

| 1. Rappresentazione dell'informazione ed operazioni

| Basi per rappresentare un'informazione

Per essere rappresentabile, un informazione deve avere due elementi:

- Un *alfabeto di supporto*, cioè un insieme di simboli, che concatenati tra di loro in un certo modo vanno a definire delle "parole", che hanno un senso
- Una *codifica* dell'informazione, ovvero una funzione che associa le informazioni di un insieme I alle "parole" nell'insieme codice C

$$f : I \rightarrow C$$

Considerando come N_i la cardinalità di I e come N_c la cardinalità di C :

- Se $N_i < N_c$, allora la codifica è detta *ridondante* (Troppo codice per rappresentare le stesse info)
- Se $N_i > N_c$, allora la codifica è detta *ambigua* (Troppo poco codice per rappresentare tutte le info)

Esiste anche una funzione $g : C \rightarrow I$, detta *decodifica*.

Detto questo, una rappresentazione deve dunque avere:

- Economicità: minor numero possibile di caratteri
- Semplicità di codifica e decodifica
- Semplicità di elaborazione

| Rappresentazione dell'informazione numerica

| I numeri naturali

| Sistema di rappresentazione posizionale

Il nostro sistema di rappresentazione dei numeri è detto *posizionale* in *base 10*, cioè, la posizione di una cifra definisce una potenza di 10.

Esempio

Il sistema numerico posizionale permette di rappresentare i numeri come una somma di prodotti tra:

- una cifra c in posizione p (da 0 in poi, a partire da destra verso sinistra) e

- il valore della base b elevata alla p

$$\dots (c_2 \cdot b^2) + (c_1 \cdot b^1) + (c_0 \cdot b^0)$$

Prendiamo per esempio il numero 2348 in base 10: questo numero è una somma di prodotti tra una cifra in una certa posizione e la relativa potenza del 10

$$(2 \cdot 10^3) + (3 \cdot 10^2) + (4 \cdot 10^1) + (8 \cdot 10^0) = 2348$$

| Range di numeri rappresentabili da una base

Le cifre utilizzabili da ogni base b (cioè l'alfabeto di supporto) sono definite nell'insieme intervallo $[0, b - 1]$, e con x cifre si possono rappresentare numeri definiti nell'insieme intervallo $[0, b^x - 1]$

☰ Esempio

In base 10, posso utilizzare le cifre dell'intervallo

$$[0, 10 - 1] = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

E con 4 cifre posso rappresentare i numeri nell'intervallo

$$[0, 10^4 - 1] = [0, 9999]$$

ⓘ Definizione della base

Per specificare in quale base è un numero, posso scriverlo come

$$(\text{numero})_{\text{base}}$$

| Numeri binari

Nei sistemi digitali, si usa sempre il sistema numerico posizionale, ma in base 2. È possibile dunque usare le cifre dell'insieme $\{0, 1\}$ per rappresentare, con x cifre, i numeri dell'intervallo $[0, 2^x - 1]$

☰ Esempio

Usando 3 cifre, posso rappresentare i numeri dell'intervallo $[0, 7]$

$$\begin{aligned}(000)_2 &= (0 \cdot 2^2) + (0 \cdot 2^1) + (0 \cdot 2^0) = 0 \\(001)_2 &= (0 \cdot 2^2) + (0 \cdot 2^1) + (1 \cdot 2^0) = 1 \\(010)_2 &= (0 \cdot 2^2) + (1 \cdot 2^1) + (0 \cdot 2^0) = 2 \\(011)_2 &= (0 \cdot 2^2) + (1 \cdot 2^1) + (1 \cdot 2^0) = 3 \\(100)_2 &= (1 \cdot 2^2) + (0 \cdot 2^1) + (0 \cdot 2^0) = 4 \\(101)_2 &= (1 \cdot 2^2) + (0 \cdot 2^1) + (1 \cdot 2^0) = 5 \\(110)_2 &= (1 \cdot 2^2) + (1 \cdot 2^1) + (0 \cdot 2^0) = 6 \\(111)_2 &= (1 \cdot 2^2) + (1 \cdot 2^1) + (1 \cdot 2^0) = 7\end{aligned}$$

Per praticità, i numeri binari possono essere rappresentati in versione più compatta con i numeri *ottali* ed *esadecimali*.

| Numeri ottali

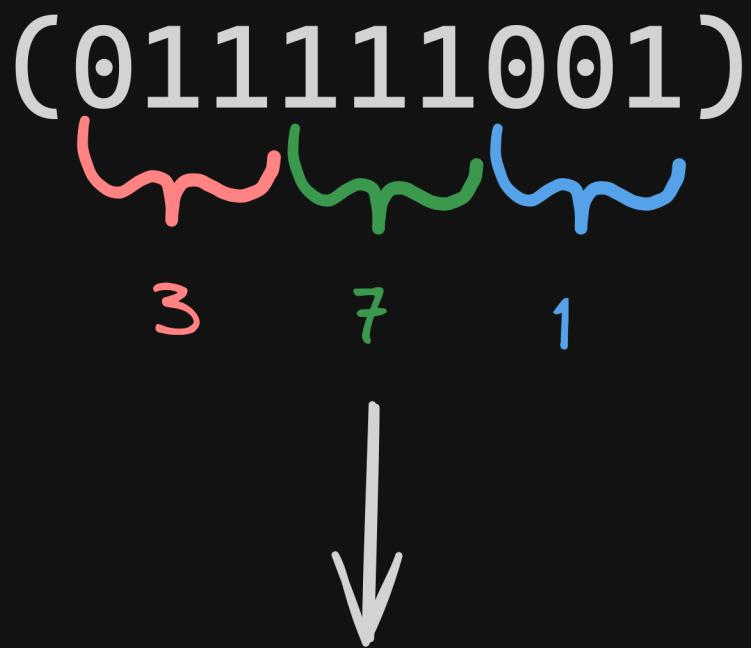
I numeri ottali seguono il sistema numerico posizionale in base 8. È possibile dunque usare le cifre dell'insieme $\{0, 1, 2, 3, 4, 5, 6, 7\}$ per rappresentare, con x cifre, i numeri dell'intervallo $[0, 8^x - 1]$

Visto che $8 = 2^3$, possiamo vedere 1 cifra ottale come se fosse 3 cifre binarie.

