La rappresentazione degli interi negativi e dei reali

Liceo G.B. Brocchi
Classi prime Scientifico – opzione scienze applicate
Bassano del Grappa, Gennaio 2023
Prof. Giovanni Mazzocchin

Gli interi con segno

- Finora abbiamo interpretato le stringhe di bit (le sequenze di *uni e zeri*) contenute nei registri e nelle celle di memoria soltanto come numeri interi positivi (se stiamo parlando di dati, non di istruzioni):
  - ad esempio, la sequenza di bit *1100,* finora, ha sempre rappresentato il numero decimale *12*
- Ovviamente, vorremmo scrivere programmi che siano in grado di lavorare anche sugli interi negativi, sui reali, sui caratteri, sulle stringhe di caratteri etc…
- Dobbiamo studiare dei metodi per rappresentare tutti questi oggetti della realtà diversi dagli interi positivi, ma sempre tramite sequenze di 1 e 0, perché internamente un calcolatore digitale non conosce niente altro oltre ai bit! Non possiamo scrivere cose come «virgole» e «segni meno» in memoria…

Codifica con segno e modulo (*sign-magnitude*)

- **Nella codifica con *segno e modulo*, il bit più a sinistra (più significativo - MSB) rappresenta il segno del numero:**
    - **0 per il +**
    - **1 per il –**
    - i bit rimanenti rappresentano il modulo (valore assoluto) del numero
- Bisogna stabilire a priori quanti bit si utilizzano per la codifica
- Ipotizziamo di voler rappresentare i numeri interi con segno utilizzando 4 bit

- **NB:** *modulo* significa *valore assoluto*

Codifica con segno e modulo – 4 bit

Abbiamo utilizzato 4 bit per rappresentare gli interi relativi con segno e modulo

Come potete notare, ovviamente le permutazioni sono sempre 16 (2^4), ma se prima utilizzavamo le stesse permutazioni per rappresentare i numeri interi positivi nell'intervallo 0-15, ora le usiamo per rappresentare i numeri relativi nell'intervallo -7 – +7

Ad esempio: prima di questa lezione, 1111 significava 15, mentre con questa rappresentazione significa -7

decimale	binario - segno e modulo
+7	**0** 1 1 1
+6	**0** 1 1 0
+5	**0** 1 0 1
+4	**0** 1 0 0
+3	**0** 0 1 1
+2	**0** 0 1 0
+1	**0** 0 0 1
+0	**0** 0 0 0
-0	**1** 0 0 0
-1	**1** 0 0 1
-2	**1** 0 1 0
-3	**1** 0 1 1
-4	**1** 1 0 0
-5	**1** 1 0 1
-6	**1** 1 1 0
-7	**1** 1 1 1

Codifica con segno e modulo – 4 bit

Gli interi opposti differiscono soltanto per il bit di segno, quello più a sinistra (*Most Significant Bit*)

Ad esempio: 0 1 0 1 e 1 1 0 1 codificano, rispettivamente, $+5_{dec}$ e -5_{dec}

NB: lo 0 è rappresentato 2 volte, come +0 e come -0. Non è utile rappresentare 2 volte lo 0, che non è né negativo, né positivo. Questa rappresentazione sarebbe molto scomoda per una macchina, che deve valutare molto spesso se il risultato di un'operazione è 0

decimale	binario - segno e modulo
+7	**0** 1 1 1
+6	**0** 1 1 0
+5	**0** 1 0 1
+4	**0** 1 0 0
+3	**0** 0 1 1
+2	**0** 0 1 1
+1	**0** 0 0 1
+0	**0** 0 0 0
-0	**1** 0 0 0
-1	**1** 0 0 1
-2	**1** 0 1 0
-3	**1** 0 1 1
-4	**1** 1 0 0
-5	**1** 1 0 1
-6	**1** 1 1 0
-7	**1** 1 1 1

