

Architettura degli Elaboratori

Lezione 1

Docente: R.Prevete
a.a. 2022/2023
8/3/2023

Organizzazione e scopi del corso



Questo corso si focalizza su “come sono fatti” e “come funzionano” i sistemi digitali.

- Cosa sono i sistemi digitali: ‘1’ e ‘0’
- Porte logiche digitali che accettano ‘1’ e ‘0’ come input e danno ‘1’ e ‘0’ come output
- Come combinare porte logiche digitali in circuiti/moduli più complessi, come memorie e sommatori
- Linguaggio nativo di un microprocessore: l’assembler e programmi in assembler
- Architettura di un microprocessore che è capace di eseguire programmi in linguaggio assembler

Organizzazione e scopi del corso

Capacità raggiunte alla fine del corso:

- Progettare moduli composti da porte logiche digitali
- L'architettura di un microprocessore
- Scrivere programmi in assembler

Libro di testo

Harris, S. L., & Harris, D.. Digital design and computer architecture: arm edition.
Morgan Kaufmann.

Ricevimento: Giovedì, Martedìforse la cosa migliore è mettersi d'accordo tramite email per luogo ed orario (Teams).

Mia email: rprevete@unina.it

Pagina web: www.docenti.unina.it/roberto.prevete

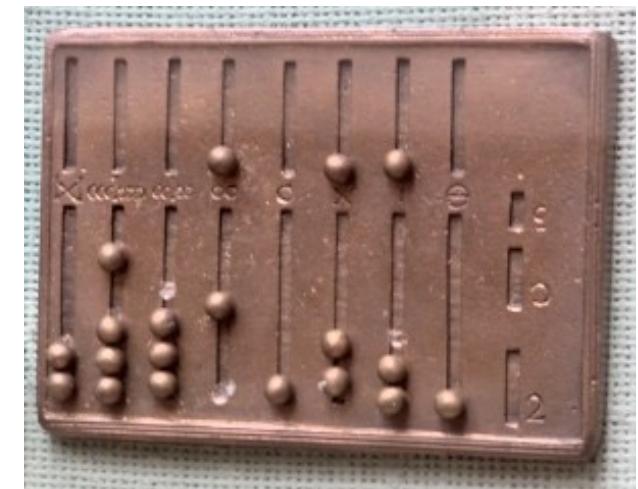
Valutazione

- Prova scritta
- Prova orale

Breve storia delle «macchine per il calcolo»

L'abaco

- Introdotto circa nel 2000 a.C
- Facilità la rappresentazione dei numeri ed il calcolo
- Diversi tipi di abaco (a lapilli, a bottoni, etc..)
- Per diversi migliaia di anni nessun miglioramento sostanziale



Ricostruzione Abaco di epoca romana

Breve storia delle «macchine per il calcolo»

Primi miglioramenti

- 1642, la “pascalina”, progettata da Blaise Pascal (addizionare e sommare numeri con al massimo 12 cifre)
- 1671, la macchina da calcolo di Leibniz, (moltiplicazione, e persino di estrarre la radice quadrata)
- Entrambe, ovviamente, meccaniche



Breve storia delle «macchine per il calcolo»

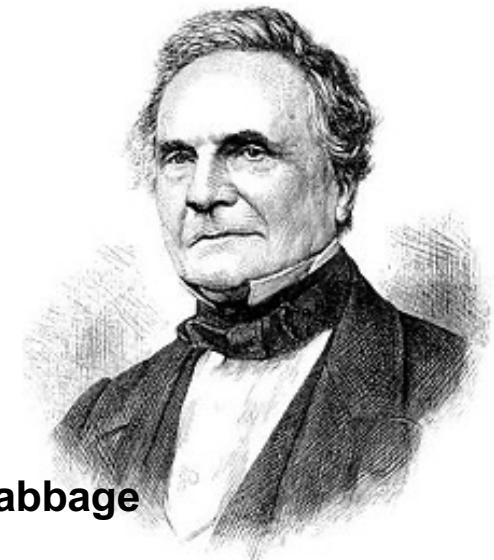
Nuove idee: Calcolatrice vs Calcolatore

- Calcolatrice: Funzioni specifiche
- Calcolatore: Può ricevere delle istruzioni per essere programmato

Breve storia delle «macchine per il calcolo»

Charles Babbage (primo studio teorico, 26 dicembre 1837, *On the Mathematical Power of the Calculating Engine*)^[1]

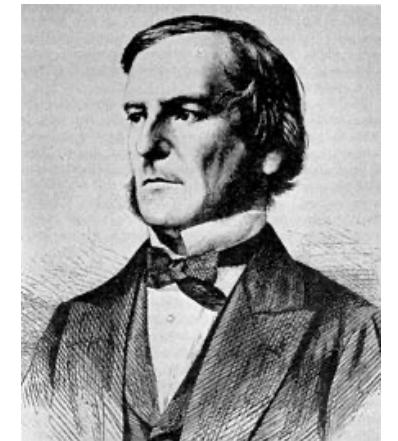
- Il “motore analitico” prevedeva non più una serie fissa di operazioni ma una “programmabilità” vera e propria
- Il motore analitico prevedeva anche un’istruzione di salto condizionale, ovvero di modifica dell’ordine delle istruzioni in base a una condizione
- Questa macchina non fu però mai realizzata
- Rappresenta un primo salto dalla calcolatrice al calcolatore:



Charles Babbage

Breve storia delle «macchine per il calcolo»

- **1842** primi programmi della storia pensati proprio per la macchina analitica di Babbage, da parte della contessa di Lovelace, Ada Byron (1815-1852), figlia del poeta Lord Byron.
- **1854** l'alebra di booleana basata sul codice binario da parte di George Boole (1815-1864)



Breve storia delle «macchine per il calcolo»

- 1936 il logico inglese Alan Turing definisce la cosiddetta 'macchina di Turing':

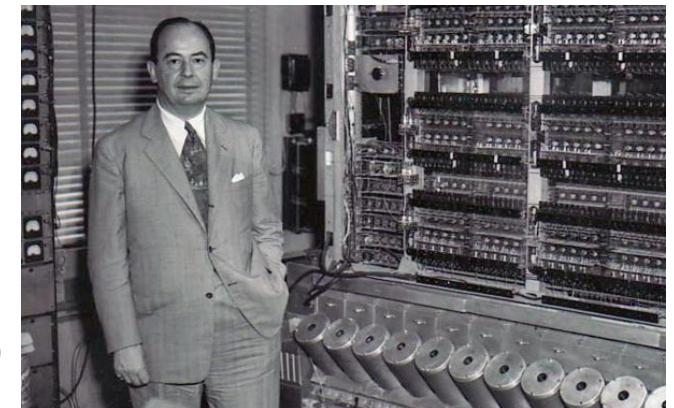
- In grado di eseguire «ogni tipo di calcolo» sulla base di semplici operazioni elementari
- E' una macchina programmabile
- Non ne verranno costruiti esemplari reali, ma la sua idea costituirà la base di tutta l'informatica



Alan Turing (1912-1954)

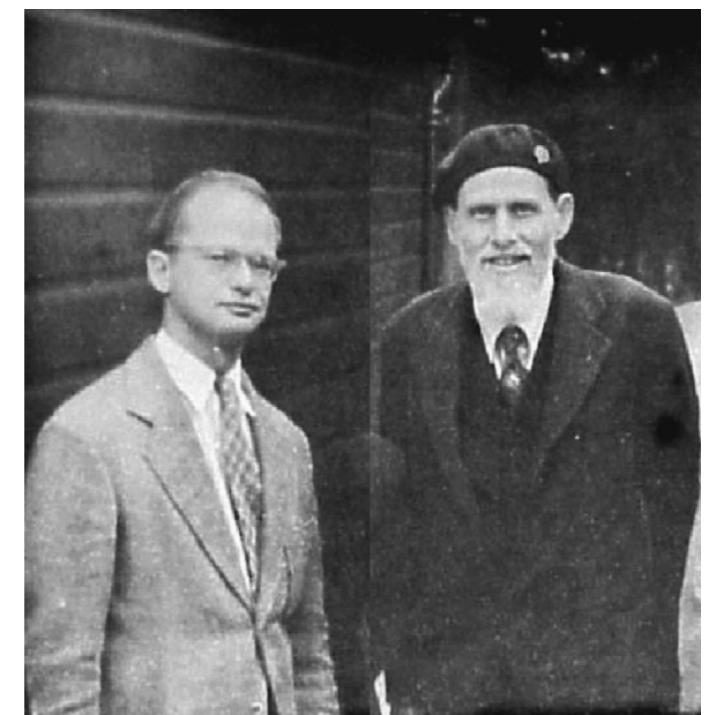
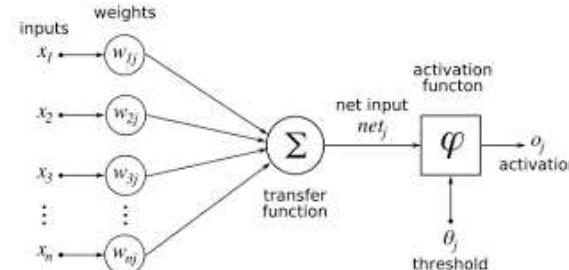
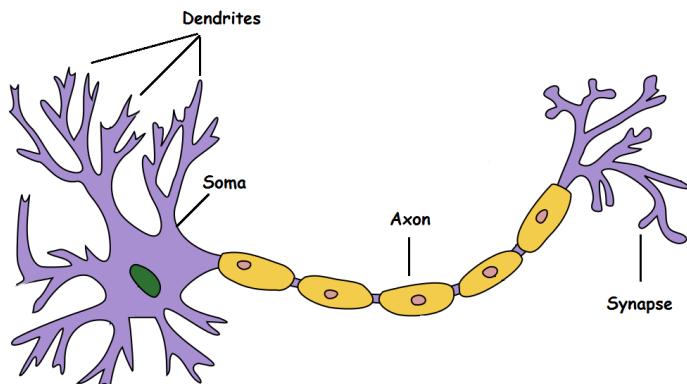
Breve storia delle «macchine per il calcolo»

- 1946 Architettura di Von Neumann
- Si basa su uno schema molto semplice composto da pochi elementi:
 - Processore
 - Memoria centrale
 - Linee di connessione
 - Input/Output
- E' una macchina Programmabile (general purpose)
- Il risultato fu il calcolatore ENIAC. Il primo grande calcolatore elettronico programmabile basato sull'uso di valvole termoioniche.



Breve storia delle «macchine per il calcolo»

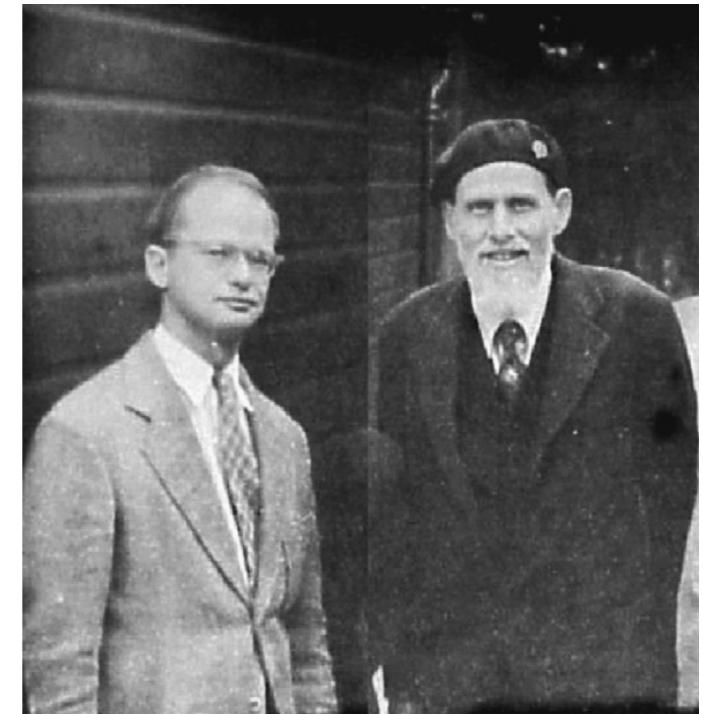
- Altri tipi di architetture: Warren McCulloch & Walter Pitts (1943)
 - Calcolo ispirato dal funzionamento dei neuroni biologici



Pitts (sinistra) e McCulloch (destra)

Breve storia delle «macchine per il calcolo»

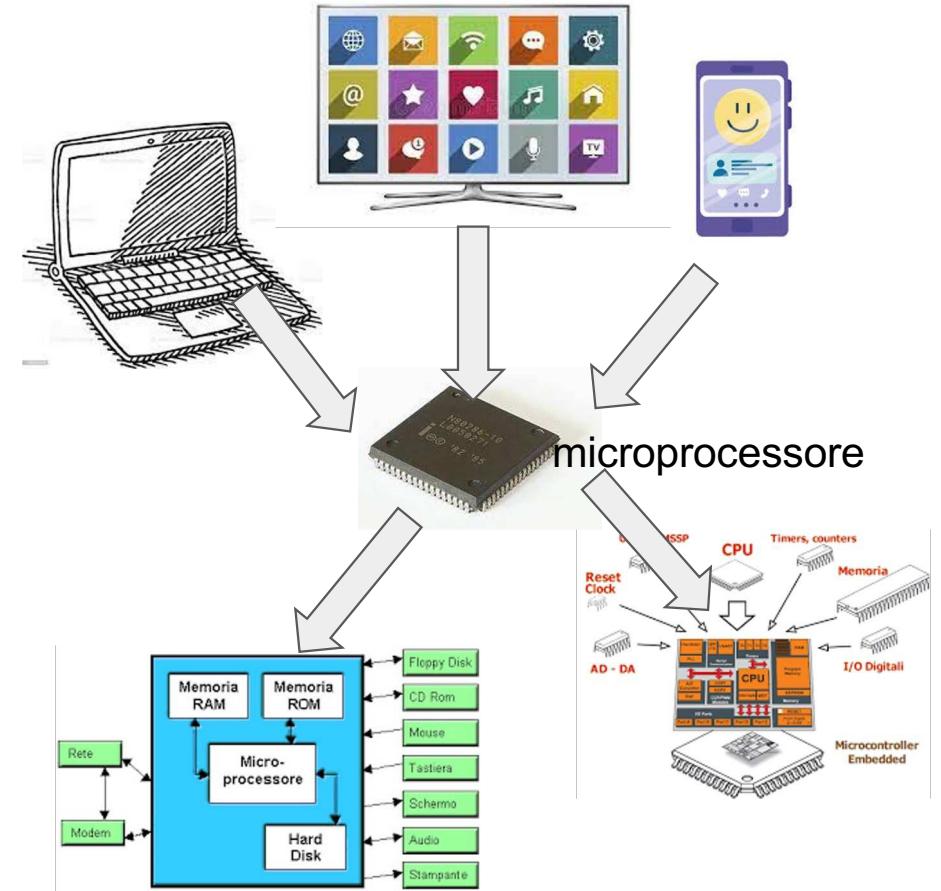
- 1955 -- IBM 702
- 1969 -- processore Intel4004
- 1976 – Z80
- 1984 -- Apple il personal computer Macintosh
- ...fino ai giorni nostri



Pitts (sinistra) e McCulloch (destra)

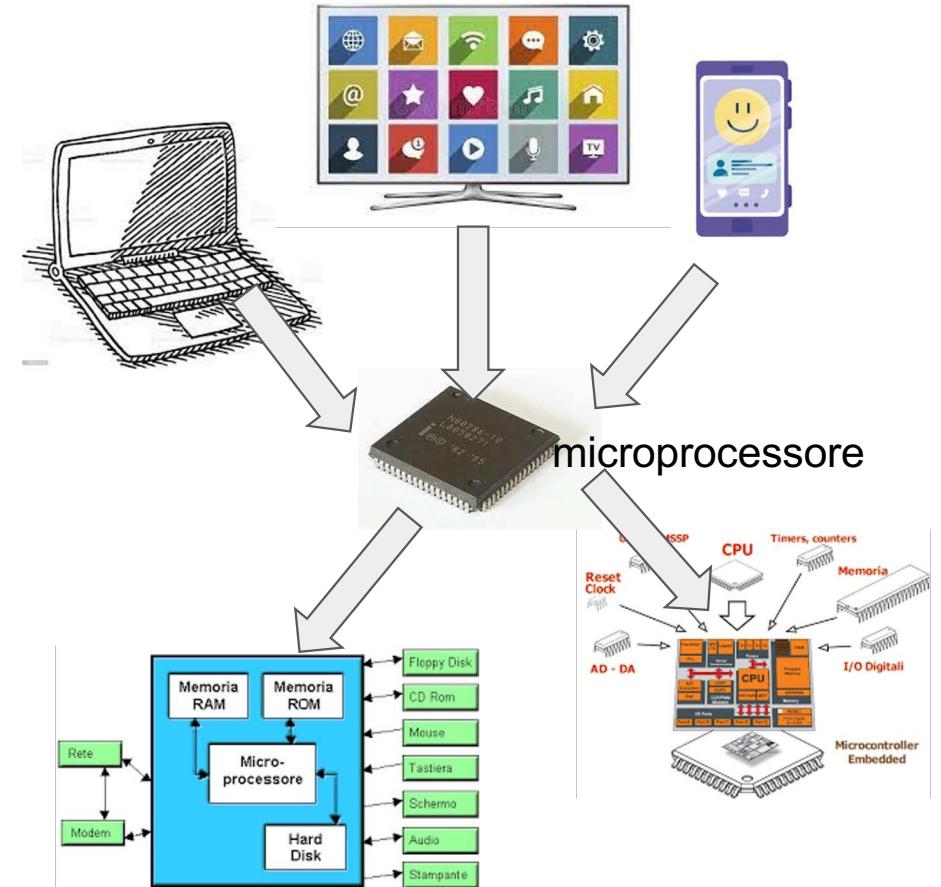
Complessità ed astrazione

- Attuali sistemi digitali (notebook, smartphone, etc..) sono sistemi altamente complessi.
- Essi sono costituiti da milioni o bilioni di componenti come i transistor
- Conoscere e descrivere in maniera “piatta” la loro struttura e il loro comportamento è una impresa impossibile!!!



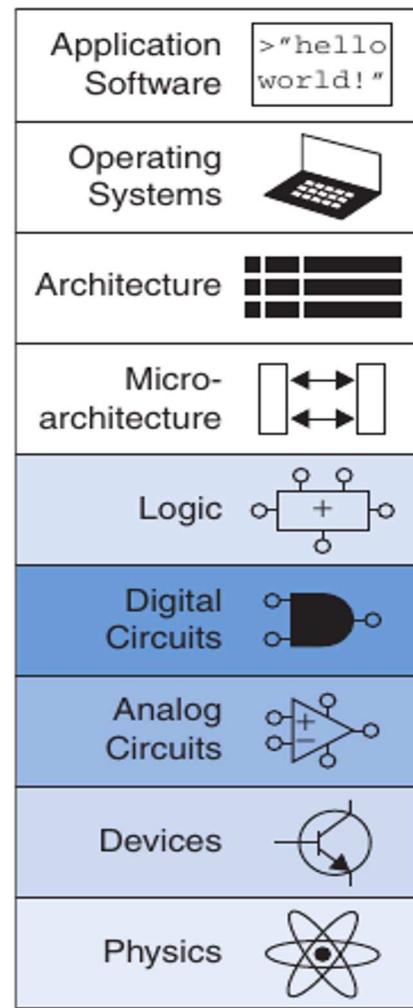
Complessità ed astrazione

- Maniera “piatta” (?):
 - Decrivendo simultaneamente cosa accade in tutto il sistema ad un unico livello di descrizione, ad esempio come gli elettronni si spostano da un elemento all’altro.
- Abbiamo bisogno di affrontare il problema su diversi livelli e poter poi combinare tali livelli tra di loro
- La parola giusta: **ASTRAZIONE!!!**



Complessità ed astrazione

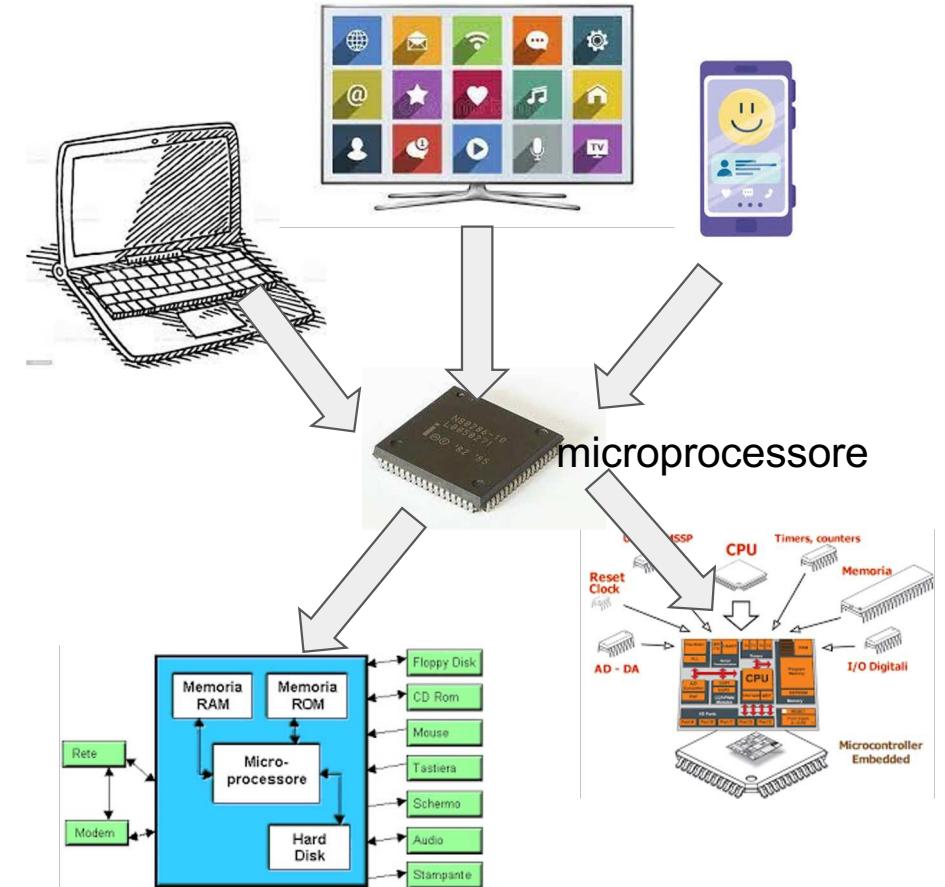
Descrizione a più
alto livello



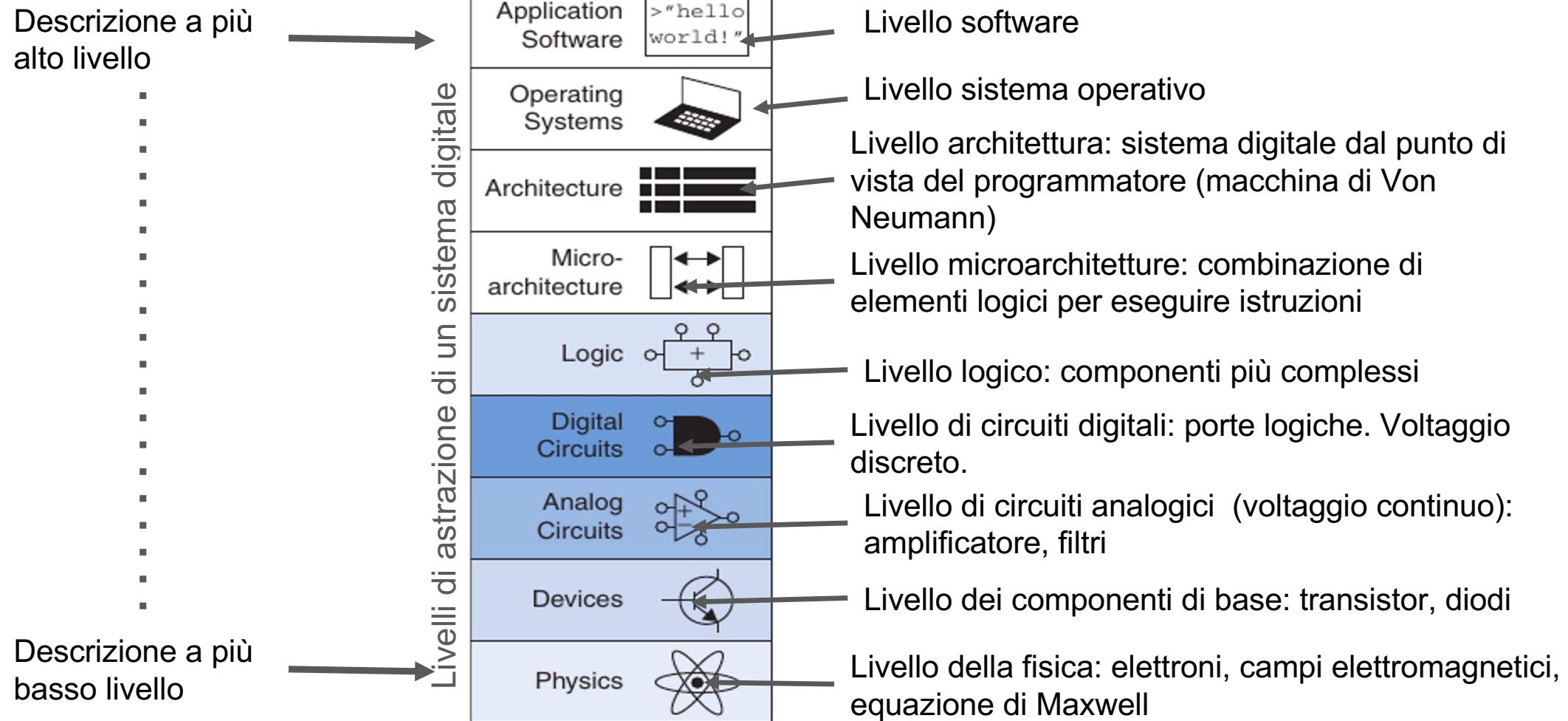
Descrizione a più
basso livello



Livelli di astrazione di un sistema digitale



Complessità ed astrazione

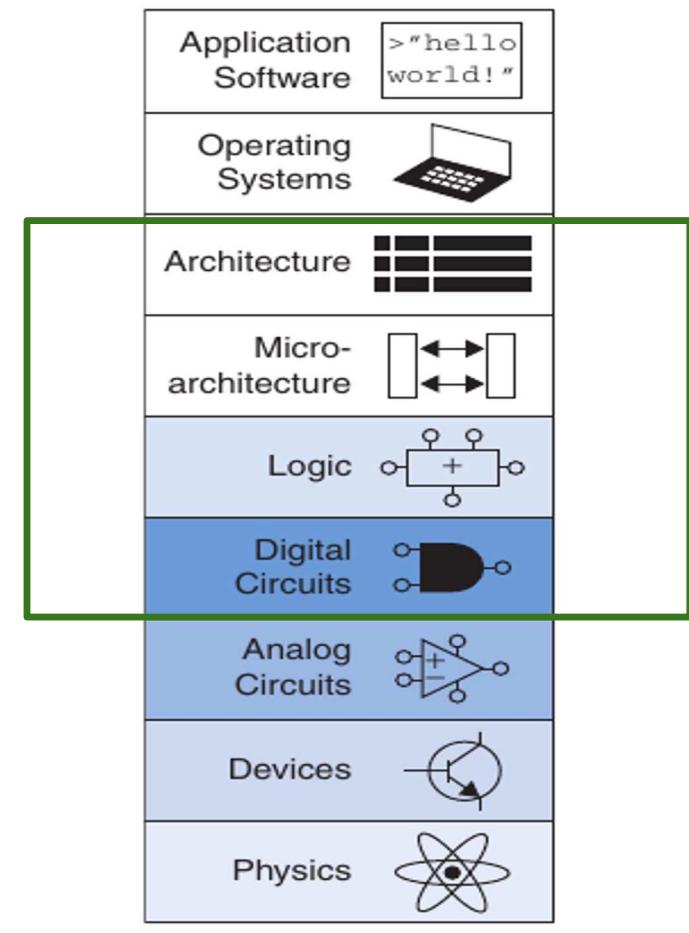


Su cosa ci focalizzeremo

Questo corso, quindi, si focalizzerà sui livelli di astrazione dai circuiti digitali fino all'architettura del computer.

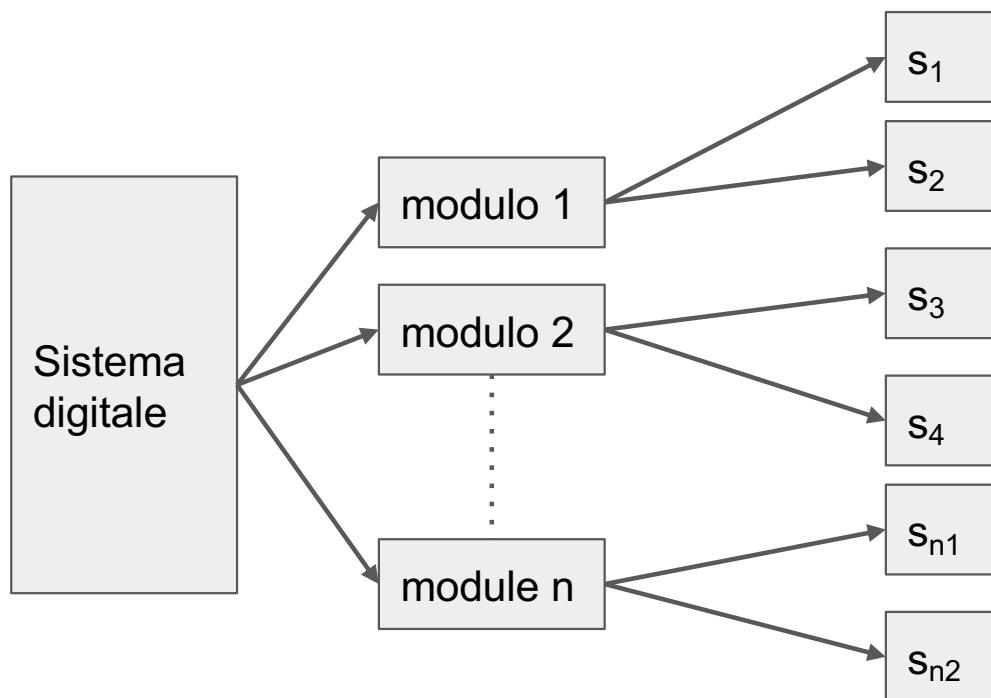
Importante: quando ci si concentra su un livello di astrazione aiuta sapere gli elementi chiave del livello di astrazione precedente e sapere qualcosa del livello di astrazione successivo.

Ad esempio: un informatico riesce ad ottimizzare il proprio codice se comprende l'architettura per la quale il programma (il codice) è scritto.



Ottener una buona gerarchica di livelli di astrazione

Astrazione: tramite il processo di astrazione un sistema è diviso in tanti moduli, poi tali moduli sono allora volta divisi in tanti sotto-moduli, i quali ancora sono divisi in sotto-moduli e così via. Al termine di tale processo si hanno “pezzi” che sono facilmente comprensibili!!



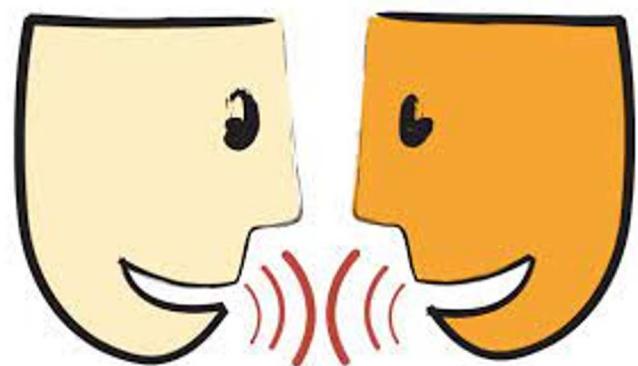
Bottom-up

moduli sempre più facilmente
comprendibili

Quale altre caratteristiche devo avere per ottenere una buona gerarchia di livelli di astrazione?

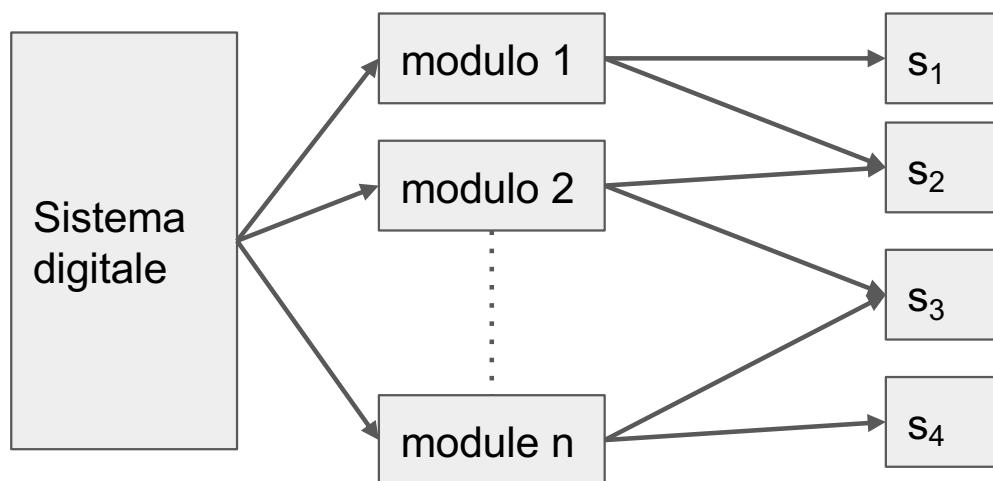
Ottener una buona gerarchica di livelli di astrazione

Modularità: I moduli devono poter “parlare” tra di loro in maniera semplice senza side-effect: *ben definite funzioni ed interfacce*.



Ottener una buona gerarchica di livelli di astrazione

Regolarità (o ri-usabilità): Un sotto-modulo può/deve essere utilizzato da più moduli, in modo da ridurre il numero di sotto-moduli (o in generale moduli) distinti, diversi.



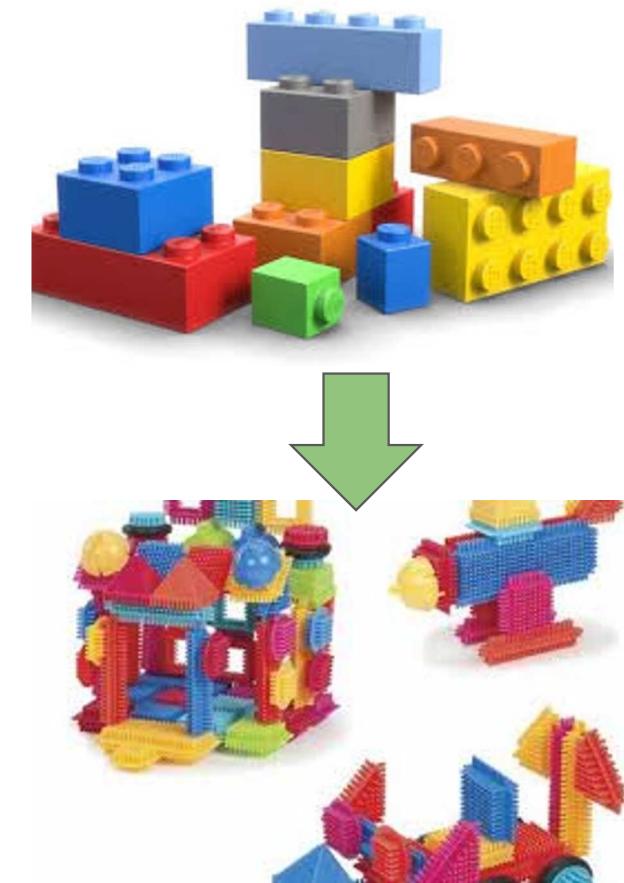
Nell'esempio, s_2 e s_3 sono utilizzati da più moduli

Ottener una buona gerarchica di livelli di astrazione

Riassumendo:

- Gerarchia (Hierarchy)
- Modularità (Modularity)
- Regolarità (Regularity)

Un po' come per i mattoncini lego!



Regola delle tre Y.

ASTRAZIONE DIGITALE

Astrazione digitale: dall'analogico al digitale

La gran parte delle grandezze fisiche sono rappresentate da variabili continue:

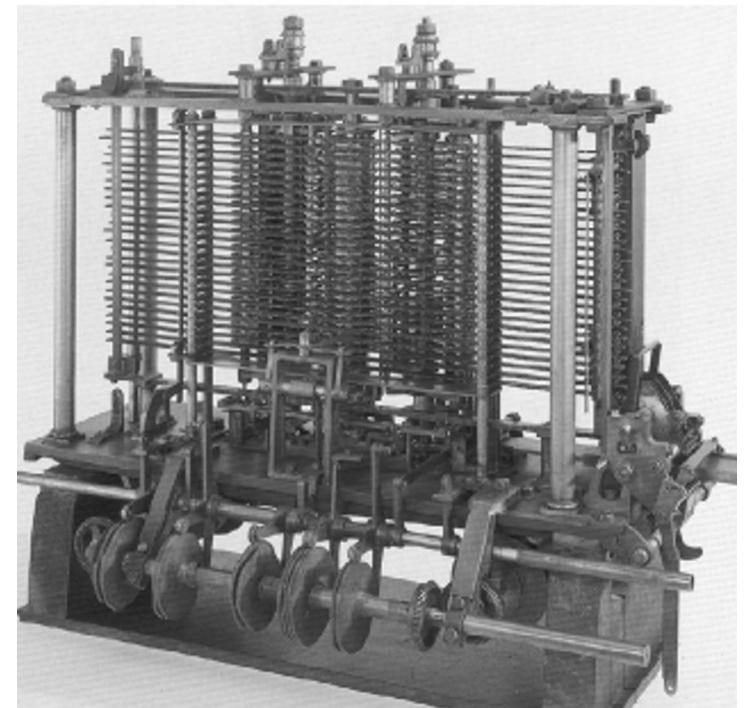
- Il peso
- La quantità di carica
- La differenza di potenziale
- La posizione
- La velocità
- L'accelerazione
- etc...

In questi casi, cioè, si usano variabili, ad esempio x , che assumono valori continui. Ad esempio, $x \in [a,b] \subset R$ (cioè x appartiene ad un intervallo di estremi a e b , il quale è incluso nell'insieme dei numeri Reali)

Astrazione digitale: dall'analogico al digitale

La macchina analitica di Charles Babbage
(1871)

- 25 file (righe) di ingranaggi
- Ciascun ingranaggio ha 10 diverse posizioni (0-9)



Astrazione digitale: dall'analogico al digitale

Oggi:

- fenomeno fisico (continuo): differenza di potenziale
- alto voltaggio '1' (~5V)
- basso voltaggio '0' (~0V)
- **rappresentazione binaria**

Astrazione digitale: dall'analogico al digitale

Osserviamo:

- fenomeno fisico (continuo): qualunque
- Sia possibile distinguere due “condizioni” diversi
- Ad una associamo ‘1’, all’altra ‘0’
- Si ottiene: rappresentazione binaria

Una rappresentazione binaria (digitale) può essere ottenuta indipendentemente dallo substrato fisico (differenza di potenziale, ingranaggi o altro)

Astrazione digitale: dall'analogico al digitale

Osserviamo:

- Nel 1854 George Boole propose un sistema di operazione logiche basate su variabili binarie (dette variabili Booleane).
- In questo caso i due possibili valori vengono detti:
 - TRUE (VERO)
 - False (FALSO)

In questo corso, quindi, useremo in maniera intercambiabile i termini:

- TRUE, VERO, 1, ALTO, HIGH
- FALSE, FALSO, 0, BASSO, LOW

Astrazione digitale: dall'analogico al digitale

Riassumendo:

L'astrazione digitale consiste nel fatto che ci si può focalizzare su '0' e '1' come valori astratti indipendentemente dalla loro rappresentazione fisica, cioè se corrispondono a specifici voltaggi (differenze di potenziale), ingranaggi, livelli di un fluido, etc...

Architettura degli Elaboratori

Lezione 2

Docente: R.Prevete
a.a. 2022/2023
2023.03.10

Sistemi numerici

Sistemi numerici

Numeri decimali

- 10 simboli diversi: 0-9
- Un numero è una sequenza di tali simboli
- Ciascun simbolo ha un significato diverso a seconda della posizione che occupa
- Da destra a sinistra: la prima posizione corrisponde a 10^0 , la seconda posizione a $10^1, \dots$, la k-th posizione a $10^{(k-1)}$
- Un simbolo **c** in k-th posizione corrisponde a c volte $10^{(k-1)}$: $c \cdot 10^{k-1}$, diremo che c è pesato con 10^{k-1}
- Data un numero di n cifre, esso corrisponde alla somma delle opportunamente pesati
- Dato che ci sono 10 simboli, diremo che tali numeri sono a base **10**

$$c_2 c_1 c_0 = c_2 10^2 + c_1 10^1 + c_0 10^0$$

Sistemi numerici

Numeri decimali: esempi

- 527
 - $c_2 \cdot 10^2 + c_1 \cdot 10^1 + c_0 \cdot 10^0$ con $c_0=7$, $c_1=2$, $c_2=5$
- 52
 - $c_1 \cdot 10^1 + c_0 \cdot 10^0$ con $c_0=2$, $c_1=5$
- 7135
 - $c_3 \cdot 10^3 + c_2 \cdot 10^2 + c_1 \cdot 10^1 + c_0 \cdot 10^0$ con $c_0=5$, $c_1=3$, $c_2=1$, $c_3=7$
- 52
 - $c_2 \cdot 10^2 + c_1 \cdot 10^1 + c_0 \cdot 10^0$ con $c_0=2$, $c_1=5$, $c_2=0$

Sistemi numerici

Numeri decimali

- Riassumendo
 - Il “nostro” sistema di numerazione è un sistema di numerazione posizionale in base 10
 - L'insieme di cifre deve necessariamente essere ordinato;
 - le cifre assumono “pesi” diversi a seconda della posizione in cui si trovano all'interno del numero.

Sistemi numerici

In generale:

- Sistemi posizionali
 - Il nostro sistema decimale
- Sistemi non posizionali

Sistemi numerici

Non posizionale

- Il sistema di conteggio ``a tacche''.
 - Ogni tacca ha il significato di una unità e non importa in quale ordine siano disposte;
- Numeri romani.
 - C'è un ordine di lettura (da sinistra verso destra), ma i simboli assumono sempre lo stesso valore: per esempio X è sempre 10, indipendentemente dalla sua posizione

Sistemi numerici

- **Parleremo solo di sistemi numerici posizionali (ovviamente)**
 - **esistono differenti sistemi numerici posizionali**
 - La differenza fondamentale è la base scelta

Per evitare confusione tra numeri a base diversa, la base è indicate come pedice:

$(759)_{10}$ oppure semplicemente 759_{10}

Sistemi numerici

- Ci focalizzeremo:
 - **Sistema binario (2 simboli)**
 - **Sistema esadecimale (16 simboli)**
 - **Trasformazione da un sistema all'altro**
 - **Vantaggi e svantaggi dei differenti sistemi**

Numeri binari

- Simboli usati: ‘1’ e ‘0’
- **Esempi:** 1011, 101, 1000, 1111
- BASE: 2

ESEMPIO: $101 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

Ricorda:

$2^0=1$ $2^1=2$ $2^2=4$ $2^3=8$ $2^4=16$ $2^5=32$ $2^6=64$ $2^7=128$ $2^8=256$ $2^9=512$ $2^{10}=1024$

$2^{11}=2048$ $2^{12}=4096$ $2^{13}=8192$ $2^{14}=16384$ $2^{15}=32768$ $2^{16}=65536$

Numeri binari

- Ciascun elemento del numero binario è detto bit.
- Un numero binario formato da N bit è detto di *lunghezza N*
- Un numero binario di N bit può avere 2^N configurazioni diverse e può rappresentare uno di 2^N numeri: 0, 1, 2, ..., 2^N-1

Ad esempio , se $N=2$ $2^N=4$

| | | | |
|----|---|-------------------------------|------------|
| 00 | = | $0 \times 2^1 + 0 \times 2^0$ | $= 0_{10}$ |
| 01 | = | $0 \times 2^1 + 1 \times 2^0$ | $= 1_{10}$ |
| 10 | = | $1 \times 2^1 + 0 \times 2^0$ | $= 2_{10}$ |
| 11 | = | $1 \times 2^1 + 1 \times 2^0$ | $= 3_{10}$ |

Numeri binari

Esempi di numeri binari e corrispondente numero decimale

| 1-Bit Binary Numbers | 2-Bit Binary Numbers | 3-Bit Binary Numbers | 4-Bit Binary Numbers | Decimal Equivalents |
|----------------------------|----------------------------|----------------------------|----------------------------|------------------------|
| 0 | 00 | 000 | 0000 | 0 |
| 1 | 01 | 001 | 0001 | 1 |
| | 10 | 010 | 0010 | 2 |
| | 11 | 011 | 0011 | 3 |
| | | 100 | 0100 | 4 |
| | | 101 | 0101 | 5 |
| | | 110 | 0110 | 6 |
| | | 111 | 0111 | 7 |
| | | | 1000 | 8 |
| | | | 1001 | 9 |
| | | | 1010 | 10 |
| | | | 1011 | 11 |
| | | | 1100 | 12 |
| | | | 1101 | 13 |
| | | | 1110 | 14 |
| | | | 1111 | 15 |

Numeri binari: conversione da decimale a binario

- E' abbastanza semplice la conversione da numero binario a numero decimale.
- **Come, invece, convertire da decimale a binario?**
- Possiamo trovare due soluzioni:
 - La prima, forse più intuitiva, è quella di andare da sinistra a destra del numero binario
 - La seconda, algoritmicamente più valida, da destra a sinistra del numero binario

Conversione da decimale a binario: da sinistra a destra

Esempio

$68_{10} = c_k \times 2^k + c_{k-1} \times 2^{k-1} + \dots + c_0 \times 2^0$, chi sono k e c_k, c_{k-1}, \dots, c_0 ?

- Prendo il massimo tra le potenze di due minori o uguali a 68_{10} :
 - $2^6 = 64_{10}$ ottengo che il digit in posizione $6+1$ ($k=6$) è pari a 1, cioè 1000000
- Sottraggo al numero $68_{10} - 64_{10} = 4_{10}$
- Di nuovo, prendo il massimo tra le potenze di due minori o uguali a 4_{10}
 - $2^2 = 4_{10}$ ottengo che il digit in posizione $2+1$ ($k=2$) è pari a 1, cioè 1000100
- Sottraggo al numero $4_{10} - 4_{10} = 0$
- Ho raggiunto 0 mi fermo

$$68_{10} = 1000100 = 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$

Conversione da decimale a binario: da sinistra a destra

PSEUDO-CODICE:

INPUT: n_{10}

OUTPUT: b_2

- $b_2 \leftarrow 0$
- WHILE $n_{10} > 0$
 - $k \leftarrow 0$
 - WHILE $2^k \leq n$
 - $k \leftarrow k+1$
 - ENDWHILE
 - $b_2(k) \leftarrow 1$
 - $n_{10} \leftarrow n_{10} - 2^{k-1}$
- ENDWHILE

Massimo delle potenze di 2 minori o uguali a n_{10}

Architettura degli Elaboratori

Lezione 3

Docente: R.Prevete
a.a. 2022/2023
13 marzo 2023

Conversione da decimale a binario: da destra a sinistra

PSEUDO-CODICE:

INPUT: n_{10}

OUTPUT: b_2

- $\text{ris} \leftarrow n_{10}$
- $k \leftarrow 1$
- $b_2(k) \leftarrow 0$
- WHILE $\text{ris} > 0$
 - $r \leftarrow \text{ris \% } 2$ Resto
 - $\text{ris} \leftarrow \text{ris / } 2$ Risultato
 - $b_2(k) \leftarrow r$
 - $k \leftarrow k+1$
- ENDWHILE

In questo caso l'idea di base è dividere (nel campo dei numeri interi) sempre per 2 e prendere risultato e resto

Esempio

$68_{10} = c_k \times 2^k + c_{k-1} \times 2^{k-1} + \dots + c_0 \times 2^0$, chi sono k e c_k, c_{k-1}, \dots, c_0 ?

- **Divido** 68_{10} per 2 ed ottengo resto $r=0$ ed il risultato ris= 34_{10} :
 - Il digit in posizione 1 è pari a $r=0$, $c_0=0$,
- Divido ris= 34_{10} per 2 ed ottengo resto $r=0$ ed il risultato ris= 17_{10} :
 - Il digit in posizione 2 è pari $r=0$, $c_1=0$,
- Divido ris= 17_{10} per 2 ed ottengo resto $r=1$ ed il risultato ris= 8_{10} :
 - Il digit in posizione 3 è pari $r=1$, $c_2=1$
- Divido ris= 8_{10} per 2 ed ottengo resto $r=0$ ed il risultato ris= 4_{10} :
 - Il digit in posizione 4 è pari $r=0$, $c_3=0$
-continua prossima slide!

Esempio

$$68_{10} = c_k x 2^k + c_{k-1} x 2^{k-1} + \dots + c_0 x 2^0, \text{ chi sono } k \text{ e } c_k, c_{k-1}, \dots, c_0?$$

- ...
- Divido $\text{ris}=4_{10}$ per 2 ed ottengo resto $r=0$ ed il risultato $\text{ris}=2_{10}$:
 - Il digit in posizione 5 è pari $r=0$, $c_4=0$
- Divido $\text{ris}=2_{10}$ per 2 ed ottengo resto $r=0$ ed il risultato $\text{ris}=1_{10}$:
 - Il digit in posizione 6 è pari $r=0$, $c_5=0$
- Divido $\text{ris}=1_{10}$ per 2 ed ottengo resto $r=1$ ed il risultato $\text{ris}=0_{10}$:
 - Il digit in posizione 7 è pari $r=1$, $c_6=1$
- Mi fermo perchè $\text{ris}=0_{10}=0$

$$68_{10} = c_6 x 2^6 + c_5 x 2^5 + c_4 x 2^4 + c_3 x 2^3 +$$

$$c_2 x 2^2 + c_1 x 2^1 + c_0 x 2^0 = 1 x 2^6 + 0 x 2^5 + 0 x 2^4 + 0 x 2^3 + 1 x 2^2 + 0 x 2^1 + 0 x 2^0 = 1000100$$

Basi differenti: Numeri esadecimali

- **16 Simboli:** '0', '1', ..., '9', 'A', 'B', 'C', 'D', 'E', 'F'
- **Esempi:** 10A, FBC, 108, EF8
- **BASE:** 16

ESEMPIO: $FBC = F \times 16^2 + B \times 16^1 + C \times 16^0$

In decimali :

$$(FBC)_{16} = F \times 16^2 + B \times 16^1 + C \times 16^0 = 15 \times 16^2 + 11 \times 16^1 + 12 \times 16^0 = (4028)_{10}$$

- Con sequenze più corte di simboli possiamo ottenere molte più configurazioni differenti
- Se N è la lunghezza del numero, 16^N configurazioni

Numeri esadecimali

Più simboli,
stringhe
meno lunghe

| Hexadecimal Digit | Decimal Equivalent | Binary Equivalent |
|-------------------|--------------------|-------------------|
| 0 | 0 | 0000 |
| 1 | 1 | 0001 |
| 2 | 2 | 0010 |
| 3 | 3 | 0011 |
| 4 | 4 | 0100 |
| 5 | 5 | 0101 |
| 6 | 6 | 0110 |
| 7 | 7 | 0111 |
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| A | 10 | 1010 |
| B | 11 | 1011 |
| C | 12 | 1100 |
| D | 13 | 1101 |
| E | 14 | 1110 |
| F | 15 | 1111 |

Numeri esadecimali: da esadecimale a binari

- Ciascun digit esadecimale corrisponde a 4 digit binari:

RICORDANDO:

- $(0)_{16} = 0000_2$ $(1)_{16} = 0001_2$ $(2)_{16} = 0010_2$ $(3)_{16} = 0011_2$
- $(4)_{16} = 0100_2$ $(5)_{16} = 0101_2$ $(6)_{16} = 0110_2$ $(7)_{16} = 0111_2$
- $(8)_{16} = 1000_2$ $(9)_{16} = 1001_2$ $(A)_{16} = 1010_2$ $(B)_{16} = 1011_2$
- $(C)_{16} = 1100_2$ $(D)_{16} = 1101_2$ $(E)_{16} = 1110_2$ $(F)_{16} = 1111_2$

SI OTTIENE FACILMENTE:

- $(A0)_{16} = 10100000_2$ $(F9B)_{16} = 111110011011_2$
- $(1C9B)_{16} = 0001110010011011_2$

Numeri esadecimali: da esadecimali a decimali

- Ciascun digit esadecimale corrisponde a 4 digit binari:

RICORDANDO:

- $A_{16} = 10_{10}$ $B_{16} = 11_{10}$ $C_{16} = 12_{10}$ $D_{16} = 13_{10}$ $E_{16} = 14_{10}$ $F_{16} = 15_{10}$

SI OTTIENE FACILMENTE:

- $(1C9B)_{16} = 1 \times 16^3 + C \times 16^2 + 9 \times 16^1 + B \times 16^0 =$
 $1 \times 16^3 + 12 \times 16^2 + 9 \times 16^1 + 13 \times 16^0 = 4096 + 3072 + 144 + 11 = 7323$

Numeri esadecimali: da binari a esadimali

- Ciascun digit esadecimale corrisponde a 4 digit binari:

RICORDANDO:

- $(0)_{16} = 0000_2$ $(1)_{16} = 0001_2$ $(2)_{16} = 0010_2$ $(3)_{16} = 0011_2$
- $(4)_{16} = 0100_2$ $(5)_{16} = 0101_2$ $(6)_{16} = 0110_2$ $(7)_{16} = 0111_2$
- $(8)_{16} = 1000_2$ $(9)_{16} = 1001_2$ $(A)_{16} = 1010_2$ $(B)_{16} = 1011_2$
- $(C)_{16} = 1100_2$ $(D)_{16} = 1101_2$ $(E)_{16} = 1110_2$ $(F)_{16} = 1111_2$

Possiamo, anche in questo caso, partire da destra e sostituire considerando 4 digit alla volta (se ne sono di meno, si pongono a zero i mancanti)

Esempi: $011_2 \rightarrow 0011_2 \rightarrow 3_{16}$, $1101011_2 \rightarrow (0110)(1011)_2 \rightarrow 6B_{16}$

Numeri esadecimali: da decimale a esadecimali (sinistra-destra)

PSEUDO-CODICE:

INPUT: n_{10}

In maniera simile alla conversione da decimale a binario, cambiando la base

OUTPUT: es_{16}

- $es_{16}(1) \leftarrow 0$
- WHILE $n_{10} > 0$

- $k \leftarrow 0$

- WHILE $16^k \leq n_{10}$

- $k \leftarrow k+1$

- ENDWHILE

- $c \leftarrow n_{10}/16^{k-1}$

Massimo delle potenze di 16 minori o uguali a n_{10}

- IF $c > 9$ THEN $c \leftarrow \text{convert}(c)$

- $es_{16}(k) \leftarrow c$

- $n_{10} \leftarrow n_{10} - c_k \times 16^{k-1}$

Quante volte 16^{k-1} va in n_{10}

Se c è maggiore di 9, converto nella lettera appropriata

- ENDWHILE

Numeri esadecimali: da decimale a esadecimali (destra-sinistra)

PSEUDO-CODICE:

INPUT: n_{10}

OUTPUT: b_{16}

- $\text{ris} \leftarrow n_{10}$
- $k \leftarrow 1$
- $b_{16}(k) \leftarrow 0$
- WHILE $\text{ris} > 0$
 - $r \leftarrow \text{ris \% } 16$ Resto
 - $\text{ris} \leftarrow \text{ris / } 16$ Risultato
 - IF $r > 9$ THEN $r \leftarrow \text{convert}(r)$
 - $b_{16}(k) \leftarrow r$
 - $k \leftarrow k+1$
- ENDWHILE

Numeri binari: *un po' di nomenclatura*

- un gruppo di 8 bit è detto **byte**
- un gruppo di 4 bit è detto **nibble** (non è comune!) o metà di un byte:
 - un digit esadecimale necessita di metà byte (nibble), 2 digit esadecimali di un byte pieno.
- Microprocessori manipolano dati in “chunks” chiamati *parole*:
 - 64-bit
 - 32-bit (vecchi microprocessori)
 - 16-bit (semplici)

Numeri binari: un po' di nomenclatura

- 2^{10} bit = 1024 bits $\approx 10^3$ bit kilobit (Kb)
- 2^{20} bit = 1048576 bit $\approx 10^6$ bit megabit (Mb)
- 2^{30} bit $\approx 10^9$ bit gigabit (Gb)

Similmente

- 2^{10} byte = 1024 byte $\approx 10^3$ byte kilobyte (KB)
- 2^{20} byte = 1048576 byte $\approx 10^6$ byte megabyte (MB)
- 2^{30} byte $\approx 10^9$ byte gigabyte (GB)

- La capacità delle memorie sono usualmente misurate in byte
 - Ad esempio: 64 GB
- La velocità di comunicazione in bits/sec. Ad esempio: 56 kbit/sec.

Numeri binari: un po' di nomenclatura

msb: bit più significativi
(a sinistra)

101100
most significant bit least significant bit

lsb: bit meno significativi
(a destra)

MSB: byte più significativi
(a sinistra)

DEAFDAD8
most significant byte least significant byte

LSB: byte meno significativi
(a destra)

Esercizi

1. Quanti numeri differenti possono essere memorizzati con 16 bit?
2. Quale è il più grande numero intero positivo che posso memorizzare con 32 bit?
3. Converti i seguenti numeri binari a decimali ed esadecimali
 - a. 1010, 110110, 111100002, 110101112
4. Converti i seguenti numeri esadecimali a binari e decimali
 - a. A5, 3B, FFFF, 7C, ED3A
5. Converti i seguenti numeri decimali a binari ed esadecimali
 - a. 8, 15, 711, 845, 10825
6. **Scrivere una funzione in C per convertire numeri decimali in binari**
7. Scrivere una funzione in C per convertire numeri decimali in esadecimali

Architettura degli Elaboratori

Lezione 4

Docente: R.Prevete
a.a. 2022/2023
2023/03/15

Operazioni su numeri binari

Addizione tra numeri binari

- Ricordiamo cosa accade nel sistema decimale:

Riporto

Addendo 1

Addendo 2

Somma o
risultato

| | k (10^3) | h (10^2) | da (10^1) | u (10^0) | |
|----------------------|--------------|--------------|---------------|--------------|---|
| Riporto | 1 | | 1 | | |
| Addendo 1 | 2 | 9 | 7 | 8 | + |
| Addendo 2 | 3 | 5 | 1 | 6 | = |
| Somma o risultato | 6 | 4 | 9 | 4 | |

Addizione tra numeri binari

- Tra i numeri binari, in maniera analoga:

| | 2^6 | 2^5 | 2^4 | 2^3 | 2^2 | 2^1 | 2^0 | |
|-------------------|-------|-------|-------|-------|-------|-------|-------|---|
| Riporto | 1 | 1 | 1 | 1 | 1 | | | |
| Addendo 1 | | 1 | 0 | 1 | 1 | 1 | 0 | + |
| Addendo 2 | | 0 | 1 | 0 | 1 | 1 | 1 | = |
| Somma o risultato | 1 | 0 | 0 | 0 | 1 | 0 | 1 | |

Addizione tra numeri binari

Si hanno, cioè, i seguenti casi (**non c'è un precedente riporto**):

caso 0+0

| | | |
|-----------|-------|--|
| 2^{k+1} | 2^k | |
| | | |
| 0 | + | |
| 0 | = | |
| 0 | | |

caso 0+1

| | | | | | |
|-----------|-------|--|-----------|-------|--|
| 2^{k+1} | 2^k | | 2^{k+1} | 2^k | |
| | | | | | |
| 0 | + | | 1 | + | |
| 1 | = | | 0 | = | |
| 1 | | | 1 | | |

caso 1+0

| | | |
|-----------|-------|--|
| 2^{k+1} | 2^k | |
| | | |
| 1 | | |
| 1 | = | |
| 0 | | |

caso 1+1

Ho due elementi di peso 2^k , quindi $2 \cdot 2^k = 2^{k+1}$, cioè un solo elemento da 2^{k+1}

Addizione tra numeri binari

Si hanno, cioè, i seguenti casi (**c'è un precedente riporto**):

caso 1+0+0

| | | |
|-----------|-------|--|
| 2^{k+1} | 2^k | |
| | 1 | |
| 0 | + | |
| 0 | = | |
| 1 | | |

caso 1+0+1

| | | |
|-----------|-------|--|
| 2^{k+1} | 2^k | |
| 1 | 1 | |
| 0 | + | |
| 1 | = | |
| 0 | | |

caso 1+1+0

| | | |
|-----------|-------|--|
| 2^{k+1} | 2^k | |
| 1 | 1 | |
| 1 | + | |
| 0 | = | |
| 0 | | |

caso 1+1+1

| | | |
|-----------|---|--|
| 2^{k+1} | | |
| 2 k | | |
| 1 | 1 | |
| 1 | + | |
| 1 | = | |
| 1 | | |

Ho tre elementi di peso 2^k , quindi $3 \cdot 2^k = 2^k + 2^{k+1}$, cioè un solo elemento 2^k ed un solo elemento da 2^{k+1}

Addizione tra numeri binari: overflow

I sistemi digitali operano con un numero fissato di digit (bit). Se il risultato è troppo grande per essere memorizzato nei digit disponibile, si dice che l'addizione va in **overflow**. In questo caso ottengo un risultato non corretto!!

| | 2^4 | 2^3 | 2^2 | 2^1 | 2^0 | |
|-------------------|-------|-------|-------|-------|-------|---|
| Riporto | 1 | 1 | 1 | | | |
| Addendo 1 | | 1 | 1 | 1 | 0 | + |
| Addendo 2 | | 0 | 1 | 1 | 1 | = |
| Somma o risultato | | 0 | 1 | 0 | 1 | |

ESEMPIO:
ho solo 4 digit,
si possono
rappresentare
solo i numeri
interi da 0 a 15

Numeri interi negativi

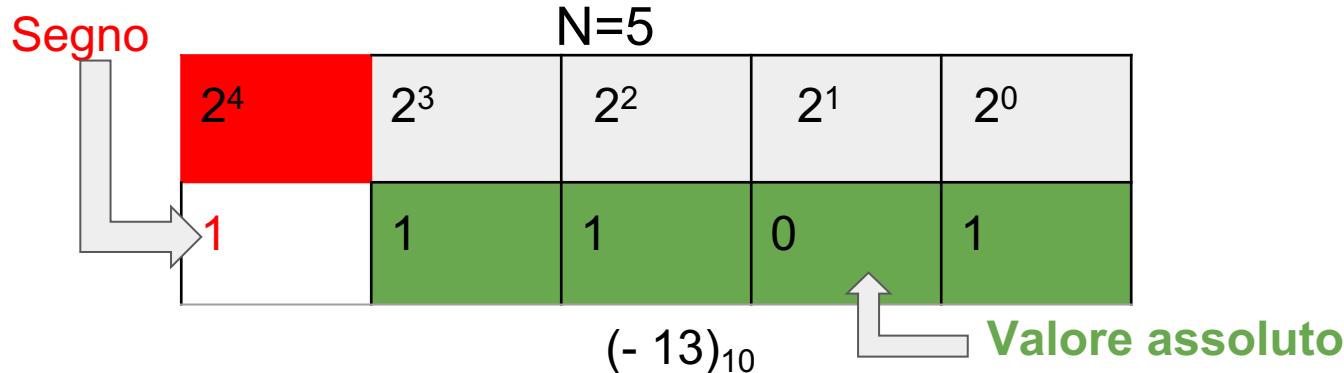
Due differenti modi per rappresentare i numeri interi con segno (positivi e negativi) in codifica binaria:

- **Segno più modulo**
- **Complemento a due**

Numeri interi negativi: segno più modulo

Codifica ad N bit:

- Il bit più significativo è usato per indicare il segno: 1 negativo, 0 positivo
- I rimanenti $N-1$ bit sono usati per codificare il valore assoluto del numero



Range:
 $[-2^{N-1}+1 , +2^{N-1}-1]$

Se $N=5$
Range:
 $[-15 , +15]$

Numeri interi negativi: segno più modulo

| N=5 | | | | |
|-------|-------|-------|-------|-------|
| 2^4 | 2^3 | 2^2 | 2^1 | 2^0 |
| 1 | 0 | 0 | 0 | 0 |

$(- 0)_{10}$

Attenzione!

Lo zero ammette due rappresentazioni distinte

| N=5 | | | | |
|-------|-------|-------|-------|-------|
| 2^4 | 2^3 | 2^2 | 2^1 | 2^0 |
| 0 | 0 | 0 | 0 | 0 |

$(+ 0)_{10}$

Numeri interi negativi: addizione

| 2^4 | 2^3 | 2^2 | 2^1 | 2^0 |
|-------------|-------|-------|-------|-------|
| $(-5)_{10}$ | 1 | 0 | 1 | 0 |
| | 1 | 0 | 1 | 1 |



| 2^4 | 2^3 | 2^2 | 2^1 | 2^0 |
|-------------|-------|-------|-------|-------|
| $(+5)_{10}$ | 0 | 0 | 1 | 0 |
| | 0 | 0 | 1 | 1 |



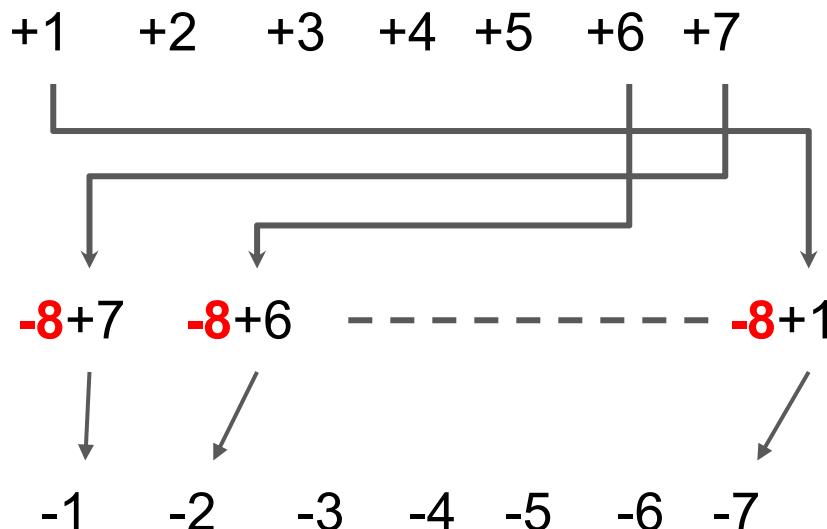
| 2^4 | 2^3 | 2^2 | 2^1 | 2^0 |
|--------------|-------|-------|-------|-------|
| $(-10)_{10}$ | 1 | 1 | 0 | 1 |
| | 1 | 1 | 0 | 0 |

Attenzione!

L'addizione così come vista non è corretta!!!

Numeri interi negativi: complemento a due

Idea



Numeri interi negativi: complemento a due

In una codifica binaria ad N bit, il bit più a sinistra viene interpretato, pesato come -2^{N-1}

| -2^4 | 2^3 | 2^2 | 2^1 | 2^0 |
|--------|-------|-------|-------|-------|
| 1 | 0 | 1 | 0 | 1 |

$$-1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = (-16 + 4 + 1)_{10} = -11_{10}$$

Range sempre: $[-2^{N-1}, +2^{N-1}-1]$

Numeri interi negativi complemento a 2: addizione

| 2^4 | 2^3 | 2^2 | 2^1 | 2^0 |
|-------------|-------|-------|-------|-------|
| $(-5)_{10}$ | | | | |
| 1 | 1 | 0 | 1 | 1 |



| 2^4 | 2^3 | 2^2 | 2^1 | 2^0 |
|-------------|-------|-------|-------|-------|
| $(+5)_{10}$ | | | | |
| 0 | 0 | 1 | 0 | 1 |

$$(-16+8+2+1) = -5$$

Attenzione!

L'addizione così come vista è corretta!!!

| 2^4 | 2^3 | 2^2 | 2^1 | 2^0 |
|------------|-------|-------|-------|-------|
| $(0)_{10}$ | | | | |
| 0 | 0 | 0 | 0 | 0 |

Numeri interi negativi complemento a 2: lo zero

| 2^4 | 2^3 | 2^2 | 2^1 | 2^0 |
|-------------------|-------|-------|-------|-------|
| (0) ₁₀ | 0 | 0 | 0 | 0 |

Unica rappresentazione per lo zero !!!

Numeri interi negativi complemento a 2

Se voglio controllare solo il segno, l'ultimo bit a sinistra mi permette ugualmente di immediatamente verificarlo

| 2^4 | 2^3 | 2^2 | 2^1 | 2^0 | $(-5)_{10}$ |
|-------|-------|-------|-------|-------|-------------|
| 1 | 1 | 0 | 1 | 1 | |

| 2^4 | 2^3 | 2^2 | 2^1 | 2^0 | $(+5)_{10}$ |
|-------|-------|-------|-------|-------|-------------|
| 0 | 0 | 1 | 0 | 1 | |

Architettura degli Elaboratori

Lezione 5

Docente: R.Prevete
a.a. 2022/2023
2023/03/17

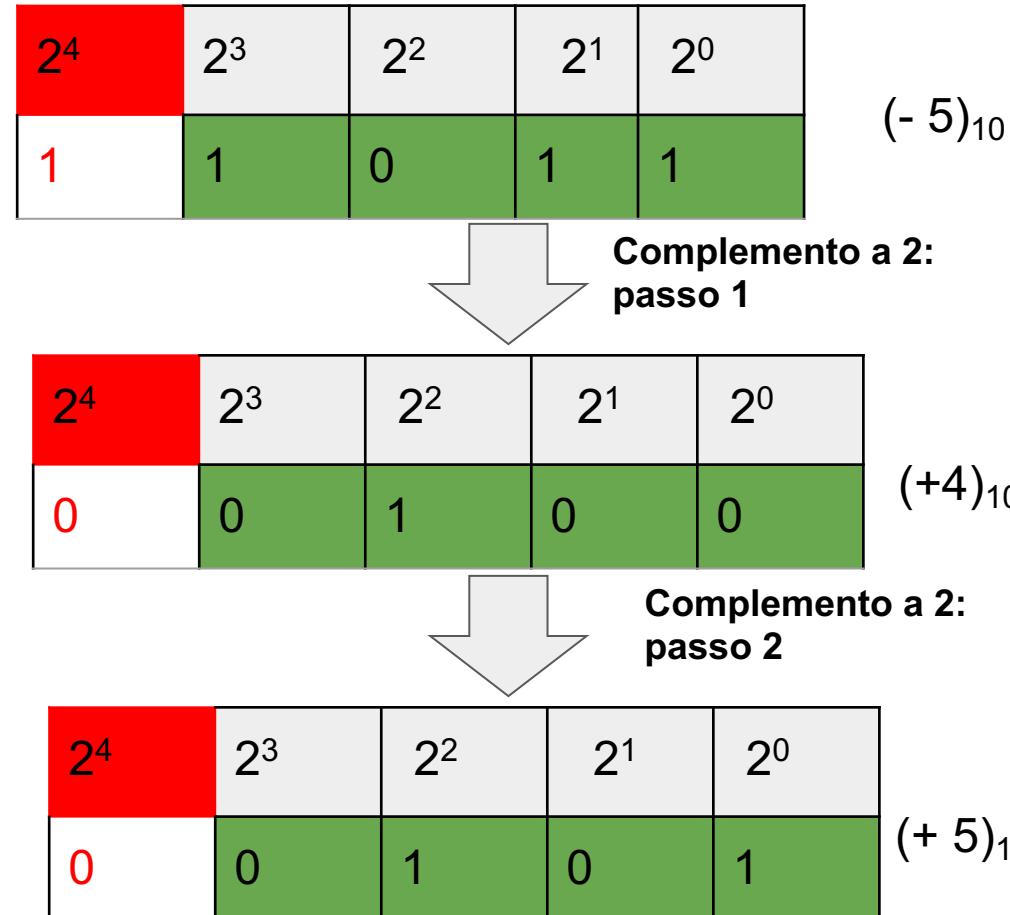
Operazioni su numeri binari

Numeri interi negativi: cambio di segno

Se voglio **cambiare il segno** di numero (nella rappresentazione a complemento a 2) si usa una procedura detta “complemento a 2 o complemento alla base”:

Complemento a 2:

1. Inverto il valore di tutti i digit
2. Sommo il valore +1



Numeri interi negativi complemento a 2

Complemento a 2:

1. Inverto il valore di tutti i digit
2. Sommo il valore +1

| 2^4 | 2^3 | 2^2 | 2^1 | 2^0 | $(+5)_{10}$ |
|-------|-------|-------|-------|-------|-------------|
| 0 | 0 | 1 | 0 | 1 | |

↓ Complemento a 2:
passo 1

| 2^4 | 2^3 | 2^2 | 2^1 | 2^0 | $(-6)_{10}$ |
|-------|-------|-------|-------|-------|-------------|
| 1 | 1 | 0 | 1 | 0 | |

↓ Complemento a 2:
passo 2

| 2^4 | 2^3 | 2^2 | 2^1 | 2^0 | $(-5)_{10}$ |
|-------|-------|-------|-------|-------|-------------|
| 1 | 1 | 0 | 1 | 1 | |

Altro esempio:

Complemento a 2: sottrazione (1)

Considero: $5_{10} - 3_{10}$

- Rappresento 5_{10} , 3_{10} come complemento a 2 (ad esempio con 5 bit)

| 2^4 | 2^3 | 2^2 | 2^1 | 2^0 | $(+5)_{10}$ |
|-------|-------|-------|-------|-------|-------------|
| 0 | 0 | 1 | 0 | 1 | |

| 2^4 | 2^3 | 2^2 | 2^1 | 2^0 | $(3)_{10}$ |
|-------|-------|-------|-------|-------|------------|
| 0 | 0 | 0 | 1 | 1 | |

Complemento a 2: sottrazione (1)

Considero: $5_{10} - 3_{10}$

- Cambio di segno 3_{10} , ottenendo -3_{10}

| 2^4 | 2^3 | 2^2 | 2^1 | 2^0 | $(+5)_{10}$ |
|-------|-------|-------|-------|-------|-------------|
| 0 | 0 | 1 | 0 | 1 | |

$(-3)_{10}$

+1

| 2^4 | 2^3 | 2^2 | 2^1 | 2^0 |
|-------|-------|-------|-------|-------|
| 1 | 1 | 1 | 0 | 1 |

| 2^4 | 2^3 | 2^2 | 2^1 | 2^0 | $(3)_{10}$ |
|-------|-------|-------|-------|-------|------------|
| 0 | 0 | 0 | 1 | 1 | |

Inverti

| 2^4 | 2^3 | 2^2 | 2^1 | 2^0 |
|-------|-------|-------|-------|-------|
| 1 | 1 | 1 | 0 | 0 |

$(-4)_{10}$

Complemento a 2: sottrazione (1)

Considero: $5_{10} - 3_{10}$

- Sommo 5_{10} e -3_{10}

| 2^4 | 2^3 | 2^2 | 2^1 | 2^0 | $(+5)_{10}$ |
|-------|-------|-------|-------|-------|-------------|
| 0 | 0 | 1 | 0 | 1 | |

+

| 2^4 | 2^3 | 2^2 | 2^1 | 2^0 | $(-3)_{10}$ |
|-------|-------|-------|-------|-------|-------------|
| 1 | 1 | 1 | 0 | 1 | |

=

| 2^4 | 2^3 | 2^2 | 2^1 | 2^0 | $(+2)_{10}$ |
|-------|-------|-------|-------|-------|-------------|
| 0 | 0 | 0 | 1 | 0 | |

Osservo: da decimale a complemento a 2

Considero: n_{10}

conversioneDecimaleComplemento2

INPUT: n_{10}

OUTPUT: b

1. IF $n_{10} > 0$
2. $b_2 \leftarrow \text{conversioneStandard}(n_{10})$
3. ELSE
4. $n_{10} \leftarrow -n_{10}$
5. $b_2 \leftarrow \text{conversioneStandard}(n_{10})$
6. $b_2 \leftarrow \text{inverttoDigit}(b_2)$
7. $b_2 \leftarrow \text{sommoUno}(b_2)$
8. ENDIF
9. output b_2

Complemento a 2: addizione con overflow

Considero: $4_{10} + 5_{10}$ usando 4 bit

| $(+4)_{10}$ | | | | $+$ | $(+5)_{10}$ | | | |
|-------------|-------|-------|-------|-----|-------------|-------|-------|-------|
| 2^3 | 2^2 | 2^1 | 2^0 | | 2^3 | 2^2 | 2^1 | 2^0 |
| 0 | 1 | 0 | 0 | | 0 | 1 | 0 | 1 |

$(-7)_{10}$

| 2^3 | 2^2 | 2^1 | 2^0 |
|-------|-------|-------|-------|
| 1 | 0 | 0 | 1 |

Overflow: Se 2 numeri dello stesso segno sono sommati ed il risultato ha **segno opposto**

Complemento a 2: estensione dei bit

Il valore del bit del segno deve essere copiato **nei bit più significativi** che vengono aggiunti

$(+5)_{10}$

| 2^3 | 2^2 | 2^1 | 2^0 |
|-------|-------|-------|-------|
| 0 | 1 | 0 | 1 |

Estensione da 4 a 6 bit

$(+5)_{10}$

| 2^5 | 2^4 | 2^3 | 2^2 | 2^1 | 2^0 |
|-------|-------|-------|-------|-------|-------|
| 0 | 0 | 0 | 1 | 0 | 1 |

$(-6)_{10}$

| 2^3 | 2^2 | 2^1 | 2^0 |
|-------|-------|-------|-------|
| 1 | 0 | 1 | 0 |

Estensione da 4 a 6 bit

$(-6)_{10}$

| 2^5 | 2^4 | 2^3 | 2^2 | 2^1 | 2^0 |
|-------|-------|-------|-------|-------|-------|
| 1 | 1 | 1 | 0 | 1 | 0 |

Complemento a 2: alcune note

| System | Range |
|------------------|-------------------------------|
| Unsigned | $[0, 2^N - 1]$ |
| Sign/Magnitude | $[-2^{N-1} + 1, 2^{N-1} - 1]$ |
| Two's Complement | $[-2^{N-1}, 2^{N-1} - 1]$ |

- Rappresentazioni: unsigned, two's complement, e sign/magnitude.
- Complemento a due conveniente:
 - Negativi e Positivi, addizione nello stesso modo
 - Sottrazione è realizzata invertendo i digit del secondo numero, aggiungendo 1 e poi facendo la normale addizione
 - Unica rappresentazione per lo zero

A meno che non diversamente specificato si userà la rappresentazione di complemento a 2

Alcuni esercizi

- Convertire i seguenti numeri in rappresentazione “complemento a 2” in decimali:
 - 1101_2 , 01101_2 , 11101_2 , 01101_2 , 100101_2
- Ripetere il precedente esercizio assumendo che la rappresentazione sia sign/magnitude
- Scrivere un programma che faccia la conversione da rappresentazione “complemento a 2” a decimale
- Estendere i seguenti numeri rappresentati come “complemento a 2” da 4 digit a 7 digit
 - 1101_2 , 0101_2 , 0111_2 , 1001_2 , 1100_2
- Eseguire le seguenti addizioni con numeri rappresentati come “complemento a 2” e verificare se c’è overflow
 - $1101_2 + 0101_2$, $0111_2 + 1001_2$, $0101_2 + 0100_2$, $1001_2 + 1010_2$

Altri esercizi da 1.30 a 1.60 , del libro di testo Harris & Harris

Architettura degli Elaboratori

Lezione 6

Docente: R.Prevete
a.a. 2022/2023
2023/03/20

Rappresentazione dei reali

Rappresentazione numeri decimali frazionari

| 10^0 | 10^{-1} | 10^{-2} | 10^{-3} |
|--------|-----------|-----------|-----------|
| 1, | 2 | 3 | 4 |

Peso

Rappresentazione decimale

| 2^0 | 2^{-1} | 2^{-2} | 2^{-3} |
|-------|----------|----------|----------|
| 1, | 1 | 0 | 1 |

Peso

Rappresentazione binaria

decimale corrispondente

$$1+1*0.5+0*0.25+1*0.125 = 1.625$$

Rappresentazione numeri decimali frazionari

Da decimale frazionario
a frazionario binario

| 10^0 | 10^{-1} | 10^{-2} | 10^{-3} |
|--------|-----------|-----------|-----------|
| 5 | 1 | 2 | 5 |

| 2^2 | 2^1 | 2^0 |
|-------|-------|-------|
| 1 | 0 | 1 |

| 2^{-1} | 2^{-2} | 2^{-3} |
|----------|----------|----------|
| 0 | 0 | 1 |

101,001

Come si ottiene??

Rappresentazione numeri decimali frazionari

Da decimale frazionario
a frazionario binario

| 10^0 | 10^{-1} | 10^{-2} | 10^{-3} |
|--------|-----------|-----------|-----------|
| 0, | 1 | 2 | 5 |



| | | | |
|-------|-------|-------|------|
| 0,125 | 0,250 | 0,500 | 1,00 |
| | 0 | 0 | 1 |

- Si procede dalla cifra più significativa a quella meno significativa (da sinistra a destra)
- Moltiplico sempre per 2
- Prendo la parte intera
- Se la parte intera è >0 , sottraggo 1
- Mi fermo quando la parte frazionaria è 0

Rappresentazione numeri decimali frazionari

Esempio: $(0,625)_{10}$ con 3 bit

0.625

Unità

Quante volte
«ci va» 2^{-1} in
 $0.625 \rightarrow$
 $0.625 / 2^{-1} =$
 $0.625 \times 2 = 1.250$

Cioè 1 volta e
mi resta da
rappresentare
0.125

$$0.5 = 2^{-1}$$

$$0.25 = 2^{-2}$$

$$0.125 = 2^{-3}$$

Rappresentazione numeri decimali frazionari

Esempio: $(0,625)_{10}$ con 3 bit

$$0.5 = 2^{-1}$$

Unità

Quante volte «ci va» 2^{-2} in 0.125
→
 $0.125 / 2^{-2} =$
 $0.125 \times 4 = 0.25 \times 2$
 $= 0.5$

Non «ci va»!

| 2^{-1} | 2^{-2} | 2^{-3} |
|----------|----------|----------|
| 1 | 0 | 0 |

$$0.25 = 2^{-2}$$

$$0.125 = 2^{-3}$$

Rappresentazione numeri decimali frazionari

Esempio: $(0,625)_{10}$ con 3 bit

$$0.5 = 2^{-1}$$

Unità

Quante volte «ci va» 2^{-3} in 0.125
→

$$0.125 / 2^{-3} =$$

$$0.125 \times 2 \times 2 \times 2 =$$

$$1.0$$

Esattamente
una volta

| 2^{-1} | 2^{-2} | 2^{-3} |
|----------|----------|----------|
| 1 | 0 | 0 |

$$0.25 = 2^{-2}$$

$$0.125 = 2^{-3}$$

Rappresentazione numeri decimali frazionari

Esempio: $(0,625)_{10}$ con 3 bit

$$0.5 = 2^{-1}$$

$$0.125 = 2^{-3}$$

Unità

| 2^{-1} | 2^{-2} | 2^{-3} |
|----------|----------|----------|
| 1 | 0 | 1 |

Rappresentazione numeri decimali frazionari

Quindi ricapitolando:

- Si procede dalla cifra più significativa a quella meno significativa (da sinistra a destra)
- Moltiplico sempre per 2
- Prendo la parte intera
- Se la parte intera è >0 , sottraggo 1
- Mi fermo quando la parte frazionaria è 0

| | |
|------|---|
| 0,75 | |
| 1,50 | 1 |
| 1,00 | 1 |
| 0 | |



Rappresentazione numeri decimali frazionari

Attenzione!

Se avessimo avuto ..

Parte frazionaria = 0,6

Allora, ripetendo il procedimento
precedente

Parte frazionaria binaria:

0,100110011001

o nell'usuale notazione
compatta:

$0.\overline{1001}$

| | |
|-----|---|
| 0,6 | |
| 1,2 | 1 |
| 0,4 | 0 |
| 0,8 | 0 |
| 1,6 | 1 |
| 1,2 | 1 |

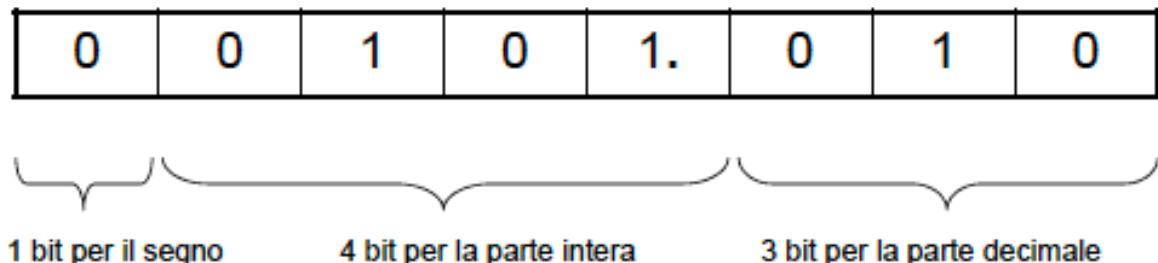
Se il risultato della
moltiplicazione per 2 si
ripete allora si ha un
numero periodico

Rappresentazione dei reali

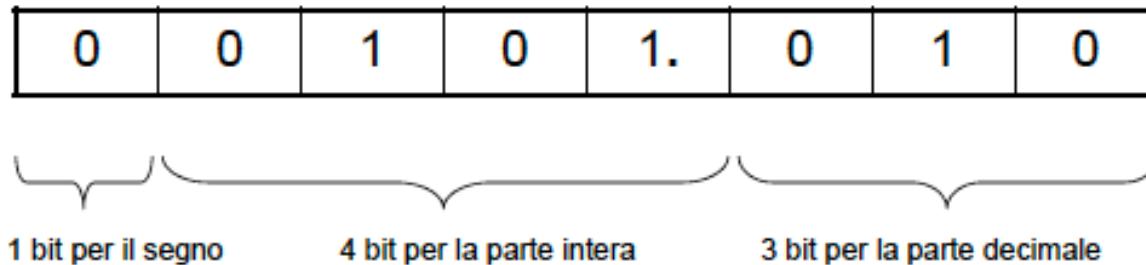
Stringa binaria di lunghezza fissata

Abbiamo limitazioni:

- nel range
- nella risoluzione
- Nella uniformità della rappresentazione



Rappresentazione dei reali

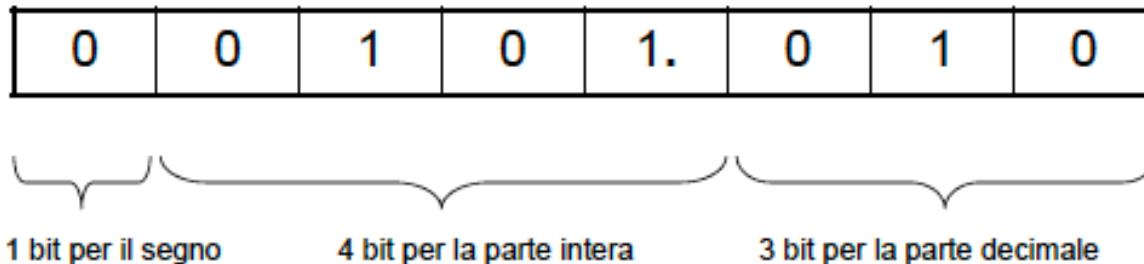


Virgola fissa

- **RANGE:** [-1111.111 ; -1111.111], cioè da -15.875 a +15.875.
- **Precisione** della rappresentazione: nel nostro caso è, evidentemente, determinata dal peso dell'ultimo bit: 2^{-3} , cioè 0.125.
- In altre parole la rappresentazione è un'approssimazione limitata e discreta del sistema dei Reali.

Rappresentazione dei reali

Virgola fissa



Ad esempio, il numero reale **5.26** che dista da quello rappresentato 5.25 meno di 0.125 non potrà avere una rappresentazione autonoma e sarà anch'esso rappresentato dalla stringa **00101.010**.

