LABORATORIO DI ARCHITETTURA DEI CALCOLATORI

lezione n° 21

Prof. Rosario Cerbone

rosario.cerbone@libero.it

http://digilander.libero.it/rosario.cerbone

a.a. 2005-2006

Cache completamente associativa

- Il principale problema pratico da risolvere nella realizzazione di un dispositivo Cache é trovare un buon modo per decidere in brevissimo tempo se esiste o meno nella memoria cache una copia del dato cui la CPU chiede di accedere.
- La richiesta di accesso avviene sotto forma di un ciclo di lettura o scrittura su un bus che mette in comunicazione la CPU con il controllore di cache, ed il dato coinvolto viene identificato mediante l'indirizzo di una cella di RAM.

Cache completamente associativa

- Occorre quindi che il controllore di cache mantenga un elenco di tutti gli indirizzi (o meglio di tutti i numeri di linea di cache) effettivamente presenti in quel momento in Cache e che sia in grado di determinare "in un solo passo" se l'indirizzo desiderato appartiene o no ad una linea contenuta in Cache, anche se l'elenco é relativamente lungo.

Memorie Associative

- In alternativa alla organizzazione di tipo RAM della memoria, dove le celle vengono individuate mediante il loro numero d'ordine (indirizzo), in molte circostanze risulterebbe utile poter disporre di un dispositivo in grado di individuare una cella sulla base di parte del suo contenuto.

Memorie Associative

- Un tale dispositivo che consente di realizzare in modo naturale una operazione di ricerca del contenuto viene detto **memoria associativa**. Per fissare le idee possiamo prendere come riferimento il problema dell'anagrafe tributaria, dove ciascuno di noi é identificato dal proprio codice fiscale, e dove uno può essere interessato a vedere tutte le informazioni relative ad una singola persona cercando tutte le informazioni che fanno riferimento in qualche modo a quel codice fiscale.

Memorie Associative

- Il primo passo verso la realizzazione di una memoria di tipo associativo consiste quindi nel suddividere le informazioni contenute in una singola cella di memoria in due sottoinsiemi di bit: un primo sottoinsieme (normalmente chiamato "tag") che sarà soggetto alle ricerche di tipo associativo, ed un secondo sottoinsieme (normalmente detto "value") che codifica i valori ignoti che si stanno cercando.

Memorie Associative

- Le informazioni sono inserite in ordine qualsiasi all'interno di una memoria associativa, per cui l'unico modo per individuare la cella contenente il valore desiderato sarà quello di confrontare il contenuto del campo di ricerca (tag) di ciascuna cella col valore che interessa. La ricerca associativa vera e propria verrà quindi realizzata mediante un circuito comparatore di uguaglianza associato al campo tag di ciascuna cella.

Cache completamente associativa

- Si arriva quindi a progettare una memoria cache "completamente associativa“. Il "numero di linea di cache" (ossia il numero ottenuto interpretando la codifica binaria dei bit più significativi dell'indirizzo proveniente dalla CPU, dopo aver tolto i bit meno significativi corrispondenti alla potenza di 2 che costituisce la dimensione prescelta delle linee di cache) viene usato come "campo di ricerca" (in questo contesto identificato come campo **tag**). Il dato associato al campo di ricerca é l'insieme delle copie del contenuto delle parole di memoria RAM individuate come appartenenti a quella linea di cache.

Cache completamente associativa

- Oltre ai dati veri e propri, volendo realizzare un algoritmo di inserzione dei dati mancanti con rimpiazzamento di tipo LRU, sarà necessario associare ad ogni linea anche almeno un bit che indichi il fatto che quella linea é "vuota" oppure "occupata" per contenere una copia dei dati originali in RAM, ed un insieme di bit destinati a memorizzare una indicazione di quali dati sono stati usati recentemente e quali no.

Cache completamente associativa

- Non andiamo oltre nella descrizione di una possibile realizzazione completamente associativa, in quanto questo metodo non viene quasi mai utilizzato per le memorie cache. La principale controindicazione che ne limita la diffusione é il costo molto elevato per bit "utile" memorizzato. Normalmente si adottano soluzioni tecnicamente meno vicine al principio di funzionamento "teorico" di una Cache, ma più convenienti ed accettabili nella realizzazione di un sistema dal punto di vista economico.

Cache a corrispondenza diretta

- Una organizzazione a corrispondenza diretta (**direct mapping**) segue un approccio diametralmente opposto rispetto a quello di una organizzazione completamente associativa. Anziché studiare i vari dispositivi e vedere quale tra questi é in grado di risolvere meglio il problema, si parte invece dalla scelta di un dispositivo molto semplice ed economico da realizzare, e poi si studia un modo di utilizzarlo che approssimi nel miglior modo possibile il comportamento che si vuole ottenere.

Cache a corrispondenza diretta

- Per ridurre i costi di realizzazione, una Cache a corrispondenza diretta viene realizzata a partire da una memoria con indirizzamento di tipo RAM. Ovviamente si farà ricorso ad una tecnologia di realizzazione statica (anziché dinamica come nel caso della "RAM di sistema"), per mantenere basso il tempo di accesso.

Cache a corrispondenza diretta

- É evidente che se l'elenco delle linee presenti in Cache é memorizzato in un dispositivo con indirizzamento RAM non sarà possibile confrontare l'indirizzo proveniente dalla CPU con tutti gli elementi dell'elenco prima di poter rispondere: tale soluzione richiederebbe, nel peggiore dei casi, un numero di accessi sequenziali pari alla dimensione dell'elenco, quindi un tempo proporzionale al numero di linee di cache, vanificando completamente l'idea di poter reperire velocemente le informazioni da una Cache con un numero sufficientemente alto di linee.

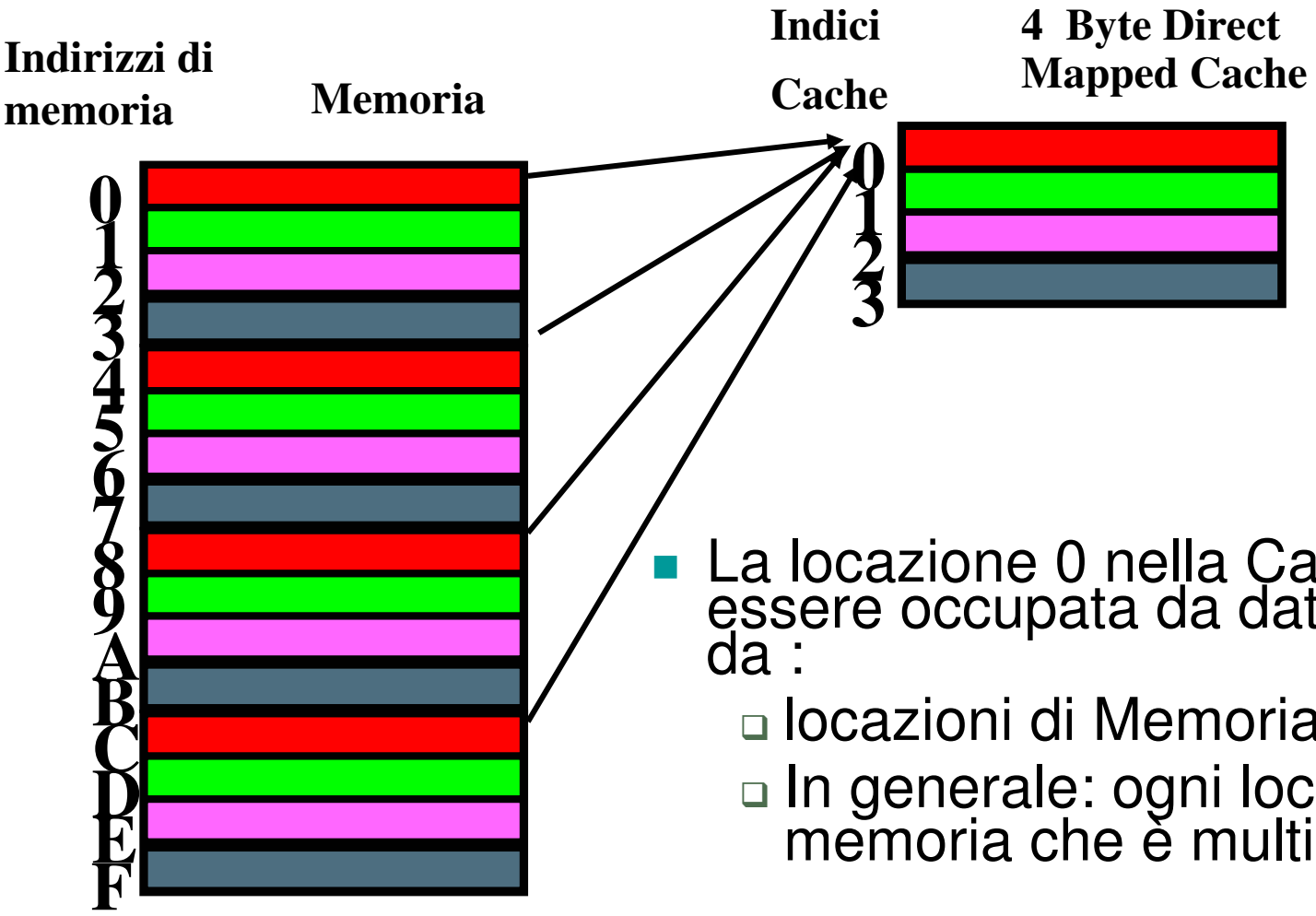
Cache a corrispondenza diretta

- La soluzione adottata consiste quindi nel ridurre ad un solo confronto di valori la determinazione della presenza o assenza della linea contenente l'indirizzo indicato dalla CPU. Ciò viene ottenuto facendo in modo che ogni numero di linea in realtà possa essere memorizzato in una sola posizione predefinita all'interno dell'elenco. Sapendo dove si potrebbe trovare nell'elenco, possiamo completare la verifica a colpo sicuro, in un singolo passo: se il numero trovato leggendo il valore in quella posizione predeterminata dell'elenco corrisponde, bene, altrimenti possiamo subito concludere che la linea non é contenuta nella Cache, senza dover procedere con altri confronti.

