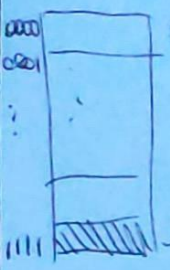
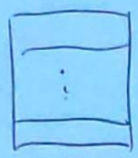
DIRECT MAPPING:

È il metodo più semplice: ogni blocco in memoria è associato 1-1 con una linea in cache.

Facciamo un es:





Blocco RAM = Blocco cache = 8 byte (IN QUESTO CASO)

$NB^{RAM}_{cache} = \frac{128}{32} = 16 \text{ block} = 2^4 \text{ blocchi}$

(numero blocchi in cache)

4 bit per INDEX in RAM.

$N_{LINEE} = \frac{32}{8} = 4$ linee 2 bit per cache index

I 2 bit meno significativi dell'indice del blocco (in questo caso 2) indicano l'indice della linea di cache, ad esempio:

0000 0
0100 4
1000 8
1100 12

} saranno mappati alle linee 0 della cache.

L'ALLINEAMENTO NON TUTTI ASSIEME, MA IN BASE AD UNA LOGICA

La CPU avrà un meccanismo per mettere i blocchi nelle linee.

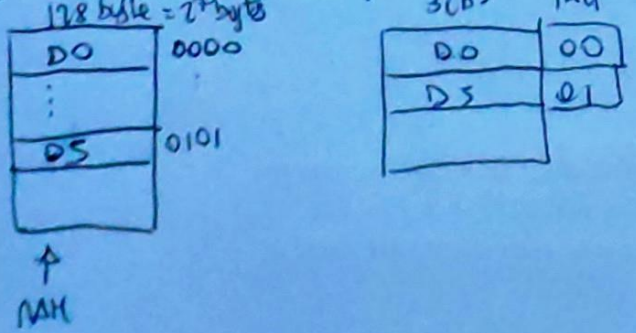
Per capire, nel caso nostro (quindi 00 - linea 1) a quale blocco corrisponde il dato che c'è nella linea, la CPU usa i bit di TAG.

TAG BITS = Bindex - Cache index = 4 - 2 = 2 bit. I tag tengono i bit più significativi (BLOCK INDEX SIZE − CACHE INDEX SIZE).

del blocco. Pertanto, se sto memorizzando il blocco 12 (1100), nell'index avrò 00 e nel tag 11.

Come fa la CPU a decodificare l'indirizzo?

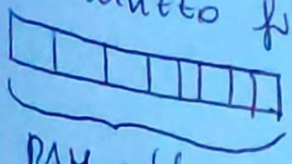
Prendiamo, ad esempio



→ 7 bit per indicare un blocco
5 bit per indicare una linea.
SIZE Blocco = 8 byte = 2^3 offset.

DIRECT MAPPING 2:

L'indirizzo fisico → Blocco è composto da 7 bit.



Poiché ogni blocco è grande 1 byte (2^3 bits), avrò bisogno di 3 bit per calcolare l'offset. L'offset mi serve per determinare il byte che devo accedere nelle linee della cache!

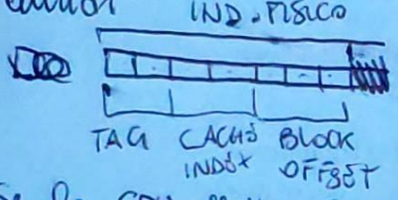
Pertanto, 3 bit sono di offset 7-3=4 bit sono per indicare il block index. Inoltre, poiché sappiamo che un indirizzo della cache è a 5 bit, i 5 bit meno significativi dell'indirizzo fisico identificheranno le linee della cache.

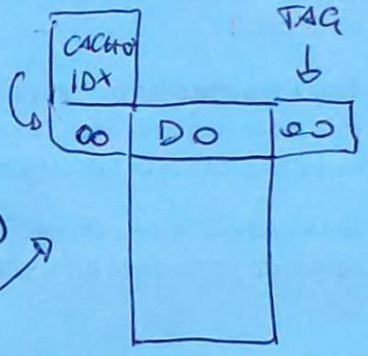
All'interno di questi 5 bit, 3 identificano il tag offset, i restanti 2 identificano l'index della cache. → avero il tag.

