

# Laboratorio di Informatica

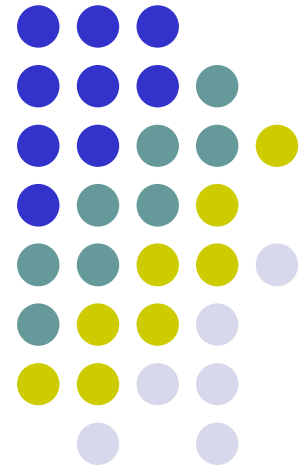
## Corso di Laurea in Matematica

### A.A. 2007/2008

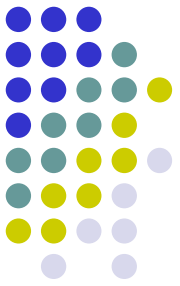
---

Dott. Davide Di Ruscio

Dipartimento di Informatica  
Università degli Studi di L'Aquila

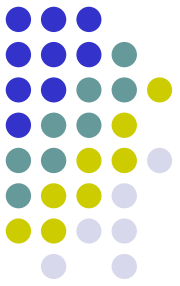


# Nota



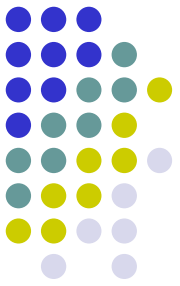
Questi lucidi sono tratti dal materiale distribuito dalla McGraw-Hill e basati su del materiale fornito dal Prof. Flammini Michele

# Sommario

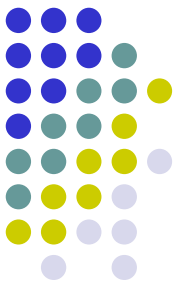


- Concetti fondamentali
- Aspetti architettureali di un sistema di calcolo
  - hardware
  - software
    - software di base
    - software applicativo
- **Codifica dell'informazione**
  - **numeri naturali, interi, reali**
  - **caratteri**
  - **immagini**
- Macchina di Von Neumann
  - CPU (UC, ALU, registri, clock)
  - memoria centrale
  - bus di sistema
  - periferiche
- Linguaggio macchina
- Linguaggio assembler
- Sistema operativo
- Ambiente di programmazione

# Codifica numeri frazionari (reali compresi tra $0$ ed $1$ )



- I numeri frazionari sono reali  $N$  t.c.  $0 < N < 1$
- Non sono rappresentabili in maniera precisa in quanto per farlo servirebbero un numero infinito di cifre (e quindi di bit)



- Estendendo quanto detto per i sistemi di numerazione alla parte del numero dopo la virgola si ha che

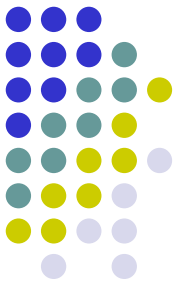
$$N_{10} = 0, a_{-1} a_{-2} \dots a_{-n} = a_{-1} \cdot 10^{-1} + a_{-2} \cdot 10^{-2} + \dots + a_{-n} \cdot 10^{-n} = \sum_{i=-1}^{-n} a_i \cdot 10^i$$

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

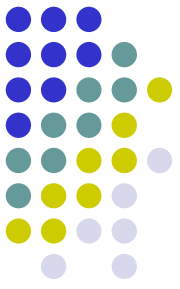
- Per una generica base  $b$

$$N_b = (0, a_{-1} \dots a_{-n})_b = a_{-1} \cdot b^{-1} + \dots + a_{-n} \cdot b^{-n} = \sum_{i=-1}^{-n} a_i \cdot b^i \quad (2)$$

Il peso delle cifre  $a_i$  dipende dalla base prescelta nel sistema di numerazione

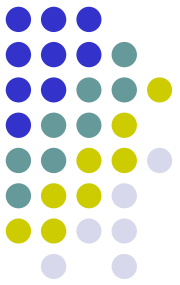


- Come convertire un numero frazionario in base  $b$  nel suo equivalente in una base  $b' \neq b$ ?
- Di nuovo abbiamo due regole generali che effettuano le operazioni aritmetiche coinvolte rispettivamente nella base partenza  $b$  e nella base di arrivo  $b'$



