Rappresentazione dei dati nell'elaboratore

Fulvio Ferroni *fulvioferroni#teletu.it*

2010.08.31

Indice generale

Introduzione

Una possibile definizione di informatica è la seguente:

«scienza della rappresentazione e dell'elaborazione (e archiviazione e trasmissione a distanza) delle informazioni».

Da questa definizione, e da altre simili, emerge l'importanza del concetto di informazione e delle modalità con cui essa viene rappresentata, per poi essere elaborata, nel sistema di elaborazione.

1.1 Dati e informazioni

Questi due termini vengono spesso usati come sinonimi (anche nel seguito di queste dispense) ma tra essi esiste una sottile differenze che deve comunque essere chiarita:

una informazione è qualcosa che permette di accrescere la conoscenza su un qualche fatto, oggetto o più in generale su una qualche entità anche astratta;

un dato è la parte costitutiva di una informazione ma da solo non permette di accrescere la conoscenza di alcunché.

Si può anche dire che una informazione è costituita da un dato (o da più dati) accompagnato dal suo significato.

Una ulteriore osservazione che permette di differenziare i due oggetti è la seguente:

una informazione viene memorizzata nella memoria umana, un dato viene memorizzato nelle memorie artificiali del sistema di elaborazione.

1.2 Misura dell'informazione

L'informazione richiede un supporto fisico per essere rappresentata, trasmessa, archiviata; essa però non coincide con il supporto, non è un'entità fisica ma logica e si può creare e distruggere (contrariamente al supporto fisico che è soggetto al ben noto principio di conservazione della materia-energia).

Il supporto fisico deve poter assumere configurazioni diverse per essere in grado di rappresentare informazioni; se infatti se la configurazione fosse sempre la stessa, non avremmo alcuna informazione dalla sua osservazione.

In termini più formali possiamo dire che: tanto minore è la probabilità che si presenti una configurazione, tanto maggiore è l'informazione che essa porta (quindi una configurazione certa, cioè con probabilità 1, non porta alcuna informazione).

Il concetto può essere espresso anche con una formula che permette di ottenere la misura dell'informazione veicolata da un supporto o da un messaggio che prevede N configurazioni equiprobabili con probabilità P = 1/N (Shannon 1948):

$$I = \log_2\left(\frac{1}{P}\right) = \log_2\left(\frac{1}{\left(\frac{1}{N}\right)}\right) = \log_2(N)$$

La formula conferma che se la probabilità della configurazione è 1 la quantità di informazione è nulla.

Nel caso del sistema di elaborazione le informazioni sono memorizzate su supporti in grado di assumere due configurazioni con probabilità 1/2 ciascuna; in questo caso quindi l'unità di misura dell'informazione è:

$$I = \log_2(2) = 1$$

Questa unità di misura prende il nome di *bit*, parola derivata da *binary digit*, cioè cifra binaria, (che può valere 0 oppure 1).

Se consideriamo il sistema di numerazione decimale possiamo calcolare la quantità di informazione di ogni cifra (che ha probabilità 1/10):

$$I = \log_2(10) = 3,32$$

e concludere quindi che per rappresentare una cifra decimale occorre un numero di bit pari a 4 (intero superiore a 3.32).

Se consideriamo invece un numero composto da 3 cifre decimali vediamo che occorrono 10 bit (3.32 + 3.32 + 3.32).

Questi aspetti, relativi all'uso delle cifre binarie per rappresentare i valori decimali, vengono approfonditi nel prossimo capitolo.

Capitolo 2

I sistemi di numerazione

Il sistema di numerazione che usiamo comunemente è quello decimale, o in base 10, costituito dalle cifre da 0 a 9 e portato nel mondo occidentale durante il medioevo dai matematici arabi.

La proprietà fondamentale di questo sistema di numerazione e degli altri che ci accingiamo ad esaminare (binario, ottale, esadecimale) è quella di essere «posizionali»:

all'interno di un numero, ogni cifra assume un valore che dipende dal suo valore intrinseco e dalla sua posizione nel numero stesso.

Tutti infatti sappiamo che nel numero (decimale) 153, il 5 è nella posizione delle decine e vale cinquanta mentre nel numero 175 è nella posizione delle unità e vale cinque.

Connessa alla nozione di posizione di una cifra è quella di «peso»; con tale termine si indica un valore ottenuto facendo la potenza che ha per base la base del sistema di numerazione e per esponente il numero di cifre che, nel numero, stanno alla destra della cifra considerata.

Applicando questa definizione vediamo che nei numeri 153 e 175 la cifra 5 ha peso rispettivamente uguale a 10 e 1; quindi in pratica peso e posizione di una cifra sono la stessa cosa e si può parlare indifferentemente dell'uno o dell'altra.

2.1 Rappresentazione polinomiale dei numeri

Nella notazione posizionale è fondamentale il concetto di '**`base di rappresentazione`**' che indichiamo genericamente con '**`B`**'.

Il valore V di un numero composto da N cifre nella base B si può scrivere:

$$V = \sum_{i=0}^{N-1} D_i * B^i$$

dove D_0, D_1, ... D_{N-1} sono le cifre (comprese fra 0 e B-1) della stringa si simboli che rappresenta il numero.

Questa rappresentazione del valore di un numero si dice «polinomiale» perché ricalca la rappresentazione di un polinomio generico di grado N-1, con la B al posto dell'incognita X.

Il valore di un numero si può quindi ottenere facendo la somma dei prodotti fra le sue cifre e i rispettivi pesi.

A titolo di esempio consideriamo la rappresentazione polinomiale del numero decimale (B = 10) 836:

$$V = 6 * 10^0 + 3 * 10^1 + 8 * 10^2$$

Il valore che si ottiene per V è ovviamente il già noto 836.

L'uso di questa rappresentazione, almeno per i numeri decimali, pare quindi di poca utilità; diviene però molto importante quando si usano sistemi di numerazione con altre basi, come mostrato più avanti.

2.2 Sistema di numerazione binario

2.2.1 Numeri binari interi

Nel sistema di numerazione binario (o in base 2) si hanno a disposizione solo le due cifre 0 e 1.

Prima di procedere occorre stabilire una convenzione che permetta di distinguere i numeri binari da quelli decimali scritti con le stesse cifre; in mancanza di una regola certa infatti si creerebbero ambiguità come la seguente: 110 è il numero decimale centodieci o il numero binario uno - uno - zero?

Per risolvere il problema decidiamo di individuare i numeri binari con un due aggiunto a pedice di questi ultimi: 110_2.

Il successivo problema che affrontiamo è quello delle conversioni tra basi diverse iniziando da quella da binario a decimale.

Supponiamo quindi di volere conoscere il valore rappresentato dal numero binario 1100_2; per fare questo ricorriamo alla rappresentazione polinomiale, che come visto in precedenza, permette di ottenere il valore di un numero:

$$V = 0 * 2^0 + 0 * 2^1 + 1 * 2^2 + 1 * 2^3$$

Svolgendo i semplici calcoli otteniamo il valore 12.

Come ulteriore esempio convertiamo il numero binario 1100110111_2:

$$V = 1 * 2^0 + 1 * 2^1 + 1 * 2^2 + 0 * 2^3 + 1 * 2^4 + 1 * 2^5 + 0 * 2^6 + 0 * 2^7 + 1 * 2^8 + 1 * 2^9$$

ottenendo il valore decimale 823.

Per effettuare la conversione inversa si deve usare un metodo leggermente meno banale denominato «algoritmo delle divisioni successive» che si origina dal seguente procedimento applicato alla rappresentazione polinomiale di un numero V in una base B:

$$V = D_0 * B^0 + D_1 * B^1 + D_2 * B^2 + D_3 * B^3 + D_4 * B^4 + ...$$

ovvero:

$$V = D_0 + B * (D_1 + D_2 * B^1 + D_3 * B^2 + D_4 * B^3 + ...)$$

che si può riscrivere come:

$$V = D_0 + B * (D_1 + B * (D_2 + B * (D_3 + ...))..)$$

e quindi si può ricavare D_0 come resto della divisione intera fra V e B.

Il quoziente di tale divisione è:

$$Q = D_1 + B * (D_2 + B * (D_3 + ...))..)$$

Su di esso si può iterare il procedimento per ricavare D_1 e via via tutte le altre cifre arrestandosi quando il quoziente della divisione diviene zero.

Si deve osservare che in questo modo si ottengono le cifre da quella meno significativa a quella più significativa (da destra a sinistra).

Applichiamo l'algoritmo per convertire il valore decimale 86 in binario:

```
86 / 2  Q = 43  R = 0
43 / 2  Q = 21  R = 1
21 / 2  Q = 10  R = 1
10 / 2  Q =  5  R = 0
 5 / 2  Q =  2  R = 1
 2 / 2  Q =  1  R = 0
 1 / 2  Q =  0  R = 1
```

Quindi il valore binario ottenuto è 1010110_2.

Come ulteriore esempio convertiamo in binario il valore decimale 255:

```
255 / 2  Q = 127  R = 1
127 / 2  Q =  63  R = 1
 63 / 2  Q =  31  R = 1
 31 / 2  Q =  15  R = 1
 15 / 2  Q =   7  R = 1
  7 / 2  Q =   3  R = 1
  3 / 2  Q =   1  R = 1
  1 / 2  Q =   0  R = 1
```

ottenendo la rappresentazione binaria 11111111_2.

2.2.2 Combinazioni con N bit

Essendo il bit l'unità fondamentale di misura della memoria, è importante chiedersi quante (e quali) sono le combinazioni diverse che si possono ottenere con N bit.

La risposta è in una semplice formula:

$$C = 2^N$$

Dimostriamo la validità di questa formula con il procedimento di induzione:

1. si verifica sia vera per N=1;
2. si suppone che sia vera per N=K;
3. se si riesce dimostrare che è vera per N=K+1 allora è vera sempre.

Per N=1 la formula è vera perché con un bit si possono avere esattamente 2 combinazioni di valori: 0 e 1;

supposto sia vera per N=K abbiamo che:

$$C_{K+1} = C_K * 2 = 2^K * 2 = 2^{(K+1)}$$

e quindi è vera anche per N=K+1 e dunque per qualsiasi N.

Usando la formula possiamo quindi affermare che le combinazioni ottenibili con 4 bit sono 16 e, essendo un numero esiguo, le scriviamo anche per esteso con a fianco il relativo valore decimale:

```
0000 = 0
0001 = 1
0010 = 2
0011 = 3
0100 = 4
0101 = 5
0110 = 6
0111 = 7
1000 = 8
1001 = 9
1010 = 10
1011 = 11
1100 = 12
1101 = 13
1110 = 14
1111 = 15
```

Notiamo che i valori associati alle combinazioni sono quelli compresi tra 0 e 2^N-1 (tra 0 e 15 con 4 bit) e anche questa formula è vera per qualsiasi valore di N.

2.2.3 Numeri binari non interi

I numeri binari non interi si rappresentano (in modo del tutto analogo ai decimali) con una parte intera, un punto e una parte frazionaria.

Il problema della conversione da binario a decimale di un numero non intero si risolve estendendo semplicemente la rappresentazione polinomiale di un numero anche alla parte non intera:

$$V = \sum_{i=0}^{N-1} D_i * B^i + \sum_{j=1}^{M} F_j * B^{-j}$$

dove le D_i sono le cifre della parte intera e le F_j quelle della parte frazionaria.

Quindi volendo convertire il numero binario 11010.101_2 in decimale abbiamo:

$$V = 0*2^0 + 1*2^1 + 0*2^2 + 1*2^3 + 1*2^4 + 1*2^{-1} + 0*2^{-2} + 1*2^{-3}$$

da cui otteniamo il valore 26.625 .

Il problema della conversione inversa, da decimale a binario, può essere suddiviso in due sottoproblemi:

- conversione della parte intera, con il già noto algoritmo delle divisioni successive;
- conversione della parte frazionaria sulla quale ci concentriamo adesso.

Supponiamo di avere quindi un numero frazionario che possiamo esprimere come:

$$FRAZ = F_1 * B^{-1} + F_2 * B^{-2} + F_3 * B^{-3} + F_4 * B^{-4} + F_5 * B^{-5} + ...$$

che si può riscrivere come:

$$FRAZ = B^{-1} * (F_1 + F_2 * B^{-1} + F_3 * B^{-2} + F_4 * B^{-3} + F_5 * B^{-4} + ...)$$

quindi, se si moltiplica FRAZ per B, la parte intera del risultato risulta essere proprio la prima cifra frazionaria F_1, mentre sulla parte frazionaria:

$$FRAZ2 = F_2 * B^{-1} + F_3 * B^{-2} + F_4 * B^{-3} + F_5 * B^{-4} + ...$$

si può ripetere il procedimento per ottenere F_2 e, iterando ancora, le altre cifre frazionarie.

