Architettura degli Elaboratori

Algebra booleana e circuiti logici

slide a cura di Salvatore Orlando, Andrea Torsello, Marta Simeoni

Algebra & Circuiti Elettronici

I computer operano con *segnali elettrici* con valori di potenziale discreti

- sono considerati significativi soltanto due *potenziali* (high/low)
- i potenziali intermedi, che si verificano durante le transizioni di potenziale, non vengono considerati

L'aritmetica binaria è stata adottata proprio perché i bit sono rappresentabili naturalmente

- tramite elementi elettronici in cui siamo in grado di distinguere i 2 stati del potenziale elettrico (high/low)

Algebra & Circuiti Elettronici

Il funzionamento dei circuiti elettronici può essere modellato tramite l'Algebra di Boole

- solo 2 valori:
  - valore logico *True* (1 o *asserted*) ⇒ livello di potenziale *alto*
  - valore logico *Falso* (0 o *deasserted*) ⇒ livello di potenziale *basso*
- operazioni logiche Booleane: *somma* (OR), *prodotto* (AND) e *inversione* (NOT) *logica*
  - OR (A+B): risultato uguale ad 1 (true) se almeno un input è 1 (true)
  - AND (A·B): risultato uguale ad 1 (true) solo se tutti gli input sono 1 (true)
  - NOT (~A): risultato uguale all'inverso dell'input (0→1 oppure 1→0)

Blocco logico

- circuito elettronico con linee (*fili*) in *input* e *output*
- possiamo associare variabili logiche con le varie linee in input/output
  - i valori che le variabili possono assumere sono quelli dell'Algebra di Boole

- il circuito calcola una o più funzioni logiche, ciascuna esprimibile tramite una combinazione di operazioni dell'Algebra di Boole sulle variabili in input

Circuiti combinatori/sequenziali

Circuito *combinatorio*

- senza elementi di *memoria* - produce *output* che dipende funzionalmente solo dall'*input*

Circuito *sequenziale*

- con elementi di *memoria* - produce *output* che dipende non solo dall'*input* ma anche dallo *stato* della memoria

Per ora ci concentriamo sui *circuiti combinatori*

Funzioni Logiche

Una funzione logica è completamente specificata da

- una tabella di verità o, equivalentemente, da
- una equazione logica

Vediamo in dettaglio…

Tabelle di Verità

Funzione logica completamente specificata tramite *Tabella di Verità*

Dati n input bit, il numero di configurazioni possibili degli input, ovvero il numero di righe della *Tabella di Verità*, è 2^n

- per ogni bit in output, la tabella contiene una colonna, con un valore definito per ognuna delle combinazioni dei bit in input

Tabella con 3 input A, B e C
e 2 output D ed E

A	B	C	D	E
0	0	0	1	0
0	0	1	0	1
0	1	0	1	0
0	1	1	0	0
1	0	0	0	0
1	0	1	0	0
1	1	0	1	1
1	1	1	0	0

Algebra Booleana

Funzione logica completamente specificata tramite una *Equazione logica* dell'algebra di Boole

Esempio: E = ~A~BC + AB~C

– bit in *input* e *output* rappresentati tramite *variabili logiche* (con valori 0 o 1)

– input *combinati* tramite le operazioni di *somma* (OR), *prodotto* (AND) e *inversione* (NOT) *logica* dell'algebra di Boole

- **OR (A+B): risultato uguale ad 1 (true) se almeno un input è 1 (true)**
- **AND (A·B): risultato uguale ad 1 (true) solo se tutti gli input sono 1 (true)**
- **NOT (~A): risultato uguale all'inverso dell'input (0→1 oppure 1→0)**

Algebra Booleana

Tabelle di verità ed equazioni logiche delle operazioni di NOT, AND, OR:

A	X
0	1
1	0

$X = \sim A$

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

$X = A \cdot B$

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

$X = A + B$

Proprietà dell'algebra di Boole

PROPRIETÀ

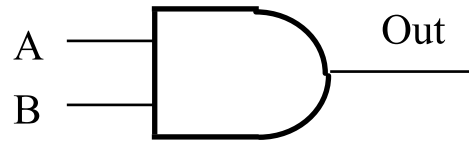
Identità:	$A+0=A$	$A \cdot 1=A$
Nullo:	$A+1=1$	$A \cdot 0=0$
Idempotente:	$A+A=A$	$A \cdot A=A$
Inverso:	$A+(\sim A)=1$	$A \cdot (\sim A)=0$
Commutativa:	$A+B=B+A$	$A \cdot B=B \cdot A$
Associativa:	$A+(B+C)=(A+B)+C$	$A \cdot (B \cdot C)=(A \cdot B) \cdot C$
Distributiva:	$A \cdot (B+C)=(A \cdot B)+(A \cdot C)$	$A+(B \cdot C)=(A+B) \cdot (A+C)$
DeMorgan:	$\sim(A+B)=(\sim A) \cdot (\sim B)$	$\sim(A \cdot B)=(\sim A)+(\sim B)$

Ad esempio, gli output D ed E della precedente *Tabella di verità* possono essere espresse come *Equazioni logiche,* semplificabili applicando le proprietà qui specificate

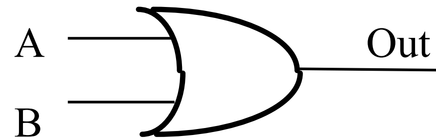
Dalle equazioni logiche ai circuiti combinatori

Porte logiche

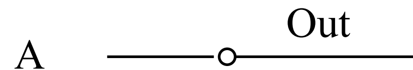
- *AND*: Out = A · B

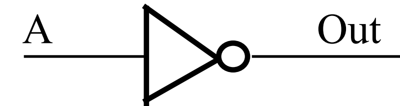


- *OR*: Out = A + B



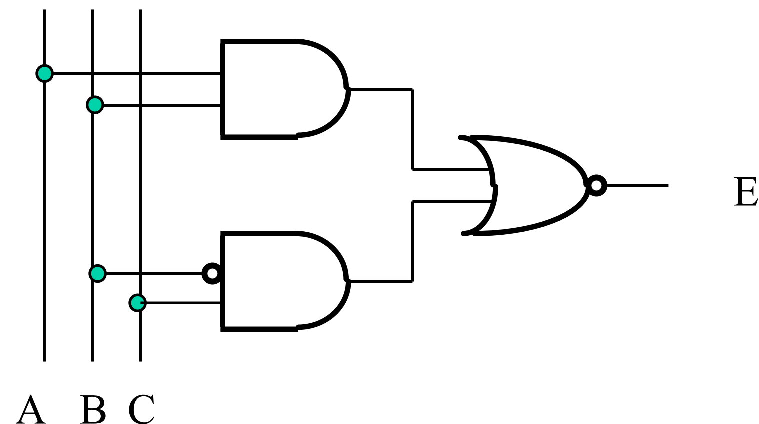
- *NOT*: Out = ~A





Esempio di equazione e corrispondente circuito:

- E = ~((AB) + (~BC))



NAND e NOR

NAND (inverso dell'operazione AND): $\sim(A \cdot B) = A_{\text{NAND}} B$

NOR (inverso operazione OR): $\sim(A + B) = A_{\text{NOR}} B$

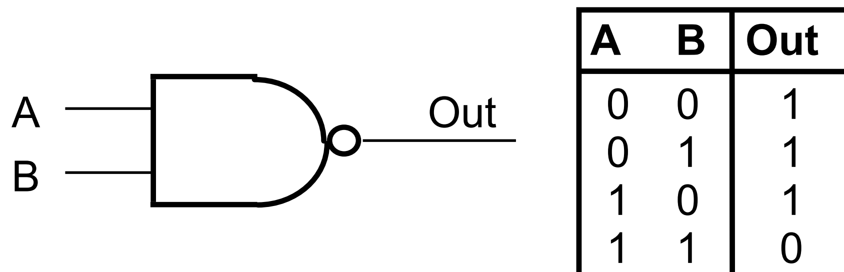
– Si può dimostrare che le operazioni NAND o NOR (e le corrispondenti porte) sono sufficienti per implementare qualsiasi funzione logica

NAND e NOR

Vediamo come realizzare le operazioni fondamentali dell'algebra Booleana NOT, AND e OR con l'operazione NAND:

- $\sim A = \sim A + 0 = \sim(A \cdot 1) = A$ NAND 1
- $A+B = \sim\sim(A+B) = \sim(\sim A \cdot \sim B) = \sim(\sim(A \cdot 1) \cdot \sim(B \cdot 1)) =$
  $= (A$ NAND $1)$ NAND $(B$ NAND $1)$
- $A \cdot B = (A \cdot B)+0 = \sim\sim((A \cdot B)+0) = \sim(\sim(A \cdot B) \cdot 1) =$
  $= ((A$ NAND $B)$ NAND $1)$

NAND: porta e tabella di verità



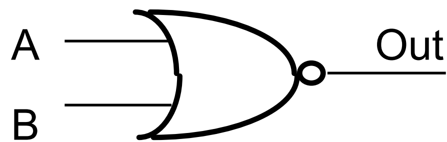
A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

NAND e NOR

Vediamo come realizzare le operazioni fondamentali dell'algebra Booleana NOT, AND e OR con l'operazione NOR:

- $\sim A \quad = \sim A \cdot 1 = \sim(A + 0) = A \text{ NOR } 0$
- $A+B \quad = (A+B) \cdot 1 = \sim\sim((A+B) \cdot 1) =$
  $= \sim(\sim(A+B) + 0) = ((A \text{ NOR } B) \text{ NOR } 0)$
- $A \cdot B \quad = \sim\sim(A \cdot B) = \sim(\sim A + \sim B) =$
  $= \sim(\sim(A+0) + \sim(B+0)) = (A \text{ NOR } 0) \text{ NOR } (B \text{ NOR } 0)$

NOR: porta e tabella di verità



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

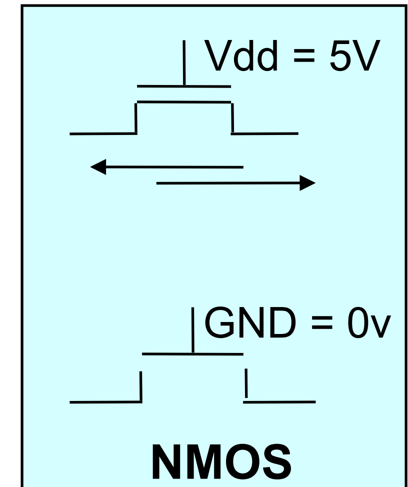
Porte logiche e transistor

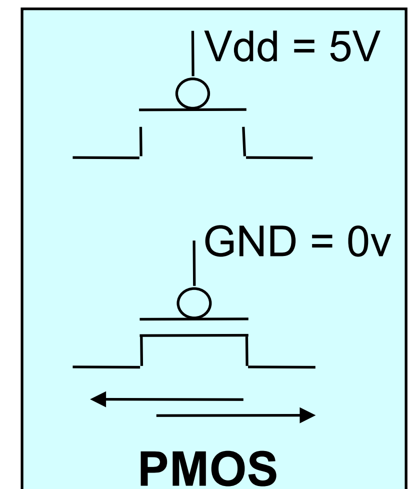
Tecnologia CMOS (Complementary Metal Oxide Semiconductor) per realizzare transistor sul silicio

- I transistor sono degli interruttori velocissimi che lasciano (o meno) passare la corrente, e sono comandati da un segnale elettrico

- NMOS (N-Type Metal Oxide Semiconductor) transistor
  - Se applichi un ALTO voltaggio (Vdd), il transistor diventa un "conduttore"
  - Se applichi un BASSO voltaggio (GND), il transistor interrompe la conduzione (resistenza infinita)

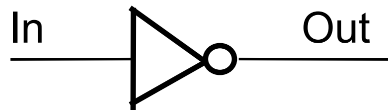
- PMOS (P-Type Metal Oxide Semiconductor) transistor
  - Se applichi un ALTO voltaggio (Vdd), il transistor interrompe la conduzione (resistenza infinita)
  - Se applichi un BASSO voltaggio (GND), il transistor diventa un "conduttore"