Cache a corrispondenza diretta

- Operativamente si ottiene questa associazione univoca tra numero di linea e posizione nell'elenco, facendo corrispondere parte della rappresentazione binaria del numero di linea di cache proprio con la posizione all'interno dell'elenco.

Cache a corrispondenza diretta



- La locazione 0 nella Cache può essere occupata da dati che arrivano da :
  - locazioni di Memoria 0, 4, 8, ...
  - In generale: ogni locazione di memoria che è multiplo di 4

Cache a corrispondenza diretta

Dividiamo gli indirizzi di memoria in tre campi:

tag	Indice	offset
Controlla se c'è il blocco corretto	**Indice all'interno della cache**	**Byte all'interno del blocco**

Terminologia della Direct-Mapped Cache

- Tutti i campi vanno intesi come numeri senza segno (non in complemento a 2).
- Indice: specifica l'indice nella cache (in quale riga dobbiamo guardare)
- Offset: una volta trovato il blocco corretto, l'offset specifica il byte all'interno del blocco che vogliamo
- Tag: gli altri bit rimanenti servono a distinguere tra i vari indirizzi di memoria che vanno a finire sullo stesso blocco

Esempio

- Supponiamo di avere una 16KB direct-mapped cache con blocchi di 4 words.
- Determiniamo le dimensioni del tag, indice e offset se usiamo un'architettura a 32-bit.
- Offset
  - Serve a specificare il byte nel blocco
  - Un blocco contiene: 4 parole = 16 bytes = 2^4 bytes
  - Abbiamo bisogno di 4 bit per specificare il byte corretto

Esempio

- Indice
- Specifica la riga corretta nella cache
  - cache contiene 16 KB = 2^{14} bytes
  - Un blocco contiene 2^4 bytes (4 words)
  - # righe della cache =# blocchi della cache (dato che c'è una riga per blocco)
- La cache ha:
  - 2^{14} bytes/2^4 bytes = 2^{10} righe
  - Abbiamo bisogno di 10 bit per le righe (indici)

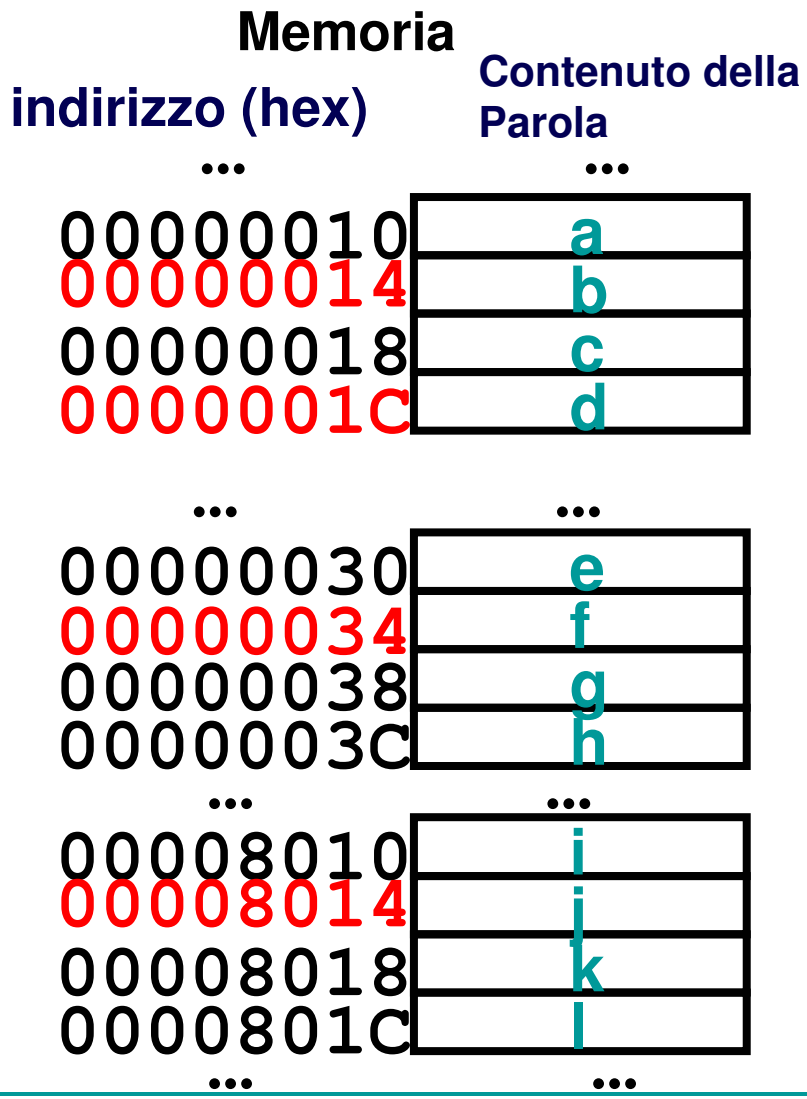
Esempio

- Tag
  - Usiamo i bit restanti come tag

  Lunghezza del tag = lung. ind. mem-offset-Indice = 32 - 4 - 10 bit = 18 bit
  - Quindi il tag è formato dai 18 bit più a sinistra dell'indirizzo di memoria

Accedere ai dati in una direct mapped cache

- Esempio: 16KB, direct-mapped, blocchi da 4 parole
- abbiamo 4 indirizzi
  - 0x00000014, 0x0000001C, 0x00000034, 0x00008014
- I contenuti sono sulla destra



Accedere ai dati in una direct mapped cache

- 4 indirizzi:
  - 0x00000014, 0x0000001C, 0x00000034, 0x00008014
- 4 indirizzi divisi nei campi Tag, Indice e Byte Offset

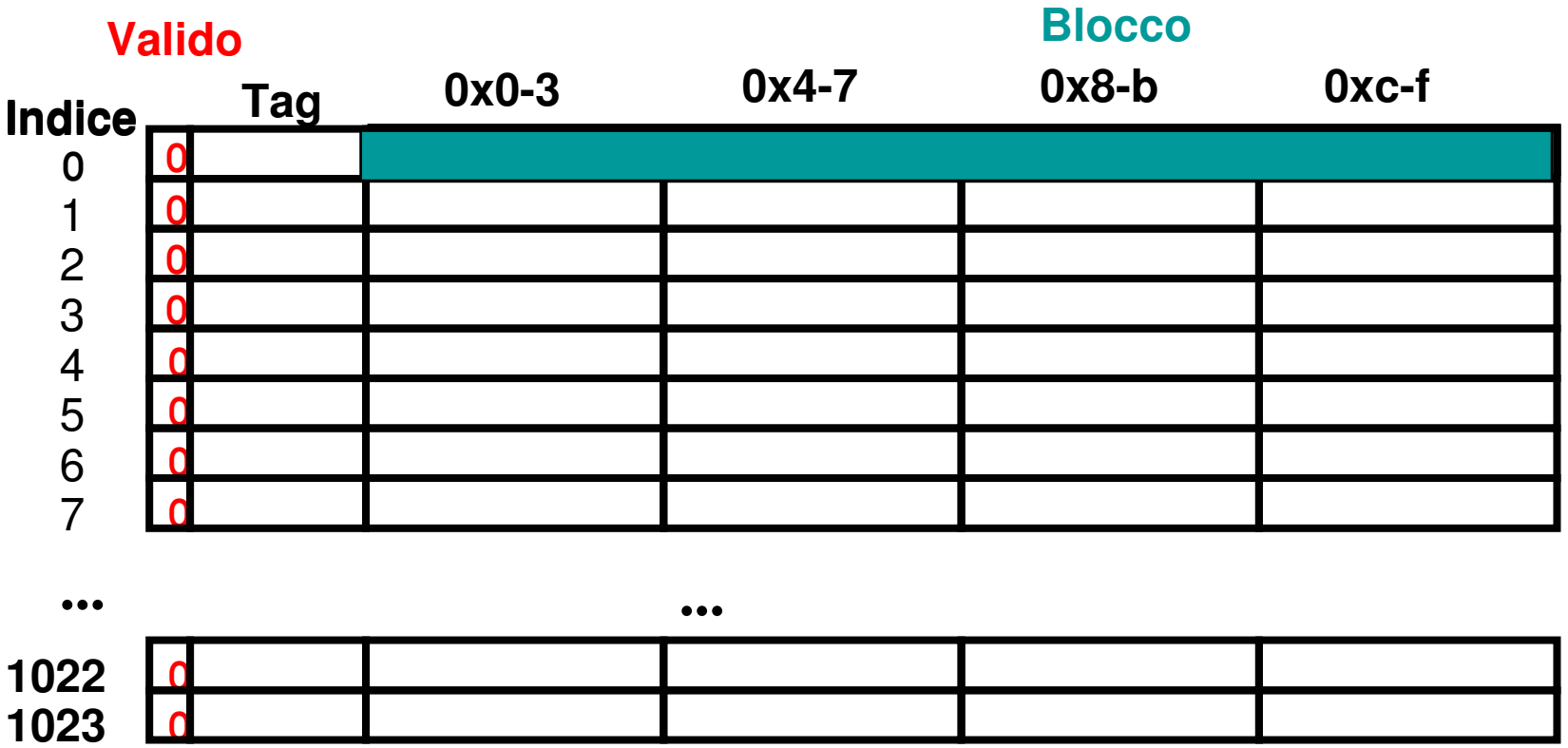
```
00000000000000000 0000000001 0100
00000000000000000 0000000001 1100
00000000000000000 0000000011 0100
00000000000000010 0000000001 0100
      Tag                    Indice    Offset
```

