# LSS:
# Reti Logiche: circuiti combinatori

Piero Vicini

A.A. 2016-2017

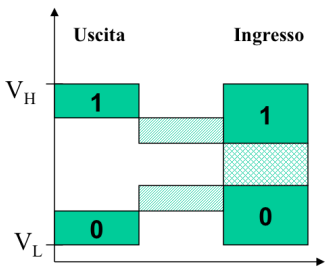
# Introduzione

Argomenti:

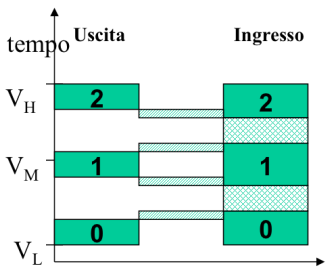
- Codici e aritmetica
- Operatori dell'algebra booleana
- Minimizzazione e sintesi di funzioni
- Esempi di implementazione hardware di circuiti combinatori

# Algebra e circuiti elettronici

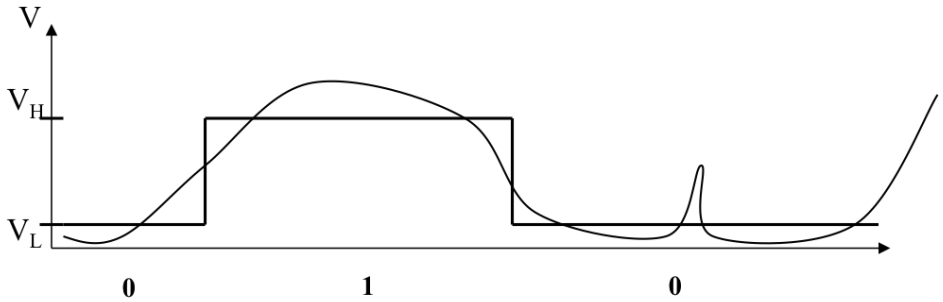
- Computers operano con segnali elettrici con valori discreti di potenziale
- Significativi soltanto 2 valori (intervalli) di potenziale elettrico (high/low, true/false, 1/0)
- Gli elementi elettronici binari sono semplici per definizione (interruttori...)
- Algebra di Boole permette di modellare il funzionamento dei circuiti elettronici binari.



Sistema binario



Sistema a piu' valori

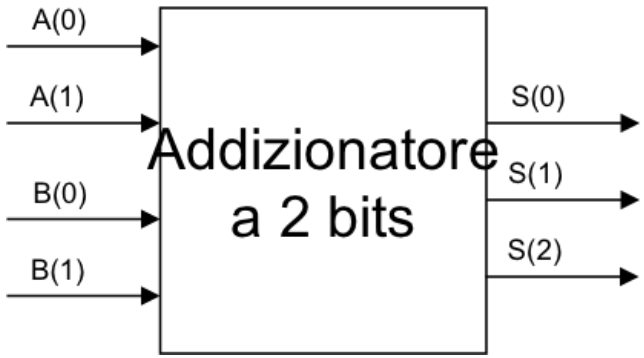


# Circuiti combinatori e sequenziali

- Un *blocco logico* e' un circuito elettronico con linee in input e output a cui sono associate variabili binarie.
- Il circuito calcola *funzioni logiche* come combinazioni di operazioni algebriche booleane sulle variabili in input.
- Due classi: circuiti *combinatori* e circuiti *sequenziali*

# Circuiti combinatori e sequenziali

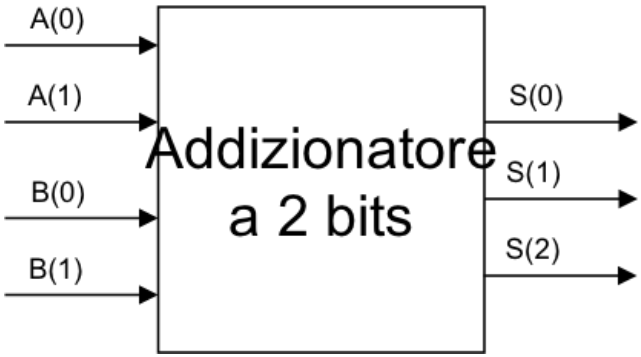
- Un *blocco logico* e' un circuito elettronico con linee in input e output a cui sono associate variabili binarie.
- Il circuito calcola *funzioni logiche* come combinazioni di operazioni algebriche booleane sulle variabili in input.
- Due classi: circuiti *combinatori* e circuiti *sequenziali*



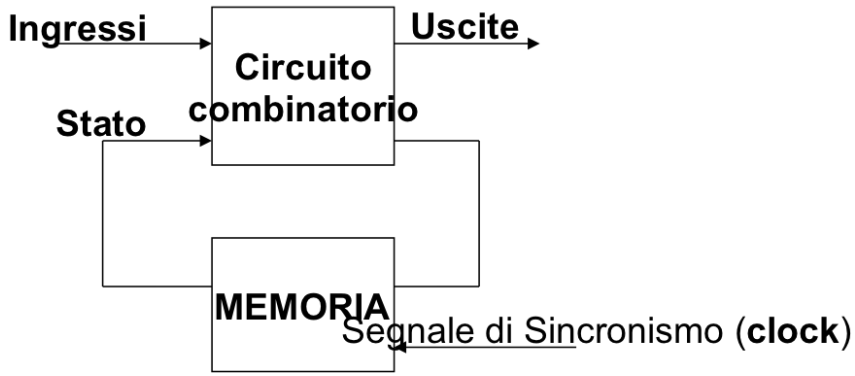

L'uscita di un circuito combinatorio e' determinata completamente dal valore istantaneo della combinazione degli ingressi

# Circuiti combinatori e sequenziali

- Un *blocco logico* e' un circuito elettronico con linee in input e output a cui sono associate variabili binarie.
- Il circuito calcola *funzioni logiche* come combinazioni di operazioni algebriche booleane sulle variabili in input.
- Due classi: circuiti *combinatori* e circuiti *sequenziali*




L'uscita di un circuito combinatorio e' determinata completamente dal valore istantaneo della combinazione degli ingressi




Un circuito sequenziale e' un sistema composto da circuiti combinatori e *elementi di memoria* in cui le uscite sono una funzione del valore degli ingressi e dello *stato passato* del circuito.

# Tavola della verita'

- La funzione logica di un circuito combinatorio e' completamente descritta e specificata da una *tavola della verita'*

- Dati $n$ bits di ingresso, il numero delle configurazioni possibili degli ingressi e' $2^n$
- La tavola possiede quindi $2^n$ righe con valore delle uscite per quella particolare combinazione degli ingressi.

| A(1) | A(0) | B(1) | B(0) | C(2) | C(1) | C(0) |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 |

# Numeri in notazione binaria

- La rappresentazione decimale di numeri interi positivi e' una rappresentazione **posizionale** in base **10**

$$(288)_{10} = 2 * 10^2 + 8 * 10^1 + 8 * 10^0$$

- il cui intervallo va da 0 a $10^N - 1$, con N uguale all'estensione della rappresentazione

# Numeri in notazione binaria

- La rappresentazione decimale di numeri interi positivi e' una rappresentazione **posizionale** in base **10**

$$(288)_{10} = 2 * 10^2 + 8 * 10^1 + 8 * 10^0$$

- il cui intervallo va da 0 a $10^N - 1$, con N uguale all'estensione della rappresentazione

- In perfetta equivalenza anche la rappresentazione binaria e' posizionale ma in base **2**. Dato un numero a *n* bits:

$$(x)_2 = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + ... + x_1 2^1 + x_0 2^0$$

- dove l'intervallo e' $0 : 2^n - 1$
- A 32 bits l'intervallo va da 0 a +4.294.967.295

# Numeri in notazione binaria

- Esempio:

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$$
$$= 0 + ... + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$$
$$= 0 + ... + 8 + 0 + 2 + 1 = 11_{10}$$

- Lo stesso ragionamento vale anche per la rappresentazione di numeri frazionari.
- In base 10:

$$(0.571)_{10} = .... + 5 * 10^{-1} + 7 * 10^{-2} + 1 * 10^{-3}$$

- In base 2:

$$(0.00101)_2 = ....+0*2^{-1}+0*2^{-2}+1*2^{-3}+0*2^{-4}+1*2^{-5}$$

# Rappresentazione esadecimale

- La rappresentazione esadecimale e' un codice la cui base e' 16
- E' una rappresentazione compatta per stringhe di bit

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0000 | 4 | 0100 | 8 | 1000 | c | 1100 |
| 1 | 0001 | 5 | 0101 | 9 | 1001 | d | 1101 |
| 2 | 0010 | 6 | 0110 | a | 1010 | e | 1110 |
| 3 | 0011 | 7 | 0111 | b | 1011 | f | 1111 |

- Ogni gruppo di 4 bits viene rappresentato da una cifra *Hex*
- Esempi:

$$C1A0\ C1A0 = 1100\ 0001\ 1010\ 0000\ \ 1100\ 0001\ 1010\ 0000$$

# Rappresentazione esadecimale

- La rappresentazione esadecimale e' un codice la cui base e' 16
- E' una rappresentazione compatta per stringhe di bit

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0000 | 4 | 0100 | 8 | 1000 | c | 1100 |
| 1 | 0001 | 5 | 0101 | 9 | 1001 | d | 1101 |
| 2 | 0010 | 6 | 0110 | a | 1010 | e | 1110 |
| 3 | 0011 | 7 | 0111 | b | 1011 | f | 1111 |

