Rappresentazione dell'informazione in un calcolatore:

Informazioni

- testi, numeri interi e reali, immagini, , suoni, etc.;

Come viene rappresentata l'informazione in un calcolatore?

- uso di tecnologia **digitale**: all'interno della CPU le informazioni vengono rappresentate da 2 possibili valori di tensione elettrica {V_{high}, V_{low}}

- In generale, a seconda del tipo di dispositivo considerato, i valori zero e uno sono rappresentati:
  - da una tensione elettrica (alta, bassa);
  - da un differente stato di polarizzazione magnetica (positiva, negativa);
  - da luce e buio;
  - etc.

- Unità di informazione nel calcolatore: **bit**.

☞ Ogni informazione viene trasformata nel calcolatore in una sequenza di bit (forma **BINARIA**), cioè in una sequenza di 0 e 1.

00011010..

Codifica dei numeri

Il sistema di numerazione che utilizziamo si dice **arabico** e fu introdotto in Europa dagli arabi nel Medio Evo.

- **È decimale** (o in base 10): esso rappresenta i numeri tramite sequenze di cifre che vanno da 0 a 9 (dieci cifre).

- **È posizionale:** il peso attribuito ad ogni cifra è funzione della posizione che occupa.

  (Esistono anche sistemi **additivi**, in cui ogni unità è rappresentata da un unico simbolo o non-posizionali come quello romano).

☞ I sistemi posizionali consentono di rappresentare numeri grandi con un numero limitato di cifre, e di svolgere su di essi calcoli più efficienti.

I sistemi di numerazione posizionale sono caratterizzati da una <u>base</u> b e un <u>alfabeto</u> α:

- **Alfabeto** (α): è l'insieme delle cifre disponibili per esprimere i numeri. A ogni cifra corrisponde un valore compreso tra 0 e (b-1).
  (Ad esempio, nella numerazione decimale l'alfabeto è α={0,1,2,3,4,5,6,7,8,9})

- **Base** (b): è il numero degli elementi che compongono l'alfabeto. Ad esempio, nel caso decimale, b=10.

Esempi:

Base 8 α={0,1,2,3,4,5,6,7}

Base 16 α={0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}

Numerazione in base p

α={0,1,2,...p-1} ,

b=p

un numero generico N in base p è rappresentato da una sequenza di cifre:

$$\mathbf{a_n\ a_{n-1}\ ...\ a_1\ a_0}$$

dove $\mathbf{a_i} \in \alpha$, $\forall$i=0,..n.

(**a_n** è la cifra più significativa, mentre **a_0** è la meno significativa)

Codifica dei Numeri Naturali

Consideriamo l'insieme dei numeri naturali.

- Dato un sistema di numerazione posizionale <α, b>:
  α={0,1,2,...p-1} ,
  b=p

- Sia $\mathbf{a_n a_{n-1} \ldots a_1 a_0}$ la codifica di un numero naturale N in base p; allora il valore di N, in base decimale è dato dalla formula:

$$N_p = \mathbf{a_n} * p^n + \mathbf{a_{n-1}} * p^{n-1} + \ldots \mathbf{a_1} * p^1 + \mathbf{a_0}$$

o, in forma più compatta:

$$N_p = \bullet_{i=0,..n}\ a_i * p^i$$

☞ Con **n** cifre in base **p** è possibile rappresentare $\mathbf{p^n}$ numeri naturali diversi da 0 a p^{n-1} (i due limiti si ottengono sostituendo a tutti gli n coefficienti ai 0 o p-1 rispettivamente).

Infatti, il numero massimo si ottiene utilizzando la cifra massima (p-1) per ogni posizione:

$$\mathbf{(p-1)*p^{n-1} + (p-1)*p^{n-2} + \ldots (p-1)*p^1 + (p-1) = p^n - 1}$$

ESEMPI

- **Conversione di un numero da base b a base decimale:**

Esempio 1: Codifica decimale. **b=10, N_{10}=5870**

$$5870 = 5 \cdot 10^3 + 8 \cdot 10^2 + 7 \cdot 10^1 + 0 \cdot 10^0$$

Esempio 2: Codifica **binaria**. **b=2,** α={0,1}

N_2=101001011

$$101001011_2=(1 \cdot 2^8 + 0 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3+0 \cdot 2^2+1 \cdot 2^1+1.2^0)_{10}= (331)_{10}$$

Esempio3: Codifica ottale. **b=8,** α={0,1,2,3,4,5,6,7}

N_8=534

$$(534)_8= (5 \cdot 8^2+3 \cdot 8^1+4)_{10}= 348_{10}$$

Esempio3: Codifica esadecimale:
b=16
α={0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}

N_{16}=B7F

$$B7F_{16}= (11 \cdot 16^2+7 \cdot 16^1+15)_{10}= (2943)_{10}$$

Conversione di un numero naturale in base 10, in base non decimale

La formula

$$N_p = \mathbf{a_n * p^n + a_{n-1} * p^{n-1} + ... a_1 * p^1 + a_0}$$

si può riscrivere come:

$$N_p = a_0 + p * (a_1 + p * (...+p * (a_{n-1} + a_n * p))..)$$

- Eseguendo la **divisione intera** per p:

  $Np \text{ div } p = (a_1 + p * (... p * (a_{n-1} + a_n * p))..) = Q_1$

  $Np \text{ mod } p = \mathbf{a_0}$ [resto della divisione intera]

- Applichiamo la **divisione intera** per p sul risultato Q_1 della divisione precedente:

  $Q_1 \text{ div } p = (a_2 + p . (...p . (a_{n-1} + a_n . p))..) = Q_2$

  $Q_1 \text{ mod } p = \mathbf{a_1}$ [resto della divisione intera]

- Ripetiamo il procedimento su Q_2, Q_3, etc. per ottenere le cifre rimanenti (a_2, a_3, .. a_n).

☞ In pratica, il procedimento da seguire è il seguente:

Algoritmo delle divisioni successive

Sia N il numero.

1. Si divide N per la nuova base p; sia Q il quoziente e R il resto.
2. Si converte R nella corrispondente cifra della nuova base p.
3. Si aggiunge la cifra così ottenuta a sinistra delle cifre ottenute in precedenza.
4. Se Q=0, fine; Altrimenti poni N = Q e torna al passo 1.

Conversione binaria

Si vuole convertire un numero N (in base 10), nella corrispondente rappresentazione in base 2.

- Applicando il procedimento visto, bisogna effettuare successive divisioni per 2.

- Il risultato è la sequenza di 0 e 1 ottenuti considerando i resti delle divisioni dalla meno significativa alla più significativa.

Esempio: Convertire in forma binaria $\mathbf{N_{10}=331}$

Divisione	Quoziente	Resto (a_i)	
331 : 2	165	1	$\mathbf{a_0}$
165 : 2	82	1	$\mathbf{a_1}$
82 : 2	41	0	$\mathbf{a_2}$
41 :2	20	1	$\mathbf{a_3}$
20 : 2	10	0	$\mathbf{a_4}$
10 : 2	5	0	$\mathbf{a_5}$
5 : 2	2	1	$\mathbf{a_6}$
2 : 2	1	0	$\mathbf{a_7}$
1 : 2	0	1	$\mathbf{a_8}$

quindi: $(331)_{10} = (101001011)_2$

Esempio: Convertire in forma binaria $N_{10}=44$

Divisione	Quoziente	Resto (a_i)
44 : 2	22	0
22 : 2	11	0
11 : 2	5	1
5 : 2	2	1
2 : 2	1	0
1 : 2	0	1

quindi: $(44)_{10} = (101100)_2$

Conversione in base p $\neq$ 2

Esempio: Convertire in forma ottale $N_{10}=44$

Divisione	Quoziente	Resto (a_i)
44 : 8	5	4
5 : 8	0	5

quindi: $(44)_{10} = (54)_8$

Esempio: Convertire in forma esadecimale $N_{10}=44$

Divisione	Quoziente	Resto (a_i)
44 : 16	2	12 (C_{16})
2 : 16	0	2

quindi: $(44)_{10} = (2C)_{16}$

Tabella Riassuntiva

Sistema di numerazione

Decimale	Binario	Ottale	Esadecimale
0	0000	00	0
1	0001	01	1
2	0010	02	2
3	0011	03	3
4	0100	04	4
5	0101	05	5
6	0110	06	6
7	0111	07	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Conversioni da forma binaria a ottale ed esadecimale.

