Calcolatori Elettronici: la memoria centrale

G. Lettieri

1 Marzo 2024

Nell'architettura moderna ogni singolo byte della memoria ha il proprio indirizzo e le istruzioni del processore permettono di accedere al singolo byte. Per esempio, nel processore Intel, `mov %al, (%rbp)` permette di sovrascrivere il singolo byte all'indirizzo contenuto in **rbp**, senza modificare nessun altro byte. Il sottosistema di memoria, quindi, deve essere in grado di supportare questo tipo di operazioni. Allo stesso tempo, però, il processore possiede istruzioni che leggono e scrivono in unità più grandi: l'istruzione `mov %rax, (%rbp)` scrive 8 byte all'indirizzo contenuto in **rbp**. Vorremmo che il sottosistema di memoria fosse in grado di supportare in maniera efficiente tutti i tipi di accessi che il processore supporta: al byte, alla parola (2 byte, nella terminologia Intel/AMD64), doppia parola (4 byte) o parola quadrupla (8 byte).

Il byte rappresenta l'unità di indirizzamento: non è possibile leggere o scrivere una unità più piccola di un byte in una singola operazione di lettura o scrittura. Se, per esempio, è necessario modificare un singolo bit all'interno di un byte, è necessario leggere l'intero byte, modificare il bit lasciando gli altri inalterati, quindi riscrivere il nuovo byte.

Quando si passa a oggetti di più byte, la prima cosa da chiedersi è che in che ordine i byte che compongono l'oggetto si trovino in memoria. Architetture diverse possono utilizzare ordini diversi. Nel nostro caso l'architettura usa l'ordinamento detto *little endian*: il byte meno significativo si trova all'indirizzo più piccolo, seguito dagli altri byte in ordine di significatività. Per esempio, supponiamo di avere una parola quadrupla in memoria, a partire da un indirizzo i, che memorizza il numero esadecimale 0x1122334455667788. Il byte di indirizzo i conterrà 0x88, il byte di indirizzo $i + 1$ conterrà 0x77, e così via fino al byte di indirizzo $i + 7$ che conterrà 0x11. Un altro ordinamento molto utilizzato (per esempio da noi stessi quando scriviamo i numeri su un foglio) è quello detto *big endian*[1], che consiste nello scrivere il numero partendo dalla parte più significativa. In Fig. 1 abbiamo ordinato i byte di ogni riga da destra verso sinistra proprio per ovviare alla differenza tra il modo in cui l'architettura AMD64 memorizza le parole e il modo in cui noi siamo abituati a leggerle.

Possiamo realizzare un modulo di memoria che possa anche leggere o scrivere più di un byte in una singola operazione e nello stesso tempo impiegato per un

[1] I nomi *little endian* e *big endian* vengono indirettamente da *I viaggi di Gulliver* di Jonathan Swift, per il tramite di un famoso articolo del 1980 reperibile a questo indirizzo: `https://www.ietf.org/rfc/ien/ien137.txt`

numero di riga	+7	+6	+5	+4	+3	+2	+1	+0	indirizzo di riga
0									0
1									8
2									16
3									24
				...					
$2^{61}-1$									$2^{64}-8$

Figura 1: Organizzazione dello spazio di memoria su righe di 64 bit. Ogni casella rappresenta un byte. I numeri +0, +1, ... sono gli offset all'interno della riga.

singolo byte. Nel nostro caso il modulo può leggere o scrivere una intera parola di 64 bit (8 byte contigui) purché questa sia *allineata naturalmente*. Il nostro modulo di memoria può anche leggere o scrivere, in una sola operazione, una parola di 32 bit (4 byte) o una parola di 16 bit (2 byte), purché queste non attraversino i confini di 8 byte (cioè siano interamente contenute in una regione naturale di 8 byte).

Per visualizzare più facilmente queste limitazioni conviene rappresentare lo spazio di indirizzamento di memoria non come una sequenza di indirizzi di byte, ma come una sequenza di *righe* o *linee* di 8 indirizzi di byte (il numero massimo di byte che può essere trasferito in una singola operazione da un modulo di memoria). Si veda la Fig. 1, dove le righe sono disposte in orizzontale. In questo modo i dati accessibili in un'unica operazione saranno sempre contenuti in una singola riga. In pratica adottiamo una scomposizione degli indirizzi in regioni di 2^3 e disegniamo le regioni in orizzontale.