Accedere ai dati in una direct mapped cache

- Accediamo a qualche dato nella cache
  - 16KB, direct-mapped, blocchi di 4 parole
- Ci sono 3 tipi di eventi:
- cache miss: non c'è niente nella cache e nel blocco, recuperare dalla memoria
- cache hit: il blocco nella cache è Valido e contiene l'indirizzo corretto, si può leggere la parola desiderata
- cache miss, rimpiazzamento del blocco: il dato sbagliato è nella cache nel blocco appropriato, va rimosso e va recuperato il dato desiderato dalla memoria

16 KB Direct Mapped Cache, 16B blocks

- Bit di Validità: determina se qualche cosa è stato memorizzato nel blocco (quando accendiamo il computer, sono tutti non Validi)



leggiamo 0x00000014 = 0…00 0..001 0100

- 00000000000000000 0000000001 0100

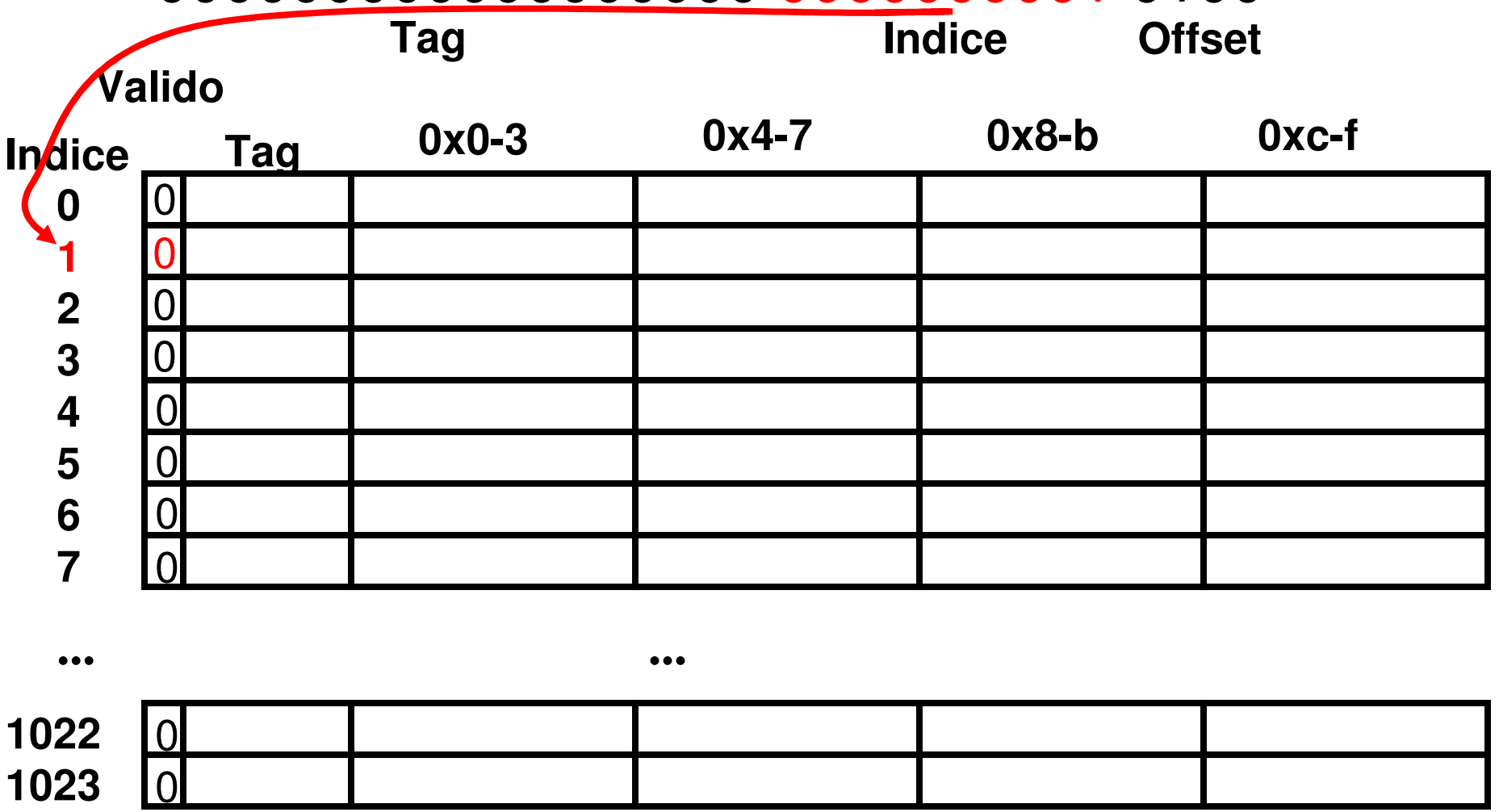
Tag Indice Offset

Indice	Valido	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
…			…			
1022	0					
1023	0					

Dobbiamo leggere il blocco 1 (0000000001)

