

Sistemi numerici

Il sistema che utilizza il pc per elaborare le istruzioni è il **sistema binario**, esso è composto da due sole cifre 0 e 1.

Utilizzando il sistema binario e avendo n cifre a disposizione possiamo rappresentare 2^n numeri, questo per via del calcolo combinatorio:

$$2 \cdot 2_1 \dots 2_n$$

• Numeri decimali

$$5374_{10} = 5 \cdot 10^3 + 3 \cdot 10^2 + 7 \cdot 10^1 + 4 \cdot 10^0$$

• Numeri binari

$$1101_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13_{10}$$

Potenze di 2

• $2^0 = 1$	• $2^8 = 256$
• $2^1 = 2$	• $2^9 = 512$
• $2^2 = 4$	• $2^{10} = 1024$
• $2^3 = 8$	• $2^{11} = 2048$
• $2^4 = 16$	• $2^{12} = 4096$
• $2^5 = 32$	• $2^{13} = 8192$
• $2^6 = 64$	• $2^{14} = 16384$
• $2^7 = 128$	• $2^{15} = 32768$

Conversione numerica

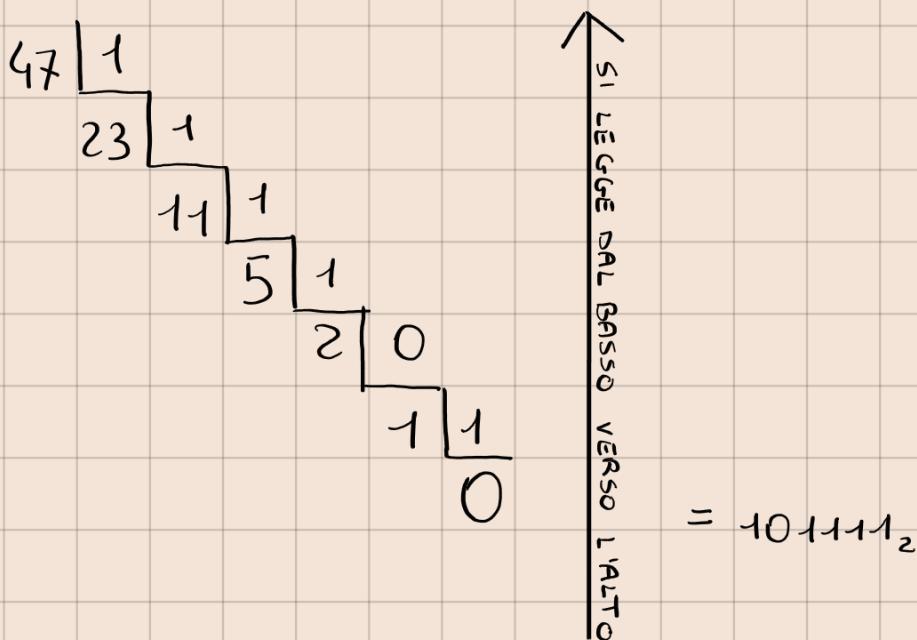
• Conversione da binario a decimale:

Un esempio di conversione in decimale può essere 10011₂ che in decimale diventa:

$$\begin{aligned} 10011_2 &= 2^4 \cdot 1 + 2^3 \cdot 0 + 2^2 \cdot 0 + 2^1 \cdot 1 + 2^0 \cdot 1 = \\ &= 16 \cdot 1 + 8 \cdot 0 + 4 \cdot 0 + 2 \cdot 1 + 1 \cdot 1 = \boxed{19_{10}} \end{aligned}$$

• Conversione da decimale a binario:

Un esempio di conversione in decimale può essere 47_{10} che in decimale diventa:



Per la conversione da decimale a binario ci sono 2 metodi:

• Metodo 1:

- Trova la più grande potenza di 2 più adatta;
- sottrai;
- se non esce 0 ripeti.

