

Architettura degli elaboratori lezione 15


Appunti di Davide Scarlata 2024/2025

 **Prof:** Claudio Schifanella

 **Mail:** claudio.schifanella@unito.it

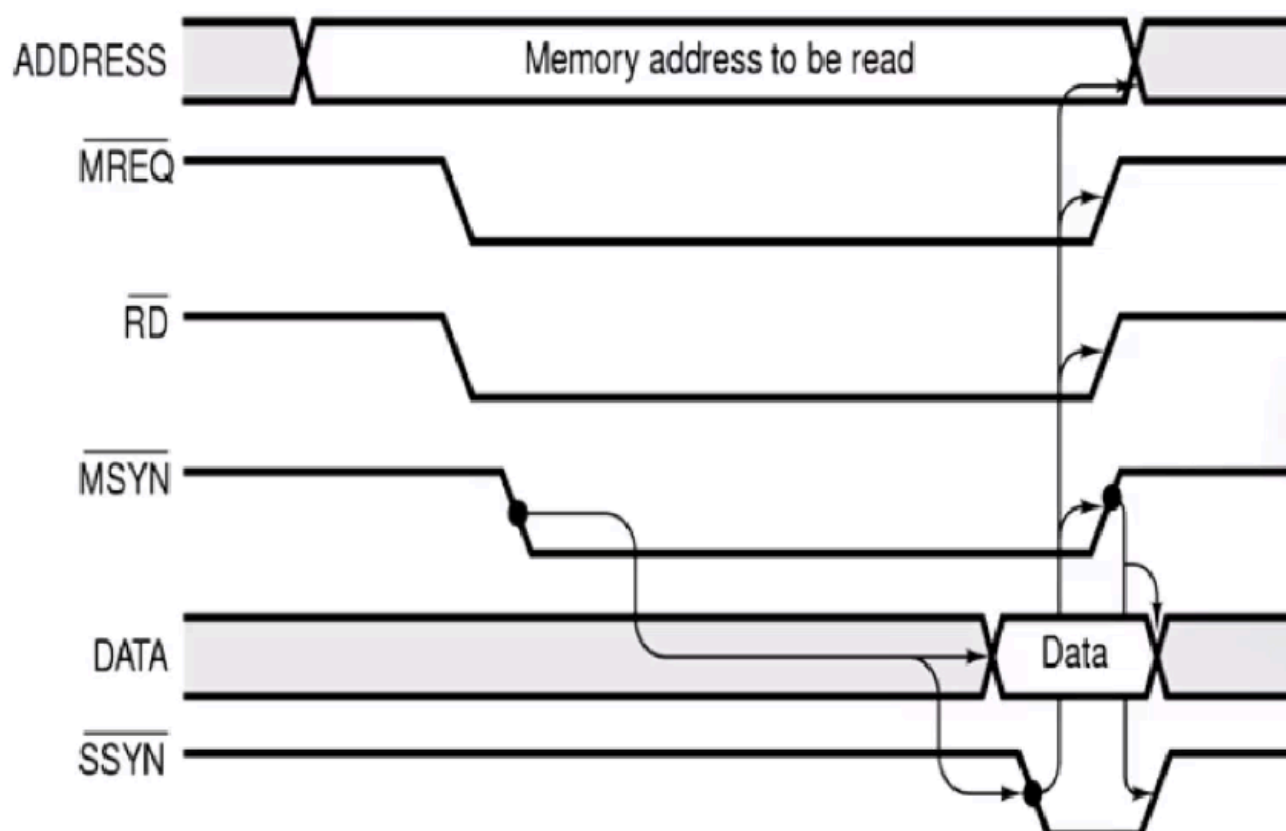
 **Corso:** C

 **Moodle** [Unito](#)

 **Data:** 13/05/2025

Bus asincrono

A differenza del bus sincrono, non abbiamo un clock che permette di scandire il tempo.



Prendiamo, di nuovo, il caso in cui la CPU (master) deve leggere qualcosa dalla memoria (slave) :

1. Il processore rende stabile il segnale nel bus degli indirizzi.
2. Il processore asserisce (mette a 0) MREQ (memory request) e RD (read).
3. Il processore asserisce MSYN (master synchronization), ovvero, il processore dice che è pronto a ricevere il dato.