Esempio

$$(249)_{10} = (011111001)_2$$

Posso trasformare il numero binario in ottale, per avere una notazione più compatta, prendendo 3 bit alla volta



$$(249)_{10} = (371)_8 = (011111001)_2$$

I Numeri Esadecimali

I numeri esadecimali seguono il sistema numerico posizionale in base 16. È possibile dunque usare le cifre dell'insieme $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$ per rappresentare, con x cifre, i numeri dell'intervallo $[0, 16^x - 1]$

• A, B, C, D, E, F in esadecimale

Nel sistema esadecimale, le lettere rappresentano i numeri da 10 a 15 (cioè $16 - 1$)

$$\begin{aligned}A &= 10 \\B &= 11 \\C &= 12 \\D &= 13 \\E &= 14 \\F &= 15\end{aligned}$$

Visto che $16 = 2^4$, possiamo vedere 1 cifra esadecimale come se fosse 4 cifre binarie.

Esempio

$$(249)_{10} = (011111001)_2$$

Posso trasformare il numero binario in esadecimale per avere una notazione più compatta, prendendo 4 bit alla volta

Ovviamente posso ignorare 0

$$(011111001)$$

0 F 9



$$(249)_{10} = (F9)_{16} = (011111001)_2$$

Potenze importanti

Alcune potenze importanti in informatica sono:

- Kilo = $2^{10} \approx 10^3$
- Mega = $2^{20} \approx 10^6$
- Giga = $2^{30} \approx 10^9$

Conversione da base 10 ad una base generica

Dato un numero n in base 10, è possibile convertirlo ad una base generica b in questo modo:

- Dividi n per b
- Tieni da parte il resto della divisione
- Continua a dividere il risultato della divisione per b ed a salvarti i resti finché il quoziente non diventa 0
- La sequenza dei resti letta in ordine inverso di ottenimento rappresenta il numero n in base b

$$n = 62$$

$$b = 3$$

Quoziente Resto

62	
20	2
6	2
2	0
0	2

Leggi in questa
direzione

$$2022_3 = 2 \cdot 3^3 + 0 \cdot 3^2 + 2 \cdot 3^1 + 2 \cdot 3^0 = 62_{10}$$

| Operazioni binarie

| Addizione

Le somme binarie sono eseguite in questo modo:

- $0 + 0 = 0$
- $0 + 1 = 1$
- $1 + 0 = 1$
- $1 + 1 = 0$ con riporto di 1

Il riporto di 1 verrà sommato al bit successivo.

$$9+3=12$$



$$1001 + 0011 = 1100$$

Eseguo la somma, aggiungendo
in rosso gli 1 di riporto

$$\begin{array}{r} 11 \\ 1001 + (9) \\ \hline 0011 = (3) \\ \hline 1100 (12) \end{array}$$

| Sottrazione

Le somme binarie sono eseguite in questo modo:

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

- $0 - 1 = 1$ prendendo in prestito un bit dalla cifra a sinistra di ordine maggiore

Il caso $0 - 1$ funziona come nelle sottrazioni normali quando bisogna sottrarre un numero più grande ad un numero più piccolo: chiedendo in prestito un bit dalla cifra a sinistra. Se questo bit è ancora 0 , a sua volta lo chiede alla cifra a sinistra. Se questa volta è 1 , il bit in questione diventa 0 , ed il bit che riceve il prestito diventa 10 (cioé 2), che a sua volta passerà il bit alla cifra che l'ha richiesto, la quale diventerà 10 , mentre la cifra che ha prestato decrementerà ad 1 .

$$9-3=6$$



$$1001-0011=0110$$

Eseguo la differenza, aggiungendo in rosso gli 1 più significativi che vengono prestati.

$$\begin{array}{r} \cancel{0} \cancel{1} \\ \cancel{1} \cancel{0} \cancel{0} 1 - \end{array} \quad \begin{array}{l} (9) \\ (3) \\ \hline (6) \end{array}$$

In questo caso:

- 0 ha chiesto un bit in prestito alla cifra di sinistra

- La cifra di sinistra, essendo **0**, ha chiesto il bit in prestito alla cifra alla sua sinistra
 - La cifra di sinistra, essendo **1**, ha prestato il bit ed è diventata **0**
- La cifra, avendo preso in prestito il bit, diventa **10**. Adesso può prestare il bit alla cifra di destra, decrementandosi di 1 e diventando **1**
- Il bit viene ricevuto e la cifra diventa **10**. Il risultato di questa differenza è **1**, perché $2 - 1 = 1$

| Moltiplicazione

Le moltiplicazioni binarie sono eseguite in questo modo:

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

Si svolgono come le moltiplicazioni normali, moltiplicando ogni cifra del secondo fattore con quelle del primo, ed i risultati ottenuti si sommano.

