# Codifica dei numeri

# Codifica dei numeri

Come si rappresentano in un elaboratore digitale

- Numeri naturali
- Numeri interi
- Numeri reali

# Numeri naturali

# Numeri naturali in varie basi

250 in base 10, in base 2 e in base 16 (*A*=10, *B*=11, *C*=12, … *F*=15):

$$(250)_{10} = (11111010)_2 = (FA)_{16}$$

| Esponente | 2 | 1 | 0 |
|---|---|---|---|
| | 2 | 5 | 0 |
| Base | 10 | 10 | 10 |

$2 \times 10^2 + 5 \times 10^1 + 0 \times 10^0$

| Esponente | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| Base | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |

$1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$

| Esponente | 1 | 0 |
|---|---|---|
| | *F* | *A* |
| Base | 16 | 16 |

$15 \times 16^1 + 10 \times 16^0$

# Numeri naturali in un elaboratore digitale

In un elaboratore digitale, i numeri naturali sono rappresentati dalla loro conversione in binario

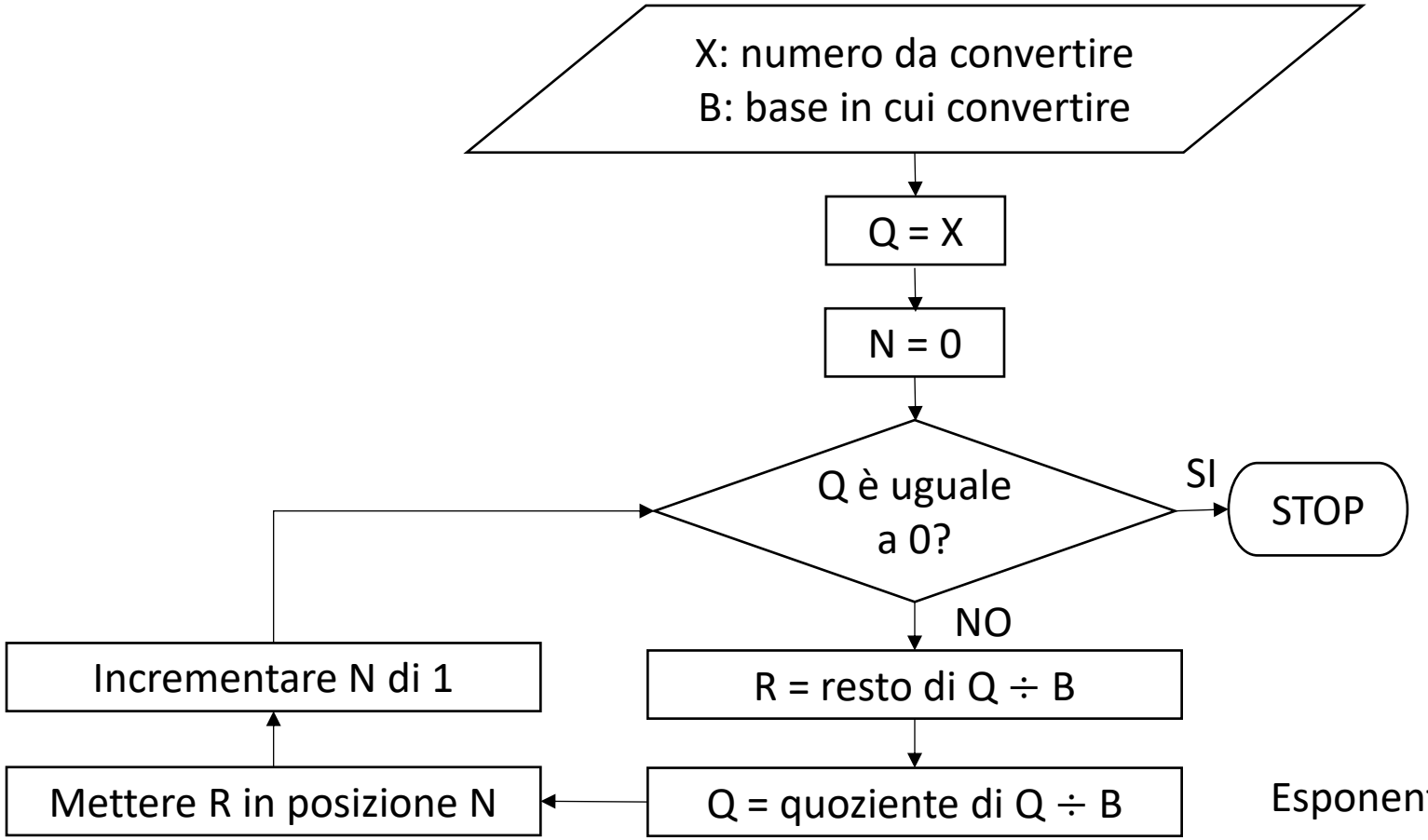
Come convertire un numero da base 10 ad un’altra base B?

