Architettura degli Elaboratori e Laboratorio

Matteo Manzali

Università degli Studi di Ferrara

Anno Accademico 2016 - 2017

Numeri razionali

- Sono numeri esprimibili come rapporto di due numeri interi.
- L'insieme dei numeri razionali contiene, oltre ai numeri interi, i numeri decimali (numeri con virgola).
- In un sistema posizionale con base B, *n* cifre intere ed *m* cifre frazionarie, i numeri razionali sono così rappresentati:

$$b_{n-1} \cdot B^{n-1} + \ldots + b_1 \cdot B^1 + b_0 \cdot B^0 + b_{-1} \cdot B^{-1} + b_{-2} \cdot B^{-2} + \ldots + b_{-m} \cdot B^{-m}$$

- La notazione con *n* + *m* cifre è detta a **virgola fissa** (fixed point).

Conversione in virgola fissa

- Per convertire un numero razionale da base 10 a base 2 si procede in due passi:
  - Si converte la parte intera (sappiamo già farlo)
  - Si converte la parte decimale
- Per la parte decimale:
  - si moltiplica per 2 la parte decimale
  - si prende la parte intera del risultato come cifra utile
  - si ripetono i due passi precedenti fino a quando la parte decimale diventa nulla

Esempio 1

- Es.: $(19.375)_{dieci}$ convertire in base 2 in virgola fissa.

  sappiamo che $(19)_{dieci} = (10011)_{due}$

  prendiamo la parte decimale:

  0.375 • 2 = **0**.75
  0.75 • 2 = **1**.5
  0.5 • 2 = **1**

  N.B. leggo i valori dall'alto verso il basso (al contrario di come fatto nella conversione della parte intera).

  Ottengo quindi che $(19.375)_{dieci} = (10011.011)_{due}$

Esempio 2

- Es.: $(2.8125)_{dieci}$ convertire in base 2 in virgola fissa.

  Provate a farlo in maniera autonoma.

Esempio 2

- Es.: $(2.8125)_{dieci}$ convertire in base 2 in virgola fissa.

  sappiamo che $(2)_{dieci} = (10)_{due}$

  prendiamo la parte decimale:

  $0.8125 \cdot 2 = \mathbf{1}.625$
  $0.625 \cdot 2 = \mathbf{1}.25$
  $0.25 \cdot 2 = \mathbf{0}.5$
  $0.5 \cdot 2 = \mathbf{1}$

  Ottengo quindi che $(2.8125)_{dieci} = (10.1101)_{due}$

Virgola fissa: problemi

- La notazione in virgola fissa non permette di rappresentare numeri molto grandi o molto piccoli:

  Es. con n=5 e m=3 posso rappresentare i numeri compresi tra -99999.999 e 99999.999

- Nel caso precedente non mi è permesso rappresentare ad esempio:

  0.000001 perchè ho solo 3 cifre dopo la virgola

  oppure

  1000000 perchè ho solo 5 cifre per la parte intera

Numeri razionali (virgola mobile)

- Per ovviare a questo problema (ottimizzando le risorse a disposizione) si è inventata la cosiddetta notazione in **virgola mobile** (floating point).

- Si utilizza la seguente notazione:

$$N = \pm\, m \cdot b^{e}$$

- b → base del sistema di numerazione

- m → mantissa del numero

- e → esponente (intero con segno)

Numeri razionali (virgola mobile)

- Fissata la base, per rappresentare un numero reale è necessario rappresentare segno, mantissa ed esponente.
- La mantissa si suppone in virgola fissa con una sola cifra non nulla a sinistra dell virgola (**notazione scientifica normalizzata**).
- L'intervallo di valori della mantissa è quindi: **1 <= m < b**
- Es.:

$(19.375)_{dieci} = (1.9375 \cdot 10^{1})_{dieci}$ // esponente positivo
$(0.0015)_{dieci} = (1.5 \cdot 10^{-3})_{dieci}$ // esponente negativo
$(7.8)_{dieci} = (7.8 \cdot 10^{0})_{dieci}$ // esponente nullo
$(-321.1)_{dieci} = (-3.211 \cdot 10^{2})_{dieci}$ // segno negativo