Quantità di bit necessaria per la moltiplicazione

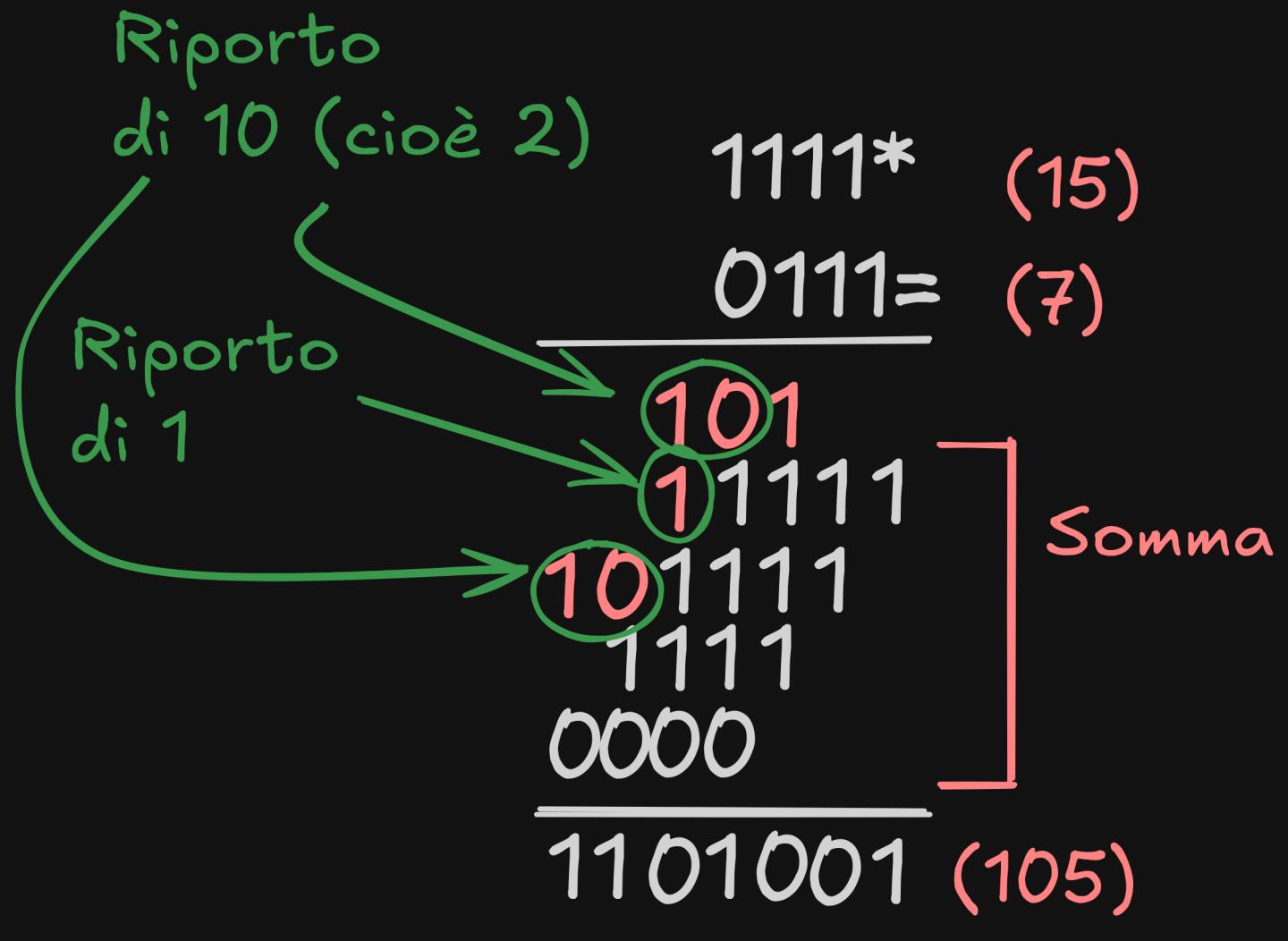
Per rappresentare la moltiplicazione di un numero a n bit, necessito di $2n$ bit.

Per esempio, con 4 bit posso rappresentare i numeri dell'intervallo $[0, 15]$, ma per rappresentare $15 \times 15 = 225$ servono $2 \cdot 4 = 8$ bit.

$$15 * 7 = 105$$



$$1111 * 0111 = 1101001$$



| Overflow

Nei numeri naturali, un overflow avviene quando eseguendo una somma il risultato è così grande da non essere rappresentabile dalla quantità di bit che ho a disposizione. Il risultato è corretto, ma non ho abbastanza bit per rappresentarlo.

$$10+7=17$$



$$1010 + 0111 = 10001$$

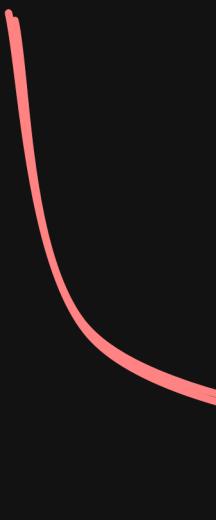
overflow

11

1010 + (10)

0111 = (7)

10001 (17)



Vedremo poi che quando ciò avviene nei [circuiti sommatori](#), i bit di overflow saranno inviati su una uscita apposita per essere trattati come riporto.

| Numeri interi

Per rappresentare i numeri interi ci sono principalmente 3 modi:

- Modulo e segno
- Complemento A1
- Complemento A2, il più conveniente da usare

Sapere se un numero è intero

Indipendentemente dalla tecnica di rappresentazione usata, non si può distinguere un numero intero da un naturale. Per sapere il valore rappresentato dal numero, ci deve

essere detto in modo esplicito quale tipo di numero abbiamo davanti (naturale, intero, reale), e quale tecnica di rappresentazione è stata utilizzata.

| Modulo e segno

Con la tecnica del modulo e segno, il segno è rappresentato dal bit più significativo: **0** è il segno "+" ed **1** è il segno "-".