Codifica con segno e modulo – 4 bit

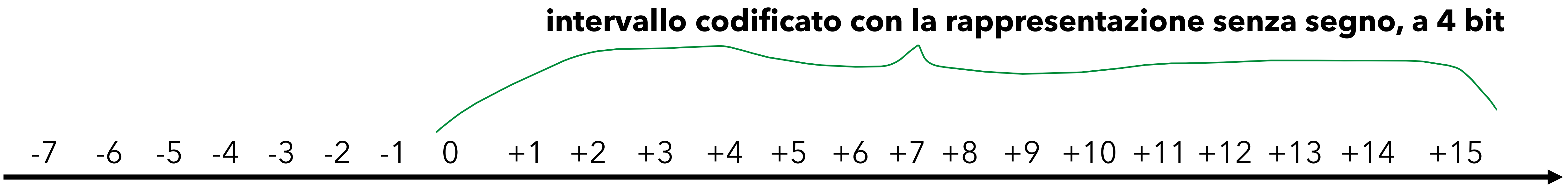
Con n bit, la rappresentazione segno e modulo degli interi relativi codifica l'intervallo:

$[-2^{n-1} + 1, 2^{n-1} - 1]$

In questo esempio:
$[-2^{4-1} + 1, 2^{4-1} - 1] =$
$[-2^{3} + 1, 2^{3} - 1] =$
$[-7, +7]$

decimale	binario - segno e modulo
+7	**0** 1 1 1
+6	**0** 1 1 0
+5	**0** 1 0 1
+4	**0** 1 0 0
+3	**0** 0 1 1
+2	**0** 0 1 1
+1	**0** 0 0 1
+0	**0** 0 0 0
-0	**1** 0 0 0
-1	**1** 0 0 1
-2	**1** 0 1 0
-3	**1** 0 1 1
-4	**1** 1 0 0
-5	**1** 1 0 1
-6	**1** 1 1 0
-7	**1** 1 1 1

Una visualizzazione utile



Una visualizzazione utile

intervallo codificato con la rappresentazione con segno e modulo, a 4 bit

-7 -6 -5 -4 -3 -2 -1 0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +10 +11 +12 +13 +14 +15

NB: la dimensione dell'intervallo è sempre la stessa e dipende dal numero di bit usati per la codifica (n bit → 2^n permutazioni)

La complementazione

- La complementazione, o metodo dei complementi, è una tecnica utilizzata per codificare intervalli simmetrici di interi relativi, utilizzata per semplificare l'operazione di sottrazione (sia nei moderni calcolatori elettronici, sia nelle vecchie calcolatrici meccaniche)

- Esplicitiamo il concetto in base 10, prima di procedere con i numeri binari

- **Il complemento a 9 di una cifra decimale *x* è la cifra *y* per cui:**

$$\mathbf{x + y = 9}$$

La complementazione

cifra	complemento a 9
0	9
1	8
2	7
3	6
4	5
5	4
6	3
7	2
8	1
9	0

La complementazione

- Il complemento a **1** di una cifra binaria **x** è quella cifra binaria per cui:

$$\mathbf{x + y = 1}$$

- Qui non abbiamo tanta scelta …

cifra	complemento a 1
0	1
1	1

La complementazione

- Per calcolare il **complemento a 9 di un numero decimale**, si complementa a 9 ciascuna cifra:

8 4 3 9_dec

1 5 6 0_dec

La somma tra un numero decimale di 4 cifre e il suo complemento a 9 dà come risultato 9999

La complementazione

- Per calcolare il **complemento a 9 di un numero decimale**, si complementa a 9 ciascuna cifra:

2 3 1_dec

↓ ↓ ↓

7 6 8_dec

La somma tra un numero decimale di 4 cifre e il suo complemento a 9 dà come risultato 999

La complementazione

- Per calcolare il **complemento a 1 di un numero binario**, si complementa a 1 ciascuna cifra, ossia la si inverte:

$$\begin{array}{ccc} 1 & 0 & 1\_bin \\ \downarrow & \downarrow & \downarrow \\ 0 & 1 & 0\_bin \end{array}$$

La somma tra un numero binario di 4 cifre e il suo complemento a 1 dà come risultato 111

La complementazione

- Per calcolare il **complemento a 1 di un numero binario**, si complementa a 1 ciascuna cifra, ossia la si inverte:

$$
\begin{array}{cccc}
1 & 1 & 1 & 0\_bin \\
\downarrow & \downarrow & \downarrow & \downarrow \\
0 & 0 & 0 & 1\_bin
\end{array}
$$