4. La memoria nota i cambiamenti di MREQ, RD e MSYN (e sa che, implicitamente, il processore si impegnerà a non cambiare il valore nel bus degli indirizzi).
5. La memoria legge l'indirizzo e mette nel bus dei DATI il dato letto stabile.
6. La memoria asserisce SSYN (slave synchronization) per far capire al master (il processore) che il dato è pronto.
7. Una volta che il processore ha letto il dato, impone a 1 MSYN (e gli altri due segnali) e la memoria capisce che il dato è stato letto.
8. La memoria toglie il SSYN.

Note

Quando ADDRESS e DATA si "chiudono", non vuol dire che non diventano stabile, ma che possono variare senza influire la lettura/scrittura.

Pro e contro dei bus sincroni/asincroni

Sincrono

Pro

- Realizzazione device semplice.
- Se la durata di un'operazione è fissa, non occorre una linea di wait.

Contro

- Durata di un'operazione di comunicazione deve necessariamente avere una durata pari ad un numero intero di cicli.

Asincrono

Pro

- Flessibilità : la durata di un'operazione è determinata unicamente dalla velocità della coppia di device

Contro

- Per completare un'operazione di comunicazione sono sempre necessarie 4 azioni
- Occorre inserire nei device i circuiti necessari a rispondere opportunamente al protocollo

Arbitraggio del bus

Lo stesso bus può essere richiesto da più dispositivi contemporaneamente. Per risolvere questo problema è necessario implementare un meccanismo di arbitraggio del bus, per evitare ambiguità delle informazioni immesse sul bus stesso. Sono possibili due strade :

- arbitraggio centralizzato
- arbitraggio decentralizzato

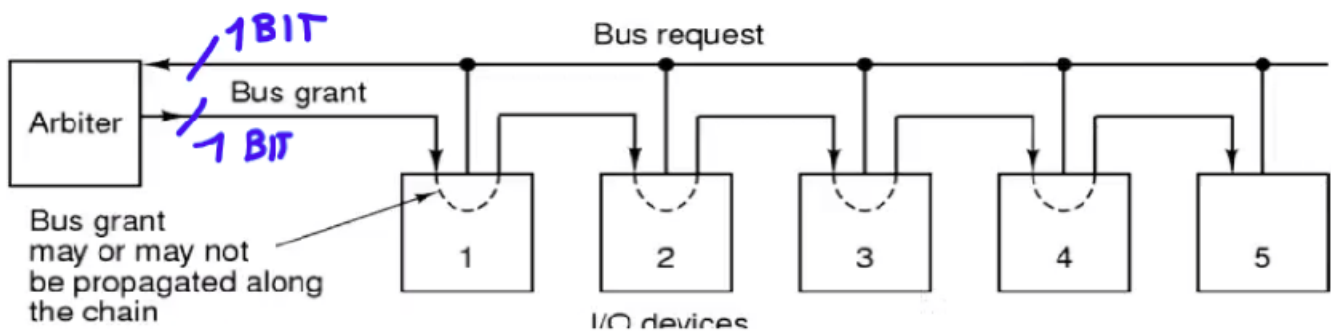
Meccanismo del grant

Meccanismo per cui ho tanti dispositivi e io assegno il grant ad uno di questi. Quel dispositivo ora sa che può utilizzare quel bus. Il grant può essere assegnato in :

- grant esplicito, qualcuno assegna esplicitamente il grant.
- grant implicito, in base al contesto, un dispositivo si assegna autonomamente il grant.

Arbitraggio centralizzato

In questo modello, quando l'arbitro "nota" una richiesta, attiva la linea di grant del bus.



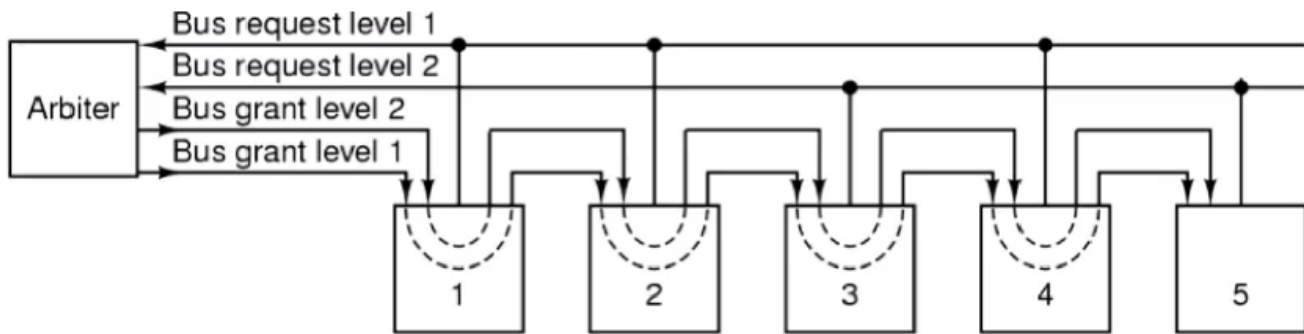
Il segnale di grant viene inviato seguendo la tecnica del "Daisy chaining". Ovvero, grant viene passato da ogni dispositivo finché un dispositivo non accetta l'assegnamento. Anche se il dispositivo 2 dovesse chiedere il grant, il dispositivo 1 potrebbe prenderlo quando passa da lui. Si dice che il dispositivo più vicino "vince".

