Università di Trieste

Laurea in ingegneria elettronica e informatica

Enrico Piccin - Corso di Reti Logiche - Prof. Stefano Marsi

Anno Accademico 2022/2023 - 6 Ottobre 2022

Indice

6 Ottobre 2022

1 Sistemi di numerazione e codici

Di seguito si espone la definizione di **sistema di numerazione**:

> **SISTEMA DI NUMERAZIONE**
>
> Un **sistema di numerazione** è un **insieme di simboli** (cifre) e regole, le quali consentono di associare ad una stringa di cifre il corrispondente valore numerico.
> I codici decimale, binario, ottale o esadecimale sono tutti codici posizionali, il cui valore dipende dalla posizione delle cifre.

Osservazione: La base 2 è la più piccola possibile, in cui i bit sono associati agli stati ON/OFF. Le basi 8 e 16, invece, permettono rappresentazioni più compatte del numero binari, soprattutto perché il passaggio da base 2 a base 8 o 16 e viceversa è particolarmente facile

$$55_10 = 110111_2 \tag{1}$$

$$110111_2 = 37_16 = 67_8 \tag{2}$$

1.1 Conversione tra basi diverse di numeri interi

La conversione da base 10 a base 2, prevede di adottare il metodo delle **divisioni successive**: si divide ripetutamente il numero per la base voluta fino ad ottenere un quoziente nullo e si memorizzano i resti (la seq. dei resti ordinata rappresenta la notazione).
Per quanto detto, il passaggio da basi B a B^n e viceversa risulta particolarmente semplice:

$$157_10 = 10011101_2 = 235_8 = 9D_16$$

Osservazione: Si osservi che per convertire un numero da base 2 a base 10, non solo è possibile usare le potenze del due, ma è anche possibile partire dal bit più significativo e moltiplicarlo per 2, sommarlo al bit successivo e moltiplicare per 2, e via dicendo fino ad esaurire tutti i bit.

1.2 Conversione tra basi diverse di numeri frazionari

Com'è noto, la virgola distingue le cifre che vanno moltiplicate per la base B con esponente positivo da quelle con esponente negativo, per cui

$$0.101 = 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = 0.5 + 0.125 = 0.625$$

Oppure si può anche traslare di 3 posizioni la virgola (che in binario vuole dire moltiplicare per $2^3 = 8$) e convertire il numero binario come se fosse intero e poi dividerlo ancora per $2^3 = 8$, per cui

$$0.101 \quad \rightarrow \quad 0.101 \cdot 2^3 = 101 = 5 \quad \rightarrow \quad 0.101 = \frac{5}{2^3} = 0.625$$

Se, invece, bisogna passare da decimale a binario, si deve procedere per moltiplicazioni successive:

$$0.375_10 \quad \rightarrow \quad 0.375 \cdot 2 = 0.750 \quad \rightarrow \quad \mathbf{0} + 0.750$$

$$0.750_10 \quad \rightarrow \quad 0.750 \cdot 2 = 1.500 \quad \rightarrow \quad \mathbf{1} + 0.500$$

$$0.500_10 \quad \rightarrow \quad 0.500 \cdot 2 = 1.000 \quad \rightarrow \quad \mathbf{1} + 0.000$$

Ecco, quindi, che il valore binario è stato ottenuto:

$$0.375_10 = 0.011_2$$

Osservazione: Ovviamente, è possibile che il processo sopra descritto cada in una ripetizione periodica. Allora, il processo di approssimazione può avvenire secondo due modalità:

1. Per **troncamento**, in cui si lascia semplicemente il valore binario ottenuto così com'è;
2. Per **arrotondamento**, in cui si considera il bit immediatamente successivo all'ultimo di quelli che si sta considerando e lo si somma all'ultima cifra, come mostrato di seguito:

$$011011|\boxed{1}01101 \rightarrow 011011 + \boxed{1} = 011100$$

Non sorprende, poi, osservare che se con una base una notazione frazionaria richiede un numero finito di cifre, potrebbe richiederne infinite con una diversa notazione.

1.3 Aritmetica binaria

L'addizione binaria è molto semplice, mentre la sottrazione risulterebbe particolarmente ostica, a meno che non si considerasse la complementazione. Infatti, dovendo eseguire, in decimale, la differenza $123 - 73$, è sufficiente eseguire la somma $123 + \text{comp}_{10}(73)$, in cui $\text{comp}_{10}(73)$ si calcola come segue

$$\begin{array}{cccc} 9 & 9 & 9 & - \\ & 7 & 3 & + \\ & & 1 & = \\ \hline 9 & 2 & 7 & \end{array}$$

per cui si ottiene $123 + \text{comp}_{10}(73) = 123 + 927 = 1|050 \rightarrow 50$, eliminando l'1 del migliaio, in quanto aggiunto prima per la complementazione. Analogamente in binario.

Osservazione: Si osservi che, per ogni base B, esistono due complementi per un numero N:

- Complemento a B, definito come $C_B = B^n - N$
- Complemento a $B - 1$, definito come $C_{B-1} = B^n - 1 - N$

Non solo, ma dalla differenza di N_1 ed N_2, vi possono essere due casi:

- $N_1 \geq N_2$: il risultato risulta maggiore o uguale a B^n, che pertanto va eliminato dal risultato finale (eliminazione dell'1 più significativo oltre il range del numero stesso)
- $N_1 < N_2$: il risultato risulta minore di B^n, e deve essere inteso come complemento a B (pertanto rappresentante di un numero negativo) del risultato. Per conoscerne il valore assoluto, è necessario ri-complementarlo.

Si consideri, infatti, l'esempio seguente, in cui si esegue la differenza $21 - 46$, ovvero:

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

In cui non è stato ottenuto un 1 nell'ultima operazione finale, pertanto il risultato 110111 deve essere ulteriormente complementato, ottenendo $011001_2 = 25$, che è il valore assoluto della differenza.

1.4 Rappresentazione dei numeri negativi

I numeri negativi possono pertanto essere rappresentati in base al loro complemento a B. In base $B = 10$, ciò non risulta essere usuale, ma si preferisce impiegare un segno grafico $-$.
In binario, ciò non risulta essere possibile, in cui i numeri negativi vengono rappresentati in base al loro complemento a 2, usando il bit più significativo viene impiegato come **bit di segno**:

1	0	1	1	1

in cui la convenzione sul bit di segno è

- **0**: numero positivo
- **1**: numero negativo

Attenzione che, eliminato il bit di segno in un numero binario

- nel caso di un numero positivo (quindi avendo eliminato il bit 0), i restanti numeri rappresentano il numero stesso:
$$01001_2 = +9_{10}$$
- nel caso di un numero negativo (quindi avendo eliminato il bit 1), i restanti numeri rappresentano il numero complementato:
$$11001_2 = -C_2(1001) = -0111_2 = -7_{10}$$

1.5 Errori nei risultati

Il risultato di un'operazione somma/sottrazione è coerente solo se il risultato non esce dal range dei numeri rappresentabili, per cui

- il risultato è **corretto** se
  - non si è avuto alcun riporto, nè nel bit di segno nè fuori dalla parola;
  - si sono avuti riporti in entrambi;
- il risultato è **errato** se si è avuto un solo riporto, o sul segno, o fuori dalla parola;

Dal punto di vista circuitale, per determinare se si è ottenuto un risultato corretto o meno, sarà sufficiente considerare il bit di riporto sul segno e quello fuori dalla parola e porli in XOR: se lo XOR produce come uscita 1, allora si è verificato un errore, altrimenti il risultato è corretto.

