Indirizzi e oggetti

G. Lettieri

29 Febbraio 2024

Gli indirizzi sono relativi ad un bus: tutti i componenti collegati al bus, in grado di rispondere a richieste di lettura o scrittura, devono avere degli indirizzi assegnati in modo univoco. I possibili indirizzi vanno praticamente sempre da 0 fino ad un massimo della forma $2^n - 1$ per qualche n che dipende dal bus. Questo perché gli indirizzi viaggiano su un numero prefissato di piedini, fili o tracce, ciascuna delle quali può assumere solo i valori 0 o 1. Se queste tracce sono n, il numero totale di indirizzi possibili è dunque 2^n.

Gli indirizzi esistono sempre tutti, nel senso che è sempre possibile richiedere una lettura o una scrittura a qualunque indirizzo, anche se l'indirizzo non è assegnato a nessun componente (il risultato di una tale richiesta dipende dal bus; come abbiamo detto, assumiamo per ora che le scritture non abbiano effetto e le letture restituiscano un valore casuale).

Per questi motivi, il modo più naturale di rappresentare gli indirizzi di un bus è come la sequenza di tutti i numeri binari, senza segno, su n bit. Conviene acquisire familiarità con le rappresentazioni dei numeri in base 16 e 8, in quanto queste basi permettono di rappresentare gli indirizzi in forma molto più compatta e, allo stesso tempo, facilmente convertibile da e verso la base 2. Alcuni numeri molto frequenti devono subito richiamare alla mente la loro rappresentazione nelle varie basi: in base 2, 2^a è un 1 seguito da a zeri mentre $2^a - 1$ è composto da a 1. Se a è un multiplo di 4, allora 2^a in base 16 è 1 seguito da $a/4$ zeri, mentre $2^a - 1$ è rappresentato come $a/4$ cifre F (questo perché ogni cifra esadecimale corrisponde a 4 cifre binarie). Similmente per la base 8: se a è multiplo di 3, allora 2^a è 1 seguito a $a/3$ zeri e $2^a - 1$ è composto da $a/3$ cifre 7. In generale, conviene usare queste basi ogni volta che dobbiamo ragionare sulla struttura binaria di un qualche valore. È bene familiarizzarsi con i seguenti casi notevoli:

- un byte corrisponde a 2 cifre esadecimali;
- 512 corrisponde a 1000 in base 8;
- 4 Ki corrisponde a 1000 in base 16 e 1.0000 in base 8;
- 1 Mi corrisponde a 10.0000 in base 16;
- 4 Gi corrisponde a 1.0000.0000 in base 16.

Le operazioni sugli indirizzi (come aggiungere una costante a un indirizzo, o sottrarre due indirizzi) vanno sempre pensate come operazioni modulo 2^n. Gli indirizzi hanno un ordinamento naturale, in cui l'indirizzo successivo di un indirizzo x è $x + 1$ e il precedente è $x - 1$. Si noti però che, dal momento che le operazioni vanno intese modulo 2^n, l'indirizzo che precede l'indirizzo 0 è l'indirizzo $2^n - 1$ e l'indirizzo successivo a $2^n - 1$ è l'indirizzo zero.

1 Scostamenti (*offset*)

Dati due indirizzi x e y su n bit, possiamo chiederci quanto y *si discosta* da x calcolando il valore $y - x$ modulo 2^n, detto *offset* di y rispetto a x.

In ogni caso, l'offset conta il numero di indirizzi che è necessario "saltare", partendo da x, per raggiungere y nell'ordine naturale. In particolare, l'offset di x rispetto a se stesso è zero. Gli offset possono essere anche negativi: il loro valore assoluto rappresenta comunque il numero di indirizzi da saltare partendo da x per raggiungere y, ma andando nella direzione degli indirizzi descrescenti. Per esempio, l'offset -1 rappresenta l'indirizzo che precede x (nel caso $x = 0$ si ricordi che l'indirizzo precedente è $2^n - 1$).

1.1 Caso particolare: offset in complemento a 2 su n bit

Se rappresentiamo gli offset come numeri in complemento a 2 su n bit (cioè lo stesso numero di bit dello spazio di indirizzamento), diventa indifferente considerarli con o senza segno. Facciamo un esempio con $n = 4$. Gli indirizzi vanno dunque da 0 a 15. L'offset -1 si rappresenta come 1111 in complemento a 2 su 4 bit. Prendiamo $x = 3$. Se interpretiamo 1111 come numero con segno, l'offset è -1 e saltando un indirizzo all'indietro arrivamo a $y = 2$. Se ora interpretiamo l'offset 1111 come il numero senza segno 15, dobbiamo saltare 15 indirizzi in avanti partendo da $x = 3$. Dopo averne saltati 12 arrivamo all'indirizzo 15, saltandone un altro ripartiamo dall'indirizzo 0 e con gli ultimi due salti arriviamo a $y = 2$, come prima.

Se, però, l'offset è rappresentato su un numero di bit inferiore a n, diventa importante sapere se la rappresentazione è con segno o senza, in quanto l'offset va esteso a n bit prima di poterlo sommare all'indirizzo, e l'estensione va realizzata in modo diverso a seconda della rappresentazione. In particolare, gli offset di 4 byte (o un byte) che si trovano all'interno delle istruzioni AMD64 usano la rappresentazione in complemento a 2 *con* segno.

