Esercitazione del 21/05/2010 - Soluzioni

1. Introduzione

Una CPU moderna ha generalmente una velocità di esecuzione delle istruzioni molto più alta della capacità delle memorie DRAM di fornire dati. In generale, possiamo dire che più alta è la quantità di dati memorizzabili in una memoria più bassa è la velocità con cui si può accedere ai dati memorizzati.

Tempo di accesso ai dischi	Tempo di accesso alle memorie DRAM	Tempo di esecuzione di una istruzione nella CPU
~10000ns	~60ns	~2ns

Se la CPU dovesse, per ogni lettura da memoria, caricare il dato direttamente dalla DRAM (o peggio direttamente dai dischi) passerebbe la maggior parte del tempo ad attendere il completamento dell'accesso ai dati.
Una soluzione possibile è interporre tra la memoria e la CPU una memoria molto più piccola ma allo stesso tempo in grado di lavorare alla stessa velocità della CPU. In questa memoria-tampone, detta **cache**, verranno memorizzate di volta in volta le istruzioni più richieste dalla CPU. Quando la CPU richiede un dato, prima di accedere alla DRAM, il dato viene cercato all'interno della cache. Se il dato è presente (**hit**) allora può essere fornito velocemente alla CPU. Se il dato non è presente (**miss**) allora deve essere caricato dalla DRAM e messo in una qualche posizione della cache, eventualmente eliminando un altro dato (si ricordi che una cache è notevolmente più piccola dello spazio disponibile nella DRAM). L'interposizione di una cache tra CPU e DRAM in generale peggiora in qualche misura l'accesso alla DRAM rispetto al singolo accesso (oltre al tempo di accesso vero e proprio occorre aggiungere il tempo di verifica della condizione di **miss**). Questo peggioramento però è largamente compensato dalla accelerazione che si ottiene nel caso si verifichi (la maggior parte delle volte) una condizione di **hit.**

2. Località spaziale e temporale

Il vantaggio in termini di velocità ottenuto usando le cache si basa su due principi teorici:

- Località temporale: *una CPU tende a richiedere dati che sono stati usati di recente (valido soprattutto per la memoria dati)*
- Località spaziale: *una CPU tende a richiedere dati vicini a quelli usati di recente (valido soprattutto per la memoria istruzioni)*

Se i dati usati di recente (*località temporale*) vengono mantenuti nella cache allora il tempo per accedere a questi dati (quelli maggiormente usati dalla CPU) sarà notevolmente minore del tempo necessario per recuperare i dati usati raramente presenti nella DRAM.
Attraverso cache organizzate a blocchi e trasferimenti paralleli tra DRAM e cache è possibile migliorare le prestazioni del sistema sfruttando il principio di *località spaziale*. Nel caso si verifichi una condizione di **miss** invece di caricare un singolo dato nella cache viene caricato un blocco di più dati spazialmente vicini, ad esempio la parola richiesta e le 3 parole successive (organizzando opportunamente l'architettura delle DRAM e delle linee di comunicazione con la cache è possibile

