

Codifica binaria

October 19, 2018

Rappresentazione dei numeri nel calcolatore

Introduzione

- La **rappresentazione numerica** consiste nell'utilizzare una sequenza di n simboli b_i (chiamati **cifre** o caratteri) per specificare un **numero**.
- Useremo la notazione

$$\llbracket b_{n-1} \dots b_0 \rrbracket_B$$

per indicare il valore di tale sequenza se interpretiamo la serie di cifre b_i in base B .

- Se $B = 10$ ometteremo sia le parentesi che la base, ad esempio $\llbracket 5 \rrbracket_{10} \equiv 5$.

- Un sistema di rappresentazione numerica può essere *posizionale* o *non posizionale*.
- Nel primo caso (tra cui troviamo il sistema decimale), ogni cifra assume un peso diverso a seconda della posizione in cui è scritta. Nel secondo no.

La rappresentazione unaria è un esempio di rappresentazione **non posizionale**:

Rappresentazione unaria	Rappresentazione decimale
I	1
II	2
III	3
IIII	4
IIII	5
IIII I	6

Rappresentazione decimale

- Nelle notazioni posizionali, distinguiamo la base e le cifre.
- Nella rappresentazione decimale, la base è il numero 10, e le cifre possono essere da 0 a 9:

$$\llbracket b_{n-1} \dots b_0 \rrbracket_{10} = b_{n-1}10^{n-1} + \dots + b_010^0 = \sum_{i=0}^{N-1} b_i \cdot 10^i.$$

dove i pesi sono calcolati come potenze della base, in questo caso dieci.

Rappresentazione in base arbitraria

E' possibile usare una qualsiasi base K (e.g., K=8 oppure K=16)

$$\llbracket b_{n-1} \dots b_0 \rrbracket_K = b_{n-1}K^{n-1} + \dots + b_0K^0 = \sum_{i=0}^{n-1} b_i \cdot K^i.$$

esempio

$$\llbracket 10 \rrbracket_{16} = 1 * 16 + 0 * 1 = 16$$

$$\llbracket 24 \rrbracket_8 = 2 * 8 + 4 = 20$$

Rappresentazione esadecimale (K=16)

Codifica molto frequente nelle applicazioni informatiche; le cifre di base sono 16:

$$b_i \in \{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$$

ove, $\llbracket A \rrbracket_{16} = 10 \dots \llbracket F \rrbracket_{16} = 15$. Ad esempio:

$$\llbracket 1A \rrbracket_{16} = 1 * 16 + 10 * 1 = 26$$

Codifica binaria naturale

Rappresentazione binaria

- Nella rappresentazione binaria naturale, la base è il numero 2 e le cifre (bit) possono essere 0 oppure 1:

$$\llbracket b_{n-1} \dots b_0 \rrbracket_2 = b_{n-1}2^{n-1} + \dots + b_02^0 = \sum_{i=0}^{n-1} b_i \cdot 2^i.$$

dove n è il numero di bit della cifra binaria.

- Ad esempio, avremmo

$$\llbracket 101101 \rrbracket_2 = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 45$$

Conversione in codifica binaria - Metodo dei resti

Sia 44 il valore che vogliamo convertire in binario. Il metodo dei resti permette di calcolare la codifica binaria dividendo iterativamente per due e considerando la colonna dei resti dal basso verso l'alto.

Valore	Divisone per due	Risultato	Resto	nota
44	44/2	22	0	cifra più a destra
22	22/2	11	0	
11	11/2	5	1	
5	5/2	2	1	
2	2/2	1	0	
1	1/2	0	1	cifra più a sinistra

Quindi $\llbracket 101100 \rrbracket_2 = 44$.

$$2^{24} = 2^{4+20} = 16 \text{ Mega} \quad (\text{leggi "16 milioni"})$$

$$2^{35} = 2^{5+30} = 32 \text{ Giga} \quad (\text{leggi "32 miliardi"})$$

$$2^{48} = 2^{8+40} = 256 \text{ Tera} \quad (\text{leggi "256 bilioni"})$$

Cambio del numero dei bit

- premettendo in modo progressivo un bit 0 a sinistra, il valore del numero non muta

$$\llbracket 100 \rrbracket_2 = \llbracket 0100 \rrbracket_2 = \llbracket 00100 \rrbracket_2 = \dots \llbracket 000000000100 \rrbracket_2 = 4$$

- cancellando in modo progressivo un bit 0 a sinistra, il valore del numero non muta, ma bisogna arrestarsi quando si trova un bit 1!

$$7 = \llbracket 00111 \rrbracket_2 = \llbracket 0111 \rrbracket_2 = \llbracket 111 \rrbracket_2 - - - STOP!$$

Somma di numeri binari

La somma si esegue colonna per colonna come per le somme decimali. Attenzione che si ha un riporto già per $1 + 1 = 10$!

1	riporto	1	-	-	-
2	4 (decimale)	(0)	1	0	0
3	7 (decimale)	(0)	1	1	1
4		-----			
5	11 (decimale)	1	0	1	1

Se avessi avuto a disposizione solo 3 bit per codificare il numero, questo sarebbe stato un errore di **overflow**.

