Università degli Studi di Verona

CORSO DI LAUREA IN INFORMATICA

Architettura degli Elaboratori

Federico Brutti
federico.brutti@studenti.univr.it

"Bits are bits, oramai lo sapete... o forse dovrei ricordarvi il discorso sui CFU?" - Franco F.

Contents

Chapter 1

Codifica dell'Informazione

1.1 Sistemi ed elaborazione dati

Cominciamo col dire che lo scopo dell'informatica è la risoluzione dei problemi attraverso insiemi di istruzioni non ambigue, ovvero gli **algoritmi**. In questa materia ci occuperemo di studiare la progettazione ed ottimizzazione di sistemi digitali tramite programmi di **sintesi logica**[1] e rivolgeremo in seguito l'attenzione al linguaggio Assembly, per una corretta comprensione delle funzionalità di un'architettura. Ad oggi esistono due macrocategorie di architetture:

- **Sistemi embedded**: Macchine composte puramente da hardware, capaci di eseguire un solo algoritmo.
- **Sistemi general purpose**: Macchine composte dal connubio hardware-software, capaci di eseguire diversi algoritmi.

Generalmente, ogni sistema operativo lavora con il **linguaggio macchina** o codice oggetto; si tratta di una sequenza di "0" ed "1" con un significato specifico per la macchina e per essere leggibile dalle persone è necessaria una traduzione. Poniamoci quindi la domanda: "In che modo è possibile passare informazioni dal mondo reale ad un computer?"; stiamo parlando di un processo di due passi:

- **Input**: Le informazioni vengono prima recepite dalla macchina per la loro codifica e poi inviate al sistema operativo per l'elaborazione.
- **Output**: L'elaborato viene decodificato per risultare leggibile alle persone e poi mostrato all'utente.

[1] Comunicazione da algoritmo a hardware.

Figure 1.1: Ciclo di elaborazione informazioni

Questo vale come discorso generale; nello specifico è giusto chiarire che, avendo risorse limitate, non è possibile dare in input infinite informazioni. La soluzione a questo problema si ottiene con il seguente algoritmo:

1. **Campionamento**: Divisione in intervalli dell'informazione registrata.

2. **Discretizzazione**: Approssimazione degli stessi quanto possibile ad un numero leggibile dalla macchina.

Le informazioni sono recepite in un determinato arco di tempo, il quale viene misurato in **Hertz** ($1Hz = 1ms$). Per ogni elaborazione vale inoltre il seguente teorema:

Teorema 1. ***Teorema di Shannon***
Data una funzione in un intervallo di campionamento e discretizzazione, è garantita la presenza di un errore.

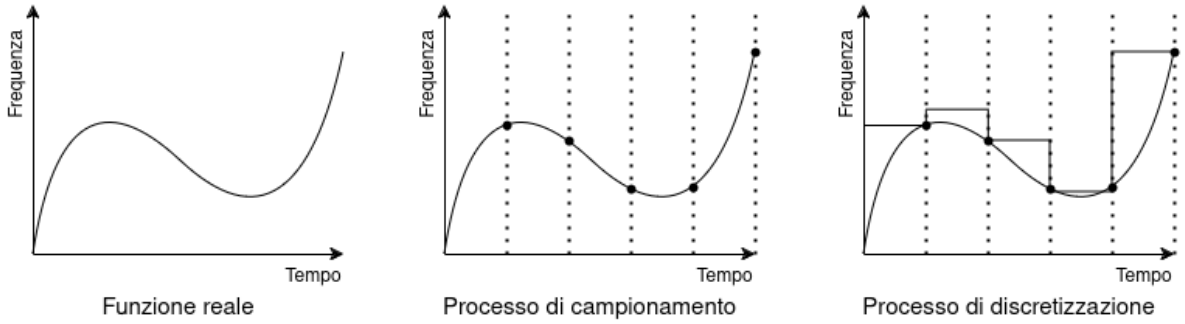


Figure 1.2: Processo di ricezione delle informazioni

Ma in che modo vengono codificate le informazioni? Si tratta di un processo che vede l'assegnazione di un codice binario ad ogni frammento di informazione con un certo numero di *bits*. Con "bit" intendiamo il numero totale di cifre binarie usate per un dato.