Segno = $\begin{cases} 0 & \text{positivo} \\ 1 & \text{negativo} \end{cases}$

000	+0
001	+1
010	+2
011	+3
100	-0
101	-1
110	-2
111	-3

Usando il bit più significativo per rappresentare il segno, l'intervallo rappresentabile con n bit passa da $[0, 2^n - 1]$ a $[-2^{n-1} - 1, 2^{n-1} - 1]$.

| Complemento A1

Con la tecnica del complemento A1 (CA1), il segno è sempre rappresentato dalla cifra più significativa come nella tecnica del modulo e segno, ma i numeri negativi hanno anche i bit

invertiti.

$$\text{Segno} = \begin{cases} 0 & \text{positivo} \\ 1 & \text{negativo} \end{cases}$$

000	+0
001	+1
010	+2
011	+3
100	-3
101	-2
110	-1
111	-0

Usando il bit più significativo per rappresentare il segno, l'intervallo rappresentabile con n bit passa da $[0, 2^n - 1]$ a $[-2^{n-1} - 1, 2^{n-1} - 1]$.

| Complemento A2

Con la tecnica del complemento A2 (*CA2*), il bit più significativo non rappresenta il segno, ma il suo valore da solo è negativo, a cui vanno sommati i bit meno significativi: Per esempio, possiamo leggere **101** in CA2 come $-4 + 1 = -3$, perché il bit più significativo da solo rappresenta **100**, e proprio perché più significativo è negativo, mentre i restanti bit rappresentano la quantità positiva da sommarci, quindi in questo caso **01** = $+1$.

000	0
001	+1
010	+2
011	+3
100	-4
101	-3
110	-2
111	-1

Questo metodo permette anche di rappresentare 0 con una singola combinazione di bit, quindi in questo caso l'intervallo rappresentabile con n bit passa da $[0, 2^n - 1]$ a $[-2^{n-1}, 2^{n-1} - 1]$.

Il CA2 è il metodo preferito per rappresentare i numeri interi:

- L'intervallo rappresentabile è più grande di un numero
- Se so che il numero è rappresentato in CA2, posso subito sapere quale numero è rappresentato, in modulo e segno o CA1 dovrei anche analizzare il bit più significativo per sapere il segno del numero
- È più facile eseguire addizioni e sottrazioni, perché un'addizione è la somma tra numeri con lo stesso segno, mentre la sottrazione è la somma tra un numero positivo ed un numero negativo.
- Per ottenere il numero opposto, mi basta complementare tutti i bit e sommare +1.

💡 Estensione dei bit in CA2

A volte è possibile dover estendere un numero rappresentabile con un certo numero di bit per avere lo stesso numero ma con più bit (Per esempio, i numeri naturali `11` e `00000011` rappresentano sempre il numero 3, ma il secondo è stato esteso per usare 8 bit invece di 2).

In CA2, quando è necessario eseguire questa operazione, si estende il bit più significativo.

Per esempio, in CA2:

$+3 = 011$ con 3 bit = `00000011` con 8 bit.

$-5 = 1011$ con 4 bit = `11111011` con 8 bit.

| Overflow nel CA2

Nei numeri interi CA2, un overflow avviene quando eseguendo una somma il risultato è così grande da non essere rappresentabile dalla quantità di bit che ho a disposizione.

In particolare, poiché il valore dipende strettamente dal bit più significativo, un overflow può causare anche l'ottenimento di un valore errato.

$$\begin{array}{r}
 11 \\
 0110 + (6) \\
 \hline
 0111 = (7) \\
 \hline
 1101 (-4)
 \end{array}$$



Ho sommato due numeri positivi ed ho ottenuto un negativo.

$$\begin{array}{r}
 0010 + (+2) \\
 \hline
 1101 = (-3) \\
 \hline
 1111 (-1)
 \end{array}$$



Corretto

$$\begin{array}{r}
 111 \\
 0110 + (6) \\
 \hline
 1011 = (-5)
 \end{array}$$

 10001 (+1)

È avvenuto un overflow,
ma poiché il risultato è
corretto se consideriamo solo
i bit a nostra disposizione,
l'overflow verrà inviato sull'uscita
del riporto, come nei numeri interi.

$$\begin{array}{r} 1 \ 1 \\ 1011+ (-5) \\ \hline 1010= (-6) \\ \hline \boxed{1}01\ 01 \quad (+5) \end{array}$$

