L'informazione nel calcolatore

I calcolatori gestiscono dati di varia natura: numeri, testo, immagini, suoni, filmati, . . .

I dati sono codificati come opportune sequenze di bit. Presentiamo le rappresentazioni dei dati gestite dall'hardware del calcolatore: numeri e caratteri.

- rappresentazione di numeri naturali, interi, reali

L'informazione nel calcolatore

I calcolatori gestiscono dati di varia natura: numeri, testo, immagini, suoni, filmati, ...

I dati sono codificati come opportune sequenze di bit. Presentiamo le rappresentazioni dei dati gestite dall'hardware del calcolatore: numeri e caratteri.

- rappresentazione di numeri naturali, interi, reali
- operazioni aritmetiche eseguite dall'hardware

L'informazione nel calcolatore

I calcolatori gestiscono dati di varia natura: numeri, testo, immagini, suoni, filmati, . . .

I dati sono codificati come opportune sequenze di bit. Presentiamo le rappresentazioni dei dati gestite dall'hardware del calcolatore: numeri e caratteri.

- rappresentazione di numeri naturali, interi, reali
- operazioni aritmetiche eseguite dall'hardware
- rappresentazione di caratteri

L'informazione nel calcolatore

I calcolatori gestiscono dati di varia natura: numeri, testo, immagini, suoni, filmati, ...

I dati sono codificati come opportune sequenze di bit. Presentiamo le rappresentazioni dei dati gestite dall'hardware del calcolatore: numeri e caratteri.

- rappresentazione di numeri naturali, interi, reali
- operazioni aritmetiche eseguite dall'hardware
- rappresentazione di caratteri
- codici di correzione degli errori

L’informazione nel calcolatore

I calcolatori gestiscono dati di varia natura: numeri, testo, immagini, suoni, filmati, . . .

I dati sono codificati come opportune sequenze di bit. Presentiamo le rappresentazioni dei dati gestite dall’hardware del calcolatore: numeri e caratteri.

- rappresentazione di numeri naturali, interi, reali
- operazioni aritmetiche eseguite dall’hardware
- rappresentazione di caratteri
- codici di correzione degli errori
- convenzioni di memorizzazione.

Codifica dei dati

Codifica di un insieme di dati rappresentabili D:

Codifica dei dati

Codifica di un insieme di dati rappresentabili D:

- insieme di simboli A (alfabeto)

Codifica dei dati

Codifica di un insieme di dati rappresentabili D:

- insieme di simboli A (alfabeto)
- mappa tra dati rappresentabili e sequenze di simboli: $D \to A^*$ (codice).
  A^* é l'insieme di tutti i possibili sottoinsiemi che possiamo generare adoperando elementi di A.

Quasi sempre si considera l'alfabeto binario. In tal caso è $A = \{0, 1\}$, $A^* = \left\{\emptyset, \{0\}, \{1\}, \{0, 0\}, \ldots\right\}$.

Codifica dei dati

Codifica di un insieme di dati rappresentabili D:

- insieme di simboli A (alfabeto)
- mappa tra dati rappresentabili e sequenze di simboli: $D \to A^*$ (codice).
  A^* é l'insieme di tutti i possibili sottoinsiemi che possiamo generare adoperando elementi di A.

Quasi sempre si considera l'alfabeto binario. In tal caso è $A = \{0, 1\}$, $A^* = \Big\{\emptyset, \{0\}, \{1\}, \{0, 0\}, \ldots\Big\}$.

Un codice a lunghezza costante a n bit rappresenta fino a 2^n dati diversi.

Un codice a lunghezza variabile a n bit rappresenta i dati con un numero di bit $\leq n$.

Rappresentazione dei numeri

"Esistono 10 tipi di persone: quelle che capiscono la notazione binaria e quelle che non la capiscono."

Rappresentazione dei numeri

“Esistono 10 tipi di persone: quelle che capiscono la notazione binaria e quelle che non la capiscono.”

La base di un numero in genere si specifica come pedice che segue la sequenza di cifre:

257_{ten}, 257_{10}, -257_{eight}, 257_8, -1011.1001_2, -0.1_{two}.