Il più piccolo dei numeri > 5.25 con una rappresentazione propria sarà, come è semplice intuire, **5.25+0.125**, cioè **5.575** :

| | | | | | | | |
|---|---|---|---|----|---|---|---|
| 0 | 0 | 1 | 0 | 1. | 0 | 1 | 1 |
|---|---|---|---|----|---|---|---|

Rappresentazione dei reali

Virgola fissa

| | | | | | | | |
|---|---|---|---|----|---|---|---|
| 0 | 0 | 1 | 0 | 1. | 0 | 1 | 1 |
|---|---|---|---|----|---|---|---|

- Ma se $0.125/5.26 \sim 2\%$ è una approssimazione relativa in qualche caso accettabile
- In altri casi, abbiamo errori relativi più significativi, ad esempio se vogliamo rappresentare 0.30, si avrebbe $.010 = (0.25)_{10}$.
- Errore relativo: $0.05/0.30 \sim 17\%!!$

Rappresentazione dei reali: rappresentazione in virgola mobile

- Per alleviare le difficoltà prima esposta si usa la tecnica della rappresentazione a **virgola mobile (floating point)**
- Ogni numero reale è posto nella forma: **M*B^E**
- **Esempi:**
- **31.4*10²** **M=31.4** **B=10 E=2**
- **0.314*10²** **M=0.314** **B=10 E=2**
- **2.7*10⁻⁴** **M=2.7** **B=10 E=-4**
- **1.723*2⁸** **M=1.723** **B=2 E=8**

Rappresentazione dei reali: rappresentazione in virgola mobile

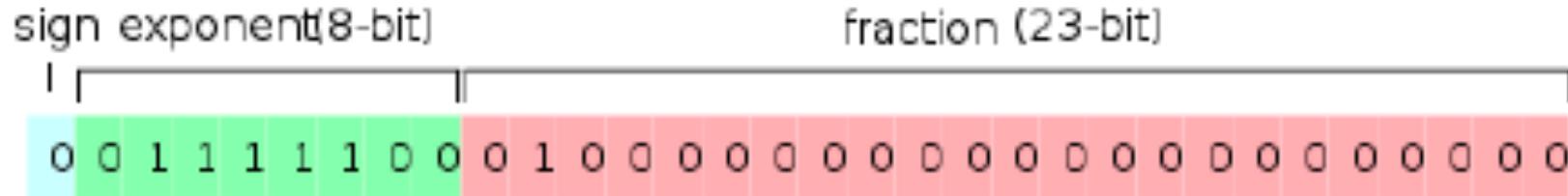
- Possiamo uniformare la rappresentazione (**forma Normalizzata**)
- Mantissa M: una sola cifra (>0) nella parte intera
- **Esempi in base 10:**
 - $31,4 \cdot 10^2 \rightarrow (31,4/10) \cdot 10^2 \cdot 10 = 3,14 \cdot 10^3$
 - $0,314 \cdot 10^2 \rightarrow (0,314 \cdot 10) \cdot 10^2 / 10 = 3,14 \cdot 10^1$
 - $322,7 \cdot 10^{-4} \rightarrow (322,7/100) \cdot 10^{-4} \cdot 100 = 3,227 \cdot 10^{-2}$

NOTA: Guardando solo l'esponente possiamo stabilire una relazione d'ordine completa

Rappresentazione in virgola mobile: Standard IEEE 754

$$X = (-1)^s \cdot M \cdot B^E$$

- Base B: 2
- Singola precisione (“float” del C). 32 bit per memorizzare:
1 bit segno s, 8 bit esponente E, 23 bit mantissa M

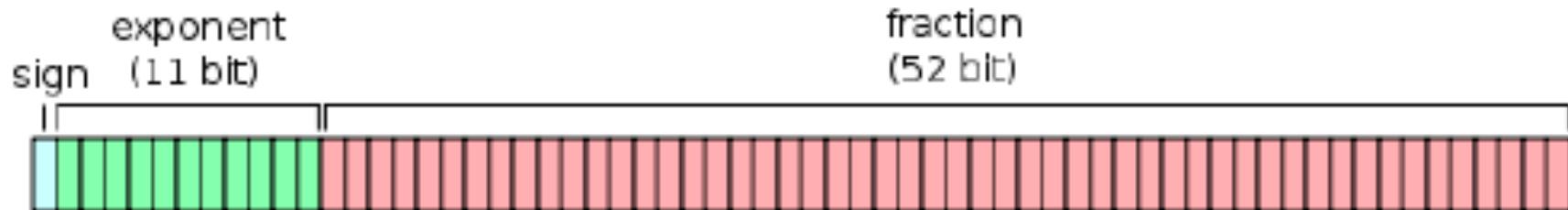


La mantissa è anche detta **frazione** o **significando**

Rappresentazione in virgola mobile: Standard IEEE 754

$$X = (-1)^s \cdot M \cdot B^E$$

- Base B: 2
- Doppia precisione (“double” del C). 64 bit per memorizzare: 1 bit segno s, 11 bit esponente E, 52 bit mantissa M



Rappresentazione in virgola mobile: Standard IEEE 754

$$X = (-1)^s \cdot M \cdot B^E$$

NOTA CHE:

- **Mantissa** o significando sono rappresentati in forma **normalizzata**:
 - si moltiplica o si divide la codifica binaria della mantissa per una certa potenza di 2. In tal modo rappresentazione della mantissa rimarrà con una sola cifra prima della virgola, cioè 1.
- Inoltre, dato che la cifra prima della virgola è sempre 1, **questa non viene rappresentata**, risparmiando così 1 bit.

$$1110001.01 = 1.11000101 \cdot 2^6 \rightarrow M = 11000101$$

$$\rightarrow X = (-1)^s \cdot (1+M) \cdot B^E$$

Architettura degli Elaboratori

Lezione 7

Docente: R.Prevete
a.a. 2022/2023
2023/03/22

Rappresentazione in virgola mobile: Standard IEEE 754

$$X = (-1)^s \cdot M \cdot B^E$$

NOTA CHE:

- **Mantissa** o significando sono rappresentati in forma **normalizzata**:
 - si moltiplica o si divide la codifica binaria della mantissa per una certa potenza di 2. In tal modo rappresentazione della mantissa rimarrà con una sola cifra prima della virgola, cioè 1.
- Inoltre, dato che la cifra prima della virgola è sempre 1, **questa non viene rappresentata**, risparmiando così 1 bit.

$$1110001.01 = 1.11000101 \cdot 2^6 \rightarrow M = 11000101$$

$$\rightarrow X = (-1)^s \cdot (1+M) \cdot B^E$$

Rappresentazione in virgola mobile: Standard IEEE 754

$$X = (-1)^S \cdot M \cdot B^E$$

NOTA CHE:

-- **Esponente E:**

- Rappresentazione con bias (o Polarizzata)
- Valori positivi e negativi
- BIAS = $2^{(n-1)} - 1$ se ho n bit. n=8 BIAS = 127, n=11 BIAS=1023
- RANGE = $[-(2^{(n-1)} - 1), 2^{(n-1)}]$

-- **ATTENZIONE:** $2^{(n-1)} - 1$ e $2^{(n-1)}$ sono valori riservati, quindi:

- RANGE = $[-(2^{(n-1)} - 2), 2^{(n-1)} - 1]$

Rappresentazione in virgola mobile: Standard IEEE 754

$$X = (-1)^{S^*} (M+1)^* B^E$$

Esempio: Calcolo della mantissa (o significando) e dell'esponente a singola precisione (M=23 bit ed E=8 bit) per il numero $(-113.25)_{10}$

-- Si calcola la codifica in base 2 del valore assoluto del numero:

$$113.25_{10} = 1110001.01_2$$

-- Per normalizzare, spostiamo la virgola di 6 posti :

$$1110001.01 = 1.11000101 \times 2^{+6}$$

-- La cifra alla sinistra delle virgola non si rappresenta, sarà:

$$M = 110001010000000000000000$$

2^{-12} 2^{-23}

Rappresentazione in virgola mobile: Standard IEEE 754

$$X = (-1)^s \cdot (M+1) \cdot B^E$$

$$M = 1100010100000000000000000$$

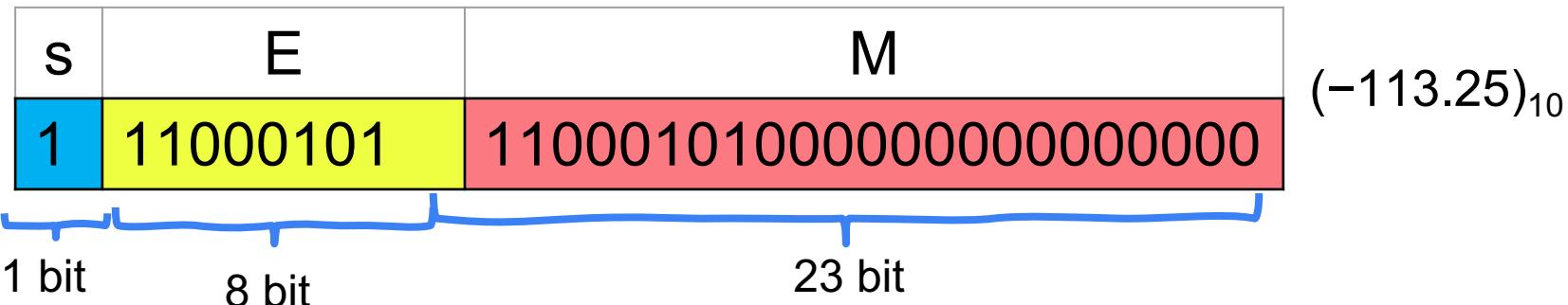
Passiamo al calcolo dell'esponente $E=+6$

-- Si aggiunge il bias $2^{n-1}-1=127$ ($n=8$ è il numero di bit in singola precisione):
 $E=127+6=133$

-- Numero binario corrispondente: $E=10000101$

-- Quindi otteniamo:

$$s=1, E=01000101 M=1100010100000000000000000$$



Rappresentazione in virgola mobile: Standard IEEE 754

$$X = (-1)^s * (M+1)^* B^E$$

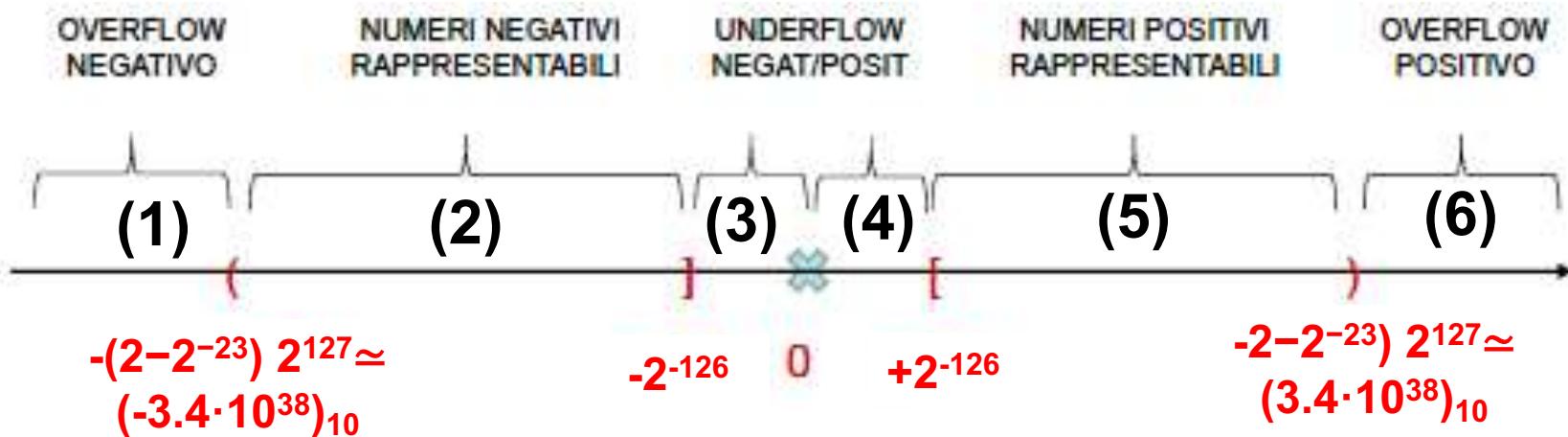
RICORDO:

- Per l'**esponente E** con 8 bit (single precision) o 11 bit (double precision), in teoria si avrebbero 256 combinazioni (single) o 2048 (double).
- **In pratica**: i valori 0 e 255 (o 2048 in double precision) sono riservati per **usi speciali** (vedremo nelle prossime slide)
Si rappresentano $2^n - 2$ valori, da $-(2^{(n-1)} - 2)$ a $+ 2^{(n-1)} - 1$

Rappresentazione in virgola mobile: Standard IEEE 754

COSA POSSIAMO RAPPRESENTARE:

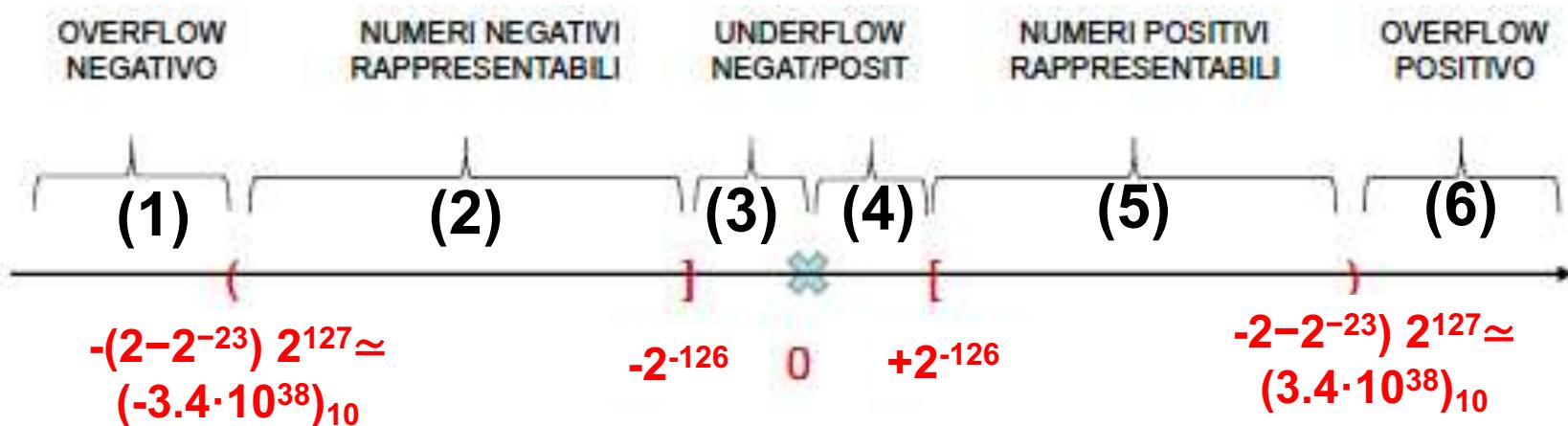
- **Intervallo (2)** e **(5)**: presenti i numeri che possiamo rappresentare:
 - costituiscono un numero finito di elementi (sottoinsieme di \mathbb{R}).
- **Intervallo 1 e 6**: non rappresentabili perché troppo grandi in valore assoluto
- **Intervallo 3 e 4**: non rappresentabili perché troppo piccoli in valore assoluto



Rappresentazione in virgola mobile: Standard IEEE 754

Alcuni casi particolari:

- **OVERFLOW**: tutti 1 all'esponente (codifica riservata) e M =0
- **Not a Number (NaN)**: casi indeterminati (0/0), tutti 1 all'esponente (codifica riservata) e M !=0
- **UNDERFLOW**: tutti 0 all'esponente (codifica riservata) e mantissa !=0



Rappresentazione in virgola mobile: Standard IEEE 754

Esempi di numeri "notevoli":

0 → 0 0000000 00000000000000000000000000000000

1 → 0 01111111 00000000000000000000000000000000

minore rappresentabile (2^{-126}) → 0 00000001 00000000000000000000000000000000

maggior rappres. ($2 - 2^{-23} \cdot 2^{127}$) → 0 11111110 1111111111111111111111111111111

∞ → 0 11111111 00000000000000000000000000000000

NaN → 0 11111111 XXXXXXXXXXXXXXXXXXXXXXXX

s

e

M

$$X = (-1)^s * (M+1) * B^E$$

Rappresentazione in virgola mobile: Standard IEEE 754

Altro esempio: $(-0.75)_{10}$

-- rappresentazione binaria di 0.75: 0.11

- forma normalizzata: 1.1 2⁻¹

-1 in forma polarizzata: 126

quindi: 01111110

-- Rappresentazione Standard IEEE 754

1 01111110 10000000000000000000000000000000

$$X = (-1)^s * (M+1)^B * E$$

Rappresentazione in virgola mobile: Standard IEEE 754

Altro esempio: $(0.6)_{10}$

-- rappresentazione binaria di 0.6:

0.1001100110011001100110011001...periodico

- forma normalizzata:

1.001100110011001100110011001... * 2^{-1}

- ma M ha solo 23 cifre:

M=00110011001100110011001

--1 in forma polarizzata: 00111110

-- Rappresentazione Standard IEEE 754

1 00111110 00110011001100110011001

C'è una approssimazione!!! 0.6 non esattamente rappresentabile

Standard IEEE 754. Precisione di macchina (o ϵ - macchina):

Differenza tra +1 ed il più piccolo numero rappresentabile maggiore di 1 (in genere indicato con $1+$)

Da una valore per eccesso dell'errore di arrotondamento

-- In single precision

$$1 = (0\ 0111111\ 000\ 0000\ 0000\ 0000\ 0000\ 0000)_{\text{IEEE-754-SP}}$$

$$= (1.000\ 0000\ 0000\ 0000\ 0000\ 0000)_2$$
$$= (1)_{10}$$

$$\begin{array}{|c|c|} \hline \text{esponente=0} & \text{sottointeso 1.} \\ \hline \end{array}$$

$$1+ = (0\ 0111111\ 000\ 0000\ 0000\ 0000\ 0001)_{\text{IEEE-754-SP}}$$

$$= (1.000\ 0000\ 0000\ 0000\ 0000\ 0001)_2$$
$$\approx (1.000000119209289\dots)_{10}$$

$$\begin{array}{|c|c|} \hline \text{esponente= -23} & \text{sottointeso 1.} \\ \hline \end{array}$$

$$\epsilon_{\text{mach}} = (0\ 01101000\ 000\ 0000\ 0000\ 0000\ 0000)_{\text{IEEE-754-SP}}$$

$$= (0.000\ 0000\ 0000\ 0000\ 0000\ 0001)_2$$

$$= (2^{-23})_{10} \approx (0.000000119209289\dots)_{10} \approx (1.19209289\dots \cdot 10^{-7})_{10}$$

Standard IEEE 754. Precisione di macchina (o ϵ - macchina):

In generale

$e_{mach} = b^{-(p-1)} = b^{1-p}$ (es. nel caso IEEE-754 single-precision $p=24$)

-- $p-1$ è il numero di cifre dopo la virgola

-- b è la base

Riassumendo...

- L'aritmetica del calcolatore è limitata da precisione finita
 - Gli interi possono essere rappresentati esattamente in un dato intervallo
 - I reali possono essere rappresentati in un dato intervalli, ma non tutti esattamente
 - Per i reali esiste uno standard IEEE 754
 - Per i reali ci può essere un errore di approssimazione
- I pattern di bit da soli non hanno un particolare significato
 - L'interpretazione determina il loro significato

Architettura degli Elaboratori

Lezione 8

Docente: R.Prevete
a.a. 2022/2023
2023/03/24

Algebra di Boole

Un nuovo concetto: Porte logiche (1)

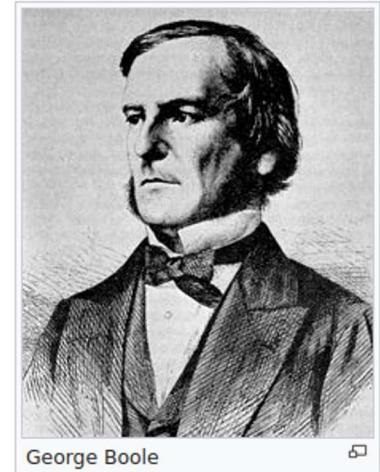
- Abbiamo visto come rappresentare informazioni in un sistema digitale, in particolare numeri interi
- Vedremo ora come un sistema digitale può realizzare operazioni su tali rappresentazioni
- In particolare vedremo come un semplice sistema digitale può prendere una o più variabili di input rappresentanti numeri binari e dare in output un numero binario.

Un nuovo concetto: Porte logiche (2)

- Le porte logiche (logic gate) realizzano operazioni logiche considerando:
 - 1 come valore di Verità (TRUE)
 - 0 come valore di Falsità (FALSE)
- Realizzano, in particolare, espressioni della logica Booleana

Porte logiche

- Come detto, le porte logiche (logic gates) sono una implementazione di espressioni/funzioni della logica Booleana
- Cosa è la logica (o algebra) Booleana (o di Boole)?
 - Fu introdotta nel 1847 e completata nel 1854 da George Boole
 - The Mathematical Analysis of Logic (Boole, George. *The mathematical analysis of logic.* Philosophical Library, 1847.)



Algebra di Boole

- E' una particolare algebra (o logica) le cui variabili e funzioni possono assumere solo due valori: 1 (True) e 0 (False)
 - Quindi il loro dominio è l'insieme $\{0,1\}$
- Le variabili in tale logica sono comunemente espresse con lettere maiuscole: A,B, C, ..., Y, Z
- Le funzioni sono un mapping funzionale dal dominio $\{0,1\}^N$ al codominio $\{0,1\}$:
 - $f: \{0,1\}^N \rightarrow \{0,1\}$
 - N rappresenta il numero di variabili
- Esempio: $Y=f(A,B)$, $Y=g(A)$, $Z=f(A,B,C)$

Algebra di Boole (2)

- In genere le prime lettere dell'alfabeto sono usate come variabili di input (A,B,C) , le ultime come variabili di output (Y,Z)
- Dato che una funzione Booleana di N variabili ha 2^N possibili input differenti e corrispondenti output, possiamo esprimerla tramite una tabella di coppie input-output, detta **tabella di verità**:

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 1 |

Algebra di Boole (3)

- Ci sono tre funzioni principali:
 - **AND**, indicato anche con \wedge
 - **OR**, indicato anche con \vee
 - **NOT**, indicato anche con \neg

$Y=AND(A,B)$

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

$Y=OR(A,B)$

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 1 |

$Y=NOT(A)$

| A | Y |
|---|---|
| 0 | 1 |
| 1 | 0 |

Algebra di Boole: AND

$Y=AND(A,B)$

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

Oltre a AND e \wedge , tale funzione logica può essere espressa con i simboli \cap , \cdot , & oppure omettendo ogni simbolo.

- $Y = AND(A,B)$
- $Y = A \wedge B$
- $Y = A \cap B$
- $Y = A \cdot B$
- $Y = A \& B$
- $Y = AB$

Sono tutti modi equivalenti di scrivere la stessa funzione logica

Algebra di Boole: OR

$Y=OR(A,B)$

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 1 |

Oltre a OR e V , tale funzione logica può essere espressa con i simboli U, +, |.

- $Y= OR(A,B)$
- $Y= A \cup B$
- $Y= A + B$
- $Y= A | B$

Sono tutti modi equivalenti di scrivere la stessa funzione logica

Algebra di Boole: NOT

$Y = \text{NOT}(A)$

| A | Y |
|---|---|
| 0 | 1 |
| 1 | 0 |

Oltre a NOT e \neg , tale funzione logica può essere espressa con i simboli $!$, \sim e con un trattino sopra la variabile.

- $Y = \text{NOT}(A)$
- $Y = !A$
- $Y = \sim A$
- $Y = \bar{A}$

Sono tutti modi equivalenti di scrivere la stessa funzione logica

Architettura degli Elaboratori

Lezione 9

Docente: R.Prevete
a.a. 2022/2023
27 marzo 2023

Algebra di Boole: Composizione

- Possiamo comporre insieme funzioni Booleane per ottenere nuove funzioni:
 - $Y = \text{NUOVA}(A, B, C) = \text{OR}(\text{AND}(A, B), C)$

Z=AND(A,B)

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

Y=OR(Z,C)

| Z | C | Y |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 1 |

Y=NUOVA(A,B,C)

| A | B | C | Y |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

Algebra di Boole: Composizione

- ALTRI ESEMPI:
 - $Y = F(A,B,C) = \text{OR}(\text{AND}(A,B), \text{NOT}(C))$
 - $Y = F(A,B) = \text{OR}(\text{AND}(A, B), \text{NOT}(B))$
 - $Y = F(A,B,C) = \text{AND}(\text{OR}(A,B), C)$

Provare a calcolare tavelle di verità!!

Algebra di Boole: AND, OR e NOT sono sufficienti

- Tramite l'operazione di composizione possiamo realizzare qualunque funzione booleana usando solamente AND, OR e NOT

Utile da ricordare: valgono la proprietà associativa e commutativa per AND e OR

- (COMMUTATIVA)
 - $A \wedge B = B \wedge A$
 - $A \vee B = B \vee A$
- (ASSOCIAUTIVA)
 - $(A \wedge B) \wedge C = A \wedge (B \wedge C)$
 - $(A \vee B) \vee C = A \vee (B \vee C)$

AND, OR E NOT COME SISTEMI DIGITALI

$Y=AND(A,B)$

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

AND



AND gate

$Y=OR(A,B)$

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 1 |

OR

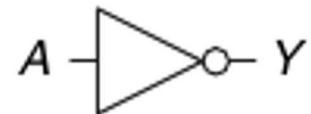


OR gate

$Y=NOT(A)$

| A | Y |
|---|---|
| 0 | 1 |
| 1 | 0 |

NOT



NOT gate

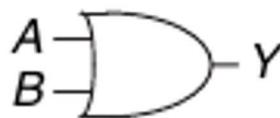
AND, OR E NOT COME SISTEMI DIGITALI

AND



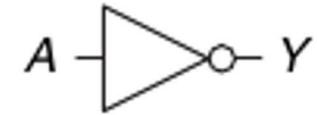
AND gate

OR



OR gate

NOT



NOT gate

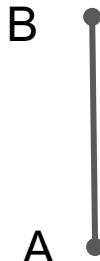
In che modo è possibile utilizzare fenomeni fisici continui (**sistema analogico**) per rappresentare variabili binarie e funzioni logiche?

- Esempi di quantità fisiche continue:
 - La differenza di potenziale (ddp) o differenza di *voltaggio* tra due punti
 - Il livello di un fluido in un recipiente
 - La posizione di una rotella
- Quindi bisogna trovare un modo per “legare” tali valori continui a valori discreti

Mapping tra DDP (o Voltaggio) e valori discreti

Dato un sistema elettrico, si considera:

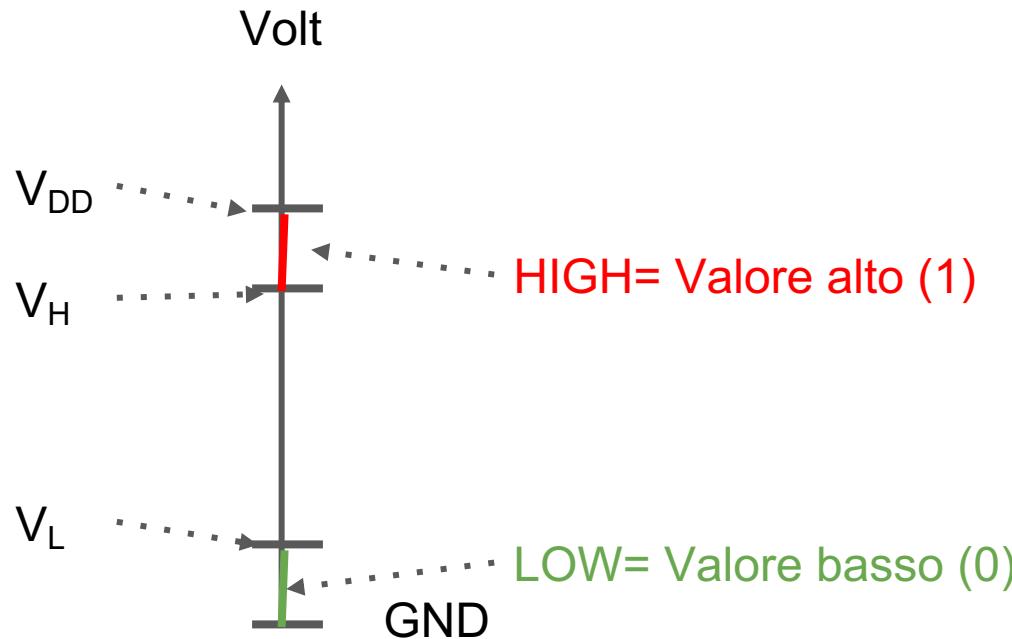
- Il minimo potenziale elettrico presente nel sistema: **ground (GND, terra)**
 - Tale minimo potenziale viene fatto corrispondere a 0 V (Volt). Ricordo che un potenziale è sempre definito a meno di una costante)
- La massima differenza di potenziale: generalmente chiamato **V_{DD}**
 - Valori tipici di V_{DD} sono: 5.0 V, 3.3 V, 2.5 V, 1.8 V, 1.5 V, 1.2 V



- Minima differenza di potenziale tra B e A: 0
- Massima differenza di potenziale tra B e A: V_{DD}

Mapping tra DDP (o Voltaggio) e valori discreti

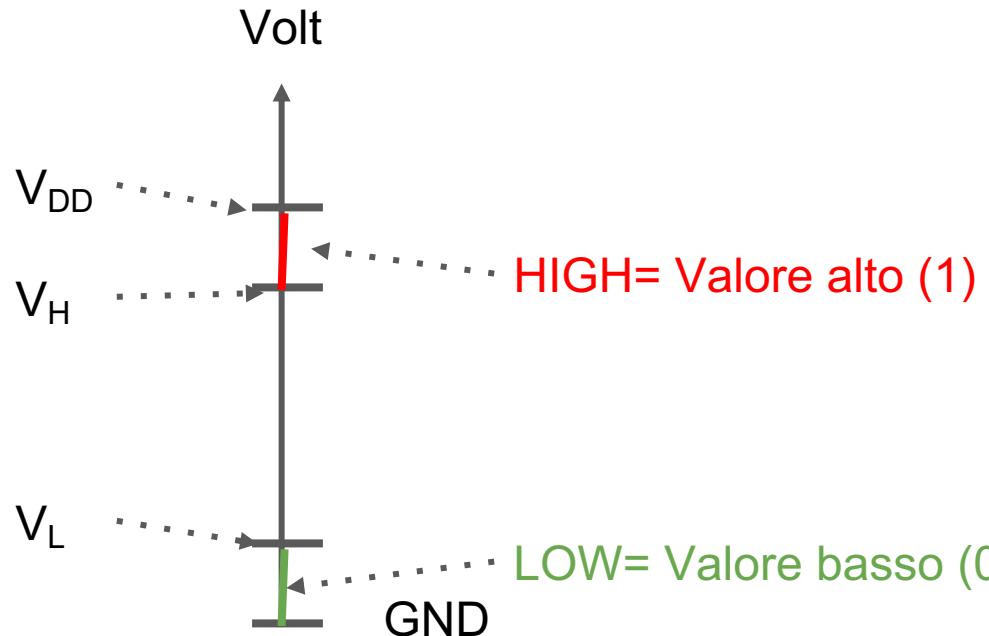
Allora il mapping tra ddp e valori discreti è realizzato definendo dei valori logici:



- L'intervallo tra V_H e V_{DD} è considerato HIGH (alto,1)
- L'intervallo tra GND e V_L è considerato LOW (basso,0)

Mapping tra DDP (o Voltaggio) e valori discreti

Allora il mapping tra ddp e valori discreti è realizzato definendo dei valori logici:

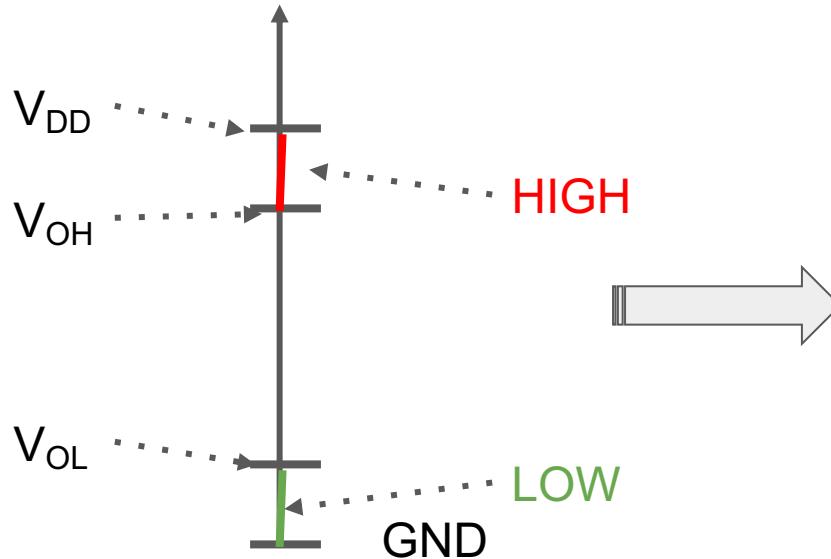


- Quindi posso considerare questa suddivisione per mappare la ddp in un sistema elettrico a valori logici. In questo modo ottengo un sistema digitale.
- **ATTENZIONE: Se ci sono due sistemi digitali che comunicano tra di loro ci può essere del rumore!**
- **Voglio che l'interazione sia tollerante al rumore!**

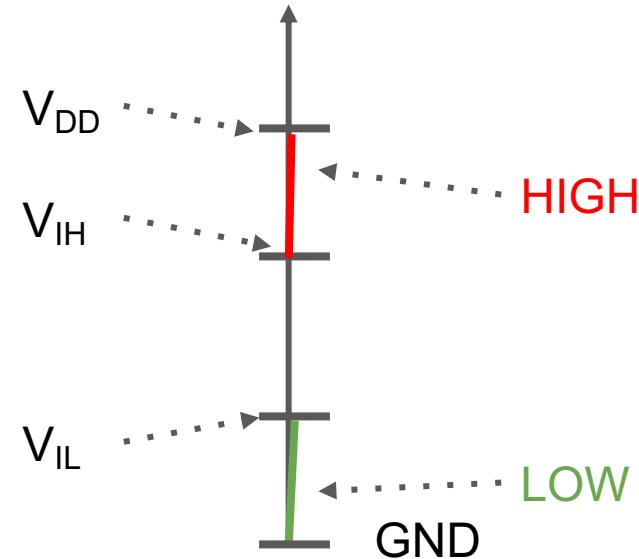
Mapping tra DDP (o Voltaggio) e valori discreti

Due sistemi digitali che interagiscono:

Output del sistema che invia
(Driver)



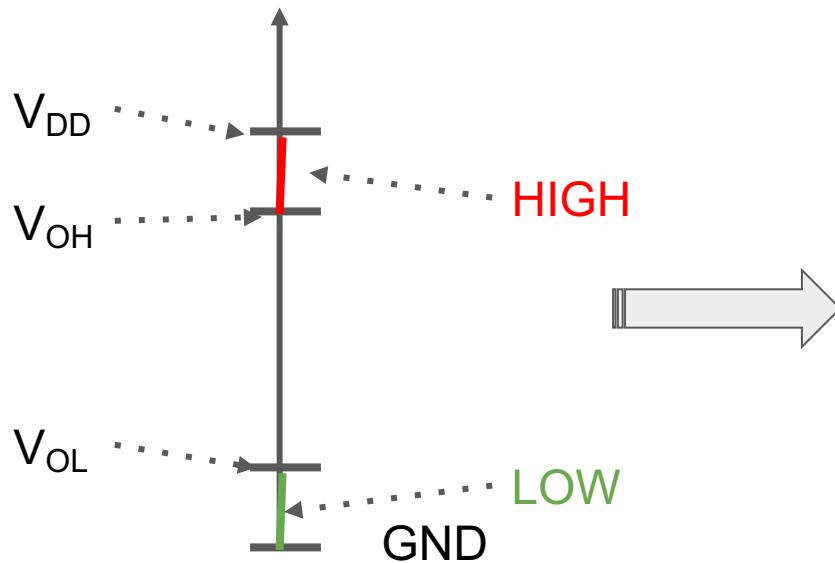
Input del sistema che riceve
(Receiver)



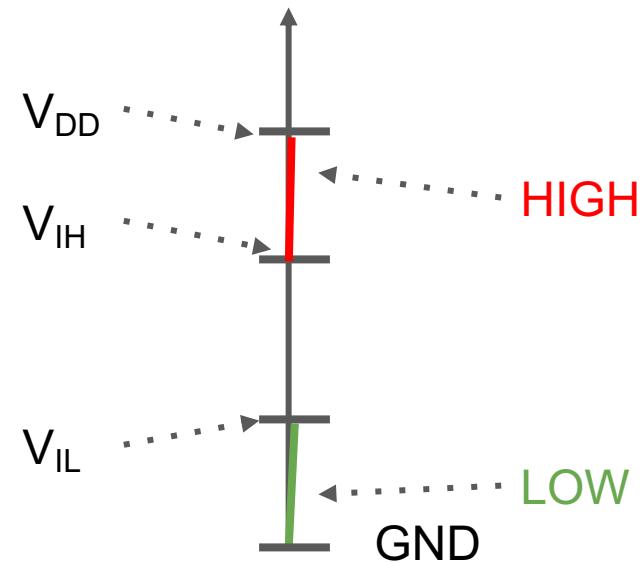
Mapping tra DDP (o Voltaggio) e valori discreti

- $V_{OH} > V_{IH}$ e $V_{IL} > V_{OL}$ permettono al receiver di essere tollerante al rumore
- $NM_H = V_{OH} - V_{IH}$ e $NM_L = V_{IL} - V_{OL}$ sono detti *noise margin*

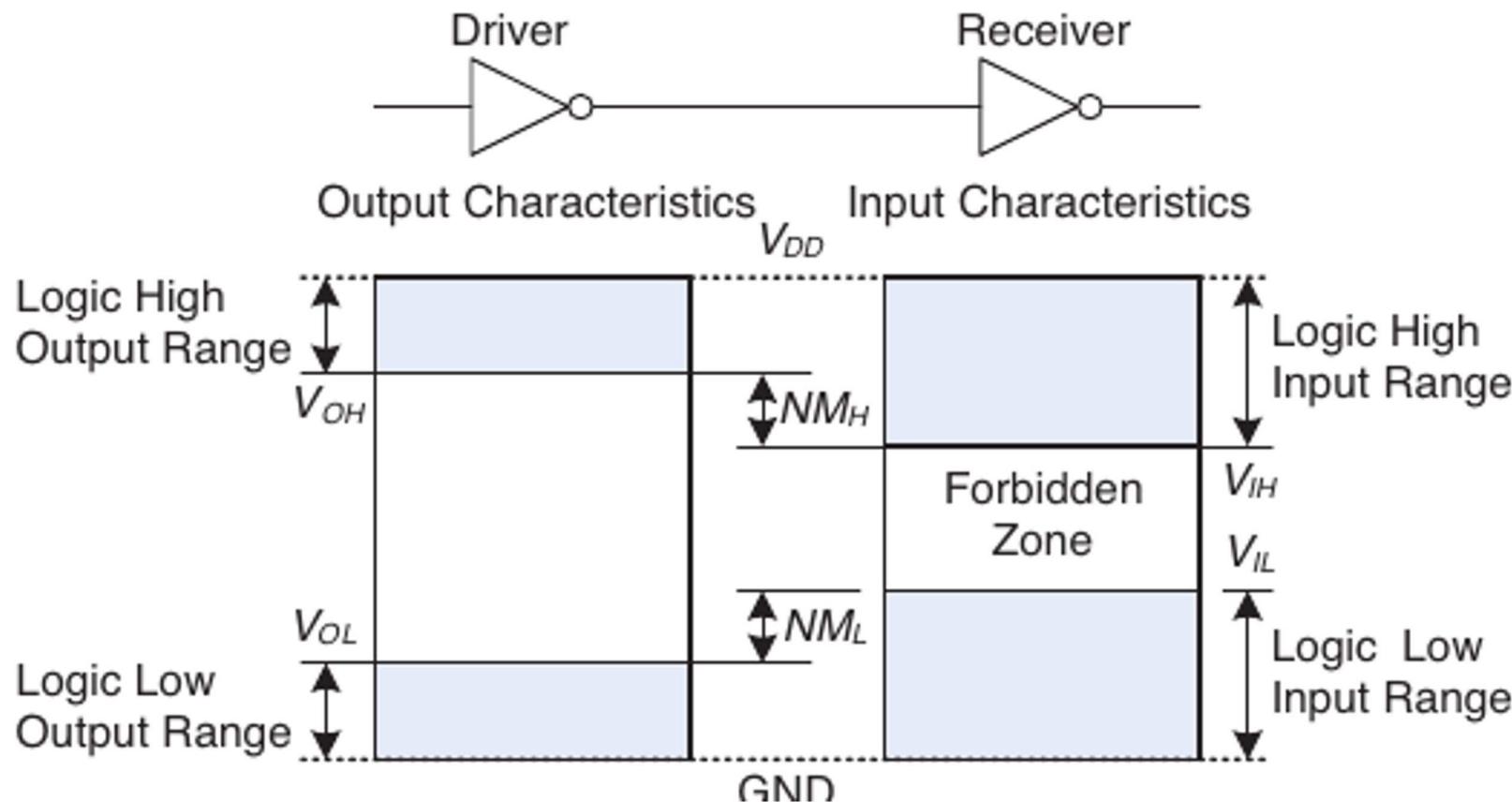
Output del sistema che invia
(Driver)



Input del sistema che riceve
(Receiver)



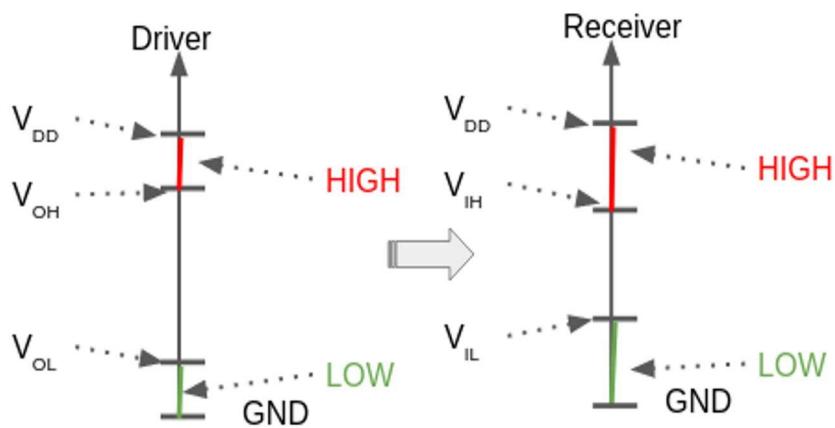
Mapping tra DDP (o Voltaggio) e valori discreti



Mapping tra DDP (o Voltaggio) e valori discreti

Facciamo un esempio:

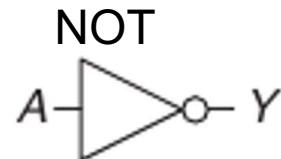
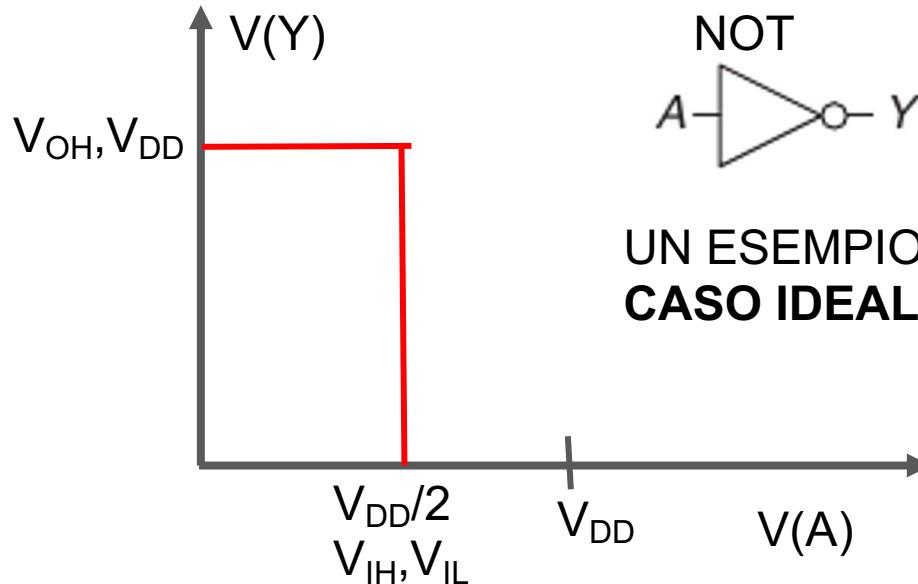
$$V_{DD} = 5 \text{ V}, V_{IL} = 1.35 \text{ V}, V_{IH} = 3.15 \text{ V}, V_{OL} = 0.33 \text{ V}, \text{ and } V_{OH} = 3.84 \text{ V}$$



- Per i valori alti ho una tolleranza (NM) di:
 - $3.84 - 3.15 = 0.69 \text{ V}$
- Per i valori bassi ho una tolleranza (NM) di:
 - $1.35 - 0.33 = 1.02 \text{ V}$
- Quindi per valori bassi il sistema ha una tolleranza maggiore (più di 1V), mentre per valori alti la tolleranza al rumore è meno di 1V

Funzione di trasferimento (transfer characteristics)

- Quando astraiamo un sistema analogico in un sistema digitale un altro elemento chiave è la funzione di trasferimento (o transfer characteristics)
- Tale funzione mette in relazione input con output del sistema

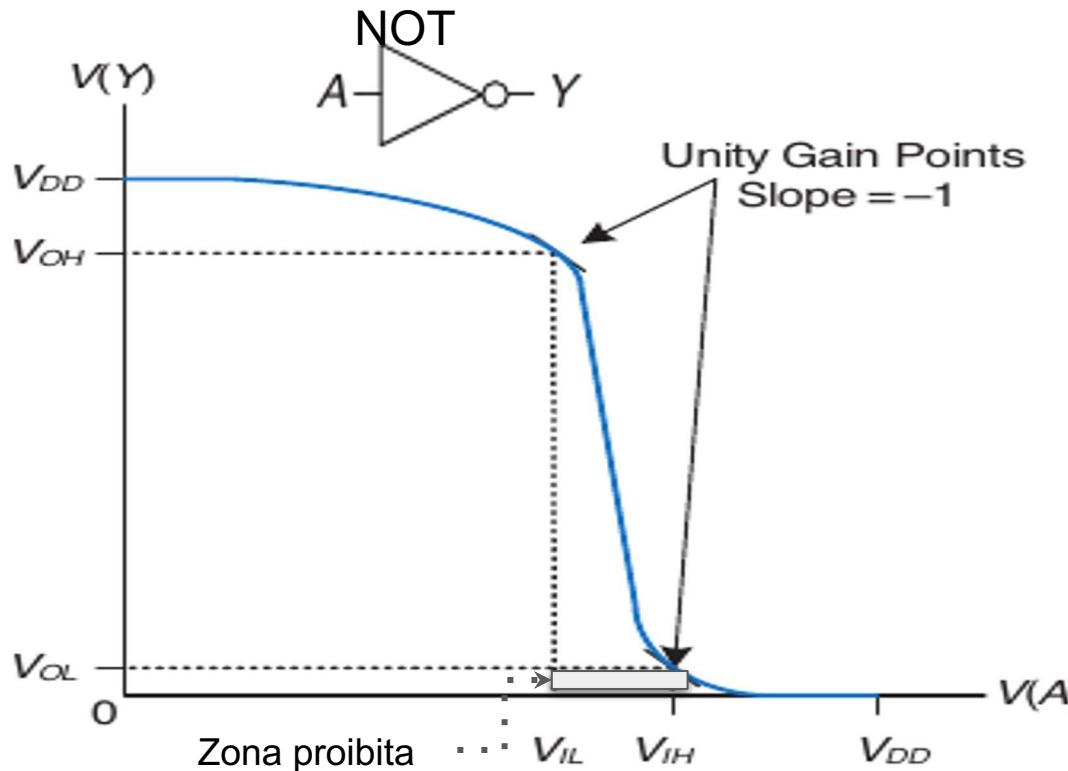


UN ESEMPIO:
CASO IDEALE!

- $V_{OH}=V_{DD}$
- $V_{OL}=0$
- $V_{IH}=V_{IL}=V_{DD}/2$

Funzione di trasferimento (transfer characteristics)

- Caso reale (un segnale analogico non cambia mai bruscamente!!)



- I livelli logici **non** sono scelti arbitrariamente
- Il “posto” più ragionevole dove scegliere i livelli logici è dove la pendenza($dV(Y) / dV(A)$) è -1 (cioè dove c’è un cambiamento)
- Si ottengono, così, due punti detti: **unity gain points**.

Hardware e scelta dei livelli logici

- Come sono realizzate tali porte logiche?
- Si hanno fondamentalmente 4 famiglie differenti a seconda dei materiali usati e della scelta dei livelli logici:
 - **Transistor-Transistor Logic (TTL),**
 - **Complementary Metal-Oxide-Semiconductor Logic (CMOS, pronunciato sea-moss),**
 - **Low Voltage TTL Logic (LV-TTL),**
 - **Low Voltage CMOS Logic (LVC-MOS).**

Hardware e scelta dei livelli logici

Table 1.4 Logic levels of 5 V and 3.3 V logic families

| Logic Family | V_{DD} | V_{IL} | V_{IH} | V_{OL} | V_{OH} |
|--------------|---------------|----------|----------|----------|----------|
| TTL | 5 (4.75–5.25) | 0.8 | 2.0 | 0.4 | 2.4 |
| CMOS | 5 (4.5–6) | 1.35 | 3.15 | 0.33 | 3.84 |
| LVTTL | 3.3 (3–3.6) | 0.8 | 2.0 | 0.4 | 2.4 |
| LVCMOS | 3.3 (3–3.6) | 0.9 | 1.8 | 0.36 | 2.7 |

Hardware e scelta dei livelli logici

Table 1.5 Compatibility of logic families

| | | Receiver | | | |
|--------|--------|----------|-----------------------|--------------------|--------------------|
| | | TTL | CMOS | LVTTL | LVCMOS |
| Driver | TTL | OK | NO: $V_{OH} < V_{IH}$ | MAYBE ^a | MAYBE ^a |
| | CMOS | OK | OK | MAYBE ^a | MAYBE ^a |
| | LVTTL | OK | NO: $V_{OH} < V_{IH}$ | OK | OK |
| | LVCMOS | OK | NO: $V_{OH} < V_{IH}$ | OK | OK |

^a As long as a 5 V HIGH level does not damage the receiver input.

Architettura degli Elaboratori

Lezione 10

Docente: R.Prevete
a.a. 2022/2023
29 marzo 2023

Composizione porte logiche: prime considerazioni

$Y = AND(A, B)$

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

$Y = OR(A, B)$

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 1 |

$Y = NOT(A)$

| A | Y |
|---|---|
| 0 | 1 |
| 1 | 0 |

$Y = AND(NOT(A), NOT(B))$

| A | B | Y |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 0 |

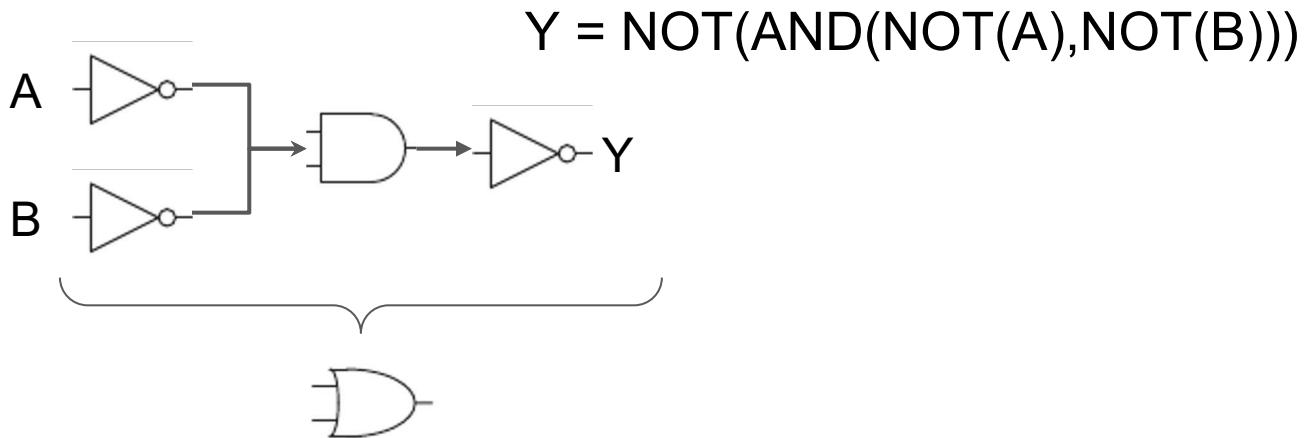
$Y = NOT(AND(NOT(A), NOT(B)))$



| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 1 |

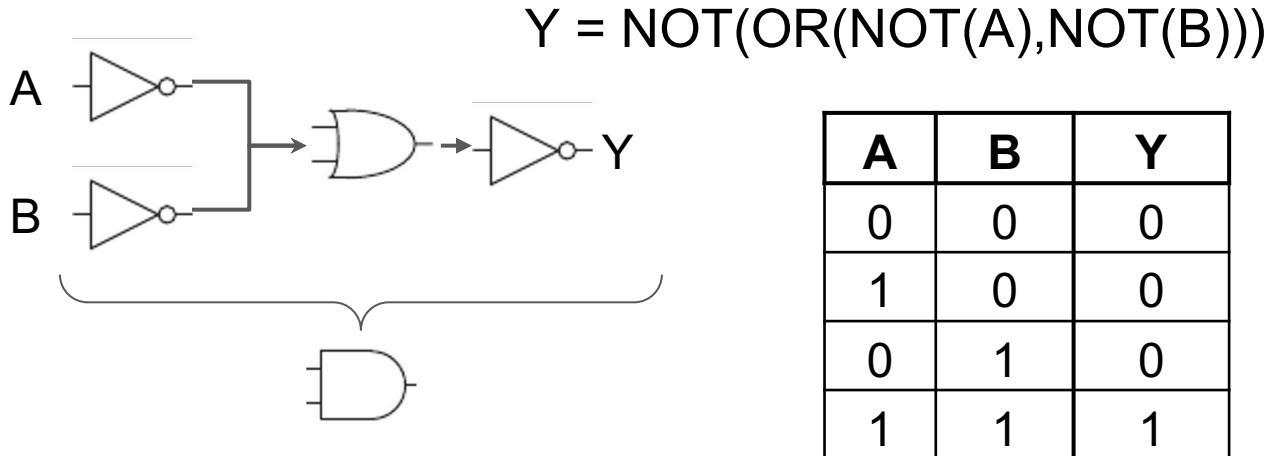
OR!!!!!!

Composizione porte logiche: prime considerazioni



- Componendo AND e NOT posso ottenere un OR
 - Quindi posso utilizzare solo AND e NOT
 - Ma...

Composizione porte logiche: prime considerazioni

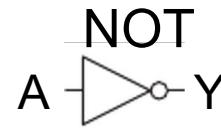
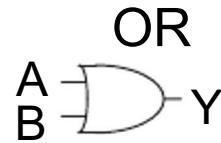
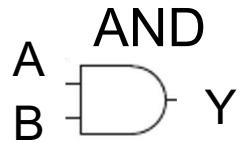


- Componendo OR e NOT posso ottenere un AND
 - Quindi posso anche utilizzare solo AND e OR

Composizione porte logiche: prime considerazioni

- CONSIDERAZIONI:
 - Solo OR e NOT oppure solo AND e NOT sono sufficienti per ogni funzione Booleana
 - Posso comporre varie porte logiche e considerare la funzione ottenuta come una nuova porta logica “primitiva”

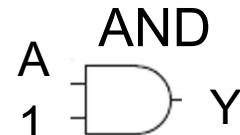
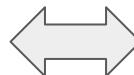
Alcune porte logiche di base



BUFFER (BUF)

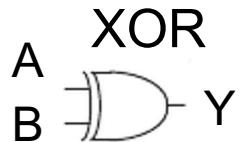
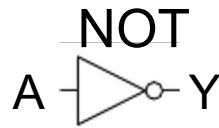
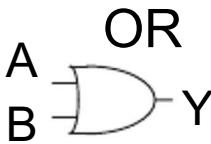
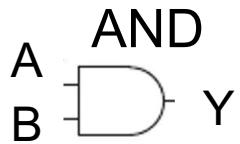


| A | Y |
|---|---|
| 0 | 0 |
| 1 | 1 |

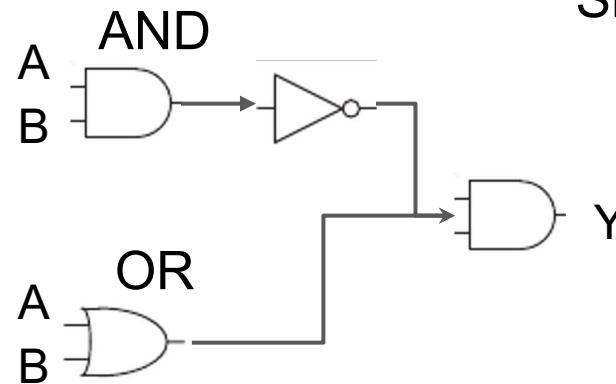


| A | B | Y |
|---|---|---|
| 0 | 1 | 0 |
| 1 | 1 | 1 |

Alcune porte logiche di base

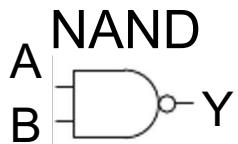
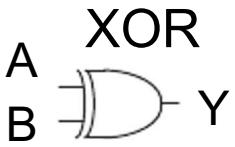
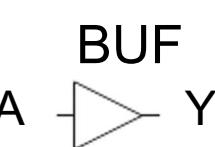
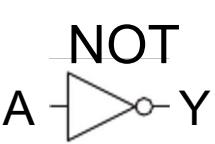
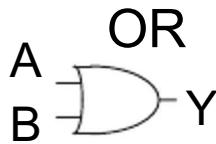
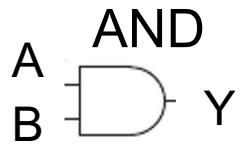


| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

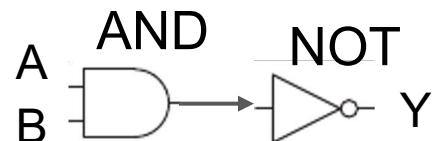


$$Y = \text{XOR}(A, B) = \text{AND}(\text{NOT}(\text{AND}(A, B)), \text{OR}(A, B))$$

Alcune porte logiche di base

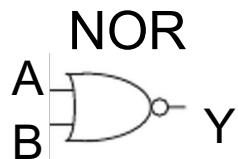
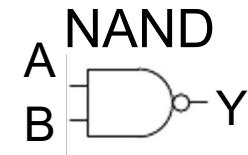
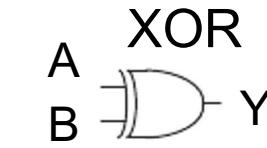
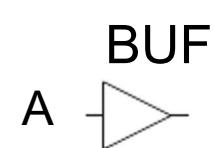
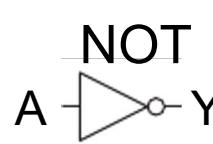
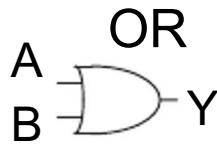
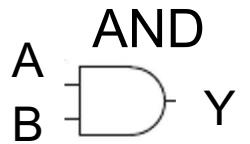


| A | B | Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

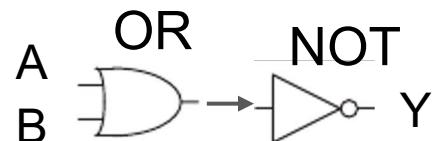


$$Y = \text{NOT}(\text{AND}(A, B))$$

Alcune porte logiche di base

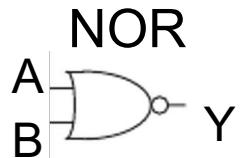
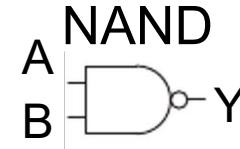
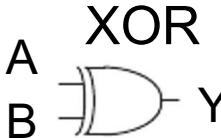
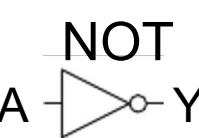
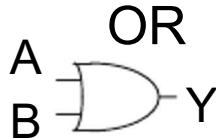
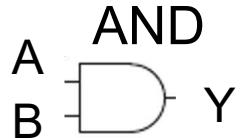


| A | B | Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

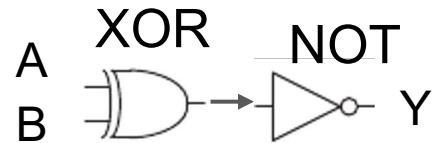


$$Y = \text{NOT}(\text{OR}(A, B))$$

Alcune porte logiche di base

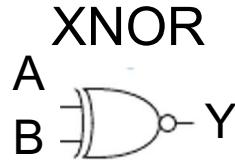
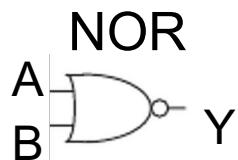
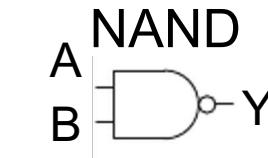
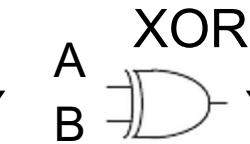
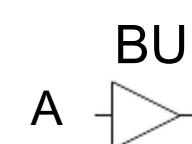
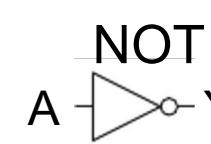
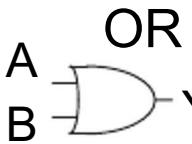
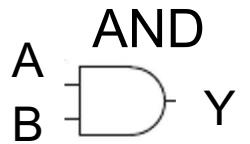


| A | B | Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |



$$Y = \text{NOT}(\text{XOR}(A, B))$$

Alcune porte logiche di base



- Rappresentano porte logiche comunemente utilizzate come elementi di base
- Anche se molte di loro possono essere viste come composizione di altre porte logiche
- Si può dimostrare che sono sufficienti:
 - OR e NOT
 - AND e NOT
 - solo NAND

Porte logiche a più input

- E' possibile costruire porte logiche a N input (argomenti)
 - Possono essere viste come composizione di porte logiche ad 1 o 2 input
- Le più usate sono:
 - AND, OR, XOR, NAND, NOR, e XNOR con tre o più input
- AND: 1 se tutti gli input sono 1, altrimenti 0
- OR: 1 se almeno un input è 1, altrimenti 0
- XOR: 1 se un numero dispari di input è 1, altrimenti 0. E' anche detto **parity gate**.

Porte logiche a più input

- ESERCIZIO:
 - Provare a costruire AND, OR e XOR con N-input usando porte logiche con 1 o 2 input, considerare N=3 e N=5

Un po' di più sulla terminologia (1)

- Il **complemento** di una variabile **A** è il suo negato **not(A)** (o semplicemente \bar{A})
- Una variabile o il suo negato è chiamato **letterale**,
 - A, B, \bar{A}
- Un letterale non negato A è chiamato “**true form**”, il suo negato \bar{A} “**complementary form**”
- AND di più letterali può anche essere chiamato **prodotto (o implicante)**
 - $\text{AND}(A,B,\bar{A})= A \cdot B \cdot \bar{A}$
- OR di uno o più letterali è chiamato **somma**
 - $\text{OR}(A,B,\bar{A})= A + B + \bar{A}$

Un po' di più sulla terminologia (2)

- Se ho una funzione di N variabili, se tutte le variabili si presentano in un prodotto questo di chiama **minterm**
 - Ad esempio, se ho una funzione di A, B e C
 - ABC minterm
 - $\bar{A}BC$ minterm
 - $\bar{A}B$ non è un minterm perchè non coinvolge C

Un po' di più sulla terminologia (3)

- Se ho una funzione di N variabili, se tutte le variabili si presentano in una somma, questa di chiama **maxterm**
 - Ad esempio, se ho una funzione di A, B e C
 - $A+B+C$ maxterm
 - $\bar{A}+B+C$ maxterm
 - $\bar{A}+B$ non è un maxterm perche' non coinvolge C

Un po' di più sulla terminologia: ordine delle operazioni

- L'ordine delle operazioni è fondamentale
 - NOT ha la precedenza più alta
 - poi, c'è l'operazione AND
 - infine, l'operazione OR
- Ad esempio:
 - $\bar{A}B + C = (\text{NOT}(A) \text{ AND } B) \text{ OR } C$

Da tabella di verità a espressione Booleana: somma dei prodotti

| A | B | Y | minterm | nome minterm |
|---|---|---|--------------|--------------|
| 0 | 0 | 0 | not(A)not(B) | m_0 |
| 1 | 0 | 0 | A not(B) | m_1 |
| 0 | 1 | 0 | not(A) B | m_2 |
| 1 | 1 | 1 | A B | m_3 |



minterm TRUE per ciascuna riga

$$Y = m_3$$

Ogni tabella di verità la possiamo trasformare nella somma dei minterm corrispondenti a Y vero!

NOTA: un minterm può essere vero solo quando tutti i suoi letterali (true o complementary form) sono veri

Da tabella di verità a espressione Booleana: somma dei prodotti

| A | B | Y | minterm | nome minterm |
|---|---|---|--------------|--------------|
| 0 | 0 | 0 | not(A)not(B) | m_0 |
| 1 | 0 | 1 | A not(B) | m_1 |
| 0 | 1 | 1 | not(A) B | m_2 |
| 1 | 1 | 1 | A B | m_3 |



minterm TRUE per ciascuna riga

$$Y = m_1 + m_2 + m_3$$

Ogni tabella di verità la possiamo trasformare nella somma dei minterm corrispondenti a Y vero!

Da tabella di verità a espressione Booleana: somma dei prodotti

| A | B | Y | miniterm | nome minterm |
|---|---|---|--------------|--------------|
| 0 | 0 | 0 | not(A)not(B) | m_0 |
| 1 | 0 | 1 | A not(B) | m_1 |
| 0 | 1 | 1 | not(A) B | m_2 |
| 1 | 1 | 0 | A B | m_3 |



minterm TRUE per ciascuna riga

$$Y = m_1 + m_2 = A \text{not}(B) + \text{not}(A)B$$

Ogni tabella di verità la possiamo trasformare nella somma dei minterm corrispondenti a Y vero!

Da tabella di verità a espressione Booleana: somma dei prodotti

| A | B | Y | miniterm | nome minterm |
|---|---|---|--------------|--------------|
| 0 | 0 | 0 | not(A)not(B) | m_0 |
| 1 | 0 | 1 | A not(B) | m_1 |
| 0 | 1 | 1 | not(A) B | m_2 |
| 1 | 1 | 0 | A B | m_3 |



minterm TRUE per ciascuna riga

$$Y = m_1 + m_2$$

Questo modo di esprimere una tabella di verità è detto:
forma canonica in somma dei prodotti

Da tabella di verità a espressione Booleana: somma dei prodotti

| A | B | Y | miniterm | nome minterm |
|---|---|---|--------------|--------------|
| 0 | 0 | 0 | not(A)not(B) | m_0 |
| 1 | 0 | 1 | A not(B) | m_1 |
| 0 | 1 | 1 | not(A) B | m_2 |
| 1 | 1 | 0 | A B | m_3 |



minterm TRUE per ciascuna riga

$$Y = m_1 + m_2$$

- La forma canonica in somma dei prodotti può essere espressa in notazione sigma (Σ)
- La Σ mi rappresenta la sommatoria

$$Y = \Sigma (m_1, m_2)$$

oppure: $Y = \Sigma (1, 2)$

Da tabella di verità a espressione Booleana: somma dei prodotti

| A | B | Y | miniterm | nome minterm |
|---|---|---|--------------|--------------|
| 0 | 0 | 0 | not(A)not(B) | m_0 |
| 1 | 0 | 1 | A not(B) | m_1 |
| 0 | 1 | 1 | not(A) B | m_2 |
| 1 | 1 | 1 | A B | m_3 |



minterm TRUE per ciascuna riga

Altro esempio:

$$Y = m_1 + m_2 + m_3$$

- La forma canonica in somma dei prodotti può essere espressa in notazione sigma (Σ)
- La Σ mi rappresenta la sommatoria

$$Y = \Sigma (m_1, m_2, m_3)$$

oppure: $Y = \Sigma (1, 2, 3)$

Da tabella di verità a espressione Booleana: Prodotto di somme

| A | B | Y | maxterm | nome maxterm |
|---|---|---|-------------------------------|--------------|
| 0 | 0 | 0 | $A+B$ | M_0 |
| 1 | 0 | 0 | $\text{not}(A) + B$ | M_1 |
| 0 | 1 | 0 | $A + \text{not}(B)$ | M_2 |
| 1 | 1 | 1 | $\text{not}(A)+\text{not}(B)$ | M_3 |



maxtem FALSE per ciascuna riga

$$Y = M_0 M_1 M_2$$

Ogni tabella di verità la possiamo trasformare nel prodotto dei maxterm corrispondenti a Y FALSE! (**Forma canonica come prodotto di somme**)

NOTA: un maxterm può essere falso solo quando tutti i suoi letterali (true o complementary form) sono falsi

Da tabella di verità a espressione Booleana: Prodotto di somme

| A | B | Y | maxterm | nome maxterm |
|---|---|---|-------------------------------|--------------|
| 0 | 0 | 0 | $A+B$ | m_0 |
| 1 | 0 | 0 | $\text{not}(A) + B$ | m_1 |
| 0 | 1 | 0 | $A + \text{not}(B)$ | m_2 |
| 1 | 1 | 1 | $\text{not}(A)+\text{not}(B)$ | m_3 |



maxterm FALSE per
ciascuna riga

$$Y = M_0 M_1 M_2$$

Possiamo anche utilizzare la
notazione pi (Π)

$$Y = \Pi(M_0, M_1, M_2) \text{ oppure}$$

$$Y = \Pi(0, 1, 2)$$

Architettura degli Elaboratori

Lezione 11

Docente: R.Prevete
a.a. 2022/2023
31/3/2023

Esempi forme canoniche

ESEMPIO

| A | B | C | Y |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

Convertire in forma canonica come somma di prodotti

Funzione di 3 variabili: A, B, C

Output: una sola variabile Y

ESEMPIO

| A | B | C | Y | Minterm | nomi |
|---|---|---|---|--------------------|-------|
| 0 | 0 | 0 | 0 | not(A)not(B)not(C) | m_0 |
| 0 | 0 | 1 | 1 | not(A)not(B)C | m_1 |
| 0 | 1 | 0 | 0 | not(A)Bnot(C) | m_2 |
| 0 | 1 | 1 | 1 | not(A)B C | m_3 |
| 1 | 0 | 0 | 0 | A not(B)not(C) | m_4 |
| 1 | 0 | 1 | 1 | A not(B) C | m_5 |
| 1 | 1 | 0 | 0 | A B not(C) | m_6 |
| 1 | 1 | 1 | 1 | A B C | m_7 |

ESEMPIO

| A | B | C | Y | Minterm | nomi |
|---|---|---|---|--------------------|-------|
| 0 | 0 | 0 | 0 | not(A)not(B)not(C) | m_0 |
| 0 | 0 | 1 | 1 | not(A)not(B)C | m_1 |
| 0 | 1 | 0 | 0 | not(A)Bnot(C) | m_2 |
| 0 | 1 | 1 | 1 | not(A)B C | m_3 |
| 1 | 0 | 0 | 0 | A not(B)not(C) | m_4 |
| 1 | 0 | 1 | 1 | A not(B) C | m_5 |
| 1 | 1 | 0 | 0 | A B not(C) | m_6 |
| 1 | 1 | 1 | 1 | A B C | m_7 |

$$Y = m_1 + m_3 + m_5 + m_7$$

$$Y = \Sigma (m_1, m_3, m_5, m_7)$$

$$Y = \Sigma (1, 3, 5, 7)$$

$$Y = \Sigma (1, 3, 5, 7) = \text{not}(A)\text{not}(B)C + \text{not}(A)BC + A\text{not}(B)C + ABC$$

ESEMPIO

| A | B | C | Y |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

**Convertire in forma canonica come
prodotto di somme**

ESEMPIO

| A | B | C | Y | Maxterm | nomi |
|---|---|---|---|---|-------|
| 0 | 0 | 0 | 0 | $A+B+C$ | M_0 |
| 0 | 0 | 1 | 1 | $A+B+\text{not}(C)$ | M_1 |
| 0 | 1 | 0 | 0 | $A+\text{not}(B)+C$ | M_2 |
| 0 | 1 | 1 | 1 | $A+\text{not}(B)+\text{not}(C)$ | M_3 |
| 1 | 0 | 0 | 0 | $\text{not}(A)+B+C$ | M_4 |
| 1 | 0 | 1 | 1 | $\text{not}(A)+B+\text{not}(C)$ | M_5 |
| 1 | 1 | 0 | 0 | $\text{not}(A)+\text{not}(B)+C$ | M_6 |
| 1 | 1 | 1 | 1 | $\text{not}(A)+\text{not}(B)+\text{not}(C)$ | M_7 |

ESEMPIO

| A | B | C | Y | MAXTERM | nomi |
|---|---|---|---|---|-------|
| 0 | 0 | 0 | 0 | $A+B+C$ | M_0 |
| 0 | 0 | 1 | 1 | $A+B+\text{not}(C)$ | M_1 |
| 0 | 1 | 0 | 0 | $A+\text{not}(B)+C$ | M_2 |
| 0 | 1 | 1 | 1 | $A+\text{not}(B)+\text{not}(C)$ | M_3 |
| 1 | 0 | 0 | 0 | $\text{not}(A)+B+C$ | M_4 |
| 1 | 0 | 1 | 1 | $\text{not}(A)+B+\text{not}(C)$ | M_5 |
| 1 | 1 | 0 | 0 | $\text{not}(A)+\text{not}(B)+C$ | M_6 |
| 1 | 1 | 1 | 1 | $\text{not}(A)+\text{not}(B)+\text{not}(C)$ | M_7 |

$$Y = M_0 M_2 M_4 M_6$$

$$Y = \prod(M_0 M_2 M_4 M_6)$$

$$Y = \prod(0, 2, 4, 6)$$

$$Y = \prod(0, 2, 4, 6) = (A+B+C)(A+\text{not}(B)+C)(\text{not}(A)+B+C)(\text{not}(A)+\text{not}(B)+C)$$

Da tabella di verità a espressione Booleana

- Forma canonica come somma di prodotti
- Forma canonica come prodotto di somme

Quale scegliere?

Dipende se la tabella di verità presenta come Y (uscita) più 1 (True) o più 0 (False)

- Più 1 → Forma canonica come prodotto di somme
- Più 0 → Forma canonica come somma di prodotti

Algebra di Boole, un po' più formali

- VISTO:
 - Costruire espressioni booleane a partire dalla tabella di verità
 - “comporre” porte logiche
- Quale espressione booleana per una data tabella di verità?
- Possiamo utilizzare proprio l'algebra di Boole, così come utilizziamo l'algebra per semplificare espressioni aritmetiche.

$$Y = (A \wedge B) \vee (A \wedge C)$$



$$Y = A \wedge (B \vee C)$$

| A | B | C | Y |
|---|---|---|----|
| 0 | 0 | 0 | 10 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

Algebra di Boole, un po' più formali

- **ASSIOMI:** Un insieme di espressioni Booleane che assumiamo siano vere
- **Teoremi:** espressioni Booleane che dimostriamo vere a partire dagli assiomi o da altri teoremi già dimostrati

Algebra di Boole: assiomi

- **ASSIOMA 1:** $A=0$ se A è diversa da 1 (variabili dicotomiche)
 - **Una variabile Booleana è falsa (0) se non è vera (1)**
- **Duale ASSIOMA 1:**
 - $A=1$ se A è diversa da 0
 - **Una variabile Booleana è vera (1) se non è falsa (0)**
- **Questo assioma ci dice che stiamo trattando con valori binari**

Algebra di Boole: assiomi

- **ASSIOMA 2:** $\text{NOT}(0)=1$
 - Esiste una operazione NOT : NOT applicato su falso (0) è vero (1)
- **Duale ASSIOMA 2:**
 - $\text{NOT}(1)=0$
 - Esiste una operazione NOT : NOT applicato su vero (1) è falso (0)

Algebra di Boole: assiomi

- ASSIOMA 3: $0 \wedge 0 = 0$
 - Esiste una operazione \wedge (**and**) : applicato su due valori falsi (**0**) da falso (**0**)
- Duale ASSIOMA 3:
 - $1 \vee 1 = 1$
 - Esiste una operazione \vee (**or**) : applicato su due valori veri (**1**) da valore (**1**)

Algebra di Boole: assiomi

- ASSIOMA 4: $1 \wedge 1 = 1$
 - Esiste una operazione \wedge (**and**) : applicato su due valori veri **(1) da vero (1)**
- Duale ASSIOMA 4:
 - $0 \vee 0 = 0$
 - Esiste una operazione \vee (**or**) : applicato su due valori falsi **(0) da valore (0)**

Algebra di Boole: assiomi

- ASSIOMA 5: $0 \wedge 1 = 1 \wedge 0 = 0$
 - Esiste una operazione \wedge (and) : applicato su un valore vero (1) e su un valore falso (0) da falso (0)
- Duale ASSIOMA 5:
 - $0 \vee 1 = 1 \vee 0 = 1$
 - Esiste una operazione \vee (or) : applicato su un valore vero(1) e su un valore falso (0) da vero (1)

Algebra di Boole: Assiomi

Table 2.1 Axioms of Boolean algebra

| | Axiom | | Dual | Name |
|----|---------------------------------|-----|-----------------------|--------------|
| A1 | $B = 0$ if $B \neq 1$ | A1' | $B = 1$ if $B \neq 0$ | Binary field |
| A2 | $\overline{0} = 1$ | A2' | $\overline{1} = 0$ | NOT |
| A3 | $0 \bullet 0 = 0$ | A3' | $1 + 1 = 1$ | AND/OR |
| A4 | $1 \bullet 1 = 1$ | A4' | $0 + 0 = 0$ | AND/OR |
| A5 | $0 \bullet 1 = 1 \bullet 0 = 0$ | A5' | $1 + 0 = 0 + 1 = 1$ | AND/OR |

Algebra di Boole: Assiomi

Table 2.1 Axioms of Boolean algebra

| Axiom | Dual | Name |
|--------------------------------|---------------------------|--------------|
| A1 $B = 0$ if $B \neq 1$ | A1' $B = 1$ if $B \neq 0$ | Binary field |
| A2 $\bar{0} = 1$ | A2' $\bar{1} = 0$ | NOT |
| A3 $0 \cdot 0 = 0$ | A3' $1 + 1 = 1$ | AND/OR |
| A4 $1 \cdot 1 = 1$ | A4' $0 + 0 = 0$ | AND/OR |
| A5 $0 \cdot 1 = 1 \cdot 0 = 0$ | A5' $1 + 0 = 0 + 1 = 1$ | AND/OR |

NOTA: per ogni assioma o teorema esiste il suo duale ottenuto scambiano **0 con 1, 1 con 0, \wedge (and) con \vee (or)** e **\vee (or) con \wedge (and)**

Algebra di Boole: Teoremi su una variabile

- **Teorema 1 (Identità):** $A \wedge 1 = A$
 - L'operazione \wedge applicata su una variabile **A** e su un valore **vero (1)** è uguale alla sola variabile **A**
- **Duale Teorema 1:**
 - $A \vee 0 = A$
 - L'operazione \vee (**or**) applicata su una variabile **A** e su un valore **falso (0)** è uguale alla sola variabile **A**

Algebra di Boole: Teoremi su una variabile

- **Teorema 1 (Identità):** $A \wedge 1 = A$
 - L'operazione \wedge applicata su su una variabile A e su un valore vero (1) è uguale alla sola variabile A
- **Dimostrazione:**
 - $A \wedge 1$
 - Se $A=0$, $A \wedge 1 = 0 = A$ per assioma 5
 - Se $A=1$, $A \wedge 1 = 1 = A$ per assioma 4
- Similmente per il suo duale

Algebra di Boole: Teoremi su una variabile

- **Teorema 2 (elemento nullo):** $A \wedge 0 = 0$
 - L'operazione \wedge applicata su una variabile A e su un valore falso (0) è uguale al valore falso
- **Duale Teorema 2:**
 - $A \vee 1 = 1$
 - L'operazione \vee (or) applicata su una variabile A e su un valore vero (1) è uguale al solo valore vero (1)

Algebra di Boole: Teoremi su una variabile

- **Teorema 2 (elemento nullo):** $A \wedge 0 = 0$
 - L'operazione \wedge applicata su una variabile **A** e su un valore falso (**0**) è uguale al valore falso
- **Dimostrazione:**
 - $A \wedge 0 = 0$
 - Se $A=0$, allora $0 \wedge 0 = 0$ per l'assioma 3
 - Se $A=1$, allora $1 \wedge 0 = 0$ per l'assioma 5
- Analogamente per il duale

Algebra di Boole: Teoremi su una variabile

- **Teorema 3 (idempotenza):** $A \wedge A = A$
 - L'operazione \wedge applicata su una variabile **A** e se stessa è uguale ancora ad **A**
- **Duale Teorema 3:**
 - $A \vee A = A$
 - L'operazione \vee applicata su una variabile **A** e se stessa è uguale ancora ad **A**

Algebra di Boole: Teoremi su una variabile

- **Teorema 3 (idempotenza):** $A \wedge A = A$
 - L'operazione \wedge applicata su una variabile **A** e se stessa è uguale ancora ad **A**
- **Dimostrazione:**
 - $A \wedge A = A$
 - Se $A=1$, allora $1 \wedge 1 = 1 = A$ per assioma 4
 - Se $A=0$, allora $0 \wedge 0 = 0 = A$ per assioma 3
- Analogamente per il duale

Algebra di Boole: Teoremi su una variabile

- **Teorema 4 (involuzione o doppia negazione):** $\text{NOT}(\text{NOT}(A))=A$
 - L'operazione NOT applicata due volte su una variabile A è uguale ancora ad A
- **Dimostrazione:**
 - $\text{NOT}(\text{NOT}(A))=A$
 - SE $A=1$, allora $\text{NOT}(\text{NOT}(1))= \text{NOT}(0)=1=A$, Assioma 2 e suo duale
 - SE $A=0$, allora $\text{NOT}(\text{NOT}(0))= \text{NOT}(1)=0=A$, Assioma 2 e suo duale

Algebra di Boole: Teoremi su una variabile

- **Teorema 5 (complemento):** $A \wedge \text{NOT}(A) = 0$
 - L'operazione \wedge applicata su una variabile A e la sua negata è uguale a falso (0)
- **Duale Teorema 5:**
 - $A \vee \text{NOT}(A) = 1$
 - L'operazione \vee applicata su una variabile A e la sua negata è uguale a vero (1)

Algebra di Boole: Teoremi su una variabile

- **Teorema 5 (complemento):** $A \wedge \text{NOT}(A) = 0$
 - L'operazione \wedge applicata su una variabile A e la sua negata è uguale a falso (0)
- **Dimostrazione:**
 - $A \wedge \text{NOT}(A) = 0$
 - Se $A=0$, allora $0 \wedge \text{NOT}(0) = 0 \wedge 1 = 0$ assiomi 2 e 5
 - Se $A=1$, allora $1 \wedge \text{NOT}(1) = 1 \wedge 0 = 0$ duale assioma 2 ed assioma 5
- Analogamente per il duale

Algebra di Boole: Teoremi su una variabile

Table 2.2 Boolean theorems of one variable

| | Theorem | | Dual | Name |
|----|------------------------------|-------------------------------|------------------------|--------------|
| T1 | $B \bullet 1 = B$ | T1' | $B + 0 = B$ | Identity |
| T2 | $B \bullet 0 = 0$ | T2' | $B + 1 = 1$ | Null Element |
| T3 | $B \bullet B = B$ | T3' | $B + B = B$ | Idempotency |
| T4 | | $\overline{\overline{B}} = B$ | | Involution |
| T5 | $B \bullet \overline{B} = 0$ | T5' | $B + \overline{B} = 1$ | Complements |

Architettura degli Elaboratori

Lezione 12

Docente: R.Prevete
a.a. 2022/2023
3 Aprile 2023

Algebra di Boole: Teoremi su una variabile e porte logiche

Table 2.2 Boolean theorems of one variable

| Theorem | | Dual | | Name |
|---------|----------------------------|-------------------------------|------------------------|--------------|
| T1 | $B \cdot 1 = B$ | $T1'$ | $B + 0 = B$ | Identity |
| T2 | $B \cdot 0 = 0$ | $T2'$ | $B + 1 = 1$ | Null Element |
| T3 | $B \cdot B = B$ | $T3'$ | $B + B = B$ | Idempotency |
| T4 | | $\overline{\overline{B}} = B$ | | Involution |
| T5 | $B \cdot \overline{B} = 0$ | $T5'$ | $B + \overline{B} = 1$ | Complements |

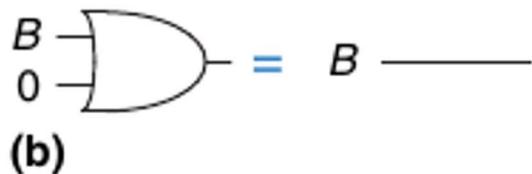
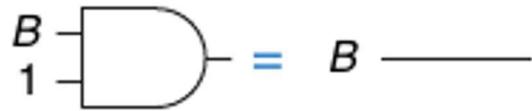


Figure 2.14 Identity theorem in hardware: (a) T1, (b) T1'

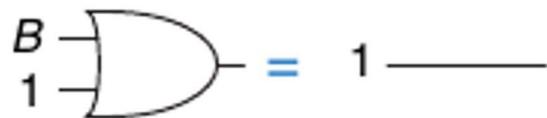
Algebra di Boole: Teoremi su una variabile e porte logiche

Table 2.2 Boolean theorems of one variable

| Theorem | | Dual | | Name |
|---------|----------------------------|-------------------------------|------------------------|--------------|
| T1 | $B \cdot 1 = B$ | T1' | $B + 0 = B$ | Identity |
| T2 | $B \cdot 0 = 0$ | T2' | $B + 1 = 1$ | Null Element |
| T3 | $B \cdot B = B$ | T3' | $B + B = B$ | Idempotency |
| T4 | | $\overline{\overline{B}} = B$ | | Involution |
| T5 | $B \cdot \overline{B} = 0$ | T5' | $B + \overline{B} = 1$ | Complements |



(a)



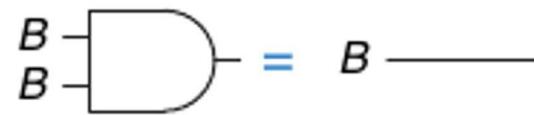
(b)

Figure 2.15 Null element theorem in hardware: (a) T2, (b) T2'

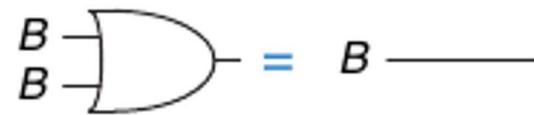
Algebra di Boole: Teoremi su una variabile e porte logiche

Table 2.2 Boolean theorems of one variable

| Theorem | | Dual | | Name |
|---------|----------------------------|-------------------------------|------------------------|--------------|
| T1 | $B \cdot 1 = B$ | T1' | $B + 0 = B$ | Identity |
| T2 | $B \cdot 0 = 0$ | T2' | $B + 1 = 1$ | Null Element |
| T3 | $B \cdot B = B$ | T3' | $B + B = B$ | Idempotency |
| T4 | | $\overline{\overline{B}} = B$ | | Involution |
| T5 | $B \cdot \overline{B} = 0$ | T5' | $B + \overline{B} = 1$ | Complements |



(a)



(b)

Figure 2.16 Idempotency theorem in hardware: (a) T3, (b) T3'

Algebra di Boole: Teoremi su una variabile e porte logiche

Table 2.2 Boolean theorems of one variable

| Theorem | | Dual | | Name |
|---------|----------------------------|-------------------------------|------------------------|--------------|
| T1 | $B \cdot 1 = B$ | T1' | $B + 0 = B$ | Identity |
| T2 | $B \cdot 0 = 0$ | T2' | $B + 1 = 1$ | Null Element |
| T3 | $B \cdot B = B$ | T3' | $B + B = B$ | Idempotency |
| T4 | | $\overline{\overline{B}} = B$ | | Involution |
| T5 | $B \cdot \overline{B} = 0$ | T5' | $B + \overline{B} = 1$ | Complements |

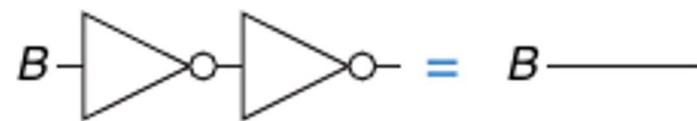
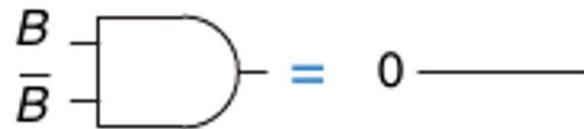


