Percorsi Abilitanti Speciali – 2014
Classe: A042 (informatica)

Metodi per l'insegnamento delle architetture degli elaboratori

Prof. Federico Pedersini
Università degli Studi di Milano
Dipartimento di Informatica

Riepilogo argomenti (pre-requisiti):

1. Rappresentazione binaria dell'informazione
2. Algebra di Boole
3. Progetto di circuiti combinatori (forme canoniche, semplificazioni)
4. Circuiti combinatori notevoli (mux/demux, circuiti aritmetici, ...)
5. Circuiti con memoria notevoli (bistabili, FF, registri, contatori)
6. Progetto di circuiti sequenziali (FSM)
7. Architettura e funzionamento di un elaboratore

Rappresentazione dell'informazione

Definizione di **rappresentazione** di informazione:

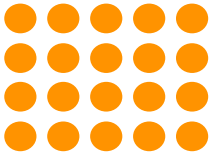
- Corrispondenza tra informazione I e parola P(I) composta da cifre a_i di un alfabeto A di simboli

$$\mathbf{I} \longrightarrow P(\mathbf{I}) = \{a_i\}, \quad a_i \in \mathbf{A}$$

ALFABETO A = $\{a_i\}$: insieme dei simboli per la rappresentazione

Esempi: {A ... Z} ; {0 ... 9} ; {0 , 1}

- I simboli dell'alfabeto possono essere di varia natura (segni su carta, suoni, livelli di tensione, fori su carta, segnali di fumo...)
- Diversi alfabeti possono essere usati per rappresentare la stessa informazione



Informazione (quantità)

$S = \{0 \cdots 9\}$ $S = \{0,1\}$ $S = \{a \cdots z\}$

20 10100 *venti*

esempi di rappresentazione dell'informazione

Capacità di rappresentazione

- Dato un alfabeto S={s_1, s_2, … s_N} composto da N simboli, quante "informazioni diverse" (quanta informazione) riesco a rappresentare con parole di k cifre?

  $C = N^k$ **C: capacità di rappresentazione**

  Quanti oggetti posso rappresentare con k bit?
  S={0,1} ➔ N=2 ➔ (2 x 2 x 2… x 2) ➔ **$C = 2^k$** oggetti
  Quanti oggetti posso rappresentare con k cifre decimali?
  (10 x 10 x 10… x 10) = **10^k** oggetti

- Date **C** informazioni diverse, quante cifre dell'alfabeto S mi servono per poterli decrivere tutti?

  $k = \log_N C$ *k intero* ➔ $k = \sup(\log_N C)$

  Quanti bit mi servono per identificare N oggetti diversi?
  Es: N = 21: (A,B,…,Z) $2^4 = 16 < 21 < 32 = 2^5 \rightarrow$ **5 bit**

Numerazione

Se l'informazione da rappresentare è una **quantità**, allora la rappresentazione è detta **numerazione**

- Numerazione DECIMALE
  - **S = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} – Base=10**
- Numerazione ESADECIMALE
  - **S = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F} – Base=16**
- Numerazione BINARIA:
  - **S = {0, 1} – Base=2**
- Numerazione OTTALE:
  - **S = {0, 7} – Base=8**

Numerazione posizionale

- Sistema di **numerazione a conteggio**
  - Sistema di numerazione mediante simboli
  - Numerazione romana: **I, V, X, L, C, M**
  - La cifra "**X**" ha sempre lo stesso valore

$$E:\langle c_k\ c_{k-1}\cdots\ c_0\rangle \qquad E=\sum_{i=0}^{k} val(c_k)$$

- Sistema di **numerazione posizionale**
  - **cifra + peso**
  - Il peso è la base elevata alla posizione della cifra.
  - La cifra "**1**" ha un valore diverso nelle parole **100** e **1000**

$$E:\langle c_k\ c_{k-1}\cdots\ c_0\rangle \qquad E=\sum_{i=0}^{k} val(c_i)\cdot b_i \quad , \quad b_i = N^i$$

Codifica posizionale di un numero

Dato un alfabeto di N elementi, detto:

base N: $B_N = \{ b_0, b_1, b_2, b_3, \ldots b_{N-1} \}$

Ciascun numero **E**, può essere rappresentato come combinazione lineare degli elementi della base:

$$E:\langle c_k\ c_{k-1}\cdots\ c_0\rangle \qquad E=\sum_{i=0}^{k} val(c_i)\cdot b_i \quad , \quad b_i = N^i$$

b_k sono i PESI delle cifre, di valore: **$b_k = N^k$**

Basi di numerazione:

- $B_2 = \{\ldots 16, 8, 4, 2, 1, ½, ¼, \ldots\}$ base 2 (binaria)
- $B_{10} = \{\ldots 1000, 100, 10, 1, 0.1, 0.01, \ldots\}$ base 10 (decimale)
- $B_{16} = \{\ldots 4096, 256, 16, 1, 1/16, 1/256, \ldots\}$ base 16 (esadecimale)

- Esempi:

$12_{10} = 1 \cdot 10^1 + 2 \cdot 10^0$

$100_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 4_{10}$

Conversione base 10 → base N

Algoritmo di conversione di un numero ***x*** in base 10 in base *N:*

- *i = 0*
- Divido il numero ***x*** per ***N***
- Resto della divisione: cifra *i*-esima in base *N*
- *i = i+1*
- Quoziente della divisione ➔ ***x***

Si prosegue fino a che il quoziente ***x* = 0**

(l'ultimo resto è la cifra più significativa del numero in base *N*)

Conversione base 10 → base 2

Esempio: vogliamo rappresentare 1492_{10} in base 2:

1492 = 2 x 746 + 0 ← *Bit meno significativo*
746 = 2 x 373 + 0
373 = 2 x 186 + 1
186 = 2 x 93 + 0
93 = 2 x 46 + 1
46 = 2 x 23 + 0
23 = 2 x 11 + 1
11 = 2 x 5 + 1
5 = 2 x 2 + 1
2 = 2 x 1 + 0
1 = 2 x 0 + 1 ← *Bit più significativo*

$1492_{10} = 10111010100_2$

Conversione: base n → base 10

Un numero a k cifre, in base *n*: $E = <c_{k-1}\ c_{k-2}\ \ldots\ c_0>$

$b_i = n^i$

si trasforma in base 10, facendo riferimento alla formula:

$$E = \sum_{i=0}^{k-1} c_i \cdot b_i = \sum_{i=0}^{k-1} c_i \cdot n^i, \quad n = 10$$

Esempio:

$\mathbf{101\ 1101\ 0100}_{\mathbf{2}} = 1x2^{10} + 0x2^9 + 1x2^8 + 1x2^7 + 1x2^6 + 0x2^5 + 1x2^4 + 0x2^3 + 1x2^2 + 0x2^1 + 0x2^0 =$

$1024 + 256 + 128 + 64 + 16 + 4 = \mathbf{1492}_{\mathbf{10}}$

Esercizi

- Si trasformino i numeri decimali: 121331, 2453, 11101 in base 3; in base 7; in base 2.
- Convertire in base 10 i numeri: 3456_7 , 121331_5 , 2453_8 , 111010101_2
- Data la base: B = { ❍ ⊙ ★ ♪ # ✔ }:
  - convertire in tale base il numero: 120_{10}
  - convertire in decimale il numero: ⊙♪✔★

Codifica esadecimale

Codifica esadecimale: base 16

molto utilizzata in alternativa alla codifica binaria

16 simboli: S_{16} = {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}

Valori: 0 … 15

Notazioni comunemente utilizzate: $9F_{16}$, 0x9F , 9Fhex

Perché base 16?

$16 = 2^4$ ➔ 1 cifra esadecimale = 4 bit

4 cifre binarie tradotte in 1 cifra esadecimale

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

8	9	A/10	B/11	C/12	D/13	E/14	F/15
1000	1001	1010	1011	1100	1101	1110	1111

Conversione binario – esadecimale

Esempi:
conversione di $1101011_{\mathbf{due}}$ in esadecimale:

- 1011_{due} → B_{hex}
- 110_{due} → 6_{hex} (viene aggiunto un "leading" 0)
- 1101011_{due} → $6B_{hex}$

conversione da esadecimale a binario

Ogni cifra esadecimale convertita in un numero binario di 4 cifre:

- 9F hex → $1001\ 1111_2$

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

8	9	A/10	B/11	C/12	D/13	E/14	F/15
1000	1001	1010	1011	1100	1101	1110	1111

Rappresentazione binaria di numeri negativi

- Codifica a **modulo e segno**: il primo bit indica il segno, il resto il numero in modulo.
- Codifica in **complemento a 1**: il numero negativo si ottiene cambiando 0 con 1 e viceversa.
- Svantaggi:
  - Ridondanti: doppia codifica per lo 0
  - Scomode per calcolo automatico
- Codifica in **complemento a 2**: il numero negativo si ottiene cambiando 0 con 1 e sommando 1.

Modulo e segno	
dec	bin
0	00
+1	01
0	10
−1	11

Compl. a 1	
dec	bin
0	00
+1	01
−1	10
0	11

Compl. a 2	
dec	bin
0	00
+1	01
−2	10
−1	11

Complemento a 2

Complemento a 2 su **N bit**

Dato numero **E** compreso nel range: $-2^{N-1} \leq E \leq 2^{N-1}-1$

- Se **E≥0** ➔ codifico **E**
- Se **E<0** ➔ codifico $E_{C2} = 2^N + E$

Proprietà:

- Il bit più significativo (**MSB**) corrisponde al **segno**
- Comoda inversione di segno
  1. inverto tutti i bit
  2. aggiungo "1"
- Comodo per calcolo automatico: sottrazione fatta come somma.

  A – B = A + (–B)

N = 3 $-2^2 \leq E \leq 2^2-1$	
codifica	valore
0 0 0	**0**
0 0 1	**1**
0 1 0	**2**
0 1 1	**3**
1 0 0	**–4**
1 0 1	**–3**
1 1 0	**–2**
1 1 1	**–1**

Sottrazione fra interi

- Sfruttando i numeri negativi, gestisco la sottrazione come una somma:

  $11 - 13 = 11 + (-13)$

- Utilizzo la rappresentazione: complemento a 2

  $+11_{10} \rightarrow 01011_2$

  $-13_{10} \rightarrow 10011_2$

- Vantaggio della rappr. a complemento a 2

```
   11 ← riporti
 01011 +
 10011 =
-----------
 11110  → -2₁₀
```


Capacità di rappresentazione interi

Rappresentazione binaria di **numeri interi**:

N bit → $\mathbf{2^N}$ valori rappresentabili

- Interi "**unsigned**" (senza segno)

  Da 0 (00…0) a $2^N - 1$ (111…1)

- Interi "**signed**"

  Da -2^{N-1} (100…0) a $2^{N-1} - 1$ (011…1)

Standard C/C++: **`int`** è rappresentato con 4 byte ➔ N=32

(signed) int: $-2.147.483.650 \le E \le +2.147.483.649$

unsigned int: $0 \le E \le +4.294.967.295$

Rappresentazione binaria di numeri **frazionari**

Conversione di un numero ***x,y*** in base 10 in base *N:*

- Per la parte intera ***x*** → vedi algoritmo precedente
- Per la parte frazionaria ***0,y***:
  - *i = 1*
  - Moltiplico ***0,y*** per ***N***
  - Parte intera: cifra decimale *i*-esima in base *N*
  - *i = i+1*
  - Parte frazionaria → ***0,y***
  - Si prosegue fino a che ***y = 0***

Problema: potrebbe **non finire mai**!

Conversione dei numeri decimali

Esempio:
$10{,}75_{10} = 1010{,}11_2$

$10 : 2 = 5, \mathbf{0}$
$5 : 2 = 2, \mathbf{1}$
$2 : 2 = 1, \mathbf{0}$
$1 : 2 = 0, \mathbf{1}$

→ 1010,...
(parte intera)

$0{,}75 \times 2 = 1.5 \rightarrow \mathbf{1}$
$0{,}5 \times 2 = 1.0 \rightarrow \mathbf{1}$

→ ...,11
(parte frazionaria)

- Errori di approssimazione:
  - arrotondamento e troncamento.

Esempio:
$10{,}76_{10} = 1010{,}1100001\ldots_2$

$0{,}76 \times 2 = 1.52 \rightarrow \mathbf{1}$
$0{,}52 \times 2 = 1.04 \rightarrow \mathbf{1}$
$0.04 \times 2 = 0.08 \rightarrow \mathbf{0}$
$0.08 \times 2 = 0.16 \rightarrow \mathbf{0}$
$0.16 \times 2 = 0.32 \rightarrow \mathbf{0}$
$0.32 \times 2 = 0.64 \rightarrow \mathbf{0}$
$0.64 \times 2 = 1.28 \rightarrow \mathbf{1}$
$0.28\ldots?$

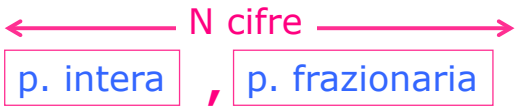
→ ,1100001
errore = $0.28 \cdot 2^{-8}$

Rappresentazione di numeri decimali

- Dato un certo numero di cifre **N** (finito) per codificare il numero decimale, esistono due tipi di codifiche possibili:

Virgola fissa (fixed point):

- lascio la virgola dov'è
- date N cifre, le divido tra parte intera e parte frazionaria



Esempio: **N=8** → **4** p. intera | **4** p. frazionaria

CAPACITÀ: MIN: 0,0001 MAX: 9999,9999

RISOLUZIONE: ΔE = 0,0001 costante!

- Capacità: 9 ordini di grandezza.
- Risoluzione insufficiente per numeri piccoli, ma esagerata per numeri grandi

Rappresentazione di numeri decimali

Virgola mobile (floating point):

- sposto la virgola dove mi fa più comodo (es. 0,xxxxxx) utilizzando la rappresentazione normalizzata (mantissa + esponente)

$$127{,}35 = 0{,}12735 \times 10^{3}$$

- date N cifre, le divido tra mantissa ed esponente

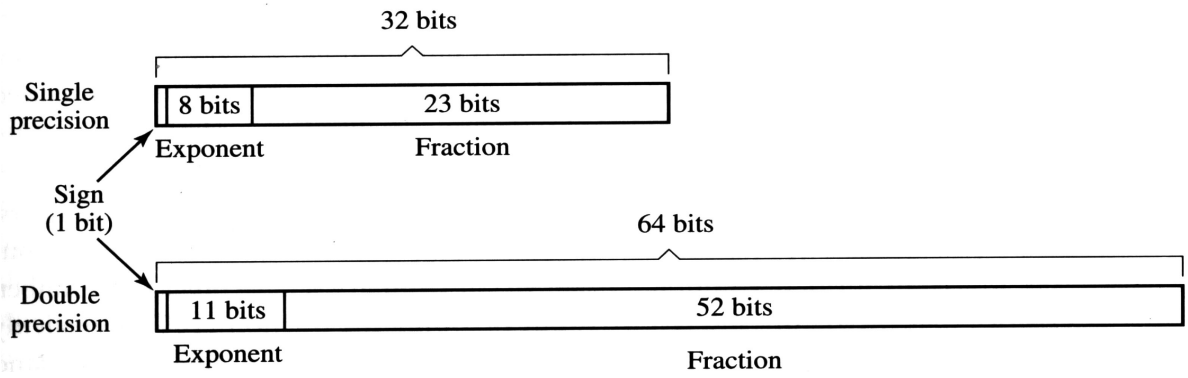
Esempio: **N=8** → **6** mantissa | **2** esponente

CAPACITÀ: Min: $0{,}000001\times10^{-50}$ Max: $0{,}999999\times10^{+49}$

RISOLUZIONE: $\Delta E = 0{,}000001\times10^{ESP}$ varia con l'esponente

- Capacità molto maggiore che in virgola fissa (100 ord. grandezza)
- La risoluzione è proporzionale all'ordine di grandezza del numero

Lo standard: IEEE-754 (1980)

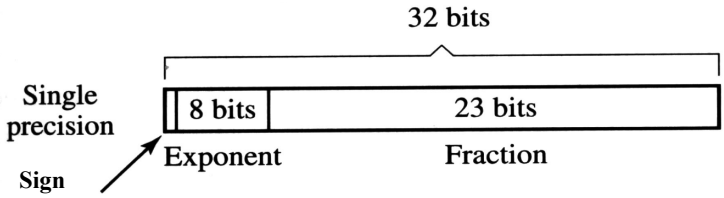


- Solo la parte frazionaria della mantissa
  - formato: 1,xxxxxxxx...
- Rappresentazione polarizzata dell'esponente:
  127 per **singola** precisione
  - 1 viene codificato come: 1000 0000

  1023 in **doppia** precisione
  - 1 viene codificato come: 100 0000 0000

Codifica standard IEEE-754

Single-precision:



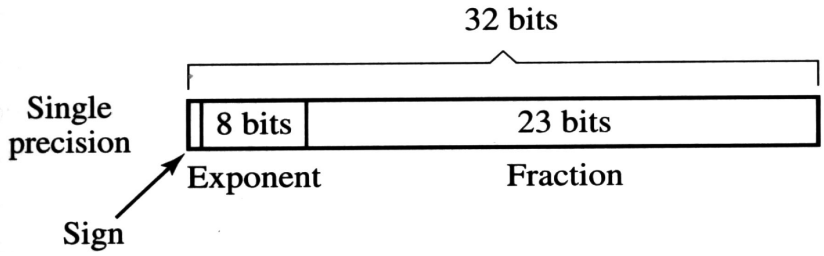
Esempio: N = –10,75

1. Conversione a binario: $-10{,}75_{10} = -1010{,}11_2$
2. Normalizzazione: $\pm 1{,}xxxxxx$ $-1{,}01011 \times 2^3$
3. Codifica del segno: 1 = "–" ; 0 = "+"
4. Calcolo dell'esponente in rappresentazione polarizzata:

$$e = 3 + 127 = 130_{10} = 10000010_2$$

10.75_{10} =
1 ___8___ _____ 23 _____
= 1 | 1000 0010 | 01011000 00000000 0000000

IEEE 754: Configurazioni notevoli

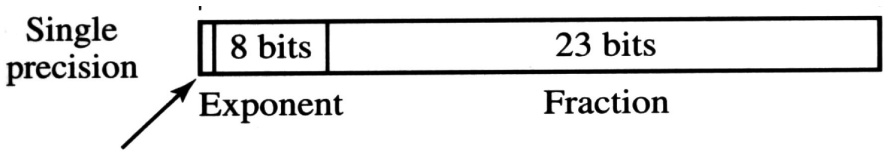


Numero	Mantissa	Esponente
0	= 0	0000 0000
∞	= 0	1111 1111
NaN	≠ 0	1111 1111
N. denormalizzato	≠ 0	0000 0000

Range esponenti: **1...254** → $-126 \le e \le +127$

Range float (32 bit): $1{,}00\ldots0 \cdot 2^{-126} \le x \le 1{,}11\ldots1 \cdot 2^{+127}$
$1.175\ldots \cdot 10^{-38} \le x \le 3.4028\ldots \cdot 10^{+38}$

IEEE-754: numeri denormalizzati



MIN_float: $1{,}00\ldots00 \cdot 2^{-126} = 1.175\ldots \cdot 10^{-38}$

Float successivo: $1{,}00\ldots01 \cdot 2^{-126} = \text{MIN\_float} + 2^{-126-23}$

→ Risoluzione float: $2^{-126-23} = 2^{-149} = 1.41298\ldots \cdot 10^{-45}$

Discontinuità tra ZERO e MIN_float !!

- Soluzione: **numeri denormalizzati**

Non si procede alla normalizzazione:

Esponente: **00...0** (si assume che sia: −126)

Mantissa: $m_1 \ldots m_{23}$ → $\mathbf{0,}m_1\ldots m_{23} \cdot 2^{-126}$

→ MIN_denorm: $0{,}00\ldots01_2 \cdot 2^{-126} = 2^{-149} \approx 1.41_{10} \cdot 10^{-45}$

Algebra di Boole
le funzioni logiche
sintesi di circuiti combinatori

George Boole, 1854:
"An Investigation of the Laws of Thought on which to found the Mathematical Theories of Logic and Probabilities"

Algebra Booleana

- Variabili binarie: FALSE(=0); TRUE(=1)
- Operatori logici sulle variabili: NOT, AND, OR

- Applicazioni:
  - Analisi dei circuiti digitali
    - Descrizione del funzionamento in modo economico.
  - Sintesi (progettazione) dei circuiti digitali
    - Data una certa funzione logica, svilupparne una implementazione efficiente.

Operatore NOT

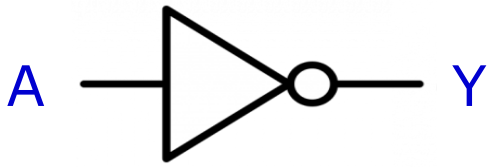
- Operazione logica di negazione
  - Se **A** è vera, **NOT(A)** è falsa

$$Y = \text{NOT}\, A = \overline{A}$$

- Operazione definita dalla **tabella della verità**
  - Funzione definita per tutte le combinazioni di variabili

Tabella della verità

A	Y
0	1
1	0



Negazione logica
("Inverter")

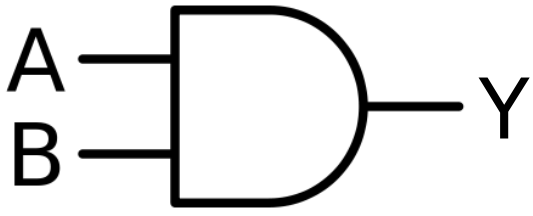
Operatore AND

- Operazione di **prodotto logico**
  - Solo se sia A che B sono veri, A AND B è vera.

$$\mathbf{Y = A\ AND\ B = A \cdot B = AB}$$

Tabella della verità

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1



Prodotto logico
(porta **AND**)

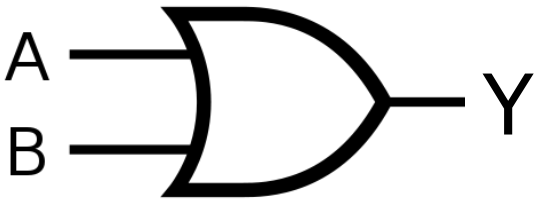
Operatore OR

- Operazione di **somma logica**
  - Se A o B sono veri, che A OR B è vera.

$$Y = A \text{ OR } B = A + B$$

Tabella della verità

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1



Somma logica
(porta **OR**)

Priorità degli operatori

- Priorità
  - In assenza di parentesi, AND ha la priorità sull'OR ed il NOT su entrambi:

NOT → AND → OR

- Esempi:

$$A \text{ OR } B \text{ AND } C = A + B \cdot C = A + (B \cdot C)$$

$$\text{NOT } A \text{ AND } C = \text{NOT } A \cdot C = (\text{NOT } A) \cdot C = \overline{A} \cdot C$$

Dualità e Postulati

- Principio di dualità
  se un'espressione booleana è vera, lo è anche la sua duale

  il DUALE di un'espressione booleana si ottiene:
  - scambiando **AND** con **OR**
    (OR→AND , AND→OR)
  - scambiando **TRUE (1)** con **FALSE (0)**
    (0→1 , 1→0)

- Postulati
  - Le proprietà commutativa, distributiva, identità, inverso sono postulati: assunti veri per definizione.
  - Le altre proprietà sono teoremi dimostrabili.

Proprietà degli operatori logici

	AND	OR (duale)
• Identità	$1 \cdot x = x$	$0 + x = x$
• Elemento 0	$0 \cdot x = 0$	$1 + x = 1$
• Idempotenza	$x \cdot x = x$	$x + x = x$
• Inverso	$x \cdot \sim x = 0$	$x + \sim x = 1$
• Commutativa	$x \cdot y = y \cdot x$	$x + y = y + x$
• Associativa	$(x \cdot y)\, z = x\, (y \cdot z)$	$(x+y) + z = x + (y+z)$
	AND rispetto a OR	**OR rispetto a AND**
Distributiva	$x \cdot (y + z) = x \cdot y + x \cdot z$	$x + y \cdot z = (x+z) \cdot (x+y)$
I assorbimento	$x \cdot (x + y) = x$	$x + x \cdot y = x$
II assorbimento	$x \cdot (\sim x + y) = xy$	$x + \sim x \cdot y = x + y$
• De Morgan	$\overline{(xy)} = \overline{x} + \overline{y}$	$\overline{(x + y)} = \overline{x} \cdot \overline{y}$

Operatore: XOR (OR esclusivo)

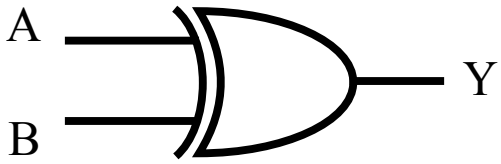
- Operazione di **mutua esclusione:**
  Y è vera se e solo se **o A o B sono veri**, ma **non entrambi**

$$Y = A\ XOR\ B = A \oplus B$$

Tabella della verità

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Porta logica XOR



Espresso mediante gli operatori fondamentali:

$$A \oplus B = A\overline{B} + \overline{A}B = \left(A + B\right)\left(\overline{A} + \overline{B}\right)$$

Proprietà: A XOR B è **VERA** quando A e B sono **DIVERSI**

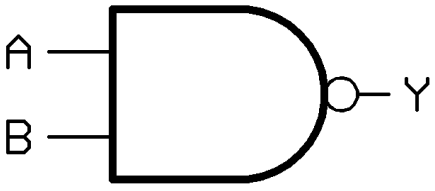
Operatore NAND

- Operatore **AND negato**

A **NAND** B = **NOT**(A **AND** B)

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

operatore "NAND"



=

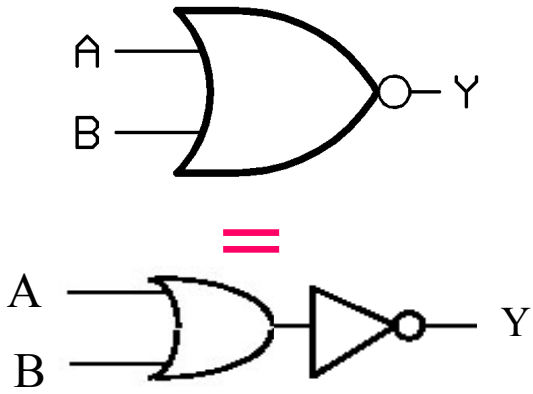
Operatore NOR

- Operatore **OR negato**

A **NOR** B = **NOT**(A **OR** B)

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

operatore "NOR"



Porte universali

Quale è il numero minimo di porte con cui è possibile implementare tutte le altre?