T_{BITS} = BIT IND. FISICO - BIT CACHE ADDRESS = 7 - 5 = 2

! = B_{INDEX} - $CACHE_{INDEX}$ = 4 - 2 = 2

Quindi





Se la CPU effettua il seguente codice di richiesta:

0000010 → richiede un dato alla linea 00 (CI) con tag 00 (TB)

TB CI BO

Pertanto la CPU guarda nella cache e trova alla linea 00 il tag corrispondente! 00 - 00

Usando il block offset, la CPU prenderà il dato al byte n° 2! (0000010)

Abbiamo un hit!

(N.B: ogni blocco/linea contiene 8 byte).

Se, ad esempio, la CPU emette:

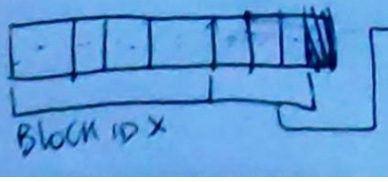
0100011 → Non abbiamo nessuna corrispondenza! MISS.

TAG CI OFFSET

In questo caso la CPU dovrà cercare nella RAM il dato nuovamente.

Per identificare il blocco nella RAM si usa l'indirizzo fisico.

Nell'ind. fisico



→ BLOCK OFFSET = 3 bit
BLOCK IDX = 5 bit

Pertanto il byte 011 (3) del blocco 0100 (4) deve essere messo nella linea 0 della cache, con tag 1 al byte 3 (011).

FORMULE D-MAPPING:

RAM SIZE = $M_{SIZE} = 2^{\text{bit ind. fisico}}$

CACHE SIZE = $C_{SIZE} = 2^{\text{bit ind cache}}$

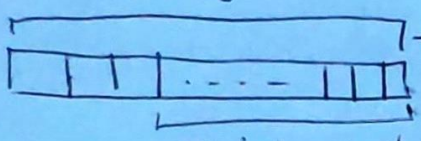
BLOCK SIZE = $B_{SIZE} = 2^{\text{offset}}$

N° BLOCKS = $\frac{M_S}{B_S} = 2^{\text{Bindex}}$ N°LINEE CACHE = $\frac{C_S}{B_S} = 2^{C_i}$

es: Una cache da 32kB con linee da 32 byte.
La CPU genera indirizzi da 32 bit.
Quanti sono i bit dell'indice della cache?
i bit di tag?

$C_S = 32kB = 2^{10} \cdot 2^5 = 2^{15}$ BYTE → 15 bit per l'ind. cache

$B_S = 32 \text{ bytes} = 2^5 B \Rightarrow B_{offset} = 5$ bit.



15 bit = indirizzo di cache
5 block offset.
15 - 5 = 10 cache index
tag bits = 32 - 15 = 17 bits

$T_B = 17$

$C_I = 10$

I tag sono memorizzati nella TAG DIRECTORY!

$T_{DS} = T_B \cdot L$ (linee). (T_B: n° bit di tag)

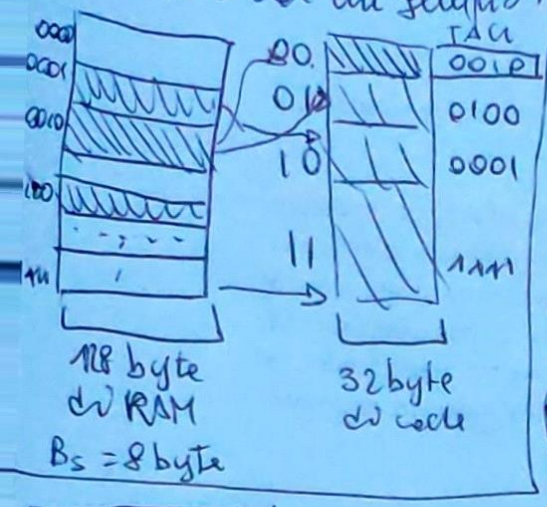
Spesso nella TD sono memorizzati bit di uso:
V: VALID (0 se non c'è dato, 1 se c'è).
D: DIRTY (1 se il dato è stato modificato)
} METABITS

Se ci sono METABITS, la dimensione della TDS è $(T_B + N_{METABITS}) \cdot L$.