# Algoritmo di conversione

Dati in ingresso un numero X (in base 10) da convertire e una base B in cui convertire

1. Assegna X ad una variabile Q
2. Assegna ad un contatore N il valore 0
3. Se Q è uguale a zero ferma l'algoritmo, altrimenti esegui i passi 3a – 3f
   a) Assegna ad una variabile R il resto della divisione di Q per B
   b) Metti R in posizione N nel numero in base B prodotto dall'algoritmo
   c) Incrementa N di 1
   d) Assegna a Q il quoziente di Q diviso B
   e) Torna al passo 3

# Algoritmo di conversione




| Esponente (N) | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| Base (B) | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |

# Esempio - Metodo «pratico»

Conversione in base due del numero 13 in base dieci

```
13 : 2 = 6 resto 1

 6 : 2 = 3 resto 0

 3 : 2 = 1 resto 1

 1 : 2 = 0 resto 1 -> STOP  (Poiché il quoziente è 0)
```

La conversione binaria di 13 possiamo ottenerla scrivendo da sinistra a destra i resti presi dal basso verso l'alto

$$(13)_{10} = (1101)_2$$

# Esempio - Metodo «pratico» - Prova

Come prova convertiamo 1101 da base due a base dieci. La posizione di ogni bit da destra verso sinistra ci informa dell'esponente da applicare alla base 2, a partire da zero.

$$(1101)_2 = (1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0)_{10} =$$
$$= (8 + 4 + 0 + 1)_{10} = (13)_{10}$$

$$\mathbf{(1101)_2 = (13)_{10}}$$

# Aritmetica modulare – Modulo $2^N$

I numeri naturali sono infiniti, ma la memoria di un elaboratore digitale è finita!

Posto che riserviamo *N* bit per rappresentare i numeri naturali in un elaboratore digitale, possiamo rappresentare qualunque numero naturale *M* con la convenzione «modulo $2^N$»

$$|M|_{2^N}$$

Se $M_1 = k2^N + M$, con $k$ numero intero, allora $|M_1|_{2^N} = |M|_{2^N}$

In altre parole, se due numeri differiscono di un multiplo di $2^N$, allora hanno la stessa rappresentazione.

# Modulo $2^N$ – Esempio

Supponiamo di lavorare con $N = 3$, cioè con 3 bit. In questo caso $2^N = 2^3 = 8$

| Dec | Bin | Dec | Bin | Dec | Bin |
|---|---|---|---|---|---|
| 0 | 000 | 8 | 000 | 16 | 000 |
| 1 | 001 | 9 | 001 | 17 | 001 |
| 2 | 010 | 10 | 010 | 18 | 010 |
| 3 | 011 | 11 | 011 | 19 | 011 |
| 4 | 100 | 12 | 100 | 20 | 100 |
| 5 | 101 | 13 | 101 | 21 | 101 |
| 6 | 110 | 14 | 110 | 22 | 110 |
| 7 | 111 | 15 | 111 | 23 | 111 |

# Addizione numeri naturali

L'addizione segue le regole dell'aritmetica come in base 10!

Supponiamo di dover sommare due numeri naturali ***X*** e ***Y*** (la loro somma sarà ***S***).

- $x_i$ e $y_i$ rappresentano i bit di ***X*** e ***Y*** in posizione $i$
- $s_i$ rappresenta il bit di ***S*** in posizione $i$
- $r_i$ rappresenta il bit del riporto in posizione $i$

# Addizione numeri naturali

| | N-1 | … | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| ***X*** | 1 | … | 1 | 1 | 1 | 0 | 1 | 0 | + |
| ***Y*** | 0 | … | 1 | 1 | 1 | 0 | 1 | 0 | = |
| ***S*** | ? | … | 1 | 1 | 0 | 1 | 0 | 0 | |

| $x_i$ | $y_i$ | $r_i$ | $s_i$ | $r_{i+1}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

# Sottrazione numeri naturali

Anche la sottrazione seguirebbe le stesse regole della base 10 ma…

Considerando che

- $2^N - 1 - Y$ è il complemento a 1 di Y, cioè il numero risultante dall'inversione dei bit di Y (gli 0 diventano 1 e viceversa)
- $\boldsymbol{D = |X - Y|_{2^N} = |X - Y + 2^N|_{2^N} = |1 + X - Y + 2^N - 1|_{2^N} =}$
  $\boldsymbol{= |1 + X + compl_1(Y)|_{2^N}}$

La sottrazione $X - Y$ può essere implementata come somma di $X$ e del complemento a 1 di $Y$, più 1. Gli stessi circuiti per l'addizione possono essere usati per la sottrazione.

# Numeri interi

# Numeri interi in un elaboratore digitale

Come rappresentare i numeri interi {…, -2, -1, 0, 1, 2, …} in un elaboratore digitale?

La soluzione più semplice sarebbe quella di codificarli esattamente come i numeri naturali, riservando il bit più a sinistra per rappresentare il segno (0 = positivo, 1 = negativo).

Questa codifica è detta modulo e segno e consiste nell'usare un bit per il segno, lasciando gli altri bit a rappresentare il valore assoluto del numero intero…

# Modulo e segno

Tale codifica però include una doppia rappresentazione dello 0 che può essere fonte di ambiguità.