Si noti che per avere le parole da 2 byte e le parole da 4 byte sempre contenute in una riga è sufficiente che anch'esse siano allineate naturalmente, ma non è necessario. Per esempio, una parola da 4 byte che inizi in una riga alla colonna +3 sarebbe interamente contenuta nella riga, pur non essendo allineata naturalmente (per essere allineata naturalmente dovrebbe iniziare alla colonna +0 o alla colonna +4).

Nel seguito assumeremo che gli indirizzi della CPU e del BUS siano su n bit. Non assumiamo che n sia 64, in quanto questo è solo il valore massimo teorico dell'architettura Intel/AMD64. I processori di questa famiglia usciti fino ad ora hanno, di fatto, un numero inferiore di bit di indirizzo (per esempio, 36) e altre considerazioni, che vedremo, limitano il massimo numero di bit dei processori futuri della stessa famiglia a un valore comunque inferiore a 64.

Per specificare completamente una richiesta di operazione di lettura la CPU deve presentare alla memoria l'indirizzo del primo byte e il numero di byte (per una operazione di scrittura dovremo anche presentare il nuovo contenuto dei byte in questione). Queste due informazioni vengono specificate in modo indiretto nel seguente modo:

1. la CPU scompone l'indirizzo del primo byte in numero di riga e offset all'interno della riga;

2. la CPU e il bus prevedono $n-3$ fili, che chiamiamo A$\{n-1\}$–A3, destinati a trasportare soltanto il numero di riga;

3. al posto dell'offset del primo byte (che sarebbe stato contenuto nei fili A2, A1 e A0) e del numero di byte sono previsti 8 fili di *byte enable*, uno per ogni byte della riga selezionata da A$\{n-1\}$–A3.

Lo scopo delle linee di byte enable, che chiamiamo /BE7–/BE0, è di selezionare singolarmente i byte della riga che la CPU intende leggere (o scrivere) nell'operazione.

Esempi:

- Supponiamo che la CPU stia eseguendo una istruzione **`movq`** `512, %`**`rax`**. Deve dunque ordinare una operazione di lettura di una parola da 8 byte all'indirizzo 512. Si tratta della riga n. $512/8 = 64$, che va dagli indirizzi 512 a 519. Avremo A$\{n-1\}$ = A$\{n-2\}$ = $\cdots$ = A10 = 0, A9 = 1, A8 = A7 = $\cdots$ = A3 = 0, in quanto 512 è 1000000000 in binario, e /BE7 = /BE6 = $\cdots$ = /BE0 = 0 (tutti i byte abilitati);

- Supponiamo invece che l'istruzione sia **`movl`** `512, %`**`eax`**. Si tratta ora di una lettura di una parola da 4 byte all'indirizzo 512: avremo A$\{n-1\}$–A3 come prima, ma /BE7 = /BE6 = /BE5 = /BE4 = 1 (byte 7–4 disabilitati) e /BE3 = /BE2 = /BE1 = /BE0 = 0 (byte 3–0 abilitati);

- Consideriamo ora **`movl`** `516, %`**`eax`**. Ora la lettura coninvolge una parola da 4 byte all'indirizzo 516: avremo A$\{n-1\}$–A3 ancora come prima, in quanto siamo sempre all'interno della stessa riga. I byte enable saranno però diversi: /BE7 = /BE6 = /BE5 = /BE4 = 0 (byte 7–4 abilitati) e /BE3 = /BE2 = /BE1 = /BE0 = 1 (byte 3–0 disabilitati).

Si noti che questo tipo di codifica permetterebbe di specificare molti più casi di quelli che ci servono. Per esempio, con /BE7 = /BE5 = /BE3 = /BE1 = 1 e /BE6 = /BE4 = /BE2 = /BE0 = 0 potremmo leggere solo i byte di indirizzo pari all'interno della riga selezionata. Di fatto tali possibilità non sono sfruttate e sono utilizzate solo le combinazioni che corrispondono a byte contigui.

1 Collegamento al bus

Consideriamo ora un modulo di memoria da 2^k byte, per esempio $k = 30$ per una memoria di 1 GiB. Possiamo realizzare le funzionalità che ci servono se costruiamo il modulo usando 8 dispositivi di memoria, ciascuno capace di memorizzare $2^k/8 = 2^{k-3}$ byte (nell'esempio precedente, con $k = 30$, ci servono 8 moduli da 128 MiB). I dispositivi devono essere collegati come in Fig. 2. Ciascuna riga di Fig. 1 è memorizzata utilizzando tutti e 8 i dispositivi: il byte della colonna "+0" sarà memorizzato nel dispositivo contrassegnato con 0, il byte "+1" nel dispositivo 1, e così via. Ogni dispositivo memorizza una intera colonna di Fig. 1. Si noti come la presenza dei byte enable semplifichi l'implementazione:

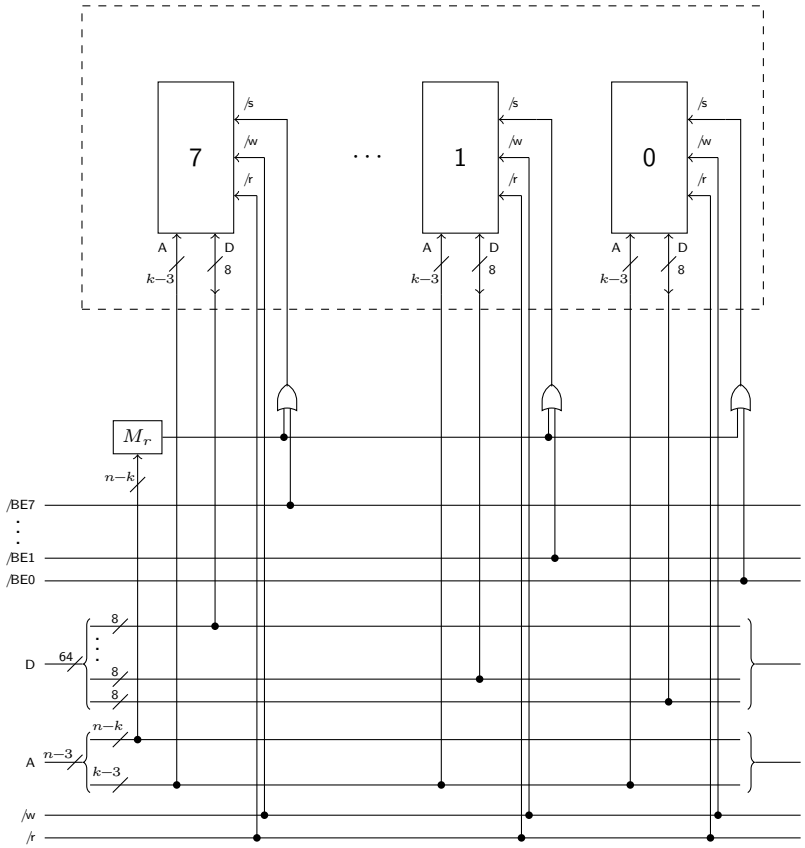

Figura 2: Realizzazione di una memoria organizzata a linee di 64 bit (parole quadruple), ma accessibile anche a byte, parole e doppie parole. Le parti dentro e fuori del rettangolo tratteggiato possono essere realizzate separatemente e poi collegate. La parte interna è una scheda di memoria, la parte esterna è il bus.

ogni byte enable contribuisce a genererare il *chip select* del chip che contiene il corrispondente byte.

Il nostro modulo di memoria convive nello spazio di indirizzamento di memoria insieme ad altri dispositivi (altri moduli di memoria, oppure anche interfacce di I/O con registri mappati in memoria). Dobbiamo assegnargli un intervallo di indirizzi e fare in modo che risponda solo alle richieste di lettura e scrittura che ricadono in quell'intervallo. Per farlo conviene scegliere una regione naturale di 2^k byte e fare in modo che la nostra memoria la occupi interamente (facciamo in modo, cioè, che il nostro modulo sia allineato naturalmente all'interno dello spazio di indirizzamento). Sia r il numero della regione grande 2^k che abbiamo scelto. Data una operazione di lettura o scrittura ad un certo numero di riga i, la nostra memoria deve sapere se ignorarla o no in base al fatto che il numero di riga i appartenga o meno alla regione r. Abbiamo già visto come fare a vedere se un indirizzo su n bit appartiene o no ad una regione grande 2^k: è sufficiente guardare gli $n - k$ bit più significativi dell'indirizzo. In questo caso non abbiamo tutto l'indirizzo (di byte) ma solo il numero di riga. Questo non è un problema: ci mancano solo i 3 bit meno significativi dell'indirizzo di byte, e questi sicuramente rientrano nei k che già dovevamo ignorare. Un altro modo per visualizzare l'operazione è di riportare tutto alle righe: una regione di 2^k byte è anche una regione di 2^{k-3} righe. I numeri di regione restano gli stessi che per i byte e per sapere se una riga appartiene o no alla regione scelta è sufficiente ignorare $k - 3$ bit meno significativi del numero di riga e confrontare gli altri con il numero di regione r. I $k - 3$ bit meno significativi, invece, sono l'offset *della riga* all'interno della regione. Questi possono essere usati per selezionare il byte corretto in ciascun chip di RAM.