Figure 2.17 Involution theorem in hardware: T4

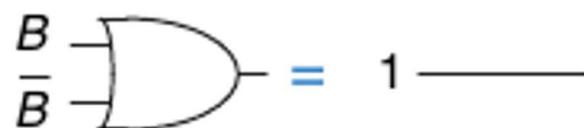
Algebra di Boole: Teoremi su una variabile e porte logiche

Table 2.2 Boolean theorems of one variable

| Theorem | Dual | Name |
|-------------------------------|-------------------------------|--------------|
| T1 $B \cdot 1 = B$ | T1' $B + 0 = B$ | Identity |
| T2 $B \cdot 0 = 0$ | T2' $B + 1 = 1$ | Null Element |
| T3 $B \cdot B = B$ | T3' $B + B = B$ | Idempotency |
| T4 | $\overline{\overline{B}} = B$ | Involution |
| T5 $B \cdot \overline{B} = 0$ | T5' $B + \overline{B} = 1$ | Complements |



(a)



(b)

Figure 2.18 Complement theorem in hardware: (a) T5, (b) T5'

Algebra di Boole. Teoremi su più variabili

- **Teorema 6 (commutatività):** $A \wedge B = B \wedge A$
 - Cambiando l'ordine degli input per l'operazione **AND** il risultato non cambia
- **Duale Teorema 6:**
 - $A \vee B = B \vee A$
 - Cambiando l'ordine degli input per l'operazione **OR** il risultato non cambia

Algebra di Boole. Teoremi su più variabili

- **Teorema 6 (commutatività):** $A \wedge B = B \wedge A$

- Cambiando l'ordine degli input per l'operazione \wedge (**AND**) il risultato non cambia

- **DIMOSTRAZIONE:**

- $A \wedge B$
- Se $A = 0$ e $B = 0$, $A \wedge B = 0$ e $B \wedge A = 0$ (A3), $A \wedge B = B \wedge A$
- Se $A = 1$ (0) e $B = 0$ (1), $A \wedge B = 0$ e $B \wedge A = 0$ (A5), $A \wedge B = B \wedge A$
- Se $A = 1$ e $B = 1$, $A \wedge B = 1$ e $B \wedge A = 1$ (A4), $A \wedge B = B \wedge A$

Table 2.1 Axioms of Boolean algebra

| | Axiom | | Dual | Name |
|----|-----------------------------|--|-------|-----------------------|
| A1 | $B = 0$ if $B \neq 1$ | | $A1'$ | $B = 1$ if $B \neq 0$ |
| A2 | $\bar{0} = 1$ | | $A2'$ | $\bar{1} = 0$ |
| A3 | $0 \cdot 0 = 0$ | | $A3'$ | $1 + 1 = 1$ |
| A4 | $1 \cdot 1 = 1$ | | $A4'$ | $0 + 0 = 0$ |
| A5 | $0 \cdot 1 = 1 \cdot 0 = 0$ | | $A5'$ | $1 + 0 = 0 + 1 = 1$ |

Algebra di Boole. Teoremi su più variabili

- **Teorema 7 (Associatività):** $(A \wedge B) \wedge C = A \wedge (B \wedge C)$
 - Lo specifico raggruppamento degli input per l'operazione \wedge (**AND**) non cambia il risultato
- **Duale Teorema 7:**
 - $(A \vee B) \vee C = A \vee (B \vee C)$
 - Lo specifico raggruppamento degli input per l'operazione \vee (**OR**) non cambia il risultato

Algebra di Boole. Teoremi su più variabili

- **Teorema 7 (Associatività):** $(A \wedge B) \wedge C = A \wedge (B \wedge C)$

- Lo specifico raggruppamento degli input per l'operazione \wedge (**AND**) non cambia il risultato

- **DIMOSTRAZIONE:**

- $(A \wedge B) \wedge C$
 - Se $C=0$, $(A \wedge B) \wedge C=0$ (A5). Se $C=1$, $A \wedge B$. (T1, **Identity**)
- $A \wedge (B \wedge C)$
 - Se $C=0$, $A \wedge (B \wedge C)=0$ (A5). Se $C=1$, $A \wedge (B \wedge C)=A \wedge B$ (T1, **Identity**)

- **Teorema 1 (Identità):** $A \wedge 1 = A$

Table 2.1 Axioms of Boolean algebra

| | Axiom | | Dual | Name |
|----|---------------------------------|--|-------|-----------------------|
| A1 | $B = 0$ if $B \neq 1$ | | $A1'$ | $B = 1$ if $B \neq 0$ |
| A2 | $\bar{0} = 1$ | | $A2'$ | $\bar{1} = 0$ |
| A3 | $0 \bullet 0 = 0$ | | $A3'$ | $1 + 1 = 1$ |
| A4 | $1 \bullet 1 = 1$ | | $A4'$ | $0 + 0 = 0$ |
| A5 | $0 \bullet 1 = 1 \bullet 0 = 0$ | | $A5'$ | $1 + 0 = 0 + 1 = 1$ |

Algebra di Boole. Teoremi su più variabili

- **Teorema 8 (Distributività):** $(A \wedge B) \vee (A \wedge C) = A \wedge (B \vee C)$
 - \wedge (**AND**) si distribuisce su \vee (**OR**)
- **Duale Teorema 8:**
 - $(A \vee B) \wedge (A \vee C) = A \vee (B \wedge C)$
 - \vee (**OR**) si distribuisce su \wedge (**AND**)

Algebra di Boole. Teoremi su più variabili

- **Teorema 8 (Distributività):** $(A \wedge B) \vee (A \wedge C) = A \wedge (B \vee C)$
 - \wedge (**AND**) si distribuisce su \vee (**OR**)

Table 2.1 Axioms of Boolean algebra

| Axiom | | Dual | | Name |
|-------|---------------------------------|------|-----------------------|--------------|
| A1 | $B = 0$ if $B \neq 1$ | A1' | $B = 1$ if $B \neq 0$ | Binary field |
| A2 | $\bar{0} = 1$ | A2' | $\bar{1} = 0$ | NOT |
| A3 | $0 \bullet 0 = 0$ | A3' | $1 + 1 = 1$ | AND/OR |
| A4 | $1 \bullet 1 = 1$ | A4' | $0 + 0 = 0$ | AND/OR |
| A5 | $0 \bullet 1 = 1 \bullet 0 = 0$ | A5' | $1 + 0 = 0 + 1 = 1$ | AND/OR |

• DIMOSTRAZIONE:

- $(A \wedge B) \vee (A \wedge C)$
 - Se $A=0$, $(A \wedge B) \vee (A \wedge C)=0 \vee 0 = 0$ (T1, **Identity**, A4')
 - Se $A=1$, $(A \wedge B) \vee (A \wedge C)=B \vee C$ (T1, **Identity**)
- $A \wedge (B \vee C)$
 - Se $A=0$, $A \wedge (B \vee C)=0$ (T2, elemento nullo)
 - Se $A=1$, $A \wedge (B \vee C)=B \vee C$ (T1, **Identity**)

Algebra di Boole. Teoremi su più variabili

- **Teorema 9 (Assorbimento,covering):**

- $A \wedge (A \vee B) = A$

- **Duale Teorema 9:**

- $A \vee (A \wedge B) = A$

Algebra di Boole. Teoremi su più variabili

- **Teorema 9 (Assorbimento):**

- $A \wedge (A \vee B) = A$

Table 2.1 Axioms of Boolean algebra

| | Axiom | | Dual | Name |
|----|---------------------------------|--|-------|-----------------------|
| A1 | $B = 0$ if $B \neq 1$ | | $A1'$ | $B = 1$ if $B \neq 0$ |
| A2 | $\bar{0} = 1$ | | $A2'$ | $\bar{1} = 0$ |
| A3 | $0 \bullet 0 = 0$ | | $A3'$ | $1 + 1 = 1$ |
| A4 | $1 \bullet 1 = 1$ | | $A4'$ | $0 + 0 = 0$ |
| A5 | $0 \bullet 1 = 1 \bullet 0 = 0$ | | $A5'$ | $1 + 0 = 0 + 1 = 1$ |

- **DIMOSTRAZIONE:**

- $A \wedge (A \vee B) = A$

- Se $B=0$, $A \wedge (A \vee B) = A \wedge A = A$ (Idempotenza)
- Se $B=1$, $A \wedge (A \vee B) = A \wedge 1 = A$ (Identity)

Algebra di Boole. Teoremi su più variabili

- **Teorema 10 (Combining):**

- $(A \wedge B) \vee (A \wedge \text{not}(B)) = A$

- **Duale Teorema 10:**

- $(A \vee B) \wedge (A \vee \text{not}(B)) = A$

Algebra di Boole. Teoremi su più variabili

- **Teorema 10 (Combining):**

- $(A \wedge B) \vee (A \wedge \neg B) = A$

Table 2.1 Axioms of Boolean algebra

| | Axiom | | Dual | Name |
|----|---------------------------------|--|--|--------------|
| A1 | $B = 0 \text{ if } B \neq 1$ | | $A1' \quad B = 1 \text{ if } B \neq 0$ | Binary field |
| A2 | $\bar{0} = 1$ | | $A2' \quad \bar{1} = 0$ | NOT |
| A3 | $0 \bullet 0 = 0$ | | $A3' \quad 1 + 1 = 1$ | AND/OR |
| A4 | $1 \bullet 1 = 1$ | | $A4' \quad 0 + 0 = 0$ | AND/OR |
| A5 | $0 \bullet 1 = 1 \bullet 0 = 0$ | | $A5' \quad 1 + 0 = 0 + 1 = 1$ | AND/OR |

- **DIMOSTRAZIONE:**

- $(A \wedge B) \vee (A \wedge \neg B) = A$

- Se $B=0$, $(A \wedge B) \vee (A \wedge \neg B) = 0 \vee (A \wedge 1) = 0 \vee A = A$
- Se $B=1$, $(A \wedge B) \vee (A \wedge \neg B) = A \vee (A \wedge 0) = A \vee 0 = A$

Algebra di Boole. Teoremi su più variabili

- **Teorema 11 (Consenso):**
 - $(A \wedge B) \vee (\text{not}(A) \wedge C) \vee (B \wedge C) = (A \wedge B) \vee (\text{not}(A) \wedge C)$
- **Duale Teorema 11:**
 - $(A \vee B) \wedge (\text{not}(A) \vee C) \wedge (B \vee C) = (A \vee B) \wedge (\text{not}(A) \vee C)$

Algebra di Boole. Teoremi su più variabili

- **Teorema 11 (Consenso):**
 - $(A \wedge B) \vee (\text{not}(A) \wedge C) \vee (B \wedge C) = (A \wedge B) \vee (\text{not}(A) \wedge C)$
- **DIMOSTRAZIONE:**
 - $(A \wedge B) \vee (\text{not}(A) \wedge C) \vee (B \wedge C) = (A \wedge B) \vee (\text{not}(A) \wedge C)$
 - Se $A=1$, $(1 \wedge B) \vee (0 \wedge C) \vee (B \wedge C) = B \vee 0 \vee (B \wedge C) = B \vee (B \wedge C) = B$
 - Se $A=0$, $(0 \wedge B) \vee (1 \wedge C) \vee (B \wedge C) = 0 \vee C \vee (B \wedge C) = C \vee (B \wedge C) = C$
 - $(A \wedge B) \vee (\text{not}(A) \wedge C)$
 - Se $A=1$, $(1 \wedge B) \vee (0 \wedge C) = B \vee 0 = B$
 - Se $A=0$, $(0 \wedge B) \vee (1 \wedge C) = 0 \vee C = C$

Teoremi di de Morgan

- **Teorema 1:**
 - La negazione della congiunzione di 2 o più variabile è equivalente alla disgiunzione delle negazioni delle variabili:
 $\text{NOT}(\text{AND}(A_1, A_2, \dots, A_N)) = \text{OR}(\text{NOT}(A_1), \text{NOT}(A_2), \dots, \text{NOT}(A_N))$
- **Teorema 2:**
 - La negazione della disgiunzione di 2 o più variabile è equivalente alla congiunzione delle negazioni delle variabili:
 $\text{NOT}(\text{OR}(A_1, A_2, \dots, A_N)) = \text{AND}(\text{NOT}(A_1), \text{NOT}(A_2), \dots, \text{NOT}(A_N))$

Teoremi di de Morgan

- **Teorema 1:**

- La negazione della congiunzione di 2 o più variabile è equivalente alla disgiunzione delle negazioni delle variabili:

$$\text{NOT}(\text{AND}(A_1, A_2, \dots, A_N)) = \text{OR}(\text{NOT}(A_1), \text{NOT}(A_2), \dots, \text{NOT}(A_N))$$

$\text{NOT}(\text{AND}(A, B))$

| A | B | Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

$\text{OR}(\text{NOT}(A), \text{NOT}(B))$

| NOT(A) | NOT(B) | Y |
|--------|--------|---|
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

Teoremi di de Morgan

- **Teorema 2:**
 - La negazione della disgiunzione di 2 o più variabile è equivalente alla congiunzione delle negazioni delle variabili:
 $\text{NOT}(\text{OR}(A_1, A_2, \dots, A_N)) = \text{AND}(\text{NOT}(A_1), \text{NOT}(A_2), \dots, \text{NOT}(A_N))$

$\text{NOT}(\text{OR}(A, B))$

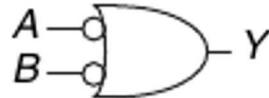
| A | B | Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

$\text{AND}(\text{NOT}(A), \text{NOT}(B))$

| NOT(A) | NOT(B) | Y |
|--------|--------|---|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

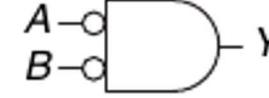
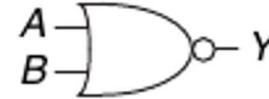
Teoremi di de Morgan

NAND



$$Y = \overline{AB} = \overline{A} + \overline{B}$$

NOR



$$Y = \overline{A+B} = \overline{A} \overline{B}$$

- UN NAND (NOT di un AND) è equivalente al OR con input negati (si dice che i due simboli sono duali)
- UN NOR (NOT di un OR) è equivalente ad un AND con input negati (si dice che i due simboli sono duali)

Riassumendo

| Theorem | Dual | Name |
|---------------------------------|-------------------------------|--------------|
| T1 $B \bullet 1 = B$ | T1' $B + 0 = B$ | Identity |
| T2 $B \bullet 0 = 0$ | T2' $B + 1 = 1$ | Null Element |
| T3 $B \bullet B = B$ | T3' $B + B = B$ | Idempotency |
| T4 | $\overline{\overline{B}} = B$ | Involution |
| T5 $B \bullet \overline{B} = 0$ | T5' $B + \overline{B} = 1$ | Complements |

| Theorem | Dual | Name |
|--|---|------------------------|
| T6 $B \bullet C = C \bullet B$ | T6' $B + C = C + B$ | Commutativity |
| T7 $(B \bullet C) \bullet D = B \bullet (C \bullet D)$ | T7' $(B + C) + D = B + (C + D)$ | Associativity |
| T8 $(B \bullet C) + (B \bullet D) = B \bullet (C + D)$ | T8' $(B + C) \bullet (B + D) = B + (C \bullet D)$ | Distributivity |
| T9 $B \bullet (B + C) = B$ | T9' $B + (B \bullet C) = B$ | Covering |
| T10 $(B \bullet C) + (B \bullet \overline{C}) = B$ | T10' $(B + C) \bullet (B + \overline{C}) = B$ | Combining |
| T11 $(B \bullet C) + (\overline{B} \bullet D) + (C \bullet D) \\ = B \bullet C + \overline{B} \bullet D$ | T11' $(B + C) \bullet (\overline{B} + D) \bullet (C + D) \\ = (B + C) \bullet (\overline{B} + D)$ | Consensus |
| T12 $\overline{B_0 \bullet B_1 \bullet B_2 \dots} \\ = (\overline{B_0} + \overline{B_1} + \overline{B_2} \dots)$ | T12' $\overline{B_0 + B_1 + B_2 \dots} \\ = (\overline{B_0} \bullet \overline{B_1} \bullet \overline{B_2} \dots)$ | De Morgan's Theorem |

A cosa ci servono i teoremi introdotti fin qui?

- Tramite i teoremi precedentemente visti possiamo semplificare le espressioni Booleane.
- Ottenendo così circuiti equivalenti più semplici

Architettura degli Elaboratori

Lezione 13

Docente: R.Prevete
a.a. 2022/2023
2023/04/05

A cosa ci servono i teoremi introdotti fin qui?

- Tramite i teoremi precedentemente visti possiamo semplificare le espressioni Booleane.
- Ottenendo così circuiti equivalenti più semplici

Esempio

Tabella Verità

| A | B | C | Y |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

**Espressione Booleana in forma cananonica
come somma di minterm:**

- $Y = \text{not}(A) \text{ not}(B) \text{ not}(C) + A \text{ not}(B) \text{ not}(C) + A \text{ not}(B) C$

Esempio

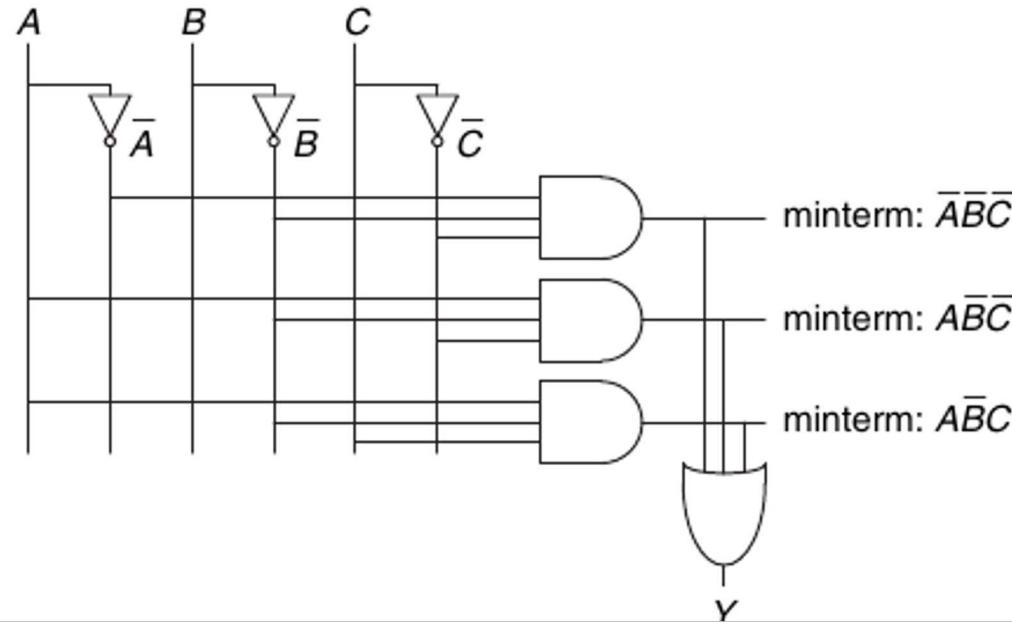
- $\text{not}(A) \text{ not}(B) \text{ not}(C) + A \text{ not}(B) \text{ not}(C) + A \text{ not}(B) C$
- $\text{not}(B) (\text{not}(A)\text{not}(C) + A \text{ not}(C) + A C)$ (distributiva)
- $\text{not}(B) (\text{not}(C) (\text{not}(A)+A) + A C)$ (distributiva)
- $\text{not}(B) (\text{not}(C)(1) + A C)$ (complemento)
- $\text{not}(B) (\text{not}(C) + A C)$ (identità)
- $\text{not}(B)(\text{not}(C)+A)$ (assorbimento)
- $\text{not}(B)\text{not}(C)+A\text{not}(B)$ in termini di somma di prodotti

Altro modo di procedere:

- $\text{not}(A) \text{ not}(B) \text{ not}(C) + A \text{ not}(B) \text{ not}(C) + A \text{ not}(B) C$
- $\text{not}(B) (\text{not}(A)\text{not}(C) + A \text{ not}(C) + A C)$ (distributiva)
- $\text{not}(B) (\text{not}(A)\text{not}(C) + A \text{ not}(C) + A \text{ not}(C) + A C)$ (idempotenza)
- $\text{not}(B) (\text{not}(C) (\text{not}(A) +A) + A (\text{not}(C) + C))$ (distributiva)
- $\text{not}(B) (\text{not}(C) (1) + A (1))$ (complemento)
- $\text{not}(B) (\text{not}(C) + A)$ (identità)
- $\text{not}(B)\text{not}(C)+A\text{not}(B)$ in termini di somma di prodotti

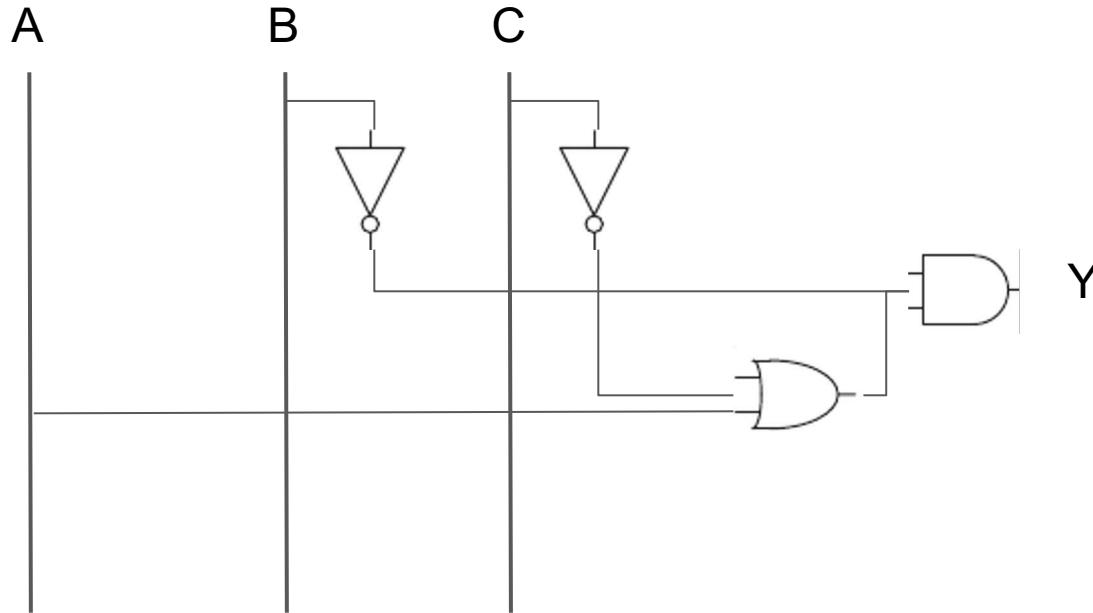
In termini di porte logiche

- $\text{not}(A) \text{ not}(B) \text{ not}(C) + A \text{ not}(B) \text{ not}(C) + A \text{ not}(B) C$



In termini di porte logiche

- $\text{not}(A) \text{ not}(B) \text{ not}(C) + A \text{ not}(B) \text{ not}(C) + A \text{ not}(B) C =$
 - $\text{not}(B) (\text{not}(C) + A)$



Semplificazione di espressioni: esempi

- $Y = AC + \text{not}(A) \text{ not}(B) C$
 - $(A + \text{not}(A)) \text{ not}(B) C$ distributiva
 - $(A(\text{not}(B)+1) + \text{not}(A)) \text{ not}(B) C$ elemento nullo
 - $A = A * 1 = A * (1 + \text{not}(B))$
 - $(\text{not}(B) + A + \text{not}(A)) \text{ not}(B) C$ distributiva
 - $((A + \text{not}(A)) \text{ not}(B) + A) C$ distributiva
 - $(\text{not}(B) + A) C$ complemento
 - $\text{not}(B) C + AC$ in termini somma di prodotti

Semplificazione di espressioni: verifico

$$Y = AC + \text{not}(A) \text{ not}(B) C$$

- $\text{not}(B)C + AC$

versione semplificata

Tabella Verità

| A | B | C | Output |
|---|---|---|--------|
| 0 | 0 | 0 | F |
| 0 | 0 | 1 | T |
| 0 | 1 | 0 | F |
| 0 | 1 | 1 | F |
| 1 | 0 | 0 | F |
| 1 | 0 | 1 | T |
| 1 | 1 | 0 | F |
| 1 | 1 | 1 | T |

- Dalla **forma canonica somma prodotti**:

- $\text{not}(A)\text{not}(B)C + A\text{not}(B)C + ABC$
- $(\text{not}(A)\text{not}(B) + A\text{not}(B) + AB)C$
- $(\text{not}(A)\text{not}(B) + A(\text{not}(B) + B))C$
- $(\text{not}(A)\text{not}(B) + A)C$
- $(\text{not}(A)\text{not}(B) + A(\text{not}(B) + 1))C$
- $(\text{not}(A)\text{not}(B) + A\text{not}(B) + A)C$
- $((\text{not}(A) + A)\text{not}(B) + A)C$
- $(\text{not}(B) + A)C$

distr.

distr.

complemento

elemento nullo

distr

distr,elemento nullo
complemento

Semplificazione di espressioni: esempi

- $Y = \neg(A) \neg(B) + \neg(A)B \neg(C) + \neg(A + \neg(C))$
 - $\neg(A)\neg(B) + \neg(A)B \neg(C) + \neg(A)C$ De Morgan 2
 - $\neg(A)(\neg(B) + B \neg(C) + C)$ distr.
 - $\neg(A)(\neg(B) + B \neg(C) + C(B+1))$ null element (duale)
 - $\neg(A)(\neg(B) + B \neg(C) + CB+C)$ distr
 - $\neg(A)(\neg(B) + B+C)$ distr, compl (si può ottenere direttamente per assormimento: $B \neg(C) + C = B+C$)
 - $\neg(A)(1+C)$ compl
 - $\neg(A)$ null element (duale)

Semplificazione di espressioni: verifico

- $Y = \text{not}(A) \text{ not}(B) + \text{not}(A)B\text{not}(C) + \text{not}(A + \text{not}(C))$
 - $\text{not}(A)$ versione semplifica

Tabella Verità

| A | B | C | Output |
|---|---|---|--------|
| 0 | 0 | 0 | T |
| 0 | 0 | 1 | T |
| 0 | 1 | 0 | T |
| 0 | 1 | 1 | T |
| 1 | 0 | 0 | F |
| 1 | 0 | 1 | F |
| 1 | 1 | 0 | F |
| 1 | 1 | 1 | F |

Semplificazione di espressioni: esempi

- $Y = \text{not}(A)BC + \text{not}(A)B\text{not}(C)$
 - $\text{not}(A)B(C + \text{not}(C))$ distributiva
 - $\text{not}(A)B$ complemento

Tabella di
Verità

| A | B | C | Output |
|---|---|---|--------|
| 0 | 0 | 0 | F |
| 0 | 0 | 1 | F |
| 0 | 1 | 0 | T |
| 0 | 1 | 1 | T |
| 1 | 0 | 0 | F |
| 1 | 0 | 1 | F |
| 1 | 1 | 0 | F |
| 1 | 1 | 1 | F |

Semplificazione di espressioni: esempi

- $Y = \text{not}(ABC) + A \text{ not}(B)$
 - $\text{not}(A)+\text{not}(B) +\text{not}(C)+A \text{ not}(B)$ De Morgan 1
 - $\text{not}(A) +\text{not}(C) +\text{not}(B)(1+A)$ distr.
 - $\text{not}(A) +\text{not}(C) +\text{not}(B)$ null element (duale)

Semplificazione di espressioni: esempi

- $Y = \text{not}(ABC) + A \text{ not}(B)$
 - forma semplificata: $\text{not}(A) + \text{not}(C) + \text{not}(B)$

Tabella Verità

| A | B | C | Output |
|---|---|---|--------|
| 0 | 0 | 0 | T |
| 0 | 0 | 1 | T |
| 0 | 1 | 0 | T |
| 0 | 1 | 1 | T |
| 1 | 0 | 0 | T |
| 1 | 0 | 1 | T |
| 1 | 1 | 0 | T |
| 1 | 1 | 1 | F |

- Forma canonica come prodotto di somme
 - Max-term:
 - $\text{not}(A)+\text{not}(B)+\text{not}(C)$
- Ricordo che prendo il prodotto dei maxterm corrispondenti a FALSE, ed il maxterm è l'OR dei valori corrispondenti delle variabili di ingresso in modo da dare FALSE

Semplificazione di espressioni: esempi

- $Y = ABC \text{ not}(D) + A \text{not}(BCD) + \text{not}(ABCD)$
 - $ABC \text{ not}(D) + A(\text{not}(B) + \text{not}(C) + \text{not}(D)) + \text{not}(A) + \text{not}(B) + \text{not}(C) + \text{not}(D)$
De Morgan 1
 - $\text{ABC not}(D) + \text{Anot}(B) + \text{Anot}(C) + \text{Anot}(D) + \text{not}(A) + \text{not}(B) + \text{not}(C) + \text{not}(D)$
distr.
 - $\text{not}(D)(ABC+A+1) + \text{not}(B)(A+1) + \text{not}(C)(A+1) + \text{not}(A)$ distr.
 - $\text{not}(A) + \text{not}(B) + \text{not}(C) + \text{not}(D)$ null element

Semplificazione di espressioni: esempi

| A | B | C | D | Output |
|---|---|---|---|--------|
| 0 | 0 | 0 | 0 | T |
| 0 | 0 | 0 | 1 | T |
| 0 | 0 | 1 | 0 | T |
| 0 | 0 | 1 | 1 | T |
| 0 | 1 | 0 | 0 | T |
| 0 | 1 | 0 | 1 | T |
| 0 | 1 | 1 | 0 | T |
| 0 | 1 | 1 | 1 | T |
| 1 | 0 | 0 | 0 | T |
| 1 | 0 | 0 | 1 | T |
| 1 | 0 | 1 | 0 | T |
| 1 | 0 | 1 | 1 | T |
| 1 | 1 | 0 | 0 | T |
| 1 | 1 | 0 | 1 | T |
| 1 | 1 | 1 | 0 | T |
| 1 | 1 | 1 | 1 | F |

- $Y = ABC \text{ not}(D) + A \text{not}(BCD) + \text{not}(ABCD)$
 - forma semplificata: $\text{not}(A) + \text{not}(B) + \text{not}(C) + \text{not}(D)$
 - Forma canonica come prodotto di somme
 - Max-term:
 - $\text{not}(A) + \text{not}(B) + \text{not}(C) + \text{not}(D)$

Tabella Verità

Semplificazione di espressioni: esempi

- $Y = \text{not}(A + \text{not}(A)B + \text{not}(A)\text{not}(B)) + \text{not}(A + \text{not}(B))$
 - $Y = \text{not}(A)\text{not}(\text{not}(A)B)\text{not}(\text{not}(A)\text{not}(B)) + \text{not}(A)\text{not}(\text{not}(B))$ De Mor. 2
 - $Y = \text{not}(A)(A + \text{not}(B))(A + B) + \text{not}(A)B$ De Mor. 1
 - $\text{not}(A)AA + \text{not}(A)\text{not}(B)A + \text{not}(A)AB + \text{not}(A)\text{not}(B)B + \text{not}(A)B$ distr.
 - $0 + 0 + 0 + 0 + \text{not}(A)B$
 - $\text{not}(A)B$ null element

Semplificazione di espressioni: esempi

- $Y = \text{not}(A + \text{not}(A)B + \text{not}(A)\text{not}(B)) + \text{not}(A + \text{not}(B))$
 - forma semplificata: $\text{not}(A)B$

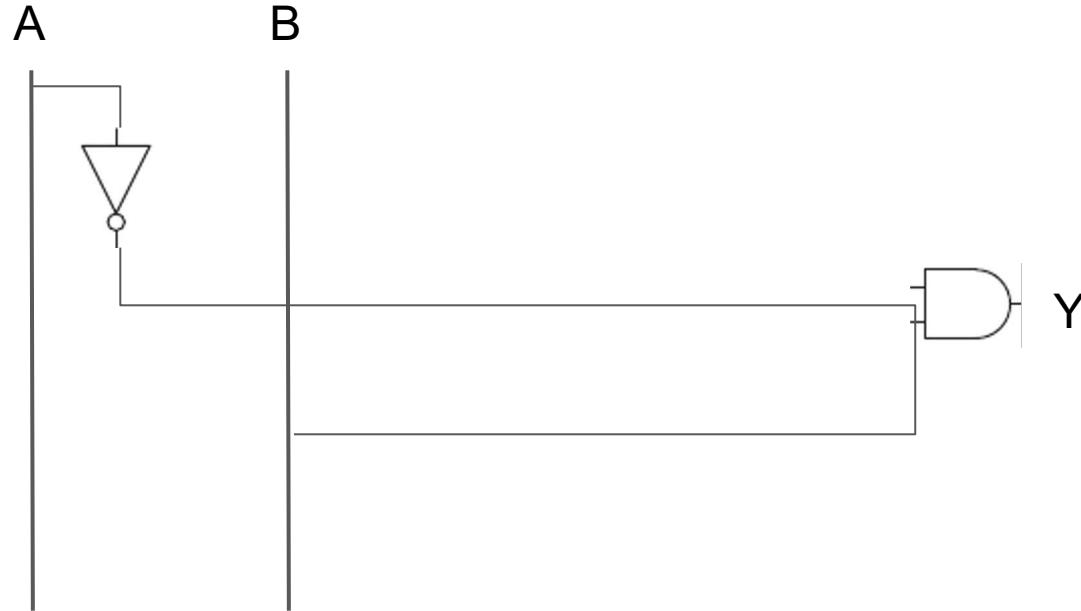
| A | B | Output |
|---|---|--------|
| 0 | 0 | F |
| 0 | 1 | T |
| 1 | 0 | F |
| 1 | 1 | F |

- Forma canonica come somma di prodotti
 - minterm:
 - $\text{not}(A)B$

Tabella Verità

Esempio di realizzazione in termini di circuito

- $Y = \text{not}(A + \text{not}(A)B + \text{not}(A)\text{not}(B)) + \text{not}(A + \text{not}(B))$
 - forma semplificata: $\text{not}(A)B$



Creazione e semplificazione di espressioni Booleane

Abbiamo visto:

- Creazione di espressioni da tabelle di verità
 - Forma Canonica come somma di prodotti di minterm
 - Forma Canonica come prodotto di maxtem
- Semplificare espressioni tramite i teoremi dell'algebra Booleana

Introduzione di un metodo grafico/visivo per aiutarsi nel semplificare le espressioni:

- **Mappe di Karnaugh** (o Karnaugh maps o K-maps)

Esercizi

Dal libro:

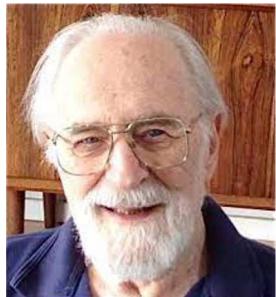
- Es 2.2
- Es 2.3
- Es 2.5
- Es 2.6
- Es. 2.7

Architettura degli Elaboratori

Lezione 14

Docente: R.Prevete
a.a. 2022/2023
12 aprile 2023

Mappe di Karnaugh



Maurice Karnaugh (Kaggi, 4 ottobre 1924) è un ingegnere delle telecomunicazioni statunitense. Laureato all'università Yale nel 1952, è attualmente governatore emerito dell'ICCC, (International Council for Computer Communication).

Ha lavorato come ricercatore ai Laboratori Bell dal 1952 al 1966 e al centro di ricerca IBM dal 1966 al 1993. Ha insegnato informatica al Politecnico della New York University dal 1980 al 1999, e dal 1975 è membro dell'IEEE per i suoi lavori sull'utilizzo di tecniche numeriche nelle telecomunicazioni. *Wikipedia*

- Mappe di Karnaugh sono state introdotte nel 1953
- Sfruttano le regole per la semplificazione di espressioni Booleane
- Sfruttano un approccio grafico/visivo

Alcuni richiami ...

- **Implicante:** Il prodotto di più letterali (variabili) in forma true o complemento
 - Es. ABC, not(A)C, Anot(B)CD
- **Implicante primo** (prime implicant): In una espressione Booleana un implicante è chiamato **primo implicante se non** può essere combinato con un altro implicante della formula per formare un **implicante con meno letterali**
- **Equazione/espressione minimale:** una espressione (o equazione) minimale è formata **tutti i implicanti primi**

Alcuni richiami ...

- **Implicante primo** (prime implicant): In una espressione Booleana un implicante è chiamato primo implicante se non può essere combinato con un altro implicante della formula per formare un implicante con meno letterali
- **Esempi:**
 - ABC+AB non sono implicanti primi, infatti:
 - $ABC+AB=AB(C+1)=AB$ posso ridurre ottenendo un implicante con meno letterali
 - AB+BC sono entrambi implicanti primi, infatti anche se posso combinare gli implicanti, non riduco i letterali $AB+BC =B(A+C)$
 - AB +not(A)C sono entrambi implicanti primi, non possono essere combinati
 - AB+BC+Cnot(B)
 - AB implicante primo, BC e Cnot(B) solo implicanti (non primi)

Alcuni richiami ...

- **Equazione/espressione minimale:** una espressione (o equazione) minimale è formata **tutti implicanti primi**
 - $AB+BC+C\text{not}(B)$ non è minimale
 - $AB+BC+C\text{not}(B) = AB+C(B+\text{not}(B))= AB+C$
 - $AB+C$ è la sua forma minimale
- $AB +\text{not}(A)C$
 - è già in forma minimale

K-mappe...un semplice esempio

| A | B | C | Output |
|---|---|---|--------|
| 0 | 0 | 0 | T |
| 0 | 0 | 1 | T |
| 0 | 1 | 0 | F |
| 0 | 1 | 1 | F |
| 1 | 0 | 0 | F |
| 1 | 0 | 1 | F |
| 1 | 1 | 0 | F |
| 1 | 1 | 1 | F |

Primo passo: **costruzione di una tabella**

| | | AB | | C | | |
|---|---|----|----|----|----|----|
| | | Y | 00 | 01 | 11 | 10 |
| Y | 0 | | | | | |
| | 1 | | | | | |

- Prima riga i possibili valori delle due variabili AB
- Prima colonna i possibili valori della variabile C
- **NOTA:** C'è un ordine specifico, configurazioni adiacenti differiscono solo per il valore di una unica variabile (**Gray code**)

K-mappe: grey code

- **Gray code:**

- 00, 01, 10, 11 NON va bene!!
 - 01 e 10 differiscono per il valore di più che una variabile
- 000, 001, 101, 100, 110, 010, 011, 111 gray code!!

| | | AB | | | |
|---|---|--------|------|------|------|
| | | 00 | 01 | 11 | 10 |
| C | 0 | Yellow | Blue | Blue | Blue |
| | 1 | Yellow | | | |

Con il gray code:

- caselle contigue sulla stessa **riga** possono differire solo del valore di una variabile
- caselle contigue sulla stessa **colonna** possono differire solo del valore di una variabile

K-mappe: grey code

Con il gray code:

- caselle contigue sulla stessa **riga** possono differire solo nel valore di una variabile
- caselle contigue sulla stessa **colonna** possono differire solo del valore di una variabile

| | | AB | | | | | |
|---|--|----|----|----|----|----|---|
| | | Y | 00 | 01 | 11 | 10 | |
| C | | 0 | ■ | | | | ■ |
| | | 1 | | | | | |

| | | AB | | | | |
|---|--|----|----|----|----|----|
| | | Y | 00 | 01 | 11 | 10 |
| C | | 0 | | | ■ | |
| | | 1 | | | | |

Vero anche se lo vedo in maniera circolare:

- Dall'ultima colonna salto alla prima e viceversa
- Dall'ultima riga salto alla prima e viceversa

K-mappe...un semplice esempio

| A | B | C | Output |
|---|---|---|--------|
| 0 | 0 | 0 | T |
| 0 | 0 | 1 | T |
| 0 | 1 | 0 | F |
| 0 | 1 | 1 | F |
| 1 | 0 | 0 | F |
| 1 | 0 | 1 | F |
| 1 | 1 | 0 | F |
| 1 | 1 | 1 | F |

Primo passo: **costruzione di una tabella**

AB

C

| Y | 00 | 01 | 11 | 10 |
|---|----|----|----|----|
| 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |

- I valori di ciascuna cella corrispondono ai valori di output della riga corrispondente nella tabella di verità

K-mappe...un semplice esempio

| A | B | C | Output |
|---|---|---|--------|
| 0 | 0 | 0 | T |
| 0 | 0 | 1 | T |
| 0 | 1 | 0 | F |
| 0 | 1 | 1 | F |
| 1 | 0 | 0 | F |
| 1 | 0 | 1 | F |
| 1 | 1 | 0 | F |
| 1 | 1 | 1 | F |

Secondo passo: **costruzione degli implicanti primi**

AB

| Y | 00 | 01 | 11 | 10 |
|---|----|----|----|----|
| | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |

**Cerchio i minterm
che devono
apparire nella
forma canonica**

- Formula minimale composta da implicanti primi
 - Formula canonica somma di prodotti:
 - $\text{not}(A)\text{not}(B)\text{not}(C) + C\text{not}(A)\text{not}(B)$
 - $\text{not}(A)\text{not}(B)(\text{not}(C)+C)$
 - $\text{not}(A)\text{not}(B)$ -- teorema complemento

K-mappe...un semplice esempio

| A | B | C | Output |
|---|---|---|--------|
| 0 | 0 | 0 | T |
| 0 | 0 | 1 | T |
| 0 | 1 | 0 | F |
| 0 | 1 | 1 | F |
| 1 | 0 | 0 | F |
| 1 | 0 | 1 | F |
| 1 | 1 | 0 | F |
| 1 | 1 | 1 | F |

Secondo passo: **costruzione degli implicanti primi**

| Y | AB | | | |
|---|----|----|----|----|
| | 00 | 01 | 11 | 10 |
| C | 0 | 1 | 0 | 0 |
| | 1 | 1 | 0 | 0 |

- Formula minimale composta da implicanti primi
 - Formula canonica somma di prodotti:
 - $\text{not}(A)\text{not}(B)$

Caselle adiacenti si differenziano solo per il valore di una variabile, quindi i corrispondenti minterm presentano la stessa variabile in forma true e in forma complemento questo fa sì che il letterale scompare

$$\text{not}(A)\text{not}(B)\text{not}(C) + C\text{not}(A)\text{not}(B) \rightarrow \text{not}(A)\text{not}(B)$$

K-mappe...un semplice esempio

| A | B | C | Output |
|---|---|---|--------|
| 0 | 0 | 0 | T |
| 0 | 0 | 1 | T |
| 0 | 1 | 0 | F |
| 0 | 1 | 1 | F |
| 1 | 0 | 0 | F |
| 1 | 0 | 1 | F |
| 1 | 1 | 0 | F |
| 1 | 1 | 1 | F |

Ultimo passo: **costruzione della formula minimale**

AB

C

| Y | 00 | 01 | 11 | 10 |
|---|----|----|----|----|
| 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |

$$Y = \text{not}(A)\text{not}(B)$$

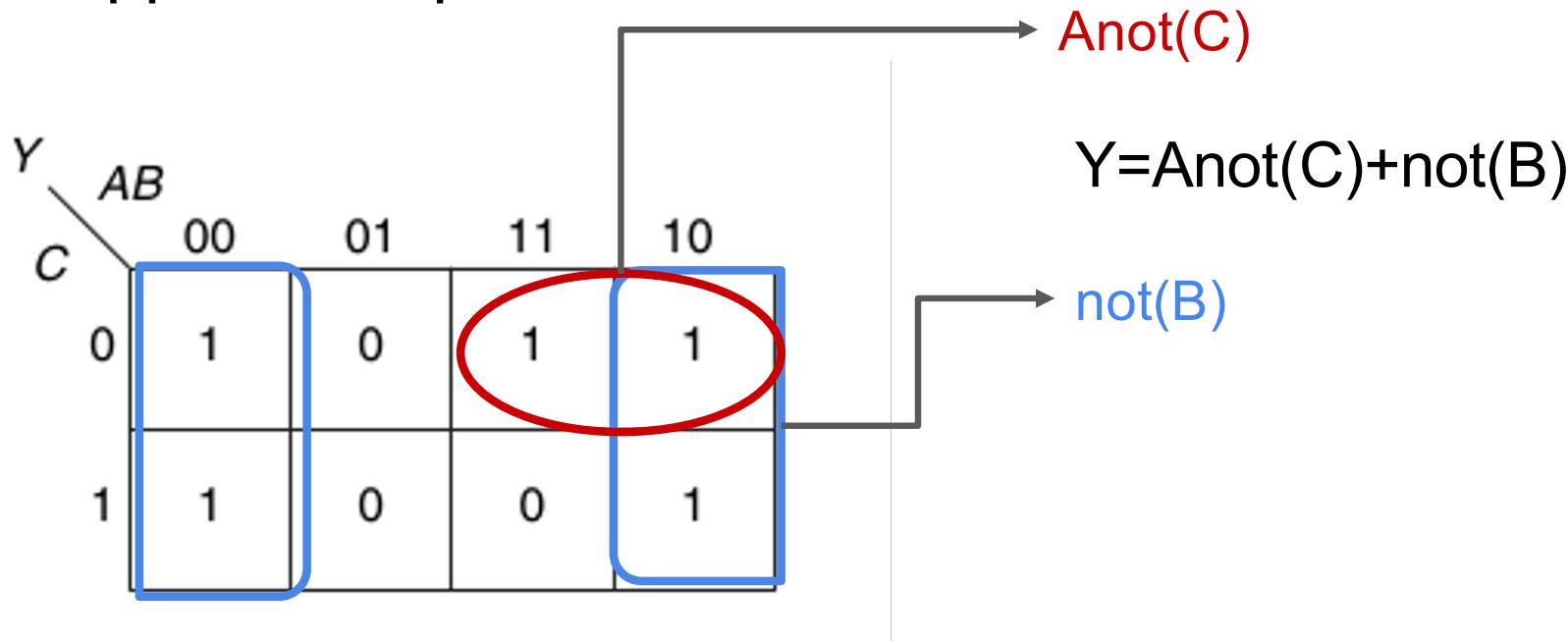
- Ricordiamo che una espressione Booleana è minimizzata quando è scritta come somma del più piccolo numero di implicanti primi
- **Quindi l'idea è:**
 - Cerchi che ricoprono quante più caselle contigue con valore 1 (corrispondenti ad un unico implicantе primo)
 - Quanto meno cerchi possibili (più piccolo numero di implicanti)

K-mappe: regole

- k1: Usare quanto meno cerchi possibili per ricoprire **tutte** le caselle con 1
- k2: In ciascun cerchio tutte le caselle devono contenere 1
- k3: Ciascun cerchio deve ricoprire un numero di caselle che sia una potenza di 2 (sia lungo le colonne sia lungo le righe)
- k4: Ciascun cerchio deve essere quanto più grande possibile
- k5: La tabella è vista come circolare, quindi un cerchio può passare da destra a sinistra (e viceversa) o dall'alto al basso (e viceversa)
- k6: Una casella può essere inclusa in più cerchi se così facendo si riducono il numero di cerchi

Formula minima risultante → somma degli implicanti primi corrispondenti a ciascun cerchio

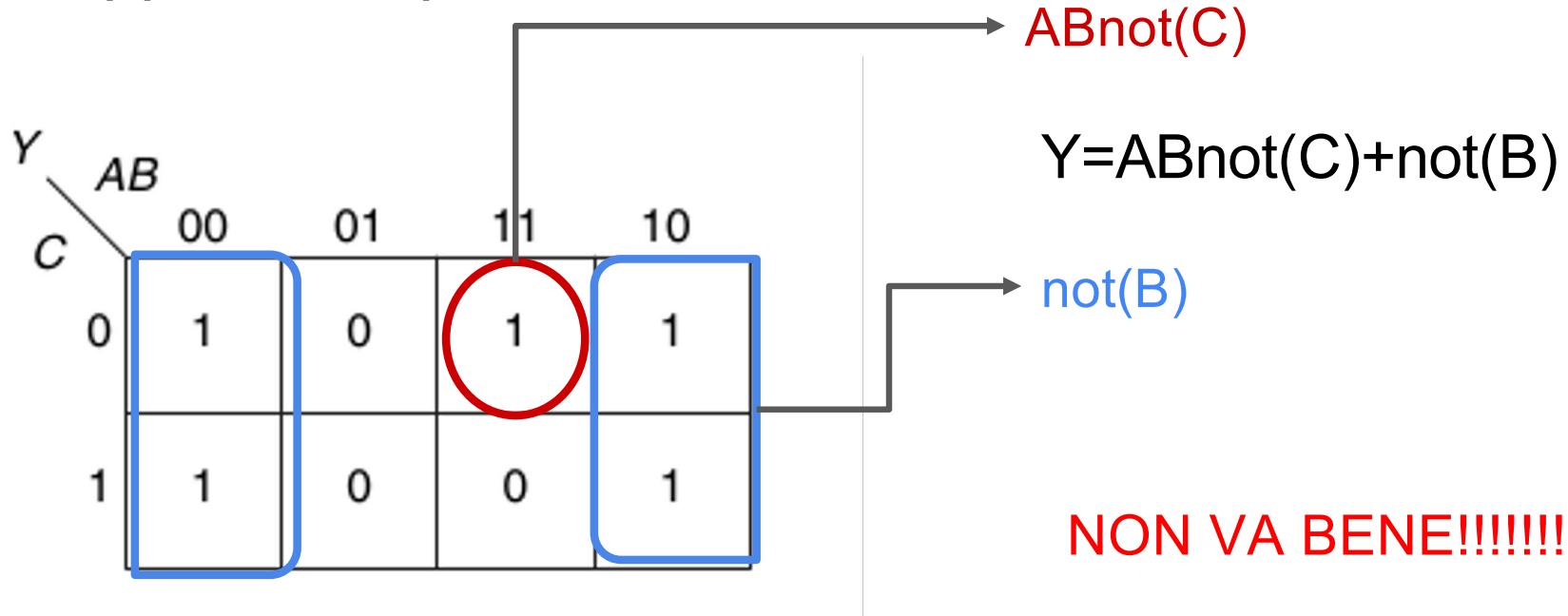
K-mappe: esempio



NOTA:

- Il cerchio blu è circolare (k5)
- I cerchi sono il più grande possibile (k4)
- I cerchi ricoprono solo caselle con 1 (k2)

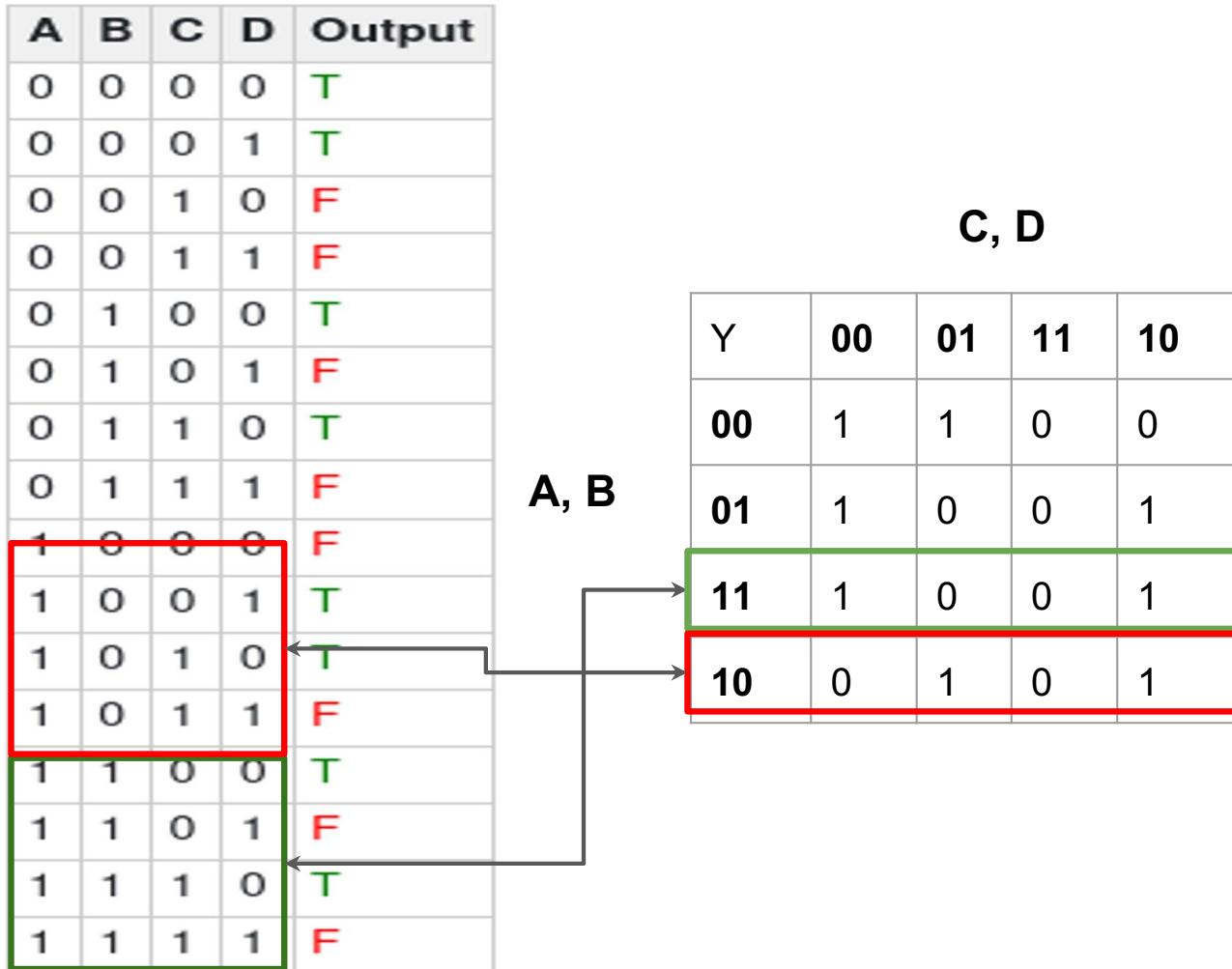
K-mappe: esempio



NOTA:

- Il cerchio blu è circolare (k5)
- I cerchi sono il più grande possibile (k4) **NON RISPETTATO!**
- I cerchi ricoprono solo caselle con 1 (k2)

K-mappe: partiamo da tabella verità



- In questo caso 4 variabili
- **Costruiamo la tabella mantenendo il gray code**

K-mappe: partiamo da tabella verità

| A | B | C | D | Output |
|---|---|---|---|--------|
| 0 | 0 | 0 | 0 | T |
| 0 | 0 | 0 | 1 | T |
| 0 | 0 | 1 | 0 | F |
| 0 | 0 | 1 | 1 | F |
| 0 | 1 | 0 | 0 | T |
| 0 | 1 | 0 | 1 | F |
| 0 | 1 | 1 | 0 | T |
| 0 | 1 | 1 | 1 | F |
| 1 | 0 | 0 | 0 | F |
| 1 | 0 | 0 | 1 | T |
| 1 | 0 | 1 | 0 | T |
| 1 | 0 | 1 | 1 | F |
| 1 | 1 | 0 | 0 | T |
| 1 | 1 | 0 | 1 | F |
| 1 | 1 | 1 | 0 | T |
| 1 | 1 | 1 | 1 | F |

A, B

| | C, D | | | |
|----|------|----|----|----|
| Y | 00 | 01 | 11 | 10 |
| 00 | 1 | 1 | 0 | 0 |
| 01 | 1 | 0 | 0 | 1 |
| 11 | 1 | 0 | 0 | 1 |
| 10 | 0 | 1 | 0 | 1 |

- Costruiamo i cerchi
 - massima ampiezza
 - potenze di 2
 - circolari

K-mappe: partiamo da tabella verità

| A | B | C | D | Output |
|---|---|---|---|--------|
| 0 | 0 | 0 | 0 | T |
| 0 | 0 | 0 | 1 | T |
| 0 | 0 | 1 | 0 | F |
| 0 | 0 | 1 | 1 | F |
| 0 | 1 | 0 | 0 | T |
| 0 | 1 | 0 | 1 | F |
| 0 | 1 | 1 | 0 | T |
| 0 | 1 | 1 | 1 | F |
| 1 | 0 | 0 | 0 | F |
| 1 | 0 | 0 | 1 | T |
| 1 | 0 | 1 | 0 | T |
| 1 | 0 | 1 | 1 | F |
| 1 | 1 | 0 | 0 | T |
| 1 | 1 | 0 | 1 | F |
| 1 | 1 | 1 | 0 | T |
| 1 | 1 | 1 | 1 | F |

A, B

| | | C, D | | | | |
|------|----|------|----|----|----|----|
| | | Y | 00 | 01 | 11 | 10 |
| A, B | 00 | 1 | 1 | 0 | 0 | 0 |
| | | 1 | 0 | 0 | 1 | |
| A, B | 01 | 1 | 0 | 0 | 0 | 1 |
| | | 1 | 0 | 0 | 1 | |
| A, B | 11 | 1 | 0 | 0 | 0 | 1 |
| | | 0 | 1 | 0 | 1 | |
| A, B | 10 | 0 | 1 | 0 | 1 | |
| | | 1 | 0 | 1 | 1 | |



- Costruiamo gli implicanti primi
 - $\text{not}(B)\text{not}(C)D$
 - $B\text{not}(D)$
 - $\text{not}(A)\text{not}(B)\text{not}(C)$
 - $A\text{Cnot}(D)$

K-mappe: partiamo da tabella verità

| A | B | C | D | Output |
|---|---|---|---|--------|
| 0 | 0 | 0 | 0 | T |
| 0 | 0 | 0 | 1 | T |
| 0 | 0 | 1 | 0 | F |
| 0 | 0 | 1 | 1 | F |
| 0 | 1 | 0 | 0 | T |
| 0 | 1 | 0 | 1 | F |
| 0 | 1 | 1 | 0 | T |
| 0 | 1 | 1 | 1 | F |
| 1 | 0 | 0 | 0 | F |
| 1 | 0 | 0 | 1 | T |
| 1 | 0 | 1 | 0 | T |
| 1 | 0 | 1 | 1 | F |
| 1 | 1 | 0 | 0 | T |
| 1 | 1 | 0 | 1 | F |
| 1 | 1 | 1 | 0 | T |
| 1 | 1 | 1 | 1 | F |

C, D

| Y | 00 | 01 | 11 | 10 |
|----|----|----|----|----|
| 00 | 1 | 1 | 0 | 0 |
| 01 | 1 | 0 | 0 | 1 |
| 11 | 1 | 0 | 0 | 1 |
| 10 | 0 | 1 | 0 | 1 |

A, B

$$Y = \text{not}(B)\text{not}(C)D + \text{not}(A)\text{not}(B)\text{not}(C) + B\text{not}(D) + A\text{Cnot}(D)$$

- Costruiamo la formula minimale
 - $\text{not}(B)\text{not}(C)D$
 - $B\text{not}(D)$
 - $\text{not}(A)\text{not}(B)\text{not}(C)$
 - $A\text{Cnot}(D)$

Architettura degli Elaboratori

Lezione 15

Docente: R.Prevete
a.a. 2022/2023
14 aprile 2023

K-maps: altro esempio

| A | B | C | Output |
|---|---|---|--------|
| 0 | 0 | 0 | F |
| 0 | 0 | 1 | T |
| 0 | 1 | 0 | F |
| 0 | 1 | 1 | F |
| 1 | 0 | 0 | F |
| 1 | 0 | 1 | T |
| 1 | 1 | 0 | F |
| 1 | 1 | 1 | T |

In una lezione precedente, abbiamo semplificato l'espressione

- $Y = AC + \text{not}(A) \text{ not}(B) C$

in

- $Y = \text{not}(B)C + AC$

usando i teoremi dell'algebra Booleana

Proviamo ora anche ad utilizzare le mappe di Karnaugh per verificare il risultato raggiunto

K-maps: altro esempio

| A | B | C | Output |
|---|---|---|--------|
| 0 | 0 | 0 | F |
| 0 | 0 | 1 | T |
| 0 | 1 | 0 | F |
| 0 | 1 | 1 | F |
| 1 | 0 | 0 | F |
| 1 | 0 | 1 | T |
| 1 | 1 | 0 | F |
| 1 | 1 | 1 | T |

- $Y = AC + \text{not}(A) \text{ not}(B) C$
semplificata in
- $Y = \text{not}(B)C + AC$

A, B

C

| Y | 00 | 01 | 11 | 10 |
|---|----|----|----|----|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |

Creazione tabella

K-maps: altro esempio

| A | B | C | Output |
|---|---|---|--------|
| 0 | 0 | 0 | F |
| 0 | 0 | 1 | T |
| 0 | 1 | 0 | F |
| 0 | 1 | 1 | F |
| 1 | 0 | 0 | F |
| 1 | 0 | 1 | T |
| 1 | 1 | 0 | F |
| 1 | 1 | 1 | T |

- $Y = AC + \text{not}(A) \text{ not}(B) C$
semplificata in
- $Y = \text{not}(B)C + AC$

A, B

| C | Y | 00 | 01 | 11 | 10 |
|---|---|----|----|----|----|
| | | 0 | 0 | 0 | 0 |
| A | 1 | 1 | 0 | 1 | 1 |
| | 0 | 0 | 0 | 0 | 0 |

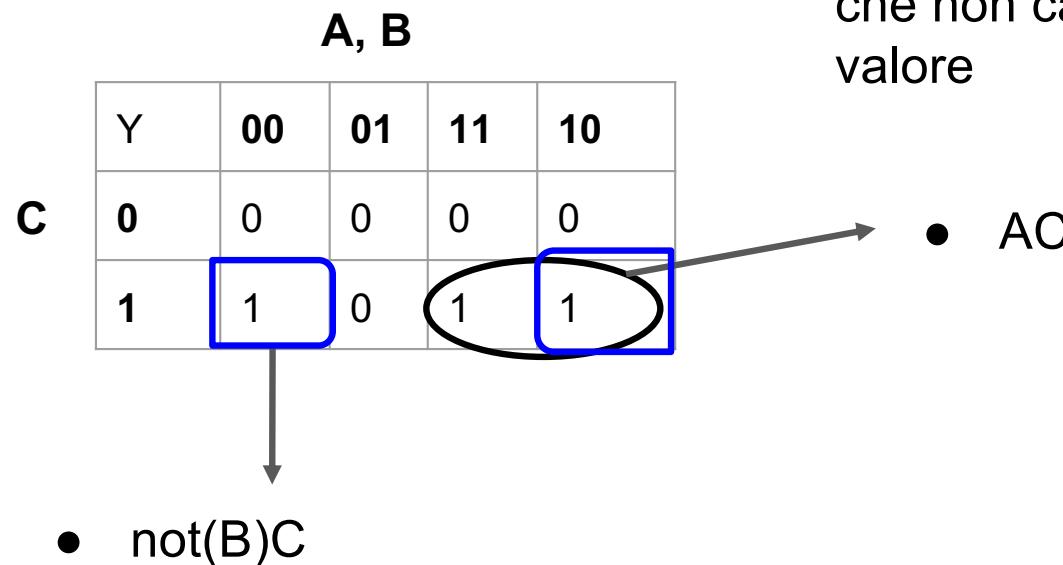
Creo cerchi

- Ricordo che devono coprire un numero di caselle pari a potenze di 2

K-maps: altro esempio

| A | B | C | Output |
|---|---|---|--------|
| 0 | 0 | 0 | F |
| 0 | 0 | 1 | T |
| 0 | 1 | 0 | F |
| 0 | 1 | 1 | F |
| 1 | 0 | 0 | F |
| 1 | 0 | 1 | T |
| 1 | 1 | 0 | F |
| 1 | 1 | 1 | T |

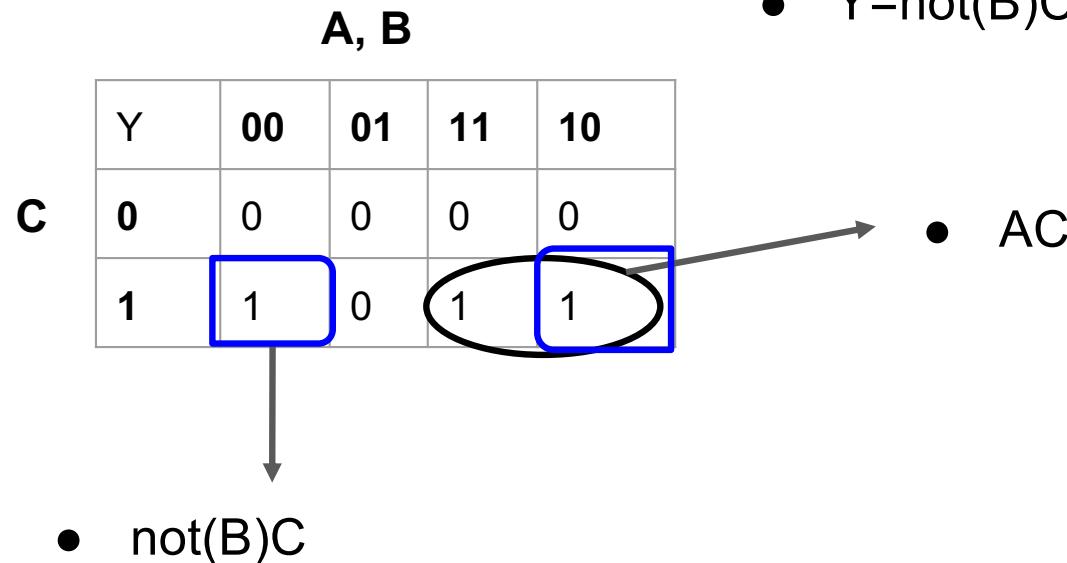
- $Y = AC + \text{not}(A) \text{ not}(B) C$ semplificata in
- $Y = \text{not}(B)C + AC$



K-maps: altro esempio

| A | B | C | Output |
|---|---|---|--------|
| 0 | 0 | 0 | F |
| 0 | 0 | 1 | T |
| 0 | 1 | 0 | F |
| 0 | 1 | 1 | F |
| 1 | 0 | 0 | F |
| 1 | 0 | 1 | T |
| 1 | 1 | 0 | F |
| 1 | 1 | 1 | T |

- $Y = AC + \text{not}(A) \text{ not}(B) C$
semplificata in
- $Y = \text{not}(B)C + AC$

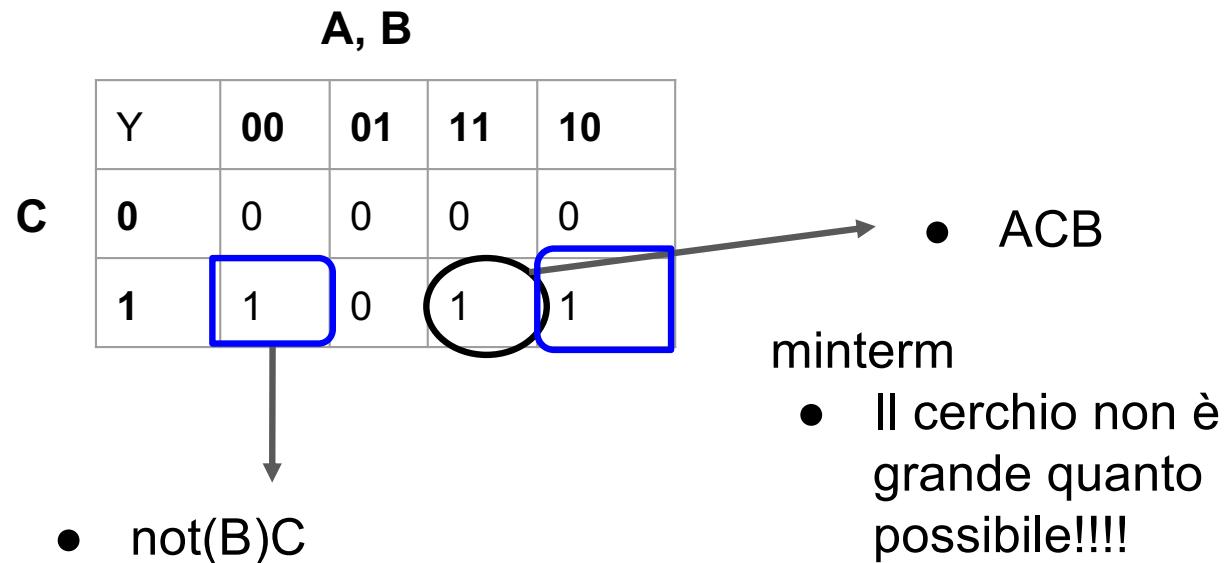


- Creo formula minimale
- addiziono gli implicanti primi
 - $Y = \text{not}(B)C + AC$

K-maps: altro esempio...errori possibili

| A | B | C | Output |
|---|---|---|--------|
| 0 | 0 | 0 | F |
| 0 | 0 | 1 | T |
| 0 | 1 | 0 | F |
| 0 | 1 | 1 | F |
| 1 | 0 | 0 | F |
| 1 | 0 | 1 | T |
| 1 | 1 | 0 | F |
| 1 | 1 | 1 | T |

- $Y = AC + \text{not}(A) \text{ not}(B) C$
semplificata in
- $Y = \text{not}(B)C + AC$



K-maps: altro esempio...errori possibili

| A | B | C | Output |
|---|---|---|--------|
| 0 | 0 | 0 | F |
| 0 | 0 | 1 | T |
| 0 | 1 | 0 | F |
| 0 | 1 | 1 | F |
| 1 | 0 | 0 | F |
| 1 | 0 | 1 | T |
| 1 | 1 | 0 | F |
| 1 | 1 | 1 | T |

- $Y = AC + \text{not}(A) \text{ not}(B) C$
semplificata in
- $Y = \text{not}(B)C + AC$

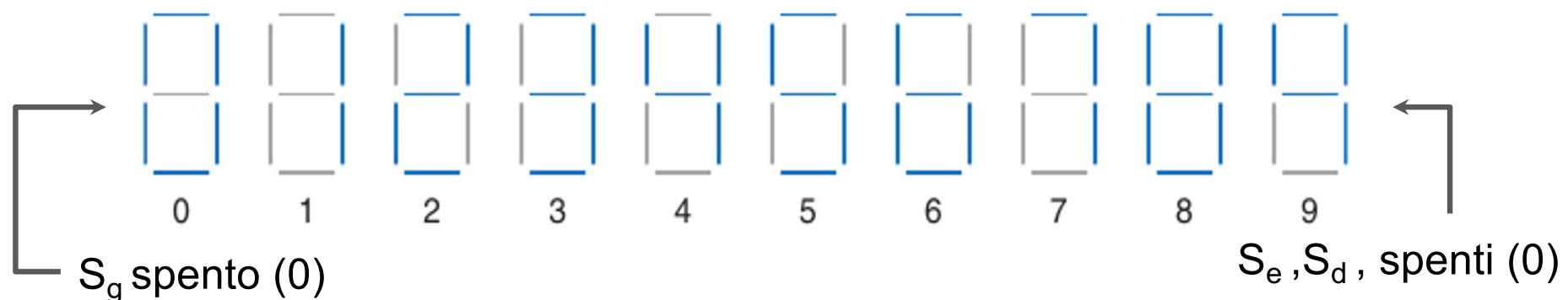
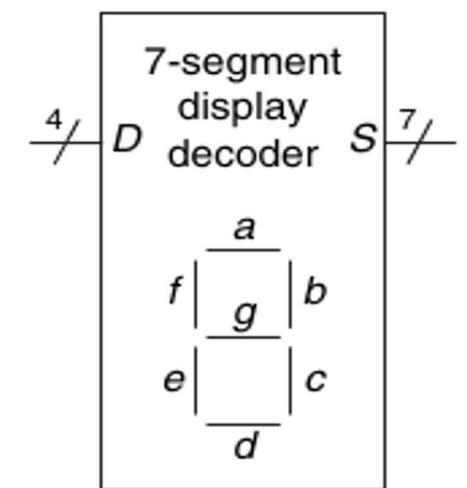
A, B

| C | Y | 00 | 01 | 11 | 10 |
|------|---|----|----|----|----|
| | | 0 | 0 | 0 | 0 |
| A, B | 1 | 1 | 0 | 1 | 1 |
| | 1 | 1 | 0 | 1 | 1 |

- Il cerchio coinvolge un numero di celle che non è una potenza di 2

Mappe di Karnaugh: Seven-Segment display decoder

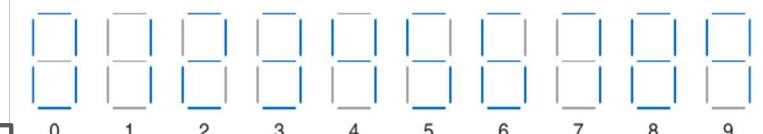
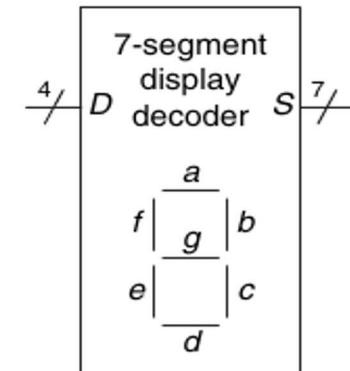
Un decodificatore per un display a sette segmenti (seven-segment display decoder) accetta un input di dati a 4 bit $D_{3:0}$ e produce sette uscite per controllare i diodi emettitori di luce per visualizzare una cifra da 0 a 9. Le sette uscite sono spesso chiamate segmenti da a a g, o $S_a - S_g$.



Mappe di Karnaugh: Seven-Segment display decoder

Table 2.6 Seven-segment display decoder truth table

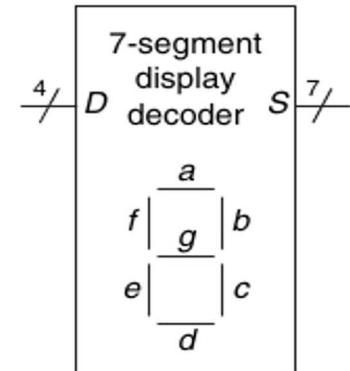
| $D_{3:0}$ | S_a | S_b | S_c | S_d | S_e | S_f | S_g |
|-----------|-------|-------|-------|-------|-------|-------|-------|
| 0000 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0001 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0010 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 0011 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 0100 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0101 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0110 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 0111 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1000 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1001 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| others | 0 | 0 | 0 | 0 | 0 | 0 | 0 |



Input Indicati come
 D_3, D_2, D_1, D_0 od anche $D_{3:0}$,

Esempio: $D_3=0, D_2=1, D_1=0, D_0=1$, cioè 0101 (5), corrisponde $S_b=0$ ed $S_e=0$

Mappe di Karnaugh: Seven-Segment display decoder



- Abbiamo 7 uscite
- Per ciascuna dobbiamo trovare una formula Booleana minimale
- Dobbiamo realizzare 7 mappe di Karnaugh

Mappe di Karnaugh: Seven-Segment display decoder

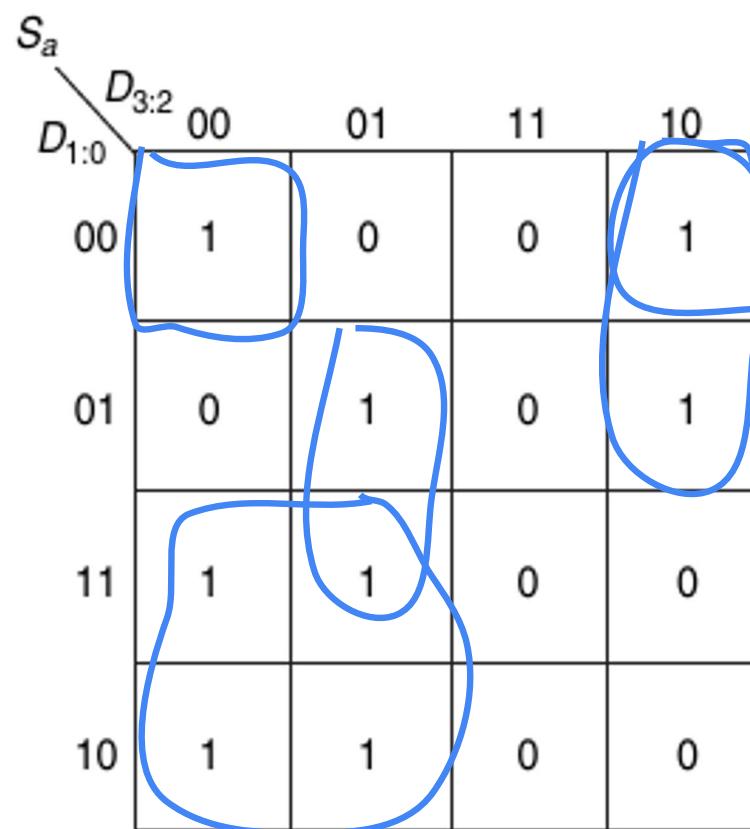
| $D_{3:0}$ | S_a |
|-----------|-------|
| 0000 | 1 |
| 0001 | 0 |
| 0010 | 1 |
| 0011 | 1 |
| 0100 | 0 |
| 0101 | 1 |
| 0110 | 1 |
| 0111 | 1 |
| 1000 | 1 |
| 1001 | 1 |
| others | 0 |

| S_a | $D_{3:2}$ | 00 | 01 | 11 | 10 |
|-------|-----------|----|----|----|----|
| | $D_{1:0}$ | 00 | 01 | 11 | 10 |
| 00 | 00 | 1 | 0 | 0 | 1 |
| 00 | 01 | 0 | 1 | 0 | 1 |
| 01 | 11 | 1 | 1 | 0 | 0 |
| 01 | 10 | 1 | 1 | 0 | 0 |

Ricordiamo che
dobbiamo
rispettare il **gray
code**

Mappe di Karnaugh: Seven-Segment display decoder

| $D_{3:0}$ | S_a |
|-----------|-------|
| 0000 | 1 |
| 0001 | 0 |
| 0010 | 1 |
| 0011 | 1 |
| 0100 | 0 |
| 0101 | 1 |
| 0110 | 1 |
| 0111 | 1 |
| 1000 | 1 |
| 1001 | 1 |
| others | 0 |

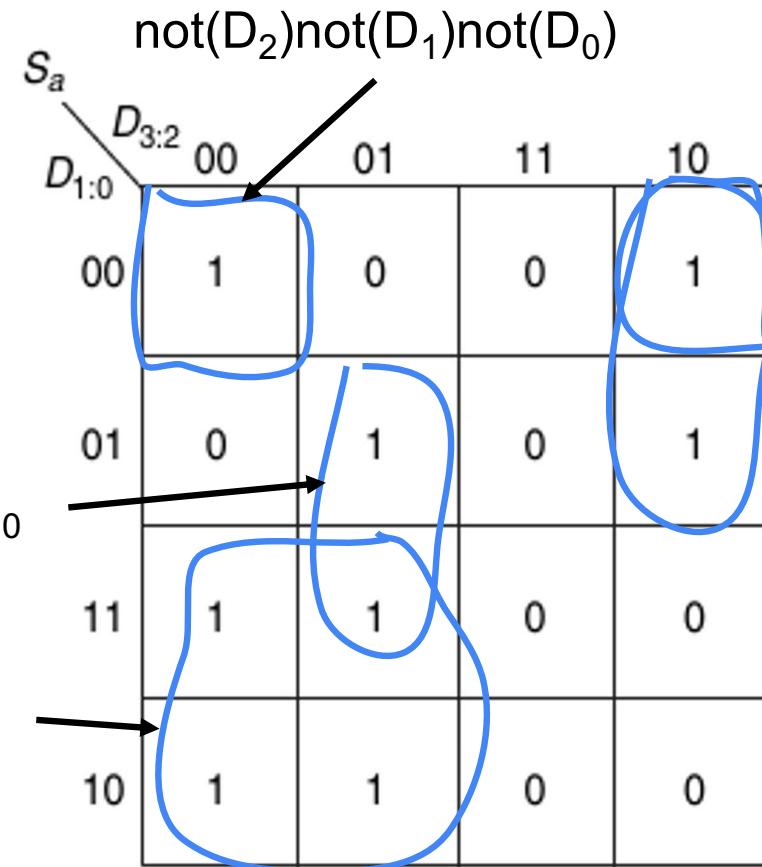


Costruiamo i cerchi

- Cerchi quanto più “ampi” possibili
- Anche “circolari”
- Comprendono solo 1
- Comprendono solo potenze di 2
- Numero minimo di cerchi

Mappe di Karnaugh: Seven-Segment display decoder

| $D_{3:0}$ | S_a |
|-----------|-------|
| 0000 | 1 |
| 0001 | 0 |
| 0010 | 1 |
| 0011 | 1 |
| 0100 | 0 |
| 0101 | 1 |
| 0110 | 1 |
| 0111 | 1 |
| 1000 | 1 |
| 1001 | 1 |
| others | 0 |

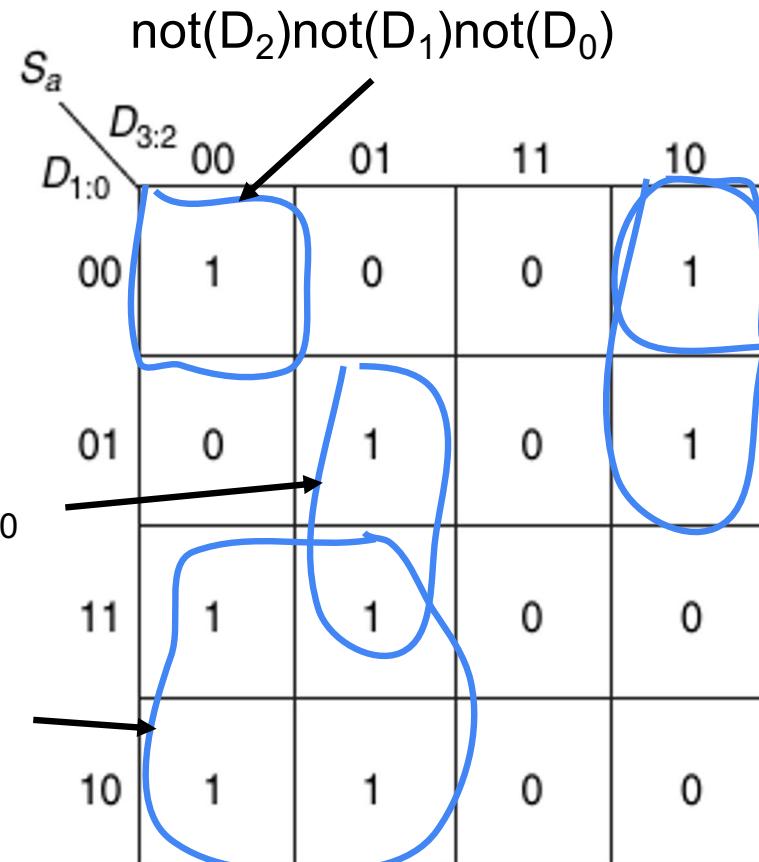


Costruiamo gli implicanti primi

D_3 not(D_2)not(D_1)

Mappe di Karnaugh: Seven-Segment display decoder

| $D_{3:0}$ | S_a |
|-----------|-------|
| 0000 | 1 |
| 0001 | 0 |
| 0010 | 1 |
| 0011 | 1 |
| 0100 | 0 |
| 0101 | 1 |
| 0110 | 1 |
| 0111 | 1 |
| 1000 | 1 |
| 1001 | 1 |
| others | 0 |



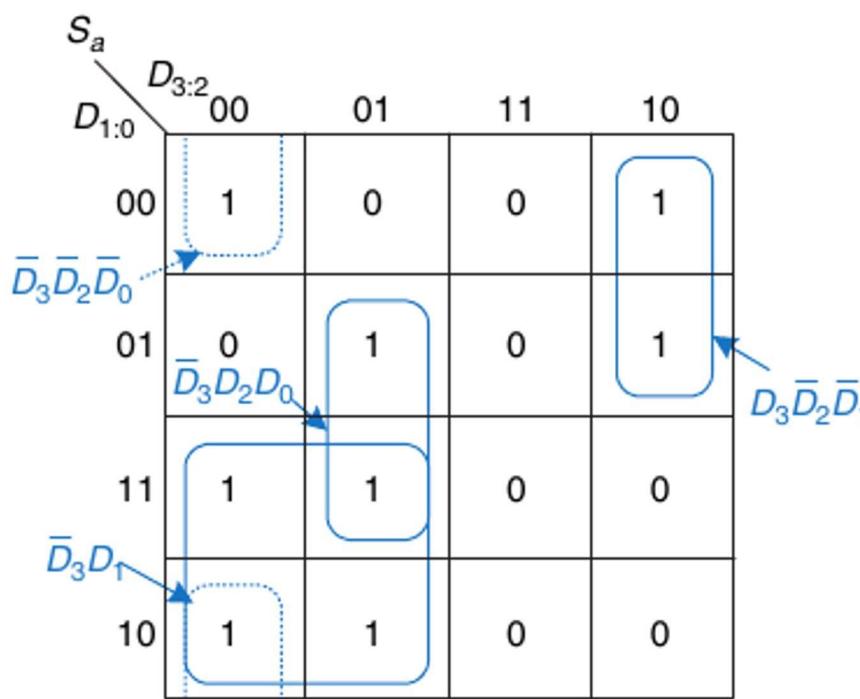
Costruiamo
formula minimale

$D_3\overline{D}_2\overline{D}_1$

$$S_a = \overline{D}_3D_1 + \overline{D}_3D_2D_0 + D_3\overline{D}_2\overline{D}_1 + \overline{D}_2\overline{D}_1\overline{D}_0$$

Mappe di Karnaugh: Seven-Segment display decoder

| $D_{3:0}$ | S_a |
|-----------|-------|
| 0000 | 1 |
| 0001 | 0 |
| 0010 | 1 |
| 0011 | 1 |
| 0100 | 0 |
| 0101 | 1 |
| 0110 | 1 |
| 0111 | 1 |
| 1000 | 1 |
| 1001 | 1 |
| others | 0 |



Osservazioni:

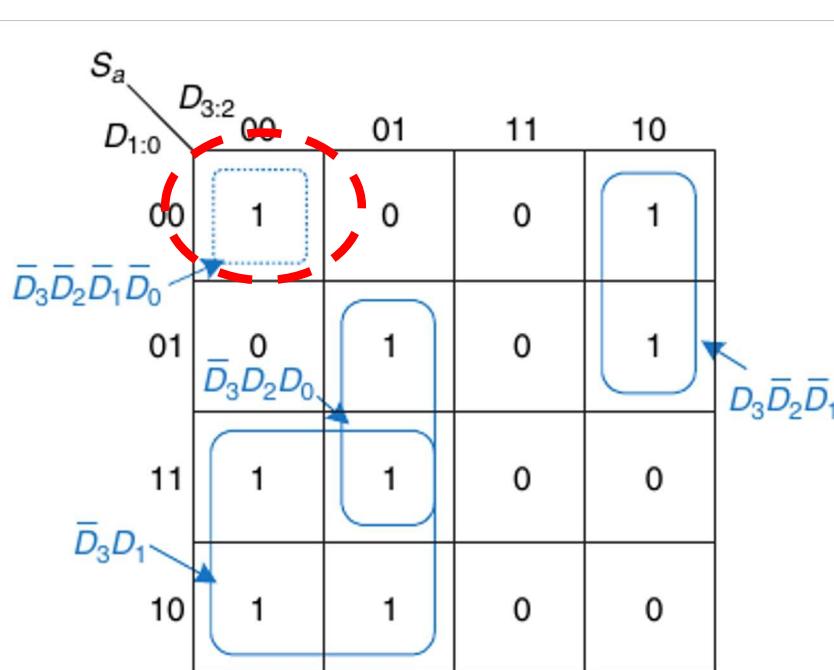
- Possiamo trovare un set di implicanti primi diverso

$$S_a = \bar{D}_3 D_1 + \bar{D}_3 D_2 D_0 + D_3 \bar{D}_2 \bar{D}_1 + \boxed{\bar{D}_3 \bar{D}_2 \bar{D}_0}$$

$$S_a = \text{not}(D_3)D_1 + \text{not}(D_3)D_2D_0 + D_3\text{not}(D_2)\text{not}(D_1) + \boxed{\text{not}(D_2)\text{not}(D_1)\text{not}(D_0)}$$

Mappe di Karnaugh: Seven-Segment display decoder

| $D_{3:0}$ | S_a |
|-----------|-------|
| 0000 | 1 |
| 0001 | 0 |
| 0010 | 1 |
| 0011 | 1 |
| 0100 | 0 |
| 0101 | 1 |
| 0110 | 1 |
| 0111 | 1 |
| 1000 | 1 |
| 1001 | 1 |
| others | 0 |



Osservazioni:

- **Attenzione nel costruire i cerchi giusti**
- **qui un esempio di costruzione errata, non è rispettata regola k4: Ciascun cerchio deve essere quanto più grande possibile**

Mappe di Karnaugh: Seven-Segment display decoder

| $D_{3:0}$ | S_a |
|-----------|-------|
| 0000 | 1 |
| 0001 | 0 |
| 0010 | 1 |
| 0011 | 1 |
| 0100 | 0 |
| 0101 | 1 |
| 0110 | 1 |
| 0111 | 1 |
| 1000 | 1 |
| 1001 | 1 |
| others | 0 |

NOTA: Don't care (non mi importa)

- Possiamo introdurre il concetto "don't care", "non mi interessa" per alcune variabili di input della tabella della verità, allo scopo di ridurre il numero di righe nella tabella quando queste non influiscono sull'output.
- Sono indicati dal simbolo X, che significa che la voce può essere 0 o 1.
- **Possono comparire anche negli output della tabella della verità** in cui il valore di output non è importante o la combinazione di input corrispondente non può mai verificarsi. Tali output possono essere trattati come 0 o 1 a discrezione del progettista

Mappe di Karnaugh: Seven-Segment display decoder

| $D_{3:0}$ | S_a |
|-----------|-------|
| 0000 | 1 |
| 0001 | 0 |
| 0010 | 1 |
| 0011 | 1 |
| 0100 | 0 |
| 0101 | 1 |
| 0110 | 1 |
| 0111 | 1 |
| 1000 | 1 |
| 1001 | 1 |
| others | 0 |

Nuova tabella con “Don’t care” output

| | $D_{3:2}$ | | | | |
|-----------|-----------|----|----|----|----|
| $D_{1:0}$ | S_a | 00 | 01 | 11 | 10 |
| 00 | 00 | 1 | 0 | X | 1 |
| 01 | 01 | 0 | 1 | X | 1 |
| 11 | 11 | 1 | 1 | X | X |
| 10 | 10 | 1 | 1 | X | X |

Anziché considerare a 0, “non mi importa” del loro valore

Mappe di Karnaugh: Seven-Segment display decoder

| $D_{3:0}$ | S_a |
|-----------|-------|
| 0000 | 1 |
| 0001 | 0 |
| 0010 | 1 |
| 0011 | 1 |
| 0100 | 0 |
| 0101 | 1 |
| 0110 | 1 |
| 0111 | 1 |
| 1000 | 1 |
| 1001 | 1 |
| others | 0 |

Nuova tabella con “Don’t care” output

$D_{3:2}$

| S_a | 00 | 01 | 11 | 10 |
|-------|----|----|----|----|
| 00 | 1 | 0 | X | 1 |
| 01 | 0 | 1 | X | 1 |
| 11 | 1 | 1 | X | X |
| 10 | 1 | 1 | X | X |

Perché è importante?