ASSOCIATIVE MAPPING:

Nell'associative mapping, un qualsiasi blocco della RAM può essere posizionato in una qualsiasi linea.

Tornando ad un esempio simile al precedente:



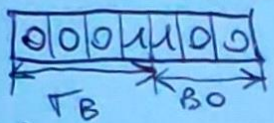
Notiamo come i bit di tag sono tanti quanti i bit di blocco.

$T_B = B_I$, in questo caso $T_B = B_I = 4$

La CPU fa dunque una richiesta per l'indirizzo:

0001100

Poiché ogni blocco è 8 byte (2^3) servono 3 bit per il block offset.



Poiché le linee non sono prefissate come nel D-MAPPING, la CPU dovrà scorrere tutte le linee della cache e confrontare gli indici. In questo caso il blocco richiesto è nella cache, in base al block offset (100 = 4) prende il dato e lo serve.

Se la CPU emette la seguente richiesta:

0011,001 → notiamo che questo tag non è nella cache, pertanto abbiamo un MISS.

Pertanto la CPU prende il blocco corrispondente all'indice 0100 (4) e cerca di caricarlo nella cache. La cache è PIENA, pertanto la CPU deve rimpiazzare una linea della cache con quella che ha fetchato ora. Qui entrano in gioco vari algoritmi di rimpiazzamento come LRU, MRU o FIFO. Statisticamente si è osservato che migliore è l'associatività della cache e più la policy RANDOM ha risultati molto simili a LRU, pertanto spesso si opta per questo.

FORMULE ASS-MAPPING:

$M_{SIZE} = 2^{\text{BIT INDIRIZZO FISICO}}$

$C_{SIZE} = 2^{\text{CACHE ADD. BITS}}$.

$B_{SIZE} = 2^{\text{BOFFSET}}$

$N_{BLOCKS} = \frac{M_{SIZE}}{B_{SIZE}} = 2^{\text{BINDEX}}$

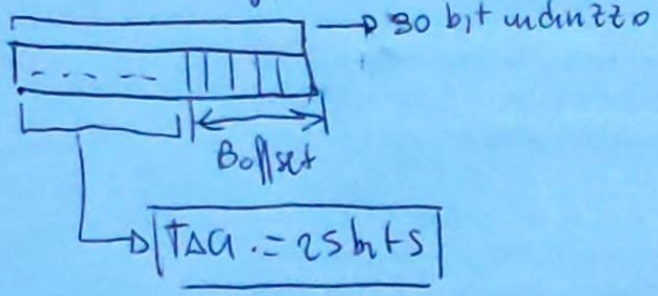
$N_{LINES} = \frac{C_{SIZE}}{B_{SIZE}} = 2^{\text{CINDEX}}$.

IN ASS-MAPPING l'ind. fisico è spezzato in BLOCK OFFSET (meno significativo) e TAG BITS (più significativi).

es: ho una ASS-CACHE DI 8KB con ~~blocchi~~ linee da 32byte.
la RAM è di 1GB. Quanti sono i BIT DI TAG?

$C_{SIZE} = 2^{10} \cdot 2^3 = 2^{13}$ BYTE $M_{SIZE} = 2^{30}$ byte → 1-byte allineate.
└ INDIRIZZO = 30 bit

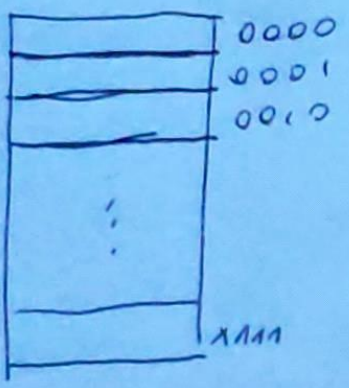
$B_{SIZE} = 2^5$ byte
└ Boffset = 5 bits



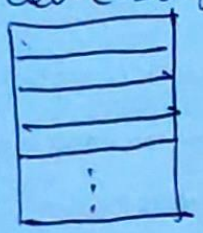
SET-ASSOCIATIVE (N-WAY).
└ Risolve l'overhead introdotto dalle FULLY ASSOCIATIVE dato la RICERCA PER COMPARAZIONE.