7 Ottobre 2022

Non deve sorprendere che un numero in una base non risulta essere periodico, mentre in altre basi esso lo è, in quanto le frazioni a disposizione sono differenti a seconda della base stessa; per esempio, le frazioni a disposizione per la base 3 sono

$$3^{-1} = \frac{1}{3}, 3^{-2} = \frac{1}{9}, 3^{-3} = \frac{1}{27}$$

ed ecco che quindi $\frac{1}{3}$, in base 3, si rappresenta come $0,1_3$.
Inoltre, se si adotta una notazione **unsigned** su n bit, i possibili numeri rappresentabili sono 2^n, da 0 a $2^n - 1$. Invece, se si adotta una notazione **signed** su n bit, i numeri rappresentabili sono i valori da 0 a $2^{n-1} - 1$, e da -1 a -2^{n-1}; tuttavia l'intervallo è il medesimo.
Nella tabella seguente si espongono i valori interi **signed** su 4 bit e i corrispondenti valori decimali se si considera la notazione con la virgola **signed** su 4 bit (utilizzando 2 bit dopo la virgola), ottenendo lo schema seguente:

Intero					Virgola
7	0	1	1	1	1.75
6	0	1	1	0	1.50
5	0	1	0	1	1.25
4	0	1	0	0	1.00
3	0	0	1	1	0.75
2	0	0	1	0	0.50
1	0	0	0	1	0.25
0	0	0	0	0	0.00
-1	1	1	1	1	-0.25
-2	1	1	1	0	-0.50
-3	1	1	0	1	-0.75
-4	1	1	0	0	-1.00
-5	1	0	1	1	-1.25
-6	1	0	1	0	-1.50
-7	1	0	0	1	-1.75
-8	1	0	0	0	-2.00

1.6 Moltiplicazione e Divisione

La moltiplicazione binaria è molto semplice, e segue la regola seguente:

- $0 \cdot 0 = 0$
- $1 \cdot 0 = 0$
- $0 \cdot 1 = 0$
- $1 \cdot 1 = 1$

e si basa sull'automatismo dello **shift & add**, come mostrato nel seguito:

$$\begin{array}{cccc|c} & 1 & 1 & 0 & \times \\ & & 1 & 0 & = \\ \hline & 0 & 0 & 0 & + \\ 1 & 1 & 0 & & = \\ \hline 1 & 1 & 0 & 0 & \end{array}$$

e la divisione viene eseguita, di solito, per sottrazioni successive, particolarmente semplice da meccanizzare tramite automatismi informatici.

1.7 Casting

Quando si considera un numero binario, risulta fondamentale capire se il valore risulta essere unsigned oppure signed; la differenza è fondamentale perché i due valori

$$\begin{array}{ccccc} 1 & 0 & 0 & 0 & 0 \\ \hline 0 & 1 & 1 & 1 & 1 \end{array}$$

in una notazione unsigned differirebbero solamente di Δ minimo, mentre se si trattasse di uan notazione signed, essi sarebbero agli opposti della scala numerica: il primo rappresenta il massimo numero negativo, mentre il secondo è il massimo numero positivo.
Appurata la notazione prescelta, si distinguono le seguenti casistiche

- Nel caso di notazione **unsigned**
  - Per aumentare il numero di cifre decimali, si aggiungono in coda degli 0
  - Per aumentare il numero di cifre intere, si aggiungono in testa degli 0
  - Per ridurre il numero di cifre decimale, si considera il primo bit oltre il range prescelto, e lo si somma all'ultimo bit del range prescelto.
  - Per ridurre il numero di cifre intere, se essi sono 0 non si altera la rappresentazione del numero. Se essi sono 1, si possono scegliere due opzioni: si segnala l'allarme di overflow, oppure si rappresenta il massimo valore possibile con il range di bit a disposizione.
- Nel caso di notazione **signed**:
  - Se il numero è **positivo**
    * Per aumentare il numero di cifre decimali, si aggiungono in coda degli 0
    * Per aumentare il numero di cifre intere, si aggiungono in testa degli 0 (si replica il bit di segno)
    * Per ridurre il numero di cifre decimale, si considera il primo bit oltre il range prescelto, e lo si somma all'ultimo bit del range prescelto.
    * Per ridurre il numero di cifre intere, se essi sono 0 non si altera la rappresentazione del numero. Se essi sono 1, si possono scegliere due opzioni: si segnala l'allarme di overflow, oppure si rappresenta il massimo valore possibile con il range di bit a disposizione (saturazione).
  - Se il numero è **negativo**:
    * Per aumentare il numero di cifre decimali, si aggiungono in coda degli 0
    * Per aumentare il numero di cifre intere, si aggiungono in testa degli 1 (si replica il bit di segno)
    * Per ridurre il numero di cifre decimale, si considera il primo bit oltre il range prescelto, e lo si somma all'ultimo bit del range prescelto.
    * Per ridurre il numero di cifre intere, se essi sono 1 non si altera la rappresentazione del numero. Se essi sono 0, si possono scegliere due opzioni: si segnala l'allarme di overflow, oppure si rappresenta il massimo valore possibile con il range di bit a disposizione (saturazione).

Pertanto, nel caso di notazione **signed**, si replicano i bit di segno per aumentare i bit di rappresentazione, si eliminano senza problemi i bit in testa se essi coincidono con il bit di segno. Per quanto riguarda la parte decimale, non c'è differenza: per aumentare il range di rappresentazione si aggiungono 0, per l'arrotondamento si considera il primo bit oltre il range prescelto, e lo si somma all'ultimo bit del range prescelto.

Osservazione: Nel caso di notazione **signed**, il più piccolo valore positivo è 00000.0001, mentre il più piccolo valore negativo è 11111.1111: in un oscilloscopio, quindi, anche il più piccolo rumore fa saltare l'onda da 00000.0001 a 11111.1111.

1.8 Codici

Un codice è un **insieme di parole** $\mathcal{C}$ adottato per rappresentare gli elementi di un insieme $\mathcal{C}^*$.
I **simboli** sono gli elementi costituenti le parole di codice, mentre la **codifica** è la procedura di associazione di una parola di $\mathcal{C}$ a un elemento di $\mathcal{C}^*$. A tal proposito, si distinguono:

- **Codice non ambiguo**, in cui la corrispondenza tra una parola di $\mathcal{C}$ e un elemento di $\mathcal{C}^*$ è **univoca**;
- **Codice ambiguo**, in cui almeno una parola di $\mathcal{C}$ rappresenta 2 o più elementi di $\mathcal{C}^*$.