- Ogni gruppo di 4 bits viene rappresentato da una cifra *Hex*
- Esempi:

$$C1A0\ C1A0 = 1100\ 0001\ 1010\ 0000\ \ 1100\ 0001\ 1010\ 0000$$

$$????\ ???? \quad = 1111\ 1110\ 1101\ 1110\ \ 1100\ 1110\ 1100\ 1010$$

# Cambiamenti di base

- Per convertire da base *qualsiasi* a base 10 sfruttiamo il fatto che la rappresentazione e' posizionale:
$$(427)_8 = 4 * 8^2 + 2 * 8^1 + 7 * 8^0 = 4 * 64 + 2 * 16 + 7 = (279)_{10}$$

- Per convertire da base 10 a base *qualsiasi* procediamo per divisioni successive:

```
35 : 2 = 17 resto 1
17 : 2 =  8 resto 1
 8 : 2 =  4 resto 0
 4 : 2 =  2 resto 0
 2 : 2 =  1 resto 0
 1 : 2 =  0 resto 1
```

$$(35)_{10} = (100011)_2$$

Numeri *signed* complemento a 2

- Dato un numero ad n bit:
  $x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + ... + x_1 2^1 + x_0 2^0$
- l'intervallo e' $-2^{n-1}$ a $2^{n-1} - 1$

- Esempio:
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1 * 2^{31} + 1 * 2^{30} + ... + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0$
    $= -2.147.483.648 - 2.147.483.644 = -4_{10}$

- A 32 bits l'intervallo va da $-2.147.483.648$ a $+2.147.483.647$

# Numeri e rappresentazioni: signed (2)

Numeri *signed* complemento a 2 (continua)

- Il bit 31 e' il bit di segno: 1 per negativi, 0 per positivi
- la rappresentazione non e' completa i.e. intervallo non simmetrico: $-(-2^{n-1})$ ????
- I numeri non-negativi hanno la stessa rappresentazione unsigned e 2s-complement (utile per aritmetica...)
- Alcuni numeri specifici:
  - 0 : 0000 0000 ... 0000
  - $-1$ : 1111 1111 ... 1111
  - Massimo numero negativo: 1000 0000 ... 0000
  - Massimo numero positivo: 0111 1111 ... 1111

# Numeri e rappresentazioni: signed (3)

Per passare dal numero positivo al suo equivalente negativo (e viceversa) si effettua un'operazione di complementazione e somma (+1) numeri negativo

- *Complementare* un numero binario significa invertirne tutti i bit
  $x + \bar{x} = 1111 \ldots 111_2 = -1$
  $\bar{x} + 1 = -x$

- Esempio negazione di +2
  - $+2 = 0000\ 0000 \ldots 0010_2$
  - $-2 = 1111\ 1111 \ldots 1101_2 + 1$
    $= 1111\ 1111 \ldots 1110_2$

# Addizione e sottrazione

Abbiamo imparato alle scuole elementari che addizione e sottrazione si eseguono cosi'

Addizione

```
  111111
   1011011000000 +
  10110110000000 =
-----------------------
 100010001000000
```

Sottrazione (come in base 10 ma occhio ai "prestiti")

```
  1 1 1
  1 0 0 0 -       (8)
  0 1 0 1 =       (5)
----------------------------
  0 0 1 1
```

# Moltiplicazione ed estensione di segno

## Moltiplicazione

```
          1 1 1 1 *
          1 1 1 1 =
--------------------
          1 1 1 1
        1 1 1 1
      1 1 1 1
    1 1 1 1
--------------------
 1 1 1 0 0 0 0 1
```

Estensione di segno: se $N < 0$ si riempe a sinistra di 1 altrimenti di 0

```
          0 1 0 0 =  4
0 0 0 0 0 1 0 0 =  4

          1 1 0 0 = -4
1 1 1 1 1 1 0 0 = -4
```

# Operazioni signed

- Se la rappresentazione e' in complemento a 2, addizione e sottrazione si riducono alla sola addizione trascurando il bit MSB del risultato
- La rappresentazione e' limitata: dobbiamo fare attenzione agli overflow...

```
+7      0 1 1 1 +
-3      1 1 0 1 =
  ---------------
+4  (1) 0 1 0 0
```

```
          1 1
+7      0 1 1 1 +
+3      0 0 1 1 =
  ---------------
    (0) 1 0 1 0 ?????
```

- Sottrazione -> $a - b = a + (-b)$

$6 - 2 = (6) + (-2)$

```
    1 1
    0 1 1 0 +    (+6)
    1 1 1 0 =    (-2)
  -------------
(1) 0 1 0 0   (+4)
```

$-4 - 2 = (-4) + (-2)$

```
      1
    1 1 0 0 +    (-4)
    1 1 1 0 =    (-2)
  -------------
(1) 1 0 1 0     (-6)
```

$-4 - 5 = (-4) + (-5)$

```
    1 1 0 0 +    (-4)
    1 0 1 1 =    (-5)
  -------------
(1) 0 1 1 1     (+7) ?????
```

# Overflow

*Overflow* compare quando il risultato dell'operazione non puo' essere rappresentato nel formato degli operandi (es 32 bits) ovvero quando il bit di segno non e' coerente con quello aspettato dal tipo di operazione e dal segno degli operandi

- No overflow quando si addizionano un numero positivo ed uno negativo...
- No overflow quando in una sottrazione i segni degli operandi sono uguali

| Operazione | Operando A | Operando B | Sign Result per Ovf |
|---|---|---|---|
| $A + B$ | $\geq 0$ | $\geq 0$ | $< 0$ |
| $A + B$ | $< 0$ | $< 0$ | $\geq 0$ |
| $A - B$ | $\geq 0$ | $< 0$ | $< 0$ |
| $A - B$ | $< 0$ | $\geq 0$ | $\geq 0$ |

# Rappresentazione in virgola mobile

$$R = (-1)^S \times 1.M \times b^E$$

- In virgola mobile (i.e. *floating point*) il numero reale perde la sua esattezza algebrica ma si rappresenta con
  - un bit di *segno* **S**,
  - un'ordine di grandezza (*esponente*) **E**
  - un numero arbitrario di cifre significative (*mantissa*) **M**

# Rappresentazione in virgola mobile

$$R = (-1)^S \times 1.M \times b^E$$

- In virgola mobile (i.e. *floating point*) il numero reale perde la sua esattezza algebrica ma si rappresenta con
  - un bit di *segno* **S**,
  - un'ordine di grandezza (*esponente*) **E**
  - un numero arbitrario di cifre significative (*mantissa*) **M**
- La rappresentazione in virgola mobile si puo' usare con qualsiasi base e con qualsiasi rappresentazione di esponente e mantissa
- Esempi:

  $1.27E^{12} = 1.27x10^{12}$

  $12,433x10^{-4}, 11.10010x10^{11101}, 1110.10110x10^{-10}$

- Di conseguenza c'e' bisogno di accordarsi su uno standard

# Standard FP IEEE 754

- Nei computers si usano abitualmente rappresentazioni IEEE da 32 bit (*singola precisione*) e 64 bit (*doppia precisione*)
- La singola precisione (SP) ha 7 cifre decimali significative ed un range di $10^{\pm 38}$
- La doppia precisione (DP) ha 16 cifre decimali significative ed un range di $10^{\pm 308}$

# Esempio FP

Numero float $= +15213.0$
$15213_{(10)} = 11101101101101_{(2)} = 1.1101101101101 \times 2^{13}$

Mantissa:
$M = 1.1101101101101$
$frac = 1101101101101\ 0000000000$

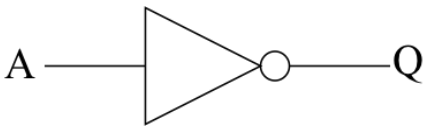
Esponente:
$E = 13$
$bias = 127$
$E = 127 + 13 = 140 = 10001100$

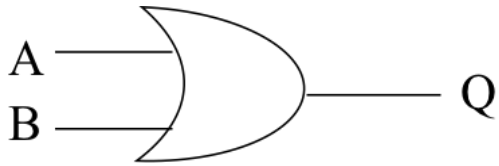
Rappresentazione in SP
0 10001100 11011011011010000000000

# Algebra Booleana e porte logiche

La funzione logica di un circuito combinatorio e' completamente descritta e specificata da una *Equazione Logica*

- Le variabili (i segnali...) in input e output sono *variabili logiche*.
- la funzione e' la combinazione dei 3 *operatori fondamentali* dell'algebra booleana
  - **OR** $(A + B)$ out $= 1$ *se almeno un input* $= 1$
  - **AND** $(A * B)$ out $= 1$ *se tutti gli input* $= 1$
  - **NOT** $(\overline{A})$ out $=$ inverso dell'input



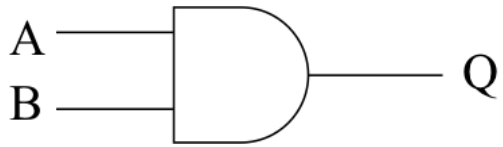




Table: Operatore NOT

| $A$ | $Q$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

Table: Operatore OR

| $A$ | $B$ | $Q$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Table: Operatore AND

| $A$ | $B$ | $Q$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

# Proprieta' dell'algebra booleana

Proprieta' degli operatori:

| | | |
|---|---|---|
| Identita': | $A + 0 = A$ | $A * 1 = A$ |
| Nullo: | $A + 1 = 1$ | $A * 0 = 0$ |
| Idempotente: | $A + A = A$ | $A * A = A$ |
| Inverso: | $A + \overline{A} = 1$ | $A * \overline{A} = 0$ |