I linguaggi di programmazione oltre ai decimali normalmente accettano numeri binari, ottali ($B = 8$) ed esadecimali ($B = 16$). Nel caso non decimale la base è specificata da un prefisso:

-.37, H328*C*, H-*ABCD*, 0x*ABCD*, O127, 0b-10.11.

Conversioni di base: da B a 10

Si applica direttamente la definizione di notazione posizionale: sommare le singole cifre ciascuna col suo peso.

Es. ($B = 2$): $101110110111.01_{two} =$
$2^{11} + 2^9 + 2^8 + 2^7 + 2^5 + 2^4 + 2^2 + 2^1 + 2^0 + 2^{-2} =$
$2048 + 512 + 256 + 128 + 32 + 16 + 4 + 2 + 1 + 1/4 =$
2999.25_{ten}

Non é computazionalmente efficente.

Accumulazione della parte intera

Nelle conversioni intere da base B a base 10 è più efficiente accumulare progressivamente il peso delle cifre nella sequenza.

A ogni passo aggiorno il risultato:

"risultato" $\leftarrow$ "risultato" $* B +$ "nuova cifra" (0 o 1).

Es. ($B = 2$): $101110110111_{two} = ?_{ten}$

$1_{two} = 1_{ten}$

Accumulazione della parte intera

Nelle conversioni intere da base B a base 10 è più efficiente accumulare progressivamente il peso delle cifre nella sequenza.

A ogni passo aggiorno il risultato:

"risultato" $\leftarrow$ "risultato" $* B +$ "nuova cifra" (0 o 1).

Es. $(B = 2)$: $101110110111_{two} = ?_{ten}$

$1_{two} = 1_{ten}$

$10_{two} = 1 * 2 + 0 = 2_{ten}$

Accumulazione della parte intera

Nelle conversioni intere da base B a base 10 è più efficiente accumulare progressivamente il peso delle cifre nella sequenza.

A ogni passo aggiorno il risultato:

"risultato" $\leftarrow$ "risultato" $* B +$ "nuova cifra" (0 o 1).

Es. $(B = 2)$: $101110110111_{two} = ?_{ten}$

$1_{two} = 1_{ten}$

$10_{two} = 1 * 2 + 0 = 2_{ten}$

$101_{two} = 2 * 2 + 1 = 5_{ten}$

Accumulazione della parte intera

Nelle conversioni intere da base B a base 10 è più efficiente accumulare progressivamente il peso delle cifre nella sequenza.

A ogni passo aggiorno il risultato:

"risultato" $\leftarrow$ "risultato" $* B +$ "nuova cifra" (0 o 1).

Es. $(B = 2)$: $101110110111_{two} = ?_{ten}$

$1_{two} = 1_{ten}$

$10_{two} = 1 * 2 + 0 = 2_{ten}$

$101_{two} = 2 * 2 + 1 = 5_{ten}$

$1011_{two} = 5 * 2 + 1 = 11_{ten}$

Accumulazione della parte intera

Nelle conversioni intere da base B a base 10 è più efficiente accumulare progressivamente il peso delle cifre nella sequenza.

A ogni passo aggiorno il risultato:

"risultato" $\leftarrow$ "risultato" $* B +$ "nuova cifra" (0 o 1).

Es. $(B = 2)$: $101110110111_{two} =?_{ten}$

$1_{two} = 1_{ten}$

$10_{two} = 1 * 2 + 0 = 2_{ten}$

$101_{two} = 2 * 2 + 1 = 5_{ten}$

$1011_{two} = 5 * 2 + 1 = 11_{ten}$ eccetera.

Accumulazione della parte intera

Nelle conversioni intere da base B a base 10 è più efficiente accumulare progressivamente il peso delle cifre nella sequenza.

A ogni passo aggiorno il risultato:

"risultato" $\leftarrow$ "risultato" $* B +$ "nuova cifra" (0 o 1).

Es. $(B = 2)$: $101110110111_{two} = ?_{ten}$

$1_{two} = 1_{ten}$

$10_{two} = 1 * 2 + 0 = 2_{ten}$

$101_{two} = 2 * 2 + 1 = 5_{ten}$

$1011_{two} = 5 * 2 + 1 = 11_{ten}$ eccetera.

