Codifica dei numeri negativi

Rappresentazione in complemento a 2

Per rappresentare numeri interi negativi si usa la cosiddetta *rappresentazione in complemento a 2*. Ad esempio, supponiamo di avere a disposizione *n* bit. Se vogliamo rappresentare numeri interi senza segno, sappiamo che possiamo rappresentare numeri nell'intervallo $[0,2^n-1]$. Se invece vogliamo rappresentare anche numeri negativi, allora le configurazioni che hanno il bit più significativo uguale a zero, cioè $[0,2^{n-1}-1]$, rappresentano se stesse, mentre le configurazioni col bit più significativo uguale a uno, cioè $[2^{n-1},2^n-1]$, rappresentano i numeri negativi che si ottengono traslando a sinistra l'intervallo di 2^n, cioè l'intervallo $[-2^{n-1},-1]$. Per questo, nella rappresentazione in complemento a 2, il bit più significativo viene chiamato *bit di segno*.

Con 8 bit, ad esempio, possiamo rappresentare i numeri naturali nell'intervallo $[0,2^8-1]$, cioè [0,255], oppure i numeri relativi nell'intervallo $[-2^7,2^7-1]$, cioè [-128,127]. Con 16 bit (2 byte) possiamo rappresentare i numeri naturali nell'intervallo $[0,2^{16}-1]$, cioè [0,65535], oppure i numeri relativi nell'intervallo $[-2^{15},2^{15}-1]$, cioè [-32768,32767].

Per ottenere la rappresentazione in complemento a 2 di un numero negativo:

"si parte dalla rappresentazione binaria del valore assoluto (che avrà il bit di segno = 0) e si prende il complemento a 1 di ciascun bit, quindi si aggiunge 1 al risultato".

Es. (si supponga una parola di 8 bit):

$$27_{10} = 00011011_2$$

$$\begin{aligned} \text{complemento a 1} \quad &: \quad 11100100 \\ &\qquad\qquad\quad + 1 \\ &\qquad\qquad\quad \text{----------} \\ -27_{10} \quad &= \quad 11100101_2 \end{aligned}$$

Viceversa, se abbiamo una sequenza di 8 bit e sappiamo che essa rappresenta un numero intero con segno, con i numeri negativi rappresentati in complemento a 2, allora, per ottenere il numero rappresentato, cominciamo con l'esaminare il bit di segno. Se esso è zero, il numero rappresentato è non negativo e lo otteniamo con la normale conversione binario-decimale. Se invece il bit di segno è uno, allora sappiamo che si tratta di un numero negativo. Per ottenere il modulo del numero applichiamo l'algoritmo di sopra, cioè complementiamo tutti i bit e sommiamo 1 al risultato.

Per esempio, se il numero binario 11100101 è la rappresentazione in complemento a 2 di un numero, il valore assoluto del numero rappresentato si ottiene così:

$$\begin{aligned} \text{complemento a 1} \quad &: \quad 00011010 \\ &\qquad\qquad\quad + 1 \\ &\qquad\qquad\quad \text{----------} \\ 27_{10} \quad &= \quad 00011011_2 \end{aligned}$$

Per una parola di *n* bit, i numeri N rappresentabili in complemento a 2 sono tali per cui

$$-2^{n-1} <= N <= 2^{n-1}-1$$

Per parole di 16 bit si ha:

$$-2^{15} <= N <= 2^{15}-1$$

cioè

$$-32768 <= N <= 32767$$

Configurazioni binarie (4 bit)	Numero rappresentato (senza segno)	Traslazione	Numero rappresentato (con segno)
0000	0		0
0001	1		1
0010	2		2
0011	3		3
0100	4		4
0101	5		5
0110	6		6
0111	7		7
1000	8	-16	-8
1001	9	-16	-7
1010	10	-16	-6
1011	11	-16	-5
1100	12	-16	-4
1101	13	-16	-3
1110	14	-16	-2
1111	15	-16	-1

Operazioni di somma e sottrazione nella rappresentazione in complemento a 2

Supponiamo di lavorare con 4 bit: i numeri rappresentabili sono gli interi nell'intervallo [-8,7].