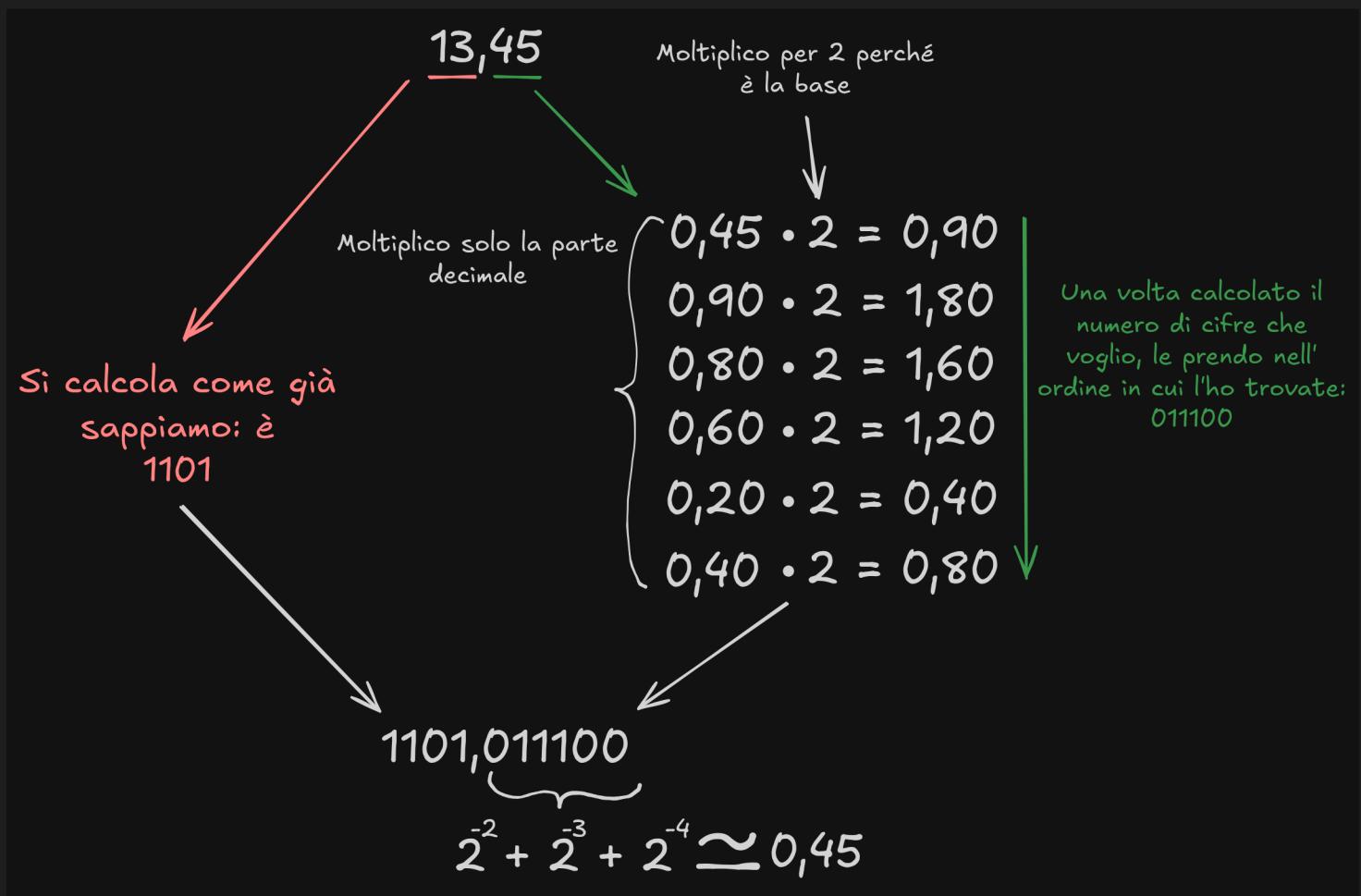
 È avvenuto un overflow,
e nonostante il bit di overflow
venga mandato sull'uscita del riporto,
il risultato è sbagliato, perché
ho sommato due numeri negativi
ed ho ottenuto un positivo.

La differenza più grande che posso fare con n bit tra due numeri in CA2 con segno opposto è $(2^{n-1} - 1) - 2^{n-1} = -1$. Ciò significa che gli overflow con risultati errati possono avvenire solo tra addendi dello stesso segno, e si rilevano controllando il bit più significativo del risultato: se è opposto al bit più significativo dei due addendi, allora è avvenuto un overflow.

I numeri reali sono composti da una parte intera ed una parte decimale dopo la virgola. La parte intera viene rappresentata come già sappiamo, mentre la parte decimale è rappresentata da potenze della base con esponente negativo.

Per convertire la parte decimale di un numero in binario, dobbiamo:

- 1) Moltiplicare la parte decimale del numero per la base da raggiungere (in questo caso, 2) e mettere da parte il risultato
- 2) Ripetere l'operazione sui risultati che si ottiene, per il numero di cifre decimali che vogliamo avere
- 3) La rappresentazione è data dalle parti intere nell'ordine in cui le abbiamo ottenute



⚠ Precisione dei numeri reali in binario

Già da questo esempio, possiamo notare che i risultati ottenuti non sono esatti, ma approssimazioni. Successivamente vedremo anche che nelle operazioni tra numeri reali vengono fatti dei cambiamenti di scala e spostamenti di bit. Ciò, insieme al fatto che non è possibile rappresentare una quantità infinita di numeri decimali in una quantità finita di memoria, vuol dire che inevitabilmente si andranno a creare degli errori di approssimazione, che se sommati possono piano piano rendere i numeri sempre meno precisi.

Per rappresentare i numeri reali in binario, ci sono due tecniche:

- Rappresentazione a virgola fissa (*fixed point*)
- Rappresentazione a virgola mobile (*floating point*)

| Rappresentazione fixed point

La rappresentazione *fixed point* rappresenta i numeri decimali con n bit. Per la parte intera si usano una certa quantità k di bit, e per la parte decimale si usano tutti i restanti.