Le rappresentazioni ottali ed esadecimali sono interessanti per la facilità di conversione dalla base due, e viceversa.

- Osserviamo che:

$$\mathbf{8 = 2^3}$$
$$\mathbf{16 = 2^4}$$

☞ La conversione da base 2 a base 8 si ottiene scomponendo il numero binario in **triple** di cifre binarie (partendo dalla meno significativa), e per ogni tripla ricavando la corrispondente cifra ottale.

$(101100)_2=(101.100)_2$
$(101)_2=(5)_8$
$(100)_2=(4)_8$

quindi $(101100)_2=(54)_8$

☞ La conversione da base 2 a base 16 si ottiene scomponendo il numero binario in **quadruple** di cifre binarie (partendo dalla meno significativa), e per ogni quadrupla ricavando la corrispondente cifra ottale.

$(101100)_2=(0010.1100)_2$
$(0010)_2=(2)_{16}$
$(1100)_2=(C)_{16}$

quindi $(101100)_2=(2C)_{16}$

Aritmetica Binaria: operazioni sui naturali

Definizione delle operazioni aritmetiche elementari:

A	B	r/p	riporto	Somma	prestito	Diff.
0	0	0	0	0	0	0
0	1	0	0	1	1	1
1	0	0	0	1	0	1
1	1	0	1	0	0	0
0	0	1	0	1	1	1
0	1	1	1	0	1	0
1	0	1	1	0	0	0
1	1	1	1	1	1	1

(r/p esprime il riporto/prestito della colonna precedente)

Somma di due numeri binari naturali:

Viene eseguita incolonnando i numeri e sommando tra loro i bit incolonnati, partendo dai meno significativi, in ordine di peso crescente.

☞ Per la somma di due numeri positivi di lunghezza K possono essere necessari K+1 posti. Se sono disponibili solo K cifre si genera un errore di **overflow** (o trabocco).

Esempio: $(11011)_2+(00110)_2$

```
  11011 +   (27)
  00110 =   (6)
  --------
 100001     (33)
```

Il bit più alto (***bit di carry***) corrisponde a un bit del registro Program Status Word (PSW).

Sottrazione di numeri binari naturali:

Viene eseguita incolonnando i numeri e sottraendo tra loro i bit incolonnati, partendo dai meno significativi, in ordine di peso crescente.

Hp: si suppone che si generi sempre un numero positivo

```
1100 -    (12)
0011 =    (3)
-----
1001 (9)
```

Moltiplicazione di numeri binari naturali:

Si utilizza la stessa tecnica usata anche per i numeri in base 10 (somma e scorrimento):

$A = (1011)_2 = (11)_{10}$
$B = (1101)_2 = (13)_{10}$

$A * B = 1011 * 1101 =$

```
    1011   +
   00000   +
  101100   +
 1011000
---------
10001111
```

Divisione di numeri binari naturali:

Si usa la tecnica usata anche per i numeri in base 10 (differenza e scorrimento).

$A=(101101)_2=(45)_{10}$
$B=(11)_2=(3)_{10}$

Calcolare A/B:

```
101101     |11
00         |01111
--
101
011
---
0101
 011
----
  100
  011
-----
   011
   011
------
   000
```

Il risultato della operazione A/B è quindi $(1111)_2=(15)_{10}$

ESERCIZI:

Effettuare le seguenti operazioni aritmetiche tra numeri binari naturali, ipotizzando di lavorare con un elaboratore con lunghezza di parola (*word*) pari a un byte:

- **Differenza**: A-B, $A=(35)_{10}$, $B=(22)_{10}$

$(35)_{10} = (23)_{16} = (0010\ 0011)_2$
$(22)_{10} = (16)_{16} = (0001\ 0110)_2$

```
0010 0011 -
0001 0110 =
-----------
0000 1101      Prestito: 0
```

Risultato: $A - B = (0000\ 1101)_2 = (0D)_{16} = (13)_{10}$

- **Somma**: A+B, $A=(42)_{10}$, $B=(31)_{10}$

$(42)_{10} = (2A)_{16} = (0010\ 1010)_2$
$(31)_{10} = (1F)_{16} = (0001\ 1111)_2$

```
0010 1010 +
0001 1111 =
-----------
0100 1001
```

Risultato: $A + B = (0100\ 1001)_2 = (49)_{16} = (73)_{10}$

- **Moltiplicazione**: A∗B, A=(7)$_{10}$, B=(12)$_{10}$

  (7)$_{10}$ = (7)$_{16}$ = (0000 0111)$_2$
  (12)$_{10}$= (0C)$_{16}$ = (0000 1100)$_2$

```
          00000111 *
          00001100 =
          ---------
          00000000
         00000000-
        00000111-
      000000111-
     00000000
    00000000
   00000000
 00000000
 ------------------
 000000001010100
```

Risultato: (0101 0100)$_2$ = (54)$_{16}$ = (84)$_{10}$

- **Divisione**: A/B, A=(23)$_{10}$, B=(7)$_{10}$

  (23)$_{10}$= (17)$_{16}$ = (0001 0111)$_2$
  (7)$_{10}$ = (7)$_{16}$ = (0000 0111)$_2$

```
10111       |111
000         |011
--
1011
0111
----
01001
 0111
-----
 0010
```

Risultato: Quoziente = (0000 0011)$_2$ = (3)$_{10}$
Resto = (0000 0010)$_2$ = (2)$_{10}$

Numeri Relativi

Consideriamo l'insieme dei numeri relativi (**interi**)

$$Z=\{-\infty, ..., -1, 0, +1,... +\infty\}$$

Vogliamo rappresentare gli elementi di questo insieme in forma binaria.

Avendo sempre un numero limitato di bit, possiamo utilizzarne uno per la rappresentazione del segno (+ o -).

Rappresentazione con modulo e segno

Il primo bit di un numero intero viene utilizzato come bit di segno (0 positivo, 1 negativo). Gli altri bit indicano il modulo (valore assoluto) del numero

Ad esempio:

- parole di 5 bit:

  +5 = 00101
  -10 = 11010

- parole a 16 bit:

  +13 = 0000000000001101
  -13 = 1000000000001101

Rappresentazione con modulo e segno

☞ Tramite m cifre in base 2 è possibile rappresentare $\mathbf{2^m-1}$ numeri diversi, da $\mathbf{-(2^{m-1}-1)}$ a $\mathbf{+(2^{m-1}-1)}$.

Ad esempio:

m = 16, da -32767 a +32767
m = 32, da -2147483647 a + -2147483647

☞Abbiamo due diverse rappresentazioni per lo zero:

-0 = 10000
+0 = 00000

Ad esempio:

m = 2 ==> posso rappresentare 2^2 -1 = 3 numeri:

Valore Binario	Valore Decimale
00	+0
01	+1
10	-0
11	-1

Rappresentazione in Complemento

Per semplificare le operazioni su interi (con segno) si adotta una rappresentazione dei **numeri negativi** in **complemento**.

Complemento alla base:

dato un numero X in base b di n cifre, il complemento alla base è definito come:

$$b^n - X$$

Esempi:

- n = 2 , b = 10, X = 64
  Il complemento a 10 di 64 è $10^2 - 64 = 36$

- n = 5 , b = 2, X = 01011
  Il complemento a 2 di X è
  $2^5 - X = 100000-01011 = 10101$.

☞ In pratica, il principio è che la somma del numero più il suo complemento dia una stringa di n cifre a zero.

Complemento alla base -1:

dato un numero X in base b di n cifre, il complemento alla base -1 è definito come:

$$(b^n-1)-X$$

Esempio:

- n = 5 , b = 2, X = 01011
  Il complemento a 1 di X è
  $(2^n - 1)-X = 11111 - 01011 = 10100.$

☞ Osserviamo che il risultato è la sequenza di cifre che si ottiene complementando a 1 ogni singolo bit.