$(10011.001)_{due} = (1.0011001 \cdot \underbrace{10}_{2}{}^{\underbrace{100}_{4}})_{due}$ // b = 2

Numeri razionali (virgola mobile)

- Osservazioni:
  - la mantissa determina la precisione
  - l'esponente determina la gamma dei valori
  - moltiplicare un numero per una potenza della base equivale a far scorrere il numero di un numero di posizioni pari all'esponente (destra o sinistra, dipende dal segno)

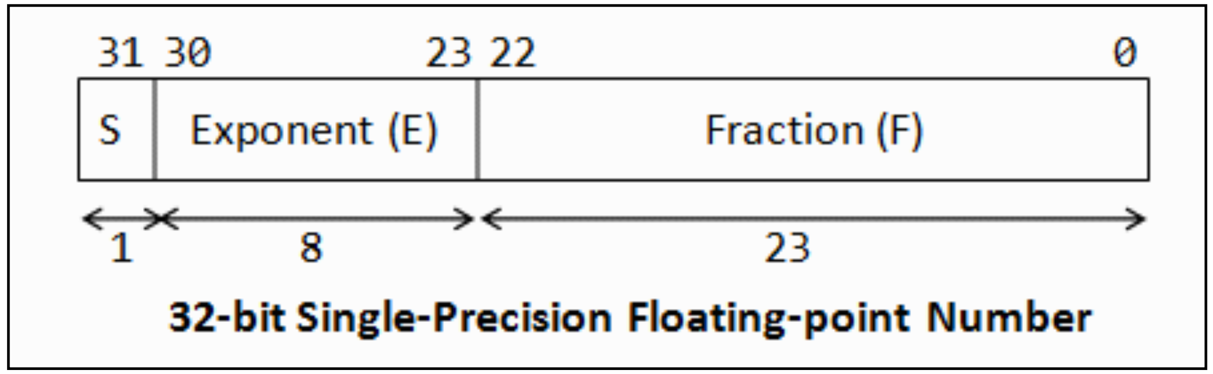
    $(1.9375 \cdot 10^{2})_{dieci} = (193.75)_{dieci}$

  - si possono rappresentare numeri con ordini di grandezza molto differenti utilizzando per la rappresentazione un insieme limitato di cifre

Standard IEEE-754

- E' lo standard internazionale adottato per la rappresentazione dei numeri reali in binario:
  - fino all'introduzione dello standard IEEE (nel 1985) ogni produttore di calcolatori aveva un proprio formato
- Lo standard IEEE-754 adotta la notazione scientifica.
- Esistono diversi formati definiti dallo standard, noi vedremo i due più usati:
  - **singola precisione (SP)** → numeri a 32 bit
    - es.: tipo di dato float
  - **doppia precisione (DP)** → numeri a 64 bit
    - es.: tipo di dato double

IEEE-754 singola precisione



- 32 bit di rappresentazione:
  - 1 bit di segno (*sign* in inglese)
  - 8 bit di esponente (*exponent* in inglese)
  - 23 bit di mantissa (*fraction* in inglese)

IEEE-754 singola precisione

- Il valore del numero è calcolabile con la formula:

$$\mathbf{v = (-1)^{S} \cdot (1 + 0.M) \cdot 2^{(E - bias)}}$$

- **S** specifica il segno del numero:
    - S = 0 → numero positivo
    - S = 1 →numero negativo
    - come per i numeri in complemento a 2

IEEE-754 singola precisione

- Il valore del numero è calcolabile con la formula:

$$v = (-1)^{S} \cdot (1 + 0.M) \cdot 2^{(E - bias)}$$

- **M** rappresenta la mantissa:
  - nella notazione scientifica normalizzata abbiamo $1 <= M < base$
  - essendo un numero binario la parte intera sarà sempre 1, si può quindi omettere assieme alla virgola
  - nella notazione binaria ci sarà quindi solo la parte decimale della mantissa (23 bit)

IEEE-754 singola precisione

- Il valore del numero è calcolabile con la formula:

$$\mathbf{v = (-1)^{S} \cdot (1 + 0.M) \cdot 2^{(E - bias)}}$$

- **E** rappresenta l'esponente:
    - l'esponente nella notazione scientifica può essere negativo
    - per risparmiare il bit del segno l'esponente è rappresentato in forma *biased* con bias = 127
    - i valori di E vanno da 0 a 255 (8 bit → 256 valori)
    - 0 e 255 sono riservati per funzioni speciali (lo vedremo poi)
    - I valori dell'esponente senza il bias vanno quindi da:
      -126 (1-bias) a 127 (254-bias)