## Regola 1:

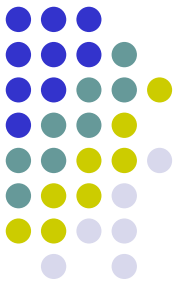
- svolge le operazioni nella base di arrivo  $b'$ , per cui è molto adatta al caso in cui  $b'=10$
- consiste nell'applicare in modo diretto la sommatoria (2) nel seguente modo:
  1. si esprimono le cifre  $a_i$  e la base  $b$  nella base  $b'$  (solitamente banale)
  2. si calcola la sommatoria (2)



## Esempi

- $0,1101_2 =$   
 $= 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} =$   
 $= 1/2 + 1/4 + 1/16 = 0,8125_{10}$
- $0,1011_2 =$   
 $= 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} =$   
 $= 1/2 + 1/8 + 1/16 = 0,6875_{10}$
- $0,452_8 =$   
 $= 4 \cdot 8^{-1} + 5 \cdot 8^{-2} + 2 \cdot 8^{-3} =$   
 $= 4/8 + 5/64 + 2/512 = 0,58203125_{10}$
- $0,1A8_{16} =$   
 $= 1 \cdot 16^{-1} + 10 \cdot 16^{-2} + 8 \cdot 16^{-3} =$   
 $= 1/16 + 10/256 + 8/4096 = 0,103515625_{10}$





## Regola 2 (delle moltiplicazioni successive):

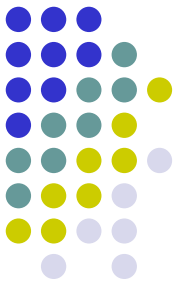
- svolge le operazione nella base di partenza  $b$ , per cui è molto adatta al caso in cui  $b=10$
- si basa sull'osservazione che, moltiplicando il numero per  $b'$ :
  - la parte intera corrisponde alla prima cifra dopo la virgola del numero nella base  $b'$
  - la parte frazionaria al numero ottenuto cancellando la prima cifra dopo la virgola dal numero di partenza espresso in base  $b'$
  - le cifre successive alla prima dopo la virgola possono essere quindi determinate riapplicando ricorsivamente lo stesso metodo alla parte frazionaria

### Esempio

$0.623 \cdot 10 = 6.23$ , ossia parte intera 6 e frazionaria 0.23

$0.23 \cdot 10 = 2.3$ , ossia parte intera 2 e frazionaria 0.3

$0.3 \cdot 10 = 3$ , ossia parte intera 3 e frazionaria 0



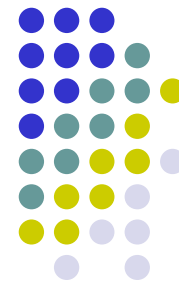
## Regola 2:

1. si determinano la parte intera e frazionaria della moltiplicazione del numero per  $b'$
2. si prosegue come al passo 1. considerando di volta in volta come numero di partenza la parte frazionaria della moltiplicazione effettuata nel passo precedente, finché non si determina una parte frazionaria nulla
3. si scrivono tutte le parti intere nell'ordine in cui sono state ottenute, esprimendole nella base  $b'$  (solitamente banale)

### NB:

- la prima parte intera ottenuta (al passo 1.) corrisponde alla prima cifra dopo la virgola, mentre l'ultima a quella più lontana
- se  $b' < b$  tutte le parti intere ottenute sono già espresse nella base  $b'$

# Esempi



1)  $0.375$  ( $b=10$  e  $b'=2$ )

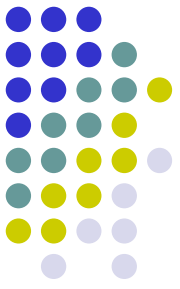
$$\begin{array}{l} 0.375 \cdot 2 = 0 + 0.75 \\ 0.75 \cdot 2 = 1 + 0.5 \\ 0.5 \cdot 2 = 1 + 0 \end{array}$$