Osservazione: Si osservi che se vi sono k simboli, n elementi e le parole sono di lunghezza l, allora il numero di combinazioni possibili è k^l: appare evidente che per non avere ambiguità deve essere che

$$N \leq k^l \quad \to \quad \log_k(N) \leq l$$

Pertanto, se

- se $l = \log_k(N)$, allora il codice si dice **efficiente**;
- se $l > \log_k(N)$, allora il codice si dice **ridondante**;
- se $l < \log_k(N)$, allora il codice si dice **ambiguo**;

Osservazione: Si osservi che il motivo principale per impiegare un codice ridondante è quello di rendere l'informazione **robusta al rumore**.

1.8.1 Codici efficienti

Alcuni codici efficienti sono, per esempio, i codici su 4 bit, in cui si rappresentano i numeri decimali da 0 a 9. Ovviamente, impiegando 4 bit, si avrebbero complessivamente 16 configurazioni distinte, per cui 6 configurazioni non sono utilizzate.
Di questi codici se ne espongono 3 tipologie:

- **Codice BCD**, il quale è un codice ponderato (detto anche codice 8421); tale codice impiega la tabella di codifica seguente:

0	0000	9	1001
1	0001	8	1000
2	0010	7	0111
3	0011	6	0110
4	0100	5	0101

  Tale codice viene impiegato per codificare i numeri da visualizzare in display 7 segmenti, attivando alcuni LED e disattivandone altri. Infatti, dovendo rappresentare il numero 137 su un display 7 segmenti, si converte 137 in BCD, ottenendo

  $$0001\_0011\_0111$$

  e successivamente ogni quattro bit vengono convertiti nella corrispettiva sequenza di 0 e 1 per il comando dei LED. Si capisce facilmente che il codice BCD viene impiegato per semplificare l'interfaccia uomo-macchina.
  Le operazioni in BCD vengono svolte esattamente come in binario; tuttavia, se il risultato dell'operazione eccede il massimo valore rappresentabile in BCD, ossia il 9, si deve sommare 6.
  Se si dovesse eseguire la somma $136 + 247 = 383$ in BCD, si procederebbe nel modo seguente:

0000 0000 1100	
0001_0011_0110	+
0010_0100_0111	=
0011_0111_1101	

Tuttavia, dal momento che 1101 non è un valore nel range BCD, si somma 0110_2 in binario, per cui si ottiene

0000 0001 1000	
0011_0111_1101	+
0000_0000_0110	=
0011_1000_0011	

Ecco che si è ottenuto il risultato previsto: 383 codificato in BCD.

- **Codice eccesso tre**, il quale è un codice in cui ogni numero decimale viene codificato come in binario, ma aggiungendo il valore 3, ottenendo un un codice auto-complementante; tale codice impiega la tabella di codifica seguente:

0	0011	9	1100
1	0100	8	1011
2	0101	7	1010
3	0110	6	1001
4	0111	5	1000

Le operazioni di somma in eccesso a 3 vengono svolte in modo molto simile al binario: una volta codificate le cifre da 0 a 9 in eccesso a 3, si sommano bit a bit e si ottiene un risultato che deve essere sempre corretto, a differenza del BCD; la correzione prevede di sommare 3 ad una quartina se vi è stato un riporto, sottrarre 3 (o sommare 13 senza tenere conto dell'ultimo riporto) se non vi è stato riporto.

- **Codice Aiken** (o 2421), anch'esso auto-complementante e ponderato; tale codice impiega la tabella di codifica seguente:

0	0000	9	1111
1	0001	8	1110
2	0010	7	1101
3	0011	6	1100
4	0100	5	1011

11 Ottobre 2022

I codici efficienti sono codici per cui non vi è ridondanza; i principali sono BCD, eccesso a 3 e Aiken, ossia dei codici che permettono di rappresentare su 4 bit ciascun digit di un numero decimale, usando l'opportuna codifica.
Non solo, ma i codici eccesso a 3 e Aiken sono anche auto-complementanti, ma sempre nell'ottica del complemento a 10: infatti, dato 0 in eccesso a 3, codificato come 0011, per ottenere il suo rispettivo complementare in eccesso a 10, ossia 9, è sufficiente invertire bit a bit, ottenendo 1100 e sommare 1. La stessa cosa per il codice Aiken.

Esercizio 1: Si esegua l'operazione $47 + 35$ in BCD. La prima cosa da fare è codificare gli addenti in codice BCD, ottenendo

$$
\begin{aligned}
47_{10} =& 0100\_0111_{\text{BCD}} \\
35_{10} =& 0011\_0101_{\text{BCD}}
\end{aligned}
$$

e si esegue la somma esattamente come in binario, ottenendo

$$
\begin{array}{ll}
0000\ 1110 & \\ \hline
0100\_0111 & + \\ \hline
0011\_0101 & = \\ \hline
0111\_1100 &
\end{array}
$$

Giacché il primo valore eccede il valore 9, il risultato non deve essere corretto andandovi a sommare 6 in binario, per cui

$$
\begin{array}{ll}
1111\ 1000 & \\ \hline
0111\_1100 & + \\ \hline
0000\_0110 & = \\ \hline
1000\_0010 &
\end{array}
$$

Dal momento che, ora, nessun valore eccede 9, si è ottenuto il risultato esatto in BCD, ovvero

$$1000\_0010_{\text{BCD}} = 82_{10}$$

Osservazione: Il processo di correzione tramite l'aggiunta di 6 deve essere implementato come fosse una somma binaria, considerando tutti i riporti che, dalle unità vanno alle decine. Se, dopo una prima correzione, le decine risultano eccedenti il valore 9, si deve ripetere il processo correttivo in modo iterativo. Ciò palesa un problema di automazione del processo noto come **catena del carry**: per ottenere il valore risultante in binario è necessario attendere che la catena del carry si esaurisca, con un tempo che aumenta esponenzialmente all'aumentare nel numero di bit con cui si sta operando.

