

RAPPRESENTAZIONE DELL'INFORMAZIONE

INTRODUZIONE ALLA RAPPRESENTAZIONE DIGITALE DELL'INFORMAZIONE:

I calcolatori digitali rappresentano e processano informazioni utilizzando set di bit. Ogni bit rappresenta un valore binario (0 o 1), e una sequenza di bit può codificare numeri, caratteri e altre informazioni digitali.

Nelle architetture più datate, i calcolatori utilizzavano generalmente set da 8 a 16 bit, che limitavano la quantità di dati che potevano essere elaborati contemporaneamente. Tuttavia, l'evoluzione delle tecnologie ha portato allo sviluppo di architetture a 32 e 64 bit nei calcolatori moderni. Questa maggiore ampiezza del set di bit ha permesso un incremento significativo delle **capacità di elaborazione** e rappresentazione dei dati, migliorando sia **l'efficienza che la velocità dei sistemi**.

VANTAGGI DELL'EVOLUZIONE :

L'uso di set di bit più ampi offre vari vantaggi:

- **Maggiore precisione e intervallo di valori:** I sistemi a 64 bit possono **rappresentare numeri molto più grandi** e precisi rispetto a quelli a 32 bit.
- **Maggiore efficienza nei calcoli complessi:** I calcolatori a 64 bit possono **gestire grandi quantità di dati** e operazioni su numeri più complessi in modo più rapido, grazie alla possibilità di effettuare calcoli in un'unica operazione.
- **Supporto per maggiori quantità di memoria:** Le architetture a 64 bit permettono l'accesso diretto a uno **spazio di memoria molto più ampio** rispetto a quelle a 32 bit, un fattore essenziale per le applicazioni che richiedono grandi quantità di RAM.

FORMULA PER IL CALCOLO DEI VALORI RAPPRESENTABILI CON UN SET DI BIT

Con un set di **N bit** è possibile rappresentare tutti i numeri interi positivi e lo zero, senza segno. Per calcolare la quantità di valori rappresentabili, si utilizza la formula:

$$2^N - 1$$

Così facendo otteniamo il range di valori rappresentabile compreso tra 0 e $2^N - 1$

ESEMPIO:

$$N = 8 \rightarrow 2^8 - 1 = 255$$

CASO DI OVERFLOW

L'**overflow** è una condizione che si verifica quando il risultato di un'operazione aritmetica **eccede la capacità di rappresentazione del set di bit a disposizione**. In altre parole, il calcolatore non ha abbastanza bit per contenere il numero risultante, causando una perdita di precisione o l'errata interpretazione del valore.

ESEMPIO:

Supponiamo di dover eseguire l'operazione $C = A + B$, dove $A = 30945$ e $B = 54667$. Il risultato in C sarebbe 85612. Tuttavia, se disponiamo di un set di soli 16 bit (insufficiente in questo caso), si genera un overflow nei registri, rendendo impossibile la rappresentazione corretta del risultato.

Conseguenze dell'Overflow:

- **Con valori senza segno:** il risultato si "riavvolge" partendo da zero, generando un valore inferiore a quello atteso.
- **Con valori con segno:** l'overflow può causare un salto nei numeri negativi, portando a risultati inattesi.

Rilevamento dell'Overflow:

Per prevenire e rilevare l'overflow, i calcolatori implementano meccanismi di verifica, come il controllo del bit di overflow nel registro di stato.

RAPPRESENTAZIONE DI CIFRE CON SEGNO

Nei calcolatori, un problema fondamentale è la rappresentazione di numeri con segno (positivi e negativi) utilizzando la numerazione binaria. Per gestire i numeri con segno, occorre un sistema che non solo consenta la rappresentazione dei valori positivi e negativi, ma permetta anche di eseguire operazioni tra numeri con e senza segno in modo corretto.