Torniamo sempre al nostro esempio:

RAM = 128 byte

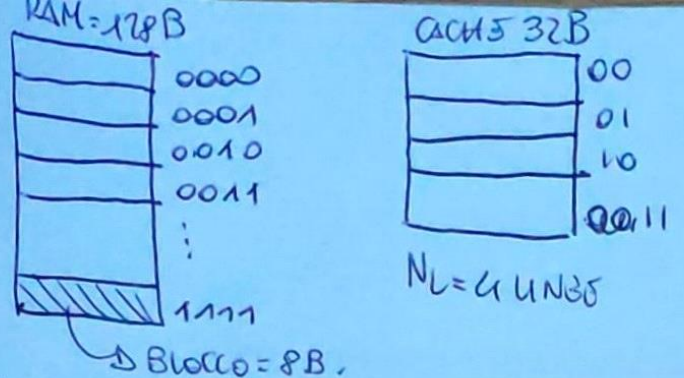


Cache = 32byte



Del nome (N-WAY) si capisce che la cache sarà splittata in N set (2, 4, 8...).
I blocchi di memoria RAM sono collegati ai set utilizzando la logica del DIRECT MAPPING, dentro ai set il collegamento è fatto come nelle fully assoc. Il vantaggio di queste cache è dato dal fatto che la ricerca di una linea all'interno del set è molto più ridotta rispetto a quella fully-assoc.
Continuiamo con l'esempio:

└ PLOT RAG.



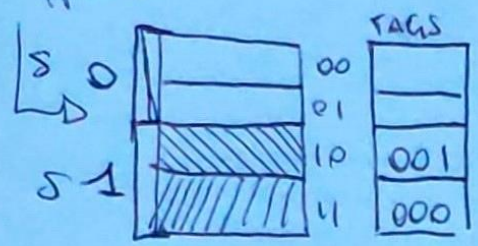
N_L = 4 LINEE

N_B = 16 BLOCCHI

2 - WAY ASSOCIATIVE

↳ la cache è partizionata in $N_S = \frac{N_L}{n} = \frac{4}{2} = 2$ SETS.

Pertanto ci basta 1 bit per indirizzare i SET.



↳ lo 0-esimo set contiene tutti i blocchi di RAM con bit MENO SIGNIFICATIVO uguale a 0. Stesso discorso per l'1-esimo set.

Nell'esempio sopra ↑ supponiamo che il 4° blocco della RAM (0011) è caricato in cache (nel SET 1). Il blocco può essere caricato in qualsiasi linea; scegliamo la 1ª. I restanti bit più significativi (001) indicano IL TAG.

Oltre al 4° blocco, anche il 2° viene caricato in cache (0001), ovviamente nel 0 set 1-esimo.

Supponendo la richiesta della CPU: 000|1|10

↳ block offset

↳ set index

tag

Pertanto la CPU cercherà:

- Nel SET 1
- IL TAG 000 (lo trova per comparazione)
- E PRENDERÀ IL 3° BYTE DI TALE BLOCCO (010) → 2+1 perché i byte vengono contati da 0.

↳ HIT!

Notiamo come ci i tag bit che il numero di comparazioni si riduce.

N.B.:

$N_{LINEE} = \frac{C_{SIZE}}{B_{SIZE}} = 2^{CI}$ → CACHE-INDEX BITS

$N_{SET} = \frac{N_{LINEE}}{n} = \frac{2^{CI}}{n}$

$N_{SET} = \frac{2^{CI}}{2^{\log_2 n}} = 2^{CI - \log_2 n} = 2^{SI}$ → S.index bits

↳ $\boxed{SI = CI - \log_2 n}$

FORMULE N-ASSOC:

$M_{SIZE} = 2^{n^\circ \text{ bit indirizzo fisico}}$

$C_{SIZE} = 2^{n^\circ \text{ bit cache}}$

$B_{SIZE} = 2^{\text{block offset}}$

$N_{BLOCKS} = \frac{M_{SIZE}}{B_{SIZE}} = 2^{BLOCK\ INDEX}$

$N_{LINES} = \frac{C_S}{B_S} = 2^{CI}$

$N_{SET} = \frac{N_{LINES}}{n} = 2^{SI}$

$SI = CI - \log_2 k$