Proprieta' dell'algebra:

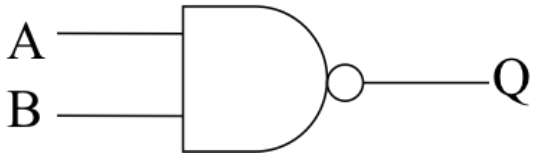
| | | |
|---|---|---|
| Commutativa: | $A + B = B + A$ | $A * B = B * A$ |
| Associativa: | $A + (B + C) = (A + B) + C$ | $A * (B * C) = (A * B) * C$ |
| Distributiva: | $A * (B + C) = (A * B) + (A * C)$ | $A + (B * C) = (A + B) * (A + C)$ |

Leggi di De Morgan:

$$\overline{(A + B)} = \overline{A} * \overline{B} \quad \overline{(A * B)} = \overline{A} + \overline{B}$$

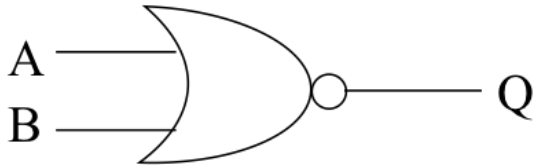
# Operatori NAND e NOR

NAND: e' l'operatore NOT(AND)




| $A$ | $B$ | $Q$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

NOR: e' l'operatore NOT(OR)




| $A$ | $B$ | $Q$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

Si puo' dimostrare che il NAND(NOR) e' un'*operatore universale* i.e e' l'unico necessario per implementare qualsiasi funzione logica

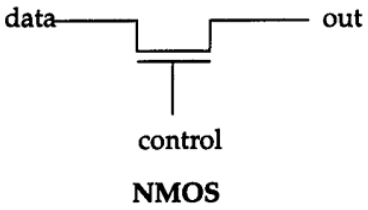
$$\overline{A} = \overline{A} + 0 = \overline{A * 1}$$

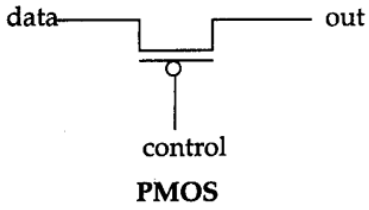
$$A + B = \overline{\overline{A + B}} = \overline{\overline{A} * 1 * \overline{B} * 1}$$

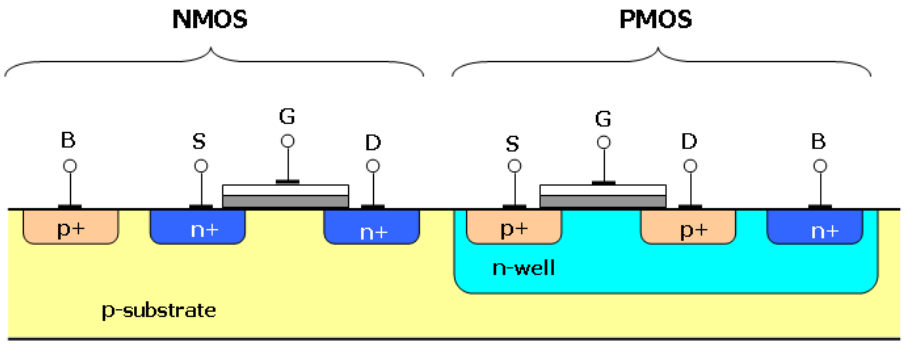
$$A * B = (A * B) + 0 = \overline{\overline{(A * B) + 0}} = \overline{\overline{A * B} * 1}$$

# Porte logiche e transistor

- Tecnologia *CMOS* (Complementary Metal Oxide Semiconductor) viene usata per realizzare transistor planari su silicio
- I principali vantaggi rispetto ad implementazione BJT sono la maggiore semplicita' della tecnologia planare usata (che permette densita' maggiori) e la potenza statica dissipata quasi nulla.



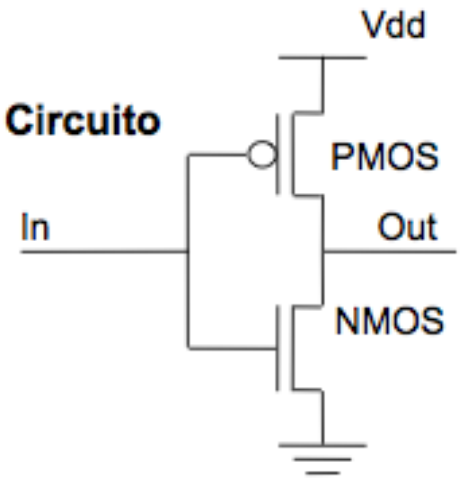




- Il comportamento e' quello di un interruttore comandato dal gate Control
  - *NMOS* (N-Type Metal Oxide Semiconductor) transistor conduce se $C = 1$. Resistenza infinita se $C = 0$
  - *PMOS* (P-Type Metal Oxide Semiconductor) transistor conduce se $C = 0$. Resistenza infinita se $C = 1$

- In tecnologia CMOS un PMOS e' sempre accoppiato ad un NMOS.
- Si escludono quindi path *statici* tra VDD e GND –> potenza dissipata statica (quasi) nulla...
- Solo in presenza del cambiamento di stato del gate si apre un canale VDD-GND –> potenza dissipata dinamica bassa

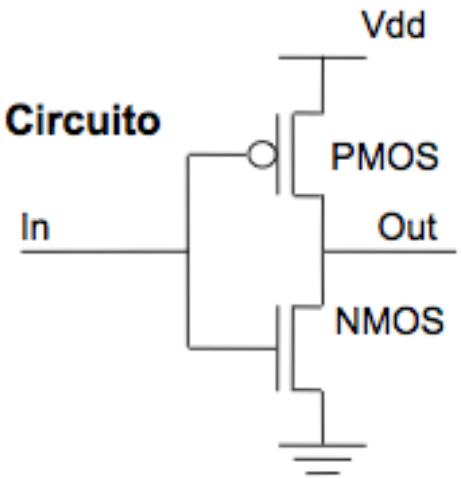
Es: Inverter

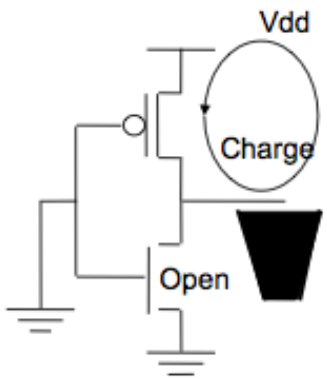


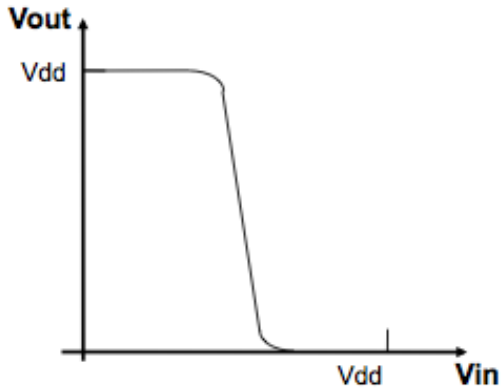
# Porte logiche e transistor

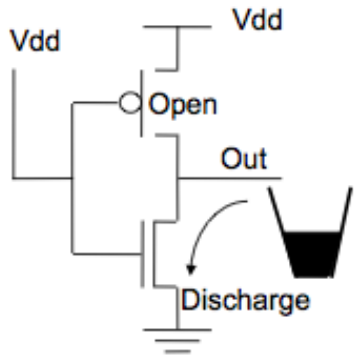
- In tecnologia CMOS un PMOS e' sempre accoppiato ad un NMOS.
- Si escludono quindi path *statici* tra VDD e GND –> potenza dissipata statica (quasi) nulla...
- Solo in presenza del cambiamento di stato del gate si apre un canale VDD-GND –> potenza dissipata dinamica bassa