Il problema di questo approccio è che anche bilanciando perfettamente la quantità di bit tra le due parti, non posso rappresentare contemporaneamente ne numeri minuscoli, ne enormi. È necessario dunque usare la notazione scientifica, approccio usato dalla seconda tecnica di rappresentazione.

| Rappresentazione floating point IEEE 754

La rappresentazione floating point rappresenta il numero reale tramite una tripla di valori:

- Il segno (0 = positivo, 1 = negativo)
- L'esponente per la notazione scientifica
- La mantissa, cioè il numero di cifre dopo la virgola

13,45



1101,0111000



1,1010111000 • 2³ Esponente
0011

0000111010111000

S

E

M

Il segno è rappresentato sempre con 1 solo bit, mentre esponente e mantissa possono variare. Lo standard *IEEE 754* definisce le caratteristiche di 3 tipi di floating point:

- 16 bit ⇒ half-precision, con 5 bit di esponente, 10 bit di mantissa ed un valore bias di 15
- 32 bit ⇒ single-precision, con 8 bit di esponente, 23 bit di mantissa ed un valore bias di 127
- 64 bit ⇒ double-precision, con 11 bit di esponente, 52 bit di mantissa ed un valore bias di 1023

Osserviamo le categorie di tipi rappresentate dall'*IEEE 754*:

- **NaN** : Not a number. È rappresentato con i bit dell'esponente tutti ad **1**, e la mantissa rappresenta un numero diverso da **0**
- **Infinity** : Infinito. È rappresentato con i bit dell'esponente tutti ad **1**, e la mantissa rappresenta **0**, cioè tutti i bit della mantissa sono a **0**
- **Zero** : il numero 0. È rappresentato con tutti i bit sia dell'esponente, che della mantissa, impostati a **0**
- Numeri *normalizzati*: sono i numeri rappresentati normalmente. I bit dell'esponente devono rappresentare un numero nell'intervallo $[1, 2^n - 2]$ dove n è il numero di bit dedicati all'esponente. Praticamente, può essere qualsiasi valore diverso da tutti **0** o tutti **1**. Il valore della mantissa non ha importanza.
- Numeri *denormalizzati*: sono numeri compresi tra 0 ed 1 così vicini allo zero da non essere normalmente rappresentabili. I bit dell'esponente sono tutti impostati a **0**, e la mantissa può essere qualsiasi valore diverso da tutti **0**

I Uso del bias e differenza tra numeri normalizzati e denormalizzati

Cerchiamo di capire la differenza tra numeri normalizzati e denormalizzati. Questo è il formato di un numero in IEEE 754, dove S è il segno, E è l'esponente ed M è la mantissa:

$$(-1)^S \cdot 1, M \cdot 2^E$$

Essendo un numero in notazione scientifica, la virgola deve essere sempre dopo la prima cifra, che è implicito sia 1.

Immaginiamo sia in half-precision: per rappresentare l'esponente abbiamo a disposizione 5 bit, di cui uno è usato per indicare l'esponente. Inoltre, dobbiamo tenere conto che come abbiamo appena visto le combinazioni di esponente con i bit impostati a tutti **1** sono riservate per rappresentare **NaN** e **infinity**, e le combinazioni di esponente con i bit impostati a tutti **0** sono riservate ai numeri denormalizzati ed a **Zero**. Il valore più piccolo normalizzato rappresentabile è quindi quello con esponente minimo e mantissa minima.

L'esponente non è in CA2, ma è *biased*: invece di riservare un bit al segno, l'esponente usa tutti e 5 i bit per codificare un numero tra 1 e 30 (perché come abbiamo appena detto **00000** e **11111** sono riservati), a cui poi verrà sottratto il bias (15 in half-precision, cioè $2^{E-1} - 1$) per ottenere il vero valore dell'esponente. Il vantaggio di questo approccio, e quindi lo scopo del bias, è che nel numero in IEEE754 posso rappresentare l'esponente come un numero positivo esponente codificato = esponente effettivo + bias, e per fare i calcoli posso ottenere esponente effettivo = esponente codificato - bias. Ciò mi permette di confrontare numeri già codificati in IEEE754 senza preoccuparmi del segno dell'esponente.

Detto questo, visto che l'esponente codificato minimo nei numeri normalizzati è **00001**, il valore esponente effettivo è $1 - 15 = -14$ e dunque il più piccolo numero normalizzato rappresentabile in half-precision è $1,0000000001 \cdot 2^{-14}$.

Il fatto che ci sia quell'1 prima della virgola è problematico, perché se fosse 0, allora si potrebbero rappresentare numeri ancora più piccoli con la stessa quantità di bit.

Qui entra il concetto di numero denormalizzato: quando l'esponente ha tutti i bit impostati a 0, non viene seguita la regola del bias che quindi fa rimanere l'esponente fisso a -14, e l'1 implicito a sinistra della virgola diventa 0. Dunque il più piccolo numero non normalizzato rappresentabile in half precision è $0,000000001 \cdot 2^{-14}$.