Riassumendo:

Supponiamo di voler rappresentare un numero X tramite n cifre binarie.

- la rappresentazione in **complemento a 2** si ottiene sottraendo X da 2^n
- la rappresentazione in **complemento a 1** si ottiene sottraendo X da 2^n-1.

In Pratica:

- Il **complemento a 1** di un numero binario X (rappresentato come intero positivo) si ottiene complementando tutti i bit (cioè scambiando 0 con 1, e viceversa).
- Il **complemento a 2** si ottiene prima calcolando il complemento a 1, e poi sommandovi 1.
  (Infatti, si può scrivere l'operazione come:
  $2^n - X$ come $2^n - 1 - X + 1$)

Rappresentazione in Complemento

Esempio: n=4

Intero	Modulo	Segno+mod	Compl. a 1	Compl. a 2
-7	0111	1111	1000	1001
-6	0110	1110	1001	1010
-5	0101	1101	1010	1011
-4	0100	1100	1011	1100
-3	0011	1011	1100	1101
-2	0010	1010	1101	1110
-1	0001	1001	1110	1111
-0	0000	1000	1111	---

- **Rappresentazione in complemento a 1:** I numeri positivi sono rappresentati dal loro modulo e hanno il bit più significativo a zero. I numeri negativi si ottengono complementando a 1 i corrispondenti moduli, segno compreso. Pertanto hanno il primo bit sempre a 1.

- **Rappresentazione in complemento a 2:** I numeri positivi sono rappresentati dal loro modulo e hanno il bit più significativo a zero. I numeri negativi si ottengono complementando a 2 i corrispondenti moduli, segno compreso. Pertanto hanno il bit di segno sempre a 1. (Una sola rappresentazione dello zero: 00000).

☞ Su 5 bit si possono rappresentare i numeri da -16 (10000) a +15 (01111).

☞ Il numero più piccolo rappresentabile (-16) non ha il corrispettivo positivo (per rappresentare +16 occorrerebbero 6 bit).

Esercizi:

- Si rappresentino i seguenti numeri in complemento a 2 avendo 8 bit a disposizione:

	Modulo	Compl.a 1	Compl.a2
- 34 →	0010 0010	→1101 1101 →	1101 1110
-25 →	0001 1001	→1110 0110 →	1110 0111
+46			→ 0010 1110
- 23 →	0001 0111	→1110 1000	→ 1110 1001
+66			→ 0100 0010
+72			→ 0100 1000
-120 →	0111 1000	→1000 0111	→ 1000 1000
- 10 →	00001010	→1111 0101	→ 1111 0110

- Interpretare la sequenza di bit 1001 0001 1100 0101 come:
  a) numero naturale
  b) numero relativo in modulo e segno
  c) numero relativo in complemento a due.

a) Interpretando la sequenza come **numero naturale**:

$1001\ 0001\ 1100\ 0101 = 91C5_{16} = 9 * 16^3 + 1 * 16^2 + 12 * 16 + 5 = 37317_{10}$

b) Interpretando la sequenza come numero relativo in modulo e segno:

1 001 0001 1100 0101 = **-** $11C5_{16}$ = **-** $(1 * 16^3 + 1 * 16^2 + 12 * 16 + 5)$ = - 4549

c) Interpretando la sequenza come numero relativo in complemento a due:

1001 0001 1100 0101

- ricaviamo il complemento a 1
  1001 0001 1100 0101 - 1 = 1001 0001 1100 0100

- complementiamo i bit:
  - 0110 1110 0011 1011 =
  - $6E3B_H$ = **-** $(6 * 16^3 + 14 * 16^2 + 3 * 16 + 11)$ =
  - 28219

Operazioni aritmetiche sui numeri relativi

Somma algebrica:

Sfrutta la rappresentazione in complemento a 2.
Si sommano i numeri per colonna incluso il bit del segno, ignorando l'eventuale riporto sul bit del segno.

Esempi:

```
   0 000101  (+5)       0 000101  (+5)

   0 001000  (+8)       1 111000  (-8)
   --------             --------
   0 001101  (+13)      1 111101  (-3)


   1 111011  (-5)       1 111011  (-5)

   0 001000  (+8)       1 111000  (-8)
   --------             --------
(1)0 000011  (+3)    (1)1 110011  (-13)
```

- Consideriamo 8 bit: (il numero è compreso fra -128 e 127)

  70 = 01000110
  70 + 70 =

  01000110 +
  01000110 =

  10001100 (OVERFLOW)

  -70 - 70 = cioè (-70) + (-70)
  -70 in complemento a 2: 10111010

  10111010 +
  10111010 =
  ----
(1)01110100 (OVERFLOW)

☞ Si può verificare **overflow** solo quando gli operandi hanno lo stesso segno (e l'operazione produce un numero di segno opposto !!).

Sottrazione:

Se i numeri sono rappresentati in complemento a 2 l'operazione di sottrazione si effettua mediante somma (vantaggio della rappresentazione in complemento a 2).

Esercizio:

Effettuare le seguenti operazioni, supponendo di operare con una rappresentazione dei numeri su un byte e in complemento a due:

-34 + 25	-34 + 46	-23 - 34
66 + 72	-120 -10	

Soluzione

I valori indicati risultano così rappresentati:

```
- 34      1101 1110
+ 25      0001 1001
+ 46      0010 1110
- 23      1110 1001
+ 66      0100 0010
+ 72      0100 1000
-120      1000 1000
- 10      1111 0110
```

Da cui:

```
-34=  1101 1110
+25=  0001 1001
      ---------
      1111 0111 = - 00001001= -9

-34=  1101 1110
+46=  0010 1110
      ---------
   (1)0000 1100 = +00001100 = +12
```

```
-34=   1101 1110
-23=   1110 1001
          ---------
       (1)1100 0111 =-00111001=-57
```

```
+66=   0100 0010
+72=   0100 1000
       ---------
       1000 1010= - 01110110=-118
```

OVERFLOW : Stiamo sommando due numeri positivi e otteniamo un numero negativo

```
-120   =   1000 1000
-10    =   1111 0110
          ---------
       (1)0111 1110= +01111110=+126
```

OVERFLOW : Stiamo sommando due numeri negativi e otteniamo un numero positivo

Moltiplicazione e divisione tra numeri relativi:

- Utilizzare i numeri in valore assoluto e utilizzare le operazioni viste sui numeri naturali (somma/differenza e scorrimento).

- Il segno si determina in base al segno degli operandi.

☞ L'applicazione delle operazioni direttamente sui numeri in complemento è in generale scorretta.

Infatti: (-3) * (+3)

I valori indicati risultano così rappresentati con 4 bit in complemento a 2:

-3 = - 0011 = 1101
+3 = 0011

Il risultato del calcolo, -9, non è rappresentabile su soli 4 bit. Supporremo disponibile allo scopo una parola di un byte.

```
        1101 *
        0011 =
        ----
        1101
      1101
    0000
  0000
---------
00100111
```

che, su un byte, è interpretabile come numero positivo, di valore 39.

Operazione di shift

Shift verso sinistra:

- Dalla rappresentazione dei numeri discende immediatamente che lo scorrimento verso sinistra di tutte le cifre del numero di una posizione con l'inserimento di uno zero nella posizione di destra equivale a moltiplicare il numero per la base.

☞ Lo scorrimento di k posizioni verso sinistra equivale a moltiplicare il numero per b^k.

Shift verso destra:

- Lo scorrimento (shift) verso destra di tutte le cifre del numero di una posizione con l'inserimento di uno zero nella posizione di sinistra, equivale a dividere il numero per la base (cioè a moltiplicare il numero per $b^{-1)}$.

☞ Lo scorrimento di k posizioni verso destra equivale a dividere il numero per b^{-k}.

Moltiplicazione/Divisione per 2 attraverso shift

Rappresentazione in modulo e segno:

Si replica sempre il segno.

