Rappresentazione dell'informazione

L' informazione si presenta sotto vari aspetti: testi, numeri interi e reali con segno e senza segno, immagini, suoni, etc.
In un calcolatore essa viene rappresentata facendo uso di tecnologia digitale, ovvero di soltanto 2 possibili valori di tensione elettrica {alto, basso}, logicamente associati alle cifre 0 ed 1. Ciò implica che ogni informazione viene trasformata nel calcolatore in una sequenza di 0 e 1.

1. Codifica dei numeri

Normalmente, per la rappresentazione dei numeri facciamo uso del sistema di numerazione arabico**,** introdotto in Europa dagli arabi nel Medio Evo. Esso è caratterizzato dalle seguenti proprietà:

- *è decimale* (o in *base 10*): poichè rappresenta i numeri tramite sequenze di cifre che vanno da 0 a 9, ovvero come sequenze di cifre prese dall'insieme D = {0,1,2,…..,9} a cardinalità 10;
- *è posizionale pesato*: poiché ogni cifra ha un *peso* diverso a seconda della posizione che occupa nella sequenza.

Ad esempio, il valore rappresentato dalla sequenza di cifre decimali 3333 è:

$$3 \times 10^3 + 3 \times 10^2 + 3 \times 10^1 + 3 \times 10^0$$

ovvero la somma di 3 unità (peso 10^0), 3 decine (peso 10^1) , 3 centinaia (peso 10^2) e 3 migliaia (peso 10^3). E' evidente da questo esempio che la cifra 3, pur essendo la stessa in tutte le posizioni, contribuisce al valore complessivo del numero con *pesi* diversi, in dipendenza della sua posizione nella sequenza. E' altrettanto evidente che nella sequenza le posizioni sono indicizzate nel verso che va da destra *(cifra meno significativa, LSD)* verso sinistra *(cifra più significativa, MSD)*.
I vantaggi derivanti dall'uso dei sistemi posizionali pesati sono fondamentalmente legati alla possibilità di rappresentare numeri molto grandi con un numero limitato di cifre e di svolgere su di essi calcoli in maniera molto efficiente. I sistemi posizionali pesati possono utilizzare come *base* un qualsiasi numero strettamente maggiore di 1 e sono caratterizzabili attraverso una *base b* > 1 ed un *alfabeto*.

- L'*alfabeto* e' l'insieme delle cifre disponibili per la rappresentazione dei numeri. Ad ogni cifra corrisponde un valore compreso tra 0 e (b-1).

Ad esempio, nel sistema di numerazione decimale l'alfabeto e` D = {0,1,2,…..,9}.

- La *base* (b) e` la cardinalità dell'alfabeto, ovvero il numero di cifre presenti nell'alfabeto.

Ad esempio, nel sistema di numerazione decimale b=10.
In generale, un numero naturale N (ovvero appartenente a $\mathbb{N}$={0,1,2…..}) può essere rappresentato in un sistema posizionale pesato qualsiasi con base *b* > 1. Infatti, fissato b ed il relativo alfabeto A = {0,1,2,….b-1}, la sequenza di n cifre

$a_{n-1}\, a_{n-2}\, \ldots\, a_1\, a_0$ con $a_i \in A\ \forall i = 0, 1, 2, \ldots\ldots n-1$

rappresenta il numero

$$N = a_{n-1}\times b^{n-1} + a_{n-2}\times b^{n-2} + \ldots + a_1\times b^1 + a_0\times b^0 = \sum_{i=0}^{n-1} a_i \times b^i$$

con a_0 cifra meno significativa (LSD) e a_{n-1} cifra più significativa (MSD).
Le basi più comunemente usate sono:

- Base binaria: b = 2, B = {0,1}; i simboli di B prendono il nome di bit, acronimo di binary digit (cifra binaria)
- Base ottale: b = 8, O = {0,1,2,3,4,5,6,7}
- Base esadecimale: b = 16, H = {0,1,…,9,A,B,C,D,E,F}, dove A vale 10, B vale 11,…, F vale 15

Esempi

$(101)_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 4 + 0 + 1 = (5)_{10}$

$(205)_8 = 2 \times 8^2 + 0 \times 8^1 + 5 \times 8^0 = 128 + 5 = (133)_{10}$

$(2F)_{16} = 2 \times 16^1 + 15 \times 16^0 = 32 + 15 = (47)_{10}$

Ricordiamo che per dimostrare la veridicità di una proposizione di tipo matematico, è spesso utile il principio di induzione di seguito riportato.

Principio di induzione

La proposizione si dimostra vera per induzione tramite una *base induttiva* ed un *passo induttivo*.
(a)*Base induttiva*: la proposizione è vera per n = 1;
(b) *Passo induttivo*: supponendo che la proposizione sia vera per n, verifichiamo che essa è vera per n + 1.
Essendo n qualsiasi, la proposizione è valida per tutti i numeri naturali.

Il principio si enuncia per proposizioni che asseriscono che qualcosa è vera per ogni numero naturale e, per poterlo applicare, è necessario dimostrare due cose: prima di tutto che la proposizione in questione è vera per n = 1, in secondo luogo che, se la proposizione è vera per il numero n che precede un qualsiasi numero n + 1, allora essa è vera per il numero n + 1 stesso. In queste circostanze potremo concludere che la proposizione è vera per ogni numero naturale.

Per induzione è possibile dimostrare che per una base b ed un numero n di cifre:

1. rappresentiamo b^n numeri naturali distinti,
2. rappresentiamo come valore minimo 0 (quando tutti gli a_i sono uguali a 0) e come valore massimo b^n-1 (quando tutti gli a_i sono uguali a b-1).

Per semplicità, e senza ledere di generalità, prendiamo b=2.

Dimostrazione 1.
Ipotesi: b=2, n numero di bit
Tesi: rappresentiamo 2^n numeri naturali distinti

(a)*Base induttiva*: n = 1 implica $2^n = 2^1 = 2$ ed, infatti, avendo un unico bit possiamo rappresentare soltanto due numeri distinti, 0 ed 1.
(b) *Passo induttivo*: supponendo che con n bit rappresentiamo 2^n distinti numeri, chiediamoci quanti ne rappresentiamo quando agli n bit ne aggiungiamo uno ulteriore. Di fatto, aggiungere un bit significa considerare i 2^n numeri scritti con n bit e riscriverli, una prima volta, aggiungendo uno 0 in una qualsiasi posizione e, una seconda volta, aggiungendo un 1 in una qualsiasi posizione. In

totale, scriviamo $2^n + 2^n = 2^{n+1}$ distinti numeri. Essendo n qualsiasi, la proposizione è valida per tutti i numeri naturali. (c.v.d.)