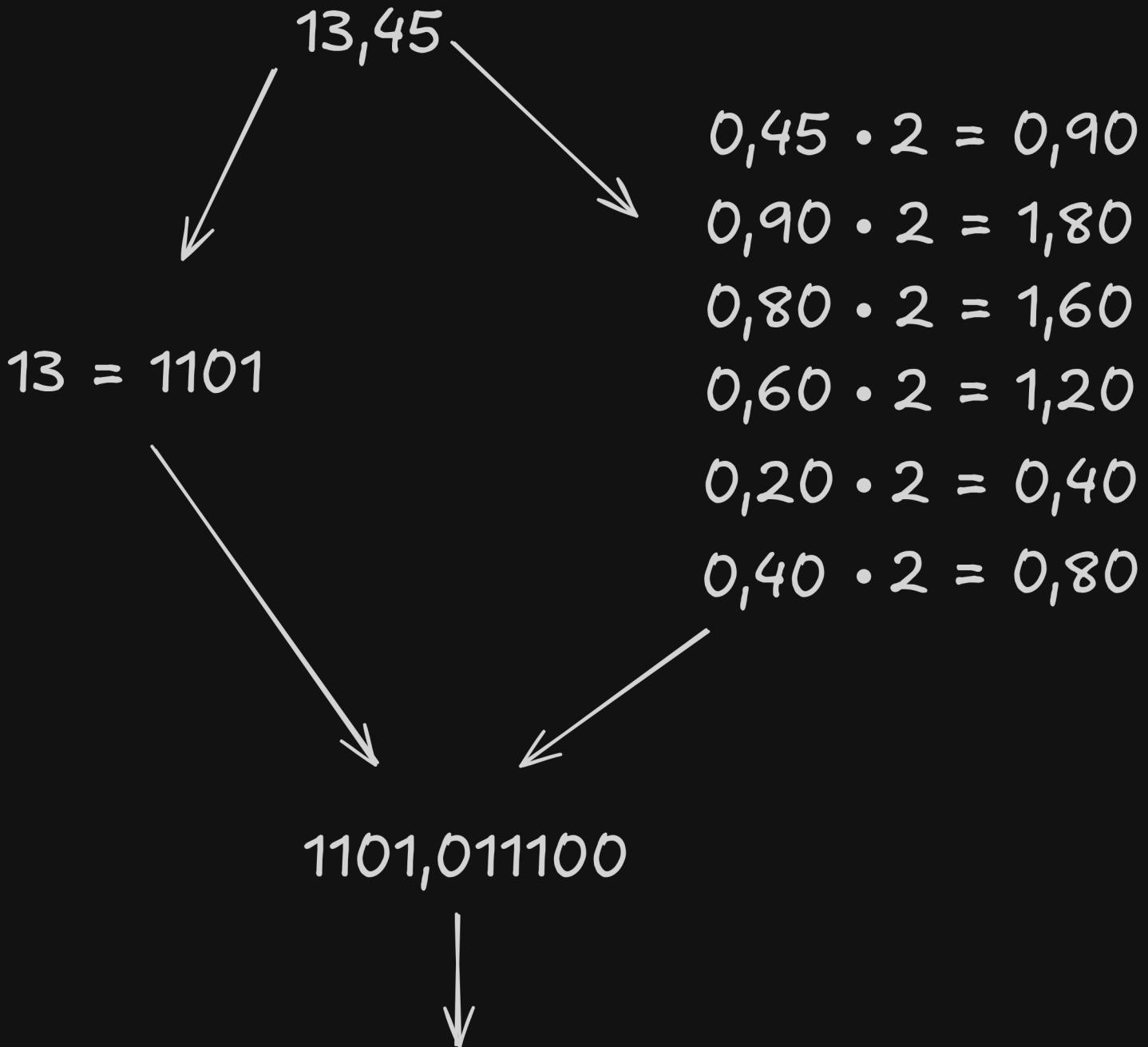
Non dovessero essere ancora chiari i concetti appena presentati, si allegano dei collegamenti esterni ad altre spiegazioni riguardanti bias^[1] e differenza tra numeri normalizzati e denormalizzati^[2].

| Conversione da decimale ad half precision e viceversa

Eseguiamo i seguenti passaggi:

- 1) Convertiamo in binario sia la parte intera del numero, che quella decimale
- 2) Portiamo in notazione scientifica il numero per ottenere la mantissa
- 3) Aggiungiamo il bias al valore dell'esponente effettivo
- 4) Scriviamo il numero in forma segno + esponente codificato con bias + mantissa
- 5) Opzionalmente, scriviamo il numero in esadecimale per renderlo più corto da scrivere e leggere

1) Converto in binario



2) Converto in notazione scientifica

Aggiunto uno 0 per avere 10 bit di mantissa

$$1,1010111000 \cdot 2^3$$

M

3) Sommo il bias
all'esponente

$$3 + 15 = 18 = 10010$$

Bias dell'half precision

4) Scrivo il numero in
IEEE754

010010101010111000
S E + Bias M

5) Converto in hex per
praticità

0100|1010|1011|1000 = 4AB8
4 A B 8

Posso eseguire l'operazione inversa eseguendo le operazioni al contrario.

1) Convertto in binario
se non lo è già

$CE94 = 1100|1110|1001|0100$

C E 9 4

↓

2) Rimuovo il bias
dall'esponente
(in half precision, i 5 bit
dopo il primo)

1100111010010100

S E + Bias = 19

Il numero è negativo

$19 - 15 = 4 = 00100$

3) Riporto il numero in notazione
scientifica (ricordando che essendo
un numero normalizzato con esponente
diverso da tutti 0, è implicito ci fosse
l'1 prima della mantissa)

$1,1010010100 \cdot 2^4$

M

4) Riporto il numero in notazione classica

11010,010100



5) Converto in decimale

11010,010100

$$2^4 + 2^3 + 2^1 = 26$$

$$2^{-2} + 2^{-4} = 0.25 + 0,0625$$

→ -26,3125

| Operazioni in IEEE 754

| Somma e sottrazione

Il procedimento di sottrazione tra numeri in IEEE754 è lo stesso dell'addizione, in quanto consiste nel sommare due numeri con segno discordi.