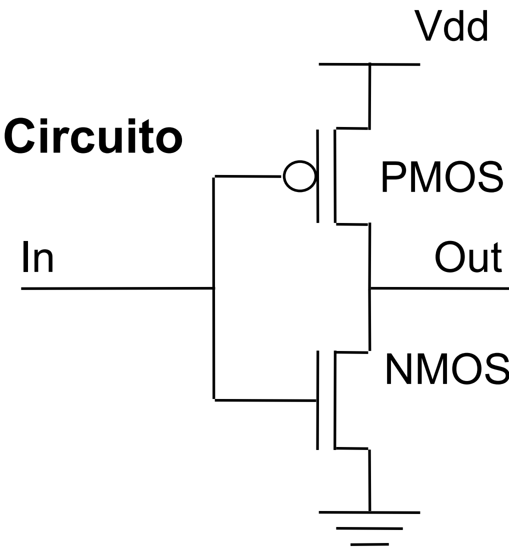


Componenti base: Inverter CMOS

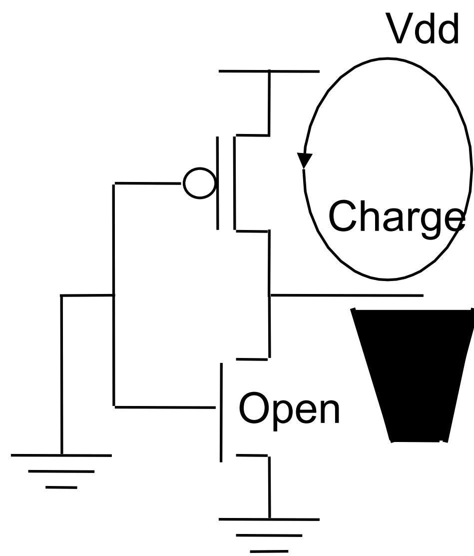
Simbolo

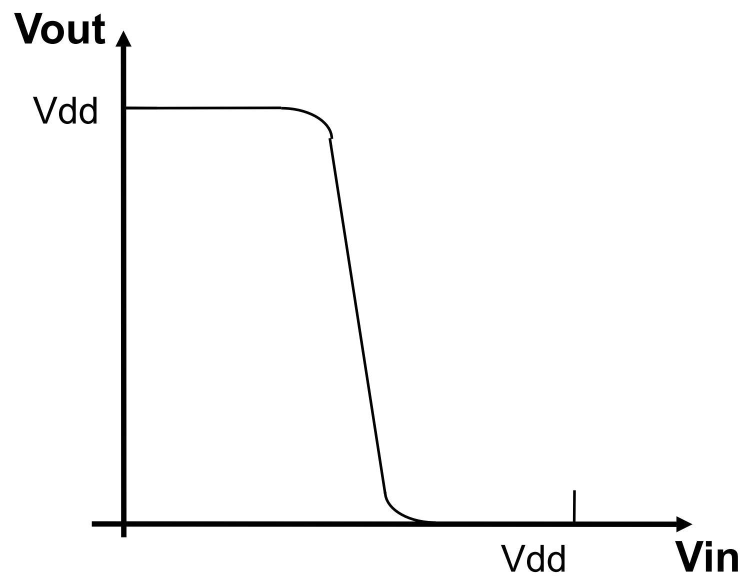


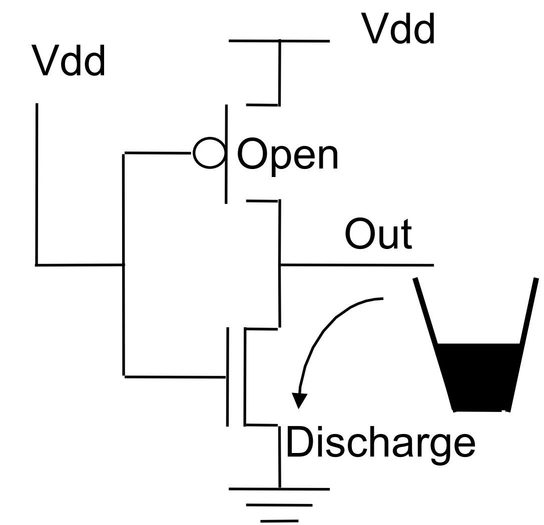
Circuito



Operazione d’inversione

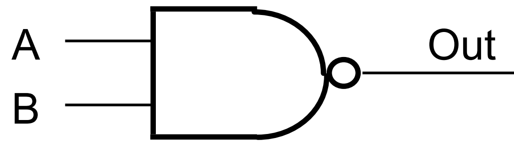




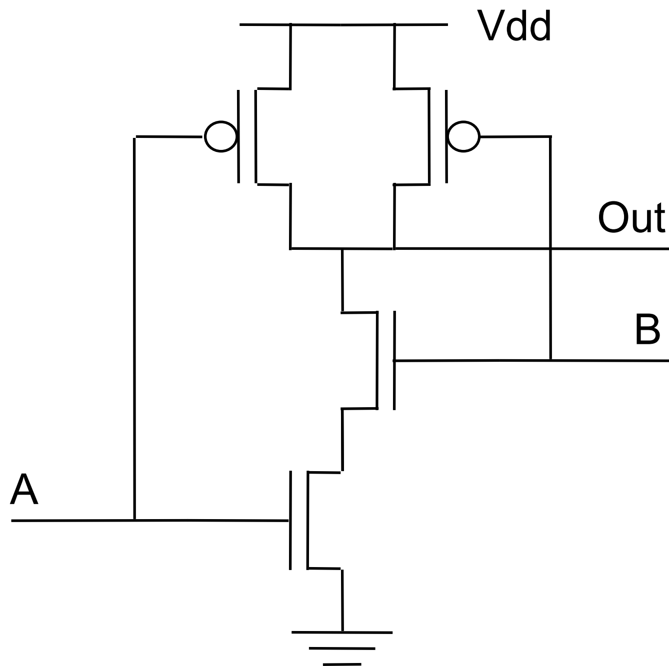


Componenti base: Porte Logiche NOR e NAND

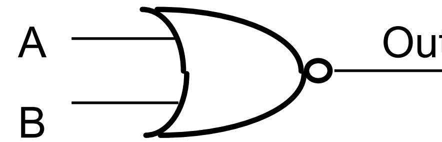
Porta NAND



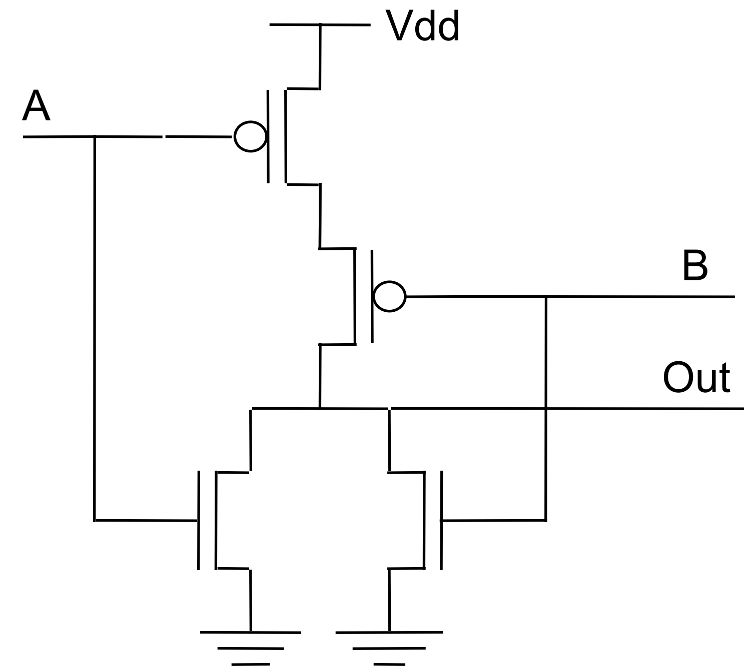
A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0



Porta NOR



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



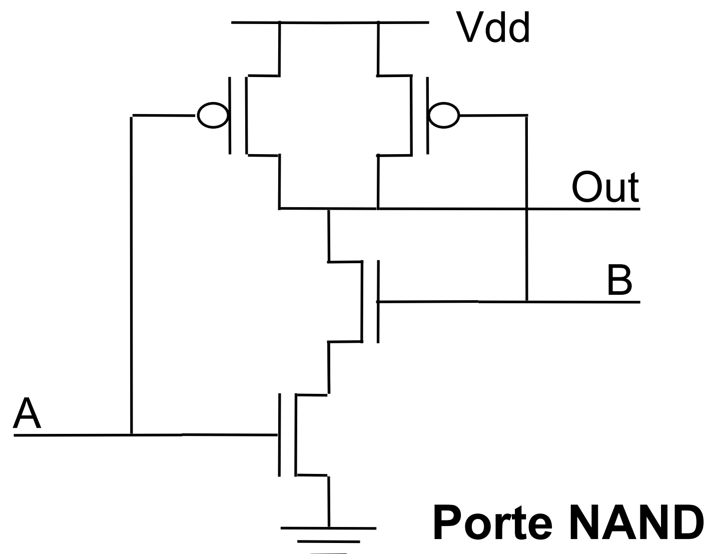
Confronto tra Porte

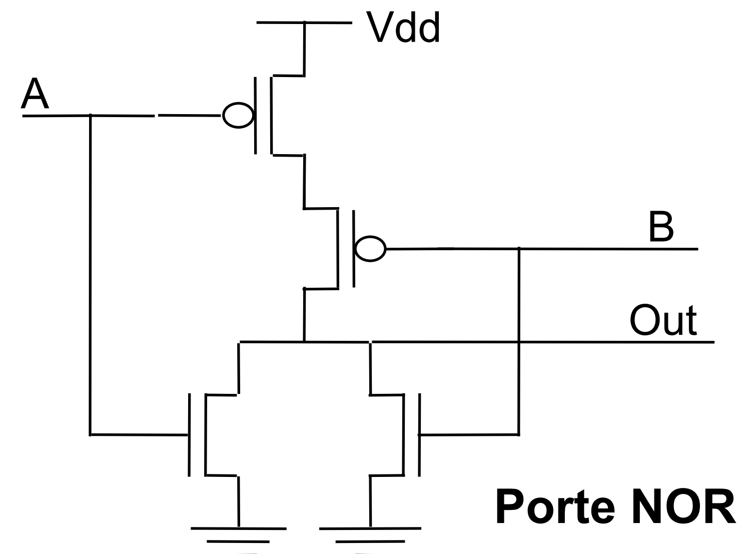
Se i transistor PMOS sono più veloci:

- È meglio avere transistor PMOS in serie
- Porte NOR preferite

Se i transistor NMOS sono più veloci:

- È meglio avere transistor NMOS in serie
- Porte NAND preferite





Forme canoniche

Ogni funzione logica può essere rappresentata come equazione logica o come tabella di verità

Ogni equazione logica può essere scritta in forma canonica tramite l'uso degli operatori AND, OR e NOT

- equazione in forma canonica derivabile dalla corrispondente tabella