Dimostrazione 2.
Ipotesi: b=2, n numero di bit
Tesi: il massimo valore rappresentabile è $2^n - 1$

(a)*Base induttiva*: n = 1 implica $2^n - 1 = 2^1 - 1 = 1$ ed, infatti, avendo un unico bit, possiamo rappresentare soltanto due numeri distinti, 0 ed 1, di cui il valore massimo è banalmente 1.
(b) *Passo induttivo*: supponendo che con n bit il massimo valore rappresentabile è $2^n - 1$, chiediamoci quanto vale il valore massimo rappresentabile quando agli n bit ne aggiungiamo uno ulteriore. Di fatto, aggiungere un bit significa considerare la stringa costituita da n bit tutti uguali a 1 ed aggiungere l'n+1esimo bit, ancora uguale a 1, in una qualsiasi posizione. Il risultato di questa operazione è l'aggiunta di un bit di peso 2^n. In totale, il valore massimo rappresentabile su n +1 bit sarà $2^n + 2^n - 1 = 2^{n+1} - 1$. Essendo n qualsiasi, la proposizione è valida per tutti i numeri naturali. (c.v.d.)

Esempi
b=2 ed n= 5 implicano che posso rappresentare 2^5 numeri naturali distinti, che vanno da un minimo di 0 ad un massimo di 2^5-1.
b=10 ed n=6 implicano che posso rappresentare 10^6 numeri naturali distinti, che vanno da un minimo di 0 ad un massimo di 10^6-1.

Fissata una base b e dato un numero N, il numero minimo n di cifre necessarie alla rappresentazione di N in base b deve essere tale che $b^n - 1 \geq N$, dove $b^n - 1$ è il più grande numero rappresentabile con n cifre. Segue che:

$$n = \lceil \log_b (N+1) \rceil$$

Esempi
b=2 ed N=10 implicano n = parte intera superiore $\lceil (\log_2 (10+1) \rceil = 4$; infatti, con 3 bit rappresentiamo i numeri da 0 a 7, mentre con 4 bit rappresentiamo i numeri da 0 a 15.
b=8 ed N=25 implicano $n = \lceil \log_8 (25+1) \rceil = 2$; infatti, con 1 cifra ottale rappresentiamo i numeri da 0 a 7, mentre con 2 cifre ottali rappresentiamo i numeri da 0 a 72.

2. Algoritmo di conversione di un numero naturale dalla base 10 alla base 2

- Dato N_{10} , la sua rappresentazione posizionale pesata in base 2 si ottiene utilizzando il seguente algoritmo basato sulla divisione intera (quoziente Q e resto R) per 2:

  - ✓ $N_{10} = Q_0 \times 2 + R_0$ *(N diviso 2 con resto R_0)*
  - ✓ $Q_0 = Q_1 \times 2 + R_1$
  - ✓ .....
  - ✓ $Q_{n-2} = Q_{n-1} \times 2 + R_{n-1}$

  fino a che il quoziente Q_{n-1} diventa 0

Porre

$$N_2 = R_{n-1} R_{n-2} \ldots R_1 R_0$$

E' opportuno notare che l'algoritmo fornisce la sequenza di bit dall'LSD all' MSD e termina non appena si ottiene un quoziente pari a 0 (continuando si genererebbe solo una sequenza di zeri non significativi alla sinistra dell'MSD).

Esempio: trovare la rappresentazione binaria di N_{10}=44
Essendo $\lceil \log_2 (N+1) \rceil = \lceil \log_2 (45) \rceil = 6$, la rappresentazione richiederà 6 bit. Infatti:

$44 = Q_0 \times 2 + R_0 = 22 \times 2 + 0$

$Q_0 = 22 = Q_1 \times 2 + R_1 = 11 \times 2 + 0$

$Q_1 = 11 = Q_2 \times 2 + R_2 = 5 \times 2 + 1$

$Q_2 = 5 = Q_3 \times 2 + R_3 = 2 \times 2 + 1$

$Q_3 = 2 = Q_4 \times 2 + R_4 = 1 \times 2 + 0$

$Q_4 = 1 = Q_5 \times 2 + R_5 = 0 \times 2 + 1$

quindi: $44_{10} = 101100_2$

L'algoritmo divisivo di conversione decimale-binario, risolve anche il problema delle conversioni decimale-ottale, decimale-esadecimale e decimale-r-ario (con r qualsiasi, purchè maggiore di 1), utilizzando come divisore rispettivamente 8, 16 ed r.

2. Algoritmo di conversione di un numero naturale dalla base 2 alla base 10

- Dato $N_2 = a_{n-1}\, a_{n-2} \ldots a_1\, a_0$, la sua rappresentazione posizionale pesata in base 10 si ottiene utilizzando il seguente algoritmo basato sulla moltplicazione per 2:

  - ✓ $S_{n-1} = a_{n-1}$
  - ✓ $S_{n-2} = a_{n-2} + 2 \times S_{n-1}$
  - ✓ .....
  - ✓ $S_0 = a_0 + 2 \times S_1$

  Porre

  $N_{10} = S_0$

E' opportuno notare che l'algoritmo scandisce la sequenza di bit dall'MSD all' LSD.

Esempio: trovare la rappresentazione decimale di N_2=1101001
In questo caso il numero è rappresentato su n=7 bit e, pertanto, l'algoritmo incomincia a calcolare le somme parziali S_i da n-1 = 6.

$S_6 = a_6 = 1$

$S_5 = a_5 + 2 \times S_6 = 1 + 2 = 3$

$S_4 = a_4 + 2 \times S_5 = 0 + 6 = 6$

$S_3 = a_3 + 2 \times S_4 = 1 + 12 = 13$

$S_2 = a_2 + 2 \times S_3 = 0 + 26 = 26$

$S_1 = a_1 + 2 \times S_2 = 0 + 52 = 52$

$S_0 = a_0 + 2 \times S_1 = 1 + 104 = 105$

$N_{10} = S_0 = 105$

L'algoritmo per la conversione binario-decimale calcola, attraverso una successione di passi elementari (moltiplicazione per 2 e somma di un bit), la somma di potenze derivante dalla definizione di sistema posizionale pesato. Infatti, sappiamo che, applicando tale definizione alla parola $N_2 = 1101001_2$, il corrispondente valore in base 10 può essere calcolato tramite la somma di potenze:

$$N_{10} = \sum_{i=0}^{6} a_i \times 2^i = 2^6 + 2^5 + 2^3 + 2^0 \qquad (1)$$

Riprendiamo, ora, l'algoritmo ed esplicitiamo formalmente tutti i conti per capire quale sia il contenuto esatto di S_0.