Es: Inverter

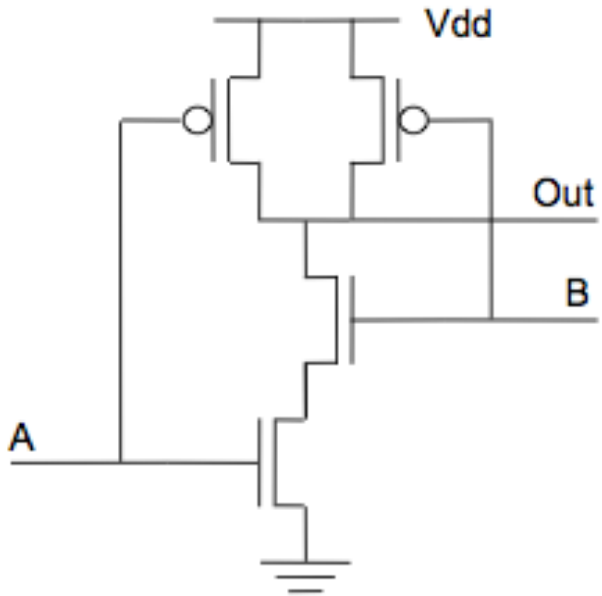


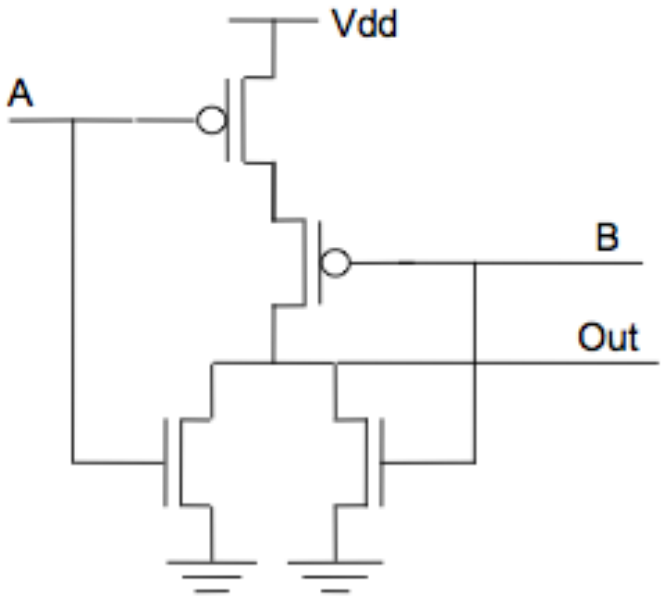



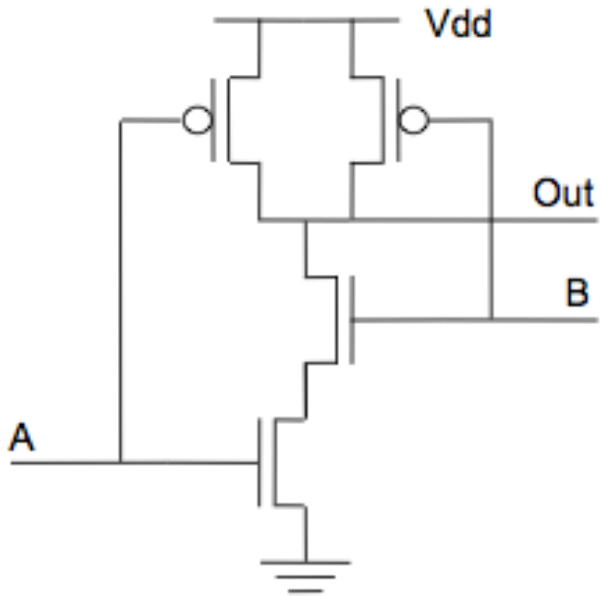



# Porte logiche e transistor

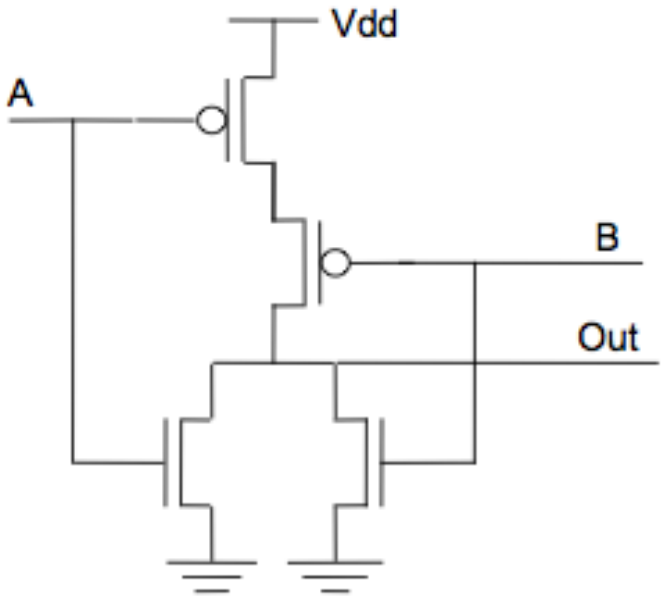


# Porte logiche e transistor

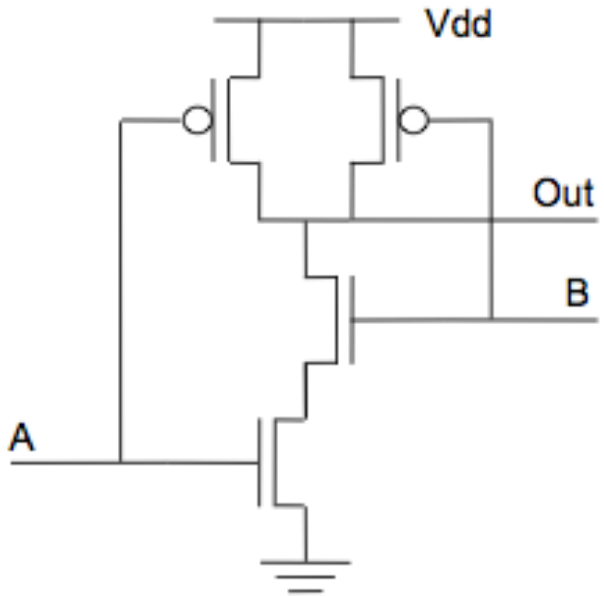



NAND

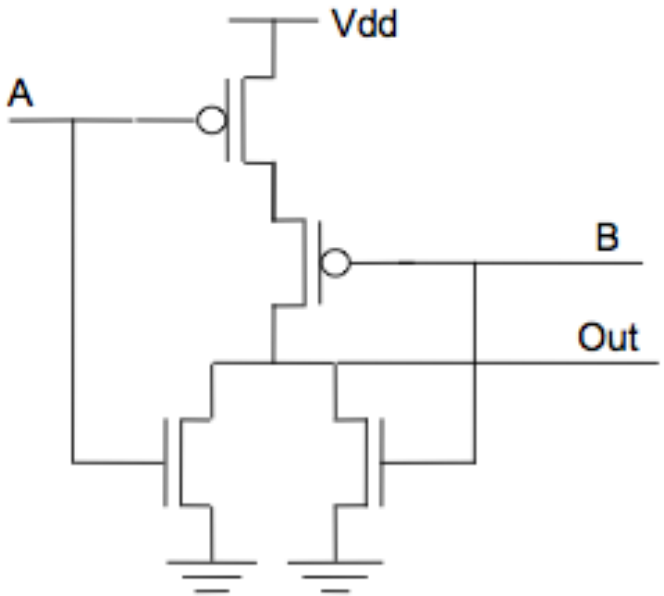



NOR

# Porte logiche e transistor




NAND




NOR

- Se in una particolare tecnologia il transistor PMOS e' piu' veloce
  - Meglio avere PMOS in serie
  - Porte NOR preferite per implementazione circuiti
- Se invece una particolare il transistor NMOS e' piu' veloce
  - Meglio avere NMOS in serie
  - Porte NAND preferite per implementazione circuiti

