

LOGICA DIGITALE

ALGEBRA BOOLEANA

L'algebra di Boole è il ramo dell'algebra che usa variabili che possono assumere solamente i valori vero e falso, generalmente denotati rispettivamente come 1 e 0; viene anche denominata algebra digitale perché utilizza i così detti bit.

Le operazioni fondamentali non sono addizioni e sottrazioni, ma gli operatori logici:

Somma logica \rightarrow OR

Prodotto logico \rightarrow AND

Negazione logica \rightarrow NOT

OR può essere indicato con: $+$, $|$, \parallel , \vee

AND può essere indicato con: $*$, \cdot , $\&$, $\&\&$, \wedge

NOT può essere indicato con: \bar{X} , $!$, $!!$, \neg , \sim

Nell'algebra booleana sono soddisfatte le seguenti proprietà:

- **commutativa:**

$$A + B = B + A$$

- **associativa:**

$$A + (B + C) = (A + B) + C \quad e \quad A * (B * C) = (A * B) * C$$

- **assorbimento:**

$$A + (A * B) = A \quad e \quad A * (A + B) = A$$

- **distributiva:**

$$A * (B + C) = (A * B) + (A * C) \quad e \quad A + (B * C) = (A + B) * (A + C)$$

- **idempotenza:**

$$A + A = A \quad e \quad A * A = A$$

- **esistenza di minimo e massimo:**

$$A * 0 = 0 \quad e \quad A + 1 = 1$$

- **Esistenza del complemento:**

$$A * \bar{A} = 0 \quad e \quad A + \bar{A} = 1$$

- **Simmetria:**

$$A + B = B + A \quad e \quad A \cdot B = B \cdot A$$

- **Involuzione di NOT:**

$$\text{NOT}(\text{NOT}(A)) = A \quad \text{oppure} \quad \bar{\bar{A}} = A$$

TEOREMI DI DE MORGAN

I teoremi o leggi di De Morgan stabiliscono relazioni di equivalenza tra gli operatori logici and e or. Vengono spesso utilizzate per la **semplificazioni** di equazioni booleane.

I due teoremi sono duali:

$$\overline{A + B} = \bar{A} \cdot \bar{B}$$

$$\overline{A \cdot B} = \bar{A} + \bar{B}$$

Gli stessi teoremi sono **generalizzabili** ad un numero n di variabili:

$$\overline{A \cdot B \cdot C \cdot \dots} = \bar{A} + \bar{B} + \bar{C} + \dots$$

$$\overline{A + B + C + \dots} = \bar{A} \cdot \bar{B} \cdot \bar{C} \cdot \dots$$

SHEFTER STROKE

La base di Shefter stroke o **NAND** si basa sulle operazioni NOT e AND, tramite le quali è possibile ottenere tutte le operazioni booleane. Un'algebra booleana può essere definita sia da NOT e AND sia da NOT e OR, essendo possibile definire OR attraverso NOT e AND così come AND attraverso NOT e OR.

Qui di seguito le equazioni:

$$A \cdot B = \overline{\overline{A} + \overline{B}}$$

$$A + B = \overline{\overline{A} \cdot \overline{B}}$$

Guardando meglio le equazioni qui sopra, è palese come queste siano una trascrizione dei teoremi di De Morgan in altra forma.

OPERATORI E PORTE LOGICHE

Come nelle espressioni matematiche, anche le operazioni in algebra booleana hanno una precedenza. La logica è più o meno la stessa: prima la moltiplicazione e poi la somma: la prima operazione da eseguire è sempre il NOT, segue AND e successivamente OR.

Risulta quindi che l'espressione:

$$\text{NOT } A * B \text{ AND NOT } C$$

equivale a:

$$\overline{A} + B\overline{C}$$

In questa rappresentazione risulta ovvio che la prima operazione da eseguire è la negazione.

Aggiungiamo ora le parentesi per evidenziare quale operazione va eseguita prima, tra OR e AND:

$$\overline{A} + (B * \overline{C})$$

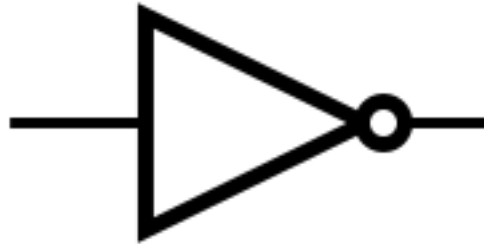
O in altra forma:

$$\bar{A} + B\bar{C}$$

Riportandola in quest'ultima forma, la sequenza di operazioni da eseguire risulta palese, ed è anche più semplice, se necessario, semplificare l'espressione attraverso i teoremi di De Morgan.

