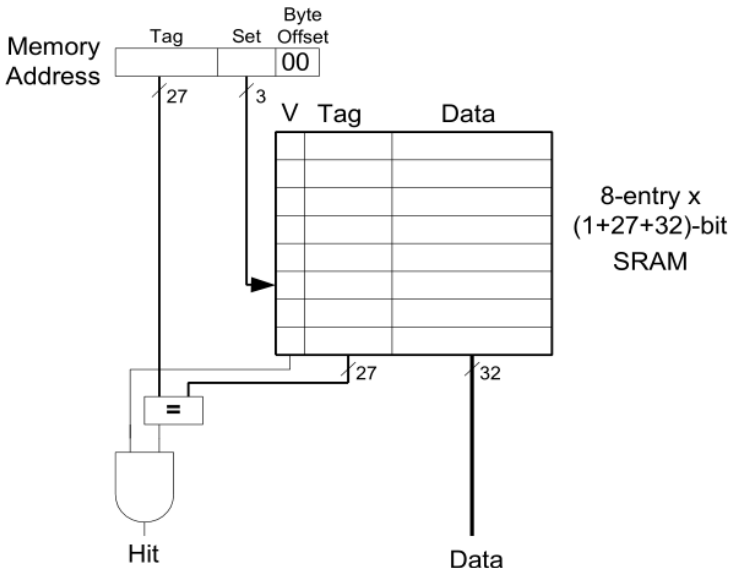
2- i **bit di set** che sono usati per indicare in quale set della cache viene mappato l'indirizzo: ad esempio se la cache è formata da 8 set terremo conto dei *log2 8 (log2 S)* = 3 bit successivi e quindi si mappa l'indirizzo nel set in cache corrispondente ai bit considerati (000, 001, 010..111). Ogni indirizzo di memoria viene mappato in un solo set della cache. Gli indirizzi sono mappati in modo ciclico o circolare, poiché molti di essi sono mappati nel medesimo set.

3- i bit restanti più significativi sono il **tag (etichetta)** e indicano quale dei tanti possibili indirizzi è effettivamente presente in quel particolare set.

La cache è realizzata con una SRAM , nella quale abbiamo 8 linee che corrispondono in questo caso a 8 set. Ogni set costituisce una riga contenente **32 bit di dato, 27 bit di tag e 1 bit di validità**.
Il **bit di validità** indica se il set contiene dati significativi cioè se il dato è affidabile oppure no. Se il bit di validità è 0 il contenuto del set non è significativo, se 1 allora è valido.

I 27 bit di tag vengono confrontati con i tag dell'indirizzo di memoria poiché il mapping non è 1:1 e dato che la cache ha una capacità minore della main memory e quindi in un particolare set possono essere memorizzati più indirizzi diversi.
Dati i bit di set dell'indirizzo, il controller verifica la linea corrispondente al set in cache.
Poi confronta il tag dell'indirizzo che la cpu ha richiesto col tag presente nella linea di cache.
Il confronto avviene con un comparatore : se i due tag sono uguali vuol dire che l'indirizzo presente nella linea di cache è proprio quello richiesto dalla CPU.
Il tag poi va in AND col bit di validità; se il bit di validità è 1 e il tag è 1 (cioè il tag di cache corrisponde al tag di indirizzo) allora si ha un **HIT. Cioè un 1 che indica che il dato è stato trovato in cache.** A questo punto si può leggere o scrivere il dato attraverso il bus data. In caso di miss il controller deve prelevare il dato dalla memoria principale.

Svantaggio: miss di conflitto.



Direct Mapped Cache

Miss obbligatori e di conflitto

Quando il processore all'inizio richiede dei dati avremo una situazione in cui abbiamo dei **miss obbligatori** dovuti al fatto che al primo accesso in cache questi dati non saranno presenti e bisogna prima caricarli dalla main memory ; il bit di validità corrispondente ai vari set sarà 0. Quindi poi i dati vengono prelevati dalla main memory e copiati in cache, settando il bit di validità a 1. Agli accessi successivi avremo degli hit in quanto i dati sono presenti nei vari set della cache con i vari bit di validità a 1. Dipendono dal fatto che una certa locazione di memoria sia essa contenente un istruzione o un dato viene richiesta per la prima volta (in lettura o scrittura), come se fosse un inizializzazione.

Quando due indirizzi richiesti di recente dal processore si mappano nel medesimo set della cache si verifica un **miss di conflitto.** Dunque l'indirizzo che deve essere mappato più recente espelle quello precedentemente mappato nel set. Il dato viene sovrascritto una volta che è stato prelevato dalla main memory. Anche il tag del set è sovrascritto.
Dipendono dal fatto che essendo la memoria cache come capacità inferiore alla memoria principale, il mapping fra gli indirizzi della memoria principale e i set della memoria cache non è 1:1 (funzione iniettiva) quindi più blocchi della main memory possono memorizzarsi nello stesso set della cache.
Siccome le cache a mappatura diretta hanno un solo blocco per ogni set, due indirizzi che si mappano nel medesimo set causano sempre un miss di conflitto. miss rate pari al 100% e tempo di accesso in memoria alto.

N-Way Associative cache

Una **tecnica efficace per ridurre i miss di conflitto** è quella di passare da una direct mapped cache ad una **n-way associative cache**, cioè una cache parzialmente associativa a N vie. Infatti lo scopo di una N-Way associative cache è proprio quello di ridurre i miss di conflitto causati dall'uso di una direct mapped cache.