Esempio:

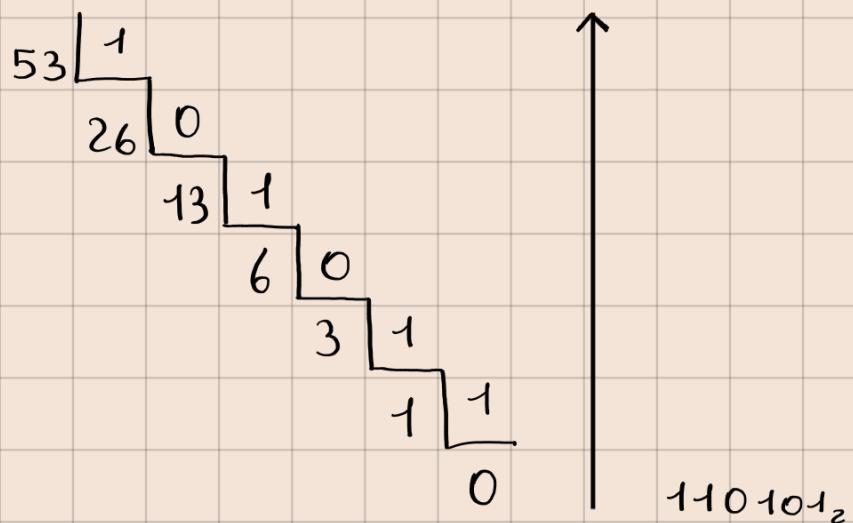
53_{10} $\rightarrow 53 - 32 = 21$ $\rightarrow 21 - 16 = 5$ $\rightarrow 5 - 4 = 1$	$32 \cdot 1$ $16 \cdot 1$ $4 \cdot 1$ $1 \cdot 1$
---	--

$\longrightarrow 110101_2$

• Metodo 2:

- dividi per 2;
- metti il resto a sinistra della rappresentazione binaria.

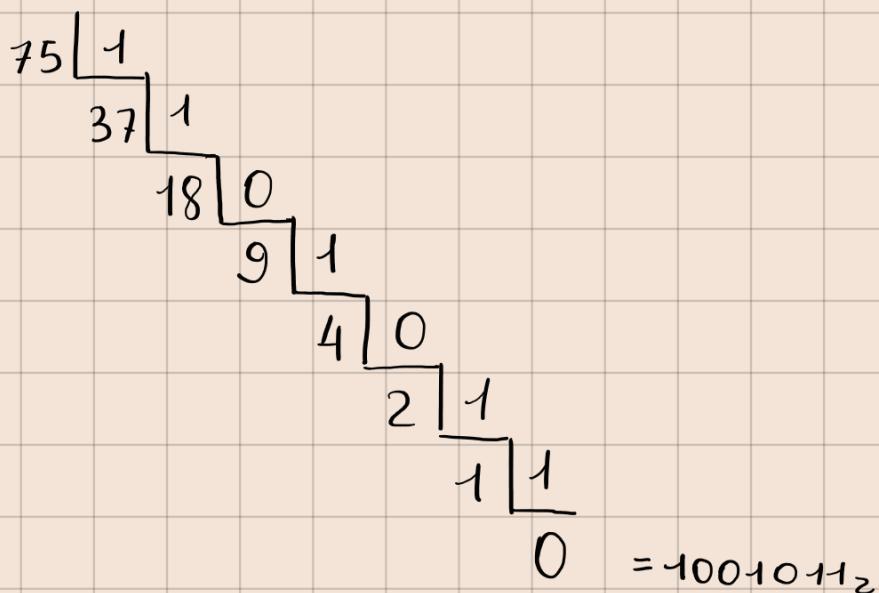
Esempio:



Altri esempi dei due metodi possono essere:

$$75_{10} = 64 + 8 + 2 + 1 = 1001011_2$$

oppure



Da x a y

Quando invece abbiamo una base x e vogliamo convertirla ad una base y prima convertiamo da x a decimale e poi da decimale a y.