0011 (+3) $*$ 2 = 0110 (+6)
1011 (-3) $*$ 2 = 1110 (-6)
(se si "scarica" un 1 è overflow)

1100 (-4) $*$ 2 = 1000 scarico un 1 (overflow)

0011 (+3) div 2 = 0001 (1)
1011 (-3) div 2 = 1001 (-1)

Rappresentazione in complemento a 2:

Moltiplicazione: si esegue uno shift di tutti i bit, compreso il segno (se si cambia segno, c'è overflow).
0011 (+3) $*$ 2 = 0110 (+6)
1101 (-3) $*$ 2 = 1010 (-6)
1010 (-6) $*$ 2 = 0100 overflow

Numeri Frazionari

Sono numeri reali compresi fra zero e 1;

Rappresentazione dei numeri frazionari:

Sia:

α={0,1,2,...p-1} ,

b=p

Dato un numero frazionario N, la sua rappresentazione in base p e' data da una sequenza di cifre :

$$\mathbf{0,a_{-1}\, a_{-2}\, a_{-3} \ldots a_{-n}}$$

dove $\mathbf{a_{-i}} \in \alpha$, $\forall$ **i=0,..n**.

Il valore di N è dato dalla formula:

$$N_p = a_{-1} * p^{-1} + a_{-2} * p^{-2} + \ldots + a_{-n} * p^{-n}$$

o, in forma più compatta:

$$N_p = \bullet_{i=1,..,n}\ a_{-i} * p^{-i}$$

Ad esempio:

- Base decimale: **p=10, N_{10}= 0,587**

$$(0,587)_{10} = 5 * 10^{-1} + 8 * 10^{-2} + 7 * 10^{-3}.$$

- Binario: **p= 2, N_2= 0,1011**

$$(0,1011)_2=(1*2^{-1}+0*2^{-2}+1*2^{-3}+1*2^{-4})_{10}$$
$$= (0,6875)_{10}$$

Conversione di un numero decimale frazionario in base $\neq$ 10

$$N_p = a_{-1} * p^{-1} + a_{-2} * p^{-2} + ... + a_{-n} * p^{-n}$$

- Moltiplico N_p per p:

$$N_p * p = a_{-1} + a_{-2} * p^{-1} + ... + a_{-1} * p^{-n+1}$$

Il risultato è un **numero reale >1**:

- Parte **intera** ($N_p * p$) = a_{-1}
- Parte **frazionaria** ($N_p * p$)
  = $a_{-2} * p^{-1} + ... + a_{-n} * p^{-n+1}$ = **M_1**

- Applichiamo ancora la **moltiplicazione** per p sulla parte frazionaria M_1 ottenuta nella divisione precedente:
  - Parte **intera** ($M_1 * p$) = a_{-2}
  - Parte **frazionaria** ($M_1 * p$)
    = $a_{-3} * p^{-1} + ... + a_{-n} * p^{-n+2}$ = **M_2**

- Ripetiamo il procedimento su M_3, M_4, etc. per ottenere le cifre rimanenti (a_{-3}, a_{-4}, .., a_{-n}).

☞ In pratica, il procedimento da seguire è il seguente:

Algoritmo delle moltiplicazioni successive

Sia N il numero frazionario.

1. Si moltiplica N per la nuova base; sia I la parte intera e M la parte frazionaria.

2. Si converte I nella corrispondente cifra della nuova base.

3. Si aggiunge la cifra così ottenuta a destra delle cifre ottenute in precedenza (la prima cifra immediatamente a destra della virgola)

4. Se M=0 (oppure si sono ottenute le cifre richieste) fine; Altrimenti poni N = M e torna al passo 1.

Conversione binaria di numeri frazionari:

- Si ottiene effettuando successive moltiplicazioni per due.

- Il risultato è la sequenza di zero e uno ottenuti considerando le parti intere delle moltiplicazioni dalla più significativa alla meno significativa.

Ad Esempio:

- Convertire in base 2: **N_{10}=0,875**

Moltiplicazione	Parte Frazionaria	Parte Intera (a_{-i})
0,875 * 2	0,75	1
0,75 * 2	0,5	1
0,5 * 2	0	1

quindi $(0,875)_{10} = (0,111)_2$

- Convertire in base 8: **N_{10}=0,875**

Moltiplicazione	Parte Frazionaria	Parte Intera (a_{-i})
0,875 * 8	0	7

quindi $(0,875)_{10} = (0,7)_8$

- Convertire in base 16: **N_{10}=0,875**

Moltiplicazione	Parte Frazionaria	Parte Intera (a_{-i})
0,875 * 16	0	14 (E_{16})

quindi $(0,875)_{10} = (0,E)_{16}$

Conversione di numeri frazionari in base non decimale

☞ Non sempre si ottiene una conversione esatta:

Moltiplicazione	Parte Frazionaria	Parte Intera (a_{-i})
$0,8 * 2$	0,6	1
$0,6 * 2$	0,2	1
$0,2 * 2$	0,4	0
$0,4 * 2$	0,8	0
$0,8 * 2$	0,6	1
....	...	...

quindi $(0,8)_{10} = (0,\underline{1100})_2$ **periodico**

- Quindi uno stesso numero può avere un numero finito di cifre in una base ed un numero infinito in un'altra.

- Quindi nella rappresentazione interna si può introdurre un **errore di troncamento**.

Rappresentazione dei Numeri Reali

Consideriamo l'insieme R dei numeri reali.

Ogni elemento di R, in generale, può essere espresso come somma di un intero con un numero frazionario.

In pratica, un numero reale è individuato univocamente da:

- una parte intera I
- una parte frazionaria F

Rappresentazione dei reali in virgola fissa

Un numero prefissato di cifre viene dedicato alla parte intera e a quella frazionaria (**rappresentazione in virgola fissa**).
Quindi:

- Si calcola la rappresentazione della parte intera nella base data con la formula:

$$N_i = a_n * p^n + a_{n-1} * p^{n-1} + \ldots + a_1 * p^1 + a_0$$

- Si calcola la rappresentazione della parte frazionaria nella base data (tenendo conto del numero di cifre disponibili) con la formula:

$$N_f = a_{-1} * p^{-1} + a_{-2} * p^{-2} + \ldots + a_{-n} * p^{-n}$$

- Si giustappongono le due rappresentazioni:

<rappresentazione I> **.** <rappresentazione F>

Ad esempio:

$331{,}6975_{10} = 00101001011{,}10110_2$
(riservando 11 bit per la parte intera e 5 per quella frazionaria).

☞ La precisione è variabile e può essere scarsa per numeri di valore prossimo allo zero

Rappresentazione dei numeri reali in virgola mobile

- A un numero reale r vengono associati due numeri:
    - **m** = **mantissa**
    - **n** = **esponente** (o caratteristica)

Mantissa:

- è un numero frazionario con segno. Il suo valore è quindi compreso nell'intervallo]-1, +1[.
- mantissa **normalizzata**: se la prima cifra dopo la virgola è diversa da 0. Il valore assoluto della mantissa, in questo caso, è compreso tra 1 ed 1/p

Esponente:

- è un intero con segno.

☞ la mantissa e l'esponente sono legati dalla relazione:

$$r = m * b^n$$

dove b è un numero intero che indica una base utilizzata per la notazione esponenziale (in generale, se p è la base del sistema di numerazione, **p** $\neq$ **b**).