Starvation

Questo modello presenta un grosso problema, ovvero che i dispositivi alla fine, più lontani, potrebbero non arrivare mai a ricevere il grant. Per risolvere questo problema si possono integrare più livelli di priorità.

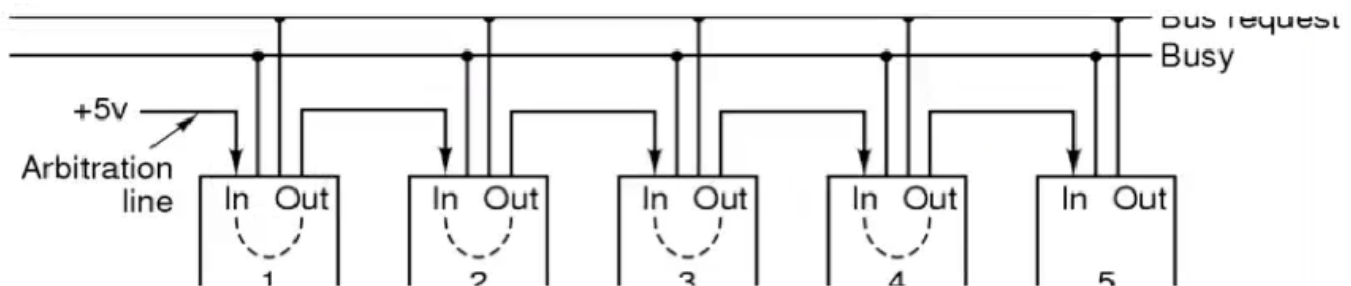
Priorità

Si possono integrare due, o più, livelli di priorità. In questo caso l'assegnamento segue lo stesso meccanismo del daisy chaining, ma i dispositivi priorità più alta hanno precedenza nell'assegnamento del grant da parte dell'arbitro.



Arbitraggio decentralizzato

In questo modello non c'è un arbitro che gestisce il bus, ma si permette ai dispositivi stessi di gestirlo.



Il dispositivo che vuole usare il bus ferma la propagazione di "arbitration line", abilita il segnale di "busy" (il quale notifica a tutti i dispositivi che il bus è occupato) e poi il bus viene usato. Una volta finito l'uso del bus, si toglie il segnale di "busy" e si lascia passare il segnale di "arbitration line". Questo sistema è più economico e veloce rispetto a "daisy chaining" centralizzato, però è poco flessibile (ma si risparmia il costo dell'arbitro).