# Porte logiche con BJT

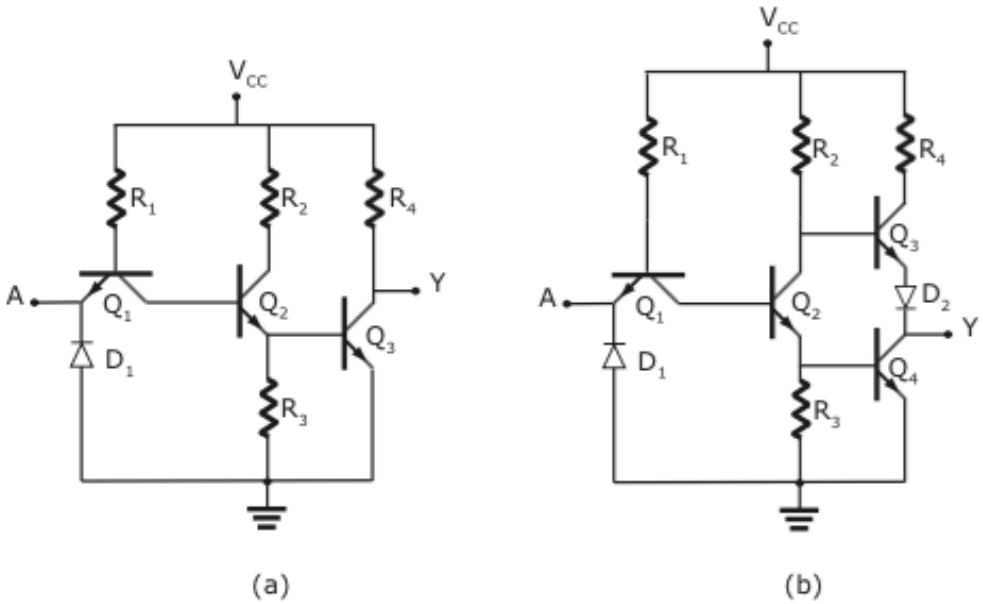



Figura 7.5: *(a) Circuito NOT; (b) circuito NOT con uscita "totem pole".*

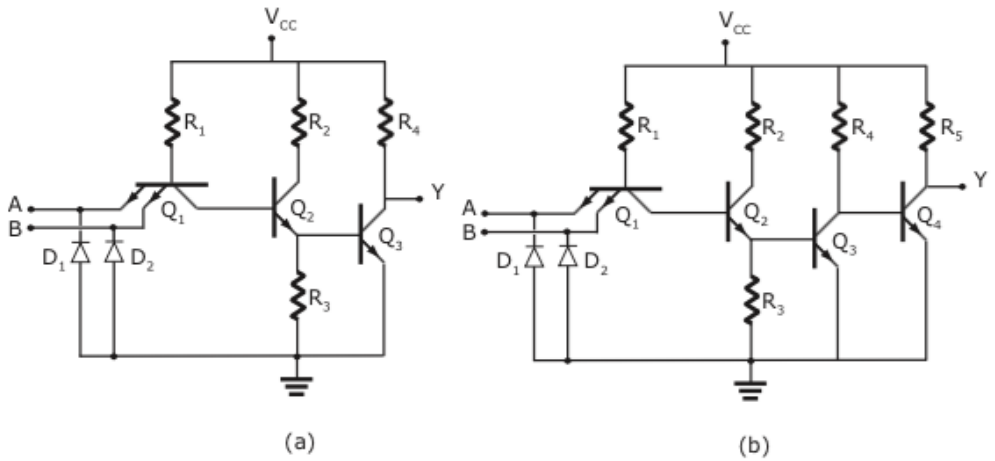
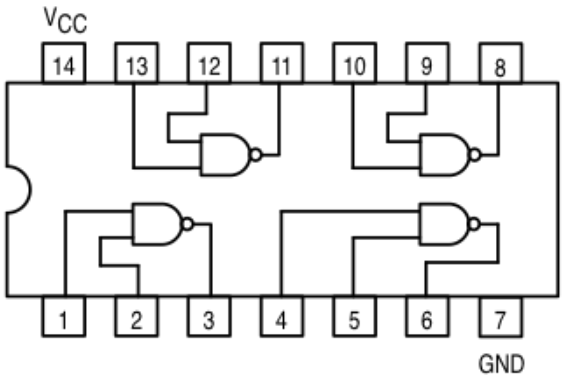



Figura 7.6: *(a) Circuito NAND;(b) Circuito AND*

MOTOROLA

# QUAD 2-INPUT NAND GATE

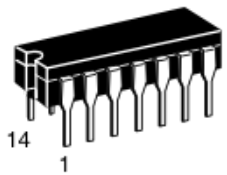
- ESD > 3500 Volts




**SN54/74LS00**

**QUAD 2-INPUT NAND GATE**

**LOW POWER SCHOTTKY**




**J SUFFIX**
CERAMIC
CASE 632-08

**GUARANTEED OPERATING RANGES**

| Symbol | Parameter | | Min | Typ | Max | Unit |
|---|---|---|---|---|---|---|
| $V_{CC}$ | Supply Voltage | 54<br>74 | 4.5<br>4.75 | 5.0<br>5.0 | 5.5<br>5.25 | V |
| $T_A$ | Operating Ambient Temperature Range | 54<br>74 | −55<br>0 | 25<br>25 | 125<br>70 | °C |
| $I_{OH}$ | Output Current — High | 54, 74 | | | −0.4 | mA |
| $I_{OL}$ | Output Current — Low | 54<br>74 | | | 4.0<br>8.0 | mA |

**DC CHARACTERISTICS OVER OPERATING TEMPERATURE RANGE** (unless otherwise specified)

| Symbol | Parameter | | Limits Min | Limits Typ | Limits Max | Unit | Test Conditions | |
|---|---|---|---|---|---|---|---|---|
| $V_{IH}$ | Input HIGH Voltage | | 2.0 | | | V | Guaranteed Input HIGH Voltage for All Inputs | |
| $V_{IL}$ | Input LOW Voltage | 54 | | | 0.7 | V | Guaranteed Input LOW Voltage for All Inputs | |
| | | 74 | | | 0.8 | | | |
| $V_{IK}$ | Input Clamp Diode Voltage | | | −0.65 | −1.5 | V | $V_{CC}$ = MIN, $I_{IN}$ = −18 mA | |
| $V_{OH}$ | Output HIGH Voltage | 54 | 2.5 | 3.5 | | V | $V_{CC}$ = MIN, $I_{OH}$ = MAX, $V_{IN}$ = $V_{IH}$ or $V_{IL}$ per Truth Table | |
| | | 74 | 2.7 | 3.5 | | V | | |
| $V_{OL}$ | Output LOW Voltage | 54, 74 | | 0.25 | 0.4 | V | $I_{OL}$ = 4.0 mA | $V_{CC}$ = $V_{CC}$ MIN, $V_{IN}$ = $V_{IL}$ or $V_{IH}$ per Truth Table |
| | | 74 | | 0.35 | 0.5 | V | $I_{OL}$ = 8.0 mA | |
| $I_{IH}$ | Input HIGH Current | | | | 20 | μA | $V_{CC}$ = MAX, $V_{IN}$ = 2.7 V | |
| | | | | | 0.1 | mA | $V_{CC}$ = MAX, $V_{IN}$ = 7.0 V | |
| $I_{IL}$ | Input LOW Current | | | | −0.4 | mA | $V_{CC}$ = MAX, $V_{IN}$ = 0.4 V | |
| $I_{OS}$ | Short Circuit Current (Note 1) | | −20 | | −100 | mA | $V_{CC}$ = MAX | |
| $I_{CC}$ | Power Supply Current<br>Total, Output HIGH | | | | 1.6 | mA | $V_{CC}$ = MAX | |
| | Total, Output LOW | | | | 4.4 | | | |

Note 1: Not more than one output should be shorted at a time, nor for more than 1 second.

**AC CHARACTERISTICS** ($T_A$ = 25°C)

# Forme canoniche

- Ogni equazione logica puo' essere rappresentata in *forma canonica* tramite uso di operatori AND, OR, e NOT
- La forma canonica si deriva (ad es..) dalla tabella della verita' in forma di *Somma di Prodotti* (SP)

| 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'output si genera un prodotto *minterm* degli input dove gli input=0 sono negati
- Per ottenere l'equazione in forma S, si sommano i prodotti cosi' ottenuti

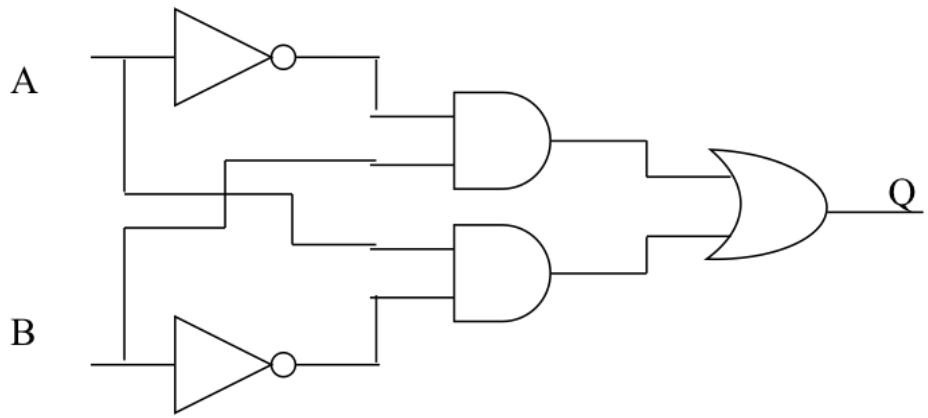
$$E = (\overline{A} * \overline{B} * C) + (A * B * \overline{C})$$

- Grazie alla proprieta' di *identita'* della somma logica ($A + 0 = A$) il contributo all'equazione logica viene solo dai minterm non 0

# Da forma canonica al circuito (logica 2-level)

- A partire da un'equazione logica espressa come SP si puo' realizzare il circuito equivalente con variabili (segnali) invertite e non invertite che attraversano:
  - un livello di porte AND per i prodotti
  - un livello di porte OR per la somma dei prodotti

| $A$ | $B$ | $Q$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |




$$Q = (\overline{A} * B) + (A * \overline{B})$$

# Minimizzazione di circuiti

- Obiettivo della minimizzazione e' la riduzione del *costo* del circuito combinatorio in termini di numero di porte e variabili necessarie all'implementazione dell'equazione richiesta
- Si ottiene un circuito equivalente che generalmente e' piu' *piccolo* e con tempi di propagazione ridotti.
- Si puo' agire per ispezione utilizzando le proprieta' dell'algebra.

Esempio:

$Q = \overline{A} * B + A * B$ –>applico proprieta' distributiva
$Q = B * (\overline{A} + A)$ –> applico inverso
$Q = B * 1 = B$

- in questo caso la variabile $A$ e' *DON'T CARE* cioe' non conta ai fini della definizione della equazione
- L'individuazione di variabili DON'T CARE e' l'obiettivo ultimo della minimizzazione

- Esempio: multiplexer a due input. Seleziona in uscita il valore di un ingresso scelto tra 2(N) diversi in base ai valori assunti dagli ingressi di *select* (S)

| S | A | B | Z |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

- Scrivo la funzione in termini di somme di prodotti

$$Z = \overline{S}A\overline{B} + \overline{S}AB + S\overline{A}B + SAB \text{ -> distributiva}$$

$$Z = \overline{S}A(\overline{B} + B) + SB(\overline{A} + A) \text{ -> inverso}$$

$$Z = \overline{S}A + SB$$

- 3 gate NOT, 8 gate AND e 3 gate OR –> 1 NOT, 2 AND, 1 OR !!!!!