Codifica binaria dei numeri con segno

Codifica modulo e segno

- il primo bit a sinistra rappresenta il segno del numero (bit di segno),
- i bit rimanenti rappresentano il valore:
 - 0 per il segno positivo
 - 1 per il segno negativo
- esempi con $n = 9$ (8 bit + 1 bit per il segno)
 - $\llbracket 000000000 \rrbracket_{MS} = 0 \$$
 - $\llbracket 000001000 \rrbracket_{MS} = + 1 \times 2^3 = 8 \$$
 - $\llbracket 100001000 \rrbracket_{MS} = - 1 \times 2^3 = -8 \$$

Svantaggi del modulo e segno

- Il bit di segno è applicato al numero rappresentato, ma non fa propriamente parte del numero in quanto tale
- E' inefficiente: due codifiche per lo 0 ed è complesso da implementare

Codifica complemento a due

- Il C_2 è un sistema di rappresentazione simile al sistema binario naturale, ma il primo bit (quello a sinistra, il più significativo) ha peso negativo.
- Il bit più a sinistra è ancora chiamato bit di segno ma questa volta ha un significato numerico poiché il valore è calcolato tramite una espressione che lo comprende:

$$-1 \cdot b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \dots + b_02^0 = -1 \cdot b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i.$$

Esempi di numeri in complemento a due

valore C2	formula per il calcolo del valore	equiv. decimale
$[\![000]\!]_{C2}$	$-0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$	+0
$[\![001]\!]_{C2}$	$-0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$	+1
$[\![010]\!]_{C2}$	$-0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$	+2
$[\![011]\!]_{C2}$	$-0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$	+3
$[\![100]\!]_{C2}$	$-1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$	-4
$[\![101]\!]_{C2}$	$-1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$	-3
$[\![110]\!]_{C2}$	$-1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$	-2
$[\![111]\!]_{C2}$	$-1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$	-1

Come si ottiene un numero in complemento a due

- L'inverso additivo (o opposto) di un numero rappresentato in C2 si ottiene:
 - Invertendo (negando) ogni bit del numero
 - Sommando 1 alla posizione meno significativa

Esplicitando la base di ciascuna rappresentazione, calcolare il valore in C2 di $\llbracket -11 \rrbracket_{10}$:

- Parto da $\llbracket 11 \rrbracket_{10}$ in complemento a 2: $\llbracket 01011 \rrbracket_{C2}$
- Inverto ciascun bit: $\llbracket 10100 \rrbracket_{C2}$
- Aggiungo 1: $\llbracket 10100 \rrbracket_{C2} + \llbracket 1 \rrbracket_{C2} = \llbracket 10101 \rrbracket_{C2} = \llbracket -11 \rrbracket_{10}$

Confronto rappresentazioni:

Dec	M & S	C2
127	01111111	01111111
126	01111110	01111110
...
2	00000010	00000010
1	00000001	00000001
+0	00000000	00000000
0	10000000	-
-1	10000001	11111111
-2	10000010	11111110
...
-126	11111110	10000010
-127	11111111	10000001
-128	-	10000000

- Binario naturale (n bit): $[0, 2^n)$
- Modulo e segno (n bit): $(-2^{n-1}, 2^{n-1})$
- Complemento a 2 (n bit): $[-2^{n-1}, 2^{n-1})$

Addizione e sottrazione di interi

Addizione di numeri naturali - no overflow

1	riporto			1	1	1				
2	77 (decimale)	0	1	0	0	1	1	0	1	+
3	156 (decimale)	1	0	0	1	1	1	0	0	=
4										
5	233 (decimale)	1	1	1	0	1	0	0	1	

Addizione di numeri naturali - con overflow

```
1          riporto
2          perduto
3
4          riporto      1  1  1  1  1  1
5  125 (decimale)      0  1  1  1  1  1  0  1  +
6  156 (decimale)      1  0  0  1  1  1  0  0  =
7
8   25 (decimale)  (1) 0  0  0  1  1  0  0  1
9
10          avrei bisogno di
11          un altro bit..
```

Addizione di numeri in C2

```
1      riporto          1  1  1
2      77 (decimale)    0  1  0  0  1  1  0  1  +
3     -100 (decimale)   1  0  0  1  1  1  0  0  =
4
5     -23 (decimale)   .  1  1  1  0  1  0  0  1
6
7           non ho un
8           riporto perduto
```

Addizione di numeri in C2 con overflow

1	riporto	1	1	1	1					
2	77 (decimale)	0	1	0	0	1	1	0	1	+
3	92 (decimale)	0	1	0	1	1	1	0	0	=
4										
5	-87 (decimale)	.	1	0	1	0	1	0	0	1
6										
7	risultato	nessun riporto								
8	errato	perduto								

- Si può avere overflow senza “riporto perduto”
 - Capita quando da due addendi positivi otteniamo un risultato negativo, come nell’esempio precedente
- Si può avere un “riporto perduto” senza overflow
 - Può essere un innocuo effetto collaterale
 - Capita quando due addendi discordi generano un risultato positivo (si provi a sommare +12 e -7)
 - Se gli addendi sono discordi, non si ha mai overflow