Questo metodo si chiama «algoritmo delle moltiplicazioni successive» e si interrompe quando, dopo una moltiplicazione, si ottiene parte frazionaria pari a zero oppure quando si sono individuate un numero di cifre frazionarie ritenuto sufficiente ad esprimere il valore con la precisione desiderata.

Come esempio consideriamo la conversione del valore decimale 0.0625:

```
0.0625 * 2 = 0.125   Parte intera 0
0.125  * 2 = 0.25    Parte intera 0
0.25   * 2 = 0.5     Parte intera 0
0.5    * 2 = 1.0     Parte intera 1
```

quindi $0.0625 = 0.0001_2$.

Come esempio ulteriore convertiamo 32.87 stabilendo a priori di ottenere al massimo 6 cifre binarie frazionarie; il criterio di arresto (parte frazionaria pari a zero) non è infatti certo che sia soddisfatto dopo poche iterazioni e può anche non esserlo mai (infatti ci sono valori che hanno rappresentazione frazionaria finita in decimale ma non in binario).

Come detto in precedenza, in un caso come questo si convertono separatamente la parte intera e la parte frazionaria:

```
32 / 2  Q = 16  R = 0
16 / 2  Q =  8  R = 0
 8 / 2  Q =  4  R = 0
 4 / 2  Q =  2  R = 0
 2 / 2  Q =  1  R = 0
 1 / 2  Q =  0  R = 1
```

```
0.87 * 2 = 1.74      Parte intera 1
0.74 * 2 = 1.48      Parte intera 1
0.48 * 2 = 0.96      Parte intera 0
0.96 * 2 = 1.92      Parte intera 1
0.92 * 2 = 1.84      Parte intera 1
0.84 * 2 = 1.68      Parte intera 1
```

Il risultato finale è dunque: $32.87 = 100000.110111_2$.

Notiamo adesso che, se si riconverte in decimale il valore binario appena trovato, si ottiene 32.859375 che si discosta dal valore di partenza; il motivo è ovviamente che nella conversione da decimale non si sono trovate «abbastanza» cifre binarie frazionarie.

Nel caso dell'esempio precedente, invece, se riconvertiamo in decimale 0.0001_2 otteniamo il valore esatto 0.0625 e ciò si spiega con il fatto che il procedimento di calcolo delle cifre frazionarie si era arrestato regolarmente e non in modo forzato.

2.3 Sistemi di numerazione ottale e esadecimale

Nei sistemi di numerazione ottale e esadecimale (base 8 e base 16) sia hanno a disposizione rispettivamente otto e sedici cifre.

Nel sistema ottale non ci sono problemi nel decidere quali sono le otto cifre da usare: molto banalmente si considerano le prime otto cifre del sistema decimale, da 0 a 7.

Nel sistema esadecimale invece abbiamo il problema di individuare altri sei simboli da aggiungere alle cifre da 0 a 9; la scelta è quella di utilizzare le prime sei lettere dell'alfabeto inglese, A, B, C, D, E, F stabilendo il rispettivo valore intrinseco nel modo più ovvio:

```
A = 10
B = 11
C = 12
D = 13
E = 14
F = 15
```

Per quanto riguarda le conversioni da e verso valori decimali si applicano i metodi introdotti in precedenza: rappresentazione polinomiale e divisioni successive (non consideriamo valori ottali e esadecimali frazionari, quindi il metodo delle moltiplicazioni successive qui non serve); vediamo alcuni esempi.

Conversione del valore 13724_8 in decimale:

$$13724_8 = 4 * 8^0 + 2 * 8^1 + 7 * 8^2 + 3 * 8^3 + 1 * 8^4 = 6100$$

Conversione del valore $F3B_{16}$ in decimale:

$$F3B_{16} = 11 * 16^0 + 3 * 16^1 + 15 * 16^2 = 3899$$

Conversione del valore 510 in ottale:

```
510 / 8  Q = 63  R = 6
 63 / 8  Q =  7  R = 7
  7 / 8  Q =  0  R = 7
```

Quindi 510 = 776_8.

Conversione del valore 3456 in esadecimale:

```
3456 / 16  Q = 216  R = 0
 216 / 16  Q =  13  R = 8
  13 / 16  Q =   0  R = 13
```

Quindi 3456 = $D80_{16}$.

2.3.1 Conversioni dirette tra ottali o esadecimali e binari e viceversa

Le rappresentazioni R1 e R2 di uno stesso numero su basi B1 e B2 che sono una potenza dell'altra sono strettamente correlate:

se B1 = 2 e B2 = 2^n, ogni cifra nella rappresentazione R2 corrisponde a n cifre nella rappresentazione R1.

In particolare:

- ogni cifra ottale corrisponde a 3 cifre binarie;
- ogni cifra esadecimale corrisponde a 4 cifre binarie.

Come conseguenza si ha un metodo per passare dalla rappresentazione di un numero in base B1 a quella in base B2 (o viceversa) senza applicare gli algoritmi di conversione; si può agire direttamente sostituendo ordinatamente ogni cifra di R1 con gruppi di n cifre di R2.

Vediamo alcuni esempi.

Conversione diretta di 416_8 in binario:

$$4 = 100_2$$
$$1 = 001_2$$
$$6 = 110_2$$

Quindi $418_8 = 100001110_2$.

Conversione diretta di $A37C_{16}$ in binario:

$$A = 1010_2$$
$$3 = 0011_2$$
$$7 = 0111_2$$
$$C = 1100_2$$

Quindi $A37C_{16} = 1010001101111100_2$.

Conversione diretta di 1010110111011_2 in esadecimale e ottale:

$1010110111011_2 = 0001\ \ 0101\ \ 1011\ \ 1011\ = 15BB_{16}$

$1010110111011_2 = 001\ \ 010\ \ 110\ \ 111\ \ 011\ = 12673_8$

2.3.2 Ruolo dei numeri esadecimali e ottali

L'interesse per i numeri esadecimali in informatica è dovuto al fatto che il computer gestisce i simboli 0 e 1 (*bit*) raggruppati in sequenze di otto cifre (*byte*) e che i codici più diffusi per la rappresentazione dei dati al suo interno sono lunghi otto bit; se si dividono a metà queste sequenze si ottengono gruppi di quattro cifre binarie (*nibble*) ognuno dei quali rappresenta una cifra esadecimale; risulta allora comodo esprimere i dati trattati dall'elaboratore attraverso i corrispondenti valori esadecimali.

Nei primi anni dell'informatica, fino agli anni '50, esistevano codici lunghi sei bit, quindi aveva un certo interesse la sequenza di sei cifre binarie il cui contenuto poteva essere rappresentato da due cifre ottali; ecco quindi che anche il sistema ottale riveste un ruolo importante nell'informatica, almeno per motivi «storici».

Queste considerazioni comunque non hanno molto interesse per gli utilizzatori dei computer, bensì per i programmatori o per i «tecnici» che hanno a che fare con la logica interna di funzionamento di tali macchine.

2.4 Operazioni aritmetiche con numeri binari, ottali, esadecimali

Essendo tutti i sistemi posizionali, le operazioni aritmetiche con numeri binari, ottali e esadecimali si svolgono con le regole già note nel sistema decimale.

Occorre però ricordare che il numero di cifre a disposizione cambia, così come il valore della base (anche se si scrive sempre con i simboli «10») e questo si ripercuote sui calcoli, in modo particolare sulle operazioni di «riporto» e di «presa a prestito».

Vediamo alcuni esempi limitandoci alle operazioni di somma e sottrazione.

Per sommare 100011_2 e 10111_2 si incolonnano i due numeri a destra e si somma cifra per cifra a partire da destra ottenendo:

```
100011+
010111=
------
111010

1 + 1         = 0 con riporto = 1
1 + 1 + 1 (r) = 1 con riporto = 1
0 + 1 + 1 (r) = 0 con riporto = 1
0 + 0 + 1 (r) = 1
0 + 1         = 1
1 + 0         = 1
```

Per fare la sottrazione tra gli stessi due numeri si fa ancora l'incolonnamento e poi si sottrae cifra per cifra partendo da destra:

```
100011-
010111=
------
001100

1 - 1 = 0
1 - 1 = 0
0 - 1 = 1 con prestito dalla sesta cifra;
          il quarto e quinto bit diventano 1
1 - 0 = 1
1 - 1 = 0
0 - 0 = 0
```

Passando a numeri ottali facciamo la somma e sottrazione tra 4047_8 e 254_8:

```
4047+
0254=
----
4323

7 + 4         = 3 con riporto = 1
4 + 5 + 1 (r) = 2 con riporto = 1
0 + 2 + 1 (r) = 3
4 + 0         = 4

4047-
0254=
----
3573

7 - 4 = 3
4 - 5 = 7 con prestito dalla quarta cifra
          (diventa 14 ottale cio 12 come valore);
          la terza cifra diviene 7
7 - 2 = 5
3 - 0 = 3
```

Concludiamo con operazioni tra esadecimali sommando $A9B_{16}$, $FF3_{16}$ e $9D_{16}$:

```
 A9B+
 FF3+
 09D=
----
1B2B

 B + 3 + D         = B (11) con riporto = 1
 9 + F + 9 + 1 (r) = 2 con riporto = 2
 A + F + 0 + 2 (r) = B (11) con riporto = 1
 0 + 0 + 0 + 1 (r) = 1
```

Infine facciamo la sottrazione tra $B073_{16}$ e $F9C_{16}$:

```
B073-
 F9C=
----
A0D7

3 - C = 7 con prestito dalla seconda cifra
          (diventa 13 esadecimale cio 19 come valore)
6 - 9 = D con prestito dalla quarta cifra
          (diventa 16 esadecimale cio 22 come valore);
          la terza cifra diviene F
F - F = 0
A - 0 = A
```

2.5 Operazioni di shift su numeri binari, ottali, esadecimali

Non abbiamo fatto esempi sulle normali operazioni di moltiplicazione e divisione (che comunque non presentano alcuna novità a livello di metodo di calcolo), ma consideriamo un caso particolare di moltiplicazione e divisione: quelle che si ottengono mediante le operazioni di *shift* o scorrimento.

L'operazione di shift consiste nello spostare a destra o sinistra in un numero una certa quantità di cifre.

L'effetto di queste operazioni è il seguente:

- shift verso sinistra di K cifre: si spostano le cifre di K posti a sinistra e le posizioni liberate si riempono con zeri; si ottiene la moltiplicazione del numero per B^K (dove B ovviamente è la base del sistema di numerazione considerato);
- shift verso destra di K cifre: si spostano le cifre di K posti a destra perdendo le K cifre meno significative; si ottiene la divisione intera tra il numero e B^K.

2.6 Appendice sui sistemi di numerazione

In questo paragrafo vediamo alcune curiosità, anche storiche, sui numeri e sul loro uso.

2.6.1 Sistemi di numerazione addizionali

Come abbiamo visto i «moderni» sistemi di numerazione sono tutti posizionali; nei tempi più antichi invece erano «addizionali».

Il più semplice si tali sistemi è quello basato su un solo simbolo: il trattino verticale |.

Ogni numero veniva rappresentato con tanti trattini quante erano le unità che lo costituivano.

Il sistema addizionale più famoso, ed in certi casi ancora utilizzato, anche se solo per motivi «coreografici», è invece quello dei numeri romani che si basa sui seguenti sette simboli:

- I = 1
- V = 5
- X = 10
- L = 50
- C = 100
- D = 500
- M = 1000

e sulle seguenti due regole:

- se un simbolo ne segue un altro di valore uguale o maggiore, esso si somma al precedente;
- se un simbolo ne precede un altro di valore maggiore, esso si sottrae dal successivo.

Ecco alcuni esempi di conversione da numero romano a valore decimale:

```
DX        = 510
CM        = 900
MCMXCVIII = 1998
```

Se invece vogliamo convertire un valore decimale, ad esempio 27, in notazione romana, procediamo nel modo seguente:

- 27 è compreso fra 20 e 30 quindi scriviamo 20 in romano: XX;
- la parte residua non rappresentata è 7 che è compreso fra 5 e 10 quindi scriviamo 5 in romano: V;
- la parte residua non rappresentata è 2 che si rappresenta direttamente: II;
- quindi in totale otteniamo: XXVII.