# Mappe di Karnaugh (MK)

- Metodo grafico utile per minimizzare funzioni booleane di poche (2-6) variabili.
- Sfrutta caratteristiche posizionali delle *Mappe di Karnaugh* che sono rappresentazioni simili ed alternative alla tavola della verita'

| A B C | Z | |
|---|---|---|
| 0 0 0 | 1 | /A/B/C |
| 0 0 1 | 1 | /A/BC |
| 0 1 0 | 1 | /AB/C |
| 0 1 1 | 0 | |
| 1 0 0 | 0 | |
| 1 0 1 | 0 | |
| 1 1 0 | 1 | AB/C |
| 1 1 1 | 0 | |

$Z = /A/B/C + /A/BC + /AB/C + AB/C$

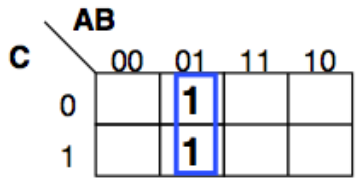
| A B \ C | 0 | 1 |
|---|---|---|
| 00 | 1 | 1 |
| 01 | 1 | 0 |
| 11 | 1 | 0 |
| 10 | 0 | 0 |

- La combinazione degli input per ogni riga(colonna) deve differire da quelle delle righe(colonne) adiacenti per l'inversione di una sola variabile
- Quindi, ogni casella differisce dalle adiacenti per l'inversione di una sola variabile
- Allora, *due caselle adiacenti generano una variabile DON'T CARE....*

# Mappe di Karnaugh (MK)

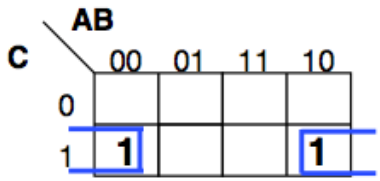
- Scopo del metodo e' quello di individuare facilmente insieme di righe della tabella della verita' che contengono variabili DON'T CARE.
- gli 1 corrispondenti a queste righe risultano adiacenti nella mappa corrispondente. Il loro insieme si definisce come *loop* o *p-sottocubo* dove *p* e' il rank del loop.
- Le mappe sono periodiche al contorno i.e. bordi orizzontali e verticali possono essere considerati adiacenti...

1-sottocubo



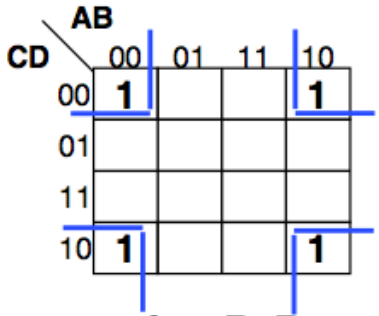

f = ~ AB

1-sottocubo



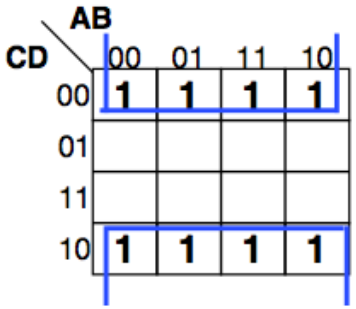

f = C~B

2-sottocubo



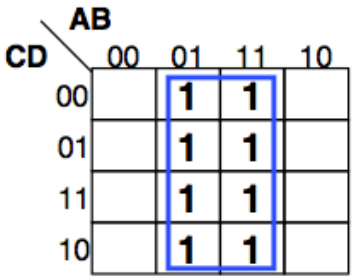

f = ~B~D

3-sottocubo



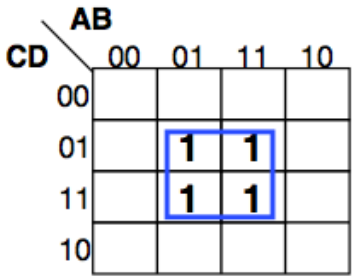

f = ~D

3-sottocubo




f = B

2-sottocubo

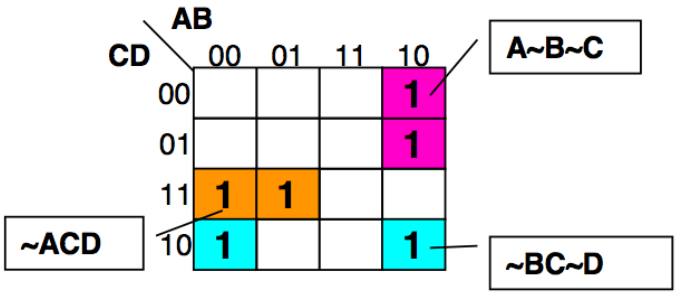



f = BD

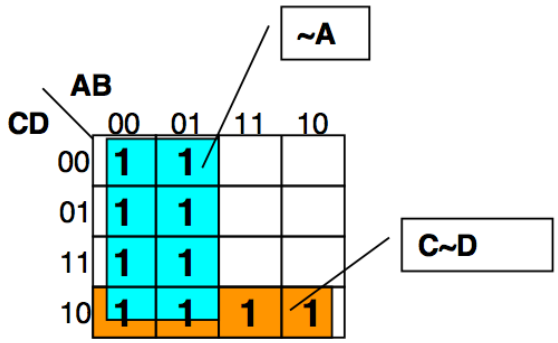
# Mappe di Karnaugh (MK)

E' intuitivo che piu' grandi sono i loop piu' efficace la minimizzazione

- Per le proprieta' dell'algebra gli stessi 1 possono essere inclusi in piu' loops

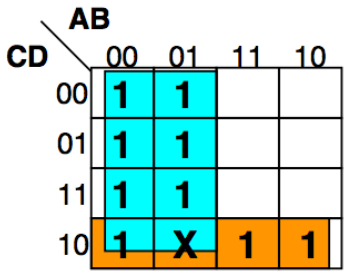



**f = ~ACD + A~B~C + ~BC~D**



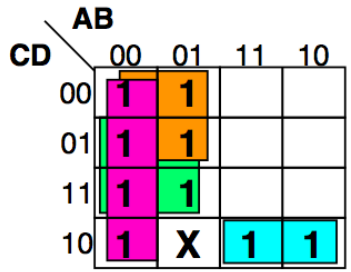

**f = ~A + C~D**

- In caso di funzioni *incomplete* alcuni output, per particolari configurazioni degli input, non sono definiti/interessanti e quindi possono valere 0 o 1 a nostra scelta
- Sfrutto questo fatto per costruire una MK piu' efficace in termini di minimizzazione




**Considerando X=1, solo 2 p-sottocubi:**

**f = ~A + C~D**

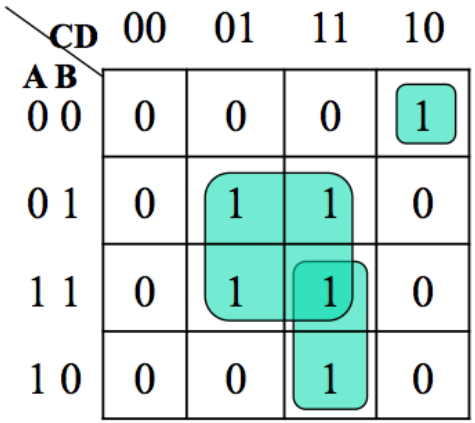



**Considerando X=0, ben 4 p-sottocubi:**

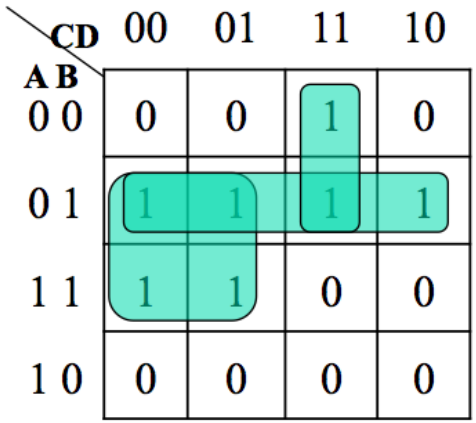
**f = ~A~B + ~A~C + ~AD + AC~D**

# Mappe di Karnaugh (MK)

1. Costruire la MK come visto
2. *Padding* delle condizioni DON'T CARE nel modo migliore
3. Ricercare nella mappa gli 1 "isolati" e crearne un gruppo per ognuno
4. Ricercare nella mappa gli 1 che sono adiacenti ad un solo altro 1 e crearne un loop-coppia.
5. Raggruppare gli eventuali ottetti anche se contengono 1 gia' inclusi in altri gruppi
6. Raggruppare gli eventuali quad anche se contengono 1 gia' inclusi in altri gruppi (facendo attenzione ad usare il numero minimo di gruppi)
7. Generare un gruppo-coppia per ogni 1 non incluso
8. Formare l'OR dei termini generati da ciascun gruppo