NOT

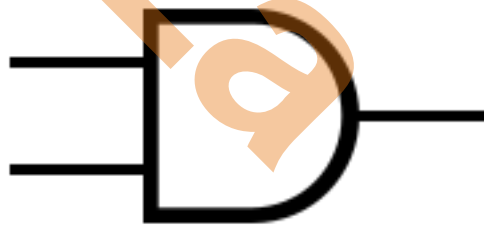
A	NOT A
0	1
1	0



Spesso per semplificare espressioni complesse, si usano operatori brevi che uniscono operazioni di NOT ad altre, ad esempio: **NOR** (OR + NOT), **NAND** (AND + NOT), **XNOR** (XOR + NOT). Attenzione, la negazione in questi casi viene sempre ed esclusivamente applicata dopo il risultato dell'operatore principale (OR, AND e **XOR**).

AND

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1



L'operatore AND restituisce 1 se e solo se tutti gli operandi hanno valore 1. L'operatore AND è del tutto simile alla moltiplicazione; viene infatti rappresentato anche attraverso gli stessi simboli aritmetici della moltiplicazione ed è spesso chiamato prodotto logico.

OR

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1



L'operatore OR restituisce 1 quando almeno uno dei due operandi è 1. È del tutto simile all'operazione aritmetica di addizione e per tale motivo è spesso rappresentata dal simbolo più e chiamata somma logica.

ALTRI OPERATORI E PORTE LOGICHE

BUFFER

A	Buffer A
0	0
1	1



Non è una vera e propria porta logica in quanto lascia passare il “segnale” inalterato; è solitamente utilizzato nei circuiti logici quando si parla di **sincronia del segnale**: il buffer è infatti considerato un **ritardo da applicare al segnale**.

XOR

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0



L'operatore XOR, detto anche **OR esclusivo** o **somma modulo 2**, restituisce 1 se e solo se il numero degli operandi uguali a 1 è dispari, mentre restituisce 0 in tutti gli altri casi. La tabella di verità qui sopra riporta il caso in cui gli operatori siano 2; più in generale ci si riferisce a questo operatore come **operatore di disparità** quando il numero di ingressi maggiore di due.

NAND

A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0



L'operatore NAND è un operatore composto da AND e successivamente NOT. La NOT è indicata dal pallino dopo la AND.

NOR

A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0



L'operatore NOR è un operatore composto da OR e successivamente NOT. La NOT è indicata dal pallino dopo la OR.

XNOR

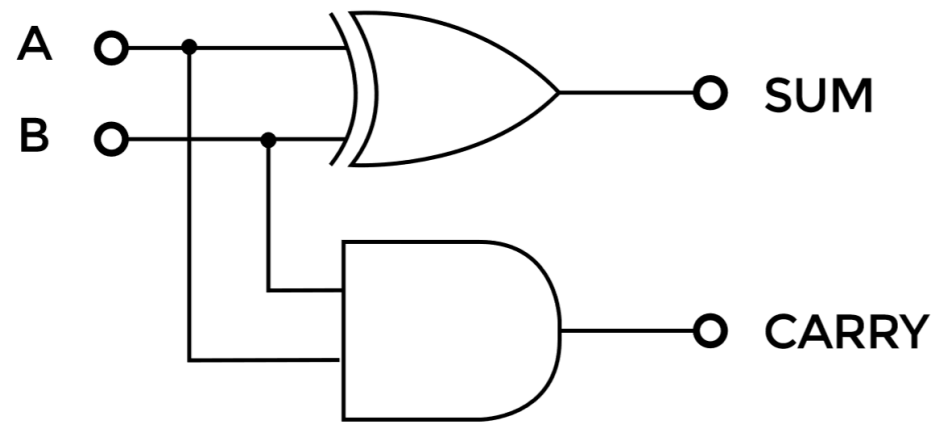
A	B	A XNOR B
0	0	1
0	1	0
1	0	0
1	1	1



L'operatore XNOR è un operatore composto da XOR e successivamente NOT. La NOT è indicata dal pallino dopo la XOR. È di solito utilizzato come **operatore di parità**, cioè restituisce il valore 1 se il numero di 1 in ingresso è pari.