2.6.2 I numeri arabi e lo zero

Nei sistemi di numerazione addizionali lo zero non ha senso di esistere; in effetti il suo utilizzo è diventato comune solo a partire dal secolo XIII, quando il matematico Pisano Leonardo Fibonacci lo introdusse nel mondo occidentale.

Questo avvenne unitamente all'introduzione del sistema di numerazione basato sulle cifre da 1 a 9, usato dagli Arabi, a loro volta ispirati dagli Indiani; grande merito a questo proposito va riconosciuto al matematico e astronomo persiano Muhammad Ibn Musa Al-Khwarizmi (considerato tra l'altro il «padre» dell'algebra) che nel secolo VIII introdusse la notazione posizionale e il concetto di zero.

Ancora oggi ci si riferisce alle cifre da 0 a 9 chiamandoli numeri «Arabi» o «Indo-Arabi».

2.6.3 Altre basi di numerazione

Ci si può chiedere come mai tra tutti i sistemi di numerazione posizionali sia stato privilegiato quello decimale che oggi viene usato praticamente in tutto il Mondo.

Gli storici della matematica sono unanimi nel ritenere che ciò sia dovuto al fatto che l'uomo ha dieci dita e può usarle per contare.

Esistono però ancora tracce di sistemi di numerazione caduti in disuso e che forse avevano basi diverse dal dieci:

- in molti contesti ha ancora una notevole importanza la «dozzina» come unità di misura; questa è una probabile reminiscenza di un sistema in base 12 per certi versi migliore del sistema decimale in quanto il 12 è divisibile per più fattori rispetto al 10;
- i francesi per dire ottanta dicono «quattro volte venti»; questa è una probabile eredità di un vecchio sistema in base venti (o visegimale);
- ancora oggi è usato, per misurare il tempo e gli angoli, il sistema di numerazione in base sessanta (o sessagesimale) degli antichi Babilonesi; il motivo per cui l'antico popolo della Mesopotamia scelse una base così «strana» è forse che il numero sessanta ha molti sottomultipli; si deve anche notare che il sistema dei Babilonesi era già posizionale ma non prevedeva il simbolo, e neppure il concetto, di zero.

2.6.4 Considerazione filosofica sui numeri binari

Concludiamo con una nota «filosofica» riguardante i numeri binari.

Essi sono stati a lungo materia di studio e curiosità per molti matematici, anche famosi, che ammiravano la loro «eleganza» e semplicità; in particolare Leibnitz (1646 - 1716) vedeva nell'aritmetica binaria un'immagine della creazione: immaginava che l'unità rappresentasse Dio e lo zero il nulla e che Dio avesse creato tutti gli esseri dal nulla così come l'unità e lo zero esprimono tutti i valori nel sistema binario.

Capitolo 3

Codifica binaria dell'informazione

In tutti gli elaboratori elettronici costruiti fino ad oggi le informazioni sono rappresentate, trasmesse ed elaborate tramite grandezze fisiche, delle quali vengono considerati significativi solo alcuni valori (*livelli discreti*).

Se ad ogni livello si associa un simbolo diverso si ottiene un *alfabeto* per esprimere le informazioni da trattare.

Negli elaboratori si usa un alfabeto costituito dai due soli simboli 0 e 1, chiamati BIT (*BInary digiT*), in quanto si preferisce avere a che fare con grandezze fisiche che hanno solo due livelli discreti e stabili.

I motivi di questa scelta si possono così riassumere:

- efficienza: i calcoli sono svolti più rapidamente in binario che con altre basi di numerazione;
- economicità: i dispositivi a due soli stati sono più semplici e meno costosi;
- affidabilità: essendo solo due gli stati possibili delle grandezze fisiche è più semplice individuare errori di rappresentazione dovuti ai disturbi che esse subiscono («rumore» ambientale).

Rimandando maggiori dettagli e definizioni generali riguardanti i codici al paragrafo (3.2.1) in cui si esaminano le codifiche dei dati alfanumerici per l'elaboratore, vediamo adesso come si rappresentano in quest'ultimo le informazioni numeriche.

3.1 Rappresentazione dei valori numerici nell'elaboratore

In questo capitolo vengono esaminati i principali metodi con i quali possono essere rappresentati i valori numerici all'interno del sistema di elaborazione distinguendo tra valori interi e valori reali.

In entrambi i casi è però importante chiarire come non ci si riferisca all'insieme dei numeri interi e all'insieme dei numeri reali conosciuti in matematica in quanto questi due insiemi sono infiniti (e quello dei reali è anche «denso», dato che tra valori vicini quanto si vuole, ci sono infiniti valori reali).

Data la finitezza dei componenti interni della macchina, non è possibile gestire infiniti valori e quindi si rappresentano solo dei sottoinsiemi dei numeri interi e dei numeri reali; inoltre quelli che fra questi ultimi hanno una rappresentazione infinita, possono essere solo approssimati.

In base a tali considerazioni sarebbe più corretto parlare di *numeri interi macchina* e *numeri reali macchina* quando si fa riferimento ai valori rappresentati nell'elaboratore; in queste dispense si fa comunque uso delle definizioni più brevi dando per scontato si intende parlare dei sottoinsiemi gestibili dalla macchina.

3.1.1 Rappresentazione dei valori interi

I metodi per la rappresentazione dei numeri interi nell'elaboratore sono fondamentalmente tre:

- *modulo (valore assoluto) e segno;*
- *metodo dell'eccesso o bias (polarizzazione);*

- *complemento a due.*

3.1.1.1 Rappresentazione in modulo e segno

Viene usato il bit più a sinistra, cioè il più significativo o MSB (*Most Significant Bit*) per indicare esplicitamente il segno con la seguente convenzione (che è quasi sempre valida anche in altri contesti):

- 0 = segno positivo;
- 1 = segno negativo.

Gli altri bit disponibili sono utilizzati per rappresentare il valore assoluto come numero binario «puro».

L'intervallo dei valori rappresentabili con N bit è:

$-2^{(N-1)}-1 <= X <= +2^{(N-1)}-1$

La quantità di valori è comunque 2^N (come è lecito attendersi) e non 2^N-1 , come appare dall'ampiezza dell'intervallo, perché ci sono due rappresentazioni del valore zero.

Se supponiamo ad esempio che i bit a disposizione per la gestione dei valori interi siano quattro (ovviamente tal valore è irrealisticamente basso, in realtà si hanno rappresentazioni con 8 o, meglio ancora, 16 o 32 bit) otteniamo i seguenti valori interi:

```
0 000 = +0      1 000 = -0
0 001 = +1      1 001 = -1
0 010 = +2      1 010 = -2
0 011 = +3      1 011 = -3
0 100 = +4      1 100 = -4
0 101 = +5      1 101 = -5
0 110 = +6      1 110 = -6
0 111 = +7      1 111 = -7
```

Osserviamo che il segno è completamente disgiunto dal valore assoluto e che, in linea di principio, la posizione del bit di segno entro la stringa di bit è irrilevante (ma si preferisce scegliere il bit più a sinistra).

Notiamo anche che, come detto, il valore zero ha due rappresentazioni e che non si possono usare i metodi usuali per eseguire le operazioni aritmetiche; addirittura non è vero che:

$X + (-X) = 0$

Infatti, facciamo 3 + (-3):

```
0011+
1011=
----
1110  = -6
```

Il problema è proprio nel fatto che il bit del segno è disgiunto dal resto della rappresentazione e quindi deve essere trattato a parte nei calcoli.

Ciò rende necessari circuiti specifici o software più complessi per la realizzazione delle operazioni aritmetiche da parte della macchina e quindi, pur trattandosi di un metodo semplice e intuitivo, ha un uso concreto molto scarso.

3.1.1.2 Rappresentazione con il metodo dell'eccesso

Per usare questo metodo occorre prima di tutto calcolare il valore dell'eccesso o *bias*.

Supponendo che N sia il numero di bit disponibili per la rappresentazione dei valori, il calcolo si può fare in due modi:

- bias = $2^{(N-1)}$
- bias = $2^{(N-1)}$-1

Il numero X da rappresentare viene prima sommato al bias e poi tradotto in binario, in modo che sia sempre positivo; viceversa se abbiamo una rappresentazione binaria, dobbiamo sottrarre il bias al valore rappresentato per avere il valore effettivo.

Come esempio supponiamo di avere ancora quattro bit bit a disposizione; in tal caso il bias vale 8 oppure 7 i valori rappresentati nei due casi sono:

```
bias = 8

  0000 = -8      1000 =  0
  0001 = -7      1001 = +1
  0010 = -6      1010 = +2
  0011 = -5      1011 = +3
  0100 = -4      1100 = +4
  0101 = -3      1101 = +5
  0110 = -2      1110 = +6
  0111 = -1      1111 = +7

bias = 7

  0000 = -7      1000 = +1
  0001 = -6      1001 = +2
  0010 = -5      1010 = +3
  0011 = -4      1011 = +4
  0100 = -3      1100 = +5
  0101 = -2      1101 = +6
  0110 = -1      1110 = +7
  0111 =  0      1111 = +8
```

L'intervallo dei valori rappresentabili con N bit è:

$-2^{(N-1)} <= X <= +2^{(N-1)}-1$

oppure:

$-2^{(N-1)}-1 <= X <= +2^{(N-1)}$

in dipendenza di come viene calcolato il valore di polarizzazione.

Anche stavolta il primo bit rappresenta il segno, ma, contrariamente alla convenzione più diffusa, i numeri negativi hanno il bit del segno pari a 0 e i numeri positivi pari a 1; il valore zero ha 1 come bit del segno nella rappresentazione eccesso $2^{(N-1)}$ e 0 nella rappresentazione eccesso $2^{(N-1)}$-1.

Questo metodo di rappresentazione è meno intuitivo del precedente, non è usato effettivamente per gestire i valori interi ma viene «ripescato» a proposito di un aspetto riguardante i numeri reali.

3.1.1.3 Rappresentazione in complemento a due

Vediamo prima di tutto come si definisce il complemento alla base B di un valore espresso, con N cifre, appunto, nella base B:

$$C_B(V) = B^N - V$$

Ad esempio il complemento a dieci di 753 è 247 e possiamo vedere che si può ottenere, oltre che con l'applicazione della formula, anche facendo il complemento a nove di ogni singola cifra e sommando 1 al risultato.

Questa osservazione è molto utile per fare il complemento a due di un valore binario.

Supponiamo ad esempio di volerlo calcolare per il valore 110010_2; possiamo applicare la definizione ottenendo:

$$C_2(110010_2) = 1000000_2 - 110010_2 = 001110_2$$

ma in modo più agevole possiamo fare il complemento a uno di ogni cifra (che consiste banalmente nello scambiare i valori zero con uno e viceversa) e poi sommare 1:

```
001101+
     1=
------
001110
```

Un altro metodo, ancora più semplice e veloce per ottenere il complemento a due di un numero binario è il seguente:
«scorrere il numero da destra lasciando i bit inalterati fino al primo uno (compreso) e invertire il valore dei successivi bit».

Anche con il metodo del complemento a due il bit più significativo rappresenta il segno, stavolta rispettando la convenzione più diffusa:

- 0 = segno positivo;
- 1 = segno negativo.

La regola generale per questa rappresentazione è la seguente:

- se il valore è positivo lo si rappresenta normalmente in binario avendo però cura di scrivere anche gli zeri non significativi a sinistra (in modo che sia sempre presente il bit del segno);
- se il valore è negativo si converte in binario e poi si effettua il complemento a due.

Facciamo due esempi per i valori 61 e -76, supponendo di avere a disposizione otto bit per la rappresentazione degli interi:

```
61 / 2  Q = 30  R = 1
30 / 2  Q = 15  R = 0
15 / 2  Q =  7  R = 1
 7 / 2  Q =  3  R = 1
 3 / 2  Q =  1  R = 1
 1 / 2  Q =  0  R = 1
```

Quindi: 61 = 00111101_2.

```
76 / 2  Q = 38  R = 0
38 / 2  Q = 19  R = 0
19 / 2  Q =  9  R = 1
 9 / 2  Q =  4  R = 1
 4 / 2  Q =  2  R = 0
 2 / 2  Q =  1  R = 0
 1 / 2  Q =  0  R = 1
```

Allora 76 sarebbe 01001100_2 e passando al complemento a due otteniamo: -76 = 10110100_2.