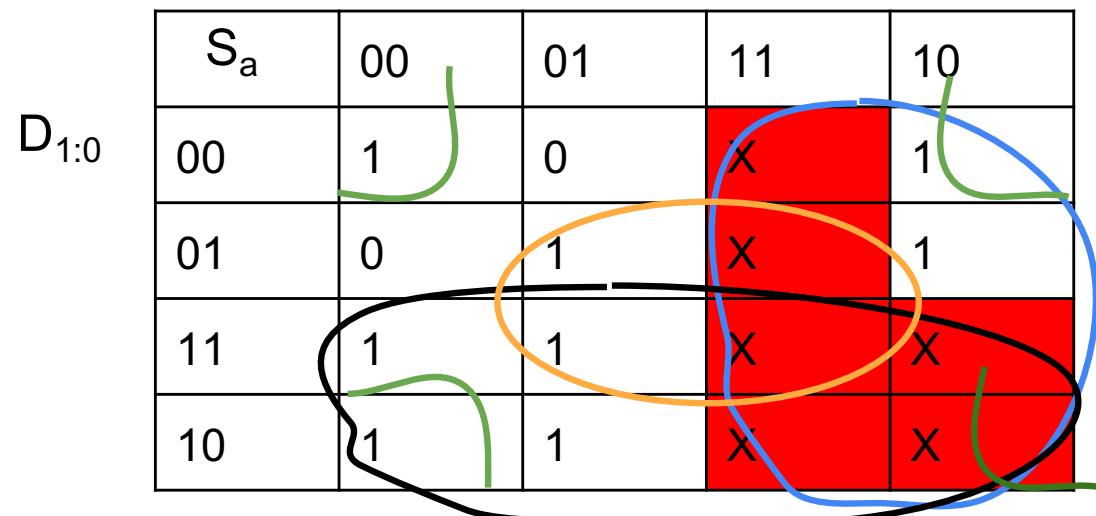
- Possiamo considerare le X con il valore che vogliamo se questo permette di fare cerchi più ampi e/o meno cerchi

Mappe di Karnaugh: Seven-Segment display decoder

| $D_{3:0}$ | S_a |
|-----------|-------|
| 0000 | 1 |
| 0001 | 0 |
| 0010 | 1 |
| 0011 | 1 |
| 0100 | 0 |
| 0101 | 1 |
| 0110 | 1 |
| 0111 | 1 |
| 1000 | 1 |
| 1001 | 1 |
| others | 0 |

Nuova tabella con “Don’t care” output

$D_{3:2}$



Abbiamo sempre 4 cerchi ma molto più ampi!!!

Mappe di Karnaugh: Seven-Segment display decoder

| $D_{3:0}$ | S_a |
|-----------|-------|
| 0000 | 1 |
| 0001 | 0 |
| 0010 | 1 |
| 0011 | 1 |
| 0100 | 0 |
| 0101 | 1 |
| 0110 | 1 |
| 0111 | 1 |
| 1000 | 1 |
| 1001 | 1 |
| others | 0 |

Nuova tabella con “Don’t care” output

$D_{3:2}$

| S_a | 00 | 01 | 11 | 10 |
|-------|----|----|----|----|
| 00 | 1 | 0 | X | 1 |
| 01 | 0 | 1 | X | 1 |
| 11 | 1 | 1 | X | X |
| 10 | 1 | 1 | X | X |

- $\text{not}(D_2) \text{ not}(D_0)$
- $D_2 D_0$
- D_3
- D_1

$$S_a = D_3 + D_2 D_0 + \text{not}(D_2) \text{ not}(D_0) + D_1$$

Mappe di Karnaugh: Seven-Segment display decoder

Con variabile di output “don’t care”:

$$S_a = D_3 + D_2 D_0 + \text{not}(D_2) \text{ not}(D_0) + D_1$$

Senza variabile di output “don’t care”:

$$S_a = \text{not}(D_3)D_1 + \text{not}(D_3)D_2D_0 + D_3\text{not}(D_2)\text{not}(D_1) + \text{not}(D_2)\text{not}(D_1)\text{not}(D_0)$$

Per le prime 10 configurazioni di input a cui sono interessato le due espressioni sono equivalenti da un punto di vista funzionale !!!! Questo significa che nel primo caso (quella in alto) le successive (dalla 11-sima in poi) configurazioni non sono ammesse!!!

Mappe di Karnaugh: Seven-Segment display decoder

Ritornando al nostro problema.....

Tabella di Verità generale

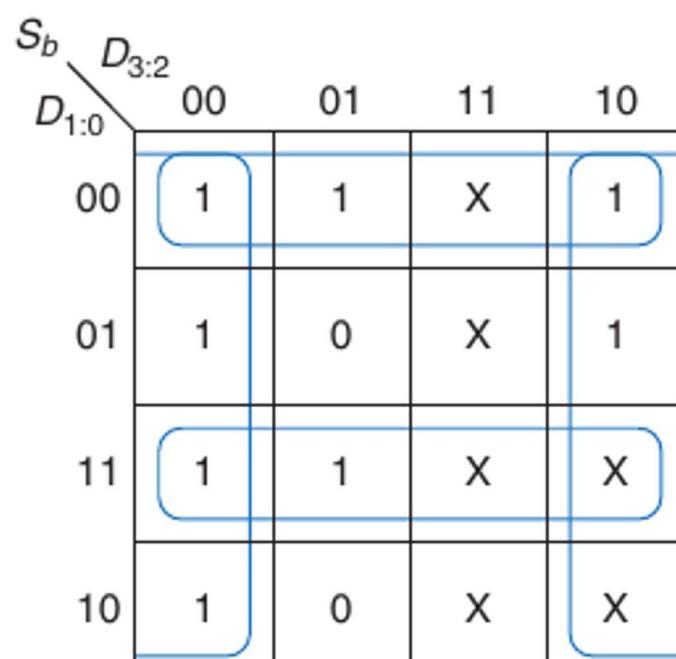
Dobbiamo ripetere lo stesso ragionamento per tutte le altre uscite!!!

| $D_{3:0}$ | S_a | S_b | S_c | S_d | S_e | S_f | S_g |
|-----------|-------|-------|-------|-------|-------|-------|-------|
| 0000 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0001 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0010 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 0011 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 0100 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0101 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0110 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 0111 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1000 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1001 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| others | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Mappe di Karnaugh: Seven-Segment display decoder

| $D_{3:0}$ | S_b |
|-----------|-------|
| 0000 | 1 |
| 0001 | 1 |
| 0010 | 1 |
| 0011 | 1 |
| 0100 | 1 |
| 0101 | 0 |
| 0110 | 0 |
| 0111 | 1 |
| 1000 | 1 |
| 1001 | 1 |
| others | 0 |

Per S_b



$$S_b = \bar{D}_2 + D_1 D_0 + \bar{D}_1 \bar{D}_0$$

Anche qui l'uso del “Don't care” permette una soluzione più sintetica.

- Abbiamo solo tre implicanti primi
- Ciascun implicante ha meno letterali

Mappe di Karnaugh: altro esempio “don’t care”

| A | B | C | D | Y |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | X |
| 0 | 0 | 1 | 1 | X |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | X |
| 0 | 1 | 1 | 0 | X |
| 0 | 1 | 1 | 1 | X |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | X |
| 1 | 1 | 1 | 1 | 1 |

AB

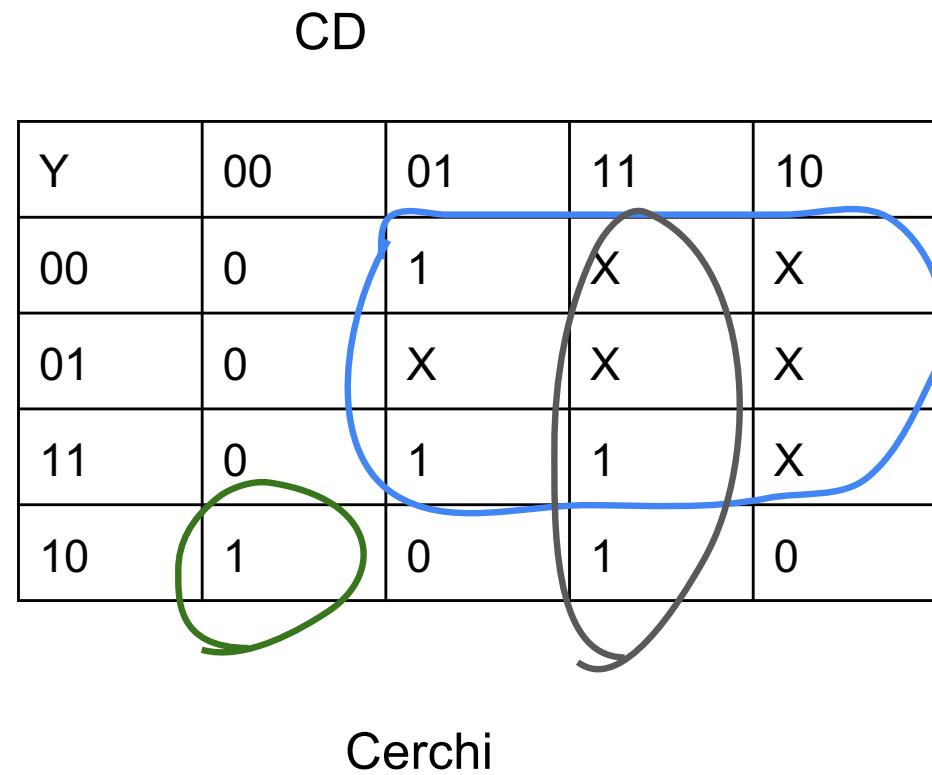
| | | CD | | | |
|----|----|----|----|----|--|
| Y | 00 | 01 | 11 | 10 | |
| 00 | 0 | 1 | X | X | |
| 01 | 0 | X | X | X | |
| 11 | 0 | 1 | 1 | X | |
| 10 | 1 | 0 | 1 | 0 | |

TAbella

Mappe di Karnaugh: altro esempio “don’t care”

| A | B | C | D | Y |
|-----|-----|-----|-----|-----|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | X |
| 0 | 0 | 1 | 1 | X |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | X |
| 0 | 1 | 1 | 0 | X |
| 0 | 1 | 1 | 1 | X |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | X |
| 1 | 1 | 1 | 1 | 1 |

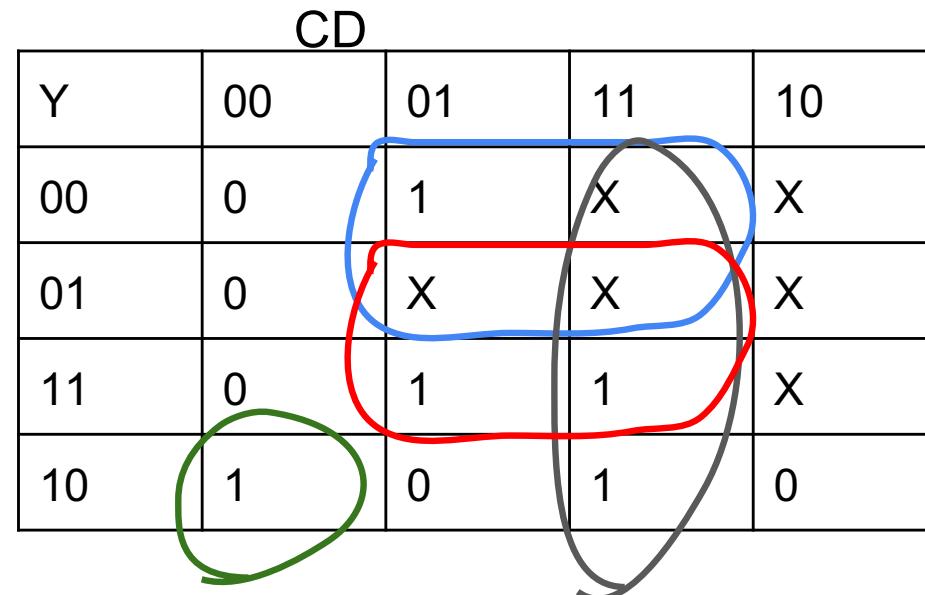
AB



Mappe di Karnaugh: altro esempio “don’t care”

| A | B | C | D | Y |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | X |
| 0 | 0 | 1 | 1 | X |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | X |
| 0 | 1 | 1 | 0 | X |
| 0 | 1 | 1 | 1 | X |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | X |
| 1 | 1 | 1 | 1 | 1 |

AB



Implicanti primi:

- **not(A)D** **B varia, C varia**
- **BD** **A varia, C Varia**
- **CD** **A, B variano**
- **Anot(B)not(C)not(D)** **non varia nessuno**

Mappe di Karnaugh: altro esempio “don’t care”

| A | B | C | D | Y |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | X |
| 0 | 0 | 1 | 1 | X |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | X |
| 0 | 1 | 1 | 0 | X |
| 0 | 1 | 1 | 1 | X |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | X |
| 1 | 1 | 1 | 1 | 1 |

Implicanti primi:

- $\text{not}(A)D$ B varia, C varia
- BD A varia, C Varia
- CD A, B variano
- $\text{Anot}(B)\text{not}(C)\text{not}(D)$ non varia nessuno

Formula minimale

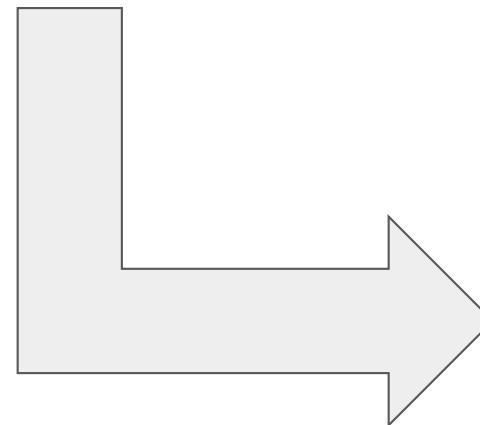
- $Y = \text{not}(A)D + BD + CD + \text{Anot}(B)\text{not}(C)\text{not}(D)$

Mappe di Karnaugh: altro esempio “don’t care”

| A | B | C | D | Y |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | X |
| 0 | 0 | 1 | 1 | X |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | X |
| 0 | 1 | 1 | 0 | X |
| 0 | 1 | 1 | 1 | X |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | X |
| 1 | 1 | 1 | 1 | 1 |

Formula minimale

- $Y = \text{not}(A)D + BD + CD + \text{Anot}(B)\text{not}(C)\text{not}(D)$



| A | B | C | D | Output |
|---|---|---|---|--------|
| 0 | 0 | 0 | 0 | F |
| 0 | 0 | 0 | 1 | T |
| 0 | 0 | 1 | 0 | F |
| 0 | 0 | 1 | 1 | T |
| 0 | 1 | 0 | 0 | F |
| 0 | 1 | 0 | 1 | T |
| 0 | 1 | 1 | 0 | F |
| 0 | 1 | 1 | 1 | T |
| 1 | 0 | 0 | 0 | T |
| 1 | 0 | 0 | 1 | F |
| 1 | 0 | 1 | 0 | F |
| 1 | 0 | 1 | 1 | T |
| 1 | 1 | 0 | 0 | F |
| 1 | 1 | 0 | 1 | T |
| 1 | 1 | 1 | 0 | F |
| 1 | 1 | 1 | 1 | T |

Architettura degli Elaboratori

Lezione 16

Docente: R.Prevete
a.a. 2022/2023
17 aprile 2023

Mappe di Karnaugh: Priority circuit

- Supponiamo che $n+1$ persone, A_0, A_1, \dots, A_n , possano usufruire di un certo servizio (ad esempio l'occupazione di un'aula).
- Se una singola persona vuole utilizzare tale servizio non ci sono problemi.
- Ma se $1 < k \leq n+1$ vogliono utilizzare contemporaneamente tale servizio, dobbiamo fissare una ordine (priorità). Ad esempio:
 - $A_n > A_{n-1} > \dots > A_0$

$$A_i > A_{i-1} \quad i = n, n-1, \dots, 1$$

Priority circuit

- Tabella verità con n=4

| A_3 | A_2 | A_1 | A_0 | Y_3 | Y_2 | Y_1 | Y_0 |
|-------|-------|-------|-------|-------|-------|-------|-------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |

- Ho una funzione con n entrate ed n uscite
- Variabili di ingresso: A_0, A_1, A_2, A_3 ($A_{3:0}$)
- Variabili di output: Y_0, Y_1, Y_2, Y_3 ($Y_{3:0}$)
- $Y_k=1 \Rightarrow$ risorsa assegnata a A_k
- $Y_k=1 \Rightarrow Y_h=0$ con $h <> k$

Priority circuit

- Tabella verità con n=4

| A_3 | A_2 | A_1 | A_0 | Y_3 | Y_2 | Y_1 | Y_0 |
|-------|-------|-------|-------|-------|-------|-------|-------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |

- Don't care (variabili di ingresso)

| A_3 | A_2 | A_1 | A_0 | Y_3 | Y_2 | Y_1 | Y_0 |
|-------|-------|-------|-------|-------|-------|-------|-------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | X | 0 | 0 | 1 | 0 |
| 0 | 1 | X | X | 0 | 1 | 0 | 0 |
| 1 | X | X | X | 1 | 0 | 0 | 0 |

Priority circuit

- Tabella verità con n=4

| A_3 | A_2 | A_1 | A_0 | Y_3 | Y_2 | Y_1 | Y_0 |
|-------|-------|-------|-------|-------|-------|-------|-------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | X | 0 | 0 | 1 | 0 |
| 0 | 1 | X | X | 0 | 1 | 0 | 0 |
| 1 | X | X | X | 1 | 0 | 0 | 0 |

| $A_{3:2}$ | $A_{1:0}$ | Y ₃ | 00 | 01 | 11 | 10 |
|-----------|-----------|----------------|----|----|----|----|
| 00 | 0 | 0 | 0 | 0 | 0 | 0 |
| 01 | 0 | 0 | 0 | 0 | 0 | 0 |
| 11 | 1 | 1 | 1 | 1 | 1 | 1 |
| 10 | 1 | 1 | 1 | 1 | 1 | 1 |

Priority circuit

- Tabella verità con n=4

| A_3 | A_2 | A_1 | A_0 | Y_3 | Y_2 | Y_1 | Y_0 |
|-------|-------|-------|-------|-------|-------|-------|-------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | X | 0 | 0 | 1 | 0 |
| 0 | 1 | X | X | 0 | 1 | 0 | 0 |
| 1 | X | X | X | 1 | 0 | 0 | 0 |

$$Y_3 = A_3$$

| $A_{3:2}$ | $A_{1:0}$ | 00 | 01 | 11 | 10 |
|----------------|-----------|----|----|----|----|
| Y ₃ | | | | | |
| 00 | 0 | 0 | 0 | 0 | 0 |
| 01 | 0 | 0 | 0 | 0 | 0 |
| 11 | 1 | 1 | 1 | 1 | 1 |
| 10 | 1 | 1 | 1 | 1 | 1 |

Come, in questo caso, si evince facilmente dalla tabella di verità

Priority circuit

- Tabella verità con n=4

| A_3 | A_2 | A_1 | A_0 | Y_3 | Y_2 | Y_1 | Y_0 |
|-------|-------|-------|-------|-------|-------|-------|-------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | X | 0 | 0 | 1 | 0 |
| 0 | 1 | X | X | 0 | 1 | 0 | 0 |
| 1 | X | X | X | 1 | 0 | 0 | 0 |

| $A_{3:2}$ | $A_{1:0}$ |
|-----------|-----------|
| Y_2 | 00 |
| 00 | 0 |
| 01 | 1 |
| 11 | 0 |
| 10 | 0 |

Priority circuit

- Tabella verità con n=4

| A_3 | A_2 | A_1 | A_0 | Y_3 | Y_2 | Y_1 | Y_0 |
|-------|-------|-------|-------|-------|-------|-------|-------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | |
| 0 | 0 | 1 | X | 0 | 0 | 1 | 0 |
| 0 | 1 | X | X | 0 | 1 | 0 | 0 |
| 1 | X | X | X | 1 | 0 | 0 | 0 |

$A_{1:0}$

| Y_2 | 00 | 01 | 11 | 10 |
|-------|----|----|----|----|
| 00 | 0 | 0 | 0 | 0 |
| 01 | 1 | 1 | 1 | 1 |
| 11 | 0 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 |

$A_{3:2}$

$$Y_2 = \text{not}(A_3)A_2$$

Come, in questo caso, si evince facilmente dalla tabella di verità

Priority circuit

- Tabella verità con n=4

| A_3 | A_2 | A_1 | A_0 | Y_3 | Y_2 | Y_1 | Y_0 |
|-------|-------|-------|-------|-------|-------|-------|-------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | X | 0 | 0 | 1 | 0 |
| 0 | 1 | X | X | 0 | 1 | 0 | 0 |
| 1 | X | X | X | 1 | 0 | 0 | 0 |

| $A_{3:2}$ | $A_{1:0}$ |
|-----------|-----------|
| Y_1 | 00 |
| 00 | 0 |
| 01 | 01 |
| 11 | 11 |
| 10 | 10 |

Priority circuit

- Tabella verità con n=4

| A_3 | A_2 | A_1 | A_0 | Y_3 | Y_2 | Y_1 | Y_0 |
|-------|-------|-------|-------|-------|-------|-------|-------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | X | 0 | 0 | 1 | 0 |
| 0 | 1 | X | X | 0 | 1 | 0 | 0 |
| 1 | X | X | X | 1 | 0 | 0 | 0 |

$A_{1:0}$

| Y_1 | 00 | 01 | 11 | 10 |
|-------|----|----|----|----|
| 00 | 0 | 0 | 1 | 1 |
| 01 | 0 | 0 | 0 | 0 |
| 11 | 0 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 |

$A_{3:2}$

$$Y_1 = \text{not}(A_3)\text{not}(A_2)A_1$$

Come, in questo caso, si evince facilmente dalla tabella di verità

Priority circuit

- Tabella verità con n=4

| A_3 | A_2 | A_1 | A_0 | Y_3 | Y_2 | Y_1 | Y_0 |
|-------|-------|-------|-------|-------|-------|-------|-------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | X | 0 | 0 | 1 | 0 |
| 0 | 1 | X | X | 0 | 1 | 0 | 0 |
| 1 | X | X | X | 1 | 0 | 0 | 0 |

| $A_{3:2}$ | $A_{1:0}$ |
|-----------|-----------|
| Y_0 | 00 |
| 00 | 0 |
| 01 | 1 |
| 11 | 0 |
| 10 | 0 |

Priority circuit

- Tabella verità con n=4

| A_3 | A_2 | A_1 | A_0 | Y_3 | Y_2 | Y_1 | Y_0 |
|-------|-------|-------|-------|-------|-------|-------|-------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | X | 0 | 0 | 1 | 0 |
| 0 | 1 | X | X | 0 | 1 | 0 | 0 |
| 1 | X | X | X | 1 | 0 | 0 | 0 |

| $A_{1:0}$ | | | | |
|-----------|-------|----|----|----|
| $A_{3:2}$ | Y_0 | 00 | 01 | 11 |
| 00 | 0 | 0 | 1 | 0 |
| 01 | 0 | 0 | 0 | 0 |
| 11 | 0 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 |

$$Y_0 = \text{not}(A_3)\text{not}(A_2)\text{not}(A_1)A_0$$

Come, in questo caso, si evince facilmente dalla tabella di verità

Priority circuit

- Tabella verità con n=4

| A_3 | A_2 | A_1 | A_0 | Y_3 | Y_2 | Y_1 | Y_0 |
|-------|-------|-------|-------|-------|-------|-------|-------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | X | 0 | 0 | 1 | 0 |
| 0 | 1 | X | X | 0 | 1 | 0 | 0 |
| 1 | X | X | X | 1 | 0 | 0 | 0 |

$$Y_3 = A_3$$

$$Y_2 = \text{not}(A_3)A_2$$

$$Y_1 = \text{not}(A_3)\text{not}(A_2)A_1$$

$$Y_0 = \text{not}(A_3)\text{not}(A_2)\text{not}(A_1)A_0$$

Come, in questo caso, si evince facilmente dalla tabella di verità

Priority circuit

- Tabella verità con n=4

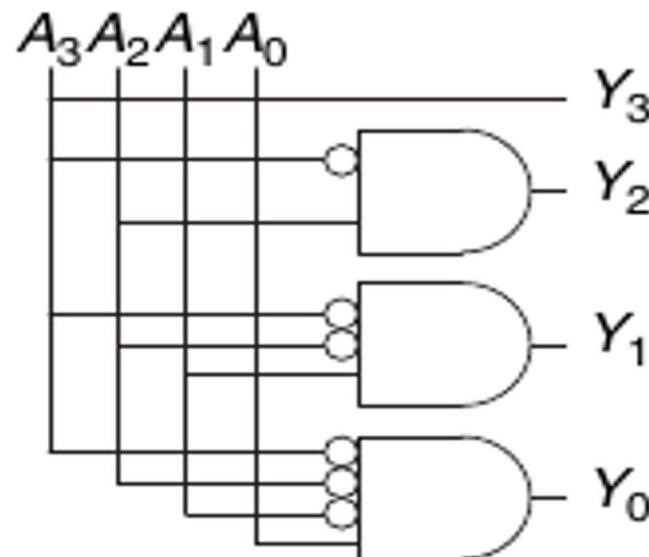
| A_3 | A_2 | A_1 | A_0 | Y_3 | Y_2 | Y_1 | Y_0 |
|-------|-------|-------|-------|-------|-------|-------|-------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | X | 0 | 0 | 1 | 0 |
| 0 | 1 | X | X | 0 | 1 | 0 | 0 |
| 1 | X | X | X | 1 | 0 | 0 | 0 |

$$Y_3 = A_3$$

$$Y_2 = \text{not}(A_3)A_2$$

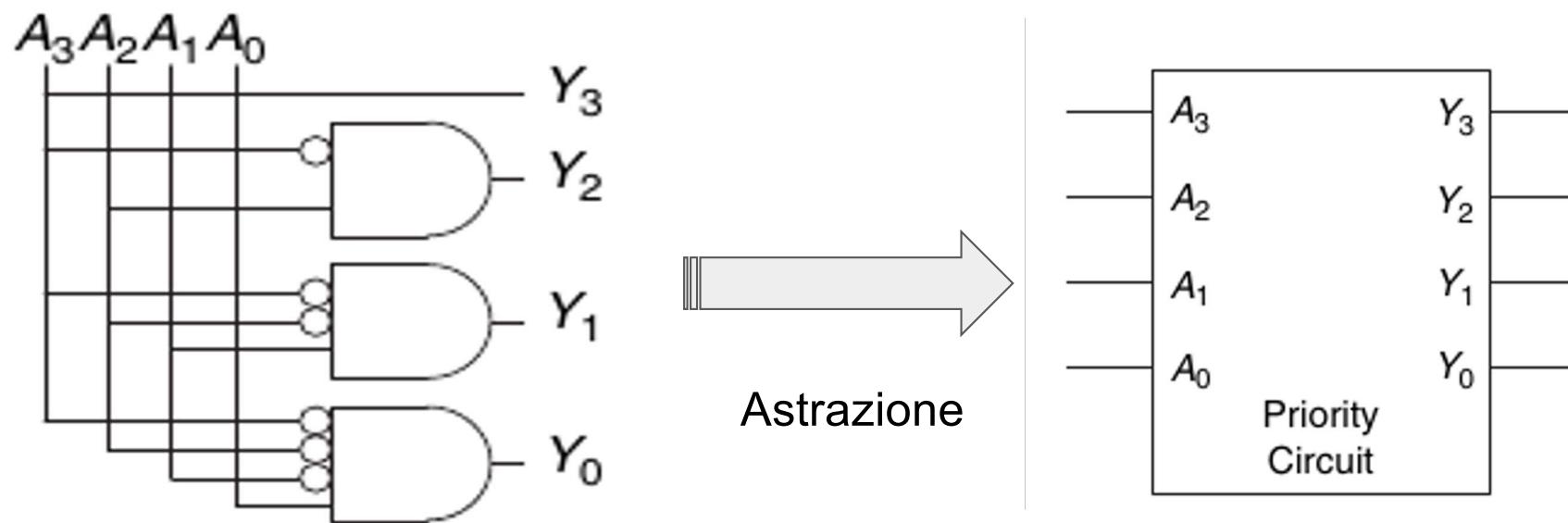
$$Y_1 = \text{not}(A_3)\text{not}(A_2)A_3$$

$$Y_0 = \text{not}(A_3)\text{not}(A_2)\text{not}(A_1)A_0$$

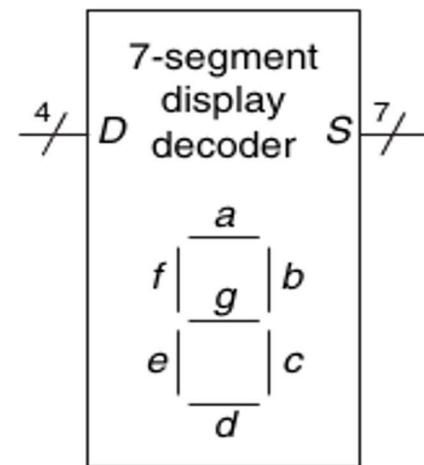
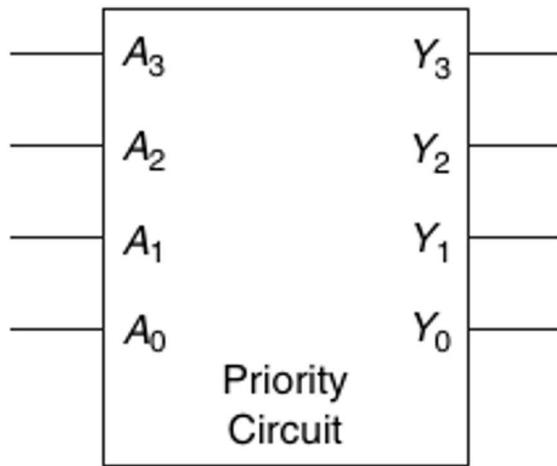


Da un punto di
vista circuitale

Priority circuit



Combinando circuiti (astrarre)



- Da elementi di più basso livello costruiamo circuiti più complessi che poi possiamo utilizzare a loro volta come elementi di base
- In questo modo sfruttiamo il principio di astrazione

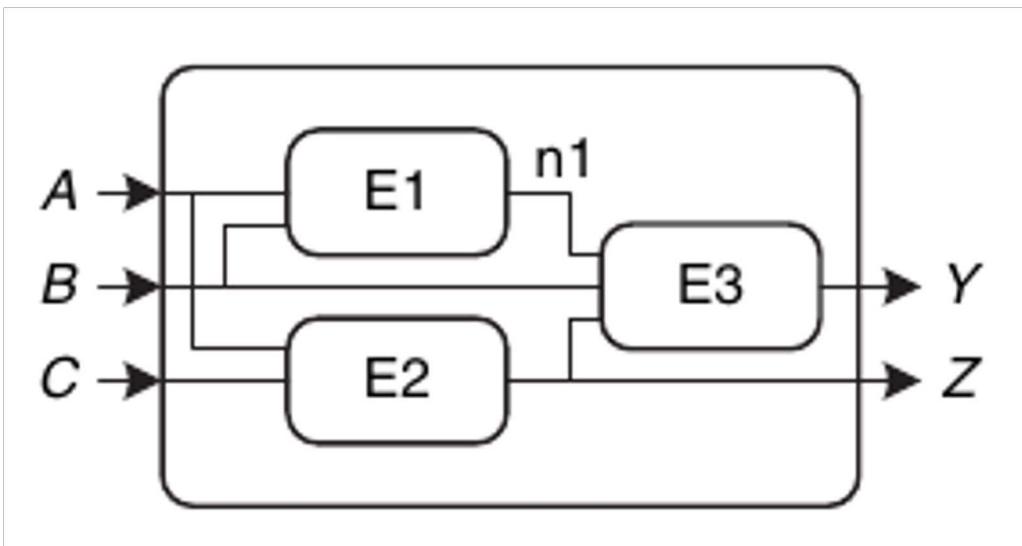
Di cosa parleremo

- **Logica combinatoria: mettere insieme più circuiti**
- **Il tempo**

Combinando circuiti (astrarre)

- Da elementi di più basso livello costruiamo circuiti più complessi che poi possiamo utilizzare a loro volta come elementi di base
- In questo modo sfruttiamo il principio di astrazione

Circuito combinatorio (Combinational circuit)

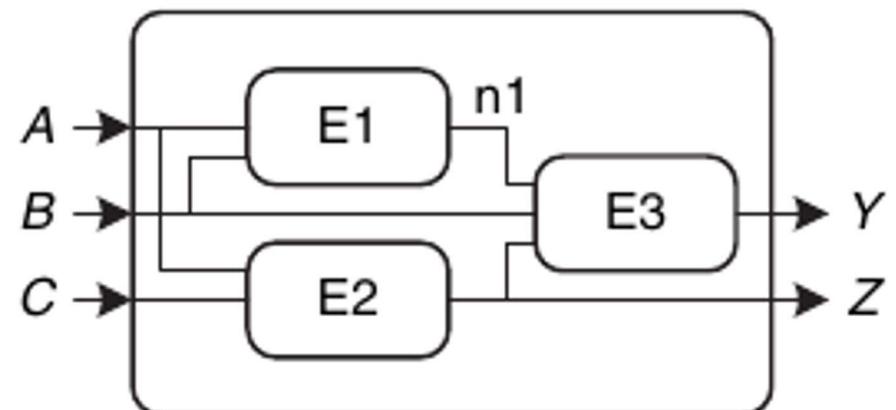


- E1, E2, E3 elementi
- A,B, C, Y, Z, n1 nodi
- A,B,C nodi di input
- Y,Z nodi di output
- n1 nodo interno

- Un elemento è esso stesso un circuito o una porta logica di base con input e output
- Un nodo è una linea su cui viaggia il segnale

Combinando circuiti (regole di base)

- Un circuito è combinatorio (combinational circuit) se consiste nella interconnessione di elementi tale che:
 - Ogni elemento è un circuito combinatorio.
 - Ogni nodo del circuito è o progettato come un input al circuito o si connette esattamente ad 1 terminale di output di un elemento del circuito
 - **Il circuito non contiene cicli**



Architettura degli Elaboratori

Lezione 17

Docente: R.Prevete
a.a. 2022/2023
19 aprile 2023

Circuiti combinatori

Due circuiti combinatori molto comuni sono:

- multiplexer (sceglie, seleziona quale input va in uscita)
- decoder (decodifica un valore codificato)

Multiplexer

I **multiplexer** sono tra i circuiti combinatori più comunemente usati.

- Scelgono il valore di uscita tra diversi possibili ingressi in base al valore di un segnale specifico.

Un multiplexer è talvolta chiamato ***mux***

Multiplexer

| S | D ₁ | D ₀ | Y |
|---|----------------|----------------|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

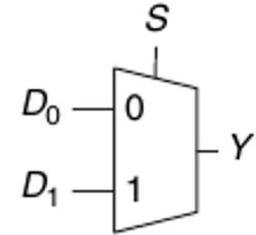
| S | | D _{1:0} | | K-map | |
|---|--|------------------|----|-------|----|
| Y | | 00 | 01 | 11 | 10 |
| 0 | | 0 | 1 | 1 | 0 |
| 1 | | 0 | 0 | 1 | 1 |

- Un segnale di controllo: S
- N variabili di ingresso: N=2
- Una variabile di uscita: Y
- S sceglie il valore di quale variabile di ingresso è trasmesso in output

Multiplexer

Esempio 2:1 multiplexer

- multiplexer con due ingressi D_0 e D_1 , un ingresso di selezione S e un'uscita Y .
- Il multiplexer sceglie tra i due input di dati in base alla valore di S :
 - se $S = 0$, $Y = D_0$ e se $S = 1$, $Y = D_1$.
- S è anche chiamato **segnale di controllo**
- **2:1 multiplexer**, significa 2 variabili di ingresso, 1 variabile di uscita



| S | D_1 | D_0 | Y |
|-----|-------|-------|-----|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

Multiplexer

| S | D ₁ | D ₀ | Y |
|---|----------------|----------------|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

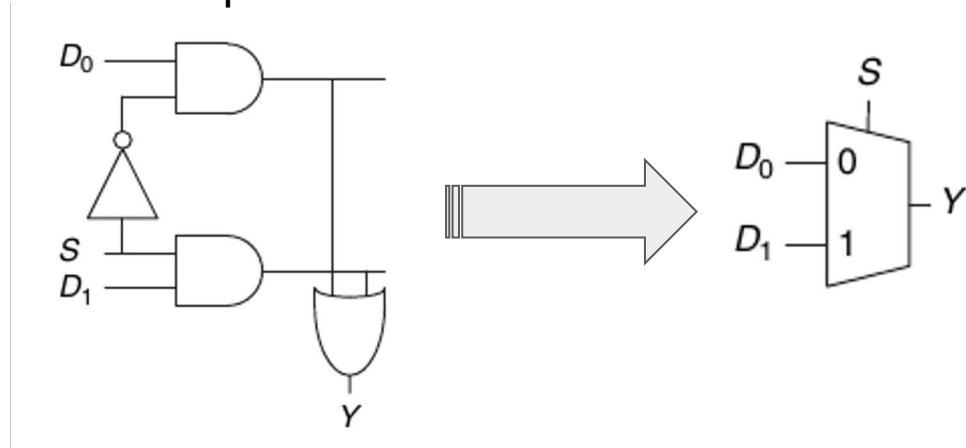
| S | | D _{1:0} | | K-map | | |
|------------------|---|------------------|----|-------|----|----|
| | | Y | 00 | 01 | 11 | 10 |
| D _{1:0} | 0 | 0 | 1 | 1 | 0 | |
| | 1 | 0 | 0 | 1 | 1 | |

$$Y = \text{not}(S)D_0 + SD_1$$

Multiplexer

Multiplexer con “**logica a due livelli**” perché consiste di letterali collegati a un livello di porte AND , a loro volta collegate a un livello di porte OR.

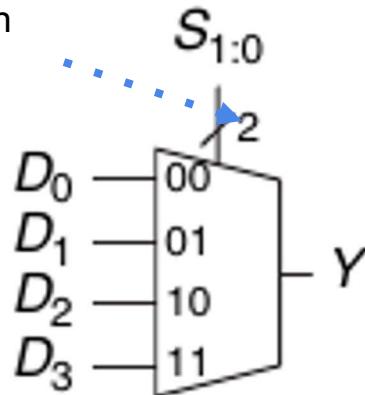
| S | D ₁ | D ₀ | Y |
|---|----------------|----------------|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |



$$Y = \text{not}(S)D_0 + SD_1$$

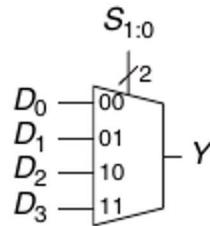
Multiplexer: 4:1 multiplexer

Notazione con
slash

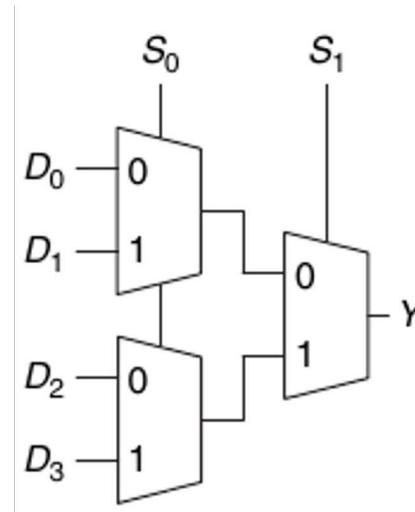
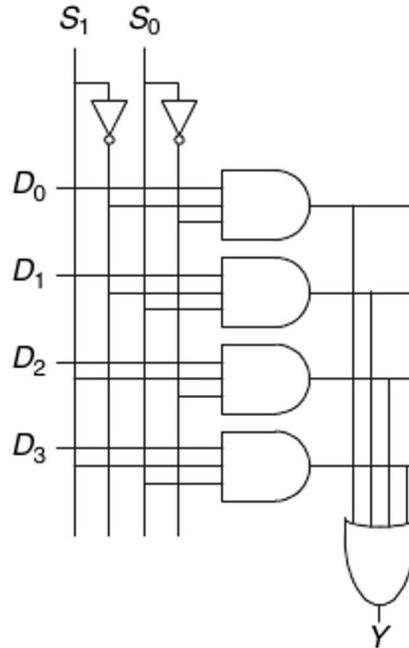


- Un multiplexer 4:1 ha quattro ingressi di dati $D_{3:0}$ e un'uscita, Y
- Sono necessari due segnali di controllo $S_{1:0}$ per scegliere tra i quattro dati ingressi.

Multiplexer: 4:1 multiplexer



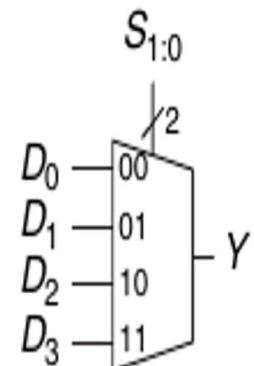
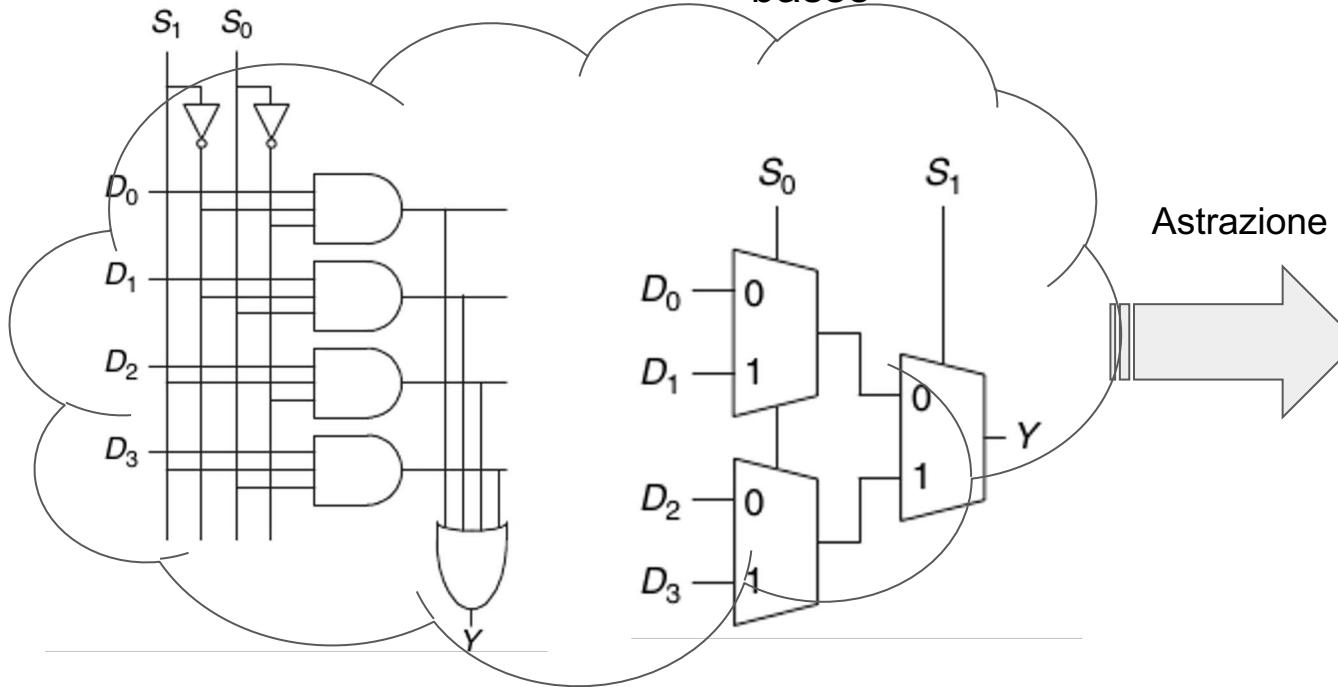
- Il multiplexer 4:1 può essere costruito utilizzando la logica della somma dei prodotti (ad esempio K-maps) o come multiplexer multipli 2:1



Multiplexer: 4:1 multiplexer

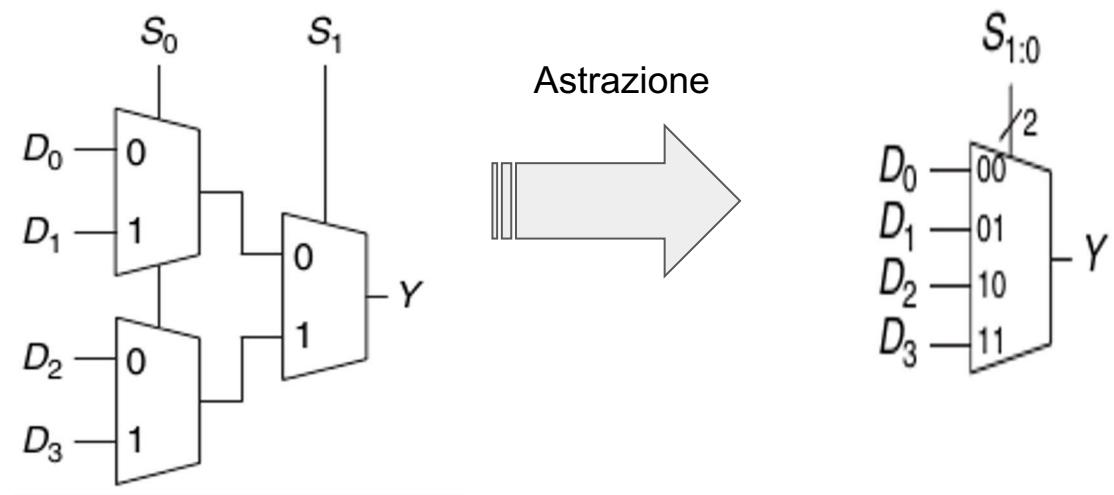
- Osservazione 1

Nel processo di astrazione non ci interessa come le cose sono state implementate nel livello più basso



Multiplexer: 4:1 multiplexer

- Osservazione 2
 - Uso del multiplexer 2:1 per costruire 4:1
 - costruire “mattoncini” via via più complessi



Multiplexer: 4:1 multiplexer

- Importante:
 - In generale, un multiplexer con ingresso 2^N può essere programmato per formare **qualsiasi funzione Booleana** di N-input applicando gli appropriati valori alle variabili di controllo e di ingresso
 - Le varibili di controllo rappresenteranno gli input della funzione Booleana, I valori di ingresso saranno invece utilizzati per assegnare il valore di output della funzione Booleana desiderata.
 - In altre parole, il multiplexer può essere riprogrammato per svolgere una funzione diversa.

Multiplexer: 4:1 multiplexer

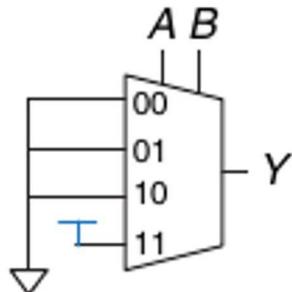
- Esempio:

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

$Y = AB$

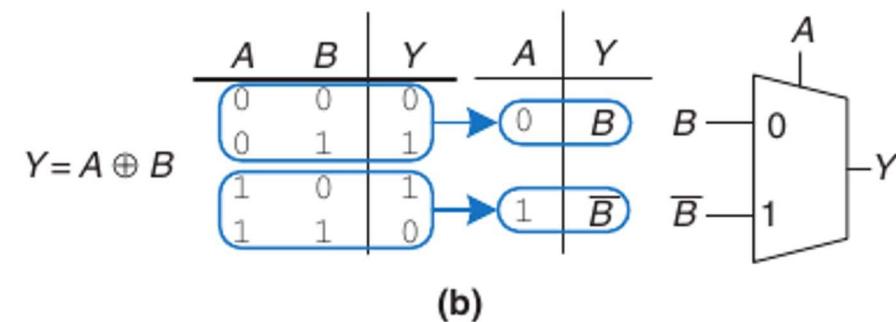
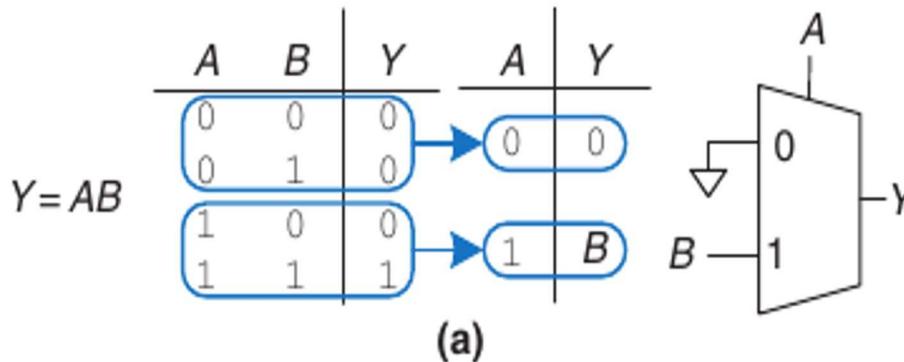
- $A=S_1, B=S_0$
- $D_0=0, D_1=0, D_2=0, D_3=1$

In questo modo il multiplexer 4:1 si comporta come un AND rispetto alle variabili A e B



Multiplexer come porte logiche universali

- Nota che possiamo usare un multiplexer a 2^{N-1} ingressi per eseguire qualsiasi funzione logica a N ingressi.
- La strategia consiste nel fornire uno dei letterali agli ingressi dati del multiplexer.



Decoder (decodificatore)

- Un decoder riceve N input ed ha 2^N uscite
- Ad ogni configurazione di ingresso corrisponde una configurazione di uscita con una unica linea ad 1 e tutte le altre a zero
 - Abbiamo già visto un esempio: decodificatore per un display a sette segmenti
- Le configurazioni in output sono chiamati anche *one-hot* (*perchè solo una linea assume il valore 1*)

Decoder (decodificatore)

- Esempio Decoder 2:4:

| 2:4 Decoder | | A_1 | A_0 | Y_3 | Y_2 | Y_1 | Y_0 |
|----------------|----|-------|-------|-------|-------|-------|-------|
| A_1 | 11 | 0 | 0 | 0 | 0 | 0 | 1 |
| A_0 | 10 | 0 | 1 | 0 | 0 | 1 | 0 |
| | 01 | 1 | 0 | 0 | 1 | 0 | 0 |
| | 00 | 1 | 1 | 1 | 0 | 0 | 0 |

Decoder (decodificatore)

- Esempio Decoder 2:4:

| | | A_1 | A_0 | Y_3 | Y_2 | Y_1 | Y_0 | |
|----------------|--|-------|-------|-------|-------|-------|-------|----------------------------------|
| 2:4 Decoder | | 0 | 0 | 0 | 0 | 0 | 1 | $\text{not}(A_1)\text{not}(A_0)$ |
| A_1 | | 0 | 1 | 0 | 0 | 1 | 0 | $\text{not}(A_1)A_0$ |
| A_0 | | 1 | 0 | 0 | 1 | 0 | 0 | $A_1\text{not}(A_0)$ |
| | | 1 | 1 | 1 | 0 | 0 | 0 | A_1A_0 |

Calcolo minterm

Decoder (decodificatore)

- Esempio Decoder 2:4:

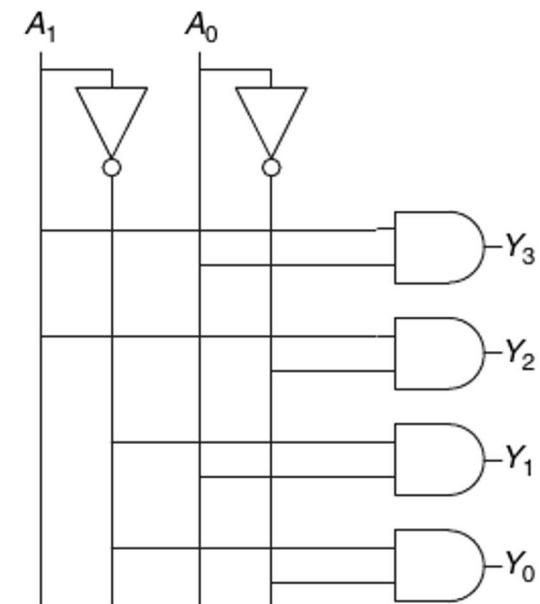
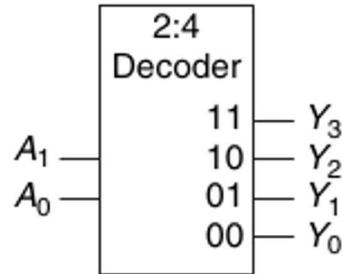
| A_1 | A_0 | Y_3 | Y_2 | Y_1 | Y_0 |
|-------|-------|-------|-------|-------|-------|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |
| --- | | ----- | | | |

$\text{not}(A_1)\text{not}(A_0)$

$\text{not}(A_1)A_0$

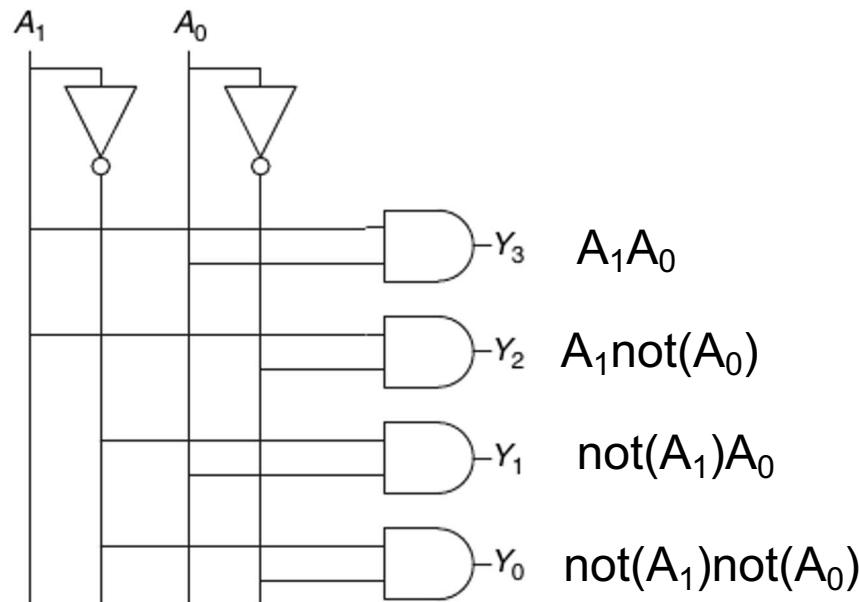
$A_1\text{not}(A_0)$

A_1A_0



Decoder (decodificatore)

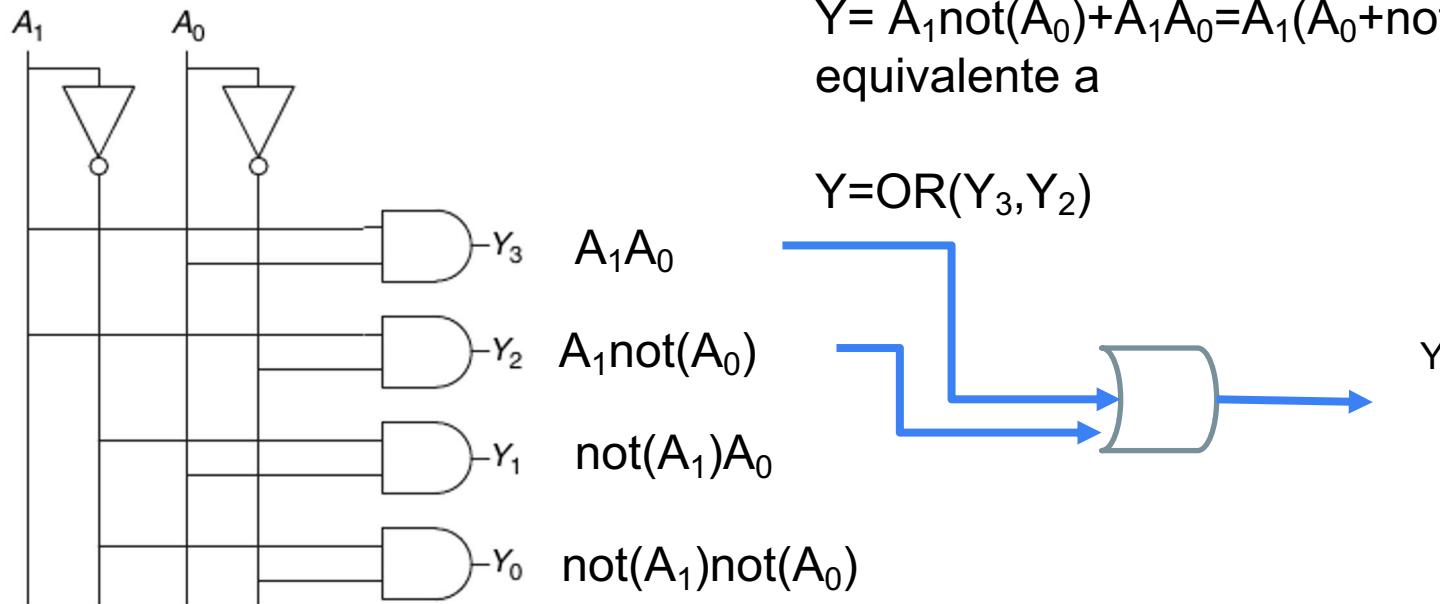
- Decoder possono essere utilizzati per costuire funzioni logiche combinandole con porte OR



Ogni uscita corrisponde ad una particolare configurazione dei due input

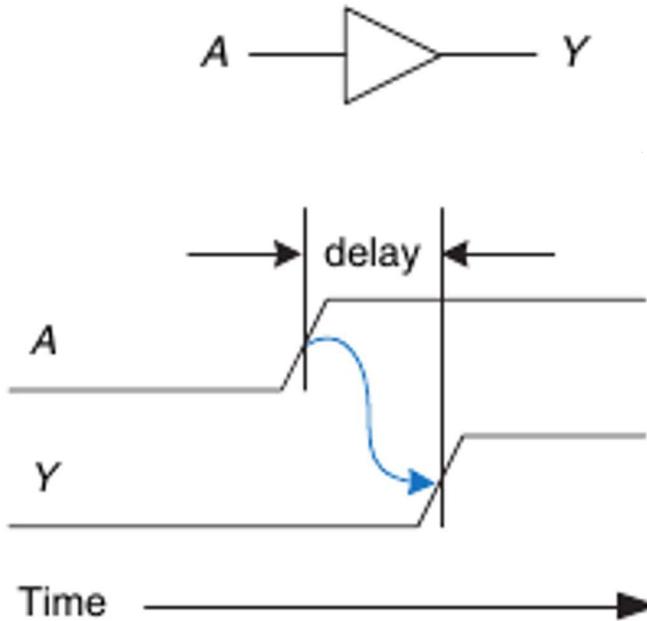
Decoder (decodificatore)

- Decoder possono essere utilizzati per costuire funzioni logiche combinandole con porte OR



Il tempo

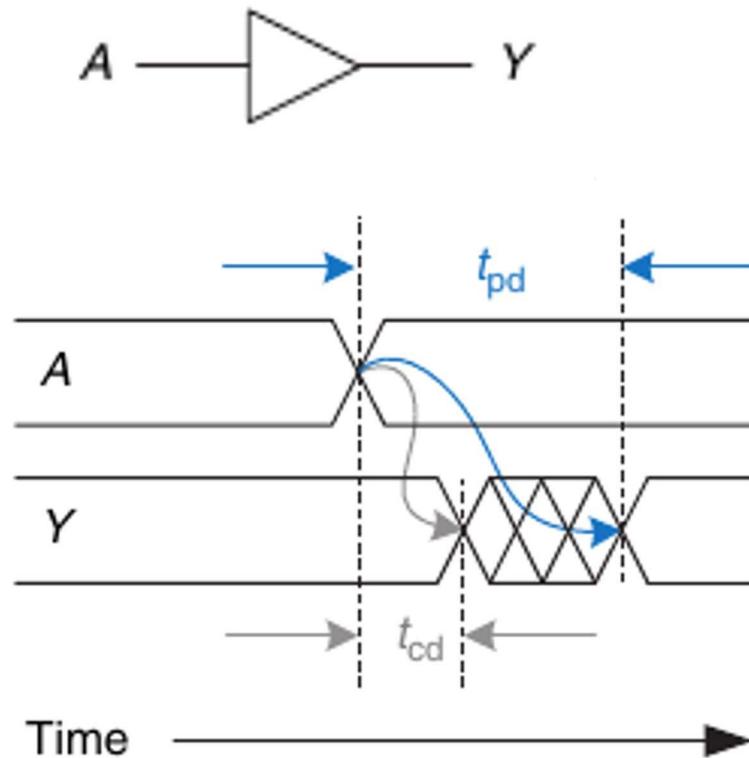
- Ciascun componente impiega un certo tempo a “calcolare” il suo output
- I segnali in ingresso possono variare valore
 - Tali variazioni da “alto” (1) a “basso” (0) (e viceversa) impiegano un certo tempo



In figura la freccia ricurva indica il ridardo con cui vado la variazione di valore su Y

Circuit delays are ordinarily on the order of picoseconds ($1 \text{ ps} = 10^{-12} \text{ seconds}$) to nanoseconds ($1 \text{ ns} = 10^{-9} \text{ seconds}$). Trillions of picoseconds have elapsed in the time you spent reading this sidebar.

Il tempo



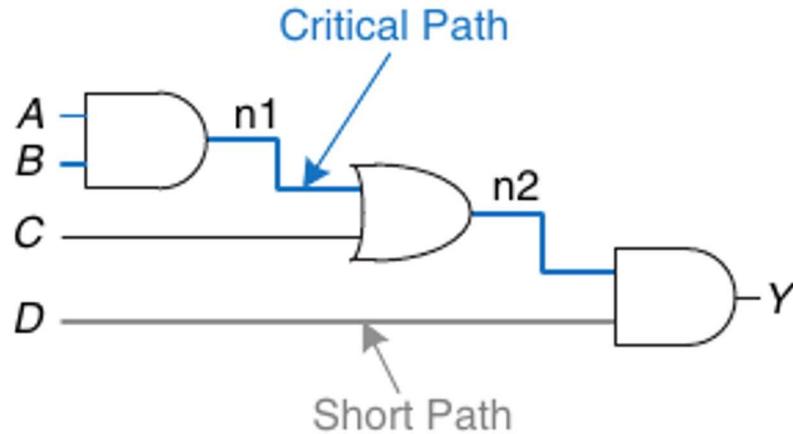
- Supponiamo che il segnale su A (ingresso) **cambi** valore da **ALTO** a **basso** (o viceversa)
- t_{cd} (**contamination delay**) è il tempo minimo per osservare un cambiamento sull'uscita Y
- t_{pd} (**propagation delay**) è il tempo massimo affinché su Y il segnale si stabilizzi al suo nuovo valore
In figura le frecce ricurve indicano i due tempi t_{cd} e t_{pd}

Ci possono essere diverse cause che portano ad un ritardo nella stabilizzazione del segnale di uscita:

- input multipli
- temperatura
- etc...

Il tempo

- t_{cd} (contamination delay) e t_{pd} (propagation delay) dipendono anche dal particolare “path” (cammino) che il segnale percorre



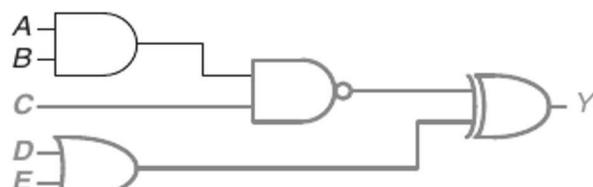
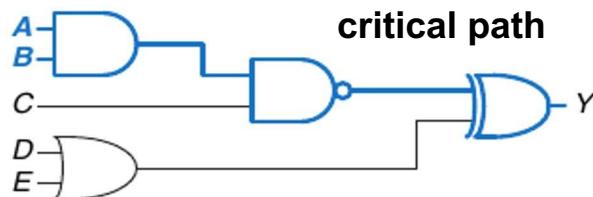
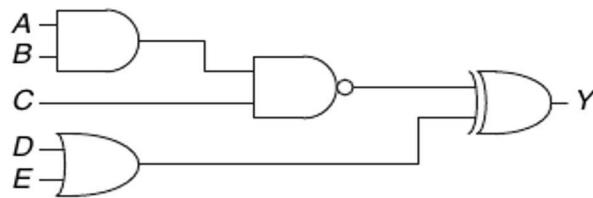
- **Critical path:** tempi più lunghi
- **Short path:** tempi più brevi

- **Ritardi per una composizione di circuiti:**

- t_{cd}^{tot} : somma dei t_{cd} corrispondenti agli elementi dello short path (t_{cd}^{and})
- t_{pd}^{tot} : somma dei t_{pd} corrispondenti degli elementi del critical path ($2t_{pd}^{\text{and}} + t_{pd}^{\text{or}}$)

Il tempo: altro esempio

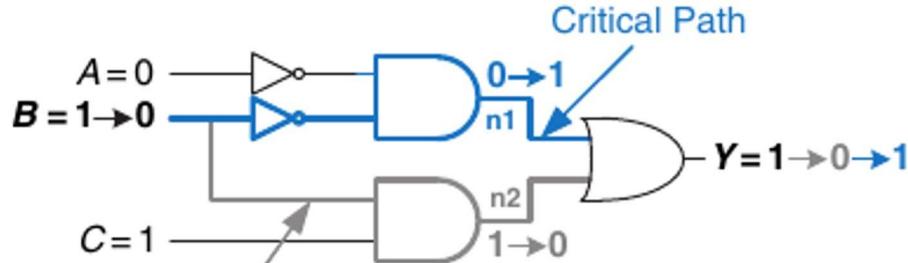
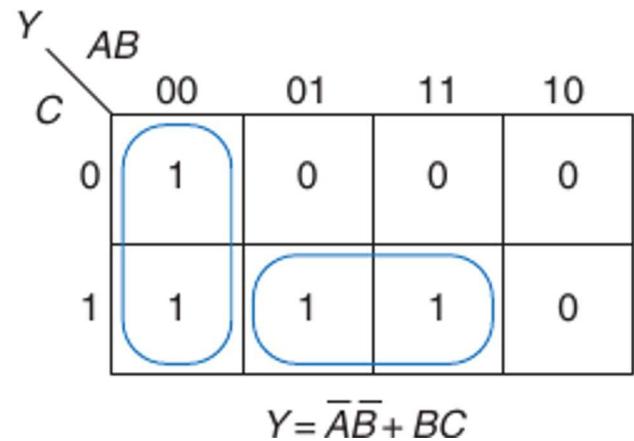
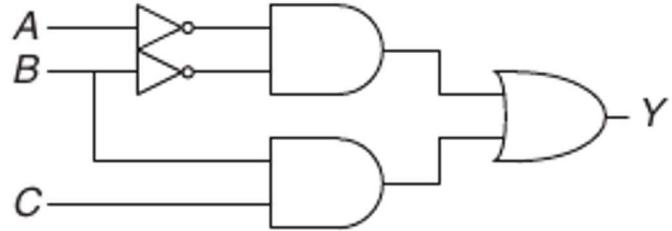
- t_{cd} (contamination delay) e t_{pd} (propagation delay) dipendono anche dal particolare “path” (cammino) che il segnale percorre



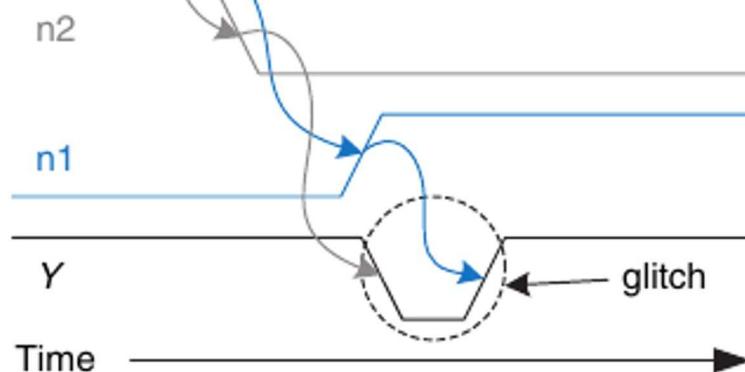
- $t_{cd} = t_{cd}^{\text{or}} + t_{cd}^{\text{xor}}$

- $t_{pd} = t_{pd}^{\text{and}} + t_{pd}^{\text{nand}} + t_{pd}^{\text{xor}}$

Il tempo: glitch



$A=0, B=1 \rightarrow 0, C=1 \rightarrow 1$ $Y=1 \rightarrow 0 \rightarrow 1$

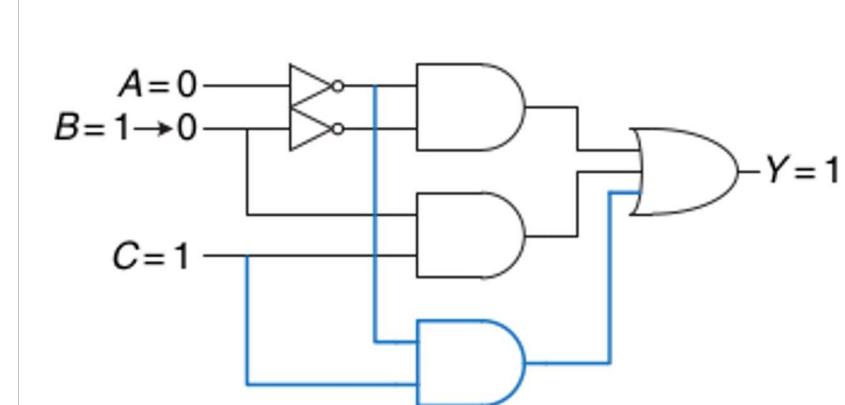
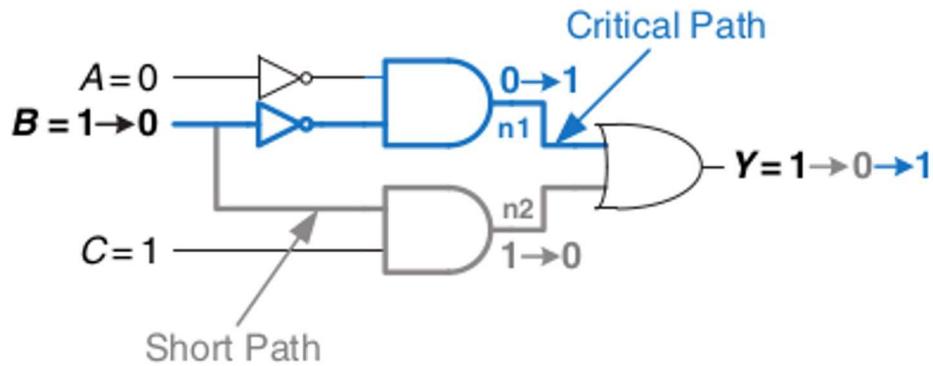


La risposta del circuito ha un “glitch” prima di stabilizzarsi

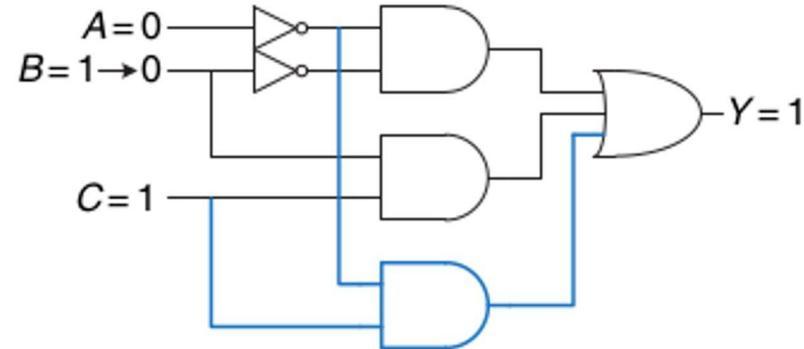
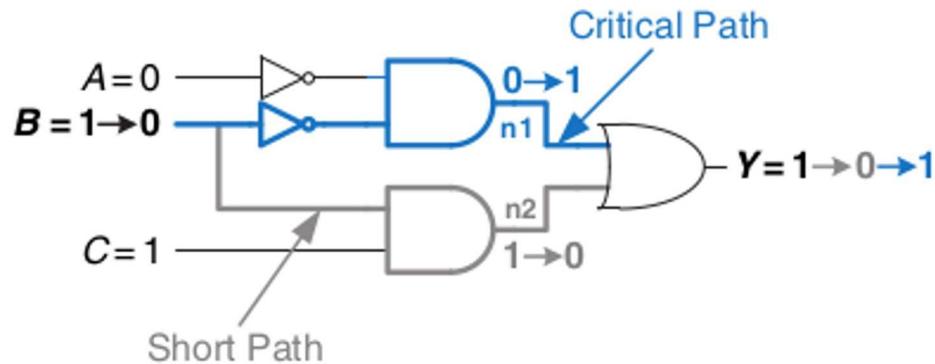
Il tempo: glitch

In genere il glitch non è un problema perché il circuito si stabilizza (non oscilla!)

Se vogliamo evitarlo bisogna cambiare il circuito, in genere aggiungendo componenti:



Il tempo: glitch



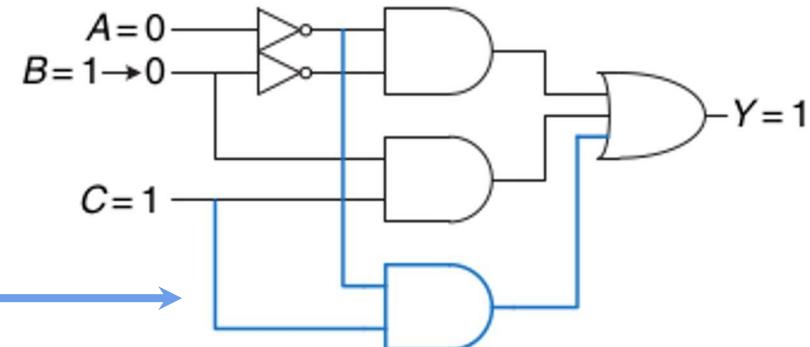
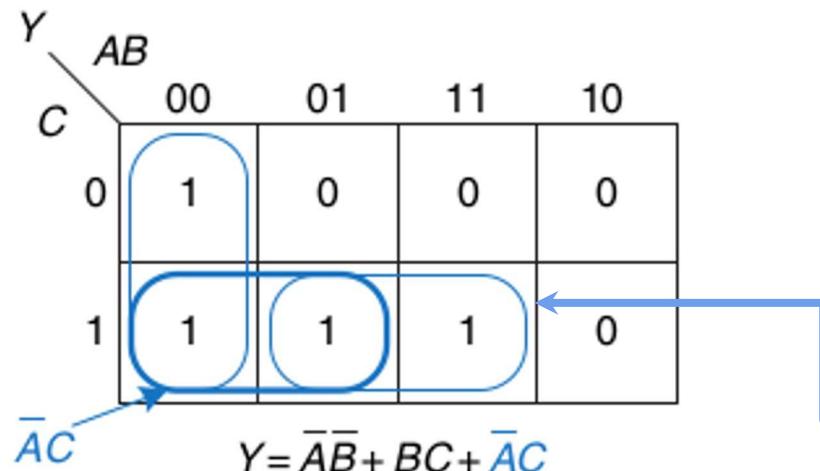
Come abbiamo eliminato il glitch?

- In genere un glitch può presentarsi quando c'e' un cambiamento di una singola variabile attraversando il confine tra 2 implicanti primi in una K-map
- Per evitarlo possiamo aggiungere un implicante ridondante che copra tale confine

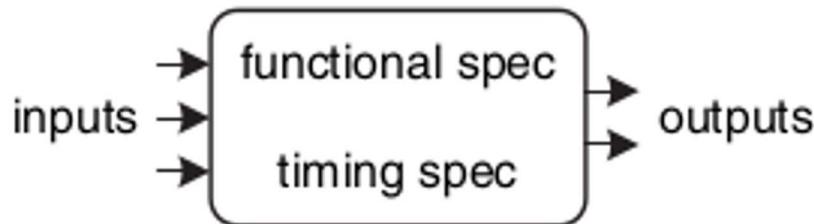
Il tempo: glitch

Come abbiamo eliminato il glitch?

- In genere un glitch può presentarsi quando c'e' un cambiamento di una singola variabile attraversando il confine tra 2 implicanti primi in una K-map
- Per evitarlo possiamo aggiungere un implicante ridondante che copra tale confine



Circuiti combinatori: Riassumendo

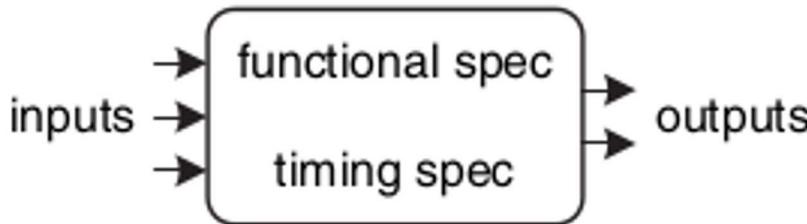


- Abbiamo un certo numero di variabili di terminali (linee,nodi) di **input** su cui è presente un valore discreto (0/1)
- Abbiamo un certo numero di variabili di terminali (linee,nodi) di **output** su cui è presente un valore discreto (0/1)
- E' specificato una **relazione funzionale** tra input ed output
- C'è una **specifica temporale** che ci dice il ritardo tra quando cambia l'input e quando vedo tale cambiamento in output

Circuiti combinatoriali: Riassumendo



Simbolo per un circuito combinatoriale



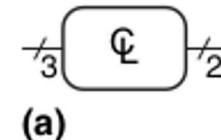
$$Y = F(A, B) = A + B$$

Esempi: **Figure 2.3** Combinational logic circuit

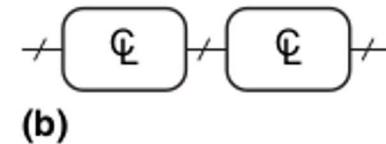


$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

Figure 2.5 Multiple-output combinational circuit



(a)



(b)

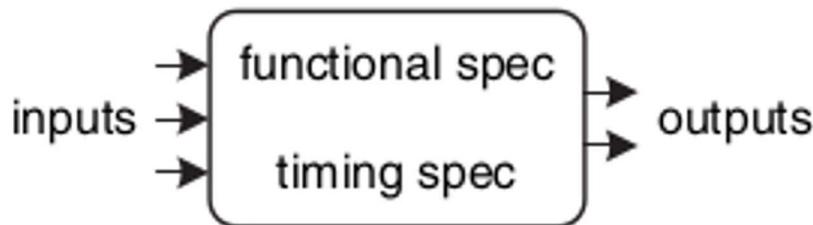
Figure 2.6 Slash notation for multiple signals

Architettura degli Elaboratori

Lezione 18

Docente: R.Prevete
a.a. 2022/2023
26 aprile 2023

Circuiti combinatori: Riassumendo

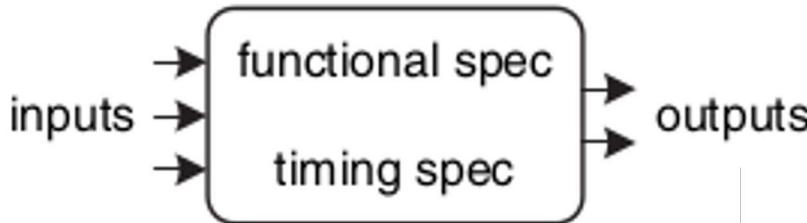


- Abbiamo un certo numero di variabili di terminali (linee,nodi) di **input** su cui è presente un valore discreto (0/1)
- Abbiamo un certo numero di variabili di terminali (linee,nodi) di **output** su cui è presente un valore discreto (0/1)
- E' specificato una **relazione funzionale** tra input ed output
- C'è una **specifica temporale** che ci dice il ritardo tra quando cambia l'input e quando vedo tale cambiamento in output

Circuiti combinatori: Riassumendo



Simbolo per un circuito combinatorio



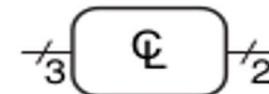
$$Y = F(A, B) = A + B$$

Esempi: **Figure 2.3** Combinational logic circuit



$$\begin{aligned}S &= A \oplus B \oplus C_{in} \\C_{out} &= AB + AC_{in} + BC_{in}\end{aligned}$$

Figure 2.5 Multiple-output combinational circuit



(a)

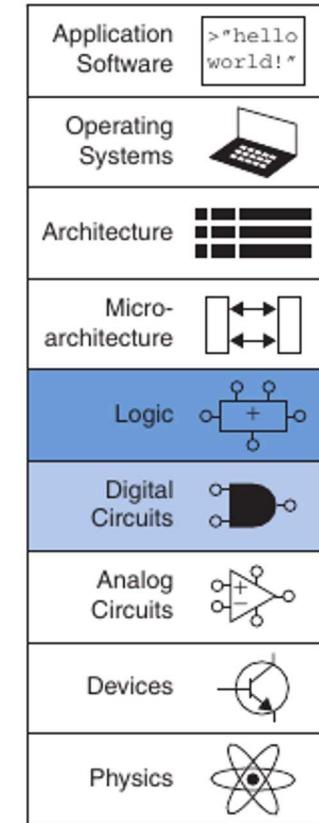


(b)

Figure 2.6 Slash notation for multiple signals

Logica sequenziale

- Nelle precedenti lezioni abbiamo visto come creare circuiti logici il cui output dipende dallo stato corrente degli input
- E' possibile costruire circuiti che dipendono anche da cosa è accaduto nel passato?
 - SI!!!
 - In questo caso si parla di ***logica sequenziale***



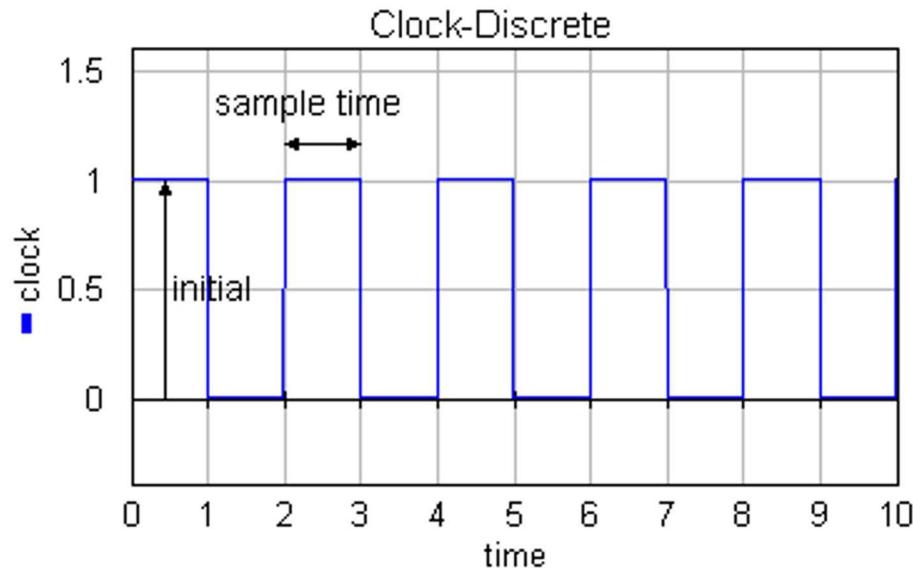
Logica sequenziale

- Due caratteristiche principali dei circuiti sequenziali
 - **Il tempo** (è da prendere in considerazione)
 - **Stato del sistema** (stato interno)
 - È una memoria, mantiene informazioni su cosa è “capitato” nel passato (input od altro)
 - Lo stato del sistema è rappresentato da una o più variabili dette **variabili di stato**
 - queste *variabili di stato* permettono, insieme al input corrente, di determinare l'output del sistema ed il proprio prossimo valore

Logica sequenziale

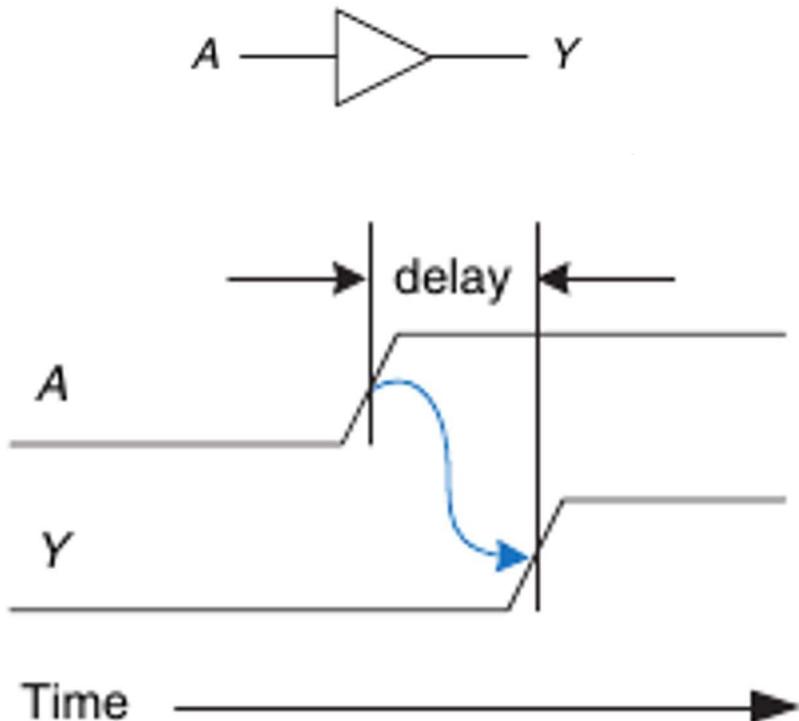
- **Di cosa parleremo**
 - In che senso prendiamo il tempo in considerazione? (oggi)
 - Elemento Bi-stabile (oggi)
 - Latches (oggi)
 - Flip-flop
 - Circuiti sincroni di logica sequenziale
 - Composti da flip-flop e latches
 - Macchine a stati finiti

Il tempo: un modo di vedere le cose



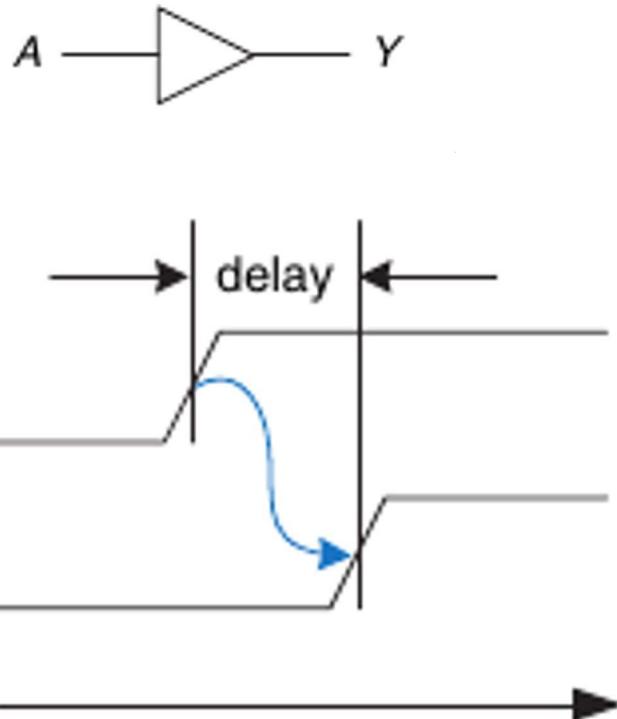
- Possiamo immaginare i tempo come una entità discreta con passo DT
- DT sarà la nostra unità di tempo
- Tutte le variabili di ingresso e di uscita saranno aggiornate in maniera **sincrona ad ogni passo**

Il tempo: un modo di vedere le cose



- **Avremo, quindi:**
- $t=0, A(t), Y(t) = A(0), Y(0)$
- $t=1, A(t), Y(t) = A(1), Y(1)$
- $t=2, A(t), Y(t) = A(2), Y(2)$
- $t=3, A(t), Y(t) = A(3), Y(3)$
- ...