- 1) Ottengo i due numeri da sommare/sottrarre nello standard IEEE754
- 2) Allineo i due operandi nella notazione scientifica: incremento l'esponente più piccolo n volte per farlo diventare uguale a quello più grande e poi faccio uno shift a destra di n posizioni del numero che ho modificato.
- 3) Eseguo l'operazione di somma o differenza in base al segno dei due numeri
- 4) Se necessario, normalizzo il risultato per avere 1 davanti alla virgola
- 5) Riporto il risultato in standard IEEE 754

Perdita di precisione

Come accennato qualche paragrafo fa, l'operazione di allineamento al punto 2 va a causare una perdita di precisione, in quanto eseguendo lo shift a destra, i bit meno significativi della mantissa vengono persi.

Per vedere dal vivo questo effetto collaterale, provate ad eseguire in python `print(0.1+0.2)` e noterete che dopo un certo numero di cifre decimali, inizieranno ad essere generati valori errati, o nel caso di `print(0.1+0.7)`, il valore è proprio approssimato.

1) Prendiamo due numeri in half precision

$$A = 26,42 = \underbrace{0100111}_{S} \underbrace{010011010}_{E + Bias} \underbrace{10}_{M}$$

$$B = -37,68 = \underbrace{110100}_{S} \underbrace{001010110101}_{E + Bias} \underbrace{01}_{M}$$



2) Troviamo il numero con esponente più piccolo, e lo allineiamo con quello più grande

$$A = 10011 = 19 \rightarrow E_A = 19 - 15 = 4 \rightarrow A = 1,1010011010 \cdot 2^4$$

$$B = 10100 = 20 \rightarrow E_B = 20 - 15 = 5 \rightarrow B = 1,0010110101 \cdot 2^5$$

Quindi devo shiftare a destra A di $5 - 4 = 1$ posizione

$$A = 1,1010011010 \cdot 2^4$$



$$A = 0,1101001101 \cdot 2^5$$

Shiftando ho perso lo 0 meno significativo perché la mantissa deve rimanere a 10 bit.



3 e 4) Eseguiamo l'operazione tra le mantisse ed il valore a sinistra della virgola. In questo caso è una differenza perché i segni sono discordi.

$$\begin{array}{r} 0,1101001101 \\ 1,0010110101- \\ \hline 0,1101001101 \\ \hline 0,0101101000 \end{array}$$

Ho aggiunto gli 0 per mantenere la mantissa a 10 bit

$$\text{Il risultato è } 0,0101101000 \cdot 2^5$$

Normalizzandolo diventa $1,0110100000 \cdot 2^3$

\downarrow

5) Riportiamo nello standard IEEE754

Segno

$$-37,68 + 26,42 = \text{risultato negativo}$$

$$\boxed{S = 1}$$

Segno

Esponente

$$\boxed{E = \textcircled{3} + 15 = 18 = 10010}$$

Esponente + bias

Mantissa

È quella che abbiamo ottenuto come risultato della somma e che poi abbiamo normalizzato

$$M = \boxed{0110100000}$$

Mantissa

Rappresentazione IEEE754

$$\boxed{\begin{matrix} 1 & 10010 & 0110100000 \\ S & E + \text{bias} & M \end{matrix}}$$

$$1100|1001|1010|0000$$



$$(C940)_{16}$$

Overflow in IEEE754

Se l'esponente esce dall'intervallo rappresentabile, avviene un overflow.

I Moltiplicazione

La moltiplicazione si esegue in un modo un po' diverso rispetto alla somma:

- 1) Trova il segno del risultato confrontando il segno dei numeri da moltiplicare
- 2) Trova l'esponente del risultato sommando i due esponenti effettivi senza i relativi bias

3) Moltiplica le mantisse, considerando l'1 a sinistra della vigola (Abbiamo considerato l'esponente prima, quindi qui non c'è bisogno di fare shift). Se necessario, normalizzo il risultato per avere 1 davanti la virgola

4) Unisci il tutto nel formato dello standard

$A = \boxed{0100010101000000}$

S E

M

$B = \boxed{1100000110000000}$

S

E

M

1) Trova il segno del risultato

$$A = 0 = +$$

$$B = 1 = -$$

Come nelle moltiplicazioni normali,
positivo \times negativo = negativo

$\boxed{S = 1}$

Segno



2) Trova l'esponente del risultato

$$E_A = 10001 = 17 - 15 = 2$$

$$E_B = 10000 = 16 - 15 = 1$$

$$2 + 1 = 3$$

L'esponente del risultato sarà

$$E_R = 3 + 15 = 18 = \boxed{10010}$$

Esponente + bias



3) Moltiplica le mantisse, considerando anche l'1 che si trova prima della virgola

nella notazione scientifica

Eseguendo la moltiplicazione, posso togliere gli 0 meno significativi, perché tanti saranno 0

$$\begin{array}{r} 1,0101 \times \\ 1,0110 = \\ \hline 00000 \\ 110101 \\ 110101 \\ 00000 \\ 10101 \\ \hline 1,11001110 \end{array}$$

Quindi la mantissa è: Ho aggiunto gli 0 per arrivare a 10 bit di mantissa

$$1,1100111000 \cdot 2^3$$

Mantissa

Esponente calcolato nel punto 2

↓

4) Unisco il tutto nel formato dello standard



C B 3 8
1100|1011|0011|1000

↓
 $(CB38)_{16}$

1) https://it.wikipedia.org/wiki/IEEE_754#Numeri_a_32bit ↵