Per la conversione inversa, da binario espresso in complemento a due a decimale, si può procedere in due modi:

1. se il valore è positivo lo si converte normalmente; se è negativo prima si fa il complemento, poi lo si converte, infine si moltiplica il risultato per -1;
2. si converte normalmente il valore considerando però il peso del bit più significativo con esponente negativo.

Vediamo esempi dei due metodi considerando numeri negativi in quanto più «interessanti» e supponendo di avere 10 bit per la rappresentazione.

Convertiamo 1010110010_2; il complemento a due è: 0101001110_2 che passiamo in decimale:

$$0*2^0+1*2^1+1*2^2+1*2^3+0*2^4+0*2^5+1*2^6+0*2^7+1*2^8+0*2^9=334$$

quindi il risultato è: -334.

Convertiamo ora 1110001101_2 con l'altro metodo:

$$1*2^0+0*2^1+1*2^2+1*2^3+0*2^4+0*2^5+0*2^6+1*2^7+1*2^8+1*2^{-9}=-115$$

Come si vede il secondo metodo è molto più rapido.

La rappresentazione degli interi in complemento a due è senza dubbio la più utilizzata in quanto, pur essendo meno intuitiva di quella in modulo e segno ha i seguenti pregi:

- si possono usare le regole dell'aritmetica binaria senza segno;
- c'è una sola rappresentazione dello zero;
- la convenzione sul bit del segno è rispettata;
- non è necessario nessun circuito particolare per trattare i valori negativi;
- è verificata la proprietà X + (-X) = 0 scartando però il bit di riporto che si ottiene a sinistra del bit del segno.

Vediamo un esempio relativo all'ultima osservazione supponendo che X sia 3 e che si usino quattro bit per scrivere i valori interi:

```
 0011+
 1101=
 ----
10000
```

L'intervallo dei valori rappresentabili in complemento a due con N bit è:

$-2^{(N-1)} <= X <= +2^{(N-1)}-1$

Il massimo intero positivo rappresentabile è in modulo inferiore di uno al minimo negativo rappresentabile perché il valore zero fa intrinsecamente parte dei valori positivi.

Se supponiamo che N sia 4, vediamo quali sono i valori rappresentati:

```
0000 =  0      1000 = -8
0001 = +1      1001 = -7
0010 = +2      1010 = -6
0011 = +3      1011 = -5
0100 = +4      1100 = -4
0101 = +5      1101 = -3
0110 = +6      1110 = -2
0111 = +7      1111 = -1
```

Si noti che con questo tipo di notazione si perde la «somiglianza» tra valori opposti che invece c'era nella rappresentazione in modulo e segno; questo però non è un problema perché per il sistema di elaborazione tale perdita non porta alcuno svantaggio.

3.1.1.4 Somme e sottrazioni in complemento a due

La rappresentazione in complemento a due è la più utilizzata anche per un motivo legato ai calcoli aritmetici: la sottrazione tra due valori espressi in questa forma può infatti essere effettuata comodamente facendo una somma tra il primo valore ed il complemento a due del secondo (scartando il bit di riporto oltre il bit del segno).

A dire il vero il «trucco» è sfruttabile anche in decimale facendo la somma tra il primo valore ed il complemento a dieci del secondo; ad esempio:

per fare 842 - 245 facciamo 842 + 755 ottenendo = 1597 e, scartando il primo 1, abbiamo 597 che è il risultato atteso.

Il fatto che le sottrazioni si trasformino in somme è molto importante ai fini della semplificazione dei circuiti interni del sistema di elaborazione in quanto non sono necessari dispositivi adibiti a svolgere queste operazioni.

Se poi si tiene conto che le moltiplicazioni si possono ottenere «via software» come successioni di somme e le divisioni analogamente come successioni di sottrazioni, emerge che l'unico circuito aritmetico necessario alla macchina è, almeno in linea teorica, il circuito sommatore.

Nello svolgere le operazioni di somma e sottrazione occorre però porre attenzione agli errori che si possono verificare e che sono dovuti alla limitatezza dell'intervallo dei valori rappresentabili.

Si possono infatti verificare situazioni di *overflow* o traboccamento dovute la fatto che il risultato di una operazione esce da tale intervallo.

Ci sono vari sistemi per accorgersi di questo tipo di errori:

- la somma tra due valori positivi pare fornire un risultato negativo o comunque il segno del risultato è opposto a quello che sarebbe lecito attendersi;
- esaminando i valori coinvolti nell'operazione ci si accorge che il risultato eccede i limiti dei valori rappresentabili prima ancora di svolgere l'operazione;
- si riscontra una situazione particolare relativamente al riporto sul bit del segno e sul bit ulteriore a sinistra di quello del segno.

Chiariamo meglio l'ultimo punto:

- se si ha un riporto sul bit del segno e sul bit a sinistra di esso, oppure, non si ha riporto su nessuno dei due bit, allora il risultato è corretto (scartando l'eventuale bit eccedente a sinistra dal risultato);
- se si ha un riporto sul bit del segno e non sul bit a sinistra di esso o viceversa, allora il risultato non è corretto e si ha una situazione di overflow.

Vediamo alcuni esempi in cui si suppone di avere a disposizione otto bit per la rappresentazione dei valori interi (per brevità le conversioni non vengono dettagliate).

somma tra -50 e +90:

```
  -50 = 11001110
  +90 = 01011010

  11001110+
  01011010=
  --------
1 00101000 = +40
```

differenza tra -50 e +90 (fatta come somma tra -50 e il complemento di +90):

```
   -50 = 11001110
C(+90) = 10100110

  11001110+
  10100110=
  --------
1 01110100 = Overflow
```

somma tra 84 e -75:

```
  +84 = 01010100
  -75 = 10110101

  01010100+
  10110101=
  --------
1 00001001 = +9
```

differenza tra 84 e -75 (fatta come somma tra 84 e il complemento di -75):

```
   +84 = 01010100
C(-75) = 01001011

  01010100+
  01001011=
  --------
  10011111 = Overflow
```

somma tra 55 e 99:

```
+55 = 00110111
+99 = 01100011

00110111+
01100011=
--------
10011010 = Overflow
```

differenza tra 55 e 99 (fatta come somma tra 55 e il complemento di 99):

```
   +55 = 00110111
C(+99) = 10011101

  00110111+
  10011101=
  --------
  11010100 = -44
```

3.1.1.5 Osservazioni finali sui numeri interi macchina

A causa della limitatezza dei dispositivi dedicati alla rappresentazione dei valori interi macchina, per essi non valgono la proprietà associativa e commutativa della somma e del prodotto e la proprietà distributiva della somma.

Infatti se ad esempio X è il massimo intero rappresentabile, l'espressione (X-X)+X (che, calcolata, dà il risultato corretto pari a X) non equivale a X-(X+X), in quanto il risultato parziale (X+X) non è rappresentabile.

Nell'elaboratore, oltre che di aritmetica finita, si parla anche di aritmetica *modulare* in cui i numeri interi «si avvolgono su se stessi» e si possono rappresentare su una circonferenza anziché sulla usuale retta infinita.

Notiamo infatti che se si dedicano sedici bit alla rappresentazione dei valori interi (come per gli interi in linguaggio Pascal) si ha un intervallo di valori compreso tra -32.768 e +32.767 ma, al di fuori di esso non c'è il «vuoto» in quanto, essendo i valori disposti su una circonferenza, dopo il valore 32.767 si troverà il valore -32.768, poi -32.767 e così via.

Aumentando il numero di bit a disposizione si amplia ovviamente l'intervallo dei valori rappresentabili: con trentadue bit (valori interi per il linguaggio c) si va da -2.147.483.648 a 2.147.483.647; rimane però valido il concetto di disposizione «circolare» di tali valori.

3.1.2 Rappresentazione dei valori reali

Abbiamo già accennato al fatto che la rappresentazione finita dei reali nel calcolatore è impossibile quando i numeri hanno infinite cifre nella parte frazionaria; questo accade per:

- numeri irrazionali, non rappresentabili finitamente in forma esplicita in nessuna base (ad esempio: $2^{1/2}$, π);
- numeri razionali *periodici* nella base di rappresentazione utilizzata.

A tale proposito si deve notare che un valore razionale può essere periodico o meno a seconda della base usata per rappresentarlo; ad esempio 1/3 = 0.333333333333... è periodico in base 10

ma non in base 3 dove si scrive 0.1_3, oppure 8/7 = 1.142857142857... periodico in base 10 ma non in base 7 dove si scrive 1.1_7.

In generale abbiamo che se un valore è periodico quando rappresentato in base B, lo è anche quando rappresentato in qualunque base B_1 che sia un fattore di B ($B = k*B_1$).

Quindi se X è periodico in base 10, lo è anche in base 2 o in base 5 (ma non vale necessariamente il viceversa).

Inoltre abbiamo che se un valore è periodico in base B lo è anche in una base B_2 che è una sua potenza ($B_2 = B^k$).

Quindi se Y è periodico in base 2 lo è anche in base 8 e 16.

3.1.2.1 Rappresentazione in virgola fissa

La rappresentazione in virgola fissa è così denominata in quanto, stabilito il numero K di bit da usare per memorizzare i numeri reali (valori possibili di k sono: 16, 32, 64, 80), si usano:

- un bit per rappresentare il segno;
- i rimanenti K-1 bit per la memorizzazione della parte intera e della parte frazionaria, suddividendoli in due gruppi di ampiezza fissa (la virgola di separazione tra parte intera e frazionaria assume idealmente una posizione fissa).

Supponiamo ad esempio che i bit a disposizione in una ipotetica macchina siano 32 e che la suddivisione sia: 1 per il segno, 15 per la parte intera e 16 per la parte frazionaria; in questo caso avremmo un intervallo di valori rappresentabili compreso tra $0.1*2^{-16}$ e 2^{15} per i valori positivi ed uno analogo per i valori negativi.

Questo metodo di rappresentazione dei numeri reali macchina non è conveniente in quanto i valori rappresentabili sono relativamente pochi e la precisione che si ottiene non è elevata.

3.1.2.2 Rappresentazione in virgola mobile

La rappresentazione in virgola mobile (*floating point*), detta anche *a mantissa e caratteristica*, si basa sulla notazione esponenziale o scientifica dei numeri.

Secondo tale notazione un numero reale può essere espresso nella seguente forma:

$$V = m * B^{esp}$$

dove:

- *m* è un valore frazionario chiamato *mantissa*;
- *B* è la base del sistema di numerazione;
- *esp* è un valore intero chiamato *esponente* o *caratteristica*.

Ad esempio il valore 1.125 lo possiamo rappresentare come $1.125*10^0$ ma anche come $0.1125*10^1$ oppure come $11.25*10^{-1}$.

In effetti, data una base B, esistono infinite coppie (m, esp) in grado di rappresentare, lo stesso valore reale V.

Questo fatto, in vista della definizione di uno standard che sia universale per la rappresentazione dei reali macchina e che elimini le ambiguità, non è molto positivo; occorre dunque fare una scelta tra le varie forme esponenziali, in modo che la rappresentazione di un dato valore sia unica.

La scelta ricade sulla notazione esponenziale *normalizzata* che è quella per cui la mantissa è compresa tra 0 (incluso) e 1 (escluso) e la sua prima cifra (se il valore di m non è zero) è diversa da zero.

Tornando all'esempio precedente la notazione esponenziale normalizzata (che è univoca) è $0.1125*10^1$.

Osserviamo che, nel caso di valore normalizzato, vale:

$$\frac{1}{B} < m < 1$$

Infatti, ricordando che m è un numero frazionario, abbiamo:

$$m = F_1 * B^{-1} + F_2 * B^{-2} + F_3 * B^{-3} + F_4 * B^{-4} + F_5 * B^{-5} + ...$$

pertanto affinché in qualunque base la prima cifra della mantissa sia diversa da zero deve esserci il termine F_1*B^{-1} e quindi si ha che $1/B < m$.

La rappresentazione normalizzata è una scelta efficiente perché sfrutta tutte le cifre disponibili per la mantissa senza sprecarle per zeri iniziali che non portano vera informazione.

Nell'elaboratore la rappresentazione di valori reali è espressa ovviamente in binario, quindi dobbiamo preoccuparci di convertire dei valori reali decimali in binario con notazione esponenziale normalizzata.