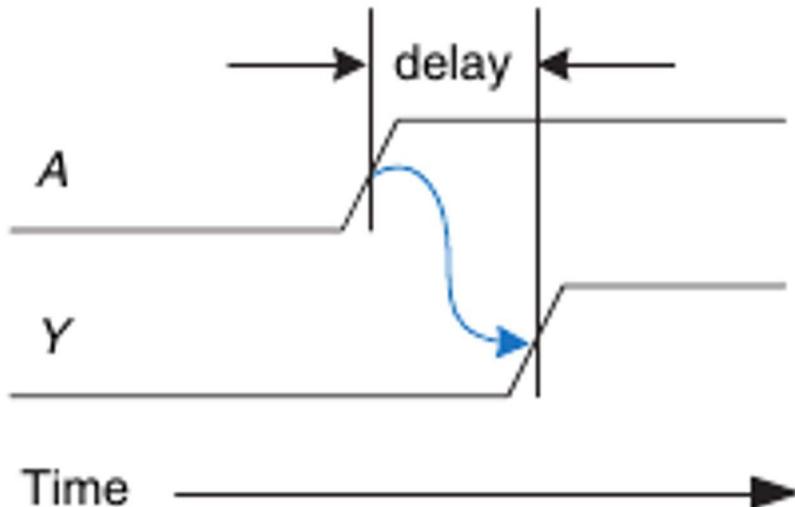
Il tempo: un modo di vedere le cose



- I valori delle variabili di output e delle variabili di stato al tempo $t+1$ saranno calcolati sulla base dei valori di ingresso e delle variabili di stato al tempo t

- $t=0$, $A(0)$, $Y(0)$ $Y(0)$ indeterminato
- $t=1$, $A(1)$, $Y(1)$ $Y(1)$ dipende da $A(0)$
- $t=2$, $A(2)$, $Y(2)$ $Y(2)$ dipende da $A(1)$
- ...

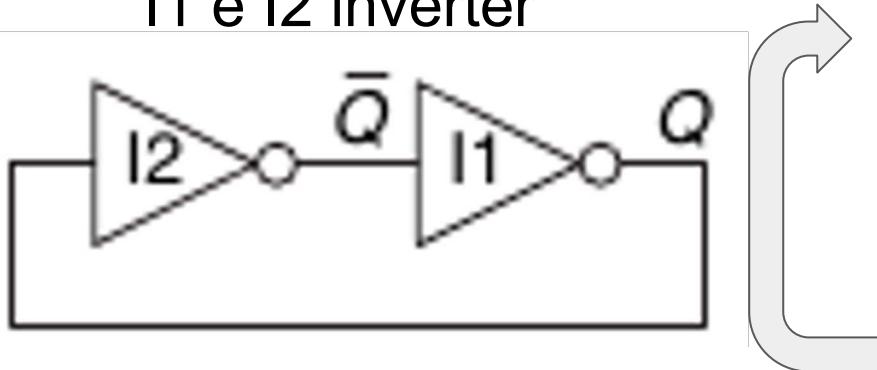
Il tempo: un modo di vedere le cose



- Per specificare il funzionamento di un circuito dobbiamo specificare, allora, dati i valori delle variabili al tempo t (tempo corrente) quali sono i valori delle variabili al tempo $t+1$ (tempo successivo)
 - (tempo corrente) → (tempo succivo)
- **Tabella di verità**, allora, deve catturare proprio tale relazione funzionale!

Logica sequenziale: elemento bi-stabile

I1 e I2 inverter



Mantiene stabile un
valore: 0 o 1

- **Q=0**

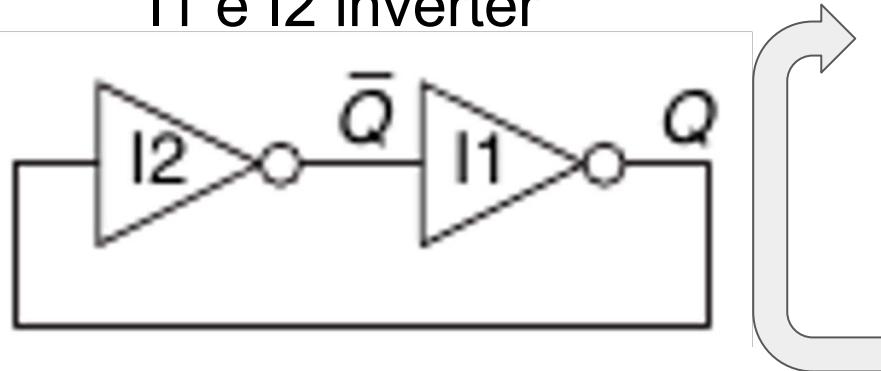
- I2 riceve 0, e come output ha 1
- Quindi **Q segnato** diventa 1, ed è in input a I1
- I1 inverte 1 a 0, e Q rimane 0

- **Q=1**

- I2 riceve 1, e come output ha 0
- Quindi Q segnato diventa 0, ed è in input a I1
- I1 inverte 0 a 1, e Q rimane 1

Logica sequenziale: elemento bi-stabile

I1 e I2 inverter



- **Osserviamo:**

- Andamento ciclico
- Doppialmente (bi) stabile, sia $Q=0$ sia $Q=1$ restano stabili

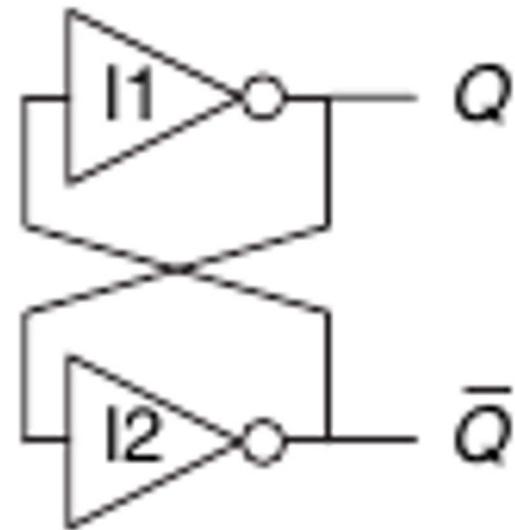
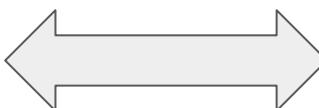
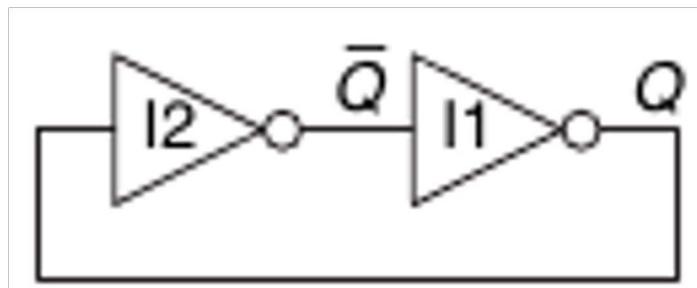
- $Q=0$

- I2 riceve 0, e come output ha 1
- Quindi \bar{Q} diventa 1, ed è in input a I1
- I1 inverte 1 a 0, e Q rimane 0

- $Q=1$

- I2 riceve 1, e come output ha 0
- Quindi \bar{Q} diventa 0, ed è in input a I1
- I1 inverte 0 a 1, e Q rimane 1

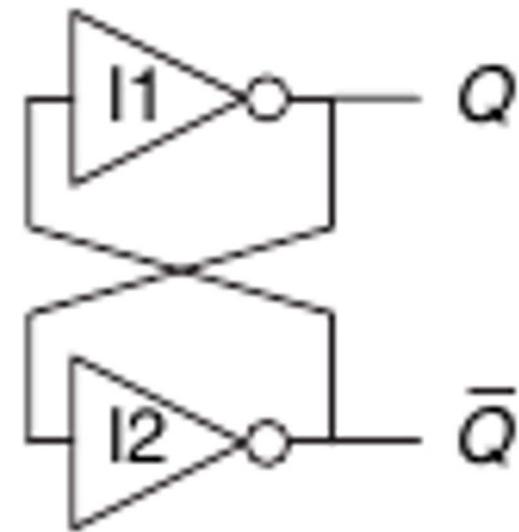
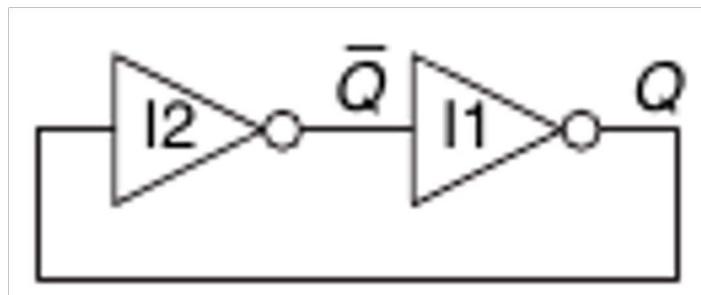
Logica sequenziale: elemento bi-stabile



| Q corrente | Q' (Q prossimo) |
|------------|-----------------|
| 0 | 0 |
| 1 | 1 |

Tabella di verità

Logica sequenziale: elemento bi-stabile



In questo tipo di circuito la variabile Q nel
tempo rimane invariata

Nota: Q è usato, in genere, come simbolo per la
variabile di output dei circuiti sequenziali

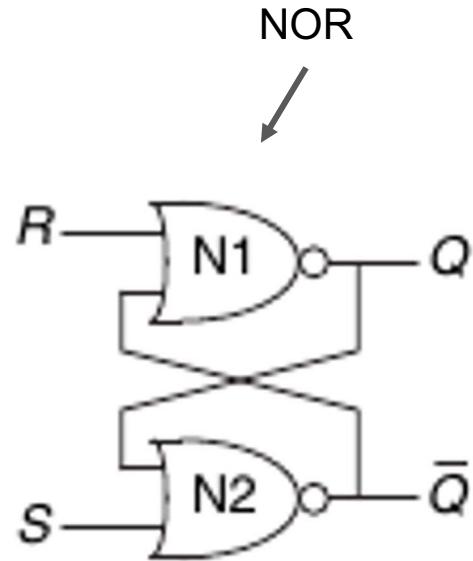
Logica sequenziale: elemento bi-stabile

Alcune osservazioni:

- Un circuito bistabile mantiene 2 stati possibili, quindi corrisponde ad 1 bit
- N circuiti bistabile mantengono 2^N stati (configurazioni) possibili
- Un circuito con N stati stabili mantiene $\log_2 N$ bit
- La variabile di stato corrisponde con Q: il valore di Q ci dice qualunque cosa sui valori futuri e su quelli passati
- Un circuito bistabile, così come descritto, non ha un input (**per questo motivo ci serve a poco!**)

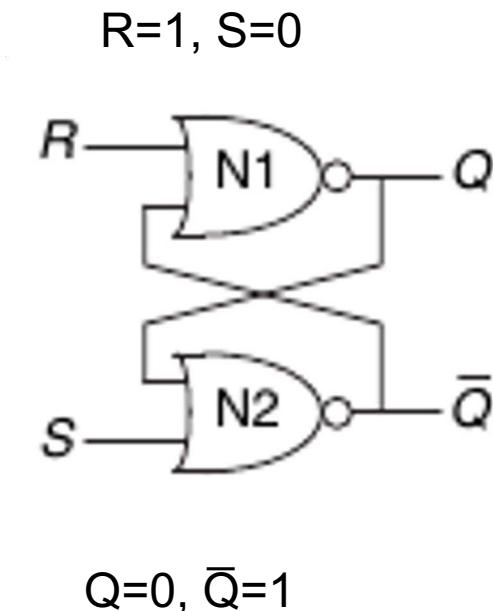
Logica sequenziale: SR Latch

- Ha due input: S, R
- Due Output: Q e \bar{Q} (Q negato)
- È simile al circuito bi-stabile
- Il suo stato può essere determinato dagli input R,S
 - *S determina il valore in Q*
 - *R azzera (reset) il valore di Q*



Logica sequenziale: SR Latch

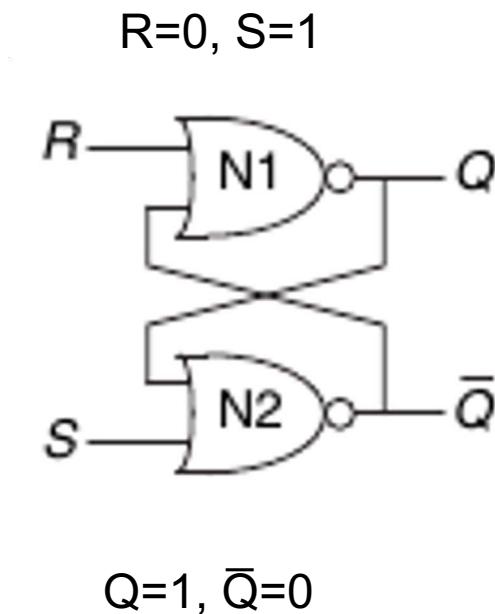
- CASO 1: $R=1, S=0$
 - Q e \bar{Q} inizialmente indeterminati, $t=0$**
 - $N1$ riceve $R=1$ e \bar{Q} indeterminato, quindi $Q=0$ a $t+1$**
 - $N2$ riceve $S=0$, ma Q ancora indeterminato, quindi \bar{Q} indeterminato a $t+1$**
 - $t=t+1$ ($Q=0$, \bar{Q} indeterminato)**
 - $N1$ riceve sempre $R=1$, e \bar{Q} indeterminato, quindi $Q=0$ a $t+1$**
 - $N2$ riceve sempre $S=0$ e $Q=0$, quindi $\bar{Q}=1$ a $t+1$**
 - $t=t+1$ ($Q=0$, $\bar{Q} = 1$)**
 - Ripeti da e. ($Q=0$, $\bar{Q} = 1$ restano invariati)**



Logica sequenziale: SR Latch

- CASO 2: $R=0, S=1$

- Q e \bar{Q} inizialmente indeterminati, $t=0$
- N1** riceve $R=0$ e \bar{Q} indeterminato, quindi Q indeterminato a $t+1$
- N2** riceve $S=1$, e Q indeterminato, quindi $\bar{Q} = 0$ a $t+1$
- $t=t+1$ ($\bar{Q} = 0$, Q indeterminato)
- N1** riceve sempre $R=0$, ma $\bar{Q} = 0$, quindi $Q = 1$ a $t+1$
- N2** riceve sempre $S=1$ e Q indeterminato, quindi ancora $\bar{Q}=0$ t+1
- $t=t+1$ ($Q=1, \bar{Q} = 0$)
- Ripeti da e. con $Q=1, \bar{Q} = 0$, restano invariati

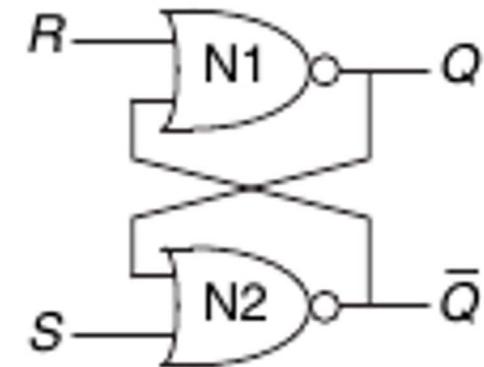


Logica sequenziale: SR Latch

- CASO 3: $R=1, S=1$

- Q e \bar{Q} inizialmente indeterminati, $t=0$
- N1** riceve $R=1$ e \bar{Q} indeterminato, quindi $Q=0$ a $t+1$
- N2** riceve $S=1$, e Q indeterminato, quindi $\bar{Q}=0$ a $t+1$
- $t=t+1$ ($\bar{Q}=0$, $Q=0$)
- N1** riceve sempre $R=1$, ma $\bar{Q}=0$, quindi $Q=0$ a $t+1$
- N2** riceve sempre $S=1$ e $Q=0$, quindi ancora $\bar{Q}=0$ a $t+1$
- $t=t+1$ ($Q=0$, $\bar{Q}=0$)
- Ripeti da e., restano invariati

$R=1, S=1$



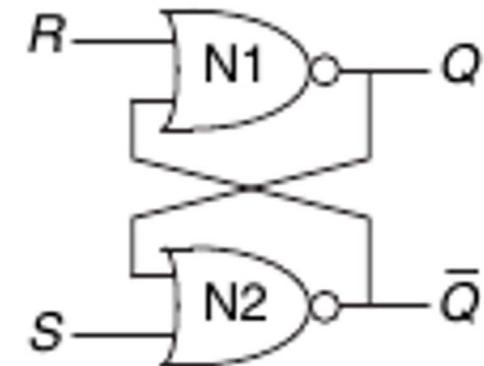
$Q=0, \bar{Q}=0$

Non è quello che vogliamo!!!

Logica sequenziale: SR Latch

- CASO 4: R=0, S=0
 - a. Q e \bar{Q} inizialmente indeterminati, t=0
 - b. N1 riceve R=0 e \bar{Q} indeterminato, quindi Q indeterminato a t+1
 - c. N2 riceve S=0, e Q indeterminato, quindi \bar{Q} indeterminato a t+1
 - d. t=t+1 (\bar{Q} indeterminato, Q indeterminato)
 - e. Ripeti da b., \bar{Q} e Q rimangono indeterminati

R=0, S=0



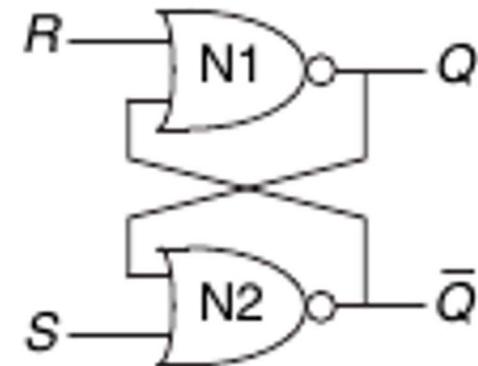
Q indeterminato, \bar{Q} indeterminato

In questo caso, allora, i valori di Q e \bar{Q} non possono essere determinati senza sapere i loro valori iniziali!!!

Logica sequenziale: SR Latch

- CASO 4a: $R=0$, $S=0$, $Q=0$, $\bar{Q}=1$
 - a. $Q=0$ e $\bar{Q}=1$, $t=0$
 - b. N1 riceve $R=0$ e $\bar{Q}=1$, quindi $Q=0$ $t+1$
 - c. N2 riceve $S=0$, e $Q=0$, quindi $\bar{Q}=1$ a $t+1$
 - d. $t=t+1$ ($Q=0$, $\bar{Q}=1$)
 - e. Ripeti da a., restano invariati

$$R=0, S=0, \\ Q=0, \bar{Q}=1$$

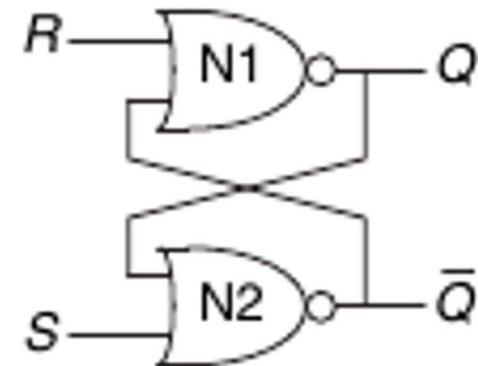


Q , \bar{Q} assumono il valore precedente

Logica sequenziale: SR Latch

- CASO 4b: $R=0$, $S=0$, $Q=1$, $\bar{Q}=0$
 - a. $Q=1$ e $\bar{Q}=0$, $t=0$
 - b. N1 riceve $R=0$ e $\bar{Q}=0$, quindi $Q=1$ $t+1$
 - c. N2 riceve $S=0$, e $Q=1$, quindi $\bar{Q}=0$ a $t+1$
 - d. $t=t+1$ ($Q=1$, $\bar{Q}=0$)
 - e. Ripeti da b., restano invariati

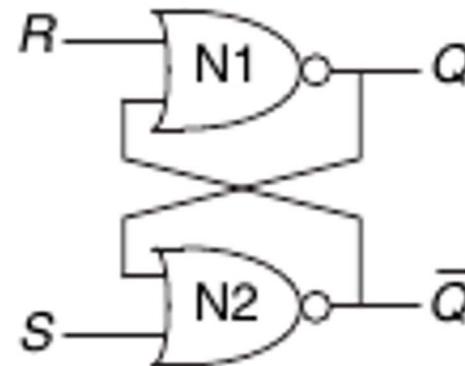
$$R=0, S=0, \\ Q=0, \bar{Q}=1$$



Q , \bar{Q} assumono il valore precedente

Logica sequenziale: SR Latch

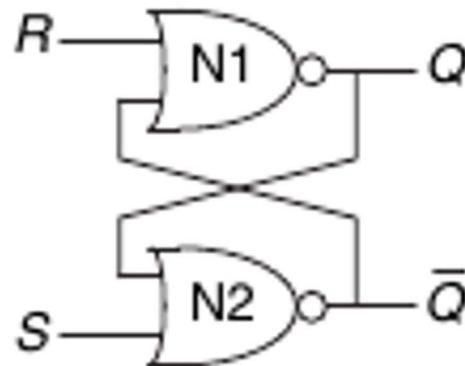
| S | R | Q_{n+1} | \bar{Q}_{n+1} | prossimo stato |
|---|---|-----------|-----------------|----------------|
| 0 | 0 | Q_n | \bar{Q}_n | stato attuale |
| 0 | 1 | 0 | 1 | reset |
| 1 | 0 | 1 | 0 | set |
| I | I | X | X | non ammesso |



Osservazione 1: Quando gli ingressi vengono attivati attraverso opportuni livelli logici, l'uscita Q e di conseguenza il suo complemento \bar{Q} possono essere portate in uno dei due stati 0 o 1, nei quali può rimanere stabilmente anche quando gli ingressi vengono disattivati.

Logica sequenziale: SR Latch

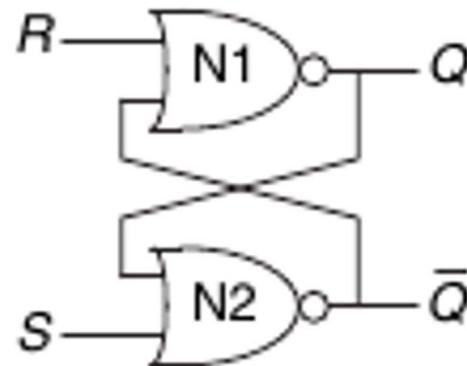
| S | R | Q_{n+1} | \bar{Q}_{n+1} | |
|-----|-----|-----------|-----------------|--------------------|
| 0 | 0 | Q_n | \bar{Q}_n | <i>memoria</i> |
| 0 | 1 | 0 | 1 | <i>reset</i> |
| 1 | 0 | 1 | 0 | <i>set</i> |
| 1 | 1 | X | X | <i>non ammesso</i> |



Osservazione 2: Si tratta, quindi, di un dispositivo bistabile, cioè dotato di due stati stabili nei quali può rimanere bloccato (latch=chiavistello) e mantenere (memorizzare) un bit.

Logica sequenziale: SR Latch

| S | R | Q_{n+1} | \bar{Q}_{n+1} | |
|-----|-----|-----------|-----------------|--------------------|
| 0 | 0 | Q_n | \bar{Q}_n | <i>memoria</i> |
| 0 | 1 | 0 | 1 | <i>reset</i> |
| 1 | 0 | 1 | 0 | <i>set</i> |
| 1 | 1 | X | X | <i>non ammesso</i> |



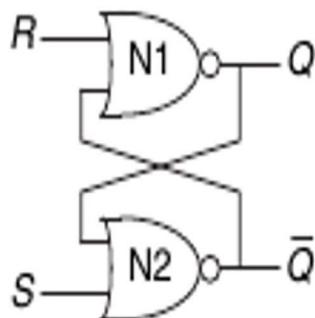
Osservazione 3: L'ultima condizione, $R=1$ e $S=1$, comporta una situazione di ambiguità ed è inutilizzabile. In questo caso, infatti, entrambe le uscite sono poste 0, ma Q e \bar{Q} devono essere complementari tra di loro altrimenti ci possono essere gravi inconvenienti (instabilità).

Logica sequenziale: SR Latch

Supponiamo, infatti,
che da $S=1$, $R=1$
(che implica
 $Q=0, \bar{Q}=0$),
passiamo a $R=0$,
 $S=0$,

Caso ambiguo !!

- **CASO: $R=0, S=0, Q=0, \bar{Q}=0$**
 - a. $Q=0$ e $\bar{Q}=0$, $t=0$
 - b. N1 riceve $R=0$ e $\bar{Q}=0$, quindi $Q=1$ a $t+1$
 - c. N2 riceve $S=0$, e $Q=0$, quindi $\bar{Q}=1$ a $t+1$
 - d. $t=t+1$ ($Q=1, \bar{Q}=1$)
 - e. N1 riceve $R=0$ e $\bar{Q}=1$, quindi $Q=0$ a $t+1$
 - f. N2 riceve $S=0$, e $Q=1$, quindi $\bar{Q}=0$ a $t+1$
 - g. Ripeti da b., Q e \bar{Q} non assumono un valore stabile



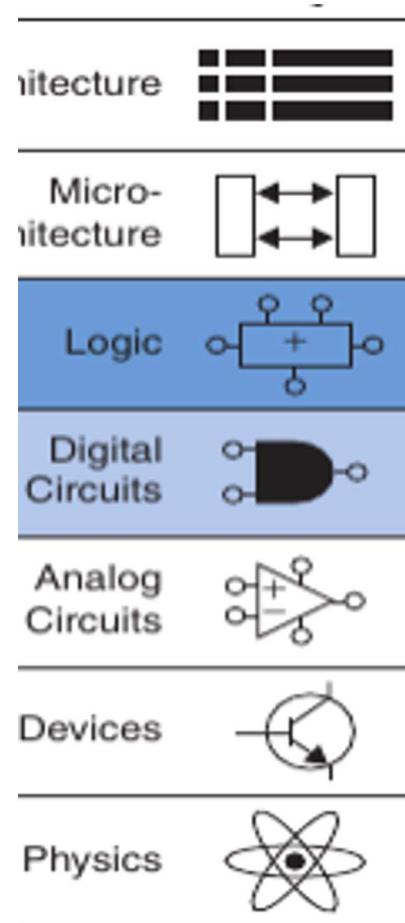
Architettura degli Elaboratori

Lezione 19

Docente: R.Prevete
a.a. 2022/2023
28 aprile 2023

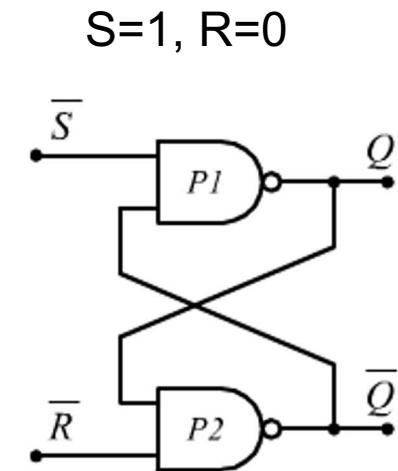
Logica sequenziale

- Di cosa parleremo
 - Ancora SR-Latch
 - D-Latch
 - flip-flop
 - Registri
 - macchine a stati finiti



Logica sequenziale: SR Latch con NAND

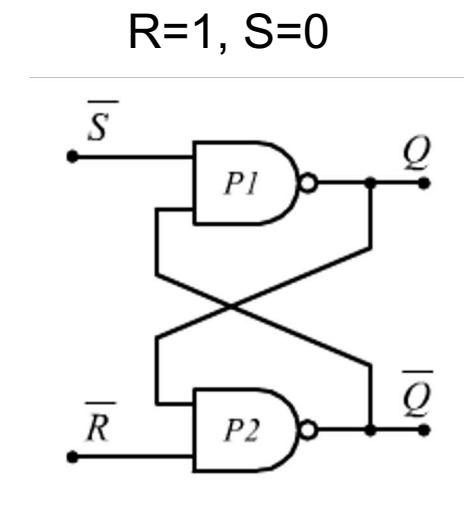
- CASO 1: S=1, R=0
 - a. Q e \bar{Q} inizialmente indeterminati, t=0
 - b. P1 riceve S=1, cioè $\bar{S} = 0$, ed \bar{Q} indeterminato, quindi **Q=1 a t+1**
 - c. P2 riceve R=0, cioè, $\bar{R}=1$, ma Q ancora indeterminato, quindi \bar{Q} indeterminato a t+1
 - d. **t=t+1 (Q=1, \bar{Q} indeterminato)**
 - e. P1 riceve sempre 0 e \bar{Q} indeterminato, quindi **Q=1 a t+1**
 - f. P2 riceve sempre 1 e Q=1, quindi $\bar{Q}=0$ t+1
 - g. **t=t+1 (Q=1, $\bar{Q} = 0$)**
 - h. Ripeti da e. (**Q=1, $\bar{Q} = 0$ restano invariati**)



Q=1, $\bar{Q}=0$

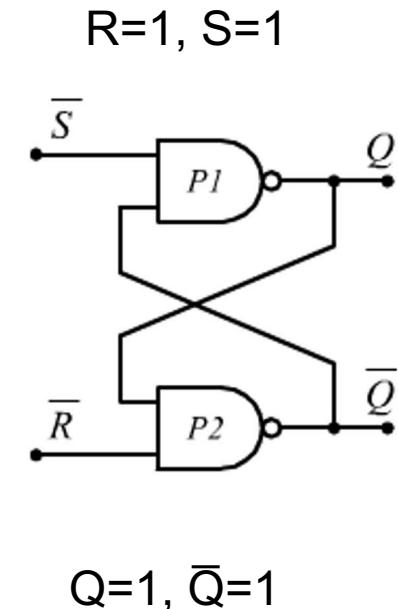
Logica sequenziale: SR Latch con NAND

- CASO 2: S=0, R=1
 - a. Q e \bar{Q} inizialmente indeterminati, t=0
 - b. P1 riceve S=0, cioè $\bar{S} = 1$, ed \bar{Q} indeterminato, quindi Q indeterminato a t+1
 - c. P2 riceve R=1, cioè, $\bar{R}=0$, e Q ancora indeterminato, quindi $\bar{Q} = 1$ a t+1
 - d. t=t+1 (Q indeterminato, $\bar{Q} = 1$)
 - e. P1 riceve sempre $\bar{S} = 1$ e $\bar{Q} = 1$, quindi Q=0 a t+1
 - f. P2 riceve sempre $\bar{R}=0$ e Q indeterminato quindi $\bar{Q}=1$ t+1
 - g. t=t+1 (Q=0, $\bar{Q} = 1$)
 - h. Ripeti da e. (Q=0, $\bar{Q} = 1$ restano invariati)

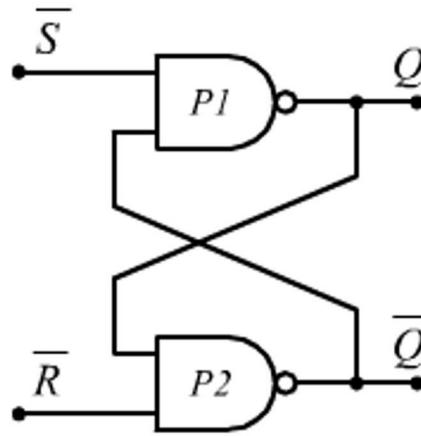


Logica sequenziale: SR Latch con NAND

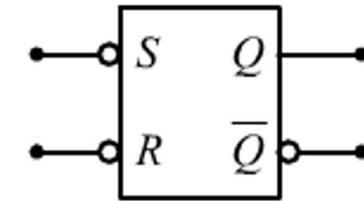
- CASO 3: S=1, R=1 (non ammesso)
 - a. Q e \bar{Q} inizialmente indeterminati, t=0
 - b. P1 riceve S=1, cioè $\bar{S} = 0$, ed \bar{Q} indeterminato, quindi $Q = 1$ a t+1
 - c. P2 riceve R=1, cioè, $\bar{R}=0$, e Q ancora indeterminato, quindi $\bar{Q} = 1$ a t+1
 - d. t=t+1 ($Q = 1$, $\bar{Q} = 1$)
 - e. P1 riceve sempre $\bar{S} = 0$ e $\bar{Q} = 1$, quindi $Q = 1$ a t+1
 - f. P2 riceve sempre $\bar{R}=0$ e $Q=1$ quindi $\bar{Q}=1$ t+1
 - g. t=t+1 ($Q=1$, $\bar{Q} = 1$)
 - h. Ripeti da e. ($Q=1$, $\bar{Q} = 1$ restano invariati)



Logica sequenziale: SR Latch con NAND



| \bar{S} | \bar{R} | Q_{n+1} | \bar{Q}_{n+1} | |
|-----------|-----------|-----------|-----------------|-------------|
| 0 | 0 | X | X | non ammesso |
| 0 | 1 | 1 | 0 | set |
| 1 | 0 | 0 | 1 | reset |
| 1 | 1 | Q_n | \bar{Q}_n | memoria |



| S | R | Q_{n+1} | \bar{Q}_{n+1} | |
|-----|-----|-----------|-----------------|-------------|
| 0 | 0 | Q_n | \bar{Q}_n | memoria |
| 0 | 1 | 0 | 1 | reset |
| 1 | 0 | 1 | 0 | set |
| 1 | 1 | X | X | non ammesso |

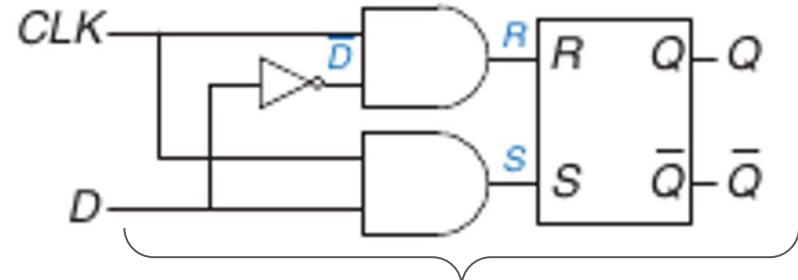
Con NOR

D latch

- Risolve problemi di SR latch
- Due input:
 - D (data input) controlla **quale** è il prossimo stato
 - **CLK** (clock input) controlla **se** lo stato deve cambiare



Concetto Fondamentale!!! — —



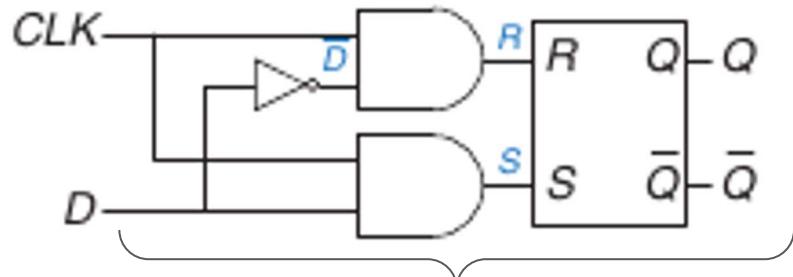
Costruendo su un circuito precedente

| S | R | Q_{n+1} | \bar{Q}_{n+1} | |
|---|---|-----------|-----------------|-------------|
| 0 | 0 | Q_n | \bar{Q}_n | memoria |
| 0 | 1 | 0 | 1 | reset |
| 1 | 0 | 1 | 0 | set |
| 1 | 1 | X | X | non ammesso |

SR latch

D latch

- Caso **CLK=1, D=0**
 - R=1 S=0
 - Q=0 , $\bar{Q}=1$
- Caso **CLK=1, D=1**
 - R=0,S=1
 - Q=1 , $\bar{Q}=0$
- Caso **CLK=0, D=0**
 - R=0,S=0
 - Q, \bar{Q} stato precedente
- Caso **CLK=0, D=1**
 - R=0,S=0
 - Q, \bar{Q} stato precedente



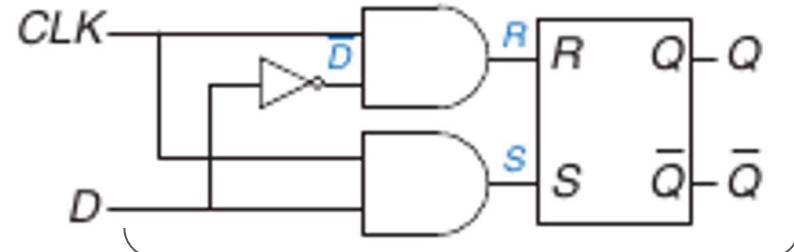
Costruendo su un circuito precedente

| S | R | Q_{n+1} | \bar{Q}_{n+1} | |
|---|---|-----------|-----------------|-------------|
| 0 | 0 | Q_n | \bar{Q}_n | memoria |
| 0 | 1 | 0 | 1 | reset |
| 1 | 0 | 1 | 0 | set |
| 1 | 1 | X | X | non ammesso |

SR latch

D latch

| CLK | D | \bar{D} | S | R | Q | \bar{Q} |
|-------|-----|-----------|-----|-----|------------|------------------|
| 0 | X | X | 0 | 0 | Q_{prev} | \bar{Q}_{prev} |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| .. | | | | | | |



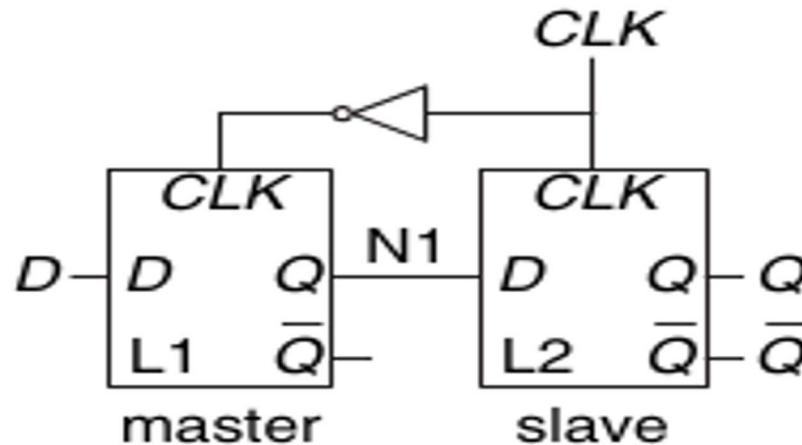
Costruendo su un circuito precedente

- $CLK = 0$ D latch è “opaco” (**Non** lascia passare il segnale di input)
- $CLK = 1$ D latch è “trasparente” (lascia passare il segnale di input)

| S | R | Q_{n+1} | \bar{Q}_{n+1} | |
|-----|-----|-----------|-----------------|-------------|
| 0 | 0 | Q_n | \bar{Q}_n | memoria |
| 0 | 1 | 0 | 1 | reset |
| 1 | 0 | 1 | 0 | set |
| I | I | X | X | non ammesso |

SR latch

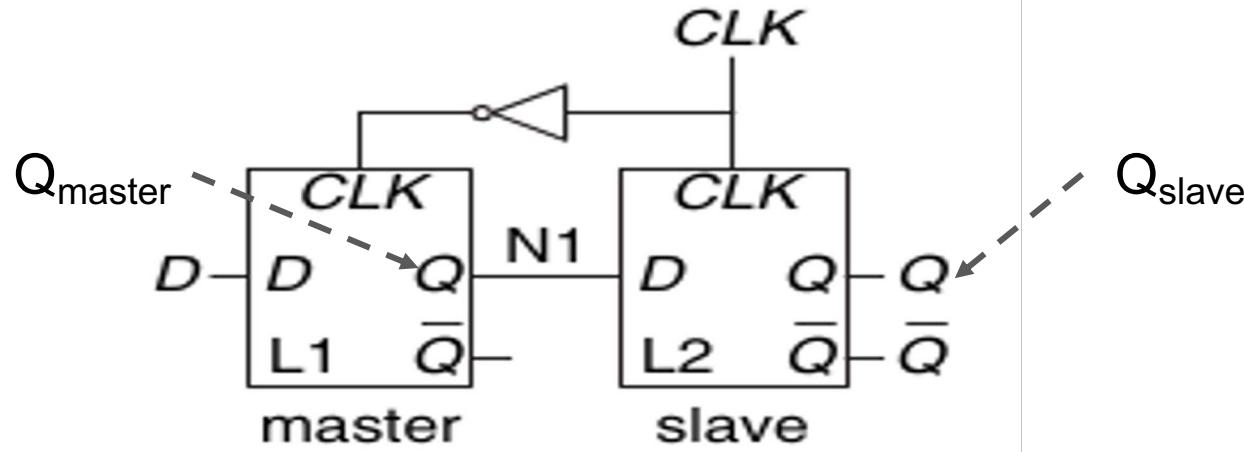
Flip-Flop



Composizione di 2 D-Latch:

- Il D-Latch a sinistra (detto **master**):
 - **Input**: riceve dati D e il negato del CLK
 - **Output**: Q(N1) inviato come input dati al secondo D-Latch (a destra)
- Il D-Latch a destra (detto **slave**):
 - **Input**: riceve Q (N1) dal master e CLK
 - **Output**: Q e \bar{Q}

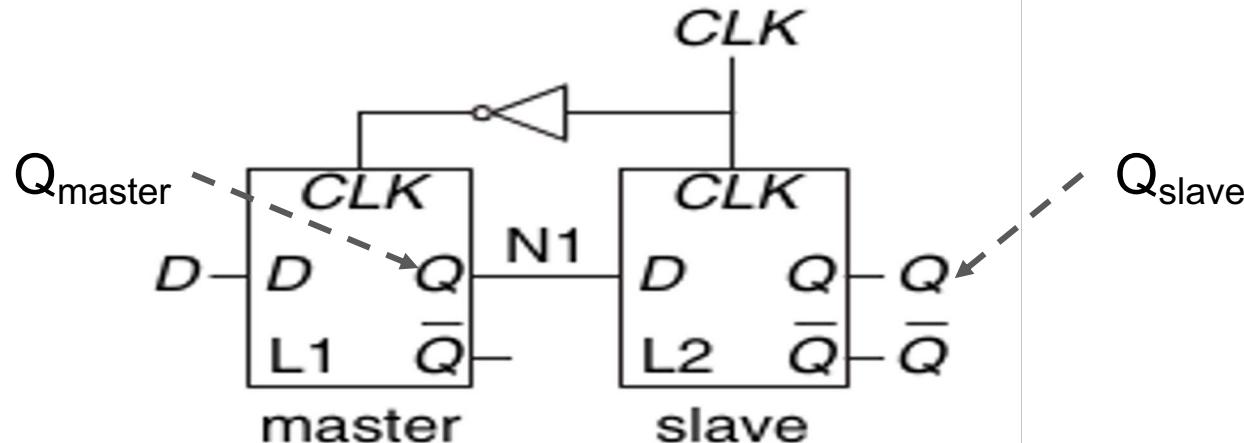
Flip-Flop



CLK=0:

- Il master è “attivato”, trasparente, quindi i dati D passano e Q_{master} cambierà valore a D. Cioè $Q_{master}^{next}=D$
- Lo slave è “disattivo”, opaco quindi non fa passare i dati, quindi Q_{slave} rimarrà invariato, cioè $Q_{slave}^{new}=Q_{slave}^{old}$

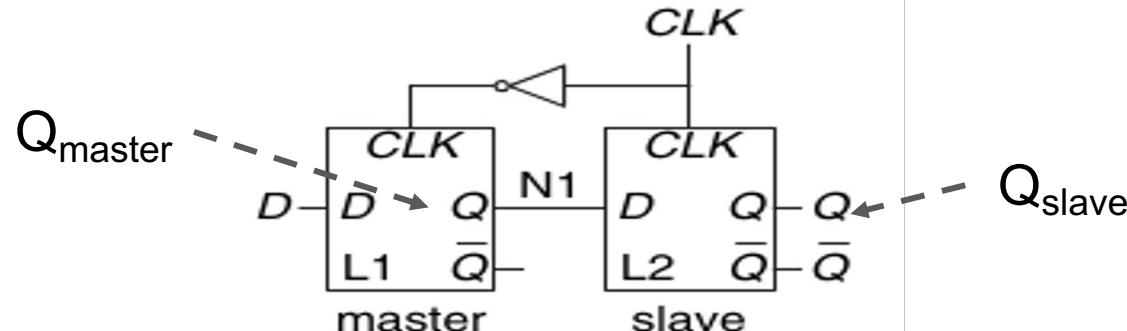
Flip-Flop



CLK=1:

- Il master è “disattivato”, opaco quindi i dati D stavolta non passano e resta il vecchio valore di Q_{master}
- Lo slave è “attivo” trasparente quindi Q_{slave} assume il valore dell’attuale D_{slave} , cioè Q_{master}

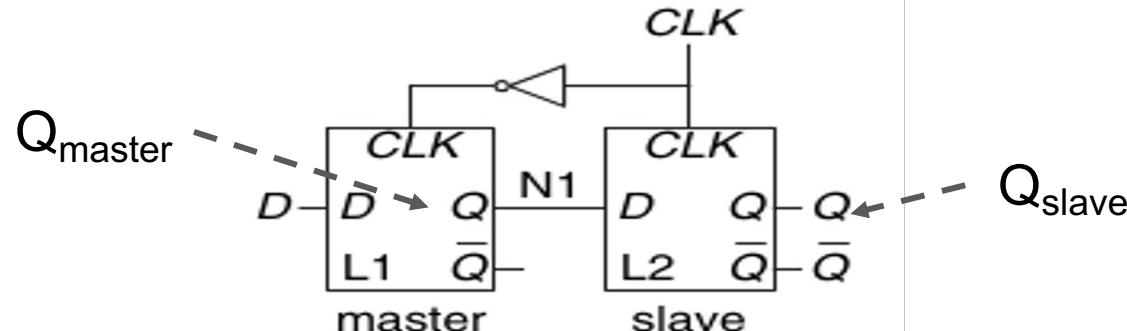
Flip-Flop



Con X indico
un valore non
definito

| | t | t+1 |
|---------------------|-------------|-------------|
| CLK | 0 | 1 |
| master | trasparente | opaco |
| D | 1 | X |
| Q_{master} | 1 | 1 |
| Slave | opaco | trasparente |
| D_{slave} | 1 | 1 |
| Q_{slave} | X | 1 |

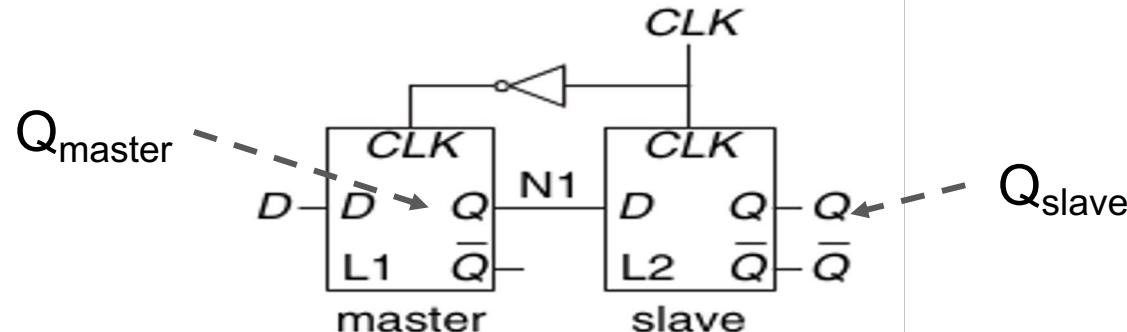
Flip-Flop



| | t | t+1 |
|--------------|-------------|-------------|
| CLK | 0 | 0 |
| master | trasparente | trasparente |
| D | 1 | X |
| Q_{master} | 1 | X |
| Slave | opaco | opaco |
| D_{slave} | 1 | X |
| Q_{slave} | $X=X_{pre}$ | $X=X_{pre}$ |

Con X indico
un valore non
definito

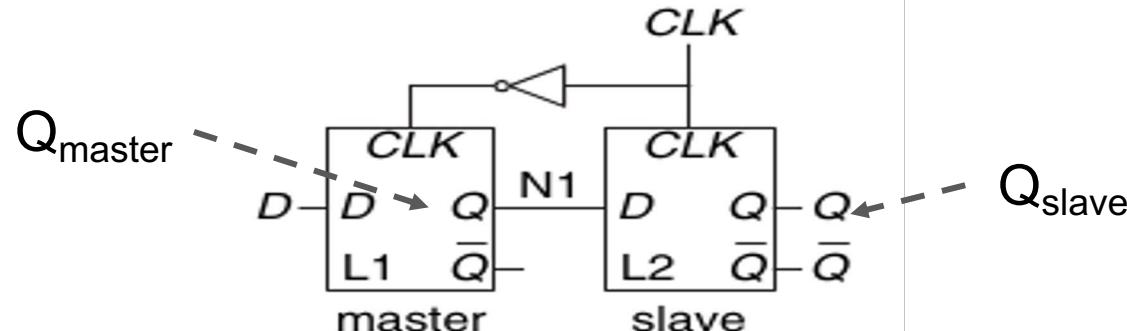
Flip-Flop



| | t | t+1 |
|---------------------|--------------------|--------------------|
| CLK | 1 | 1 |
| master | opaco | opaco |
| D | 1 | X |
| Q_{master} | $X=X_{\text{pre}}$ | $X=X_{\text{pre}}$ |
| Slave | trasparente | trasparente |
| D_{slave} | $X=X_{\text{pre}}$ | $X=X_{\text{pre}}$ |
| Q_{slave} | $X=X_{\text{pre}}$ | $X=X_{\text{pre}}$ |

Con X indico
un valore non
definito

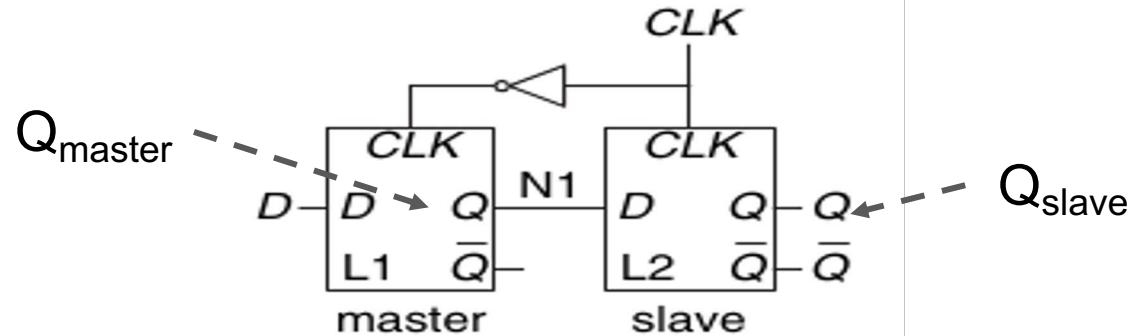
Flip-Flop



| | t | t+1 |
|---------------------|--------------------|--------------------|
| CLK | 1 | 0 |
| master | opaco | trasparente |
| D | 1 | X |
| Q_{master} | $X=X_{\text{pre}}$ | X |
| Slave | trasparente | opaco |
| D_{slave} | $X=X_{\text{pre}}$ | $X=X_{\text{pre}}$ |
| Q_{slave} | $X=X_{\text{pre}}$ | $X=X_{\text{pre}}$ |

Con X indico
un valore non
definito

Flip-Flop

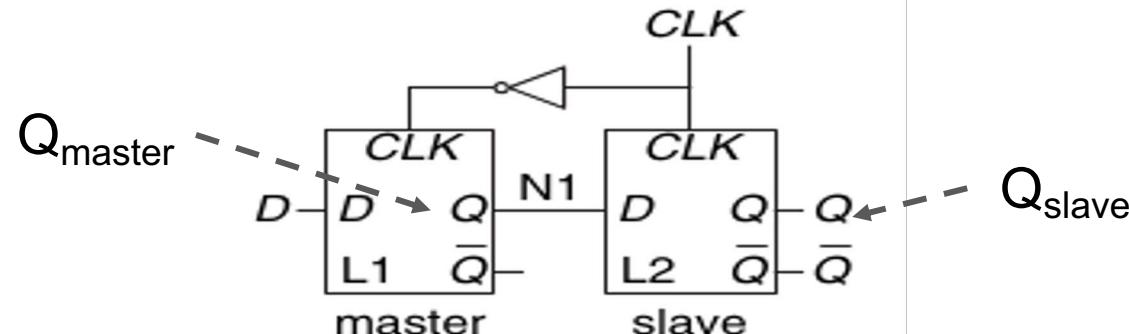


| | t | $t+1$ |
|---------------------|-------------|-------------|
| CLK | 0 | 1 |
| master | trasparente | opaco |
| D | 1 | X |
| Q_{master} | 1 | 1 |
| Slave | opaco | trasparente |
| D_{slave} | 1 | 1 |
| Q_{slave} | X | 1 |

Quindi l'unico caso in cui il valore D (ad esempio 1) passa come valore di Q_{slave} è quando CLK passa da 0 ad 1

Flip-Flop

Questa riga è solo per indicare il tempo che incrementa, ma non c'è realmente un tempo discreto (il tempo è continuo)



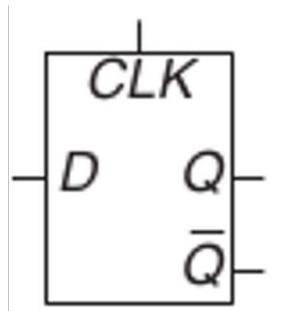
Esempio

| | t | t+1 | t+2 | t+3 | t+4 |
|---------------------|-------------|-------------|-------------|-------------|-------------|
| CLK | 0 | 1 | 1 | 0 | 1 |
| master | trasparente | opaco | opaco | trasparente | opaco |
| D | 1 | 0 | 0 | 0 | 1 |
| Q_{master} | 1 | 1 | 1 | 0 | 0 |
| Slave | opaco | trasparente | trasparente | opaco | trasparente |
| D_{slave} | 1 | 1 | 1 | 0 | 0 |
| Q_{slave} | X | 1 | 1 | 1 | 0 |

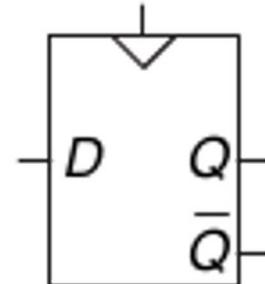
Flip-Flop

Importante:

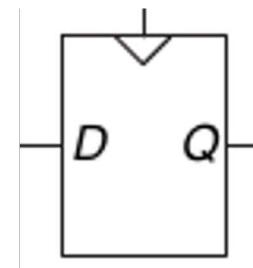
Mentre con il **D-latch** il segnale di controllo (o di clock) ad 1 fa sempre memorizzare il dato D inviato in input, con il **flip-flop** solo il passaggio da 0 ad 1 permette la memorizzazione del dato



Simbolo D-latch



Simbolo flip-flop



Simbolo
semplificato

Flip-Flop

Importante:

Mentre con il D-latch il segnale di controllo ad 1 fa sempre memorizzare il dato D inviato in input,

con il flip-flop solo il passaggio da 0 ad 1 permette la memorizzazione del dato

| CLK | D | \bar{D} | S | R | Q | \bar{Q} |
|-------|-----|-----------|-----|-----|------------|------------------|
| 0 | X | \bar{X} | 0 | 0 | Q_{prev} | \bar{Q}_{prev} |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 |

D-Latch

Flip-Flop

Questa riga è solo per indicare il tempo che incrementa, ma non c'è realmente un tempo discreto (il tempo è continuo)

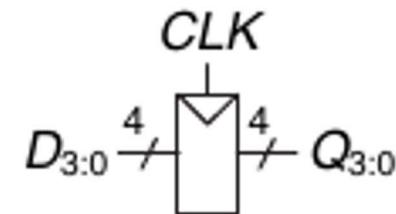
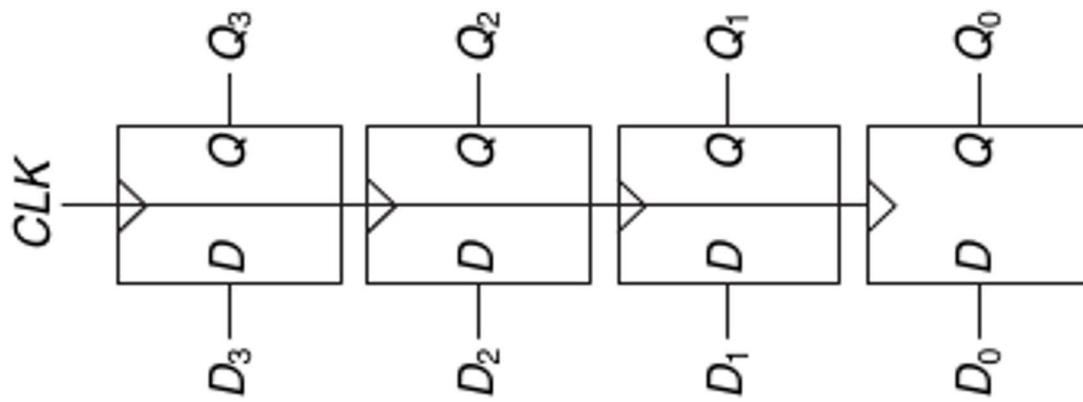
Importante:

Mentre con il D-latch il segnale di controllo ad 1 fa sempre memorizzare il dato D inviato in input, con il flip-flop solo il passaggio da 0 ad 1 permette la memorizzazione del dato,

| | t | t+1 | t+2 | t+3 | t+4 |
|---------------------|-------------|-------------|-------------|-------------|-------------|
| CLK | 0 | 1 | 1 | 0 | 1 |
| master | trasparente | opaco | opaco | trasparente | opaco |
| D | 1 | 0 | 0 | 0 | 1 |
| Q _{master} | 1 | 1 | 1 | 0 | 0 |
| Slave | opaco | trasparente | trasparente | opaco | trasparente |
| D _{slave} | 1 | 1 | 1 | 0 | 0 |
| Q _{slave} | X | 1 | 1 | 1 | 0 |

Registri

- Un **registro** (o registro ad N bit) è un insieme di N flip-flop che condividono lo stesso segnale di controllo o di clock (CLK)
 - Comunemente detto **banco** di N flip-flop

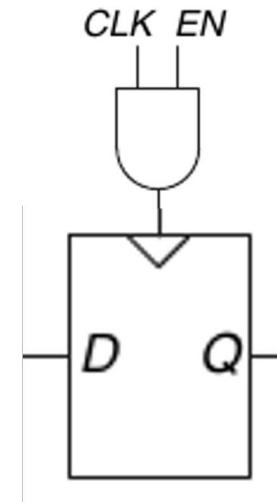
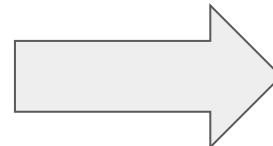
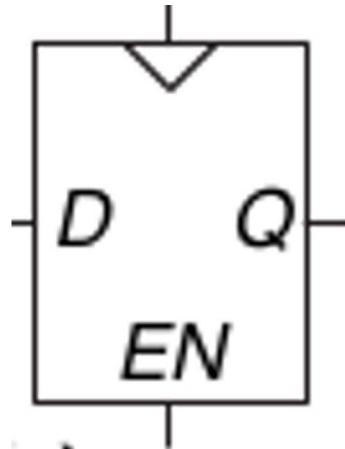


Simbolo usato:

- Nell'esempio per $N=4$

Enabled Flip-Flop

- In questo caso si aggiunge un altro input EN, che ci dice se il flip-flop è abilitato a funzionare oppure no!
 - EN=1 il flip-flop si comporta normalmente
 - EN=0 il flip-flop mantiene il suo stato



Possible
realizzazione

Esercizio 1: SR-latch

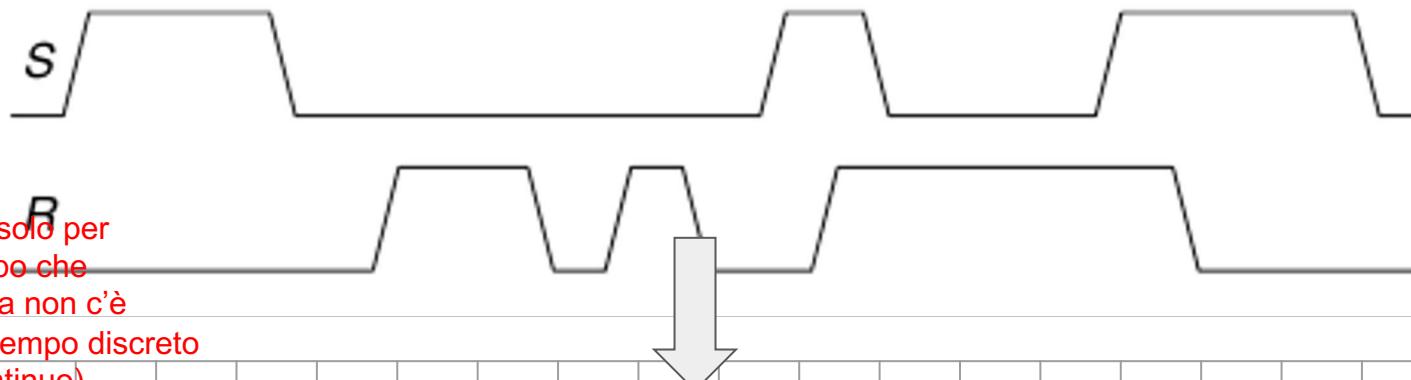
- Calcolare il valore Q per un SR-latch dato il seguente andamento di S e R



| S | R | Q_{n+1} | \bar{Q}_{n+1} | |
|---|---|-----------|-----------------|-------------|
| 0 | 0 | Q_n | \bar{Q}_n | memoria |
| 0 | 1 | 0 | 1 | reset |
| 1 | 0 | 1 | 0 | set |
| I | I | X | X | non ammesso |

Esercizio 1: SR-latch

- Calcolare il valore Q per un SR-latch dato il seguente andamento di S e R



Questa riga è solo per indicare il tempo che incrementa, ma non c'è realmente un tempo discreto (il tempo è continuo)

Esercizio 1: SR-latch

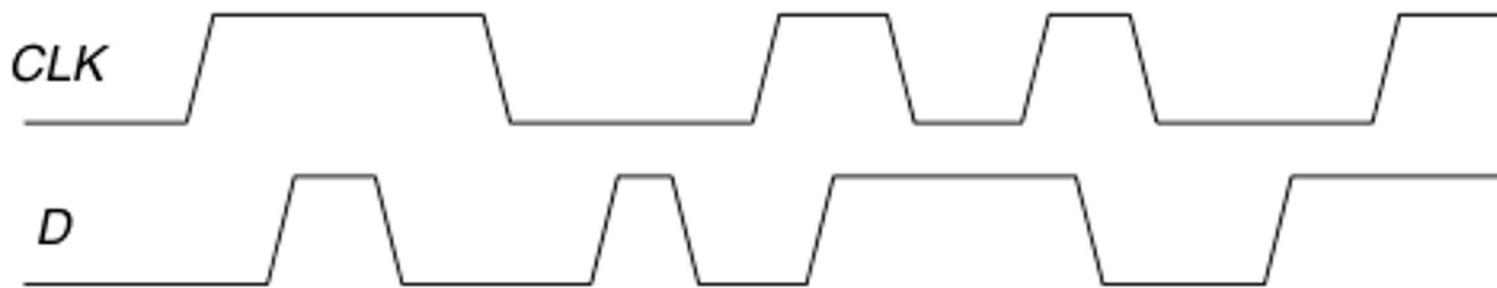
- Calcolare il valore Q per un SR-latch dato il seguente andamento di S e R

| | t | t+1 | t+2 | t+3 | t+4 | t+5 | t+6 | t+7 | t+8 | t+9 | t+10 | t+11 | t+12 | t+13 | t+14 | t+15 | t+16 | t+17 | t+18 |
|---|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|------|------|------|------|------|------|------|------|
| S | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| R | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| Q | | X | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | X | 0 | 0 | 0 | X | 1 | 1 | 1 |

| S | R | Q_{n+1} | \bar{Q}_{n+1} | | |
|---|---|-----------|-----------------|-------------|--|
| 0 | 0 | Q_n | \bar{Q}_n | memoria | |
| 0 | 1 | 0 | 1 | reset | |
| 1 | 0 | 1 | 0 | set | |
| I | I | X | X | non ammesso | |

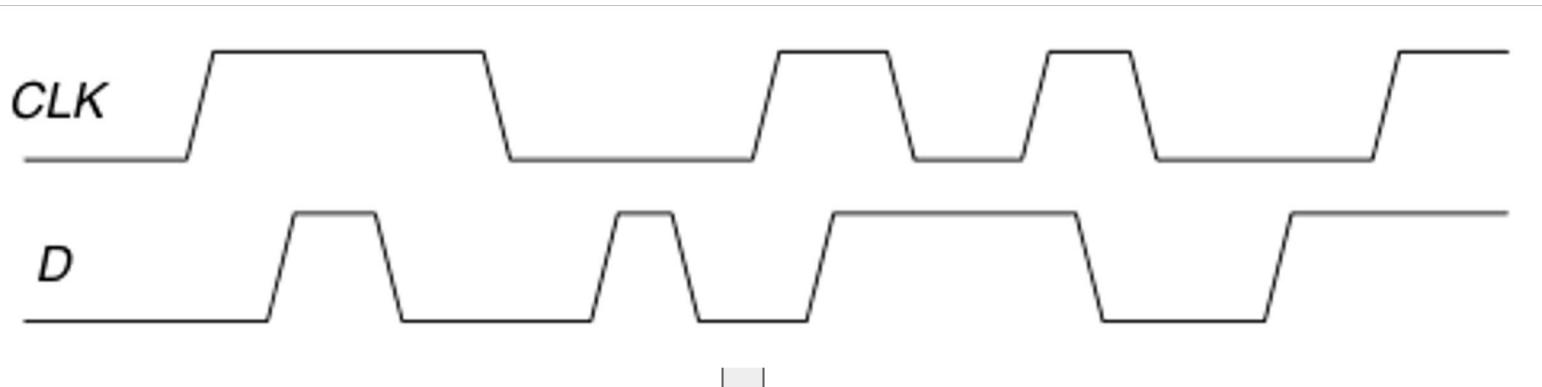
Esercizio 2: D-latch

- Calcolare il valore Q per un D-latch dato il seguente andamento di CLk e D



| <i>CLK</i> | <i>D</i> | \bar{D} | <i>S</i> | <i>R</i> | <i>Q</i> | \bar{Q} |
|------------|----------|-----------|----------|----------|------------|------------------|
| 0 | X | \bar{X} | 0 | 0 | Q_{prev} | \bar{Q}_{prev} |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 |

Esercizio 2: D-latch



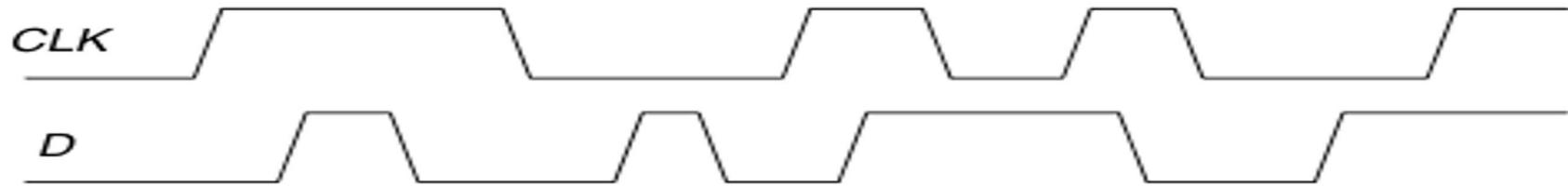
Esercizio 2: D-latch

| | t | t+1 | t+2 | t+3 | t+4 | t+5 | t+6 | t+7 | t+8 | t+9 | t+10 | t+11 | t+12 | t+13 | t+14 | t+15 | t+16 | t+17 | t+18 |
|-----|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|------|------|------|------|------|------|------|------|
| CLK | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| D | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| Q | | X | X | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |

| CLK | D | \bar{D} | S | R | Q | \bar{Q} |
|-----|---|-----------|---|---|------------|------------------|
| 0 | X | X | 0 | 0 | Q_{prev} | \bar{Q}_{prev} |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 |

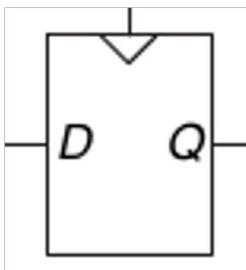
Esercizio 3: flip-flop

- Calcolare il valore Q per un flip-flop con il seguente andamento di CLk e D



Esercizio 3: flip-flop

| | t | t+1 | t+2 | t+3 | t+4 | t+5 | t+6 | t+7 | t+8 | t+9 | t+10 | t+11 | t+12 | t+13 | t+14 | t+15 | t+16 | t+17 | t+18 |
|-----|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|------|------|------|------|------|------|------|------|
| CLK | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| D | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| Q | X | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |

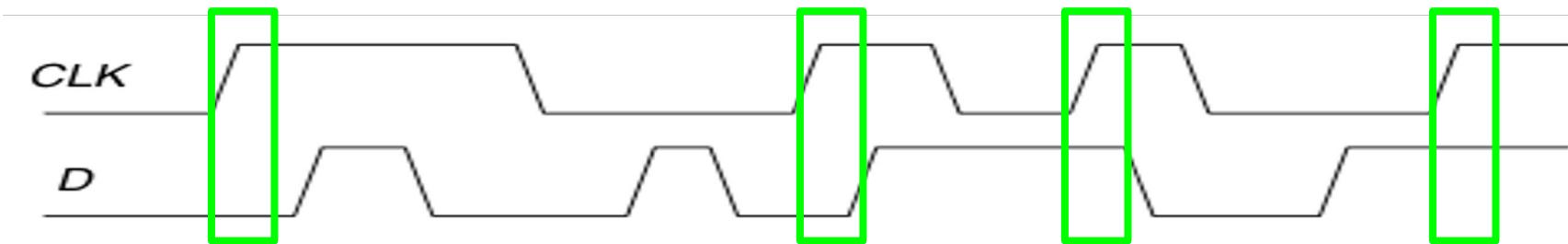


Solo quando CLK passa da 0 ad 1
faccio passare il valore D

Logica sequenziale

OSSERVO (RICORDO):

Solo quando CLK passa da 0 ad 1 faccio passare il valore D

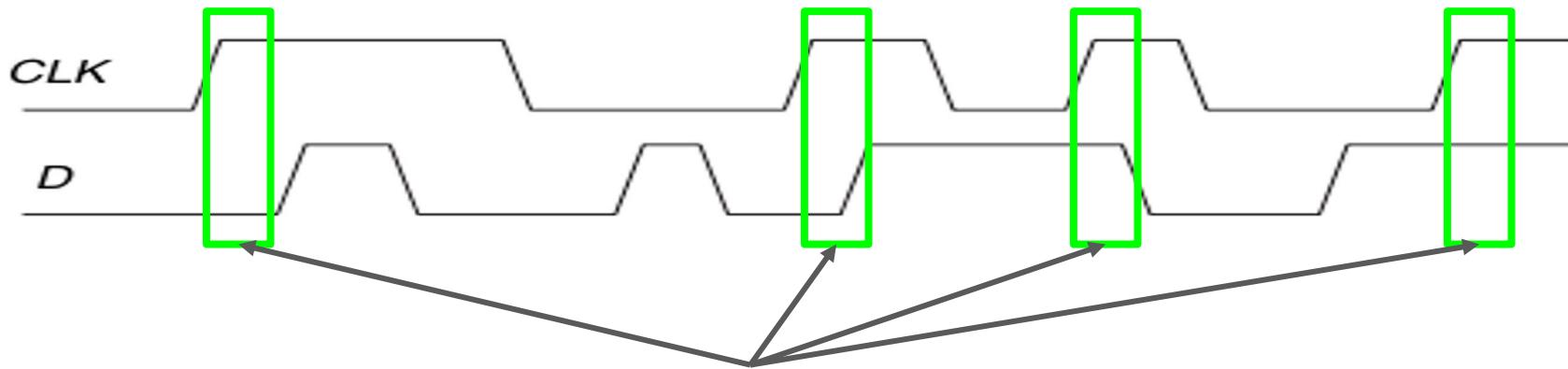


| | t | t+1 | t+2 | t+3 | t+4 | t+5 | t+6 | t+7 | t+8 | t+9 | t+10 | t+11 | t+12 | t+13 | t+14 | t+15 | t+16 | t+17 | t+18 |
|-----|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|------|------|------|------|------|------|------|------|
| CLK | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| D | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| Q | X | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |

Logica sequenziale

OSSERVO (RICORDO):

Solo quando CLK passa da 0 ad 1 faccio passare il valore D



Ho, allora, una sequenza di specifici tempi in cui accadono le transizioni da uno valore all'altro, QUINDI è il CLOCK che rende il tempo discreto, che permette di parlare effettivamente di un tempo t ed un tempo successivo t+1

Logica sequenziale

Circuito sequenziale sincrono:

- Un insieme finito di possibili stati: $\{S_0, S_1, \dots, S_{k-1}\}$
- Un input di clock che specifica a quali tempi avviene una transizione di stato

Logica sequenziale: circuito sincrono

Circuito sequenziale sincrono:

- Un insieme finito di possibili stati: $\{S_0, S_1, \dots, S_{k-1}\}$
- Un input di clock che specifica a quali tempi avviene una transizione di stato

In questo modo posso parlare di:

- **stato corrente** (t)
- **prossimo stato** ($t+1$, quando il segnale di clock sale da 0 ad 1)

Logica sequenziale: circuito sincrono

- **stato corrente** (t)
- **prossimo stato** ($t+1$, quando il segnale di clock sale da 0 ad 1)

La specifica funzionale di un circuito sequenziale sincrono definisce il prossimo stato e gli output del sistema sulla base dello stato corrente e dei suoi input

Circuito sincrono: regole

Un circuito sequenziale è sincrono se consiste di elementi interconnessi tale che:

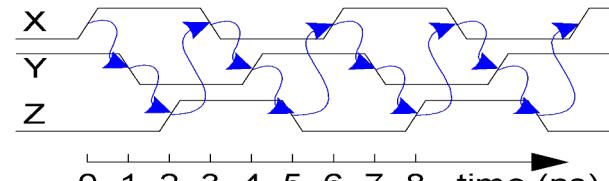
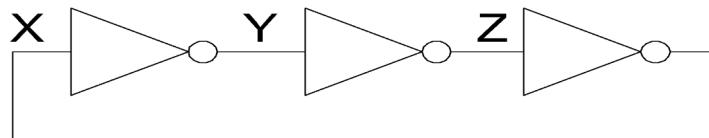
- Ogni elemento del circuito o è un registro o è un circuito combinatorio
- Almeno un elemento è un registro
- Tutti i registri ricevono lo stesso segnale di clock
- Ogni eventuale ciclo contiene almeno 1 registro

Circuito asincroni

Un circuito sequenziale è **asincrono** se non è sincrono:

- Ci focalizzeremo su circuiti sincroni
- I circuiti sincroni sono molto più facili da progettare
- Quasi tutti i circuiti digitali sono sincroni
- I circuiti asincroni presentano delle forti criticità

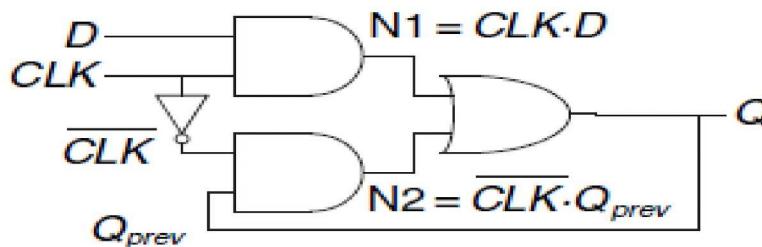
Criticità nella logica sequenziale



- Questi circuiti vengono detti “astabili” poiché hanno un comportamento oscillante
- Il periodo di oscillazione dipende dai ritardi degli inverter
- Idealmente è di 6 ns tuttavia può variare a causa di diversi fattori
 - differenze nella manifattura
 - temperatura
- Circuito *asincrono*: l’output è retroazionato in maniera diretta

Criticità nella logica sequenziale

- In casi più complessi che comprendono l'uso di più porte AND, NOT, OR il comportamento di una rete asincrona può dipendere fortemente dai ritardi accumulati sui singoli cammini

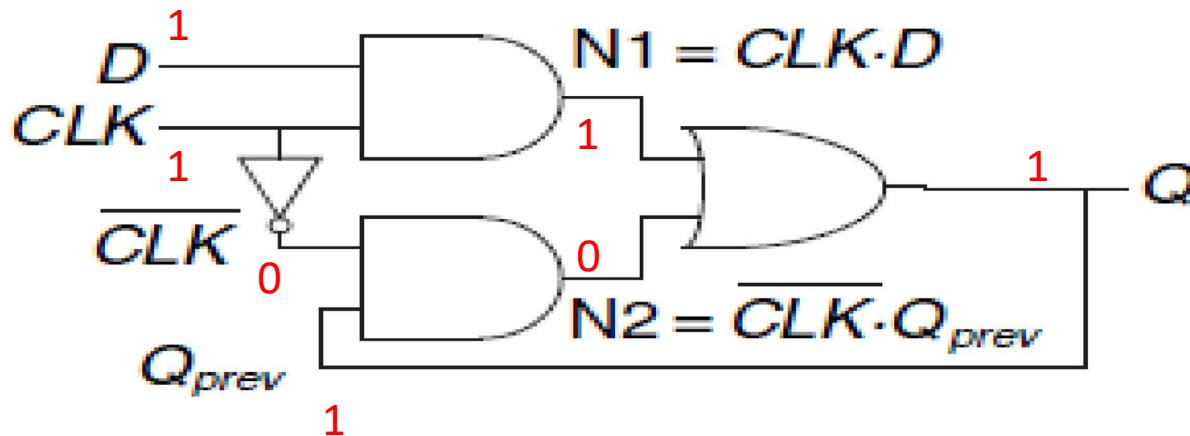


$$Q = CLK \cdot D + \overline{CLK} \cdot Q_{prev}$$

| CLK | D | Q_{prev} | Q |
|-------|-----|------------|-----|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

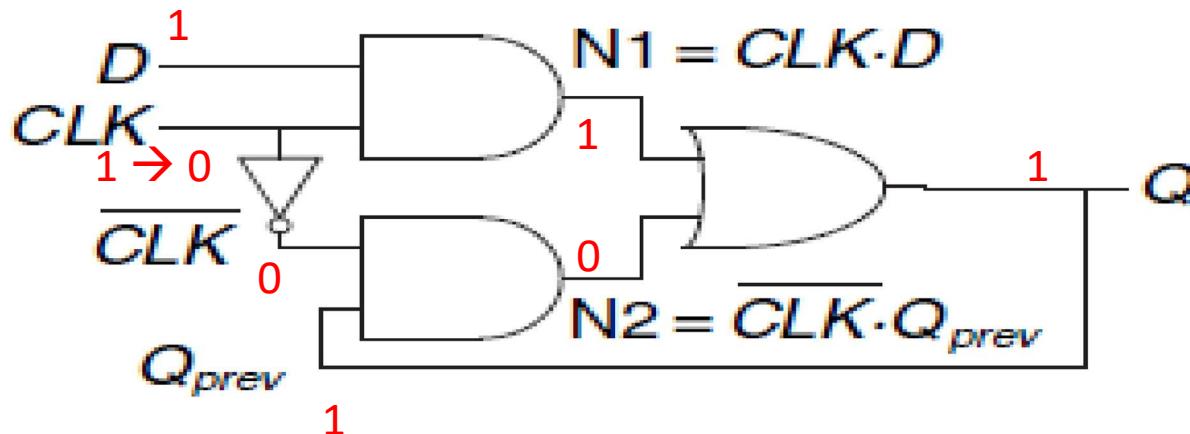
Criticità nella logica sequenziale

- D=1, CLK=1 $\rightarrow Q=1$



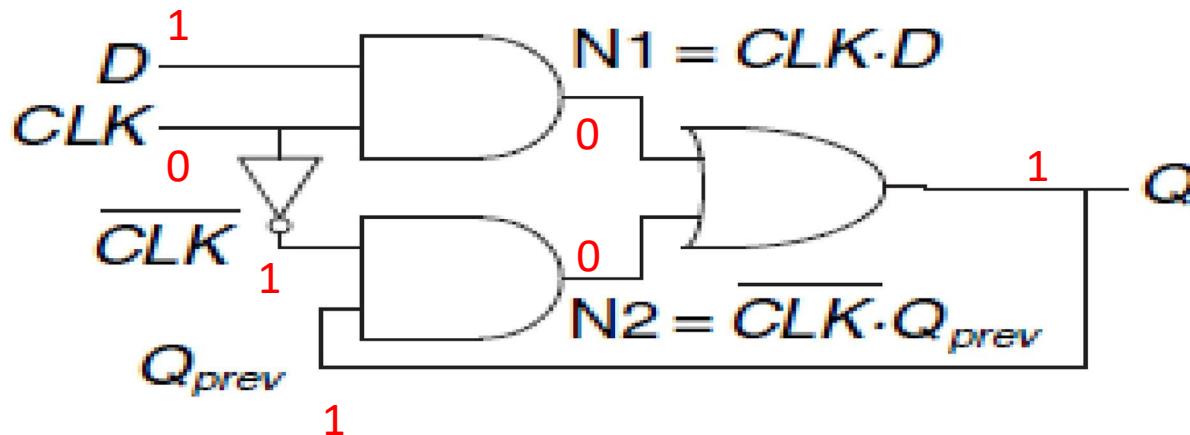
Criticità nella logica sequenziale

t_0 D=1, CLK da 1 a 0



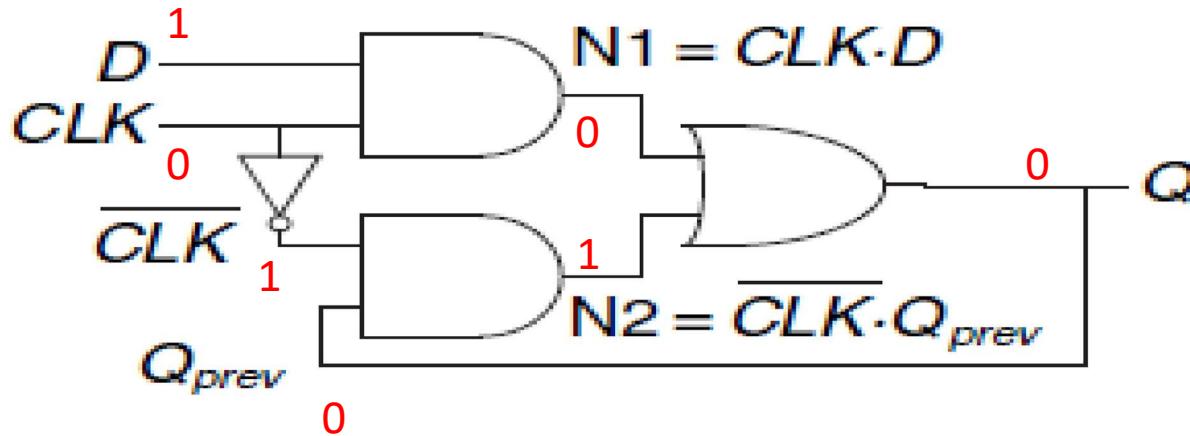
Criticità nella logica sequenziale

$t_1 D=1, CLK = 0$



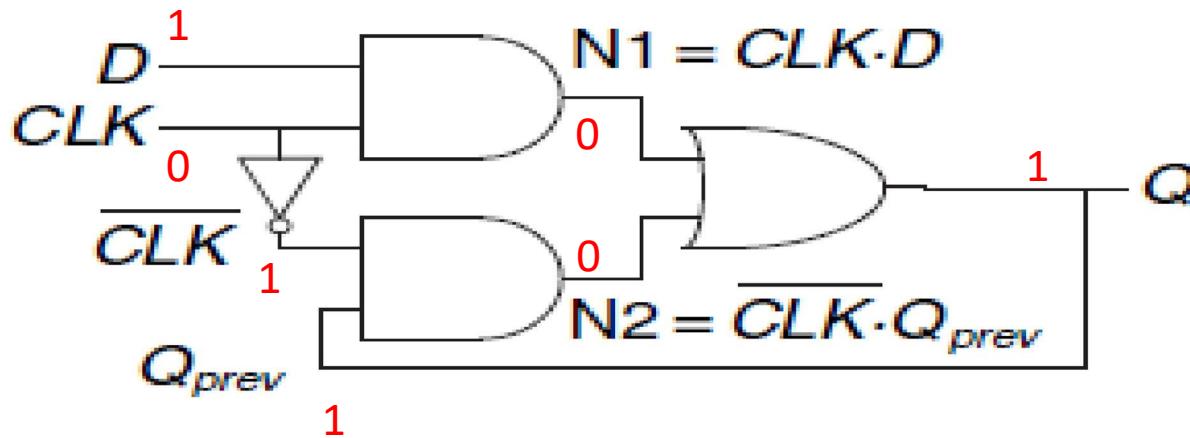
Criticità nella logica sequenziale

t_2 D=1, CLK = 0



Criticità nella logica sequenziale

$t_3 D=1, CLK = 0$



Criticità nella logica sequenziale

- In casi più complessi che comprendono l'uso di più porte AND, NOT, OR il comportamento di una rete asincrona può dipendere fortemente dai ritardi accumulati sui singoli cammini



- $D=1, CLK=1 \rightarrow Q=1$
- $CLK=0 \rightarrow Q$ oscilla $(Q=Q_{prev}=1)$

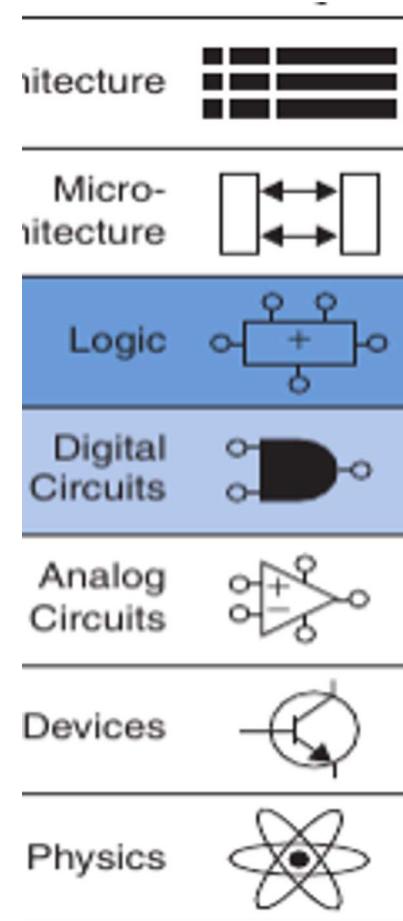
Architettura degli Elaboratori

Lezione 20

Docente: R.Prevete
a.a. 2022/2023
3 maggio 2023

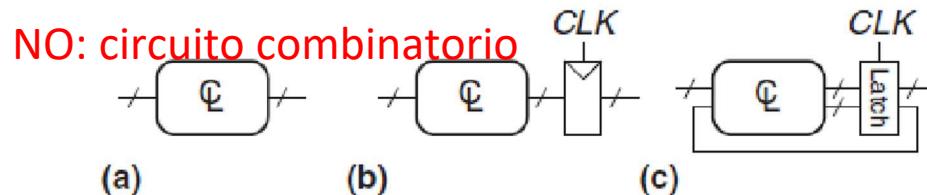
Logica sequenziale

- Di cosa parleremo
 - Macchine a stati finiti

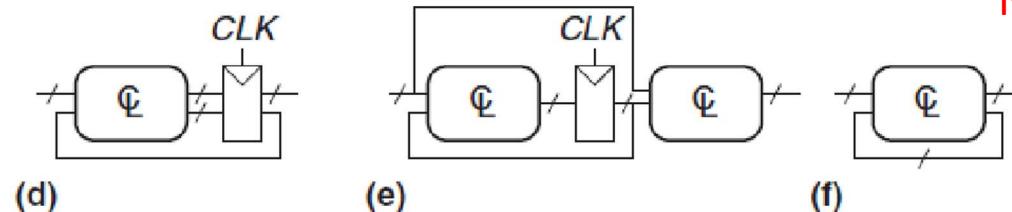


Ricordo cosa è un circuito sequenziale sincrono ('l' significa eventualmente più linee)