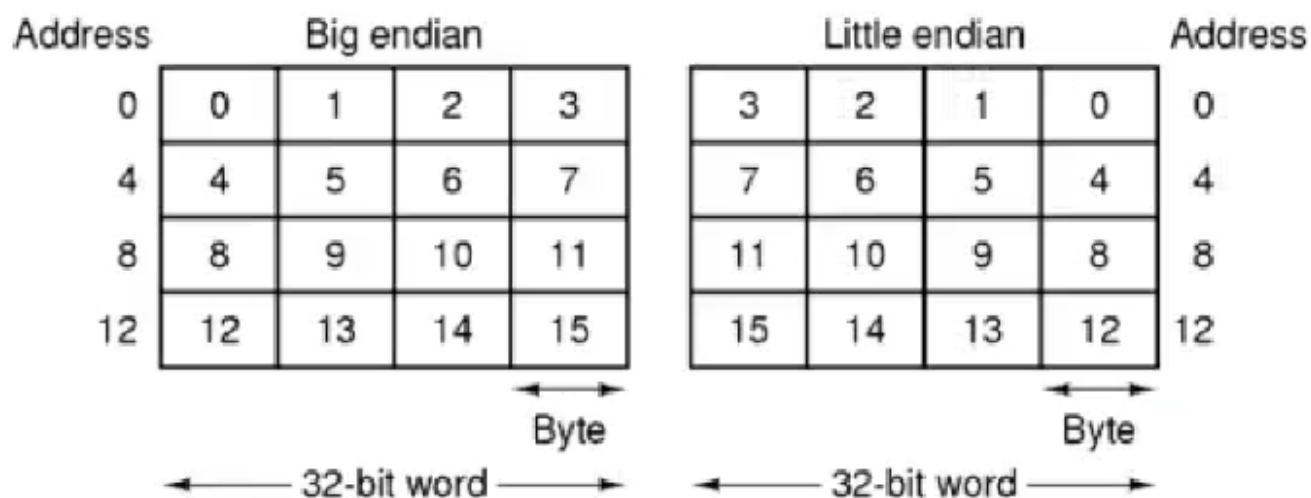
Memoria

Alcuni fatti per ripassare la memoria :

- La memoria è un insieme di celle aventi tutte la stessa dimensione
- Ogni cella di memoria è identificata tramite un indirizzo
- Se un calcolatore ha n celle, allora gli indirizzi vanno da 0 a $n - 1$, indipendentemente dalla dimensione della cella
- Il numero di bit nell'indirizzo determina il numero massimo di celle indirizzabili
- La cella è la minima unità indirizzabile (possiamo caricare gli 8 bit di una cella e poi vederne, ad esempio, 4, ma ne dobbiamo sempre caricare 8)
- In RISC-V una cella di memoria è grande 8 bit e ci sono 2^{32} celle.

Ordinamento dei byte

Abbiamo due modi di ordinare i byte :



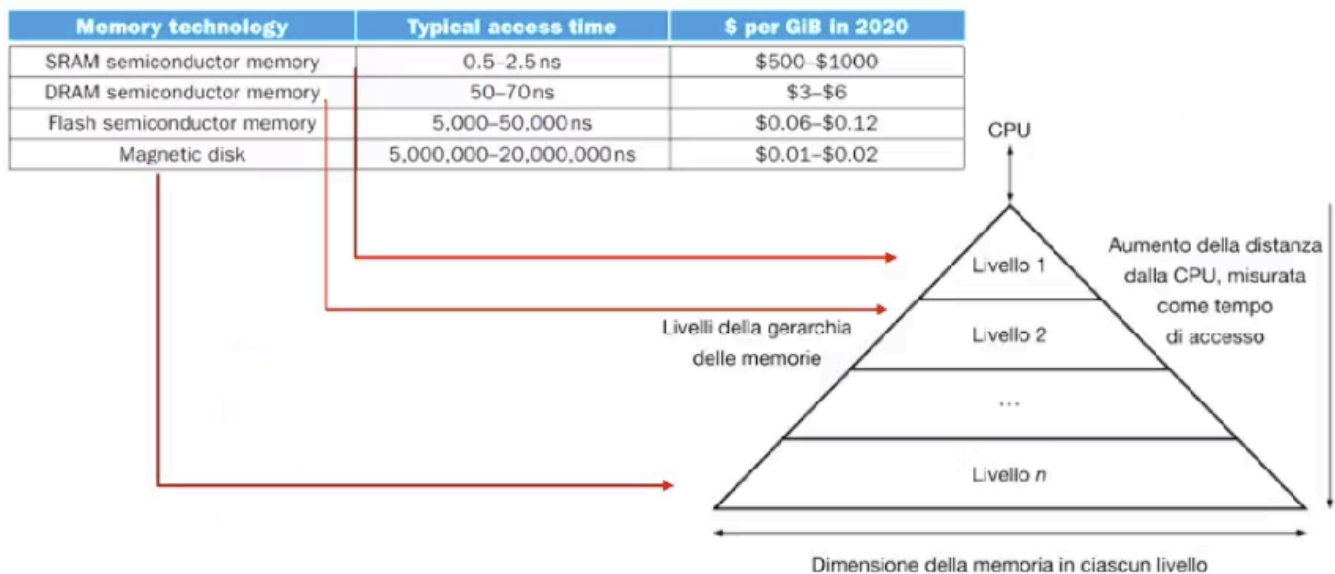
- Big endian, la numerazione va da sinistra verso destra. Big end first (SPARC, mainframe IBM).
- Little endian, la numerazione va da destra verso sinistra. Little end first (Intel, RISC-V).