In questa tipologia di cache per ogni set possiamo mappare più blocchi.

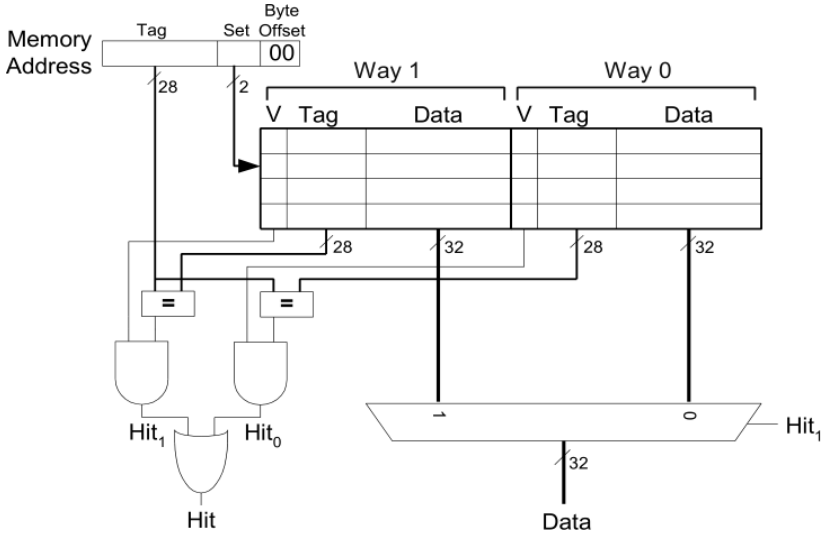
Quindi ci sono N blocchi in ciascun set: l'indirizzo di memoria viene mappato in uno specifico set, ma può essere copiato in uno qualsiasi degli N blocchi del set. Gli N blocchi presenti in ciascun set rappresentano il grado di associatività della cache. Nell'esempio considerato abbiamo 2 blocchi, quindi 2 vie.

In ciascun set sono presenti quindi 2 parole, ognuna con il suo tag e i dati corrispondenti ed il bit di validità. Siccome ogni set contiene 2 parole e la cache è formata da 8 parole, vuol dire che avremo 4 set in totale, quindi ci servono solo 2 ***bit identificativi di set***.

I primi due bit dell'indirizzo sono sempre il ***byte offset*** per discriminare il byte all'interno della parola. Il ***tag dell'indirizzo di memoria*** viene confrontato contemporaneamente con entrambi i tag (delle 2 way presenti) del set corrispondente. **Quindi per ogni indirizzo di memoria si estrae il tag e lo si confronta con gli N tag delle way associate al set corrispondente, con N che dipende dal grado di associatività della cache**.

Il confronto dei tag avviene sempre attraverso dei comparatori , dopodiché si mette il risultato in AND col bit di validità. Abbiamo 2 possibili situazioni di hit *Hit1 (way 1)* e *Hit0 (way 0)* che corrispondono al fatto che la prima via può combaciare col tag dell'indirizzo di memoria o che l'altra via combacia col tag. Abbiamo un hit se il tag dell'indirizzo richiesto è presente in almeno una delle vie corrispondenti al set, quindi usiamo una porta OR per stabilire quale dei due Hit sia andato a buon fine. Se si verifica un hit in una delle due vie, dobbiamo selezionare i dati della via in cui si è verificato l'hit: per farlo usiamo un multiplexer 2:1 che seleziona di volta in volta i dati o della way1 o della way0. Esso è pilotato dal valore di *Hit1:* se esso è pari a 1 allora il mux seleziona il dato corrispondente alla way1, se esso è pari a 0 allora il mux seleziona il dato corrispondente alla way0, nel caso in cui il valore finale *Hit* sia pari a 1.

Questa tipologia di cache ha un miss rate inferiore alla cache direct mapped, tuttavia è generalmente più lenta e più costosa da realizzare a causa del mux di uscita e dei comparatori aggiuntivi. Sorge inoltre il problema di quale via "sostituire" quando entrambe sono piene. L'associatività riduce i miss di conflitto. Abbiamo solo i miss obbligatori iniziali

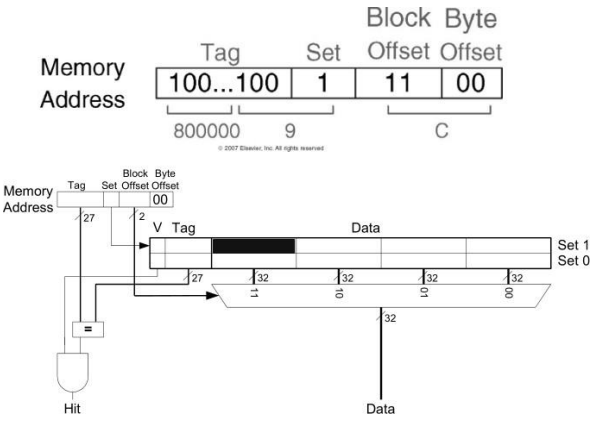


Sfruttare la località spaziale: dimensione del block size

Nell'ipotesi che **aumentiamo il block size**,cioè il numero di parole presenti in ogni blocco, sfruttiamo al meglio il principio di **località spaziale e riduciamo i miss obbligatori** poiché carichiamo indirizzi vicini nel blocco senza doverli andare a riprendere uno per uno in main memory. Sfruttiamo la località spaziale in quanto è probabile che nel futuro ci servirà uno degli indirizzi vicini a quello richiesto. Quindi una cache utilizza blocchi di dimensioni maggiori per memorizzare più parole di memoria consecutive col vantaggio che in caso di miss vengono copiate in cache la parola non trovata e anche le parole adiacenti nel blocco di memoria principale. Gli accessi successivi hanno dunque maggiore probabilità di dare luogo a hit grazie alla località spaziale.