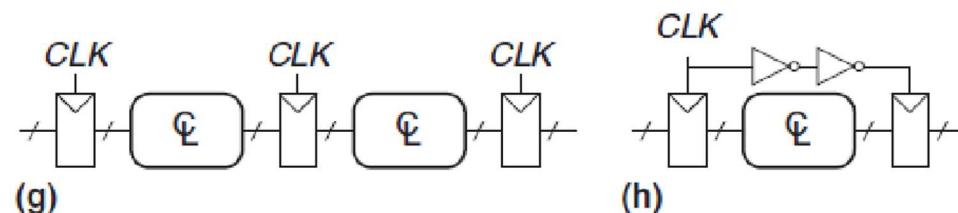
NO: circuito combinatorio



NO: latch e non flip-flop



NO: circuito sequenziale asincrono



NO: i registri non hanno lo stesso clock

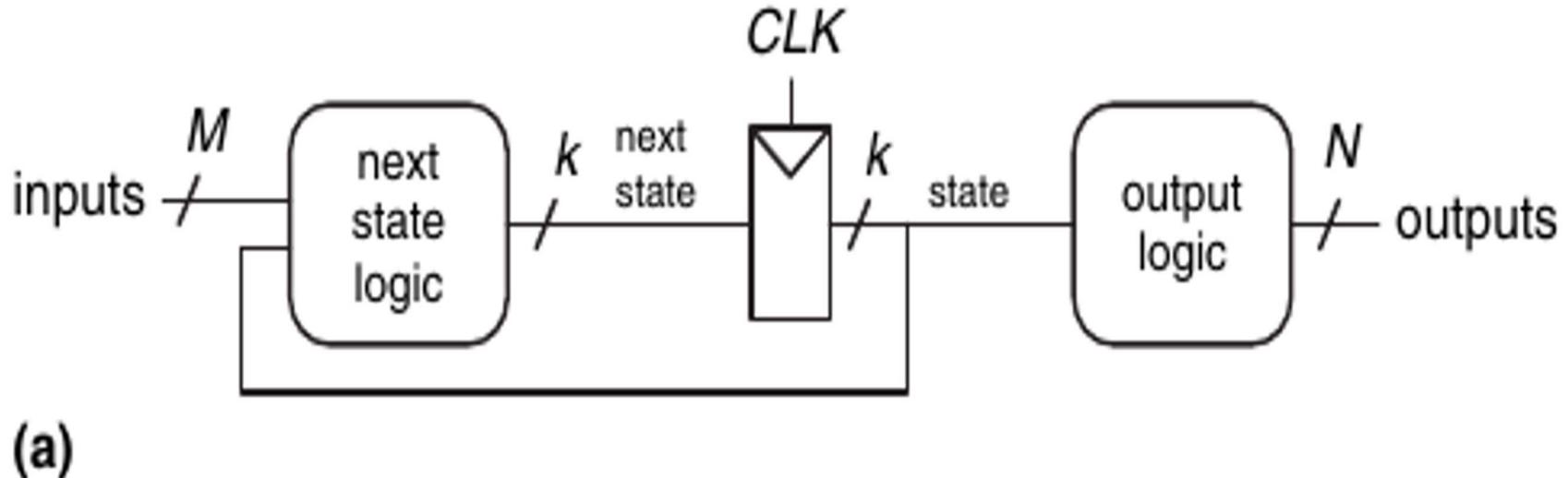
Macchine a stati finiti (Finite State Machine-FSM)

Macchine a stati finiti

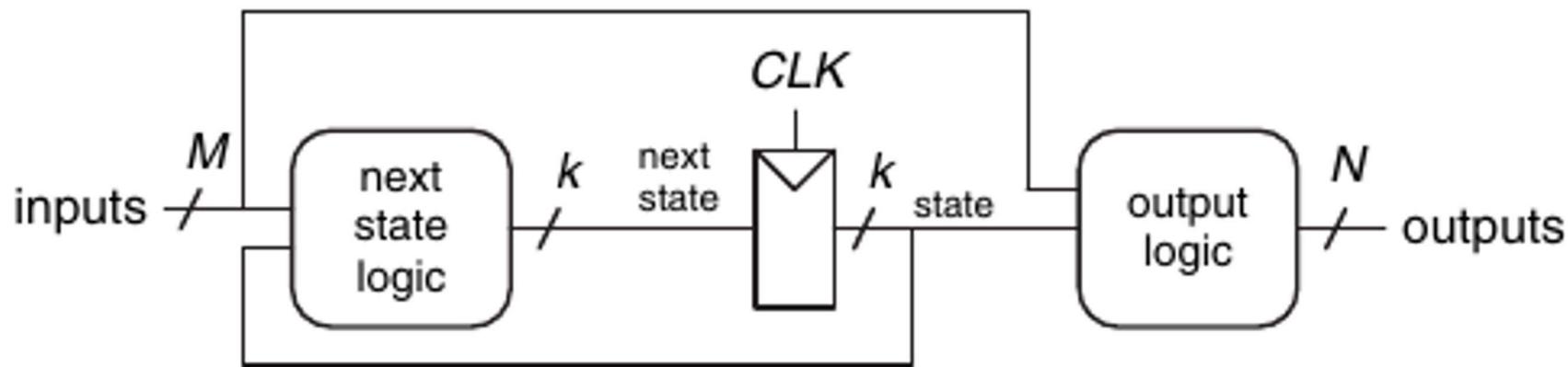
Concetti di base:

- Un registro ad N bit può avere un numero finito di configurazioni (stati), 2^N
- Due tipologie di macchine a stati finiti:
 - L'output dipende solo dallo stato corrente (*macchine di Moore*)
 - L'output dipende dallo stato corrente e dall'input (*macchine di Mealy*)

Macchine a stati finiti: macchine di Moore



Macchine a stati finiti: macchine di Mealy

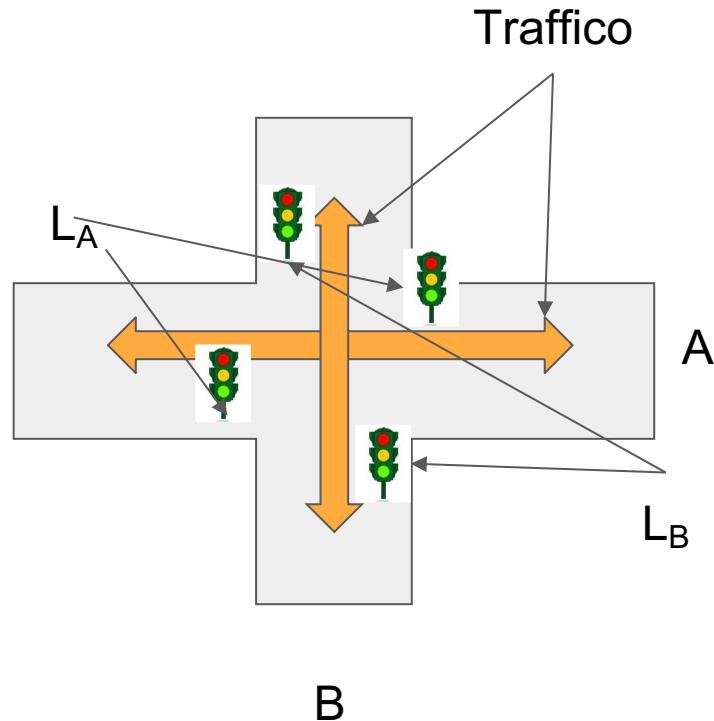


(b)

Macchine a stati finiti: un esempio

Semafori intelligenti

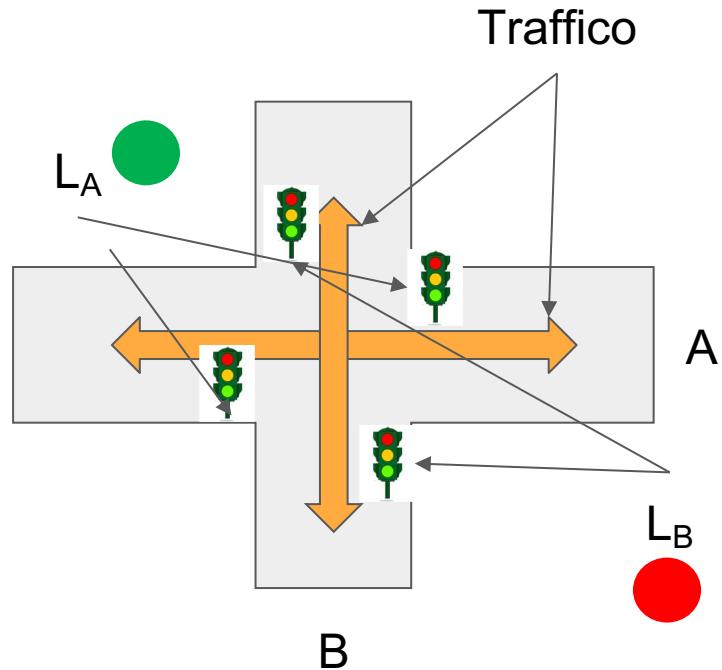
- T_A e T_B sensori traffico (**input del sistema**)
 - $T_A, T_B = 0$ (no traffico), 1 (si traffico)
- L_A, L_B colori semaforo (**stato del sistema**)
 - $L_A, L_B = \text{rosso, giallo, verde}$



Macchine a stati finiti: un esempio

Semafori intelligenti

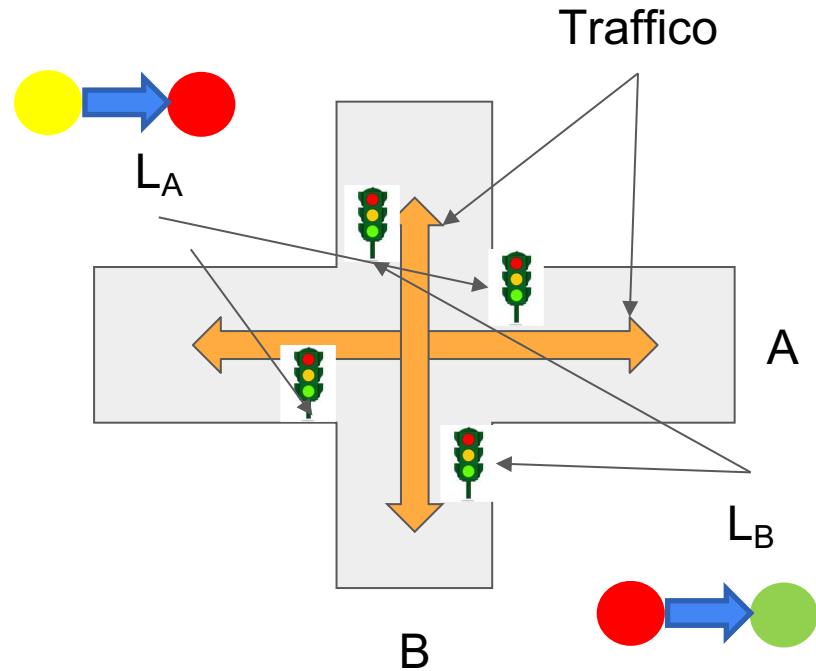
- Configuraione iniziale:
- $L_A = \text{verde}$, $L_B = \text{rosso}$ (situazione, stato **S0**)
- SE $T_A = 1$
 - Resta invariato
 - $L_A = \text{verde}$, $L_B = \text{rosso}$
- ALTRIMENTI ($T_A = 0$)
 - $L_A = \text{giallo}$, $L_B = \text{rosso}$



Macchine a stati finiti: un esempio

Semafori intelligenti

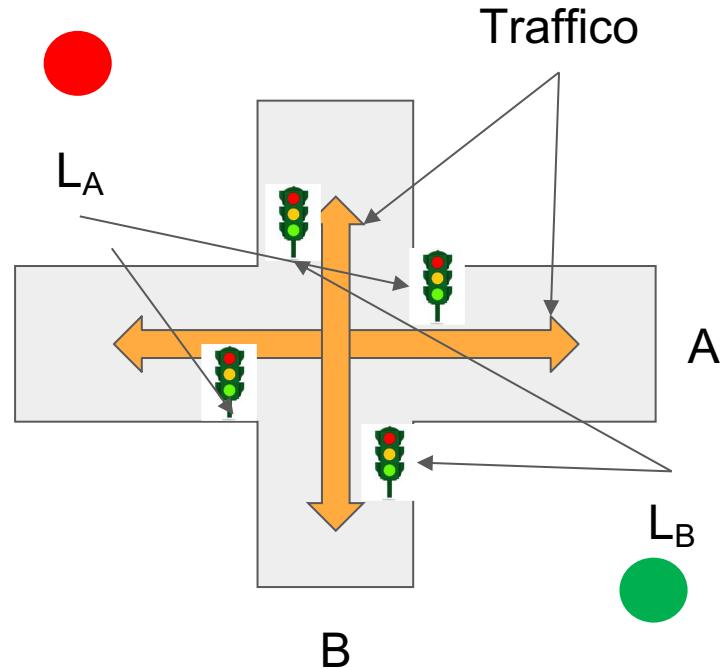
- Possibile prossimo stato:
 - $L_A = \text{giallo}$, $L_B = \text{rosso}$
- (situazione, stato **S1**)
- DOPO UN Tempo DT fissato
 - $L_A = \text{rosso}$, $L_B = \text{verde}$



Macchine a stati finiti: un esempio

Semafori intelligenti

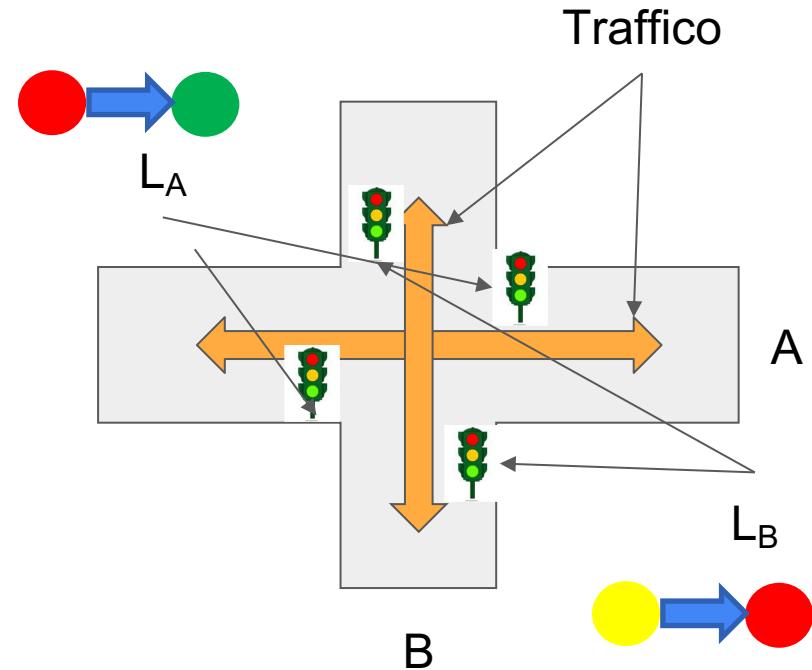
- $L_A = \text{rosso}$, $L_B = \text{verde}$
(situazione, stato **S2**)
- SE $T_B = 1$
 - Resta invariato $L_A = \text{rosso}$,
 $L_B = \text{verde}$
- ALTRIMENTI
 - $L_A = \text{rosso}$, $L_B = \text{giallo}$



Macchine a stati finiti: un esempio

Semafori intelligenti

- L_A = rosso, L_B = giallo (situazione, stato **S3**)
- DOPO UN Tempo DT fissato (in questo esempio 5s)
 - L_A = verde, L_B = rosso



Macchine a stati finiti: un esempio

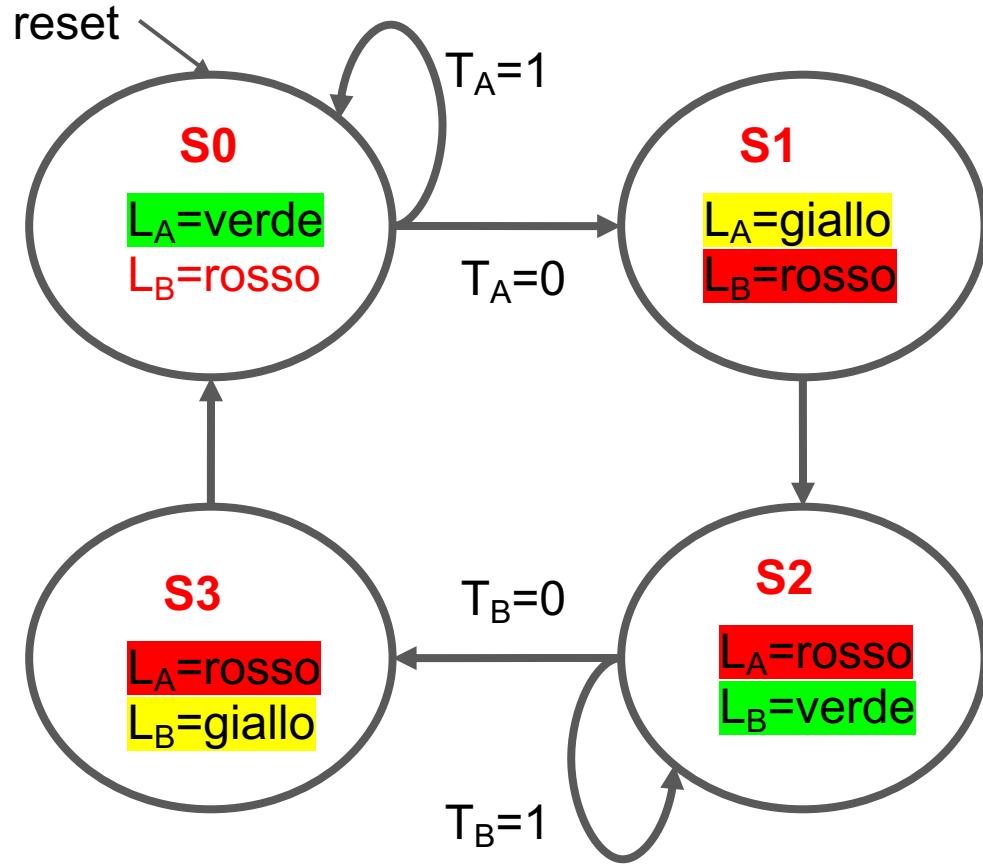
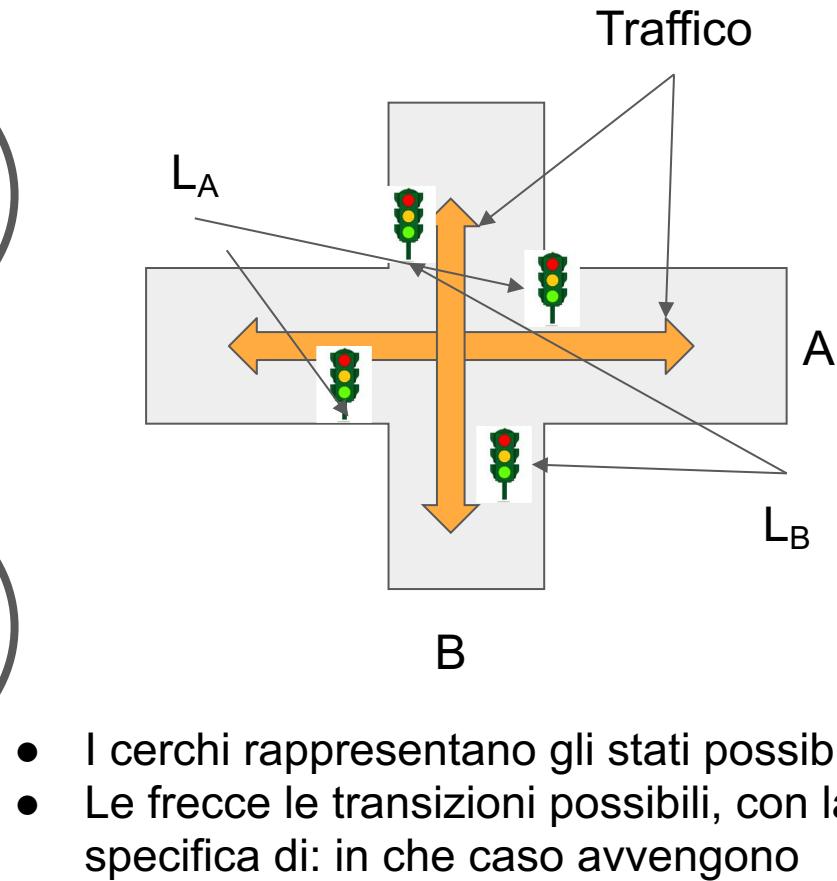
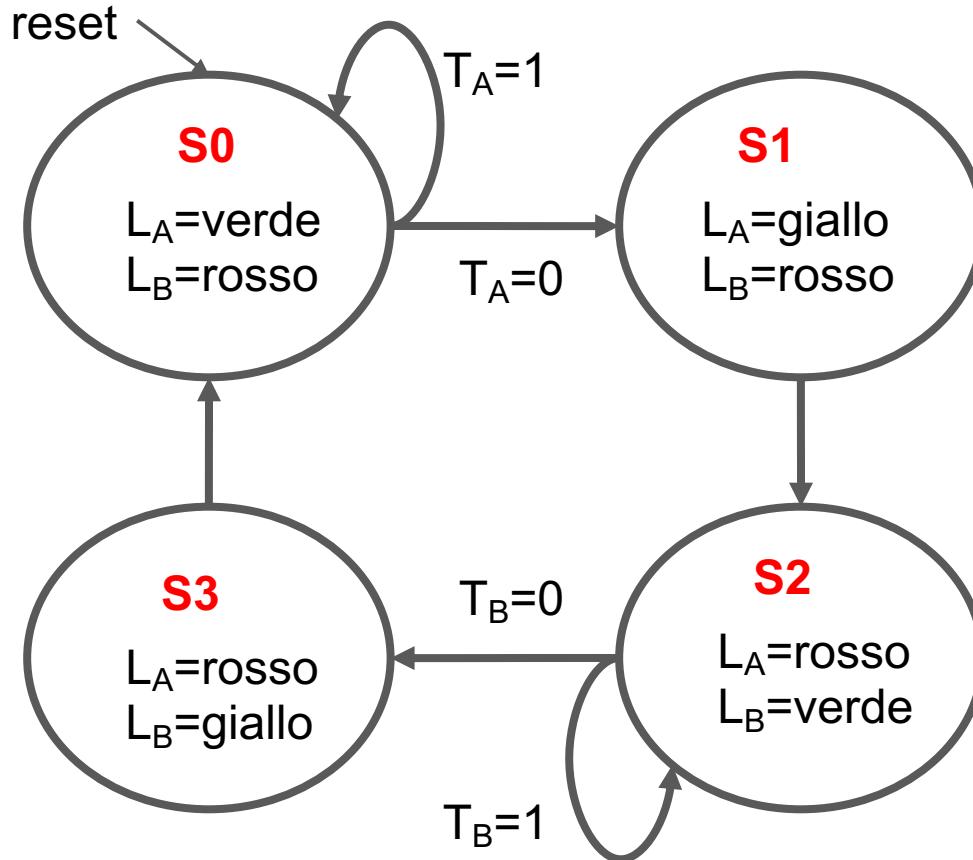


Diagramma delle transizioni



Macchine a stati finiti: un esempio



| Current State S | Inputs | | Next State S' |
|-------------------|--------|-------|-----------------|
| | T_A | T_B | |
| S0 | 0 | X | S1 |
| S0 | 1 | X | S0 |
| S1 | X | X | S2 |
| S2 | X | 0 | S3 |
| S2 | X | 1 | S2 |
| S3 | X | X | S0 |

Tabella delle transizioni
(ricordo che X significa “Don’t care”, non importa)

Macchine a stati finiti: un esempio

Osservazione:

- Abbiamo usato dei “simboli” per gli stati del sistema (S_0, \dots, S_3) e per gli output (rosso, verde, giallo)
- Per costruire un circuito reale abbiamo bisogno di passare a 0,
1

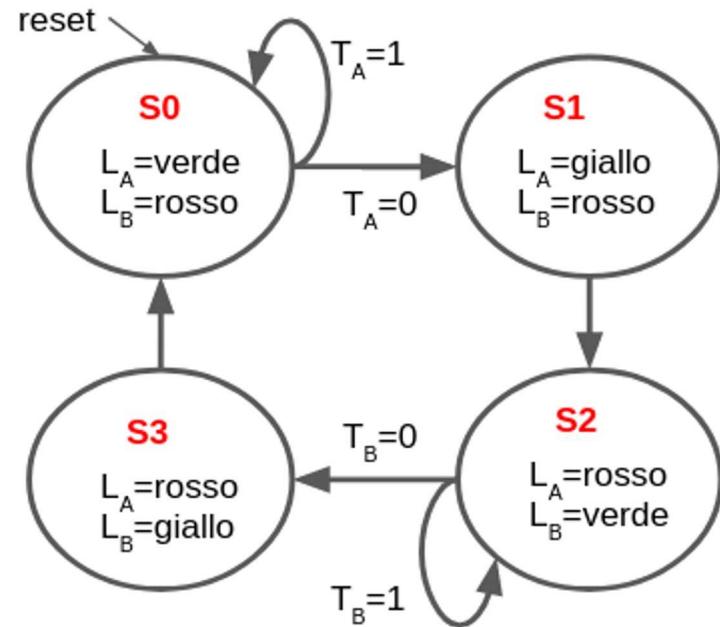
Macchine a stati finiti: un esempio

| State | Encoding $S_{1:0}$ |
|-------|--------------------|
| S0 | 00 |
| S1 | 01 |
| S2 | 10 |
| S3 | 11 |

| Output | Encoding $L_{1:0}$ |
|--------|--------------------|
| green | 00 |
| yellow | 01 |
| red | 10 |

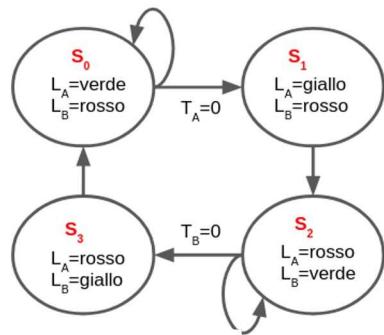
Due bit :
 S_0 e S_1
codifica binaria
degli stati

Due bit :
 L_0 e L_1
codifica binaria
del output



Macchine a stati finiti: un esempio

| Current State | | Inputs | | Next State | |
|---------------|-------|--------|-------|------------|--------|
| S_1 | S_0 | T_A | T_B | S'_1 | S'_0 |
| 0 | 0 | 0 | X | 0 | 1 |
| 0 | 0 | 1 | X | 0 | 0 |
| 0 | 1 | X | X | 1 | 0 |
| 1 | 0 | X | 0 | 1 | 1 |
| 1 | 0 | X | 1 | 1 | 0 |
| 1 | 1 | X | X | 0 | 0 |



| State | Encoding $S_{1:0}$ |
|-------|--------------------|
| S0 | 00 |
| S1 | 01 |
| S2 | 10 |
| S3 | 11 |

| Output | Encoding $L_{1:0}$ |
|--------|--------------------|
| green | 00 |
| yellow | 01 |
| red | 10 |

Tabella delle transizioni con codifica binaria

Macchine a stati finiti: un esempio

Possiamo esprimere la tabella di transizione in termini di **tabella di verità**:

- Lo **stato successivo** diventa l'output della tabella di verità

Input

| Input | | | | output | |
|-------|-------|-------|-------|--------------|--------------|
| S_1 | S_0 | T_A | T_B | S^{next}_1 | S^{next}_0 |
| 0 | 0 | 0 | X | 0 | 1 |
| 0 | 0 | 1 | X | 0 | 0 |
| 0 | 1 | X | X | 1 | 0 |
| 1 | 0 | X | 0 | 1 | 1 |
| 1 | 0 | X | 1 | 1 | 0 |
| 1 | 1 | X | X | 0 | 0 |

| Current State S_1 | S_0 | Inputs T_A | T_B | Next State S'_1 | S'_0 |
|------------------------|-------|-----------------|-------|----------------------|--------|
| 0 | 0 | 0 | X | 0 | 1 |
| 0 | 0 | 1 | X | 0 | 0 |
| 0 | 1 | X | X | 1 | 0 |
| 1 | 0 | X | 0 | 1 | 1 |
| 1 | 0 | X | 1 | 1 | 0 |
| 1 | 1 | X | X | 0 | 0 |

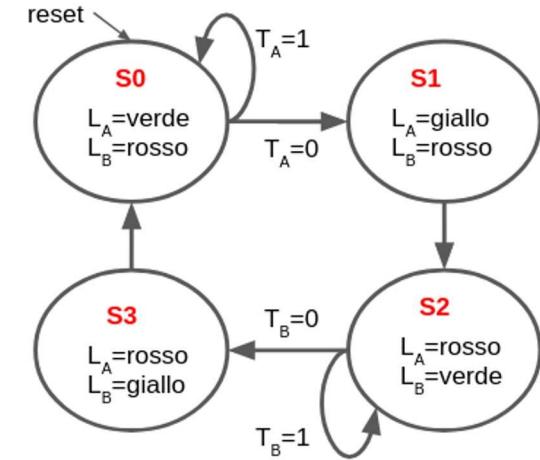
Tabella di verità

Macchine a stati finiti: un esempio

Anche per l'output possiamo scrivere la corrispondente **tabella di verità**:

| Input | | output | | | |
|-------|-------|----------|----------|----------|----------|
| S_1 | S_0 | L_{A1} | L_{A0} | L_{B1} | L_{B0} |
| 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 1 |

Tabella di verità



| State | Encoding $S_{1:0}$ | Output | Encoding $L_{1:0}$ |
|-------|--------------------|--------|--------------------|
| S0 | 00 | green | 00 |
| S1 | 01 | yellow | 01 |
| S2 | 10 | | |
| S3 | 11 | red | 10 |

Macchine a stati finiti: un esempio

- Espressione Booleana per lo stato successivo

Input

output

| S_1 | S_0 | T_A | T_B | S^{next}_1 | S^{next}_0 |
|-------|-------|-------|-------|--------------|--------------|
| 0 | 0 | 0 | X | 0 | 1 |
| 0 | 0 | 1 | X | 0 | 0 |
| 0 | 1 | X | X | 1 | 0 |
| 1 | 0 | X | 0 | 1 | 1 |
| 1 | 0 | X | 1 | 1 | 0 |
| 1 | 1 | X | X | 0 | 0 |

$$S^{next}_1 = \bar{S}_1 S_0 + S_1 \bar{S}_0 \bar{T}_B + S_1 \bar{S}_0 T_B$$

Possiamo anche usare mappe di Karnaugh



$$S^{next}_1 = \bar{S}_1 S_0 + S_1 \bar{S}_0$$

$$S^{next}_1 = S_1 \oplus S_0$$

Macchine a stati finiti: un esempio

- Espressione Booleana per lo stato successivo

| Input | | | | output | |
|-------|-------|-------|-------|--------------|--------------|
| S_1 | S_0 | T_A | T_B | S^{next}_1 | S^{next}_0 |
| 0 | 0 | 0 | X | 0 | 1 |
| 0 | 0 | 1 | X | 0 | 0 |
| 0 | 1 | X | X | 1 | 0 |
| 1 | 0 | X | 0 | 1 | 1 |
| 1 | 0 | X | 1 | 1 | 0 |
| 1 | 1 | X | X | 0 | 0 |

Possiamo anche usare mappe di Karnaugh

$$S^{next}_0 = \bar{S}_1 \bar{S}_0 \bar{T}_A + S_1 \bar{S}_0 \bar{T}_B$$

Macchine a stati finiti: un esempio

- Espressione Booleana per lo stato successivo

| Input | | | | output | |
|-------|-------|-------|-------|--------------|--------------|
| S_1 | S_0 | T_A | T_B | S^{next}_1 | S^{next}_0 |
| 0 | 0 | 0 | X | 0 | 1 |
| 0 | 0 | 1 | X | 0 | 0 |
| 0 | 1 | X | X | 1 | 0 |
| 1 | 0 | X | 0 | 1 | 1 |
| 1 | 0 | X | 1 | 1 | 0 |
| 1 | 1 | X | X | 0 | 0 |

$$S^{next}_1 = S_1 \oplus S_0$$

$$S^{next}_0 = \bar{S}_1 \bar{S}_0 \bar{T}_A + S_1 \bar{S}_0 \bar{T}_B$$

Macchine a stati finiti: un esempio

- Espressione Booleana per definire l'output

| Current State | | Outputs | | | |
|---------------|-------|----------|----------|----------|----------|
| S_1 | S_0 | L_{A1} | L_{A0} | L_{B1} | L_{B0} |
| 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 1 |

| Output | Encoding |
|--------|----------|
| green | 00 |
| yellow | 01 |
| red | 10 |

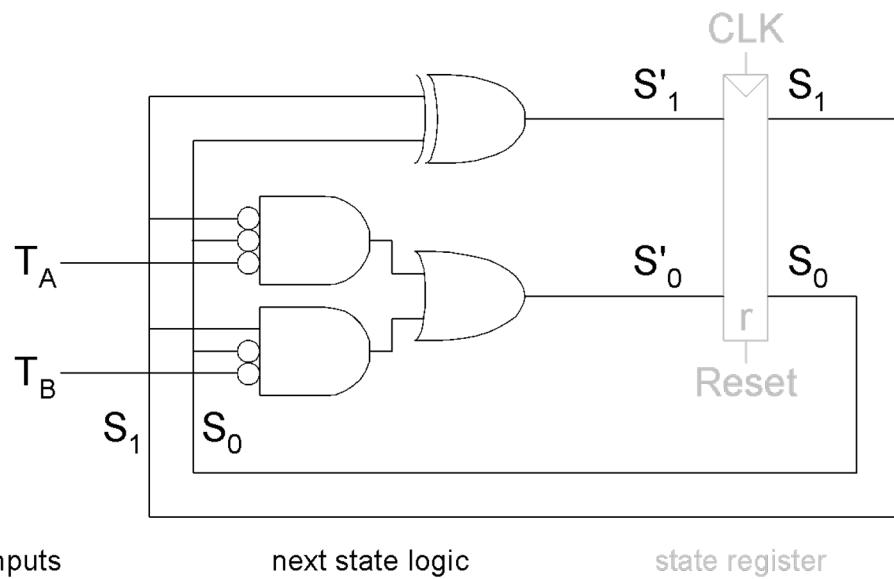
$$L_{A1} = S_1$$

$$L_{A0} = \bar{S}_1 S_0$$

$$L_{B1} = \bar{S}_1$$

$$L_{B0} = S_1 S_0$$

Schema della logica di transizione

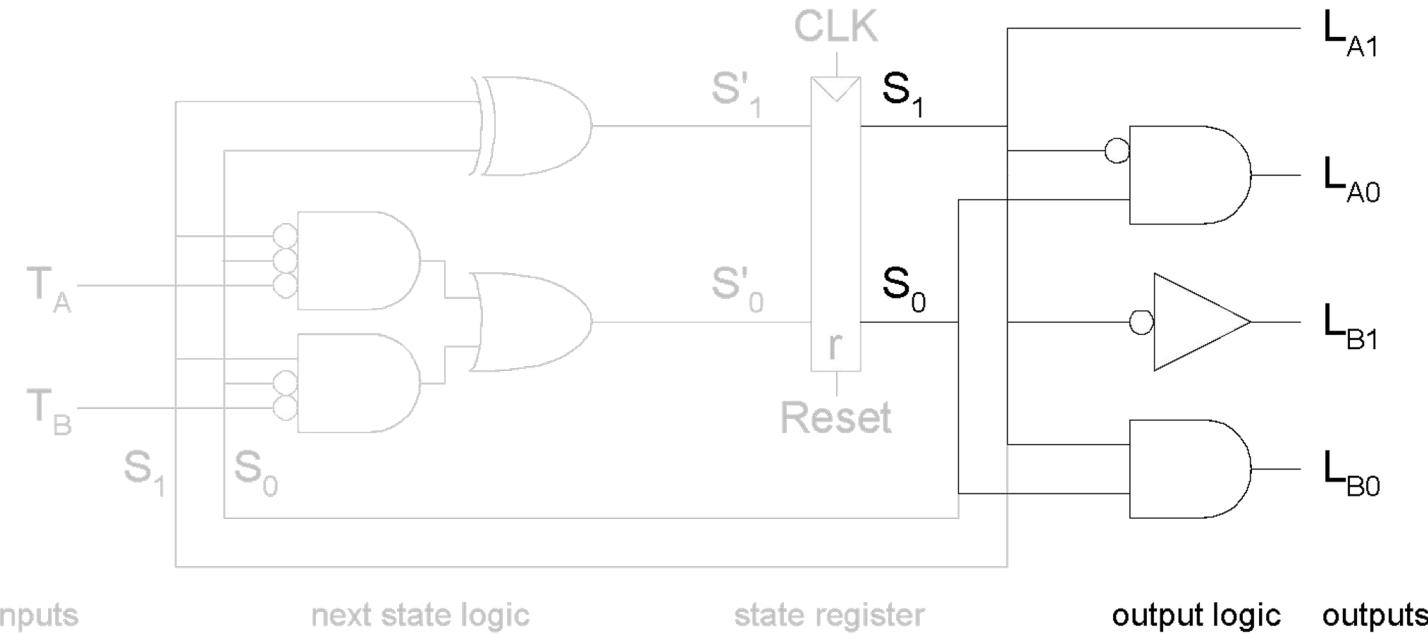


$$S'^{next}_1 = S_1 \oplus S_0$$

$$S'^{next}_0 = \overline{S}_1 \overline{S}_0 \overline{T}_A + S_1 \overline{S}_0 \overline{T}_B$$

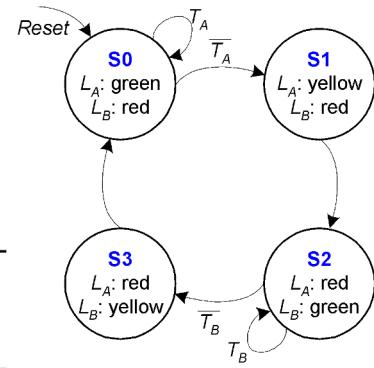
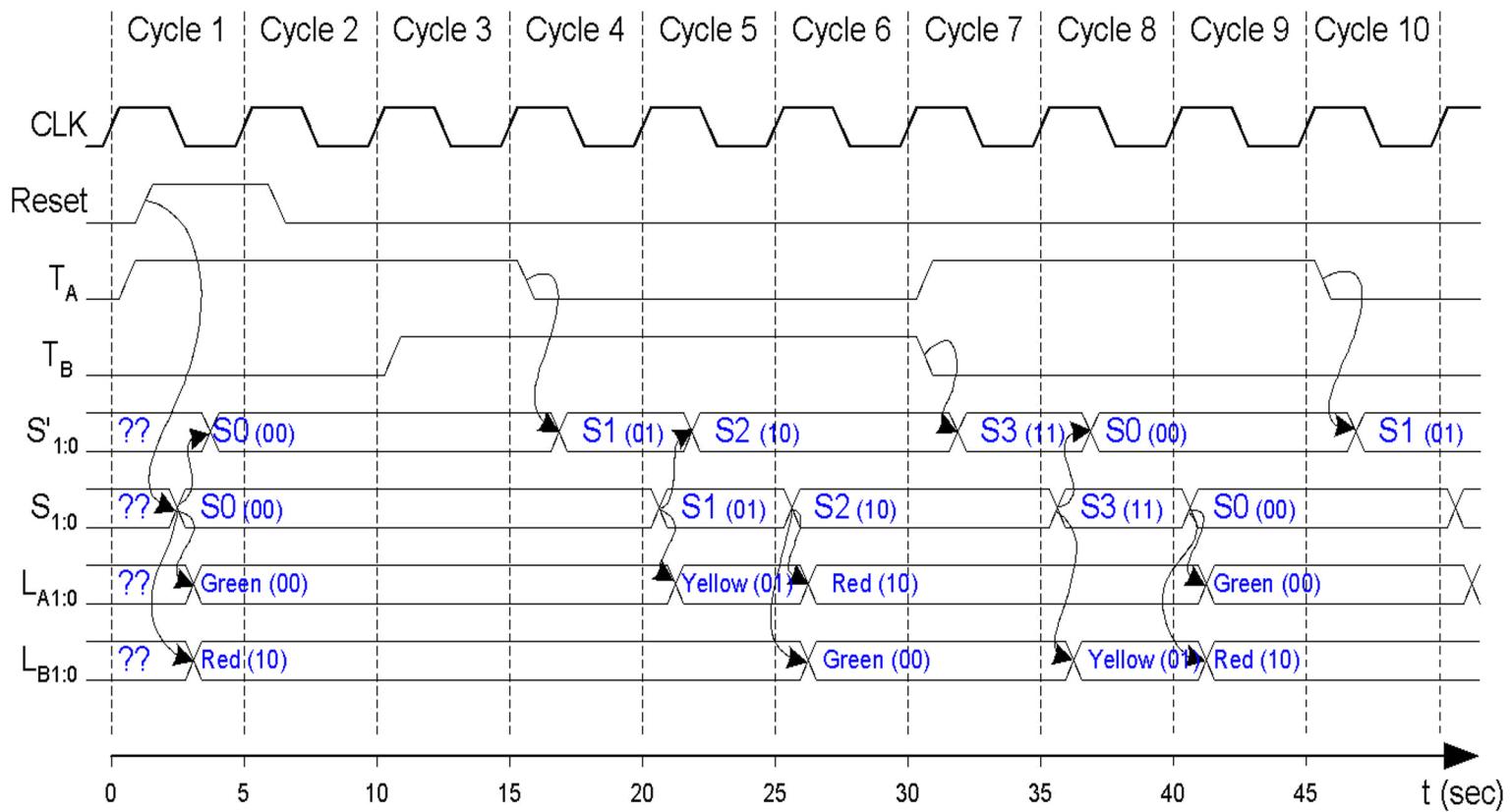
Nota che nel diagramma il simbolo S' è usato al posto di S^{next}

Schema della logica di output



$$\begin{aligned}L_{A1} &= S_1 \\L_{A0} &= \bar{S}_1 S_0 \\L_{B1} &= \bar{S}_1 \\L_{B0} &= S_1 S_0\end{aligned}$$

FSM Timing Diagram

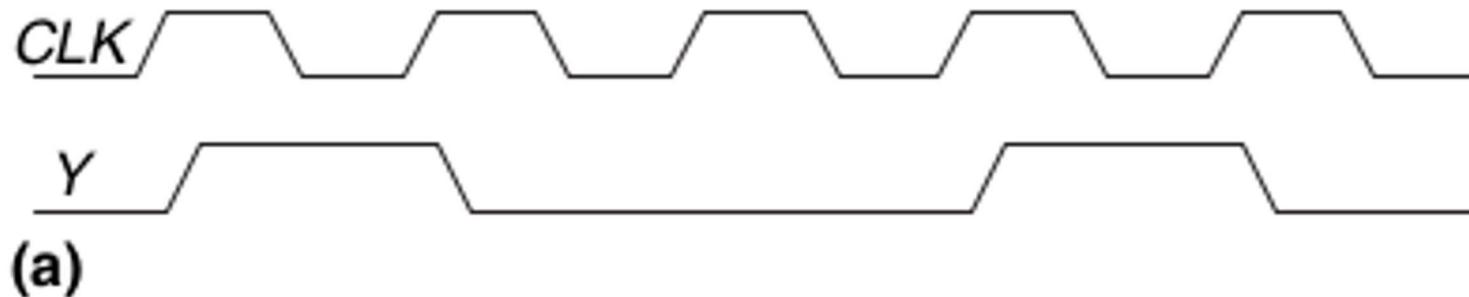


Encoding degli stati: due possibili soluzioni

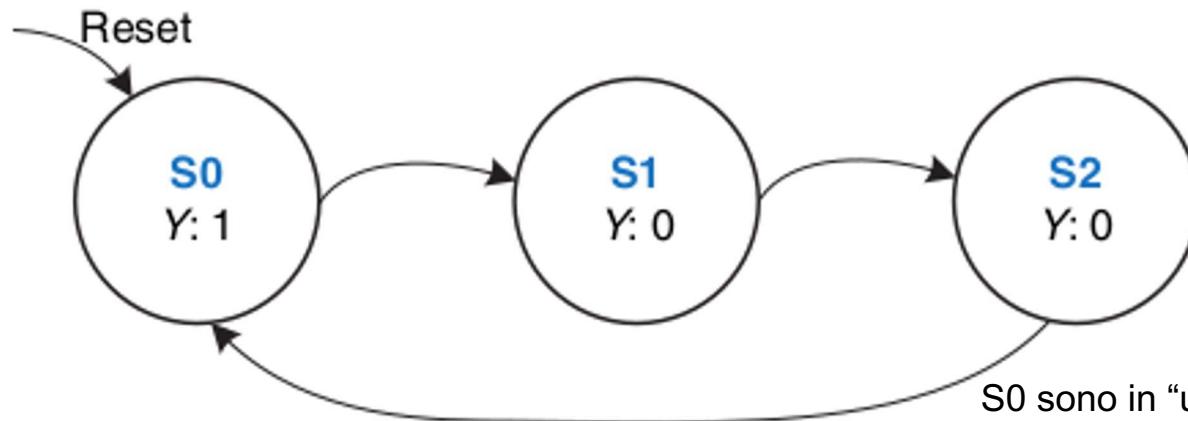
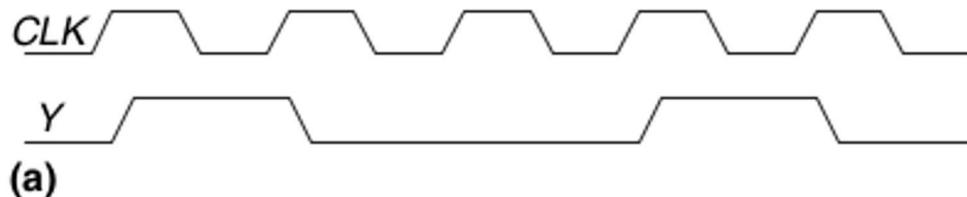
- Encoding binario:
 - i.e., per 4 stati, 00, 01, 10, 11
- Encoding *one-hot*
 - Un bit per stato
 - Solo un bit HIGH alla volta
 - i.e., per 4 stati, 0001, 0010, 0100, 1000
 - Richiede più flip-flops
 - Spesso la logica combinatoria associata è più semplice

Scelta della codifica: un esempio

- **divide-by-N counter**
 - Ha un output e nessun input.
 - L'output Y è alto per ogni N cicli
 - In altre parole l'output divide la frequenza di clock per N



divide-by-N counter: un esempio ($N=2$)



(b)

Diagramma delle transizioni

S0 sono in “un valore alto”, quando arriva il clock
devo andare in uno “valore basso”
S1 sono in uno valore basso , ma “devo aspettare”
S2 sono in uno valore basso, il “prossimo sarà alto”

divide-by-N counter: un esempio

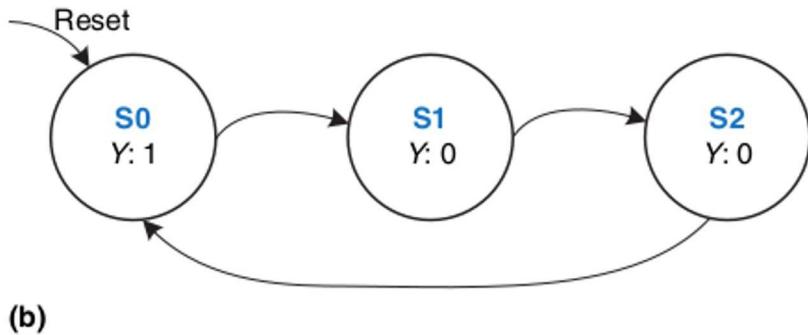


Tabella delle transizioni

| Stato corrente | Prossimo stato |
|----------------|----------------|
| S0 | S1 |
| S1 | S2 |
| S2 | S0 |

Tabella output

| Stato corrente | Y |
|----------------|---|
| S0 | 1 |
| S1 | 0 |
| S2 | 0 |

divide-by-N counter: codifica binaria

Tabella delle transizioni

| Stato corrente | Prossimo stato |
|----------------|----------------|
| S0 | S1 |
| S1 | S2 |
| S2 | S0 |

Tabella output

| Stato corrente | Y |
|----------------|---|
| S0 | 1 |
| S1 | 0 |
| S2 | 0 |



| Stato corrente | Y |
|----------------|----------------|
| S ₁ | S ₀ |
| 0 | 0 |
| 0 | 1 |
| 1 | 0 |

↓

| Stato corrente | | | |
|----------------|----------------|-----------------|-----------------|
| S ₁ | S ₀ | S' ₁ | S' ₀ |
| 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |

| State | One-Hot Encoding | | | Binary Encoding | |
|-------|------------------|----------------|----------------|-----------------|----------------|
| | S ₂ | S ₁ | S ₀ | S ₁ | S ₀ |
| S0 | 0 | 0 | 1 | 0 | 0 |
| S1 | 0 | 1 | 0 | 0 | 1 |
| S2 | 1 | 0 | 0 | 1 | 0 |

divide-by-N counter: codifica binaria

Tabella di verità delle transizioni

| Stato corrente | | Prossimo stato | |
|----------------|-------|----------------|--------|
| S_1 | S_0 | S'_1 | S'_0 |
| 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |

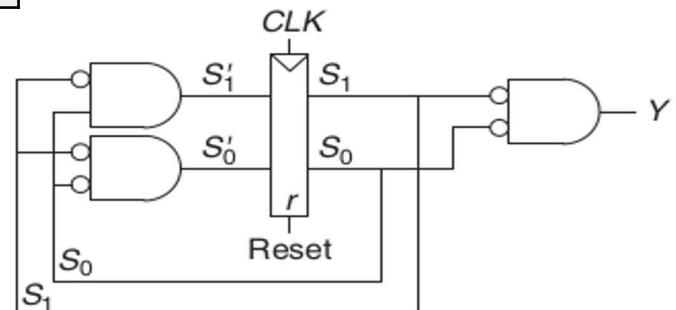
$$S'_1 = \overline{S}_1 S_0$$

$$S'_0 = \overline{S}_1 \overline{S}_0$$

Tabella verità output

| Stato corrente | | Y |
|----------------|-------|---|
| S_1 | S_0 | |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |

$$Y = \overline{S}_1 \overline{S}_0$$



next state logic state register
(a)

output logic output

divide-by-N counter: codifica one-hot

Tabella delle transizioni

| Stato corrente | Prossimo stato |
|----------------|----------------|
| S0 | S1 |
| S1 | S2 |
| S2 | S0 |

Tabella output

| Stato corrente | Y |
|----------------|---|
| S0 | 1 |
| S1 | 0 |
| S2 | 0 |



| Stato corrente | Y | | |
|----------------|----------------|----------------|---|
| S ₂ | S ₁ | S ₀ | |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 |



| Stato corrente | Prossimo stato | | | | |
|----------------|----------------|----------------|-----------------|-----------------|-----------------|
| S ₂ | S ₁ | S ₀ | S' ₂ | S' ₁ | S' ₀ |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 |

| State | One-Hot Encoding | | | Binary Encoding | |
|-------|------------------|----------------|----------------|-----------------|----------------|
| | S ₂ | S ₁ | S ₀ | S ₁ | S ₀ |
| S0 | 0 | 0 | 1 | 0 | 0 |
| S1 | 0 | 1 | 0 | 0 | 1 |
| S2 | 1 | 0 | 0 | 1 | 0 |

divide-by-N counter: one-hot

Tabella di verità delle transizioni

| Stato corrente | | | Prossimo stato | | |
|----------------|-------|-------|----------------|--------|--------|
| S_2 | S_1 | S_0 | S'_2 | S'_1 | S'_0 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 |

Tabella verità output

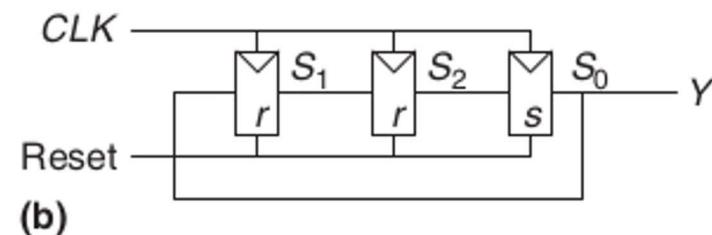
| Stato corrente | | | Y |
|----------------|-------|-------|---|
| S_2 | S_1 | S_0 | |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 |

$$S'_2 = S_1$$

$$S'_1 = S_0$$

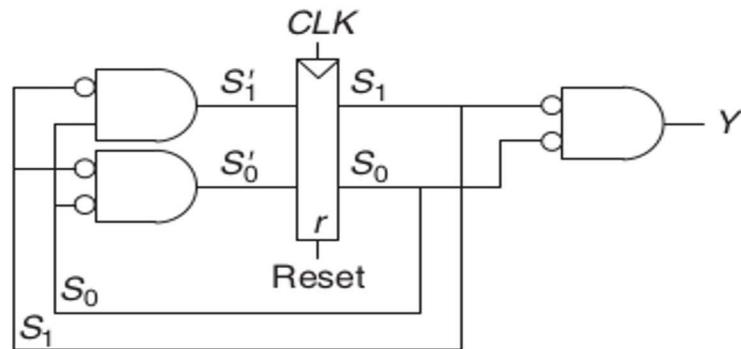
$$S'_0 = S_2$$

$$Y = S_0$$



settable (s) and resettable (r) flip-flops

divide-by-N counter: binary vs one-hot

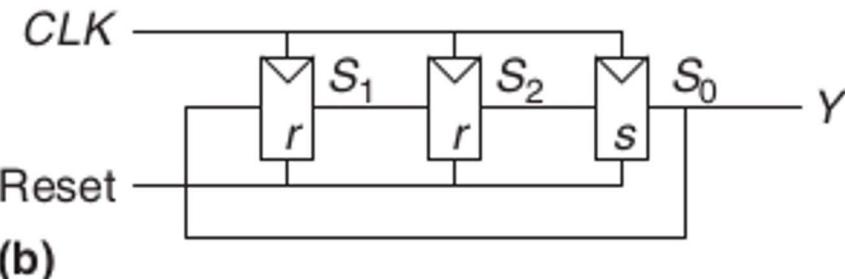


next state logic state register

(a)

output logic

output



settable (s) and resettable (r) flip-flops

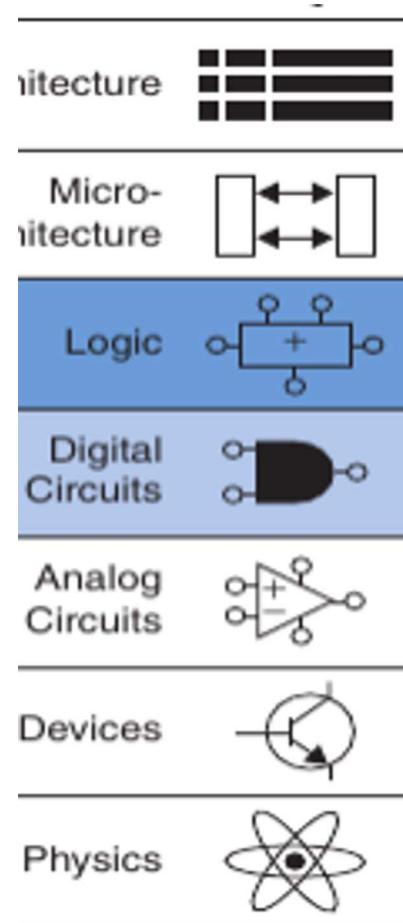
Architettura degli Elaboratori

Lezione 21

Docente: R.Prevete
a.a. 2022/2023
2023.05.05

Logica sequenziale

- Di cosa parleremo
 - Moore vs Mealy
 - Fattorizzazione



Confronto tra macchine di Moore e Mealy

- Moore: output dipende solo dallo stato interno
- Mealy: output dipende dallo stato interno e dall'input



Abbiamo una lumaca che striscia su un nastro di carta con 1 e 0 su di esso. La lumaca sorride ogni volta che le ultime due cifre su cui è strisciata sono 0 e 1. Progettare FSM di Moore e Mealy del cervello della lumaca (Alyssa P. Hacker snail)

Moore vs Mealy FSM

Abbiamo una lumaca che striscia su un nastro di carta con 1 e 0 su di esso. La lumaca sorride ogni volta che le ultime due cifre su cui è strisciata sono 0 e 1. Progettare macchine di Moore e Mealy del cervello della lumaca



Tre stati, situazioni possibili:

- S2: Ho raggiunto il goal (sorrido), Output Y=1
- S1: No ho raggiunto il goal (non sorrido) e sono su 0, Output Y=0
- S0: Non ho raggiunto il goal (non sorrido) e sono su 1, Output Y=0

10011000101101

goal=1

dist_prox_goal ≥ 2

10011000101101

goal=0

dist_prox_goal ≥ 2

10011000101101

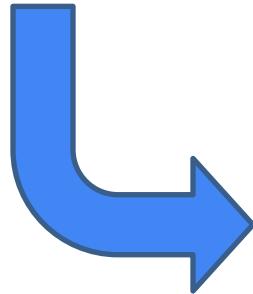
goal=0

dist_prox_goal ≥ 1

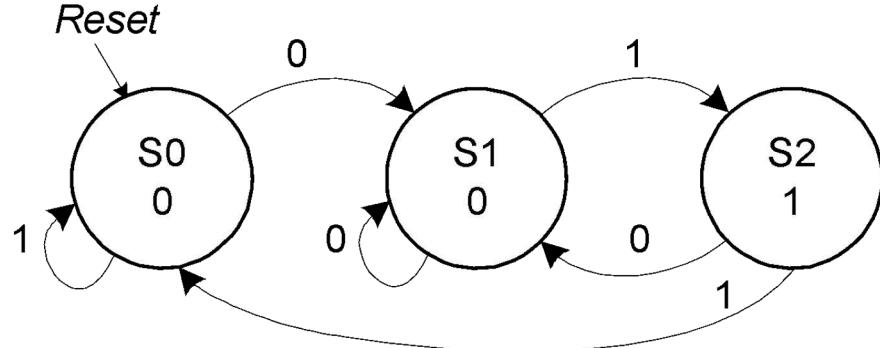
10011000101101

Moore vs Mealy FSM

Abbiamo una lumaca che striscia su un nastro di carta con 1 e 0 su di esso. La lumaca sorride ogni volta che le ultime due cifre su cui è strisciata sono 0 e 1. Progettare macchine di Moore e Mealy del cervello della lumaca



Moore FSM

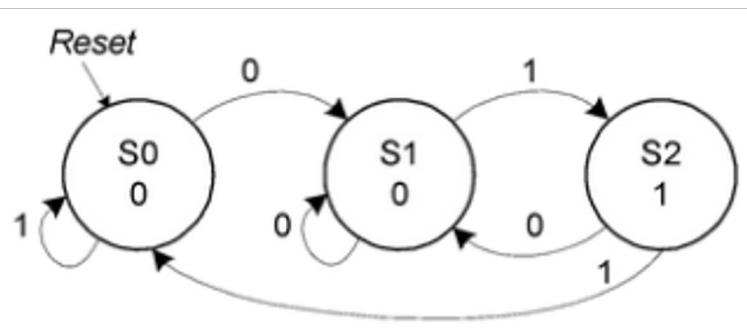


Tre stati, situazioni possibili:

- **S2: Ho raggiunto il goal (sorrido), Output Y=1**
- **S1: Non ho raggiunto il goal (non sorrido) e sono su 0, Output Y=0**
- **S0: Non ho raggiunto il goal (non sorrido) e sono su 1, Output Y=0**

Tabella transizione Moore FSM

| Current State | | Inputs A | Next State | |
|---------------|-------|---------------|------------|--------|
| S_1 | S_0 | | S'_1 | S'_0 |
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 |



S_1S_0 codificano gli stati

| State | Encoding |
|-------|----------|
| S_0 | 00 |
| S_1 | 01 |
| S_2 | 10 |

$$S'_1 = S_0 A$$

$$S'_0 = \overline{A}$$

Tabella transizione Moore FSM

| Current State | | Inputs | Next State | |
|---------------|-------|--------|------------|--------|
| S_1 | S_0 | A | S'_1 | S'_0 |
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 |

| State | Encoding |
|-------|----------|
| S0 | 00 |
| S1 | 01 |
| S2 | 10 |

$$S_1' = S_0 A$$
$$S_0' = \overline{A}$$

Manca S_1 negato, perché?

Tabella transizione Moore FSM

| Current State | | Inputs A | Next State | |
|---------------|-------|---------------|------------|--------|
| S_1 | S_0 | | S'_1 | S'_0 |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | X | 0 |
| 1 | 1 | 1 | X | 0 |

A

Se uso i don't care

| $S_1 S_0$ | | Y | 00 | 01 | 11 | 10 |
|-----------|---|---|----|----|----|----|
| 0 | 0 | 0 | 0 | X | 0 | |
| 1 | 0 | 0 | 1 | X | 0 | |

$$S'_1 = S_0 A$$

Tabella output Moore FSM

| Current State | | Output |
|---------------|-------|--------|
| S_1 | S_0 | Y |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |

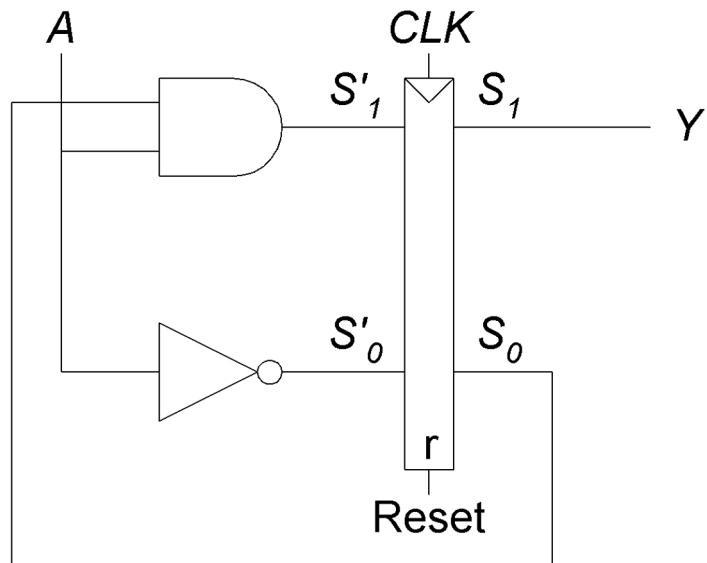
| State | Encoding |
|-------|----------|
| S0 | 00 |
| S1 | 01 |
| S2 | 10 |

$$Y = S_1$$

Tre stati, situazioni possibili:

- S2: Ho raggiunto il goal (sorrido), Output Y=1
- S1: Non ho raggiunto il goal (non sorrido) e sono su 0, Output Y=0
- S0: Non ho raggiunto il goal (non sorrido) e sono su 1, Output Y=0

Schema Moore FSM



$$S_1' = S_0 A$$

$$S_0' = \overline{A}$$

$$Y = S_1$$

Macchina di Mealy

Mealy machine:

- solo 2 stati: S1 sono sul digit 0, S0 sono su digit 1
- ciascun arco labellato con input /output: A/Y
- A è il valore di input che causa la transizione
- Y il corrispondente output

Mealy FSM

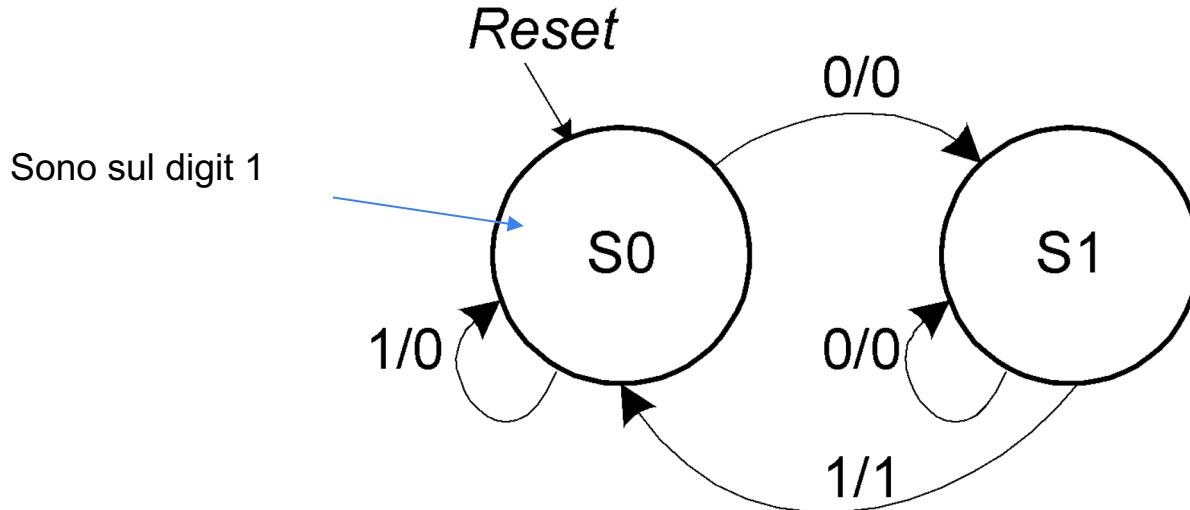


Tabella transizione/output Mealy FSM

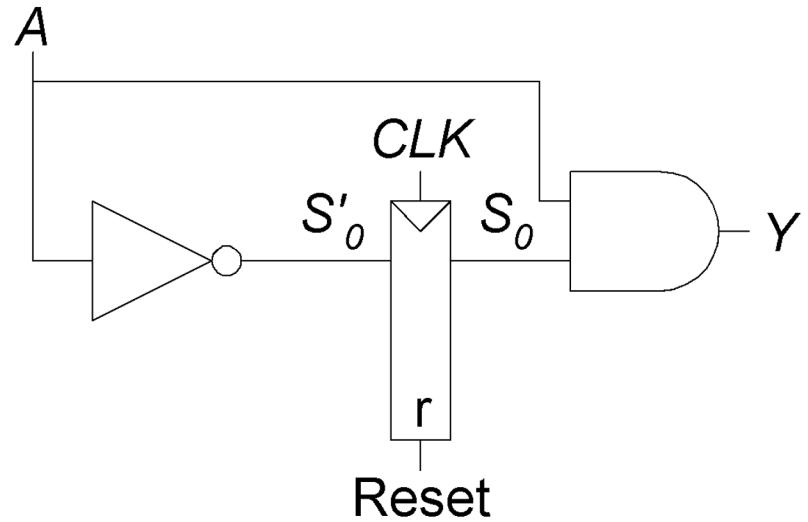
| Current State | Input | Next State | Output |
|---------------|-------|------------|--------|
| S | A | S' | Y |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

| State | Encoding |
|-------|----------|
| S0 | 0 |
| S1 | 1 |

$$S' = \bar{A}$$

$$Y = SA$$

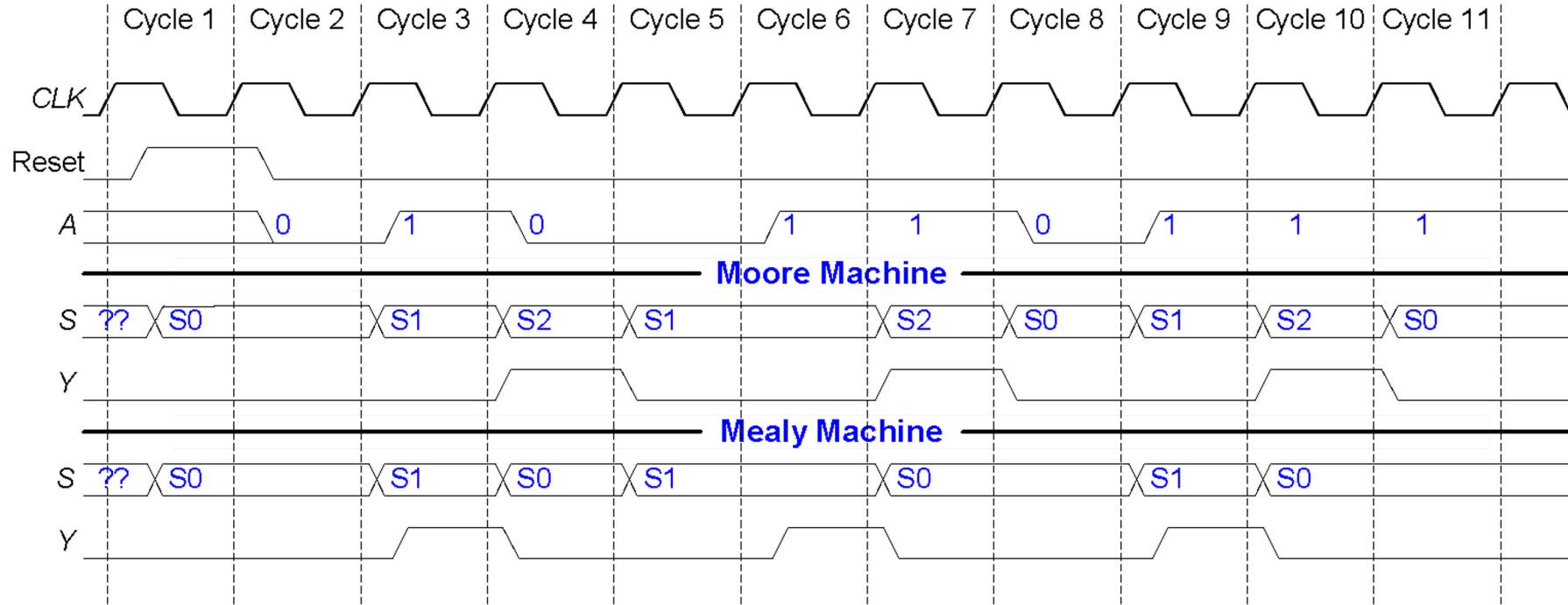
Schema Mealy FSM



$$S' = \overline{A}$$

$$Y = SA$$

Moore & Mealy Timing

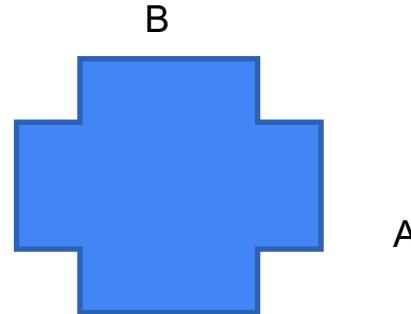


Fattorizzazione di FSM

- Fattorizzare consiste nel suddividere una FSM complessa in FSM più piccole che interagiscono fra loro
- Output di una o più FSM diventa input di una o più altre FSM
- Organizzazione modulare e gerarchica

Fattorizzazione di FSM

- **Esempio:** Considerate di voler modificare il controller di semafori per tener conto di possibili parate
 - Altri due inputs: P, R
 - Se $P = 1$, entra in modalità *Parade* e il semaforo di Bravado Blvd (strada B) rimane verde
 - Se $R = 1$, lascia la modalità *Parade*



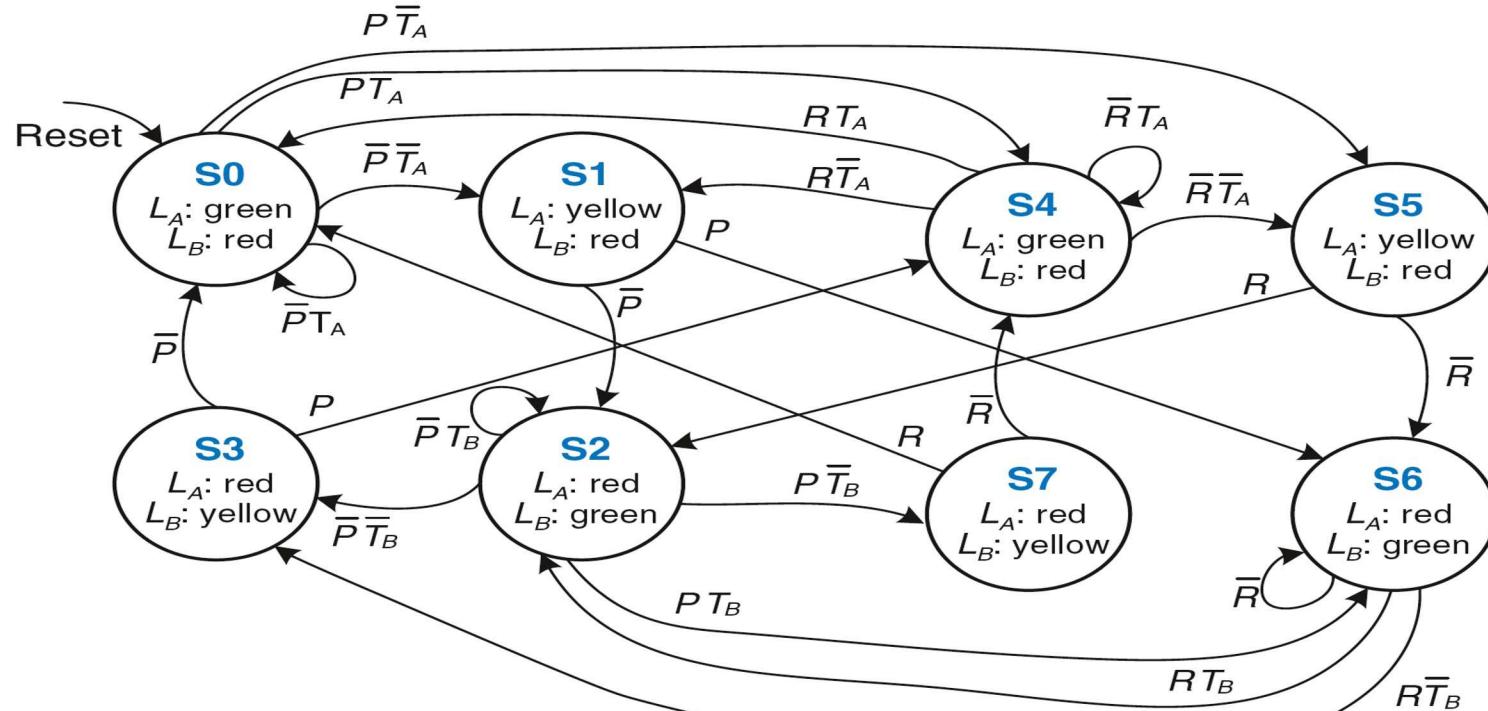
Semafori intelligenti

- T_A e T_B sensori traffico
 - $T_A, T_B = 0$ (no traffico), 1 (si traffico)
- L_A, L_B colori semaforo
 - $L_A, L_B = \text{rosso, giallo, verde}$

Parade FSM: non fattorizzato

Diagramma delle
transizioni

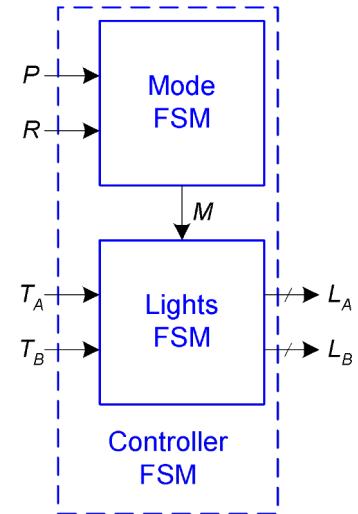
FSM non fattorizzato



Parade FSM: fattorizzato

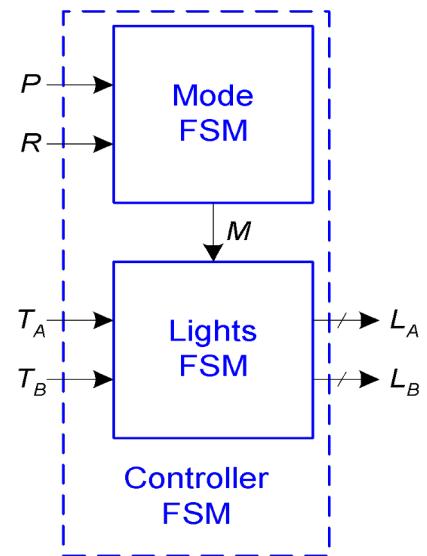
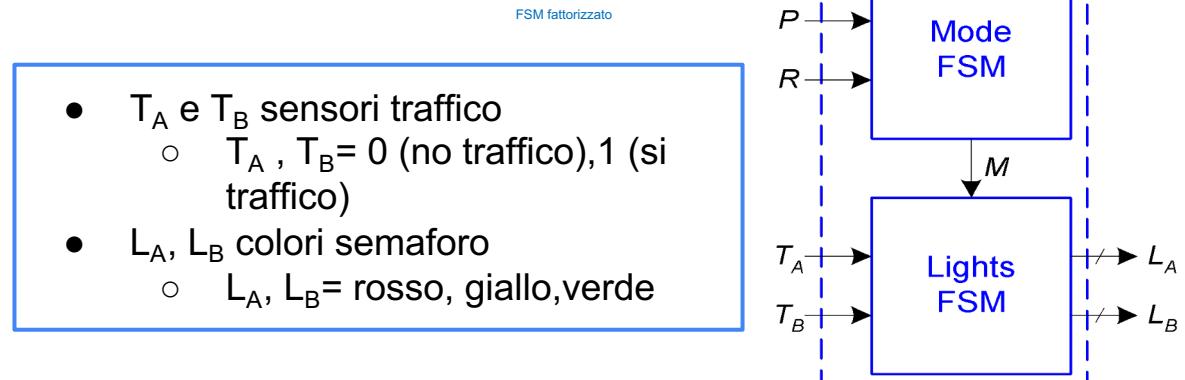
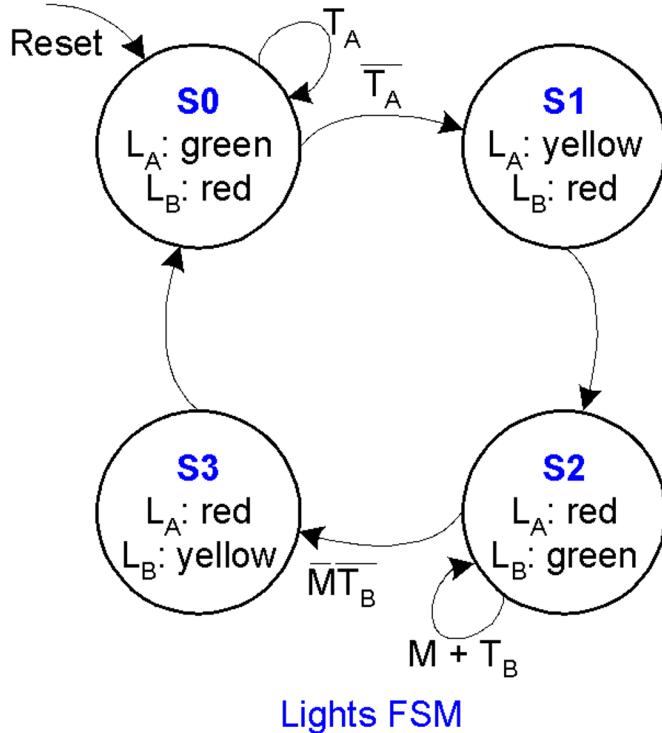
FSM fattorizzato

- T_A e T_B sensori traffico
 - $T_A, T_B = 0$ (no traffico), 1 (si traffico)
- L_A, L_B colori semaforo
 - $L_A, L_B = \text{rosso, giallo, verde}$

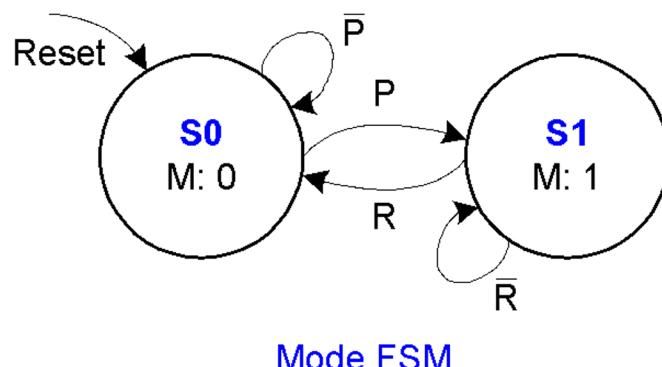


- $P = 1$, entra in modalità *Parade* e il semaforo di Bravado Blvd (strada B) rimane verde
- Se $R = 1$, lascia la modalità *Parade*

Parade FSM: fattorizzato



- **P** = 1, entra in modalità Parade semaforo strada B resta verde
- **R** = 1, lascia modalità Parade



Progettare una FSM

1. Identificare gli input e output
2. Abbozzare uno state transition diagram
3. Scrivere la state transition table
4. Selezionare un encoding degli stati, input ed output
5. Macchina di Moore:
 - a. Riscrivere la state transition table con l'encoding degli stati e degli input
 - b. Scrivere la output table
5. Macchina di Mealy:
combinare la state transition table e la output table con gli encoding degli stati, ed input ed output.
6. Scrivere le equazioni booleane relative alla logica di prossimo stato e alla logica di output e minimizzare le equazioni
7. Logica prossimo strato e logica di output corrispondono a due circuiti combinatori
8. Fare uno schema del circuito

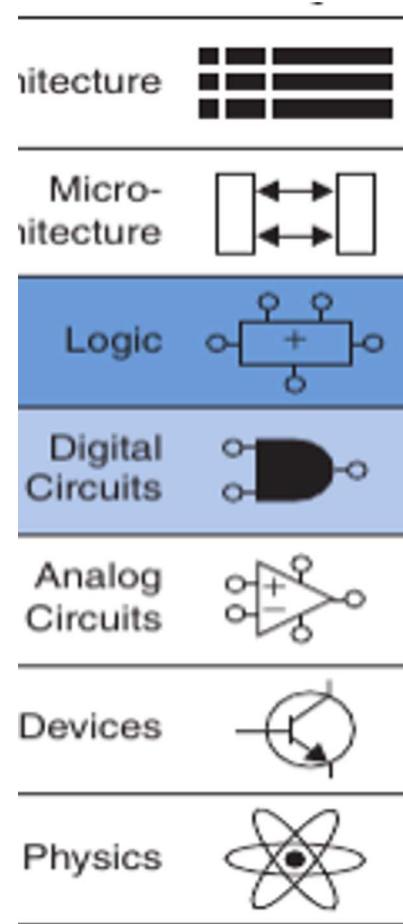
Architettura degli Elaboratori

Lezione 22

Docente: R.Prevete
a.a. 2022/2023
2023.05.08

Logica sequenziale

- Di cosa parleremo
 - Esercizi su macchine a stati finiti



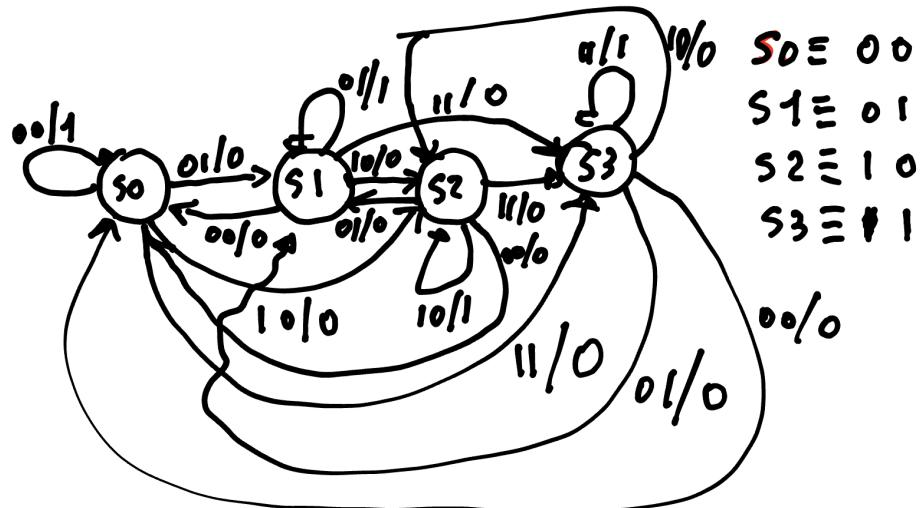
Esercizio

- Progettare una Mealy FSM F con due input (A e B) e un output Q.
 - $Q=1$ sse A e B assumono rispettivamente il valore precedente

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

Esempio

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



$$\begin{aligned} S_0 &\equiv 00 \\ S_1 &\equiv 01 \\ S_2 &\equiv 10 \\ S_3 &\equiv 11 \end{aligned}$$

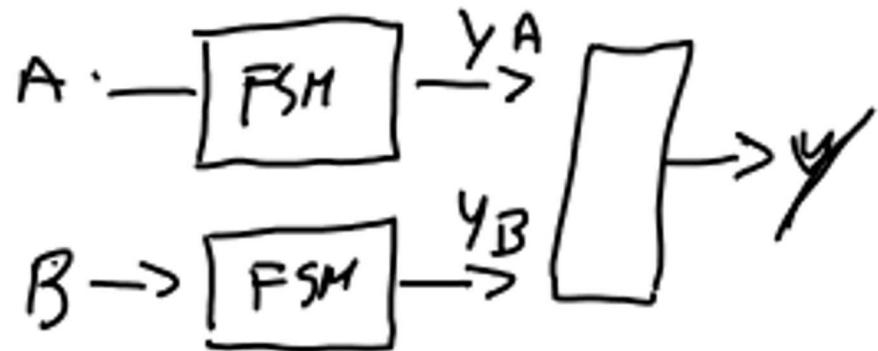
$$\begin{aligned} S_0 &\equiv 00 \\ S_1 &\equiv 01 \\ S_2 &\equiv 10 \\ S_3 &\equiv 11 \end{aligned}$$

AUTOMA A STATI FINITO MA COMPLICATO!!!

NON FATTORIZZATA!!!!

Esempio

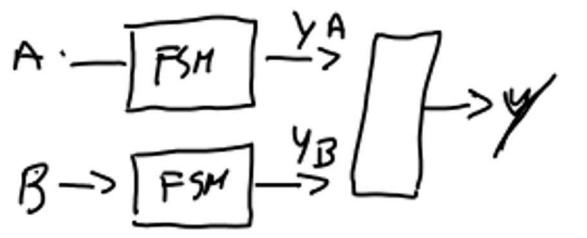
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| A | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| B | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| Q | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |



FATTORIZZATA!!!!