Quindi  $0.375 = 0,011_2$

2)  $0.84375$  ( $b=10$  e  $b'=16$ )

$$\begin{array}{l} 0.84375 \cdot 16 = 13 (D) + 0.5 \\ 0.5 \cdot 16 = 8 (8) + 0 \end{array}$$

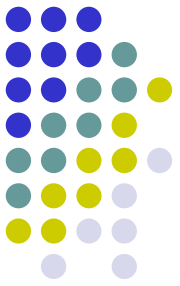
Quindi  $0.84375 = 0,D8_{16}$



# Nota

- I numeri frazionari possono introdurre approssimazioni dovute alla presenza di un numero limitato di cifre dopo la virgola
- L'approssimazione è comunque inferiore a  $b^{-n}$  dove  $n$  è il numero di cifre utilizzate

# Esempio 1



1)  $0.587$  ( $b=10$  e  $b'=2$ )

$$0.587 \cdot 2 = 1 + 0.174$$

$$0.174 \cdot 2 = 0 + 0.348$$

$$0.348 \cdot 2 = 0 + 0.696$$

$$0.696 \cdot 2 = 1 + 0.392$$

$$0.392 \cdot 2 = 0 + 0.784$$

$$0.784 \cdot 2 = 1 + 0.568$$

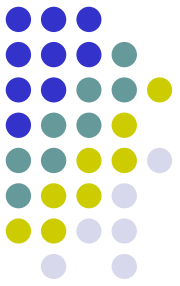
$$0.568 \cdot 2 = 1 + 0.136$$

...

$$0.587_{10} = 0.1001_2 \text{ (approssimazione } < 2^{-4} \text{)}$$

$$0.587_{10} = 0.1001011_2 \text{ (approssimazione } < 2^{-7} \text{)}$$

# Esempio 2



2)  $0.35$  ( $b=10$  e  $b'=2$ )

$$0.35 \cdot 2 = 0 + 0.7$$

$$0.7 \cdot 2 = 1 + 0.4$$

$$0.4 \cdot 2 = 0 + 0.8$$

$$0.8 \cdot 2 = 1 + 0.6$$

$$0.6 \cdot 2 = 1 + 0.2$$

$$0.2 \cdot 2 = 0 + 0.4$$

$$0.4 \cdot 2 = 0 + 0.8$$

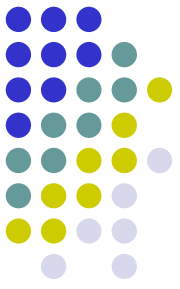
...

$m$

$$0.35_{10} = 0.010110..._2$$

$m$

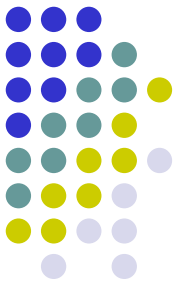
(approssimazione  $< 2^{-m}$ )



- Per codificare i numeri frazionari di nuovo viene utilizzato il sistema di numerazione posizionale in base 2
- Fissato il numero di bit  $k$  da utilizzare nella rappresentazione
  1. si converte il numero di partenza nella base 2
  2. si considerano solo i primi  $k$  bit dopo la virgola, eventualmente tagliando i rimanenti o aggiungendo bit uguali a 0 alla parte finale fino ad ottenere complessivamente esattamente  $k$  bit

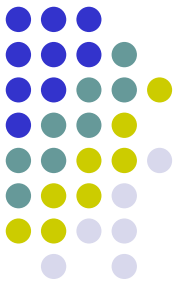
NB: non è necessario rappresentare lo 0 iniziale e la virgola, perché sono comuni a tutti i numeri frazionari

# Codifica numeri interi (positivi e negativi)



- Esistono diverse codifiche
- Tra le più note:
  - rappresentazione in modulo e segno
    - più semplice e diretta
  - rappresentazione in complemento a 2
    - ha il pregio di poter effettuare somme algebriche, ossia la sottrazione  $a-b$  equivale ad effettuare la somma  $a+(-b)$
    - a livello di circuiteria elettronica ciò consente di poter effettuare somme e sottrazioni in modo unificato tramite un unico dispositivo sommatore che opera sulle codifiche di  $a$  e  $b$
  - ...

