11.4 Il trattamento dei bit

Per scrivere applicazioni che controllano dispositivi hardware è necessario disporre di operatori in grado di lavorare sui singoli bit di un registro. Se per esempio si ha una centralina di controllo di un impianto di illuminazione, dove un registro di 16 bit comanda lo stato di 16 lampade, mettere a 1 un bit del registro significa accendere una lampada se questa è spenta, e metterlo a 0 significa spegnerla se questa è accesa. Il C fornisce un ricco insieme di operatori per il trattamento dei bit:

`&`	AND bit a bit
`|`	OR bit a bit
`^`	OR esclusivo
`<<`	shift sinistra
`>>`	shift destra
`~`	complemento a 1

Gli operatori per il trattamento dei bit – si veda più avanti in Figura 11.1 la tavola di priorità complessiva – sono gli stessi disponibili in tutti i linguaggi assembler, e rendono il C quello che alcuni hanno definito “il linguaggio di più alto livello tra quelli a basso livello”.

Analizziamo ora uno per uno gli operatori servendoci di esempi. Come premessa occorre precisare che questi operatori non lavorano su variabili di tipo `float` o `double`, ma su variabili di tipo `char` e `int`. In effetti, pensando agli elementi di memoria di un qualsiasi dispositivo hardware (una centralina, una scheda di rete, il controller di un disco) i valori in virgola mobile non hanno alcun significato: i bit sono bit di comando e le uniche operazioni che servono sono quelle di abilitazione/disabilitazione. In altre parole, sui bit di un registro non c'è bisogno di compiere operazioni aritmetiche: i registri sono semplicemente una collezione di bit cui si attribuisce un preciso significato. Se ne deduce, allora, che le operazioni per il trattamento dei bit, con l'eccezione delle operazioni di shift e complemento, si applicheranno tipicamente a variabili di tipo `unsigned char` e `unsigned int`, poiché il segno aritmetico non ha significato.

L'istruzione di AND bit a bit è usata per *spegnere* uno o più bit di una variabile. Infatti, nel confronto bit a bit dell'operazione booleana AND basta avere uno dei due operandi a 0 per produrre 0.

b1	b2	b1 & b2
0	0	0
1	0	0
0	1	0
1	1	1

Se per esempio volessimo mettere a 0 i quattro bit meno significativi di una variabile `x` di tipo `unsigned char`, e lasciare inalterato la stato di quattro bit più significativi, basterebbe mettere in AND `x` con la sequenza:

```
1 1 1 1 0 0 0 0
```

il cui valore ottale è 360. In pratica si tratta di formare una sequenza di bit dove sia posto il valore 1 in corrispondenza dei bit di `x` che devono rimanere invariati e il valore 0 in corrispondenza dei bit di `x` che devono essere trasformati in zero. Se lo stato iniziale dei bit di `x` fosse

```
1 0 1 0 1 0 1 0
```

dopo l'istruzione

```
x = x & '\360';
```

lo stato dei bit di `x` diventerebbe:

```
1 0 1 0 0 0 0 0
```

L'istruzione di OR bit a bit è usata per *accendere* uno o più bit di una variabile. Infatti, nel confronto bit a bit dell'operazione booleana OR basta avere uno dei due operandi a 1 per produrre 1.

b1	b2	b1 | b2
0	0	0
1	0	1
0	1	1
1	1	1

Se per esempio volessimo mettere a 1 i quattro bit meno significativi di una variabile `unsigned char`, e lasciare inalterato la stato di quattro bit più significativi, basterebbe usare l'istruzione:

```
x = x | '\017';
```

Se prima dell'istruzione lo stato dei bit di `x` fosse

```
1 0 1 0 1 0 1 0
```

corrispondendo la costante `'\017'` alla sequenza

```
0 0 0 0 1 1 1 1
```

dopo l'istruzione

```
x = x | '\017';
```

lo stato dei bit di `x` sarebbe:

```
1 0 1 0 1 1 1 1
```

L'istruzione di OR esclusivo bit a bit è usata per *commutare* da 0 a 1 e viceversa un insieme di bit di una variabile. Infatti, nel confronto bit a bit dell'operazione booleana OR esclusivo se i due bit sono uguali si produce 0, se sono diversi si produce 1.

b1	b2	b1 ^ b2
0	0	0
1	0	1
0	1	1
1	1	0

Se per esempio volessimo scambiare 0 con 1 e 1 con 0 nei primi quattro bit più significativi di una variabile `unsigned char`, e lasciare inalterato lo stato dei quattro bit meno significativi, basterebbe usare l'istruzione:

```
x = x ^ '\360';
```

Se prima dell'istruzione lo stato dei bit di `x` fosse

```
1 0 1 0 1 0 1 0
```

corrispondendo la costante `'\360'` alla sequenza

```
1 1 1 1 0 0 0 0
```

dopo l'istruzione

```
x = x ^ '\360';
```

lo stato dei bit di x sarebbe:

```
0 1 0 1 1 0 1 0
```

Si osservi che effettuare l'OR esclusivo di una variabile con se stessa ha l'effetto di mettere a 0 la variabile. L'istruzione

```
x = x ^ x;
```

trasforma sempre `x` in:

```
0 0 0 0 0 0 0 0
```

Il lettore potrebbe obiettare che lo stesso risultato si sarebbe potuto ottenere con l'assegnazione:

```
x = 0;
```