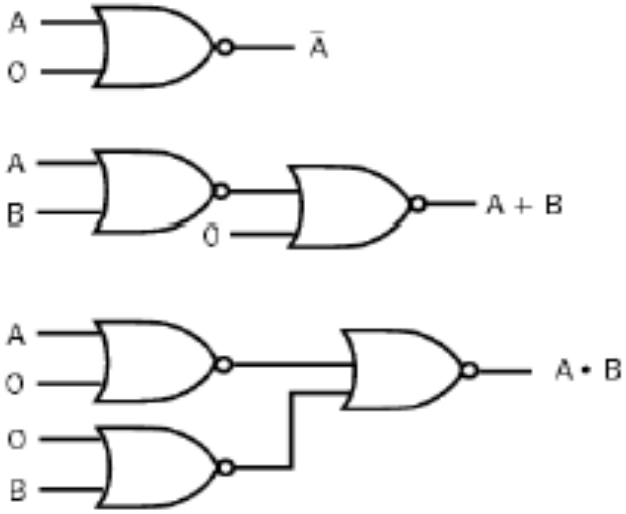
- Con la legge di De-Morgan riusciamo a passare da 3 a 2:
  - con NOT e AND si ottiene OR:

NOT(NOT(**A**) AND NOT(**B**)) = **A** OR **B**

- E' possibile usarne una sola?
  - Sì, ad esempio la porta **NAND** e la **NOR** che sono chiamate **porte universali**

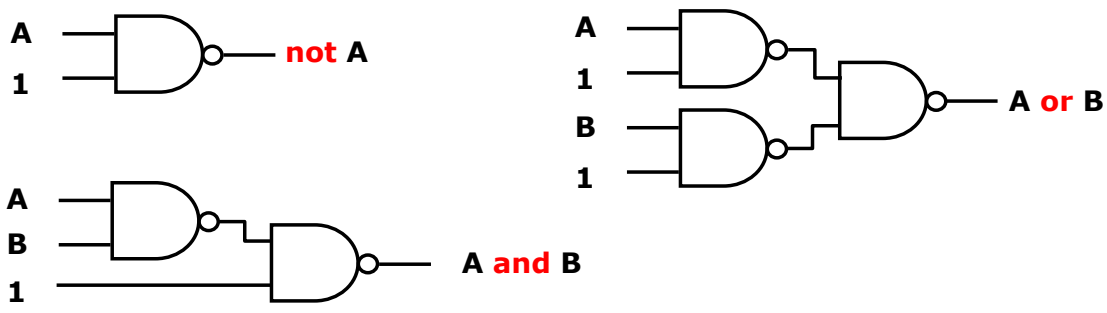
Porta Universale NOR

NOT A = 0 NOR A = A NOR A
A OR B = (A NOR B) NOR 0
A AND B = (A NOR 0) NOR (B NOR 0)



Porta Universale NAND

NOT **A** = **1** NAND **A** = **A** NAND **A**
A AND **B** = (**A** NAND **B**) NAND **1**
A OR **B** = (**A** NAND **1**) NAND (**B** NAND **1**)



Circuiti digitali

Ricordando che:

- Un oggetto di **materiale conduttore** si trova tutto allo stesso potenziale elettrico (**equipotenziale**)
- Un **generatore di tensione** (batteria, alimentatore) genera una **differenza di potenziale** tra due conduttori detti **POLI**: **positivo (+)** e **negativo (–)**

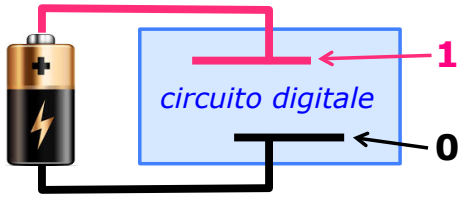
Definiamo:

- **TENSIONE** su un conduttore: **differenza di potenziale** tra il conduttore ed un conduttore di riferimento ➔ **polo negativo**

In un circuito digitale ho **2 TENSIONI possibili** per ogni conduttore:

- Tensione MASSIMA (potenziale del polo +) ➔ **"1"**
- Tensione MINIMA: 0 Volt (potenziale del polo –) ➔ **"0"**

"1": collegamento elettrico a **"+"**
"0": collegamento elettrico a **"–"**

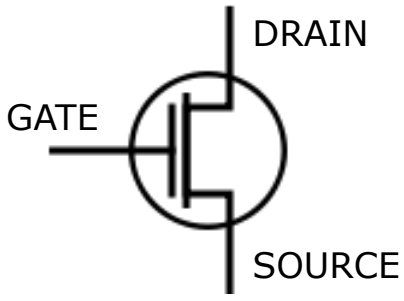


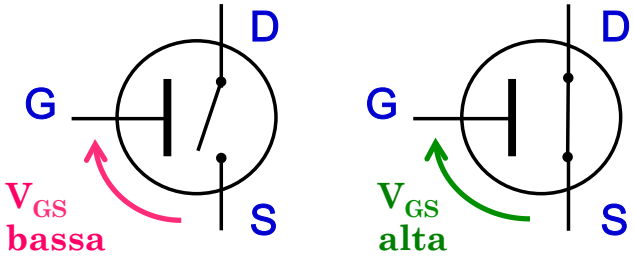
Il transistore **MOSFET**

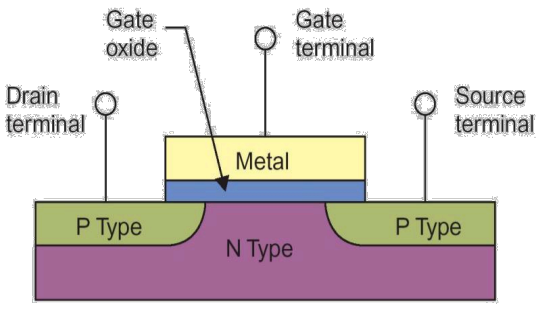
MOSFET: **M**etal-**O**xide-**S**emiconductor **F**ield **E**ffect **T**ransistor

Modello di funzionamento MOSFET:
collegamento tra **DRAIN** e **SOURCE**
comandato dalla tensione su **GATE:**

- Tensione **V_{GS} bassa → D, S isolati**
  - MOSFET in stato di **INTERDIZIONE**
- Tensione **V_{GS} alta → D, S collegati**
  - MOSFET in stato di **SATURAZIONE**





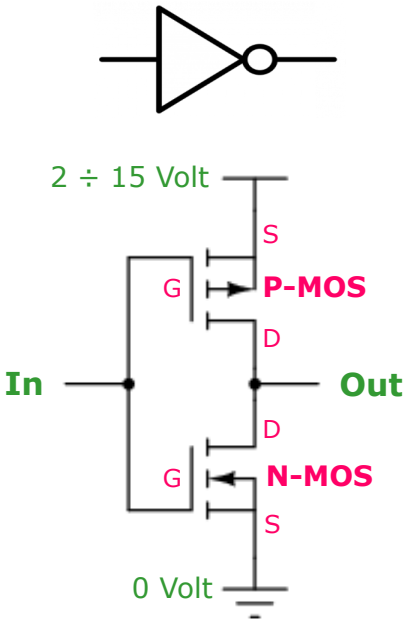


La tecnologia CMOS (1980 – oggi)

- **CMOS: Complementary–MOS**
  MOSFET a coppie complementari (**N-MOS** + **P-MOS**) che lavorano in "contrapposizione"
  - Se un **N-MOS conduce** ➔ il corrispondente **P-MOS è isolato** e viceversa

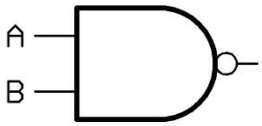
- Vantaggi tecnologia CMOS:
  - Tensione di alimentazione "flessibile":
    - $V_{CC} = 1 \div 15$ Volt
    - $V_{LOW} = 0 \div V_{CC}/2$
    - $V_{HIGH} = V_{CC}/2 \div V_{CC}$
  - Consumo bassissimo:
    - Consuma solo nella transizione
    - In condizioni statiche, consumo nullo!

Inverter CMOS

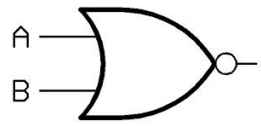


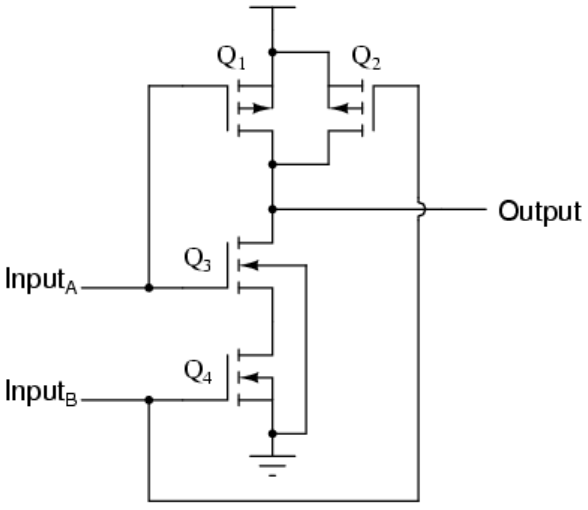
Porte CMOS

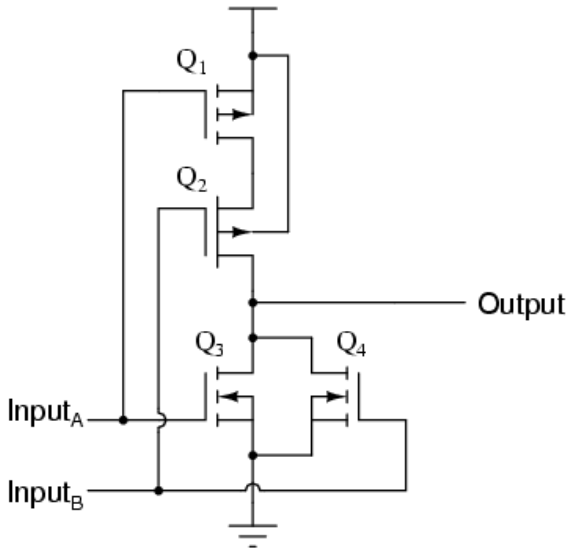
Porta **NAND**



Porta **NOR**







Funzione logica / circuito logico

Funzione logica: $f: B^n \rightarrow B$

- Funzione booleana di **n** variabili booleane
- Può essere rappresentata mediante combinazione di variabili e operatori elementari (NOT,AND,OR)
- Definita per tutte le 2^n combinazioni delle variabili (ingressi)

Può essere rappresentata in tre diversi modi:

Espressione: $Y = f(x_1, x_2, \ldots, x_n)$

Circuito logico

- **Y: Uscita,** funzione booleana di **n ingressi** (variabili) booleane

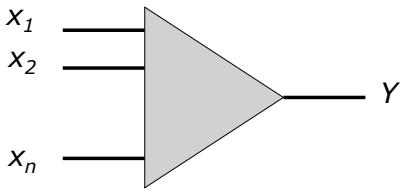


Tabella di verità (Truth Table, TT)

- Definizione della funzione per **elenco** di tutti i valori possibili delle variabili.

Funzione / circuito / tab. verità

Esempio: $F(A,B,C) = A \cdot B + B \cdot \overline{C}$

3 variabili: $F = f(A,B,C)$ → $2^3 = 8$ combinazioni possibili delle variabili

Circuito logico

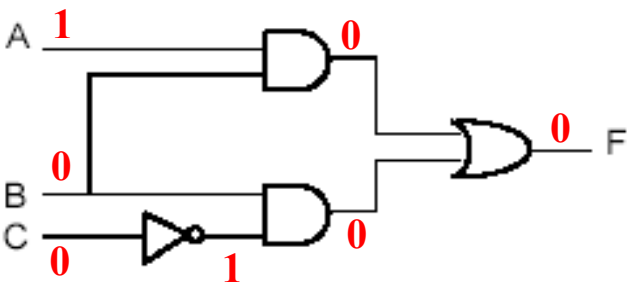


Tabella di verità

A	B	C	A•B	B•not C	F
0	0	0	0	0	**0**
0	0	1	0	0	**0**
0	1	0	0	1	**1**
0	1	1	0	0	**0**
1	0	0	0	0	**0**
1	0	1	0	0	**0**
1	1	0	1	1	**1**
1	1	1	1	0	**1**

*Data una funzione **F**,*
*esistono **infinite espressioni** e **infiniti circuiti**,*
*ma **una sola tabella di verità** che la rappresenta.*

Problema della sintesi (progetto) di circuiti combinatori:

Come passare ***da tabella di verità***

a espressione logica circuito digitale?

Data la tabella di verità:

A	B	C	F
0	0	0	0
0	0	1	0
0	**1**	**0**	**1**
0	1	1	0
1	0	0	0
1	0	1	0
1	**1**	**0**	**1**
1	**1**	**1**	**1**

F = 1 ***se e solo se:***

A = 0 AND B = 1 AND C = 0

OR

A = 1 AND B = 1 AND C = 0

OR

A = 1 AND B = 1 AND C = 1

F = 1 ***se e solo se:***

A = 0 AND B = 1 AND C = 0

OR

A = 1 AND B = 1 AND C = 0

OR

A = 1 AND B = 1 AND C = 1

$F = 1$ se e solo se:

$\left(\overline{A} = 1 \text{ and } B = 1 \text{ and } \overline{C} = 1\right)$ or

$\left(A = 1 \text{ and } B = 1 \text{ and } \overline{C} = 1\right)$ or

$\left(A = 1 \text{ and } B = 1 \text{ and } C = 1\right)$

$F = 1$ se e solo se: $\overline{A}B\overline{C} = 1$ or $AB\overline{C} = 1$ or $ABC = 1$

$F = 1$ se e solo se: $\overline{A}B\overline{C} + AB\overline{C} + ABC = 1$

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

La prima forma canonica

$$F = A \cdot B + B \cdot \overline{C} = \overline{A}B\overline{C} + AB\overline{C} + ABC$$

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Implicante:
Prodotto delle variabili (in forma naturale o negata) per le quali la funzione vale 1

Mintermine m_j:
implicante che contiene tutte le ***n*** variabili della funzione (e.g. ABC).

Prima forma canonica (SoP): $F = \sum_{j=1}^{Q} m_j \quad , \quad Q \le 2^n$

Prima forma canonica (SoP) di **F**:
la **somma logica** dei suoi **mintermini**

Somma di Prodotti

Considero i MINTERMINI (casi in cui: **F = 1**)

- MINTERMINI: prodotti di tutte le variabili, con le variabili **NEGATE** se nella tabella di verità sono **0**, **NATURALI** se sono **1**

Prima forma canonica (SoP): $F = \sum_{j=1}^{Q} m_j \quad , \quad Q \le 2^n$

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

m_2 → $\overline{A}B\overline{C} +$

m_6 → $AB\overline{C} +$

m_7 → ABC

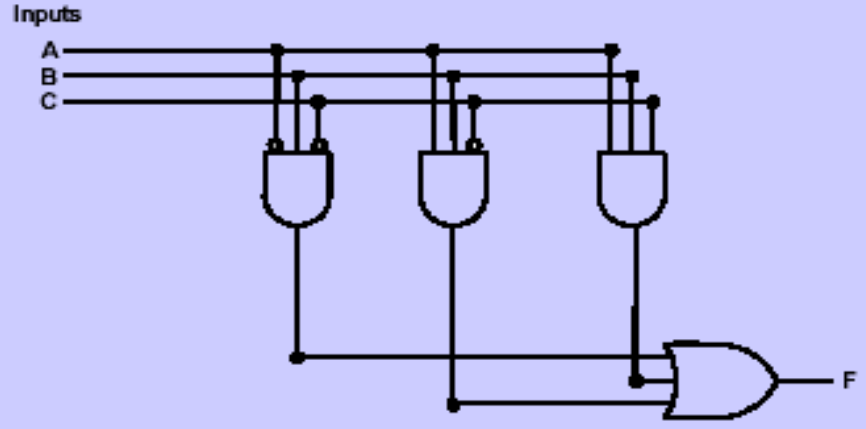
$F =$

I forma canonica: dall'espressione al circuito

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

Circuito a **due stadi**:

1. Stadio **AND**: **Q porte AND a n ingressi**, una per ogni mintermine
2. Stadio **OR**: **1 porta OR a Q ingressi**



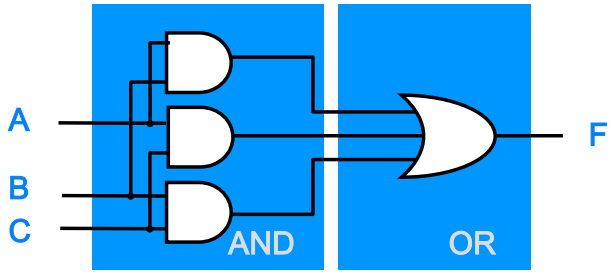
*Esercizio: **funzione maggioranza***

Funzione logica di 3 variabili ➔ 3 ingressi, 1 uscita

1. Costruzione tabella di verità o espressione logica
2. Trasformazione a forma SOP
3. Eventuale semplificazione

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

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



Seconda forma canonica

Seconda forma canonica di ***F(A,B,C)***:

- Approccio DUALE rispetto alla I forma canonica: considero i casi in cui: **F = 0**

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

F = 0 *se e solo se:*

A=0 and B=0 and C=0
OR
A=0 and B=0 and C=1
OR
A=0 and B=1 and C=1
OR
A=1 and B=0 and C=0
OR
A=1 and B=0 and C=1

Funzione: espressione / tabella di verità

F = 0 *se e solo se:*

A=0 and B=0 and C=0 OR
A=0 and B=0 and C=1 OR
A=0 and B=1 and C=1 OR
A=1 and B=0 and C=0 OR
A=1 and B=0 and C=1

$F=0$ se e solo se:
$\left(\overline{A}=1 \text{ and } \overline{B}=1 \text{ and } \overline{C}=1\right)$ or
$\left(\overline{A}=1 \text{ and } \overline{B}=1 \text{ and } C=1\right)$ or
$\left(\overline{A}=1 \text{ and } B=1 \text{ and } C=1\right)$ or
$\left(A=1 \text{ and } \overline{B}=1 \text{ and } \overline{C}=1\right)$ or
$\left(A=1 \text{ and } \overline{B}=1 \text{ and } C=1\right)$

$$F=0 \text{ se e solo se: } \overline{A}\overline{B}\overline{C}=1 \text{ or } \overline{A}\overline{B}C=1 \text{ or } \overline{A}BC=1 \text{ or } A\overline{B}\overline{C}=1 \text{ or } A\overline{B}C=1$$

$$F=0 \text{ se e solo se: } \left(\overline{A}\overline{B}\overline{C}+\overline{A}\overline{B}C+\overline{A}BC+A\overline{B}\overline{C}+A\overline{B}C\right)=1$$

$$\overline{F}=\overline{A}\overline{B}\overline{C}+\overline{A}\overline{B}C+\overline{A}BC+A\overline{B}\overline{C}+A\overline{B}C$$

Seconda forma canonica

Nuova definizione di **F:**

- Elenco dei termini per cui: **F = 0 → ~F = 1**

$$\overline{F} = \sum_{i=1}^{W} M_i \quad , \quad W \le 2^N$$

Maxtermine, M_j:
Prodotto di tutte le variabili di ingresso al quale corrisponde un valore di funzione F = 0

I forma can.: $F = \sum_{j=1}^{Q} m_j \quad , \quad Q \le 2^N \longrightarrow$ $Q + W = 2^N$

Seconda forma canonica

Esprimiamo F come: **somma di MIN-termini m_i'** per i quali **F=0**

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

$$\overline{F} = \sum_{i=1}^{W} m_i'$$

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$$\overline{F} = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$$

Seconda Forma Canonica: POS

$$\overline{F} = \overline{A}\overline{B}\overline{C} + \overline{A}\overline{B}C + \overline{A}BC + A\overline{B}\overline{C} + A\overline{B}C$$

- Negando entrambi i membri ed applicando il II teorema di De Morgan si ottiene:

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

II Forma Canonica – PoS (Product of Sums): Prodotto delle somme rappresentanti i **MAXtermini**

In generale:

$$\overline{F} = \sum_{i=1}^{W} m_i' \quad , \quad W \le 2^N$$

$$\overline{\overline{F}} = F = \overline{\left(\sum_{i=1}^{W} m_i'\right)} = (2^{\circ}\ \text{Th. De Morgan}) = \prod_{i=1}^{W} \overline{m_i'} = \prod_{i=1}^{W} M_i$$

$$m_i' = a \cdot b \cdot c \longrightarrow \overline{m_i'} = M_i = \overline{a} + \overline{b} + \overline{c}$$

Somma di Prodotti

- I termini-somma sono i casi in cui: **F = 0**

$$\overline{M_i} = 0 \longrightarrow F = 0, \quad \forall i = 1..N$$

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

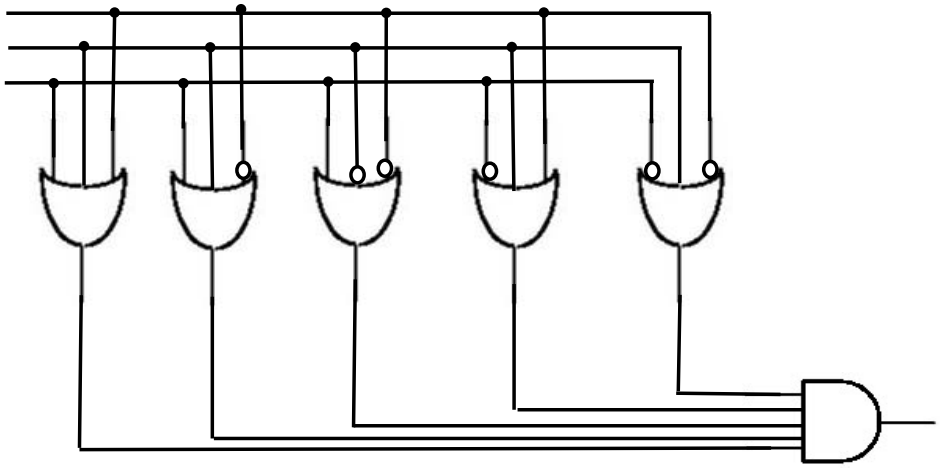
$$F = (A+B+C)\cdot \quad (M_0)$$
$$\cdot(A+B+\overline{C})\cdot \quad (M_1)$$
$$\cdot(A+\overline{B}+\overline{C})\cdot \quad (M_3)$$
$$\cdot(\overline{A}+B+C)\cdot \quad (M_4)$$
$$\cdot(\overline{A}+B+\overline{C}) \quad (M_5)$$

Circuito 2ª forma canonica: POS

Circuito a **due stadi**:

1. Stadio **OR**: **W porte OR a n ingressi**, una per ogni MAXtermine
2. Stadio **AND**: **1 porta AND a W ingressi**

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



Semplicità e prestazioni di un circuito

Criteri di valutazione delle prestazioni:

Semplicità (area)

- numero di porte in totale

Velocità (tempo di commutazione)

- numero di porte attraversate

Soddisfazione di altri vincoli

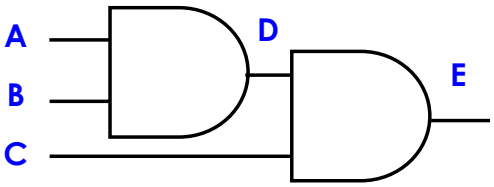
- potenza dissipata,
- facilità di debug...