Esempio:

Abbiamo 25_{20} e vogliamo convertirlo in base 4:

1) Convertiamo prima in base 10 quindi:

$$25_{20} = 2 \cdot 20^1 + 5 \cdot 20^0 = 40 + 5 = 45$$

2) convertiamo 45 in base 4 e per fare questo basterà dividere 45 finché non esce 0 e ad ogni divisione metteremo da parte i resti:

RESTI DI SOTTRAZIONI

$$\begin{array}{r} 45 \\ | \\ 11 \quad 3 \\ | \\ 2 \quad 2 \\ | \\ 0 \end{array}$$

$= 231_4$

Valori binari e intervalli

- Numero decimale a N cifre

- Quanti valori? 10^n

- Range? $[0, 10^n - 1]$

Esempio con un numero decimale a 3 cifre:

- $10^3 = 1000$ possibili valori;

- Range: $[0, 999]$

- Numero binario a n cifre

- Quanti valori? 2^n

- Range $[0, 2^n - 1]$

Esempio con un numero binario a 3 cifre:

- $2^3 = 8$ possibili valori

- Range: $[0, 7] = [000_2, 111_2]$

Numeri esadecimali

Hex Digit	Decimal Equivalent	Binary Equivalent
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Le caratteristiche principali dei numeri esadecimali sono 2:

- ha come base 16
- abbreviazione binaria

Conversione da esadecimale a binario

Proviamo per esempio a convertire 4AF (anche scritto 0x4AF) in binario:

0 1 0 0 | 1 0 1 0 | 1 1 1 1
 4 A F 2

Conversione da esadecimale a decimale

Per la conversione da esadecimale a decimale invece proviamo a convertire sempre 4AF₁₆ in decimale:

$$16^2 \cdot 4 + 16^1 \cdot 10 + 16^0 \cdot 15 = 1199_{10}$$

Bits, Bytes, Nibble...

Un bit è una singola unità di un numero binario ad esempio 0 o 1.

Invece un **byte** sono 8 bit ed un nibble 4 bit, quindi 2 nibble formano un byte.

- Un esempio di bits può essere:

- Esempio di Bytes & Nibble invece può essere:

byte

1 0 0 1 0 1 1 0

 Nibble

- Infine un esempio di ByTes può essere.

Grandi potenze di 2

Esempi di grandi potenze di z sono:

- $2^{10} = 1 \text{ Kilo} \approx 1000(1024)$
 - $2^{20} = 1 \text{ mega} \approx 1 \text{ milione}(1048576)$
 - $2^{30} = 1 \text{ giga} \approx 1 \text{ miliardo}(1073741824)$

Addizioni

Le addizioni con numeri decimali e binari si fa in due modi diversi:

- ## • decimalle

$$\begin{array}{r} 1 \\ 37341 + \\ 5168 = \\ \hline 8902 \end{array}$$

REPORT

• binario

$$\begin{array}{r} \text{1} \ 0 \ 1 \ 1 + \\ \text{0} \ 0 \ 1 \ 1 = \\ \hline \text{1} \ 1 \ 1 \ 0 \end{array}$$

RIPORTI

In binario le cose che cambiano nell'addizione sono:

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

Esempi di operazioni esadecimali

Facciamo 2 esempi di operazione con numeri esadecimali:

• Addizione:

$$\begin{array}{r} \text{3 A } \overset{1}{\text{0}} \ 9 + \\ \text{1 B } \overset{1}{\text{1}} \ 7 = \\ \hline \text{5 5 2 0} \end{array}$$

↓
G
A
B
C
D
E
F

① con riporto di 1

• Sottrazioni:

$$\begin{array}{r} \text{3 D } \overset{1}{\text{0}} \ 9 - \\ \text{1 B } \overset{1}{\text{2}} \ 7 = \\ \hline \text{2 1 E 2} \end{array}$$

Overflow

Poiché i sistemi digitali operano con un numero fisso di Bit, è necessario considerare quei casi in cui, durante un'addizione o una moltiplicazione, il numero di bit sia insufficiente a poter rappresentare il risultato.

Questo problema prende il nome di overflow (ossia strabordare)

Un esempio di overflow può essere $101 \cdot 111$

$$\begin{array}{r} 101 \times \\ 111 = \\ \hline 101 + \\ 101 // + \\ \hline 101111 = \\ 100011 \end{array}$$

In questo caso i due moltiplicandi utilizzano entrambi 3 bit ma il loro prodotto ne richiede almeno 6.