Il risultato è ancora intero.

Accumulazione della parte intera: esempio grafico

1 0 1 1 1 0 1 1 0 1 1 1

$1 + 2 \times 1499 = 2999$ Result

$1 + 2 \times 749 = 1499$

$1 + 2 \times 374 = 749$

$0 + 2 \times 187 = 374$

$1 + 2 \times 93 = 187$

$1 + 2 \times 46 = 93$

$0 + 2 \times 23 = 46$

$1 + 2 \times 11 = 23$

$1 + 2 \times 5 = 11$

$1 + 2 \times 2 = 5$

$0 + 2 \times 1 = 2$

$1 + 2 \times 0 = 1$ Start here

Accumulazione della parte frazionaria

Nelle conversioni frazionarie da base B a base 10 è più efficiente accumulare progressivamente il peso delle cifre nella sequenza.

A ogni passo aggiorno il risultato:

"risultato" $\leftarrow$ "risultato" $/\ B +$ "nuova cifra" (0 o $1/B$).

Es. $(B = 2)$: $.01_{two} = ?_{ten}$

$.1_{two} = .5_{ten}$

Accumulazione della parte frazionaria

Nelle conversioni frazionarie da base B a base 10 è più efficiente accumulare progressivamente il peso delle cifre nella sequenza.

A ogni passo aggiorno il risultato:

"risultato" $\leftarrow$ "risultato" $/ \; B +$ "nuova cifra" (0 o $1/B$).

Es. ($B = 2$): $.01_{two} = ?_{ten}$

$.1_{two} = .5_{ten}$

$.01_{two} = .5/2 + 0 = .25_{ten}$.

Accumulazione della parte frazionaria

Nelle conversioni frazionarie da base B a base 10 è più efficiente accumulare progressivamente il peso delle cifre nella sequenza.

A ogni passo aggiorno il risultato:

"risultato" $\leftarrow$ "risultato" $/\ B +$ "nuova cifra" (0 o $1/B$).

Es. $(B = 2)$: $.01_{two} = ?_{ten}$

$.1_{two} = .5_{ten}$

$.01_{two} = .5/2 + 0 = .25_{ten}$.

Il risultato è ancora frazionario, eventualmente costituito da un numero infinitamente grande di cifre.

Ogni numero in base B può quindi essere convertito alla base 10.

Conversioni di base: da 10 a B

Si applica direttamente la definizione di notazione posizionale: scomporre in termini potenze di 2 per approssimazioni successive sulla parte intera e su quella frazionaria.

Es. ($B = 2$): $1492.875_{ten} =$
$1024 + 468 = 1024 + (256 + 212) + (.5 + .375) =$
$1024 + (256 + (128 + 84)\ldots) + (.5 + .25 + 0.125) =$
$1024 + 256 + 128 + 64 + 16 + 4 + 1/2 + 1/4 + 1/8 =$
$2^{10} + 2^8 + 2^7 + 2^6 + 2^4 + 2^2 + 2^{-1} + 2^{-2} + 2^{-3} =$
10111010100.111_{two}.

Non é efficiente, nè intuitivo per grandi numeri.

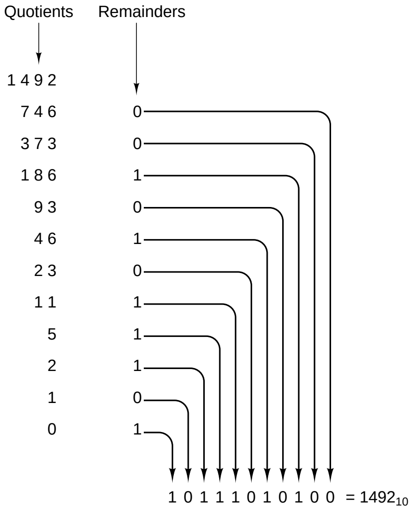
Isolamento della parte intera

Nelle conversioni intere da base 10 a base *B* è più efficiente e comprensibile isolare progressivamente le cifre nella nuova base.

Dividendo un numero decimale per *B* ottengo la cifra intera più a destra nella nuova base come resto della divisione.