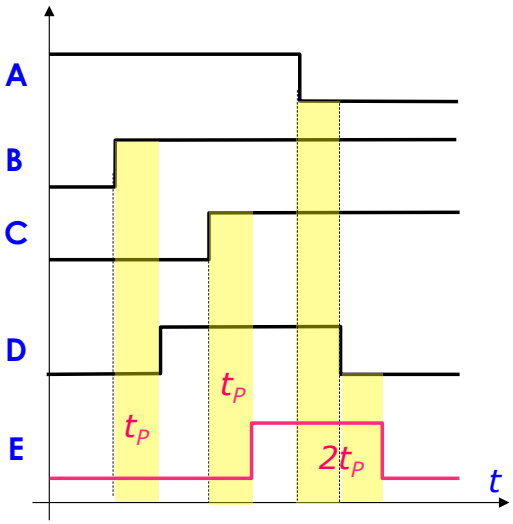
Velocità di un circuito

- Ogni circuito logico è caratterizzato da un tempo di commutazione

CAMMINO CRITICO: **massimo** numero di porte da attraversare da un qualsiasi ingresso a una qualsiasi uscita

- Non si contano gli inverters (inclusi nelle porte)

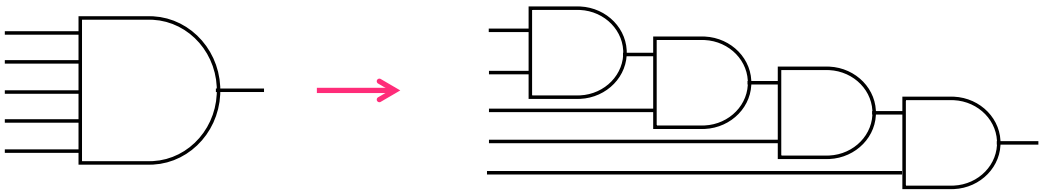




Implementazione con porte a 2 ingressi

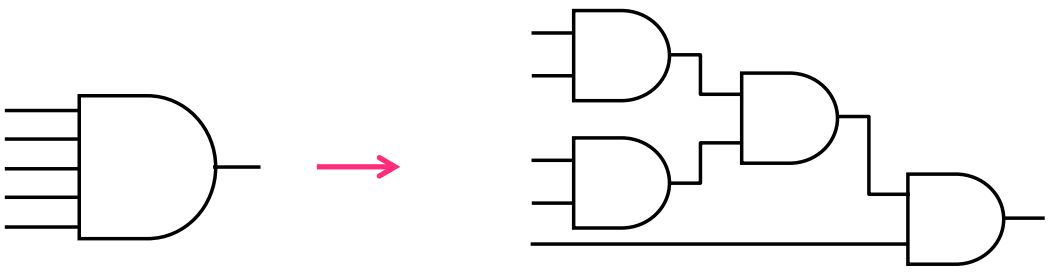
Gli elementi costruttivi di base tipici sono porte a 2 ingressi

- **Porta a N ingressi → N–1 porte a 2 ingressi**



Porta a **N** ingressi → Cammino Critico: **N-1**

Ottimizzazione del cammino critico:



Porta a **N** ingressi → Cammino Critico: **$\log_2(N)$**

Esempio di semplificazione algebrica

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

- raccolgo: $B\cdot \overline{C}$

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

- inverso: $\overline{A} + A = 1$

$$= 1\cdot B\cdot \overline{C} + A\cdot B\cdot C =$$

- identità: $(1\cdot B = B)$

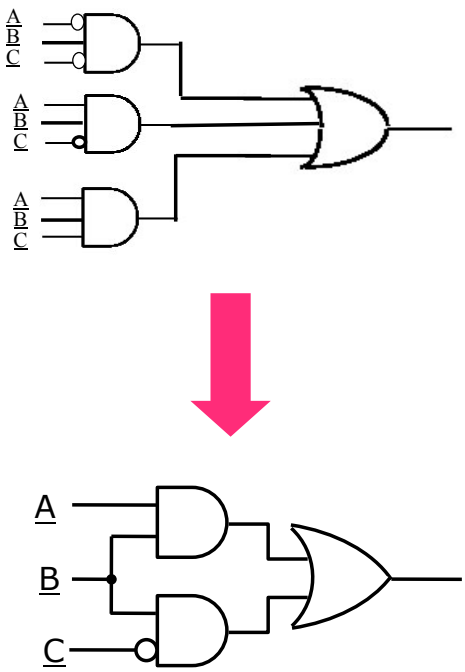
$$= B\cdot \overline{C} + A\cdot B\cdot C$$

- raccolgo: B

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

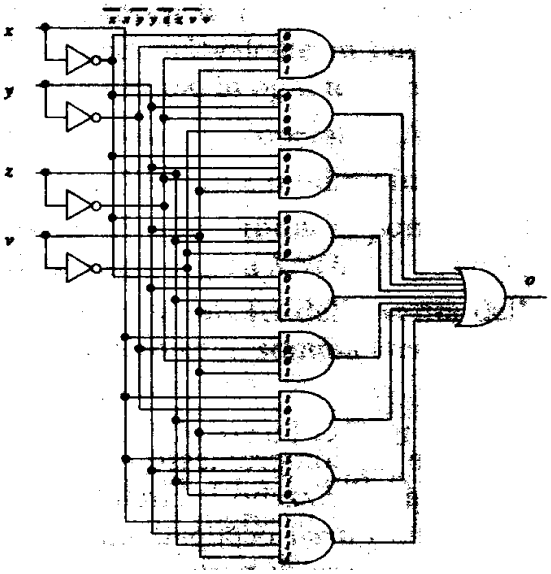
- II legge assorb.: $(A + \overline{A}B = A + B)$

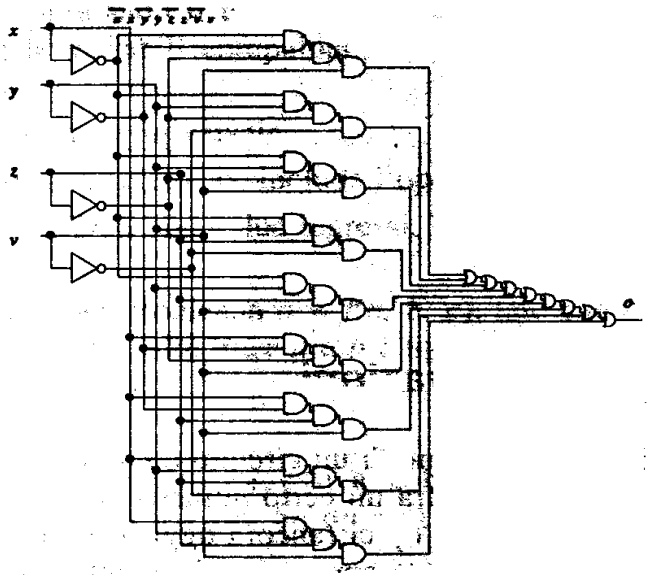
$$= B\cdot \left(\overline{C} + A\right) = AB + B\overline{C}$$



Riduzione a circuiti con porte a 2 ingressi

$$O = \overline{x}\,\overline{y}\,\overline{z}\,\overline{v} + \overline{x}\,\overline{y}\,z\,\overline{v} + \overline{x}\,y\,\overline{z}\,\overline{v} + \overline{x}\,y\,z\,\overline{v} + x\,\overline{y}\,\overline{z}\,\overline{v} + x\,\overline{y}\,z\,\overline{v} + x\,y\,\overline{z}\,\overline{v} + x\,y\,z\,\overline{v} + x\,y\,z\,v$$





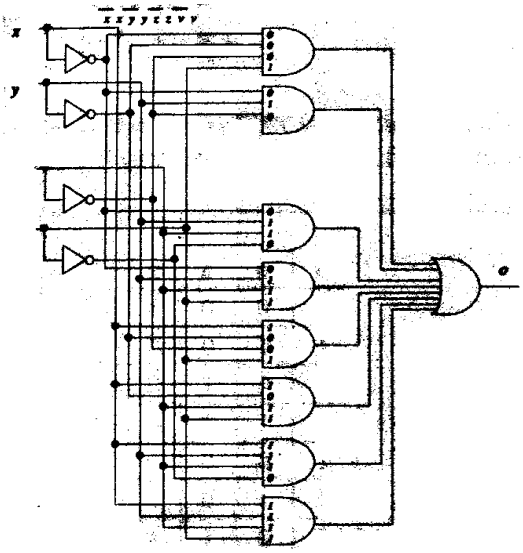
Cammino critico = 2 , N. porte = 10

Cammino critico = 11 , N. porte = 35

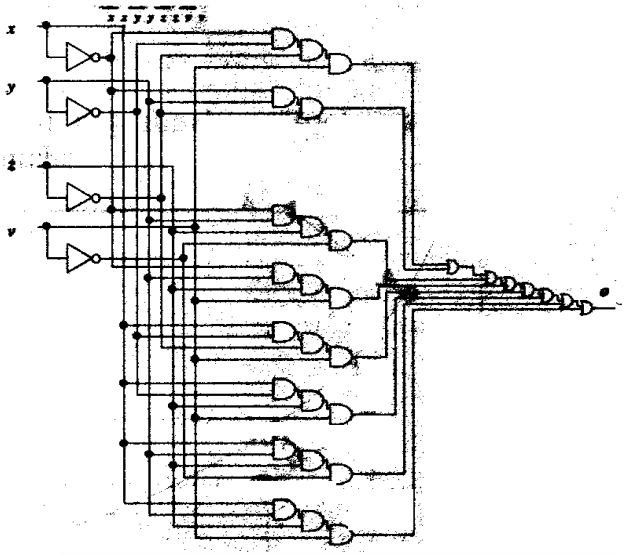
Semplificazione

- Semplificando la prima parte dell'espressione logica...

$$\bar{x}\bar{y}zv + \bar{x}y\bar{z}v = \bar{x}y\bar{z}\left(\bar{v} + v\right) = \bar{x}y\bar{z}$$



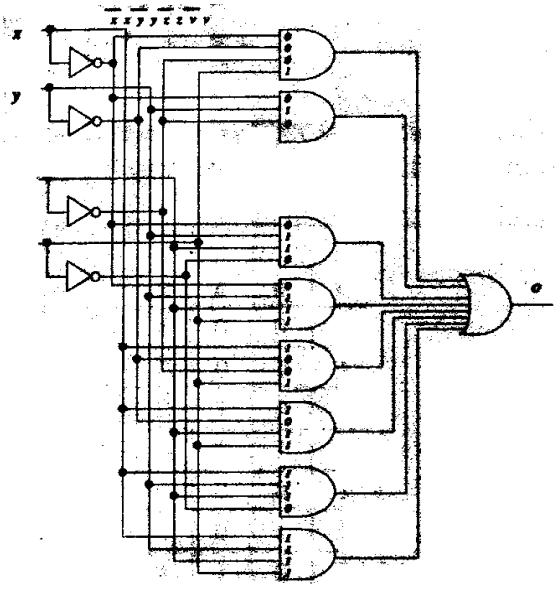
Cammino critico = 2 N. porte = 9

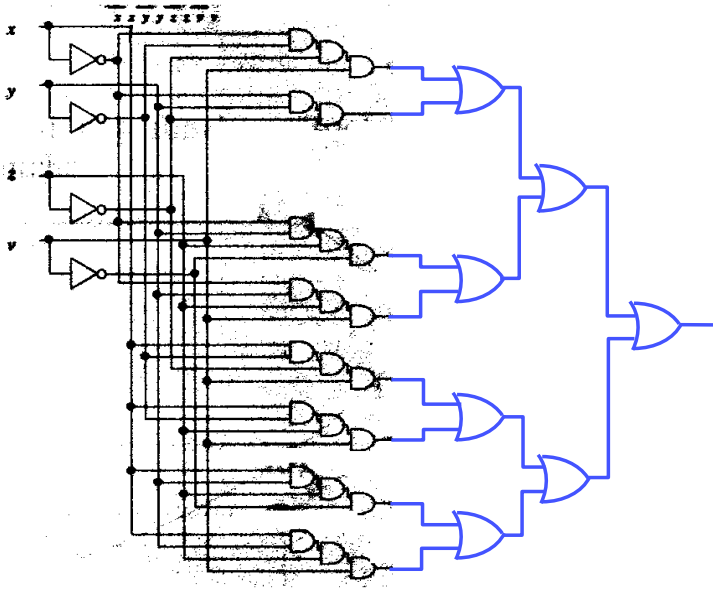


Cammino critico = 10 N. porte = 30

Ottimizzazione del cammino critico

- Collegando le porte in modo ottimizzato, si riduce significativamente il cammino critico...





Cammino critico = 6 N. porte = 30

Semplificazione

$$O = \bar{x}\bar{y}\bar{z}\bar{v} + \bar{x}y\bar{z}\bar{v} + \bar{x}yz\bar{v} + \bar{x}yzv + \bar{x}\bar{y}zv + \bar{x}yzv + xyz\bar{v} + xyzv$$

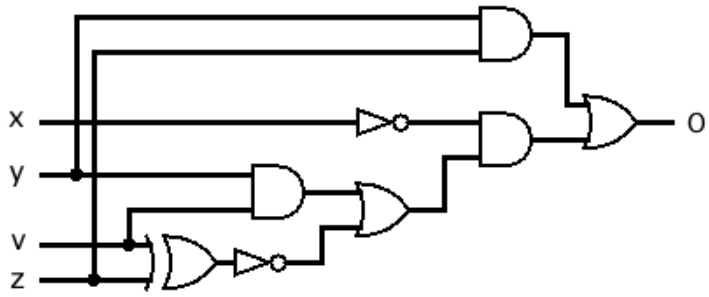
$$= \bar{x}\bar{z}\bar{v}\left(y+\bar{y}\right) + \bar{x}yz\bar{v} + \bar{x}yz\left(\bar{v}+v\right) + \bar{x}zv\left(\bar{y}+y\right) + xyz\left(\bar{v}+v\right) =$$

$$= \bar{x}\left(\bar{z}\bar{v} + zv + yz\bar{v}\right) + yz = \bar{x}\left(\bar{z}\bar{v} + v\left(z+\bar{z}y\right)\right) + yz =$$

$$= \bar{x}\left(\bar{z}\bar{v} + v\left(z+y\right)\right) + yz =$$

$$= \bar{x}\left(\bar{z}\bar{v} + vz + vy\right) + yz =$$

$$= \bar{x}\left(\overline{\left(v \oplus z\right)} + vy\right) + yz$$



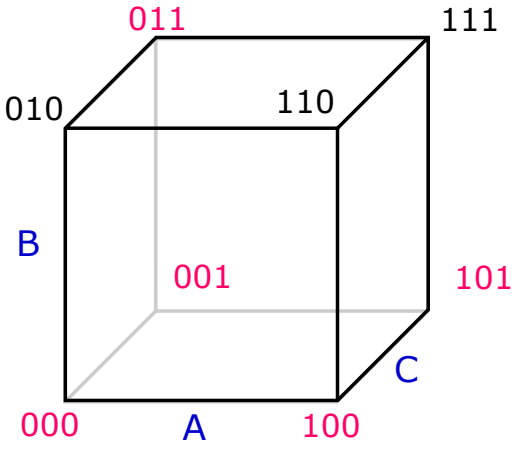
Cammino critico = 5 N. porte = 8

Semplificazione: mappe di Karnaugh

- Tecnica di semplificazione, a partire dalla tabella di verità
- Esempio: funzione di 3 variabili
  - Rappresentazione cubica di funzioni logiche a 3 variabili: F = *f*(a,b,c)
  - Muovendosi sui lati, la configurazione di variabili cambia di un solo bit
  - Distanza di HAMMING: ***d(v1, v2)*** = n. di bit diversi tra le sequenze

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

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

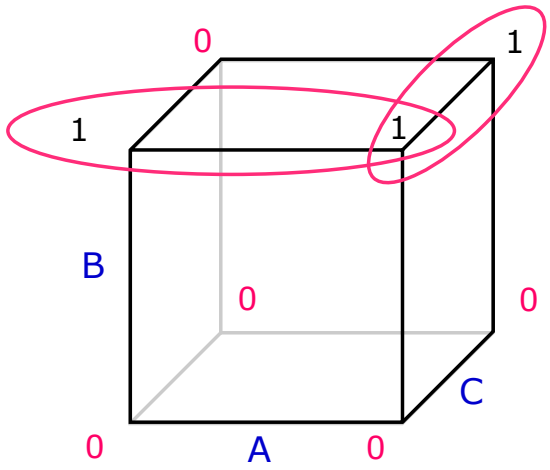


Semplificazione: mappe di Karnaugh

- **Copertura**: ricerca di tutti gli implicanti
- Se i vertici di un lato sono entrambi 1, l'implicante è indipendente dalla variabile corrispondente al lato

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

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



- Per **N>3** variabili, la rappresentazione diviene complessa...

Semplificazione: mappe di Karnaugh

- Rappresentazione piana della funzione:
  - "srotolo" il cubo
  - Codifica di Gray (codice riflesso) lungo ogni direzione

AB / C	00	01	11	10
0	000	010	110	100
1	001	011	111	101

indipendente da a: **b~c**

indipendente da c: **ab**

AB / C	00	01	11	10
0	0	1	1	0
1	0	0	1	0

F = ab + b~c

Semplificazione: mappe di Karnaugh

- Rappresentazione piana, utilizzabile per: **2 ≤ N ≤ 4**

B \ A	0	1
0	1	0
1	1	0

F = ~a

CD \ AB	00	01	11	10
00	0	1	1	0
01	0	0	1	0
11	1	1	1	1
10	0	0	1	0

F = ab + cd + b~c~d

Semplificazione: mappe di Karnaugh

- Mappa di Karnaugh: rappresentazione piana e ciclica

CD \ AB	00	01	11	10
00	0	1	1	0
01	0	0	1	0
11	1	0	1	1
10	0	0	1	0

F = ab + b~c~d + ~bcd

Uscite indifferenti di una funzione logica

Situazione tipica in sintesi (progetto) di funzioni/circuiti logici:

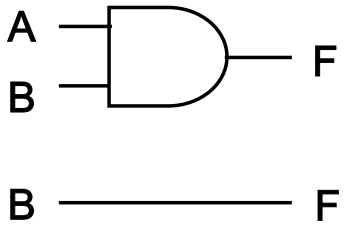
- *Per alcune combinazioni degli ingressi, il valore assunto dall'uscita è INDIFFERENTE*
  - **Simbolo: X**

- Come si risolve?
  - **Si sceglie il caso che rende il circuito più semplice**

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

X=0 → F=AB

X=1 → F=B



Esercizi

Si progetti un circuito caratterizzato da un ingresso a 4 bit rappresentante un numero binario intero senza segno A, e un'uscita che vale '1' se e solo se:

(A<4 ed è divisibile per 2) oppure
(4≤A<8) oppure
(A≥8 ed è divisibile per 4).

a) Determinare la tabella di verità della funzione logica di uscita;
b) scrivere la funzione nella forma canonica più adatta;
c) semplificarla mediante mappa di Karnaugh.

Generatore di parità dispari su 3 bit:

Si progetti un circuito caratterizzato da 3 ingressi (a,b,c) e da un'uscita P tale che:
P = 1 se e solo se il n. di "1" sugli ingressi è dispari

a) Determinare la tabella di verità della funzione logica di uscita;
b) semplificarla mediante mappa di Karnaugh;
c) semplificarla ulteriormente, se possibile, mediante trasformazioni algebriche;
d) disegnarne il corrispondente circuito digitale.

Lezione 5

Circuiti combinatori notevoli e circuiti aritmetici

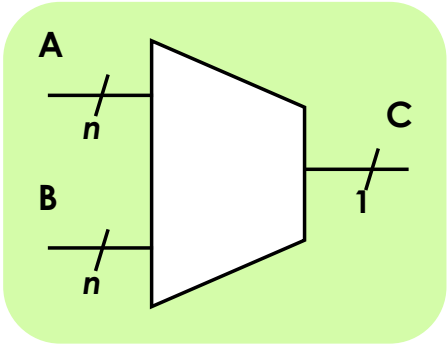
F. Pedersini

Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano

Comparatore

COMPARATORE

- Confronta parole di **n** bit
  - IN: **2** gruppi di **n** bit
  - OUT: **1** bit
  - OUT = 1 se i due IN sono uguali
  - OUT = 0 se diversi.



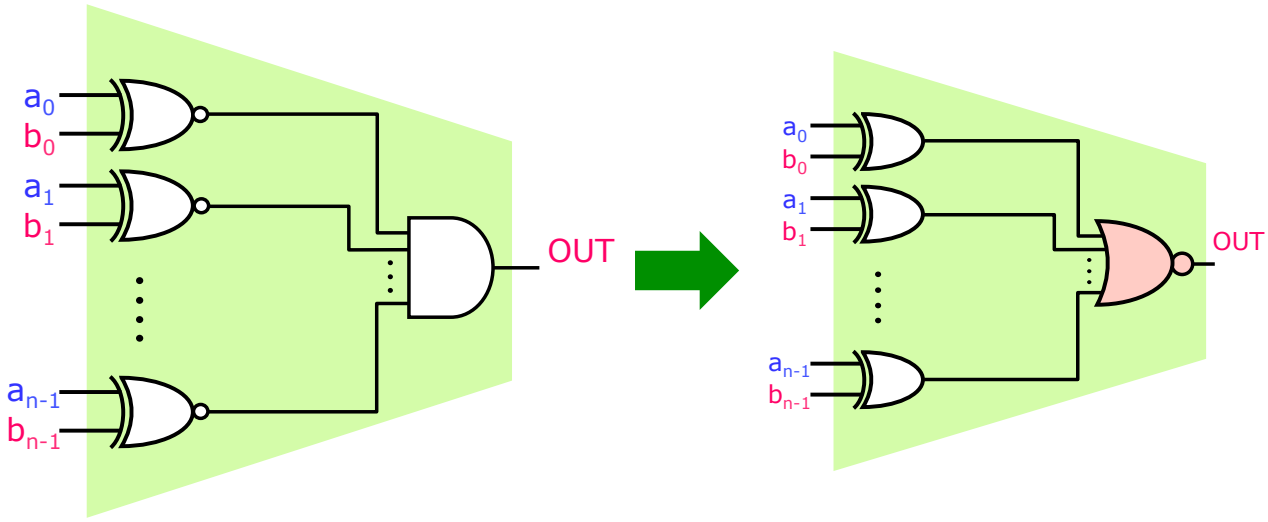
A_0	B_0	C_0
0	0	1
0	1	0
1	0	0
1	1	1

A_1	B_1	C_1
0	0	1
0	1	0
1	0	0
1	1	1

$$c_i = \overline{(a_i \oplus b_i)}$$

Comparatore

- Comparatore a n ingressi: schema circuitale



n porte **XNOR** + 1 **AND** ad **n** ingressi

n porte **XOR** + 1 **NOR** ad **n** ingressi

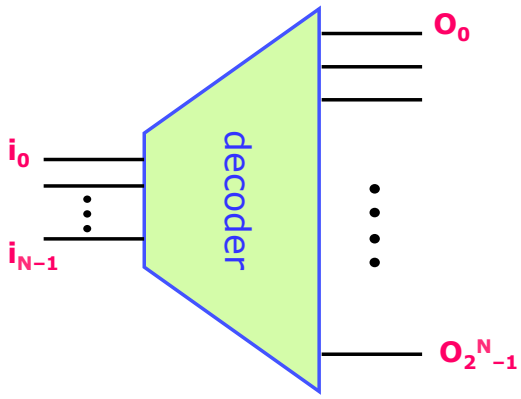
Decodificatore (decoder)

DECODER:

N ingressi, 2^N uscite

- Dato un numero ***I*** espresso sugli ingressi, viene asserita (porre a "1") l'uscita O_I corrispondente (tutte le altre uscite a "0")

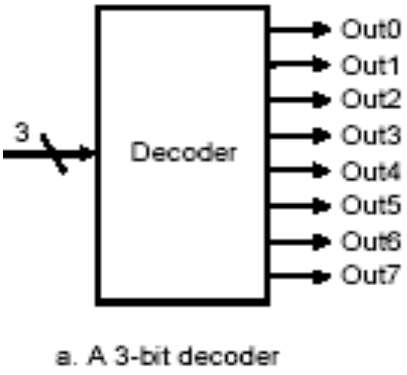
- Utilizzato ad es. nelle memorie, per selezionare una cella mediante il suo indirizzo.



La funzione decoder

Decoder a **3 ingressi ➔ 2^3=8 uscite**

Tabella di verità:



a. A 3-bit decoder

A	B	C	U_0	U_1	U_2	U_3	U_4	U_5	U_6	U_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

Ogni uscita corrisponde ad uno dei mintermini

- 2^N uscite ←→ 2^N mintermini

Cammino critico ?

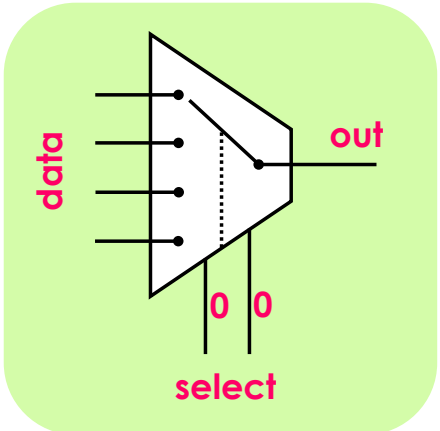
Multiplexer

MULTIPLEXER (**MUX**):

Operatore di **selezione**

IN: **n** linee di input (**data**)
k linee di controllo (**select**)

OUT: **1** linea



Funzionamento:
il valore fornito sulla linea di controllo viene connessa all'uscita la linea di ingresso selezionata.

Quante linee di selezione?

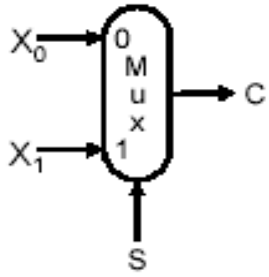
$$\mathbf{k} = \text{ceil}\ (\log_2 n)$$

Linee di input: $n = 4$
Linee di controllo: $k = ceil(\log_2 4) = 2$

Multiplexer binario

MUX a **2 ingressi**: **n = 2, k = 1**

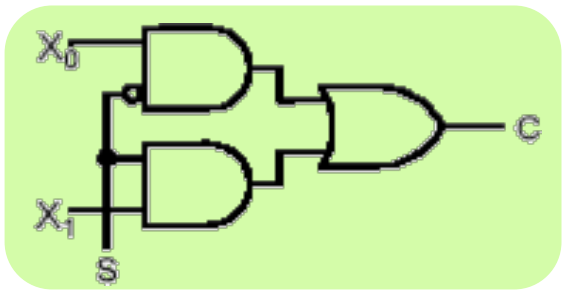
- Selezione S "apre" la porta opportuna
- Circuito logico a **3 ingressi, 1 uscita**



S	X_0	X_1	C
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**

I forma canonica (SoP):

$$C = \overline{S}X_0\overline{X_1} + \overline{S}X_0X_1 + S\overline{X_0}X_1 + SX_0X_1 =$$
$$= \overline{S}X_0\left(\overline{X_1} + X_1\right) + SX_1 + \left(\overline{X_0} + X_0\right)$$
$$= \overline{S}X_0 + SX_1$$



Sintesi Multiplexer

*Esercizio: sintesi MUX a 2 ingressi in **II forma canonica***

$$Y = \left(S + X_0 + X_1\right)\left(S + X_0 + \overline{X_1}\right)\left(\overline{S} + X_0 + X_1\right)\left(\overline{S} + \overline{X_0} + X_1\right) =$$

$$\begin{cases} a = \overline{S} + X_1 \\ b = S + X_0 \end{cases}$$

$$= \left[\left(b + X_1\right)\left(b + \overline{X_1}\right)\right] \cdot \left[\left(a + X_0\right)\left(a + \overline{X_0}\right)\right] =$$

$$= \left[b + b\left(X_1 + \overline{X_1}\right) + X_1\overline{X_1}\right] \cdot \left[a\right] = ba =$$

$$= \left(S + X_0\right)\left(\overline{S} + X_1\right) = \quad \textit{(PoS)}$$

S	X_0	X_1	C
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**

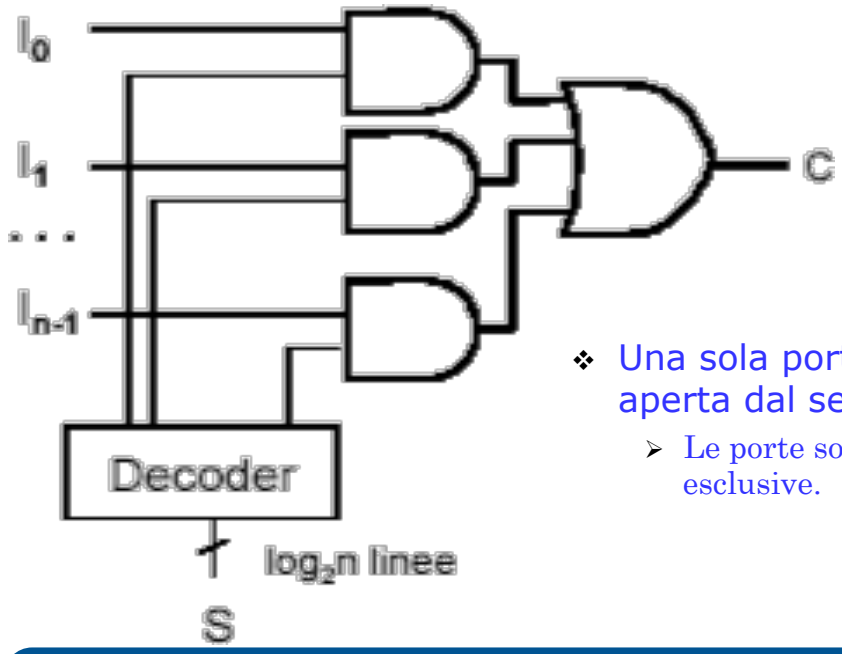
$$= S\overline{S} + SX_1 + \overline{S}X_0 + X_0X_1 =$$

$$= SX_1 + \overline{S}X_0 + X_0X_1\left(S + \overline{S}\right) = SX_1\left(1 + X_0\right) + \overline{S}X_0\left(1 + X_1\right) = SX_1 + \overline{S}X_0 \quad \textit{(SoP)}$$

Multiplexer (Mux) a più vie.