Rappresentazione in virgola mobile

Esempi:

- **p=b=10, r = -331,6975**

  r si rappresenta con m=-0,3316975 e n=3

- Conversione in binario, virgola mobile (**p=b=2)**:
  r = $12,5_{10}$

  $12,5_{10} = 1100,1_2$

  $m = 0,11001_2$ $n = 100_2$

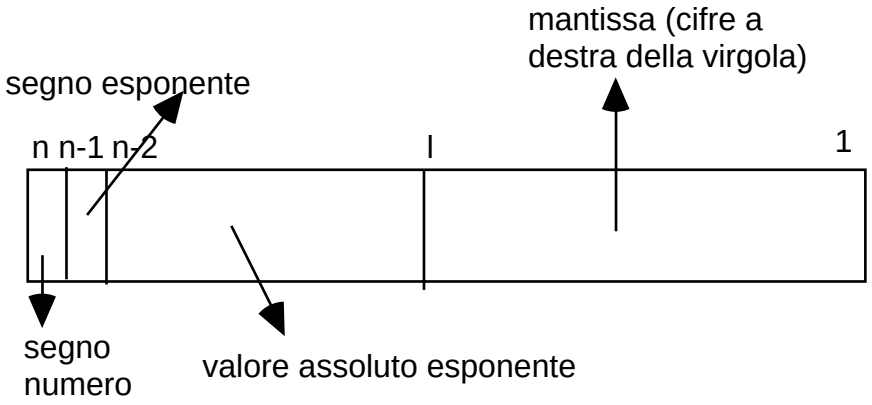
- Conversione in binario, virgola mobile (p=b=2):
  r=$26,5_{10}$

  $26,5_{10} = 11010,1_2$

  $m = 0, 110101_2$ $n = 101_2$

Rappresentazione binaria in virgola mobile:

Per ogni numero reale vengono utilizzati n bit:



Ad esempio:

Supponiamo di avere a disposizione **<u>32 bit</u>**, di cui:

- 1 bit per il segno della mantissa;
- 1 bit per il segno dell'esponente;
- 8 bit per il valore assoluto dell'esponente;
- 22 bit per la mantissa.

→ Rappresentazione in virgola mobile di **r= 26,5**

$$r = 0{,}110101_2 * 2^{101_2}$$

esponente		mantissa
0 | 0 |	00000101 |	11010100000000000000

→ Rappresentazione in virgola mobile di **r=0,3125**

$$r = 0{,}101_2 * 2^{-1_2}$$

esponente		mantissa
0 | 1 |	00000001 |	10100000000000000000

Rappresentazione binaria in virgola mobile

- Poiché la mantissa comincia sempre con 1 (***mantissa normalizzata***), si può utilizzare questo come bit di segno.

- invece di riservare un bit per il segno dell'esponente, in alcuni casi si adotta la rappresentazione in complemento a 2 dell'esponente o la rappresentazione con esponente aumentato.

Esempio: complemento a 2 per l'esponente.

32 bit:

1 bit segno mantissa;
8 bit valore dell'esponente (in compl. a 2);
23 bit mantissa.

- $r=26,5 = 0,110101_2 * 2^{101_2}$

esponente mantissa

0 | 00000101 | 11010100000000000000000

- $r=0,3125 = 0,101_2 * 2^{-1_2}$

0 | 11111111| 10100000000000000000

Rappresentazione in virgola mobile con esponente aumentato

- Ad un numero reale r vengono associati due numeri:
  - **m** = **mantissa**
  - **e** = **esponente aumentato**

☞ la mantissa e l'esponente aumentato sono legati dalla relazione:

$$r = m * b^{e-q}$$

dove:

- b è la base
- q è una quantità chiamata **eccesso**

Definendo l'eccesso in modo opportuno, si può evitare di memorizzare il segno dell'esponente poiché l'esponente aumentato risulta sempre positivo.

Ad esempio:
Se la rappresentazione dell'esponente è su 1 byte e l'eccesso q=128:

n	e
-128	0
-127	1
...	...
0	128
1	129
...	...
127	255

Esempio:
Consideriamo come esempio il numero reale 0.1 (in base 10).

La rappresentazione in base 2 è costituita da un numero infinito e periodico di cifre binarie:
$(0.1)_{10}$ = .00011001100110011....

Supponiamo di voler rappresentare il numero con 6 byte (rappresentazione Turbo-Pascal).
L'esponente occupa il primo byte e la mantissa i successivi 5. Il byte meno significativo precede quelli più significativi. Il bit più significativo dell'ultimo byte viene utilizzato implicitamente per il segno della mantissa. L'esponente e` codificato con eccesso 128.

Rappresentazione interna:

$(0.1)_{10}$ = .00011001100110011....

Normalizzazione: = .11001100110011....$*$ 2^{-3}
Codifica esponente= 128 - 3 = 125_{10} = 01111101
Segno della mantissa: .**0**1001100110011.....

Rappresentazione interna finale:

esponente byte 5 byte 4 ..... byte 1
01111101|11001100|..... | |01001100

ESERCIZIO:

Rappresentare su 16 bit (di cui 8 di esponente in complemento a due, uno per il segno e 7 di mantissa) i seguenti valori:

0.75 0.1 13.33

Soluzione

a) r=0.75

Parte intera: 0
Parte frazionaria: algoritmo delle moltiplicazioni successive. Si ha:

$0.75 * 2 = 1.5 \rightarrow F = 0.5, \quad I = 1$
$0.5 * 2 = 1.0 \rightarrow F = 0.0, \quad I = 1$
$0.0 * 2 = 0.0 \rightarrow F = 0.0, \quad I = 0$
.....

Il valore richiesto è dunque:
<parteIntera>.<parteFrazionaria>=0.11

Non è necessaria alcuna normalizzazione.

Rappresentazione finale di 0.75:

esponente	segno	mantissa
00000000	**0**	**1100000**

b) r=0.1

Parte **intera**: 0
Parte **frazionaria**: si usa l'algoritmo delle moltiplicazioni successive:

$0.1 * 2 = 0.2 \rightarrow F = 0.2, \quad I = 0$
$0.2 * 2 = 0.4 \rightarrow F = 0.4, \quad I = 0$
$0.4 * 2 = 0.8 \rightarrow F = 0.8, \quad I = 0$
$0.8 * 2 = 1.6 \rightarrow F = 0.6, \quad I = 1$
$0.6 * 2 = 1.2 \rightarrow F = 0.2, \quad I = 1$
.....

rappresentazione **approssimata** perché si ottiene un numero periodico.

$0.1_{10} \rightarrow$ 0. 0<u>0011</u>= 0.0001100110011...

- Normalizzando:
  mantissa = 0. 1 100110
  esponente=-3= -(11)=11111100+1=11111101

esponente	segno	mantissa
11111101	**0**	**1100110**

Si noti che la mantissa viene troncata al numero disponibile di bit *solo dopo aver normalizzato*.

☞ C'è di un ***<u>errore di troncamento</u>***: il numero realmente rappresentato non è 0.1, ma:

$0.110011 * 2^{-3} = (1/2 + 1/4 + 1/32 + 1/64) * 1/8 =$
$= 0.099609375$

c) r= 13.33

Parte **intera**: 13 = $(1101)_2$

Parte **frazionaria**: 0.33 con l'algoritmo delle moltiplicazioni successive:

0.33 * 2 = 0.66 $\rightarrow$ F = 0.66 I = 0
0.66 $*$ 2 = 1.32 $\rightarrow$ F = 0.32, I = 1
0.32 $*$ 2 = 0.64 $\rightarrow$ F = 0.64, I = 0
0.64 $*$ 2 = 1.28 $\rightarrow$ F = 0.28, I = 1
0.28 $*$ 2 = 0.56 $\rightarrow$ F = 0.56, I = 0
0.56 $*$ 2 = 1.12 $\rightarrow$ F = 0.12, I = 1
.....

quindi: r=13.33=1101.010101....

Normalizzando:

m=0.1101 010101...
esponente = +4=100

esponente	segno	mantissa
00000100	**0**	**1101010**

Valutiamo l'errore: il numero realmente rappresentato vale:

$0.1101010 * 2^4 = (1/2+1/4+1/16+1/64) * 16 = 53/64 * 16 = 13.25$

Precisione nella rappresentazione dei numeri reali

Si può, in generale, osservare che:

- Quanto maggiore è il numero di bit riservati alla mantissa tanto maggiore è il numero di cifre significative che possono essere memorizzate (precisione);

- Quanto maggiore è il numero di bit riservati all'esponente tanto maggiore è l'ordine di grandezza della cifra che può essere rappresentata.

Precisione:

è data dal numero di cifre in base 10 rappresentabili con la mantissa.

Ad esempio:
Se la mantissa è rappresentata da 20 bit, la precisione è di 6 cifre (max valore rappresentabile dalla mantissa 2^{20}-1 circa $= 10^6 = 1000000$).

☞ Le cifre meno significative della mantissa che non possono essere rappresentate nel numero di bit a disposizione vengono eliminate mediante **troncamento** o **arrotondamento**.