In questi casi i bit in eccesso vengono **scartati** generando un risultato sballato perché togliendo l'eccesso verrebbe $101 \cdot 111 = 011$, dunque $5 \cdot 7 = 3$

Shift di Bit a destra o a sinistra

Un operatore binario aggiuntivo rispetto alla normale aritmetica decimale è l'operazione di **shift** (o spostamento).

Lo shift è un operatore che prevede lo spostamento **a destra o a sinistra** di una certa quantità di caselle di tutti i bit del numero binario sul quale viene applicato

$$000101_2 \ll 2 = 010100_2$$

$$011000_2 \gg 2 = 000110_2$$

Numeri binari negativi

Fino ad ora, abbiamo parlato di numeri binari positivi.

Per comprendere ulteriori operazioni aritmetiche è necessario introdurre il **segno** nei numeri binari.

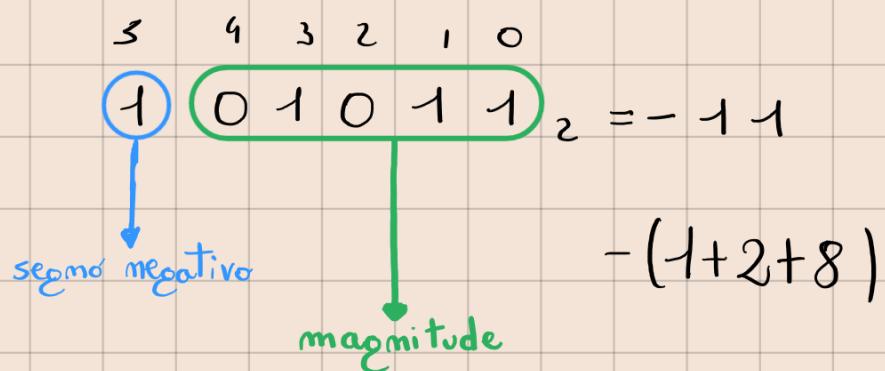
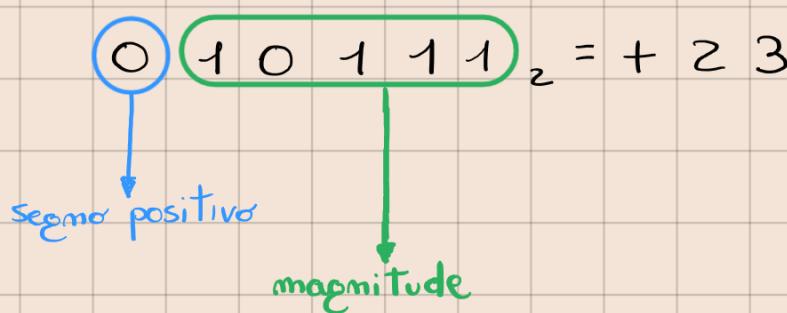
Esistono 2 diversi metodi di rappresentare i numeri negativi e sono:

- 1) i numeri in **Sign / Magnitude**
- 2) i numeri in **Complemento a 2**

Numeri in Sign / Magnitude

Nel primo metodo di rappresentazione la differenza è poca: il most significant Bit del numero rappresenta il segno, mentre tutti gli altri rappresentano il suo valore (o **magnitude**).

Nel caso in cui il bit del segno assume valore 0 allora il numero sarà positivo mentre se è 1 allora il numero sarà negativo:



Quindi dati n bit nei 2 metodi di rappresentazione abbiamo i seguenti intervalli di valori disponibili:

$$[0, 2^n - 1]$$

senza segno

$$[-(2^{n-1} - 1), 2^{n-1} - 1]$$

sign / magnitude

Nonostante ciò, anche questo formato ha delle problematiche:

- le addizioni non funzionano;
- abbiamo 2 modi per rappresentare lo 0 (1000 e 0000, corrispondenti a -0 e +0).

$$\begin{array}{r} 1\ 1\ 1\ 0 + \\ 0\ 1\ 1\ 0 = \\ \hline 1\ 0\ 1\ 0 0 \end{array}$$

In questa addizione per esempio c'è un errore perché anche se scartassimo l'overflow avremmo comunque $-6 + 6 = +4$

Numeri in complemento a 2

Per risolvere questo problema è stato introdotto il sistema binario basato sul **complemento a 2**, che va ad aggiungere un ulteriore significato al most significant Bit.