Per questo, la codifica in modulo e segno, che sarebbe esprimibile come

$$codifica(x) = \begin{cases} x & se\ x \geq 0 \\ |x| + 2^{N-1} & se\ x \leq 0 \end{cases}$$

non viene usata. La tabella a lato mostra la codifica modulo e segno con 4 bit ($N = 4$)

| Decimale | Mod e segno |
|---|---|
| 7 | 0111 |
| 6 | 0110 |
| 5 | 0101 |
| 4 | 0100 |
| 3 | 0011 |
| 2 | 0010 |
| 1 | 0001 |
| 0 | 0000 |
| -0 | 1000 |
| -1 | 1001 |
| -2 | 1010 |
| -3 | 1011 |
| -4 | 1100 |
| -5 | 1101 |
| -6 | 1110 |
| -7 | 1111 |
| -8 | - |

# Numeri interi in un elaboratore digitale

Come rappresentare i numeri interi {…, -2, -1, 0, 1, 2, …} in un elaboratore digitale?

Un’altra soluzione sarebbe codificare gli interi positivi come i numeri naturali e i numeri negativi come il complemento a 1 (cioè l’inversione di tutti i bit) dei positivi.

Es: 4 potrebbe essere codificato come 0100, mentre -4 come 1011

# Complemento a 1

Anche tale codifica però include una doppia rappresentazione dello 0 che può essere fonte di ambiguità.

Nemmeno la codifica basata sul complemento è adatta per rappresentare i numeri interi in un elaboratore digitale

| Decimale | Compl. a 1 |
|---|---|
| 7 | 0111 |
| 6 | 0110 |
| 5 | 0101 |
| 4 | 0100 |
| 3 | 0011 |
| 2 | 0010 |
| 1 | 0001 |
| 0 | 0000 |
| -0 | 1111 |
| -1 | 1110 |
| -2 | 1101 |
| -3 | 1100 |
| -4 | 1011 |
| -5 | 1010 |
| -6 | 1001 |
| -7 | 1000 |
| -8 | - |

# Numeri interi in un elaboratore digitale

Come rappresentare i numeri interi {…, -2, -1, 0, 1, 2, …} in un elaboratore digitale?

Una soluzione al problema delle codifiche in modulo e segno e in complemento a 1 è traslare verso l'alto tutti i valori che possono essere inclusi su N bit.

$$codifica(x) = \begin{cases} 2^{N-1} + |x| & se\ x \geq 0 \\ 2^{N-1} - |x| & se\ x \leq 0 \end{cases}$$

In questo modo la codifica non è più ambigua: sia 0 che -0 sono rappresentati come $2^{N-1}$

# Traslazione

Un metodo «intuitivo» è partire a contare dal minimo numero rappresentabile (con N = 4 bit tale numero è -8) e ad ogni incremento si aggiunge 1 (-8 = 0000, -7 = 0001, -6 = 0010, …).

Tuttavia, la somma algebrica di due numeri traslati <u>non è</u> la traslazione della somma

| | |
|---|---|
| 2 + | 1010 + |
| -3 = | 0101 = |
| -1 | 1111 |

<u>1111 non è la codifica in traslazione di -1!</u> Nemmeno questa codifica si usa per gli interi.

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

# Numeri interi in un elaboratore digitale

Come rappresentare i numeri interi {…, -2, -1, 0, 1, 2, …} in un elaboratore digitale?

I numeri interi in un elaboratore digitale sono universalmente rappresentati con la codifica in «complemento a 2», definita come segue

$$codifica(x) = \begin{cases} x & se\ x \geq 0 \\ 2^N - |x| & se\ x \leq 0 \end{cases}$$

In altre parole, su *N* bit, gli interi positivi sono rappresentati esattamente come i naturali, gli interi negativi sono rappresentati come $2^N - |x|$

# Complemento a 2

Si noti che su 4 bit ($N = 4$) nel caso di «-0» la rappresentazione sarebbe $2^N - |x|$ e cioè $(16)_{10} = (10000)_2$.

Tuttavia, non c'è posto per il bit più a sinistra. Dunque -0 continua ad essere 0000 (si ricordi l'aritmetica modulare…).

Inoltre la somma algebrica di due numeri in complemento a 2 restituisce il risultato già correttamente codificato!

| Decimale | Compl. a 2 |
|---|---|
| 7 | 0111 |
| 6 | 0110 |
| 5 | 0101 |
| 4 | 0100 |
| 3 | 0011 |
| 2 | 0010 |
| 1 | 0001 |
| 0 | 0000 |
| -0 | 0000 |
| -1 | 1111 |
| -2 | 1110 |
| -3 | 1101 |
| -4 | 1100 |
| -5 | 1011 |
| -6 | 1010 |
| -7 | 1001 |
| -8 | 1000 |

# Complemento a 2

- Se $R_x$ e $R_y$ sono le rappresentazioni ad *N* bit in complemento a 2 di due numeri interi $x$ ed $y$, allora la rappresentazione della loro somma $s$ è data da

$$R_s = \left| R_x + R_y \right|_{2^N}$$

- Analogamente, la loro differenza $d$ è data da

$$R_d = \left| R_x - R_y \right|_{2^N}$$

- In altre parole, i circuiti che servono a compiere operazioni di addizione e sottrazione fra numeri naturali sono utilizzabili anche per compiere le stesse operazioni fra numeri interi.