Troncamento e arrotondamento

Esempio:

$r=1029_{10} = 0{,}10000000101 * 2^{11}$

con:

- 10 bit per la mantissa
- 5 bit per l'esponente

Rappresentazione:

- con **troncamento**:
  0 01011 1000000010 (=1028)

- con **arrotondamento**:
  0 01011 1000000011 (=1030)

Esempio:

$r= 0{,}8_{10} = 0{,}110011001100..... * 2^0$ (periodico)

con:

- 10 bit per la mantissa
- 5 bit per l'esponente

Rappresentazione:

- con **troncamento (**e con **arrotondamento)**:

  0 00000 1100110011 (~0,7998)

Valori massimi e minimi rappresentabili:

Dipendono dall'esponente e dal metodo di rappresentazione.

Se il valore assoluto dell'esponente è rappresentato da 10 bit:

- il **massimo numero reale** è (in valore assoluto):

$$1 * 2^{1023} \sim 10^{307}$$

- il **minimo numero reale** è (in valore assoluto):

$$0{,}1_2 * 2^{-1023} \sim 0{,}5 * 10^{-307}$$

Accuratezza della macchina

È il più piccolo numero che, aggiunto a 1,0 produce un risultato diverso da 1,0.

tipicamente per n=32 e' circa $3 * 10^{-8}$

Operazioni aritmetiche con numeri reali

Somma:
Si opera nel modo seguente:

1)	si aumenta l'esponente del più piccolo, contemporaneamente spostando la mantissa a destra, fino a che i due esponenti sono uguali.
2)	si sommano le due mantisse insieme.
3)	si normalizza il risultato.

In questo caso si possono generare errori:

Errore di incolonnamento:

Si manifesta nel sommare numeri con esponenti diversi.

Ad esempio:
Supponiamo di lavorare in base 10 con 5 bit per la mantissa normalizzata.

$0.12465 * 10^{1} + 0.32999 * 10^{-2} =$
$0.12465 * 10^{1} + 0.00032 * 10^{1} =$

riportando tutto all'esponente maggiore si perdono le ultime 3 cifre del secondo addendo (cioè la quantità $0.00999 * 10^{-2}$).

Errore di cancellazione:

Si manifesta nel sottrarre numeri simili fra loro quando nel primo o in entrambi si è verificato un errore di troncamento

☞ Questo errore fa sì che la somma non goda sempre della proprietà associativa.

Esempio:

A = $-0.12344 * 10^0$ B = $0.12345 * 10^0$ C= $0.32741 * 10^{-4}$

Supponiamo di utilizzare solo cinque cifre decimali.

- Valutiamo **A + (B+C)**

B + C = $0.12345 * 10^0$ +

$0.00003 * 10^0$ errore di incolonnamento

$0.12348 * 10^0$ +

A $- 0.12344 * 10^0$

$0.00004 * 10^0$

Normalizzazione: $0.40000 * 10^{-4}$ errore di cancellazione

- Valutiamo **(A + B)+C** =

A + B= $0.12345 * 10^0$ +

$-0.12344 * 10^0$

$0.00001 * 10^0$

Normalizzazione: $0.10000 * 10^{-4}$ errore di cancellazione

$0.10000 * 10^{-4}$ +

C $0.32741 * 10^{-4}$

$$\frac{}{0.42741 * 10^{-4}}$$

ESERCIZIO

Effettuare la somma:

$$4.6 + 0.5$$

con numeri rappresentati in virgola mobile su 16 bit (di cui 8 di esponente in complemento a due, uno per il segno e 7 di mantissa), verificando i risultati ottenuti.

Soluzione

$4.6 = 0100.1001\ 1001.... = 0.100\ 1001 * 2^3$

$0.5 = 0000.10000000... = 0.100\ 0000 * 2^0$

Allineando i numeri in modo che abbiano identico esponente:

$4.6 = 0.100\ 1001 * 2^3 +$

$0.5 = 0.000\ 1000 * 2^3 =$

$0.101\ 0001 * 2^3 = 101.0001 =$

$5 + 1/16 =$

5.0625

Valore atteso per il risultato 5.1.

ESERCIZIO

Effettuare la seguente moltiplicazione fra numeri rappresentati in virgola mobile su 16 bit (di cui 8 di esponente in complemento a due, uno per il segno e 7 di mantissa), verificando il risultato ottenuto:

$2.15 * (-4.8)$ Risultato teorico: -10.32

Soluzione

$2.15 = 0010.00100110.... = 0.100\ 0100 * 2^2$

$-4.8 = -0100.1100110... = -0.100\ 1100 * 2^3$

Per moltiplicare i due valori è sufficiente moltiplicare le mantisse e sommare gli esponenti. Poiché però il prodotto di due numeri di 7 bit richiede in generale 14 bit per essere esattamente rappresentato, mentre anche la mantissa risultante dovrà essere rappresentata in 7 bit (i più significativi), l'operazione potrà dare luogo ad approssimazioni.

```
          1000100 x
          1001100 =
          -------
          0000000
         0000000
        1000100
       1000100
      0000000
     0000000
    1000100
    -------------
(0)1010000110000
```

Isolando i 7 bit più significativi si ha:

$R = -0.01010000 * 2^5 = -0.1010000 * 2^4 = -10$

Codici

Un codice è un sistema di simboli per rappresentare un'informazione di qualsiasi genere (parole, numeri, caratteri, etc.)

Codice binario, è un codice che utilizza come simboli le cifre binarie 0 ed 1.

Con N cifre binarie è possibile codificare 2^N oggetti distinti (esistono, infatti, 2^N configurazioni distinte che possono essere poste in corrispondenza biunivoca con i naturali da 0 a 2^N-1).

Per codificare M oggetti distinti, il numero minimo di cifre binarie necessarie è il minimo numero intero N tale che:

$$N \geq \log_2 M$$

Definire un codice significa attribuire un *significato unico e convenzionale* alle configurazioni binarie.

Ad esempio:
Codifica di quattro colori (rosso, giallo, blu, bianco):

Oggetto	Codifica
bianco	00
rosso	01
giallo	10
blu	11

Codifica dei caratteri

I caratteri di un testo vengono codificati tramite sequenze di bit, utilizzando un **codice** di traduzione.

Quello più usato è il **codice ASCII** (American Standard Code for Information Interchange).

Utilizza 7 bit. Rappresenta 128 caratteri. Mancano, ad esempio i caratteri accentati, greci etc.

Esempio:
0 è codificato come 048_{10} e 30_{16} (e $011\ 0000_2$)
9 è codificato come 057_{10} e 39_{16}
a è codificato come 097_{10} e 61_{16}
z è codificato come 122_{10} e $7A_{16}$

Vengono inseriti in un byte per cui 1 bit viene normalmente ignorato. In trasmissione funziona come **bit di parità** per individuare errori.
Il valore (0 o 1) del bit di parità è scelto in modo che la sequenza di 1 sia pari.

Ad esempio: 0 => 30_{16} => **0** $011\ 0000_2$

Il codice **ASCII esteso** utilizza invece 8 bit (256 caratteri)

Tabella dei Codici ASCII

Viene mantenuto l’ordinamento alfabetico.

Rappresentazioni Binarie

Riassumendo:

- **100** come intero con n = 32 bit (4 bytes)

100 : 2 0
50 : 2 0
25 : 2 1
12 : 2 0
6 : 2 0
3 : 2 1
1 : 2 1

00000000000000000000000001100100

- **-100**

11111111111111111111111110011100

- **100. come reale in virgola mobile**
    - 1 byte per l'esponente
    - 3 bytes per la mantissa.

$(100.)_{10} = (1100100.)_2$
Normalizzato:

$$0.1100100 * 2^7$$

Esponente: 7 = 00000111
0 00000111 10010000000000000000000

Il primo bit è il segno. Il primo bit della parte frazionaria è omesso perché essendo normalizzata è sempre 1

- **-100. come reale in virgola mobile:**

100000111 10010000000000000000000

- **'100'** come carattere con codifica ascii

00110001 31_{16}
00110000 30_{16}
00110000

- **'-100'** come carattere con codifica ascii

00101101 ....