$S_6 = a_6$

$S_5 = a_5 + 2 \times S_6 = a_5 + 2a_6$

$S_4 = a_4 + 2 \times S_5 = a_4 + 2(a_5 + 2\,a_6) = a_4 + 2\,a_5 + 2^2\,a_6$

$S_3 = a_3 + 2 \times S_4 = a_3 + 2(a_4 + 2a_5 + 2^2\,a_6) = a_3 + 2a_4 + 2^2\,a_5 + 2^3\,a_6$

$S_2 = a_2 + 2 \times S_3 = a_2 + 2(a_3 + 2a_4 + 2^2\,a_5 + 2^3\,a_6) = a_2 + 2a_3 + 2^2\,a_4 + 2^3\,a_5 + 2^4\,a_6$

$S_1 = a_1 + 2 \times S_2 = a_1 + 2(a_2 + 2a_3 + 2^2\,a_4 + 2^3\,a_5 + 2^4\,a_6) = a_1 + 2a_2 + 2^2\,a_3 + 2^3\,a_4 + 2^4\,a_5 + 2^5\,a_6$

$S_0 = a_0 + 2 \times S_1 = a_0 + 2(a_1 + 2a_2 + 2^2\,a_3 + 2^3\,a_4 + 2^4\,a_5 + 2^5\,a_6) =$

$= a_0 + 2a_1 + 2^2\,a_2 + 2^3\,a_3 + 2^4\,a_4 + 2^5\,a_5 + 2^6\,a_6$

che è proprio la somma (1) poiché a_4, a_2 ed a_1 valgono 0 in N_2 .
L'algoritmo moltiplicativo di conversione binario-decimale, risolve anche il problema delle conversioni ottale-decimale, esadecimale-decimale e r-ario-decimale (con r qualsiasi, purchè maggiore di 1), utilizzando come moltiplicatore rispettivamente 8, 16 ed r.

3. Conversioni da rappresentazione binaria a rappresentazioni ottale ed esadecimale.

Le rappresentazioni ottali ed esadecimali consentono algoritmi molto semplici di conversione dalla base 2 e verso la base 2.
Ricordando che $8 = 2^3$ e $16 = 2^4$, la conversione da binario a ottale (da binario a esadecimale) si ottiene suddividendo, a partire dall'LSD, il numero binario in triple (quadruple) di bit e calcolando per ciascuna di esse la corrispondente cifra ottale (esadecimale).

Esempio: dato 101110_2 , calcolarne l'equivalente ottale ed esadecimale.
Per ricavare la rappresentazione ottale è necessario suddividere la rappresentazione binaria in triple partendo dall'LSD, ovvero:
$101110_2 \rightarrow 101_2 \mid 110_2$
ora: $101_2 = 5_8$ e $110_2 = 6_8$
da cui segue che $101110_2 = 56_8$

Per ricavare la rappresentazione esadecimale è necessario suddividere la rappresentazione binaria in quadruple partendo dall'LSD , ovvero:
$101110_2 \rightarrow 10_2 \mid 1110_2$
ora: $10_2 = 2_{16}$ e $1110_2 = E_{16}$
da cui segue che $101110_2 = 2E_{16}$

4. Rappresentazione dei numeri interi

L'insieme **Z** dei numeri interi è formato dall'unione dei numeri naturali (0, 1, 2, ...) e dei numeri negativi (-1, -2, -3,...). Tra le infinite possibili rappresentazioni dei numeri interi quelle di maggiore interesse sono le due seguenti:

- rappresentazione in modulo e segno;
- rappresentazione in complemento a due.

Queste rappresentazioni si possono adottare con qualsiasi base perché le proprietà su cui si basano sono generali. Noi ci riferiremo alla base 2.

Nella rappresentazione in modulo e segno, data una sequenza di n bit $a_{n-1}\ a_{n-2}\ \ldots\ a_1\ a_0$, il bit più significativo, a_{n-1}, viene riservato alla rappresentazione del segno, associando, di solito, al segno "più" il bit 0 ed al segno "meno" il bit 1, mentre la rimanente porzione della sequenza, $a_{n-2}\ \ldots\ a_1\ a_0$, viene riservata alla rappresentazione del valore assoluto del numero in accordo con le regole del sistema posizionale pesato.

Esempio: per n= 5 bit, le sequenze 01100 e 11100 in modulo e segno rappresentano, rispettivamente, +12 e -12.

E' evidente che con la notazione in modulo e segno su n bit è possibile rappresentare 2^{n-1} numeri non negativi ed altrettanti negativi. Infatti, poichè la porzione numerica della sequenza assume valori compresi tra 0 e $2^{n-1}-1$, rappresentiamo tutti i numeri compresi nell'intervallo:

$$[\ -(2^{n-1}-1),\ldots\ldots -0,\ +0,\ldots..+(2^{n-1}-1)]$$

in cui lo 0 viene rappresentato sia come negativo (tutti 0 con l'MSD ad 1), sia come non negativo (tutti tutti 0 con l'MSD ad 0).

Esempio: per n = 6, l'intervallo di rappresentabilità dei numeri interi in modulo e segno è:

$$[\ -(2^{5}-1),\ldots\ldots -0,\ +0,\ldots..+(2^{5}-1)]$$

cui, corrisponde l'insieme delle sequenze binarie {111111, 111110,……..100000, 000000, 000001,……011111}

Gli algoritmi di conversione decimale-binario e binario-decimale per la notazione in modulo e segno sono ovviamente banali.

La notazione in modulo e segno, tuttavia, non risulta efficiente per le operazioni aritmetiche. Pensando, infatti, ad un'unità logico-aritmetica (ALU) progettata per effettuare le operazioni aritmetiche (somma e sottrazione) con numeri rappresentati in modulo e segno, avremmo necessità di due sottosistemi diversi, un addizionatore e un sottrattore, da scegliere in relazione all'operazione desiderata ed ai segni degli operandi, come illustrato nella tabella seguente.

	Stesso segno degli operandi	Segno opposto degli operandi
Addizione	Addizionatore	Sottrattore
Sottrazione	Sottrattore	Addizionatore

Più precisamente, le somme algebriche vanno eseguite in modo diverso a seconda del segno concorde o discorde degli operandi, richiedendo una serie di decisioni logiche dipendenti dai dati in input.