Somma di due numeri positivi:

Es. 1

2_{10} : 0010_2 +
4_{10} : 0100_2 =

6_{10} : 0110_2

Es. 2

5_{10} : 0101_2 +
4_{10} : 0100_2 =

-7_{10} : 1001_2 (overflow)

Somma di due numeri negativi:

Es. 1

-2_{10} : 1110_2 +
-4_{10} : 1100_2 =

-6_{10} : [1]1010_2

Es. 2

-5_{10} : 1011_2 +
-4_{10} : 1100_2 =

7_{10} : [1]0111_2 (overflow)

Somma di due numeri di segno opposto:

Es. 1

$+2_{10}$: 0010_2 +
-4_{10} : 1100_2 =

-2_{10} : 1110_2

Es. 2

-5_{10} : 1011_2 +
$+7_{10}$: 0111_2 =

$+2_{10}$: [1]0010_2

Rappresentazione dei numeri reali

Il numero decimale 341.801 equivale a

$$3\times10^{2}+4\times10^{1}+1\times10^{0}+8\times10^{-1}+0\times10^{-2}+1\times10^{-3}$$

Analogamente il numero binario 101.011 equivale a

$$1\times2^{2}+0\times2^{1}+1\times2^{0}+0\times2^{-1}+1\times2^{-2}+1\times2^{-3}$$

Conversione in binario di un numero reale

Si convertono separatamente la parte intera e la parte frazionaria. Per la parte intera si può applicare l'algoritmo di conversione visto in precedenza. Per la parte frazionaria si moltiplica per 2 e si toglie la parte intera del risultato, che diventa la prima cifra dopo il punto. Si procede allo stesso modo per le successive cifre, finchè la parte frazionaria non si annulla o finchè non abbiamo ottenuto un numero sufficiente di cifre binarie.

Esempio: convertire in binario il numero 5.375_{10}

La parte intera è 101_2
Rimane da convertire la parte frazionaria 0.375

0.375	× 2	=	0.750	la parte intera (0) diventa la prima cifra binaria dopo il punto
0.750	× 2	=	1.5	la parte intera (1) diventa la seconda cifra binaria dopo il punto
0.5	× 2	=	1.0	la parte intera (1) diventa la terza cifra binaria dopo il punto

Risultato: $5.375_{10} = 101.011_2 = 4 + 1 + 1/4 + 1/8$

Rappresentazione normalizzata

Il numero (base 10)

$$341.801$$

può essere rappresentato in forma normalizzata come

$$3.41801 \text{x} 10^2$$

dove 3.41801 è la *mantissa* e 2 è l'*esponente*.

Analogamente il numero binario 101.011 può essere rappresentato come

$$1.01011 \times 2^2$$

Tenendo presente che la mantissa di un numero binario normalizzato (diverso da 0) comincia sempre con la cifra 1 seguita dal punto, per rappresentare il numero in memoria è sufficiente inserire le cifre che nella mantissa seguono il punto, più l'esponente e il segno.

Standard IEEE per la rappresentazione di numeri reali in singola precisione

Questo standard è utilizzato ad esempio per rappresentare il tipo *float* del C++, che occupa 4 byte (32 bit): 1 bit per il segno, 8 bit per l'esponente e 23 bit per la mantissa.

S (1 bit)	E (8 bit)	M (23 bit)

Il bit di segno (S) è 0 per i numeri positivi e 1 per i numeri negativi.

Gli 8 bit successivi (E) rappresentano l'esponente in *codice in eccesso-127*, cioè si ottengono sommando all'esponente effettivo il *bias* \$7F = $01111111_2 = 127_{10}$. L'esponente effettivo può dunque andare da -127 a +128.

I rimanenti 23 bit rappresentano i bit che seguono 1. nella mantissa.

Il numero 0.0 è rappresentato da 4 byte nulli. Il numero -0.0 ha il bit di segno uguale a 1 e i rimanenti 31 bit uguali a 0.

In memoria i 4 byte vengono messi nell'ordine basso-alto, cioè il byte all'indirizzo più basso contiene gli ultimi 8 bit della mantissa, mentre il byte all'indirizzo più alto contiene il bit di segno seguito dai primi 7 bit dell'esponente.

Tipo *float* del Turbo C++
Esempi di rappresentazione interna

	Byte offset: 3	2	1	0
1.0	3F	80	00	00
-1.0	BF	80	00	00
2.0	40	00	00	00
-2.0	C0	00	00	00
3.0	40	40	00	00
-3.0	C0	40	00	00
4.0	40	80	00	00
-4.0	C0	80	00	00
0.5	3F	00	00	00
-0.5	BF	00	00	00
0.75	3F	40	00	00
-0.75	BF	40	00	00
100.25	42	C8	80	00
-100.25	C2	C8	80	00
25.125	41	C9	00	00
-25.125	C1	C9	00	00
0.0	00	00	00	00
-0.0	80	00	00	00