Per fare questo si può convertire il valore decimale V in binario (separatamente per la parte intera e la parte frazionaria, come visto in precedenza) e poi normalizzare il valore ottenuto.

Esiste però anche un metodo alternativo:

- individuare le potenze di 2 tra le quali si trova V;
- prendere la più alta delle due ed assegnare l'indice della potenza all'esponente esp;
- assegnare alla mantissa m il risultato della seguente operazione:

$$m = \frac{V}{2^{esp}}$$

- convertire in binario la mantissa m così ottenuta.

Ad esempio troviamo la rappresentazione normalizzata in base 2 di 5.875:

- siccome $2^2 < 5.875 < 2^3$ abbiamo esp = 3
- m = 5.875/8 = 0.734375
- convertiamo 0.734375 in binario:

```
0.734375 * 2 = 1.46875   Parte intera 1
0.46875  * 2 = 0.9375    Parte intera 0
0.9375   * 2 = 1.875     Parte intera 1
0.875    * 2 = 1.75      Parte intera 1
0.75     * 2 = 1.5       Parte intera 1
0.5      * 2 = 1.0       Parte intera 1
```

Quindi: $5.875 = 0.101111_2 * 2^3$.

3.1.2.3 Lo standard IEEE 754 per i numeri reali macchina

Nella storia dell'informatica c'è stato un periodo in cui ogni grande azienda proponeva il suo metodo di rappresentazione dei valori reali in virgola mobile senza curarsi dei problemi di compatibilità dei dati.

Per porre fine alla proliferazione dei formati di rappresentazione fu proposto uno standard dallo IEEE (*Institute of Electrical and Electronics Engineers*) denominato *IEEE 754* o *Standard for Binary Floating-Point Arithmetic*.

Lo standard definisce tre formati:

- *singola precisione* (single precision) con utilizzo di 32 bit;
- *doppia precisione* (double precision) con utilizzo di 64 bit;
- *precisione estesa* (extended precision) con utilizzo di 80 bit.

La precisione estesa è comunque usata solo internamente alle unità di calcolo FPU (*Floating Point Unit*) e non viene messa a disposizione per la definizione dei dati da parte dei linguaggi di programmazione.

In tutti e tre i formati le informazioni che vengono gestite per memorizzare i valori reali sono:

- segno del numero;
- mantissa del numero;
- caratteristica del numero.

Il numero viene considerato espresso in binario secondo la notazione esponenziale che qui chiamiamo per comodità «quasi normalizzata».

Non viene utilizzata la forma normalizzata in quanto in essa il primo bit frazionario vale sempre e sicuramente 1 e quindi non serve memorizzarlo (viene chiamato '**`bit nascosto`**'); in questo modo si guadagna un bit per la mantissa.

Nel caso della precisione estesa però il bit nascosto non esiste; vengono infatti considerati tutti i bit della mantissa partendo quindi dalla notazione normalizzata del valore reale; questo avviene per due motivi:

- la mantissa è già abbastanza grande e non serve guadagnare un bit aggiuntivo;
- essendo quella estesa la forma usata internamente per i calcoli, è meglio avere memorizzati tutti i bit della rappresentazione dei valori in modo da facilitare le operazioni su di essi.

La notazione quasi normalizzata si ottiene da quella normalizzata spostando di un posto a destra il punto frazionario e togliendo quindi dalla mantissa il primo bit che vale sempre 1 e che diventa la parte intera del valore; questo è il '**bit nascosto**' che, come detto, non viene preso in considerazione nei casi di singola e doppia precisione.

Se consideriamo allora l'esempio illustrato alla fine del paragrafo precedente, cioè quello riguardante il valore 5.875, non dobbiamo considerare come punto di partenza per la rappresentazione nello standard IEEE 754 il valore normalizzato 0.101111_2*2^3, bensì il valore quasi normalizzato 1.01111_2*2^2.

In questo caso abbiamo dunque che la mantissa è 1.01111_2 (di cui si memorizza solo la parte frazionaria 01111_2) mentre la caratteristica è 10_2.

Nella tabella che segue viene dettagliata la quantità di bit usati, nei tre formati dello standard, per rappresentare il segno, la caratteristica e la mantissa dei valori reali; si noti che le tre informazioni vengono codificate nell'ordine appena enunciato.

Formato	Bit totali	Bit segno	Bit caratteristica	Bit mantissa
Singola precisione	32	1	8	23
Doppia precisione	64	1	11	52
Precisione estesa	80	1	15	64

Una osservazione importantissima deve essere fatta per la caratteristica che viene memorizzata con il metodo dell'eccesso (per risparmiare il bit che servirebbe a rappresentare il suo segno).

Il valore di polarizzazione viene calcolato con la seconda formula tra le due possibili enunciate nel paragrafo (3.1.1.2) riguardante i numeri interi macchina e cioè:

$bias = 2^{(N-1)}-1$

dove N è il numero di bit da usare.

Quindi la caratteristica viene espressa in *eccesso 127* nel caso di singola precisione e in *eccesso 1023* nel caso di doppia precisione.

Nel nostro esempio del valore 5.875, abbiamo che la caratteristica rappresentata è 129 (10000001_2) e 1025 (10000000001_2) rispettivamente.

Le stringhe complete di bit che rappresentano il valore 5.875 nei formati singola precisione e doppia precisione sono:

0 10000001 01111000000000000000000

0 10000000001 0111100000000000000000000000000000000000000000000000

Gli spazi sono stati messi solo per far comprendere meglio la separazione tra bit del segno, caratteristica e mantissa, ovviamente l'elaboratore non li inserisce nella rappresentazione.

Per avere una forma più compatta delle stringhe appena ottenute, possiamo sostituire ai trentadue o ai sessantaquattro bit, a gruppi di quattro, le corrispondenti cifre esadecimali:

$40BC0000_{16}$

4017800000000000_{16}

Consideriamo adesso il procedimento inverso, cioè quello che ci permette di passare da una sequenza di trentadue o sessantaquattro bit al valore reale decimale rappresentato.

Come primo esempio partiamo da una rappresentazione in singola precisione i cui bit corrispondono alle cifre esadecimali $414E0000_{16}$:

i bit sono quindi:

0 10000010 10011100000000000000000

da cui si deduce che il segno è positivo, la caratteristica rappresentata è 130 e quindi quella effettiva è 3, mentre la mantissa è 1.100111_2.

Quindi il valore binario è 1100.111_2 che corrisponde a 12.875 .

Vediamo poi la seguente rappresentazione in doppia precisione, sempre con in bit raggruppati in notazione esadecimale, $BFE6000000000000_{16}$:

passiamo a scrivere i bit per esteso:

1 01111111110 0110000000000000000000000000000000000000000000000000

abbiamo quindi segno negativo, caratteristica rappresentata 1022, cioè effettiva -1, mantissa 1.011_2.

Il valore binario è dunque -0.1011_2 che corrisponde a -0.6875 .

I valori estremi della caratteristica (tutti zero o tutti uno) assumono un significato particolare:

- se i bit della caratteristica sono tutti zero ed è zero anche la mantissa si rappresenta il valore 0.0;
- se i bit della caratteristica sono tutti zero e la mantissa è diversa da zero, siamo in presenza di valori reali *denormalizzati* (che qui non approfondiamo);
- se i bit della caratteristica sono tutti uno e la mantissa è zero si tratta della rappresentazione del concetto di infinito (più o meno infinito, secondo il bit del segno);
- se i bit della caratteristica sono tutti uno e la mantissa è diversa da zero si tratta di rappresentazioni non valide («not a number»).

3.1.2.4 Precisione della rappresentazione

Per *precisione* di una rappresentazione in virgola mobile si intende il numero di cifre frazionarie che si riescono a rappresentare senza troncamenti.

A tale proposito si osservi che, se si vuole rappresentare in binario con la stessa precisione una mantissa normalizzata in base 10, occorre prevedere un numero di cifre binarie pari a più del triplo delle cifre presenti nella rappresentazione decimale.

Infatti abbiamo osservato nel paragrafo 1.2 sulla misura dell'informazione, come il numero di bit necessari a rappresentare un valore decimale è 3.32 (arrotondato a 4).

Alla stessa conclusione si arriva osservando che il numero di cifre binarie necessarie per esprimere un valore V è proporzionale a $\log_2(V)$, mentre il numero di cifre decimali necessarie per esprimere lo stesso valore V è proporzionale a $\log_{10}(V)$; quindi dobbiamo calcolare il rapporto:

$$\frac{\log_2(V)}{\log_{10}(V)} = \frac{\log_2(10) * \log_{10}(V)}{\log_{10}(V)} = \log_2(10) = 3.32$$

Se siamo interessati a gestire valori fino alla k-esima cifra decimale (ovvero ad accettare un errore di 10^{-k}), dovremo allora prevedere di usare 3.32*k cifre binarie frazionarie.

Riassumendo si ha che le cifre per la memorizzazione della mantissa servono per la precisione, mentre le cifre per la memorizzazione della caratteristica determinano l'ampiezza dell'intervallo dei valori reali macchina.

A proposito della precisione con cui si possono rappresentare i valori reali macchina, si definisce *precisione macchina* il valore più piccolo, in valore assoluto, che l'elaboratore è in grado di «apprezzare» nei calcoli; lo si può calcolare con il seguente metodo:

- si imposta *prec = 1*;
- si esegue ciclicamente *prec = prec/2* finché *1 + prec > 1*;
- quando *1 + prec = 1* vuol dire che prec non viene più «sentito» nei calcoli e quindi *prec = prec * 2* è la precisione di macchina.

Si noti che il valore della precisione di macchina è diverso (maggiore) del valore positivo più piccolo rappresentabile; nel caso di singola precisione si hanno rispettivamente i valori: $2.3*10^{-7}$ e $1.2*10^{-38}$.

3.1.2.5 Errori nei calcoli con i reali macchina

DA COMPLETARE .....

3.1.2.6 Cenni sul *calcolo numerico*

I calcoli effettuati con i numeri floating-point possono essere affetti non solo da errori (di troncamento/arrotondamento, incolonnamento, cancellazione) sui dati introdotti dal sistema quando le cifre significative sono diverse da quelle ammesse nella rappresentazione, ma anche da errori generati dal particolare algoritmo usato per sviluppare i calcoli stessi.

L'entità degli errori e l'effetto della loro propagazione nei calcoli va tenuta in considerazione.

Infatti in alcuni algoritmi iterativi la propagazione degli errori durante il ciclo dei calcoli può portare ad una situazione di instabilità o inaffidabilità dei risultati (mentre altre volte è il problema stesso che, essendo mal condizionato, può dar luogo ad instabilità).

Ad esempio nel calcolo per passi dell'integrale definito di una funzione, se si sceglie come passo di integrazione un valore troppo piccolo, nell'intento di aumentare la precisione del risultato, può accadere che il valore della variabile non venga mai incrementato (errore di incolonnamento e successivo troncamento) e il programma entri in una situazione di ciclo infinito (*loop*).

Se n è il valore esatto di un numero e $\tilde{n}$ il valore rappresentato, si definisce *errore assoluto*:

$e = n - \tilde{n}$

e *errore relativo*:

$\epsilon = (n - \tilde{n}) / n$

In genere si è interessati a minimizzare sia il modulo dell'errore assoluto che dell'errore relativo.

Questi aspetti vengono tenuti in grande considerazione quando si deve usare il sistema di elaborazione per effettuare una grande mole di calcoli scientifici; esiste anche una disciplina apposita denominata correntemente *calcolo numerico* che studia l'entità e la propagazione degli errori numerici introdotti durante l'esecuzione di istruzioni svolte su valori rappresentati in maniera finita e approssimata, al fine di minimizzarli.

3.1.2.7 Distribuzione dei valori reali macchina

L'insieme dei numeri rappresentabili secondo lo standard IEEE 754 è un sottoinsieme dei numeri razionali, distribuito in modo non uniforme: infatti essi sono estremamente vicini fra loro nell'intorno dello zero e si allontanano sempre di più al crescere del loro valore assoluto.

Per verificarlo calcoliamo la differenza (distanza) tra due numeri contigui posti a diversa distanza dal valore zero considerando di avere mantissa espressa con soli tre bit:

$num_1 = 0.100_2 * 2^1$ $num_2 = 0.110_2 * 2^1$

distano la quantità: $0.01_2 * 2^1$ (pari a 0,5).