Una rappresentazione alternativa che supera i suddetti problemi è la notazione in complemento a 2. Essa consente di:

- utilizzare per la costruzione dell'ALU un solo sottosistema aritmetico: l'addizionatore;
- evitare di controllare i segni degli operandi tutte le volte che si deve effettuare un'addizione o una sottrazione;
- evitare la duplice rappresentazione dello 0.

Un intero rappresentato in complemento a 2 si definisce come segue.

Data una sequenza $a_{n-1}\ a_{n-2}\ \ldots\ a_1\ a_0$ di n bit , a_{n-1} è il bit di segno e $a_{n-2}\ \ldots\ a_1\ a_0$ la porzione numerica. Il valore dell' intero N_{10} e':

$$N_{10} = -\ a_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} a_i \times 2^i \qquad (2)$$

ovvero, mentre alle posizioni dei bit nella porzione numerica viene dato lo stesso peso della notazione in modulo e segno, al bit di segno a_{n-1} viene dato il peso negativo -2^{n-1}. Cosicchè, se $a_{n-1} = 0$, allora N_{10} è non negativo (≥ 0); se $a_{n-1} = 1$, allora N_{10} è negativo (< 0). Infatti:

$$a_{n-1} = 0 \quad \rightarrow \quad N_{10} = \sum_{i=0}^{n-2} a_i \times 2^i \geq 0$$

e

$$a_{n-1} = 1 \quad \rightarrow \quad N_{10} = -2^{n-1} + \sum_{i=0}^{n-2} a_i \times 2^i < 0$$

Poichè $\sum_{i=0}^{n-2} a_i \times 2^i$ varia tra un valore minimo che è 0 (quando tutti gli a_i sono uguali a 0) ed un valore massimo che è $2^{n-1} - 1$ (quando tutti gli a_i sono uguali a 1), segue anche che l'intervallo di rappresentabilità per il complemento a 2 su n bit è:

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

nel quale lo 0 è chiaramente compreso una volta soltanto.

Esempio: per n = 6 bit
110011 è l'equivalente di -13 poichè $N_{10} = -2^5 + 19$
010011 è l'equivalente di +19 poichè $a_5 = 0$ e, quindi, il valore del numero è definito solo dalla porzione numerica.

Una utile proprietà dei numeri rappresentati in complemento a 2 è la cosiddetta *estensione in segno*. Se abbiamo un numero rappresentato in complemento a 2 su n bit ed abbiamo la necessità che esso sia scritto su n+m bit, basta semplicemente replicare il bit di segno m volte partendo dalla posizione n fino alla m-1esima.

Esempio: sia 1100 un numero in complemento a 2 su quattro bit. Scrivere la sua estensione in segno su 8 bit.
1100→ 11111100

5. Algoritmo di ricerca dell'opposto di un numero intero rappresentato in complemento a 2

Dato un numero intero N rappresentato in complemento a 2, possiamo trovare la rappresentazione del suo opposto, - N, con un semplice algoritmo costituito dai seguenti due passi.

1. Complemento bit a bit (0→1 e 1→0) della sequenza binaria che rappresenta il numero assegnato N
2. Somma di 1 alla sequenza ottenuta al passo 1

Esempio 1: supponendo che 011001 sia una sequenza in complemento a due, trovare il valore che essa rappresenta e la rappresentazione del suo opposto in complemento a 2.
Dalla definizione di notazione in complemento a 2 ricaviamo il valore di 011001:

$$N_{10} = - a_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} a_i \times 2^i = - a_5 \times 2^5 + \sum_{i=0}^{4} a_i \times 2^i = 0 \times 64 + S_0 = 0 + 25 = + 25$$

Applichiamo , ora, l'algoritmo di ricerca dell'opposto:

Passo 1: 011001 complementato bit a bit diventa 100110
Passo 2: 100110 sommato ad 1 diventa 100111, che è proprio la rappresentazione di -25 in complemento a due. Infatti:

$$N_{10} = - a_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} a_i \times 2^i = - a_5 \times 2^5 + \sum_{i=0}^{4} a_i \times 2^i = - 1 \times 32 + 7 = - 32 + 7 = - 25$$

Esempio 2: supponendo che 110001 sia una sequenza in complemento a due, trovare il valore che essa rappresenta e la rappresentazione del suo opposto in complemento a 2.
Dalla definizione di notazione in complemento a 2 ricaviamo il valore di 110001:

$$N_{10} = - a_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} a_i \times 2^i = - a_5 \times 2^5 + \sum_{i=0}^{4} a_i \times 2^i = - 1 \times 32 + 17 = - 32 + 17 = - 15$$

Applichiamo , ora, l'algoritmo di ricerca dell'opposto:

Passo 1: 110001 complementato bit a bit diventa 001110

Passo 2: 001110 sommato ad 1 diventa 001111, che è proprio la rappresentazione di + 15 in complemento a due. Infatti:

$$N_{10} = - a_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} a_i \times 2^i = - a_5 \times 2^5 + \sum_{i=0}^{4} a_i \times 2^i = 0 \times 32 + 15 = 0 + 15 = + 15$$

6. Algoritmi di conversione decimale-binario e binario-decimale per la notazione in complemento a 2

Dato un intero non negativo, gli algoritmi per le relative conversioni decimale-binario e binario-decimale nella notazione in complemento a 2 sono banali.
Dato un intero negativo, la conversione binario-decimale può essere fatta utilizzando l'algoritmo di conversione binario-decimale per i numeri naturali e la definizione (2).

Esempio: sapendo che 1000101 rappresenta un numero in complemento a due, trovarne l'equivalente in notazione decimale.
Poiché il bit di segno a_6 vale 1, sappiamo che la sequenza $a_6a_5a_4a_3a_2a_1a_0$ rappresenta un numero negativo. Convertiamo la porzione numerica $a_5a_4a_3a_2a_1a_0$ usando l'algoritmo di conversione binario-decimale per i naturali:

$S_5 = a_5 = 0$
$S_4 = a_4 + 2 \times S_5 = 0 + 0 = 0$
$S_3 = a_3 + 2 \times S_4 = 0 + 0 = 0$
$S_2 = a_2 + 2 \times S_3 = 1 + 0 = 1$
$S_1 = a_1 + 2 \times S_2 = 0 + 2 = 2$
$S_0 = a_0 + 2 \times S_1 = 1 + 2 = 3$

valore della porzione numerica = 3

Applichiamo la definizione (2):

$$N_{10} = - a_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} a_i \times 2^i = - a_6 \times 2^6 + \sum_{i=0}^{5} a_i \times 2^i = - 1 \times 64 + S_0 = -64 + 3 = - 61$$

Dato un intero negativo -N, la conversione decimale-binario può essere fatta utilizzando, prima, l'algoritmo di conversione decimale-binario per i numeri naturali applicato all'intero senza segno N e, successivamente, l'algoritmo per la ricerca dell'opposto di un numero in complemento a due.