Bit di Segno (MSB - Most Significant Bit)

Per distinguere i numeri positivi dai negativi, il bit più significativo (MSB) del set di bit è riservato per indicare il segno:

- **MSB = 0** → Il numero è **positivo**
- **MSB = 1** → Il numero è **negativo**

L'uso del bit di segno riduce il range di valori rappresentabili dal set di bit

ESEMPIO

Vogliamo rappresentare i valori:

-2 e +5

-2 → 1 0000010

+5 → 0 0000101

DIFETTI DELLA RAPPRESENTAZIONE CON BIT DI SEGNO

Nella rappresentazione con bit di segno, il primo bit indica il segno del numero (0 per il positivo, 1 per il negativo), mentre i restanti bit rappresentano il valore assoluto. Tuttavia, questo metodo presenta alcuni svantaggi:

- **Doppia rappresentazione dello zero:** Il valore 0 può essere rappresentato in due modi diversi, ossia con le configurazioni 00000000 (0 positivo) e 10000000 (0 negativo). Questa ridondanza può causare ambiguità e inefficienza nella gestione dei dati.
- **Incompatibilità con le operazioni di somma e sottrazione:** Gli algoritmi di somma e sottrazione standard tra numeri binari non funzionano correttamente con i numeri rappresentati in bit di segno. Ad esempio, se si somma -9 e 9 usando la rappresentazione con bit di segno, non si otterrà 0, poiché il calcolo binario tradizionale non tiene conto del segno nella posizione corretta.

SOLUZIONE AL PROBLEMA

Per ovviare a questi difetti, si utilizza la rappresentazione in **complemento a due**. In questa codifica, il bit più significativo (MSB) ha un valore negativo, mentre i restanti 7 bit continuano a rappresentare il valore in modo positivo. Il complemento a due risolve i problemi del bit di segno

poiché consente operazioni di somma e sottrazione senza modifiche agli algoritmi standard e garantisce una rappresentazione unica per lo zero.

In un set di 8 bit, il massimo valore rappresentabile è **127 (01111111)**, e il minimo è **-128 (10000000)**. Per rappresentare 128 occorrerebbe invece un set di 9 bit.

Esempio di rappresentazione in complemento a due

1. **Numero negativo**: rappresentiamo il numero -43.

- In binario: $n=11010101$.

$$n=11010101 \quad n = 11010101$$

- Decodifica: $-128+64+16+4+1=-43$.

$$-128+64+16+4+1=-43 \quad -128 + 64 + 16 + 4 + 1 = -43$$

2. **Numero positivo**: rappresentiamo il numero 71.

- In binario: $n=01000111$.

$$n=01000111 \quad n = 01000111$$

- Decodifica: $64+4+2+1=71$.

$$64+4+2+1=71 \quad 64 + 4 + 2 + 1 = 71$$

Questa rappresentazione facilita le operazioni aritmetiche binarie e garantisce un intervallo simmetrico di numeri rappresentabili tra valori positivi e negativi.

PASSAGGI PER CODIFICARE UN NUMERO IN BINARIO CON SEGNO

Prendiamo come esempio il numero $N = -74$

1. Con un set di 8 bit, il MSB ha valore 128. Calcoliamo $X = 128 - |N| \rightarrow X = 54$

2. Ignoriamo il MSB e rappresentiamo X (54) con i 7 bit rimanenti

3. Essendo N negativo, settiamo il MSB a 1, ottenendo: **1 0110110**

4. Il risultato finale è: $-128 + 32 + 16 + 4 + 2 = -74$

5. Generalizzando, per $N < 0$, la formula è: $2^N - |N|$;

EVOLUZIONE DELL'ALGORITMO DI CONVERSIONE DEL COMPLEMENTO A 2

Dato un numero X su un set di K bit, si ha la necessità di rappresentare un numero negativo:

$X \rightarrow -25$

$K \rightarrow 8$

1. **Codifichiamo X da base decimale a base binaria in valore assoluto:** $X = 00011001$

2. **Invertiamo tutti i bit (COMPLEMENTO A 1):** $X = 11100110$

3. **Sommiamo +1 per ottenere il complemento a 2:**

$$X = 11100110 + 00000001 \rightarrow 11100111$$

4. **Risultato finale:** $-128 + 64 + 32 + 4 + 2 + 1 = -25$



l'algoritmo presentato è reversibile , ovvero dal -25 , sommando +1 è possibile ottenere +25

CASI LIMITE DEL CP2 (complemento a 2)

Il complemento a due (CP2) è una rappresentazione efficiente per gestire numeri con segno, ma presenta delle limitazioni, in particolare per quanto riguarda la gestione dell'**overflow**. Questo problema si verifica quando una somma o sottrazione produce un risultato che supera il valore massimo o minimo rappresentabile con un determinato set di bit.