Invece i numeri:

$num_1 = 0.100_2 * 2^4$ $num_2 = 0.110_2 * 2^4$

distano la quantità: $0.01_2 * 2^4$ (pari a 4).

3.1.3 Riassunto dei valori numerici rappresentabili

La tabella seguente riassume i valori rappresentabili nell'elaboratore: nel caso degli interi si ha distinzione in base al numero di bit a disposizione per la notazione in complemento a due, nel caso dei reali si considerano le varianti dello standard IEEE 754.

La terminologia è quella dei tipi dati del linguaggio c, anche se non tutti i tipi elencati sono disponibili con tutte le versioni del linguaggio.

Per i valori reali viene riportato solo l'intervallo positivo, essendo quello per i valori negativi speculare rispetto allo zero.

Tipo	Bit	Valori rappresentabili
unsigned char (caratteri o interi molto corti senza segno)	8	da 0 a 255
char (caratteri o interi molto corti)	8	da -128 a 127
unsigned short int (interi corti senza segno)	16	da 0 a 65.535
short int (interi corti)	16	da -32.768 a 32.767
int (interi)	32	da -2.147.483.648 a 2.147.483.647
float (reali singola precisione)	32	da $1.2*10^{-38}$ a $3.4*10^{38}$
double (reali doppia precisione)	64	da $2.2*10^{-308}$ a $1.8*10^{308}$
long double (reali precisione estesa)	80	da $3.4*10^{-4932}$ a $1.1*10^{4932}$

Precisiamo che per ottenere i valori mostrati nel caso dei reali macchina è stata considerata la presenza del bit nascosto e non sono stati presi in considerazione i valori estremi della caratteristica (-127, 128 o -1023, 1024 o -16383, 16384 a seconda del formato considerato) in quanto servono a denotare valori speciali; ad esempio per individuare l'intervallo per la singola precisione consideriamo i valori da $1.0 * 2^{-126}$ a $1.11111111111111111111111 * 2^{127}$, cioè circa da 2^{-126} a 2^{128}, ottenendo i valori indicati in tabella.

3.2 Rappresentazione dei dati alfanumerici nell'elaboratore

Passiamo adesso ad occuparci della codifica dei dati alfanumerici, non senza avere esaminato qualche nozione sui codici e le codifiche.

3.2.1 Codici

Per *codice* si intende un insieme di parole che sono associate ad un insieme di oggetti da codificare.

Esistono codici naturali (ad esempio le lingue parlate) e codici artificiali, cioè creati dall'uomo per vari scopi.

Ogni codice è basato su un *alfabeto* di simboli che sono usati per creare le *parole* del codice; la *codifica* consiste nell'associare un oggetto dell'insieme da codificare ad una parola di codice, la *decodifica* è il procedimento inverso che permette di risalire da una parola all'oggetto corrispondente.

Spesso, creando una certa ambiguità, di denomina *codice* anche la singola parola di codice; di solito però il contesto in cui i termini sono usati permette di evitare confusione tra gli stessi.

I codici possono essere a *lunghezza fissa* o a *lunghezza variabile* rispettivamente se tutte le parole hanno la stessa lunghezza oppure no.

Un esempio di codice a lunghezza variabile è il codice Morse in cui l'alfabeto è costituito dai simboli di «punto» e «linea».

Quando la lunghezza è variabile ci sono problemi nel procedimento di decodifica dovuti al fatto che la lunghezza delle parole non è uniforme; questo è vero anche nel codice Morse dove è stato introdotto il concetto di «pausa» per separare lettere, parole e frasi facendo in pratica aumentare il numero di simboli dell'alfabeto usato in quel codice.

Torniamo adesso brevemente al concetto di misura dell'informazione introdotto nel paragrafo 1.2; avevamo visto come la quantità di informazione veicolata da un supporto o da un messaggio che prevede N configurazioni equiprobabili con probabilità P = 1/N è:

$$I = \log_2\left(\frac{1}{P}\right) = \log_2\left(\frac{1}{\left(\frac{1}{N}\right)}\right) = \log_2(N)$$

Definiamo ora il concetto di *entropia* di una fonte o sorgente di informazione (qui ci interessa la fonte costituita dai simboli che sono alla base di un codice):

Si dice *entropia* la quantità di informazione totale relativa a tutti i messaggi che la sorgente può emettere (o tutte le configurazioni che il supporto può assumere), calcolata come media ponderata, rispetto alle probabilità, delle quantità di informazioni dei messaggi.

In formula (formula di Shannon):

$$H = \sum_{i=1}^{N} P_k * I_k = \sum_{i=1}^{N} P_k * \log_2\left(\frac{1}{P_k}\right)$$

L'entropia è massima quando tutti gli N messaggi (o tutte le N configurazioni, o tutti gli N simboli dell'alfabeto) sono equiprobabili con probabilità $P_k = 1/N$; in tal caso:

$$H = \sum_{i=1}^{N} \frac{1}{N} * \log_2(N) = \log_2(N)$$

Quindi l'entropia del codice definito con le dieci cifre decimali è:

$$H = \log_2(10) = 3.32$$

Si definisce poi la *lunghezza media di un codice*, misurata in bit per parola (bit/parola), come:

$$L = \sum_{i=1}^{M} Q_i * L_i$$

dove Q_i è la probabilità di una parola di codice e L_i la sua lunghezza.

Se le M parole di un codice sono equiprobabili e della stessa lunghezza, la lunghezza media è pari alla lunghezza di una parola:

$$L = \log_2(M)$$

Il *primo teorema di Shannon* afferma che:

«la lunghezza di un codice non può mai essere inferiore all'entropia, cioè: L >= H»

In caso contrario si avrebbero infatti codici *ambigui* in cui ad una parola di codice corrispondono due o più oggetti dell'insieme da codificare.

Si dice *efficienza* di un codice il rapporto E = H/L

Tale rapporto può essere al massimo pari a 1 e in tal caso si parla di codice *efficiente*; se invece è minore di 1 si ha un codice *ridondante*.

Ad esempio se codifichiamo le sedici cifre esadecimali in binario abbiamo:

- L = 4
- H = 4
- E = 1

Se invece codifichiamo le dieci cifre decimali in binario abbiamo:

- L = 4
- H = 3.32
- E = 0.83

Quindi nel primo caso il codice è efficiente, nel secondo ridondante.

La ridondanza di un codice aumenta ovviamente al diminuire del valore di E.

Esistono, soprattutto riguardo la trasmissione delle informazioni, dei codici che sono volutamente inefficienti (E molto piccola), in quanto prevedono l'introduzione nelle parole di codice di bit aggiuntivi al fine di controllare e, talvolta, correggere errori di codifica e di trasmissione (codici a rilevazione o a correzione di errore); questo argomento non viene comunque qui approfondito.

3.2.2 Il codice BCD

Il codice BCD (*Binary Coded Decimal*) è un codice, proposto all'inizio degli anni '60, con il quale si possono codificare i valori decimali.

Vengono usati 4 bit per ogni cifra (quindi il codice è ridondante, secondo quanto mostrato nel precedente paragrafo) e le parole di codice corrispondono alla traduzione in binario delle varie cifre:

```
0000 = 0
0001 = 1
0010 = 2
0011 = 3
0100 = 4
0101 = 5
0110 = 6
0111 = 7
1000 = 8
1001 = 9
```

Si dice allora che il codice è *ponderato* (i bit in ogni parola hanno un peso, nell'ordine da sinistra: 8, 4, 2, 1).

Altri codici simili, utilizzati sempre per codificare i valori decimali sono il *codice Stibitz* o *codice eccesso 3*:

```
0011 = 0
0100 = 1
0101 = 2
0110 = 3
0111 = 4
1000 = 5
1001 = 6
1010 = 7
1011 = 8
1100 = 9
```

e il *codice Aiken:* o *codice 2421*:

```
0000 = 0
0001 = 1
0010 = 2
0011 = 3
0100 = 4
1011 = 5
1100 = 6
1101 = 7
1110 = 8
1111 = 9
```

Il primo non è ponderato ma è *autocomplementante* in quanto per ottenere il complemento a nove di una cifra basta invertire il valore dei bit (questa è una caratteristica importante in quanto le sottrazioni tra due valori si effettuano come somme tra il primo e il complemento a nove del secondo).

Il codice Aiken è sia autocomplemantante che ponderato (con pesi 2, 4, 2, 1, da cui il nome).

Di questi due codici non forniamo qui ulteriori dettagli.

Tornando al codice BCD possiamo osservare che con esso si possono comodamente rappresentare le cifre di un numero in un *nibble* (4 bit) avendo quindi la possibilità di memorizzare due cifre in ogni byte; in questo caso si parla di *Packed BCD*.

Il segno del numero decimale viene memorizzato nell'ultimo nibble o nell'ultima cifra con la seguente convezione:

Cifra	Bit	Segno
C	1100	+
D	1101	-
F	1111	unsigned

Il valore decimale +134.67 viene quindi memorizzato nel seguente modo su tre byte:

0001 0011 0100 0110 0111 1100

Gli spazi sono inseriti solo per migliorare la leggibilità; il punto frazionario non viene memorizzato perché la quantità di cifre frazionarie viene assunta come fissa e nota a priori.

Esiste anche la rappresentazione *Zoned BCD* in cui ogni cifra viene memorizzata in un byte andando ad occupare il nibble più basso (denominato *digit*) mentre quello più alto (denominato *zonatura*) viene riempito con quattro bit fissi che possono essere:

- 1111, come avviene nel codice EBCDIC;
- 0011, come avviene nel codice ASCII.

L'eventuale segno viene memorizzato nella zonatura della cifra meno significativa con le rappresentazioni illustrate in precedenza.

Per i codici EBCDIC e ASCII si rimanda al prossimo paragrafo, qui notiamo solo che il vantaggio della rappresentazione Zoned BCD è che i valori numerici vengono memorizzati rappresentando in memoria i simboli decimali corrispondenti alle singole cifre; questo aumenta le chiarezza della rappresentazione dei dati in memoria al prezzo di un notevole spreco di bit (un byte per ogni cifra anziché un byte per due cifre).

I numeri Packed BCD hanno un grosso vantaggio rispetto ai numeri espressi con i metodi visti nel paragrafo 3.1 in quanto si possono rappresentare certi valori con maggiore precisione; ad esempio il numero decimale 12.3 in BCD diventa 00010010.0011 (qui il punto frazionario è stato lasciato per chiarezza di esposizione, ma, come detto, non viene memorizzato).

Lo stesso valore in binario «puro» dovrebbe essere necessariamente approssimato in quanto periodico.

Un altro punto a favore del codice BCD è la facilità della conversione del codice di una cifra nel relativo carattere o nella forma utile alla sua rappresentazione nei *display* a sette segmenti.

Questo vantaggi hanno fatto si che il codice Packed BCD (e anche lo Zoned BCD) sia stato largamente utilizzato in tutte le applicazioni, come quelle commerciali, in cui c'è l'esigenza di

memorizzare e gestire valori con poche cifre decimali ma espressi senza errori di rappresentazione; fra il linguaggi di programmazione il COBOL (*COmmon Business Oriented Language*) è quello che fa maggior uso di questi sistemi di codifica dei valori numerici.

Gli elaboratori o i linguaggi che prevedono la memorizzazione di valori decimali packed o zoned e che possiedono istruzioni per la loro manipolazione si dicono elaboratori o linguaggi con *aritmetica decimale*.

Una diffusione maggiore del codice BCD è stata ostacolata da due suoi grossi difetti causati dalla ridondanza:

- «spreco» di bit; il numero di bit usati per rappresentare un valore in BCD è in genere maggiore rispetto al numero necessario per rappresentarlo in binario: ad esempio 129 in binario è 10000001_2 mentre in BCD è 000100101001.
- maggiore complessità delle operazioni con necessità di dotare i microprocessori di istruzioni apposite per i valori BCD.

Chiariamo meglio il secondo aspetto con un esempio in cui si sommano i valori 77 e 11, 88 e 4 e 68 e 19 rappresentati in BCD:

```
01110111+
00010001=
--------
10001000   = 88

10001000+
00000100=
--------
10001100   = 8?

01101000+
00011001=
--------
10000001   = 81 ??
```

Nel primo caso il risultato è corretto, nel secondo si ha una cifra (quella a destra) «fuori codice» e il terzo risultato è palesemente errato (la causa è che c'è stato un riporto tra le due cifre decimali).