Il circuito risultante è quello di Figura 2. La maschera M_r serve ad abilitare o disabilitare complessivamente tutta la scheda di memoria.

La parte interna al rettangolo tratteggiato in Fig. 2 rappresenta la scheda di memoria vera e propria, mentre la parte esterna rappresenta i circuiti del bus a cui la scheda è collegata. Il bus può proseguire a destra e sinistra e avere altre maschere che riconoscono valori di r diversi. Tipicamente avremo una "scheda madre" che prevede un certo numero di slot in cui si possono inserire le schede di memoria. Ogni slot avrà una maschera diversa. Le schede di memoria possono invece essere tutte uguali tra loro ed essere montate in un qualunque slot; più schede possono essere montate contemporaneamente sul bus, ciascuna ovviamente inserita in uno slot diverso.

2 Accessi non allineati

I processori AMD/Intel a 64 bit permettono anche accessi non allineati. Per esempio, è possibile leggere con una sola istruzione una parola quadrupla che inizi ad un indirizzo che non è multiplo di 8:

```
        mov 4097, %rax
```

numero di riga	+7	+6	+5	+4	+3	+2	+1	+0	indirizzo di riga
				...					
511									4088
512	22	33	44	55	66	77	88		4096
513								11	4104
514									4112
				...					

Figura 3: Accesso non allineato.

Supponiamo che all'indirizzo 4097 sia memorizzato il numero 1122334455667788 (base 16). I byte che compongono il numero si troveranno nelle posizioni indicate in Figura 3. In questo caso il processore eseguirà due accessi in memoria: uno alla riga numero 512 (4096/8) con tutti i byte enable attivi, tranne /BE0; un altro alla riga successiva (513), con tutti i byte enable *non* attivi, tranne /BE0. In questo modo riesce a recuperare tutti i byte che compongono il numero, ma non basta: la prima lettura porterà i byte 22–88 in un registro interno del processore, ma in una posizione che è traslata di 8 bit a sinistra rispetto a quella desiderata; la seconda lettura porterà il byte 11 nella posizione meno significativa del registro interno, invece che in quella più significativa. Il processore, automaticamente, provvederà ad eseguire i necessari shift in modo che **`rax`**, alla fine, contenga il valore corretto.

Si noti come il soggetto di tutte queste azioni è *il processore*: il software contiene solo l'istruzione "`mov 4097, %rax`" e non menziona in alcun modo le due letture e gli shift. Queste sono azioni svolte dal processore per eseguire correttamente l'istruzione richiesta.

Da quanto abbiamo descritto, possiamo capire che gli accessi non allineati comportano un costo in termini di tempo, e richiedono hardware aggiuntivo nel processore. Alcuni tipi di processori (per es., quelli di ARM) non contengono questo hardware e non ammettono accessi non allineati.

3 Allineamento dei dati in C++

Anche nei processori che possiedono l'hardware per l'accesso non allineato, come quelli Intel/AMD64, ha senso cercare di non portarsi mai nella situazione in cui occorrono accessi disallineati, per non pagarne il costo in termini di tempo. Per questo motivo, l'ABI (Application Binary Interface) di un sistema definisce delle regole di allineamento che il compilatore deve rispettare. Nel nostro caso valgono le regole dell'ABI System V, che è quella adottata da Linux[2]. L'ABI stabilisce quale deve essere il risultato degli operatori **`sizeof`** e `alignof` per ogni tipo, sia base che derivato.

[2] `https://calcolatori.iet.unipi.it/deep/psABI-x86_64.pdf`

tipo	**sizeof**	alignof
char	1	1
short	2	2
int	4	4
long	8	8

Tabella 1: Allineamenti dei tipi base nell'ABI System V.

```
struct S1 {
    char c;
    int i;
    long l;
};
```
(a)

```
struct S2 {
    char c;
    int i;
    long l;
    char c2;
};
```
(b)

```
struct S3 {
    char c;
    char c2;
    int i;
    long l;
};
```
(c)

Figura 4: Tre esempi di strutture.

Per i tipi base che ci interessano vale la Tabella 1. Come si vede, tutti i tipi sono allineati naturalmente. La tabella non riporta i corrispondenti tipi **unsigned**, perché non c'è differenza con i tipi **signed**.