Esempio: trovare la rappresentazione in complemento a 2 di - 18 .
Poiché $-2^5 < -18 < 2^5 - 1$, l'intero negativo di cui vogliamo trovare la rappresentazione in complemento a 2 rientra nell'intervallo di rappresentabilità con n =6 bit.
Utilizziamo l'algoritmo di conversione decimale-binario per trovare la rappresentazione del numero naturale 18 :

$18 = Q_1 \times 2 + R_1 = 9 \times 2 + 0$

$Q_1 = 9 = Q_2 \times 2 + R_2 = 4 \times 2 + 1$
$Q_2 = 4 = Q_3 \times 2 + R_3 = 2 \times 2 + 0$
$Q_3 = 2 = Q_4 \times 2 + R_4 = 1 \times 2 + 0$
$Q_4 = 1 = Q_5 \times 2 + R_5 = 0 \times 2 + 1$

Quindi: $18 \rightarrow 10010_2 \rightarrow 010010_2$ (abbiamo aggiunto uno 0 non significativo alla sinistra della sequenza ottenuta poiché la rappresentazione in complemento a 2 richiede 6 bit)

La sequenza 010010_2 può essere interpretata come la rappresentazione in complemento a due di +18. Ciò consente di applicare ad essa l'algoritmo di ricerca dell'opposto per trovare la rappresentazione di – 18.
Passo 1: 010010 complementato bit a bit diventa 101101
Passo 2: 101101 sommato ad 1 diventa 101110, che è proprio la rappresentazione di - 18 in complemento a due. Infatti:

$$N_{10} = - a_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} a_i \times 2^i = - a_5 \times 2^5 + \sum_{i=0}^{4} a_i \times 2^i = -1 \times 32 + 14 = -32 + 14 = -18$$

7. Aritmetica Binaria

La somma di due interi binari senza segno si basa sulla seguente tavola pitagorica dell' addizione binaria:

+	0	1
0	0	1
1	1	10

nella quale l'elemento 10 (1+1=10) va inteso come somma 0 con riporto di 1.
Possiamo effettuare l'operazione di addizione di due numeri binari ad n bit, $A = a_{n-1}\, a_{n-2} \ldots a_1\, a_0$ e $B = b_{n-1}\, b_{n-2} \ldots b_1\, b_0$ incolonnando gli operandi e facendo l'operazione di somma bit a bit in accordo con le regole della tavola pitagorica di cui sopra. In ogni posizione i, con i = 0, 1,…..n-1, abbiamo come input i bit a_i di A, b_i di B e c_i che abbiamo ottenuto dall'addizione nella posizione i-1 e calcoliamo come output il bit di somma s_i ed il bit di riporto c_{i+1}.

$$\begin{array}{l} \boxed{c_n}\, c_{n-1}\, c_{n-2} \ldots\ldots c_1 \boxed{c_0} \\ \quad a_{n-1}\, a_{n-2} \ldots a_1\, a_0 + \\ \quad b_{n-1}\, b_{n-2} \ldots b_1\, b_0 = \\ \hline s_{n-1}\, s_{n-2} \ldots s_1\, s_0 \end{array}$$

c_0 vale 0; c_n non deve essere considerato ai fini del risultato della somma in quanto, avendo scelto di rappresentare i numeri con n bit, anche il risultato della somma deve essere rappresentato su n bit. Tuttavia, ciò non è sempre possibile poiché il valore assoluto della somma ovviamente cresce rispetto ai valori assoluti degli operandi e potrebbe assumere un valore che non rientra nell'intervallo di rappresentabilità con n bit. Questa condizione, nota come *overflow*, conduce ad un risultato scorretto della somma. L'osservazione di c_n, congiuntamente con c_{n-1}, ci da', comunque, informazioni sulla correttezza o scorrettezza dell'operazione; in particolare, se c_n e c_{n-1}

sono uguali tra di loro l'operazione è corretta, mentre se c_n e c_{n-1} sono diversi l'operazione è scorretta.
La tavola pitagorica dell'addizione binaria si riferisce ad operandi costituiti da un solo bit. Possiamo scrivere la tavola dell'addizione binaria per operandi ad n bit come segue:

a_i	b_i	c_i	s_i	c_{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

intendendo che, $\forall i = 0, 1,2,\ldots\ldots n-1$, effettuiamo, ordinatamente dal bit meno significativo al più significativo, la somma bit a bit con riporto.

Esempio 1: calcolare le somma tra A = 01000 e B = 00110, naturali rappresentati su n =5 bit.

011100 *riporti*
01010 + operando A
00110 = operando B
10000 somma

} operazione corretta: il valore della somma rientra nell'intervallo di rappresentabilità con 5 bit

Esempio 2: calcolare le somma tra A = 01000 e B = 00110, naturali rappresentati su n =5 bit.

111100 *riporti*
11010 + operando A
01110 = operando B
01000 somma

} operazione scorretta: il valore della somma non rientra nell'intervallo di rappresentabilità con 5 bit

Abbiamo finora analizzato il problema dell'addizione pensando ai numeri naturali. Se rivolgiamo , ora, l'attenzione agli interi, non soltanto non dobbiamo modificare quanto affermato per l'addizione, ma non dobbiamo neanche affrontare la questione della sottrazione poiché la notazione in complemento a 2 permette di trattare quest'ultima operazione come una somma tra numeri di segno opposto. Infatti:

$$(A - B) = (A + (-B))$$

ovvero la notazione in complemento a 2 ci consente di utilizzare per la costruzione dell'ALU un solo sottosistema aritmetico, l'addizionatore, senza alcun bisogno di controllare i segni degli operandi.