IEEE-754 singola precisione

- I valori assunti dall'esponente E e dalla mantissa M determinano l'appartenenza del numero ad una di queste categorie:

caso standard

Categoria	Esponente	Mantissa
Zeri	0	0
Numeri denormalizzati	0	diverso da 0
Numeri normalizzati	da 1 a 254	qualunque
Infiniti	255	0
NaN (not a number)	255	diverso da 0

IEEE-754 singola precisione

- Casi particolari:
  - nello standard IEEE-754 sono presenti due tipologie di zeri e di infiniti, a seconda del segno: $+\infty$, $-\infty$, $+0$, -0
  - NaN (not a number) è il risultato di un’operazione non valida, ad esempio la divisione tra zero e zero
  - la categoria dei numeri denormalizzati esiste per poter rappresentare più piccoli di quelli rappresentabili nella forma standard, in questa configurazione particolare il valore del numero è calcolato come:

$$\mathbf{v = (-1)^S \cdot (0.M) \cdot 2^{-126}}$$

    - come si può notare la mantissa non ha più la parte intera e l’esponente è fissato a -126

Calcolare il bias

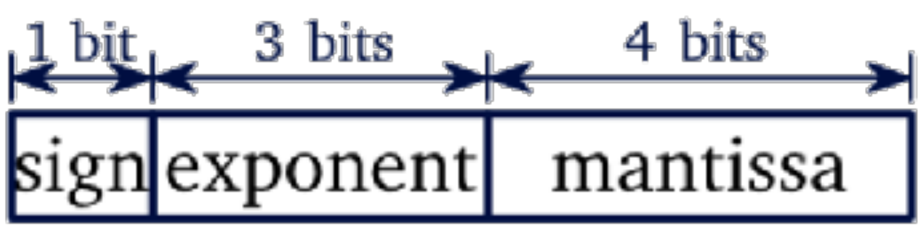
- Il bias si può calcolare in maniera generica a partire dal numero di bit riservati al campo dell'esponente, basta mettere a 1 tutti i bit eccetto il più significativo.

- Es.:

    - rappresentazione floating point 32-bit, 8 bit di esponente:

      bias = $(01111111)_{due} \rightarrow (2^8 - 1)_{dieci} = (127)_{dieci}$

    - rappresentazione floating point in 8-bit:



      bias = $(011)_{due} \rightarrow (2^3 - 1)_{dieci} = (3)_{dieci}$

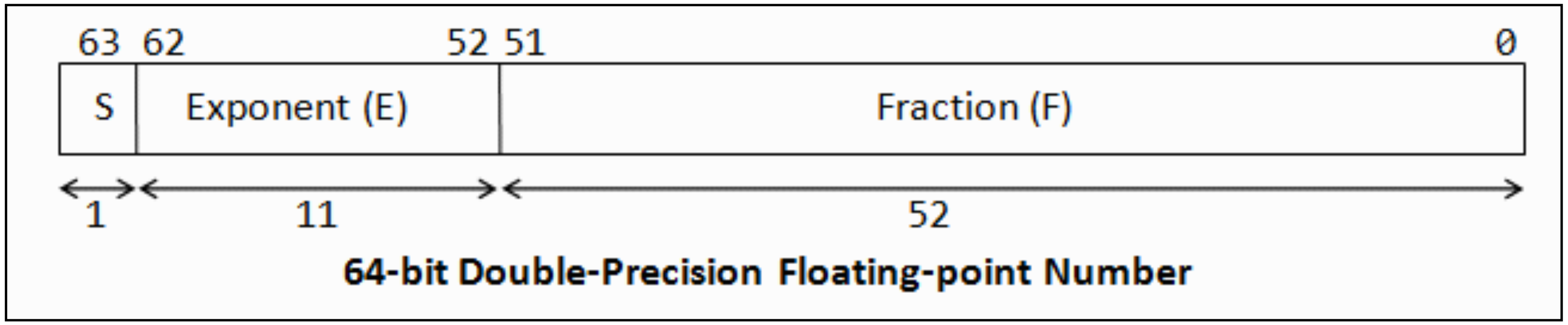
Confronto tra floating points

- L'ordine dei campi *segno*, *esponente*, *mantissa* esiste per un motivo preciso:
    - rende molto semplice il confronto tra due floating points
    - i campi sono in ordine di "importanza"
    - si confrontano i due numeri bit a bit (eccezione fatta per il segno, che segue una logica inversa)
- Es. il primo numero è più piccolo del secondo:

```
0.01234 = 0  01111000  10010100010110110110110
0.32870 = 0  01111101  01010000100101101011110
```

primo bit diverso

IEEE-754 doppia precisione



- 64 bit di rappresentazione:
  - 1 bit di segno (*sign* in inglese)
  - 11 bit di esponente (*exponent* in inglese)
  - 52 bit di mantissa (*fraction* in inglese)

IEEE-754 doppia precisione

- La doppia precisione è molto simile alla singola precisione.
- Le differenze, oltre alla quantità di bit disponibili, riguardano soprattutto l'esponente:
  - per i numeri normalizzati il bias è pari a 1023
  - per i numeri denormalizzati l'esponente è fissato a −1022
  - i NaN e gli infiniti sono rappresentati con un esponente pari a 2047

Intervallo singola precisione

- Gli esponenti 00000000 e 11111111 sono riservati
- Valore più piccolo (in valore assoluto)
  - esponente: 00000001 → 1 - bias = 1 - 127 = -126
  - mantissa: 000...00 → $(1.0)_{due}$
  - $\pm(1.0)_{due} \cdot 2^{-126} \approx \pm 1.2 \cdot 10^{-38}$
  - sotto è underflow (denormalizzazione a parte)
- Valore più grande (in valore assoluto)
  - esponente: 11111110 → 254 - bias = 254 - 127 = 127
  - mantissa: 111...11 → ≈ $(10.0)_{due}$
  - $\pm(10.0)_{due} \cdot 2^{127} \approx \pm 3.4 \cdot 10^{38}$
  - sopra è overflow

Intervallo doppia precisione

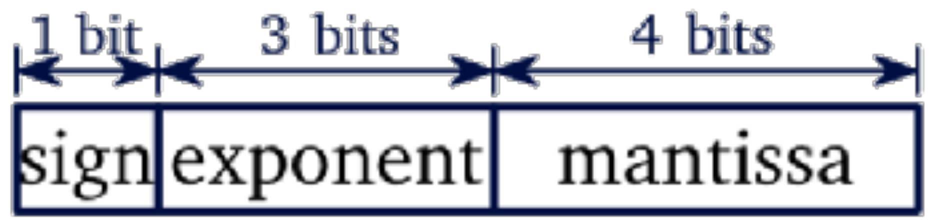
- Gli esponenti 00000000000 e 11111111111 sono riservati
- Valore più piccolo (in valore assoluto)
  - esponente: 00000000001 → 1 - bias = 1 - 1023 = -1022
  - mantissa: 000...00 → $(1.0)_{due}$
  - $\pm(1.0)_{due} \cdot 2^{-1022} \approx \pm 2.2 \cdot 10^{-308}$
  - sotto è underflow (denormalizzazione a parte)
- Valore più grande (in valore assoluto)
  - esponente: 11111111110 → 2046 - bias = 2046 - 1023 = 1023
  - mantissa: 111...11 → $\approx (10.0)_{due}$
  - $\pm(10.0)_{due} \cdot 2^{1023} \approx \pm 1.8 \cdot 10^{308}$
  - sopra è overflow

Conversione in binario

- La procedura standard per la conversione da numero decimale a numero binario IEEE-754 è la seguente:
  - si setta il bit del segno a seconda del segno del numero
  - il modulo del numero va convertito in binario
  - il numero in binario va diviso (o moltiplicato) per 2 fino ad ottenere una forma del tipo 1.xxxx
  - rimuovendo la parte intera e la virgola si ha la mantissa
  - Al numero di volte per cui si è diviso (o moltiplicato) per due bisogna sommare il bias e convertire il risultato in binario, ottenendo l'esponente