Ripeto lo stesso procedimento sul numero decurtato del resto. Procedo fino a quando non c'è più nulla da decurtare.

Isolamento della parte intera: esempio grafico



Isolamento della parte frazionaria

Nelle conversioni frazionarie da base 10 a base B è più efficiente isolare progressivamente le cifre nella nuova base.

Moltiplicando un numero frazionario per B ottengo la cifra frazionaria più a sinistra nella nuova base come parte intera della moltiplicazione.

Ripeto lo stesso procedimento sul numero decurtato della parte intera. Procedo fino a quando non c'è più nulla da decurtare.

Questo procedimento non termina se la parte frazionaria nella base B è costituita da un numero infinitamente grande di cifre.

Ogni numero può quindi essere convertito a base B.

Conversioni tra basi non decimali

Un numero in base non decimale B_1 può facilmente essere convertito alla base non decimale B_2 passando per la base 10.

Le conversioni tra basi che sono potenze di 2 si calcolano più agilmente passando per la base 2.

Infatti, la conversione di un numero in base $B = 2^k$ mappa ogni cifra in base B in k cifre binarie.

Reciprocamente, la conversione di un numero binario alla base 2^k mappa gruppi di k cifre binare in una cifra nella nuova base.

Come corollario, le conversioni tra basi che sono potenze di 2 non generano numeri costituiti da un numero infinitamente grande di cifre.

Es.: conversione numero esadecimale $\longleftrightarrow$ numero ottale

Example 1

Hexadecimal	1 9 4 8 . B 6
Binary	0 0 0 1 1 0 0 1 0 1 0 0 1 0 0 0 . 1 0 1 1 0 1 1 0 0
Octal	1 4 5 1 0 . 5 5 4

Example 2

Hexadecimal	7 B A 3 . B C 4
Binary	0 1 1 1 1 0 1 1 1 0 1 0 0 0 1 1 . 1 0 1 1 1 1 0 0 0 1 0 0
Octal	7 5 6 4 3 . 5 7 0 4

Correttezza di $B_k = 2^k \longleftrightarrow B_m = 2^m$

Es.: $2 \longleftrightarrow 8$

$(d_7 d_6 d_5 d_4 d_3 d_2 d_1 d_0)_{\text{two}} =$

Correttezza di $B_k = 2^k \longleftrightarrow B_m = 2^m$

Es.: $2 \longleftrightarrow 8$

$$(d_7 d_6 d_5 d_4 d_3 d_2 d_1 d_0)_{\text{two}} =$$

$$= 2^7 d_7 + 2^6 d_6 + 2^5 d_5 + 2^4 d_4 + 2^3 d_3 + 2^2 d_2 + 2^1 d_1 + 2^0 d_0$$

Correttezza di $B_k = 2^k \longleftrightarrow B_m = 2^m$

Es.: $2 \longleftrightarrow 8$

$$(d_7 d_6 d_5 d_4 d_3 d_2 d_1 d_0)_{\text{two}} =$$

$$= 2^7 d_7 + 2^6 d_6 + 2^5 d_5 + 2^4 d_4 + 2^3 d_3 + 2^2 d_2 + 2^1 d_1 + 2^0 d_0$$

$$= (2d_7 + d_6)2^6 + (4d_5 + 2d_4 + d_3)2^3 + (4d_2 + d_1 + d_0)$$

Correttezza di $B_k = 2^k \longleftrightarrow B_m = 2^m$

Es.: $2 \longleftrightarrow 8$

$$(d_7 d_6 d_5 d_4 d_3 d_2 d_1 d_0)_{\text{two}} =$$

$$= 2^7 d_7 + 2^6 d_6 + 2^5 d_5 + 2^4 d_4 + 2^3 d_3 + 2^2 d_2 + 2^1 d_1 + 2^0 d_0$$