HALF ADDER

A	B	SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

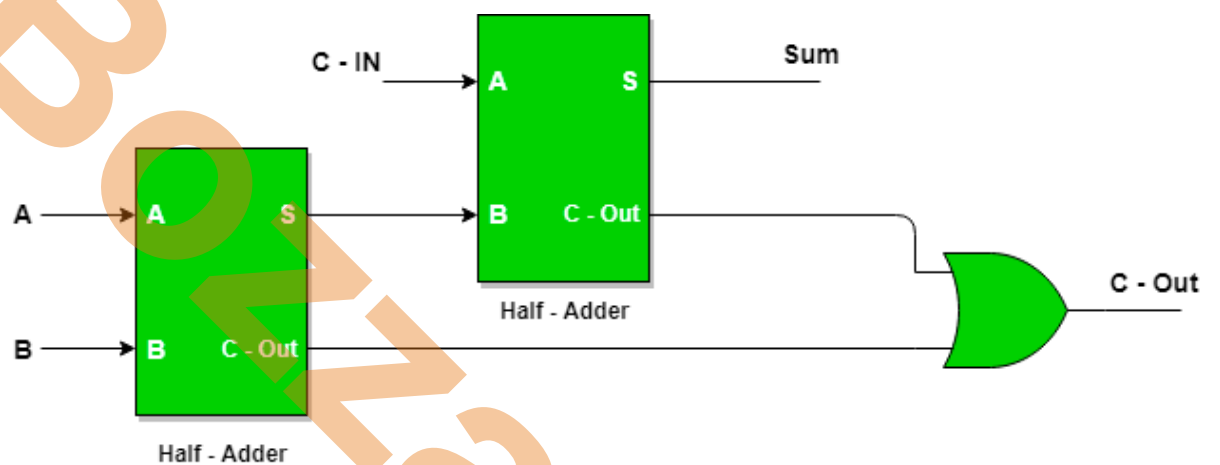


L'HALF ADDER è un circuito logico per la somma di due bit. Il circuito logico è in grado di generare due output, il risultato della somma e l'eventuale riporto.

FULL ADDER

Il FULL ADDER è un circuito digitale in grado di eseguire una somma logica completa.

Il circuito è in grado di sommare tre bit (solitamente due bit e un terzo costituito dal riporto di una operazione precedente. L'adder restituisce due bit, il risultato dell'operazione e l'eventuale riporto (carry).



A	B	C (CARRY IN)	SUM	CARRY
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

I SISTEMI NUMERICI - BASE 10, BASE 2, BASE 8 e BASE 16

Un sistema numerico identifica il numero di simboli utilizzate per la rappresentazione di un **numero**: il sistema decimale (o base 10) è il sistema che utilizziamo tutti i giorni ed utilizza 10 cifre (da 0 a 9) per rappresentare tutti i numeri.

Come dice il nome stesso, il sistema binario (o base 2) utilizza invece due soli simboli per rappresentare i numeri (0 e 1). È utilizzato nei sistemi elettronici perché ben rappresenta lo stato acceso o spento, di due differenti livelli di tensione (es. 0V e 5V).

Il sistema **esadecimale** (o base 16) utilizza 16 simboli (da 0 a 9 e lettere da A a F). E' molto utilizzato in informatica perché ben rappresenta lo stato della memoria come insieme di byte. Un **byte** è infatti composto da **8bit**, ma si può rappresentare in un modo semplice suddividendolo in due parti da **4 bit** (denominate **nibble**) che sono per cui facilmente convertibili in un numero esadecimale.

Il sistema totale (o base 8) utilizza ovviamente solo otto simboli (a 0 a 7) ed è il meno utilizzato.

La tabella sottostante mostra le diverse conversioni nei differenti sistemi numerici.

binario	ottale	decimale	esadecimale
0	0	0	0
1	1	1	1
10	2	2	2
11	3	3	3
100	4	4	4
101	5	5	5
110	6	6	6
111	7	7	7

binario	ottale	decimale	esadecimale
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F

LA CODIFICA BINARIA

Come appena descritto, il sistema binario è in grado di **rappresentare i numeri mediante due soli simboli**: rappresentare l'informazione con due simboli significa in qualche modo "cifrare" questa informazione.

Con n bit si possono codificare 2^n numeri decimali, da 0 a 2^n-1

Il metodo più semplice per convertire un numero decimale in un numero binario è il **metodo dei resti**: si calcolano i resti delle divisioni per due a partire dal numero che si vuole convertire.

Qui di seguito i passaggi del metodo:

- se il numero è dispari, il resto è 1, se pari il resto è 0.
- si dimezza il numero di partenza
- si procede con i punti precedenti fino a che il dimezzamento del numero da 1 o 0

Il numero binario si ottiene leggendo i resti dei dimezzamenti a partire dall'ultimo (cifra meno significativa) al resto della prima divisione (cifra più significativa).