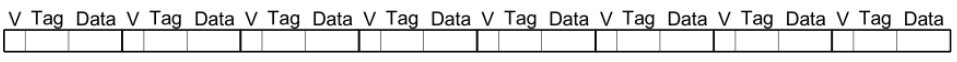
Nell'esempio considerato abbiamo una cache di capacità 8 parole di tipologia direct mapped (ogni blocco corrisponde ad un set). Se un blocco ha un block size di 4 parole, significa che avremo 2 set, con 1 bit di set. Ogni set conterrà quindi 4 parole, che corrispondono ad 1 blocco. Avremo dunque due bit prima del bit di set , detti **block offset** che ci dicono quale parola all'interno del set scegliere (1°, 2°, 3°..). Si confronta il tag e poi si sceglie la parola corrispondente attraverso un mux che seleziona il contenuto associato alla parola giusta, dato il block offset.

Una dimensione del block size maggiore significa a parità di capacità della cache che questa ha meno blocchi, aumentano quindi i miss di conflitto. Inoltre serve più tempo in caso di miss per prelevare dalla main memory tutte le parole del blocco e trasferirle in cache.



Fully associative cache

Questa tipologia di cache ha una grande versatilità in quanto è costituita da 1 unico set, nel quale sono presenti tante vie corrispondenti al numero B di blocchi. Un indirizzo di memoria può essere mappato in una qualsiasi delle vie associate all'unico set. Essa consente la massima flessibilità nella scelta della locazione nella cache in cui posizionare un blocco di memoria quindi lo spazio può essere utilizzato in modo efficiente comportando una riduzione dei miss di conflitto. Lo svantaggio principale è che il costo della ricerca di un blocco in cache è notevole: alla richiesta di un dato dobbiamo confrontare tanti tag quanti ne sono i blocchi presenti nell'unico set, dal momento che il dato potrebbe trovarsi in un blocco qualsiasi. Un mux con tanti ingressi e un'uscita seleziona il dato corretto in caso di hit. Quindi questa cache richiede hardware aggiuntivo per i confronti dei tag ed un elevato numero di comparatori.



Bit di validità: dettaglio
Siccome deve esserci una certa coerenza fra il contenuto di un blocco in cache e il contenuto di un blocco della main memory, viene introdotto un bit di validità per mantenere questa coerenza. Esso è un bit di controllo, ed è necessario per ogni blocco. Indica se il blocco contiene o meno dati validi.
Non deve però essere confuso con il ***bit di modifica***, che indica se il blocco è stato modificato o meno durante il periodo in cui è stato nella cache, esso serve soltanto nei sistemi che non fanno uso del metodo write-through.
I bit di validità dei blocchi che si trovano nella cache vengono tutti posti a 0 nell'istante iniziale di accensione del sistema e, in seguito, ogni volta che nella memoria principale vengono caricati programmi e dati nuovi dal disco.
Il bit di validità di un certo blocco della cache viene posto a "1" la prima volta che esso è chiamato a contenere istruzioni o dati trasferiti dalla memoria principale; poi, ogni volta che le locazioni di un blocco della memoria principale vengono interessate da un trasferimento di nuove istruzioni o di nuovi dati (swap) dal disco (virtual memory), viene effettuato un controllo per determinare se qualcuno dei blocchi che stanno per essere sovrascritti fossero o meno presenti nella cache.
Se nella cache c'è una copia del blocco della main memory che deve essere aggiornato mediante swap dalla virtual memory, il suo bit di validità viene posto a "0"; in tal modo, si garantisce che nella cache non ci siano dati obsoleti e si mantiene una certa coerenza tra cache e main memory. Il bit di validità a "0" candida immediatamente il blocco della cache all'interno di un set ad essere sovrascritto e quindi sostituito.

Il Direct Memory Access - DMA
I trasferimenti dal disco(memoria di massa) alla memoria principale vengono effettuati mediante una logica di controllo non gestita dalla CPU, bensì da un meccanismo detto di DMA (Direct Memory Access). La CPU non si accorge dello scambio di pagine fra la memoria virtuale e la main memory. Durante il trasferimento la CPU non interviene, perché è la logica di controllo che gestisce le linee indirizzo e le linee dati, fornendo i segnali di controllo necessari. In una operazione di questo genere vengono trasferite grandi quantità di istruzioni e dati organizzati in entità dette "**pagine**" ciascuna delle quali di solito contiene migliaia di locazioni e quindi centinaia di blocchi.
Le **pagine** quindi sono segmenti di riferimenti di memoria contigui, come i blocchi della cache , ma molto più grandi (ordine di Kb), in quanto la virtual memory e la main memory hanno capacità maggiori. Normalmente, le pagine di programma e quelle di dati vanno nella memoria centrale senza coinvolgere la cache.