Esercizio 2: Il codice eccesso 3 è un codice auto-complementante: ciò significa che consente di eseguire operazioni di somma e differenza in modo più semplice rispetto al codice BCD.
A titolo esemplificativo, si esegua l'operazione $47 + 35$ in eccesso a 3. La prima cosa da fare è codificare gli addenti in codice eccesso a 3, ottenendo

$$
\begin{aligned}
47_{10} =& 0111\_1010_{\text{Ecc}_3} \\
35_{10} =& 0110\_1000_{\text{Ecc}_3}
\end{aligned}
$$

e si esegue la somma esattamente come in binario, ottenendo

$$
\begin{array}{ll}
1111\ 0000 & \\ \hline
0111\_1010 & + \\ \hline
0110\_1000 & = \\ \hline
1110\_0010 &
\end{array}
$$

Ora è necessario inevitabilmente correggere il risultato, sempre e comunque, andando a

- sommare 3 se, dopo aver eseguito la somma sui 4 bit, si è avuto un riporto sul 5 bit;
- sottrarre 3 (che equivale a sommare 13 in binario) se, dopo aver eseguito la somma sui 4 bit, NON si è avuto un riporto sul 5 bit;

Dal momento che, in questo caso, dopo aver eseguito la somma sui bit delle unità si è avuto riporto sulle decine, si somma 3 alle unità.
Invece, riporto per le unità, siccome dopo aver eseguito la somma sui bit delle decine, non si è avuto riporto sulle centinaia, si sottrae 3 alle decine, ovvero si somma 13.
Pertanto si ottiene:

$$\begin{array}{ll} 1000\ 0100 & \\ \hline 1110\_0010 & + \\ \hline 1101\_0011 & = \\ \hline 1011\_0101 & \end{array}$$

Si è così ottenuto il valore $1011\_0101_{\mathrm{Ecc}_3}$ che corrisponde a 82_{10}.

Osservazione: Si osservi che nell'operazione di correzione si trascura l'ultimo riporto che eventualmente si dovrebbe verificare.
Non solo, ma siccome la correzione viene eseguita su ogni digit, essa può essere anche svolta in parallelo, a differenza del BCD.

Esercizio 3: Per eseguire le differenze in eccesso a 3 è necessario ricorrere alla complementazione, cosa che risultava ostica per il BCD; il codice eccesso a 3, invece, essendo auto-complementante, non ha di questo problema, in quanto è sufficiente complementare tutti i bit di un digit per ottenere il suo complementare.
A titolo esemplificativo, si esegua l'operazione $47 - 35$ in eccesso a 3. La prima cosa da fare è codificare gli addenti in codice eccesso a 3, ottenendo

$$47_{10} = 0111\_1010_{\mathrm{Ecc}_3}$$
$$35_{10} = 0110\_1000_{\mathrm{Ecc}_3}$$

Siccome deve essere sottratto il valore 35, semplicemente si esegue il suo complementare in eccesso a 3 e si somma 1: si osservi che l'operazione di somma 1 in binario non è formalmente corretta, in quanto bisognerebbe sommare 1 in eccesso a 3 (ovvero sommare 4 in binario) e successivamente correggere la somma sottraendo 3, quanto non si genera riporto: tale formalismo nasce quando si devono complementare degli 0 nelle posizioni **meno significative** che poi diventano 9 e, quindi, sommando 1 in binario non si generano i riporti necessari che si avrebbero sommando 1 in eccesso a 3; la strategia prevede semplicemente di ricopiare gli zero meno significativi in modo inalterato e complementare i digit successivi; dopodiché si dovrà sommare 1 solamente a partire dal primo digit meno significativo che è stato complementato.
Avendo appurato ciò, la procedura descritta permette di ottenere:

$$35_{10} = 0110\_1000_{\mathrm{Ecc}_3} - 35_{10} = \qquad 1001\_0111_{\mathrm{Ecc}_3} + 1 = 1001\_1000_{\mathrm{Ecc}_3}$$

e si esegue la somma esattamente come in binario, ottenendo

$$\begin{array}{lll} 1 & 1111\ 0000 & \\ \hline & 0111\_1010 & + \\ \hline & 1001\_1000 & = \\ \hline & 0001\_0010 & \end{array}$$

Ora è necessario inevitabilmente correggere il risultato, sempre e comunque, andando a

- sommare 3 se, dopo aver eseguito la somma sui 4 bit, si è avuto un riporto sul 5 bit;

- sottrarre 3 (che equivale a sommare 13 in binario) se, dopo aver eseguito la somma sui 4 bit, NON si è avuto un riporto sul 5 bit;

Dal momento che, in questo caso, dopo aver eseguito la somma sui bit delle unità e sui bit delle decine si è avuto un riporto sul digit superiore, si somma 3 in entrambi i casi.
Da notare che è fondamentale che vi sia un riporto sulle centinaia, perché se non ci fosse significherebbe che si è ottenuto un risultato negativo che, quindi, deve essere ulteriormente complementato per ottenere il valore assoluto della differenza.
Pertanto si ottiene:

0110 0100	
0001_0010	+
0011_0011	=
0100_0101	

Si è così ottenuto il valore $0100\_0101_{\mathrm{Ecc}_3}$ che corrisponde a 12_{10}.

Osservazione: Per quanto riguarda il codice Aiken, la somma viene eseguita esattamente come in BCD. Nel caso di Aiken, però, se il digit ottenuto non è conforme al codice stesso, si deve apportare la dovuta correzione, ma solo se non rispetta il codice:

- si somma 6 se non si è avuto riporto;
- si sottrae 6, ovvero si somma 10, se il riporto vi è stato.

1.9 Binary to BCD converter

Si consideri numero 237_{10} convertito in binario:

$$237_{10} = 11101101_2$$

Per convertire tale numero binario in BCD, si considera il numero binario e vi si antepongono delle fasce verticali di 4 bit; ad ogni passaggio, si esegue lo shift verso sinistra del numero binario di partenza e di controlla se all'interno di ogni fascia vi sia contenuto un valore maggiore o uguale a 5 in binario: se questo è il caso, a tale valore vi si somma 3 in binario e si considera la somma risultante come parte integrante del numero di partenza:

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

Ecco, quindi, che si è ottenuto il risultato cercato, ovvero

$$0010\_0011\_0111_{\mathrm{BCD}} = 237_{10}$$

Il risultato non deve sorprendere, in quanto sommare 3 quando si ha un valore maggiore di 5 permette di generare un riporto al passo successivo.

1.10 Codici ridondanti

I codici ridondanti sono molto utili per evidenziare e/o correggere eventuali errori: utilizzando k bit per il controllo e n bit per l'informazione, si ottengono parole di lunghezza

$$m = n + k$$

in cui viene definita **ridondanza** il rapporto tra i bit impiegati per la rappresentazione ed i bit strettamente necessari, ovvero

$$\mathcal{R} = \frac{m}{n} = \frac{n+k}{n} = 1 + \frac{k}{n}$$

In tale contesto, prende il nome di **peso** il numero di bit diversi da 0, mentre si chiama **distanza** il numero di bit per cui 2 configurazioni differiscono: la distanza tra 100_{BCD} e 101_{BCD} è pari a 1, mentre la distanza tra 000_{BCD} e 111_{BCD} è pari a 3.
La **molteplicità d'errore** rappresenta la distanza tra la configurazione trasmessa e quella (non significativa) ricevuta: in questo senso possono essere ricevuti errori singoli, doppi, tripli, etc. . . .
Non da ultimo, la **distanza di Hamming (h)** è la minima distanza tra tutte le possibili coppie di parole di un codice: sono individuabili gli errori con molteplicità minore di h, mentre se h è grande, si può operare una correzione dell'errore (attraverso i cosiddetti codici auto-correttori).