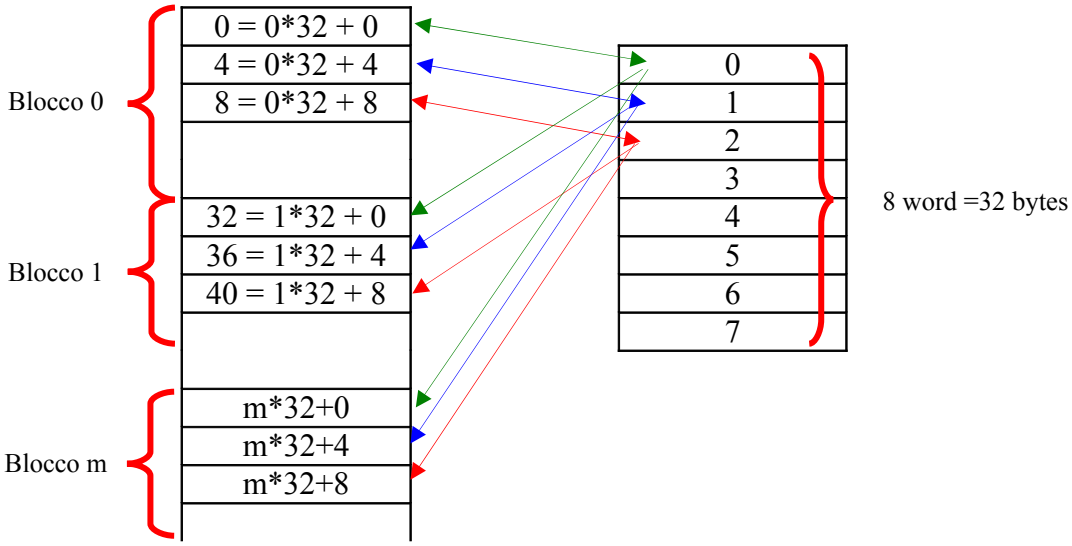
trasferire un blocco di dati in un tempo paragonabile a quello necessario per trasferire una sola parola). I successivi accessi alla cache avranno in questo caso più probabilità di trovare il dato richiesto aumentando così la frequenza di **hit**.

3. Cache a corrispondenza diretta

Supponiamo per semplicità di avere una cache di sole 2^3 parole. Lo spazio di indirizzamento del MIPS permette di indirizzare teoricamente 2^{30} parole. Occorre un metodo per decidere dove mettere di volta in volta nella cache un dato letto dalla DRAM, una regola di corrispondenza. Nelle **cache a corrispondenza diretta** ad ogni parola della memoria corrisponde ad una prefissata posizione nella cache. Ne segue che una cella della cache corrisponde a più celle nella DRAM. Se un dato viene trasferito dalla memoria alla cache, la posizione che assume all'interno della cache è fissa e determinata a priori. Il modo più semplice per determinare questa corrispondenza è sfruttare alcuni bit dell'indirizzo originario per costruire l'indirizzo nella cache. Nell'esempio considerato, possiamo pensare di "smontare" l'indirizzo in questo modo:

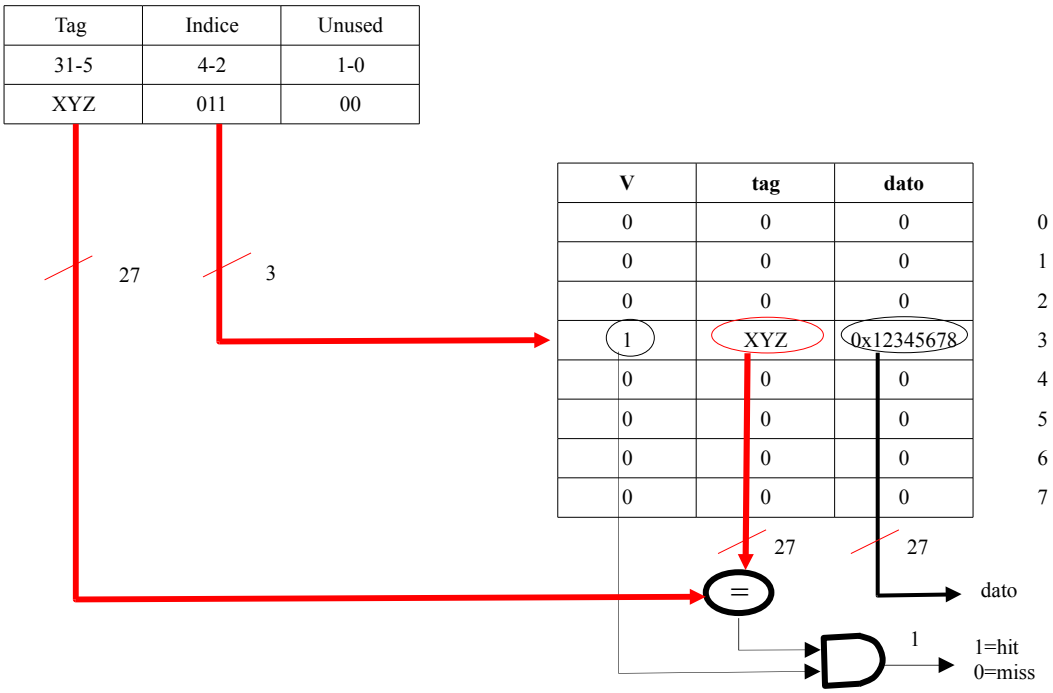
tag:31-5	cache:4-2	byte offset:1-0

Poiché i trasferimenti da e verso la memoria avvengono sempre allineati alla word i primi due bit meno significativi 1-0 saranno sempre a 0 e quindi possono essere omessi. Dei restanti bit utilizziamo i successivi tre bit 4-2 per indirizzare la cache. In questo modo otteniamo una corrispondenza di questo tipo:



Poiché ad ogni cella della cache corrispondono più celle della DRAM occorre memorizzare insieme al dato anche un identificativo (**tag**) che indichi a quale cella DRAM corrisponde il dato. In aggiunta occorre un ulteriore bit (**validate**) per indicare se il dato a cui si riferisce il tag presente in cache è valido (è stato letto effettivamente dalla DRAM) oppure va ricaricato.

La cache a corrispondenza diretta dell'esempio sarà quindi organizzata come segue:



Supponiamo che la CPU debba eseguire i seguenti accessi alla memoria: **0,4,0,32.** Supponiamo che la cache sia ancora inutilizzata. La situazione della cache nello stato iniziale sarà:

V	tag	dato	
0	0	0	0
0	0	0	1
0	0	0	2
0	0	0	3
0	0	0	4
0	0	0	5
0	0	0	6
0	0	0	7

Al primo accesso la CPU richiede il dato all'indirizzo:

0 = 0x00000000 = <u>00000 00000 00000 00000 00000 00</u> <u>000</u> 00_2

a cui corrisponde l'indice $\mathbf{000_2}$ ed il tag $\mathbf{00000000000000000000000000_2}$**.**
La cache seleziona la riga *0* della cache, estrae il tag memorizzato e lo confronta con il tag dell'indirizzo. In questo caso i tag coincidono ma il dato non è valido a causa del bit di validate a **0** (condizione di **miss**: il dato non è presente in cache). Occorre ricaricare dalla memoria il dato richiesto (supponiamo che valga **0x11111111**). Il bit di *validate* corrispondente viene settato ad **1**:

V	tag	dato	
1	0	0x11111111	0
0	0	0	1
0	0	0	2
0	0	0	3
0	0	0	4
0	0	0	5
0	0	0	6
0	0	0	7

Al secondo accesso la CPU richiede il dato all'indirizzo:

$$\mathbf{4 = 0x00000004 = \underline{00000\ 00000\ 00000\ 00000\ 00000\ 00}\ \ \underline{001}\ 00_2}$$

a cui corrisponde l'indice $\mathbf{001_2}$ ed il tag $\mathbf{00000000000000000000000000_2}$.
Come nel caso precedente i tag coincidono ma il dato non è valido a causa del bit di validate a **0**. Occorre ricaricare dalla memoria il dato richiesto (supponiamo che valga 0x22222222).

V	tag	dato	
1	0	0x11111111	0
1	0	0x22222222	1
0	0	0	2
0	0	0	3
0	0	0	4
0	0	0	5
0	0	0	6
0	0	0	7

Al terzo accesso la CPU richiede il dato all'indirizzo **0**: **id=0**, **tag=0**. La cache seleziona la riga *0* e confronta i tag che coincidono. Il bit di validate è a **1**: il dato è valido (condizione di **hit**: il dato è presente in cache).