Svuotamento (flush) della cache. Problema della coerenza fra cache, mm, vm
In una gerarchia di memoria a 3 livelli (cache, main memory, virtual memory) si deve sempre garantire che vi sia una certa coerenza fra i livelli. Questa gerarchia di livelli fa si che la cache comunichi con la main memory *(end to end)* e la memoria principale comunichi con la virtual memory. La cache non comunica direttamente con la virtual memory. La memoria principale si trova in mezzo a richieste che vengono contemporaneamente dalla cache e dalla virtual memory e c'è bisogno di un meccanismo che mantenga la coerenza in modo tale che la memoria principale soddisfi le richieste sia della cache che della virtual memory.
In particolare quando abbiamo un write hit nella cache, se la politica adottata è quella di write-back significa che si sovrascrive il blocco in cache senza sovrascrivere anche la

memoria principale. Quindi è come se si perdesse la “coerenza” fra le due memorie: si deve far si che questa perdita momentanea di coerenza non infici sull’intero sistema. Quindi si pone il bit di modifica del blocco in cache pari a 1 e quando deve essere sostituito per far spazio ad un nuovo blocco lo dobbiamo sovrascrivere in memoria principale nel suo indirizzo corrispondente , che otteniamo guardando il tag del blocco. Un altro caso lo abbiamo quando sovrascriviamo una parola all’interno del blocco in cache ma non sovrascriviamo il blocco in main memory e questo blocco fa parte di una pagina più grande che dobbiamo swappare con la virtual memory. Quando dobbiamo swappare la pagina, i dati nella pagina devono essere aggiornati : ciò significa che dobbiamo prendere tutti i blocchi che fanno parte della pagina che stanno in cache che hanno il bit di modifica pari a 1 e dobbiamo fare uno svuotamento, cioè un **flush** del blocco della cache che deve essere spostato in main memory . Poichè la pagina viene poi swappata con la virtual memory per far contenere qualcosa di nuovo, tutti i bit di validità del blocco in cache poi vengono settati a 0 poichè i dati non sono più coerenti con i nuovi presenti nella pagina aggiornata in main memory.
Il sistema operativo svolge tutte queste operazioni necessarie in modo molto efficiente. Questa necessità di garantire che due diverse entità (i sottosistemi della CPU e del DMA nel caso presente) utilizzino la stessa copia dei dati viene chiamata problema della coerenza della cache.

Algoritmi di sostituzione (di un blocco di parole in cache)
In una **cache ad indirizzamento diretto**, la posizione di ogni blocco è predefinita ed abbiamo che ogni set corrisponde ad un blocco, quindi **non esiste alcuna strategia di sostituzione** visto che la sostituzione avviene per sovrascrizione dell’unico blocco presente nel set col nuovo blocco aggiornato.
In una **cache associativa a N vie** quando dobbiamo trascrivere un nuovo blocco in un set dobbiamo scegliere quale blocco sostituire quando il set è pieno. Nel caso ci fossero dei blocchi in un set con bit di validità pari a 0 si incomincia a sovrascrivere quelli in quanto contengono dati non più significativi. Se il set è pieno e tutti i bit di validità valgono 1 e si verifica la situazione in cui dobbiamo sostituire in memoria cache un blocco con un nuovo blocco richiesto dalla cpu e prelevato dalla main memory , dobbiamo scegliere uno dei blocchi in cache con bit di validità pari a 1 da sostituire. In questo caso si sfrutta il principio della **località temporale** per cui dati a cui abbiamo fatto accesso di recente saranno quelli con più probabilità di essere riusati nel prossimo futuro.
Riassumendo, quando un nuovo blocco deve essere portato nella cache, e tutte le posizioni che potrebbe occupare contengono dati validi, il controllore della cache deve decidere quale blocco, tra quelli esistenti, sovrascrivere. In generale, l'obiettivo è quello di mantenere nella cache quei blocchi che hanno una maggior possibilità di essere nuovamente utilizzati nel prossimo futuro. Una possibile strategia è di tener presente che la località dei riferimenti suggerisce che i blocchi a cui c’è stato accesso di recente hanno elevata probabilità di essere utilizzati nuovamente entro breve tempo.

Algoritmo LRU
Una strategia di sostituzione è quella definita dall **algoritmo di sostituzione LRU (Least Recently Used)** in cui viene sovrascritto il blocco a cui non si accede da più tempo nel set. Tale blocco prende il nome di blocco utilizzato meno di recente LRU. Per utilizzare l'algoritmo LRU, il controllore della cache deve mantenere traccia di tutti gli accessi ai blocchi mentre l'elaborazione prosegue attraverso un **contatore degli accessi al blocco**. Si stabilisce una “graduatoria” dei blocchi nel set per stabilire quello meno usato di recente,

stabilendo una classifica di posizionamento dei blocchi. Ad esempio se abbiamo 4 blocchi, con due bit indicheremo con 00 il blocco usato più recente in prima posizione, 01 il secondo, 10 il terzo e con 11 il blocco usato meno di recente.