1.11 Probabilità di errore non rilevato

Posta p la probabilità di errore di ogni singolo bit. Allora, la probabilità che una parola si trasformi in un'altra a distanza esattamente r è

$$P_r = p^r \cdot (1-p)^{m-r} \cdot \binom{m}{r}$$

in cui r sono le cifre errate, mentre $m - r$ sono le cifre esatte.
La probabilità che l'errore non sia rilevato dipende da quante configurazioni significative N_r si trovano a distanza "r" dalla parola, per cui

$$P_{tr} = P_{sr} \cdot p^r \cdot (1-p)^{m-r} \cdot \binom{m}{r} \quad \text{ove} \quad P_{sr} = \frac{N_r}{\binom{m}{r}}$$

La probabilità di errore non rilevato è la somma per ogni r (singolo, doppio, triplo, etc.), ovvero

$$P_t = \sum_{h}^{m} N_r \cdot p^r \cdot (1-p)^{m-r} \cong N_h \cdot p^h$$

in quanto, tipicamente, $p << 1$, e quindi l'unico termine della sommatoria che effettivamente ha peso è solo il primo, quando $r = h$. Da notare che la sommatoria di tutte le possibile distanze delle parole trasmesse parte da $h =$ distanza di Hamming, in quanto le parole a distanza minore della distanza di Hamming vengono rilevate.

1.12 Codice a controllo di parità

Nel controllo di parità, ai vari bit che compongono la parola si aggiunge un ulteriore bit (ridondante), secondo la regola seguente:

- tale bit è 0 se il peso della parola è pari

- tale bit è 1 se il peso della parola è dispari

La parola risultante sarà **sempre a peso pari**. In questo modo, la distanza di Hamming aumenta di 1; se la distanza di partenza era pari a 1, diventando pari a 2, tale metodo consente di rilevare tutti gli errori di molteplicità dispari.

13 Ottobre 2022

Un errore (eventualmente multiplo) di trasmissione può non essere rilevato se il numero degli errori che si sono verificati fa sì che da una parola di codice se ne ottenga un'altra, ugualmente significativa, ma avente un contenuto informativo differente da quella di partenza.
Al fine di ridurre il più possibile la probabilità di non rilevare un errore (eventualmente multiplo) durante una trasmissione, quello che si fa è aumentare al massimo la distanza di Hamming tra le parole di codice.
Un primo metodo per aumentare di 1 la distanza fra le parole è il controllo di parità, in cui ai vari bit che compongono la parola si aggiunge un ulteriore bit (ridondante), al fine di rendere la parola **sempre a peso pari**. In questo modo, la distanza di Hamming aumenta di 1; se la distanza di partenza era pari a 1, diventando pari a 2, tale metodo consente di rilevare tutti gli errori di molteplicità dispari (ma già un errore doppio non viene più rilevato).

Esempio 1: Dato un codice a 7 bit, di cui 6 di informazioni e 1 di parità, per un totale di 128 parole: 64 di peso pari e significative, 64 di peso dispari e non significative. In questo senso si ha una ridondanza

$$\mathcal{R} = \frac{7}{6} \cong 1.16$$

Per ogni parola il numero di parole che distano 2 sono:

$$N_h = \binom{7}{2} = 21$$

in quanto per avere un'altra parola del codice si devono commutare 2 bit su 7.
Supponendo $p = 0.01$, la probabilità di errore non rilevato è

$$P_t = N_h \cdot p^h = 21 \cdot 0.01^2 = 0.21\%$$

In pratica coincide con la probabilità che vi sia un errore di molteplicità 2 (ma solo solo perché tutte le configurazioni a distanza 2 sono significative).

Esempio 2: Nel caso in cui tutte le parole hanno lo stesso peso w, in cui la distanza di Hamming permane, ovviamente, $h = 2$, si stanno considerando **codici a peso costante**.
Se m è la lunghezza di ogni parola

- le parole significative saranno $\binom{m}{w}$
- Mentre le configurazioni non significative saranno $2^m - \binom{m}{w}$
- Per ogni parola, il numero di parole a distanza 2 (ammissibili) è

  $$N_2 = w \cdot (m - w)$$

  in quanto per avere una parola significativa non è possibile commutare due 0, in quanto il peso della stessa si altererebbe. Bisogna, quindi, commutare necessariamente un 1 in 0 in w modi ed uno 0 in 1 in $m - w$ modi.

Non basta, quindi, che vi sia un errore doppio, ma questo deve portare anche in un'altra configurazione significativa.

Esempio 3: Considerando un codice a peso costante 2 e a lunghezza 5, il numero totale delle parole che si possono costruire è

$$n = \binom{5}{2} = 10$$

Dal momento che per indirizzare 10 parole servono 4 bit, il numero di bit strettamente necessari per l'informazione è proprio 4. Da ciò segue che la ridondanza è

$$\mathcal{R} = \frac{5}{4} = 1.25$$

il numero di parole a distanza 2 è, evidentemente, $w \cdot (m-w) = 2 \cdot (5-2) = 6$ per cui la probabilità di errore non rilevato, supposta la probabilità di errore sul singolo bit pari a $p = 0.01$ è

$$P_t = N_h \cdot p^h = 6 \cdot 0.01^2 = 0.06\%$$

Esempio 4: Si consideri un codice bi-quinario, ovverosia un codice a peso costante 2 e che presenta un doppio controllo di parità: un bit di parità sui primi 2 e sugli ultimi 5 bit. Il numero totale delle parole che si possono costruire è

$$n = \binom{2}{1} \cdot \binom{5}{2} = 2 \cdot 5 = 10$$

Analogamente a quanto visto in precedenza, per indirizzare 10 parole servono 4 bit, ma il codice bi-quinario ne prevede 7. Da ciò segue che la ridondanza è La ridondanza, ovviamente, è

$$R = \frac{7}{4} = 1.75$$

in quanto, per quello che si è detto, i bit utilizzati sono 7, quando ne sarebbero sufficiente 4.
Ovviamente si ha che il numero di parole a distanza 2 è $N_2 = 5$, in quanto si possono commutare i due bit iniziali in un solo modo possibile e ciò genera una sola nuova parola di codice; ma anche i 5 bit finali possono essere commutati, in 4 possibili modi e ciascuno di essi produce una nuova parola di codice. Pertanto, probabilità di errore non rilevato, supposta la probabilità di errore sul singolo bit pari a $p = 0.01$ è

$$P_t = N_h \cdot p^h = 5 \cdot (0.01)^2 = 0.05\%$$

in quanto le configurazioni significative a distanza 2 da ogni parola sono **solo 5**.