Codici binari pesati

Codice BCD (Binary Coded Decimal) è un codice per la rappresentazione di numeri decimali. Ciascun numero viene suddiviso nelle cifre decimali che lo compongono e ogni cifra viene codificata separatamente dalle altre.

Per rappresentare le dieci cifre decimali {0,1,2,3,4,5,6,7,8,9} sono sufficienti quattro bit. Delle 2^4 configurazioni, solo dieci vengono impiegate per la codifica.

Esistono quindi più codici BCD, corrispondenti alla scelta di quali configurazioni sono rappresentative e di che numero decimale.

- **Codice BCD-8-4-2-1**

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

Ad esempio:

$(721)_{10}$ = $(0111\ 0010\ 0001)_{BCD}$

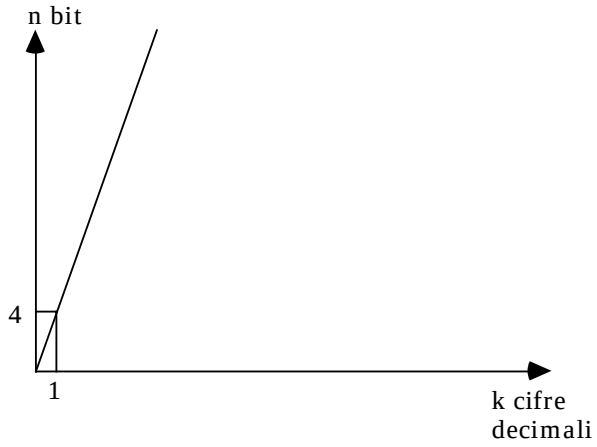
Esiste un aritmetica per numeri in rappresentazione BCD-8-4-2-1.

Questa rappresentazione è utilizzata nei sistemi a microprocessore in cui è scarsa l'elaborazione numerica, mentre sono frequenti le visualizzazioni di valori.

Vantaggi: corrispondenza diretta con le cifre decimali (facilità nell'input/output)

Con la rappresentazione BCD ciascuna cifra può essere inviata direttamente ai circuiti di visualizzazione (display).

Svantaggi: il numero di bit necessari cresce linearmente con il numero di cifre decimali da rappresentare (codice ridondante). Per rappresentare numeri di *k* cifre decimali occorrono 4*k* bit. In una rappresentazione binaria pura, invece, sarebbero sufficienti $N=[\log_2 k]$ bit.



Somma (BCD):

Si esegue, come nel sistema decimale, incolonnando e sommando le cifre di uguale peso.

È in pratica ricondotta alla somma binaria, salvo aggiustamenti dovuti alla peculiarità del codice (configurazioni non utilizzate).

Ad esempio:

a)

```
0010 0111 +        (27)
0011 0001 =        (31)
-------------
0101 1000     (58)
```

b)

```
0010 1001 +        (29)
0001 0010 =        (12)
-------------
0011 1011     (3?)
```

Si corregge il risultato aggiungendo alla configurazione non permessa (1011) il valore $6_{10}=0110_{BCD}$

```
0011 1011 +
0000 0110
-------------
0100 0001     (41)
```

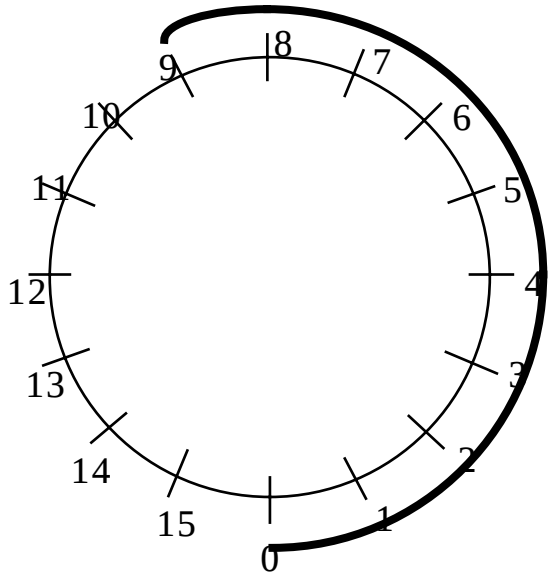
c) 0010 1000 + (28)
0001 1001 (19)

0100 0001 (41)

Le configurazioni di cifre sono lecite, ma il risultato non è corretto. Si è verificato un riporto tra la prima "cifra" e la seconda. Anche in questo caso si opera una correzione di 6_{10}.

0100 0001 +
0000 0110

0100 0111 (47)



I valori tra 10 e 15 non sono ammessi (6 valori, da cui la somma del valore 6_{10}).

Moltiplicazione e divisione per 10:

Moltiplicare o dividere un numero per 10 significa farlo scorrere a sinistra o destra di una cifra decimale, cioè di 4 bit in rappresentazione BCD.

Ad esempio:

27 * 10	0000 0010 0111	(27)
	0010 0111 0000	(270)
342 / 10	0011 0100 0010	(342)
	0000 0011 0100	(34)

Viene "scaricata" la cifra 0010 (ovvero 2_{10}) che rappresenta il resto della divisione.

Moltiplicazione e divisione per 2:

La moltiplicazione per due viene ricondotta alla somma.

La divisione per due viene effettuata con uno scorrimento a destra di tutti i bit, come per i numeri binari. Se da una "cifra" all'altra passa un bit 1 è necessario effettuare una correzione (si sottrae 3_{10}).

Ad esempio:

0011 0110 (36)

Dopo lo scorrimento:

0001 1011

Correggendo:

```
0001 1011 -
     0011
----------
0001 1000      (18)
```

Il bit che passa dalla quinta alla quarta posizione aveva peso 2^4 come binario puro e peso 10^1 come codice BCD.

Dopo lo scorrimento, il bit ha un peso 2^3 (= 8) come binario puro mentre il suo contributo dovrebbe essere per un valore pari a 10/2=5.

Per correggere la cifra si sottrae quindi 8-5=3.

Sottrazione:

Si può ricondurre al caso di somma se si adotta una rappresentazione in complemento alla base (10, in questo caso).

Ad esempio: 27-12

0010 0111 (27)

0001 0010 (12)

Il suo complemento a 10, si ottiene sommando 1 al complemento a 9 di ogni "cifra":

```
compl. a 9:     1000 0111 +   (87)
                0000 0001
                ----------
compl. a 10:    1000 1000     (88)
```

Somma algebrica:

```
                0010 0111 +   (27)
                1000 1000     (88)
                ----------
                1010 1111 +   correzione di 6
                0110 0110       per entrambe le cifre
                ----------
           (1)  0001 0101     (15)
```

La presenza di un riporto indica l'assenza di un prestito. Il risultato e` corretto ed il segno e` quello del primo addendo.

Ad esempio: 12 - 27

	0001 0010	(12)
	0010 0111	(27)
compl. a 9:	0111 0010 +	(72)
	0000 0001	
	---	
compl. a 10:	0111 0011	(73)

Somma algebrica:

	0001 0010 +	(12)
	0111 0011	(73)

	1000 0101	(85)

L'assenza di un riporto indica la presenza di un prestito. Il risultato corretto (15_{10}) di ottiene facendo di nuovo il complemento a 10. Il segno è l'opposto di quello del primo addendo.

	1000 0101	(85)
compl. a 9:	0001 0100 +	
	0000 0001	

	0001 0101	(-15)

- **Codice BCD-2-4-2-1**

Questo codice ha la proprietà di essere *autocomplementare*, ovvero la rappresentazione binaria del complemento a 9 di una cifra decimale A_d si ottiene complementando bit a bit la rappresentazione BCD di A_d:

A_d	=	3	A_{BCD}	=	0011
$9 - A_d$	=	6	$(9 - A_d)_{BCD}$	=	1100

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

La proprietà di autocomplementarietà risulta particolarmente utile in alcune operazioni aritmetiche (somma algebrica).

Non è utilizzato nella pratica.