Esempio 1 (8-bit)

- Convertire $(2.625)_{dieci}$ in binario con la rappresentazione 8-bit:



parte intera: $(2)_{dieci} \rightarrow (10)_{due}$

parte decimale:

$0.625 \cdot 2 = \mathbf{1}.25$

$0.25 \cdot 2 = \mathbf{0}.5$

$0.5 \cdot 2 = \mathbf{1}$

otteniamo quindi il numero binario: $(10.101)_{due}$

Esempio 1 (8-bit)

Normalizziamo il numero: $(10.101)_{due} \rightarrow (1.0101)_{due} \cdot 2^1$

mantissa: 0101

esponente: $1 + bias = 1 + 3 = (4)_{dieci} \rightarrow (100)_{due}$

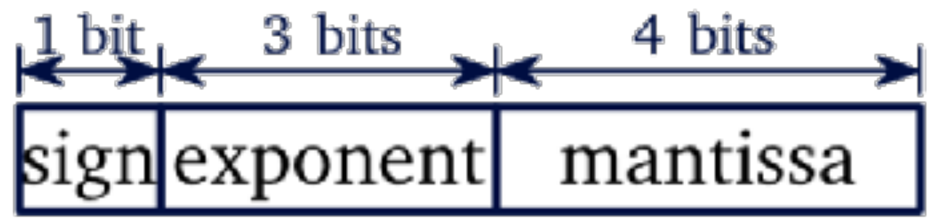
il bit di segno è a 0 (numero positivo)

risultato:

0 100 0101

Esempio 2 (8-bit)

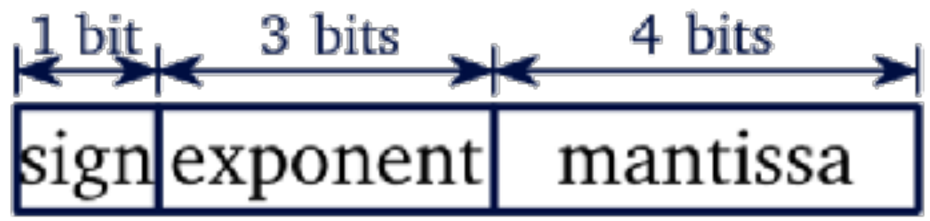
- Convertire $(-4.75)_{dieci}$ in binario con la rappresentazione 8-bit:



Provate a farlo in maniera autonoma.

Esempio 2 (8-bit)

- Convertire $(-4.75)_{dieci}$ in binario con la rappresentazione 8-bit:



parte intera: $(4)_{dieci} \rightarrow (100)_{due}$

parte decimale:
0.75 • 2 = **1**.5
0.5 • 2 = **1**

otteniamo quindi il numero binario: $(100.11)_{due}$

Esempio 2 (8-bit)

Normalizziamo il numero: $(100.11)_{due} \rightarrow (1.0011)_{due} \cdot 2^2$

mantissa: 0011

esponente: 2 + bias = 2 + 3 = $(5)_{dieci} \rightarrow (101)_{due}$

il bit di segno è a 1 (numero negativo)

risultato:

1 101 0011

Esempio 3 (8-bit)

- Convertire $(0.40625)_{dieci}$ in binario con la rappresentazione 8-bit:

  Provate a farlo in maniera autonoma.

Esempio 3 (8-bit)

- Convertire $(0.40625)_{dieci}$ in binario con la rappresentazione 8-bit:

  parte intera: $(0)_{dieci} \rightarrow (0)_{due}$

  parte decimale:
  0.40625 • 2 = **0**.8125
  0.8125 • 2 = **1**.625
  0.625 • 2 = **1**.25
  0.25 • 2 = **0**.5
  0.5 • 2 = **1**

  otteniamo quindi il numero binario: $(0.01101)_{due}$

Esempio 3 (8-bit)

Normalizziamo il numero: $(0.01101)_{due} \rightarrow (1.101)_{due} \cdot 2^{-2}$

mantissa: 101

esponente: $-2 + bias = -2 + 3 = (1)_{dieci} \rightarrow (1)_{due}$

il bit di segno è a 0 (numero positivo)

risultato:

0 001 1010

se la mantissa contiene meno bit di quelli disponibili, si aggiungono gli zero alla fine!