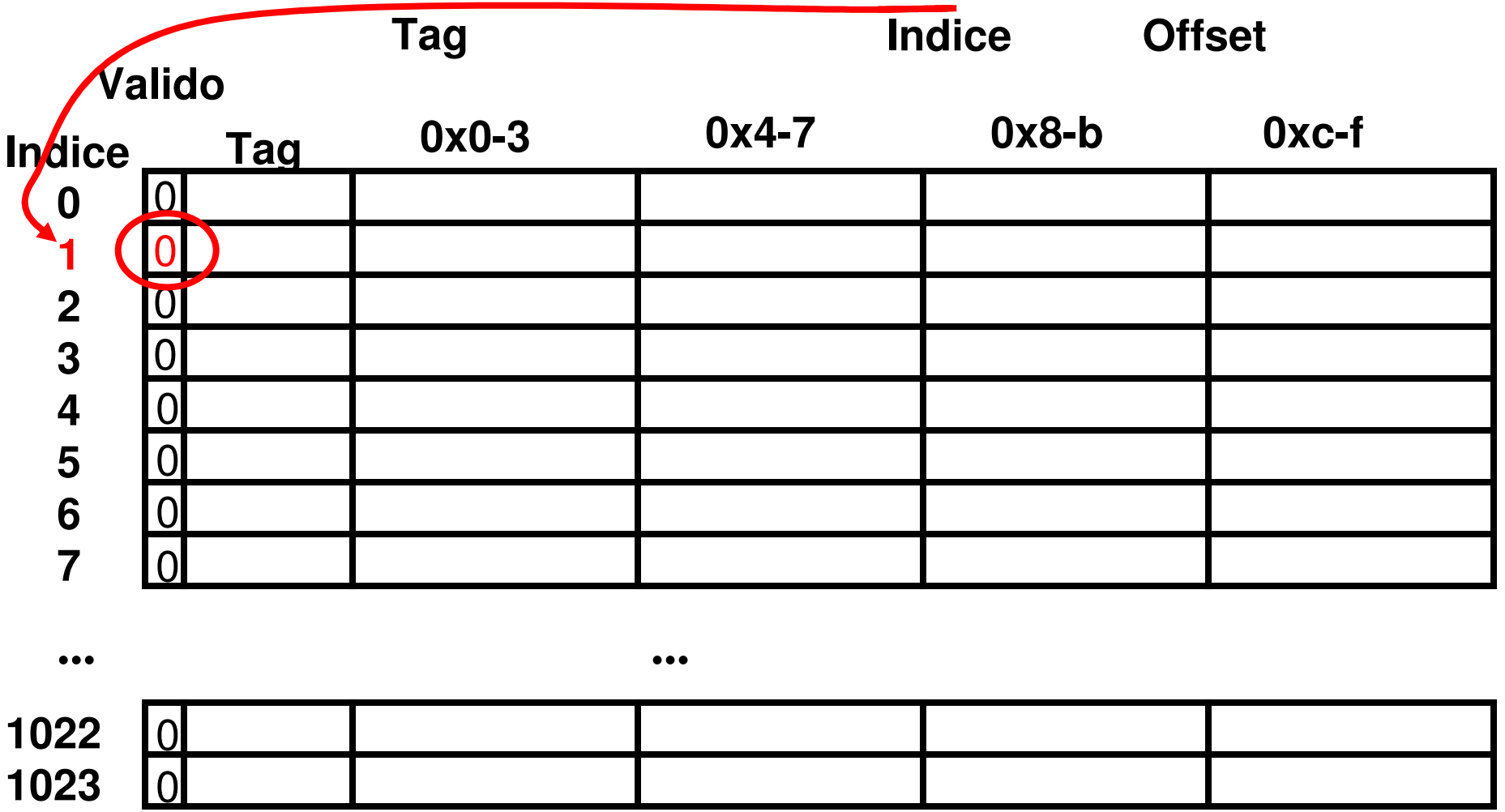
- 00000000000000000 0000000001 0100

Tag Indice Offset



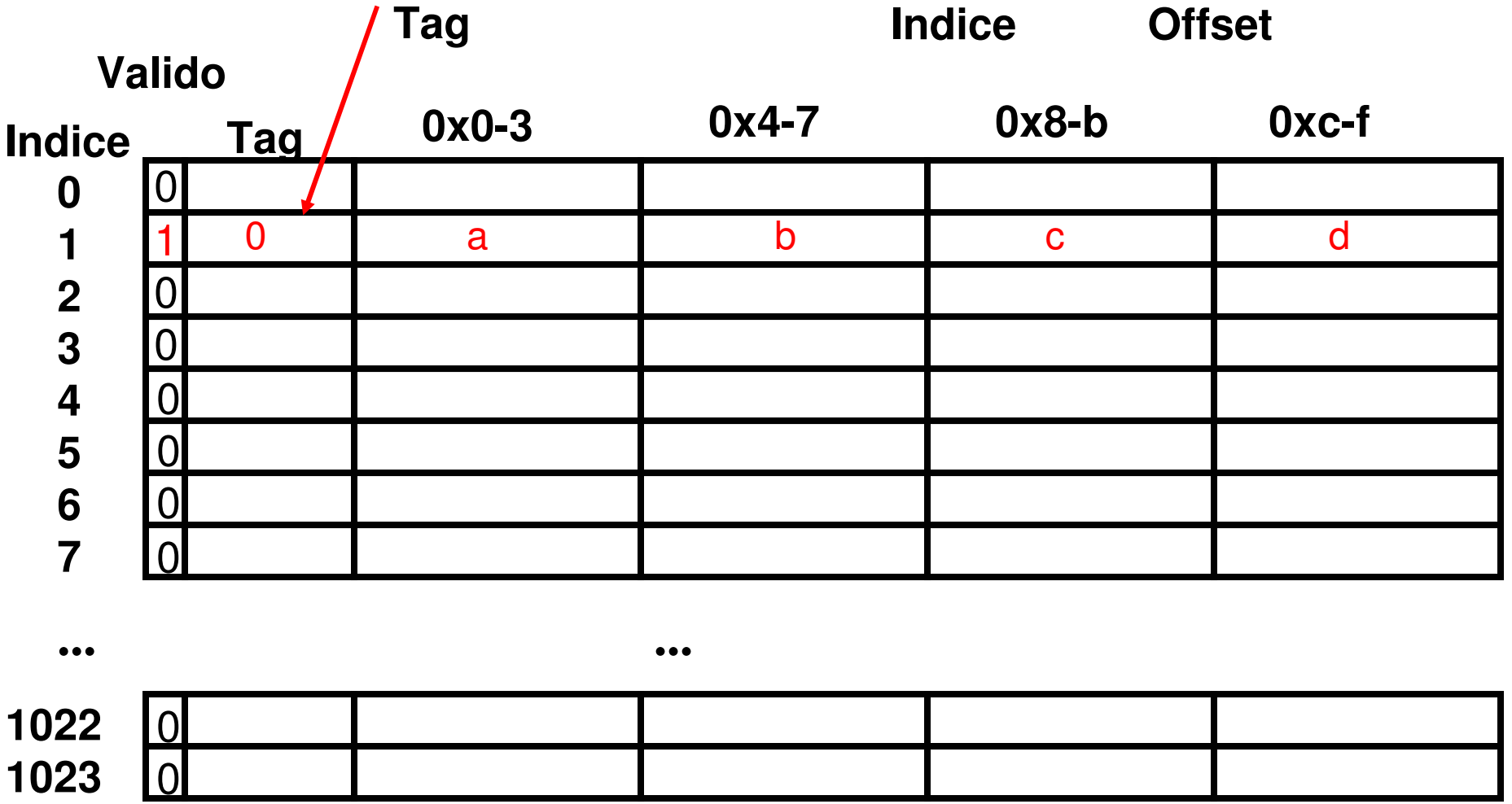
Non ci sono dati Validi

- 00000000000000000 0000000001 0100



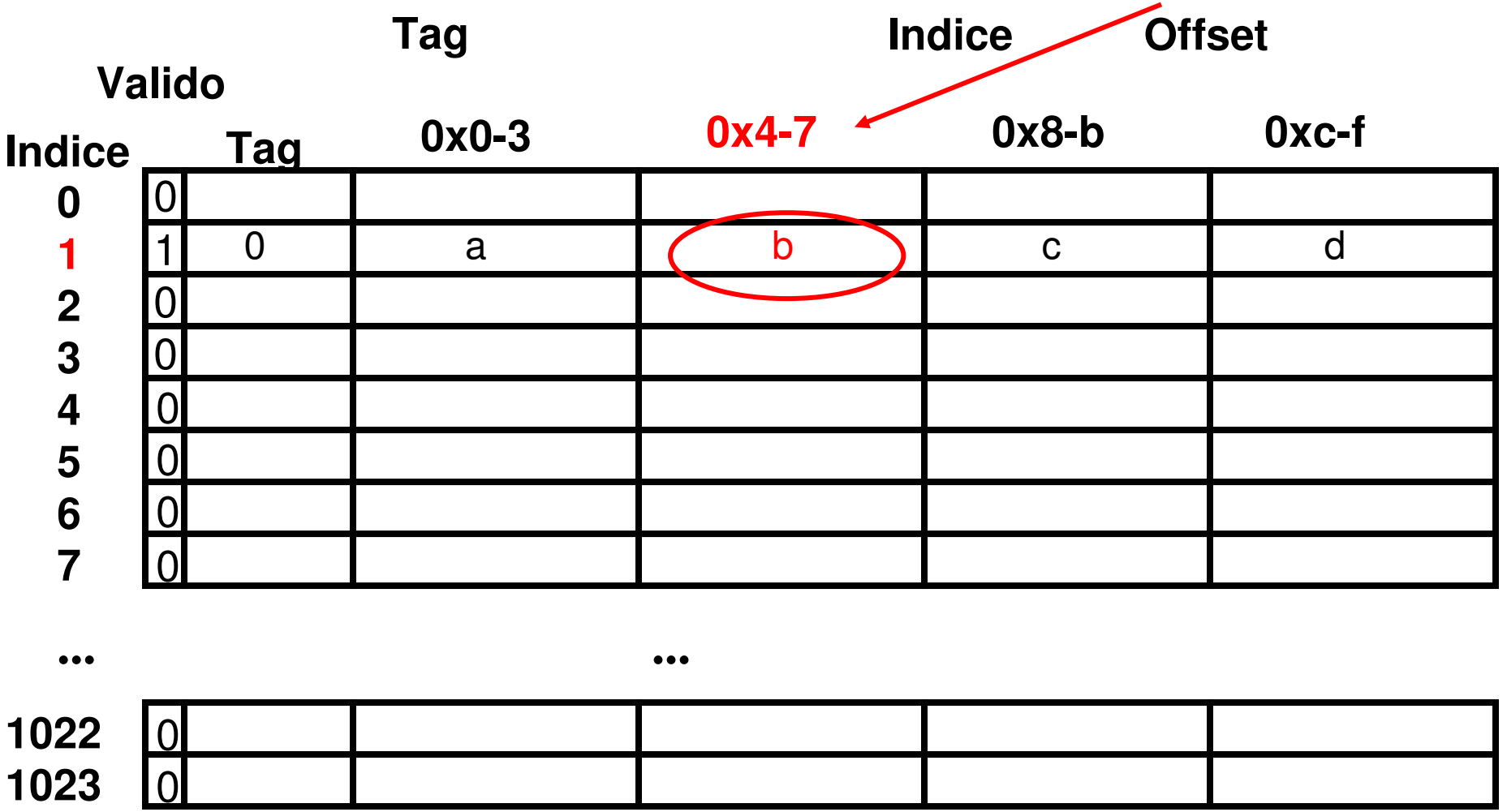
Carichiamo il dato nella cache, inseriamo il tag e settiamo il bit Valido