Forma canonica *SP (somma di prodotti)*

A	B	C	E
0	**0**	**0**	**0**
0	**0**	**1**	**1**
0	**1**	**0**	**0**
0	**1**	**1**	**0**
1	**0**	**0**	**0**
1	**0**	**1**	**0**
1	**1**	**0**	**1**
1	**1**	**1**	**0**

- Per ogni entry uguale ad 1 dell'ouput (E)
  - genera un *prodotto* (mintermine) degli input (A, B e C), dove gli input uguali a 0 appaiono negati.
- NOTA: ciascun *prodotto* vale 1 solo per quella data combinazione dei fattori (dei valori delle variabili in input).
- Per ottenere l'equazione in forma SP, *somma* i prodotti così ottenuti:

  E = (~A~BC) + (AB~C)

Forme canoniche

Forma canonica *PS (prodotto di somme)*

A	B	C	E
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

- Per ogni entry uguale ad 0 dell'ouput (E)
  - genera una *somma* (maxtermine) degli input (A, B e C), dove gli input uguali a 1 appaiono negati.
- NOTA: ciascuna *somma* vale 0 solo per quella data combinazione degli addendi (dei valori delle variabili in input).
- Per ottenere l'equazione in forma PS, effettua il *prodotto* delle somme così ottenute:

E = (A+B+C) · (A+~B+C) · (A+~B+~C) · (~A+B+C)

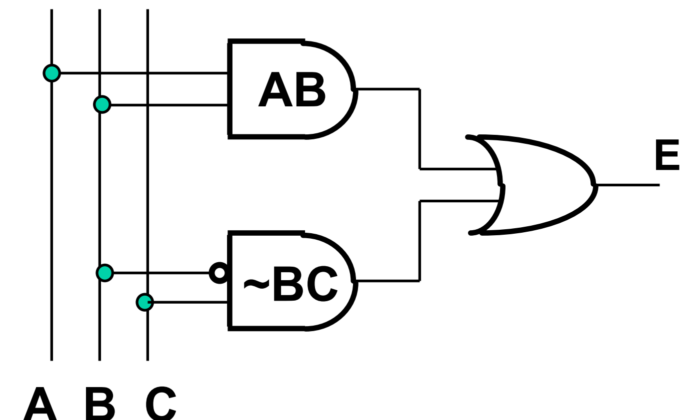
(~A+B+~C) · (~A+~B+~C)