Esempio

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



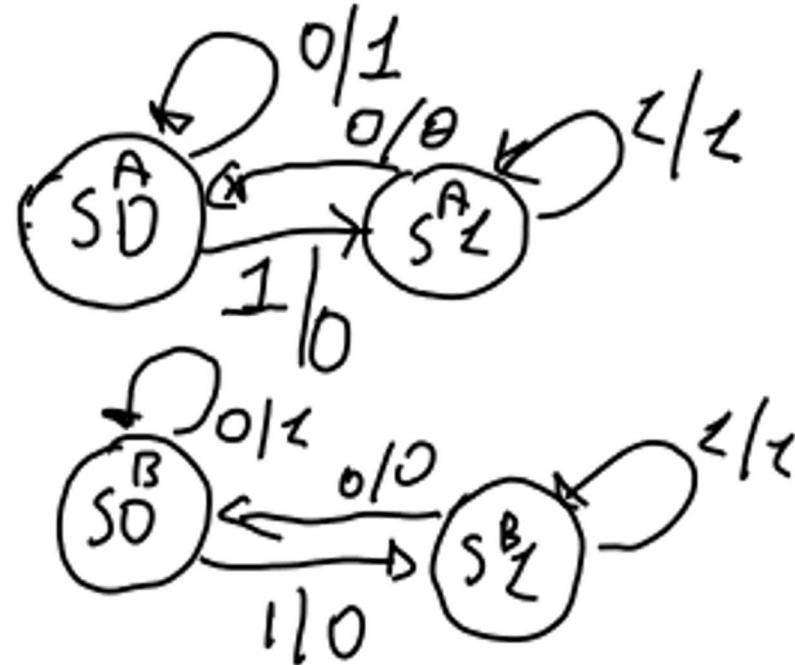
FATTORIZZATA!!!!

$$S^A_0 = 0$$

$$S^A_1 = 1$$

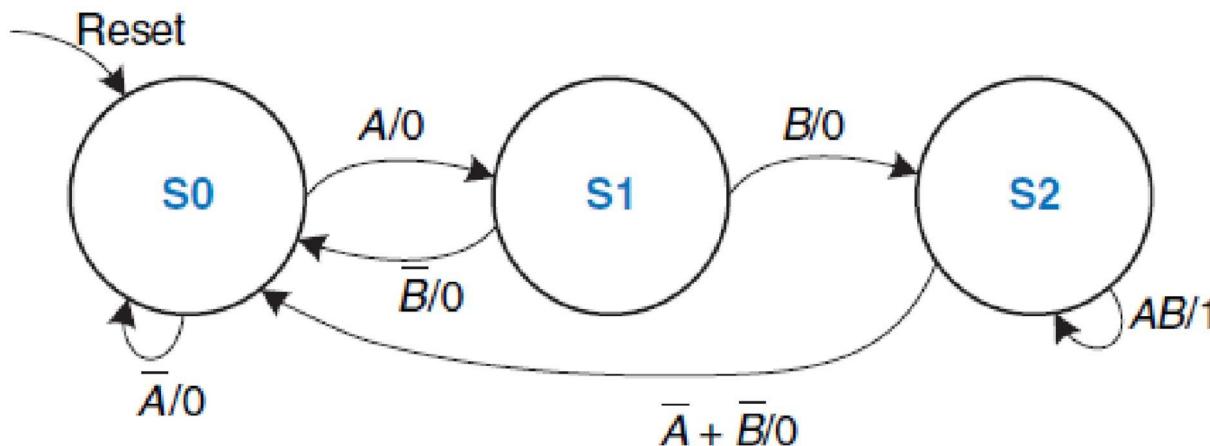
$$S^B_0 = 0$$

$$S^B_1 = 1$$

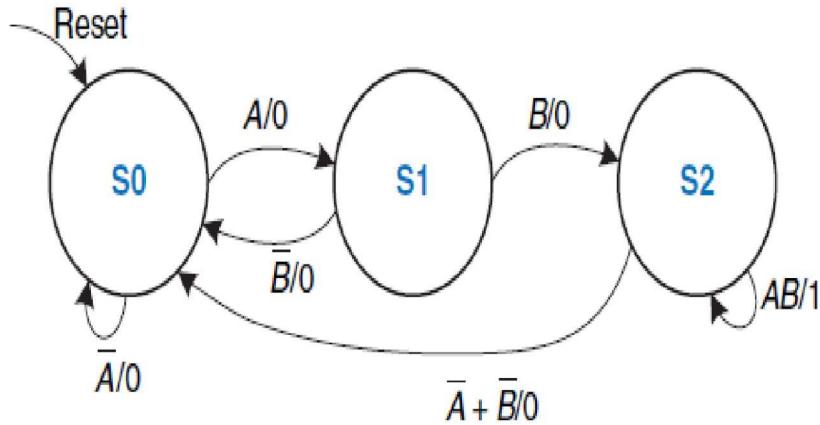


Esercizio 3.23 (dal libro di testo)

Descrivere prima “a parole” cosa fa FSM in figura. Dopodichè usando una codifica binaria, definire la tabella di transizione degli stati e la tabella di output. Scrivere le equazioni Booleane per il prossimo stato e per l’output. Infine, uno schema del circuito digitale.

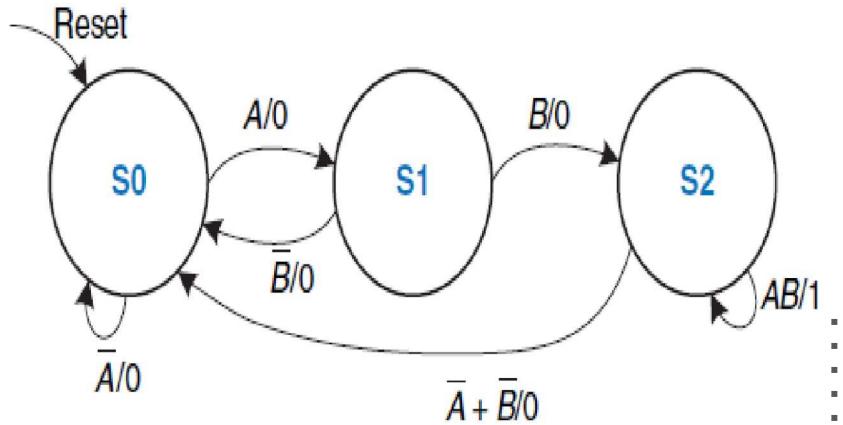


Esercizio 3.23



- FSM di Mealy
- Input A, B, output: Y
- Tre stati S_0, S_1, S_2
- Stato di Reset S_0
- Se S_0 , A falso resto in S_0 . $Y=0$
- Se S_0 , A vero vado in S_1 . $Y=0$
- Se S_1 , B vero vado in S_2 . $Y=0$
- Se S_1 , B falso ritorno in S_0 . $Y=0$
- Se S_2 , A e B veri, resto in S_2 . $Y=1$
- Se S_2 , A o B falso, ritorno in S_0 . $Y=0$

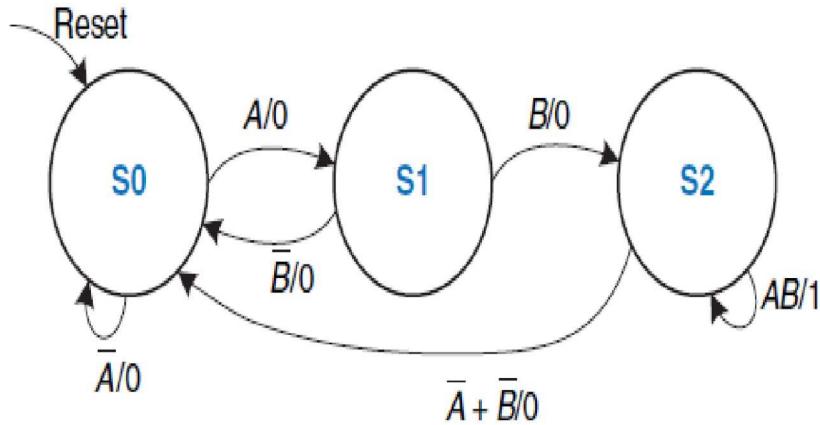
Esercizio 3.23



- In questo stato dipende solo dal input A
- Il suo output è sempre 0
- Cambia stato se A è vero

- Input A, B, output: Y
- tre stati S0,S1, S2
- Stato di Reset S0
- Se S0, A falso resto in S0. Y=0
- Se S0, A vero vado in S1. Y=0
- Se S1, B vero vado in S2. Y=0
- Se S1, B falso ritorno in S0. Y=0
- Se S2, A e B veri, resto in S2. Y=1
- Se S2, A o B falso, ritorno in S0. Y=0

Esercizio 3.23

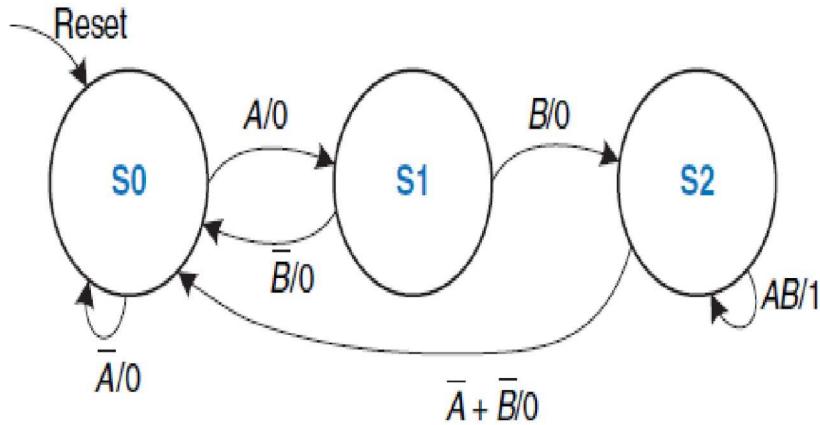


- In questo stato dipende solo dal input B
- Il suo output è sempre 0
- Cambia stato sempre
- Se B è falso ritorno in S0, altrimenti vado in S2

- Input A, B, output: Y
- tre stati S0, S1, S2
- Stato di Reset S0
- Se S0, A falso resto in S0. Y=0
- Se S0, A vero vado in S1. Y=0
- Se S1, B vero vado in S2. Y=0
- Se S1, B falso ritorno in S0. Y=0
- Se S2, A e B veri, resto in S2. Y=1
- Se S2, A o B falso, ritorno in S0. Y=0

Significa che riconosco una sequenza A=vero, B=Vero. Lo stato S1 quindi significa che ho precedentemente avuto A=vero come input

Esercizio 3.23



- In questo stato dipende sia da A sia da B
- Il suo output è 1 quando vede A e B veri, e resta nello stesso stato
- Altrimenti ritorna in S0

.....

- Input A, B, output: Y
- tre stati S0, S1, S2
- Stato di Reset S0
- Se S0, A falso resto in S0. Y=0
- Se S0, A vero vado in S1. Y=0
- Se S1, B vero vado in S2. Y=0
- Se S1, B falso ritorno in S0. Y=0
- Se S2, A e B veri, resto in S2. Y=1
- Se S2, A o B falso, ritorno in S0. Y=0

Riconosco (Es.):

A 1 X 1 1 0

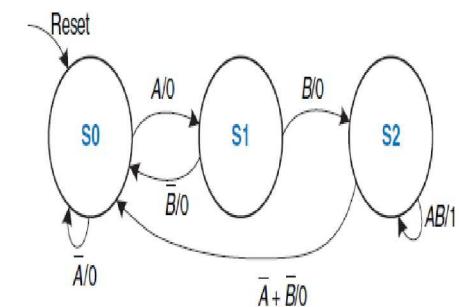
B X 1 1 1 1

Significa che la mia FSM riconosce A=vero, seguito da A, B Vero almeno 1 volta

Esercizio 3.23

| current state | inputs | next state | output | | | |
|---------------|--------|------------|--------|--------|--------|---|
| s_1 | s_0 | a | b | s'_1 | s'_0 | q |
| 0 | 0 | 0 | X | 0 | 0 | 0 |
| 0 | 0 | 1 | X | 0 | 1 | 0 |
| 0 | 1 | X | 0 | 0 | 0 | 0 |
| 0 | 1 | X | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 |

| state | encoding $s_{1:0}$ |
|-------|-----------------------|
| S0 | 00 |
| S1 | 01 |
| S2 | 10 |



Esercizio 3.23

| s_1 | s_0 | a | b | s'_1 |
|-------|-------|-----|-----|--------|
| 0 | 0 | 0 | X | 0 |
| 0 | 0 | 1 | X | 0 |
| 0 | 1 | X | 0 | 0 |
| 0 | 1 | X | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |

$$S'_1 = \bar{S}_1 S_0 B + S_1 \bar{S}_0 A B$$

?? Abbiamo dei Don't care ??

Proviamo a passare a k-map

Esercizio 3.23

| s_1 | s_0 | a | b | $s'_{1:0}$ |
|-------|-------|-----|-----|------------|
| 0 | 0 | 0 | X | 0 |
| 0 | 0 | 1 | X | 0 |
| 0 | 1 | X | 0 | 0 |
| 0 | 1 | X | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |

$$S'_1 = \bar{S}_1 S_0 B + S_1 \bar{S}_0 A B$$

AB

| | 00 | 01 | 11 | 10 |
|----|----|----|----|----|
| 00 | 0 | 0 | 0 | 0 |
| 01 | 0 | 1 | 1 | 0 |
| 11 | X | X | X | X |
| 10 | 0 | 0 | 1 | 0 |

$S_{1:0}$



$$S'_1 = S_0 B + S_1 A B$$

Esercizio 3.23

| current state | | inputs | | next state | | output |
|---------------|-------|--------|-----|------------|--------|--------|
| s_1 | s_0 | a | b | s'_1 | s'_0 | q |
| 0 | 0 | 0 | X | 0 | 0 | 0 |
| 0 | 0 | 1 | X | 0 | 1 | 0 |
| 0 | 1 | X | 0 | 0 | 0 | 0 |
| 0 | 1 | X | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 |

A green rectangle highlights the first two columns (current state) and the last column (output). A red rectangle highlights the fifth column (next state). An arrow points from the highlighted area to the equation $S'_0 = \bar{S}_1 \bar{S}_0 A$.

$$S'_0 = \bar{S}_1 \bar{S}_0 A$$

Esercizio 3.23

| current state | | inputs | | next state | | output |
|---------------|-------|--------|-----|------------|--------|--------|
| s_1 | s_0 | a | b | s'_1 | s'_0 | q |
| 0 | 0 | 0 | X | 0 | 0 | 0 |
| 0 | 0 | 1 | X | 0 | 1 | 0 |
| 0 | 1 | X | 0 | 0 | 0 | 0 |
| 0 | 1 | X | 1 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 |

$$S'_1 = S_0B + S_1AB$$

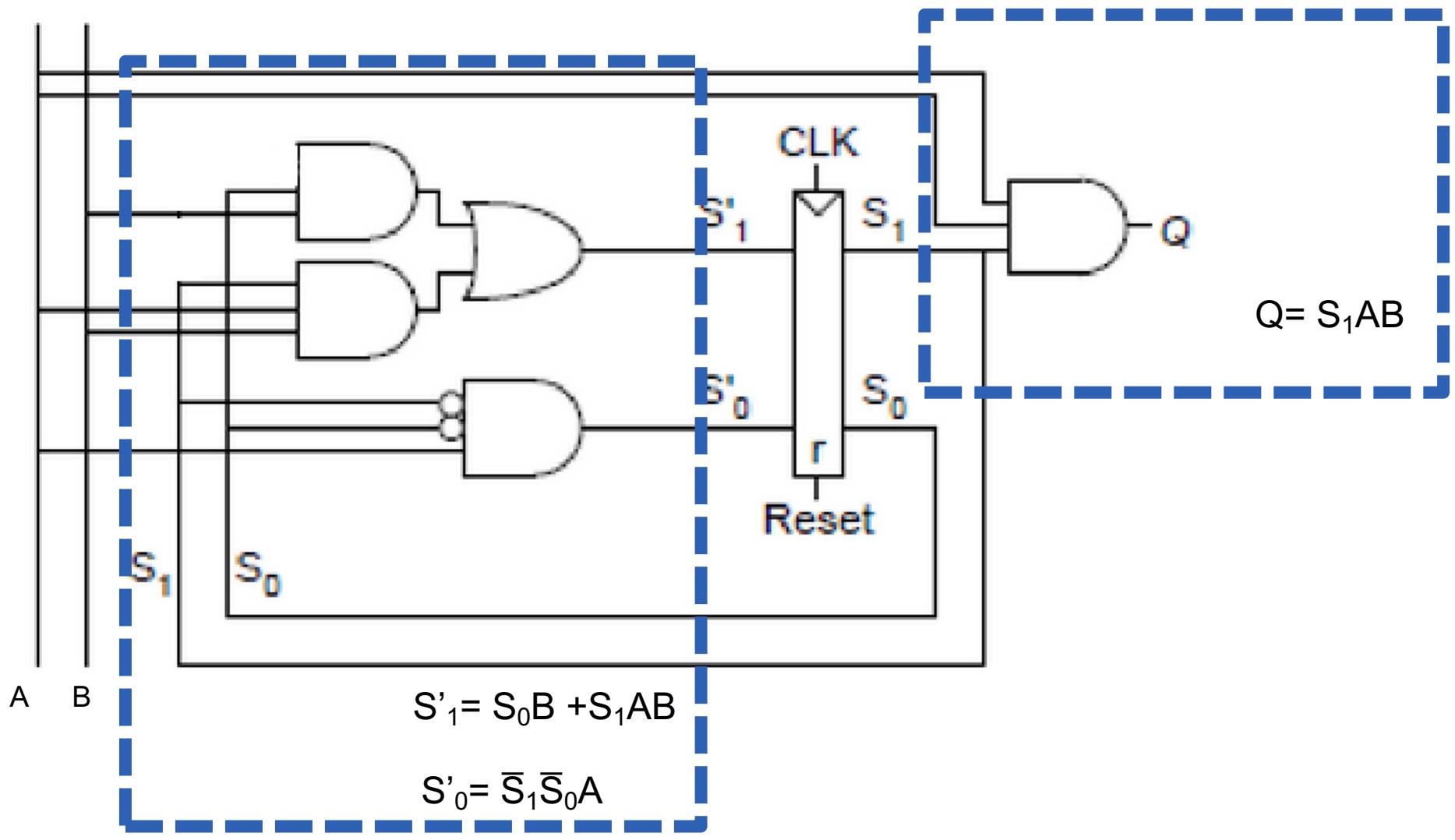
$$S'_0 = \bar{S}_1\bar{S}_0A$$

Le due formule Booleane precedenti corrispondono al circuito combinatorio per ottenere il prossimo stato

$$Q = S_1AB$$

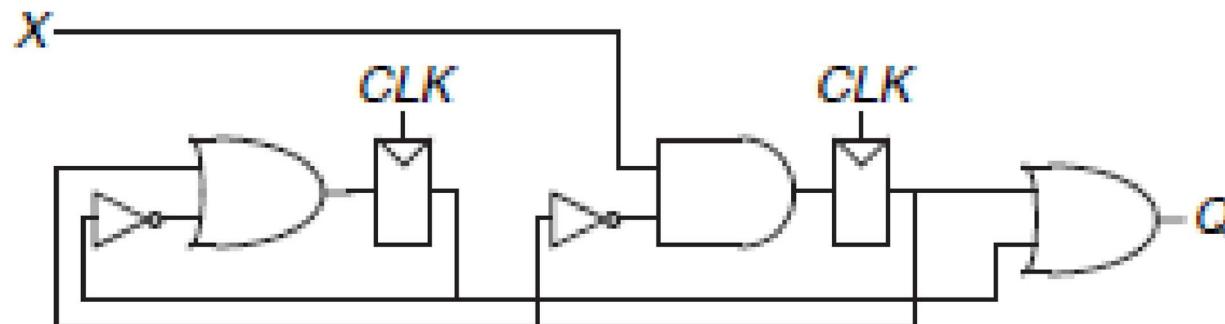
La formula Booleana precedente corrisponde al circuito combinatorio per ottenere output

Per l'output ispezionando la tabella si puo' vedere che si ottiene 1 solo quando A e B sono 1 insieme ad S_1



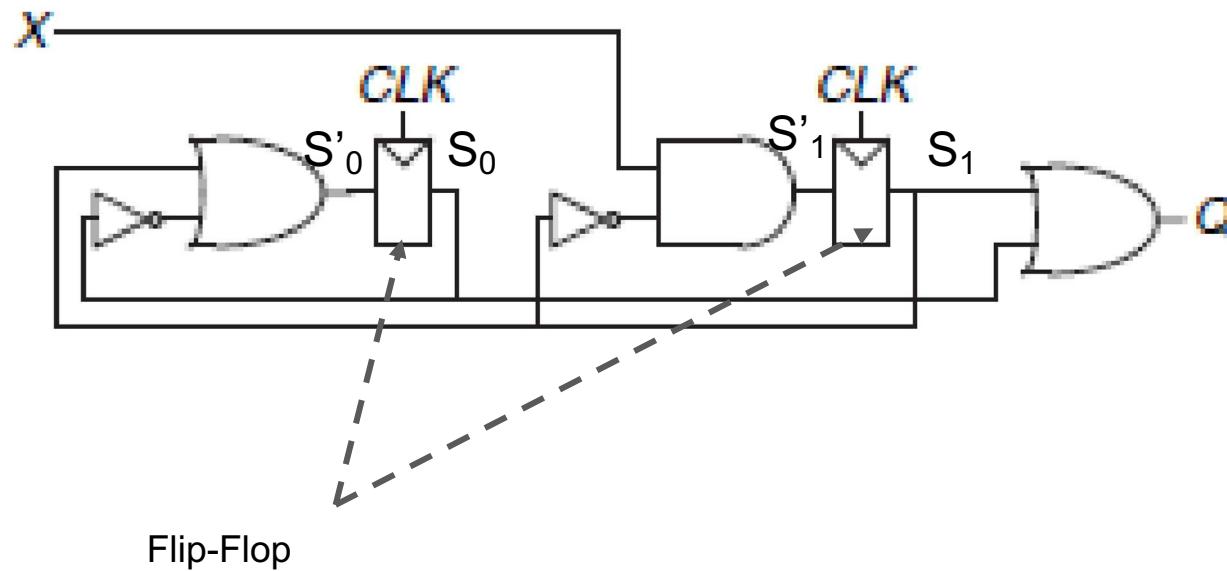
Esercizio 3.31

Analizzare la FSM in figura. Disegnare il diagramma di transizione e scrivere le tabelle di transizione e di output. Commentare cosa la FSM realizza.



Esercizio 3.31

Analizzare la FSM in figura. Disegnare il diagramma di transizione e scrivere le tabelle di transizione e di output. Commentare cosa la FSM realizza.



| | S ₁ | S ₀ |
|----|----------------|----------------|
| S0 | 0 | 0 |
| S1 | 0 | 1 |
| S2 | 1 | 0 |
| S3 | 1 | 1 |

Codifica binaria
degli stati

Esercizio 3.31

Analizzare la FSM in figura. Disegnare il diagramma di transizione e scrivere le tabelle di transizione e di output. Commentare cosa la FSM realizza.

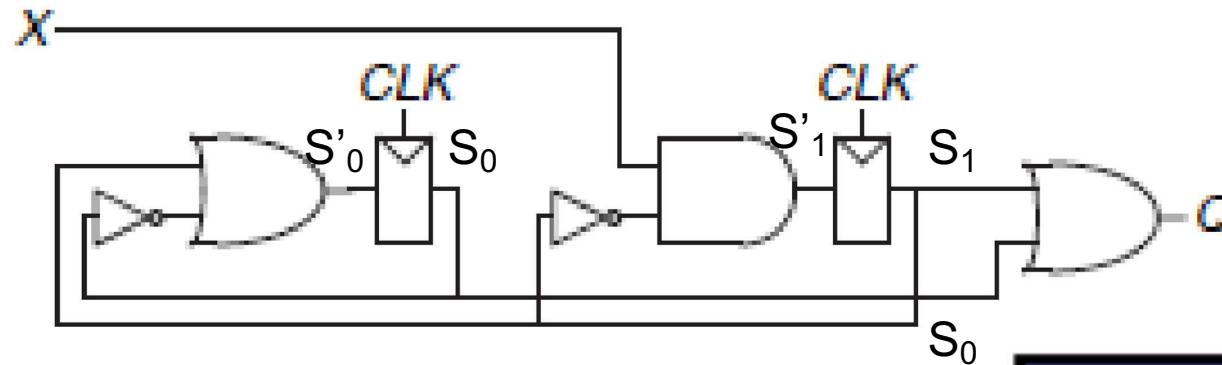
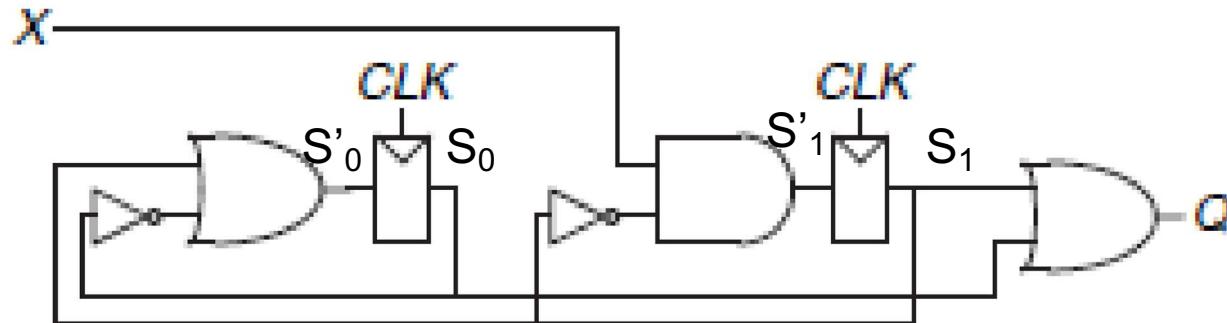


Tabella output

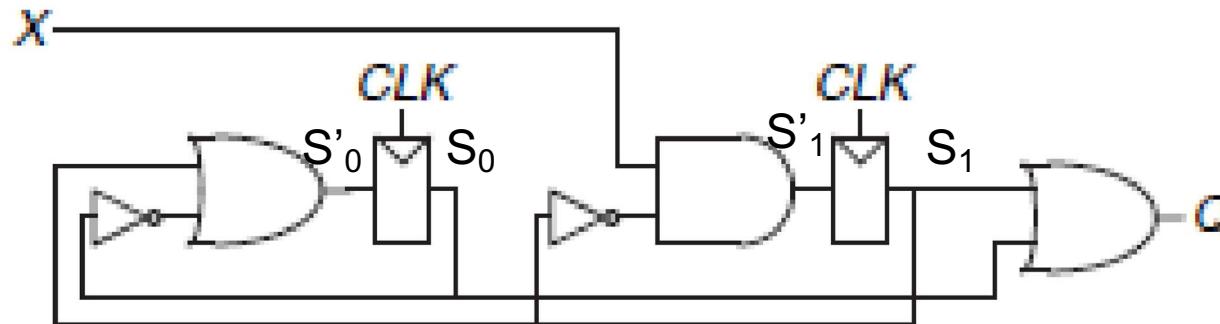
| current state | output | |
|---------------|--------|-----|
| s_1 | s_0 | q |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | X | 1 |

Esercizio 3.31



| Stato corrente | | Prossimo stato |
|----------------|-------|----------------|
| S_1 | S_0 | S'_0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Esercizio 3.31



| Input | Stato corrente | Prossimo stato |
|-------|----------------|----------------|
| X | S_1 | S'_1 |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Esercizio 3.31

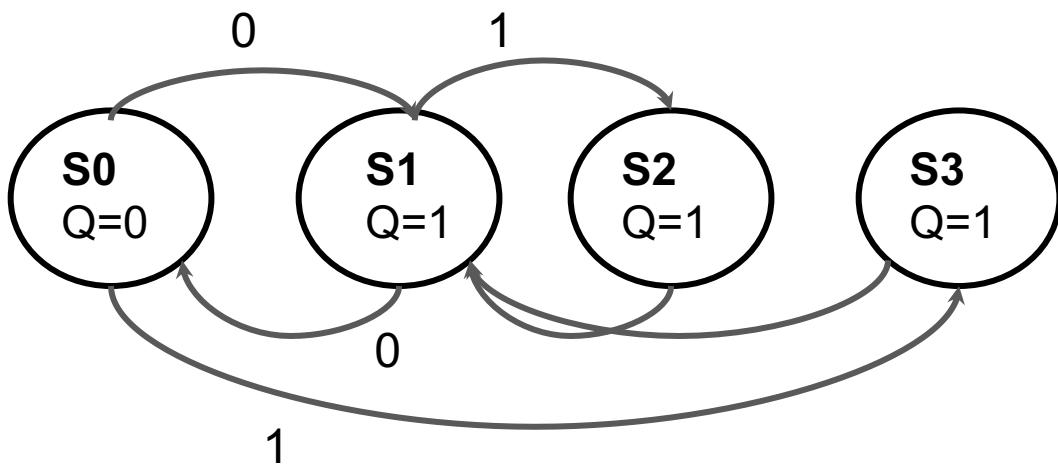
| current state | | input | next state | |
|---------------|-------|-------|------------|--------|
| s_1 | s_0 | x | s'_1 | s'_0 |
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | x | x | 0 | 1 |

Tabella transizioni

Tabella output

| current state | | output |
|---------------|-------|--------|
| s_1 | s_0 | q |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | x | 1 |

Esercizio 3.31



| | s_1 | s_0 |
|-------|-------|-------|
| s_0 | 0 | 0 |
| s_1 | 0 | 1 |
| s_2 | 1 | 0 |
| s_3 | 1 | 1 |

| current state | | output |
|---------------|-------|--------|
| s_1 | s_0 | q |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | x | 1 |

| current state | | input | next state | |
|---------------|-------|-------|------------|--------|
| s_1 | s_0 | x | s'_1 | s'_0 |
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | x | x | 0 | 1 |

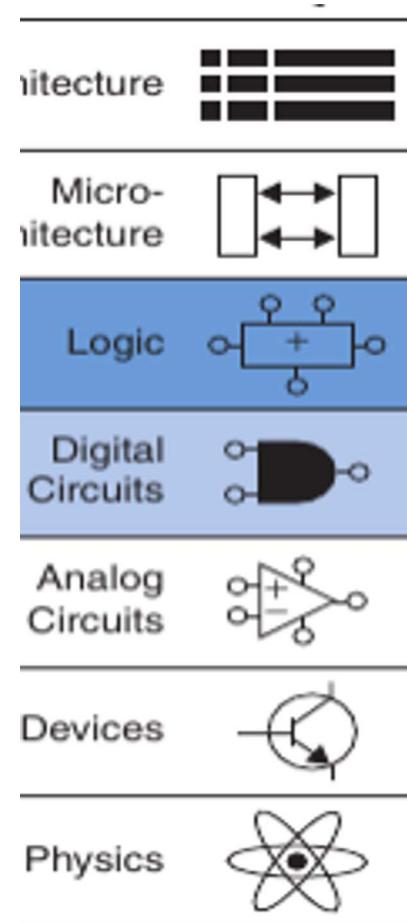
Architettura degli Elaboratori

Lezione 23

Docente: R.Prevete
a.a. 2022/2023
2023.05.10

Logica sequenziale

- Di cosa parleremo
 - Cenni sul parallelismo
 - Circuiti sequenziali più complessi



Cenni sul parallelismo

Parallelismo

- **Goal**
 - **Vogliamo velocizzare i tempi di computazione**
 - Una possibilità è legato alla possibilità di fare delle operazioni in parallelo

Parallelismo: due concetti chiave

Latenza e throughput

Definiamo:

- **Token:** Gruppo di input da processare per ottenere un output significativo

Allora:

- **Latency (latenza):** Tempo che occorre ad un token per essere processato e produrre un output
- **Throughput (capacità di trasmissione):** Numero di output prodotti per unità di tempo

Esempio di Latency e Throughput

- Ben vuole fare delle torte
- 5 minuti per preparare una torta
- 15 minuti per cucinarla

$$\text{Latency} = 5 + 15 = 20 \text{ minuti} = \mathbf{1/3 \text{ h}}$$

$$\text{Throughput} = \frac{1}{\text{latency}} = \mathbf{3 \text{ torte/h}}$$

Unità di tempo: **h**



Throughput è il reciproco
della latenza e viceversa

Parallelismo e Throughput

Latency = 5 + 15 = 20 minuti = **1/3 h**

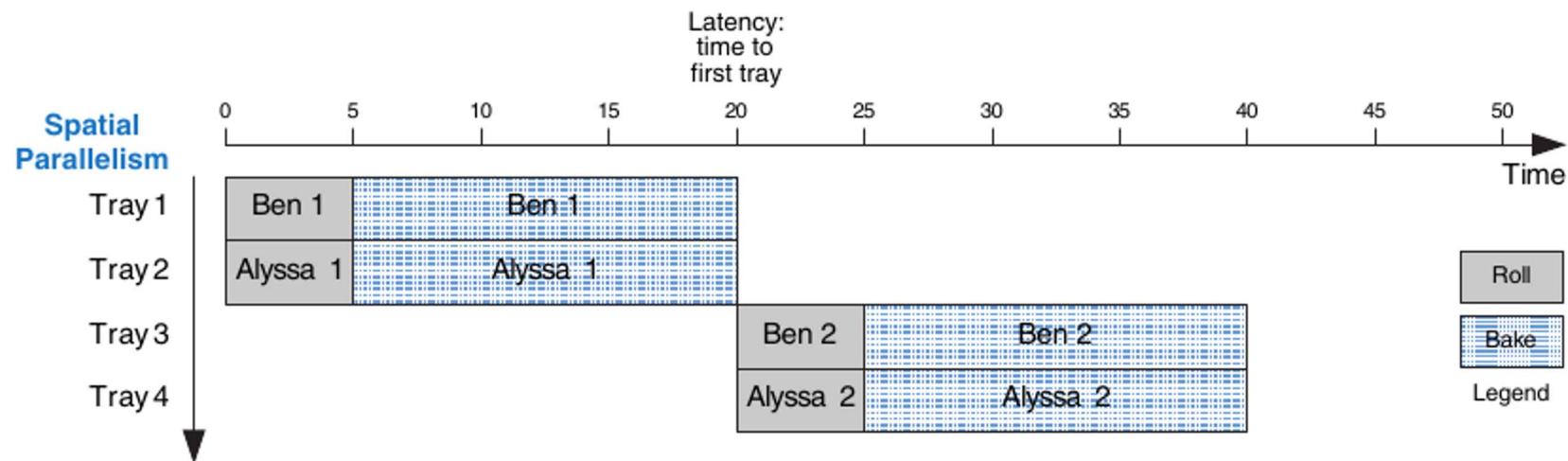
Throughput = $\frac{1}{latency}$ = **3 torte/h**

il parallelismo incrementa il throughput

Esempio di parallelismo

- **parallelismo spaziale:** Ben chiede a Allysa di aiutarlo, usando anche il suo forno
- **parallelismo temporale:**
 - 2 fasi: preparare e cucinare
 - Mentre una torta è in forno, Ben ne prepara un'altra, etc.

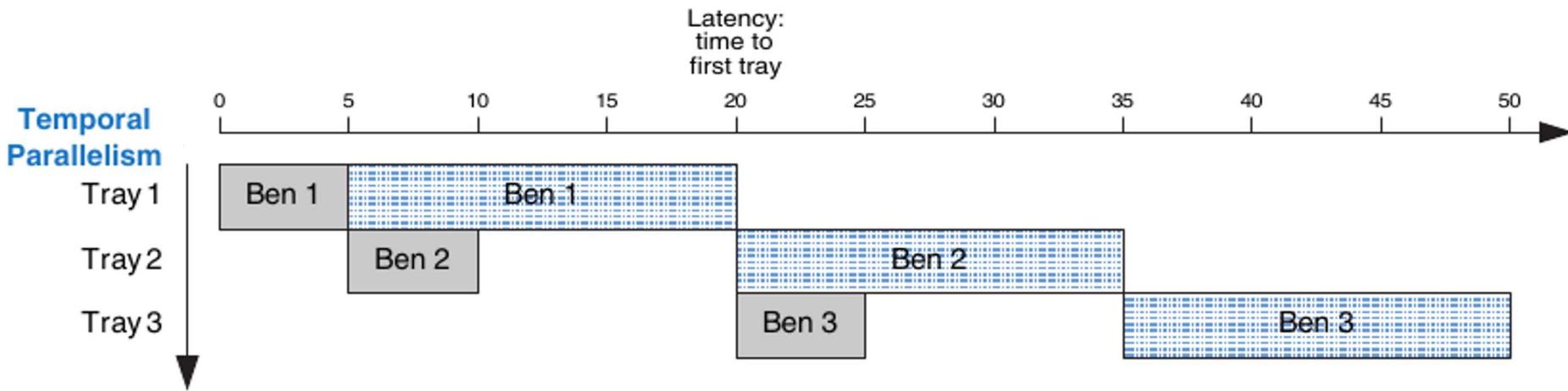
Parallelismo spaziale



$$\text{Latency} = 5 + 15 = 20 \text{ minuti} = \mathbf{1/3 \text{ h}}$$

$$\text{Throughput} = 2 \frac{1}{latency} = \mathbf{6 \text{ torte/h}}$$

Parallelismo temporale



- **Latency**= 15m = $\frac{1}{4}$ h
- **Troughput**= 4 torte /h

Parallelismo

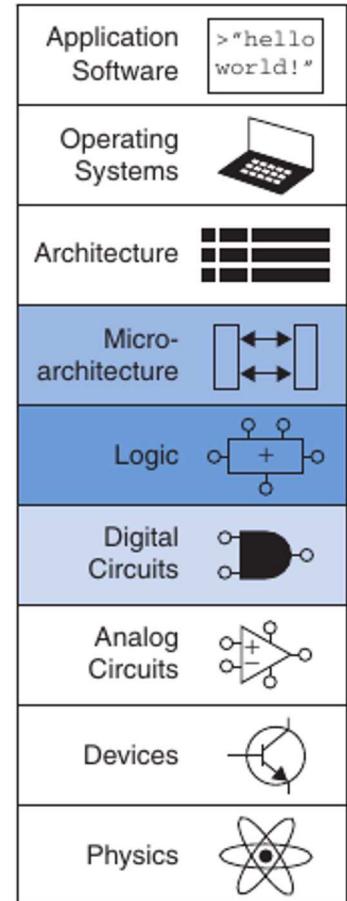
- 2 tipi di parallelismo:
 - **Spaziale**
 - duplicare l'hardware per eseguire più task contemporaneamente
 - Latenza L, N copie del hardware: Throughput = N/L
 - **Temporale**
 - Il task è suddiviso in più fasi
 - Le diverse fasi sono eseguite in pipelining
 - Latenza L, task di N fasi di uguale lunghezza (caso limite) : Throughput = N/L

Parallelismo: svantaggi

- Non si applicano a tutti situazioni. La rovina del parallelismo sono le dipendenze.
- L'attività corrente può dipendere dal risultato dell'attività precedente
 - Ad esempio, se Ben vuole controllare che il primo vassoio di biscotti abbia un buon sapore prima di iniziare a preparare il secondo, ha una dipendenza che impedisce il funzionamento in parallelo.
- Tuttavia, Il parallelismo è uno delle tecniche più importanti per la progettazione di sistemi digitali ad alte prestazioni.

Circuiti aritmetici e memorie

- Abbiamo visto logica combinatoria e sequenziale.
- **Vedremo** blocchi combinatori e sequenziali più elaborati. Questi blocchi includono:
 - circuiti aritmetici,
 - contatori, registri a scorrimento,
 - array di memoria e array logici.
- Questi blocchi esemplificano i principi di gerarchia e modularità



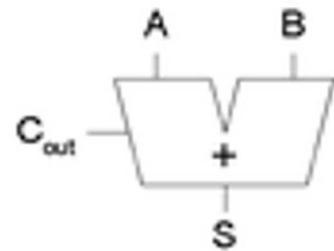
Addizione

L'addizione è l'operazione più comune nei sistemi digitali.

- Prima mostreremo come addizionare **due numeri binari ad 1 bit** e
- poi vedremo con più bit.

1-Bit Adders: half adder

Half Adder



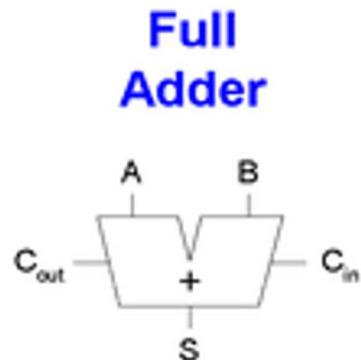
- Ho solo due input (A e B)
- Ho due uscite: la somma (S) ed il riporto (the carry, C)

$$\begin{aligned} S &= A \oplus B \\ C_{out} &= AB \end{aligned}$$

$$S = \text{not}(A)B + \text{not}(B)A$$

| A | B | C_{out} | S |
|---|---|-----------|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

1-Bit Adders: full adder



- Ho tre input (A ,B ed un riporto C_{in})
- Ho due uscite: la somma (S) ed il riporto (the carry, C_{out})

| C_{in} | A | B | C_{out} | S |
|----------|---|---|-----------|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

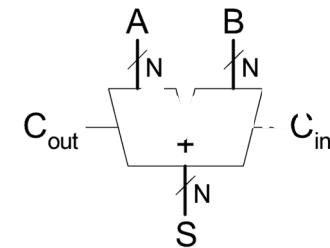
Stringa1= ... A X ...

Stringa2= ... B Y ...

Stringa1 + Stringa2, C_{in} =riporto della somma di X ed Y (tale somma a sua volta ha bisogno del riporto precedente)

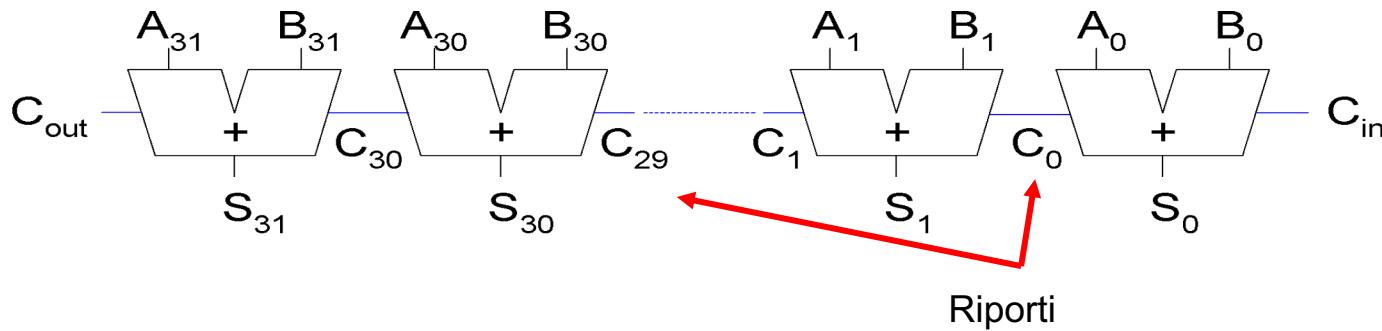
Multibit Adders (CPAs)

- Tipologie di propagazione del riporto (carry propagate adders-CPAs):
 - Ripple-carry (lento)
 - Carry-lookahead (veloce)
- Carry-lookahead adders offrono prestazioni migliori ma richiedono più hardware



Ripple-Carry Adder

- $N > 1$ 1-bit full adder concatenati insieme
- Il riporto si propaga lungo la catena
- E' un buon esempio di modularità e regolarità
- Svantaggio: **lento**



**Devo sempre attendere la computazione
precedente!!**

Ritardo del Ripple-Carry

$$t_{\text{ripple}} = N t_{FA}$$

- t_{FA} è il ritardo di un singolo 1-bit full adder
- N è il numero di 1-bit full adders

Carry-Lookahead Adder

IDEA: Calcolare il riporto C_{out} di un **blocco di k -bit** usando due funzioni separate:

- *Generate*: considera quando il blocco di k -bit genera un riporto
- *Propagate*: considera quando il blocco di k -bit propaga un riporto dal blocco precedente

Carry-Lookahead Adder

Partiamo dal consideriamo due generici bit A_i e B_i

| C_{in} | A_i | B_i | C_{out} |
|----------|-------|-------|-----------|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |

Carry-Lookahead Adder

Partiamo dal consideriamo due generici bit A_i e B_i

| C_{in} | A_i | B_i | C_{out} |
|----------|-------|-------|-----------|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |

Propago

Propago

Genero

Carry-Lookahead Adder

Il bit i -esimo produce un riporto (carry out) per generazione o per propagazione di un riporto del bit $(i-1)$ -esimo (carry in)

- **Generate:** Il bit i -esimo **genera** un carry out se A_i e B_i sono entrambi 1. $G_i = A_i B_i$ (funzione generate)
- **Propagate:** Il bit i -esimo **propaga** un carry al carry out se A_i o B_i sono 1. $P_i = A_i + B_i$ (funzione propagate)
- **Carry out:** il carry i (C_i) è data da:

$$C_i = A_i B_i + (A_i + B_i) C_{i-1} = G_i + P_i C_{i-1}$$

Block Propagate and Generate

Ora dobbiamo estendere le funzioni Propagate and Generate a blocchi di k -bits, ovvero:

- calcolare se un blocco di k -bit $(i, i+1, \dots, i+k-1)$ **propaga** il carry out del blocco precedente (ovvero da C_{i-1} a C_{i+k-1})
- calcolare se un blocco di k -bit $(i, i+1, \dots, i+k)$ **genera** un carry out (in C_{i+k-1})

Esempi

Propago
il riporto

4 bit

Riporto dal blocco precedente

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | C |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | A |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | B |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | |

Non genero
il riporto

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | C |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | A |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | B |
| 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | |

Esempi

Diagram illustrating the propagation of a 4-bit error in a 16-bit word across three stages (C, A, B) and its effect on the output.

The diagram shows two cases: "Non genero e non propago il riporto" (Case 1) and "genero il riporto" (Case 2).

4 bit Riporto dal blocco precedente

| | | | | | | | | | | | | | | | | | | |
|--|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | C |
| Non genero e non propago il riporto | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | A |
| | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | B |
| | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | |
| genero il riporto | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | C |
| | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | A |
| | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | B |
| | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | |

Esempi

non genero il
riporto

| 4 bit | | | | | | | | | | | | | | | | C |
|-------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | A |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | B |
| 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | |

Propagate e Generate per un blocco di 4 bit

- *Propagate:* $P_i = A_i + B_i$
 - $P_{3:0} = P_3 P_2 P_1 P_0$ $G_i = A_i B_i$
- *Generate:*
 - $G_{3:0} = G_3 + P_3 \quad G_2 + P_3 P_2 \quad G_1 + P_3 P_2 P_1 G_0 =$
 $G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0))$

Cioè, una colonna genera il riporto e le successive lo devono propagare

$$C_{out} = G_{3:0} + P_{3:0} C_{in}$$

Esempio

Propago il riporto

| | 4 bit | | | | | | | | | | | | | | | Riporto dal blocco precedente |
|---|-------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|-------------------------------|
| C | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| B | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |

genero il riporto

| | 4 bit | | | | | | | | | | | | | | | |
|---|-------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| C | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| B | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |

$$P_{3:0} = P_3 P_2 P_1 P_0 \quad G_{3:0} = G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0))$$

$$C_{out} = G_{3:0} + P_{3:0} C_{in}$$

Propagate e Generate per un blocco di k-bit

- In generale,

$$P_{i:j} = P_i P_{i-1} P_{i-2} \dots P_j$$

$$G_{i:j} = G_i + P_i (G_{i-1} + P_{i-1} (G_{i-2} + P_{i-2} (\dots G_j) \dots))$$

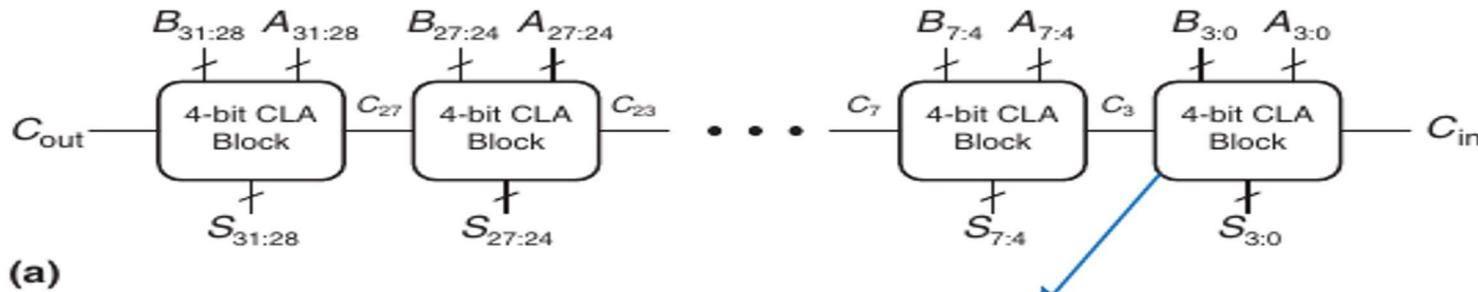
$$C_i = G_{i:j} + P_{i:j} C_{j-1} \text{ (od anche } C_{\text{out}} = G_{i:j} + P_{i:j} C_{\text{in}})$$

Con $i-j=k$

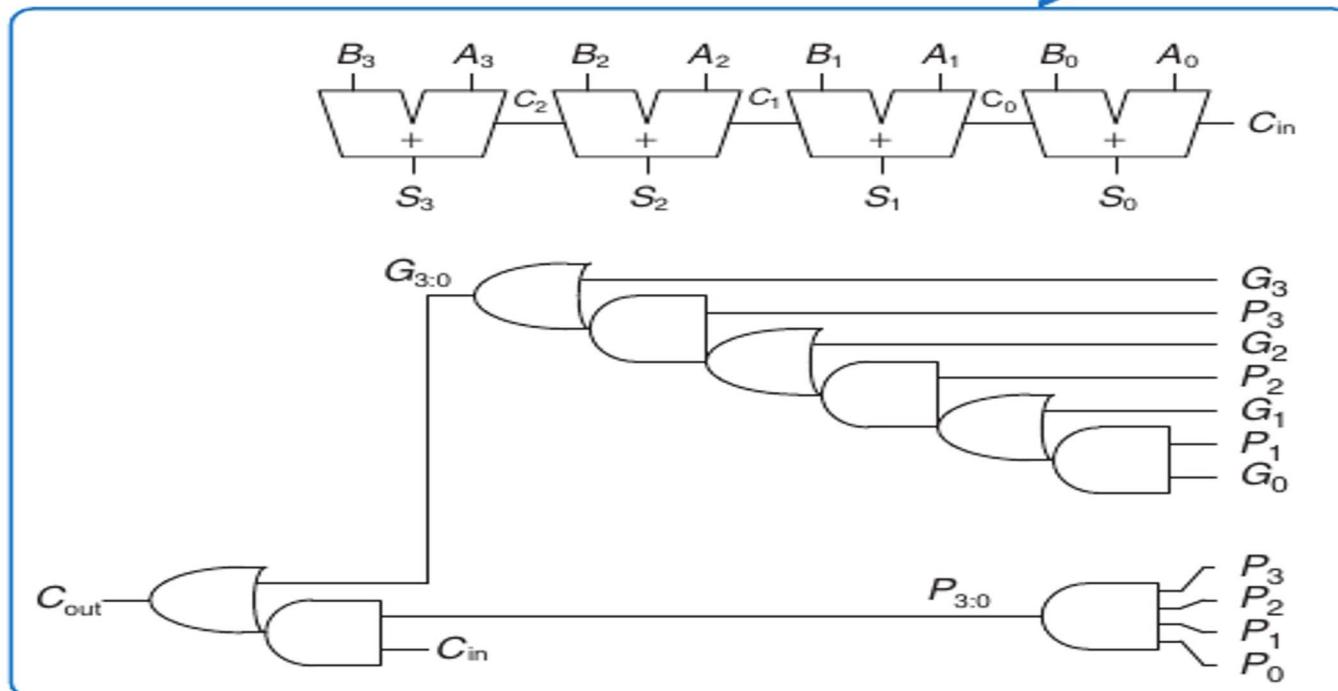
Carry-Lookahead Addition

- **Step 1:** calcola tutti i G_i and P_i
- **Step 2:** calcola G and P per blocchi di k -bit
- **Step 3:** C_{in} si propaga mediante la logica propagate/generate dei vari blocchi di k -bit

32-bit CLA con 4-bit Blocks



(a)



Ritardo del Carry-Lookahead Adder

Per un N -bit CLA con blocchi di k bit:

$$t_{CLA} = t_{pg} + t_{pg_block} + (N/k - 1)t_{AND_OR} + kt_{FA}$$

- t_{pg} : ritardo per generare P_i, G_i
- t_{pg_block} : ritardo per generare $P_{i:j}, G_{i:j}$
- t_{AND_OR} : ritardo delle porte AND/OR a monte della logica propagate/generate, questo ritardo si propaga da C_{in} a C_{out} che si propaga lungo i $N/k - 1$ blocchi

Un N -bit carry-lookahead adder è generalmente più veloce di un ripple-carry adder per $N > 16$

Confronto

- 32-bit ripple-carry vs 32-bit CLA con blocchi di 4 bit
- 2-input gate delay = 100 ps; full adder delay = 300 ps

$$\begin{aligned} t_{\text{ripple}} &= Nt_{FA} = 32(300 \text{ ps}) \\ &= \mathbf{9.6 \text{ ns}} \end{aligned}$$

$$\begin{aligned} t_{\text{CLA}} &= t_{pg} + t_{pg_block} + (N/k - 1)t_{\text{AND_OR}} + kt_{FA} \\ &= [100 + 600 + (7)200 + 4(300)] \text{ ps} \\ &= \mathbf{3.3 \text{ ns}} \end{aligned}$$

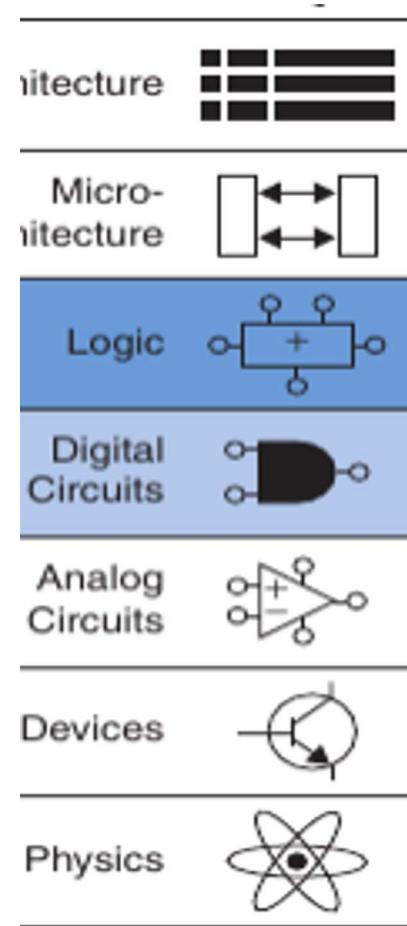
Architettura degli Elaboratori

Lezione 24

Docente: R.Prevete
a.a. 2022/2023

Logica sequenziale

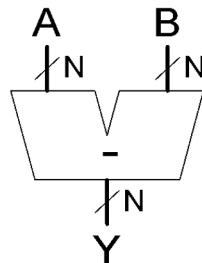
- Di cosa parleremo
 - Circuiti sequenziali più complessi:
 - sottrattore
 - comparatore
 - ALU (Arithmetic Logic Unit)



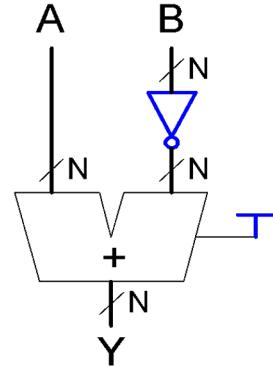
Sottrattore

- Nella rappresentazione a complemento a 2 per complementare un numero occorre invertire ogni cifra e sommare 1
 - Es: $2 = 0010_2$ $-2 = 1101_2$ $+0001_2 = 1110_2$

Symbol

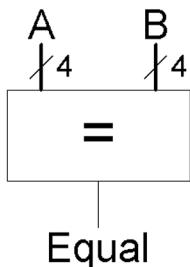


Implementation

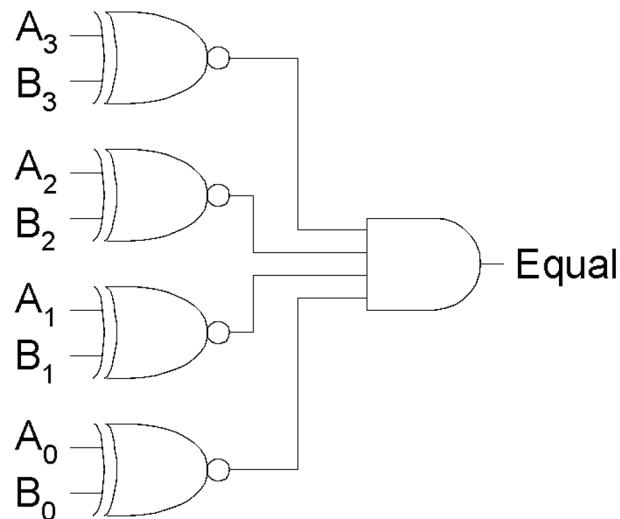


Comparatore: uguaglianza

Symbol



Implementation



| A _i | B _i | XOR | Not(XOR) |
|----------------|----------------|-----|----------|
| 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

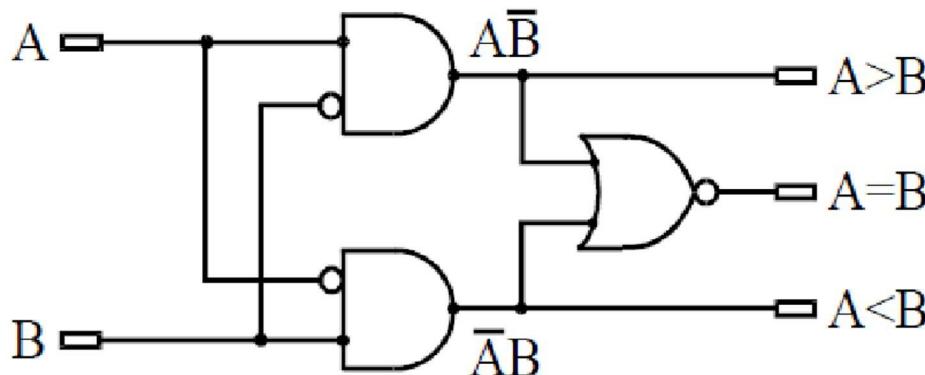
Comparatore completo a un bit

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

Voglio sapere se:

- $<$
- $>$
- $=$

Per l'uguale:
 $\text{not}(A)\text{not}(B) + \text{not}(A)B =$
 $\text{Not}(A)\text{not}(B)$ Not(not(A)B)=
 $(\text{not}(A)+B)(A+\text{not}(B))$
prodotto di MAXTERM

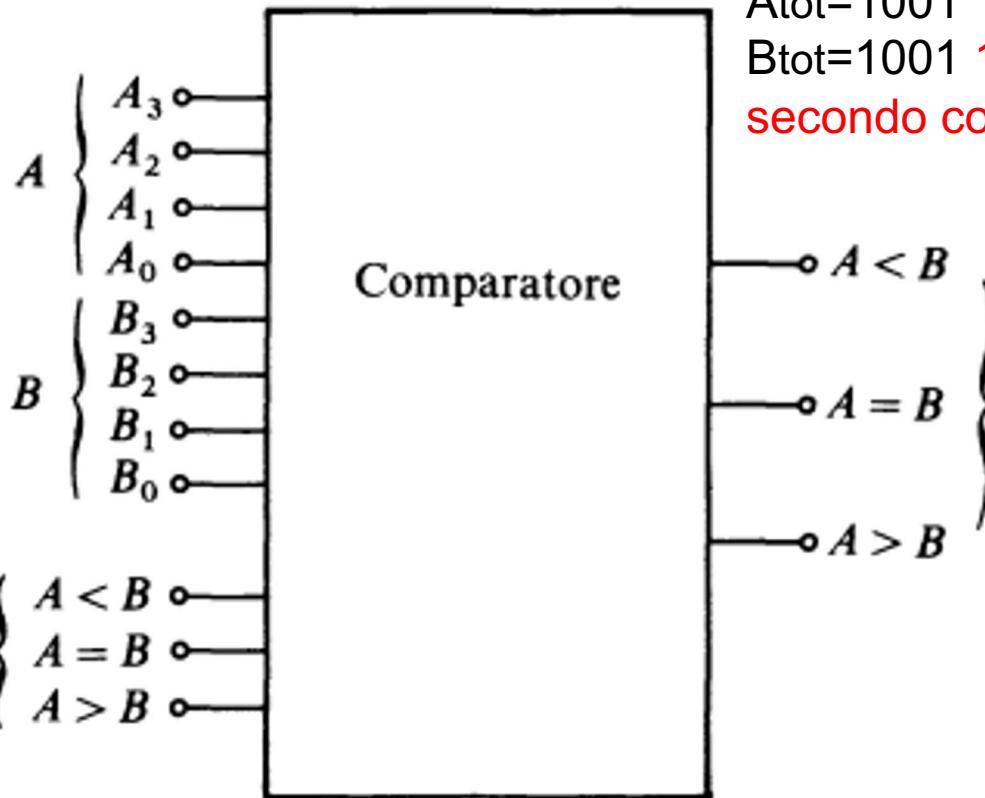


Comparatore completo a 4 bit (1)

A= 1011

B= 1111

Input dati



A_{tot}=1001 1011

B_{tot}=1001 1111 Continuo con il
secondo comparatore, altrimenti no

Output

L'idea:

21 > 17 ??

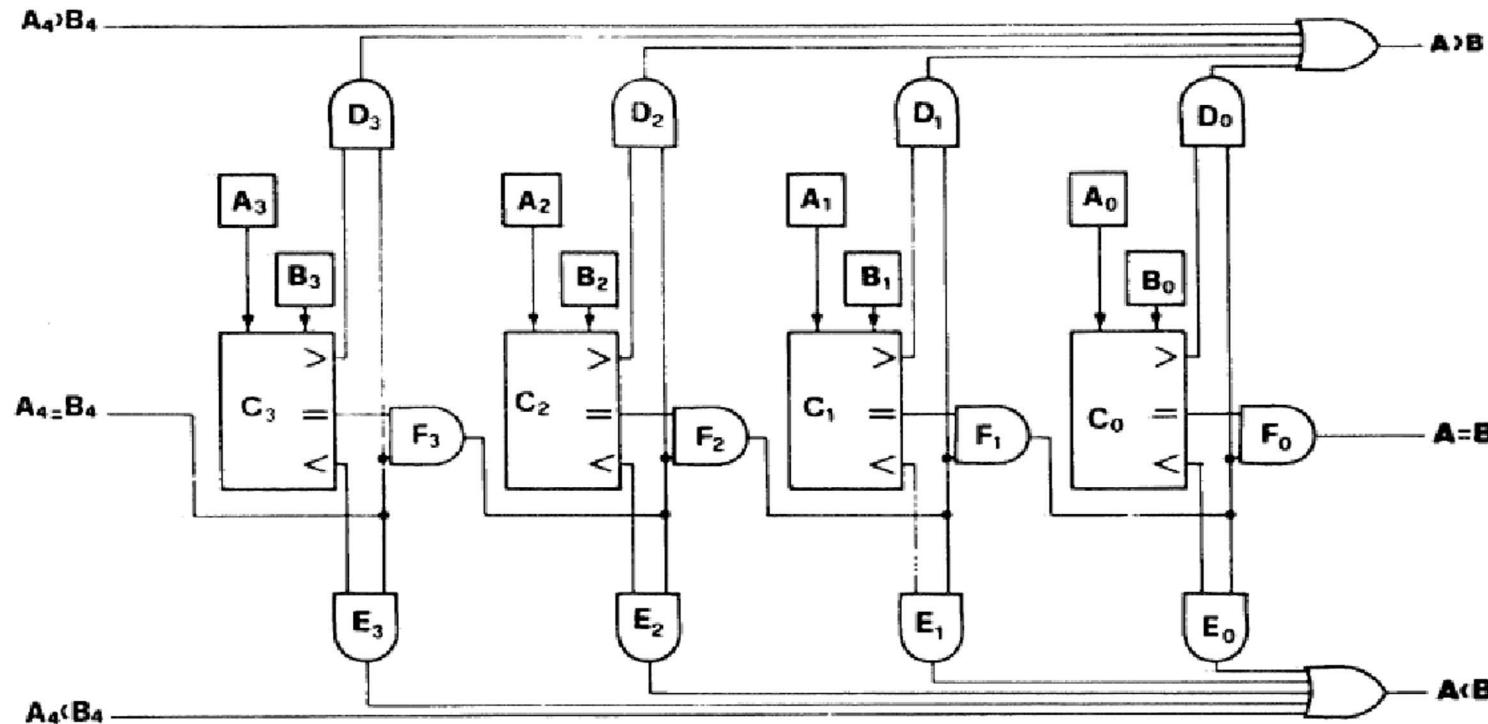
Prima confronto 2 con 1
Ok

21 > 27 ??

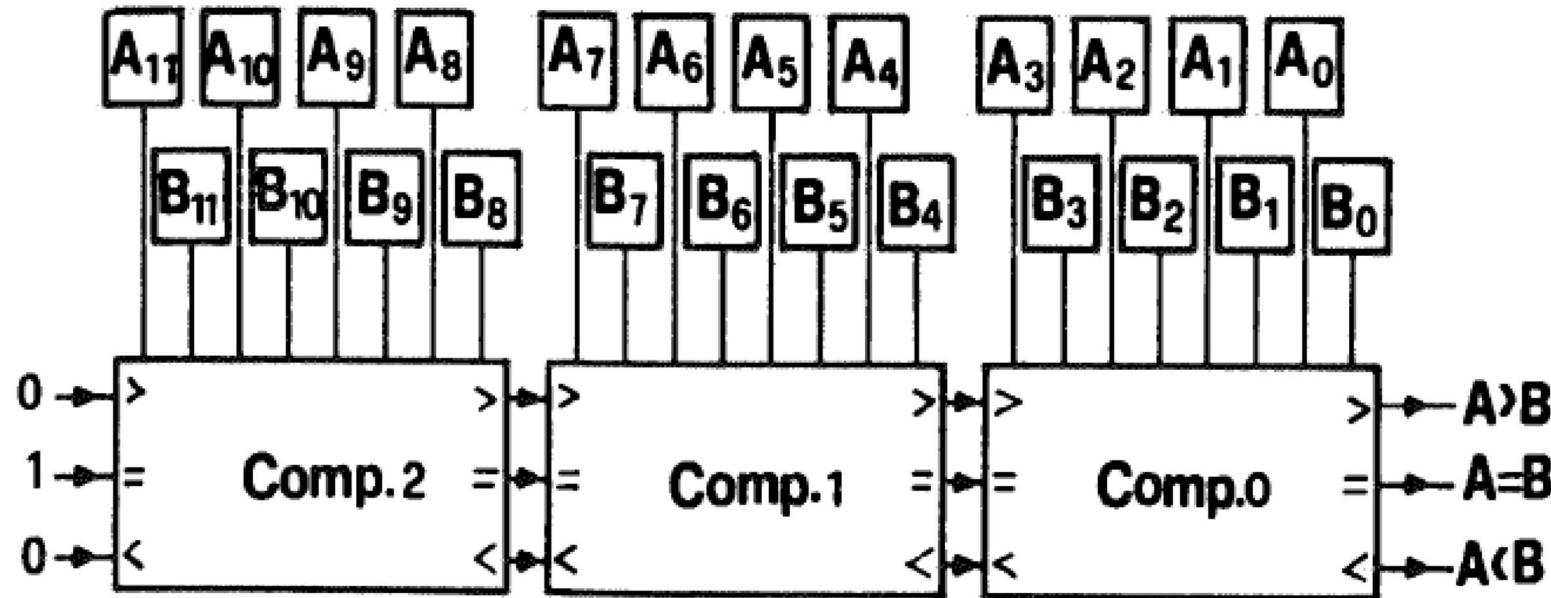
Prima confronto 2 con 2,
poi 1 con 7

Input per
collegamento
a cascata

Comparatore completo a 4 bit (2)



Comparatore completo a 12 bit



ALU: Arithmetic Logic Unit

- Abbiamo visto varie distinte operazioni logiche e matematiche implementate in circuiti differenti
- La ALU permette di avere molte di esse in una unica unità
- La ALU costituisce il cuore della maggior parte dei sistemi informatici

ALU: Arithmetic Logic Unit

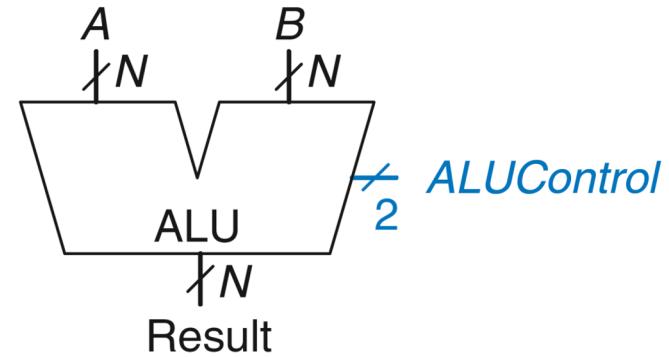
Operazioni che ALU tipicamente esegue:

- Addizione
- Sottrazione
- AND
- OR

ALU: Arithmetic Logic Unit

| ALUControl _{1:0} | Function |
|---------------------------|----------|
| 00 | Add |
| 01 | Subtract |
| 10 | AND |
| 11 | OR |

Funzioni realizzate



Simbolo

Idea di base: uso di un multiplexer (MUX)

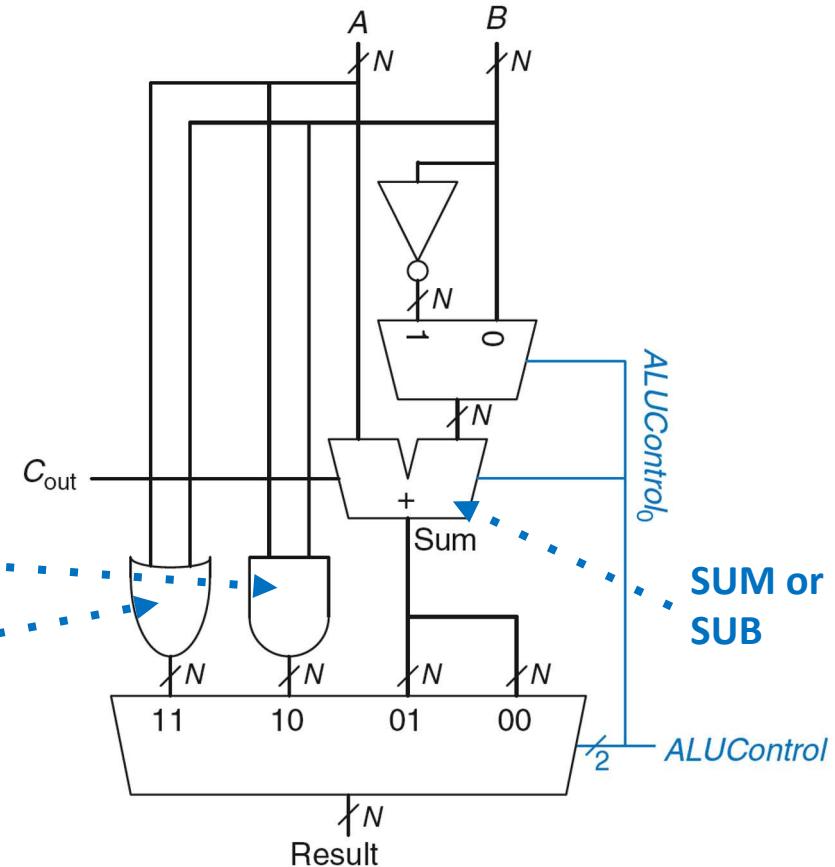
ALU: Arithmetic Logic Unit

| ALUControl _{1:0} | Function |
|---------------------------|----------|
| 00 | Add |
| 01 | Subtract |
| 10 | AND |
| 11 | OR |

Un unico circuito che contemporaneamente fa più cose

AND

OR

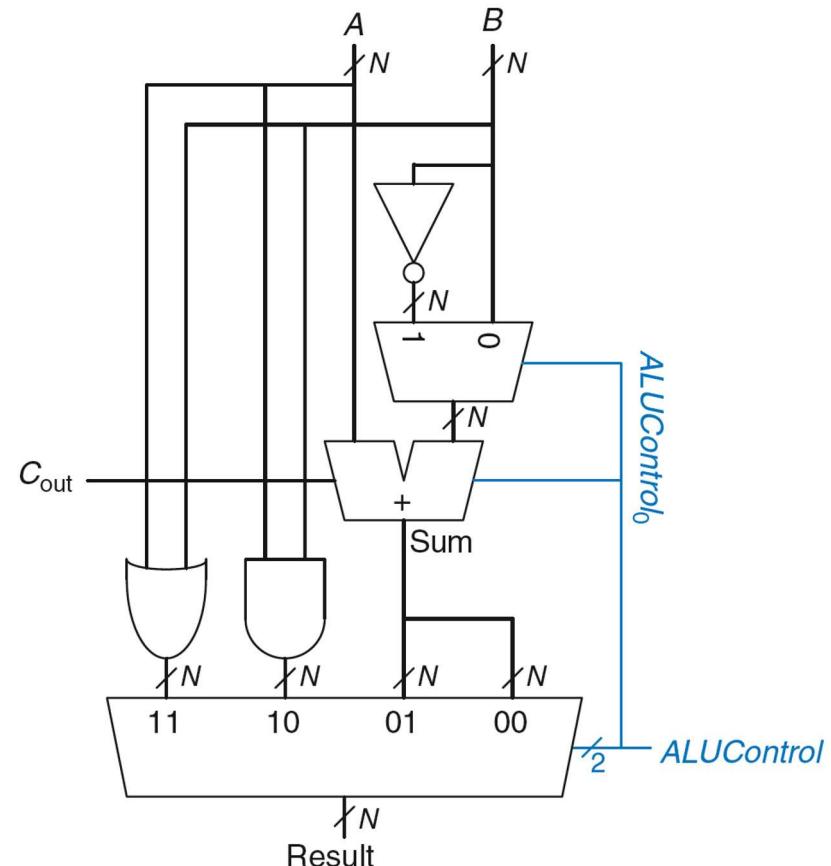


ALU: Arithmetic Logic Unit

| ALUControl _{1:0} | Function |
|---------------------------|----------|
| 00 | Add |
| 01 | Subtract |
| 10 | AND |
| 11 | OR |

Esempio: A OR B

$ALUControl_{1:0} = 11$



ALU: Arithmetic Logic Unit

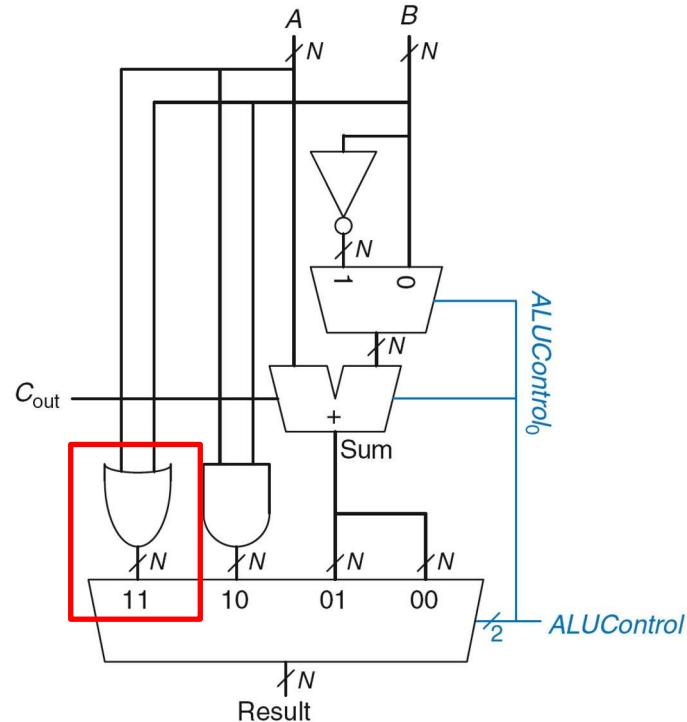
| ALUControl _{1:0} | Function |
|---------------------------|----------|
| 00 | Add |
| 01 | Subtract |
| 10 | AND |
| 11 | OR |

Esempio: A OR B

$ALUControl_{1:0} = 11$

Mux seleziona l'output della porta OR

$\text{Result} = A \text{ OR } B$



ALU: Arithmetic Logic Unit

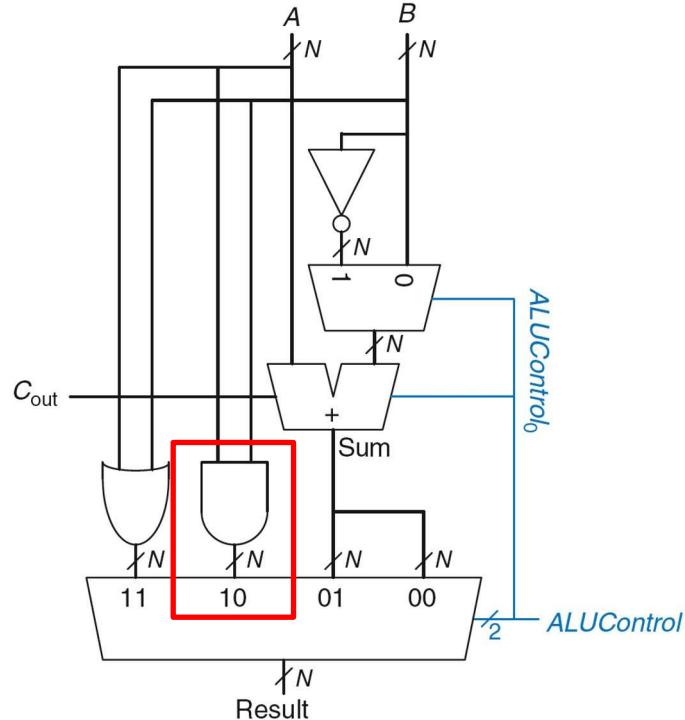
| ALUControl _{1:0} | Function |
|---------------------------|----------|
| 00 | Add |
| 01 | Subtract |
| 10 | AND |
| 11 | OR |

Esempio: A AND B

$ALUControl_{1:0} = 10$

Mux seleziona l'output della porta AND

$Result = A \text{ AND } B$

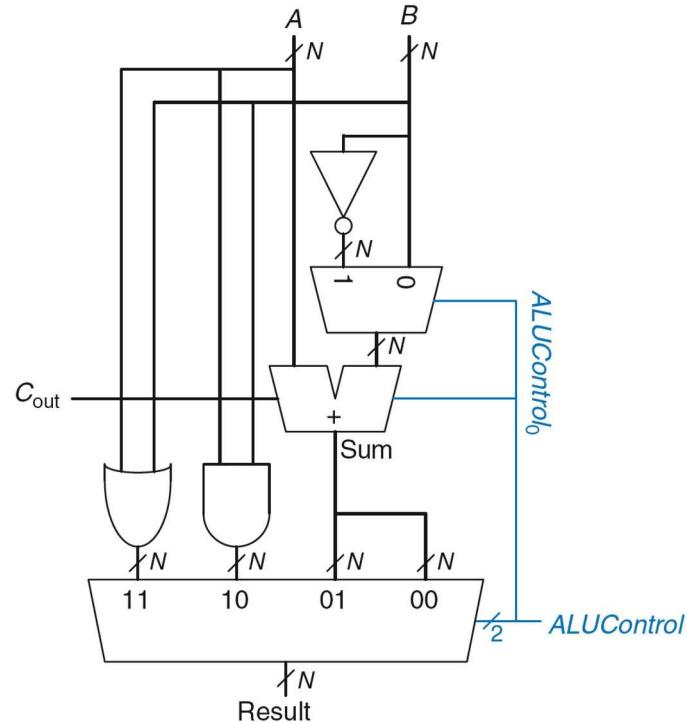


ALU: Arithmetic Logic Unit

| ALUControl _{1:0} | Function |
|---------------------------|----------|
| 00 | Add |
| 01 | Subtract |
| 10 | AND |
| 11 | OR |

Esempio: $A + B$

$ALUControl_{1:0} = 00$



ALU: Arithmetic Logic Unit

| ALUControl _{1:0} | Function |
|---------------------------|----------|
| 00 | Add |
| 01 | Subtract |
| 10 | AND |
| 11 | OR |

Esempio: $A + B$

$ALUControl_{1:0} = 00$

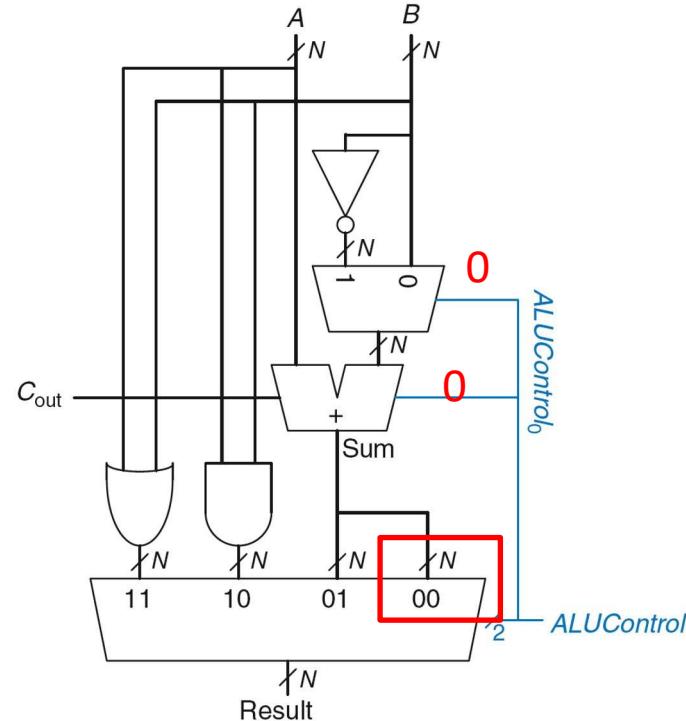
$ALUControl_0 = 0$, quindi:

$Cin = 0$

il 2nd input dell'adder è B

Mux seleziona *Sum* come *Result*

$Result = A + B$



ALU: Arithmetic Logic Unit

| ALUControl _{1:0} | Function |
|---------------------------|----------|
| 00 | Add |
| 01 | Subtract |
| 10 | AND |
| 11 | OR |

Esempio: $A - B = A + (-B)$

$ALUControl_{1:0} = 01$

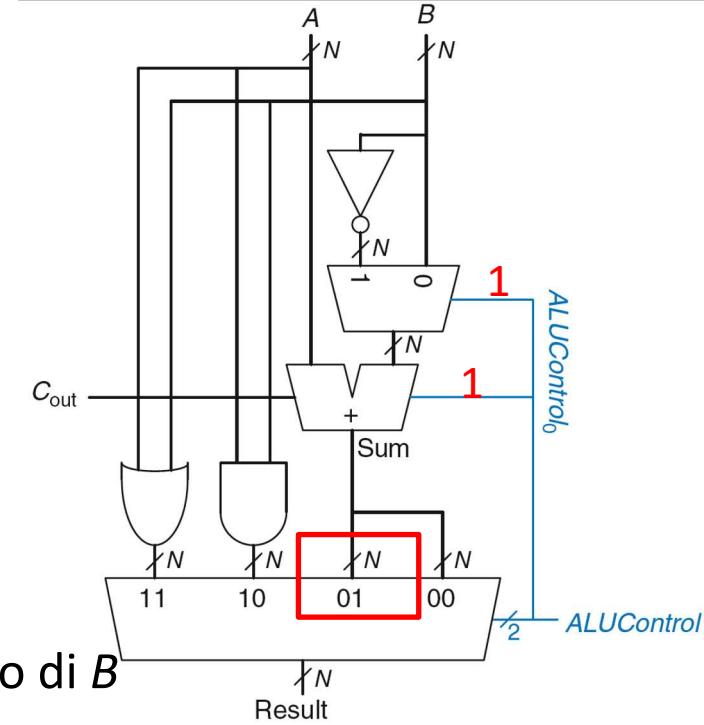
$ALUControl_0 = 1$, quindi:

Cin = 1

2nd input dell'adder è il complemento di B

Mux seleziona Sum come Result

Result = A - B



Architettura degli Elaboratori

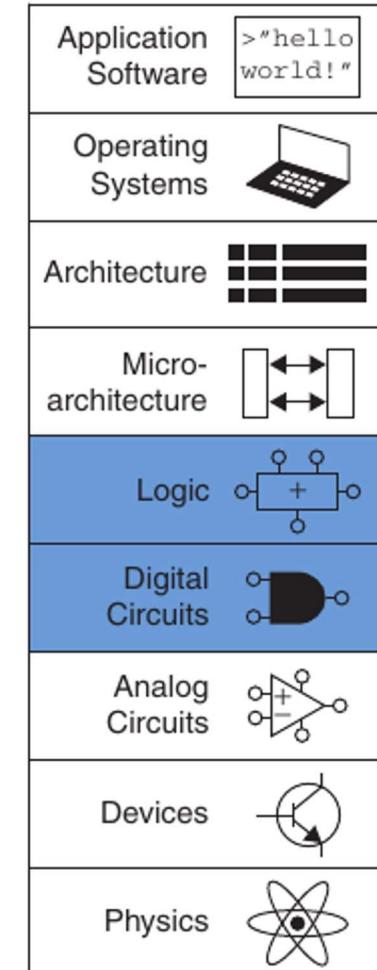
Lezione 25

Docente: R.Prevete
a.a. 2022/2023
15 maggio 2023

Logica sequenziale

- **Di cosa parleremo**

- Ancora ALU
- Un serie di “piccoli” circuiti sequenziali più o meno complessi:
 - shifter
 - counter
 - shift register

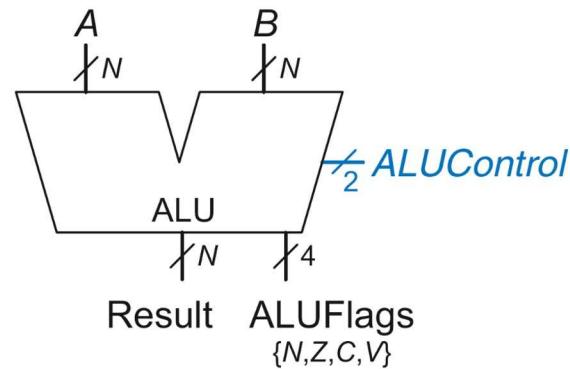


ALU con flags di stato

Le ALU possono fornire ulteriori output.

