

Calcolatori Elettronici: la memoria centrale

G. Lettieri

4 Marzo 2022

Vogliamo vedere come organizzare e collegare al bus una memoria che permetta l'accesso sia al byte che alle parole di 2, 4 o 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.

Dal momento che lavoriamo con parole di più byte, ci possiamo chiedere in che ordine i byte della parola 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 *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 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

¹I nomi *little endian* e *big endian* vengono indirettamente da *I viaggi di Gulliver* di Jonathan Swift, grazie a 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.

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 2^8 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 $A\{n-1\} = A\{n-2\} = \dots = A_{10} = 0$, $A_9 = 1$, $A_8 = A_7 = \dots = A_3 = 0$, in quanto 512 è 1000000000 in binario, e $/BE7 = /BE6 = \dots = /BE0 = 0$ (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 $A\{n-1\}$ – A_3 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 coinvolge una parola da 4 byte all'indirizzo 516: avremo $A\{n-1\}$ – A_3 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: ogni byte enable contribuisce a generare 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 di

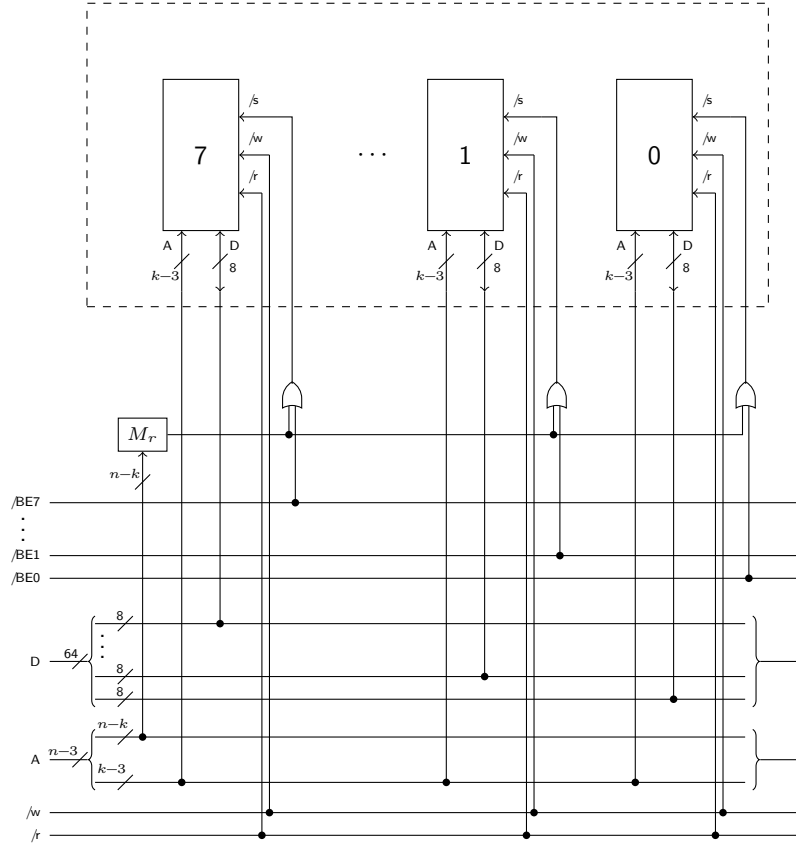


Figura 2: Realizzazione di una memoria organizzata a linee di 64 bit (parole quadruple), ma accessibile anche a byte, parole e doppie parole. Le parti dentro e fuori del rettangolo tratteggiato possono essere realizzate separatamente e poi collegate. La parte interna è una scheda di memoria, la parte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

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:

<pre>mov 4097, %rax</pre>

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 511 (4096/8) con tutti i byte enable attivi, tranne /BE0; un altro alla riga successiva (512), 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

numero di riga	+7	+6	+5	+4	+3	+2	+1	+0	indirizzo di riga
...									
510									4080
511	22	33	44	55	66	77	88		4096
512								11	4112
513									4128
...									

Figura 3: Accesso non allineato.

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. Possiamo dunque 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.