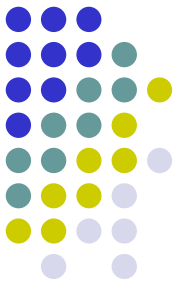




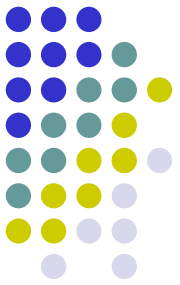
- Rappresentazione in modulo e segno a  $k$  bit:
  - 1 bit per il segno, solitamente il più significativo posto a 0 per indicare il segno + e ad 1 per indicare il segno –
  - $k-1$  bit per il modulo o valore assoluto, secondo la codifica dei numeri naturali
- L'intervallo dei numeri rappresentabili quindi è
$$[-(2^{k-1}-1), 2^{k-1}-1]$$
- Si noti che esistono due codifiche possibili per il numero 0, ossia  $00\dots 0$  (corrispondente a  $+0$ ) e  $10\dots 0$  (corrispondente a  $-0$ )

# Esempio 1

Rappresentazione in segno e modulo su tre bit



Interi	Segno/modulo
-4	-
-3	111
-2	110
-1	101
-0	100
+0	000
+1	001
+2	010
+3	011



# Esempio 2

Rappresentazione in modulo e segno a  $k=8$  bit dei numeri 26 e -26

$$26:2 = 13 \text{ con resto } 0$$

$$13:2 = 6 \text{ con resto } 1$$

$$6:2 = 3 \text{ con resto } 0$$

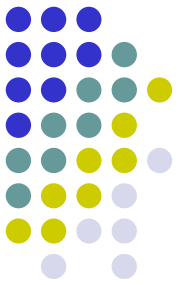
$$3:2 = 1 \text{ con resto } 1$$

$$1:2 = \mathbf{0} \text{ con resto } 1$$

Quindi  $26=11010_2$ , per cui

- rappresentazione di 26:  $00011010$

- rappresentazione di -26:  $10011010$



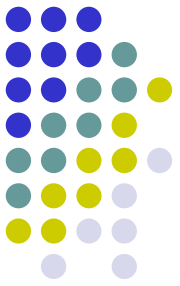
- Rappresentazione in complemento a 2 a  $k$  bit:

- Poiché esiste un'unica codifica del numero 0 (-0 non viene rappresentato), l'intervallo dei numeri rappresentabili è

$$[-2^{k-1}, 2^{k-1}-1]$$

- I numeri non negativi coincidono con la rappresentazione in modulo e segno
- Infatti la rappresentazione di un numero non negativo si ottiene semplicemente convertendolo in binario
- La rappresentazione di un numero negativo  $-N$  si ottiene facendo la conversione in binario del numero  $2^k-N$
- Una semplice regola di conversione:
  - si converte in binario il numero  $N$
  - si complementano tutti i bit
  - si somma 1

# Esempio 1 (1/3)



Rappresentazione in  $k=8$  bit dei numeri 26 e -26

$$26:2 = 13 \text{ con resto } 0$$

$$13:2 = 6 \text{ con resto } 1$$

$$6:2 = 3 \text{ con resto } 0$$

$$3:2 = 1 \text{ con resto } 1$$

$$1:2 = \mathbf{0} \text{ con resto } 1$$

Quindi  $26=11010_2$ , per cui

- rappresentazione di 26:

modulo e segno

00011010

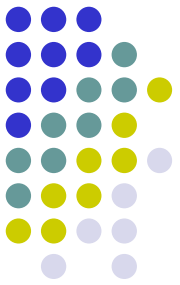
complemento a 2

00011010

- rappresentazione di -26:

10011010

# Esempio 1 (2/3)



$$2^8 - 26 = 256 - 26 = 230_{10}$$

$$230 : 2 = 115 \text{ con resto } 0$$

$$115 : 2 = 57 \text{ con resto } 1$$

$$57 : 2 = 28 \text{ con resto } 1$$

$$28 : 2 = 14 \text{ con resto } 0$$

$$14 : 2 = 7 \text{ con resto } 0$$

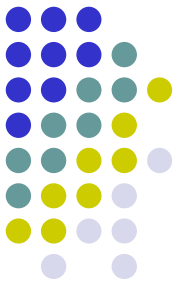
$$7 : 2 = 3 \text{ con resto } 1$$

$$3 : 2 = 1 \text{ con resto } 1$$

$$1 : 2 = 0 \text{ con resto } 1$$

$$-26_{10} = 11100110_2$$

# Esempio 1 (3/3)



Rappresentazione in  $k=8$  bit dei numeri 26 e -26

$$26:2 = 13 \text{ con resto } 0$$

$$13:2 = 6 \text{ con resto } 1$$

$$6:2 = 3 \text{ con resto } 0$$

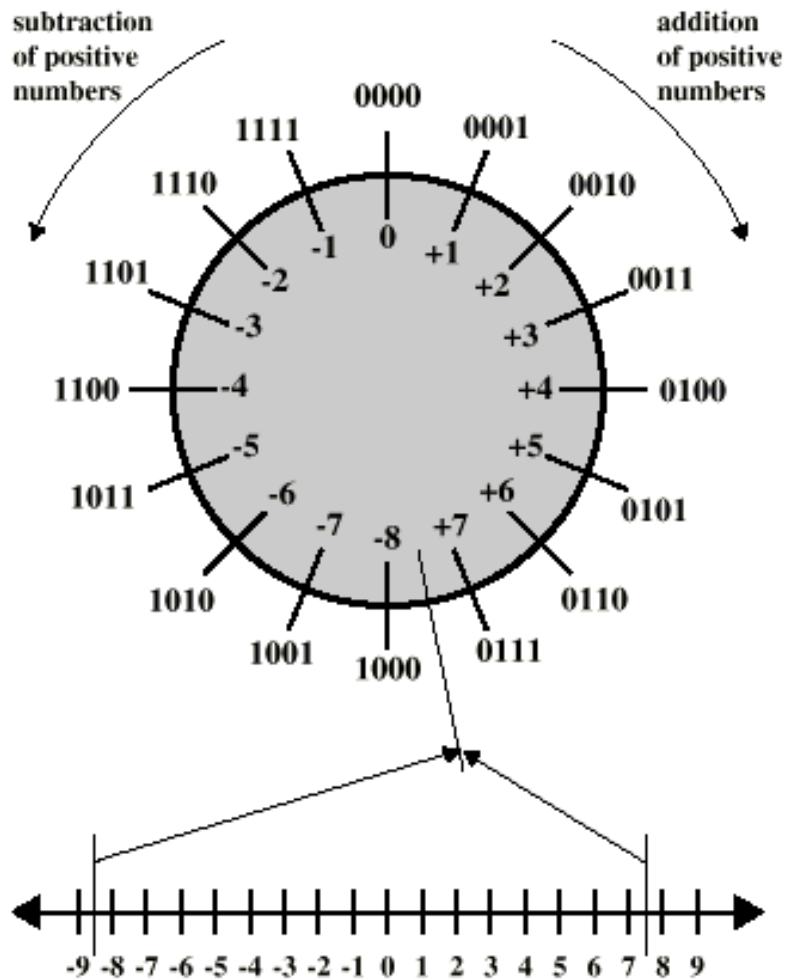
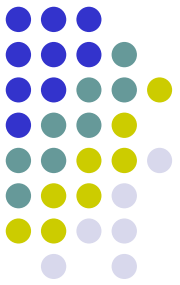
$$3:2 = 1 \text{ con resto } 1$$

$$1:2 = \mathbf{0} \text{ con resto } 1$$

Quindi  $26=11010_2$ , per cui

	modulo e segno	complemento a 2
- rappresentazione di 26:	00011010	00011010
- rappresentazione di -26:	10011010	11100110