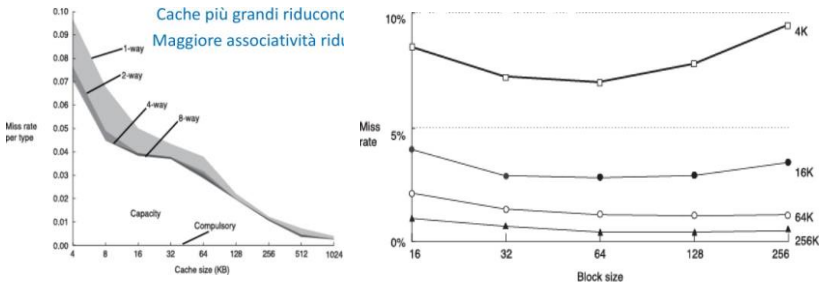
Gestione del contatore degli accessi nell'algoritmo LRU
Quando si ha un successo nell'accesso al blocco di un set (**hit**):
- il contatore di quel blocco viene posto a 0 poichè diventa il blocco usato più di recente LRU. I contatori con valori inferiori a quelli del blocco a cui si accede vengono incrementati di uno, mentre tutti gli altri con valore superiore rimangono invariati.

Quando, invece, l'accesso fallisce (**miss**) si aprono due possibilità:
- il set non è pieno, quindi abbiamo un bit di validità pari a 0 : viene caricato il nuovo blocco dalla memoria principale viene e il contatore degli accessi viene posto a 0 , il bit di validità passa a 1 e il valore di tutti gli altri contatori incrementa di 1.
- tutto il set è pieno, si rimuove il blocco meno usato di recente, il cui contatore ha valore 3, e si pone il nuovo blocco al suo posto, mettendo il contatore a 0. Gli altri tre contatori del set vengono incrementati di 1.

Questa strategia può generare molti miss ad esempio nel caso di accessi sequenziali a un vettore di elementi che è leggermente troppo grande per poter stare tutto nella cache e dunque le prestazioni sono molto scadenti con molti miss. **Quindi ci sono strategie diverse che utilizzano sempre l'algoritmo LRU ma ogni 5-6 accessi sostituiscono un blocco in modo casuale (random).**

Tipi di miss. Analisi del miss rate.
- **Compulsory**: primo accesso ad un dato ossia i ***miss obbligatori***
- **Capacity**: la cache è troppo piccola per contenere tutti i dati di interesse ***miss di capacità***
- **Conflict**: due dati sono indirizzati nella stessa locazione nella cache ossia i ***miss di conflitto***

Dai grafici si mostra che avendo fissata una dimensione del blocco in cache , all'aumentare della capacità della cache , fissato il numero di vie , il numero di set aumenta dato che aumentiamo la capacità. Quindi si riducono i miss di capacità e avremo meno miss in generale. Inoltre fissata una certa dimensione, se aumentiamo il numero di vie si diminuiscono i miss di conflitto, però aumentare a dismisura il numero di vie ad un certo punto significa complicare l'hardware e non ottenere un reale beneficio .
Mentre fissato il cache size e un numero n di vie , notiamo che all'aumentare del block size il miss rate prima migliora col numero di miss obbligatori dato che sfruttiamo la località spaziale con blocchi più grandi, ma poi ad un certo punto peggiora all'aumentare del block size. Questo perchè se aumentiamo il block size in modo molto grande arriveremo in un punto in cui diminuisce ovviamente il numero di set e quindi avremo più miss di conflitto , perchè abbiamo pochi set.

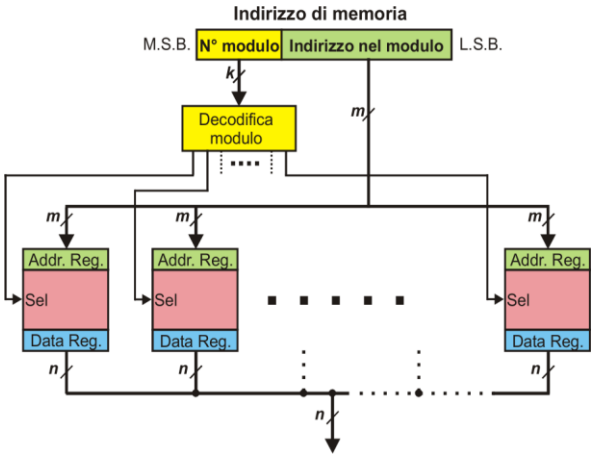


Frazionamento della memoria in moduli
E utile che i trasferimenti da/verso le unità di memoria possano essere effettuati alla stessa frequenza dell'unità più veloce, cioè la cache. Ciò non può avvenire se si accede nello stesso modo all'unità più lenta e a quella più veloce.
Tale risultato può invece essere raggiunto se si sfrutta il parallelismo nell'organizzazione dell'unità più lenta. Un modo efficiente per introdurre il parallelismo (spaziale) nella gestione della memoria principale consiste nell'impiego di un'organizzazione in più moduli della memoria. Se la memoria principale di un calcolatore è strutturata come una collezione di moduli fisicamente separati, ognuno con il proprio registro degli indirizzi (**Address Register, AR**) e registro dei dati (**Data Register, DR**), le operazioni di accesso alla memoria possono procedere contemporaneamente in più di un modulo.
Così, si può incrementare la frequenza di trasmissione delle parole da o verso il sistema di memoria principale ed inoltre possiamo trasferire più parole in parallelo, impiegando meno tempo.
Abbiamo un metodo interlacciato e un metodo non interlacciato della memoria che corrispondono a due metodi di distribuzione degli indirizzi.