Per questo motivo i numeri interi sono universalmente rappresentati nei calcolatori in complemento a 2.

# Complemento a 2

C’è un metodo intuitivo per calcolare il complemento a 2 di un intero negativo. Data la rappresentazione binaria di un numero positivo

- Esamino il numero bit per bit da destra verso sinistra
- Per scrivere il complemento a 2 del numero negativo lascio ogni bit invariato fino al primo 1 compreso
- Inverto tutti i bit che seguono il primo 1 da destra verso sinistra.

# Complemento a 2 – Esempio

Esempio: -2 rappresentato su 4 bit (*N* = 4)

- L'intero positivo $(2)_{10}$ in base due vale $(0010)_2$
- Applicando da destra verso sinistra quanto spiegato nella slide precedente per il «complemento a 2» in un elaboratore digitale si ha che $(-2)_{10} = (1110)_2$. Applicando a 0010 passaggio per passaggio quanto descritto:

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

# Complemento a 2 - Esempio

Esempio: -5 rappresentato su 4 bit (*N* = 4)

- L'intero positivo $(5)_{10}$ in base due vale $(0101)_2$
- Applicando il «complemento a 2» si ha che $(-5)_{10} = (1011)_2$.
  Applicando a 0101 passaggio per passaggio quanto descritto:

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

# Numeri interi - Riepilogo

| Decimale | Mod e segno | Compl. a 1 | Traslazione | Compl. a 2 |
|---|---|---|---|---|
| 7 | 0111 | 0111 | 1111 | 0111 |
| 6 | 0110 | 0110 | 1110 | 0110 |
| 5 | 0101 | 0101 | 1101 | 0101 |
| 4 | 0100 | 0100 | 1100 | 0100 |
| 3 | 0011 | 0011 | 1011 | 0011 |
| 2 | 0010 | 0010 | 1010 | 0010 |
| 1 | 0001 | 0001 | 1001 | 0001 |
| 0 | 0000 | 0000 | 1000 | 0000 |
| -0 | 1000 | 1111 | 1000 | 0000 |
| -1 | 1001 | 1110 | 0111 | 1111 |
| -2 | 1010 | 1101 | 0110 | 1110 |
| -3 | 1011 | 1100 | 0101 | 1101 |
| -4 | 1100 | 1011 | 0100 | 1100 |
| -5 | 1101 | 1010 | 0011 | 1011 |
| -6 | 1110 | 1001 | 0010 | 1010 |
| -7 | 1111 | 1000 | 0001 | 1001 |
| -8 | – | – | 0000 | 1000 |

# Numeri reali

# Numeri reali in un elaboratore digitale

Come rappresentare i numeri reali in un elaboratore digitale?

Fanno parte dei numeri reali:

- I numeri naturali
- I numeri interi
- I numeri razionali (cioè i numeri esprimibili come frazione)
- I numeri irrazionali (es. $\sqrt{2}$, $\pi$, $e$)

# Numeri reali – Virgola mobile

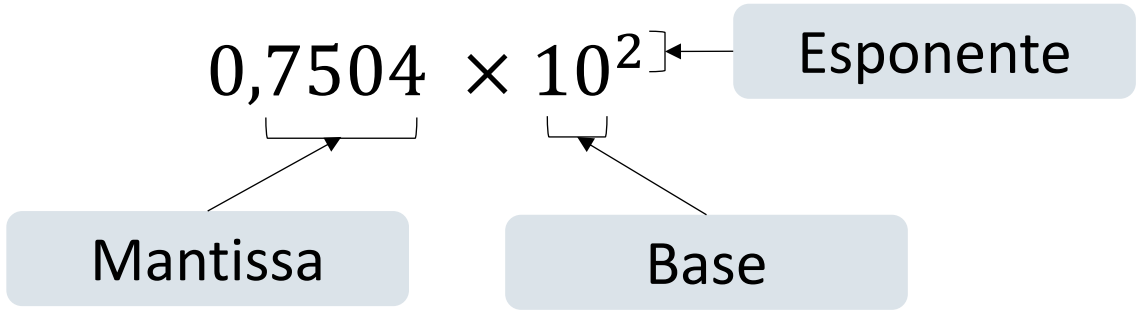
La memoria di un elaboratore digitale è finita, mentre i numeri reali sono infiniti (l'insieme dei reali è anche non numerabile…). Serve un compromesso tra il *range* di valori rappresentabili e la *precisione.*

Negli elaboratori digitali si usa la rappresentazione in virgola mobile, stabilendo un numero fisso di cifre significative e scalando l'esponente della base di conseguenza.