La somma tra un numero binario di 4 cifre e il suo complemento a 1 fa 1111

Il complemento a 2

- Per calcolare il **complemento a 2 di un numero binario b**
  1. si calcola il complemento a 1 di b, ossia si invertono tutti i suoi bit
  2. si somma 1 al risultato
- L'analogo decimale del complemento a 2 sarebbe il complemento a 10, pensateci

twos_complement(1 0 1 1) = 0 1 0 0 + 1 = 0 1 0 1
twos_complement(1 1 1 1) = 0 0 0 0 + 1 = 0 0 0 1
twos_complement(1 0 0 1 0) = 0 1 1 0 1 + 1 = 0 1 1 1 0

Il complemento a 2

- **NB:** la somma di un numero binario di n cifre e del suo <u>complemento a 2</u> dà come risultato 1 seguito da n zeri. Questo fatto deriva proprio dalla definizione di complemento a 2

```
  1 0 1 1 +
  0 1 0 1 =
1 0 0 0 0
```

```
  1 0 0 1 0 +
  0 1 1 1 0 =
1 0 0 0 0 0
```

```
  1 1 1 1 +
  0 0 0 1 =
1 0 0 0 0
```

Il complemento a 2

- Esiste un metodo veloce per calcolare il complemento a 2 di un numero binario di n cifre

- *Partire dalla cifra più a destra e mantenere le cifre inalterate fino a che non si incontra il primo 1, compreso. Dopodiché si invertono tutte le cifre seguenti (verso sinistra)*

- Ricordatevi di procedere da destra

Il complemento a 2

0	1	1	0	0	0	1	0
1	0	0	1	1	1	1	0

cifre invertite dopo il primo 1 — **cifre inalterate fino al primo 1 compreso**

Ricordatevi che si parte da destra

Rappresentazione degli interi negativi in complemento a 2

- **Sorpresa**: il complemento a 2 è utilizzatissimo nei calcolatori digitali per rappresentare gli interi negativi, al posto della rappresentazione in segno e modulo:

<u>Codifica in complemento a 2:</u>
la rappresentazione degli interi positivi è identica a quella segno e modulo. Per gli interi negativi si complementa a 2 il corrispondente intero positivo

Rappresentazione degli interi negativi in complemento a 2

- **Esempio**: per rappresentare con 4 bit il numero $-6_{dec:}$
  - si effettua la conversione di $+6_{dec}$ nella rappresentazione segno e modulo: 0 1 1 0
  - si complementa a 2 il risultato del passo precedente: **1 0 1 0**

- <u>***Anche con questa rappresentazione, come per quella segno e modulo, il bit più significativo indica il segno del numero***</u>

Rappresentazione degli interi negativi in complemento a 2 – 4 bit

decimale	binario - complemento a 2
+7	**0** 1 1 1
+6	**0** 1 1 0
+5	**0** 1 0 1
+4	**0** 1 0 0
+3	**0** 0 1 1
+2	**0** 0 1 0
+1	**0** 0 0 1
0	**0** 0 0 0
-1	**1** 1 1 1
-2	**1** 1 1 0
-3	**1** 1 0 1
-4	**1** 1 0 0
-5	**1** 0 1 1
-6	**1** 0 1 0
-7	**1** 0 0 1
-8	**1** 0 0 0

complemento a 2

Rappresentazione degli interi negativi in complemento a 2 – 4 bit

Ora abbiamo solo 1 rappresentazione dello 0, a differenza delle 2 rappresentazioni con segno e modulo

Notare che l’intero più piccolo rappresentabile in complemento a 2, con 4 bit, è -8;

Abbiamo quindi utilizzato la permutazione che prima rappresentava -0 per rappresentare un numero negativo in più

decimale	binario - complemento a 2
+7	**0** 1 1 1
+6	**0** 1 1 0
+5	**0** 1 0 1
+4	**0** 1 0 0
+3	**0** 0 1 1
+2	**0** 0 1 0
+1	**0** 0 0 1
0	**0** 0 0 0
-1	**1** 1 1 1
-2	**1** 1 1 0
-3	**1** 1 0 1
-4	**1** 1 0 0
-5	**1** 0 1 1
-6	**1** 0 1 0
-7	**1** 0 0 1
-8	**1** 0 0 0