CPU e memoria

Le CPU sono più veloci delle memoria, questo può provocare del bottleneck (un rallentamento dell'interno processo per colpa di un componente). Infatti, quando la CPU effettua una richiesta alla memoria, essa non ottiene la parola desiderata se non dopo molti cicli di CPU. Come si può risolvere questo problema? Cosa dovremmo fare, avere una memoria più piccola e molto più veloce (e molto più costosa) o una grande quantità di memoria, ma non abbastanza veloce (e relativamente economica)? Beh, si può fare una via di mezzo, ovvero, si può implementare una piccola quantità di **memoria cache** (attualmente lo standard sono pochi KB/MB in base al livello) e una media/grande quantità di memoria RAM.

Gerarchia della memoria

Ecco uno schema della gerarchia delle memoria secondo la capacità di memorizzazione :



Come possiamo vedere c'è un collegamento diretto tra il costo delle memorie e la velocità di accesso. Per quanto un disco magnetico costi davvero poco, è impensabile di usarla memoria principale dinamica.

Memoria cache

La memoria cache è una memoria che si interpone tra la CPU e la memoria principale (RAM). L'uso della memoria cache è basata sul principio di località del codice (i programmi non accedono alla memoria a caso).

Se la CPU vuole leggere una word dalla memoria che non è salvata nella cache (nella cache possono essere salvate sia istruzioni che dati), questa word viene letta dalla memoria assieme a molte altre word (viene letto un blocco di memoria) e vengono salvate nella cache. Poi la cache restituisce la word richiesta alla CPU. Questa cosa viene fatta per i due seguenti fattori :

- Località temporale

Principio secondo il quale se si accede a una determinata locazione di memoria, è molto probabile allora che vi si acceda di nuovo dopo poco tempo. Un esempio banale al quale pensare è un ciclo (sia for che while), quando finisce un'iterazione del ciclo, si ritorna indietro alle stesse istruzioni.

- Località spaziale

Principio secondo il quale se si accede a una determinata locazione di memoria, è molto probabile che si acceda alle locazioni vicine a essa dopo poco tempo. Basta pensare ad un array contiguo. Se accedo alla posizione i , è facile che io poi voglia avere $i+1$. Invece, per quanto riguarda le istruzioni, basta pensare a qualsiasi frammento di codice (dove non ci sono

dei salti), noi vogliamo fare prima un'istruzione, poi un'altra e poi un'altra ancora tutte vicine tra di loro.

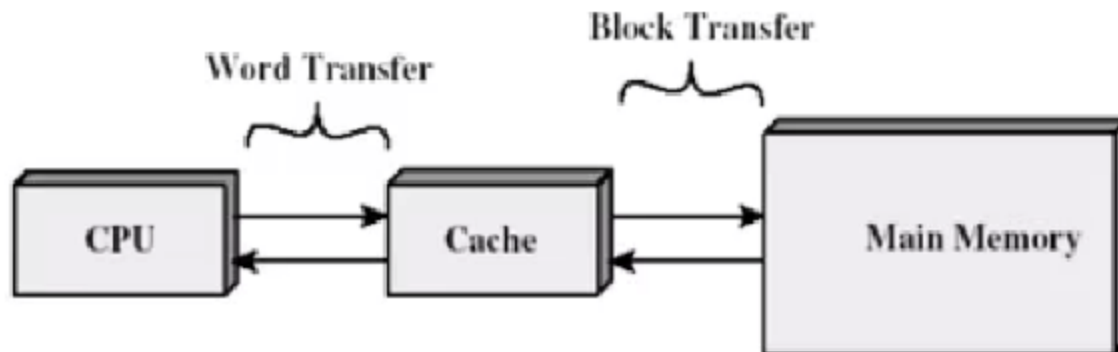
Blocchi di informazioni

La memoria principale consiste di 2^n byte indirizzabili. Logicamente possiamo pensare che questa memoria è divisa in blocchi di k byte ($M = 2^n/k$ blocchi), La memoria cache consiste in una quantità C di linee di k byte (blocchi).

Note

C (il numero di blocchi in cui è divisa la cache) è sempre minore a M (il numero di blocchi un cui è divisa la RAM). Questo è ovvio, perché altrimenti vorrebbe dire avere una cache che è grande quanto (o più) la RAM.