Esempio 1: calcolare le somma tra A = 0001100 e B = 0011110, interi rappresentati con n = 7 bit.

```
 0 0111000   riporti
   0001100 + operando A
   0011110 = operando B
   0101010   somma
```

operazione corretta: $c_n = c_{n-1} = 0$

Esempio 2: calcolare le somma tra A = 1101100 e B = 1100010, interi rappresentati con n = 7 bit.

```
 1 1100000   riporti
   1101100 + operando A
   1100010 = operando B
   1101110   somma
```

operazione corretta: $c_n = c_{n-1} = 1$

Esempio 3: calcolare le somma tra A = 0101100 e B = 0111110, interi rappresentati con n = 7 bit.

```
 0 1111000   riporti
   0101100 + operando A
   0111110 = operando B
   1101010   somma
```

operazione scorretta: $c_n = 0 \neq c_{n-1} = 1$

Esempio 4: calcolare le somma tra A = 1001100 e B = 1111110, interi rappresentati con n = 7 bit.

```
 1 0111000   riporti
   1001100 + operando A
   1011110 = operando B
   0101010   somma
```

operazione scorretta: $c_n = 1 \neq c_{n-1} = 0$

Esempio 5: calcolare le somma tra A = 0001100 e B = 1111110, interi rappresentati con n = 7 bit.

```
 0 0111000   riporti
   0001100 + operando A
   1011110 = operando B
   1101010   somma
```

operazione corretta: $c_n = c_{n-1} = 0$

Esempio 6: calcolare le somma tra A = 1111111 e B = 0111110, interi rappresentati con n = 7 bit.

```
 1 1111100   riporti
   1111111 + operando A
   0011110 = operando B
   0011101   somma
```

operazione corretta: $c_n = c_{n-1} = 1$

8. Frazioni proprie

Sia F < 1 una frazione propria. La rappresentazione binaria sarà costituita da una sequenza di m bit $a_{-1}\ a_{-2}\ .....a_{-m}$ legata ad F_{10} dalla relazione:

$$F = a_{-1} \times 2^{-1} + a_{-2} \times 2^{-2} + ... a_{-m} \times 2^{-m} = \sum_{i=-1}^{-m} a_i \times 2^i =$$

Possiamo determinare l'intervallo di rappresentabilità delle frazioni proprie su m bit come segue. Il minimo rappresentabile vale 0 e si avrà, ovviamente, quando $a_i = 0$, $\forall i = -1,.....-m$. Il massimo rappresentabile vale $1 - 2^{-m}$ e si avrà quando $a_i = 1$, $\forall i = -1,.....-m$. Infatti:

$$F_{max} = 2^{-1} + 2^{-2} + ... 2^{-m} = (2^{m-1} + 2^{m-2} + ... + 2^0)/ 2^m = (2^m - 1)/ 2^m = 1 - 2^{-m}$$

Quindi, l'intervallo di rappresentabilità delle frazioni proprie su m bit è:

$$[\ 0,\ \ 1 - 2^{-m}]$$

Un numero rappresentato in base 2 con n + m bit, di cui n riservati alla parte intera ed m alla parte frazionaria, avrà dunque valore:

$$N_{10} = a_{n-1} \times 2^{n-1} + a_{n-2} \times 2^{n-2} + ... + a_2 \times 2^2 + a_1 \times 2^1 + a_0 \times 2^0 + a_{-1} \times 2^{-1} +$$

$$+ a_{-2} \times 2^{-2} + ...+ + a_{-m} \times 2^{-m}$$

Questo tipo di rappresentazione viene detto in virgola fissa, poiché, dato il numero complessivo di bit, m + n, che si hanno a disposizione, viene stabilito a priori quanti assegnarne alla rappresentazione della parte intera del numero e quanti alla parte decimale. Essa è molto semplice da utilizzare, ma presenta il grosso svantaggio di precludere la rappresentazione di numeri molto grandi e molto piccoli. Per le conversioni, la parte frazionaria si tratta separatamente dalla parte intera.

Algoritmo di conversione decimale-binario della parte frazionaria

- Data la frazione propria F, la sua rappresentazione posizionale pesata in base 2 si ottiene utilizzando il seguente algoritmo basato sulla moltiplicazione per 2:
  - ✓ $F_0 = F$
  - ✓ $2F_0 = a_{-1} + F_{-1}$
  - ✓ $2F_{-1} = a_{-2} + F_{-2}$
  - ✓ **.....**
  - ✓ $2F_{-i} = a_{-i} + F_{-(i+1)}$
  - ✓ .....
  - ✓ .....

fino a che non viene raggiunto lo 0 come parte periodica, una parte periodica qualsiasi , o il numero m di bit prestabilito per la rappresentazione della parte frazionaria.

Porre

$F_2 = a_{-1}\, a_{-2} \ldots a_{-m}$

Esempio: siano F = .125 e G = .48 due frazioni proprie rappresentate in base 10. Supponendo di avere a disposizione 4 bit per la rappresentazione della parte frazionaria, gli algoritmi di codifica decimale-binario sono rispettivamente i seguenti:

$F_0 = F = .125$
$2 \times F_0 = a_{-1} + F_{-1} = 2x(.125) = 0 + .250$
$2 \times F_{-1} = a_{-2} + F_{-2} = 2x(.250) = 0 + .500$
$2 \times F_{-2} = a_{-3} + F_{-3} = 2x(.500) = 1 + .000$

Essendo stata raggiunta come parte periodica lo 0, l'algoritmo si ferma; segue che $F_2 = .0010$ poiché per la rappresentazione della parte frazionaria sono stati riservati quattro bit.

$G_0 = G = .48$
$2 \times G_0 = a_{-1} + G_{-1} = 2x(.48) = 0 + .96$
$2 \times G_{-1} = a_{-2} + G_{-2} = 2x(.96) = 1 + .92$
$2 \times G_{-2} = a_{-3} + G_{-3} = 2x(.92) = 1 + .84$
$2 \times G_{-3} = a_{-4} + G_{-4} = 2x(.84) = 1 + .68$

L'algoritmo si ferma poichè non è stata raggiunta alcuna parte periodica, ma è stato raggiunto il numero di bit riservato alla rappresentazione della parte frazionaria. Segue che $G_2 = .0111$

È opportuno osservare che la rappresentazione decimale di una frazione propria e la relativa rappresentazione binaria hanno precisioni diverse (cfr. G e G_2); si dimostra che, affinché le precisioni siano confrontabili, è necessario che la rappresentazione binaria di una frazione sia espressa con circa tre volte il numero di cifre della sua controparte decimale.

Algoritmo di conversione binario-decimale della parte frazionaria:

- Data la sequenza di bit $a_{-1}\, a_{-2} \ldots a_{-m}$, la sua rappresentazione in base 2 si ottiene utilizzando il seguente algoritmo basato sulla divisione per 2:
  - ✓ $F_{-(m-1)} = a_{-m}/2$
  - ✓ $F_{-(m-2)} = (a_{-(m-1)} + F_{-(m-1)})/2$
  - ✓ .....
  - ✓ $F_{-i} = (a_{-(i-1)} + F_{-(i-1)})/2$
  - ✓ .....
  - ✓ $F_0 = (a_{-(1)} + F_{-(1)})/2$

Porre $F = F_0$

L'algoritmo termina quando sono stati considerati ordinatamente tutti i bit della rappresentazione, dal meno significativo al più significativo.