- 0000000000000000 0000000001 0100



Leggiamo il dato dalla cache

- 0000000000000000 0000000001 0100



Leggiamo il dato 0x0000001C = 0…00 0..001 1100

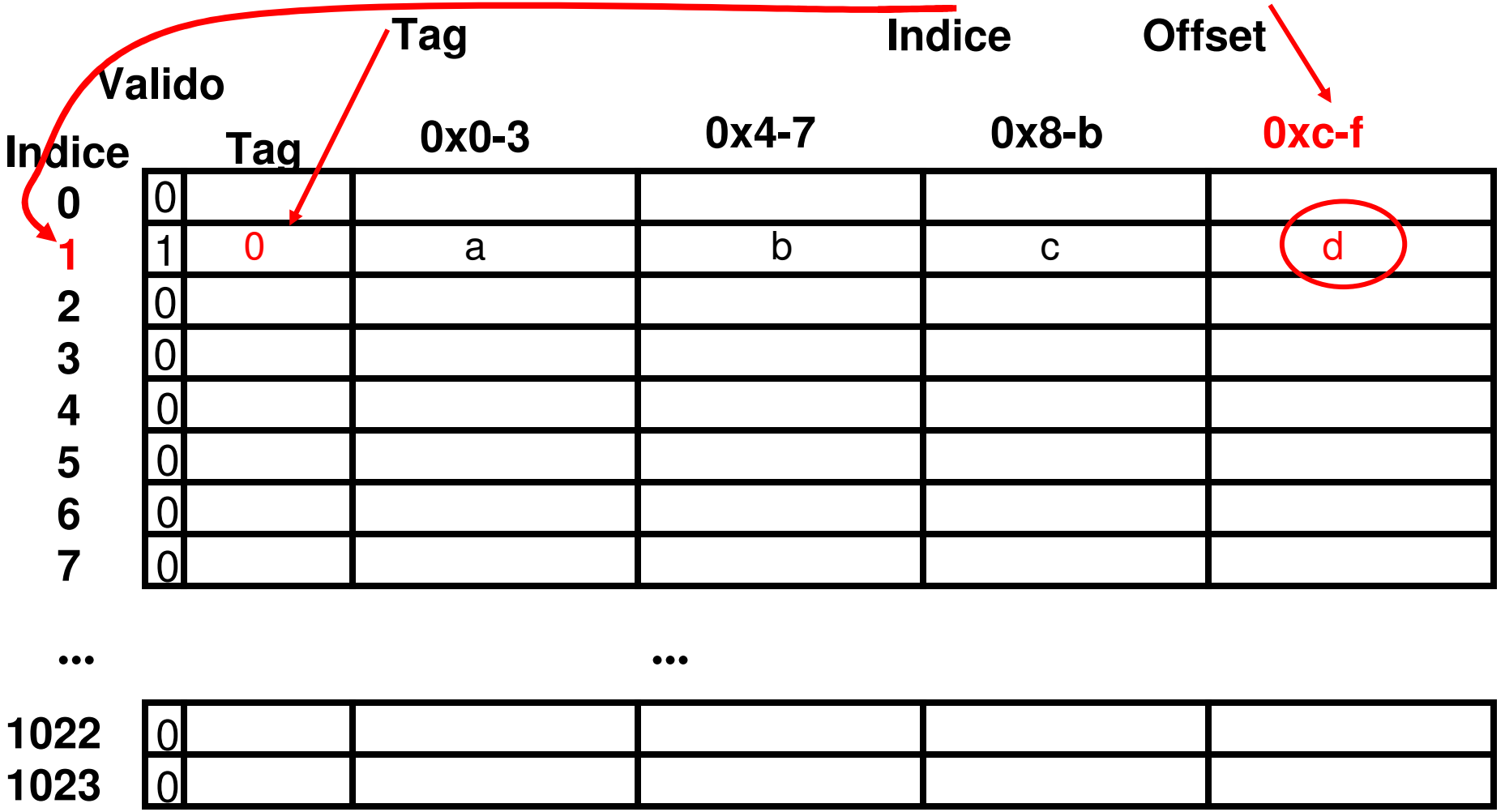
- 00000000000000000 0000000001 1100

Tag | Indice | Offset

Indice	Valido	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
…				…		
1022	0					
1023	0					

Il dato è Valido, il tag OK, leggiamo l'offset e abbiamo la parola desiderata

- 00000000000000000 0000000001 1100



leggiamo 0x00000034 = 0…00 0..011 0100

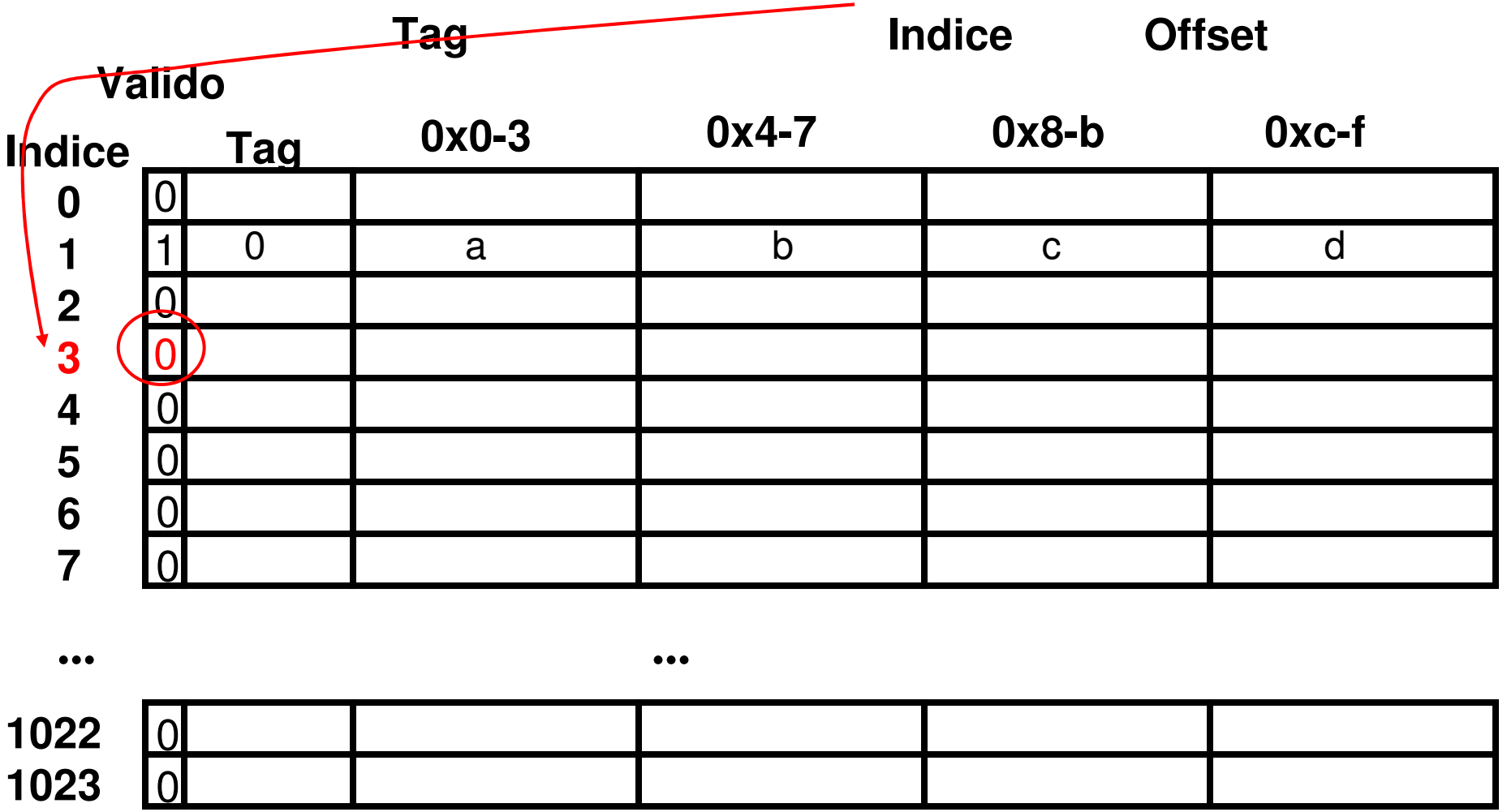
- 00000000000000000 0000000011 0100

Tag Indice Offset

Indice	Valido	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
…				…		
1022	0					
1023	0					