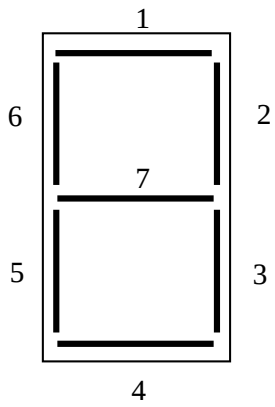
Codici per il dialogo uomo-macchina

- **Codice 1 su N**

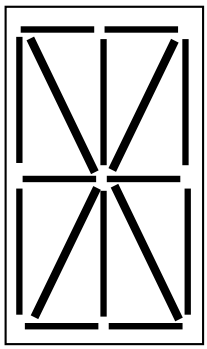
Impiega N bit per codificare N informazioni. Ogni configurazione valida contiene un solo bit a 1.

- **Codice a sette segmenti**

Impiega 7 bit per rappresentare le cifre decimali sui sette segmenti di un display.



È stato ampliato successivamente per rappresentare anche le prime sei lettere dell'alfabeto (cifre esadecimali).



Cifra decimale	Codice 1 su 10	Codice 7 seg.
0	0000000001	1111110
1	0000000010	0110000
2	0000000100	1101101
3	0000001000	1111001
4	0000010000	0110011
5	0000100000	1011011
6	0001000000	0011111
7	0010000000	1110000
8	0100000000	1111111
9	1000000000	1110011

Codici per il dialogo convertitori-macchina

La maggior parte delle grandezze fisiche varia nel tempo con continuità.

Per elaborare tali grandezze con un calcolatore e` necessaria una ***conversione analogico-digitale***.

La continuità dei valori analogici si mappa in "contiguità" delle configurazioni di bit che li rappresentano.

Due configurazioni sono contigue se differiscono al più per un bit

Con una codifica binaria dei valori (per semplicità, naturali) questo non è possibile.

Naturale	Binario
0	00
1	01
2	10
3	11

Nel passaggio dal valore 1_{10} a 2_{10}, in binario ci sono ***configurazioni spurie*** (01 --> **11** --> 10 e 01 --> **00** --> 10) poiché il cambiamento contemporaneo di due bit è un evento molto improbabile.

Questo problema si risolve con l'adozione di codici ciclici, dove le configurazioni di codice consecutive differiscono unicamente per un bit.

Si parla di codici ***completi*** se, adottando N bit, il codice contiene tutte le 2^N configurazioni possibili.

- **Codice Gray a 4 bit**

È un codice ciclico completo che utilizza 4 bit per codificare 2^4 informazioni diverse.

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

Esistono algoritmi di conversione da codice Gray a numero binario puro e viceversa.

Codici a rilevanza e a correzione d'errore

Nella trasmissione o nella memorizzazione, alcuni bit di una sequenza possono alterare il loro valore.

Esistono tecniche di codifica che consentono di rilevare errori ed, in alcuni casi, correggerli.

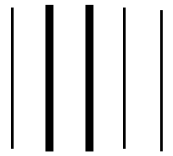
- **Codice 2 su 5**

Impiega 5 cifre binarie per codificare dieci elementi. In ciascuna configurazione lecita solo due dei cinque bit sono posti a 1.

Cifra decimale	Codifica
0	01100
1	11000
2	10100
3	10010
4	01010
5	00110
6	10001
7	01001
8	00101
9	00011

È un codice in grado di rilevare ***errori singoli*** (su singolo bit). Il controllo di errore si ottiene verificando se il numero di bit a 1 è uguale a due.

Codici di questo tipo sono usati in applicazioni industriali e commerciali (ad esempio, codici a barre)



0 1 1 0 0 ------> cifra 0

- **Codici a controllo di parità**

Per ottenere un codice a rilevazione d'errore, si può aggiungere a un codice pesato (BCD) un quinto bit, detto bit di parità. Il valore di tale bit è 0 (oppure 1) in modo che il numero di bit nella sequenza sia pari o dispari (si parla di ***parità pari***, nel primo caso, ***parità dispari***, nel secondo).

E‘ in grado di rilevare errori su un numero dispari di cifre binarie.

Cifra decimale	BCD-2-4-2-1	Con controllo di parità
0	0000	0000 0
1	0001	0001 1
2	0010	0010 1
3	0011	0011 0
4	0100	0100 1
5	1011	1011 1
6	1100	1100 0
7	1101	1101 1
8	1110	1110 1
9	1111	1111 0

Codifica delle immagini

Codificate come sequenze di numeri binari.

- In bianco e nero (b/n) (bit map)

- A livello di grigio (n bit, 2^n livelli)

  Normalmente, n=8 $\rightarrow$ 256 livelli di grigio
  00000000 nero
  11111111 bianco

- A colori rgb (8 bit per ciascun colore)
  (red, green, blue)

Esistono altri codici (televisivo con luminanza e crominanza)

Immagini elaborate dal calcolatore

- dimensioni (n. righe e n. colonne)
  ad esempio 256x256
- numero di livelli di grigio

Digitalizzazione, passaggio da una immagine ad una sequenza binaria.

L'immagine è vista come una matrice di **punti** (*pixel*).

Ad ogni punto è associato un numero che corrisponde ad un particolare **colore** o, per immagini in bianco e nero, ad un **livello di grigio** (numero potenza di 2).

Una maggiore qualità di immagine implica una maggiore occupazione di memoria:

- numero maggiore di punti per pollice (*dpi*, dot per inch; 1 pollice = 2.54 cm);
- numero maggiore di colori (*palette*).

Ad esempio, per una fotografia di bassa qualità (80-100 dpi) occorrono circa 1/8 Mbyte; per una buona riproduzione in b/n (1200 dti), circa 16 Mbyte.

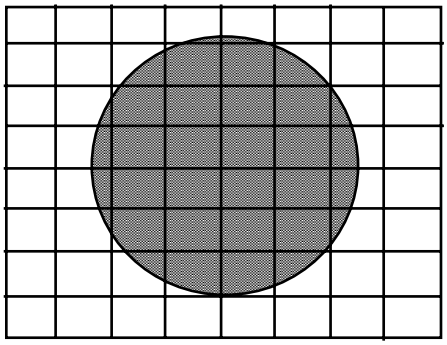
Per interpretare le sequenze di bit occorre conoscere:

- dimensioni dell'immagine;
- risoluzione (misurata in dpi);
- numero di colori o sfumature.

Codifica delle Immagini

Standard di codifica: TIFF (Tagged Image File Format) può utilizzare tecniche di compressione, GIF, etc.

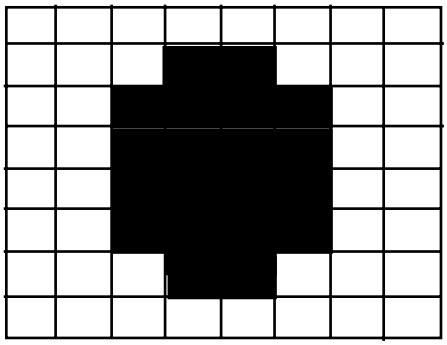
Esempio:



64 pixel, immagine a due colori (bianco e nero).

Serve un solo bit per codificare la tonalità di colore, per ciascun pixel.

In totale 64 bit (8 byte).



```
00000000
00011000
00111100
00111100
00111100
00111100
00011000
00000000
```

Codifica di sequenze sonore e filmati

Tempo di *campionamento* e successiva “digitalizzazione”.

Filmato di 10 sec (30 fotogrammi al sec.) occupa circa 6 Mbyte.

Sequenza sonora di 50 sec, da 0.5 Mbyte a 2 Mbyte a seconda della qualità richiesta.

Considerazioni:

Necessità di ridurre lo spazio per rappresentare un’informazione senza perdita del contenuto informativo.

Codifiche che tengano conto della *probabilità di occorrenza* di un simbolo (ad esempio, “.” nel codice Morse per il carattere più probabile “e”).

Tecniche di compressione, codificano in modo compatto sequenze ripetute di simboli.

Con perdita di contenuto, fattore di compressione 15-20:1, senza perdita di contenuto, fattore di compressione 2:1.

Codifica delle informazioni e fabbisogno di memoria:

Tipo di informazione	Memoria necessaria
Testi	2 Kbyte per pagina
Immagini	1/8 - 16 Mbyte
Sequenze sonore	10 - 40 Kbyte per sec.
Filmati	0.6 Mbyte per secondo