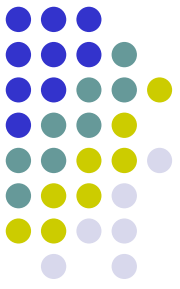
# Esempio 2



(a) 4-bit numbers

Valore binario $b_3b_2b_1b_0$	Notazione modulo e segno	Notazione complemento a 2
0000	+0	0
0001	+1	+1
0010	+2	+2
0011	+3	+3
0100	+4	+4
0101	+5	+5
0110	+6	+6
0111	+7	+7
1000	-0	-8
1001	-1	-7
1010	-2	-6
1011	-3	-5
1100	-4	-4
1101	-5	-3
1110	-6	-2
1111	-7	-1



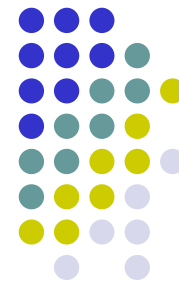


# Nota

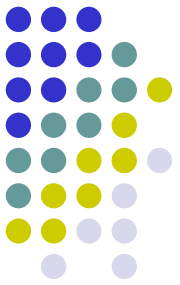
- Per fare la codifica di un numero devo capire a quale range appartiene
- Ad esempio se  $N=-74$  e considerato il range  $[-2^{m-1}, 2^{m-1}-1]$ , è necessario che  $m=8$  in modo da ottenere  $[-2^7, 2^7-1]=[-128, 127]$  ( $-74 \in [-128, 127]$ )
- Per  $N=74$ ,  $m=7$  considerando però il range  $[0, 2^7-1] = [0, 127]$  ( $74 \in [0, 127]$ )

Quando ho numeri negativi devo raddoppiare il range in quanto metà serve per la rappresentazione dei numeri positivi e metà per quelli negativi

# Codifica numeri reali

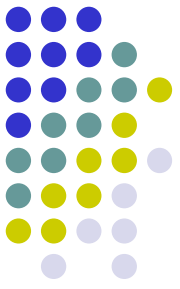


- Esistono due codifiche fondamentali:
  - rappresentazione in virgola fissa
    - più semplice e diretta
    - $k_1$  bit per la parte intera secondo la codifica dei numeri interi
    - $k_2$  bit per la parte frazionaria secondo la codifica dei numeri frazionari
    - NB: una volta stabilito che i primi  $k_1$  bit sono per la parte intera e i rimanenti  $k_2$  bit per la parte frazionaria, non è chiaramente necessario codificare la virgola
  - rappresentazione in virgola mobile (floating point)
    - si basa sull'osservazione che l'errore (assoluto) per la rappresentazione deve essere piccolo per numeri piccoli e può esser grande per numeri grandi
    - in altre parole, l'errore che bisogna limitare è l'errore percentuale, ossia l'errore assoluto rapportato alla grandezza del numero
    - ciò ha il pregio di poter rappresentare da un lato numeri reali, dall'altro numeri reali molto piccoli con precisione molto grande



## Rappresentazione in virgola fissa:

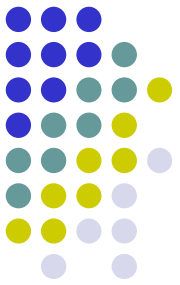
- Per la rappresentazione dei numeri reali è sufficiente giustapporre due numeri (un intero ed un frazionario)
- Es: *00101001011,10110*
  - La parte intera consta di 11 bit e la parte frazionaria di 5 bit
  - Corrisponde al numero reale  $331,6875_{10}$



- Se ho una rappresentazione in virgola fissa (es. su segno e 10 cifre con 4 cifre alla destra della virgola) rappresento numeri (base 10) compresi fra