Il fatto è che la pratica di azzerare una variabile facendo l'OR esclusivo con se stessa deriva dall'assembler, dove trova frequente applicazione. Se poi si vogliono scambiare tutti i bit di una variabile da 0 a 1 e viceversa, basta effettuare il complemento a 1 della variabile stessa:

```
x = ~x;
```

Se prima dell'istruzione lo stato dei bit di `x` fosse

```
1 0 1 0 1 0 1 0
```

dopo l'istruzione

```
x = ~x;
```

lo stato dei bit di `x` sarebbe:

```
0 1 0 1 0 1 0 1
```

Le operazioni di shift << e >> traslano il primo operando, rispettivamente a sinistra e a destra, di un numero di posizioni corrispondenti al valore del secondo operando. L'effetto delle operazioni di shift cambia a seconda che le si applichi a variabili senza segno o con segno; per quelle con segno il risultato può cambiare a seconda del tipo di macchina. In generale valgono le seguenti regole:

- se si esegue lo shift a destra di una variabile `unsigned` i bit vacanti a sinistra sono rimpiazzati con degli 0;
- se si esegue lo shift a destra di una variabile con segno i bit vacanti a sinistra sono rimpiazzati con il bit del segno su certe macchine (shift aritmetico) e con 0 su altre (shift logico);
- se si esegue lo shift a sinistra di una variabile `unsigned`, i bit vacanti a destra sono rimpiazzati con degli 0;
- se si esegue lo shift a sinistra di una variabile con segno, i bit vacanti a destra sono rimpiazzati con degli 0 e il bit del segno rimane inalterato.

Vediamo ora alcuni esempi di operazioni di shift.

1)
```
char c;
c = c << 1;
```

Se prima dello shift i bit di `c` fossero stati

```
1 0 0 0 1 0 0 1
```

dopo lo shift avremmo avuto:

```
1 0 0 1 0 0 1 0
```

Avremmo cioè mantenuto a 1 il primo bit di segno.

2)
```
char c;
c = c >> 1;
```

Se prima dello shift i bit di `c` fossero stati

```
1 0 0 0 1 0 0 1
```

dopo lo shift avremmo avuto

```
1 1 0 0 0 1 0 0
```

Il bit vacante a sinistra sarebbe cioè rimpiazzato dal bit del segno.

3)
```
unsigned char c;
c = c >> 1;
```

Se prima dello shift i bit di `c` fossero stati

```
1 0 0 0 1 0 0 1
```

dopo lo shift avremmo avuto

```
0 1 0 0 0 1 0 0
```

Il bit vacante a sinistra sarebbe cioè rimpiazzato dal bit 0.

Si noti come nello shift a sinistra i bit vacanti di destra siano sempre rimpiazzati da 0. Ciò equivale a moltiplicare per multipli di 2. Così, per esempio, l'istruzione:

```
x = x << 2;
```

produce su `x` l'effetto di una moltiplicazione per 4.
Le operazioni di shift sono spesso usate congiuntamente alle operazioni booleane bit a bit per controllare lo stato di uno o più bit di una variabile. Per sapere, per esempio, se il quarto bit a partire da destra della variabile `c` è 0 oppure 1 potremmo procedere così:

```
if( (x >> 3) & 01 )
   printf("In quarto bit di x è 1");
else
   printf("In quarto bit di x è 0");
```

Nell'esempio abbiamo raccolto `x>>3` all'interno di parentesi tonde: infatti avendo l'operatore `>>` minore priorità di `&`, in caso contrario sarebbe stato effettuato all'interno dell'espressione prima l'AND bit a bit tra 3 e 01 e poi lo shift a destra di `x`.

È il momento di ampliare la tavola gerarchica delle priorità degli operatori (Figura 11.1), dove inseriamo anche la trasformazione di tipo indicata nella tabella con `(tipo)`, che studieremo approfonditamente nel prossimo paragrafo. Si tenga conto che per gli operatori binari per il trattamento dei bit, come per quelli aritmetici, è possibile utilizzare la notazione:

```
variabile [operatore]= espressione
```

quando una variabile appare sia a sinistra sia a destra di un operatore di assegnamento. Pertanto l'istruzione:

```
x = x << 2;
```

può essere scritta come

```
x <<= 2;
```

```
        !    -       ++      --      sizeof      (tipo)

                        *       /       %

                              +       -

                            >>      <<

                         >      >=     <     <=

                                ==     !=

                                    &

                                    ^

                                    |

                                   &&

                                   ||

                                   ?:

        =   +=   -=   *=   /=   %=   &=   ^=   |=   <<=   <<=

                                   ,
```

Figura 11.1 Tavola di priorità degli operatori