Rappresentazione degli interi negativi in complemento a 2

- Vi starete chiedendo da dove salta fuori quel -8, visto che non c'è +8 nell'elenco
- Attenzione: +8 in binario è 1 0 0 0, e per rappresentarlo con il segno diventerebbe: 0 1 0 0 0. Evidentemente servono 5 bit e non 4
- Possiamo rappresentare il -8 perché «avanzava» la permutazione

1 0 0 0: notate che il complemento a 2 di 1 0 0 0 è ancora 1 0 0 0

Con n bit, la rappresentazione in complemento a 2 degli interi relativi codifica l'intervallo:
$[-2^{n-1}, 2^{n-1} - 1]$

In questo esempio:
$[-2^{4-1}, 2^{4-1} - 1] =$
$[-2^{3}, 2^{3} - 1] =$
$[-8, +7]$

Rappresentazione degli interi negativi in complemento a 2

- **Drill**: ipotizziamo che `0xFFFF` sia un numero intero rappresentato in complemento a 2. Cosa possiamo dire di `0xFFFF`?
- `0x F F F F = 0b 1111 1111 1111 1111`
- Innanzitutto, il numero è rappresentato con 16 bit. Sappiamo che la rappresentazione è in complemento a 2 e il bit più significativo è 1, quindi il numero è negativo
- Per trovare il modulo (*valore assoluto*) del numero, calcoliamone il complemento a 2, con la scorciatoia:

`0000 0000 0000 0001`

- Si deduce che 0xFFFF è la rappresentazione in complemento a 2 di -1, su 16 bit

Operazioni in complemento a 2

- La rappresentazione in complemento a 2 è molto utile per semplificare i circuiti che effettuano le operazioni aritmetiche. Con questa rappresentazione, addizioni e sottrazioni vengono effettuate dagli stessi circuiti
- In particolare, questa rappresentazione permette di evitare i prestiti nelle sottrazioni

- Per calcolare A – B:
    - si rappresenta A in complemento a 2
    - si rappresenta -B in complemento a 2
    - si effettua la normale addizione binaria tra le 2 rappresentazioni ottenute

Operazioni in complemento a 2

69_dec - 12_dec

0 1 0 0 0 1 0 1 -
0 0 0 0 1 1 0 0 =
0 0 1 1 1 0 0 1

finora abbiamo sempre effettuato la sottrazione così, convertendo in binario i numeri e sottraendoli

Operazioni in complemento a 2

Calcoliamo il complemento a 2 di $12_{dec} = 00001100_{bin}$

twos_complement(00001100) = 11110100

Operazioni in complemento a 2

0 1 0 0 0 1 0 1 +
1 1 1 1 0 1 0 0 =
0 0 1 1 1 0 0 1

Effettuiamo l'addizione tra 69 e -12, rappresentando -12 come complemento a 2 di 12.
Ignoriamo l'1 risultato dell'ultimo riporto.

Notazione esponenziale

- **<u>Notazione esponenziale</u>**: un numero decimale è in **forma esponenziale** quando è espresso in questo modo:

$$\boldsymbol{a \cdot 10^n \qquad con\ n\ intero}$$

NB: La notazione esponenziale non è univoca. È infatti possibile spostare la virgola e scalare l'esponente in infiniti modi.

- Esempi:

$$120000 = 12 \cdot 10^4 = 1.2 \cdot 10^5$$
$$0{,}000016 = 1.6 \cdot 10^{-5} = 16 \cdot 10^{-6}$$

- Notazione «informatica», da provare su Python:

$$120000 = 12E4 = 1.2E5$$
$$1.6 \cdot 10^{-5} = 1.6E-5 = 16E-6$$

- Questa notazione è utile per semplificare i calcoli con quantità molto grandi o molto piccole

Notazione scientifica

- **<u>Notazione scientifica</u>**: un numero decimale è in scritto in notazione scientifica quando è espresso in questo modo:

$$\boldsymbol{p \cdot 10^n \qquad con\ 1 \leq p \leq 9\ e\ n\ intero}$$

- Esempi:

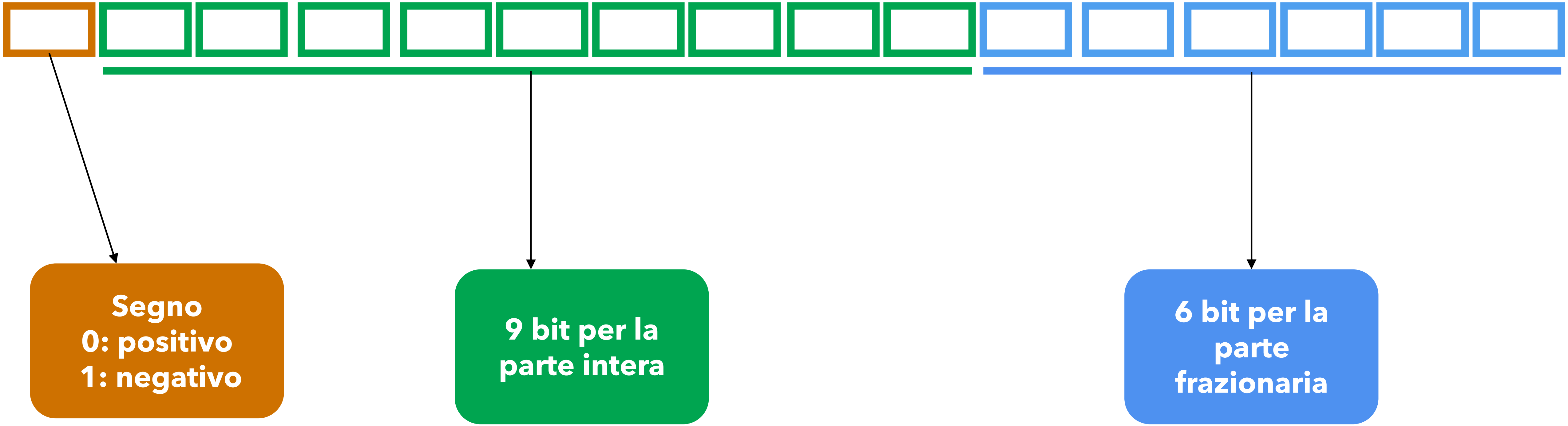
$$120000 = 1.2 \cdot 10^5 = 1.2\text{E}5$$
$$0{,}000016 = 1.6 \cdot 10^{-5} = 1.6\text{E} - 5$$

Rappresentazione in *virgola fissa*

- Avete sicuramente già capito che i numeri, all'interno dei sistemi di elaborazione elettronici digitali, sono memorizzati in locazioni di memoria di dimensione **prefissata** e **finita**. La dimensione della locazione si misura in *bit*
- Sicuramente il numero reale irrazionale *pi greco* non è rappresentabile completamente, in quando ha infinite cifre decimali

Rappresentazione in <u>virgola fissa</u>
la rappresentazione in virgola fissa prevede che la locazione che memorizza il numero sia suddivisa in: bit di segno, bit della parte intera, bit della parte frazionaria

Rappresentazione in virgola fissa su 16 bit



Rappresentazione in virgola fissa su 16 bit

Rappresentiamo il numero binario +110.101

0	0	0	0	0	0	0	**1**	**1**	**0**	**1**	**0**	**1**	0	0	0

$$+110.101 = 2^2 + 2^1 + 2^{-1} + 2^{-3}$$

Rappresentiamo il numero binario +0.000001

0	0	0	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Non siamo riusciti a rappresentarlo!

Rappresentazione in virgola fissa su 16 bit

Rappresentiamo il numero binario +101111000010.101

0	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	0	**1**	0	0	0

Non siamo riusciti a rappresentarlo! Ci siamo persi i 4 bit più significativi della parte intera

Rappresentiamo il numero binario +0.10100001

0	0	0	0	0	0	0	0	0	**0**	**1**	**0**	**1**	**0**	**0**	**0**

Ci siamo persi i 2 bit meno significativi della parte frazionaria!