$$= (2d_7 + d_6)2^6 + (4d_5 + 2d_4 + d_3)2^3 + (4d_2 + d_1 + d_0)$$

$$= (2d_7 + d_6)2^{3*2} + (4d_5 + 2d_4 + d_3)2^{3*1} + (4d_2 + d_1 + d_0)$$

Correttezza di $B_k = 2^k \longleftrightarrow B_m = 2^m$

Es.: $2 \longleftrightarrow 8$

$$(d_7 d_6 d_5 d_4 d_3 d_2 d_1 d_0)_{\text{two}} =$$

$$= 2^7 d_7 + 2^6 d_6 + 2^5 d_5 + 2^4 d_4 + 2^3 d_3 + 2^2 d_2 + 2^1 d_1 + 2^0 d_0$$

$$= (2d_7 + d_6)2^6 + (4d_5 + 2d_4 + d_3)2^3 + (4d_2 + d_1 + d_0)$$

$$= (2d_7 + d_6)2^{3*2} + (4d_5 + 2d_4 + d_3)2^{3*1} + (4d_2 + d_1 + d_0)$$

$$= (2d_7 + d_6)8^2 + (4d_5 + 2d_4 + d_3)8^1 + (4d_2 + 2d_1 + d_0)$$

Correttezza di $B_k = 2^k \longleftrightarrow B_m = 2^m$

Es.: $2 \longleftrightarrow 8$

$$(d_7 d_6 d_5 d_4 d_3 d_2 d_1 d_0)_{\text{two}} =$$
$$= 2^7 d_7 + 2^6 d_6 + 2^5 d_5 + 2^4 d_4 + 2^3 d_3 + 2^2 d_2 + 2^1 d_1 + 2^0 d_0$$
$$= (2d_7 + d_6)2^6 + (4d_5 + 2d_4 + d_3)2^3 + (4d_2 + d_1 + d_0)$$
$$= (2d_7 + d_6)2^{3*2} + (4d_5 + 2d_4 + d_3)2^{3*1} + (4d_2 + d_1 + d_0)$$
$$= (2d_7 + d_6)8^2 + (4d_5 + 2d_4 + d_3)8^1 + (4d_2 + 2d_1 + d_0)$$
$$= ((2d_7 + d_6)(4d_5 + 2d_4 + d_3)(4d_2 + 2d_1 + d_0))_{\text{eight}}$$

Questo argomento può essere ripetuto per qualunque numero rappresentato in base $B_m = 2^m$.

L'aritmetica del calcolatore

I numeri naturali (unsigned), interi (signed), reali (float) sono in ogni caso rappresentati con un numero fissato di cifre:

- naturali e interi: 16 o 32 o 64 bit (fixed point)
- reali: 32 o 64 o 128 bit (floating point).

L'insieme dei valori rappresentabili è dunque limitato a $2^{16,32,64,128}$ codifiche. Conseguenze:

- esiste un valore massimo e uno minimo rappresentabile
- i valori possono essere approssimati, inizialmente oppure a seguito di operazioni
- i valori sono memorizzati con diverse codifiche.

Codifiche di interi

Codifiche di interi (es.: 8 bit):

- segno e valore assoluto (cambio di segno: si nega il bit più significativo)
  $9_{\text{dieci}} = 0\ 0001001 \qquad -9_{\text{dieci}} = 1\ 0001001$

Codifiche di interi

Codifiche di interi (es.: 8 bit):

- segno e valore assoluto (cambio di segno: si nega il bit più significativo)
  $9_{\text{dieci}} = 0\ 0001001 \qquad -9_{\text{dieci}} = 1\ 0001001$
- complemento a uno (cambio di segno: si nega ogni bit)
  $9_{\text{dieci}} = 0\ 0001001 \qquad -9_{\text{dieci}} = 1\ 1110110$

Codifiche di interi

Codifiche di interi (es.: 8 bit):

- segno e valore assoluto (cambio di segno: si nega il bit più significativo)
  $9_{\text{dieci}} = 0\ 0001001 \qquad -9_{\text{dieci}} = 1\ 0001001$
- complemento a uno (cambio di segno: si nega ogni bit)
  $9_{\text{dieci}} = 0\ 0001001 \qquad -9_{\text{dieci}} = 1\ 1110110$
- complemento a due (cambio di segno: si nega ogni bit e si incrementa)
  $9_{\text{dieci}} = 0\ 0001001 \qquad -9_{\text{dieci}} = 1\ 1110111$