MUX a **N** ingressi:

- si utilizza un DECODER: $\log_2(N)$ ingressi, N uscite



- Una sola porta alla volta viene aperta dal segnale S.
  - Le porte sono mutuamente esclusive.

Demultiplexer (DEMUX)

Demultiplexer (DEMUX):

1 ingresso

N linee di selezione

2^N uscite

Funzionamento:

- Le linee di selezione determinano su quale uscita viene propagato l'ingresso.
- Tutte le altre uscite valgono "0".

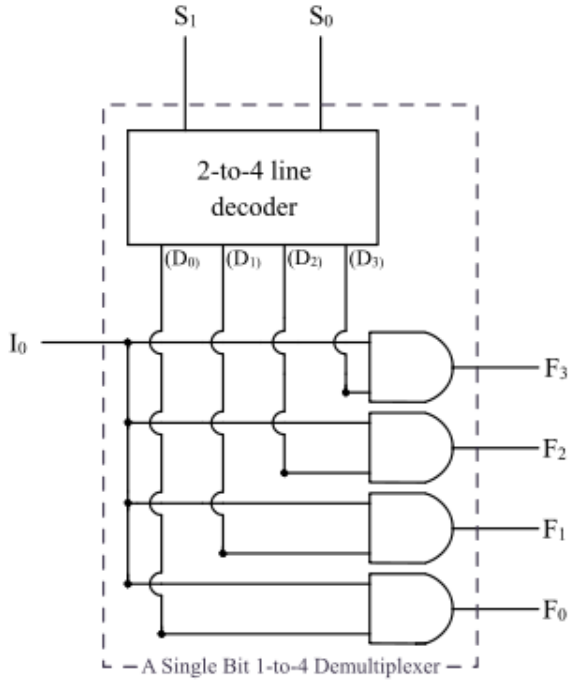


immagine da: Wikipedia

Circuiti aritmetici

circuiti combinatori che generano risultati
di operazioni aritmetiche su numeri binari

- somma
- moltiplicazione
- ALU

HALF Adder (1 bit)

- Somma aritmetica tra 2 bit
  - 2 ingressi: addendi: **a** , **b**
  - 2 uscite: somma: **s**
    riporto: **r**

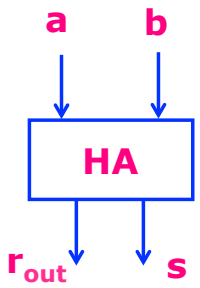
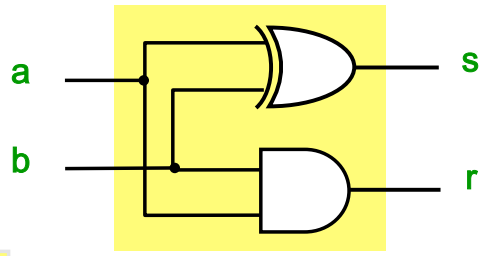


Tabella della verità

a	b	somma	riporto
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

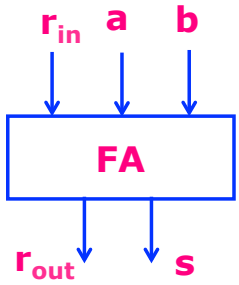


$s = a \oplus b$
$r = a \bullet b$

FULL Adder (1 bit)

Somando numeri di più bit, c'è un problema:
va gestito anche il riporto in ingresso

- 3 ingressi: **a, b, r_{IN}**
- 2 uscite: **s, r_{OUT}**



a	b	r_{in}	r_{out}	s
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

SOP:

$$s = m_1 + m_2 + m_4 + m_7$$
$$r_{out} = m_3 + m_5 + m_6 + m_7$$

$$s = \bar{a}\bar{b}r_{in} + a\bar{b}\bar{r}_{in} + \bar{a}b\bar{r}_{in} + abr_{in}$$
$$r_{out} = ab\bar{r}_{in} + \bar{a}br_{in} + a\bar{b}r_{in} + abr_{in}$$

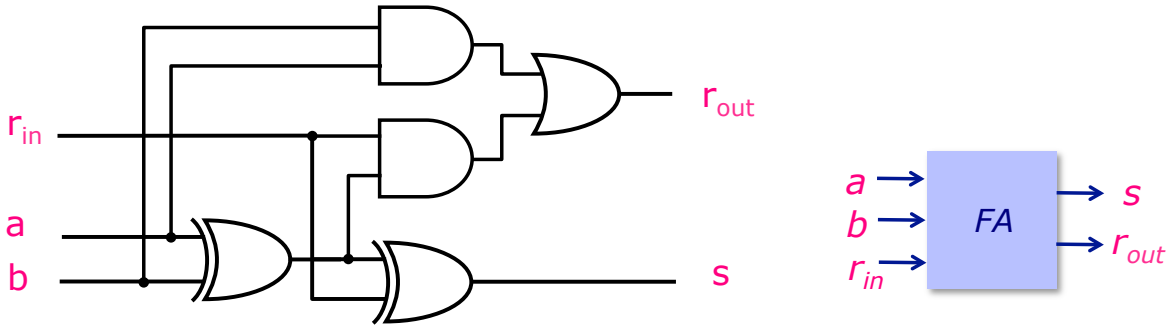
Semplificando le espressioni

$$s = \bar{a}\bar{b}r_{in} + a\bar{b}\bar{r}_{in} + \bar{a}b\bar{r}_{in} + abr_{in} =$$
$$= (a \oplus b)\bar{r}_{in} + (ab + \bar{a}\bar{b})r_{in} =$$
$$= (a \oplus b)\bar{r}_{in} + \overline{(a \oplus b)}r_{in} =$$
$$= (a \oplus b) \oplus r_{in}$$

$$r_{out} = ab\bar{r}_{in} + \bar{a}br_{in} + a\bar{b}r_{in} + abr_{in} =$$
$$= ab(r_{in} + \bar{r}_{in}) + (a\bar{b} + \bar{a}b)r_{in} =$$
$$= ab + (a \oplus b)r_{in} = \cdots = ab + (a + b)r_{in}$$

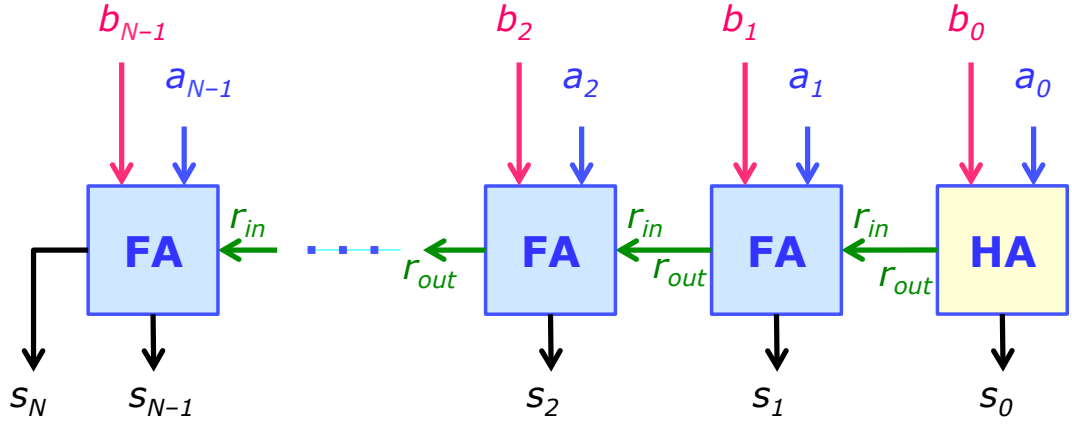
Implementazione circuitale

$$s = (a \oplus b)\overline{r_{in}} + \overline{(a \oplus b)}r_{in} = (a \oplus b) \oplus r_{in}$$
$$r_{out} = ab + (a \oplus b)r_{in}$$



Sommatore di parole di N bit

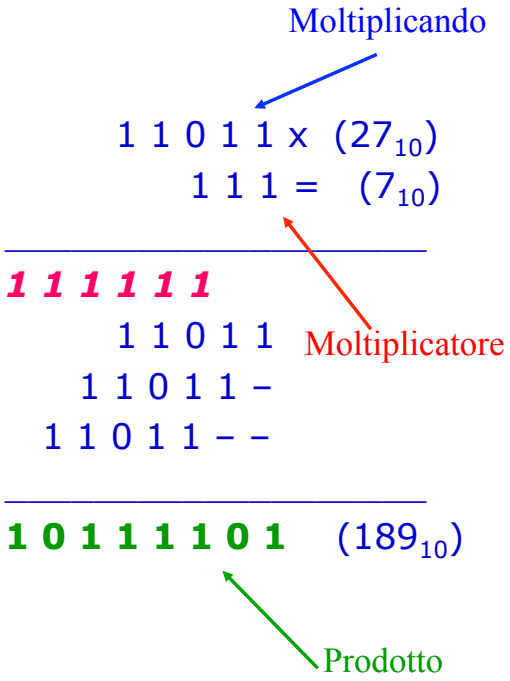
- Sommatore a propagazione di riporto
  - **IN**: **2** parole di **N** bit
  - **OUT**: somma di **N+1** bit



Cammino critico? (HA=1 ; FA=3)

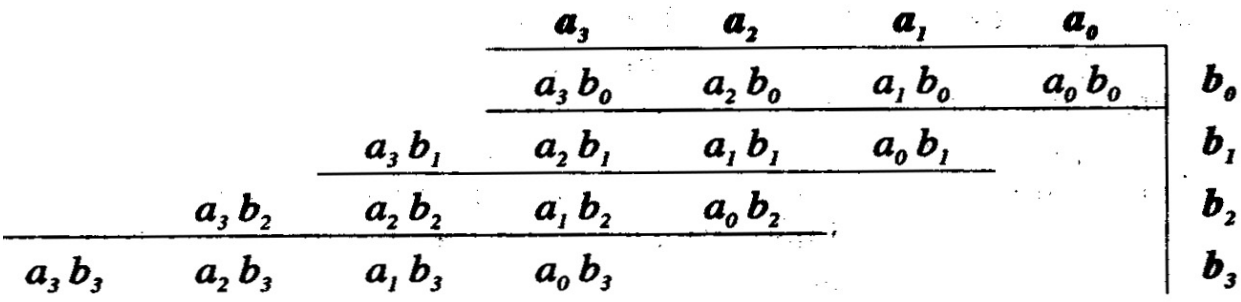
$C = 3(N-1)+1 = 3N - 2$

Moltiplicazione binaria



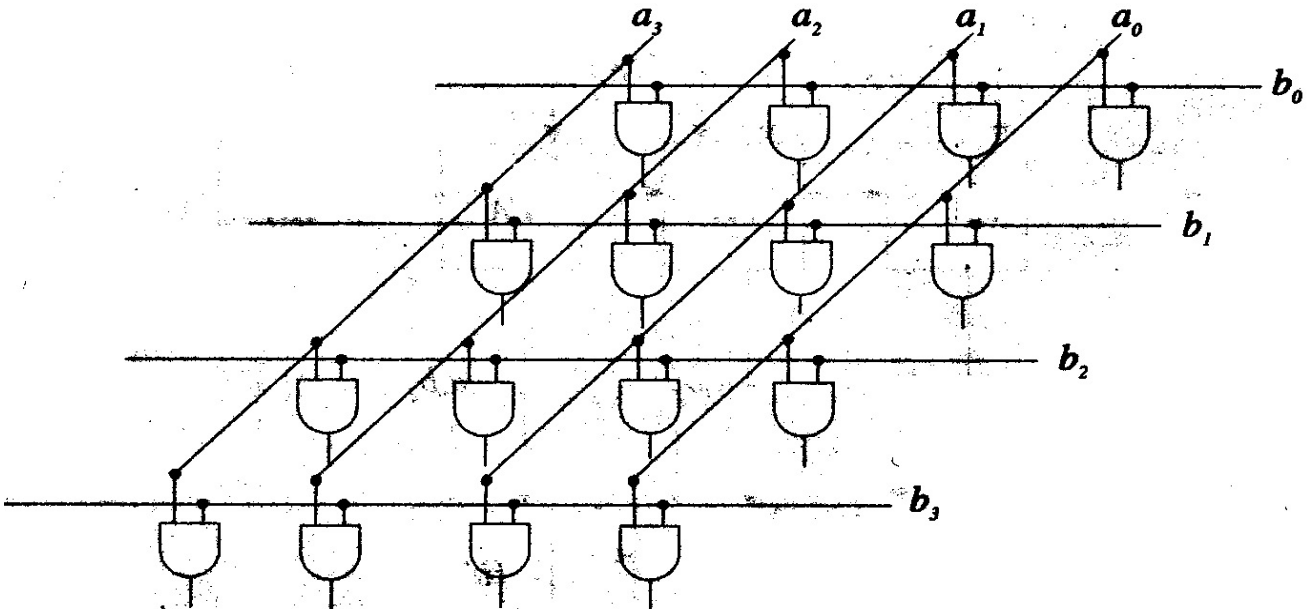
- Come fare il calcolo con circuiti logici ?
  - Possiamo scomporre l'operazione in due stadi:
  - Primo stadio: prodotti parziali
    - si mette in AND ciascun bit del moltiplicatore con i bit corrispondenti del moltiplicando
  - Secondo stadio: somme
    - si effettuano le somme (full adder) dei bit sulle righe contenenti i prodotti parziali

La matrice dei prodotti parziali

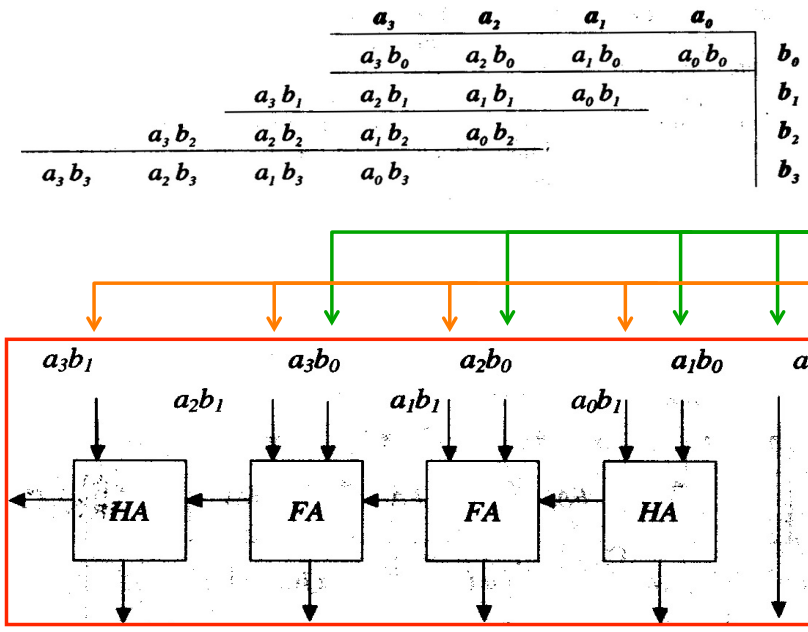


In binario i prodotti parziali sono degli AND

Il circuito che effettua i prodotti

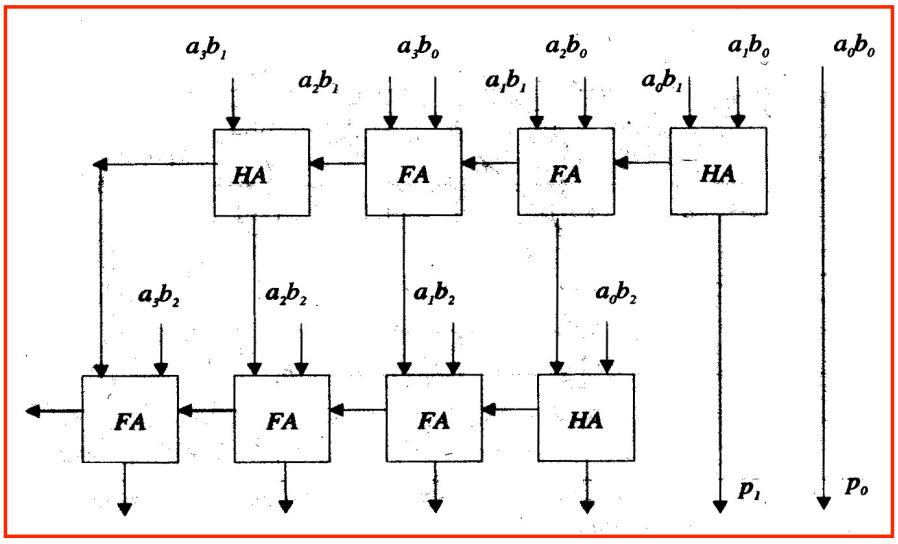


Somma – prime 2 righe dei prodotti parziali



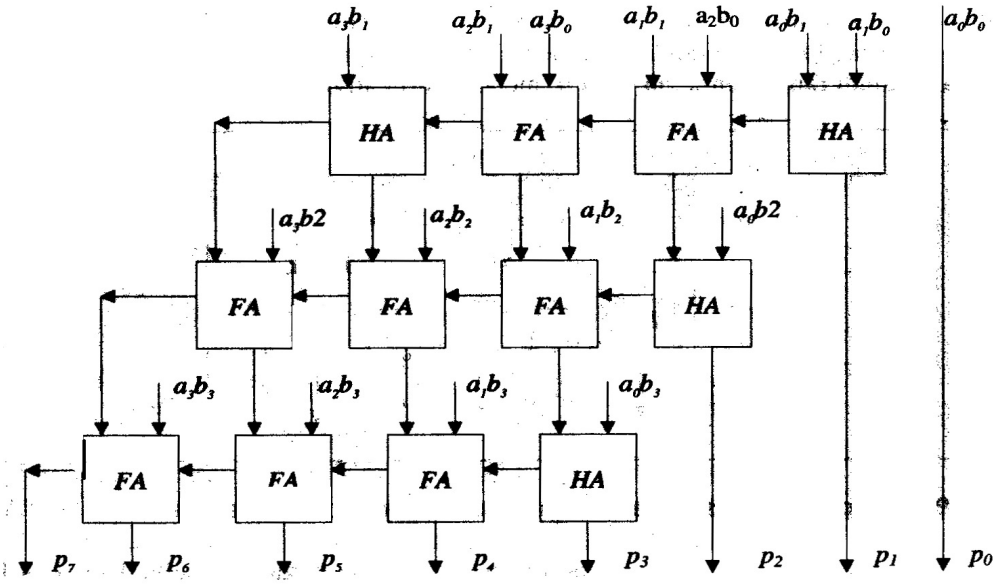

```
      1 0 1 1   x
      0 1 1 1   =
  ---------------
        1 1 1
      1 0 1 1   +
    1 0 1 1 -
  ---------------
    1
  1 0 0 0 0 1   +
  1 0 1 1 - -
  ---------------
1 1 0 1 1 0 1
```

Somma della terza riga




```
    1 1 0 1 1  x
      1 1 1  =
-------------
  1 1 1 1 1
    1 1 0 1 1 +
  1 1 0 1 1 –
-------------
1
  1 0 1 0 0 0 1 +
  1 1 0 1 1 –
-------------
1 0 1 1 1 1 0 1
```

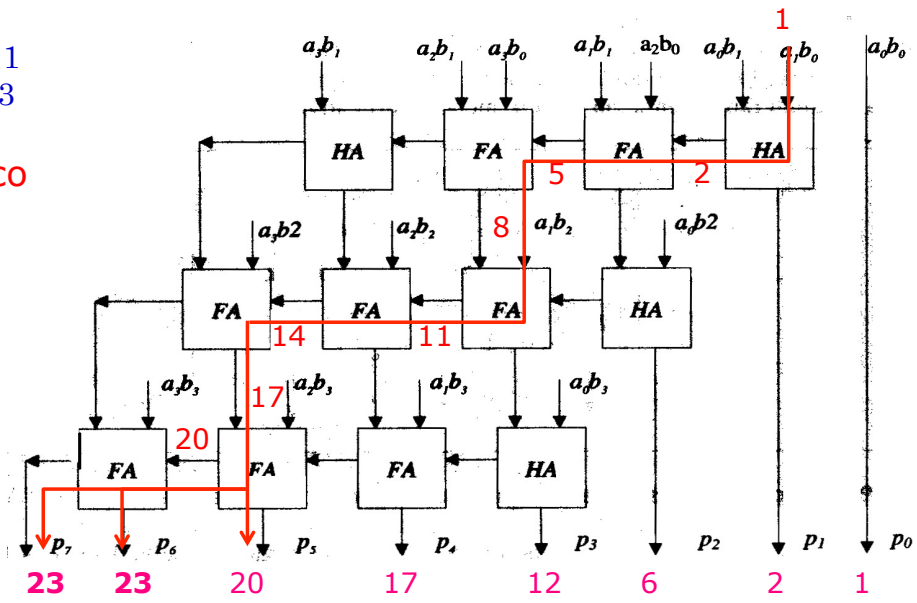

Somma prodotti parziali – circuito completo



A e B su: **N bit** → P su: **2N bit**

Sommatore a 4 bit: cammino critico

- **N = 4**
  - Ritardo AND = 1
  - Ritardo HA = 1
  - Ritardo FA = 3
- Cammino critico = **23**



ALU: Arithmetic-Logic Unit

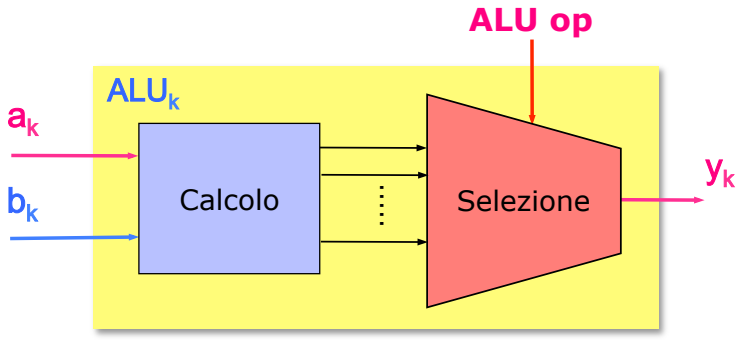
- Esegue le operazioni aritmetico-logiche
  + , − , x , ÷ , ...
  and , or , not , xor , = , ≠ , ...
- Normalmente integrata nel processore
  - Inizio anni '90 → introduzione con il nome di co-processore matematico
  - Le ALU non compaiono solamente nei micro-processori
- E' un circuito combinatorio
  - Rappresentabile come insieme di funzioni logiche
- Opera su **parole (N bit)**
  - 6502, 8080, Z-80 8 bit
  - MIPS, 80386: 32 bit
  - PowerPC G5, Athlon64: 64 bit
- Struttura modulare
  - Blocchi funzionali da 1 bit, replicati N volte

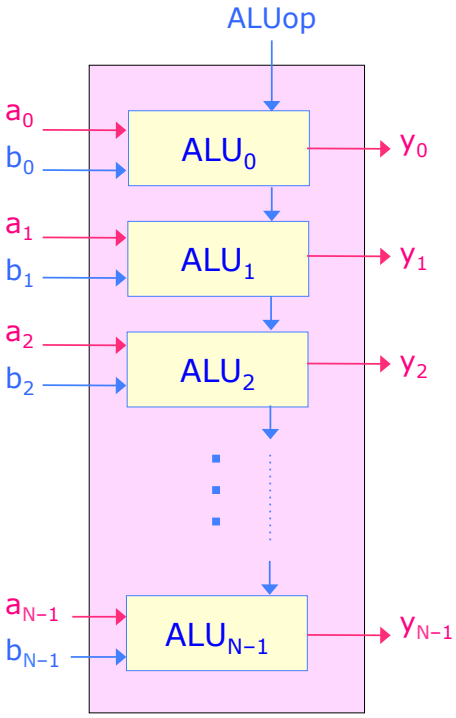
Struttura MODULARE di una ALU

ALU a N bit ➔ N ALU a 1 bit (ALU elementari)

Struttura ALU elementare:

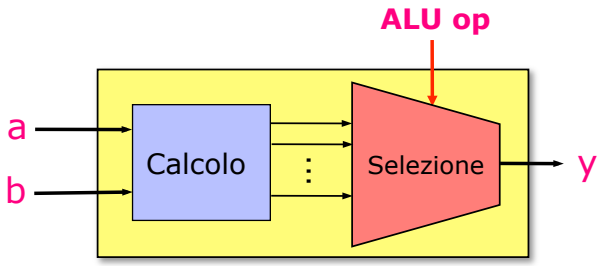
- Ingressi: Operandi: a_k, b_k
  Riporto in ingresso: r_{in}
  Selettore operazione: ALUop
- Uscite: Risultato: y_k
  Riporto in uscita: r_{out}