Dalle forme canoniche ai circuiti (2-level logic)

Prendiamo una equazione logica espressa come *somma di prodotti* (SP) che realizza una funzione logica di *n input* e *1 output*

- 1º livello di porte AND per i *prodotti*
  - una porta AND per ogni prodotto
  - arietà (fan-in) delle porte dipende dal numero di fattori dei prodotti (max *arietà = no. variabili in input*)
  - fattori dei prodotti (*variabili in input*) entrano nelle porte *direttamente* o *invertite*
- 2º livello costituito da una porta OR per la *somma*
  - arietà della porta dipende dal numero di prodotti

- i segnali in input attraversano
  - 2 livelli di porte logiche (AND e OR) + eventuali negazioni



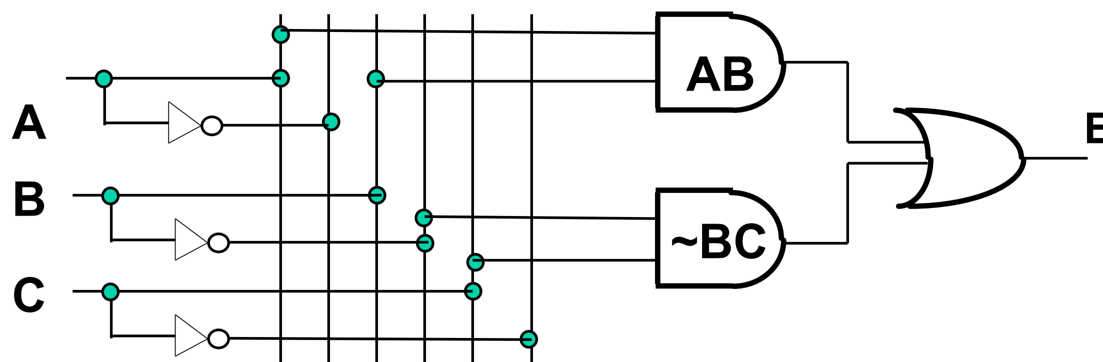
Esempio di forma SP: E = (AB) + (~BC)

Rappresentazione alternativa (circuito a 2 livelli)

Prendiamo una equazione logica espressa come *somma di prodotti* (SP) che realizza una funzione logica di *n input* e *1 output*

- una porta AND per ogni prodotto
- un invertitore per ogni variabile
- input delle porte AND collegate con le linee corrispondenti alle varie variabili (o alla loro negazione)
- l'output delle porte AND collegate in input alla porta OR

Esempio di forma SP: E = (AB) + (~BC)



Minimizzazione circuiti

Scopo della minimizzazione

- data una equazione in forma normale (es. SP), si riduce il numero di prodotti, oppure il numero di variabili coinvolte in ogni prodotto
- minimizzando si riduce quindi il costo del circuito combinatorio corrispondente => meno porte, con arietà (fan-in) ridotta

Esempio di minimizzazione

Esempio di minimizzazione usando le proprietà dell'algebra di Boole

- Funzione F che assume valori indipendentemente dal valore di A:

F	$= \sim AB + AB =$	(distributiva)
	$= B\,(\sim A + A) =$	(inverso)
	$= B \cdot 1 = \quad B$	(nullo)

- A è un input DON'T CARE (che non importa ai fini della definizione dell'equazione)

Esempio di minimizzazione

f = ~A~BC~D + ~A~B CD
+ ~ABC~D + ~ABCD

~AC compare in tutti i prodotti, combinato con tutti i possibili valori di B e D

B e D sono variabili DON'T CARE, e si può minimizzare eliminandole:

f = ~AC

Infatti:

f = ~AC (~B~D + ~BD
+ B~D + BD) =
= ~AC (1) = ~AC

A	B	C	D	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

A	B	C	D	f
0	0	0	0	0
0	0	0	1	0
0	X	1	X	1
0	1	0	0	0
0	1	0	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Tecniche di minimizzazione

Intuitivamente, per semplificare una tabella di verità di N variabili di input e minimizzare la corrispondente forma normale SP, ovvero per scoprire le variabili DON'T CARE, basta individuare:

- 2^1 (coppie di) righe con output 1 dove
  - i valori assunti da N-1 variabili appaiono fissi
  - tutti i possibili valori di una variabile (X) appaiono combinati con gli altri N-1 valori fissi
    ⇒ la variabile X è DON'T CARE
- 2^2 (4-ple di) righe con output 1 dove
  - i valori assunti da N-2 variabili appaiono fissi
  - tutti i possibili valori due variabili (X,Y) appaiono combinati con con gli altri N-2 valori fissi
    ⇒ le variabili X e Y sono DON'T CARE

Tecniche di minimizzazione (...continua)

- 2^3 (8-ple di) righe con output 1 dove
  - i valori assunti da N-3 variabili appaiono fissi
  - tutti i possibili valori di tre variabili (X,Y,Z) appaiono combinati con gli altri N-3 valori fissi
    ⇒ le variabili X, Y e Z sono DON'T CARE
- 2^4 (16-ple di) righe con output 1 dove ….

Mappe di Karnaugh

Difficile minimizzare *a mano* guardando la tabella di verità. Esistono comunque algoritmi efficienti, automatizzabili, ma difficili da usare a mano.