Osservazione: Sembra che nonostante si sia abbondantemente aumentata la ridondanza (3 bit in più), non ci sia una significativa riduzione della probabilità di errore. Ciò non deve sorprendere; infatti, preso ad esempio un codice composto solo da 2 parole di molte cifre 01000000000000 e 00100000000000, la probabilità che una passi sull'altra è circa p^2, ovvero la probabilità che la seconda cifra sia codificata errata e che lo sia anche la terza.
Si potrebbe fare molto meglio aumentando la distanza tra le due parole sfruttando la ridondanza, scrivendo

00000000000000 e 11111111111111

ossia due parole a distanza 14. Ciò non solo rende molto più robusto il codice, ma permette anche di intuire la giusta parola: se si ottiene una parola con 2 zeri e 12 uni, probabilmente la parola trasmessa originariamente conteneva tutti 1, in quanto la probabilità à sbagliare 2 uni è molto maggiore che sbagliare 12 zeri.
Ridondanza NON è sinonimo di robustezza. Un codice può essere ridondante senza essere robusto!

1.13 Codici di Hamming

I codici di Hamming sono codici con $h = 3$ o $h = 4$ usati come **rilevatori/auto-correttori di errore**.

- La molteplicità di errore rilevabile è $r < h - 1$;
- La molteplicità di errore correggibile $c < \frac{h}{2}$.

Pertanto un codice di Hamming con $h = 3$ è in grado di rilevare errori doppi e di correggere errori singoli.
Dato un codice efficiente ad n bit vi si aggiungono k bit di controllo che controllano la parità di gruppi di bit specifici. I bit aggiunti si posizionano alla posizione 2^b e, in particolare:

- bit 1: controllo di parità per $1, 3, 5, 7, 9, 11, 13, 15, 17, \ldots$
- bit 2: controllo di parità per $2, 3, 6, 7, 10, 11, 14, 15, \ldots$

- bit 4: controllo di parità per $4, 5, 6, 7, 12, 13, 14, 15, \ldots$
- bit 8: controllo di parità per $8, 9, 10, 11, 12, 13, 14, \ldots$

secondo lo schema seguente

1		3		5		7		9		11		13		15
	2	3			6	7			10	11			14	15
			4	5	6	7					12	13	14	15
							8	9	10	11	12	13	14	15

In ricezione si verifica la parità per ogni gruppo e si scrive 0 se verificata, 1 se non verificata. Il risultato (letto in binario) darà la posizione del bit errato.

Osservazione: È facile capire che la commutazione di un bit della parola comporta la commutazione di almeno due bit di parità (in quanto ogni bit è coperto da almeno due bit di parità, ma alcuni anche da 3, se non da 4), per cui la distanza minima tra le parole diviene 3.
Non solo, ma è bene notare che non tutte le parole sono distanti 3 tra loro, ma tutte le parole sono sicuramente distanti almeno 3 l'una dall'altra; le altre distano molto di più.
L'autocorrezione, inoltre, viene adottata a seconda della tipologia di informazione trasmessa: se una sequenza di bit rappresenta un messaggio estremamente importante, una volta rilevato l'errore è possibile richiedere la ritrasmissione invece che provare a correggere i bit, in quanto non è detto che si apporti una correzione esatta; se, invece, la sequenza di bit rappresenta un flusso di streaming, allora è possibile procedere alla correzione, decisamente meno onerosa rispetto ad una ritrasmissione.

Esempio: Si voglia trasmettere 0101; allora, per il funzionamento del codice di Hamming, si trasmetterà

$$b_1\, b_2\, 0\, b_4\, 1\, 0\, 1$$

in cui ovviamente

- $b_1 = 0$
- $b_2 = 1$
- $b_4 = 0$

Si trasmetterà, pertanto la stringa 0100101. Si supponga, invece, ci ricevere 0101101; allora, rieseguendo il controllo di parità si ottiene che

- $b_1 = 0$
- $b_2 = 0$
- $b_4 = 1$

Ecco che, quindi, si verificato un errore nella posizione $100_2 = 4_{10}$. Lo si sarebbe anche potuto notare lavorando a gruppi: sicuramente, infatti, l'errore si è manifestato negli ultimi 4 bit controllati dal bit di parità b_4; non solo, ma deve essere anche un bit che non viene controllato dai bit di parità b_1 e b_2, sennò essi sarebbero stati errati. Se ne deduce che l'unico bit a poter essere errato è proprio quello in posizione 4.

1.13.1 Efficienza codice di Hamming

Per il corretto funzionamento del codice di Hamming deve essere verificata la condizione

$$m \leq 2^k - 1$$

in cui $m = n + k$ è la dimensione totale della parola trasmessa, con k bit di controllo e n bit di informazione.

Si dicono **ottimi** i codici in cui per la relazione di cui sopra è verificata con il segno uguale, come nel caso in cui i bit di controllo sono 2 e il bit di informazione è uno solo. Allora le due uniche parole a distanza 3 e di lunghezza 3 che rispettano il codice di Hamming sono

$$\boxed{0}\boxed{0}0 \quad \text{e} \quad \boxed{1}\boxed{1}1$$

In questo modo si riescono ad individuare errori singoli e doppi e a correggere gli errori singoli (ovviamente non può essere noto a priori se l'errore è singolo, oppure doppio, ma a livello probabilistico l'errore più probabile è quello singolo).

1.13.2 Codice di Hamming a distanza 4

Esistono anche codici di Hamming con distanza $h = 4$ (vi è un ulteriore bit di parità globale, per cui si rilevano errori singoli, doppi e tripli e si correggono quelli singoli).

Esempio: Si consideri un codice di Hamming a 7 bit, ossia un codice con 3 bit di controllo e 4 di informazione. Allora, nell'ipotesi cautelativa che tutte le parole si trovano a distanza 3, il numero totale delle parole cercato è

$$N_3 = \binom{7}{3} = 35$$

Tuttavia non tutte risultano essere significative per il codice. Per semplificare la stima, siccome il numero di bit di informazione è pari a 4, il numero totale delle parole che si possono costruire è $2^4 = 16$. Pertanto, data una parola, essa avrà al più 15 parole nel suo universo attorno. Allora, facendo una maggiore e supponendo l'errore sul singolo bit pari a 0.01, se ne conclude che

$$P_3 \leq 15 \cdot 0.01^3 = 15 \times 10^{-7}\%$$

Se il codice di Hamming fosse stato a 8 bit, con 4 di informazione e 4 di controllo, allora il numero di parole a distanza 4 da una parola qualsiasi sarebbe stato non maggiore di 15, ma la probabilità di errore non rilevato, supponendo l'errore sul singolo bit pari a 0.01, sarebbe

$$P_4 \leq 15 \cdot 0.01^4 = 15 \times 10^{-10}\%$$

14 Ottobre 2022

Si esegua la differenza in eccesso a 3 tra 47 e 32, ottenendo:

$$
\begin{aligned}
&0111\_1010+\\
&1000\_1011 =\\
&0000\_0101+\\
&1011\_1011 =\\
&1011\_1000
\end{aligned}
$$