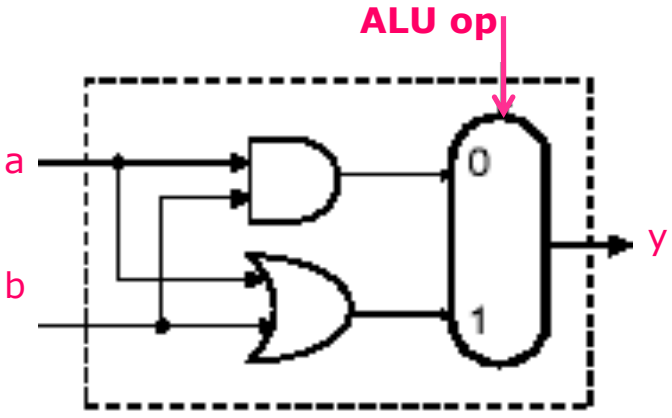




Progettazione della ALU

- Porta AND / OR
  - Selezionabile
- Componenti:
  - 1 porta AND
  - 1 porta OR
  - 1 Multiplexer (MUX)

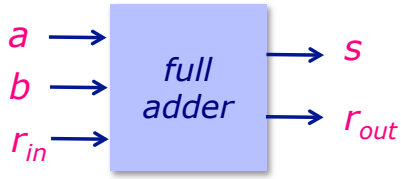




ALUop = 0 → y = AND(a,b)
ALUop = 1 → y = OR(a,b)

FULL Adder (1 bit)

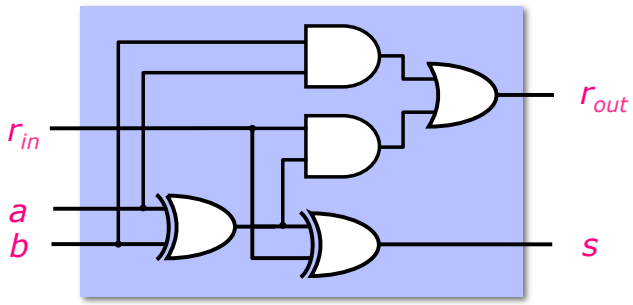
- Gestisce anche il riporto in ingresso
  - 3 ingressi: **a, b, r_{IN}**
  - 2 uscite: **s, r_{OUT}**



a	b	r_{in}	r_{out}	s
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

$$s = \bar{a}\bar{b}r_{in} + a\bar{b}\bar{r}_{in} + \bar{a}b\bar{r}_{in} + abr_{in} = a \oplus b \oplus r_{in}$$

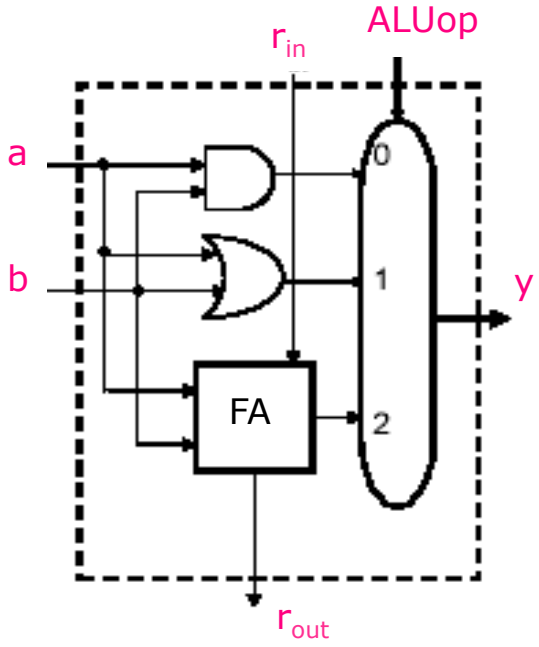
$$r_{out} = ab\bar{r}_{in} + \bar{a}br_{in} + a\bar{b}r_{in} + abr_{in} = ab + (a+b)r_{in}$$



ALU 1 bit

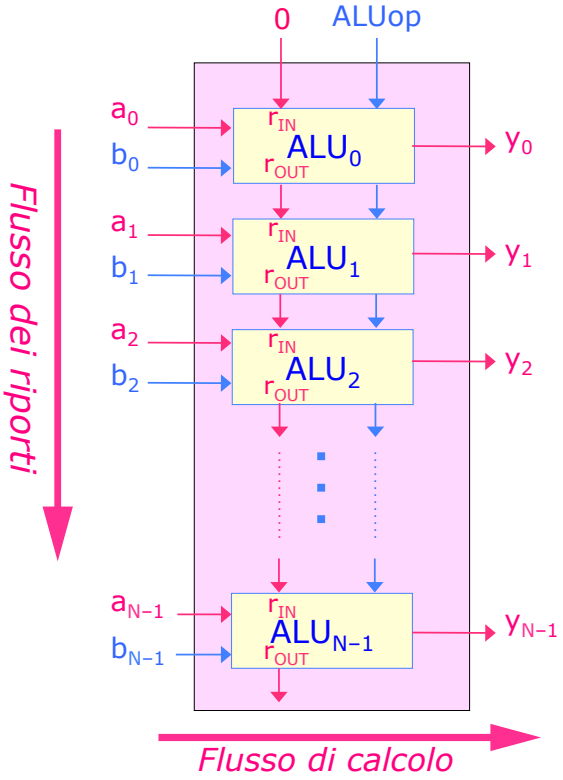
Operazioni:

- OR, AND, somma
- ALUop: 2 bit
  - **00: s = a and b**
  - **01: s = a or b**
  - **10: s = a + b + r_{in}**



ALU a 32 bit

- Come collegare N ALU a 1 bit per ottenere ALU a N bit?
- ALU a N bit:
- ALU in parallelo, ma...
  - Propagazione dei riporti
  - Limite alla velocità di calcolo



Sottrazione

Sottrazione → addizione dell'opposto: **a – b = a + (– b)**

- Posso farlo con gli stessi circuiti dell'addizione, ma devo costruire **–b** a partire da **b**

Complemento a 2: –b = not(b) + 1

- Inversione logica bit-a-bit
- Aggiunta della costante "1": pongo $r_{in}(0) = 1$

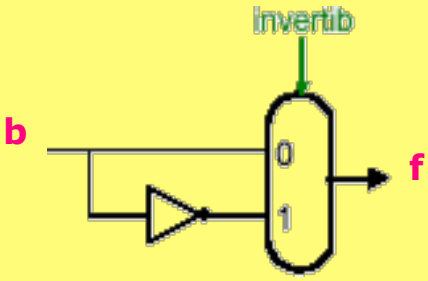
Inversione logica bit-a-bit

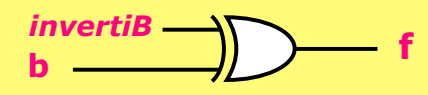
invertiB	b	f
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

→ **f = b *XOR* invertiB** →

Invertitore:

If (invertib=1) → **f = ~b**
else → **f = b**





ALU - bit i-esimo

ALU elementare: ALU_i (bit i-esimo)

Operazioni: **AND , OR , + , –**

Propagazione riporti: $r_{in}(i) = r_{out}(i-1)$, i= 0, 1, 2, ... , N–1

Addizione:

$r_{in}(0) = 0$

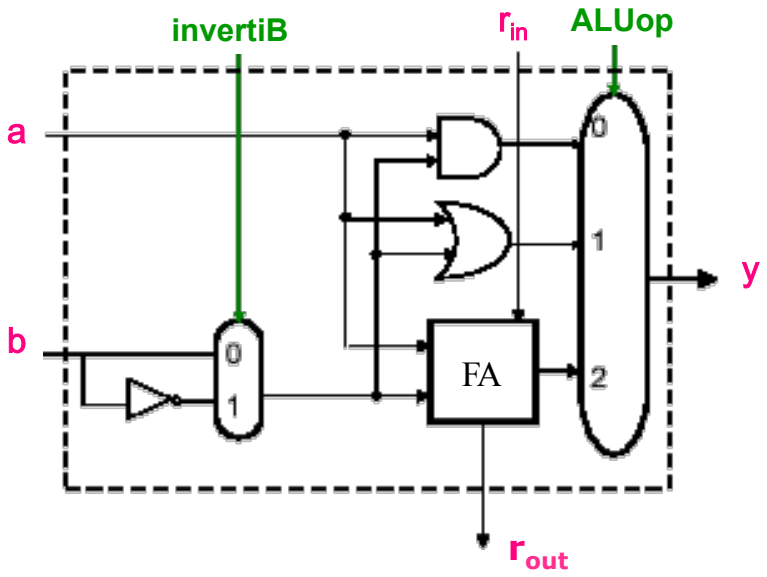
invertiB = 0

Sottrazione:

$r_{in}(0) = 1$

invertiB = 1

ALUop	funzione
0 00	and
0 01	or
0 10	+
1 10	–



Comparazione (confronto)

Comparazione:

if **a** < **b** then **y** = 1 (y = 0...01)

- Fondamentale per dirigere il flusso di esecuzione (test, cicli...)

if (a<b) → **y = [0 0 0 ... 0 1]**

else → **y = [0 0 0 ... 0 0]**

- Implementazione:

if ALUop = "comparazione" then

y(1) = y(2) = ... = y(N–1) = 0

if (a < b) **y(0) = 1**

else **y(0) = 0**

Devo:

- Imporre tutti i bit di y (tranne y_0) a 0;
- Calcolare y_0 in base alla condizione a<b

Come sviluppare la comparazione?

IDEA: in complemento a 2, il MSB della somma (bit di segno) **= 1**
per numeri negativi → $s_{MSB} = 1$

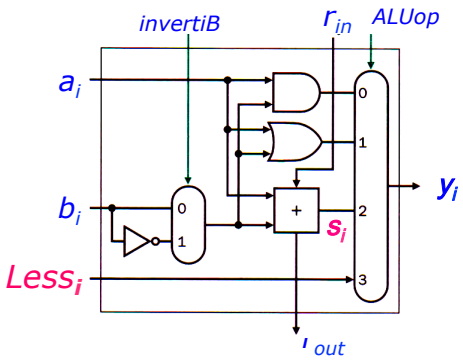
$$a < b \rightarrow a - b < 0 \rightarrow s_{N-1} = 1$$

Implementazione:

- Nuovo ingresso: **LESS**
  IF: ALUop = "comparazione" → $s_i = LESS_i$
- Operazioni:
  - Calcolare la differenza (a – b) (senza mandarla in uscita)
  - Inviare **0** a **LESS di ALU_1, ALU_2, …, ALU_{N-1}**
  - Inviare l'uscita del sommatore di ALU_{N-1} a **LESS di ALU_0**

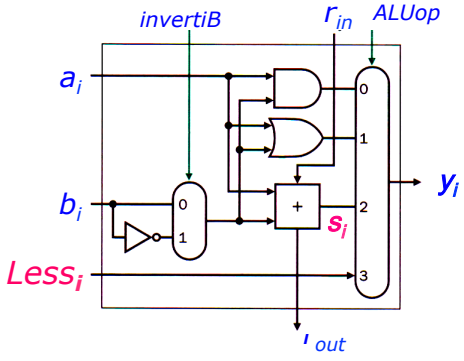
$LESS_1 \ldots LESS_{N-1} \leftarrow 0$

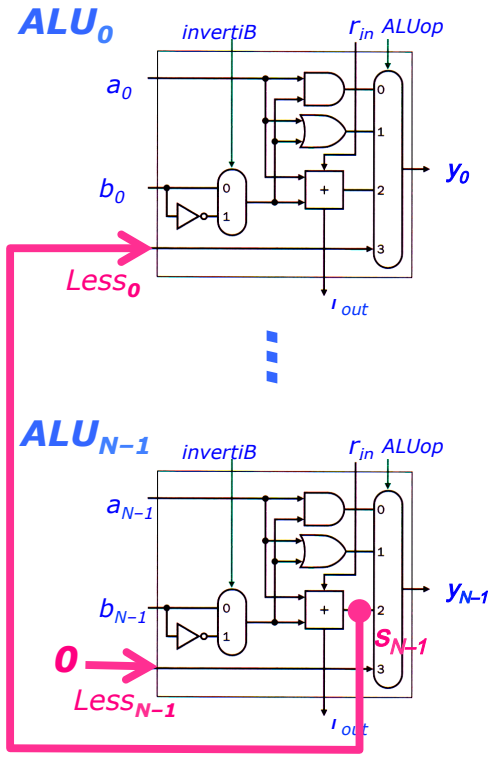
$LESS_0 \leftarrow s_{N-1}$



Struttura ALU con comparatore

if (**a < b**) & (**S = comparazione**)
ALUop = "11"
invertiB = "1"
Less(i) ← 0 i = 1, 2, 3, …. N–1
Less(0) ← s_{31}





Overflow

- Esempio decimale:
  - **a + b = c** dove **a,b,c tutti** codificati con 2 cifre decimali
  - **a = 19, b = 83**
  - **Overflow: 19 + 83 = (1)02**
- Supponendo il MSB dedicato al bit di segno...
  - **0**19 + **0**83 = **1**02
  - L'overflow modifica il **MSB** (in compl. a 2, dedicato al **segno**)
- **Overflow nella somma** quando:

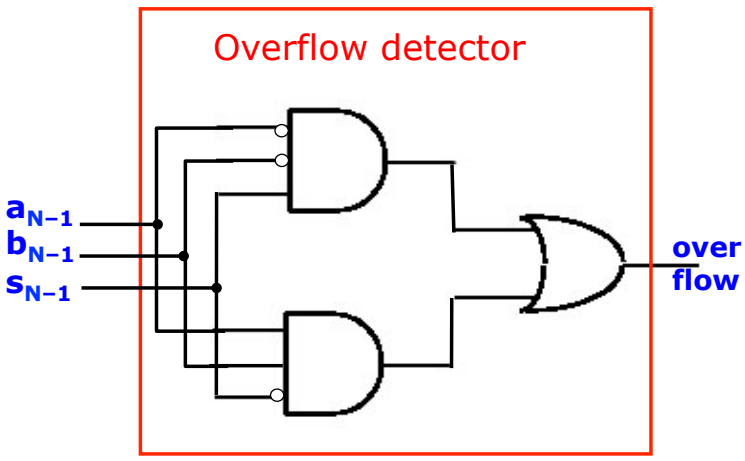
  a + b = s, a > 0, b > 0 → **MSB di a e b = 0, MSB di s = 1**
  a + b = s, a < 0, b < 0 → **MSB di a e b = 1, MSB di s = 0**

- Si può avere overflow con **a** e **b** di segno opposto ?

Circuito di riconoscimento dell'overflow

- 3 ingressi, tutti dalla ALU_{N-1}:
  - MSB di a, b e somma: a_{N-1} b_{N-1} s_{N-1}

a_{N-1}	b_{N-1}	s_{N-1}	overflow
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



Overflow

Overflow nella differenza:

019 – (–**0**83) = **1**02

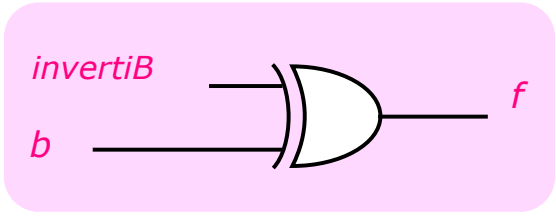
- **Overflow nella differenza** quando:

a + (–b) = s,	a > 0, (–b) > 0 →	**MSB di a e (–b) = 0, MSB di s = 1**
a + (–b) = s,	a < 0, (–b) < 0 →	**MSB di a e (–b) = 1, MSB di s = 0**

\+ : MSB di **b**

– : MSB di (**–b**) = MSB di **NOT(b)**

➔ +/– : MSB di **f**

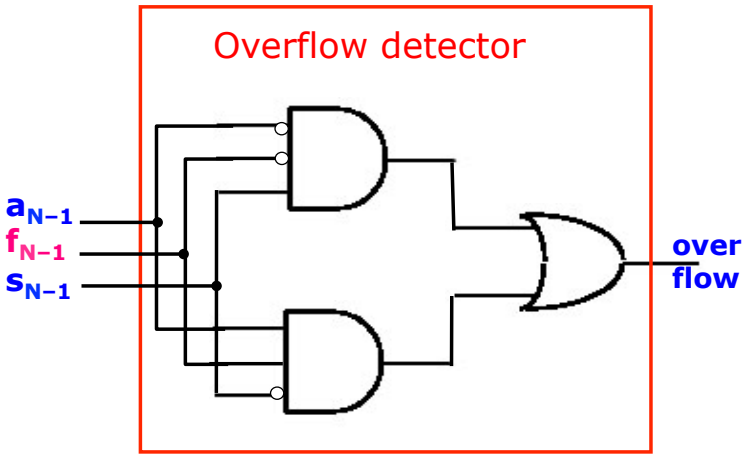


- Utilizzando **f**, a valle dello XOR, anziché **b**, il rivelatore di overflow funziona sia per la somma che per la differenza!

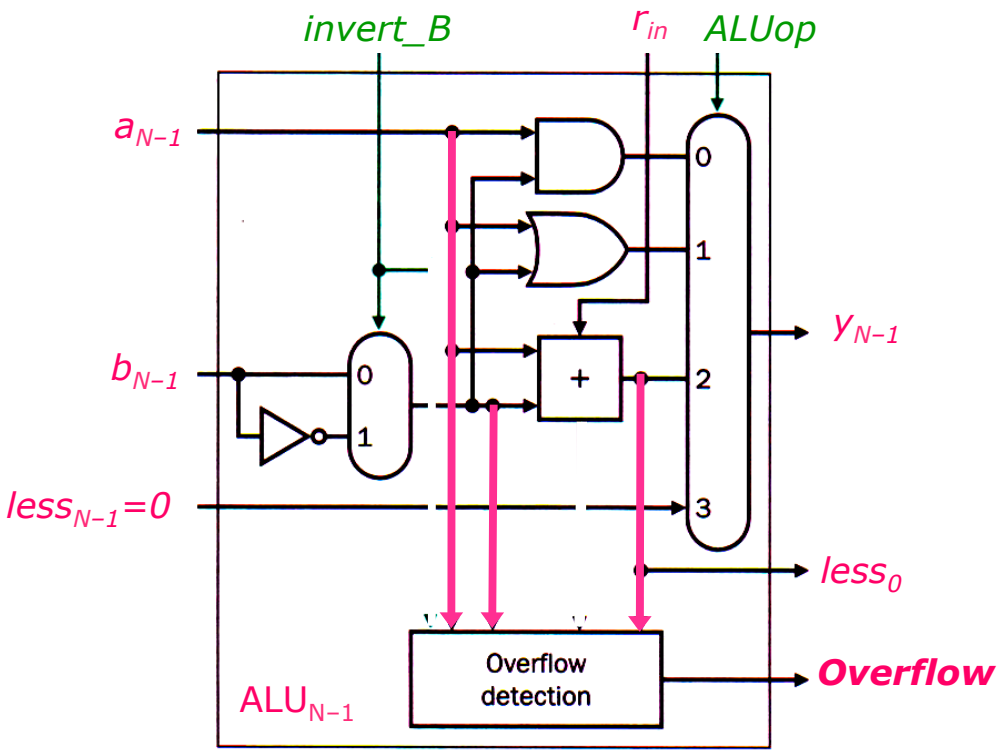
Circuito di riconoscimento dell'overflow

- 3 ingressi, tutti dalla ALU31:
  - MSB di a, b e somma: a_{31} b_{31} s_{31}

a_{N-1}	f_{N-1}	s_{N-1}	overflow
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



Struttura ALU_{N-1} con Overflow detector



ALU completa a 32 bit

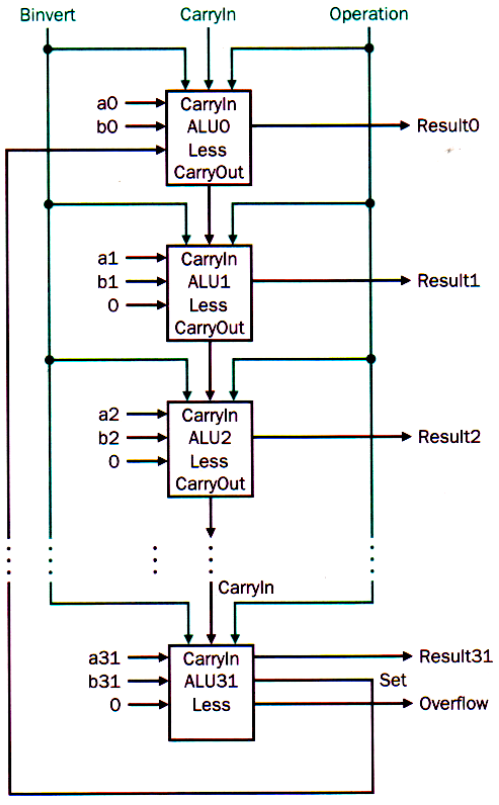
- *InvertiB e $r_{IN}(0)$ sono lo stesso segnale*

Test di uguaglianza:

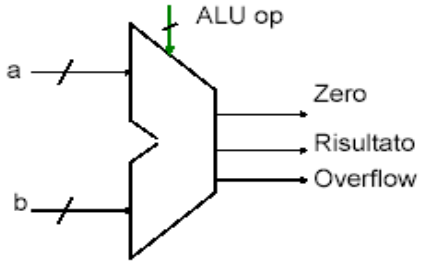
- *Operazioni necessarie*
  - Impostare una differenza.
  - Effettuare l' OR di tutti i bit somma.
  - Uscita dell' OR = 0 → i due numeri sono uguali

Operazioni possibili:

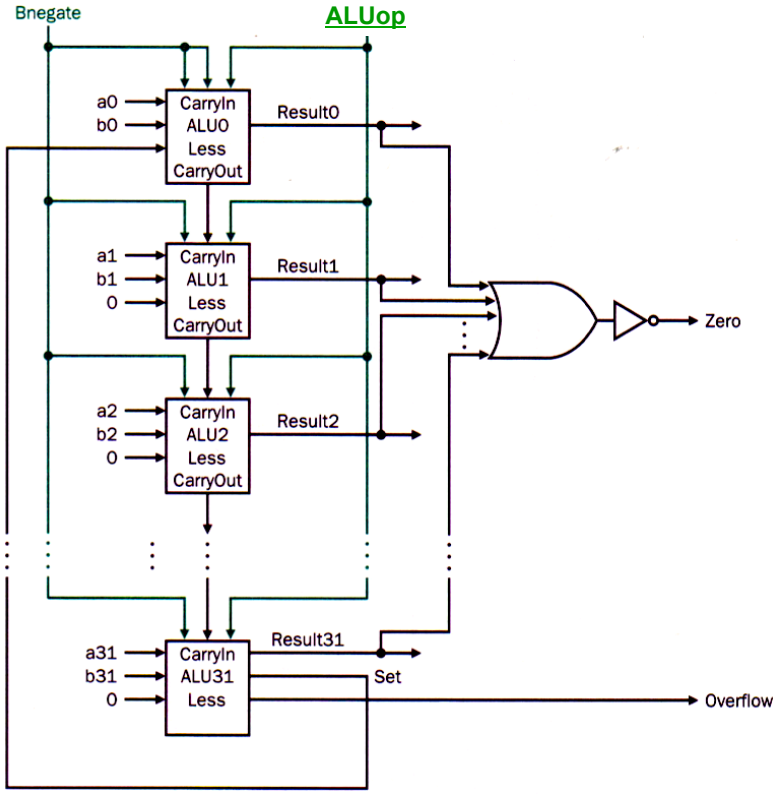
- AND
- OR
- Somma / Sottrazione
- Comparazione
- Test di uguaglianza



ALU a 32 bit: struttura finale



ALUop	funzione
000	and
001	or
010	+ (add)
110	– (sub)
111	set less than



CIRCUITI CON MEMORIA notevoli:

- Bistabili
- Circuiti notevoli
- Registri
- Contatori

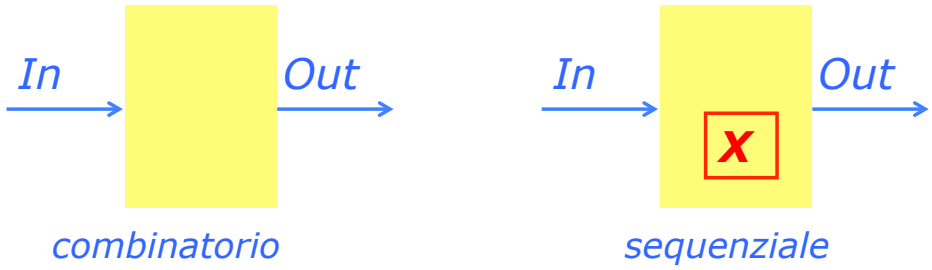
Circuiti sequenziali

- **Circuiti combinatori** = circuiti senza memoria
  - Gli output al tempo t dipendono unicamente dagli input al tempo t

    $Out = f(In)$
  - Per consentire ad un dispositivo di mantenere le informazioni, sono necessari circuiti con memoria

- **Circuiti sequenziali** = circuiti con memoria (stato)
  - La memoria contiene lo **stato (X)** del sistema:

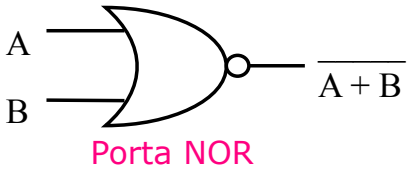
    $Out = f(In, X)$



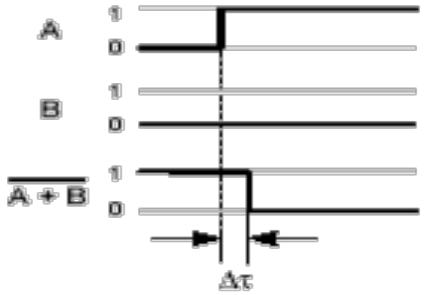
Latch & Bistabili

- Elemento cardine dei circuiti sequenziali è lo **stato**
  - Lo stato riassume il funzionamento negli istanti precedenti e deve essere immagazzinato (memorizzato)
  - Necessità della memoria (bistabili → registri → memorie)
- Elemento base della **memoria** è il **bistabile**
  - dispositivo in grado di **mantenere *indefinitamente* il valore di input**
  - Il suo valore di uscita coincide con lo stato
  - Memoria: insieme di bistabili (bistabili → registri → memorie)
- **Tipologie di bistabile**
  - Bistabili non temporizzati (asincroni) / temporizzati (sincroni).
  - Bistabili (sincroni) che commutano sul livello (**latch**) o sul fronte (**flip-flop**)

Latch Set-Clear (SR) asincrono

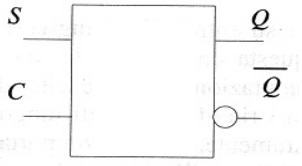


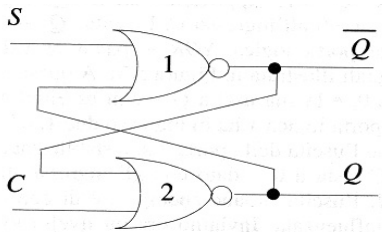
Porta NOR



- Una coppia di porte NOR retroazionate può memorizzare un bit!