Procediamo con un esempio:

Convertiamo il numero 19d

$$19:2 = 9 \text{ con resto } 1$$

Riportiamolo nella tabella seguente

Numero da convertire	19	
	9	1
Risultato della prima divisione		

Continuando con il procedimento avremo:
 $9:2 = 4 \text{ con resto } 1$

19	
9	1
4	1

$$4:2 = 2 \text{ con resto } 0$$


19	
9	1
4	1
2	0

2:2 = 1 con resto 0

19	
9	1
4	1
2	0
1	0

1: 2 = 0 con resto 1

19	
9	1
4	1
2	0
1	0
0	1



Il numero binario è composto dai resti delle divisioni, leggendolo dall'ultimo al primo (secondo il verso della freccia riportata qui sopra).

$$19d = 10011b$$

È possibile controllare se la conversione sommando il valore decimale di ogni bit.

$$n_5 \cdot 2^4 + n_4 \cdot 2^3 + n_3 \cdot 2^2 + n_2 \cdot 2^1 + n_1 \cdot 2^0$$

Con n_1 bit meno significativo, n_5 bit più significativo.

$$1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 16 + 2 + 1 = 19$$

L'aggiunta di uno zero in posizione meno significativa, con la codifica decimale moltiplica il numero per 10, nella codifica binaria raddoppia il numero. Il metodo è il medesimo in tutte le basi.

$$\begin{aligned} 10011b &= 19d \\ 100110b &= 38d \end{aligned}$$

Ecco un esempio relativo ad un byte. Un byte è composto da **8 bit**, per cui può rappresentare, in complemento a 1 numeri da **0 a 128**.

Prendiamo in considerazione il numero 26_d decimale (d indica un numero decimale).

Convertito in binario risulta: 0001 1010_b (b indica un numero binario).

Dividendolo in due nibble, la rappresentazione HEX (esadecimale) risulta:

0001 → 1_h (h indica il numero esadecimale)

1010 → A_h

26_d → 1A_h

Se si volessero rappresentare non solo numeri interi positivi ma anche negativi, occorre inserire il segno ovvero utilizzare **un bit per rappresentare il segno**.

Distaccando il bit di segno, i bit rimanenti rappresentano il valore assoluto.

I numeri positivi possono essere rappresentati in complemento a uno e complemento a due.

Il complemento a 1 aggiunge semplicemente un bit di segno davanti al numero. Consideriamo un numero a 8 bit, di cui 1 di segno: i numeri:

1000 0000

0000 0000

corrispondono entrambe al numero 0 (zero) decimale.

Si perde quindi una parte del potere di rappresentazione con 8 bit,

Il complemento a due ovvia a questo problema. Un numero in complemento a due viene calcolato invertendo il numero decimale in complemento a 1 e sommando 1.

In questo modo si ha che:

$$000_{c2} = 0_d$$

$$001_{c2} = 1_d$$

$$010_{c2} = 2_d$$

$$011_{c2} = 3_d$$

$$100_{c2} = -4_d$$

$$101_{c2} = -3_d$$

$$110_{c2} = -2_d$$

$$111_{c2} = -1_d$$

Il COMPLEMENTO a 1 con n bit è in grado di rappresentare numeri da $-2^{n-1}+1$ a $2^{n-1}-1$

Il COMPLEMENTO a 2 con n bit è in grado di rappresentare numeri da -2^{n-1} a $2^{n-1}-1$

Per convertire un numero decimale in un numero binario in complemento a due:

1. **Se il numero $D \geq 0$**

- Convertire il numero in binario

- Aggiungere uno 0 a sinistra del numero convertito (lo zero indica il segno)

2. Se il numero $D < 0$

- Convertire il numero in binario
- Aggiungere uno 0 a sinistra del numero convertito (lo zero indica il segno)
- Calcola l'opposto del numero così ottenuto, secondo la procedura di inversione del complemento a due.

Se il numero è positivo, aggiungendo zero a sinistra il numero non cambia.

Se il numero è negativo, aggiungendo uno a sinistra il numero non cambia.

Il bit di segno nel complemento a 2 è incorporato nel numero, mentre nel complemento a 1 è solamente aggiunto.

Suggerimento:

Un altro modo per convertire un numero binario in complemento a due:

- leggere e copiare il numero binario da destra a sinistra fino al primo 1.
- negare tutti gli altri numeri.

Es:

0110100

Leggo da destra e copio solamente fino al primo 1.

Nego tutti gli altri bit.

$\overline{0110100}$

Il risultato è:

1001100

OPERAZIONI TRA NUMERI BINARI

Per eseguire operazioni di somma e sottrazione tra numeri binari si procede come con i numeri decimali **sommando cifra per cifra** a partire dalla meno significativa, e nel caso di resto sommandola alla cifra più significativa successiva.

Tuttavia, quando si usa la codifica binaria, viene solitamente imposto un numero massimo di bit.

In questo caso, facendo somma e sottrazione può essere che vi sia un riporto sulla cifra significativa.