Esempio di overflow

Immaginiamo di operare con un set di **8 bit**, in cui il minimo valore rappresentabile è **-128** e il massimo è **127**.

Consideriamo:

- $X = -73$
- $Y = -73$

Se sommiamo questi valori:

$$X + Y = -73 + (-73) = -146$$

il set di 8 bit permette un valore minimo di -128 , quindi notiamo come si verifichi un caso di overflow .



se sommiamo due numeri con segno opposto , è impossibile cadere in overflow , in quanto non si supererà mai il range massimo e minimo

inoltre se il risultato della somma tra due numeri mantiene lo stesso segno degli addendi , senza modificare il bit di segno tramite i riporti , la somma non cadrà mai in overflow

Condizioni di overflow in CP2

Ci sono alcune regole per individuare quando può verificarsi un overflow:

1. **Somma di due numeri con lo stesso segno:** Se entrambi i numeri sono positivi e il risultato è negativo, o se entrambi i numeri sono negativi e il risultato è positivo, si verifica un overflow. Questo accade perché la somma esce dall'intervallo rappresentabile.
2. **Somma di numeri con segno opposto:** Quando si sommano due numeri con segno opposto, l'overflow è impossibile. In questi casi, la somma non potrà mai eccedere il limite massimo o minimo del set di bit.
3. **Stabilità del bit di segno:** Se, dopo una somma, il bit di segno rimane invariato e non viene alterato dai riporti, allora l'operazione non cadrà mai in overflow. Questo si verifica perché il risultato resta entro il range rappresentabile.

GESTIONE DEI NUMERI CON O SENZA SEGNO IN LINGUAGGIO DI PROGRAMMAZIONE

alcuni linguaggi permettono di specificare la grandezza del numero da rappresentare e il numero di bit da impiegare per rappresentarlo , con o senza segno , queste sono le generali rappresentazioni in termini dichiarativi più comuni :

8 bit —> “char” / “byte”

16 bit —> “short int” / “short”

32 bit —> “integer” / “int”

64 bit —> “long int” / “long”

per specificare se un numero ha segno o no si usa :

“unsigned char “ / “unsigned int”

RAPPRESENTAZIONE DI NUMERI CON SEGNO MEDIANTE LA RAPPRESENTAZIONE IN CODICE DI ECCESSO B (notazione polarizzata)

un altro metodo più efficiente della notazione CP2 offrendo i seguenti vantaggi :

- la posizione dei bit durante la conversione da decimale a binario viene mantenuta a differenza del complemento a 2 , rendendo la comparazione tra numeri più semplice
- facilita le operazioni in virgola mobile
- facilita le rappresentazioni del segno

questo algoritmo si basa sull'uso di un *bias* che assume un valore differente in base al numero di bit usati per la rappresentazione , in una rappresentazione ad 8 bit si ha un *bias* di 127 , se si ha un set di 11 bit si ha un *bias* di 1023 .

ESEMPIO :

vogliamo tradurre il valore $X = -50$ su un set di 8 bit

1. sommiamo il *bias* al numero da convertire
 - a. $X = 127 - 50 \rightarrow X = 77$
2. ora è necessario tradurre il valore 77 in binario , ottenendo $\rightarrow 01001101$
3. per verificare la correttezza dell'operazione , procediamo con l'algoritmo inverso
 - a. per prima cosa convertiamo $X \rightarrow 01001101$ in decimale , ottenendo $\rightarrow 64+8+4+1 = 77$
 - b. per ottenere il valore originale , basta sottrarre il *bias* al numero ottenuto ($X-bias$)
ovvero :
 - c. $X = 77-127 \rightarrow -50$, facendo così abbiamo riottenuto il valore iniziale

RAPPRESENTAZIONE DI NUMERI BINARI TI TIPO FRAZIONARIO

per rappresentare un valore decimale con virgola in notazione decimale possiamo applicare il solito algoritmo , ma considerando i valori decimali e applicando l'uso di potenze negative , come segue .