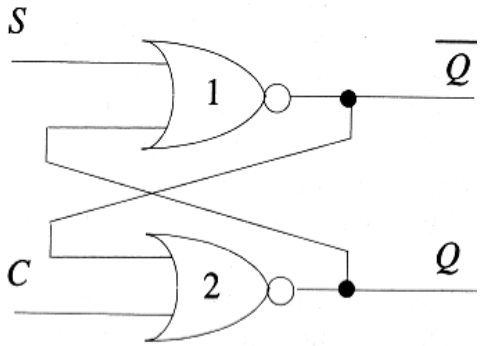
Latch Set-Clear

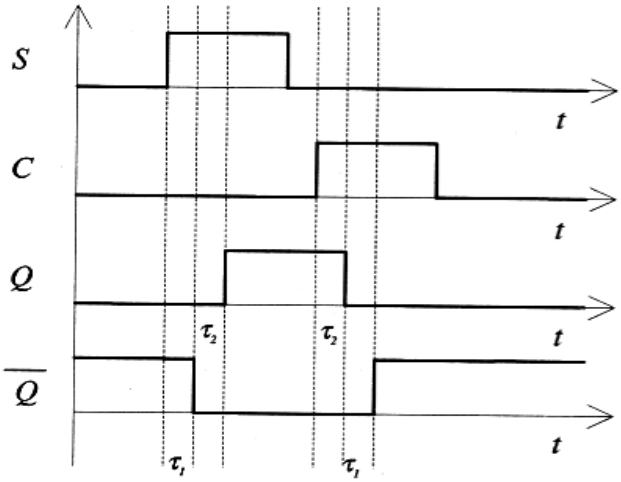




Funzionamento del Latch SC (SR)

Latch SC





- Funzionamento:
  - Set: C = 0, S → 1 Q → 1 (~Q → 0)
  - Reset: S = 0, C → 1 Q → 0 (~Q → 1)
  - Comportamenti anomali:
  - S = 1 , C: 0→1 Q=~Q=0 (anomalia)

Tabella delle transizioni

- Un circuito sequenziale non si può rappresentare mediante una tabella di verità
- Per i circuiti sequenziali si definisce la tabella delle transizioni

Tabella delle transizioni: $Q^* = f(Q, I) = f(Q, S, C)$

Q: valore dell'uscita attuale: stato **corrente**

Q*: uscita al tempo successivo: stato **prossimo**

$Q^* = f(Q, I)$

Q \ I	SC = 00	SC = 01	SC = 11	SC = 10
0	0	0	X	1
1	1	0	X	1

Q* = Q Clear / Reset SET

Tabella delle transizioni

- Tabella delle transizioni ***f*** :

$$Q^* = f(I, Q)$$

SC / Q	SC=00	SC=01	SC=11	SC=10
Q=0	0	0	X	1
Q=1	1	0	X	1

S	C	Q	Q*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	X
1	1	1	X

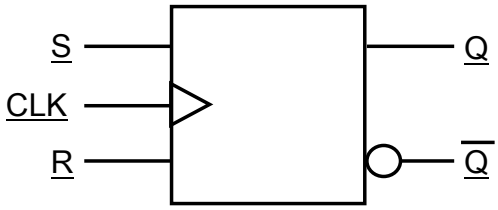
- Considerando lo stato Q come ingresso ottengo la tabella delle verità di Q*:

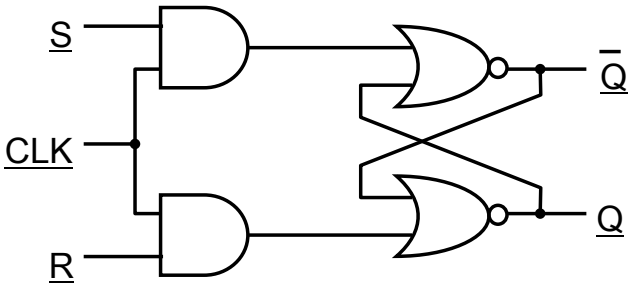
Latch SR sincrono

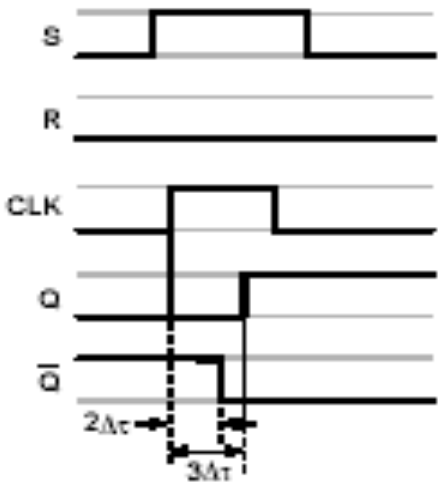
Latch Set-Reset (SR) sincrono

Struttura: Latch SR + Porte AND tra il clock e gli ingressi.

- Solo quando il clock è alto i "cancelli" (porte AND) fanno passare gli input → LATCH



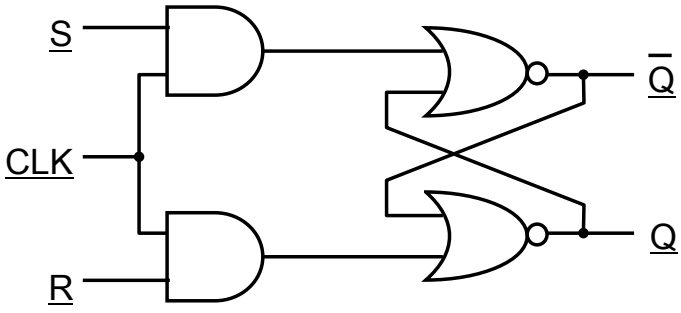




Analisi della funzione logica sintetizzata

Latch Set-Reset (SR) sincrono

$$Q^* = \overline{T}Q + T\left(Q\overline{R} + \overline{Q}S\right)$$



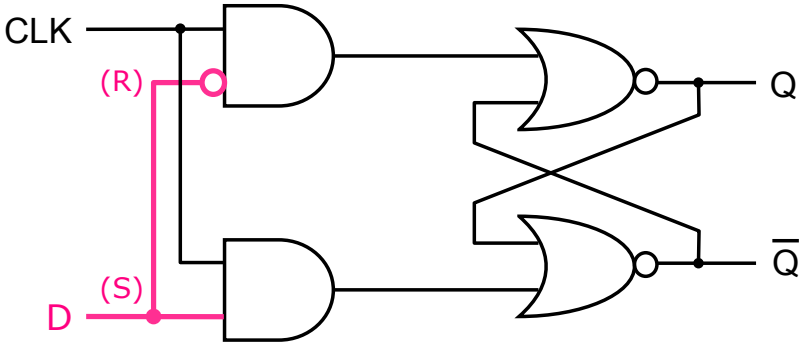
$$T = 1 \rightarrow \quad Q^* = Q\overline{R} + \overline{Q}S: \quad \begin{cases} \text{Set}: S = 1, R = 0 & Q^* = Q + \overline{Q} = 1 \\ \text{Reset}: S = 0, R = 1 & Q^* = 0 + 0 = 0 \end{cases}$$

$$T = 0 \rightarrow \quad Q^* = Q: \quad \text{"}status\ quo\text{"}$$

Comportamento del Latch D

Latch SR è un po' scomodo: 2 ingressi separati per memorizzare "0" o "1"

IDEA: piloto **S e R** con un solo ingresso: **D**
D = 1 → S=1, R=0 (Set) → **Q=1**
D = 0 → S=0, R=1 (Reset) → **Q=0**



LATCH D:
Clock ALTO (CLK=1):

- L'uscita Q insegue l'ingresso D (con 3Δτ di ritardo)

Clock BASSO (CLK=0):

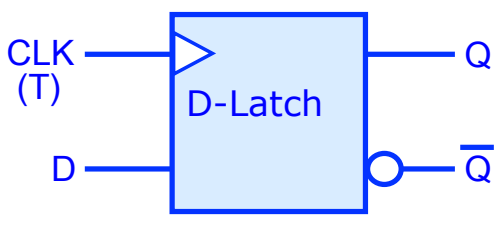
- L'uscita Q rimane bloccata sull'ultimo valore assunto

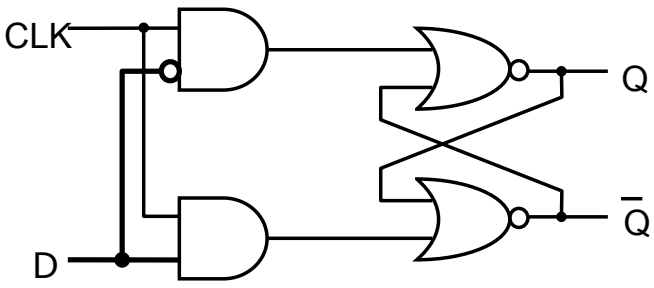
Latch D sincrono

LATCH D sincrono:

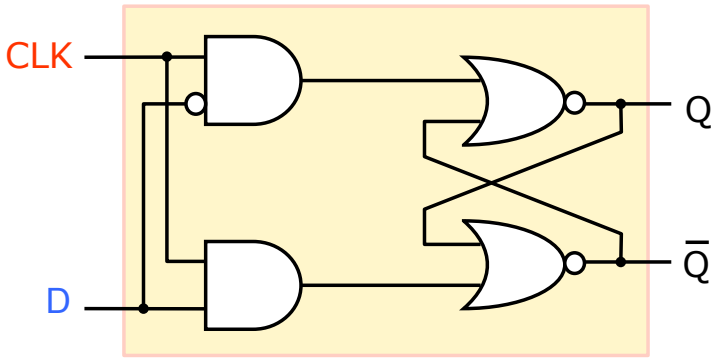
Memorizza il valore presente all'ingresso dati quando il clock è alto, altrimenti (clock=0) si mantiene sul valore memorizzato

if CLK = 1 **then** Q*=D **else** Q*=Q





La struttura del latch D



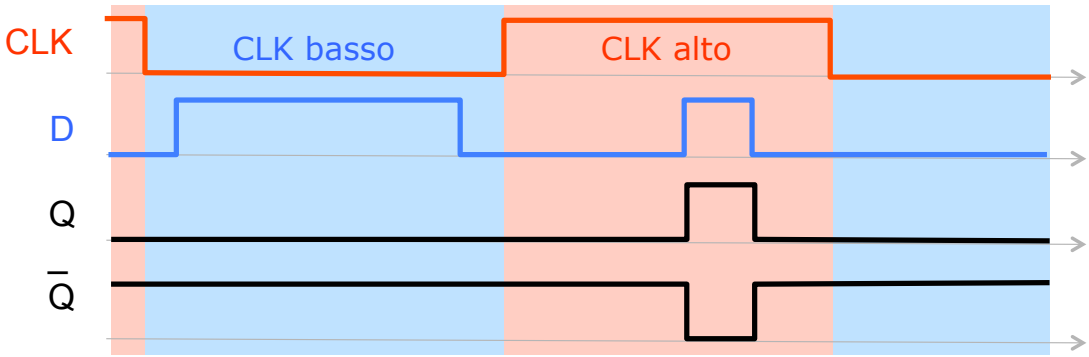


Tabella delle transizioni

- Dalla tabella delle transizioni calcolo l'espressione logica della funzione stato prossimo **Q* = f(T,Q,D):**

Tabella delle transizioni → Funzione stato prossimo Q* = f(Q, T,D):

T	Q	D	Q*
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

$$Q^* = \overline{T}Q\overline{D} + \overline{T}QD + T\overline{Q}D + TQD =$$

$$= \overline{T}Q + TD$$

Clock T alto:
→ Q* = D (input)

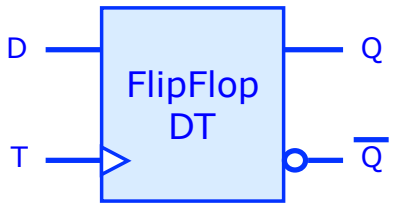
Clock T basso
→ Q* = Q (status quo)

$$T = 1 \rightarrow Q^* = D$$

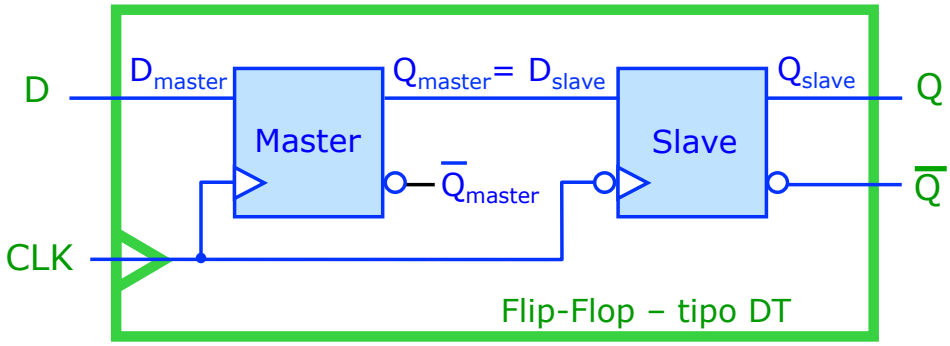
$$T = 0 \rightarrow Q^* = Q$$

Bistabili "edge sensitive": i Flip-Flop

FLIP-FLOP: bistabile *edge sensitive* (attivo sui fronti del clock):
lo stato (uscita) commuta solo in corrispondenza del fronte di salita o di discesa del clock.

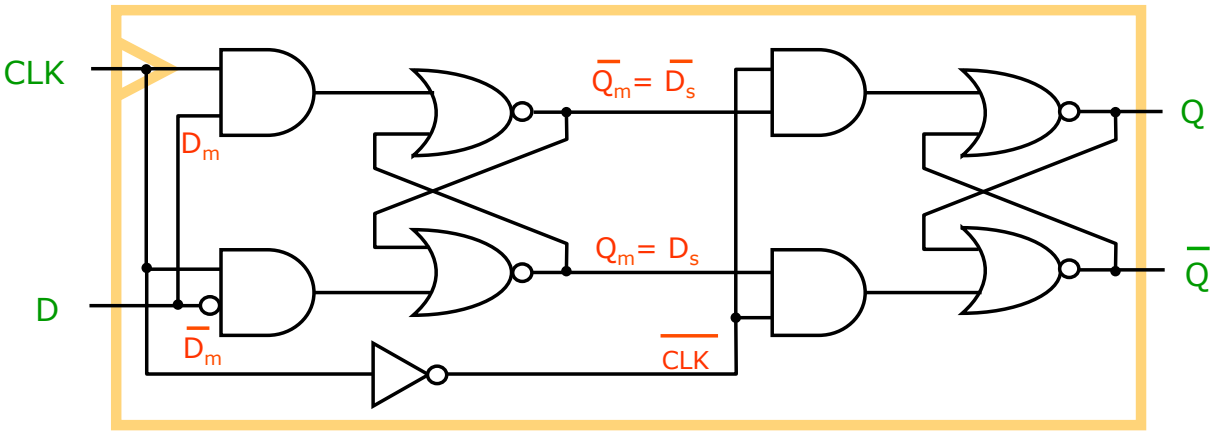


Flip-Flop tipo DT
configurazione "Master-Slave":

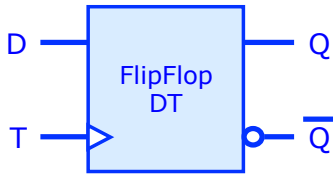


Flip-flop: struttura master–slave

- **FLIP (clock ALTO)**: L'ingresso D viene memorizzato nel latch MASTER
  - L'ingresso è aperto, ma l'uscita è bloccata
- **FLOP (clock BASSO)**: l'uscita stabile del latch MASTER viene propagato al latch SLAVE
  - L'ingresso è bloccato, l'uscita è stabile.

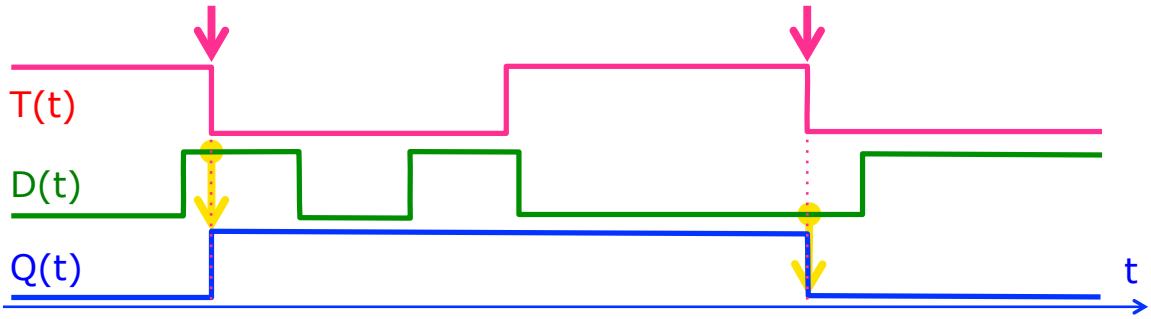


Funzionamento dei Flip-Flop



- Fronte di SALITA – FLIP
  - Attivato lo stadio **MASTER**
  - Uscita (stadio slave) invariata
- **Fronte di DISCESA – FLOP**
  - Attivato stadio **SLAVE**
  - Memorizzato il dato sull'ingresso:
  - Presenta il dato memorizzato in uscita:
  - Ingresso isolato

D → STATO
STATO → Q

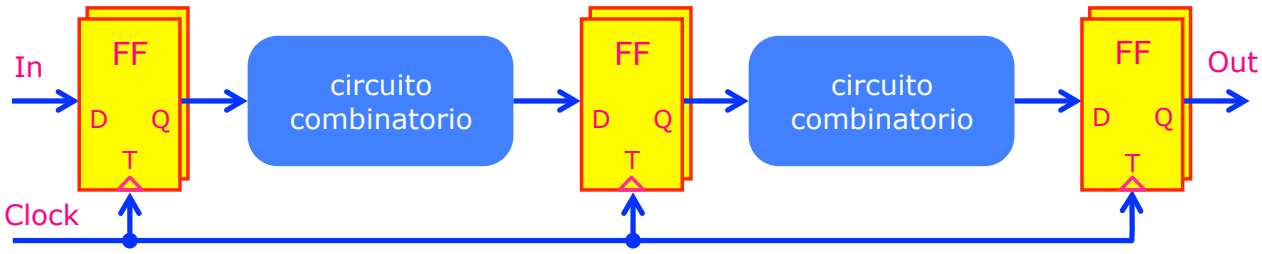


Architetture SINCRONE

Una tipica applicazione: **architetture sincrone**

- Batterie di flip-flop separano stadi successivi di circuiteria combinatoria
  - Senza separazioni, la propagazione disordinata dei segnali attraverso le porte logiche genererebbe "**data races**" che renderebbero il comportamento del circuito **imprevedibile**.
  - Ad ogni colpo di clock i segnali avanzano di uno stadio.

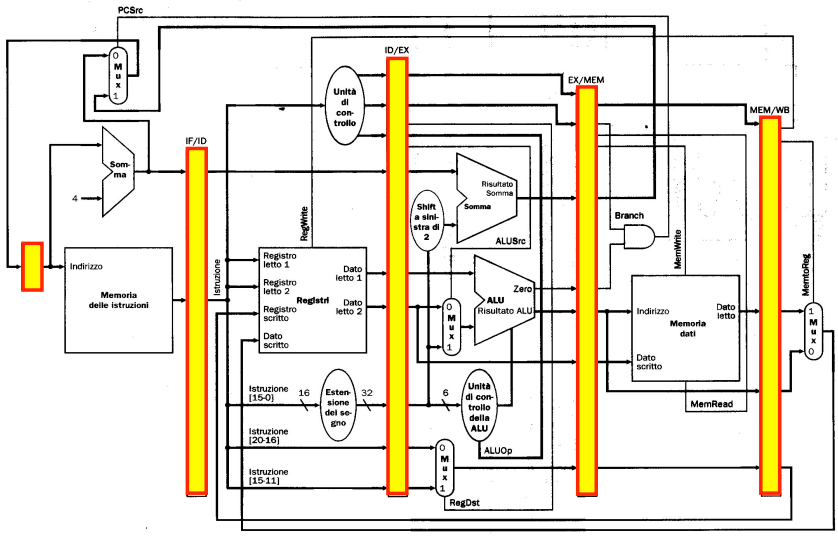
Flip-flop: cancello doppio
Quando apro in ingresso, l'uscita è chiusa, e viceversa.



Sincronizzazione mediante flip-flop

- I "cancelli" devono **disaccoppiare** i diversi sottosistemi logici
  - "memorizzare" i segnali e "rilanciarli", tutto ad un determinato istante
  - **prima e dopo**, uscite **non sensibili** agli ingressi
  - **Cancello doppio**: **ingresso** e **uscita,** mai aperti contemporaneamente

Cancelli:
registri costituiti da gruppi di **flip-flop**



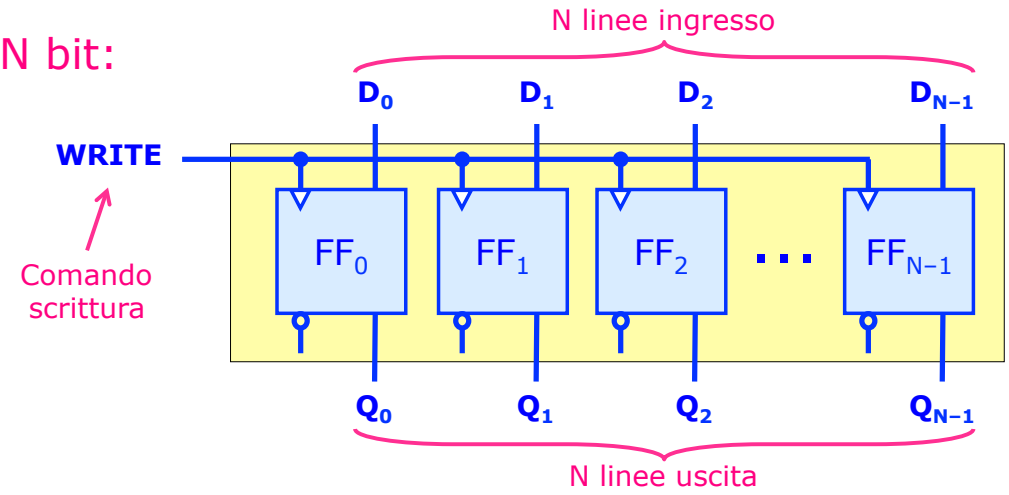
Registri

Registro: unità di memorizzazione di parole di **N bit**

Struttura: **N Flip-flop tipo DT**

Operazioni: **LETTURA**: I dati memorizzati sono **sempre leggibili** sulle uscite Q.
SCRITTURA: un **fronte** di salita (o discesa) su **WRITE** memorizza i valori presenti sugli ingressi D

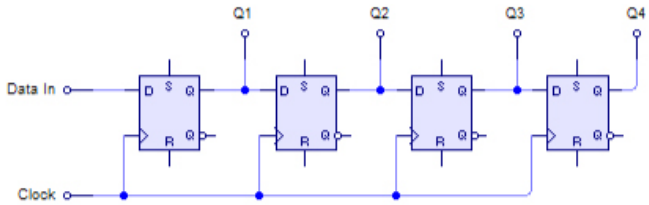
Registro N bit:



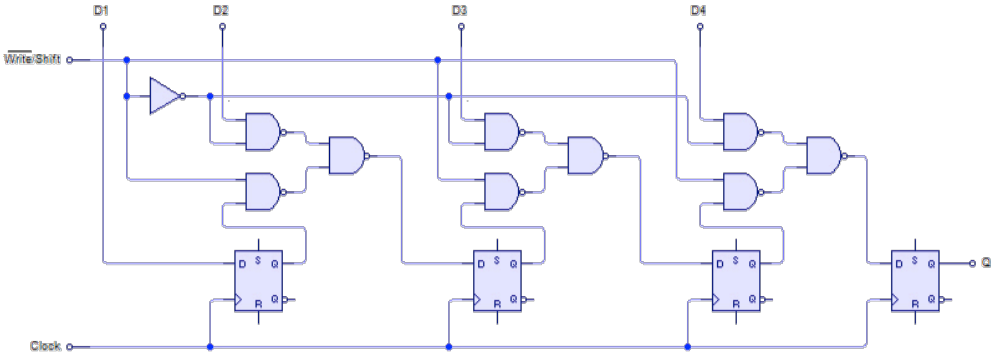
Shift registers

- Caratterizzati da ingresso e/o uscita seriali, anziché paralleli.
- **WRITE** è sostituito da: **SHIFT**

Serial in, serial/parallel out (SISO, SIPO):



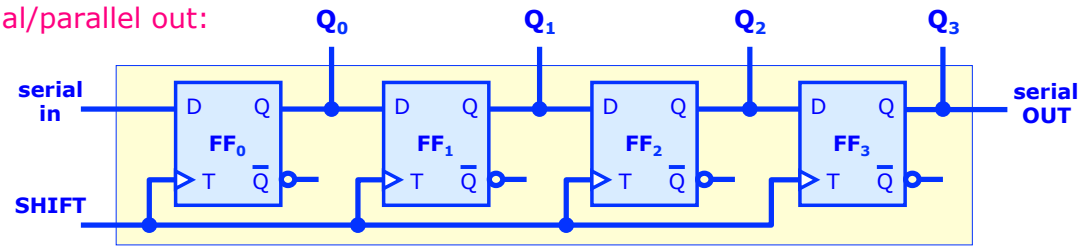
Parallel in, serial out (PISO):