Esempio: sia F = .10110110 (m = 8) una frazione propria rappresentata in base 2. L' algoritmo di codifica binario-decimale è il seguente:

F = .10101101

$F_{-7} = a_{-8}/2 = ½ = 0.5$
$F_{-6} = (a_{-7} + F_{-7})/2 = (0 + 0.5)/2 = 0.25$
$F_{-5} = (a_{-6} + F_{-6})/2 = (1 + 0.25)/2 = 0.625$
$F_{-4} = (a_{-5} + F_{-5})/2 = (1 + 0.625)/2 = 0.8125$
$F_{-3} = (a_{-4} + F_{-3})/2 = (0 + 0.8125)/2 = 0.40625$
$F_{-2} = (a_{-3} + F_{-3})/2 = (1 + 0.5)/2 = 0.703125$
$F_{-1} = (a_{-2} + F_{-2})/2 = (1 + 0.703125)/2 = 0.3515625$
$F_0 = (a_{-1} + F_{-1})/2 = (0 + 0.3515625)/2 = 0.67578125$

da cui F = F_0 = .67578125

9. Rappresentazione di numeri reali in virgola mobile

La rappresentazione in virgola fissa risulta inadeguata poiché consente di rappresentare numeri reali in un intervallo molto limitato, definito dalla quantità di bit che si hanno a disposizione. Affinchè si possano rappresentare numeri reali molto grandi e molto piccoli è necessario svincolare la virgola da una posizione fissa all'interno della sequenza di bit usati per la rappresentazione. La notazione in virgola mobile soddisfa questa richiesta. Essa fa riferimento alla notazione scientifica per la quale un numero reale N_{10} viene scritto come:

$$N_{10} = (-1)^s \cdot a \cdot 10^E$$

dove $(-1)^s$ denota il segno del numero, a è un numero reale con una sola cifra diversa da zero prima del punto decimale, ed E, l'*esponente*, è un numero intero.

Esempio:
$+3,14 = +3,14 \cdot 10^0$
$-0,000001 = -0,000001 \cdot 10^0 = 0,1 \cdot 10^{-5} = 1,0 \cdot 10^{-6}$

L'intervallo di rappresentabilità è determinato dal numero di cifre che compongono l'esponente, mentre la precisione è determinata dal numero di cifre della frazione.
Un numero in notazione scientifica che non abbia uno 0 a sinistra della virgola si dice *normalizzato*. La normalizzazione è il modo più appropriato per rappresentare numeri reali in notazione scientifica: per i numeri normalizzati, infatti, esiste un'unica forma, mentre per quelli non normalizzati ne esistono molteplici .
Quanto finora affermato per la notazione decimale, si estende alla notazione binaria; ovvero si possono rappresentare numeri reali binari in notazione scientifica normalizzata come:

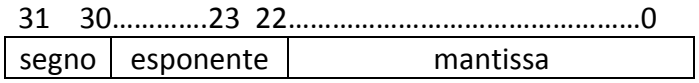
$$N_2 = (-1)^s \cdot a \cdot 2^E$$

dove a è un numero binario del tipo 1,xxxxxx…xx (normalizzazione), denominato *significando* con la sequenza di bit dopo la virgola che prende il nome di *mantissa*.

Esempio:
$1,0 \cdot 2^{-1}$ è in notazione scientifica normalizzata
$1,01011 \cdot 2^5$ è in notazione scientifica normalizzata

$10,001 \cdot 2^{-4}$ non è in notazione scientifica normalizzata
$0,1 \cdot 2^{-3}$ non in notazione scientifica normalizzata

Nel definire una rappresentazione in virgola mobile bisogna trovare un compromesso tra la dimensione della mantissa e la dimensione dell'esponente. Infatti, poiché la rappresentazione del segno, della mantissa e dell'esponente dovranno trovare spazio in una parola di dimensioni prestabilite, il compromesso di cui sopra si riflette sulla precisione della rappresentazione e sulla dimensione dell'intervallo di rappresentabilità: aumentando il numero di bit riservati alla mantissa, migliora la precisione, mentre aumentando il numero di bit riservati all'esponente migliora la dimensione dell'intervallo di rappresentabilità.
Lo standard IEEE 754 per la virgola mobile stabilisce il formato a *precisione singola* per la rappresentazione dei numeri reali, utilizzando una parola di 32 bit, in cui il bit 31 ha valore di segno (modulo e segno), gli 8 bit da 30 a 23 sono riservati alla codifica dell'esponente e i rimanenti 23 bit da 22 a 0 sono riservati alla mantissa. Il bit 1, posizionato prima della virgola nel significando, non viene rappresentato nella parola a 32 bit, ma è implicito. Il risparmio della rappresentazione di questo bit implicito è un vantaggio derivante dalla normalizzazione che impone all' l'ALU di tenerne conto.



Il segno, posizionato nel bit più significativo, rende più semplici i test di maggiore, minore o uguale a 0. L'esponente prima della mantissa semplifica l'ordinamento di numeri in virgola mobile, in quanto i numeri con esponenti maggiori sono più grandi dei numeri con esponente minore, purchè gli esponenti abbiano tutti lo stesso segno. Per questo motivo lo standard IEEE 754 non utilizza, per la rappresentazione degli esponenti, la notazione in complemento a due, ma la notazione *polarizzata*. In generale, la notazione polarizzata su n bit rappresenta un numero memorizzandone la somma con $2^{n-1}-1$; l'effetto è che tutti i numeri negativi rappresentabili su n bit diventano non negativi, mentre i non negativi conservano il loro segno, ma aumentano in valore assoluto. Con gli 8 bit a disposizione per la rappresentazione dell'esponente lo standard IEEE 754 usa un *coefficiente di polarizzazione* pari a 127 e rappresenta l'esponente più negativo con 00000000 e quello più positivo con 11111111. Pertanto, l'intervallo risultante per gli esponenti è:

$$[-127, +128]$$

L'intervallo dei significanti in forma normalizzata può essere calcolato come segue. L' estremo inferiore è 1 e si ottiene quando tutti i 23 bit della mantissa sono uguali a 0. L'estremo superiore si ottiene quando tutti i 23 bit della mantissa sono uguali a 1 :

$$1 + \sum_{i=-1}^{-23} 2^{i} = 1 + [1 - 2^{-23}] = 2 - 2^{-23}$$

L'intervallo dei significandi in forma normalizzata è quindi:

$$[1, (2 - 2^{-23})]$$

L'intervallo di rappresentabilità dei reali in virgola mobile è di conseguenza:

$$[-(2-2^{-23})\times 2^{128} \text{.....} -2^{-127} \ \underline{?????????} \ + 2^{-127} \text{.....} + (2-2^{-23})\times 2^{128}]$$