Ad esempio, tutte le seguenti notazioni esprimono la stessa quantità

$$75{,}04 = 75{,}04 \times 10^0 = 7{,}504 \times 10^1 = 0{,}7504 \times 10^2 = 7504 \times 10^{-2}$$

# Numeri reali – Virgola mobile




La mantissa è la parte frazionaria di un numero reale. Dato un reale $x$, la mantissa è la differenza tra $x$ e la sua parte intera.

In un elaboratore digitale i numeri reali sono rappresentati in virgola mobile (*floating point*). Ovviamente, la base è 2 anziché 10.

# Numeri reali – Conversione in base 2

Convertiamo separatamente la parte intera e la parte decimale.
Esempio: 6,625.

Per la parte intera:

```
6 : 2 = 3 resto 0

3 : 2 = 1 resto 1

1 : 2 = 0 resto 1 -> STOP  (Poiché il quoziente è 0)
```

$$(6)_{10} = (110)_2$$

# Numeri reali – Conversione in base 2

Per la parte frazionaria, anziché dividere per 2 si moltiplica per 2

`0,625 × 2 = 1,250` -> 1 sarà la prima cifra dopo la virgola. Si prosegue con la parte frazionaria 0,250

`0,250 × 2 = 0,5` -> 0 sarà la seconda cifra dopo la virgola. Si prosegue con la parte frazionaria 0,5

`0,5 × 2 = 1,0` -> **STOP** 1 sarà la terza cifra dopo la virgola. Ci si ferma, in quanto la parte frazionaria restante è 0.

$$(0{,}625)_{10} = (0{,}101)_2$$

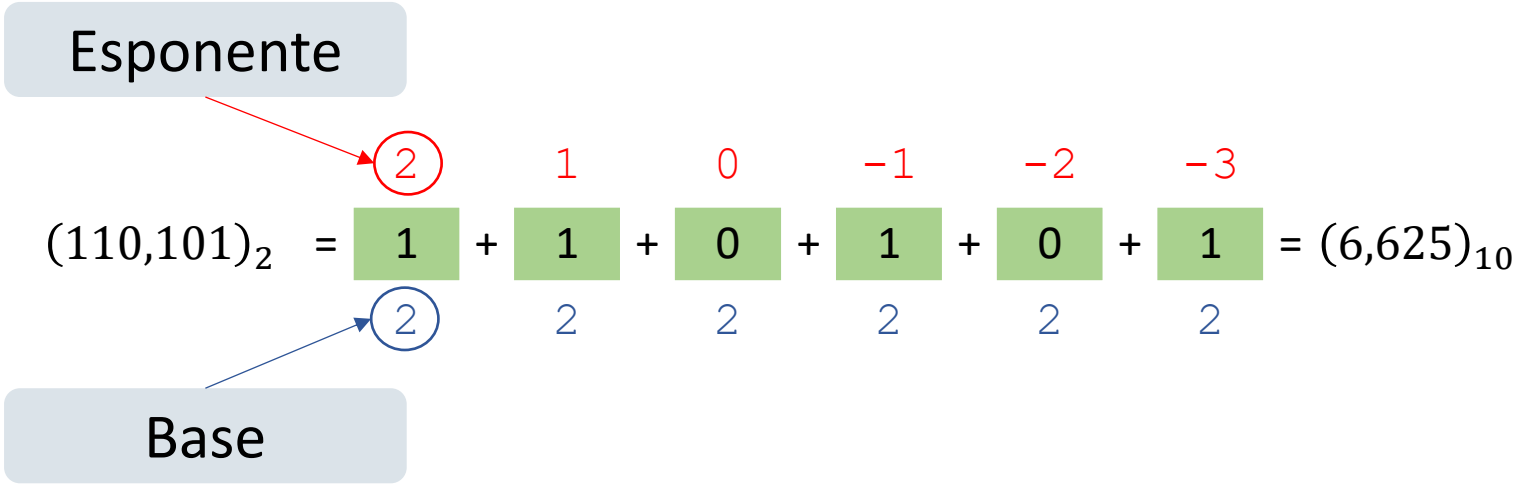
Le cifre dopo la virgola esprimono le potenze negative.

# Numeri reali – Conversione in base 2

Le posizioni dopo la virgola rappresentano gli esponenti negativi della base, che decrementano di 1 da sinistra verso destra, a partire da -1.

$$0{,}101 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = \frac{1}{2} + 0 \times \frac{1}{4} + \frac{1}{8} =$$

$$= 0{,}5 + 0 + 0{,}125 = 0{,}625$$



# Numeri reali – Conversione in base 2

Non tutte le parti frazionarie finite in base 10 sono finite anche in base 2… Ad esempio, proviamo a convertire 0,6 e 0,85

| | | |
|---|---|---|
| 0,6 | × 2 = | 1,2 |
| 0,2 | × 2 = | 0,4 |
| 0,4 | × 2 = | 0,8 |
| 0,8 | × 2 = | 1,6 |
| 0,6 | × 2 = | 1,2 |
| 0,2 | × 2 = | 0,4 |
| 0,4 | × 2 = | 0,8 |
| 0,8 | × 2 = | 1,6 |
| 0,6 | × 2 = | 1,2 |
| … | … | … |

$$(0{,}6)_{10} = (0, \overline{1001})_2$$

$$(0{,}85)_{10} = (0{,}11\overline{0110})_2$$

In base 2 diventano periodici!