Al quarto accesso accesso la CPU richiede il dato all'indirizzo:

$$\mathbf{32 = 0x00000020 = \underline{00000\ 00000\ 00000\ 00000\ 00000\ 01}\ \ \underline{000}\ 00_2}$$

id=0, **tag=1**.

La cache seleziona la riga *0* e confronta il tag dell'indirizzo con il tag contenuto nella cache e verifica che differiscono ($\mathbf{0 \neq 1}$). Il bit di validate è a **1**. Il dato è valido ma si riferisce ad un'altra zona di memoria (**miss)**. Occorre ricaricare il dato richiesto (supponiamo che valga 0x3333).

V	tag	dato	
1	1	0x33333333	0
1	0	0x22222222	1
0	0	0	2
0	0	0	3
0	0	0	4
0	0	0	5
0	0	0	6
0	0	0	7

In totale si sono verificati 1 hit e 3 miss. Supponendo il tempo di hit pari a 2ns ed il tempo di miss pari a 60ns, otteniamo il tempo totale di accesso:

$$\mathbf{1 \times 2ns + 3 \times 62ns = 188ns}$$

4. Coerenza dei dati tra memoria e cache: write-back e write-through

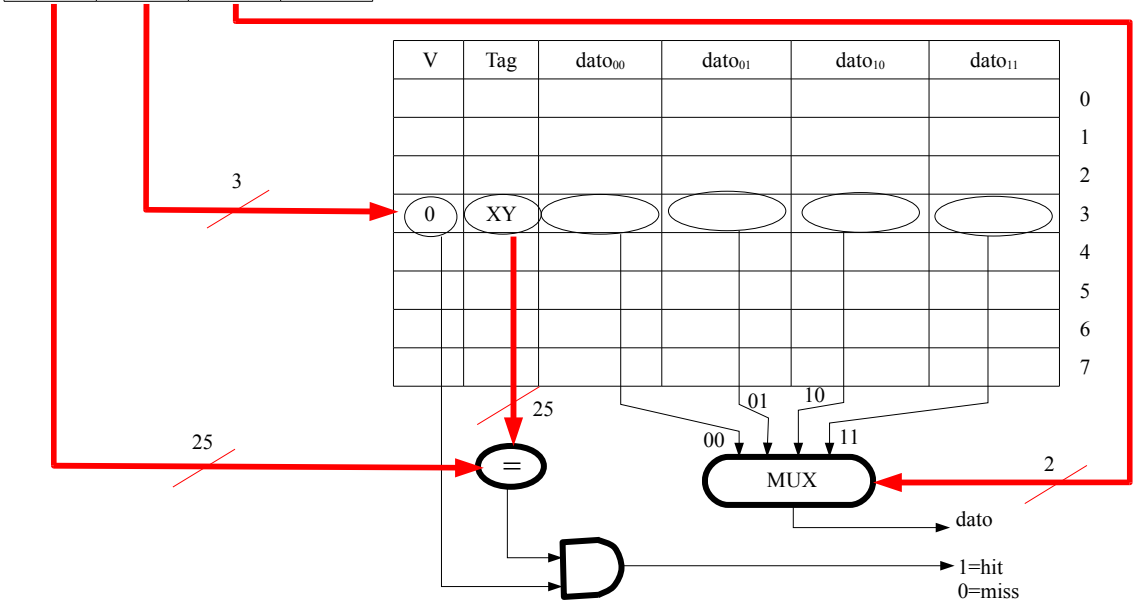
Le scritture di dati verso la memoria sollevano il problema della coerenza tra il contenuto della cache ed il contenuto della memoria stessa. Il metodo più semplice consiste nello scrivere il dato immediatamente anche nella memoria DRAM parallelamente all'aggiornamento della cache (**write-through**). In questo modo però tutte le operazioni di scrittura richiedono un tempo paragonabile ad un **miss**. Un metodo più efficiente consiste nell'aggiornare la sola cache e nel ritardare la riscrittura del dato nella memoria DRAM solo al momento in cui il dato viene sostituito nella cache da un altro più recente (**write-back**). In questo modo tutte le

operazioni di scrittura avranno tempo paragonabile ad un **hit** eccetto l'operazione di riscrittura nella memoria DRAM che peserà come una **miss.** Un ulteriore miglioramento può essere ottenuto aggiungendo un buffer che accoda le richieste di scrittura e le esegue quando la memoria non è occupata dalle operazioni di lettura (**buffer di scrittura**). Se il buffer è sufficientemente grande, allora le operazioni di scrittura avverranno nei tempi morti senza rallentare i tempi di accessi (ogni operazione di scrittura avrà un tempo paragonabile ad una **hit**).