Z=/A/BC/D + BD + ACD



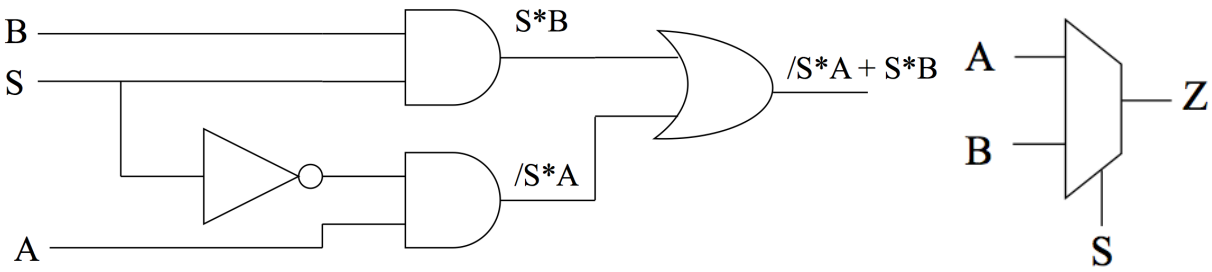

Z=/ACD + /AB + B/C

# Esempi di circuiti combinatori: Multiplexer

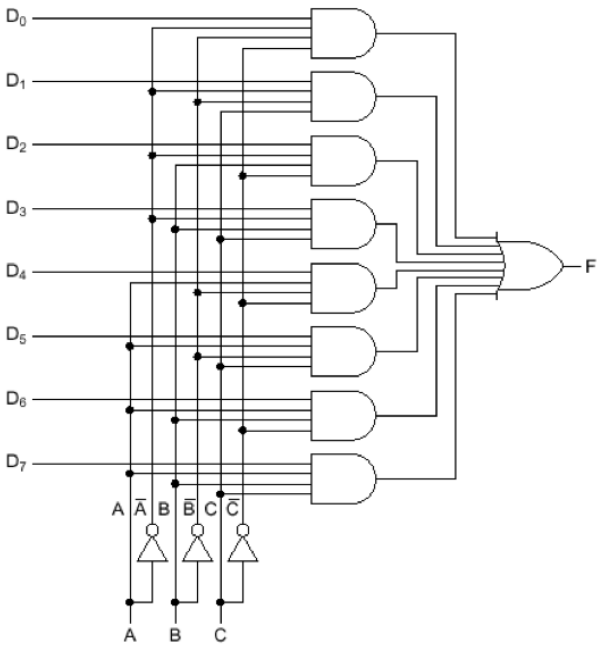
## Multiplexer 2:1

| S A \ B | 0 | 1 |
|---|---|---|
| 0 0 | 0 | 0 |
| 0 1 | 1 | 1 |
| 1 1 | 0 | 1 |
| 1 0 | 0 | 1 |

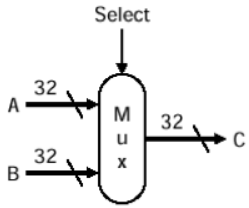
$Z = /SA + SB$

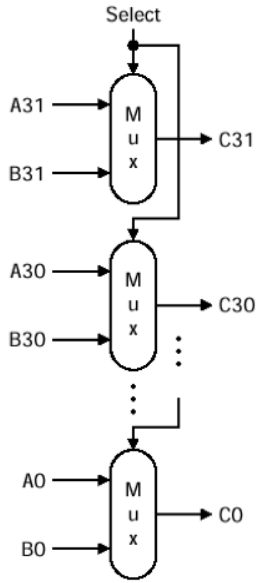



## Mux 8:1 single bit

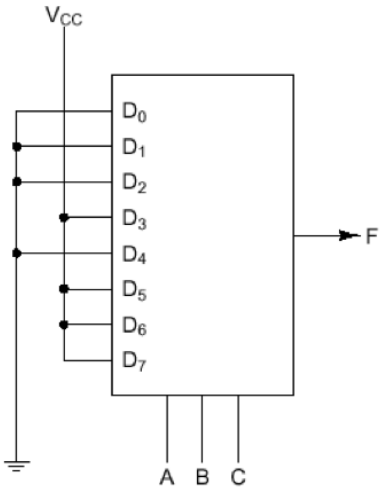



## Mux 2:1 32 bit




## Mux 8:1 come gen funzioni

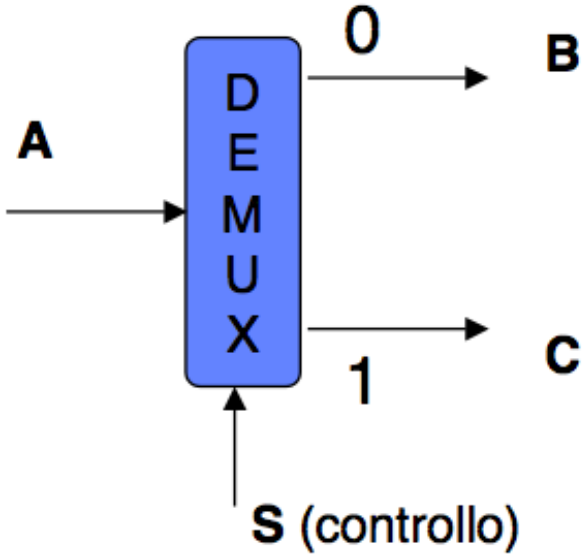



$$F = (\overline{A}BC) + (A\overline{B}C) + (AB\overline{C}) + (ABC)$$

# Esempi di circuiti combinatori: Demultiplexer

- Da 1 input a n output (scelti da select)
- $log_2 n$ segnali di controllo

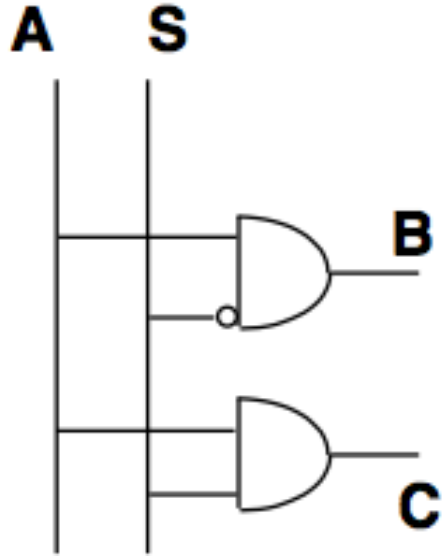
Simbolo del DeMux




| $A$ | $S$ | $B$ | $C$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

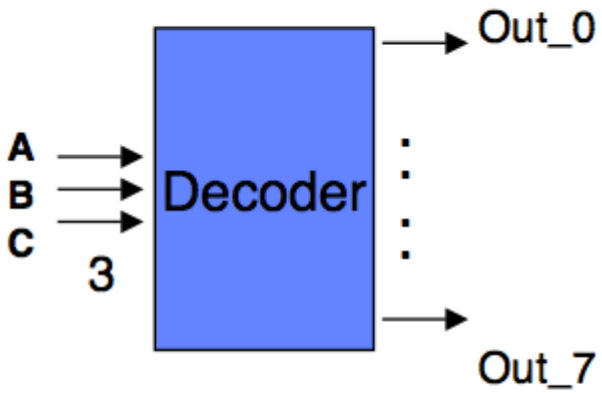
$B = A\overline{S}$, $C = AS$

DeMux schematic

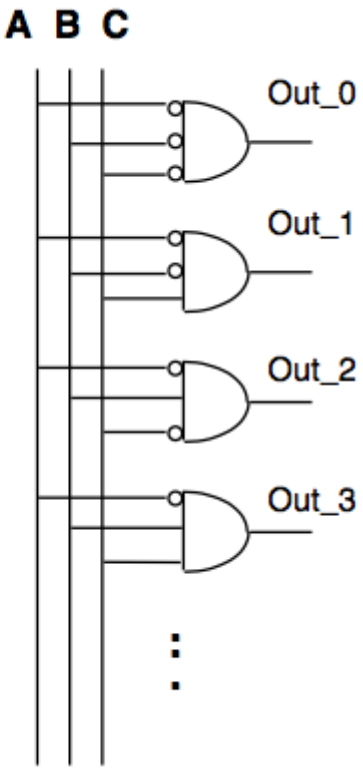


# Esempi di circuiti combinatori: Decoder

- Componente con $n$ inputs e $2^n$ outputs
- $n$ input sono un numero unsigned
- Se $n = i$ allora solo *i-esimo* bit di output uguale a 1
- Lo abbiamo visto impiegato nel blocco di selezione della locazione in un Register File



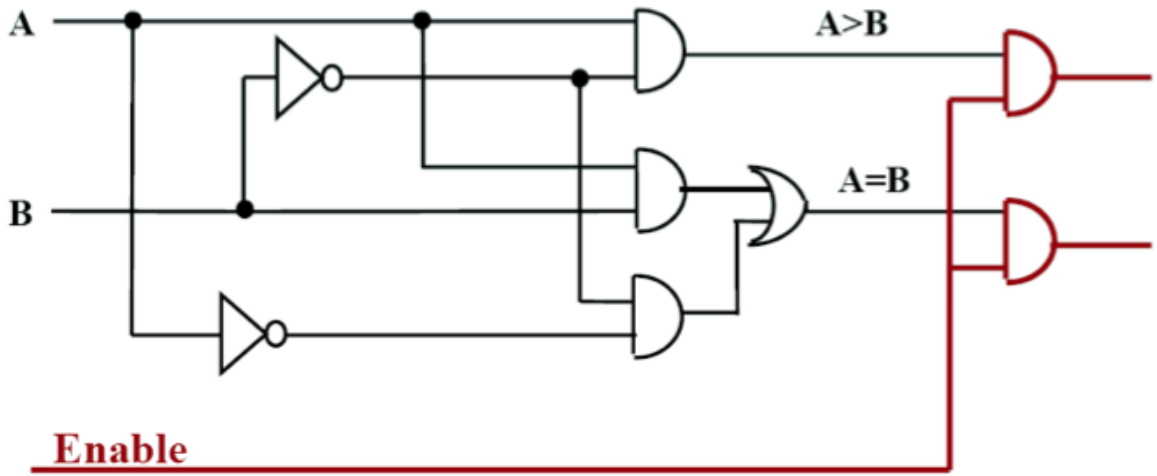

| A | B | C | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |



# Esempi di circuiti combinatori: Comparatore

- Confronto di 2 *numeri* interi positivi
- In uscita $A = B$, $A > B$
- La condizione $A < B$ ?