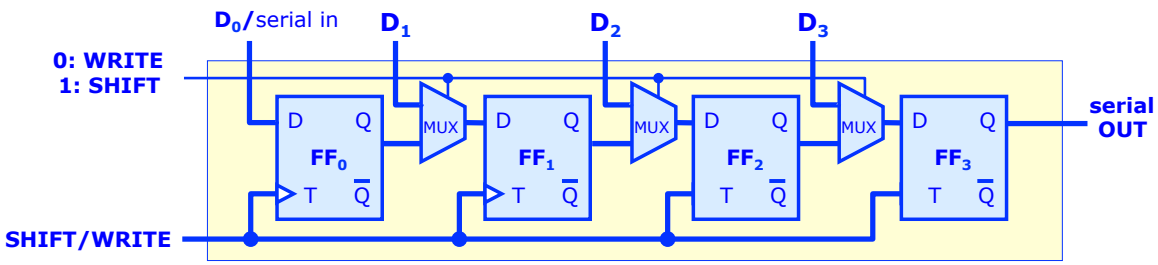
Shift registers

- Caratterizzati da ingresso e/o uscita seriali, anziché paralleli.
- **WRITE** è sostituito da: **SHIFT**

Serial in, serial/parallel out:



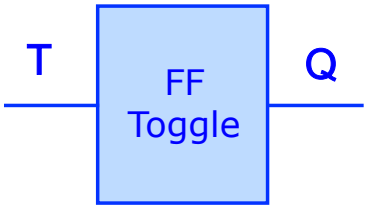
Serial/parallel in, serial out:



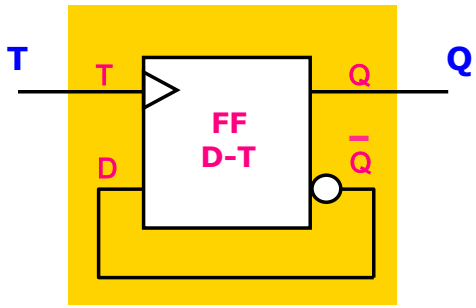
Flip-Flop di tipo TOGGLE

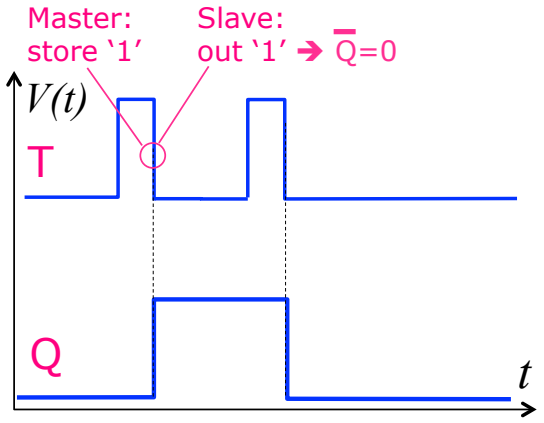
Flip-flop di tipo **T (TOGGLE)**

- Realizzato mediante un **flip-flop tipo D retroazionato**
- Ogni **impulso** in ingresso **commuta lo stato** del bistabile



FF Toggle



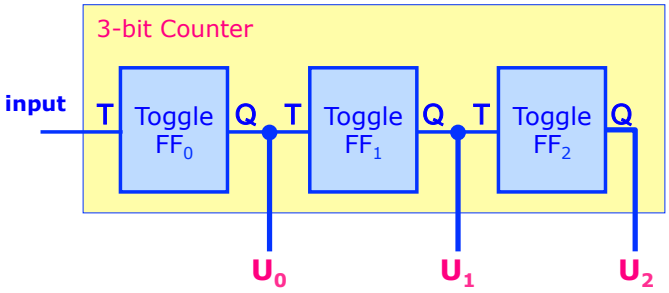


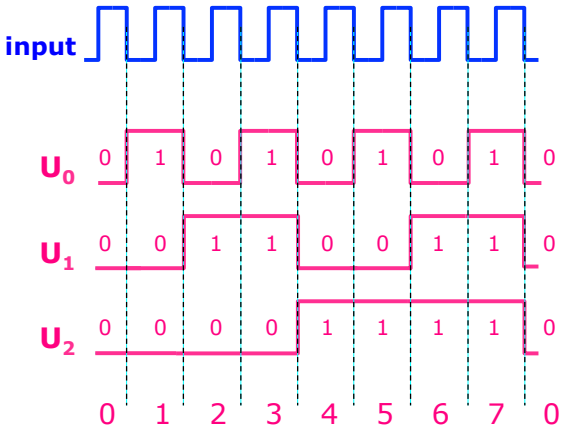
Contatori

Binary counter

- contano gli impulsi in ingresso:
  ogni impulso incrementa il n. binario (di N bit) espresso sulle N uscite

Struttura: N Flip-Flop Toggle in cascata





Sintesi di circuiti sequenziali: mediante macchine a stati finiti (FSM)

- Una Macchina a Stati Finiti (MSF) è definita dalla quintupla:

$$< X,\ I,\ Y,\ f(\cdot),\ g(\cdot) >$$

X: insieme degli stati (in numero finito).

I: alfabeto di ingresso: l'insieme dei simboli che si possono presentare in ingresso. Con **n** ingressi, avremo $\mathbf{2^n}$ possibili configurazioni.

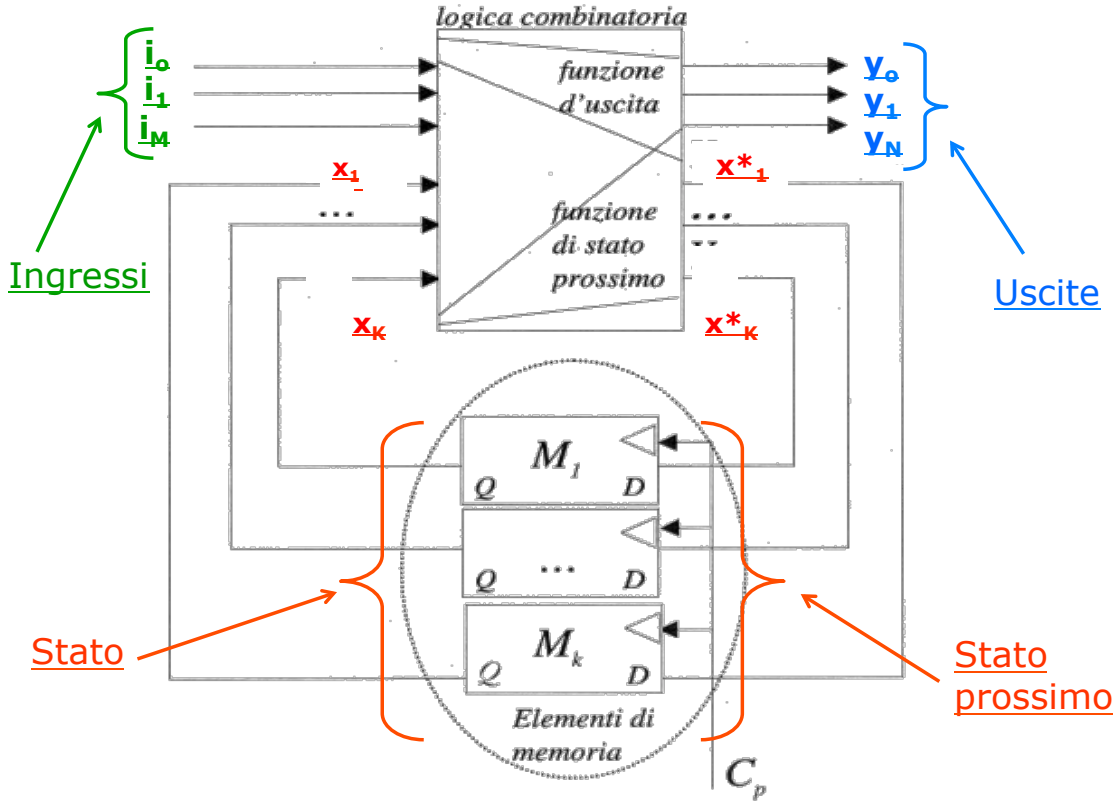
Y: alfabeto di uscita: l'insieme dei simboli che si possono generare in uscita. Con **m** uscite, avremo $\mathbf{2^m}$ possibili configurazioni

***f*(·)**: funzione stato prossimo: $\mathbf{X^* = f(X, I)}$
Definisce l'evoluzione della macchina nel tempo, in modo deterministico

***g*(·)**: funzione di uscita: $\mathbf{Y = g(X)}$ (macchina di **Moore**)
$\mathbf{Y = g(X, I)}$ (macchina di **Mealy**)

- Per il buon funzionamento della macchina è previsto uno **stato iniziale**, al quale la macchina può essere portata mediante un comando di **reset**.

La macchina sequenziale di **Huffman**



Macchina di Moore: **State Transition Table (STT)**

- **STT**: State Transition Table (Tabella delle transizioni di stato)
  - Per ogni coppia: <stato attuale, ingresso>
    definisco uscita **y** e stato prossimo **x***

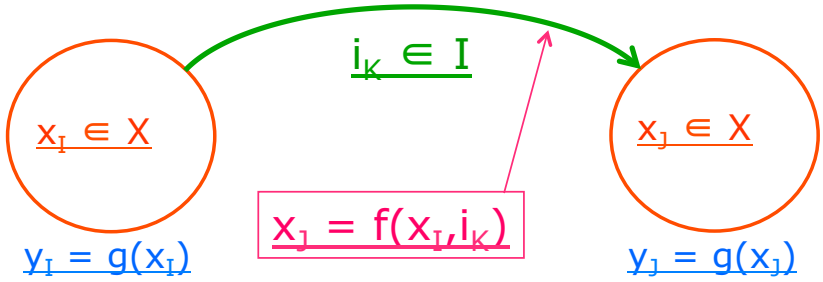
    $(x_i \in X , i_j \in I) \rightarrow y(x_i) ; x^*(x_i, i_j)$

  - Esempio: M stati (log2M bit di stato), N ingressi (log2M bit d'ingresso):

	i_1	i_2	...	i_N	Y
x_1	$x^*(x_1,i_1)$	$x^*(x_1,i_2)$		$x^*(x_1,i_N)$	$y(x_1)$
x_2	$x^*(x_2,i_1)$	$x^*(x_2,i_2)$		$x^*(x_2,i_N)$	$y(x_2)$
...					...
x_M	$x^*(x_M,i_1)$	$x^*(x_M,i_2)$		$x^*(x_M,i_N)$	$y(x_M)$

Macchina di Moore: **State Transition Graph (STG)**

- **STG: State Transition Graph**
  (*Diagramma degli Stati o Grafo delle transizioni*)
  - Ad ogni nodo è associato uno stato: $x_i \in X$
  - ... ed un valore della funzione d'uscita: $y_i \in Y$, $y_i = g(x_i)$
  - Un arco orientato da uno stato x_i ad uno stato x_j, contrassegnato da un simbolo (di ingresso) i_K, rappresenta una transizione che si verifica quando la macchina, essendo nello stato x_i, riceve come ingresso i_K

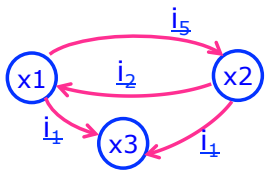


Sintesi circuito sequenziale

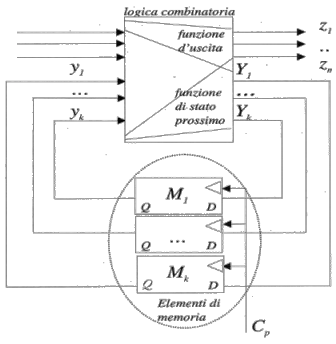
Sequenza operazioni di sintesi
di un circuito sequenziale mediante modello a FSM:

1. Definizione insiemi ingressi **I** e uscite **Y**
2. Costruzione grafo delle transizioni **STG**
   ➔ definizione insieme X, f(X,I) e g(X)
3. Traduzione **STG ➔ STT**
4. **Codifica binaria** degli elementi di **X, Y, I** nella **STT**
   ➔ f(X,I) e g(X) diventano funzioni logiche
5. Sintesi delle funzioni logiche f(X,I) e g(X)
   ➔ circuito finale con struttura di Huffman

$$I = \{i_1, i_2, \cdots, i_M\},$$
$$Y = \{y_1, y_2, \cdots, y_P\}$$

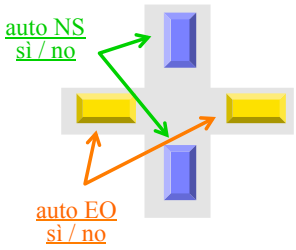


	i_1	...	i_N	Y
x_1	$x^*(x_1,i_1)$		$x^*(x_1,i_N)$	$y(x_1)$
x_2	$x^*(x_2,i_1)$		$x^*(x_2,i_N)$	$y(x_2)$
...				...
x_M	$x^*(x_M,i_1)$		$x^*(x_M,i_N)$	$y(x_M)$



$$x_i^* = f_i(x_1, \cdots, x_Q, i_1, \cdots, i_R)$$
$$y_i = g_i(x_1, \cdots, x_Q)$$

Esempio: controllore di un semaforo



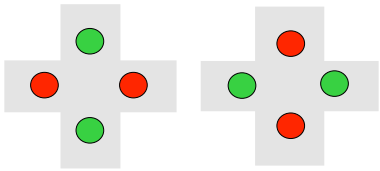
SEMAFORO:

- Incrocio tra 2 strade: nord-sud (NS) ed est-ovest (EO) controllate da un semaforo
  - per semplicità consideriamo solamente rosso e verde
- Il semaforo può commutare ogni 30 secondi
  - Macchina sincrona, clock con frequenza = ?
- E' presente un sensore in grado di "leggere", per ogni direttrice, se esiste almeno un'auto in attesa, oppure un'auto che si accinga ad attraversare (condizioni trattate allo stesso modo).
- Il semaforo deve cambiare colore, da rosso a verde, quando esiste un'auto in attesa sulla sua direttrice.
- Se ci sono auto in attesa sulle entrambe le direttrici il semaforo deve cambiare colore (al termine del tempo di commutazione)

INGRESSI

- Auto NS presente / non presente
  - **AutoNS=1/0**
- Auto EO presente / non presente
  - **AutoEO = 1/0**

→ 2 bit di INGRESSO → 4 configurazioni d'ingresso: ***I = {00, 01, 10, 11}***

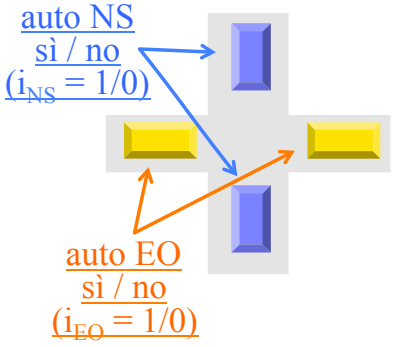


STATO

- Semaforo NS VERDE, semaforo EO ROSSO
- Semaforo NS ROSSO, semaforo EO VERDE

X = {VerdeNS, VerdeEO}

→ 1 bit di STATO (1 flip-flop):



USCITE: = STATO: ***Y = X***

- LuceEO verde (LuceNS rossa)
- LuceNS verde (LuceEO rossa)

→ 2 configurazioni d'uscita → 1 bit di USCITA

Funzionamento: **uscita**

- Funzione uscita: **Y = g(X)**
  - Per ogni stato, definire l'uscita della macchina
- Uscita ↔ STATO ➔ **Y = X**
  - VerdeNS → Verde sulla direttrice NS, rosso sulla direttrice EO
  - VerdeEO → Verde sulla direttrice EO, rosso sulla direttrice NS
- Luce Verde NS / Rosso EO = **VerdeNS**
- Luce Verde EO / Rosso NS = **VerdeEO**

Funzionamento: **stato prossimo**

- **Stato prossimo:** evoluzione dello stato, in funzione dello stato attuale e degli ingressi attuali

$$X(t+1) = X^* = f(X(t), I)$$

- **X(t+1) = X*** = "VerdeNS"
  - Se X(t)="VerdeNS" AND non ci sono auto sulla direttrice EO
  - Se X(t)="VerdeEO" AND ci sono auto sulla direttrice NS

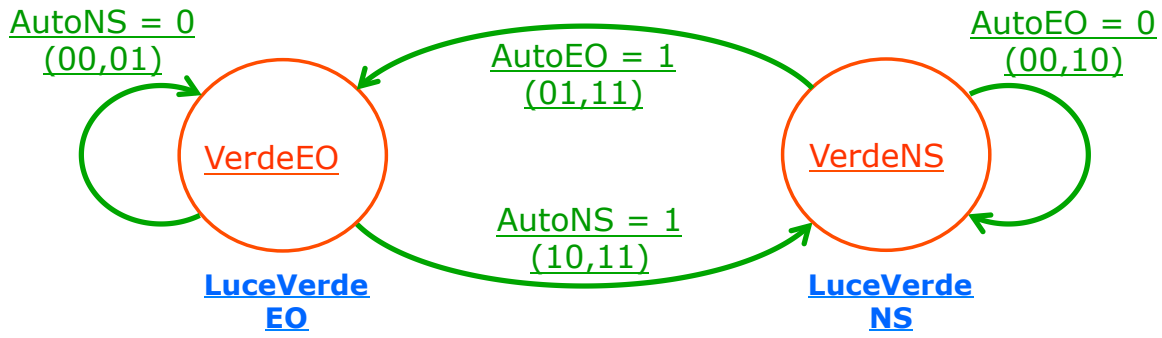
  VerdeNS · ~autoEO + VerdeEO · autoNS ➔ X* = VerdeNS

- **X(t+1) = X*** = "VerdeEO"
  - Se X(t)="VerdeEO" AND non ci sono auto sulla direttrice NS
  - Se X(t)="VerdeNS" AND ci sono auto sulla direttrice EO

  VerdeEO · ~autoNS + VerdeNS · autoEO ➔ X* = VerdeEO

STG del semaforo

Sintesi STG:



Risultato:

- Funzione stato prossimo:

  VerdeNS* = VerdeNS · ~autoEO + VerdeEO · autoNS
  VerdeEO* = VerdeEO · ~autoNS + VerdeNS · autoEO

- Funzione uscita: **Y = X**

STT del semaforo

Sintesi STT:

X \ $I_{(i_{EO},i_{NS})}$	$i_{EO}=0$ $i_{NS}=0$	$i_{EO}=0$ $i_{NS}=1$	$i_{EO}=1$ $i_{NS}=0$	$i_{EO}=1$ $i_{NS}=1$	Uscita
VerdeNS	VerdeNS	VerdeNS	VerdeEO	VerdeEO	Luce VerdeNS
VerdeEO	VerdeEO	VerdeNS	VerdeEO	VerdeNS	Luce VerdeEO

Funzione stato prossimo: X* = *f*(X, I)

Funzione uscita: Y = *g*(X)

STT del semaforo: CODIFICA binaria

Codifica binaria della STT:

Stato:
- (**VerdeNS**/RossoEO, RossoNS/**VerdeEO**) → **X = {0, 1}**

Ingresso:
- 2 Variabili: (AutoEO, AutoNS) → 1 =“presente”, 0 =“assente”
- 4 Configurazioni: **I = (i_{EO},i_{NS}) = {00, 01, 10, 11}**

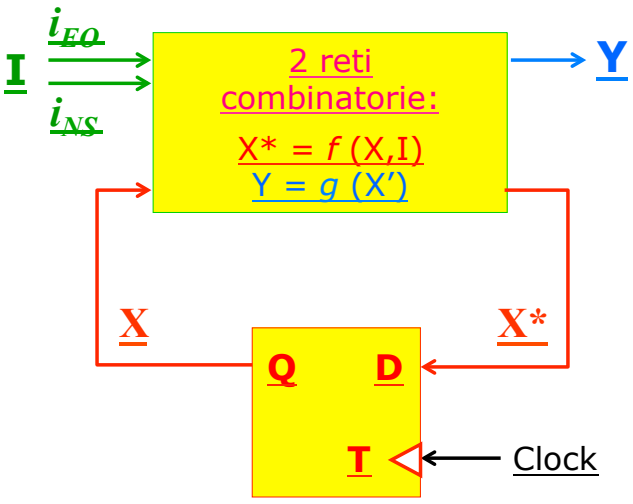
Uscita:
- (Luce_VerdeNS, Luce_VerdeEO) → **Y = {0, 1}**

X \ I(i_{EO},i_{NS})	00	01	10	11	Uscita Y
0	0	0	1	1	0
1	1	0	1	0	1

Sintesi della MSF del semaforo

Sintesi del circuito:
- 2 stati (0,1) → 1 flip-flop DT
- 2 linee di ingresso
- 1 linea d’uscita

Architettura di **Huffman:**
- Rete combinatoria che implementa:
  - stato prossimo: ***f*(X,I)**
  - uscita: ***g*(X)**



Sintesi della MSF del semaforo

- Mediante la STT codificata in binario, posso esprimere **X*** e **Y** come somma di prodotti:
  - cerco i mintermini:

$$X^* = \overline{X}\, i_{EO}\, \overline{i_{NS}} + \overline{X}\, i_{EO}\, i_{NS} + X\, i_{EO}\, \overline{i_{NS}} + X\, \overline{i_{EO}}\, \overline{i_{NS}} =$$
$$= \overline{X}\, i_{EO} + X\, \overline{i_{NS}}$$

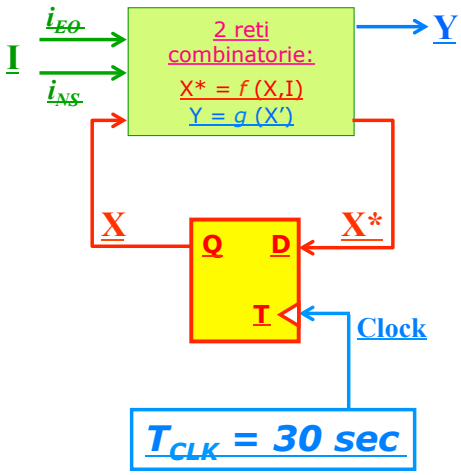
$$Y = X$$

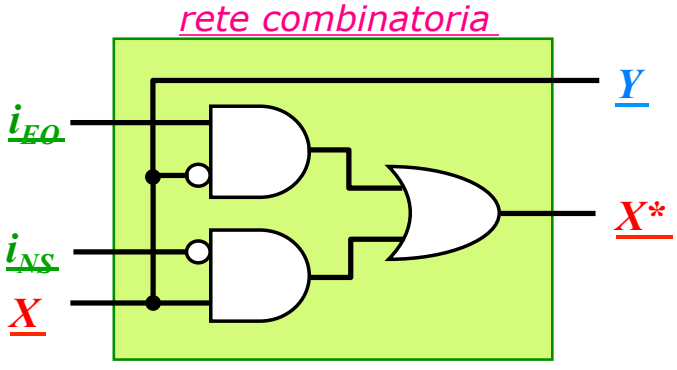
X \ I(i_{EO},i_{NS})	00	01	10	11	Y
0	0	0	1	1	0
1	1	0	1	0	1

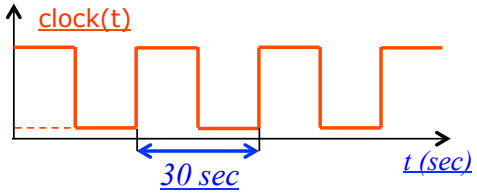
MSF del semaforo: sintesi del circuito

- funzioni logiche rete combinatoria:

$$X^* = \overline{X}\, i_{EO}\, \overline{i_{NS}} + \overline{X}\, i_{EO}\, i_{NS} + X\, i_{EO}\, \overline{i_{NS}} + X\, \overline{i_{EO}}\, \overline{i_{NS}} =$$
$$= \overline{X}\, i_{EO} + X\, \overline{i_{NS}}$$
$$Y = X$$





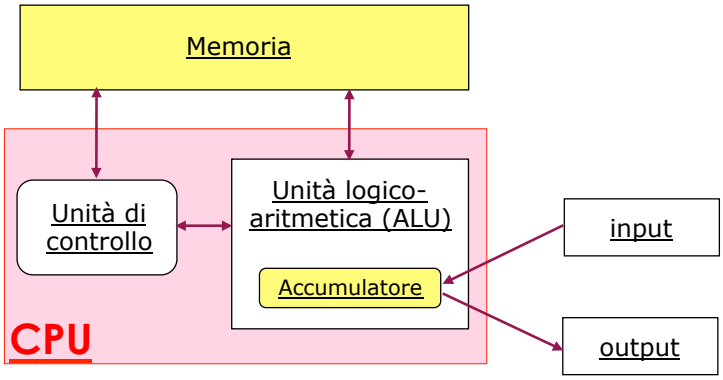


Architettura di riferimento di un elaboratore

- **Architettura di VON NEUMANN:**
  - Dati e Istruzioni sono contenute in una memoria leggibile e scrivibile
    Accesso al contenuto della memoria, in base alla sua posizione (Indirizzo)
  - Esecuzione **sequenziale** da un'istruzione alla seguente
  - I risultati di un'operazione sono sempre contenuti nell'unico registro della CPU chiamato **accumulatore**

ALU + UC incorporate nella CPU

Von Neumann: Architettura ad Accumulatore



Tipi di architetture

Architetture moderne:

Accumulator (1 register = 1 indirizzo di memoria)

1 address	**add A**	acc ← acc + mem[A]
1+x address	**addx A**	acc ← acc + mem[A + x]

Stack (posso operare solo sui dati in cima allo stack)

0 address	**add**	mem[ToS] ← mem[ToS] + mem[ToS – 1]

Register-addressed Memory (tanti indirizzi di memoria quanti sono i registri: indirizzamento indiretto)

2 address	**add A B**	mem(A) ← mem(A) + mem(B)
3 address	**add A B C**	mem(A) ← mem(B) + mem(C)

Load/Store (posso operare solamente sui dati contenuti nei registri. Devo prima caricarli dalla memoria).