-999999.9999      e      999999.9999

- Non posso rappresentare, quindi:
  - numeri che richiedono più di 6 cifre intere, cioè maggiori di 999999.9999 (es. 1000000)
  - numeri che richiedono più di 4 cifre alla destra della virgola, come ad es. 130.00001



## Rappresentazione in virgola mobile:

Un numero reale viene espresso nella forma

$$N = \pm m \cdot p^e$$

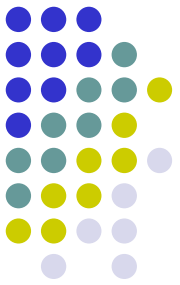
dove

- $m$  è un numero frazionario (tra 0 ed 1) chiamato **mantissa**, solitamente con la prima cifra dopo la virgola non nulla (rappresentazione normalizzata)
- $e$  è un numero intero chiamato esponente o **caratteristica**
- $p$  è la **base della rappresentazione**

## Esempi ( $p=10$ )

- $10842:$        $0,10842 \cdot 10^5$
- $0,000013:$        $0,13 \cdot 10^{-4}$

Moltiplicare (dividere) un numero per una potenza della base equivale a far scorrere a sinistra (destra) il numero di un numero di posizioni pari all'esponente, ovvero a spostare la virgola di un uguale numero di posizioni in senso opposto.



Un numero in virgola mobile si dice **normalizzato** se la posizione più significativa della mantissa contiene una cifra diversa dallo zero

Esempio:

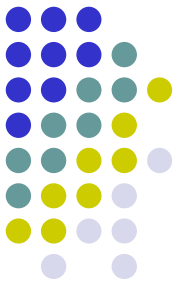
$$+0,45676 \cdot 10^2$$

(normalizzato)

$$+0,0045676 \cdot 10^4$$

(non normalizzato)

# Esempio (1/2)



$$N = m \cdot p^e$$

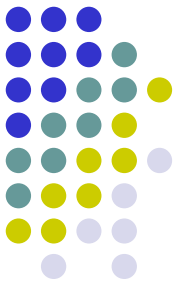
$$m = 1011$$

$$p = 2$$

$$e = 01010$$

in decimale  $N$  a quanto corrisponde?

# Esempio (2/2)



Ricordiamo che:

$$N_b = (0, a_{-1} \dots a_{-n})_b = a_{-1} \cdot b^{-1} + \dots + a_{-n} \cdot b^{-n} = \sum_{i=-1}^{-n} a_i \cdot b^i \quad (2)$$

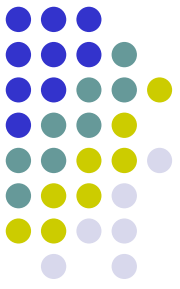
Quindi:

$$m_{10} = 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} = 0,6875$$

$$e_{10} = 10$$

$$N_{10} = 0,6875 \cdot 2^{10} = 0,6875 \cdot 1024 = 704,01$$





A livello di codifica:

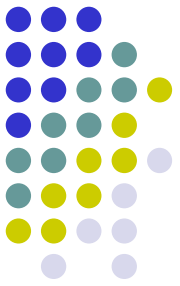
- 1 bit per il segno
- $k_1$  bit per la mantissa  $m$  secondo la codifica dei numeri frazionari
- $k_2$  bit per la caratteristica e secondo la codifica dei numeri interi

Se la caratteristica viene rappresentata in modulo e segno a  $k_2$  bit, l'intervallo dei numeri rappresentabili è

$$\left[ -(1-2^{-k_1}) \cdot p^{2^{(k_2-1)}-1}, (1-2^{-k_1}) \cdot p^{2^{(k_2-1)}-1} \right]$$

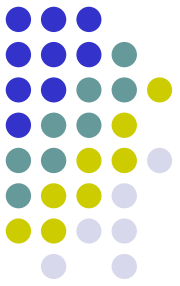
$$err.rel. = err.ass/n = |n-r(n)|/n = 2^{-k_1} p^e/n \approx 2^{-k_1} p^e/p^e = 2^{-k_1}$$