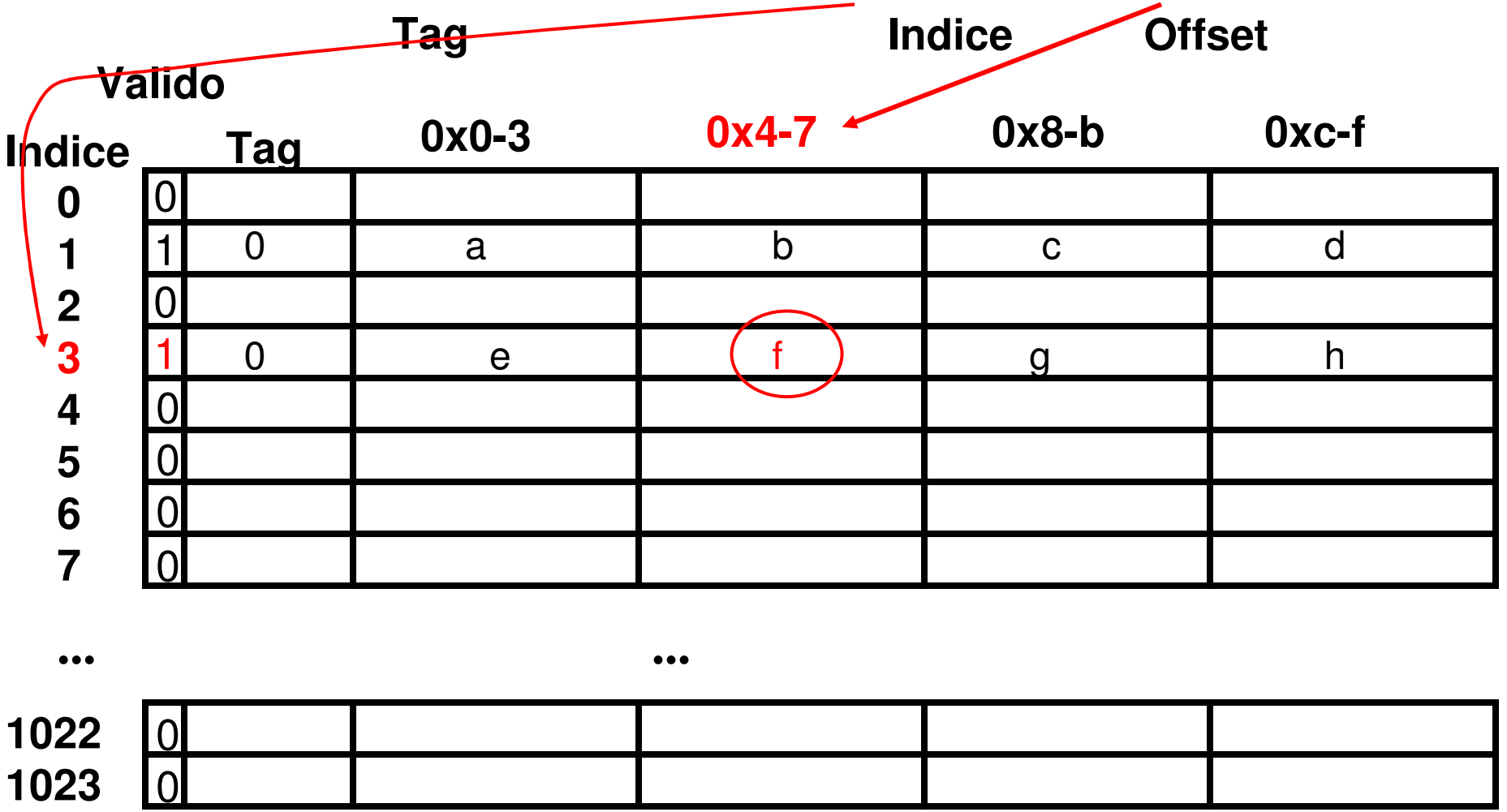
Leggiamo il blocco 3. Dato non Valido

- 00000000000000000 0000000011 0100



Come prima: caricare il blocco e leggere il dato

- 00000000000000000 0000000011 0100



leggiamo 0x00008014 = 0…10 0..001 0100

- 00000000000000010 0000000001 0100

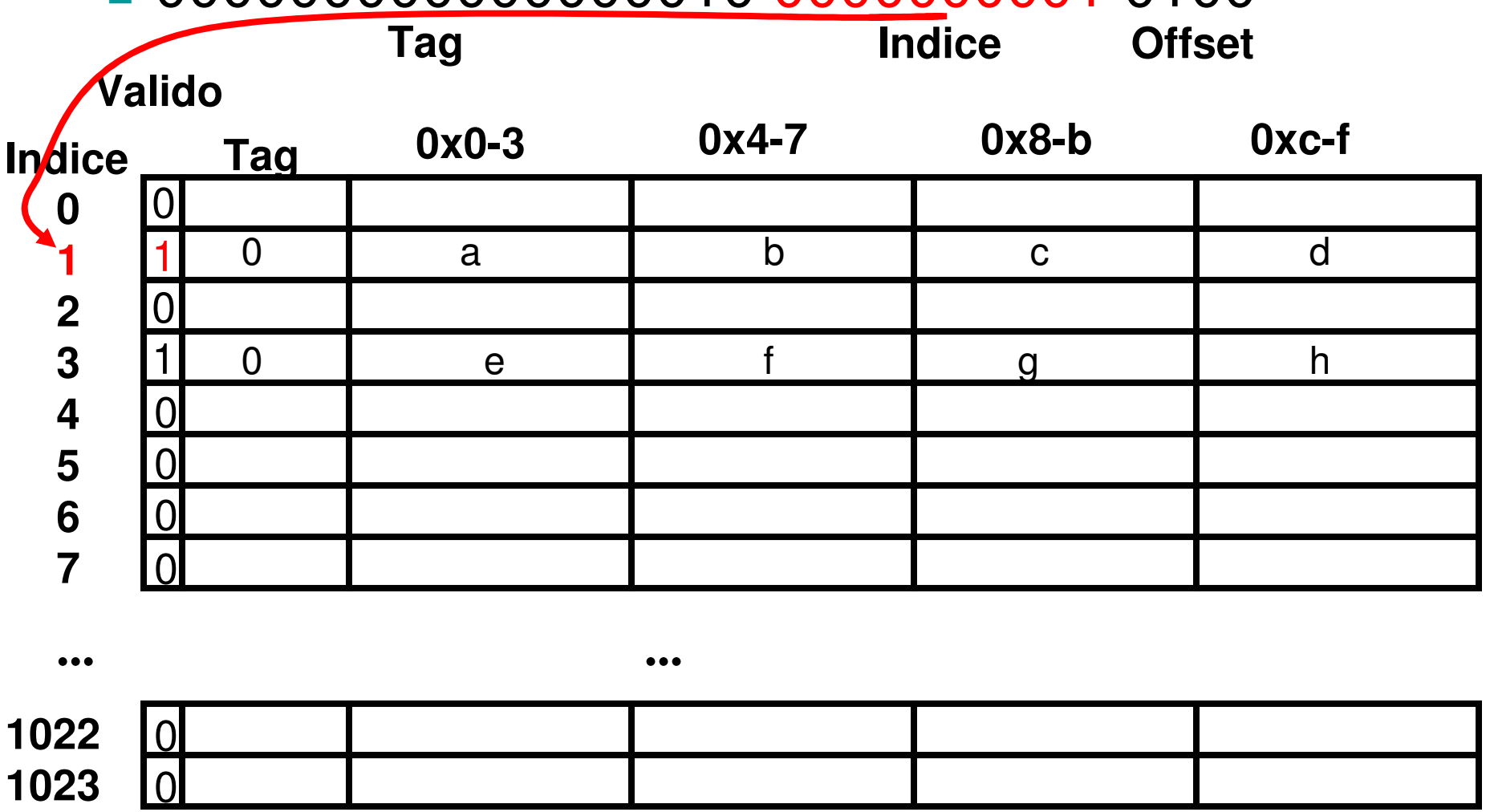
Tag | Indice | Offset

Indice	Valido	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
…				…		
1022	0					
1023	0					

Dobbiamo leggere il blocco 1, il dato è Valido

- 0000000000000010 0000000001 0100

Tag | Indice | Offset



Ma nel Blocco 1 il Tag non coincide (0 ≠ 2)