Codifiche di interi

Codifiche di interi (es.: 8 bit):

- segno e valore assoluto (cambio di segno: si nega il bit più significativo)
  $9_{\text{dieci}} = 0\ 0001001 \qquad -9_{\text{dieci}} = 1\ 0001001$
- complemento a uno (cambio di segno: si nega ogni bit)
  $9_{\text{dieci}} = 0\ 0001001 \qquad -9_{\text{dieci}} = 1\ 1110110$
- complemento a due (cambio di segno: si nega ogni bit e si incrementa)
  $9_{\text{dieci}} = 0\ 0001001 \qquad -9_{\text{dieci}} = 1\ 1110111$
- eccesso N (zero va su N. Es.: $N = 2^{8-1} = 128$)
  $9_{\text{dieci}} = 1\ 0001001 \qquad -9_{\text{dieci}} = 0\ 1110111.$

Codifica complemento a due

Codice per ottenere un numero in complemento a due a N bit:

- valori tra 0 e $2^{N-1} - 1$ sono normalmente codificati in notazione binaria
- valori tra -1 e -2^{N-1} sono codificati negando ogni bit del rispettivo valore assoluto in notazione binaria e poi sommando 1.

  Es. $(N = 8)$:

  $-1_{10} \rightarrow 11111110_2 + 1_2 = 11111111_2$.

  $-2_{10} \rightarrow 11111101_2 + 1_2 = 11111110_2$.

  $-2^{N-1}_{10} \rightarrow 01111111_2 + 1_2 = 10000000_2$.

Quindi, usando N bit la codifica del numero $-i$ equivale al numero $2^N - i$ espresso in base 2.

Vantaggi codifica complemento a due

É la più adoperata per la codifica di interi:

- come nelle codifiche modulo e segno e complemento a uno, è immediato il riconoscimento del segno in quanto il numero è negativo se e solo se il bit più significativo è uguale a uno
- la somma algebrica non necessita di complicare il sommatore che abbiamo visto
- quindi, la sottrazione si calcola sommando il secondo termine dopo averlo cambiato di segno
- infine, l'estensione a una codifica di $M > N$ bit si ottiene immediatamente aggiungendo $M - N$ cifre uguali al segno alla codifica a N bit.

Operazioni in complemento a 2

Avendo a disposizione un sommatore a N bit e un circuito per negare N bit possiamo immediatamente calcolare

- somma: ovviamente
- opposto: nego e poi incremento
- sottrazione: sommo l'opposto del secondo termine
- prodotto: sommo per un numero di volte uguale al termine positivo
- divisione: sottraggo il divisore fino ad annullare il dividendo, contando le sottrazioni.

Prodotto e divisione sono realizzati da circuiti dedicati più sofisticati, molto più veloci.

Gestione di interi decimali

La convenienza del calcolo binario ha una contropartita nell'acquisizione e presentazione dei numeri, che viceversa dev'essere fatta nella base 10.

Aritmetica BCD (Binary Coded Decimal)

- cifre da 0 a 9, ognuna rappresentata con 4 bit
- vantaggi: nessun cambio di precisione, interfaccia utente più diretta
- svantaggi: complica i circuiti per le operazioni, usa più bit del necessario.

L'aritmetica BCD si realizza limitatamente all'hardware per l'interfaccia utente, per convertire numeri da / a decimali a / da binari durante l'acquisizione / presentazione dei dati.

Esempi di conversioni di base

Es.: $0.4_{10} = ?_2$

Es.: $0.\overline{3}_{10} = ?_3$

Es. (impegnativo): dimostrare che la conversione di base mappa parti reali in parti reali e parti frazionarie in parti frazionarie.

Overflow

Quando il risultato di un’operazione non è rappresentabile l’hardware segnala un errore di overflow.

Caso tipico: la somma di numeri grandi in assoluto, aventi lo stesso segno.

Nella codifica complemento a due c’è overflow solo se sommo numeri con lo stesso segno. In tal caso, c’è overflow solo se il segno del risultato non coincide con quello dei termini della somma.