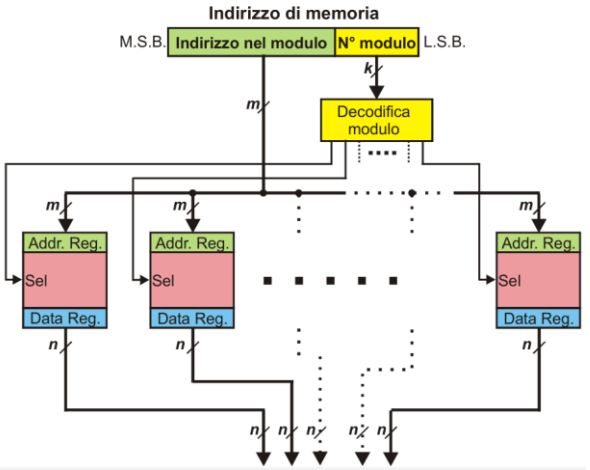
1- Non interlacciato.
L'indirizzo di memoria generato dalla CPU viene decodificato nel seguente modo:
I **k bit più significativi** indicano **uno degli n moduli,** e gli **m bit meno significativi** indicano una particolare **parola all'interno del modulo**.
I k bit attraverso un decoder vanno ad identificare un certo modulo, gli m bit meno significativi vanno a finire in un **address register** ed identificano l'offset all'interno del modulo, cioè quale parola all'interno del modulo (che contiene tutti indirizzi di memoria contigui) ci stiamo andando a riferire. La parola viene presa in considerazione e il dato corrispondente viene messo in un **data register** e poi verrà inviato dal bus fino alla CPU o alla memoria virtuale. **I blocchi di cache saranno contenuti tutti all'interno di un modulo**. Quindi quando avremo un trasferimento da cache alla main memory di un blocco questo andrà ad essere copiato nel modulo corrispondente della main memory, in base ai k bit dell'indirizzo che specificano il numero del modulo. Intanto attraverso il DMA negli altri moduli che sono liberi può avvenire il trasferimento di una pagina con la virtual memory.
Quindi può avvenire in contemporanea un trasferimento di una pagina con la memoria virtuale in un modulo ed un trasferimento di un blocco dalla cache ad un altro modulo. Questa frazionamento in moduli allora consente il trasferimento in contemporanea di dati dalla cache e dalla virtual memory verso la main memory e viceversa purché si riferiscano a moduli differenti.
Questo tipo di organizzazione **non è quello più efficiente** perchè i trasferimenti di blocchi della memoria cache che ha una ridotta capacità e quindi si satura facilmente, con la main memory sono più frequenti rispetto ai trasferimenti di pagina tra la memoria virtuale e la main memory che sono più lenti e meno frequenti. **Per questo si passa ad un tipo di indirizzamento interlacciato**

2- Interlacciato
Funziona in maniera duale: **i bit meno significativi (k) selezionano un modulo, mentre i bit più significativi (m) identificano una parola nel modulo.**

Questo vuol dire che indirizzi contigui che stanno nello stesso blocco corrispondono a moduli differenti, in cui hanno le parole dello stesso blocco sono posizionate allo stesso offset, ma ovviamente in blocchi differenti. Il trasferimento di un blocco avviene andando a leggere i differenti moduli e può avvenire in parallelo. La lettura dei blocchi dalla cache alla main memory avviene più velocemente, però siccome si impegnano tutti i moduli se c'è un trasferimento alla memoria virtuale finché non si libera un modulo questo trasferimento non può avvenire.



Non interlacciato



Interlacciato

Cache multilivello

Oggigiorno la memoria cache è costituita da una gerarchia di 2 o 3 livelli: ***L1, L2, L3.***
Questa tipologia di cache è detta multilivello. I livelli successivi sono cache un po' più grandi ma più lente ovviamente della cache di primo livello. In questi casi le politiche di gestione tra cache e main memory si ripetono anche fra i livelli di cache.
Vi possono essere due tipologie di cache multilivello: **inclusiva** o **esclusiva**

1- Inclusiva: la cache L2 contiene tutti i dati presenti in L1 , in quanto è più grande in termini di capacità di L1 e può contenerla. Quindi L1 è un sottoinsieme più veloce di L2. (similmente L2 è contenuta in L3). Il tipo di politica di solito è **write-through**, quindi si mantiene subito una coerenza fra i vari livelli di cache. Quando abbiamo un **miss** in L1, il controller vede se il blocco ricercato è contenuto in L2: se lo trova il blocco viene caricato in L1 e poi nel caso avviene una sostituzione di un blocco in L1 con un algoritmo di sostituzione che fa si che il blocco che viene sovrascritto venga caricato in L2 **(copy back)**. Se abbiamo un miss in L1 e in L2 il blocco viene prelevato dalla main memory e viene copiato sia in L1 che in L2 per mantenere la coerenza fra i livelli.

2-Esclusiva: la cache L2 contiene solo i dati di copy back da L1 (victim cache)
quindi non contiene gli stessi dati ma solo i dati che vengono passati da L1 quando c'è una sostituzione di un blocco di L1 (blocco vittima).
La memoria complessiva della cache è data dalla somma delle capacità dei vari livelli.
Se abbiamo un miss in L1 e hit in L2 il blocco trovato in L2 deve passare in L1 : quindi avviene uno scambio fra i blocchi cioè quello meno recente in L1 viene sostituito con quello più recente appena trovato (hit) in L2, ovviamente rispettando i bit di set.