Per minimizzare a mano *funzioni* di poche variabili, si possono *rappresentare le tabelle di verità con le **mappe di Karnaugh***

- ogni quadrato (cella) della mappa individua una combinazione di variabili in *input*
- il valore contenuto nel quadrato corrisponde al valore in *output* per quella particolare combinazione di variabili di input
- per convenzione nella mappa si inseriscono solo i valori uguali a 1
- da notare le combinazioni delle variabili in input che *etichettano* i due assi delle mappe:
  - codice di Gray: differenza di un singolo bit tra combinazioni consecutive

B \ A	0	1
0		
1		**1**

2 variabili

C \ AB	00	01	11	10
0		**1**		
1		**1**		

3 variabili

CD \ AB	00	01	11	10
00			**1**	
01	**1**	**1**		
11			**1**	
10				

4 variabili

Mappe di Karnaugh

Scopo mappe:

- individuare facilmente insiemi di righe (2^1, 2^2, 2^3 righe, ecc.) della tabella di verità con variabili (1, 2, 3 variabili, ecc.) DON'T CARE
- gli 1 corrispondenti a queste righe risultano infatti *adiacenti* nella mappa corrispondente
  - nel considerare l'*adiacenza* delle celle nella mappa, si tenga conto che i bordi orizzontali/verticali della mappa è come se si *toccassero*
  - le combinazioni di 2^1, 2^2, 2^3 righe della tabella di verità originale con 1,2,3 variabili DON'T CARE diventano "*rettangoli*" di valori uguali ad 1 nella mappa di Karnaugh
  - questi rettangoli sono composti da 2^p valori uguali ad 1, e sono anche noti con il termine di *p-sottocubi* (il termine deriva dal fatto che la mappa di Karnaugh è in effetti la *rappresentazione tabellare* di un grafo con topologia ad ipercubo)

Esempi di p-sottocubi

1-sottocubo

C \ AB	00	01	11	10
0		1		
1		1		

f = ~ AB

1-sottocubo

C \ AB	00	01	11	10
0				
1	1			1

f = C~B

2-sottocubo

CD \ AB	00	01	11	10
00	1			1
01				
11				
10	1			1

f = ~B~D

3-sottocubo

CD \ AB	00	01	11	10
00	1	1	1	1
01				
11				
10	1	1	1	1

f = ~D

3-sottocubo

CD \ AB	00	01	11	10
00		1	1	
01		1	1	
11		1	1	
10		1	1	

f = B

2-sottocubo

CD \ AB	00	01	11	10
00				
01		1	1	
11		1	1	
10				

f = BD

Rappresentazione differente per p-sottocubi

1-sottocubo

C \ AB	00	01	11	10
0		1		
1		1		

f = ~ AB

1-sottocubo

C \ AB	00	01	11	10
0				
1	1			1

f = C~B

2-sottocubo

CD \ AB	00	01	11	10
00	1			1
01				
11				
10	1			1

f = ~B~D

3-sottocubo

CD \ AB	00	01	11	10
00	1	1	1	1
01				
11				
10	1	1	1	1

f = ~D

3-sottocubo

CD \ AB	00	01	11	10
00		1	1	
01		1	1	
11		1	1	
10		1	1	

f = B

2-sottocubo

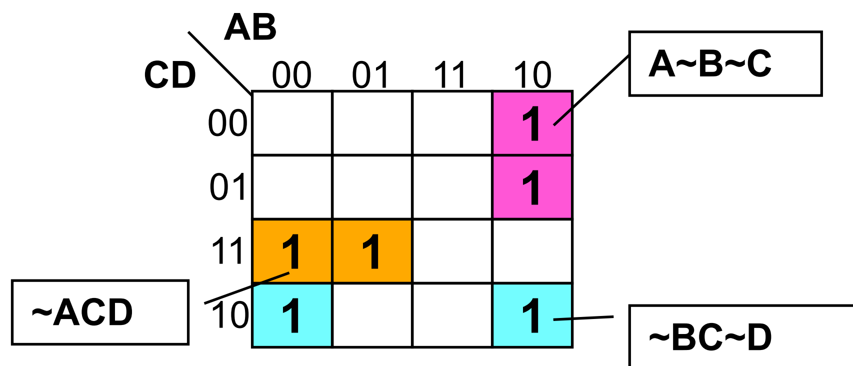
CD \ AB	00	01	11	10
00				
01		1	1	
11		1	1	
10				

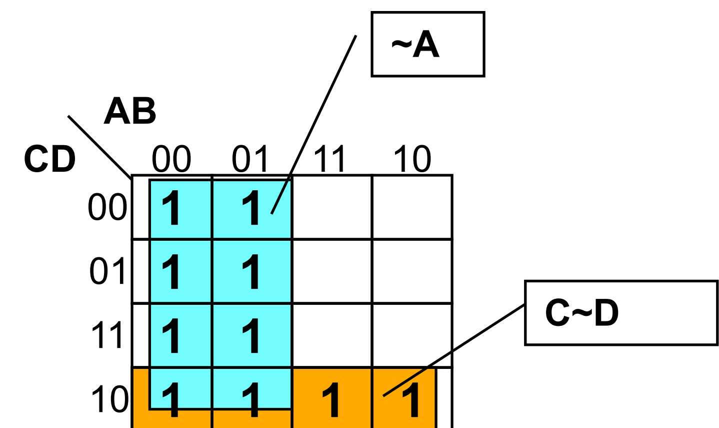
f = BD

Minimizzazione con mappe di Karnaugh

Intuitivamente

- per minimizzare il più possibile, basta scegliere i *più grandi rettangoli* (p-sottocubi) che ricoprono gli 1 della mappa
- ATTENZIONE: gli stessi 1 possono essere ricoperti da *più rettangoli* (da più p-sottocubi)