In questo sistema infatti assume il suo valore effettivo ma a differenza del sistema senza segno, il valore assoluto sarà negativo.

Dati n bit, il most significant Bit assumerà il valore $-(2^{n-1})$, mentre tutti gli altri bit manterranno il valore positivo:

$$1\ 0\ 1\ 1\ 1_2 = 1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + (-1 \cdot 2^4) = -9_{10}$$

Per poter calcolare il complemento a 2 di un numero decimale negativo, invece, è prima necessario calcolare il **Complemento a 1** del suo corrispettivo numero binario positivo.

Com complemento a 1 si intende semplicemente invertire tutti i bit, trasformando quindi tutti i suoi 0 in 1 e viceversa:

$$-25_{10} \rightarrow +25_{10} = 011001_2 \rightarrow 100110_2$$

Una volta calcolato il prossimo passo sarà semplicemente sommare 1 al risultato del complemento a 1:

$$100110 + 1 = 100\ 111_2$$

Per confermare che la conversione sia avvenuta correttamente, possiamo calcolare il valore del complemento a 2 quindi:

$$\begin{array}{r} \text{sg 3 2 1 0} \\ 100111_2 = 1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 0 \cdot 2^4 + (-1 \cdot 2^5) = -25_{10} \end{array}$$

L'utilità del complemento a 2 è evidente, perché grazie al modo in cui viene rappresentato è possibile svolgere operazioni tra numeri di segno opposto senza problematiche.

Viene anche risolto il problema dei due zeri, perché 0000 corrisponde a 0 mentre 1000 sarà -8.

$$+6 = 0110_2$$

$$-6 = 1010_2$$

$$\begin{array}{r} 1010 + \\ 0110 = \\ \hline 10000 \end{array}$$

Scartando l'overflow otteniamo $+6 - 6 = 0$, quindi il risultato è corretto.

Il range di valori negativi rappresentabili in complemento a 2 sarà:

$$\underbrace{[-(2^{m-1}), 2^{m-1}-1]}_{\text{complemento a 2}}$$

Incrementare il numero di bit

In alcuni casi nel complemento a 2 può essere necessario incrementare il numero di bit utilizzati, per far questo avremmo bisogno di tenere conto del **segno** del numero rappresentato:

- il numero 010011_2 può essere allungato aggiungendo la quantità di 0 desiderata:

$$010011_2 \longrightarrow 00001011_2$$

questo è possibile perché è un numero **positivo**.

- il numero 110011_2 può essere allungato aggiungendo la quantità di 1 desiderata:

$$110011_2 \longrightarrow 111110011_2$$

questo è possibile perché è un numero **negativo**.

Nel caso in cui fosse utilizzato lo 0 al posto dell'1, il valore finale del numero cambierebbe.

Lo shift come moltiplicazione o divisione

Lo spostamento di bit nel caso del complemento a 2 può essere utilizzato come **moltiplicazione e divisione**, ma solo se il moltiplicatore o divisore sia una potenza di 2:

$$B \cdot 2^m = B \ll m$$

$$B / 2^m = B \ggg m$$

Il motivo per cui nella divisione utilizziamo il **triplo shift a destra** (\ggg) è semplicemente un modo per segnalare la necessità di

dove tenere conto del segno in caso venga applicato lo shift ad un numero negativo.

Esempi:

- $00001 \ll 2 = 00100 \rightarrow 1 \cdot 2^2 = 4$
- $11101 \ll 2 = 10100 \rightarrow -3 \cdot 2^2 = -12$
- $01000 \ggg 2 = 00010 \rightarrow 8 \cdot 2^2 = 2$
- $10000 \ggg 2 = 11100 \rightarrow -16 \cdot 2^2 = -4$

Numeri binari razionali

Come per il sistema decimale, anche nel sistema binario è possibile rappresentare i numeri razionali.

Esistono 2 tipi di rappresentazione utilizzabili:

- virgola fissa (fixed-point numbers)
- virgola mobile (floating-point numbers)

Numeri a virgola fissa

Questo sistema è molto simile alla classica rappresentazione dei numeri razionali nel sistema decimale.