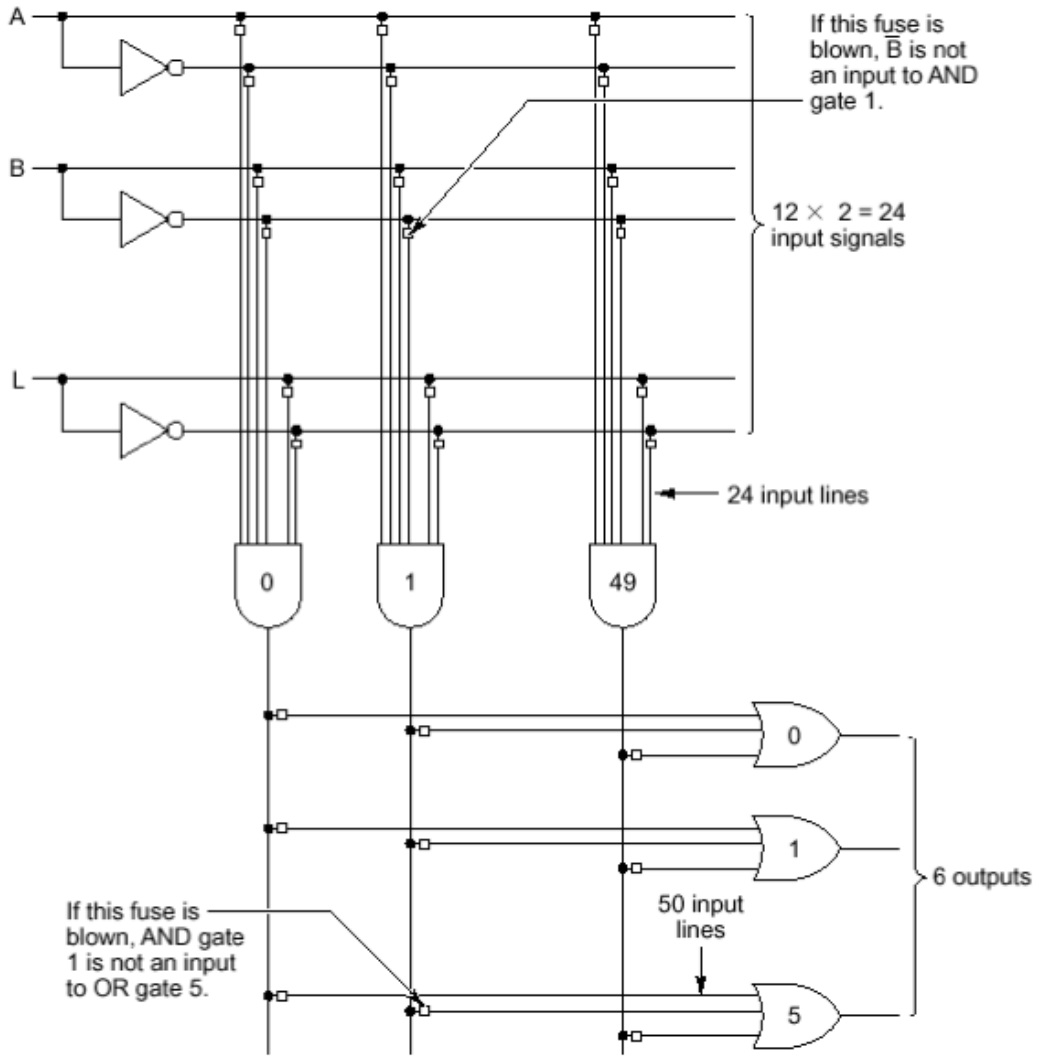
| $A$ | $B$ | $A > B$ | $A = B$ |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |



# Esempi di circuiti combinatori: PLA

*Programming Logic Array*: componente utilizzato per generare funzioni qualsiasi in termini di somma di prodotti

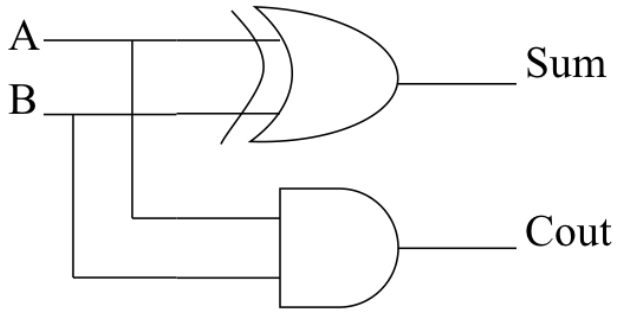
- E' una struttura con $n$ input, $o$ output, $m$ minterm esprimibili
- Strutture elettriche *fusibili* realizzano le connessioni i.e. i minterm
- La programmazione consiste nel decidere quali *fusibili* bruciare i.e. quali sono gli input ad ogni porta AND e quali quelli ad ogni port OR



# Addizionatori

- Half-Adder circuito di addizione su numeri binari unsigned ad 1 bit (non completo....)

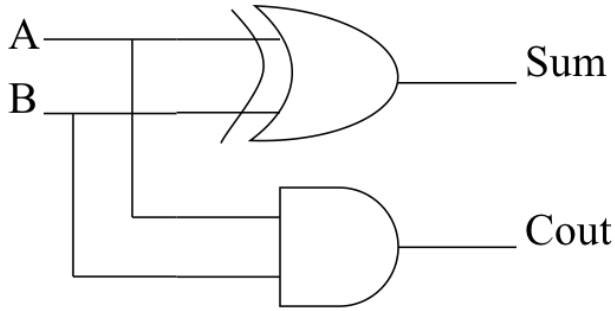
| A | B | Sum | Cout |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |



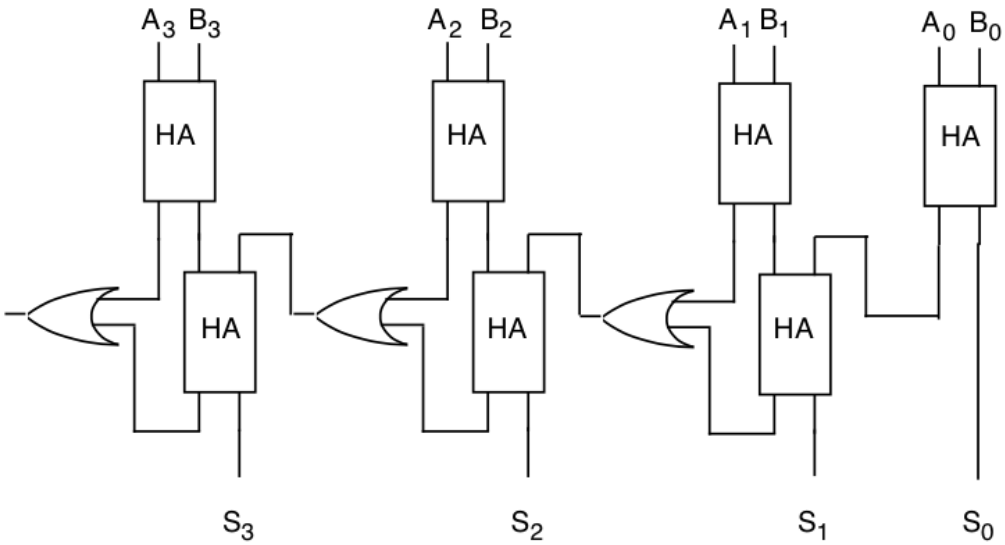
# Addizionatori

- Half-Adder circuito di addizione su numeri binari unsigned ad 1 bit (non completo....)

| A | B | Sum | Cout |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |



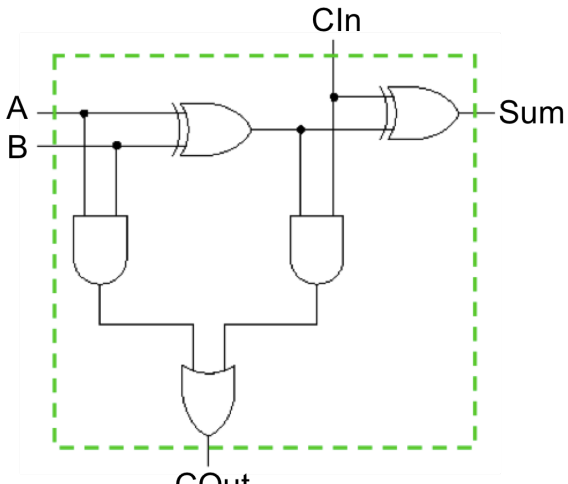

- Messi opportunamente in cascata realizzano addizionatori interi per parole a *n* bit implementando esattamente il metodo elementare *paper&pencil*



# Addizionatori

- Il *full-adder* o *sommatore completo* e' un circuito elettronico che esegue addizioni su numeri binari unsigned ad 1 bit

| $A$ | $B$ | $C_i$ | $S$ | $C_o$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |




**SUM**

| AB \ CIn | 0 | 1 |
|---|---|---|
| 0 0 | 0 | 1 |
| 0 1 | 1 | 0 |
| 1 1 | 0 | 1 |
| 1 0 | 1 | 0 |

$$S = (\overline{A}B\overline{C_i}) + (\overline{A}\overline{B}C_i) + (AB\overline{C_i}) + (ABC_i)$$

**COut**

| AB \ CIn | 0 | 1 |
|---|---|---|
| 0 0 | 0 | 0 |
| 0 1 | 0 | 1 |
| 1 1 | 1 | 1 |
| 1 0 | 0 | 1 |

$$C_o = BC_i + AB + AC_i$$