N.B.: il riporto del full adder che calcola la cifra più significativa non è di alcuna utilità per la determinazione dell’overflow.

Notazione mantissa ed esponente

I numeri reali possono essere alternativamente rappresentati in notazione mantissa ed esponente:

$X = m \cdot 10^e$, in cui

- mantissa m: numero frazionario normalizzato $(-1 < m < 1)$
- esponente e: numero intero.

L'esponente indica di quante posizioni deve essere spostata la virgola nella mantissa.

Vantaggio: si rappresenta la mantissa sempre in modo relativamente preciso.

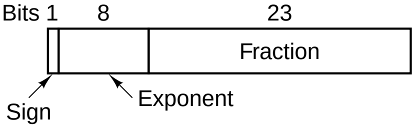
Il calcolatore eredita questa notazione adoperando esponenti di 2: $X = m \cdot 2^e$.

Rappresentazione floating point

Occupazione in bit fissa per mantissa ed esponente. Quindi, il numero di numeri in base 2 rappresentabili è finito: $X_{two} = m_{two} \cdot 2^{e_{two}}$.

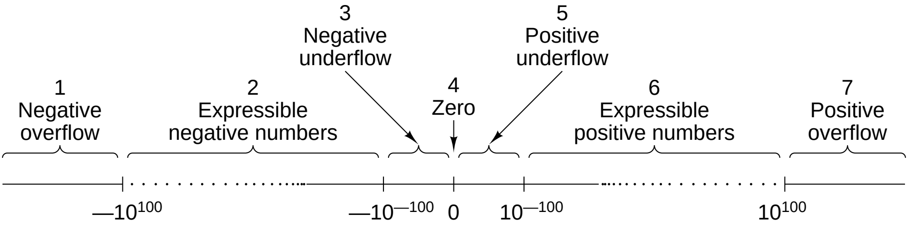
Codifica IEEE 754 32 bit (T.A. William Kahan):

- bit più a sx di segno: 0 positivo, 1 negativo
- e_{two} (8 bit) in eccesso N bit ($N = 127$)
- m_{two} (23 bit) normalizzata sempre nella forma $1.\cdots_{two}$, così il valore 1 e la virgola possono essere omessi dalla rappresentazione.



Errore d'approssimazione relativo

La granularità (relativa!) della rappresentazione dipende dalla precisione della mantissa:



- più bit per m_{two} $\Rightarrow$ granularità più fine
- più bit per e_{two} $\Rightarrow$ intervallo di rappresentazione più ampio.

Operazioni floating point

Le reti logiche viste (full adder) vanno arricchite:

- somma e sottrazione: sommo (sottraggo) le mantisse dopo avere eguagliato gli esponenti (cioè allineo la virgola). Es. (mantissa di 8 bit):
  $1.10100000_2 \cdot 2^{23} + 1.00100000_2 \cdot 2^{27} =$
  $0.00011010_2 \cdot 2^{28} + 0.10010000_2 \cdot 2^{28}$
- moltiplicazione: moltiplico le mantisse e sommo gli esponenti. Es. (mantissa di 4 bit):
  $1.1010_2 \cdot 2^{23} \cdot 1.0010_2 \cdot 2^{-27} =$
  $0.1101_2 \cdot 2^{24} \cdot 0.1001_2 \cdot 2^{-26} = 0.01110101 \cdot 2^{-2}$
- divisione: divido (= moltiplico per il reciproco) le mantisse e sottraggo gli esponenti.

Occorrono un moltiplicatore e un "reciprocatore".

Numeri speciali

Se un'operazione floating point restituisce un valore non rappresentabile, questo è rimpiazzato da numeri speciali: 0, $\pm\infty$, NaN.

Includendo anche i numeri denormalizzati che codificano la forma $0.m \cdot 2^{-126}$, le rappresentazioni floating point possibili sono:

	Sign bit	Exponent	Mantissa
Normalized	$\pm$	0 < Exp < Max	Any bit pattern
Denormalized	$\pm$	0	Any nonzero bit pattern
Zero	$\pm$	0	0
Infinity	$\pm$	1 1 1…1	0
Not a number	$\pm$	1 1 1…1	Any nonzero bit pattern