Se il risultato dell'operazione è corretto, si ha un riporto perduto; se il risultato è errato, si ha overflow, ovvero il numero di bit non è sufficiente per rappresentare il risultato dell'operazione.

Ecco delle semplici regole per controllare la presenza o meno di overflow, guardando solamente il bit più significativo:

- Se gli addendi sono tra loro **discordi** (di segno differente) **l'overflow non si verifica mai;**
- Se gli addendi sono tra loro **concordi** si ha overflow solo se il risultato è discorde:
 - addendi positivi, risultato negativo;
 - addendi negativi, risultato positivo.

NUMERI FRAZIONARI

Per convertire un numero decimale con la virgola in codifica binaria si procede convertendo la parte intera con **il metodo dei resti e la parte frazionaria** come segue:


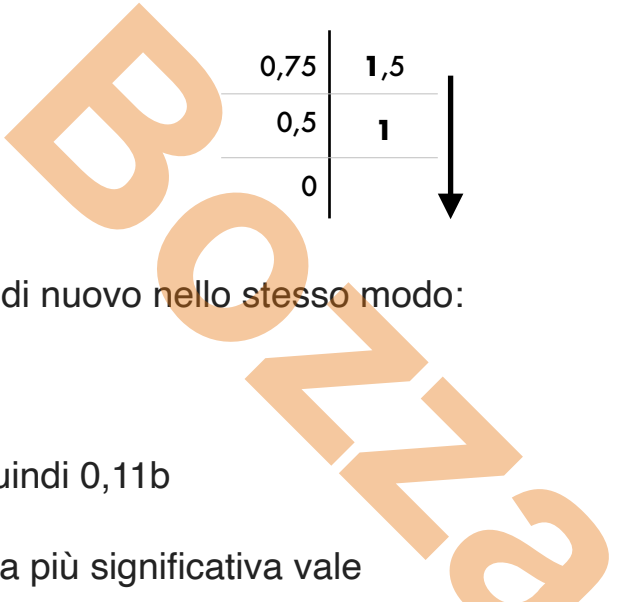
- si moltiplica per due il numero frazionario.
- Se il numero è inferiore a 1 si riporta 0, se superiore si riporta uno.
- Si procede con i due punti precedenti fino a che il valore moltiplicato per due risulta pari a 1 o si ripete un numero frazionario già moltiplicato. In questo secondo caso si è di fronte ad un numero binario periodico.

0,75	1,5
0,5	

Proviamo a convertire un numero semplice: 0,75

$$0,75 \times 2 = 1,5$$

0,75	1,5
0,5	1
0	



Sottraiamo 1 a 1,5 e procediamo di nuovo nello stesso modo:

$$0,5 \times 2 = 1$$

Il numero 0,75 in binario risulta quindi 0,11b

Ogni cifra frazionaria, a partire dalla più significativa vale

$$2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} \dots = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} \dots$$

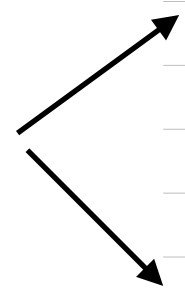
$$0,5 + 0,25 + 0,125 + 0,0625 \dots$$

0,11b equivale quindi a $0,5 + 0,25 = 0,75$

Proviamo ora a convertire il numero 0,3

Si ripete 0,6

0,3	0,6
0,6	1,2
0,2	0,4
0,4	0,8
0,8	1,6
0,6	



Il numero 0,3 nelle conversione si ripropone: si ha quindi un numero binario periodico che vale:

$$0,0\overline{1001}$$

In un computer, i numeri frazionari possono essere rappresentati a virgola fissa o virgola mobile.

Virgola fissa:

È un sistema di rappresentazione semplice, ma poco flessibile, e può condurre a sprechi di bit

- Per rappresentare in virgola fissa numeri molto grandi (o molto precisi) occorrono molti bit
- La precisione nell'intorno dell'origine e lontano dall'origine è la stessa

Anche se su numeri molto grandi in valore assoluto la parte frazionaria può non essere particolarmente significativa

Virgola mobile (floating point):

È un sistema di rappresentazione molto utilizzato in base 10 (è la così detta notazione scientifica) e si basa sulla rappresentazione:

$$R_{\text{virgolamobile}} = M \times B^E$$

M è la mantissa e E è l'esponente.

In binario si usano $m \geq 1$ bit per la mantissa e $n \geq 1$ bit per l'esponente.

La mantissa è un numero tra -1 e +1, la base B in binario non viene rappresentata.

In totale si usano $m+n$ bit.