Non sono rappresentabili i:

- ✓ numeri negativi più piccoli di $-(2-2^{-23})\times 2^{128}$ (overflow negativo)
- ✓ numeri positivi più grandi di $+(2-2^{-23})\times 2^{128}$ (overflow positivo)
- ✓ numeri negativi più grandi di -2^{-127} (underflow negativo)
- ✓ lo zero
- ✓ numeri positivi più piccoli di $+2^{-127}$ (underflow positivo)

Tuttavia, proprio per ovviare alla mancanza di rappresentabilità dello zero e di valori matematici particolarmente significativi, quali ad esempio -∞ e +∞, non tutte le sequenze di bit nei formati IEEE sono interpretate nel modo sopra esposto: piuttosto, alcune di esse vengono utilizzate per rappresentare tali valori speciali. Così,

10000000000000000000000000000000 rappresenta - 0
00000000000000000000000000000000 rappresenta + 0
11111111111111111111111111111111 rappresenta - ∞
01111111111111111111111111111111 rappresenta + ∞

con conseguente modifica dell'intervallo di rappresentabilità (si perdono almeno quattro valori nell' intervallo discusso in precedenza).

Esempio 1: dato il reale N_{10} = -19.49, darne la sua rappresentazione in virgola mobile (standard IEEE 754).

Convertiamo separatamente la parte intera e la parte frazionaria di N_{10}.

$19 = Q_1 \times 2 + R_1 = 9 \times 2 + 1$
$Q_1 = 9 = Q_2 \times 2 + R_2 = 4 \times 2 + 1$
$Q_2 = 4 = Q_3 \times 2 + R_3 = 2 \times 2 + 0$
$Q_3 = 2 = Q_4 \times 2 + R_4 = 1 \times 2 + 0$
$Q_4 = 1 = Q_5 \times 2 + R_5 = 0 \times 2 + 1$

$19_{10} \rightarrow 10011_2$

$G_0 = G = .49$
$2 \times G_0 = a_{-1} + G_{-1} = 2x(.49) = 0 + .98$
$2 \times G_{-1} = a_{-2} + G_{-2} = 2x(.98) = 1 + .96$
$2 \times G_{-2} = a_{-3} + G_{-3} = 2x(.96) = 1 + .92$
$2 \times G_{-3} = a_{-4} + G_{-4} = 2x(.92) = 1 + .84$

Pur non essendo stata raggiunta alcuna parte periodica, fermiamo l'algoritmo ricordando che non riusciremo a rappresentare la parte frazionaria in binario con la stessa precisione che avevamo in decimale.

$.49 \rightarrow .0111$

Segue che:

$N_2 = 1001.\ 0111 \cdot 2^0$

Normalizziamo N_2:

$N_2 = 1001.\ 0111 \cdot 2^0 = 1.0010111 \cdot 2^3$

Mettiamo l'esponente in forma polarizzata:

$E_p = E + 127 = 3 + 127 = 130$ dove E_p con indichiamo l'esponente polarizzato

$E_p = E + 01111111 = 00000011 + 01111111 = 10000011$

La rappresentazione cercata è quindi:

31	30……23	22……0
1	10000011	00101110000000000000000

dato che il numero è negativo e la parte intera del significando è implicita.

Esempio 2: sapendo che la seguente parola a 32 bit rappresenta un numero reale in virgola mobile (standard IEEE 754), darne l'equivalente rappresentazione decimale.

31	30……23	22……0
0	11010001	10110111000000000000000

Il segno del numero è positivo. Convertiamo in decimale l'esponente:

$S_7 = a_7 = 1$
$S_6 = a_6 + 2 \times S_7 = 1 + 2 = 3$
$S_5 = a_5 + 2 \times S_6 = 0 + 6 = 6$
$S_4 = a_4 + 2 \times S_5 = 1 + 12 = 13$
$S_3 = a_3 + 2 \times S_4 = 0 + 26 = 26$
$S_2 = a_2 + 2 \times S_3 = 0 + 52 = 52$
$S_1 = a_1 + 2 \times S_2 = 0 + 104 = 104$
$S_0 = a_0 + 2 \times S_1 = 1 + 208 = 209$

$11010001 \rightarrow N_{10} = S_0 = 209$

Essendo l'esponente polarizzato, è necessario sottrargli il coefficiente di polarizzazione:

$E = E_p - 127 = 209 - 127 = +82$

Convertiamo, ora, la mantissa .10110111 :

$F_{-7} = a_{-8}/2 = ½ = 0.5$
$F_{-6} = (a_{-7} + F_{-7})/2 = (0 + 0.5)/2 = 0.25$
$F_{-5} = (a_{-6} + F_{-6})/2 = (1 + 0.25)/2 = 0.625$
$F_{-4} = (a_{-5} + F_{-5})/2 = (1 + 0.625)/2 = 0.8125$
$F_{-3} = (a_{-4} + F_{-3})/2 = (0 + 0.8125)/2 = 0.40625$
$F_{-2} = (a_{-3} + F_{-3})/2 = (1 + 0.5)/2 = 0.703125$
$F_{-1} = (a_{-2} + F_{-2})/2 = (1 + 0.703125)/2 = 0.8515625$
$F_{0} = (a_{-1} + F_{-1})/2 = (1 + 0.8515625)/2 = 0.92578125$

da cui segue che la mantissa è $F = F_0 = 0.92578125$; ad essa dobbiamo aggiungere l'1 implicito per il calcolo del significando:

$a = 1 + F = 1.\ 92578125$

Possiamo, quindi, scrivere la rappresentazione decimale del numero assegnatoci come:

$N_{10} = (-1)^s \cdot a \cdot 2^E = +1.\ 92578125 \cdot 2^0 = +1\ 92.578125 \cdot 2^{-2} = +1\ 92578125 \cdot 2^{-8} = +0.1\ 92578125 \cdot 2^1$

E' opportuno notare che soltanto la prima notazione è normalizzata: essa è unica; delle altre se ne potrebbero trovare infinite.

Lo standard IEEE 754 stabilisce altri due formati: quello a precisione doppia su 64 bit e quello a precisione estesa su 80 bit. In particolare il formato a precisione doppia assegna un bit al segno, 11 bit all'esponente e 52 bit alla mantissa.