| | | |
|---|---|---|
| 0,85 | × 2 = | 1,7 |
| 0,7 | × 2 = | 1,4 |
| 0,4 | × 2 = | 0,8 |
| 0,8 | × 2 = | 1,6 |
| 0,6 | × 2 = | 1,2 |
| 0,2 | × 2 = | 0,4 |
| 0,4 | × 2 = | 0,8 |
| 0,8 | × 2 = | 1,6 |
| 0,6 | × 2 = | 1,2 |
| … | … | … |

# Numeri reali – Virgola mobile

Come i numeri in base dieci, anche quelli in base due possono essere espressi in virgola mobile

$$110{,}101 = 110{,}101 \times due^{0} = 11{,}0101 \times due^{1} = 1{,}10101 \times due^{2} = 11010{,}1 \times due^{-2}$$

# Standard **IEEE 754**

Standard per l'aritmetica in virgola mobile definito dall'*Institute of Electrical and Electronics Engineers* (IEEE) nel 1985.

Stabilisce regole per la rappresentazione floating point, le operazioni eseguibili, gli arrotondamenti e la gestione delle eccezioni

- Numeri reali in virgola mobile
- Numero di bit per segno, esponente, mantissa
- Precisione singola (32 bit), doppia (64 bit), quadrupla (128 bit), precisione estesa e estendibile

# IEEE 754 a precisione singola

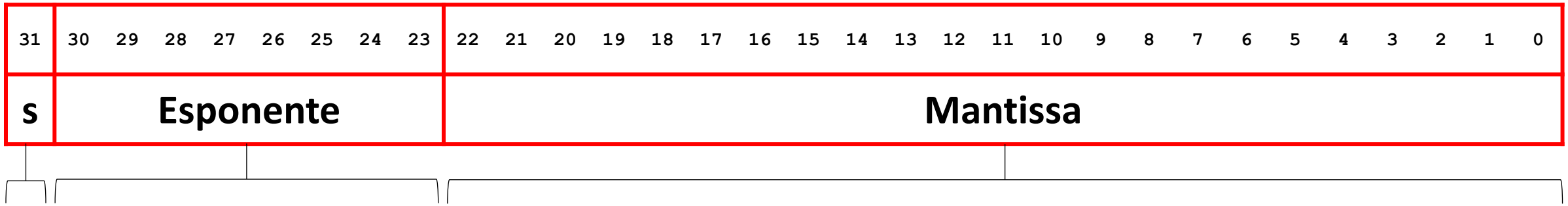
- 32 bit per rappresentare i numeri reali
- 1 bit, quello più a sinistra per il segno (0 positivo, 1 negativo)
- I successivi 8 bit contengono l'esponente, rappresentato come *intero in traslazione* di 127 (cioè viene memorizzato il valore dell'esponente + 127)
- I restanti 23 bit contengono la mantissa

# IEEE 754 a precisione singola

Per calcolare la rappresentazione binaria di un numero in IEEE 754 a precisione singola:

- Si convertono parte intera e parte frazionaria in base due
- Si sposta la virgola lasciando un solo 1 a sinistra della stessa, cambiando l'esponente in maniera coerente
- Si trasla l'esponente di 127 e se ne calcola la rappresentazione binaria
- Si riporta nel bit più a sinistra il segno (0 positivo, 1 negativo)
- Nei successivi 8 bit si scrive l'esponente traslato
- Nei successivi 23 bit si scrive la mantissa, sottintendendo l'1 a sinistra della virgola

# IEEE 754 a precisione singola




Segno
0 (+)
1 (-)

Vero esponente
+
127
(traslazione)

Numero naturale che corrisponde alla componente frazionaria del numero in virgola fissa con una sola cifra a sinistra del punto decimale

$$23{,}85 = 10111{,}11\overline{0110} = 1{,}0111110110 \times due^{4}$$

L'esponente viene traslato di 127, dunque sarà rappresentato come $(131)_{10} = (10000011)_2$

Viene sottinteso

**01000001101111101100110011001101**

# IEEE 754 a precisione singola – Esempio

Calcoliamo la rappresentazione IEEE 754 a precisione singola di 12,75. Cominciamo dalla rappresentazione binaria di 12 e di 0,75, cioè della parte intera e di quella frazionaria.

```
12 : 2 = 6 resto 0

 6 : 2 = 3 resto 0

 3 : 2 = 1 resto 1

 1 : 2 = 0 resto 1 -> 
```

$(12)_{10} = (1100)_2$

# IEEE 754 a precisione singola – Esempio

0,75 × 2 = 1,5

0,5 × 2 = 1,0 -> $(0{,}75)_{10} = (0{,}11)_2$

A questo punto abbiamo che

$$(12{,}75)_{10} = (1100{,}11)_2$$

Dobbiamo spostare la virgola verso sinistra fino al primo 1, cioè di tre posizioni

$$1100{,}11 = 1{,}10011 \times due^{3}$$

Viene sottinteso

# IEEE 754 a precisione singola – Esempio

Dobbiamo ora traslare l'esponente di 127:

```
3 + 127 = 130
```

Non ci resta che convertirlo in base due

```
130 : 2 = 65 resto 0        8 : 2 = 4 resto 0
 65 : 2 = 32 resto 1        4 : 2 = 2 resto 0
 32 : 2 = 16 resto 0        2 : 2 = 1 resto 0
 16 : 2 =  8 resto 0        1 : 2 = 0 resto 1
```

$$(130)_{10} = (10000010)_2$$

# IEEE 754 a precisione singola – Esempio

Considerando che 12,75 è positivo, il bit del segno è 0.