Ecco un esempio:

Supponiamo di poter convertire il numero

$$0,011 \cdot 2^{010}$$

che equivale a:

$$\left(\frac{1}{4} + \frac{1}{8}\right) \times 2^2$$

ovvero:

$$\frac{3}{8} \times 4 = \frac{3}{2} = 1,5_{\text{dec}}$$

La rappresentazione in virgola fissa ha il vantaggio di:

- si possono rappresentare con pochi bit numeri molto grandi oppure molto precisi (cioè con molti decimali);
- sull'asse dei valori i numeri rappresentabili si affollano nell'intorno dello zero, e sono sempre più sparsi al crescere del valore assoluto.

Avendo tuttavia un numero limitato di cifre per la mantissa, a volte si può incorrere in problemi di approssimazione, come vedremo in seguito in alcuni esercizi.

Quasi tutti i calcolatori oggi adottano lo **standard aritmetico IEEE 754**, che definisce:

- I formati di rappresentazione binario naturale, C2 e virgola mobile
- Gli algoritmi di somma, sottrazione, prodotto, ecc, per tutti i formati previsti
- I metodi di arrotondamento per numeri frazionari
- Come trattare gli errori (overflow, divisione per 0, radice quadrata di numeri negativi, ...)

Lo standard IEEE 754 rappresenta i numeri mediante:

- **un bit per il segno** della mantissa S (0 = +, 1 = -)
- **alcuni** bit per l'esponente E
- **altri** bit per la mantissa (il suo valore assoluto) M

Il segno dell'esponente è rappresentato in notazione "eccesso K" per ovviare al problema del segno: si memorizza cioè il valore dell'esponente aumentato di K: **se k bit sono dedicati all'esponente, $K = 2^{(k-1)}$**

es:

k = 8 si memorizza esponente aumentato di $K = 128 - 1 = 127$

Inoltre la **mantissa è normalizzata** scegliendo l'esponente in modo tale che il primo valore della mantissa sia 1. In questo modo è possibile eliminare il primo 1 passandolo per sottinteso e guadagnare un bit.

La tabella di seguito mostra i bit utilizzati da mantissa ed esponente per le differenti tipologie di variabili del calcolatore.

Campo	Precisione singola	Precisione doppia	Precisione quadrupla
Numero totale di bit	32	64	128
Bit per Segno	1	1	1
Bit per Esponente	8	11	15
Bit per mantissa	23	52	111
Massimo esponente	255	2047	32767
Minimo esponente	0	0	0
K	127	1023	16383

Vediamo ora un esempio di conversione completa in virgola **fissa**, virgola mobile e IEEE754

Convertiamo il numero 42.6875d

Iniziamo convertendo il numero intero 42d

42	
21	0
10	1
5	0
2	1
1	0
0	1

Convertiamo poi il numero frazionario 0,6875

0,6875	1,375
0,375	0,75
0,75	1,5
0,5	1
0	

Il numero 42.6875d in virgola fissa risulta:

101010,1011

Per convertirlo in virgola mobile, spostiamo la virgola verso sinistra di 5 posizioni ottenendo il numero:

$$1,010101011 \cdot 2^{101}$$

In formato IEEE754 a precisione singola:

Segno = 0

Mantissa (23 bit) = 01010101100000000000000b

Esponente (8 bit) = K + esponente = K + 5 = 127 + 5 = 132 = 1000 0100b

ESERCIZI CODIFICA

ESERCIZIO LOGICA DIGITALE

Es.1

Si costruisca la tabella di verità della seguente espressione booleana in tre variabili, badando alla precedenza tra gli operatori logici.

$$(!A \text{ or } B \text{ and } C) \text{ and } (A \text{ or } !C) \text{ and } !B$$

Soluzione:

L'espressione può essere scritta nel seguente metodo:

$$(\bar{A} + BC)(A + \bar{C})\bar{B}$$

Risulta più chiaro in questo modo quale operazione va eseguita prima (1. NOT, 2. AND, 3. OR)

Compiliamo ora la tabella di verità. La tabella necessita di tre variabili, A B e C. Per riempire velocemente la tabella sappiamo che con 3 variabili si necessitano di 2^3 combinazioni. Si parte dalla terza colonna C, e la si riempie alternando 0 e 1. Si procede poi con la colonna B e si alternano due 0 a due 1. Infine la colonna A alternando quattro 0 e quattro 1. Se vi fossero più variabili si procederebbe alternando otto zeri e otto uno, ecc.

Per costruire la tabella di verità, si può dividere l'espressione in più parti.