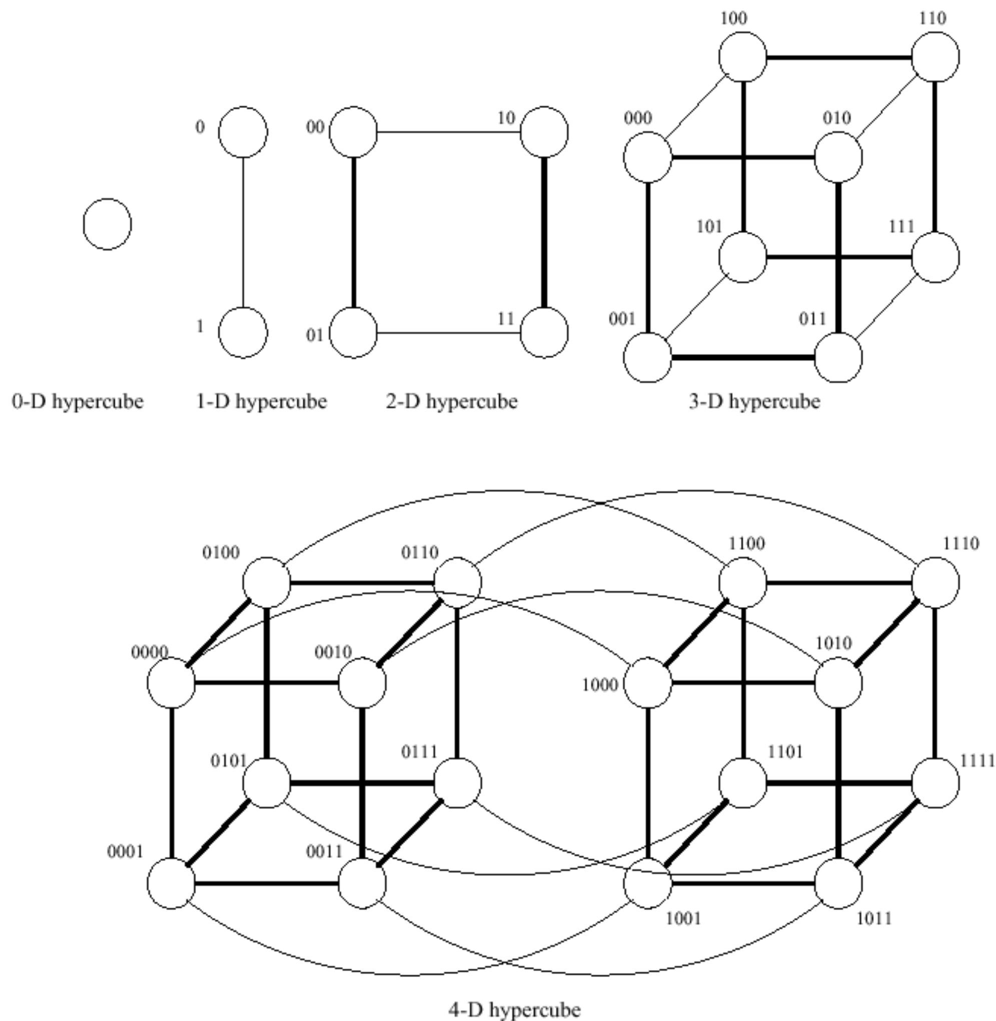




f = ~ACD + A~B~C + ~BC~D

f = ~A + C~D

Ipercubi e Mappe di Karnaugh

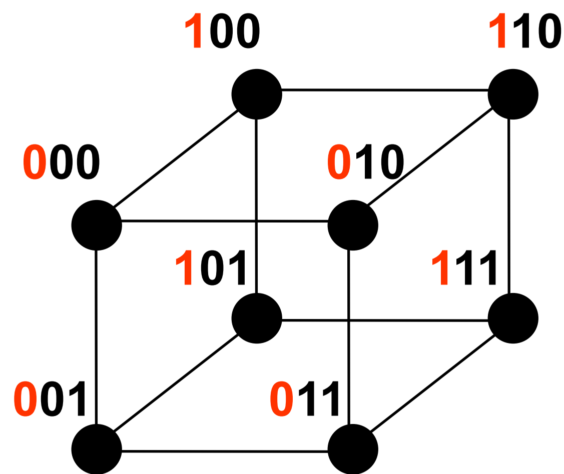


Abbiamo definito alcuni gruppi di 2^p celle delle mappe come *p-sottocubi*

La mappa di Karnaugh è in effetti la *rappresentazione tabellare* di un grafo con topologia ad ipercubo

- ogni nodo dell'ipercubo a *n* dimensioni è *etichettato* con un numero binario a *n* cifre
- due nodi dell'ipercubo sono connessi se la loro etichetta differisce per un solo bit (distanza di hamming pari ad 1)
- ipercubo a *n dimensioni* ottenuto mettendo assieme 2 ipercubi di *n-1 dimensioni*
  - aggiungendo un bit nelle etichette e aggiungendo gli archi in modo consistente
- i sottocubi si riferiscono a specifici sottoinsiemi di nodi connessi

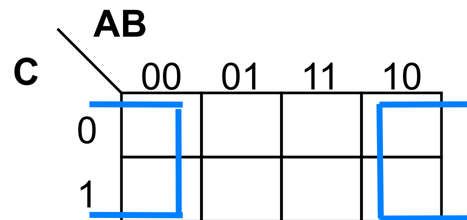
Ipercubi e Mappe di Karnaugh

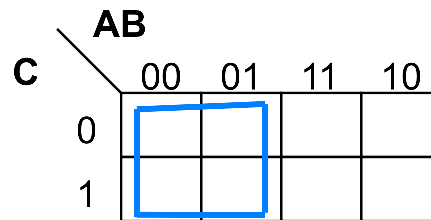


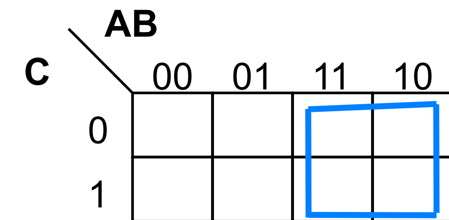
Negli ipercubi le etichette dei nodi *connessi* differiscono di 1 solo bit (distanza di Hamming = 1)

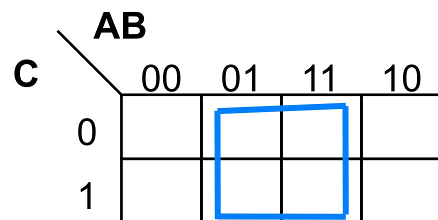
Consideriamo i 2-sottocubi dell'ipercubo a 3 dimensioni illustrato a sinistra

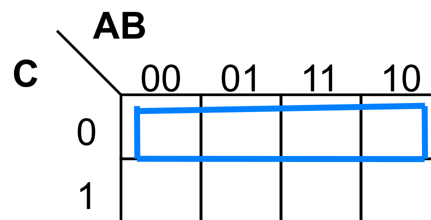
- ogni 2-sottocubo contiene 2^2=4 nodi
- ogni 2-sottocubo corrisponde ai 4 nodi che stanno su una delle 6 facce dell'ipercubo
  - abbiamo al più 6 2-sottocubi

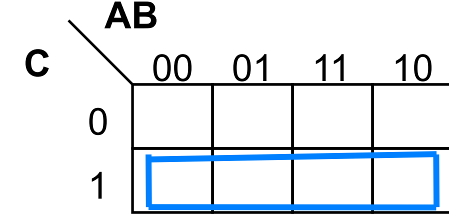












Funzioni incomplete

Alcuni output di una funzione, ovvero gli output corrispondenti a particolari configurazioni degli input, possono *non interessare* (output DON'T CARE)