5. Cache organizzate a blocchi

E' possibile sfruttare il principio di località spaziale usando cache il cui i dati sono indirizzabili a blocchi. Consideriamo il seguente schema di una cache di capacità pari a 32 parole organizzate in blocchi di 4 parole:

Tag	Indice	Offset	Unused
31-7	6-4	3-2	1-0
XY	011	01	00

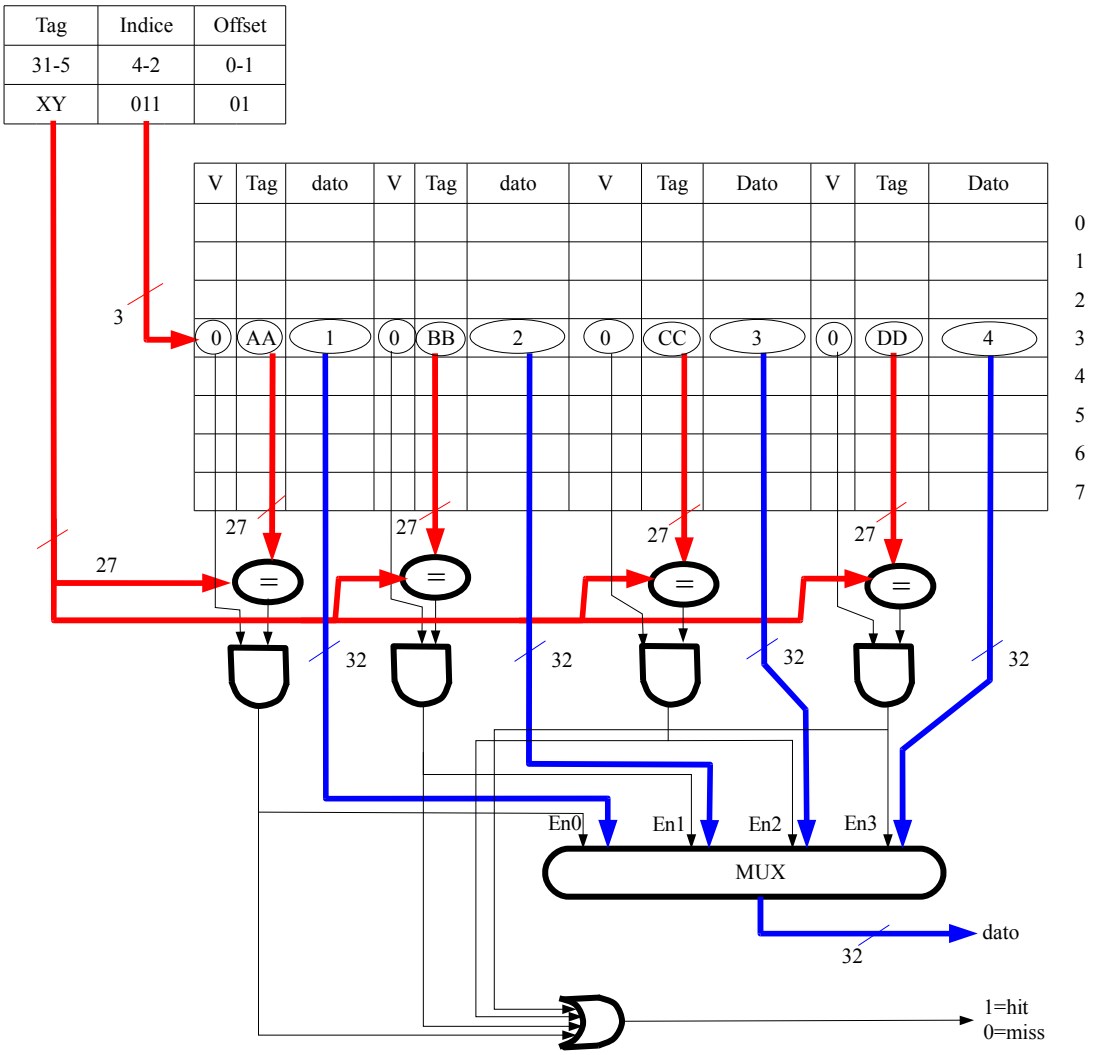


In questo esempio ogni riga indirizzabile nella cache contiene un blocco di quattro parole successive. La cache scompone l'indirizzo in tre parti: una (**indice**) usata per indirizzare la riga, una (**tag**) usata per riconoscere se il blocco selezionato è quello richiesto ed una terza (**offset**) che estrae dal blocco selezionato la parola opportuna tramite un multiplexer. Quando il tag associato ad una selezione non coincide con quello dell'indirizzo allora è necessario ricaricare l'intero blocco dalla DRAM (**miss**). Quando al contrario il tag coincide e il bit di *validate* è uguale ad 1 allora il dato selezionato dal multiplexer presente in uscita è valido (**hit**).
Una cache a corrispondenza diretta organizzata a blocchi riduce la frequenza delle **miss** per il principio di località spaziale. Ogni volta che un dato è caricato nella cache, vengono caricati anche i dati vicini del blocco di appartenenza. In pratica viene anticipato il caricamento di dati che è probabile vengano usati a breve.