Si rende quindi necessario intervenire con il cosiddetto *aggiustamento decimale* o *correzione decimale* che consiste nel sommare il valore 0110 (sei in BCD) alla cifra fuori codice e alla cifra che ha fornito il prestito; se dopo tale procedimento si hanno ancora cifre fuori codice occorre riapplicare ad esse la correzione (non però se si hanno riporti fra cifre decimali durante l'aggiustamento).

Questo modo di operare è giustificato dal fatto che usando quattro bit il riporto non si ha al raggiungimento del valore 10, bensì del valore 16, cioè con 6 unità di ritardo.

Effettuando la correzione decimale alle due operazioni errate si ottiene:

```
10001100+
    0110=
--------
10010010   = 92

10000001+
    0110=
--------
10000111   = 87
```

Vediamo un ulteriore esempio in cui si sommano i valori 423 e 879 in BCD apportando subito le necessarie correzioni decimali:

```
 010000100011+
 100001111001=
 ------------
 110010011100+
 0110    0110=
-------------
1001010100010+
     0110    =
-------------
1001100000010   = 1302
```

La differenza in BCD si effettua come somma del primo operando con il complemento a 10 del secondo operando (scartando la cifra in più a sinistra nel risultato); ad esempio 85 - 49 diventa 85 + 51 e quindi:

```
  10000101+
  01010001=
  --------
  11010110+
  0110    =
  --------
1 00110110  = 36
```

3.2.3 I codici EBCDIC e ASCII

Il codice EBCDIC (*Extended Binary Coded Decimal Interchange Code*) fu proposto dall'IBM negli anni '60 con l'avvento degli elaboratori della serie 360; non è mai divenuto uno standard ma è stato utilizzato per tutti i grossi elaboratori IBM.

Esso nacque come estensione di un codice della stessa IBM, denominato BCD, che però era diverso dal BCD di cui abbiamo già parlato, in quanto prevedeva parole di sei bit ed era in grado di codificare le cifre decimali, le lettere maiuscole dell'alfabeto inglese e alcuni caratteri speciali o di controllo.

Un altro codice con parole di sei bit, molto famoso e utilizzato in quegli anni era il *FIELDDATA* degli elaboratori della Univac.

Il codice EBCDIC è a lunghezza fissa ed è efficiente in quanto rappresenta 256 simboli alfanumerici con parole lunghe otto bit; tra l'altro è anche compatibile con il *codice Hollerith* usato per le schede perforate nei primi decenni dell'informatica.

In ogni parola di codice i due nibble che compongono il byte prendono il nome di: *zonatura*, quello più significativo, e *digit*, quello meno significativo; in pratica la zonatura permette di

distinguere i tipi di simboli (ad esempio i simboli relativi alle cifre decimali hanno tutti zonatura 1111), mentre il digit di designare i vari simboli all'interno di ogni tipo.

Nel codice EBCDIC si ha prima la codifica dei caratteri di controllo, poi dei simboli di punteggiatura e accessori, quindi delle minuscole, delle maiuscole (ovviamente in ordine alfabetico e infine delle cifre decimali.

Nella figura 3.41 è riportata la tabella EBCDIC.

Figura 3.41

Dec	Hx	Oct	Char		Dec	Hx	Oct	Char	Dec	Hx	Oct	Char	Dec	Hx	Oct	Char
0	0	000	nul	(Null)	65	41	101		130	82	202	b	195	c3	303	C
1	1	001	soh	(Start of Heading)	66	42	102		131	83	203	c	196	c4	304	D
2	2	002	stx	(Start of Text)	67	43	103		132	84	204	d	197	c5	305	E
3	3	003	etx	(End of Text)	68	44	104		133	85	205	e	198	c6	306	F
4	4	004	pf	(Punch Off)	69	45	105		134	86	206	f	199	c7	307	G
5	5	005	ht	(Horizontal Tab)	70	46	106		135	87	207	g	200	c8	310	H
6	6	006	lc	(Lower Case)	71	47	107		136	88	210	h	201	c9	311	I
7	7	007	del	(Delete)	72	48	110		137	89	211	i	202	ca	312	
8	8	010	ge		73	49	111		138	8a	212		203	cb	313	
9	9	011	rlf		74	4a	112	¢	139	8b	213		204	cc	314	
10	a	012	smm	(Start of Manual Message)	75	4b	113	.	140	8c	214		205	cd	315	
11	b	013	vt	(Vertical Tab)	76	4c	114	>	141	8d	215		206	ce	316	
12	c	014	ff	(Form Feed)	77	4d	115	(	142	8e	216		207	cf	317	
13	d	015	cr	(Carriage Return)	78	4e	116	+	143	8f	217		208	d0	320	}
14	e	016	so	(Shift Out)	79	4f	117	|	144	90	220		209	d1	321	J
15	f	017	si	(Shift in)	80	50	120	&	145	91	221	j	210	d2	322	K
16	10	020	dle	(Data Link Escape)	81	51	121		146	92	222	k	211	d3	323	L
17	11	021	dc1	(Device Control 1)	82	52	122		147	93	223	l	212	d4	324	M
18	12	022	dc2	dc2 (Device Control 2)	83	53	123		148	94	224	m	213	d5	325	N
19	13	023	tm	(Tape Mark)	84	54	124		149	95	225	n	214	d6	326	O
20	14	024	res	(Restore)	85	55	125		150	96	226	o	215	d7	327	P
21	15	025	nl	(New Line)	86	56	126		151	97	227	p	216	d8	330	Q
22	16	026	bs	(Backspace)	87	57	127		152	98	230	q	217	d9	331	R
23	17	027	il	(Idle)	88	58	130		153	99	231	r	218	da	332	
24	18	030	can	(Cancel)	89	59	131		154	9a	232		219	db	333	
25	19	031	em	(End of Medium)	90	5a	132	!	155	9b	233		220	dc	334	
26	1a	032	cc	(Cursor Control)	91	5b	133	$	156	9c	234		221	dd	335	
27	1b	033	cu1	(Customer Use 1)	92	5c	134	*	157	9d	235		222	de	336	
28	1c	034	ifs	(Interchange File Separator)	93	5d	135	)	158	9e	236		223	df	337	
29	1d	035	igs	(Interchange Group Separator)	94	5e	136	;	159	9f	237		224	e0	340	\
30	1e	036	irs	(Interchange Record	95	5f	137		160	a0	240		225	e1	341	
31	1f	037	ius	(Interchange Unit Separator)	96	60	140	-	161	a1	241	~	226	e2	342	S
32	20	040	ds	(Digit Select)	97	61	141	/	162	a2	242	s	227	e3	343	T
33	21	041	sos	(Start of Significance)	98	62	142		163	a3	243	t	228	e4	344	U
34	22	042	fs	(Field Separator)	99	63	143		164	a4	244	u	229	e5	345	V
35	23	043			100	64	144		165	a5	245	v	230	e6	346	W
36	24	044	byp	(Bypass)	101	65	145		166	a6	246	w	231	e7	347	X
37	25	045	lf	(Line Feed)	102	66	146		167	a7	247	x	232	e8	350	Y
38	26	046	etb	(End of Transmission Block)	103	67	147		168	a8	250	y	233	e9	351	Z
39	27	047	esc	(Escape)	104	68	150		169	a9	251	z	234	ea	352	
40	28	050			105	69	151		170	aa	252		235	eb	353	
41	29	051			106	6a	152	|	171	ab	253		236	ec	354	
42	2a	052	sm	(Set Mode)	107	6b	153	,	172	ac	254		237	ed	355	
43	2b	053	cu2	(Customer Use 2)	108	6c	154	%	173	ad	255		238	ee	356	
44	2c	054			109	6d	155		174	ae	256		239	eF	357	
45	2d	055	enq	(Enquiry)	110	6e	156	<	175	af	257		240	f0	360	0
46	2e	056	ack	(Acknowledge)	111	6f	157	?	176	b0	260		241	f1	361	1
47	2f	057	bel	(Bell)	112	70	160		177	b1	261		242	f2	362	2
48	30	060			113	71	161		178	b2	262		243	f3	363	3
49	31	061			114	72	162		179	b3	263		244	f4	364	4
50	32	062	syn	(Synchronous Idle)	115	73	163		180	b4	264		245	f5	365	5
51	33	063			116	74	164		181	b5	265		246	f6	366	6
52	34	064	pn	(Punch On)	117	75	165		182	b6	266		247	f7	367	7
53	35	065	rs	(Reader Stop)	118	76	166		183	b7	267		248	f8	370	8
54	36	066	uc	(Upper Case)	119	77	167		184	b8	270		249	f9	371	9
55	37	067	eot	(End of Transmission)	120	78	170		185	b9	271		250	fa	372	|
56	38	070			121	79	171	`	186	ba	272		251	fb	373	
57	39	071			122	7a	172	:	187	bb	273		252	fc	374	
58	3a	072			123	7b	173	#	188	bc	274		253	fd	375	
59	3b	073	cu3	(Customer Use 3)	124	7c	174	@	189	bd	275		254	fe	376	
60	3c	074	dc4	(Device Control 4)	125	7d	175	'	190	be	276		255	ff	377	eo
61	3d	075	nak	(Negative Acknowledge)	126	7e	176	=	191	bf	277					
62	3e	076			127	7f	177	"	192	c0	300	{				
63	3f	077	sub	(Substitute)	128	80	200		193	c1	301	A				
64	40	100	Sp	(Space)	129	81	201	a	194	c2	302	B				

Source: www.asciitable.com

Il codice ASCII (*American Standard Code for Information Interchange*) è invece uno standard approvato dallo ANSI (*American National Standard Institute*) nel 1963 e fatto proprio dalla ISO (*International Standard Organization*) come ISO 646.

La prima versione dello standard prevedeva parole di codice lunghe sette bit; il bit eccedente nel byte veniva utilizzato come *bit di parità* per una blanda forma di controllo sulla correttezza dei sette bit della codifica.

Tale bit veniva infatti posto a zero o uno in modo da far risultare pari la quantità di bit impostati al valore uno in ogni byte; in questo modo un errore di rappresentazione o di trasmissione dati in un byte che facesse diventare dispari tale quantità, poteva essere rilevato (il controllo era però inefficace nel caso di due errori sullo stesso byte che si compensavano lasciando pari la quantità di bit con valore uno).

I simboli codificabili in ASCII ISO 646 sono 128 e si suddividono in:

- *riproducibili*, cioè che si trovano sulle tastiere e possono essere stampati a video o su carta;
- *non riproducibili*, o di controllo o non stampabili la cui presenza deriva dall'importanza che buona parte di essi avevano per gestire le telescriventi, apparecchiature molto usate ai tempi della definizione dello standard (comunque molti di questi simboli conservano anche oggi una notevole utilità); essi, a loro volta, si suddividono in:
  - simboli di trasmissione, ad esempio: STX = Start Of Text (inizio testo) o ACK = Acknowledge (conferma positiva);
  - simboli di formato, ad esempio: LF = Line Feed (nuova linea), FF = Form Feed, (nuova pagina), CR = Carriage Return (ritorno a capo);
  - simboli ausiliari, ad esempio: DEL = Delete (cancella carattere), BEL = Bell (emissione di un suono), NUL = Null (carattere nullo).