Più in generale, l'ABI garantisce che, per ogni tipo `t`, sia base che derivato, **sizeof**`(t)` sia un *multiplo* di `alignof(t)`. L'utilità di ciò è semplice da vedere nel caso degli array. Se abbiamo un array `tipo a[dim]`, allora `alignof(a)` è uguale a `alignof(tipo)` (gli array hanno lo stesso allineamento dei loro elementi). Se **sizeof**`(tipo)` è un multiplo di `alignof(tipo)`, possiamo disporre in memoria gli elementi `a[0]`, `a[1]`, ..., `a[dim-1]` semplicemente uno dopo l'altro, perché ciascuno di loro rispetterà il proprio allineamento e sarà possibile calcolare l'indirizzo dell'elemento i-esimo come $\texttt{a} + i \times \texttt{sizeof(tipo)}$. Inoltre, **sizeof**`(a)` sarà semplicemente $\texttt{dim} \times \texttt{sizeof(a)}$, rispettando dunque il vincolo anche per l'array nella sua interezza.

Passiamo ora alle strutture. Calcolare l'allineamento è semplice: una struttura ha per allineamento il *massimo* degli allineamenti dei suoi membri. Calcolare la dimensione, invece, è più complicato, perché bisogna garantire che *ogni* membro della struttura rispetti il proprio allineamento, e che i membri siano disposti in memoria nell'ordine in cui il programmatore li ha dichiarati. Questo comporta che ogni tanto si debbano saltare dei byte (*internal padding*).

Consideriamo, per esempio, la struttura `S1` definita in Figura 4(a). Il suo allineamento è 8, per via del campo `l`. Per calcolare la sua dimensione dobbiamo disegnarne il *layout* in memoria, assumendo di partire da un indirizzo allineato a 8. Il risultato si può vedere in Figura 5. Il primo campo da allocare dentro la struttura è `c`, che ha allineamento uno e dunque occupa il primo byte. Il secondo campo da allocare è `i`, che ha allineamento 4. La prima posizione disponibile subito dopo `c`, però, non ha un indirizzo multiplo di 4, quindi bi-

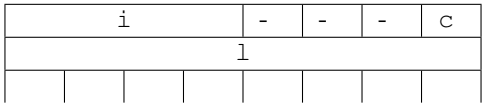

Figura 5: Layout di memoria della struttura `S1` di Figura 4(a).

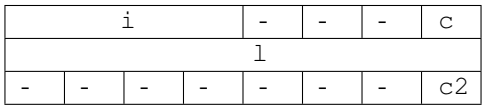

Figura 6: Layout di memoria della struttura `S2` di Figura 4(b).

sogna saltare un numero sufficiente di byte, fino ad arrivare al primo indirizzo allineato a 4: in questo caso è necessario saltare 3 byte. Questi byte concorrono a definire la dimensione della struttura, ma non sono utilizzati in alcun modo. Dopo aver allocato `i` si procede ad allocare `l`, che ha allineamento 8. In questo caso la prossima posizione è allineata a 8, e quindi `l` può occuparla. A questo punto possiamo contare i byte che abbiamo usato (inclusi quelli saltati): sono 16. Siccome 16 è multiplo di `alignof(S1)`, che avevamo detto essere 8, **`sizeof`**`(S1)` è effettivamente 16.

Consideriamo ora la struttura `S2` di Figura 4(b). Rispetto alla struttura `S1` abbiamo aggiunto un unico **`char`** `c2` in fondo. L'allineamento di `S2` è ovviamente sempre 8. Il layout della struttura è illustrato in Figura 6. Notare come `c2` deve essere allocato per ultimo, perché questo è l'ordine stabilito dal programmatore (colui che ha definito `S2`). Quindi `c2` deve finire su una terza riga. Inoltre, la dimensione di `S2` non può essere 17, perché 17 non è multiplo di 8: dobbiamo sprecare ulteriori 7 byte in modo che **`sizeof`**`(S2)` diventi 24.

È importante ribadire che queste sono *regole* stabilite dall'ABI, non suggerimenti su come rappresentare le strutture: non seguire le regole comporta incompatiblità con il compilatore e con il sistema operativo. Se il programmatore è interessato a minimizzare lo spreco di byte, deve lui stesso pensare a definire le proprie strutture in maniera opportuna. Per esempio, la struttura `S3` di Figura 4(c) contiene le stesse informazioni della struttura `S2`, ma il suo layout in memoria è quello illustrato in Figura 7, in cui si vede che solo 2 byte sono andati sprecati.

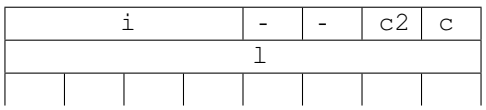

Figura 7: Layout di memoria della struttura `S3` di Figura 4(c).