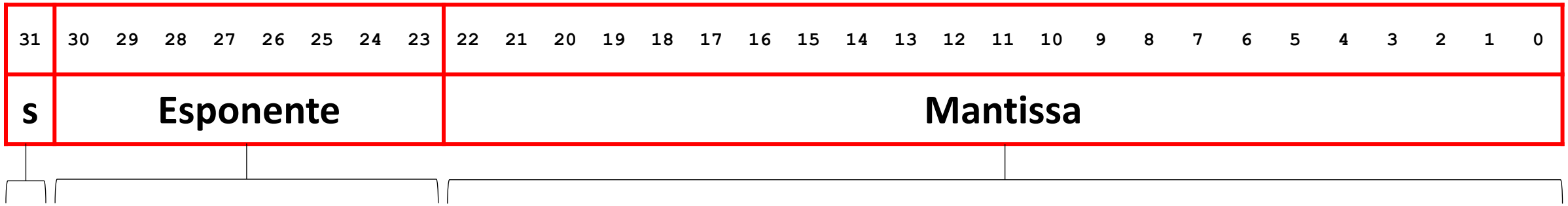
Dunque la rappresentazione in IEEE 754 a precisione singola di 12,75 è

| s | Esponente | Mantissa |
|---|---|---|
| 0 | 1 0 0 0 0 0 1 0 | 1 0 0 1 1 |

I bit restanti sono 0 (Gli zero a destra dell'ultima cifra significativa non cambiano il valore. Es: 1,575 = 1,5750000).

| | | |
|---|---|---|
| 0 | 1 0 0 0 0 0 1 0 | 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |

# IEEE 754 a precisione singola




Segno
0 (+)
1 (-)

Vero esponente
+
127
(traslazione)

Numero naturale che corrisponde alla componente frazionaria del numero in virgola fissa con una sola cifra a sinistra del punto decimale

$$23{,}85 = 10111{,}11\overline{0110} = 1{,}0111110110 \times due^{4}$$

L'esponente viene traslato di 127, dunque sarà rappresentato come $(131)_{10} = (10000011)_2$

Viene sottinteso

**01000001101111101100110011001101**

Arrotondamento

# IEEE 754 – Arrotondamento

| Mode | Example value | | | |
|---|---|---|---|---|
| | **+11.5** | **+12.5** | **−11.5** | **−12.5** |
| to nearest, ties to even | +12.0 | +12.0 | −12.0 | −12.0 |
| to nearest, ties away from zero | +12.0 | +13.0 | −12.0 | −13.0 |
| toward 0 | +11.0 | +12.0 | −11.0 | −12.0 |
| toward +∞ | +12.0 | +13.0 | −11.0 | −12.0 |
| toward −∞ | +11.0 | +12.0 | −12.0 | −13.0 |

- *To nearest, ties to even* → arrotondamento convergente (default per binari)
- *To nearest, ties away from zero* → arrotondamento commerciale
- *Toward 0* → troncamento
- *Toward +∞* → arrotondamento per eccesso
- *Toward -∞* → arrotondamento per difetto

# IEEE 754 – Arrotondamento

***To nearest, ties to even***

Troncamento all'i-esima cifra (base 10)

- Se la cifra in posizione *i+1* è **≤ 4**, arrotonda per difetto (in valore assoluto)
- Se la cifra in posizione *i+1* è **≥ 6**, arrotonda per eccesso (in valore assoluto)
- Se la cifra in posizione *i+1* è **= 5** e ci sono altre **cifre diverse da 0 nelle posizioni successive**, arrotonda per eccesso (in valore assoluto)
- Se la cifra in posizione *i+1* è **= 5** e le cifre **successive sono 0**
  - Se la cifra in posizione *i* **è pari**, arrotonda per difetto (in valore assoluto)
  - Se la cifra in posizione *i* **è dispari**, arrotonda per eccesso (in valore assoluto)

# IEEE 754 – Arrotondamento

***To nearest, ties to even***

Troncamento all'i-esima cifra (base 2)

- Se la cifra in posizione *i+1* è **0**, arrotonda per difetto (in valore assoluto)
- Se la cifra in posizione *i+1* è **1** e **ci sono altri 1 nelle cifre successive** arrotonda per eccesso (in valore assoluto)
- Se la cifra in posizione *i+1* è **1** e le **cifre successive sono tutte 0**
  - Se la cifra in posizione *i* è **0**, arrotonda per difetto (in valore assoluto)
  - Se la cifra in posizione *i* è **1**, arrotonda per eccesso (in valore assoluto)