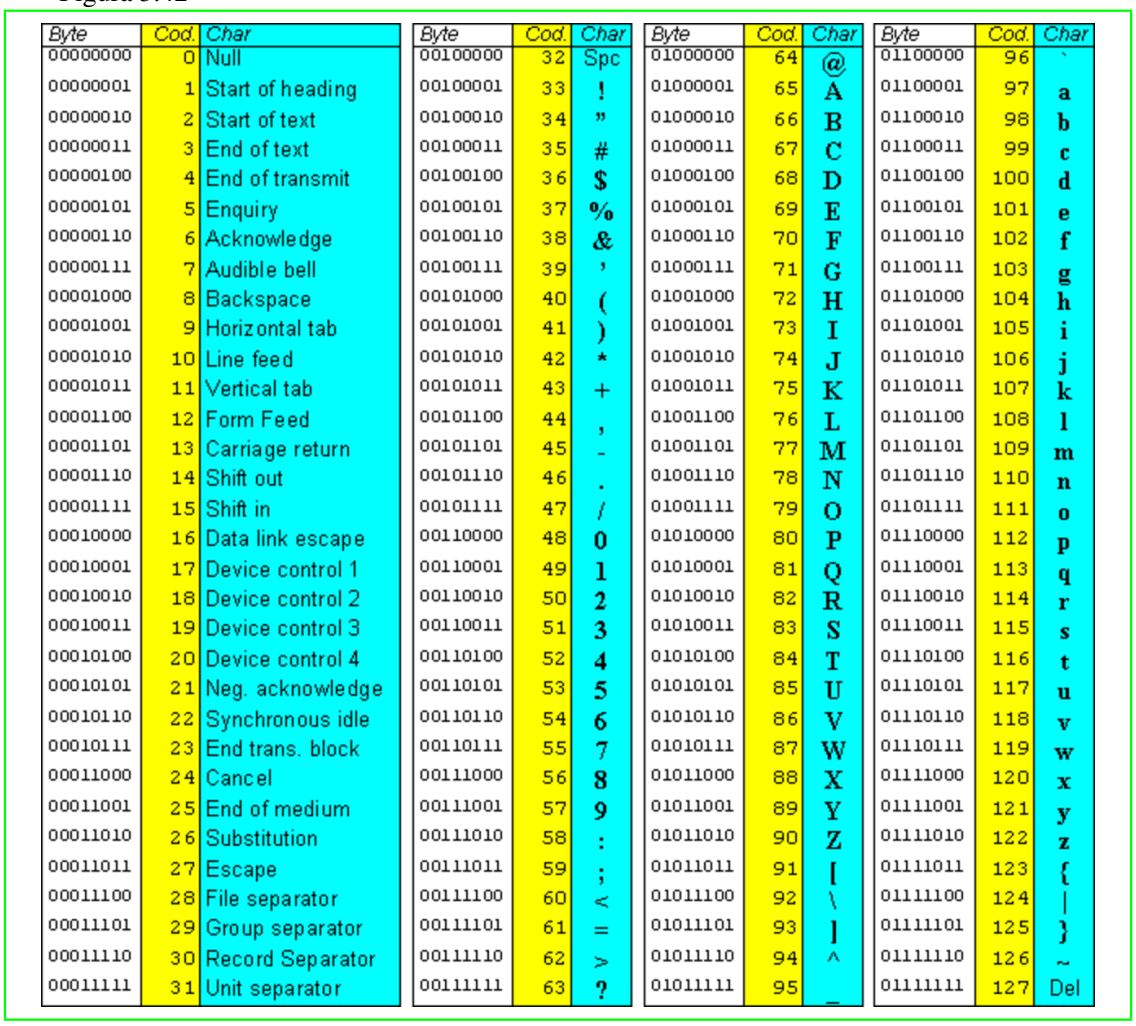
I simboli non riproducibili nella tabella dei codici ASCII sono i primi trentadue, quelli il cui codice tradotto in valore decimale è compreso tra 0 e 31.

Seguono poi, inframezzati a simboli vari e di punteggiatura, le cifre decimali, le lettere maiuscole e le lettere minuscole.

Si noti che la sequenza è diversa da quella presente in EBCDIC; questo significa che l'ordine «lessicografico» dei dati nel computer prevede la precedenza di stringhe maiuscole rispetto a minuscole in ASCII e il viceversa in EBCDIC; da questo e soprattutto dal fatto che le parole di codice associate ai vari simboli sono diverse, discende che i due codici sono incompatibili.

Il codice ASCII è usato da tutti i produttori diversi da IBM e anche dalla stessa IBM nel caso dei Personal Computer; in figura 3.42 vediamo la tabella dei codici nella quale notiamo che le parole di codice sono lunghe otto bit: si tratta in effetti della prima metà della tabella *ASCII estesa* contenente gli stessi codici di quella standard ma con un bit a zero in più a sinistra.

Figura 3.42



Byte	Cod.	Char
00000000	0	Null
00000001	1	Start of heading
00000010	2	Start of text
00000011	3	End of text
00000100	4	End of transmit
00000101	5	Enquiry
00000110	6	Acknowledge
00000111	7	Audible bell
00001000	8	Backspace
00001001	9	Horizontal tab
00001010	10	Line feed
00001011	11	Vertical tab
00001100	12	Form Feed
00001101	13	Carriage return
00001110	14	Shift out
00001111	15	Shift in
00010000	16	Data link escape
00010001	17	Device control 1
00010010	18	Device control 2
00010011	19	Device control 3
00010100	20	Device control 4
00010101	21	Neg. acknowledge
00010110	22	Synchronous idle
00010111	23	End trans. block
00011000	24	Cancel
00011001	25	End of medium
00011010	26	Substitution
00011011	27	Escape
00011100	28	File separator
00011101	29	Group separator
00011110	30	Record Separator
00011111	31	Unit separator

Byte	Cod.	Char
00100000	32	Spc
00100001	33	!
00100010	34	"
00100011	35	#
00100100	36	$
00100101	37	%
00100110	38	&
00100111	39	'
00101000	40	(
00101001	41	)
00101010	42	*
00101011	43	+
00101100	44	,
00101101	45	-
00101110	46	.
00101111	47	/
00110000	48	0
00110001	49	1
00110010	50	2
00110011	51	3
00110100	52	4
00110101	53	5
00110110	54	6
00110111	55	7
00111000	56	8
00111001	57	9
00111010	58	:
00111011	59	;
00111100	60	<
00111101	61	=
00111110	62	>
00111111	63	?

Byte	Cod.	Char
01000000	64	@
01000001	65	A
01000010	66	B
01000011	67	C
01000100	68	D
01000101	69	E
01000110	70	F
01000111	71	G
01001000	72	H
01001001	73	I
01001010	74	J
01001011	75	K
01001100	76	L
01001101	77	M
01001110	78	N
01001111	79	O
01010000	80	P
01010001	81	Q
01010010	82	R
01010011	83	S
01010100	84	T
01010101	85	U
01010110	86	V
01010111	87	W
01011000	88	X
01011001	89	Y
01011010	90	Z
01011011	91	[
01011100	92	\
01011101	93	]
01011110	94	^
01011111	95	_

Byte	Cod.	Char
01100000	96	`
01100001	97	a
01100010	98	b
01100011	99	c
01100100	100	d
01100101	101	e
01100110	102	f
01100111	103	g
01101000	104	h
01101001	105	i
01101010	106	j
01101011	107	k
01101100	108	l
01101101	109	m
01101110	110	n
01101111	111	o
01110000	112	p
01110001	113	q
01110010	114	r
01110011	115	s
01110100	116	t
01110101	117	u
01110110	118	v
01110111	119	w
01111000	120	x
01111001	121	y
01111010	122	z
01111011	123	{
01111100	124	|
01111101	125	}
01111110	126	~
01111111	127	Del

Esiste infatti, dal 1985, una versione più moderna del codice ASCII, chiamata *ASCII estesa* o ISO 8859, in cui la lunghezza delle parole è otto bit.

I simboli codificabili diventano così 256 e questo permette di codificare correttamente le lettere degli alfabeti diversi da quello inglese, latino e swahili (che sono privi di lettere accentate) i soli per i quali è sufficiente la versione a sette bit.

C'è poi la possibilità di inserire nel codice anche molti altri simboli fra cui ad esempio quelli grafici; inoltre la perdita del bit di parità non è importante in quanto il controllo realizzato con esso non era molto sofisticato e affidabile.

L'estensione del codice è stata fatta in modo «intelligente» lasciando inalterati i primi 128 simboli (a parte l'aumento di lunghezza delle parole) in modo da assicurare la «retro-compatibilità» con la versione precedente del codice.

Sono stati poi creati un certo numero di sottostandard, che si differenziano per gli ultimi 96 degli altri 128 simboli (i primi 32 sono ulteriori caratteri di controllo comuni a tutte le varianti dello standard).

Ognuno dei sottostandard è dedicato ad una particolare lingua o a un gruppo di lingue; abbiamo quindi:

- ISO 8859-1 (Latin-1 Western European): per le lingue dell'Europa Occidentale e della Scandinavia (quindi anche delle Americhe, dell'Oceania dell'India e altre ex colonie europee);

- ISO 8859-2 (Latin-2 Central European): per le lingue dell'Europa centro orientale basate sull'alfabeto latino;
- ISO 8859-3 (Latin-3 South European): per maltese, turco e esperanto;
- ISO 8859-4 (Latin-4 North European): lingue dei paesi baltici e della Groenlandia;
- ISO 8859-5 (Latin/Cyrillic): lingue basate su alfabeto cirillico;
- ISO 8859-6 (Latin/Arabic): arabo;
- ISO 8859-7 (Latin/Greek): greco moderno;
- ISO 8859-8 (Latin/hebrew): ebraico moderno;
- ISO 8859-9 (Latin-5 Turkish): come Latin-1 con l'islandese rimpiazzato dal turco;
- ISO 8859-10 (Latin-6 Nordic): modifica del Latin-4 per i linguaggi nordici;
- ISO 8859-11 (Latin/Thai): linguaggio thai;
- ISO 8859-13 (Latin-7 Baltic Rim): aggiunta di caratteri mancanti in Latin-4 e Latin-6;
- ISO 8859-14 (Latin-8 Celtic): lingua celtica;
- ISO 8859-15 (Latin-9): revisione del Latin-1 con l'introduzione, tra l'altro del simbolo dell'Euro;
- ISO 8859-16 (Latin-10 South-Eastern European): definito per alcune lingue tra cui italiano, sloveno, francese, tedesco con il simbolo di valuta sostituito da quello dell'Euro.

3.2.4 I codici UCS-2 e UTF-8

Lo standard ISO 8859, con le sue varianti, copre le necessità di molte delle lingue esistenti; rimangono però fuori tutti quei linguaggi che non sono basati su un alfabeto ma su *ideogrammi* ognuno dei quali può corrispondere ad una parola o addirittura ad una frase.

Per indicare questi linguaggi si usa l'acronimo CJK (*Chinese, Japanese, Korean*) dai nomi delle lingue più rappresentative in questo ambito.

La soluzione probabilmente definitiva ai problemi di codifica si ha con la creazione, a partire dal 1991, dell'insieme di caratteri UCS (*Universal Character Set*), standard ISO 10646, meglio noto come *Unicode*.

Con questo insieme di caratteri si associa ad ogni carattere o simbolo di ogni lingua del mondo un numero intero detto *code point* o punto di codifica.

L'insieme è in continua evoluzione e attualmente conta molte decine di migliaia di codici: sono considerati tutti gli alfabeti conosciuti, anche quelli «inventati» come il Klingon della serie televisiva Star Trek.

I punti di codifica si indicano di solito con la notazione U+xxxx, con «xxxx» valore esadecimale del codice; ad esempio il simbolo di «marchio registrato» ® si indica con U+00AD e quello del «copyright» © con U+00A9.

Grazie ad Unicode vengono definite anche le regole di composizione in modo da formare nuovi simboli partendo da da due già esistenti (ad esempio õ da «o» e «~»).

Unicode però non specifica come i simboli debbano essere codificati (che è proprio la cosa più importante dal punto di vista del sistema di elaborazione); a questo scopo esistono le *codifiche* che definiscono le associazioni tra codici e loro rappresentazioni basate su bit.

Le codifiche più importanti e utilizzate sono:

- *UCS-2* che usa due byte per codificare ogni simbolo ed è utilizzata dal 1995 nei sistemi operativi della Microsoft e dal linguaggio di programmazione Java; non è pienamente retro-compatibile con il codice ASCII perché, sebbene i primi 128 simboli siano gli stessi, la codifica di ogni simbolo occupa il doppio dei bit;
- UTF-8 (*Unicode Transformation Format 8*), utilizzata nei sistemi Unix e Linux e riconosciuta come standard per le comunicazioni in Internet; è retro-compatibile con ASCII perché i primi 128 simboli, oltre ad essere gli stessi, sono codificati con parole di 8 bit; gli altri codici sono più lunghi (fino a sei byte) e quindi siamo in presenza di una codifica a lunghezza variabile, più complessa da gestire.

La cifra 8 che appare nel nome significa che la base della codifica è il byte (8 bit); esistono anche UTF-7, UTF-16 (estensione di UCS-2) e UCS-4 ma sono molto meno utilizzate.

Come detto, in UTF-8 i primi 128 simboli sono codificati con un byte; quelli da 128 a 2047 invece con due byte (e sono i simboli sufficienti a coprire tutte le lingue occidentali) mentre i simboli da 2048 a 65535 occupano tre byte (ideogrammi e altri simboli vari).