Esempio 1. ***Calcolo del numero di bits necessari per salvare informazioni***
Supponiamo di avere $12M$ *unità di dati da registrare; dobbiamo ragionare attraverso le potenze di* 2 *ed ottenere il valore più piccolo che sia maggiore o uguale al numero di dati da registrare.*

In questo caso sarà $2^4 = 16$ *e l'esponente sarà il numero di bits necessari. Per fare ordine:*

- *Da registrare:* $12M = 12 \times 1000000 = 12000000$
- *Potenza corretta:* $2^4 = 16$
- *Numero di bits necessari:* $2^4 \implies 4b$

La codifica non è un procedimento sempre uguale; talvolta risulta necessario modificarlo in base alle specifiche del calcolatore, tra le quali notiamo in particolare la **facilità di calcolo**, che riguarda la semplicità delle istruzioni, e la **velocità di elaborazione**, la quale influenza la codifica. Se quest'ultima richiedesse più tempo rispetto alla frequenza di campionamento si perderebbero dei dati e per questo deve essere sempre maggiore o uguale al valore del campionamento.

Per il momento prenderemo in considerazione i **circuiti combinatori**, dove ad ogni codifica binaria è associata un'informazione e sequenze identiche verranno elaborate come la stessa informazione. In questo caso: **bits are bits**.

1.2 Codice Binario e operazioni in base 2

Il codice binario è fondamentalmente la "lingua" in cui è scritto il linguaggio macchina; come detto prima è una sequenza di 0 e 1 che rappresenta un'informazione. In particolare, il bit all'estremità sinitra è detto **più significativo**, mentre quello al lato opposto è il **meno significativo**.

È grazie alla codifica binaria che è possibile elaborare informazioni come immagini, musica e video, ma soprattutto numeri e caratteri, i quali hanno il codice **ASCII** con uno standard che vede i primi $127b$ comuni a tutte le lingue.

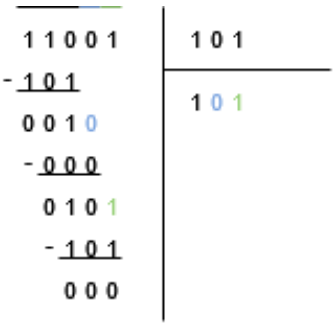
Ma per quale motivo stiamo considerando solo le potenze del 2? Procediamo a fare un collegamento: il codice binario ha solo *due* cifre utilizzabili, quindi bisognerà ragionare in loro funzione. Noterai, per esempio, che per $2b \implies 2^2 = 4$ puoi esprimere quattro combinazioni di numeri diverse senza ripetizione; estendendo il ragionamento a valori più alti avrai capito il funzionamento.

Lavorare in una base diversa dal 10 non comporta modifiche nella logica; infatti sono presenti tutte le operazioni elementari, come segue:

```
 - Addizione -              - Sottrazione -           - Moltiplicazione -
 0   0       0              0   0       0             0   0       0
```

0	1	1	0	1	1 (Carry-in)	0	1	0
1	0	1	1	0	1	1	0	0
1	1	0 (Carry-out)	1	1	0	1	1	1

La divisione può risultare contro-intuitiva a causa dell'utilizzo della sottrazione. Vediamo con un esempio: $\frac{11001}{101} = \frac{25}{5}$.



Bisogna poter sottrarre il divisore al dividendo quando questo "sta dentro" al primo.
Abbassa 110 e sottraigli 101 perché il divisore ci sta una volta. Otterrai 001, al quale dovrai aggiungere la cifra successiva del dividendo e scriverai "1" come prima cifra del risultato.
Osserva che 10 non ci sta in 101, quindi non gli si può sottrarre nulla e scriverai "0" come seconda cifra del risultato, procedendo ad abbassare l'ultima cifra del dividendo ed ottenere il numero 101 che, guarda un pò, è uguale al divisore e quindi ci sta dentro. $101 - 101 = 000$, è una divisione intera senza resto. Scrivi "1" come ultima cifra del risultato e hai finito.