Esempio 4 (32-bit)

- Convertire $(-1313.3125)_{dieci}$ in binario con la rappresentazione in singola precisione.

- Parte intera:

1313 / 2 = 656	R 1
656 / 2 = 328	R 0
328 / 2 = 164	R 0
164 / 2 = 82	R 0
82 / 2 = 41	R 0
41 / 2 = 20	R 1
20 / 2 = 10	R 0
10 / 2 = 5	R 0
5 / 2 = 2	R 1
2 / 2 = 1	R 0
1 / 2 = 0	R 1

Esempio 4 (32-bit)

- Parte decimale:

  $0.3125 \cdot 2 = \mathbf{0}.625$
  $0.625 \cdot 2 = \mathbf{1}.25$
  $0.25 \cdot 2 = \mathbf{0}.5$
  $0.5 \cdot 2 = \mathbf{1}$

- Numero binario: $(10100100001.0101)_{due}$
- Normalizzo: $(1.01001000010101)_{due} \cdot 2^{10}$
- Mantissa: 01001000010101000000000 (ho 14 bit, aggiungo 9 zeri per arrivare a 23 bit)

Esempio 4 (32-bit)

- Esponente: $(10 + 127)_{dieci} = (137)_{dieci}$

  137 / 2 = 68 R 1
  68 / 2 = 34 R 0
  34 / 2 = 17 R 0
  17 / 2 = 8 R 1
  8 / 2 = 4 R 0
  4 / 2 = 2 R 0
  2 / 2 = 1 R 0
  1 / 2 = 0 R 1

- Esponente: $(10001001)_{due}$
- Risultato:

  1 10001001 01001000010101000000000

Esempio 5

- Convertire $(1.7)_{dieci}$ in binario con la rappresentazione 32-bit.
- **Provate a fare questa conversione… Che succede?**

Conversione in decimale

- La procedura standard per la conversione da numero binario IEEE-754 a numero decimale è la seguente:
  - si separano i bit nei tre campi segno, esponente e mantissa
  - si aggiunge “1.” alla mantissa
  - si sottrae il bias all’esponente
  - si scrive il numero in formato decimale settando il segno in base al bit relativo

- N.B. Gli esempi successivi useranno come valore di partenza il risultato ottenuto dagli esempi della conversione da decimale a binario.

Esempio 1 (8-bit)

- Convertire in decimale il floating point in rappresentazione 8-bit $(01000101)_{due}$:

  identifico i 3 campi: 0 100 0101

  mantissa: **1.**0101

  esponente: $(100)_{due} \rightarrow (4)_{dieci}$ - bias $\rightarrow 4 - 3 = 1$

  risultato:

  $(+1.0101)_{due} \cdot 2^1 \rightarrow (+10.101)_{due} \rightarrow$
  $1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = (2.625)_{dieci}$

Esempio 2 (8-bit)

- Convertire in decimale il floating point in rappresentazione 8-bit $(11010011)_{due}$.

  Provate a farlo in maniera autonoma.

Esempio 2 (8-bit)

- Convertire in decimale il floating point in rappresentazione 8-bit $(11010011)_{due}$:

  identifico i 3 campi: 1 101 0011

  mantissa: **1.**0011

  esponente: $(101)_{due} \rightarrow (5)_{dieci}$ - bias $\rightarrow 5 - 3 = 2$

  risultato:

  $(-1.0011)_{due} \cdot 2^2 \rightarrow (-100.11)_{due} \rightarrow$
  $-1 \cdot (1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2}) = (-4.75)_{dieci}$

Esempio 3 (8-bit)

- Convertire in decimale il floating point in rappresentazione 8-bit $(00011010)_{due}$.

  Provate a farlo in maniera autonoma.