Il modo in cui i numeri razionali vengono rappresentati corrisponde alla seguente somma:

$$134.56_{10} = 1 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0 + 5 \cdot 10^{-1} + 6 \cdot 10^{-2}$$

In binario invece sarà:

$$110.11_2 = 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} = 6.75_{10}$$

Per poter convertire un numero decimale razionale in un numero binario razionale, è necessario scomporre il processo in due fasi:

- 1) Conversione in binario della parte intera del numero decimale

2) Conversione in binario della parte razionale del numero decimale

La prima fase siamo già in grado di svilupperla tramite le metodologie imparate fino ad ora.

Per la seconda invece bisognerà usare un nuovo metodo:

-1) Moltiplicare la parte razionale per 2:

(a) se il risultato ottenuto è uguale ad 1, passa al punto 2;

(b) se è maggiore di 1 allora sarà necessario sottrarre 1 al risultato e poi ripeti il punto 1 utilizzando l'attuale risultato parziale come parte frazionaria

(c) se il risultato è minore di 1 ripeti il punto 1 utilizzando l'attuale risultato parziale come parte frazionaria

2) una volta che la parte frazionaria non sia più presente nel risultato parziale sarà necessario riscrivere partendo dall'alto tutte le parti intere dei risultati parziali ottenuti.

Esempio:

Proviamo usando il numero 17.625_{10} :

• la parte intera sarà $17_{10} \rightarrow 010001_2$

• per la parte razionale dovremo fare:

$$1) 0.625 \cdot 2 = 1.25 \quad (1.25 - 1 = 0.25)$$

$$2) 0.25 \cdot 2 = 0.5$$

$$3) 0.5 \cdot 2 = 1$$

Una volta raggiunto l'1, consideriamo partendo dall'alto tutte le parti intere dei risultati, quindi:

.101₂

- alla fine uniremo la parte intera binaria a quella razionale:

$$17.625_{10} = 0.10001.101_2$$

Numeri a virgola mobile

I numeri a virgola fissa sono l'equivalente binario della notazione scientifica dei numeri decimali.

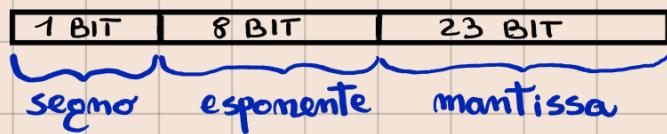
I numeri in notazione scientifica vengono rappresentati nella seguente forma:

$$\pm M \cdot B^E$$

↑ ↑
 mantissa esponente
 base

Lavorando con definite quantità di bit, solitamente i numeri floating-point vengono rappresentati utilizzando 32 bit dove:

- 1° bit rappresenta il segno
- i successivi 8 bit rappresentano l'esponente
- e i restanti 23 bit



Per convertire un numero decimale nella sua forma floating-point, è necessario eseguire questi passaggi:

- 1) Convertire il numero in un binario fixed-point unsigned, visto che il segno sarà rappresentato dal 1° bit;
- 2) riscrivere il numero in "notazione scientifica binaria" portando la virgola fino ad una sola unità

3) riscrivere il segno, la mantissa e l'esponente nei loro campi equivalenti della rappresentazione floating-point

Esempio:

Usando 17.625_{10} che è stato già convertito in fixed point nell'esempio precedente, convertiamolo in floating point:

$$1) 17.625_{10} = 10001.101_2$$

$$2) \frac{10001.101_2}{2^4} = 1.0001101_2 \cdot 2^4$$

3) converto i tre campi necessari in binario per poi riscrivere nei campi rispettivi:

- segno = 0_2 perché il numero è positivo
- esponente = 100_2 che sarebbe 4 rappresentato in binario
- mantissa = 10001101_2

0	00000100	100011010000000000000000
segno	esponente	mantissa

Guardando attentamente il secondo passaggio possiamo notare come qualsiasi numero venga rappresentato in notazione scientifica binaria abbia un 1 come unità a cui viene attribuita la virgola:

- $17.625_{10} = 10001.101 = 1.0001101_2 \cdot 2^4$
- $-58.25_{10} = 111010.01_2 = 1.1101001_2 \cdot 2^5$
- $228_{10} = 11100100_2 = 1.1100100_2 \cdot 2^7$

Per convenzione, quindi, nel momento in cui andiamo a riempire i rispettivi campi del floating-point possiamo direttamente ignorare l'1 prima della virgola, andando a guadagnare un bit aggiuntivo da poter utilizzare per rappresentare le cifre dopo la virgola:

$$17.625_{10} = 10001.101_2 = 1.0001101_2 \cdot 2^4$$

0	00000100	10001101000000000000000000000000
---	----------	----------------------------------

\Rightarrow senza ignorare l'1 davanti la virgola

0	00000100	00011010000000000000000000000000
---	----------	----------------------------------

\Rightarrow ignorando l'1 davanti la virgola

Questo è un passaggio estremamente utile, nel caso in cui dovessimo rappresentare in float un numero in notazione scientifica binaria con 23 bit dopo la virgola, saremmo costretti a dover tagliare il bit meno significativo, poiché non possiamo inserire 24 bit nello spazio di 23 predisposto dalla mantissa.

Lo standard IEEE 754

Ora che abbiamo guadagnato un bit aggiuntivo da poter usare per la precisione della mantissa, resta solo da scoprire come rappresentare i numeri in notazione scientifica binaria con esponente negativo.

Lo standard IEEE 754 propone una soluzione semplice: aggiungiamo un bias di 127 all'esponente.

$$17.625_{10} = 10001.101_2 = 1.0001101_2 \cdot 2^4$$

Una volta portato il numero in questa forma, aggiungiamo 127 all'esponente, per poi convertirlo nel formato floating-point:

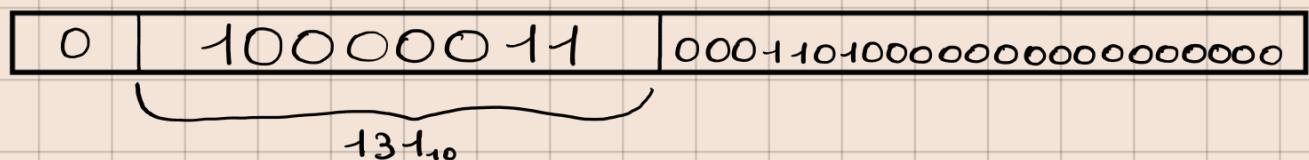
$$1.0001101_2 \cdot 2^{(4+127)} = 1.0001101_2 \cdot 2^{131}$$

Converti 131 in binario:

$$\begin{array}{r}
 131 \\
 | \\
 65 \\
 | \\
 32 \\
 | \\
 16 \\
 | \\
 0
 \end{array}$$

$$\begin{array}{r}
 80 \\
 40 \\
 20 \\
 \hline
 11
 \end{array}
 \quad
 0 = 10000011_2$$

quindi:



Grazie al bias 127 possiamo anche scrivere i numeri con un esponente negativo fino a -127. Tuttavia, perdiamo anche l'uso di una parte di esponenti positivi, poiché il valore massimo rappresentabile in 8 bit è 255.

Casi speciali, tipi di precisione e rappresentazione in esadecimale

Nel caso dei numeri a virgola mobile, sono stati stabiliti dei casi speciali per poter rappresentare dei valori particolari o non esprimibili numericamente:

Numero equivalente	Segno	EspONENTE	Mantissa
0	1 o 0	00000000	000000000000000000000000
∞	0	11111111	000000000000000000000000
$-\infty$	1	11111111	000000000000000000000000
NaN	1 o 0	00000000	diverso da 0
Num. Decimali	1 o 0	00000000	$(-1)^s \cdot 2^{-126} \cdot 0.m \text{ con } m \neq 0$

Inoltre esistono vari tipi di precisione possibili per i numeri float in base al numero di bit utilizzati per rappresentarli:

Tipologia	Bit Tot.	Segno	EspONENTE	Mantissa	Bias
Half-precision	16 bit	1 bit	5 bit	10 bit	15
Single-precision	32 bit	1 bit	8 bit	23 bit	127
Double-precision	64 bit	1 bit	11 bit	52 bit	1023

L'utilità di avere più tipi di precisione è per prevenire la possibilità di un overflow o un **underflow** (ossia quando il numero è troppo piccolo per essere rappresentato) poiché altrimenti sarebbe necessario arrotondare il valore, perdendo una parte di precisione.