Esistono anche altre due operazioni, utili per la *codifica in virgola mobile*, la quale vedremo nelle prossime sezioni, e per velocizzare moltiplicazione e divisione:

- **Shift Left**; Aggiunge uno zero alla fine del numero (Sposta tutte le cifre a sinistra di una posizione). $1101 \times SL = 11010$.
- **Shift Right**; Sposta le cifre a destra ed aggiunge uno 0 a sinistra. $1101 \times SR = 0110$.

1.3 Codifica dei numeri

In questa sezione vedremo come codificare i numeri e le particolarità di ogni algoritmo che svolge tale funzione. Si lavora con cifre binarie con la regola della *Notazione posizionale*, dove il valore di un numero è dato dalla posizione delle sue cifre. Iniziamo con la **codifica standard**.

Questo è un algoritmo utile per lavorare con numeri interi positivi. Bisogna prendere la potenza del 2 più grande che si avvicina al numero da codificare, ma che non lo supera, per poi sottrarla all'altro. Ripetere fin quando il numero iniziale non è "0".

Esempio 2. ***Codifica del numero 683***

```
683 - 512 = 171            Valore 1 in posizione bit 9
```

```
171 - 128 = 43              Valore 1 in posizione bit 7
43 - 32 = 11                Valore 1 in posizione bit 5
11 - 8 = 3                  Valore 1 in posizione bit 3
3 - 2 = 1                   Valore 1 in posizione bit 1
1 - 1 = 0                   Valore 1 in posizione bit 0
```

Codifica standard di 683*:* 1010101011*.*

1.3.1 Codifica in modulo e segno

La codifica in modulo e segno non è molto dissimile dalla precedente; infatti l'unica differenza è l'utilizzo di un ulteriore bit nella parte più significativa per marcare il segno positivo "0" o negativo "1".

Esempio 3. ***Codifica del numero -227***

```
227 - 128 = 99              Valore 1 in posizione bit 7
99 - 64 = 35                Valore 1 in posizione bit 6
35 - 32 = 3                 Valore 1 in posizione bit 5
3 - 2 = 1                   Valore 1 in posizione bit 1
1 - 1 = 0                   Valore 1 in posizione bit 0
```

Ottenuta la codifica standard; 11100011 *aggiungiamo il bit del segno:*
$227 = 011100011 \implies -227 = 111100011$*.*

1.3.2 Codifica in virgola fissa

La codifica in virgola fissa è un algoritmo capace di tradurre i numeri razionali considerando separatamente parte intera e decimale. Abbiamo La virgola rimane nella stessa posizione della base 10.

in primo luogo bisogna codificare la parte intera come una normale codifica in modulo e segno, mentre per trovare quella decimale bisogna moltiplicare per 2 il numero. Se il risultato è maggiore o uguale a 1, si scrive 1 e si verifica la stessa condizione per la parte decimale risultante. Di norma è specificato quanti bit di precisione deve avere la parte decimale, perché spesso troverai numeri periodici (dove trovi cifre decimali uguali in verifica) e andresti avanti all'infinito.

Se ti vuoi male e vuoi verificare la correttezza del tuo risultato decimale, puoi sommare tutte le potenze negative di 2 e vedere cosa ti esce. Molto probabilmente, un risultato approssimato.