2 Intervalli (*range*)

Un *intervallo* è una sequenza di indirizzi. Ci sono vari modi di rappresentare un intervallo, e purtroppo nessuno è privo di difetti se siamo vincolati ad usare numeri rappresentati su n bit. Noi adotteremo la convenzione di rappresentare gli intervalli specificando il primo indirizzo che ne fa parte, sia x, e il primo,

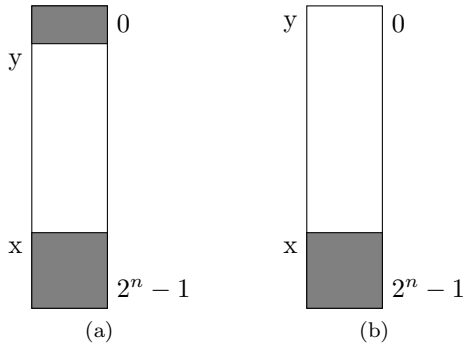

Figura 1: (a) Intervallo con wrap-around. (b) Intervallo che non fa wrap-around, ma comprende l'ultimo indirizzo.

successivo a x, che *non* ne fa parte, sia y. Diremo che x è la *base* dell'intervallo, mentre y ne è il *limite*.

Matematicamente, se x e y sono (per esempio) numeri naturali, un tale intervallo viene definito come

$$[x, y) = \{\, i \mid x \leq i < y \,\}. \tag{1}$$

Escludendo il caso $y < x$ (privo di interesse), questa definizione ha il pregio che $l = y - x$ è il numero di elementi dell'intervallo (*lunghezza*). Viceversa, se di un intervallo conosciamo la base x e la lunghezza l, l'intervallo è semplicemente $[x, x+l)$. Questo è un vantaggio rispetto ad una rappresentazione come $[x, y]$ che include il limite nell'intervallo e richiede di aggiungere 1 alla prima espressione e di sottrarre uno nella seconda. La rappresentazione $[x, y)$, viceversa, ha come difetto che l'ultimo elemento dell'intervallo (assumendo che non sia vuoto) è $y - 1$ e non y.

I problemi principali, però, derivano dal fatto che nel nostro caso x e y non sono numeri naturali, ma numeri rappresentati su n bit, con wrap-around (matematicamente, sono naturali modulo 2^n). Per fortuna le espressioni $l = y - x$ e $[x, x + l)$ valgono ancora, purché si intendano la somma e la sottrazione modulo 2^n. Purtroppo, però, la semplice definizione (1) fallisce con intervalli come quello in Figura 1(a). In questo caso, infatti, abbiamo $y < x$ e secondo la definizione (1) l'intervallo dovrebbe essere vuoto, quando chiaramente non lo è. Per non complicare la definizione[1] eviteremo sempre questo tipo di intervalli richiedendo che sia sempre $x \leq y$, ma ci resta un caso particolare da considerare: un intervallo come quello di Figura 1(b). Pur non facendo wrap-around, un intervallo del genere ha comunque $y = 0 < x$ e dunque è vuoto stando alla

[1] Una definizione che non ha problemi con gli intervalli di Figura 1 è

$$[x, y) = \{\, i \mid o(x, i) < o(x, y) \,\} \tag{2}$$

Dove $o(a, b)$ è $b - a$ modulo 2^n, cioè l'offset tra a e b.

definizione (1). Per ammettere anche questi ultimi intervalli accettiamo anche la forma $[x, 0)$ con $x \neq 0$ come valida, con l'accordo che in questo caso il limite è in realtà il numero naturale 2^n che non possiamo rappresentare.

3 Allineamenti

Diamo un po' di definizioni:

- si dice che un *indirizzo è allineato a* 2^b se è multiplo di 2^b;
- si dice che un *intervallo è allineato a* 2^b se la sua base è allineata a 2^b;
- se i è un intervallo la cui lunghezza è una potenza di 2, si dice che i è *allineato naturalmente* se è allineato alla sua lunghezza.

Un indirizzo allineato a 2^b ha i b bit meno significativi della propria rappresentazione binaria tutti nulli. Notare che se un indirizzo è allineato a 2^b, allora è anche allineato a $2^{b'}$ con $b' < b$ (se ha b bit meno significativi a zero, a maggior ragione ne ha $b' < b$ a zero). L'indirizzo 0 è allineato a 2^b per qualunque b.

4 Confini (*boundaries*) e regioni naturali

Dato un qualunque $b \leq n$, tutti gli indirizzi allineati a 2^b sono detti *confini* di 2^b. Tutti i confini di 2^b si trovano partendo da 0 e procedendo ad offset successivi di 2^b. Questi confini delimitano degli intervalli di indirizzi, allineati naturalmente, che chiamiamo *regioni naturali* di 2^b. Ometteremo l'agettivo "naturali" quando non si crea ambiguità. Useremo il termine regione (naturale) in modo generico, ma in molti casi introdurremo dei nomi particolari per regioni di particolare interesse (per esempio *riga*, *cacheline*, *pagina*, ...). Le regioni sono in totale 2^{n-b}, o 2^a se poniamo $a = n - b$ (Figura 2).