18 Ottobre 2022

20 Ottobre 2022

L'Algebra Booleana è nata per studiare problemi di logica deduttiva e prevede la presenza di 2 soli elementi, 0 e 1, rappresentabili tramite delle **variabili logiche**, ossia grandezze che assumono solo i valori 1 o 0.
In tale contesto, una **funzione logica** rappresenta la dipendenza di una grandezza logica da altre grandezze logiche. È immediato evincere che le funzioni logiche in n variabili sono finite e pari 2^{2^n}.
In funzione logica è rappresentabile con una "tabella di verità" in essa vi si contemplano tutti i casi possibili

1.14 Termini minimi

I termini minimi sono dei termini che valgono 1 solo per una certa configurazione degli ingressi. I termini minimi si ottengono come prodotto di tutte le variabili (di cui alcune dirette ed alcune negate). Per esempio, si hanno:

$$\overline{x_1} \cdot x_4 \quad \text{e} \quad x_3 \cdot \overline{x_2}$$

È importante osservare che

- Ogni funzione può essere rappresentata come somma di termini minimi, denominata **I forma canonica**;
- Ogni funzione è esprimibile come somma di prodotti;

1.15 Termini massimi

I termini massimi sono dei termini che valgono 0 solo per una certa configurazione degli ingressi. I termini massimi si ottengono come somma di tutte le variabili (di cui alcune dirette ed alcune negate). Per esempio, si hanno:

$$\overline{x_1} + x_4 \quad \text{e} \quad x_3 + \overline{x_2}$$

È importante osservare che

- Ogni funzione può essere rappresentata come prodotto di termini massimi, denominata **II forma canonica**;
- Ogni funzione è esprimibile come prodotto di somme.

1.16 Teoremi fondamentali

Di seguito si espongono alcuni fondamentali teoremi alla base della semplificazione e della comprensione dell'Algebra Booleana.

1.16.1 Principio di dualità

Tutti i postulati fino ad ora esposti, possono essere accoppiati tra loro e si può ottenere l'uno dall'altro pur di effettuare le seguenti sostituzioni

- ogni 1 deve divenire uno 0 e viceversa;
- ogni prodotto diviene una somma e viceversa.

Per esempio si ha che

$$(x_1 \cdot x_2) + (x_1 \cdot x_3) = x_1 \cdot (x_2 + x_3)$$
$$(x_1 + x_2) \cdot (x_1 + x_3) = x_1 + (x_2 \cdot x_3)$$

1.16.2 Teoremi dell'assorbimento

I teoremi dell'assorbimento si dividono in:

1. Primo teorema dell'assorbimento:

$$x + xy = x \cdot (1 + y) = x$$

2. Secondo teorema dell'assorbimento:

$$x + \overline{x}y = x \cdot (1 + y) + \overline{x}y = x + xy + \overline{x}y = x + (x + \overline{x}) \cdot y = x + y$$

3. Terzo teorema dell'assorbimento:

$$xy + yz + \overline{x}z = xy\overline{x}z$$

1.17 Teorema di De Morgan

Il teorema di De Morgan stabilisce la seguenti uguaglianze:

$$\overline{x + y} = \overline{x} \cdot \overline{y} \quad \text{e} \quad \overline{x \cdot y} = \overline{x} + \overline{y}$$

ovvero, generalizzando, si ottiene che

$$\overline{F(x_1, x_2, \ldots, +, \cdot)} = F(\overline{x_1}, \overline{x_2}, \overline{\ldots}, \cdot, +)$$

Ovvero la negazione di una funzione si ottiene negando le sue variabili e scambiando tra loro gli operatori di somma e prodotto.

Osservazione: Tramite il teorema di De Morgan si può verificare l'equivalenza fra le forme canoniche:

$$y = \sum y_i \cdot m_i = \overline{\overline{y}} = \overline{\sum \overline{y_i} \cdot m_i} = \prod(y_i + \overline{m_i}) = \prod(y_i + M_i)$$

si può verificare che l'insieme completo degli operatori necessario e sufficiente per rappresentare qualsiasi funzione logica non è AND, OR, NOT ma solamente AND e NOT oppure OR e NOT

$$x + y = \overline{\overline{x + y}} = \overline{\overline{x} \cdot \overline{y}} \qquad\qquad x \cdot y = \overline{\overline{x \cdot y}} = \overline{\overline{x} + \overline{y}}$$

1.18 Teorema di Shannon

Il teorema di Shannon afferma che è sempre possibile esprimere una funzione booleana come segue

$$f(x_1, x_2, \ldots, x_n) = x_1 \cdot f(1, x_2, \ldots, x_n) + \overline{x_1} \cdot f(0, x_2, \ldots, x_n)$$

Ciò è vero in quanto

- se $x_1 = 1$ allora

$$f(1, x_2, \ldots, x_n) = 1 \cdot f(1, x_2, \ldots, x_n) + 0 \cdot f(0, x_2, \ldots, x_n)$$

- se $x_1 = 0$ allora

$$f(0, x_2, \ldots, x_n) = 0 \cdot f(1, x_2, \ldots, x_n) + 1 \cdot f(0, x_2, \ldots, x_n)$$

1.19 Funzioni universali

Le funzioni NAND e NOR sono anche dette **funzioni universali**, in quanto tramite esse si può realizzare qualunque funzione logica:

$$\overline{x} = x|x \qquad \overline{x} = x \downarrow x$$
$$x \cdot y = \overline{x|y} \qquad x \cdot y = \overline{\overline{x} + \overline{y}} = \overline{x} \downarrow \overline{y}$$
$$x + y = \overline{\overline{x} \cdot \overline{y}} = \overline{x}|\overline{y} \qquad x + y = \overline{x \downarrow y}$$

1.20 Semplificazione di funzioni

Di seguito si espongono alcuni fondamentali concetti impiegabili nella semplificazione di funzioni booleane:

- **Letterale**: è la coppia variabile-valore; ad ogni variabile sono associati a letterali (a ed $\overline{a}$);
- **Implicante** di una funzione
  $$f(x_1, \ldots, x_n)$$
  è il prodotto di letterali
  $$P = x_i \cdot \cdots \cdot x_k$$
  in forma diretta o negata, tale per cui se $P = 1$ anche $f = 1$;
- **Termine minimo**: è implicante ove compaiono tutte le variabili, ovvero è un punto nello spazio booleano della funzione dove la funzione vale 1;
- **On set** di una funzione: è l'insieme dei suoi termini minimi;
- **Termine massimo**: è un punto nello spazio booleano della funzione dove la funzione vale 0;
- **Off set** della funzione: è l'insieme di tutti i punti dello spazio booleano della funzione che non sono termini minimi;
- **Implicante**: è un sotto-cubo di soli 1 nello spazio booleano della funzione;
- **Implicante primo**: è un implicante che è contenuto in altri implicanti;
- **Implicante essenziale**: è un implicante che contiene almeno un 1 non incluso in altri implicanti primi;
- **Copertura** di una funzione: è l'insieme di implicanti che coprono tutti i termini minimi.