Se abbiamo un miss in L1 e L2: il blocco viene letto dalla main memory ed è memorizzato direttamente in L1 e verrà trasferito in L2 solo quando esso diventa un blocco vittima cioè quando L1 diventa satura e dunque il blocco deve essere sostituito.

Cache condivise
Nelle architetture con più core i livelli esterni della cache L2 o L3 possono essere condivisi fra più core di uno stesso processore. Cio' consente un uso più efficiente del livello condiviso quando per esempio un core è inattivo e allora l'altro può utilizzare per se il livello condiviso oppure entrambi sfruttano la stessa cache L2.

La memoria virtuale
Al gradino più basso della gerarchia delle memoria abbiamo il **disco rigido** in tecnologia magnetica o a stato solido. Esso è **grande ed economico ma terribilmente lento**.
I programmi e i software risiedono in memoria di massa; **è il sistema operativo che si fa carico del mapping degli indirizzi di memoria virtuale del programma che risiede in memoria di massa in indirizzi di memoria principale**. Inoltre il sistema operativo carica il programma dalla memoria di massa in main memory in una locazione **casuale** ed in modo dinamico. La tecnica della memoria virtuale prevede quindi che il sistema operativo assegni una parte degli spazi liberi nella memoria centrale ai **segmenti** di un programma installato su disco. Infatti per un problema di efficienza il sistema operativo non è detto che carichi il programma in un insieme contiguo di indirizzi, ma in realtà lo può caricare in diversi segmenti dato che la memoria principale è suddivisa in **pagine** ossia dei blocchi di indirizzi continui (decine di kb) . **È probabile che il programma segmentato venga caricato in più pagine nella main memory non necessariamente contigue fra loro**. La suddivisione in pagine consente una gestione più efficace della memoria principale.

Swapping
Anche la **main memory si satura** : un programma molto pesante che non ha sufficiente spazio in main memory per continuare ad essere eseguito deve salvare parte di esso di nuovo nella memoria virtuale: questa viene detta un'**operazione di swapping**. La parte del programma caricata nella virtual memory viene ripresa successivamente.
Lo swapping dipende dal fatto che la main memory ha una capacità ovviamente molto minore della virtual memory. Le operazioni di swapping essendo la memoria di massa più lenta della main memory sono delle operazioni "pesanti" dal punto di vista del tempo perso. Bisogna cercare di rendere questa operazione di swapping meno onerosa possibile.
La località spaziale e temporale alleggeriscono le operazioni di swapping dato che è molto probabile che vengano richiesti indirizzi contenuti in una stessa pagina di memoria, senza necessariamente dover effettuare operazioni di swap continuamente.
Inoltre in un'operazione di swap è possibile che vengano scambiate molte pagine.

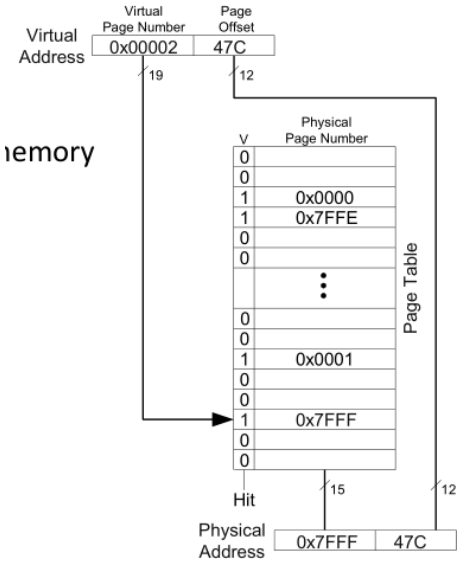
La lentezza delle operazioni di swapping dipende anche dal fatto che le memorie virtuali sono costruite con una tecnologia più lenta di quella delle cache e delle ram. Infatti un'unità a dischi rigidi magnetici contiene uno o più dischi rigidi ciascuno dotato di una testina di lettura/scrittura. La testina si sposta sulla corretta locazione e sfrutta l'elettromagnetismo per leggere o scrivere dati sul disco in rotazione sotto di lei. La testina impiega vari millisecondi per raggiungere la corretta locazione sul disco: un tempo milioni di volte più lento di quello del processore. I dischi rigidi magnetici sono progressivamente stati sostituiti da dischi a stato solido perché in questi ultimi la velocità di lettura ha ordini di grandezza maggiori.

Quindi la bandwidth è minore di una cache o una ram poichè dipende dalla velocità in cui gira il supporto magnetico ed essendo meccanico non raggiunge mai le velocità di sistemi elettronici come le sram o le dram. Inoltre c'è un overhead nel tempo di lettura e scrittura dovuto alla latenza per trovare il punto iniziale da cui andare a scrivere. Questo richiede il movimento meccanico della testina, operazione lenta. La latenza maggiore ad oggi si ha quando bisogna caricare inizialmente un programma in main memory. Per questo oggi usiamo una memoria a stato solido che ha un tempo di accesso più veloce di un disco rigido.
L'obiettivo di aggiungere un disco rigido alla gerarchia di memoria è quello di dare l'illusione di uno spazio di memoria molto grande pur mantenendo la velocità delle memorie più rapide per la maggior parte degli accessi.