Esempio 4. ***Codifica del numero 56,83 in 3b di precisione***

```
Parte intera: 56                    Parte decimale: 0,83
56 - 32 = 24    1 in bit 5          0,83 × 2 = 1,66     1 in bit -1
24 - 16 = 8     1 in bit 4          0,66 × 2 = 1,32     1 in bit -2
8 - 8 = 0       1 in bit 3          0,32 × 2 = 0,64     0 in bit -3
```

Risultato: $111000,110$

1.3.3 Codifica in virgola mobile

La codifica in virgola mobile o *Floating Point* consente di ottenere numeri particolarmente grandi e piccoli. Risulta utile per avvicinarsi al concetto di numero reale. Un tale valore si esprime nella formula di **Notazione scientifica**:

$$N = \pm Mant \times Base^{\pm exp}$$

Si divide quindi in tre parti a cui è associato un numero specifico di bits da un totale di $32b$ (float) oppure $64b$ (double); le quali sono:

- *Segno*; $1b$.
- *Esponente*; $8b$, oppure $9b$ in doppia precisione.
- *Mantissa*; $23b$, oppure $54b$ in doppia precisione.

Prima di poter lavorare sul numero è necessario **normalizzarlo**, ovvero portarlo in una forma dove rimane una singola cifra intera attraverso le operazioni di shifting a destra o sinistra.

Dipendentemente da quante posizioni sono modificate, sarà necessario sommare, se SR o sottrarre, se SL, tal numero all'esponente. Una volta ottenuto, bisogna sommargli $+127$[2] e hai fatto.

Notare che nella codifica della mantissa la cifra intera non è mai scritta perché è sempre la stessa e si può omettere.

Esempio 5. ***Codifica del numero -30,375 in virgola mobile***

```
1. Convertiamo in binario il numero con la codifica in virgola fissa:
 Parte intera: 30 = 11110            Parte decimale: 0,375 = 011
 30 - 16 = 14                        0,375 × 2 = 0,75
 14 - 8 = 6                          0,75 × 2 = 1,5
 6 - 4 = 2                           0,5 × 2 = 1
```

[2]Questa operazione si chiama **Eccesso 127** ed è necessaria per codificare l'esponente nello standard IEEE754.

```
2 - 2 = 0

Codifica in virgola fissa: 11110,011

2. Procediamo con la normalizzazione:
11110,011 / 1000 = 1,1110011 × 2^4      Sommeremo 4 all'esponente.

La mantissa sarà: 1110011...0.

3. Troviamo l'esponente:
Non c'è un esponente nel numero richiesto, quindi: 1*4 + 127 = 131.

131 - 128 = 3       1 in bit 7
3 - 2 = 1           1 in bit 1
1 - 1 = 0           1 in bit 0

131 = 10000011 - Esponente trovato!

4. Ricomponiamo il tutto
- Segno: 1
- Esponente: 10000011
- Mantissa: 1110011...0
```

La codifica in virgola mobile di $-30,375$ *è:* $1100000111110011...0$

Esempio 6. ***Trasformazione in decimale di 0100011001000110...0***

```
1. Dividiamo nelle varie parti i bits
- Segno: 0
- Esponente: 10001100
- Mantissa: 1000110...0

2. Otteniamo l'esponente decimale
128 + 8 + 4 = 140
140 - 127 = 13 - Esponente trovato!

3. Ricaviamo la mantissa
Considera che ora stai lavorando con cifre decimali, quindi le
potenze del 2 dove sta il valore 1 saranno negative. In questo caso
notiamo che si trovano nelle posizioni -1, -5 e -6, quindi:
```

```
2^-1 + 2^-5 + 2^-6 = 0,547 - Valore della mantissa trovato!

4. Ricostruiamo il decimale
Il segno è positivo, ricorda di sommare 1 alla mantissa trovata e
moltiplica ad essa l'esponente. Hai finito.
```

Risultato: $1 \times 1,547 \times 2^1 3 = 1,547 \times 2^{13}$

Ci sono infine alcuni casi di cifre particolari a cui fare attenzione:

- $+0$; Tutte le cifre sono 0.
- -0; Tutte le cifre sono 0, tranne quella del segno.
- $+\infty$; Esponente massimo, il resto a 0.
- $-\infty$; Esponente massimo e bit segno a 1, il resto a 0.
- **Not a Number**; Qualunque numero superi gli infiniti.
- 2; Tutte le cifre sono 0, tranne il bit più significativo dell'esponente.
- 2^{-145}; Tutte le cifre sono 0 tranne il bit meno significativo. Si tratta del numero più piccolo ottenibile.