Questa divisione in blocchi è usata per adempiere al principio di località. Come vediamo dall'immagine sotto, tra la RAM e la cache si passano interi blocchi e poi dalla cache alla CPU si passano semplicemente le word richieste :



Un blocco viene caricato sulla cache quando la CPU richiede una word e la cache si accorge di non avere quella word, allora, viene caricato l'intero blocco dove è contenuta quella word (un blocco ovviamente contiene più di una word, ne contiene diverse).

Miss e hit della cache

Ci sono due possibilità per le word richiesta dalla CPU :

- hit : la word richiesta si trova nella cache, quindi il tempo è ottimo.
- miss : la word richiesta non si trova nella cache, pessimo per il tempo.

Si deve cercare di alzare al massimo le hit e abbassare le miss.

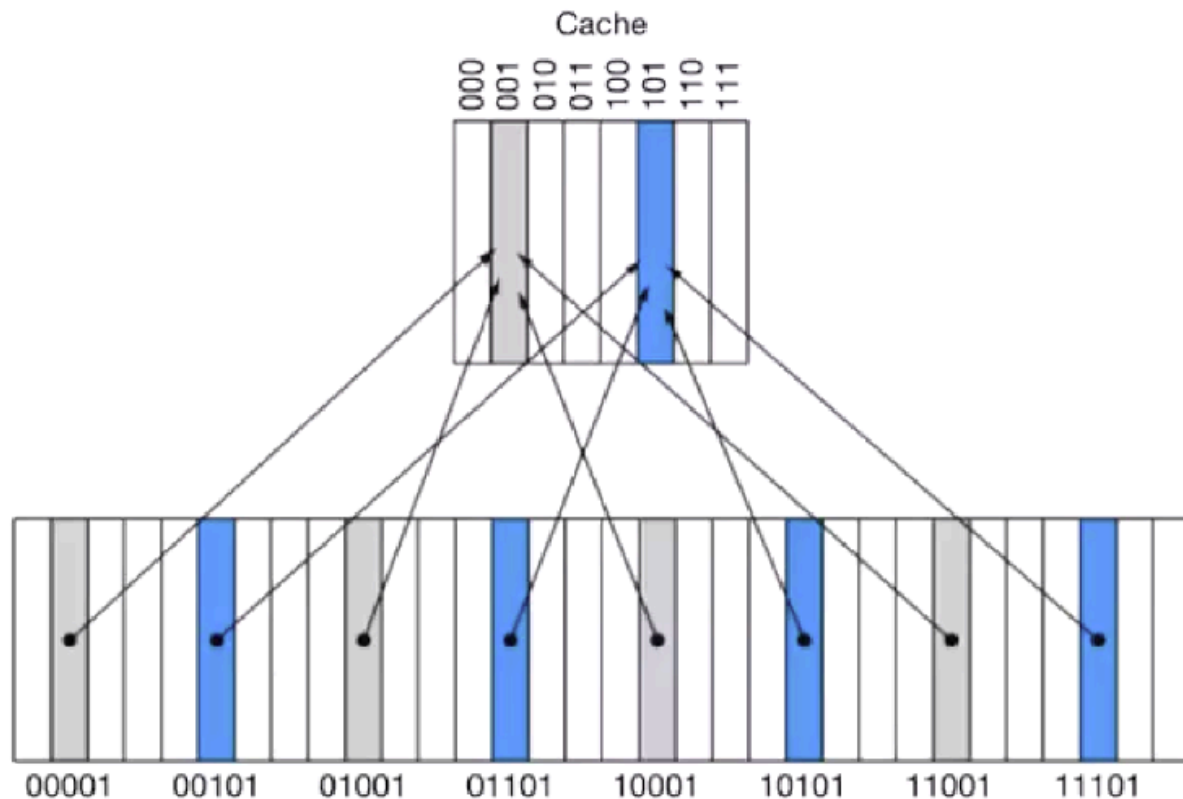
Problemi relativi alla progettazione della cache

Uno dei problemi da considerare è la dimensione della linea di cache, avere tante linee di pochi byte o poche linee di molti byte? Avere tanti blocchi, ma troppo piccoli fa generare tante miss. In modo contrario avere pochi blocchi troppo grandi, ci saranno tante sovrascritture dei blocchi.

Per questo ci sono diversi modi di mappare la cache :

Mappatura diretta (studiare)

Una cache in cui ad ogni locazione della memoria principale (ram) corrisponde una (e una sola) locazione della cache.



Si calcola con \$\$

$\text{indirizzo del blocco} \bmod (\text{numero di blocchi nella cache})$

\$\$