Un altro modo per rappresentare dei numeri float è tramite l'uso dei numeri decimali.

Poiché 32 bit corrispondono 8 gruppi da 4 bit, possiamo riscrivere ogni numero float come **8 numeri esadecimali**, per esempio:

$$17.625_{10} = \underbrace{0}_{4} \underbrace{100}_{1} \underbrace{0001}_{8} \underbrace{1000}_{D} \underbrace{1101}_{0} \underbrace{0000}_{0} \underbrace{0000}_{0} \underbrace{0000}_{0} \underbrace{0000}_{0} \text{ float} \rightarrow 0x41800000$$

Operazioni tra numeri floating-point

Per sommare due numeri in rappresentazione floating-point,

il processo necessario è molto semplice:

- 1) Rappresentare nuovamente i due numeri in forma di notazione scientifica binaria
- 2) Portare entrambi i numeri allo stesso espONENTE
- 3) Sommare le due mantisse, normalizzandole se necessario
- 4) Aggiustare l'espONENTE e riscrivere in forma floating-point, arrotondando in caso di overflow o underflow

Esempio:

Somma i due numeri float $0x3FC00000$ e $0x40500000$

$$1) 0x3FC00000 = \underbrace{0}_{\text{segno}} \underbrace{011111}_{\text{espONENTE}} \underbrace{10000000000000000000000000}_{\text{mantissa}} \text{ float} =$$

$$\text{Segno} = 0, \text{espONENTE} = 127, \text{mantissa} = .1$$

Successivamente antepomi -1 quindi: 1.1

Segmento = 0, esponente = 128, mantissa = .101

che diventerà 1.101

3) Comfronta i 2 esponenti:

$127 - 128 = -1$, quindi sposta di -1 la virgola

$$1.1 \gg 1 = 0.11 (\times 2')$$

$$4) 0.11 \cdot 2^1 + 1 \cdot 104 \cdot 2^0 = 10.011 \cdot 2^1$$

$$\begin{array}{r}
 10.110+ \\
 1.101= \\
 \hline
 10.011
 \end{array}$$

$$5) 10.011 \cdot 2^1 \rightarrow 1.0011 \cdot 2^2$$

6) Ricomponi il tutto in formato a zigzag mobile

0	10000001	00110000 0000 0000 0000 0000
---	----------	------------------------------

che in esadecimale sarà: **0x40980000**

Per moltiplicare due numeri float, invece, il processo necessario è leggermente diverso:

1) Rappresentare nuovamente i due numeri in forma di notazione scientifica binaria

2) Sommare i due esponenti.

3) Moltiplicare le due mantisse, normalizzandole se necessario

4) Aggiustare l'esponente e riscrivere in forma floating point, avvolgandolo in caso di overflow o underflow

Esempio :

Calcola il prodotto dei due numeri float 0x3FC00000 e

0x40E0000

1) $0x3fc00000 = 0.01111110000000000000000000000000$ = 1.12⁰

2023-04-14 14:43:31

$$311 \cdot 101 \cdot 2^1 + 11 \cdot 2^0 = 100111 \cdot 2^1$$

$$\begin{array}{r}
 1.101 \times \\
 1.1 \\
 \hline
 1.101 + \\
 \hline
 1.101 \ll = \\
 \hline
 10.0111
 \end{array}$$

4) $10.0111 \cdot 2^1 \rightarrow 1.00111 \cdot 2^2 = \underbrace{0}_{4} \underbrace{100}_{\circ} \underbrace{0000}_{\circ} \underbrace{1001}_{\circ} \underbrace{1100}_{\circ} \underbrace{0000}_{\circ} \underbrace{0000}_{\circ} \underbrace{0000}_{\circ} \underbrace{0000}_{\circ}$