Rappresentazione in *virgola mobile*

- Rappresentazione in **virgola mobile (floating point)**:

  in un sistema di numerazione in base ***b***, qualunque numero ***n*** si può esprimere nella forma:

$$n = m \cdot b^e$$

m: mantissa

e: esponente

- $4532.987 = 0.4532987 \cdot 10^4$

La notazione in cui la parte intera della mantissa è 0, e la cifra più significativa del numero da rappresentare si trova subito a destra della virgola, viene detta forma normalizzata
Il concetto è del tutto analogo a quello di notazione scientifica

Rappresentazione in *virgola mobile*

- Rappresentazione in **virgola mobile (floating point)**:

  in un sistema di numerazione in base ***b***, qualunque numero ***n*** si può esprimere nella forma:

$$n = m \cdot b^e$$

m: mantissa

e: esponente

- $4532.987 = 0.4532987 \cdot 10^4$

La virgola è mobile perché può essere spostata di un numero arbitrario di posizioni, scalando l'esponente di conseguenza

Lo standard *IEEE 754*

- IEEE 754 single precision: tipo **float** del C (32 bit)
- IEEE 754 double precision: tipo **double** del C (64 bit)
  - 1 bit per il segno
  - 11 bit per l’esponente
  - 52 bit per la mantissa
  - non serve rappresentare la base, dato che è sempre 2

- Il C sta alla base di quasi tutti i linguaggi di programmazione e dei sistemi operativi, quindi quello che stiamo studiando non riguarda il C, ma l’informatica in generale

Alcuni errori di approssimazione evidenti

- Aprite una shell Python e lanciate i seguenti calcoli:
  - `import math`
  - `math.sqrt(3) * math.sqrt(3)`
  - `0.3 - 0.2`
  - `0.3 - 0.2 == 0.1 - 0.0`
  - `math.sqrt(2) * math.sqrt(2)`
  - `2 ** 4 == math.sqrt(2) ** 8`
  - `2 ** 3 == math.sqrt(2) ** 6`

Alcuni errori di approssimazione evidenti

Verifichiamo come si propagano gli errori di approssimazione dovuti alla rappresentazione macchina dei reali. Eseguiamo questi calcoli:

- `sqrt(2)`^{2} (dovrebbe risultare 2^1)
- `sqrt(2)`^{4} (dovrebbe risultare 2^2)
- `sqrt(2)`^{6} (dovrebbe risultare 2^3)
- `sqrt(2)`^{8} (dovrebbe risultare 2^4)
- `sqrt(2)`^{10} (dovrebbe risultare 2^5)
- `sqrt(2)`^{12} (dovrebbe risultare 2^6)
- etc…

Alcuni errori di approssimazione evidenti

```c
for (unsigned int exp = 1; exp <= 32; exp++) {
    long double n1 = pow(2, exp);
    long double n2 = pow(sqrt(2), exp * 2);
    printf("exp:%3u; ", exp);
    printf("error: %Le\n", n1 - n2);
}
```

Rappresentazione macchina di un double (*IEEE 754 double precision – 64 bit*)

0 10000000011 1011100100010000000000000000000000000000000000000000

sign bit

characteristic's bits (exponent)

mantissa bits

interessante… ma che numero stiamo rappresentando?

Rappresentazione macchina di un double (*IEEE 754 double precision – 64 bit*)

Formula per la conversione da *IEEE 754 double precision* a decimale:

$$-1^{s} * 2^{c-1023} * (1 + f)$$

s: bit di segno
c: esponente convertito in decimale
f: mantissa convertita in decimale

Rappresentazione macchina di un double (*IEEE 754 double precision – 64 bit*)

Formula per la conversione da *IEEE 754 double precision* a decimale:

$$-1^{s} * 2^{c-1023} * (1 + f)$$

0 10000000011 1011100100010000000000000000000000000000000000000000

s **c** **f**

s: 0

c: 1027

f: $2^{-1} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-8} + 2^{-12} = 0.722900390625$

Rappresentazione macchina di un double (*IEEE 754 double precision – 64 bit*)

Formula per la conversione da *IEEE 754 double precision* a decimale:

$$-1^{s} * 2^{c-1023} * (1 + f)$$

0 10000000011 1011100100010000000000000000000000000000000000000000

s: 0

c: 1027

f: $2^{-1} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-8} + 2^{-12} = 0.722900390625$

→ sostituendo: $-1^{0} * 2^{4} * (1 + 0.722900390625) = 27.56640625$