Osservazione: Ovviamene, due funzioni sono equivalenti se hanno la stessa tavola di verità e per semplificare una funzione possono essere impiegate differenti strategie:

- attraverso le relazioni fondamentali e i teoremi dell'Algebra Booleana;
- individuando termini implicanti
- tramite le **Mappe di Karnaugh** (forma minima a 2 livelli, ma prevalentemente somma di prodotti), ovvero delle tabelle toroidali che rappresentano in più dimensioni la tabella di verità della funzione;
- attraverso il **Metodo tabellare di Quine-McCluskey** (forma minima a 2 livelli), basato sull'applicazione sistematica del teorema di Shannon
  $$f \cdot x + f \cdot \overline{x} = f$$

Ma naturalmente vi possono essere forme più "economiche" (a più livelli) per realizzare una funzione che questi metodi non evidenziano.

1.21 Condizioni non specificate (Don't care)

In una realtà circuitale, vi possono essere condizioni di ingresso che non si verificano: ad esempio, se gli ingressi dipendono a loro volta da una rete logica.
La presenza di tali condizioni può essere ben impiegata per generare ulteriori semplificazioni: infatti, una condizione non specificata (*Don't care*) si può considerare diretta o negata, in funzione di quale fornisce la miglior semplificazione.

1.22 Simmetria di funzioni

Le funzioni simmetriche possono essere implementate facilmente, ma esclusivamente, con tecniche particolari; infatti, non funzionano i normali metodi di semplificazione.
Nell'ambito della simmetria di funzioni si distinguono funzioni

- **totalmente simmetriche**, in cui un qualsiasi scambio tra le variabili lascia immutato il risultato; l'intercambiabilità può essere anche tra variabili dirette e negate (anche se meno evidente);
- **parzialmente simmetriche**, in cui la proprietà di cui sopra è limitata ad un sottoinsieme di variabili;
- **simmetriche indipendenti**, in cui la simmetria esiste solo per un sottoinsieme della funzione

In tutti i casi esposti in precedenza, le variabili interessate possono essere dirette, negate o miste.

27 Ottobre 2022

Se una funzione f di partenza viene scomposta tramite il teorema di Shannon in due funzioni f_1 e f_2 tale che

$$f = x \cdot f_1 + \overline{x} \cdot f_2$$

e si verifica che le funzioni f_1 e f_2 sono simmetriche nelle medesime variabili, allora la funzione f di partenza si dice parzialmente simmetrica e possono essere rappresentate tramite un unico sommatore e un circuito combinatorio che riorganizzi le uscite.
Potrebbe accadere che le due sotto-funzioni di partenza risultino essere simmetriche in variabili differenti, allora la funzione si dice separatamente simmetrica.

1.23 Funzioni decomponibili

Una funzione si dice decomponibile quando si riesce a riconoscere che essa può essere scomposta in più sotto-funzioni.
Tale approccio segue il **binomio area-tempo**: migliorando alcuni aspetti della funzione se ne peggiorano degli altri. Infatti, riducendo al minimo possibile la complessità della funzione e del numero di componenti circuitali richiesti per la realizzazione della stessa, aumento il numero di livelli del circuito, con più porte in cascata e un aumento della latenza.

Osservazione: Per riconoscere se una funzione può essere decomposta o meno, si impiega il teorema di Shannon. In particolare, si supponga che la funzione f in 4 variabili che può essere scomposta due volte con Shannon, come mostrato di seguito

$$f(a, b, c, d) = ab \cdot g_1(c, d) + a\overline{b} \cdot g_2(c, d) + \overline{a}b \cdot g_3(c, d) + \overline{a}\overline{b} \cdot g_4(c, d)$$

Ma se si osserva che ciascuna di ogni funzione può essere espressa in funzione di un'unica altra funzione, come, per esempio

$$g_1 = g_1 \quad \text{e} \quad g_2 = \overline{g_1} \quad \text{e} \quad g_3 = 0 \quad \text{e} \quad g_4 = 1$$

si può realizzare un selettore che a seconda della configurazione di a e b selezioni l'uscita richiesta.

$ab - cd$	00	01	11	10
00	1	0	1	1
01	0	1	0	1
11	0	0	0	0
10	1	1	1	1

Allora è possibile osservare come la prima riga si possa esprimere in funzione di cd, ottenendo α, la seconda è $\overline{\alpha}$, la terza riga è 0, la quarta è 1.
Se la scomposizione della funzione tramite il teorema di Shannon vale sempre, la decomposizione non è sempre realizzabile. Con Shannon, infatti, si è scomposta una funzione a 4 variabili, per un totale di 16 combinazione-celle, in 4 funzioni da 4 combinazioni-celle, per cui non si è guadagno niente.
Ma per capire se una funzione è decomponibile bisogna realizzare tutte le possibile mappe, organizzate con ogni combinazione delle 4 variabili a disposizione, ponendo, quindi, come riga-colonna $ab - cd$, $ac - bd$, $ad - bc$, $bc - ad$, $bd - ac$ e $cd - ab$, ma anche quando vi è una sola variabile in colonna.

1.23.1 Mappe di decomposizione

Per decretare se una funzione è decomponibile, si impiegano le cosiddette **mappe di decomposizione**. A titolo esemplificativo, si consideri la funzione seguente

$$G = ab\overline{c} + ab\overline{d} + \overline{a}bd$$

Allora si considerino i termini di cui è composta, e si determinino i termini minimi da essi generati

$$\begin{array}{ccccc}
ab\overline{c} & \rightarrow & 110- & \rightarrow & \text{termini minimi generati: } 12,13 \\
ab\overline{d} & \rightarrow & 11-0 & \rightarrow & \text{termini minim generati: } 12,14 \\
\overline{a}bd & \rightarrow & 01-1 & \rightarrow & \text{termini minimi generati: } 5,7
\end{array}$$

E una volta fatto ciò si considerano le mappe di decomposizione e si evidenziano i termini appena individuati, appartenenti all'insieme

$$\mathscr{Z} = \{5,7,12,13,14\}$$

Se si riesce a identificare una possibile decomposizione, per cui le righe di ogni mappa possono essere espresse in funzione di una funzione α e delle sue possibili varianti, $\overline{\alpha}$, 0 e 1, allora si procede con la relativa decomposizione.
Nel caso considerato della funzione G, decomponendo rispetto alla variabile b, si osserva che

$b-f$	0	1
0	0	0
1	0	1

riconoscendo, quindi, la funzione AND, da cui, banalmente, la possibilità di esprimere G come $G = b \cdot f$, cosa che poteva essere già vista in principio.