- 0000000000000010 0000000001 0100

Tag Indice Offset

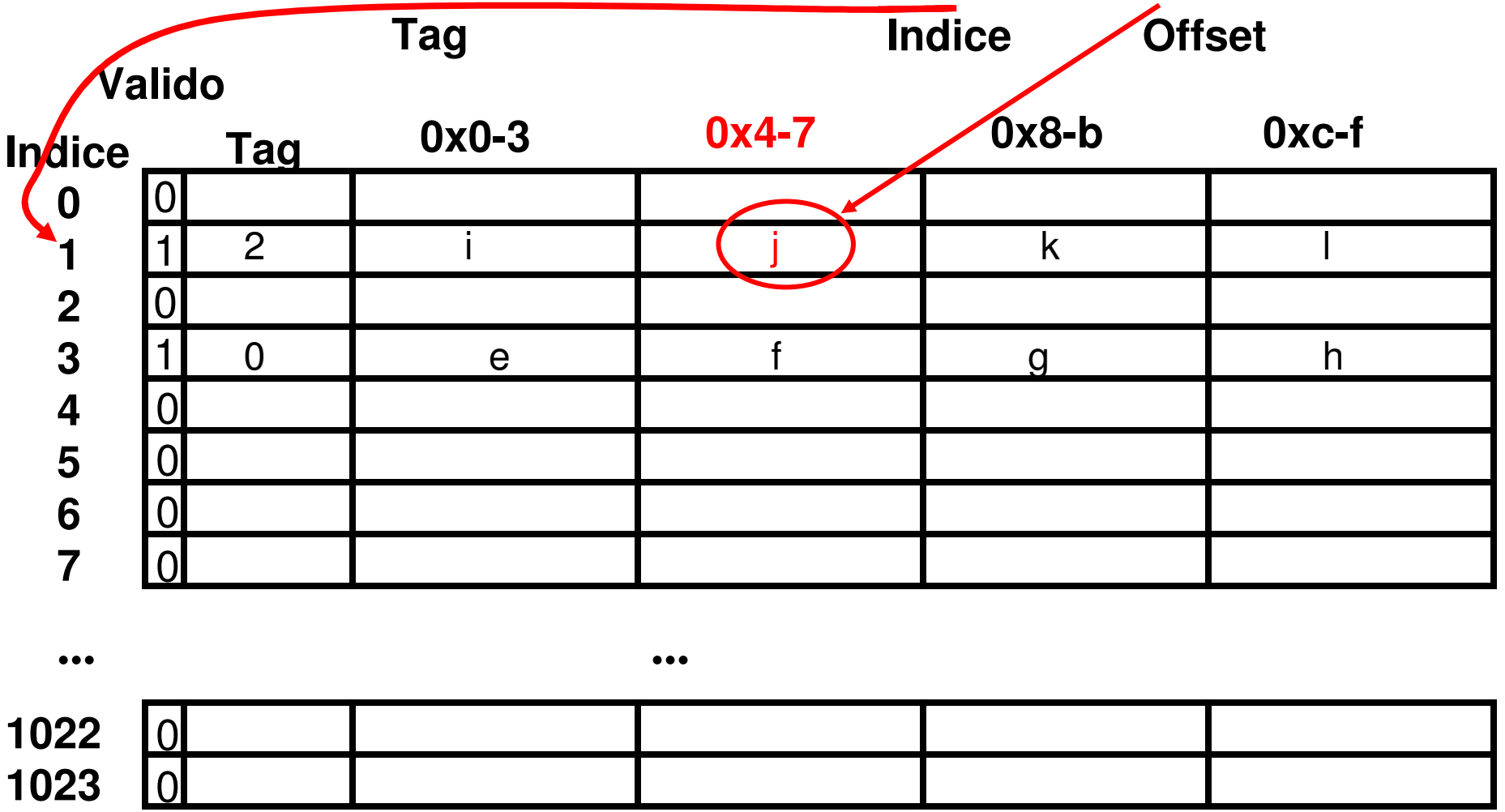
Indice	Valido	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

Miss, rimpiazzare il blocco 1 con nuovi dati & tag

- 0000000000000010 0000000001 0100

E accediamo al dato corretto

- 0000000000000010 0000000001 0100



Sempre meno Von Neumann ...

- Con l'introduzione di una sofisticata gerarchia di memoria basata su tecniche di caching siamo arrivati a individuare una soluzione tecnicamente realizzabile ed economicamente sostenibile per eliminare quasi completamente il tradizionale "collo di bottiglia" dei sistemi di calcolo costituito dalla latenza nell'accesso ai dati. A questo punto l'unico limite all'aumento delle prestazioni del sistema rimane la "relativa lentezza" di una CPU tradizionale rispetto alla frequenza di clock utilizzata per scandire i trasferimenti di dati tra registri all'interno della sua microarchitettura.

Sempre meno Von Neumann ...

- Anche ammettendo di riuscire ad eseguire qualsiasi istruzione della macchina in un singolo ciclo di clock, rimarrebbe comunque la necessità di eseguire in sequenza, in cicli di clock successivi, le diverse fasi di fetch, decodifica, ed esecuzione delle istruzioni, richiedendo quindi almeno 3 cicli di clock consecutivi per l'esecuzione di ogni singola istruzione.
- In questa parte vediamo come, abbandonando la concezione classica di macchina convenzionale sequenziale "tipo Von Neumann" sia invece possibile prima raggiungere e poi addirittura superare decisamente il "limite" di una istruzione eseguita per ogni ciclo di clock.

Pipelining

- Il risultato di arrivare a completare l'esecuzione di una istruzione per ciclo di clock può essere ottenuto rispolverando una tecnica ampiamente adottata nell'industria manifatturiera sin dai tempi di Henry Ford, ed oggi ormai "fuori moda" in quel contesto: la "catena di montaggio". Per evitare ogni riferimento a problemi di carattere sociale oltre che ogni sospetto di plagio o di nostalgie vetero industriali, gli informatici hanno scelto il nome molto più asettico di **pipelining** per riferirsi esattamente alla stessa idea.

Pipelining

- Riassumiamo brevemente l'idea di una strutturazione di tipo pipeline per il completamento di attività. Assodato che l'attività richiesta per eseguire una singola istruzione (o produrre una singola automobile) é quella che é, si rinuncia ad adottare costose tecniche di riduzione della latenza, e ci si propone invece di produrre un aumento del throughput in caso di esecuzione di una sequenza di tante istruzioni successive (ovvero della produzione di tante automobili una dopo l'altra).

Pipelining

- L'aumento di produttività viene ottenuto predisponendo molte unità attive in grado di lavorare simultaneamente. Il "trucco" che consente a queste unità attive di lavorare simultaneamente senza interferire l'una con l'altra consiste nell'assegnare a ciascuna entità attiva un compito molto limitato, che corrisponde ad una ben precisa "fase di lavorazione". Le fasi di lavorazione devono essere applicate sequenzialmente, una dopo l'altra, a tutte le istruzioni che vogliamo eseguire (o automobili da produrre), ed ogni entità attiva "lavora" su un oggetto diverso, che si trova in una fase di lavorazione diversa rispetto a tutti gli altri oggetti presenti nel sistema.

Pipelining

- Consideriamo un esempio concreto. Prendiamo il caso di un microprocessore che deve eseguire un programma sequenziale. Supponiamo che le tre fasi di fetch, decodifica ed esecuzione delle istruzioni richiedano tutte una singola unità di tempo per essere portate a termine.
- In tal caso possiamo organizzare una **pipeline a 3 stadi** costruendo 3 dispositivi diversi, uno specializzato per il fetch, uno per la decodifica, ed uno per l'esecuzione, collegandoli in cascata, con l'uscita di uno coincidente con l'ingresso del successivo.

Pipelining

- L'unità di fetch, che costituisce il primo stadio della pipeline, avrà in ingresso la connessione con la gerarchia di memoria (tipicamente, quindi, la Cache di primo livello), ed in uscita l'unità di decodifica.

Pipelining

- L'unità di decodifica avrebbe in questo caso il compito di tradurre il contenuto del registro IR in ingresso (codifica binaria dell'istruzione da eseguire), in un insieme di bit di controllo per i dispositivi del "data path".

Pipelining

- Infine l'unità di esecuzione dovrebbe contenere il "data path" vero proprio (gli altri registri tolto il registro PC, l'ALU, bus interni, ecc.) in modo da poter prendere in ingresso i segnali di controllo generati dall'unità di decodifica e, sulla base di questi, inserire nuovi valori calcolati dall'ALU in qualche registro interno oppure effettuare degli accessi a celle di RAM in lettura o scrittura.

Pipelining

- Consideriamo quindi l'esecuzione di un programma sequenziale sulla ipotetica architettura pipeline a 3 stadi appena descritta a grandi linee, e ipotizzando una realizzazione sincrona dei diversi stadi comandati da un unico segnale di clock:

```
00008008    ciclo    add #9,(a0)
0000800C             add #2,a0
0000800E             add #1,d0
00008010             cmpi #5,d0
00008014             bne ciclo
00008016             stop #$2700
0000801A             …..
```

Pipelining

- Analizziamo i cicli di clock successivi e cosa le diverse unità possono fare:
- L'unità di fetch dovrebbe "partire" con PC=8008 e nel primo ciclo di clock recuperare il codice operativo della prima istruzione, "passarlo" all'unità di decodifica alla scadere del ciclo, ed incrementare il contenuto di PC.
  Le altre due unità non dovrebbero "far nulla" in questo primo ciclo di clock.

Pipelining

- L'unità di fetch con PC=800C indirizza la seconda istruzione, ed allo scadere del ciclo incrementa il registro PC e memorizza il codice operativo dell'istruzione " add #2,a0 " nel registro IR. Simultaneamente, l'unità di decodifica sulla base del codice operativo della prima istruzione memorizzato nel registro IR riconosce che si tratta di " add #9,(a0) " e predispone i segnali di controllo relativi da inserire nel registro MIR (micro instruction register) allo scadere di questo ciclo.
  L'unità di esecuzione anche in questo secondo ciclo di clock non dovrebbero far nulla.

Pipelining

- L'unità di fetch con PC=800E indirizza la terza istruzione, ed allo scadere del ciclo incrementa il registro PC e memorizza il codice operativo dell'istruzione " add #1,d0 " nel registro IR.
  Simultaneamente, l'unità di decodifica sulla base del codice operativo della seconda istruzione memorizzato nel registro IR riconosce che si tratta di " add #2,a0 " e predispone i segnali di controllo relativi da inserire nel registro MIR allo scadere di questo ciclo.
  Simultaneamente l'unità di esecuzione adesso trova nel registro MIR le indicazioni su come comandare i dispositivi del data path per realizzare la prima istruzione e termina l'esecuzione di tale istruzione allo scadere del terzo ciclo di clock.