3 address:	**add Ra Rb Rc**	Ra ← Rb + Rc
	load Ra [Rb]	Ra ← mem[Rb]
	store Ra [Rb]	mem[Rb] ← Ra

CISC vs. RISC

Statistica di utilizzo delle istruzioni:

La maggior parte del tempo di elaborazione viene impiegato per eseguire poche istruzioni semplici.

IDEA: → Ottimizzo le istruzioni semplici, trascuro il resto

→ Set di istruzioni ridotto, solo istruzioni semplici **RISC**

Rank	Instruction	Average Execution Frequency (%)
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	96%

Architetture RISC

- **RISC**: **R**educed **I**nstruction **S**et **C**omputer
- IDEA: eseguire soltanto istruzioni semplici
  - Operazioni complesse scomposte in serie di istruzioni semplici da eseguire in un ciclo-base ridotto, ma ottimizzato.
  - Architettura semplice: ad esempio, Load-Store – gli operandi dell'*ALU* possono provenire solo dai registri, *non* dalla memoria
- Vantaggi:
  - *CPU* semplice → si riducono i tempi di esecuzione delle singole istruzioni
  - Dimensione **fissa** delle istruzioni → semplice gestione della fase di prelievo ***(fetch)*** e della decodifica delle istruzioni

Architetture di tipo CISC

- **CISC**: **C**omplex **I**nstruction **S**et **C**omputer
  Elevata complessità delle istruzioni eseguibili
  Elevato numero di istruzioni disponibili
- Numerose modalità di indirizzamento per gli operandi dell'ALU
  - possono provenire da registri oppure da memoria
  - Indirizzamento diretto, indiretto, con registro base, ecc.
- Dimensione variabile delle istruzioni a seconda della modalità di indirizzamento di ogni operando
  - complessità di gestione della fase di prelievo o fetch in quanto a priori non è nota la lunghezza dell'istruzione da caricare.
- Elevata complessità della CPU
  - Rallentano i tempi di esecuzione delle operazioni.
  - Elevata profondità dell'albero delle porte logiche, utilizzato per la decodifica.

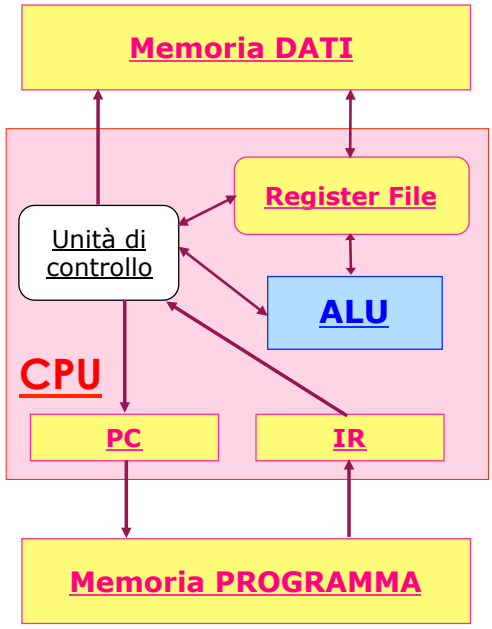
Architettura **LOAD/STORE**

- Registri ad uso generale insufficienti a memorizzare tutte le variabili di un programma
  - Esempio: 32 registri da 32 bit ciascuno
  - Assegnata una locazione di memoria ad ogni variabile nella quale trasferire il contenuto del registro quando questo deve essere utilizzato per contenere un'altra variabile.

- *Architetture **LOAD/STORE:***
  - Gli **operandi dell'ALU possono provenire soltanto dai registri** contenuti nella CPU e **non** possono provenire direttamente **dalla memoria**.
  - Sono necessarie apposite istruzioni di:
    *caricamento **(LOAD)*** dei dati da memoria ai registri;
    *memorizzazione **(STORE)*** dei dati dai registri alla memoria.

CPU – elementi principali

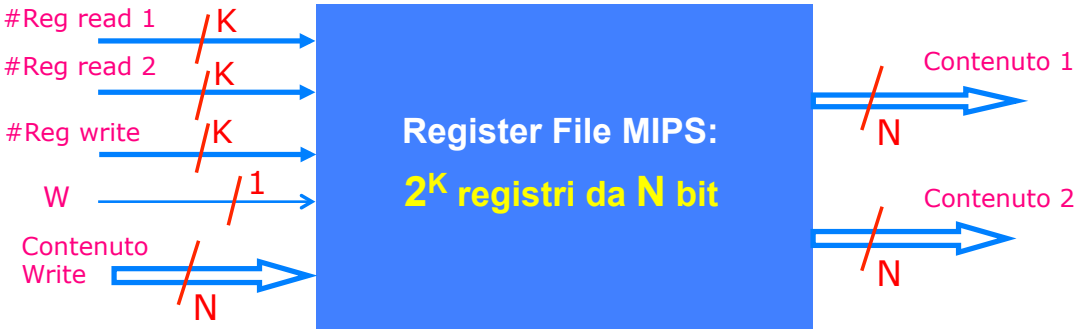
- **Registri:**
  - Banco di registri di uso generale (Register File)
    - Memoria ad accesso rapido, in cui memorizzare i dati di utilizzo più frequente
  - **Program Counter (PC)**
    - Contiene l'indirizzo dell'istruzione corrente da aggiornare durante l'evoluzione del programma, in modo da prelevare dalla memoria la corretta sequenza di istruzione
  - **Instruction Register (IR)**
    - Contiene l'istruzione in corso di esecuzione.
- **Arithmetic Logic Unit – (ALU)**
  - Effettua le operazioni aritmentico logiche fra registri, oppure direttamente dalla memoria (modalità di indirizzamento)
- **Unità di controllo**
  - "Cervello" della CPU
  - Coordina i flussi di informazione, in funzione dell'istruzione in corso
  - Seleziona l'operazione opportuna della ALU.



Registri della CPU: **Register File**

- **Register file**: elemento di una CPU che contiene tutti i registri di uso generale (general purpose registers)
  - Banco di registri: 2^K **registri** da **N bit** ciascuno
  - Solitamente, **2 porte di uscita, 1 in ingresso**

- Operazioni:
  - SELEZIONE: fornendo in ingresso il numero del registro (#reg)
  - LETTURA: non modifica il contenuto del registro selezionato
  - SCRITTURA: Inserisce <ContenutoWrite> nel registro selezionato (comando: W)



Indirizzi nella memoria principale

MEMORIA

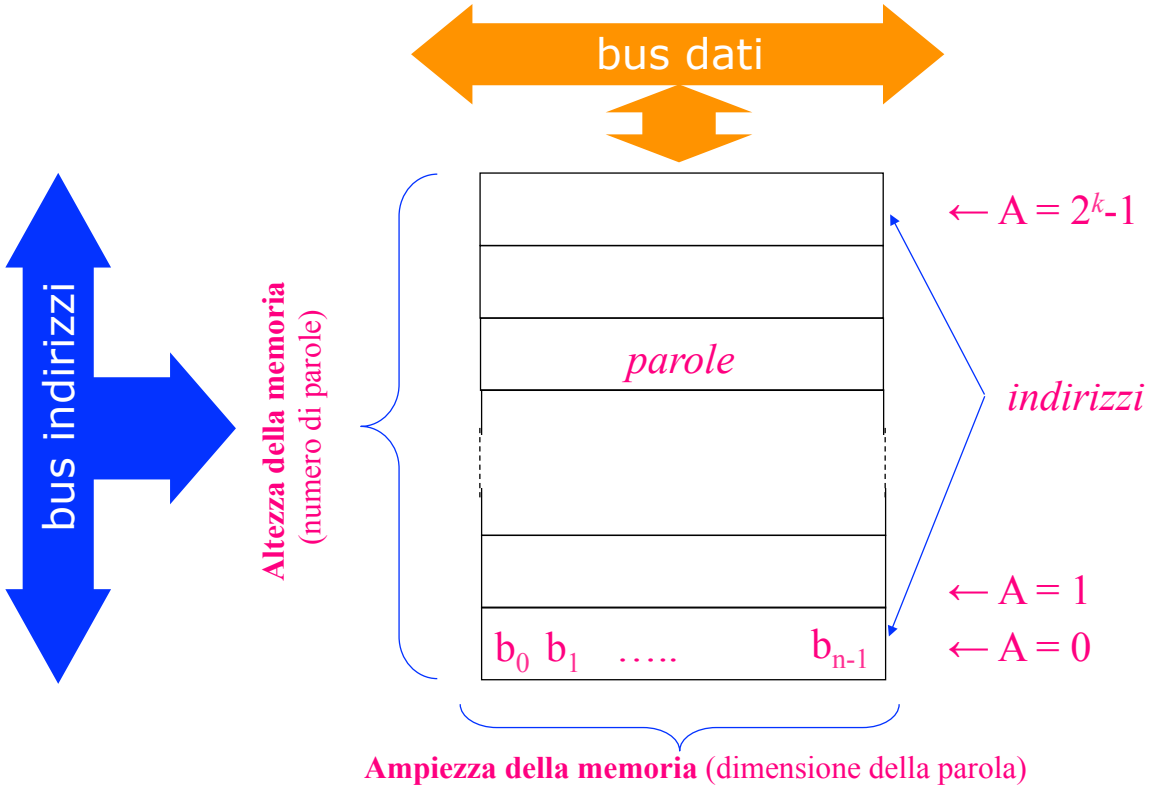
- La memoria è organizzata in 2^K ***parole (word)*** di ***N bit***
  N: ampiezza *della memoria*
  2^K***: altezza*** *della memoria*

- In genere, la dimensione della parola di memoria coincide con la dimensione dei registri della *CPU (CPU word)*
  - Le operazioni di *load/store* avvengono in un **singolo ciclo**

- **Capacità della memoria =** *(N.words)* x *(dim.word [bytes])*
  - ***k = 32 → (2^{32}= 4 Gwords) x (32 bit = 4 bytes) = 16 Gbytes***

- *Random Access Memory –* ***RAM***
  - **Il tempo di accesso alla memoria** è **fisso e indipendente dalla posizione della parola** alla quale si vuole accedere.

Indirizzi nella memoria principale

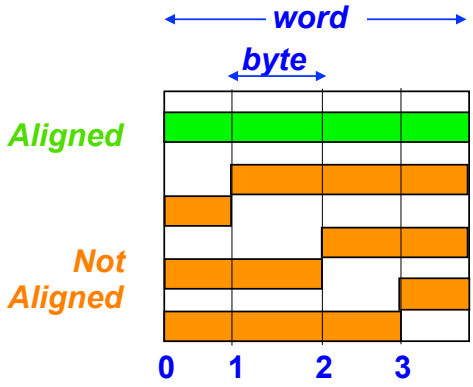


La memoria nel MIPS

- La memoria è vista come un unico grande
  array uni-dimensionale di bytes/words
  - La memoria contiene sia istruzioni che dati (Von Neumann)
  - **Indirizzo di memoria** → indice all'interno dell'array

- Indirizzamento al byte
  - l'indice punta ad un byte di memoria
- Indirizzamento alla parola
  - La memoria è vista come un array di PAROLE
  - 32 bit: gli indirizzi di parole consecutive differiscono di un fattore 4

- Alignment:
  - L'indirizzamento al byte mi permette di scrivere a cavallo tra 2 parole
  - Le parole devono avere indirizzi MULTIPLI della loro dimensione

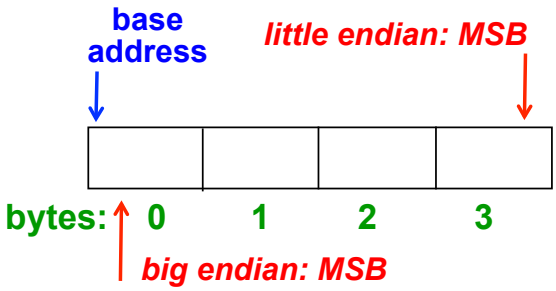


"Endianess" e "alignment"

- **Endianess**:
  come si rappresenta un dato su **più bytes**

  Big Endian:
  Word address = address of MSB
  (most significant byte)
  - Motorola 68k, PowerPC, MIPS, Sparc, HP-PA

  Little Endian:
  Word address = address of LSB
  (least significant byte)
  - Intel x86, DEC Vax, DEC Alpha



Disposizione Big-Endian

Big-endian:
- i byte sono numerati partendo dalla posizione **più** significativa;
- indirizzo di parola = indirizzo del suo byte più significativo.

	Indirizzo di byte			
Parola: **0**	0	1	2	3
Parola: **4**	4	5	6	7
Parola: **8**	8	9	10	11
…				
…				
Parola: $2^k - 4$	2^k-4	2^k-3	2^k-2	2^k-1

IEEE 754

Disposizione Little-Endian

Little-endian:

- i byte sono numerati partendo dalla posizione **meno** significativa;
- indirizzo di parola = indirizzo del suo byte meno significativo.

Indirizzo di byte

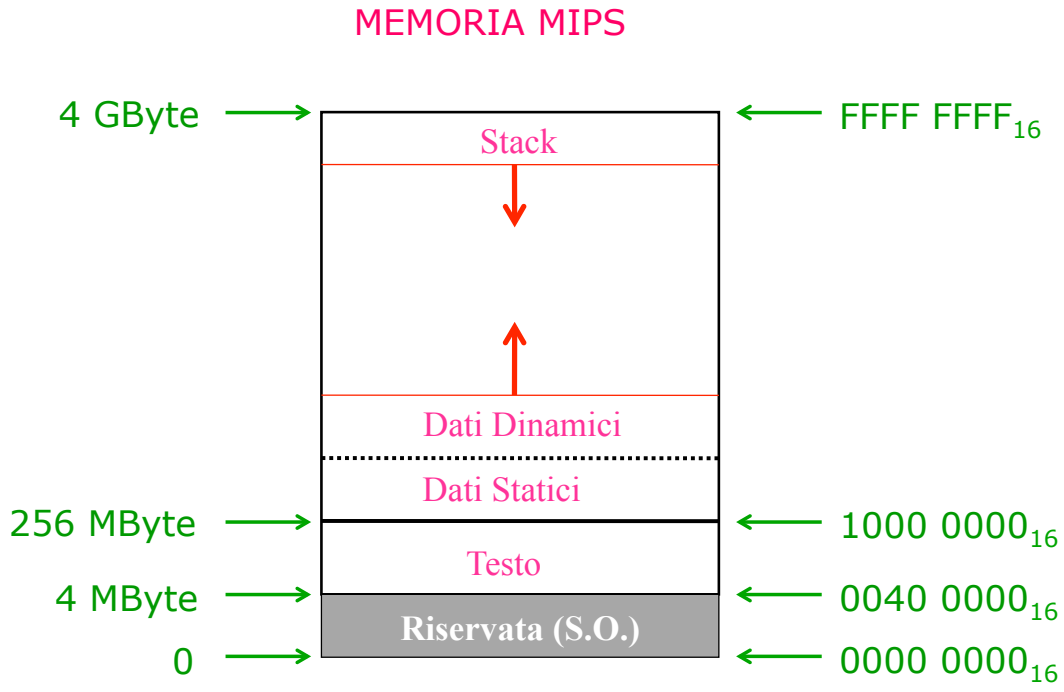
Parola: **0**	3	2	1	0
Parola: **4**	7	6	5	4
Parola: **8**	11	10	9	8
…				
…				
Parola: $2^k - 4$	2^k-1	2^k-2	2^k-3	2^k-4

IEEE 754

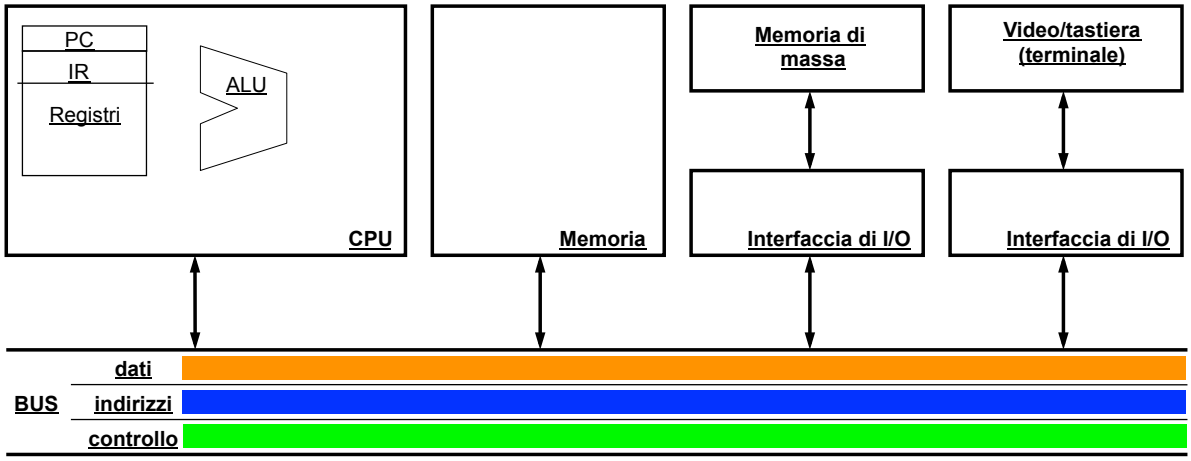
Organizzazione logica della memoria

- Organizzazione memoria MIPS Intel:
- La memoria associata un programma è divisa in **tre parti**:
  - **Segmento testo**: contiene le **istruzioni del programma**
  - **Segmento dati**: contiene i **dati di elaborazione**
    Ulteriormente suddiviso in:
    - **dati statici:** contiene dati la cui dimensione è conosciuta al momento della compilazione e il cui intervallo di vita coincide con l'esecuzione del programma
    - **dati dinamici:** contiene dati ai quali lo spazio è **allocato dinamicamente** al momento dell'esecuzione del programma su richiesta del programma stesso.
  - **Segmento stack**: contiene lo **stack** allocato automaticamente da un programma durante l'esecuzione.

Organizzazione logica della memoria



Collegamento tra componenti mediante BUS



Connessione *a nodo comune: BUS*

Tutte le unità del calcolatore sono connesse al bus

Bus di sistema

- *BUS: infrastruttura di comunicazione tra le diverse unità del calcolatore.*

  Generalmente composto da tre parti:
  - ***Bus dati***: le linee per trasferire dati e istruzioni da/verso i dispositivi.
  - ***Bus indirizzi***: su cui la *CPU* trasmette l'indirizzo di memoria da cui prelevare/depositare il dato nel caso di lettura/scrittura dalla memoria.
  - ***Bus di controllo***: informazioni ausiliarie per la corretta definizione delle operazioni da compiere e per la sincronizzazione tra *CPU* e memoria

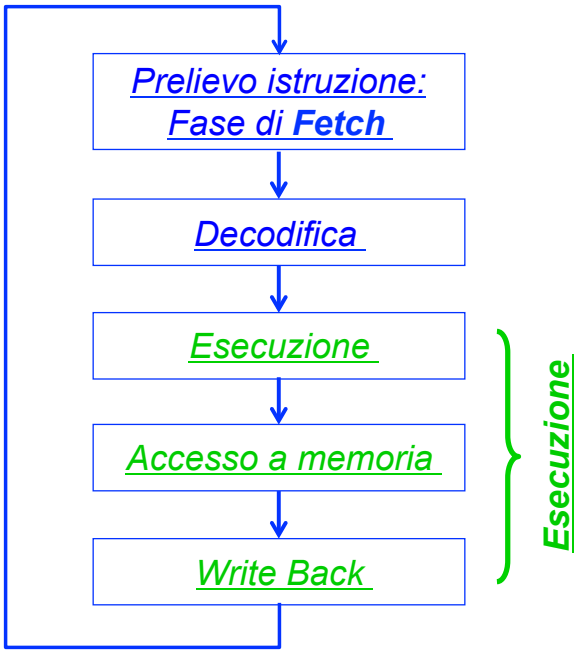
- *Esempio: lettura dalla memoria*
  1. *La CPU fornisce l'indirizzo della parola desiderata sul bus indirizzi*
  2. *viene richiesta l'operazione di **lettura** attivando il bus di controllo.*
  3. *Quando la memoria ha reso disponibile la parola richiesta, il dato viene trasferito sul bus dati e la CPU può prelevarlo dal bus dati ed utilizzarlo nelle sue elaborazioni*

Unità centrale di elaborazione (CPU)

- *Compito della CPU: eseguire in sequenza le istruzioni che costituiscono il programma assegnato all'elaboratore.*
  - *ciclo di esecuzione di istruzioni*

- Eseguire un'istruzione significa:
  - acquisire l'istruzione da eseguire: **FETCH**
  - interpretare l'istruzione e i dati a disposizione: **DECODIFICA**
  - eseguire calcoli/scelte/trasferimenti **ESECUZIONE**

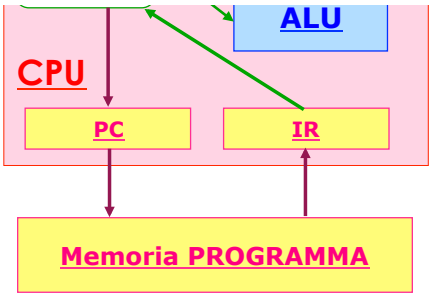
Ciclo esecuzione istruzione

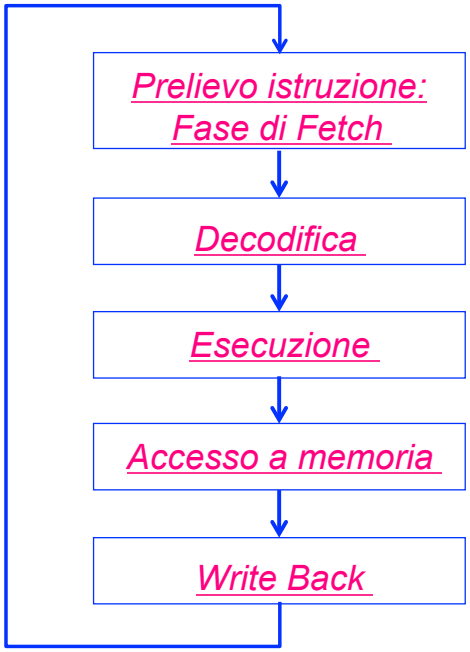


- **Istruzioni e dati** risiedono nella **memoria principale**
- L'esecuzione di un programma inizia quando il registro **PC** punta alla prima istruzione del programma
- **MIPS**: suddivisione dell'esecuzione in tre sottofasi

Lettura istruzione (Fetch)

- Fase di **FETCH**
  - Il segnale di controllo per la **lettura** viene inviato alla memoria
  - Trascorso il tempo necessario all'accesso in memoria, la parola indirizzata (istruzione) viene letta dalla memoria e trasferita nel registro IR





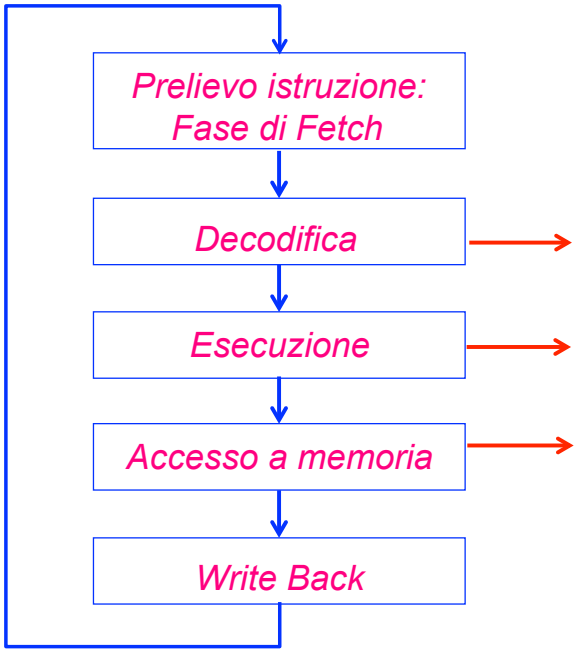
Decodifica dell'istruzione

- DECODIFICA: l'istruzione contenuta nel registro IR viene decodificata per essere eseguita
- Predisposizione della CPU all'esecuzione dell'istruzione:
  - Se l'istruzione è aritmetico-logica, è necessario recuperare gli operandi
- Apertura delle vie di comunicazione appropriate
  - Lettura/scrittura di registri
  - Lettura/scrittura verso memoria
  - Lettura/scrittura dall'esterno (I/O)
  - …

Write-back: Riscrittura risultato

- Esecuzione:
  - Vengono selezionati i circuiti combinatori appropriati per l'esecuzione dell' istruzione e determinate in fase di decodifica.
- Accesso a MEMORIA:
  - SCRITTURA: se il risultato dell'operazione deve essere posto in memoria, esso viene inviato al bus dati. L'indirizzo viene posto sul bus indirizzi e si comanda una scrittura (WRITE) in memoria.
  - LETTURA: se il dato va prelevato dalla memoria, l'indirizzo viene posto sul bus indirizzi e si attiva una lettura (READ), che porta la parola sul bus dati.
- Write-back:
  - Il risultato dell'operazione può essere memorizzato in un registro.
- Chiusura ciclo:
  - Mentre viene eseguita un'istruzione, il contenuto del PC viene incrementato in modo da puntare alla prossima istruzione da eseguire.

Ciclo di esecuzione di un'istruzione



- Il normale flusso di esecuzione di un programma può essere interrotto da *Eccezioni*:

ESTERNE: un segnale di interruzione esterno *(INTERRUPT)* per una richiesta di intervento.

INTERNE: Un segnale di errore di calcolo (es. divisione per 0)

Riferimenti

Riferimenti bibliografici:

- Queste slide (sintesi degli argomenti)
- I vostri testi di: Architetture/Calcolatori elettronici/Elettronica digitale

Elettronica digitale / reti logiche:

- M. Morris, C. Kime – ***Reti logiche*** – Pearson
- F. Fummi, M. Sami, C. Silvano – ***Progettazione digitale*** – McGraw-Hill

Architettura degli elaboratori

- D. Patterson, J. Hennessy – ***Struttura e progetto dei calcolatori*** – Zanichelli

Strumenti didattici:

- **LOGISIM:** http://sourceforge.net/projects/circuit/
- **GATESIM:** http://www.kolls.net/gatesim/