Allo stesso tempo però il tempo richiesto per la riscrittura di un intero blocco sarà in generale più lungo di quello necessario per una singola parola. Ne segue che la larghezza del blocco non può essere presa ampia a piacere: ad un certo punto l'incremento delle prestazione dovuto alla diminuzione dei **miss** è annullato dal costo

del trasferimento dei dati quando si verifica un **miss.** Un miglioramento possibile consiste nell'organizzare le memorie in banchi e usare bus di trasferimento dati più ampi tra DRAM e cache. In questo modo è possibile realizzare trasferimenti paralleli di più word in un singolo tempo di accesso.

6. Cache set-associative

Se ipotizziamo di poter memorizzare più dati di tag diversi per ogni indice di cache possibile otteniamo una **cache set-associativa**. Se consideriamo l'esempio iniziale della cache ad 8 parole e supponiamo che gli accessi siano **0,32,64,96,128,..** (si pensi ad esempio a una matrice 8x8 memorizzata per righe e letta per colonne). Possiamo notare che ogni indirizzo in questo esempio corrisponde alla cella **0** della cache data, cioè ogni accesso genera una **miss**.
In una cache set-associativa ad un identico indice possono corrispondere **n** dati (**cache set-associative a n vie**), ognuno con il proprio tag ed il proprio bit di validate.