ESEMPIO :

1. vogliamo rappresentare 235.13 in base 10



$$1. 2 \times 10^2 + 3 \times 10^1 + 5 \times 10^0 + 1 \times 10^{-1} + 3 \times 10^{-2} \rightarrow 235.13$$

2. vogliamo rappresentare 235.13 in base 2

a. ovvero $\rightarrow 101.1001$

b. applicando l'algoritmo :



$$2. 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4}$$

CONVERSIONE DA DECIMALE A BINARIO DI NUMERI FRAZIONARI CON VIRGOLA FISSA

preso un set di 8 bit , viene stabilito quali bit devono essere dedicati alla parte intera del numero e quali alla parte decimale del numero:

ESEMPIO :

rappresentiamo il valore numerico $\rightarrow 6.125$ in base 10 , definendo 4 bit per la parte intera e 4 bit per la parte decimale :

- parte intera = 0110
- parte decimale = 0010

- assemblando $\rightarrow 0110|0010$, ricordiamoci che nella parte decimale il bit a uno si trova in posizione 2^{-3} ovvero $\frac{1}{2^3}$ che risulterà essere 0.125
- sommando $6 + 0.125$, otteniamo 6.125

TRADUZIONE DELLA PARTE DECIMALE

per individuare il valore binario della parte decimale del numero è necessario applicare l'algoritmo di raddoppio della parte frazionaria :

1. vogliamo tradurre la parte decimale 0.15
2. moltiplichiamo : $0.15 \cdot 2 \rightarrow 0.30 \rightarrow 0.60 \rightarrow 1.20 \rightarrow$, una volta comparso 1 , la sequenza raddoppia ma senza la parte intera , $\rightarrow 0.40 \rightarrow 0.80 \rightarrow 1.60 \rightarrow 1.20 \rightarrow 0.40 \rightarrow 0.80 \dots\dots\dots$ viene eseguito un troncamento , perchè in questo caso è periodico
3. otteniamo $0.15 = 001001100\dots$

RAPPRESENTAZIONE DEI VALORI REALI IN FORMA NORMALIZZATA

per facilitare la rappresentazione dei numeri reali introduciamo l'applicazione della virgola mobile in forma normalizzata , implementando la MANTISSA , ovvero un valore M compreso tra 0 e 10 .

CAPACITA' DI RAPPRESENTAZIONE DI NUMERI REALI

il calcolatore permette di rappresentare 2^k numeri in base al set di bit in utilizzo , siccome i numeri sono infiniti e tra di un valore decimale e l'altro vi sono infiniti numeri , è necessario implementare un approssimazione .

LIMITI DI RAPPRESENTAZIONE

i calcolatori presentano dei limiti di rappresentazione :

1. numeri troppo grandi , o troppo piccoli
2. numeri troppo piccoli in valore assoluto , positivi o negativi

per rappresentare determinate frazioni in notazione è necessario applicare un arrotondamento o troncamento della parte decimale , per esempio , volendo rappresentare il valore $1/3$, che

risulterebbe 0.333333..... quindi verrà rappresentato come 0.333 ovvero in notazione scientifica :

$$3.33 * 10^{-1}$$

avendo più cifre per la mantissa questa tipologia di arrotondamenti non genera problemi

RAPPRESENTAZIONE DEI NUMERI REALI IN BASE 2

per rappresentare un numero reale in base 2 si applicherà lo stesso metodo della base 10 ovvero :

$$N = M * 2^E$$

usando questa notazione con mantissa avremo :

- NUMERI MOLTO GROSSI → dove la mantissa è negativa e l'esponente positivo
- NUMERI MOLTO PICCOLI → dove la mantissa assume valore positivo e l'esponente assume valore negativo



la mantissa è espressa in notazione modulo e segno , mentre l'esponente è rappresentato in notazione a codice d'eccesso

VIRGOLA MOBILE



spostare la posizione della virgola in un numero decimale determina la variazione dell'esponente , decrementandolo o incrementandolo
questo procedimento prende il nome di forma normalizzata .