A	B	C	$(\bar{A} + BC)$	$(A + \bar{C})$	\bar{B}	OUT
0	0	0	1	1	1	1
0	0	1	1	0	1	0
0	1	0	1	1	0	0
0	1	1	1	0	0	0
1	0	0	0	1	1	0
1	0	1	0	1	1	0
1	1	0	0	1	0	0
1	1	1	1	1	0	0

Es.2

Si costruisca la tabella di verità della seguente espressione booleana in tre variabili, badando alla precedenza tra gli operatori logici.

$$A \text{ or } !B \text{ and } !C \text{ or } !A \text{ and } B$$

Soluzione:

L'espressione può essere scritta nel seguente metodo:

$$A + \overline{B}\overline{C} + \overline{A}B$$

A	B	C	$\overline{B}\overline{C}$	$\overline{A}B$	OUT
0	0	0	1	0	1
0	0	1	0	0	0
0	1	0	0	1	1
0	1	1	0	1	1
1	0	0	0	0	1
1	0	1	0	0	1
1	1	0	0	0	1
1	1	1	0	0	1

Es.3

Si costruisca la tabella di verità della seguente espressione booleana in tre variabili, badando alla precedenza tra gli operatori logici.

$$\text{NOT (NOT (NOT A OR B) OR NOT B AND NOT C)}$$

Soluzione:

L'espressione può essere scritta nel seguente metodo:

$$\overline{\overline{A + B + \overline{B}\overline{C}}}$$

Possiamo o costruire la tabella di verità come segue, oppure semplificare l'espressione utilizzando le leggi di De Morgan.

Proviamo prima a risolvere il tutto partendo direttamente dall'espressione:

A	B	C	$\bar{A} + B$	$\overline{\bar{A} + B}$	$\bar{B}\bar{C}$	\overline{OUT}	OUT
0	0	0	1	0	1	1	0
0	0	1	1	0	0	0	1
0	1	0	1	0	0	0	1
0	1	1	1	0	0	0	1
1	0	0	0	1	1	1	0
1	0	1	0	1	0	1	0
1	1	0	1	0	0	0	1
1	1	1	1	0	0	0	1

Se invece proviamo a semplificare l'espressione

$$\overline{\overline{\bar{A} + B + \bar{B}\bar{C}}}$$

applicando il teorema di De Morgan:

$$\overline{\bar{A} + B} = \bar{\bar{A}} \cdot \bar{B}$$

$$\overline{\bar{A} \cdot B} = \bar{\bar{A}} + \bar{B}$$

L'espressione diventa:

$$\overline{\bar{A} + B + \bar{B}\bar{C}} = \overline{\bar{A} + B} \cdot \overline{\bar{B}\bar{C}}$$

Semplificando secondo l'involutione del NOT:

$$(\bar{A} + B) \cdot \overline{\bar{B}\bar{C}}$$

Riappliciamo De Morgan sulla seconda parte dell'espressione:

$$(\bar{A} + B) \cdot \overline{\bar{B}\bar{C}} = (\bar{A} + B) \cdot (\bar{\bar{B}} + \bar{\bar{C}})$$

$$(\bar{A} + B) \cdot (B + C)$$

Sviluppando il prodotto logico:

$$\bar{A}B + BB + \bar{A}C + BC$$

Semplifico il secondo addendo secondo la proprietà di idempotenza:

$$\bar{A}B + B + \bar{A}C + BC$$

Posso ulteriormente semplificare raccogliendo B:

$$\bar{A}C + B(1 + \bar{A} + C)$$

Secondo la proprietà di Minimo e Massimo otteniamo che $1 + \bar{A} + C = 1$, per cui l'espressione diventa:

$$\bar{A}C + B$$

La costruzione della tabella di verità risulta quindi molto più semplice.

A	B	C	$\bar{A}C$	OUT
0	0	0	0	0
0	0	1	1	1
0	1	0	0	1
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	0	1
1	1	1	0	1

Es.4

Si costruisca la tabella di verità della seguente espressione booleana in tre variabili, badando alla precedenza tra gli operatori logici.

$$(A \text{ or } B \text{ and } !C) \text{ and } A \text{ or } !C$$

Soluzione:

Iniziamo con il riscrivere l'espressione sotto altra forma:

$$(A + B\bar{C})A + \bar{C}$$

Semplificando ottengo:

$$AA + AB\bar{C} + \bar{C}$$

Semplificando e raccogliendo \overline{C} :

$$A + \overline{C}(AB + 1)$$

Secondo la proprietà di Minimo e Massimo diventa:

$$A + \overline{C}$$

Si ottiene lo stesso risultato raccogliendo dall'espressione precedente il termine A

$$A + AB\overline{C} + \overline{C}$$

$$A(1 + B\overline{C}) + \overline{C}$$

Che secondo la proprietà di Minimo e massimo diventa:

$$A + \overline{C}$$

La tabella di verità risulta quindi:

A	B	C	\overline{C}	OUT
0	0	0	1	1
0	0	1	0	0
0	1	0	1	1
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	1	1
1	1	1	0	1

Es.5

Si costruisca la tabella di verità della seguente espressione booleana in tre variabili, badando alla precedenza tra gli operatori logici.

$$A \text{ or not } C \text{ and } (B \text{ or not } A)$$

Soluzioni

$$A + \overline{C}(B + \overline{A})$$

Data la semplicità dell'espressione, si può procedere direttamente con la scrittura della tabella di verità

A	B	C	\overline{C}	$B + \overline{A}$	OUT
0	0	0	1	1	1
0	0	1	0	1	0
0	1	0	1	1	1
0	1	1	0	1	0
1	0	0	1	0	1
1	0	1	0	0	1
1	1	0	1	1	1
1	1	1	0	1	1

Es. 6

Si stabilisca il minimo numero di bit sufficiente a rappresentare in complemento a due i numeri $A = -46_{10}$ e $B = 97$, li si converta, se ne calcolino la somma $(A+B)$ e la differenza $(A-B)$ in complemento a due e si indichi se si genera riporto sulla colonna dei bit più significativi e se si verifica overflow. Non si accetteranno soluzioni senza il procedimento.

Soluzione

Iniziamo calcolando il numero di bit necessari per convertire in complemento a due i numeri. In complemento a due, con n bit, si possono codificare i numeri

$$\text{da } -2^{n-1} + 1 \text{ a } 2^{n-1} - 1$$

Avendo i numeri da convertire e volendo calcolare il numero minimo di bit, usiamo la formula inversa, ovvero il logaritmo in base 2:

$$n = \log_2(\text{dec}) + 1$$

$$\log_2(46) + 1 = 6,5 \rightarrow \text{Servono quindi 7 bit}$$

$$\log_2(97) + 1 = 7,6 \rightarrow \text{ovvero 8 bit}$$

Iniziamo convertendo il numero 46d

46	
23	0
11	1
5	1
2	1
1	0
0	1

46d —> 101110b

Possiamo verificare il risultato moltiplicando ogni bit per il suo valore decimale e somando:

$$2+4+8+32 = 46$$

Per ottenere -46 in binario, occorre utilizzare il complemento a due; come visto precedentemente, essendo il numero negativo, si deve aggiungere uno zero a sinistra del numero e calcolare l'opposto.

Procediamo per step:

101110b

Aggiungo lo zero:

0101110b

Calcolo l'opposto (con il metodo veloce, tengo i bit partendo da destra fino al primo 1 compreso, e inverte gli altri):

-46d = 1010010b

Contando il numero totale di bit ci si accorge che è proprio quello descritto nella pagina precedente, 7 bit.

Oltre ad usare il logaritmo possiamo quindi prima convertire il numero e poi controllare quanti bit servono. Tuttavia questo procedimento è più soggetto ad errore, in quando se il numero convertito risulta sbagliata, anche il numero di bit lo è di conseguenza.

Procediamo con il numero 97d

97	
48	1
24	0
12	0
6	0
3	0
1	1
0	1

97d risulta 1100001b.

Procediamo anche con la conversione di -97d, perché ci servirà per la sottrazione.

Aggiungiamo lo zero a sinistra e procediamo con l'opposto ottenendo il numero:

-97d = 1001111b

Possiamo procedere con la somma. Dalla operazione in colonna sotto si nota che il numero 97d convertito in binario ha 1 bit in più del numero -47d. Prima di procedere con la somma vanno aggiunti i bit mancanti. Essendo -47d un numero negativo in complemento a due, vanno aggiunti tanti 1 quanto basta per raggiungere il numero di bit dell'altro numero.

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

Si può procedere quindi con la somma:

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

Gli addendi sono discordi, per cui il risultato è sempre corretto e codificabile con lo stesso numero di bit degli addendi. Possiamo verificare il risultato riconvertendolo in decimale:

$$-46 + 97 = 51$$

Procediamo con la sottrazione allo stesso modo. Anche in questo caso al numero -46 va aggiunto un 1 a sinistra, per ottenere lo stesso numero di bit del numero -97. Si usa il complemento a due del numero -97d perché fare la sottrazione di due numeri è come sommare il negativo del secondo addendo.

In questo caso l'overflow è palese: la somma di due numeri concordi negativi dà infatti un risultato positivo. Otto bit riescono infatti a codificare numeri in complemento a due da -128 a +127. -46-97 dà infatti -143, che non è codificabile con 8 bit.

Riporto	1	1		1		1		
		1	1	1	0	1	0	0
		1	1	0	0	1	1	1
		1	0	1	1	0	1	1