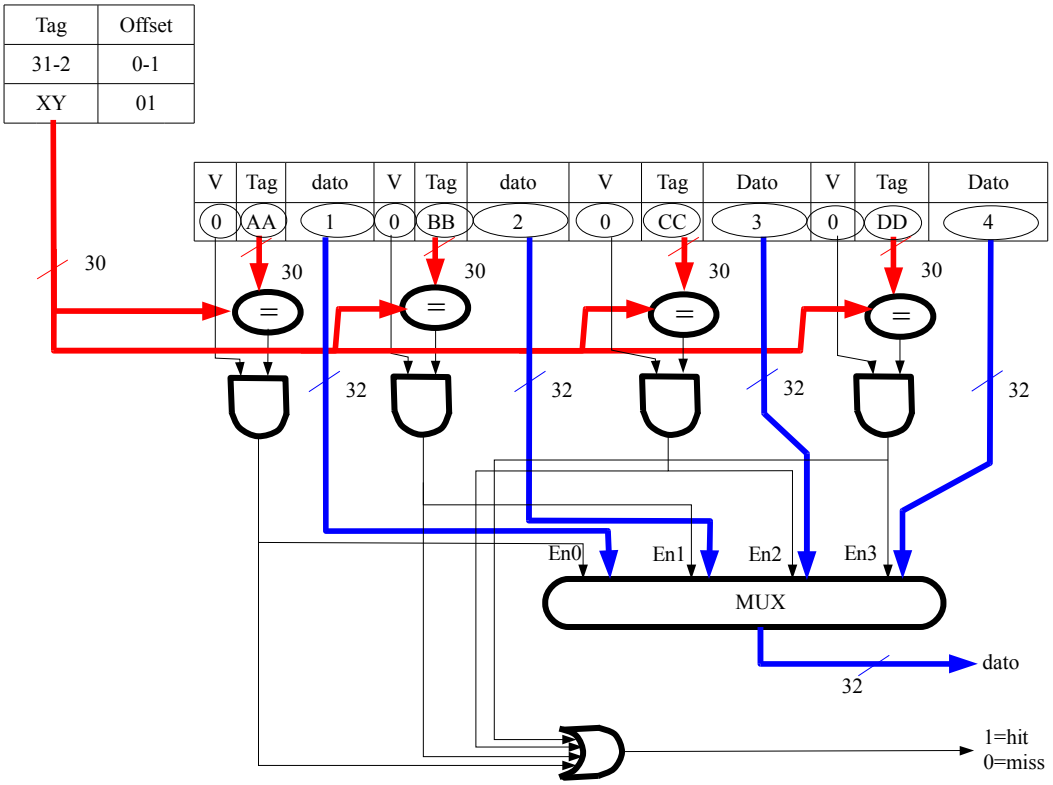


L'indirizzo di memoria viene diviso in due parti: l'**indice** di selezione che viene usato per selezionare il blocco di celle da esaminare ed il **tag** che indica quale dato si vuole leggere. Ogni tag presente nel set viene confrontato con quello di interesse. Nel caso di cache a *n* vie quindi vengono confrontati contemporaneamente *n* tag. Se il tag è presente e valido allora abbiamo un **hit**, altrimenti abbiamo un **miss** ed il dato deve

essere ricaricato dalla memoria e sistemato in una delle *n* posizioni possibili all'interno del set. Poiché ogni cella, dopo un transitorio iniziale, è occupata, occorre un algoritmo per selezionare la cella da sostituire. Un metodo efficiente ma difficile da realizzare sarebbe eliminare la cella che viene usata meno di frequente. Un metodo relativamente semplice consiste nell'eliminare la cella che non è stata richiesta da più tempo (e quindi che si suppone sia usata raramente). La gestione del tempo di accesso ad un particolare dato comunque richiede bit di memoria aggiuntiva, detti bit di **aging** che vanno eventualmente considerati, come il bit di **validate,** nel computo della dimensione effettive della cache.

7. Cache full-associative

Se i set possibili si restringono a solo uno otteniamo una **cache totalmente associativa** (full-associativa):



Tutto l'indirizzo (esclusi i due bit della word) viene usato come **tag**.
Le cache totalmente associative risultano più efficienti poiché consentono una più uniforme utilizzazione delle singole celle. Queste verranno utilizzate tutte più o meno allo stesso modo. Per contro la realizzazione fisica di un comparatore per un numero grande di celle risulta costosa e difficile.
La scrittura di una cella verso la memoria può essere organizzata come nel caso delle cache a corrispondenza diretta.

Riepilogando, una cache associativa dotata di *n* comparatori viene detta **cache associativa a n vie** (è possibile pensare le cache a corrispondenza diretta come cache a 1 via e le cache full-associative come cache a *n* vie dove *n* è il numero totale di word memorizzabili nella cache).