# IEEE 754 – Arrotondamento

***To nearest, ties to even***

| **Esempi base 10** | **Esempi base 2** |
|---|---|
| 6,3421 → 6,34 | 0,1101 → 0,11 |
| 8,2764 → 8,28 | 0,1011 → 0,11 |
| 4,3254 → 4,33 | 0,1111 → 1,00 |
| 9,1250 → 9,12 | 0,1010 → 0,10 |
| 9,1350 → 9,14 | 0,1110 → 1,00 |

# IEEE 754 a precisione doppia

- 64 bit per rappresentare i numeri reali
- 1 bit, quello più a sinistra per il segno (0 positivo, 1 negativo)
- I successivi 11 bit contengono l'esponente, rappresentato come *intero in traslazione* di 1023 (cioè viene memorizzato il valore dell'esponente + 1023)
- I restanti 52 bit contengono la mantissa

# IEEE 754 a precisione doppia


Segno
0 (+)
1 (-)

Vero esponente
+
1023

Numero naturale che corrisponde alla componente frazionaria del numero in virgola fissa con una sola cifra a sinistra del punto decimale

| | **Single precision** | **Double precision** |
|---|---|---|
| **Decimali significativi (in base 10)** | 7 | 15 |
| **Range di ampiezza (in base 10)** | Da $1{,}18 \times 10^{-38}$ a $3{,}4 \times 10^{38}$ (in valore assoluto) | Da $2{,}23 \times 10^{-308}$ a $1{,}8 \times 10^{308}$ (in valore assoluto) |
| **Esponente (Min)** | -126 | -1022 |
| **Esponente (Max)** | 127 | 1023 |

# Standard IEEE 754

Lo zero non sarebbe rappresentato… ma è un numero troppo importante!

Si effettua una denormalizzazione dello standard

# Standard IEEE 754 - Denormalizzazione

- 32 bit (a precisione singola) e 64 bit (a precisione doppia) tutti a 0 rappresentano lo zero anziché $1{,}0 \times due^{-126}$
- I numeri in IEEE 754 in cui la mantissa è composta da tutti 0 e l'esponente da tutti 1 rappresentano $+\infty$ (segno 0) e $-\infty$ (segno 1)
- I numeri in IEEE 754 in cui la mantissa contiene cifre diverse da 0 e l'esponente è composto da tutti 1, rappresentano «Not a number» (NaN)
- I numeri in IEEE 754 in cui la mantissa contiene cifre diverse da 0 e l'esponente è composto da tutti 0 sono «denormalizzati», cioè non viene sottinteso «1,», ma «0,»

# Standard IEEE 754 - Denormalizzazione

| | |
|---|---|
| 00000000000000000000000000000000 | rappresenta lo 0 |
| 01111111000000000000000000000000 | rappresenta $+\infty$ |
| 11111111000000000000000000000000 | rappresenta $-\infty$ |
| 01111111--- non tutti zero ---- | NaN (Not a number) |
| 11111111--- non tutti zero ---- | NaN (Not a number) |
| 00000000--- non tutti zero ---- | denormalizzato (pos) |
| 10000000--- non tutti zero ---- | denormalizzato (neg) |

Esempio di numero denormalizzato:

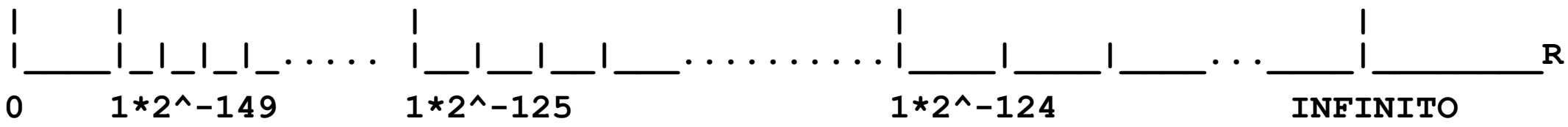
$$\mathbf{0}\mathbf{00000000}\mathbf{00000000000000000000001} =$$
$$= 0.00000000000000000000001 \times 2^{-126} = \; 2^{-23} \times 2^{-126} = \mathbf{2^{-149}}$$

# Standard IEEE 754

Con lo standard denormalizzato il minimo numero rappresentabile (in valore assoluto) oltre allo 0 è $2^{-149}$. Il successivo sarebbe

$$\mathbf{0}\mathbf{00000000}\mathbf{00000000000000000000010} =$$
$$= 0.00000000000000000010 \times 2^{-126} = \ 2^{-22} \times 2^{-126} = 2^{-148}$$

Tra di essi ci sono infiniti valori non rappresentabili (come, d'altro canto, tra 0 e $2^{-149}$). L'intervallo di valori non rappresentabile cresce al crescere della potenza di 2 considerata



# Standard IEEE 754

L’aritmetica in virgola mobile è sempre un’approssimazione