Pipelining

- L'unità di fetch con PC=8010 indirizza la quarta istruzione, ed allo scadere del ciclo incrementa il registro PC e memorizza il codice operativo dell'istruzione " cmpi #5,d0 " nel registro IR.
  Simultaneamente, l'unità di decodifica sulla base del codice operativo della terza istruzione memorizzato nel registro IR riconosce che si tratta di " add #1,d0 " e predispone i segnali di controllo relativi da inserire nel registro MIR allo scadere di questo ciclo.
  Simultaneamente l'unità di esecuzione adesso trova nel registro MIR le indicazioni su come comandare i dispositivi del data path per realizzare la seconda istruzione e termina l'esecuzione di tale istruzione allo scadere del quarto ciclo di clock.
- E cosi via…

Pipelining

- Da questa breve simulazione si possono osservare alcune caratteristiche interessanti della tecnica di pipelining.
- Anzitutto osserviamo che l'esecuzione di ogni singola istruzione continua a richiedere 3 cicli di clock consecutivi.
- In secondo luogo notiamo che durante i primi due cicli di clock non viene completata l'esecuzione di nessuna istruzione.

Pipelining

- Dal terzo ciclo di clock in poi, invece, ad ogni ciclo di clock l'unità di esecuzione termina l'esecuzione di una istruzione, quindi se misuriamo il throughput (numero di istruzioni completate per unità di tempo) da questo punto in avanti troviamo effettivamente una velocità di esecuzione di una istruzione per ciclo di clock.
- Dal terzo ciclo di clock in poi tutte e tre le unità "lavorano" in parallelo, su fasi diverse di istruzioni diverse (tra loro consecutive).
- Pur essendo una fase di esecuzione di una istruzione sovrapposta con una fase diversa di un'altra istruzione, l'ordine con cui l'unità di esecuzione tratta le istruzioni rimane esattamente quello sequenziale specificato dal programma

Pipelining

- Purtroppo però le cose non sono così semplici come potrebbero sembrare, e l'adozione di una efficiente tecnica di pipelining per l'esecuzione dei programmi di una macchina convenzionale é tutt'altro che una operazione rapida ed indolore per il progettista. Troviamo un notevole ostacolo alla realizzazione efficiente della pipeline nell'esecuzione di programmi non strettamente sequenziali, contenenti per esempio istruzioni di "salto" o di chiamata di procedura e di ritorno da tali chiamate.

Pipelining

- Infatti, se proseguiamo nella nostra simulazione, al passo numero 5 troviamo l'esecuzione dell'istruzione “**bne**", eseguita in parallelo con la decodifica dell'istruzione **stop** (che non é ancora da eseguire, ovviamente), ed addirittura col fetch del contenuto della cella di indirizzo 801A, che non contiene nemmeno una istruzione appartenente al nostro programma! L'esecuzione dell'istruzione **bne** comporta l'inserimento del nuovo valore 8008 nel registro PC, e quindi al passo numero 6 l'unità di fetch inizierà a leggere il codice operativo "giusto" della prossima istruzione da eseguire, ma l'unità di esecuzione **non dovrà eseguire** gli ordini che troverà memorizzati nel registro MIR (quelli corrispondenti alla istruzione **stop** da eseguire dopo il ritorno dal ciclo che si deve ancora eseguire).

Pipelining

- Tantomeno l'unità di esecuzione dovrà eseguire nel ciclo di clock successivo l'istruzione decodificata a partire dal contenuto della cella di indirizzo 801A! Occorre quindi prevedere la possibilità che l'unità di esecuzione nel caso di salti "ordini" alle altre unità di annullare le loro attività, "svuotando" quindi la pipeline (situazione di **stallo**). In questo modo si evita un errore nella esecuzione del programma, ma si perde l'effetto pipeline sul throughput. Indubbiamente, dopo l'esecuzione dell'istruzione di salto che svuota la pipeline, bisognerà aspettare di nuovo tre cicli di clock prima di veder completare l'esecuzione della istruzione successiva.

Superscalarità

- Per andare oltre l'idea di completare l'esecuzione di una istruzione per ogni ciclo di clock (realizzata mediante la tecnica di pipelining) occorre inevitabilmente prevedere di cominciare l'esecuzione di due o più istruzioni simultaneamente. La possibilità di procedere verso l'esecuzione di più istruzioni **in parallelo** si scontra però contro un ostacolo di natura concettuale: la macchina convenzionale dai tempi di Von Neumann é sempre stata concepita come un dispositivo **sequenziale**, ed i programmi sono tutti organizzati partendo dal presupposto che l'esecuzione di una istruzione inizi solo dopo che l'esecuzione della istruzione "precedente" é terminata.

Superscalarità

- In effetti risulta abbastanza semplice trovare degli esempi nei quali gli operandi di una istruzione sono in realtà calcolati mediante l'esecuzione della istruzione precedente. Alterando la sequenza di esecuzione di tali istruzioni verrebbe meno la relazione di causa-effetto che lega le due istruzioni, ed il risultato prodotto dall'esecuzione del programma sarebbe diverso (e quindi "sbagliato" rispetto alle specifiche di una macchina convenzionale con esecuzione sequenziale delle istruzioni). Per fortuna, risulta altrettanto semplice trovare degli esempi in cui l'ordine con cui sono elencate due istruzioni all'interno del programma non ha alcun effetto sul risultato calcolato: questo, tipicamente, é il caso di istruzioni che agiscono su operandi diversi.

Superscalarità

- Tale casistica può essere formalizzata introducendo le condizioni di dipendenza di Bernstein (dal nome del ricercatore che per primo le ha proposte). Anzitutto ogni istruzione j viene caratterizzata mediante i suoi due insiemi di ingresso I(j) e di uscita O(j).
- L'insieme di ingresso I(j)
  - rappresenta i dati (registri o celle di memoria) che vengono usati dall'istruzione j per produrre il suo risultato, senza che questi vengano modificati
- L'insieme di uscita O(j)
  - rappresenta i dati (registri o celle di memoria) che vengono modificati a seguito dell'esecuzione dell'istruzione j

Superscalarità

- Consideriamo ora una qualsiasi coppia di istruzioni facenti parte di un programma sequenziale. Diremo che le due istruzioni sono (direttamente) dipendenti se, e solo se, esiste una intersezione non vuota tra l'insieme di uscita di una istruzione e l'unione degli insiemi di ingresso e di uscita dell'altra (o viceversa). Ovvero, se indichiamo con x e y due istruzioni consecutive di un programma sequenziale, diremo che x e y sono indipendenti se tutte le intersezioni tra I(x) e O(y), tra O(x) e I(y), e tra O(x) e O(y) sono vuote. L'unica intersezione non vuota "ammissibile" tra due istruzioni indipendenti é quella tra i due insiemi di ingresso.

Superscalarità

- Istruzioni consecutive indipendenti secondo la definizione di Bernstein possono essere eseguite simultaneamente, senza necessità di rispettare l'ordine sequenziale della macchina di Von Neumann, senza per questo cambiare i risultati calcolati. Istruzioni consecutive dipendenti (in quanto condividono dati in uscita) devono essere invece eseguite nell'ordine sequenziale prestabilito, altrimenti si rischia di calcolare un risultato diverso da quello voluto.

Superscalarità

- Una realizzazione efficiente della esecuzione parallela di istruzioni sequenziali indipendenti richiede una serie di restrizioni rispetto alla impostazione generale che abbiamo dato al problema. Questa funzionalità di esecuzione simultanea di più istruzioni, che viene detta **superscalarità**, storicamente é stata introdotta nel caso di processori RISC per istruzioni di manipolazione dei dati contenuti nei registri interni della CPU, escludendo la parallelizzazione di istruzioni complesse con accesso ad operandi in memoria.

Superscalarità

- La realizzazione fisica prevede l'uso di più pipeline "in parallelo", ciascuna dedicata al fetch, decodifica ed esecuzione di istruzioni consecutive. Tale realizzazione si "sposa bene" con l'uso di linee di cache di dimensioni multiple di parole di memoria. Il fetch del codice operativo di più istruzioni consecutive viene realizzato inviando simultaneamente il contenuto delle parole di memoria raggruppate all'interno di una stessa linea di cache a più unità di decodifica, tante quanto il "livello di superscalarità" definito (di solito 2, 3 o 4).

Superscalarità

- La fase di decodifica simultanea delle istruzioni viene realizzata, oltre che replicando i dispositivi di decodifica singoli, anche introducendo una ulteriore unità di "coordinamento", che rileva la presenza di conflitti sugli operandi (condizioni di Bernstein) e sulle unità di esecuzione arrivando quindi a decidere se le istruzioni possono effettivamente essere eseguite in parallelo oppure no.

Superscalarità

- Tale unità di coordinamento può essere realizzata in modo più o meno sofisticato, ed in particolare permettendo o meno "permutazioni dinamiche di istruzioni" nel caso in cui ci siano conflitti che impediscono l'esecuzione simultanea delle istruzioni di cui si é fatto il fetch simultaneo, in modo da cercare di parallelizzare l'esecuzione almeno con le istruzioni successive.