- es. negli output della tabella di verità (o nella mappa di Karnaugh associata) possiamo avere degli X (dove X sta per DON'T CARE)

Problema:

- l'equazione logica e il corrispondente circuito NON possono essere incompleti
- essi devono produrre un risultato in corrispondenza di TUTTE le combinazioni dei valori in input
- TRUCCO: al posto delle X (valori non specificati) si sceglie 1 o 0 in modo da ottenere la migliore minimizzazione

CD	00	01	11	10
00	1	1		
01	1	1		
11	1	1		
10	1	X	1	1

Considerando X=1, solo 2 p-sottocubi:
f = ~A + C~D

CD \ AB	00	01	11	10
00	1	1		
01	1	1		
11	1	1		
10	1	X	1	1

Considerando X=0, ben 4 p-sottocubi:
f = ~A~B + ~A~C + ~AD + AC~D

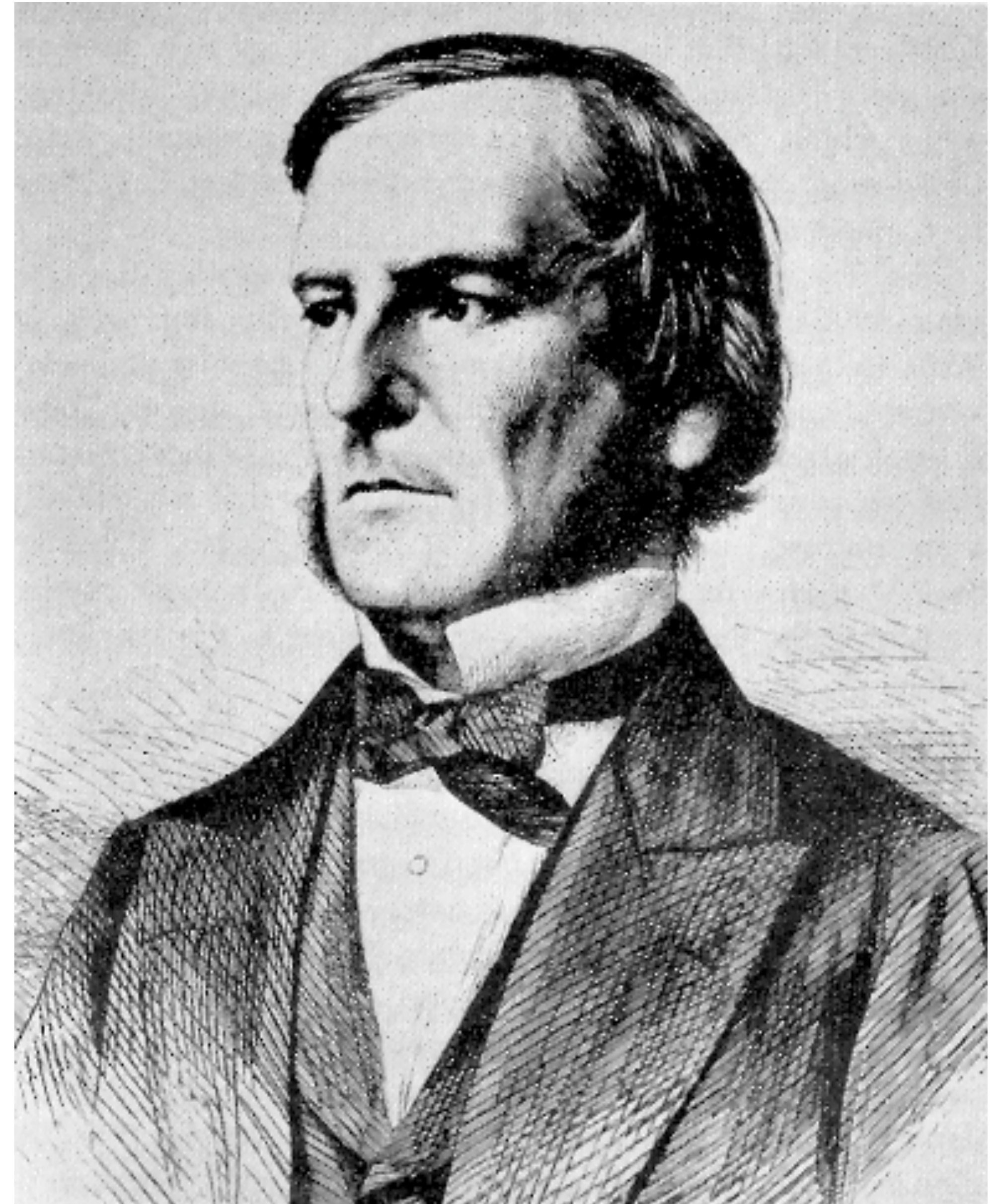
Algoritmo di Quine McCluskey

Le mappe di Karnaugh servono per la minimizzazione "a mano" delle funzioni, ma si riescono a rappresentare al massimo 5 variabili.

Esiste un algoritmo detto Algoritmo di Quine - McCluskey che serve per sintetizzare funzioni logiche con più di 5 variabili in maniera "automatica"

George Boole (1815 – 1864)

- Definisce lo strumento concettuale che sta alla base del funzionamento del calcolatore elettronico: l'algebra booleana.
- Nel suo libro del 1854 crea il legame tra logica e algebra: enunciati logici espressi mediante operazioni algebriche.
- Il suo lavoro in vita fu considerato pura matematica.
- Nel 1938 *Claude Shannon* dimostra che la logica simbolica di Boole può essere applicata per rappresentare le funzioni degli interruttori nei circuiti elettronici



Claude Shannon (1916 – 2001)

- Nella sua tesi di master del 1938 dimostra che il fluire di un segnale elettrico attraverso una rete di interruttori segue le regole dell'algebra di Boole. Questo pone la base teorica dei sistemi di codificazione, elaborazione e trasmissione digitale dell'informazione
- Nel 1948 pubblica una ricerca sul problema di ricostruire, con un certo grado di certezza, le informazioni trasmesse da un mittente. In questa ricerca conia la parola "bit" per designare l'unità elementare d'informazione
- Nel 1949 pubblica uno studio che fonda la teoria matematica della crittografia