Gestione della memoria virtuale
I programmi possono accedere a un qualsiasi dato della memoria virtuale, quindi devono utilizzare **indirizzi virtuali** che specificano le locazioni dei dati nella memoria virtuale. Quindi gli indirizzi virtuali appartengono alla memoria di massa. Poi come sappiamo abbiamo la main memory che però è una memoria con una capacità minore della memoria di massa: essa contiene un **sottoinsieme della parte di memoria virtuale più recentemente utilizzata.** In questo modo la main memory (memoria fisica) agisce da cache della memoria virtuale. Anche in questo caso avviene quindi un'**operazione di mapping** ossia **gli indirizzi virtuali vengono tradotti in indirizzi fisici della main memory**. In questo caso il mapping avviene seguendo una tabella degli indirizzi chiamata **lookup table** e non in modo automatico perchè lo swapping tra main memory e virtual memory è più raro ed è quindi gestito dal sistema operativo che gestisce la tabella degli indirizzi. Quindi c'è una politica di mapping fully associative dove si vede una tabella per effettuare il mapping da indirizzi virtuali a fisici. Infatti gli indirizzi virtuali corrispondono a varie pagine nell'hard disk ed una parte di essi, che deve essere caricata in main memory, viene tradotta in indirizzi fisici.
La memoria virtuale è suddivisa in pagine virtuali, così come la memoria fisica è suddivisa in pagine fisiche della stessa dimensione. Per evitare mancanze di pagina dovute ai conflitti ogni pagina virtuale può essere quindi mappata in una qualsiasi pagina fisica, in questo modo la memoria fisica si comporta da cache fully associative per la memoria virtuale.

Tabella delle pagine - lookup table e il TLB (Translation Lookaside Buffer)
Si usa la tabella delle pagine per effettuare la traduzione dell'indirizzo virtuale in fisico. Essa ha un entry per ogni pagina virtuale che indica in quale pagina fisica tale pagina virtuale si trova oppure che è presente solo su disco. Ogni istruzione di lettura o scrittura in memoria richiede quindi un accesso alla tabella delle pagine seguito da un accesso alla memoria fisica: l'accesso alla tabella delle pagine traduce l'indirizzo virtuale usato dal programma in indirizzo fisico, che viene poi utilizzato per l'effettivo accesso di lettura o scrittura del dato.
Ogni entry della pagina virtuale contiene quindi un numero di pagina fisica e un bit di validità: se esso vale 1, la pagina virtuale viene mappata nella pagina fisica specificata nell'elemento, altrimenti la pagina virtuale va caricata dal disco e poi si aggiorna la tabella.
La tabella delle pagine è normalmente così grande da essere memorizzata in memoria fisica.
Le operazioni di load/store richiedono 2 accessi alla main memory:
- Il primo per la traduzione (lettura della page table)
- Il secondo è l'accesso vero e proprio al dato dopo la traduzione

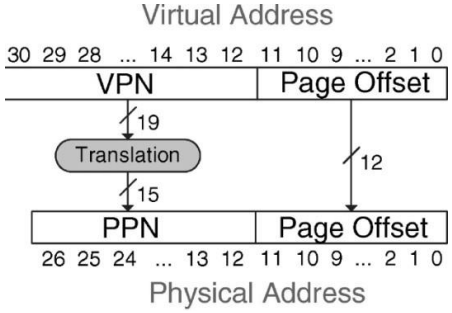
Questo dimezzerebbe le performance della main memory e per evitare ciò si usa il **Translation Lookaside Buffer TLB** che tiene in cache poche entry delle migliaia della tabella delle pagine che vengono utilizzate più di frequente. Infatti il processore tiene tracciadi un certo numero di entry della tabella delle pagine nel TLB e guarda in esso per trovare latraduzione dell'indirizzo prima di dover accedere alla memoria principale per leggerla dalla tabella delle pagine. Nei programmi reali la stragrande maggioranza degli accessi nel TLB da luogo a hit (99%) evitando i lenti accessi in memoria fisica per leggera la tabella delle pagine. Il TLB è organizzato come una cache fully associative. TLB sfrutta il fatto che gli accessi alla page table godono di una elevate località temporale: essendo le dimensioni di una pagina relativamente grandi vi è un'alta probabilità che operazioni di load o store consecutive richiedano un accesso alla medesima pagina.



Come avviene la traduzione dell'indirizzo virtuale in fisico

I 12 bit meno significativi dell'indirizzo virtuale da 31 bit indicano il page offset cioè all'interno della pagina a quale locazione ci stiamo riferendo e non richiedono traduzione.

I successivi 19 bit più significativi indicano il numero di pagina virtuale e vengono tradotti nei 15 del numero di pagina fisica. Per prendere la locazione corrispondente nella pagina fisica mappata viene preso il page offset dell'indirizzo virtuale.



La MMU - Memory Management Unit

Il compito di tradurre l'indirizzo virtuale in un indirizzo che punti ad una locazione

effettivamente esistente nel sistema (indirizzo fisico) è demandato ad una struttura logica contenuta nel chip del processore, che si chiama Memory Management Unit (MMU) e cheopera sotto il controllo software del sistema operativo. Essa controlla la lookup table e fa il mapping dall’indirizzo virtuale a fisico.