# Osservazioni conclusive sui numeri reali



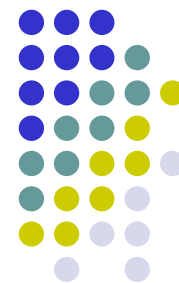
- L'utilizzo di memoria finita (numero finito di bit) per rappresentare i reali comporta, oltre ad un limite superiore al massimo numero rappresentabile, un errore di rappresentazione dovuto al fatto che una precisione infinita richiede un numero infinito di bit
- Come conseguenza, in un elaboratore non possono essere rappresentati numeri reali, ma solo loro approssimazioni razionali
- Tali approssimazioni nella rappresentazione in virgola fissa sono distribuite uniformemente, ossia ognuna alla stessa distanza ( $2^{-k_2}$ ) dalla successiva
- Nella rappresentazione in virgola mobile sono molto vicine per numeri piccoli e si allontanano progressivamente al crescere dei numeri
- Calcolo numerico: disciplina che studia la propagazione dell'errore di rappresentazione al susseguirsi delle operazioni

# Codifica immagini



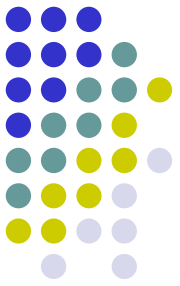
- Anche le immagini vengono codificate come una sequenza di bit
- Il passaggio da un'immagine a una sequenza binaria prende il nome di **digitalizzazione**
- Un'immagine viene vista come un insieme di punti (**pixel**) giacenti su una griglia bidimensionale
- Ad ogni punto viene associata un insieme di bit che ne codificano il colore o la tonalità di grigio (per bianco e nero)
- **Risoluzione**: numero di punti rappresentati per unità di superficie (punti per pollice quadrato o *dot per inch*)
- Grazie all'alta località, ossia al fatto che punti vicini tendono ad essere molto simili (basti pensare allo sfondo), è possibile utilizzare rappresentazioni o formati compressi in grado di ridurre il numero totale di bit utilizzati
- Le tecniche di compressione si distinguono in **tecniche lossless** e **tecniche lossly**
- Tra i formati più noti: TIFF, GIF, JPEG, ...

# Osservazioni codifica immagini



- Come i numeri reali, le immagini hanno uno spettro continuo, mentre le codifiche uno spettro discreto (come le approssimazioni razionali dei reali)
- Quindi nella digitalizzazione, ossia nella codifica di immagini con sequenze di bit (di lunghezza finita), si commette necessariamente un errore di rappresentazione
- All'aumentare della risoluzione e del numero di colori rappresentati, tale errore diventa impercettibile
- Al contrario, eventuali errori dovuti a trasmissioni, deterioramento del supporto di memorizzazione, malfunzionamenti hardware, ..., rispetto alle precedenti codifiche analogiche entro certi limiti di rumore sono recuperabili perché
  1. è più facile distinguere tra stati discreti (es. bit pari a 0 o ad 1) che tra stati continui (es. tra due reali)
  2. è possibile utilizzare codici a correzioni di errore capaci di correggere un numero limitato di bit comunque alterati

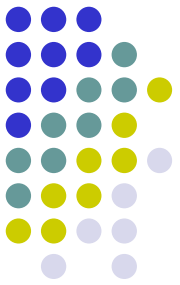
# Codifiche analogiche e digitali



- La seguente tabella riassume le differenze fondamentali tra codifiche analogiche e digitali:

	Codifiche analogiche	Codifiche digitali
Errore di rappresentazione	Teoricamente nullo	Ammesso ma impercettibile
Separazione codifica originale - rumore	Generalmente impossibile	Possibile entro certi limiti di rumore

- Tali differenze sono direttamente riscontrabili confrontando tra loro TV terrestre e TV satellitare (digitale), dischi in vinile - cassette audio (analogici) e CD audio (digitali), foto su pellicola e foto digitali, ....



# Punti Chiave

- Codifica numeri
  - Codifica numeri frazionari
  - Codifica numeri interi
  - Codifica numeri reali
- Codifica immagini