Esempio 3 (8-bit)

- Convertire in decimale il floating point in rappresentazione 8-bit $(00011010)_{due}$:

  identifico i 3 campi: 0 001 1010

  mantissa: **1.**1010

  esponente: $(001)_{due} \rightarrow (1)_{dieci}$ - bias $\rightarrow 1 - 3 = -2$

  risultato:

  $(1.101)_{due} \cdot 2^{-2} \rightarrow (0.01101)_{due} \rightarrow$
  $0 \cdot 2^{0} + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \cdot 2^{-4} + 1 \cdot 2^{-5} = (0.40625)_{dieci}$

Esempio 4 (32-bit)

- Convertire in decimale il floating point in rappresentazione 32-bit $(11000100101001000010101000000000)_{due}$:

  identifico i 3 campi: 1 10001001 01001000010101000000000

  mantissa: **1.**01001000010101

  esponente: $(10001001)_{due} \rightarrow 2^7 + 2^3 + 2^0$ - bias =
  128 + 8 + 1 - 127 = 10

  risultato:

  $(-1.01001000010101)_{due} \cdot 2^{10} \rightarrow (-10100100001.0101)_{due} \rightarrow$
  $-1 \cdot (2^{10} + 2^8 + 2^5 + 2^0 + 2^{-2} + 2^{-4}) = (-1313.3125)_{dieci}$

Addizione

1. si prende il numero con l'esponente più piccolo e lo si porta ad avere l'esponente uguale all'altro

2. dato $e_a > e_b$ si calcola $e_a - e_b =$ *steps*

3. *steps* rappresenta la quantità di shifts verso destra che deve subire la mantissa del numero più piccolo

4. per ogni shift fatto l'esponente del numero più piccolo deve essere incrementato di 1

5. si sommano le mantisse dei due numeri (tenendo in considerazione il bit nascosto "1.")

6. si normalizza il risultato (per arrivare alla forma 1.xxxx)

N.B. Nei punti 3 e 6 ci possono essere perdite di bit se non ci sono sufficienti bit a disposizione per la mantissa.

Esempio semplificato

- Eseguire $(5)_{dieci}$ + $(3.625)_{dieci}$ assumendo una precisione di 4 bit:

$(5)_{dieci} = (101)_{due} = (1.01)_{due} \cdot 2^2$
$(3.625)_{dieci} = (11.101)_{due} = (1.1101)_{due} \cdot 2^1$

Shifting: 2 -1 = 1 (steps)
$(1.1101)_{due} \cdot 2^1 = (0.11101)_{due} \cdot 2^2$

$$
\begin{array}{lr}
 & 1\ \ 1 \\
\text{Somma:} & 1.01000\ + \\
 & 0.11101\ = \\
 & \text{— — — — —} \\
 & 10.00101
\end{array}
$$

Normalizzo: $(10.00101)_{due} \cdot 2^2 = (1.000101)_{due} \cdot 2^3$

Arrotondo (4 bit di precisione): $(1.0001)_{due} \cdot 2^3$

Arrotondamento

- Lo standard IEEE prevede diverse metodi per arrotondare i floating points quando richiedono troppi bit di precisione.
- Il metodo più semplice è quello del troncamento (perdo i bit in eccesso).
- Un metodo più sofisticato è quello del sistema GRS:
  - si usano 3 bit speciali aggiuntivi (guard bit, round bit e sticky bit) che vengono utilizzati per tenere "l'eccedenza" della mantissa
  - questi bit vengono usati sono in fase di calcolo del numero, quindi la dimensione dei floating points non cambia (singola precisione, doppia precisione, etc.)

GRS

```
mantissa format plus extra bits:

   1.XXXXXXXXXXXXXXXXXXXXXXX   0   0   0

   ^         ^                 ^   ^   ^
   |         |                 |   |   |
   |         |                 |   |   -  sticky bit (s)
   |         |                 |   -  round bit (r)
   |         |                 -  guard bit (g)
   |         -  23 bit mantissa from a representation
   -  hidden bit
```

- Quando la mantissa eccede i bit a disposizione, i bit in eccesso vengono “shiftati” nei 3 bit GRS.
- Non appena un “1” raggiunge lo sticky bit, questo viene settato a 1 e non cambia più il suo valore.

GRS

- In base alla combinazione dei valori di questi bit (GRS) si decide in che modo arrotondare:
    - 0xx → round down (x significa qualsiasi valore di bit)
    - 100 → se LSB della mantissa è 1 allora round up, altrimenti round down
    - 101 → round up
    - 110 → round up
    - 111 → round up
- N.B.:
  round down → arrotondo per difetto (troncamento)
  round up → arrotondo per eccesso (aggiungo 1 alla mantissa)