1.3.4 Codifica in complemento a 1 e a 2

Parleremo solamente della codifica in complemento a 2 in quanto è un singolo passaggio in più rispetto all'altra.

Il suo scopo principale è dividere a metà il totale delle codifiche ottenibili da $2^n b$, rendendo più semplice ottenere numeri lunghi. Supponiamo di avere a disposizione 4 bits, quindi 16 combinazioni diverse. Per ottenere il complemento ad 1 basta invertire tutte le cifre, mentre per il complemento a 2 bisognerà poi sommare 1 ad ogni combinazione.

Esempio 7. ***Codifica di -3 in complemento a 2 con 4b di precisione***

```
3 = 0011 -> 1100  in compl. ad 1
1100 + 1 = 1101  in compl. a 2
```

Segue una lista indicativa di tutte le codifiche in precisione $4b$ per il complemento ad 1 e 2:

Codifica normale	Compl. ad 1	Compl. a 2
0000 = 0	1111 = -8	1111 = -1
0001 = 1	1110 = -7	1110 = -2
0010 = 2	1101 = -6	1101 = -3
0011 = 3	1100 = -5	1100 = -4
0100 = 4	1011 = -4	1011 = -5
0101 = 5	1010 = -3	1010 = -6
0110 = 6	1001 = -2	1001 = -7
0111 = 7	1000 = -1	1000 = -8

1.3.5 Codifica in esadecimale

Soon!

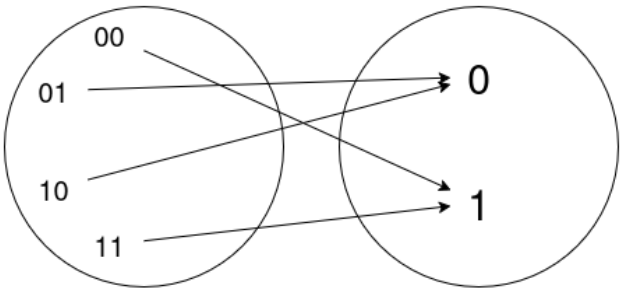
Chapter 2

Circuiti e Ottimizzazione

La domanda principale di questo capitolo è "Come realizzare un sistema digitale?" Ebbene, è necessario un modello apposito che consentirà di rappresentare appropriatamente la sua struttura. Per far ciò useremo l'**algebra di Boole**; uno spazio ad n dimensioni misurate in base all'alfabeto che voglio dare allo spazio. Qui sono presenti solo due valori come in base binaria: 0 e 1.

Secondo Boole, se si definisce una funzione che genera valori in un altro spazio, generalmente scritta $f(B^n) -> B^m$, questa potrà essere rappresentata tramite gli operatori elementari; in pratica ci puoi fare qualunque cosa.

Utilizzando questo spazio è possibile passare da una scrittura ambigua ad una formale, chiara per quelli che saranno i nostri scopi; la rappresentazione dei sistemi avviene infatti tramite le tabelle di verità, che mostrano le funzioni booleane.



A	B	Out
0	0	1
0	1	0
1	0	0
1	1	1

Figure 2.1: Funzione booleana XNOR

Nella tabella di verità vengono definiti:

- **Onset**: L'insieme dei punti dello spazio di ingresso dove la funzione vale 1. Gli elementi si dicono **mintermini**.

- **Offset**: L'insieme dei punti dello spazio di ingresso dove la funzione vale 0. Gli elementi si dicono **maxtermini**.