Possiamo assegnare ad ogni regione un numero progressivo r compreso tra 0 e $2^a - 1$. Il numero r, detto *numero di regione*, serve ad identificare univocamente ogni regione. Si noti che tutti (e soli) gli indirizzi che fanno parte della stessa regione contengono negli a bit più significativi proprio il numero della regione a cui apparengono. Invece, i b bit meno significati contengono l'offset dell'indirizzo rispetto alla base della regione (il confine), detto *offset all'interno della regione*. Ogni indirizzo è dunque identificato dal numero di regione e dal suo offset (all'interno della regione). Ovviamente, lo stesso indirizzo può essere identificato in modi diversi in base alla dimensione della regione scelta.

Dato un numero b e un indirizzo x su n bit è dunque immediato, in hardware, trovare il suo numero di regione (di 2^b) e il suo offset: gli $a = n - b$ bit più significativi x danno il numero di regione, e i b bit meno significativi danno l'offset. Supponiamo ora di voler fare la stessa cosa in software, supponendo di avere una variabile `unsigned long x` che contiene un indirizzo, e di conoscere b. Prendiamo come n la dimensione in bit di un `unsigned long`, che nel nostro caso è 64. Per ottenere il numero di regione possiamo scrivere semplicemente

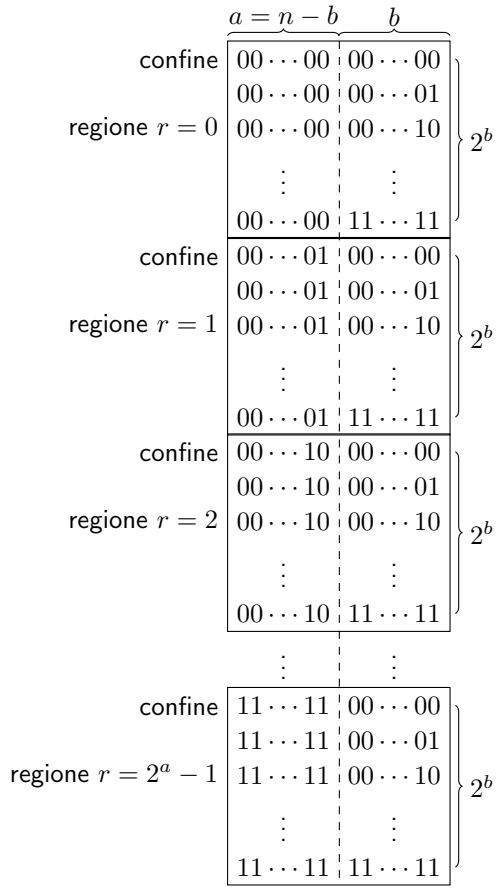

Figura 2: Scomposizione degli indirizzi in regioni naturali di 2^b.

```
r = x >> b; // numero di regione
```

Per ottenere l'offset usiamo l'AND bit a bit con una maschera che ha b cifre meno significative ad 1 e tutte le altre a zero:

```
o = x & ((1UL << b) - 1); // offset
```

I caratteri `UL` che seguono la costante 1 servono a specificarne il tipo come Unsigned Long. Si ricordi che per default le costanti numeriche del C++ hanno tipo **`int`** se, come in questo caso, sono rappresentabili su un intero.

Vediamo anche come si ottiene la base della regione a cui x appartiene. Possiamo ottenere questo indirizzo con una maschera che ha a bit significativi a 1 e gli altri a 0. Questa maschera non è altro che il NOT della maschera che abbiamo usato prima:

```
c = x & ~((1UL << b) - 1); // confine
```

Dato un intervallo non vuoto $[x, y)$, vogliamo spesso sapere quali sono la prima regione toccata dall'intervallo e la prima regione *non toccata* dall'intervallo (con questo intendiamo la prima regione che si incontra dopo la prima toccata e che non contiene indirizzi dell'intervallo).

La prima regione toccata è semplicemente la regione in cui cade x, ma per la prima regione non toccata bisogna fare più attenzione. Si noti che questa non è sempre la regione successiva a quella in cui si trova y, perché y non fa parte dell'intervallo $[x, y)$: se y cade su un confine, la regione in cui si trova y è lei stessa la prima non toccata. Possiamo operare nel modo seguente:

```
nt = (y + (1UL << b) - 1) >> b.
```

In altre parole, sommiamo prima a y il valore $2^b - 1$: se y cade su un confine, $y+2^b-1$ è ancora nella stessa regione di y; se y non cade su un confine, $y+2^b-1$ si trova nella regione successiva. Un altro metodo consiste nel sommare 1 alla regione in cui cade $y - 1$:

```
nt = ((y - 1) >> b)) + 1,
```

ma bisogna stare attenti al wrap-around quando $y = 0$: il numero di regione deve stare su $n - b$ bit, quindi `nt` andrebbe poi mascherato con una maschera di $n - b$ bit a 1.