ESEMPIO :

$$\pm 011.0110 * 2^{\pm 1011}$$

STANDARD IEEE 754

è lo standard che definisce la rappresentazione di valori decimali in virgola mobile , gestendo gli arrotondamenti ,

possiede due tipologie di precisione :

- **SINGOLA PRECISIONE** → 32 bit float [segno(1bit) esponente(8bit) mantissa(23bit)]
- **DOPPIA PRECISIONE** → 64 bit double [segno(1bit) esponente(11bit) mantissa(52bit)]

ESMPI IMPORTANTI :

convertiamo il numero -30.25 in base 10 :

1. convertiamo il -30 in notazione di modulo e segno ovvero → **MSB a 1** e poi 30 → 11110
2. convertiamo lo 0.25
 - a. $0.25 \rightarrow 0.50 \rightarrow 1.00 \rightarrow 01$
3. rappresentiamo → $-30.25 = 11110.01$
4. applichiamo la notazione scientifica , possiamo spostare la virgola di 4 posizioni a sinistra avendo → $1.111001 * 2^4$
5. **la mantissa risulta** → $M = 111001\ 0000\dots$
6. per **tradurre l'esponente** dobbiamo prendere il bias di un set di 8 bit ovvero 127 , a questo bias sommiamo l'esponente ovvero $127 + 4 = 131$
7. ora precediamo a tradurre il 131 , ottenendo 10000011 (ricordiamo che l'esponente deve essere convertito in eccesso B)
8. come risultato finale abbiamo :
 - a. **segno** → **1**
 - b. **mantissa** → **111001**
 - c. **esponente** → **111001**

RISULTATO = 1 111000 111100100000000000000000

ESERCIZIO 2 :

Tradurre il numero = -5.828125

segno = 1 (in quanto negativo)

esponente = 0

traduciamo 5 in base binaria → 101

traduciamo la parte decimale 0.828125 in binario → 110101

otteniamo che $-5.828125 = 101.110101$

spostiamo di due posizioni la virgola : $1.01110101 * 2^2$

la mantissa = 01110101

esponente → $127 + 2 = 129 \rightarrow 10000001$

RISULTATO → 1 10000001 011101010000000000000000

OPERAZIONI IN VIRGOLA MOBILE

somma e sottrazione :

nello sviluppo di somma e sottrazione con virgola mobile :

- si uguagliano gli esponenti
- si sottraggono o sommano le mantisse
- si rinormalizza la mantissa , riaggiustando l'esponente

moltiplicazione e divisione

nello sviluppo di moltiplicazione e divisione con virgola mobile :

- si moltiplicano o dividono le mantisse
- si sommano o sottraggono gli esponenti
- viene rinormalizzata la mantissa riportandola in forma normale

RAPPRESENTAZIONE DEI CARATTERI

viene definito il protocollo di rappresentazione **ASCII** , che permette la rappresentazione di 128 caratteri , venne introdotto anche un altro standard di rappresentazione **EBCDIC** , ormai in

disuso , veniva usato nei mainframe e nei vecchi calcolatori .

La diretta evoluzione dell'ASCII è il codice **UNICODE** che permette un maggior range di caratteri rappresentabile dal calcolatore , facendo uso di un numero variabile di byte .

attualmente il calcolatore non è in grado di comprendere quando 100011 si un numero intero o un carattere , quindi è compito del programmatore quello di instradare il calcolatore verso l'interpretazione della sequenza di bit , questo avviene grazie alla **tipizzazione** delle **variabili** in un **programma** .

attribuendo ad ogni variabile un determinato tipo , definiamo che tipologia di informazione il calcolatore deve elaborare e rappresentare .



assembly non permette di definire una tipizzazione per le variabili nel programma