L'unione di questi due insiemi è complementare, poiché rappresentano tutto lo spazio usato dalla funzione. Inoltre, per mettere in relazione i bit con gli operatori si utilizzano le seguenti espressioni:

$$m_3 = a \times b, m_0 = !a \times !b$$

In tal merito, definiamo **letterale** una qualunque coppia {variabile, Valore} ed è l'unità di misura usata per definire la complessità di un circuito. Infine, la funzione in output si scrive attraverso una somma di prodotti o somma di min/maxtermini, per esempio: $O = abc + !ac + b!c$ e avremo una complessità di 7 letterali.

2.1 Realizzazione di porte logiche

Le porte logiche e di conseguenza qualunque circuito elettronico sono governati dal flusso di elettricità gestito dai **transistors**; interruttori comandati. Nella tecnologia **CMOS** (Complementary Metal Oxide Semiconductor) sono implementati solo due tipi:

- **Interruttore N**: Se riceve corrente, conduce l'elettricità.
- **Interruttore P**: Se riceve corrente, ferma il flusso.

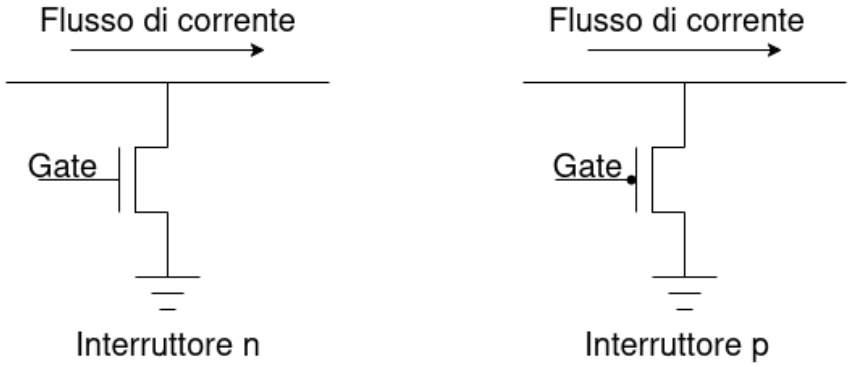


Figure 2.2: Tipi di transistors

Per usare termini corretti, quando un transistor è chiuso e scorre la corrente si dice che è in **conduzione**, mentre se è aperto e la corrente è bloccata diciamo che c'è un segnale di **interdizione**.

Di base i circuiti vanno letti da sinistra a destra, e grazie ai due interruttori appena visti è possibile creare le porte logiche elementari **AND, OR, NOT, NAND, NOR, XOR, XNOR**. Adesso abbiamo tutti gli strumenti per la creazione di un circuito elettronico. Ciò non significa tuttavia che possiamo buttarci senza cognizione di causa; ragion per cui bisognerà ragionare sui costi e le parti effettivamente necessarie al processo di realizzazione. Il nostro scopo da adesso diventa l'**ottimizzazione** dei nostri progetti.

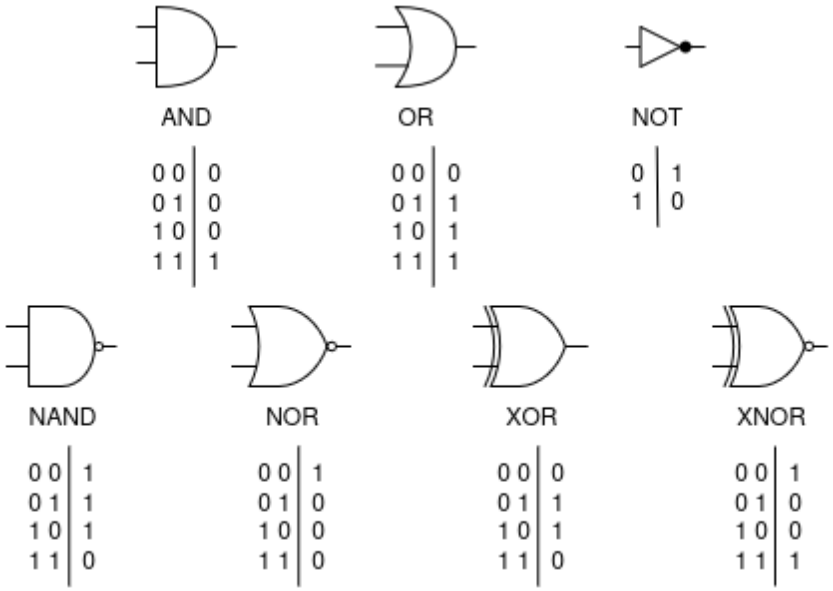


Figure 2.3: Porte logiche elementari

2.2 Minimizzazione a due livelli

Non esiste un solo modo per ottimizzare un circuito, bensì più tecniche, tutte con la loro ragion d'essere in base ai nostri scopi. Una prima semplice regola, per esempio è l'**assorbimento**, da utilizzare assieme all'algebra di Boole. Bisogna innanzitutto tenere a mente queste proprietà:

- Identità: $1 \times x = x$, $0 + x = x$
- Elemento nullo: $0 \times x = 0$, $1 + x = 1$
- Idempotenza: $x \times x = x$, $x + x = x$

- Inverso: $x \times \overline{x} = 0$, $x + \overline{x} = 1$

Valgono anche le proprietà associative, commutative e distributive. Procediamo col dare la definizione della regola:

Definizione 1. ***Regola dell'assorbimento***
Sia una funzione booleana scritta in somma di prodotti; se due di questi sono a distanza di Hamming 1, è possibile sommare le parti inverse ed ottenere il valore 1, riducendoi letterali e quindi la complessità della funzione.

$$x(x + y) = x, x + (xy) = x$$

Esempio 8. *Si ottimizzi la seguente funzione in somma di prodotti:*

$$O = \overline{x}\overline{y}\overline{z} + \overline{x}\overline{y}z + x\overline{y}\overline{z} + x\overline{y}z + xyz$$

Utilizziamo l'assorbimento; per prima cosa bisogna vedere se sono presenti termini a distanza di Hamming 1 per semplificarli; ripetere il processo fin quando non è più possibile.

$$\begin{aligned} O &= \overline{x}\overline{y}\overline{z} + \overline{x}\overline{y}z + x\overline{y}\overline{z} + x\overline{y}z + xyz \\ &= \overline{x}\overline{y}(\overline{z} + z) + \overline{y}\overline{z}(x + \overline{x}) + \overline{y}z(x + \overline{x}) + x\overline{y}(\overline{z} + z) + xz(\overline{y} + y) \\ &= \overline{x}\overline{y} + \overline{y}\overline{z} + \overline{y}z + x\overline{y} + xz \\ &= \overline{y} + \overline{y} + xz \\ &= \overline{y} + xz \end{aligned} \tag{2.1}$$

In parole molto povere, bisogna vedere uno per uno tutti i termini che presentano un solo bit di differenza fra di loro, ovvero a **distanza di Hamming** 1, e rimuovere le due parti inverse, poiché equivarranno ad 1. Nello svolgimento dell'ottimizzazione, chiameremo

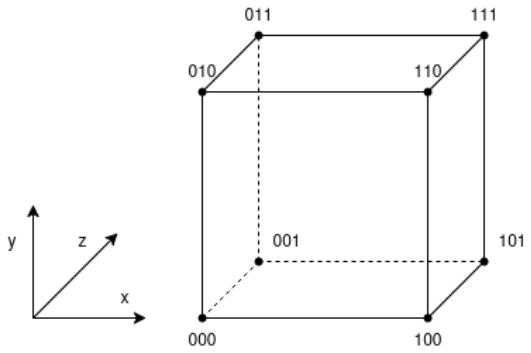


Figure 2.4: Cubo delle distanze fra codifiche

i termini che non è più possibile ridurre come **implicanti primi**; dove è vero che li vogliamo, è possibile che più implicanti primi coprano lo stesso mintermine, creando una ripetizione inutile. Ci è quindi necessario cercare gli **implicanti primi essenziali**, ovvero un implicante che tocca un mintermine non raggiunto dagli altri.

Subentra di conseguenza la questione della **copertura minima**, ovvero il problema di trovare il numero minimo di implicanti necessari per coprire ogni mintermine; per lavorare useremo i due metodi nelle prossime sezioni.

2.2.1 Mappa di Karnaugh

Da me soprannominato "Il Sarrus dell'ottimizzazione", è un algoritmo utilizzabile con funzioni $f(B^3)$ o massimo per $f(B^4)$. Servirà prendere il cubo delle distanze visto prima ed immaginare di piallarlo. Usciranno le celle corrispondenti agli spigoli del solido.

Una particolarità è che il cubo deve essere visto come se fosse impossibile andare out of bounds; quindi una volta superato un lato estremo, ci si trova subito dopo in quello opposto, esattamente come in Pacman. Le celle adiacenti sono quelle a distanza

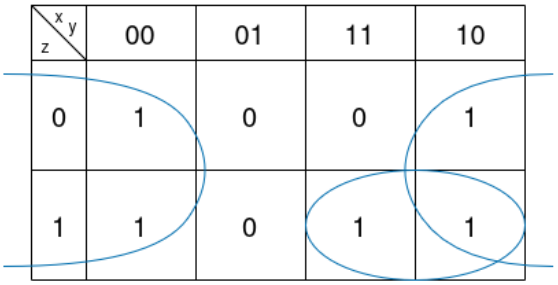


Figure 2.5: Mappa di Karnaugh

di Hamming 1 e la funzione ottimizzata è quella di prima, con risultato $O = \overline{y} + xz$.

2.2.2 Algoritmo di Quine Mc-Cluskey

2.3 Funzioni parzialmente specificate

2.4 Sintesi combinatoria multilivello

2.5 Mapping tecnologico

2.6 Dispositivi programmabili

Programmable Logic Array, Field programmable gate array, system on chip

Chapter 3

Progettazione Digitale

3.1 Circuiti sequenziali

3.1.1 FSM - Finite State Machines

3.2 Sintesi delle funzioni λ e δ, assegnazione stati

3.3 Minimizzazione degli stati

3.4 Datapath e componenti

3.5 Modello FSMD - Finite State Machine with Datapath

3.6 Derivazione FSMD da algoritmo

3.7 Modello dispositivi programmabili

Chapter 4

Architetture dei Calcolatori

4.1 Modello di Von Neumann e Unità funzionali del calcolatore

4.2 CPU - Central Processing Unit

4.2.1 CPU Cablata

4.2.2 CPU Microprogrammata

4.2.3 Microistruzioni della CPU

4.3 Metodi di I/O, Segnale Interrupt

4.4 Direct Memory Access, BUS e Arbitraggio

4.5 Stati di un processo

4.6 Pila e gestione Interrupt

4.7 Tipi di Memoria RAM

Static RAM, Dynamic RAM, Banchi di memoria ed esercizi.

4.8 Caratteristiche delle memorie con relativa gerarchia

4.9 Memoria Cache e Virtuale

4.10 Pipelining

4.11 Architettura LC-3

4.12 Modello CISC e RISC

4.13 Architetture parallele

Chapter 5

SIS e Verilog

5.1 Introduzione a SIS

5.2 Sintesi combinatoria esatta

5.3 Sintesi combinatoria approssimata multilivello

5.4 Modellazione di FSM

5.5 Modellazione di FSMD

5.6 Introduzione a Verilog

5.7 Modellazione in Verilog

5.8 Modellazione di FSM

5.9 Modellazione di FSMD

5 - comprende: Sis - Subsequential interactive synthesis Linguaggi HDL - Verilog

Chapter 6

Il linguaggio Assembly

6.1 Introduzione ad Assembly

6.2 Istruzioni e Sintassi

Stringhe e numeri

6.3 Debugging e Makefile

6.4 Relazione ISA-FSMD su LC-3

6.5 Funzioni e passaggio di parametri

6.6 Confronto con il C