Servono a controllare:

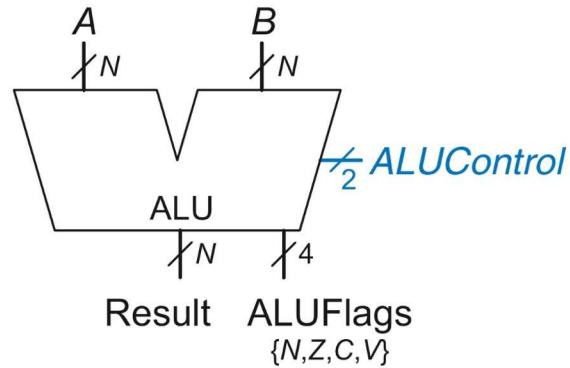
- overflow
- riporto
- output positivo
- output negativo



Tali output sono detti flags di stato

ALU con flags di stato

| Flag | Description |
|------|---------------------------|
| N | <i>Result</i> is Negative |
| Z | <i>Result</i> is Zero |
| C | Adder produces Carry out |
| V | Adder oVerflowed |

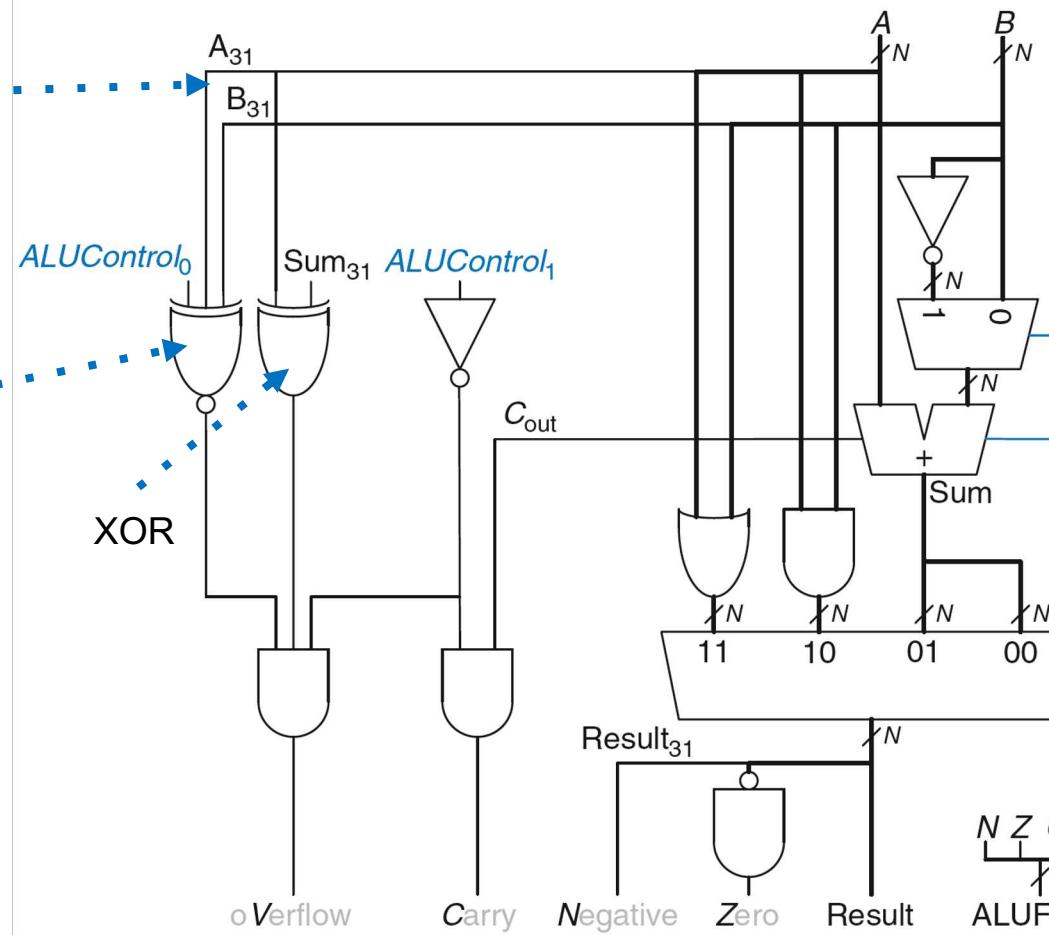


ALU con flags di stato

Bit più significativo

XOR negato

XOR

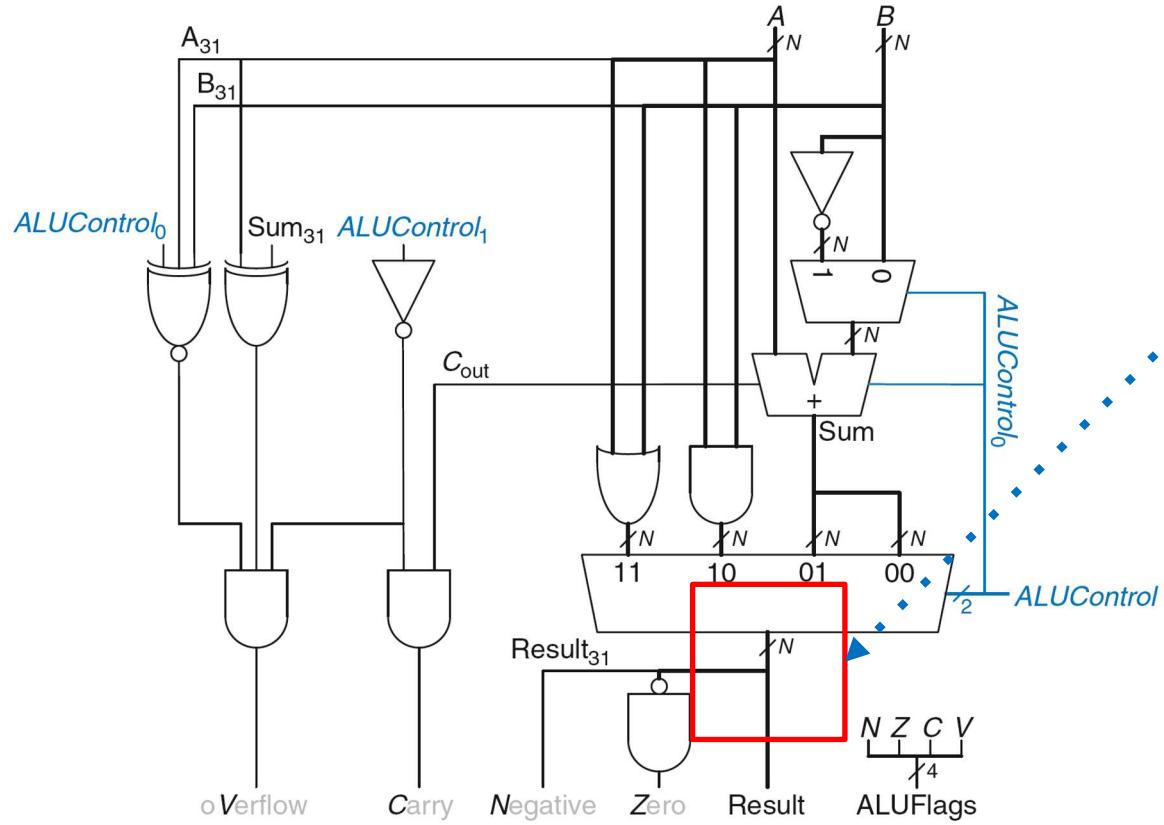


Ricorda:

In una rappresentazione complemento a 2, il bit più significativo è 1 se il numero è negativo, 0 altrimenti

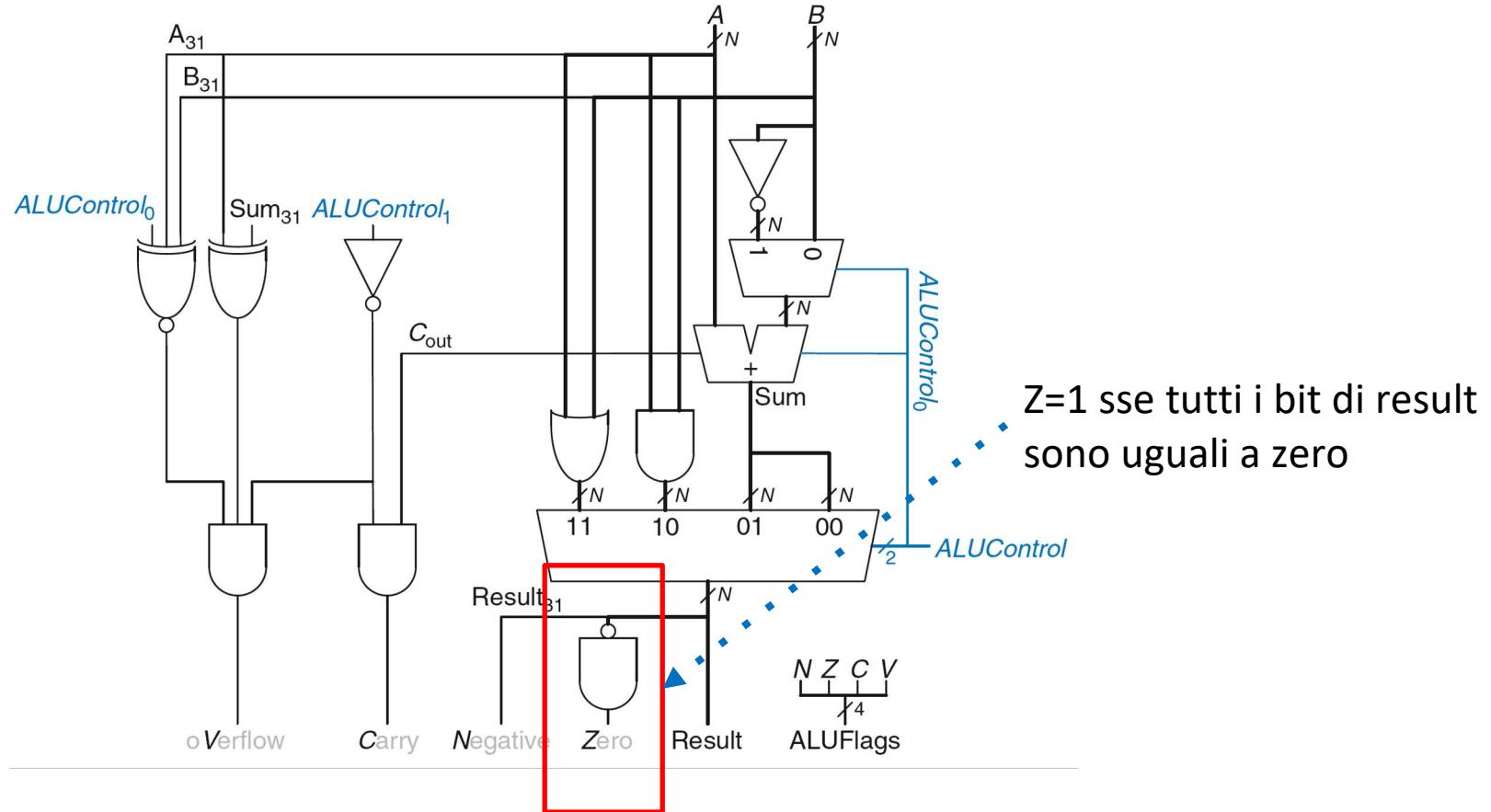
| ALUControl ₁ : | Function |
|---------------------------|----------|
| 00 | Add |
| 01 | Subtract |
| 10 | AND |
| 11 | OR |

ALU con flags di stato: Negative

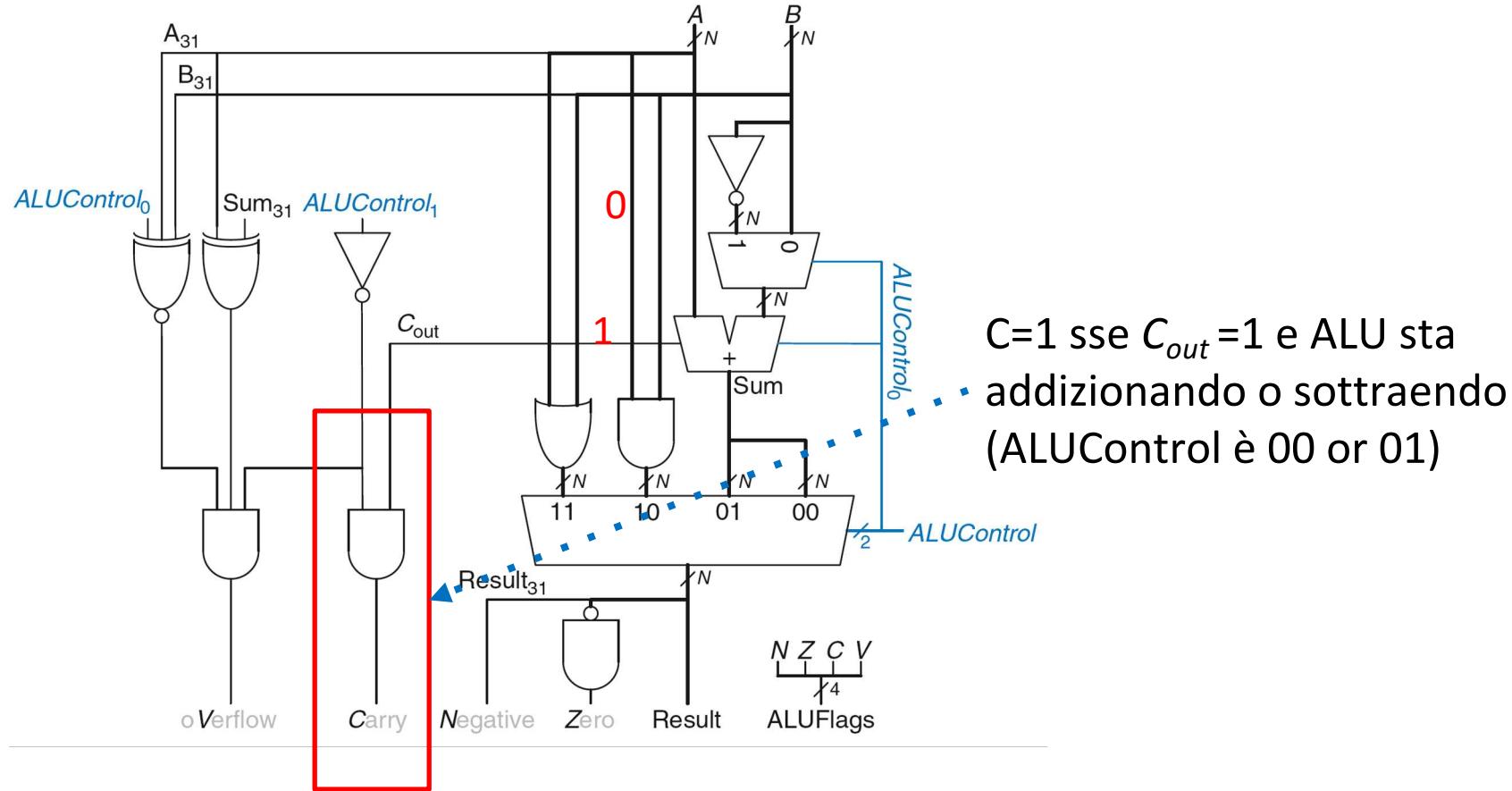


La linea N è 1 se Result negativo. Quindi, nella rappresentazione a complemento a 2, N risulta essere il bit più significativo

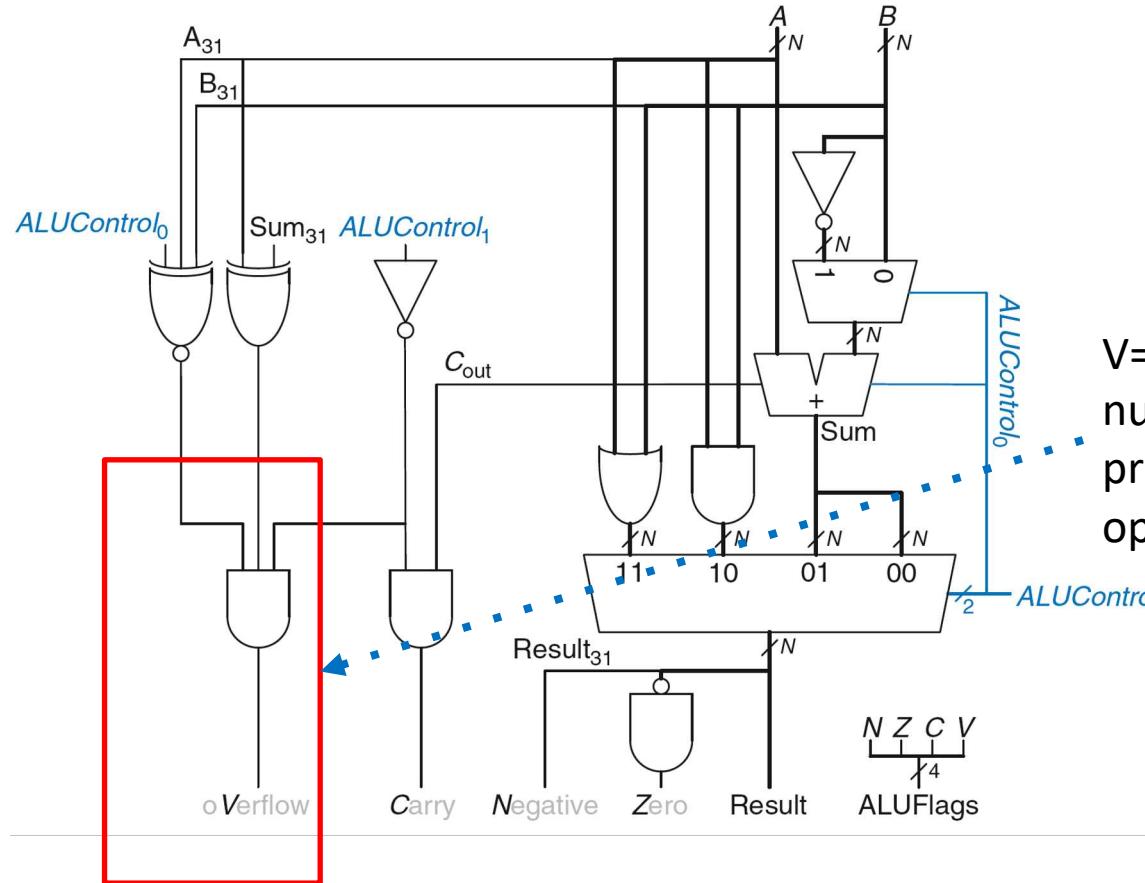
ALU con flags di stato: Zero



ALU con flags di stato: Carry



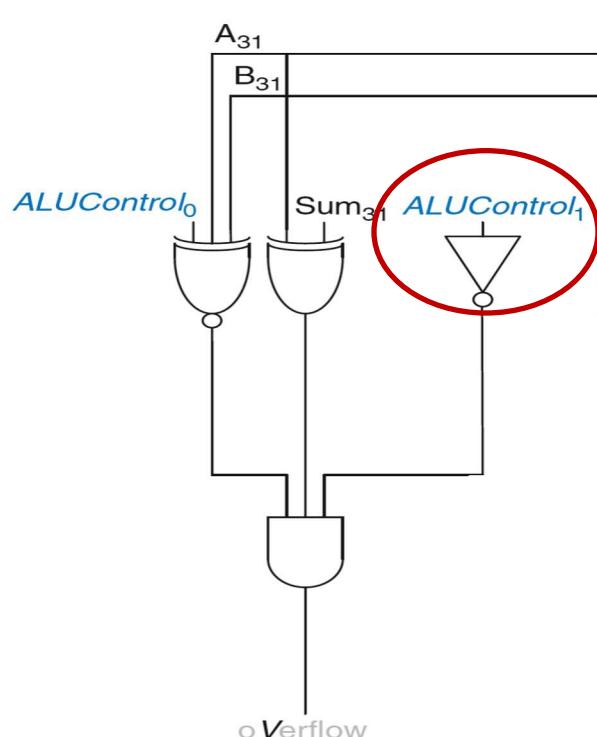
ALU con flags di stato: Overflow (1)



$V=1$ sse la somma di due numeri con lo stesso segno produce un numero di segno opposto

Vediamo più nel dettaglio...

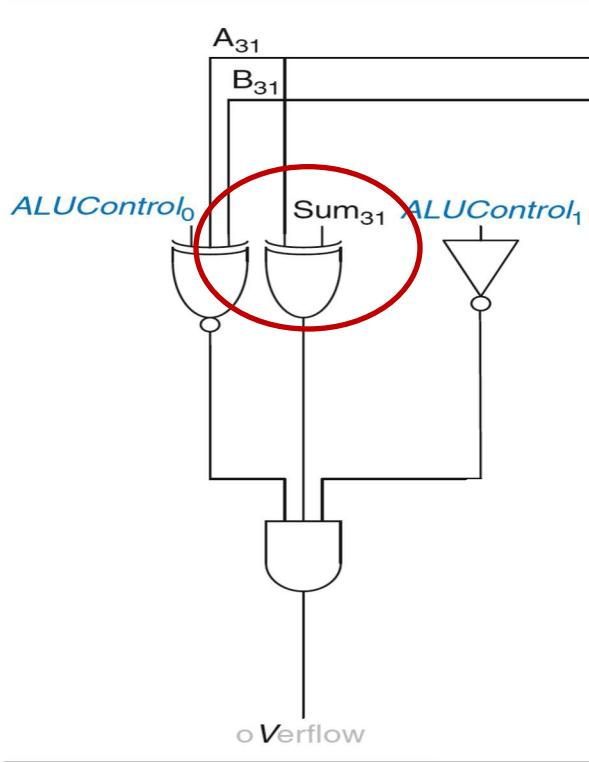
ALU con flags di stato: Overflow (2)



$V = 1$ sse:

ALU esegue una addizione o sottrazione
($ALUControl_1 = 0$)

ALU con flags di stato: Overflow (2)



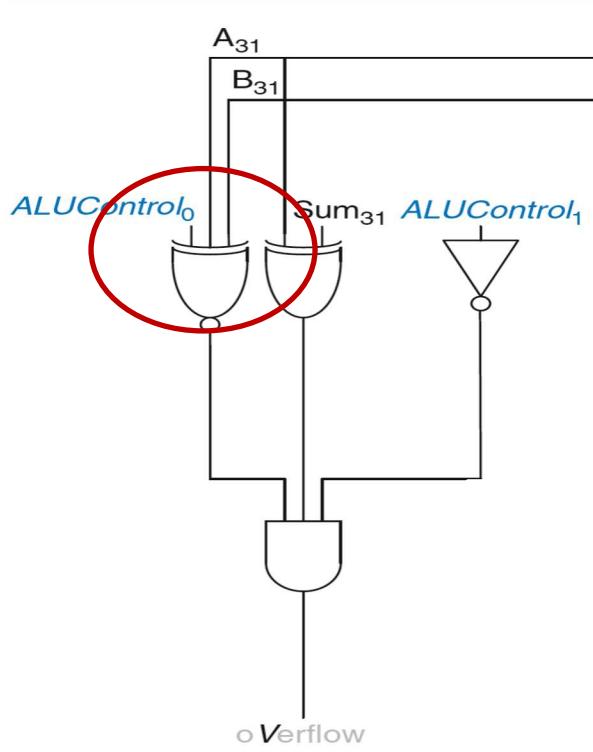
V = 1 sse:

ALU esegue una addizione o sottrazione
($ALUControl_1 = 0$)

AND

A e Sum hanno segno opposto

ALU con flags di stato: Overflow (2)



$V = 1$ sse:

ALU esegue una addizione o sottrazione
($ALUControl_1 = 0$)

AND

A e Sum hanno segno opposto

AND

A e B hanno lo stesso segno nel caso di
addizione ($ALUControl_0 = 0$) **OR**

A e B hanno segni differenti nel caso
sottrazione ($ALUControl_0 = 1$)

Shifters

Logical shifter: trasla i bit a sinistra o a destra e riempie gli spazi vuoti con degli 0

Arithmetic shifter: come il logical shifter a sinistra, nella traslazione a destra invece riempie gli spazi vuoti con il bit più significativo (msb)

Rotator: ruota i bits in cerchio, i bits che “escono” da un lato rientrano dall’ “altro”

Shifters

Logical shifter:

- Ex: 11001 >> 2 = 00110 right shifter
- Ex: 11001 << 2 = 00100 left shifter

Arithmetic shifter:

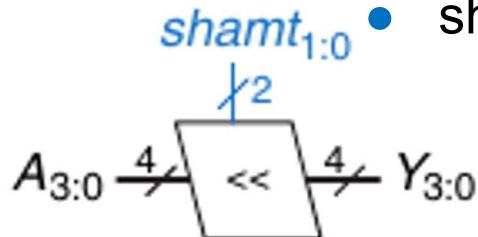
- Ex: 11001 >>> 2 = 11110 right (spazi vuoti con msb)
- Ex: 11001 <<< 2 = 00100 left (come sopra)

Rotator:

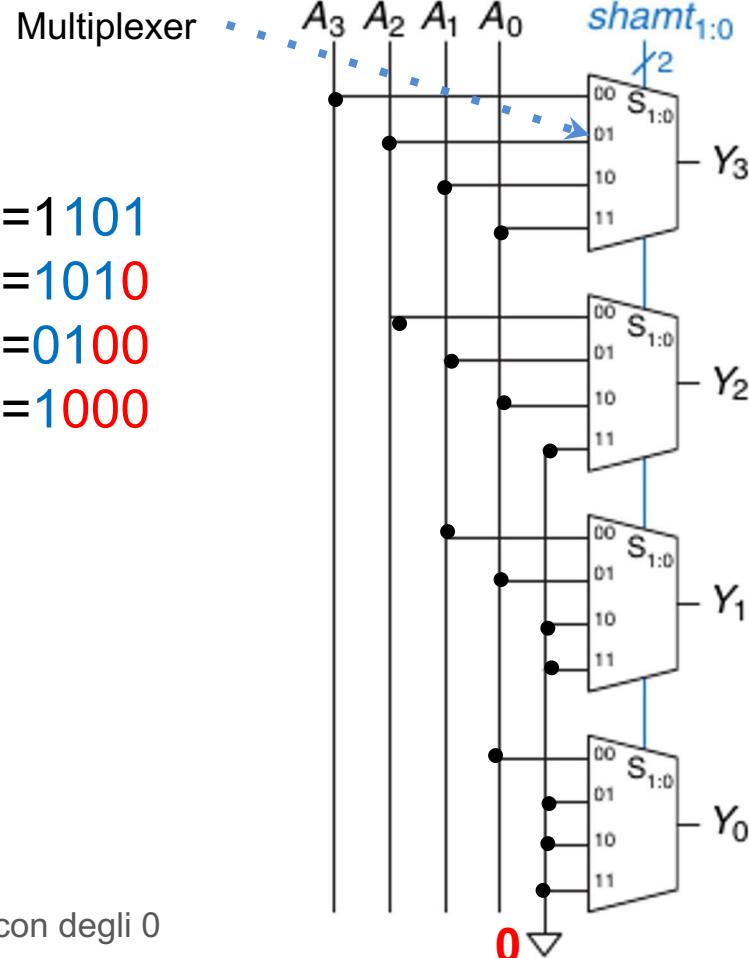
- Ex: 11001 ROR 2 = 01110 right rotator
- Ex: 11001 ROL 2 = 00111 left rotator

4-bit left logical Shifter

$$A_3 A_2 A_1 A_0 = 1\textcolor{blue}{1}01$$



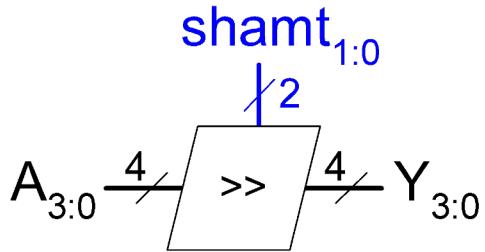
- sham_{t_{1:0}}=00 Y₃Y₂Y₁Y₀=1\textcolor{blue}{1}01
- sham_{t_{1:0}}=01 Y₃Y₂Y₁Y₀=\textcolor{red}{1}010
- sham_{t_{1:0}}=10 Y₃Y₂Y₁Y₀=\textcolor{blue}{0}100
- sham_{t_{1:0}}=11 Y₃Y₂Y₁Y₀=\textcolor{red}{1}000



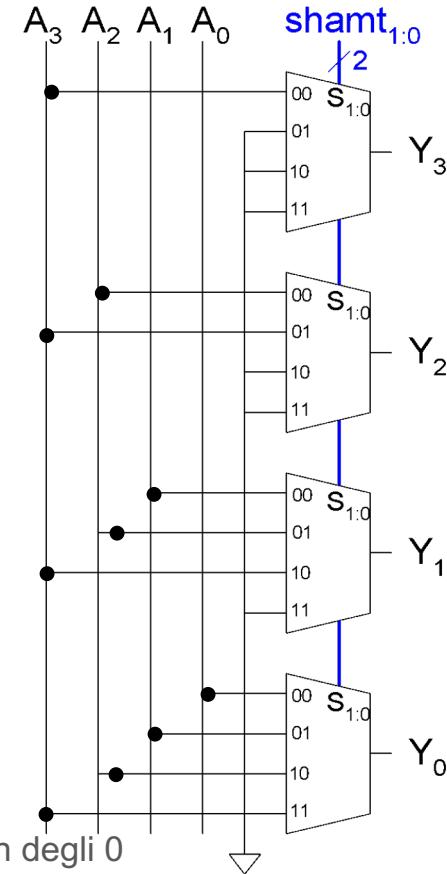
shamt= *shift amount* (*di quanti bit devo muovermi*) .

Logical shifter: trasla i bit a sinistra o a destra e riempie gli spazi vuoti con degli 0

4-bit right logical Shifter



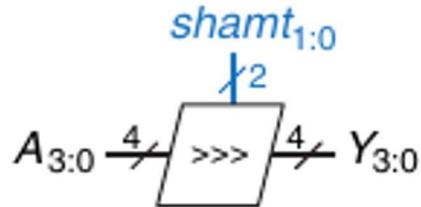
shamt= *shift amount (di quanti bit devo muovermi)* .



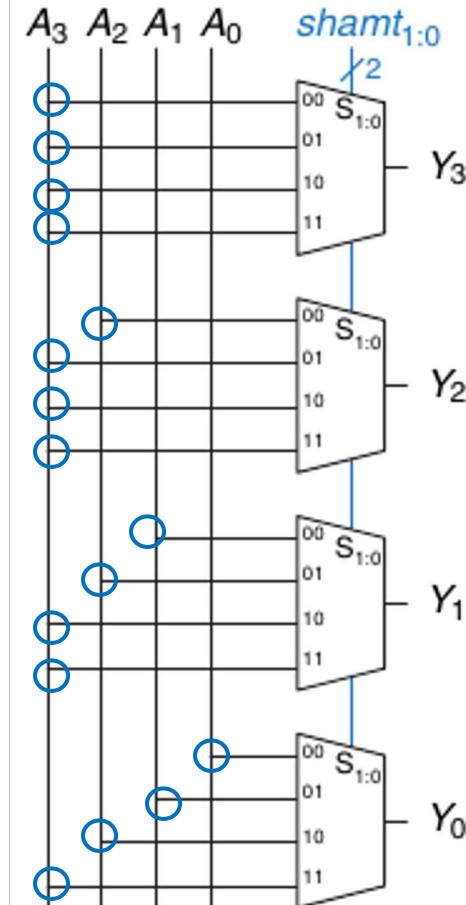
Logical shifter: trasla i bit a sinistra o a destra e riempie gli spazi vuoti con degli 0

4-bit right Arithmetic shifter:

$$A_3 A_2 A_1 A_0 = 1 \textcolor{red}{1} \textcolor{green}{0} 1$$



- $shamt_{1:0}=00 \quad Y_3 Y_2 Y_1 Y_0 = 1 \textcolor{red}{1} \textcolor{green}{0} 1$
- $shamt_{1:0}=01 \quad Y_3 Y_2 Y_1 Y_0 = 11 \textcolor{blue}{1} 0$
- $shamt_{1:0}=10 \quad Y_3 Y_2 Y_1 Y_0 = 111 \textcolor{red}{1}$
- $shamt_{1:0}=11 \quad Y_3 Y_2 Y_1 Y_0 = 1111$



Arithmetic shifter: come il logical shifter a sinistra, nella traslazione a destra invece riempie gli spazi vuoti con il bit più significativo (msb)

Shifters as Multipliers, Dividers

- $A \lll N = A \times 2^N$
 - **Example:** $00001 \ll 2 = 00100$ ($1 \times 2^2 = 4$)
 - **Example:** $11101 \ll 2 = 10100$ ($-3 \times 2^2 = -12$)
- $A \ggg N = A \div 2^N$
 - **Example:** $01000 \ggg 2 = 00010$ ($8 \div 2^2 = 2$)
 - **Example:** $10000 \ggg 2 = 11100$ ($-16 \div 2^2 = -4$)

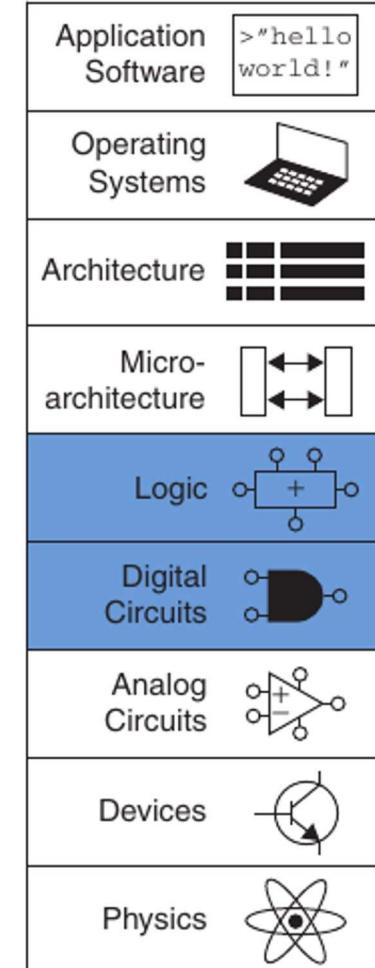
Architettura degli Elaboratori

Lezione 26

Docente: R.Prevete
a.a. 2022/2023
17 maggio 2023

Logica sequenziale

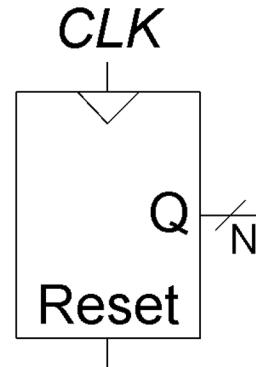
- Di cosa parleremo
 - Counters
 - Shift register
 - **Memorie**



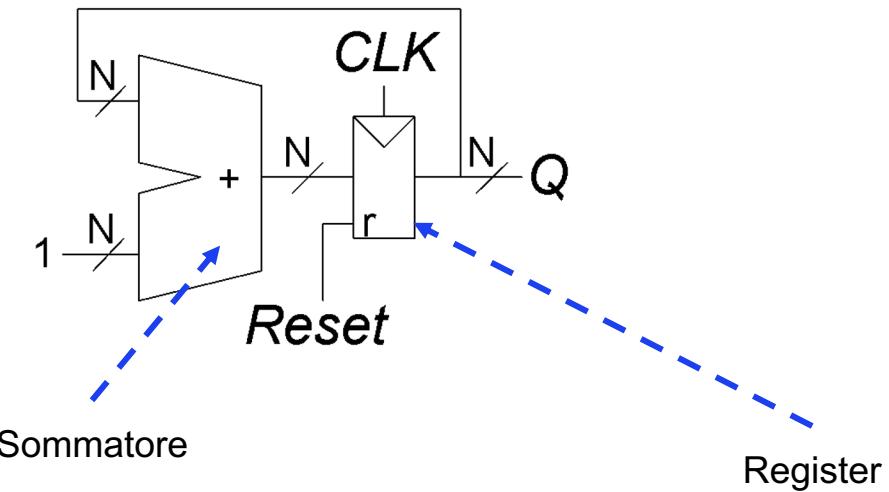
Counters

Incrementa ad ogni clock

Symbol

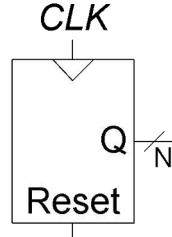
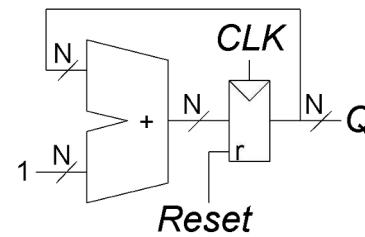


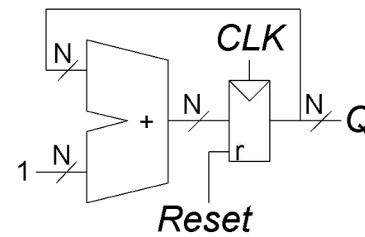
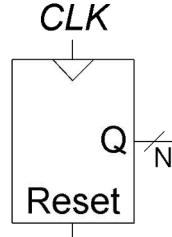
Implementation



Counters

- Usato per
 - eseguire cicli su numeri:
000, 001, 010, 011, 100, 101, 110, 111, 000, 001...
 - Come orologio digitale
 - Program counter: tiene traccia della istruzione corrente da eseguire (registro prossima istruzione)

| Symbol | Implementation |
|--|---|
|  |  |



Shift Registers

converte un input seriale (S_{in}) in un output parallelo ($Q_{0:N-1}$)

| | t-3 | t-2 | t-1 | t |
|-----------------------|------------|------------|------------|----------|
| S_{in} | 1 | 1 | 0 | 1 |



| t | | | |
|----------------------|----------------------|----------------------|----------------------|
| Q₃ | Q₂ | Q₁ | Q₀ |
| 1 | 1 | 0 | 1 |

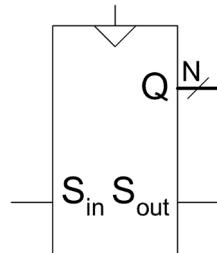
Shift Registers

Trasla di un bit ad ogni clock

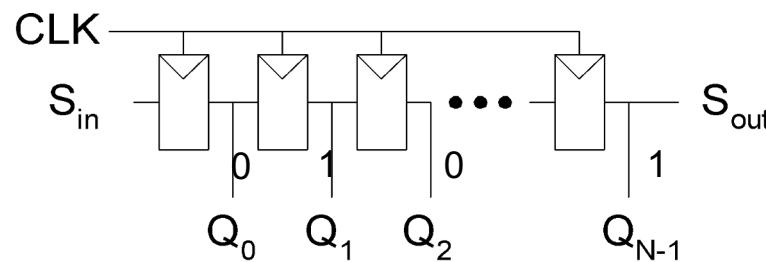
Ritorna un bit in uscita (S_{out}) ad ogni clock

Serial-to-parallel converter: converte un input seriale (S_{in}) in un output parallelo ($Q_{0:N-1}$)

Symbol:



Implementation:

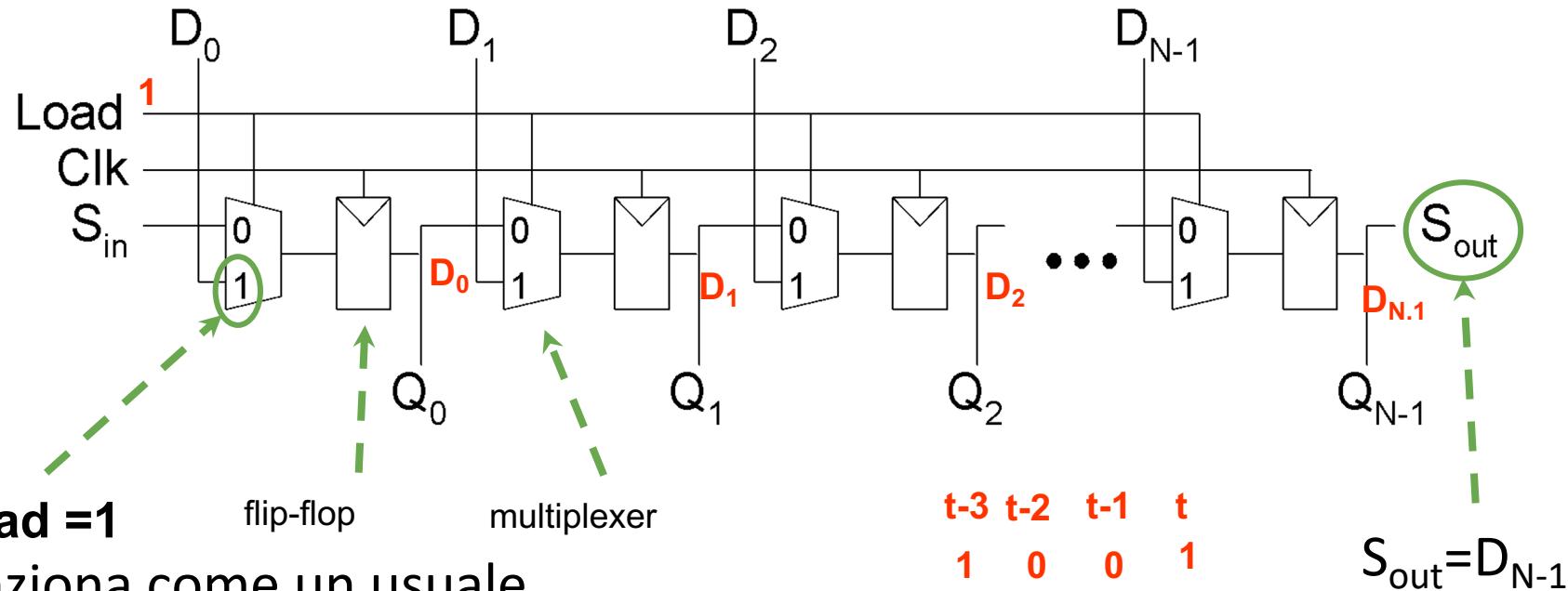


Shift Register con load parallelo

Vogliamo costruire un circuito che possa essere utilizzato per:

- Memorizzare bit in sequenza e porli in parallelo
(convertitore sequenziale -parallelo)
- Memorizzare bit in parallelo (usuale registro) e restituirli
in output in sequenza (convertitore parallelo-sequenziale)
- Variabile di controllo **Load** (permette di cambiare da una
modalità ad un'altra)

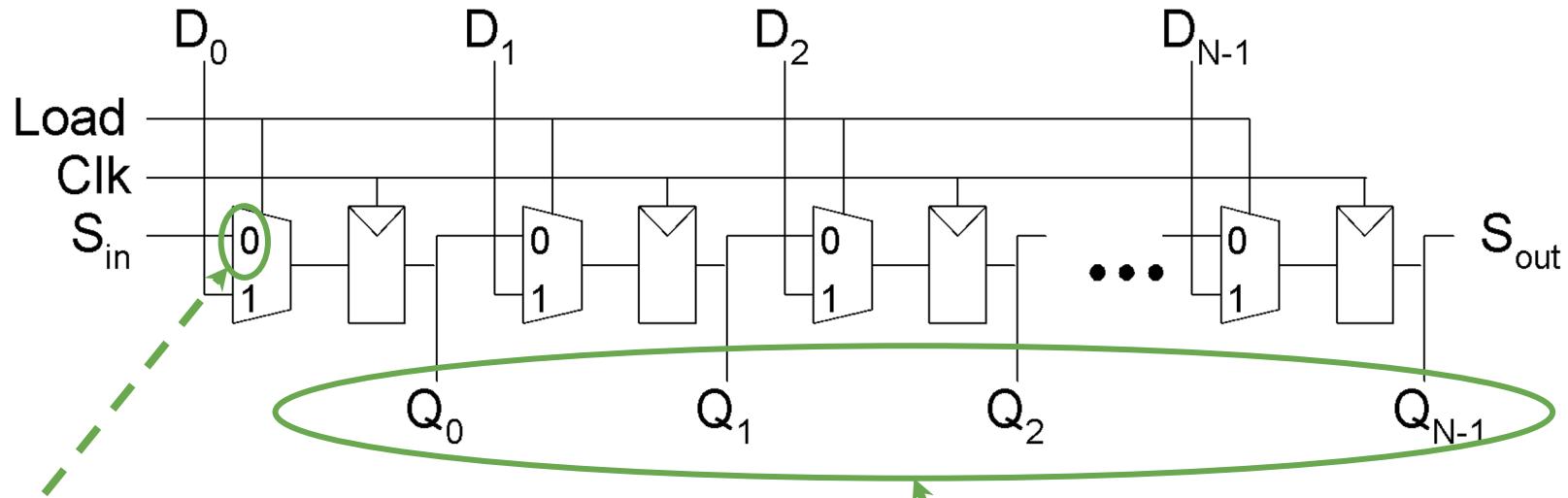
Shift Register con load parallelo



funziona come un usuale
registro a N -bit

Quando Load è rimesso a 0 gli N -bit sono
posti in sequenza in output

Shift Register con load parallelo



Load = 0

Un usuale shift register,
ricevo in input una
sequenza su S_{in}

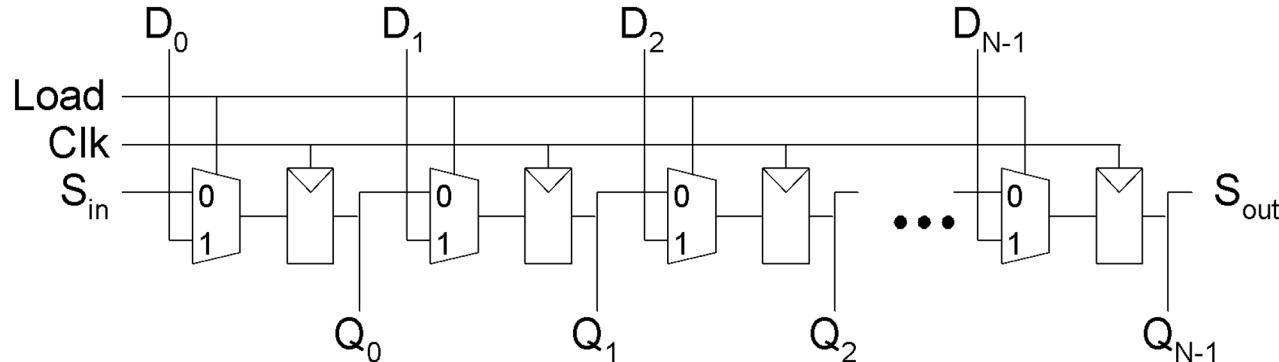
S_{in} messa in parallelo

Shift Register con load parallelo

Quando $Load = 1$, funziona come un usuale registro a N -bit

Quando $Load = 0$, agisce da shift register

Quindi può agire da convertitore *seriale-parallelo* (da S_{in} a $Q_{0:N-1}$) o da convertitore *parallelo-seriale* ($D_{0:N-1}$ to S_{out})



Parliamo ora di come memorizzare grosse quantità di dati...



Memory Arrays

Memorizzare efficacemente una grossa quantità di dati

3 tipologie:

Dynamic random access memory (DRAM)

Static random access memory (SRAM)

Read only memory (ROM)

Memory Arrays

- Memoria organizzata come array bidimensionale
- Si scrive o si legge il contenuto di una riga del array
- Ogni riga è biunivocamente associata ad un *indirizzo* (*Address*)
- Il valore letto o scritto si chiama *Dato* (*Data*)

| Address | Data |
|---------|-------|
| 11 | 0 1 0 |
| 10 | 1 0 0 |
| 01 | 1 1 0 |
| 00 | 0 1 1 |

Memory Arrays

- Un array con indirizzi di N-bit e dati di M-bit ha 2^N righe e M colonne
- Ciascuna riga dei dati è detta parola (word)
- Così l'array contiene 2^N parole di M bit

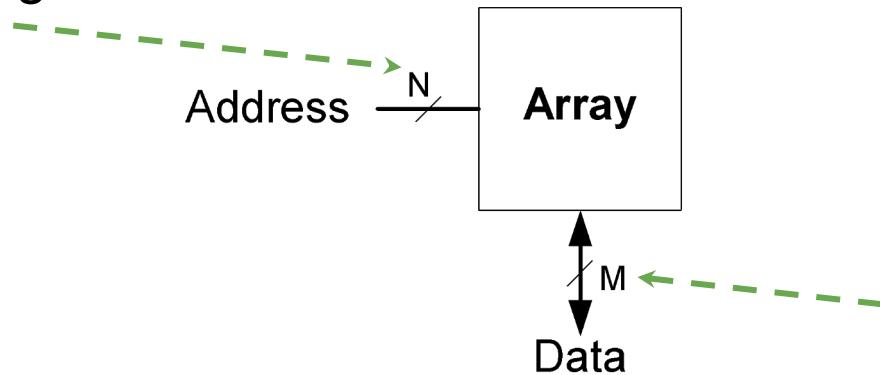
| Address | Data |
|---------|-------|
| 11 | 0 1 0 |
| 10 | 1 0 0 |
| 01 | 1 1 0 |
| 00 | 0 1 1 |

N=2

M=3

Memory Arrays

Indirizzi di N bit e
 2^N righe



Tipicamente
 $M=8$ (un byte) e
 $N=32$ (o 64)

Dati con M bit

Simbolo

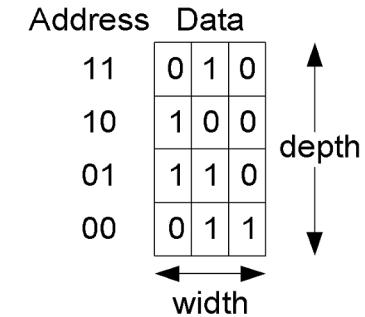
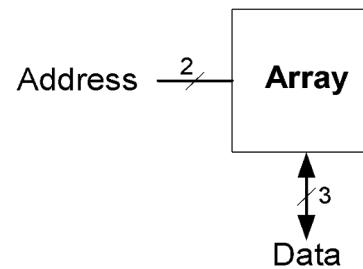
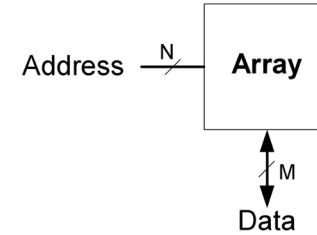
Memory Arrays

Con N bit di indirizzo e M bits data
(quindi 2^N righe e a M colonne), parliamo anche
di:

Depth: numero di righe (parole)

Width: numero di colonne (lunghezza di una
parola)

Dimensione Array : depth \times width = $2^N \times M$



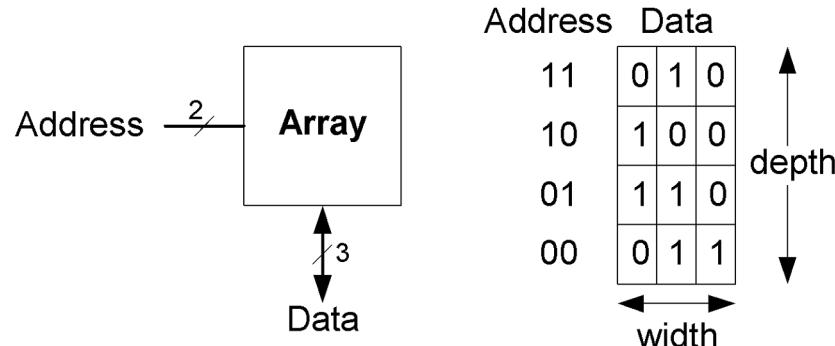
Esempio

$2^2 \times 3$ -bit array

Numero parole: 4

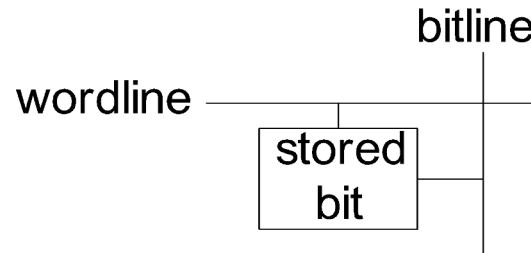
Lunghezza parola: 3-bits

All'indirizzo 10 corrisponde la parola 100

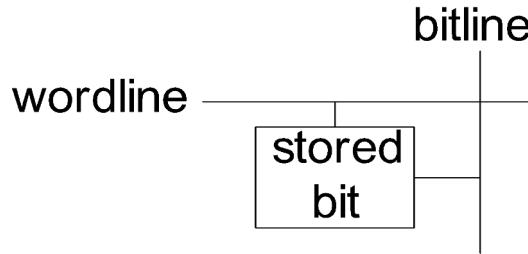


Memory Array Bit Cells

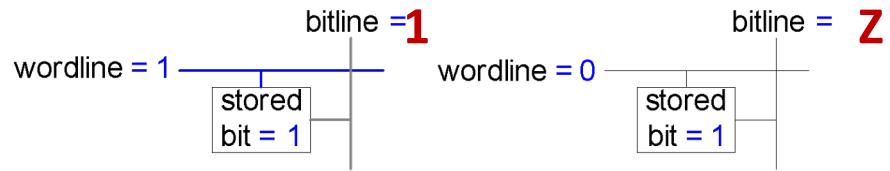
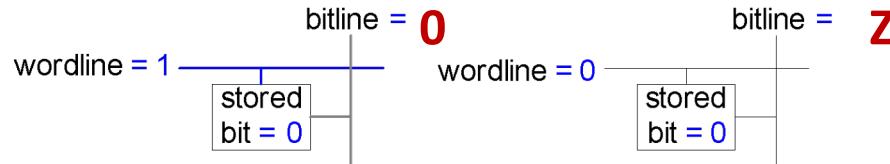
- Gli array di memoria (memory array) sono costruiti come un array di celle di bit (Bit Cells), ognuna delle quali memorizza 1 bit.
- ogni cella di bit è collegata a una **wordline** e una **bitline**.
 - La **wordline** agisce come un **enable**
 - Quando la wordline è 1, il bit è trasferito dalla bitline alla bit-cell (scrittura) o dalla bit-cell all bitline (lettura)



Memory Array Bit Cells



- Il circuito per memorizzare il bit varia con il tipo di memoria
- Per leggere una cella di bit,
 - (b) la linea di bit viene inizialmente lasciata “volante” (Z)
 - (a) Poi quando la wordline diventa 1, il bit memorizzato è “passato” alla bitline



(a)

(b)



Memory Array

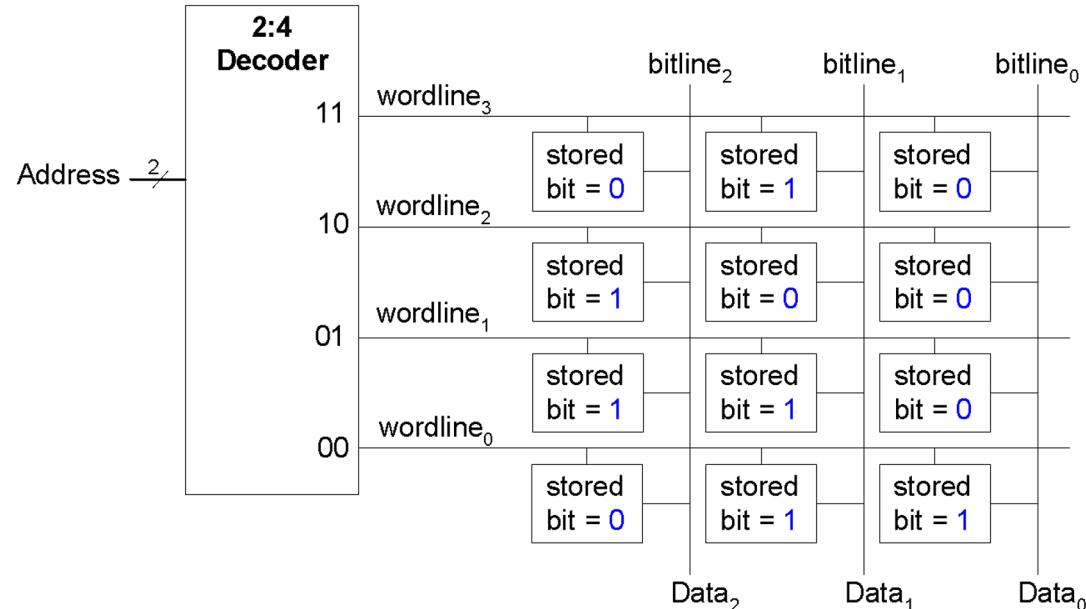
Wordline:

Agisce come un enable

Seleziona una riga nella memoria

Corrisponde ad un unico indirizzo

Solo una wordline per volta è attivata



Tipi di memoria

Random access memory (RAM): **volatile**

Read only memory (ROM): **non volatile**

RAM: Random Access Memory

Volatile: i dati sono persi quando il computer è spento

Le operazioni di lettura e scrittura sono più veloci (rispetto alle ROM)

E' la memoria primaria di un computer (DRAM)

E' chiamata *random access memory* perché il tempo di accesso ad una parola è lo stesso per ogni indirizzo (a differenza delle memorie ad accesso sequenziale come i nastri che usavo quando giocavo con il C64)

ROM: Read Only Memory

Non volatile: mantiene i dati anche quando il computer è spento

La lettura è relativamente veloce ma la scrittura è lenta o semplicemente non è possibile scrivere

Drives, flash memory, Bios etc. sono ROMs

ROM sta per *read only memory* perché inizialmente questo tipo di memorie venivano “scritte” bruciando dei fusibili. Quindi, una volta configurate, non erano più manipolabili. Chiaramente questo non è più il caso per Flash memory e altri tipi di ROMs.

Tipi di RAM

DRAM (Dynamic random access memory)

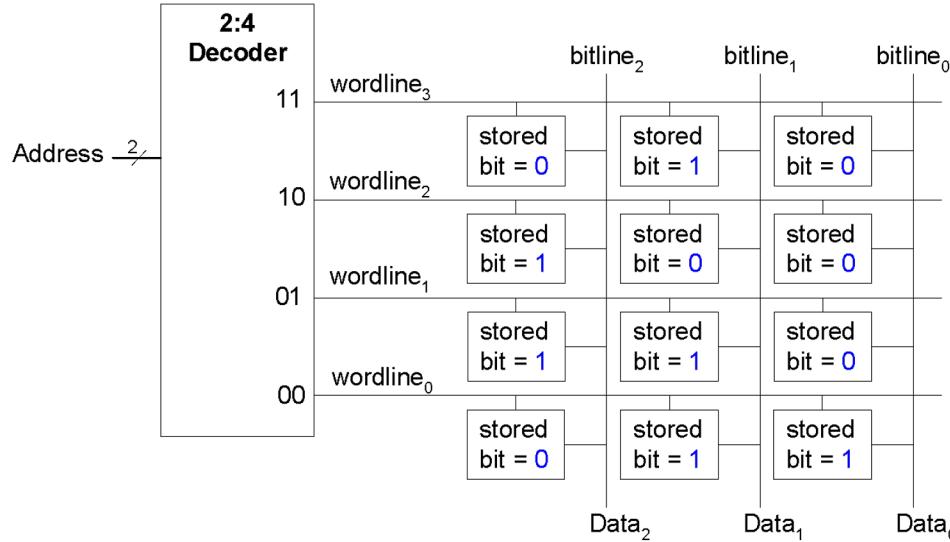
SRAM (Static random access memory)

Si differenziano per le componenti usate per memorizzare i dati:

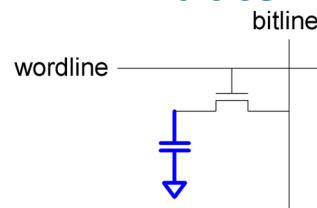
DRAM usa dei condensatori

SRAM usa invertitori (porte not)

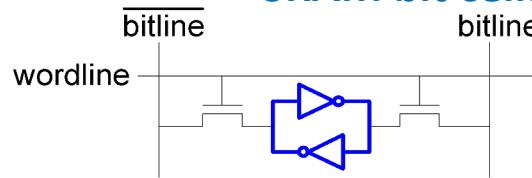
Memory Arrays: Riepilogo



DRAM bit cell:



SRAM bit cell:



DRAM vs SRAM

DRAM è più lenta

Scrittura e refresh (millisecondi) inducono un consumo maggiore di energia

DRAM, più economiche

| Memory Type | Transistors per Bit Cell | Latency |
|-------------|--------------------------|---------|
| flip-flop | ~20 | fast |
| SRAM | 6 | medium |
| DRAM | 1 | slow |

DRAM vs SRAM

DRAM è più lenta

Scrittura e refresh (millisecondi) inducono un consumo maggiore di energia

DRAM, più economiche

Possibili usi:

| Memory Type | Transistors per Bit Cell | Latency |
|-------------|--------------------------|---------|
| flip-flop | ~20 | fast |
| SRAM | 6 | medium |
| DRAM | 1 | slow |



registri

DRAM vs SRAM

DRAM è più lenta

Scrittura e refresh (millisecondi) inducono un consumo maggiore di energia

DRAM, più economiche

Possibili usi:

| Memory Type | Transistors per Bit Cell | Latency |
|-------------|--------------------------|---------|
| flip-flop | ~20 | fast |
| SRAM | 6 | medium |
| DRAM | 1 | slow |



cache

DRAM vs SRAM

DRAM è più lenta

Scrittura e refresh (millisecondi) inducono un consumo maggiore di energia

DRAM, più economiche

Possibili usi:

| Memory Type | Transistors per Bit Cell | Latency |
|-------------|--------------------------|---------|
| flip-flop | ~20 | fast |
| SRAM | 6 | medium |
| DRAM | 1 | slow |

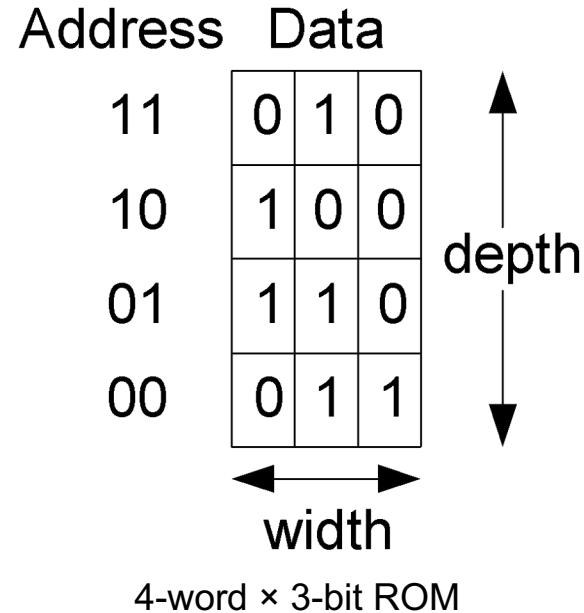
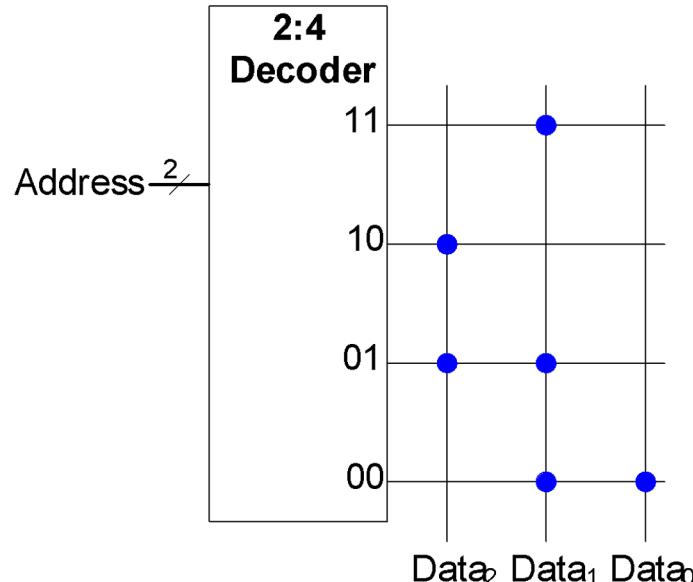


memoria centrale

Alcune precisazioni sulle Read Only Memory (ROM)

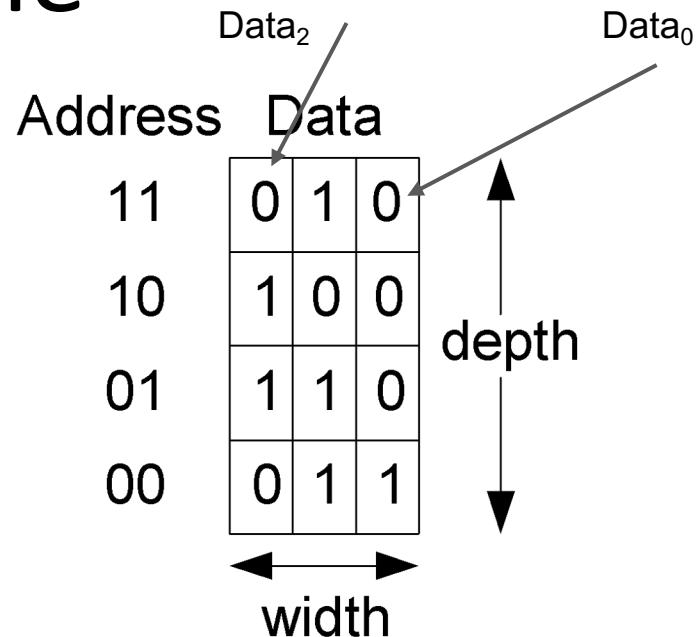
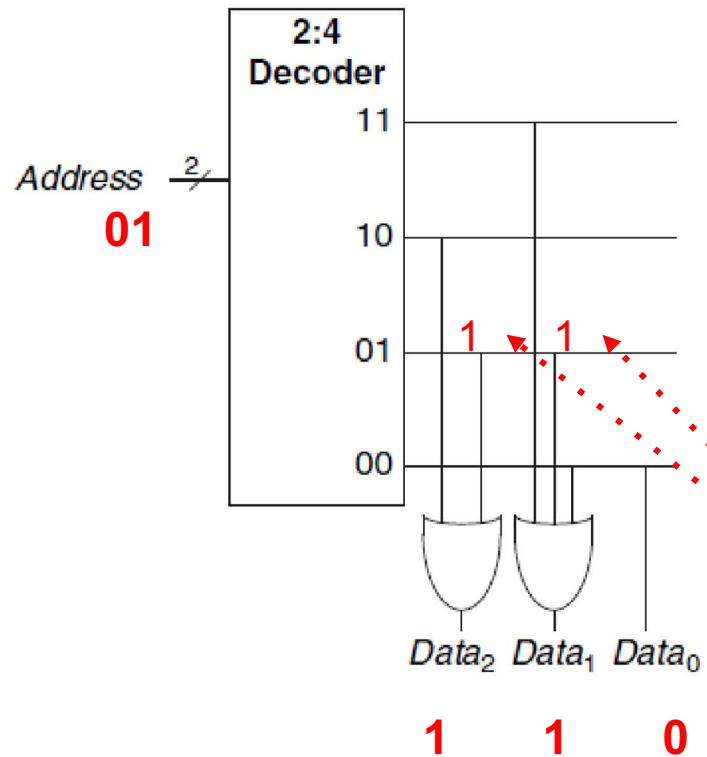
- Abbiamo detto che sono **memorie a sola lettura**
 - Ad esempio usate in un microprocessore per memorizzare una serie di istruzioni macchina elementari da non modificare.
- Il suo **contenuto** può essere rappresentato in **notazione dot**
- **Concettualmente** possono essere viste come una **logica a due livelli**
 - prima un **decoder**, poi un gruppo di **OR**

Contenuto ROM: dot notation



Un **dot** (punto) all'intersezione di una riga (wordline) e una colonna (bitline) indica che il bit di dati è 1

ROM via porte logiche



Per esempio
address **01**

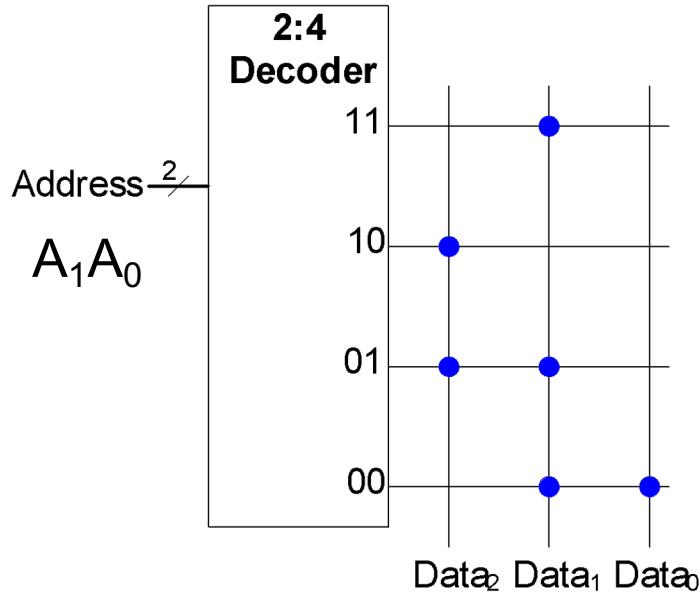
ROM possono essere programmate

- Le ROM moderne non sono realmente di sola lettura; possono essere anche programmate (scritte)
- VARI tipi:
 - Erasable PROMs (EPROMs, pronunciate “e-proms”)
 - Electrically erasable PROMs (EEPROMs, pronunciate “e-e-proms” o “double-e proms”)
 - Flash memory
- Usano meccanismi basati sulla presenza di fusibili o transistor che possono essere “bruciati”

Memorie come funzioni logiche

- Le memorie possono essere utilizzate anche per realizzare funzioni logiche
- Tali memorie sono chiamate ***lookup table***
- Ciascun indirizzo corrisponde ad una riga nella tabella di verità, e ciascun bit del contenuto della memoria (data bit) corrisponde a un valore di output
- Facciamo un esempio con una ROM ...

ROM Logic



$$Data_2 = A_1 \oplus A_0$$

$$Data_1 = \bar{A}_1 + A_0$$

$$Data_0 = \bar{A}_1 \bar{A}_0$$

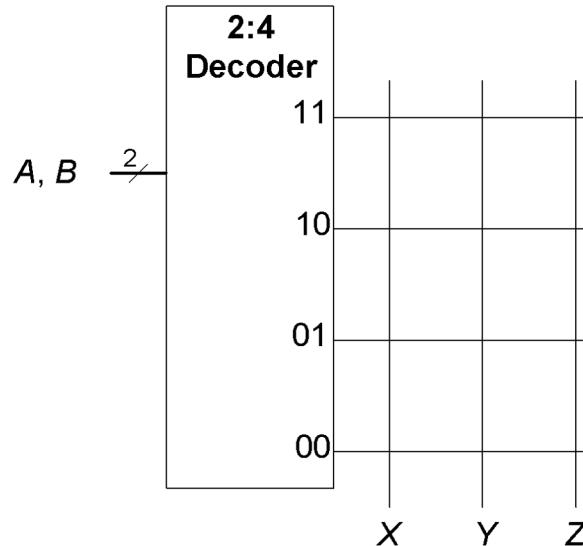
| A ₁ | A ₀ | Data ₁ |
|----------------|----------------|-------------------|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Come Prodotto di somma di MAXTERM:
Data₁= $\bar{A}_1 + A_0$

Altro esempio:

Usare un $2^2 \times 3$ -bit ROM per implementare tre funzioni booleane:

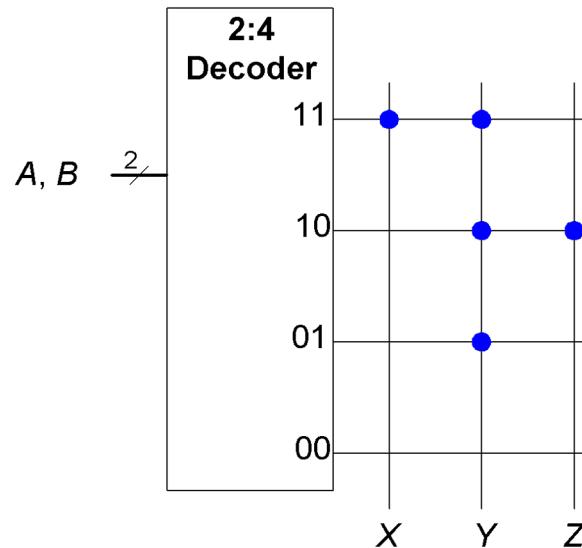
- $X = AB$
- $Y = A + B$
- $Z = A\bar{B}$



Altro esempio:

Usare un $2^2 \times 3$ -bit ROM per implementare funzioni booleane:

- $X = AB$
- $Y = A + B$
- $Z = A \bar{B}$

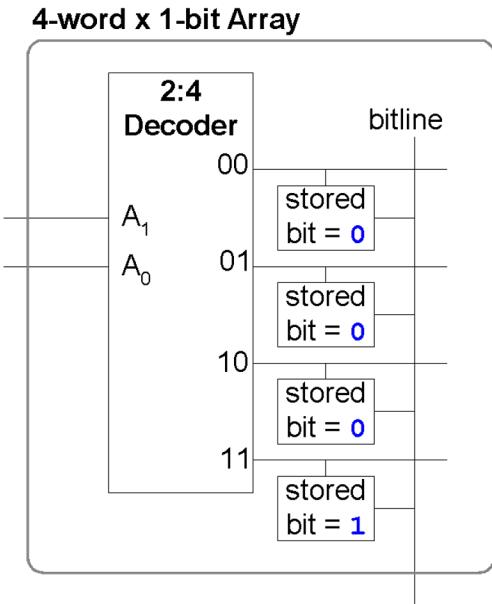


Logiche con memory array

Riassumendo:

Sono chiamate *lookup tables* (LUTs): si “guardano” gli output per ogni combinazione di input (indirizzo)

| Truth Table | | |
|-------------|---|---|
| A | B | Y |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

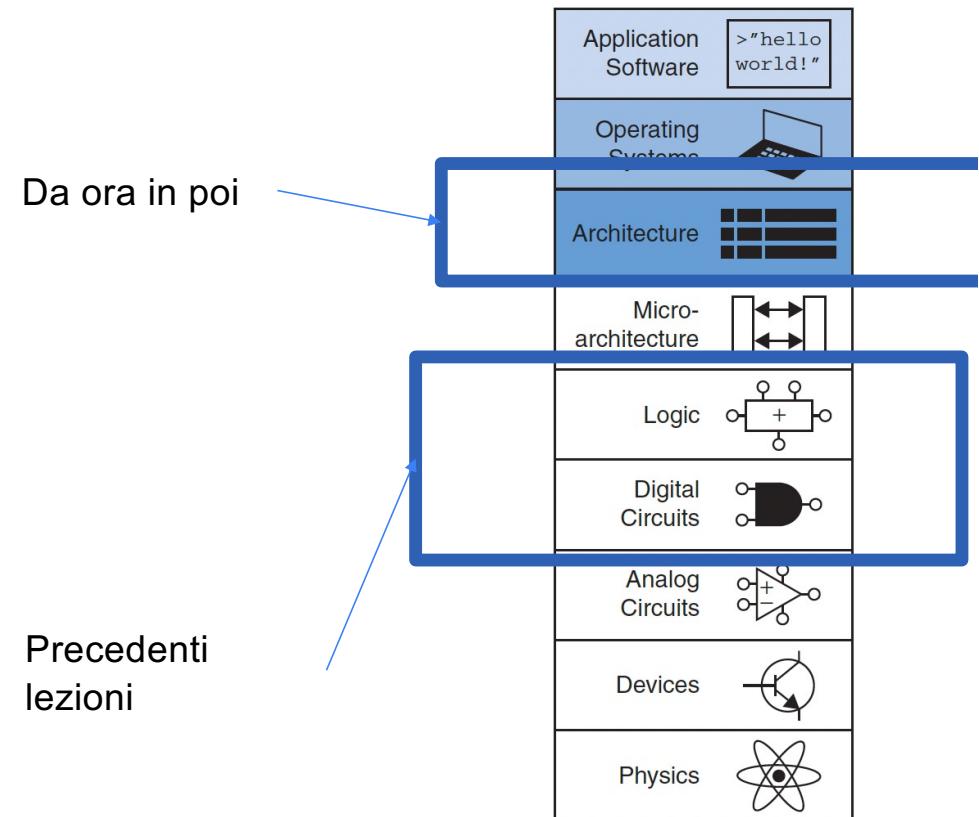


Architettura degli Elaboratori

Lezione 27

Docente: R.Prevete
a.a. 2022/2023
19 mag 2023

- Di cosa parleremo
 - Architettura di Von Neumann



Computer (Sistema digitale)

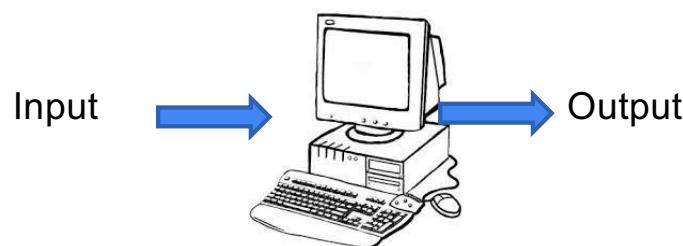
- Il computer è una macchina programmabile (general purpose)
- Grazie alla possibilità di eseguire programmi diversi, il computer può svolgere compiti molto diversi tra loro:
 - eseguire complessi calcoli matematici,
 - redigere documenti,
 - progettare edifici o automobili,
 - navigare nel Web,
 - effettuare transazioni bancarie,
 - riprodurre video o brani musicali
 - ...

Hardware e Software

- Un computer è dunque costituito da due macro-elementi:
 - **hardware**: insieme di tutti i componenti fisici del computer
 - **software**: insieme dei programmi che richiedono all'hardware di svolgere compiti specifici

Informazione

- Per eseguire le varie attività richieste dall’utente, un calcolatore memorizza e memorizza informazione
 - **riceve** dell’informazione sotto forma di dati di input
 - **memorizza** tale informazione in apposito hardware
 - **elabora** tale informazione manipolando i dati
 - **produce** nuova informazione che viene fornita in risposta sotto forma di dati di output

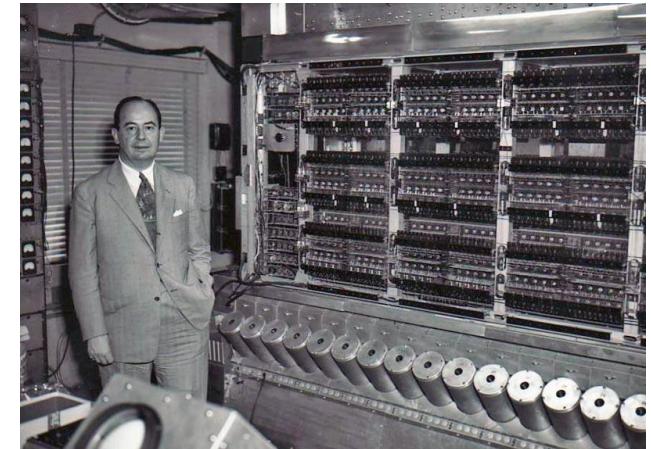


Memorizzazione della Informazione

- L'informazione è memorizzata in **appositi registri**
 - Tutta l'informazione è quindi codificata e memorizzata come sequenze di bit
 - Il **significato delle sequenze di bit dipende esclusivamente da come vengono interpretate**
 - Spesso il modo con cui deve essere interpretata una sequenza di bit (un registro) dipende dalla sua posizione

Architettura di von Neumann

- I diversi calcolatori esistenti differiscono molto tra di loro dal punto di vista dell'hardware
- Tuttavia, quasi tutti si basano su un'architettura comune chiamata **architettura di von Neumann** (o **macchina di von Neumann**)

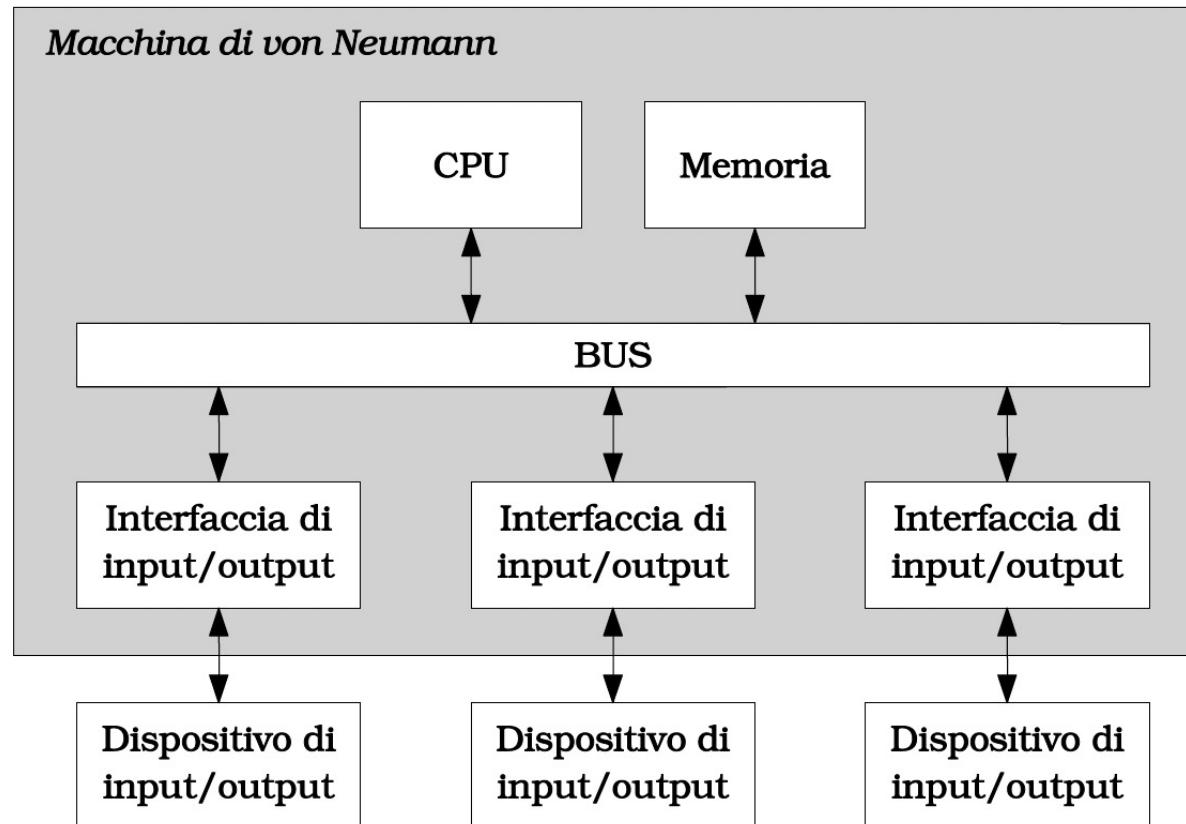


John von Neumann
(1903-1957)

First draft of a report on the EDVAC, 30 giugno 1945

EDVAC: Electronic Discrete Variable Automatic Calculator

Architettura di von Neumann



Architettura di von Neumann

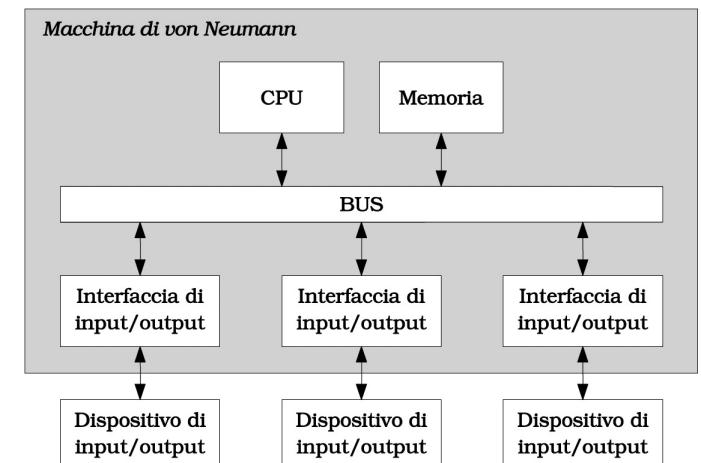
- **CPU (Central Processing Unit)**: è il dispositivo che esegue le elaborazioni. È costituito da due componenti:
 - **ALU (Arithmetic Logic Unit)**: esegue calcoli aritmetici e logici
 - **Unità di Controllo (UC)**: interpreta le istruzioni da eseguire e coordina le attività degli altri componenti
- **Memoria**: memorizza dati e programmi. È organizzata in celle di lunghezza fissa (ad es. 8 bit), identificate da un indirizzo numerico

Architettura di von Neumann

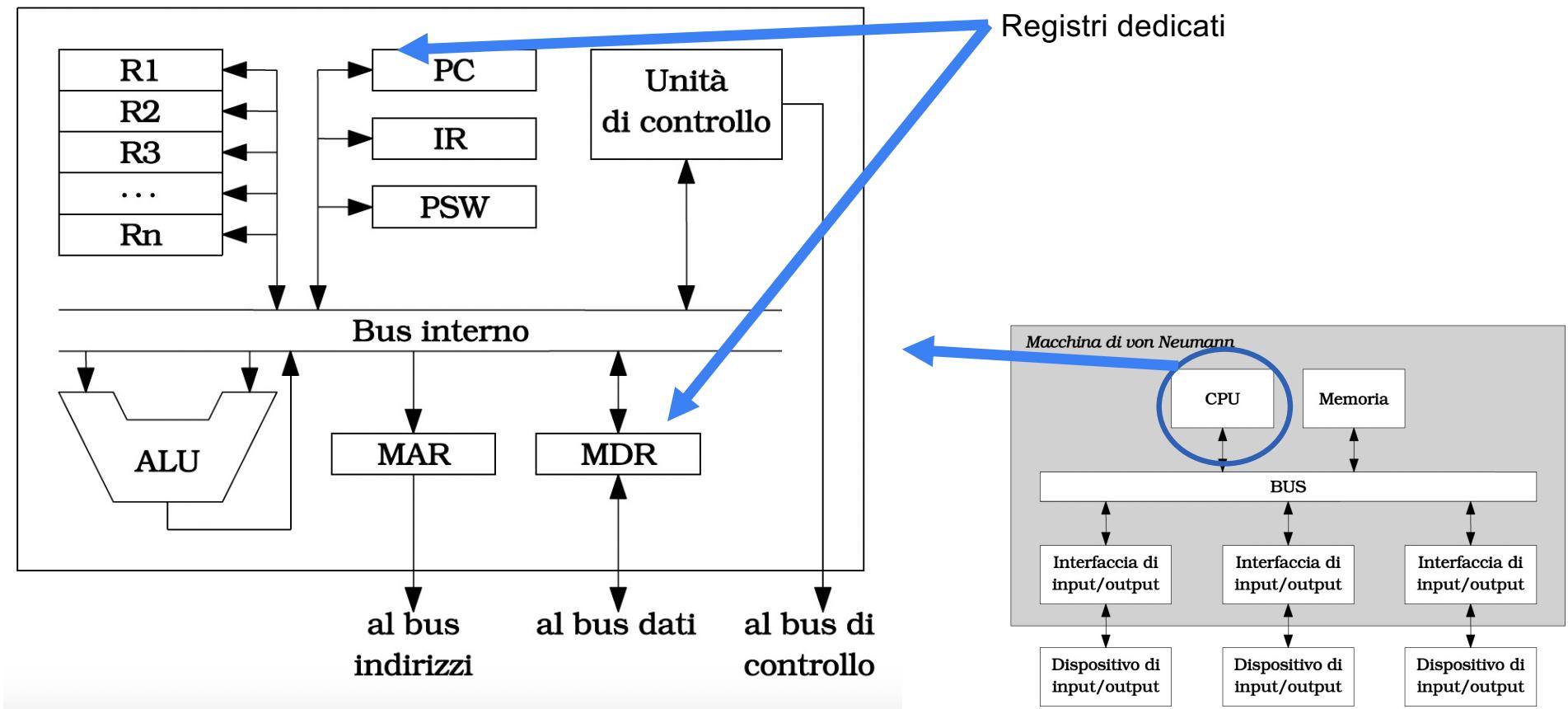
- **Interfacce di Input/Output(I/O)**: consentono al calcolatore di comunicare con le periferiche di I/O (mouse, tastiera, monitor,...) che sono considerate esterne al sistema
- **Bus**: canale di comunicazione tra i vari componenti. Logicamente composto da:
 - bus indirizzi
 - bus dati
 - bus di controllo

Processore (CPU)

- La **CPU** viene realizzata nei calcolatori come *microprocessore*
 - Un **microprocessore** è in grado di eseguire un insieme di istruzioni che costituiscono il suo **linguaggio macchina**
 - Le **istruzioni sono molto semplici:**
 - istruzioni aritmetiche
 - istruzioni logiche
 - confronti
 - letture e scritture da/in memoria

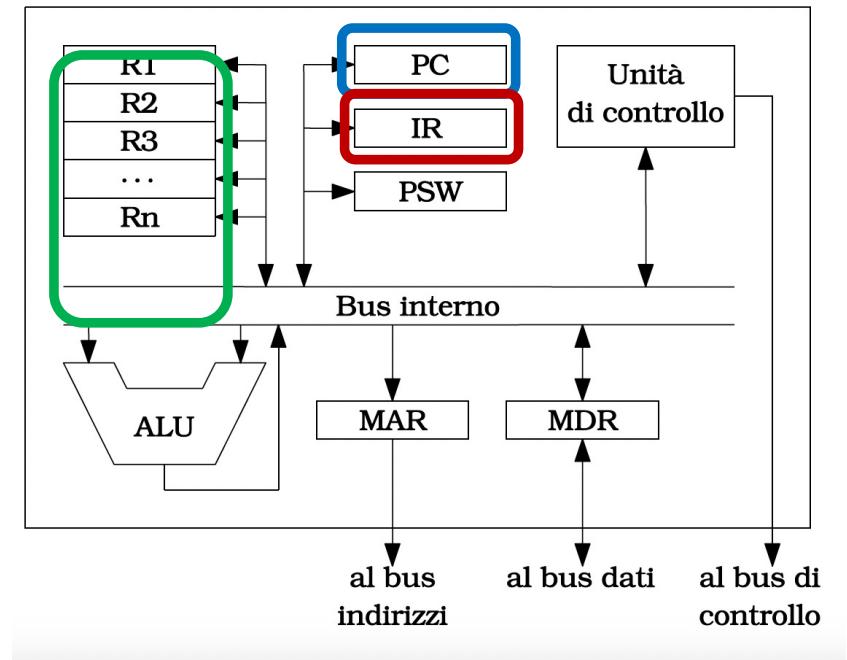


Architettura interna CPU



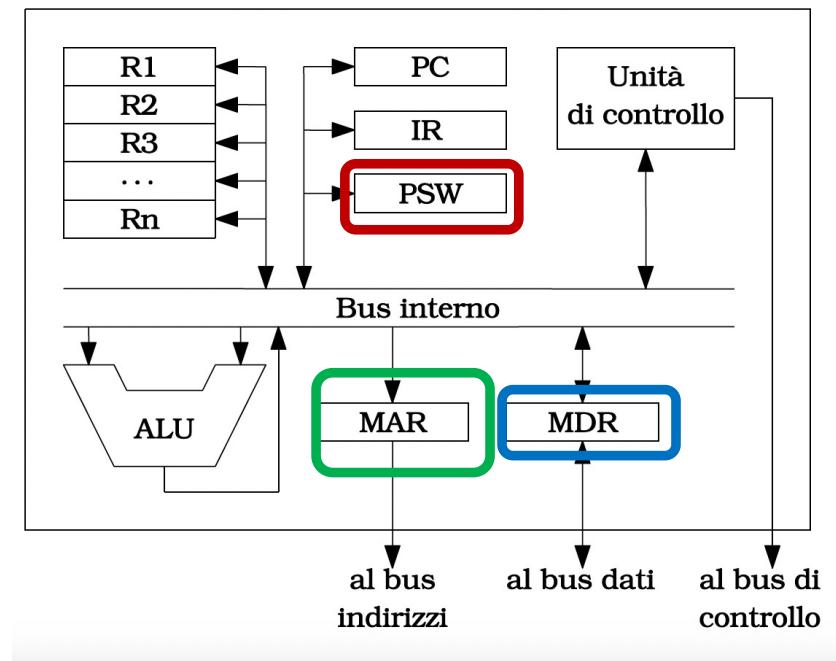
Architettura interna CPU: registri

- Oltre alla ALU e all'unità di controllo, la CPU contiene **una serie di registri**
- I registri sono celle di memoria utilizzate dalla CPU per le sue attività:
 - **Registri generali R₁...R_n**: deposito temporaneo dei dati
 - **Program Counter(PC)**: memorizza l'indirizzo della prossima istruzione da eseguire
 - **Instruction Register (IR)**: memorizza l'istruzione in esecuzione



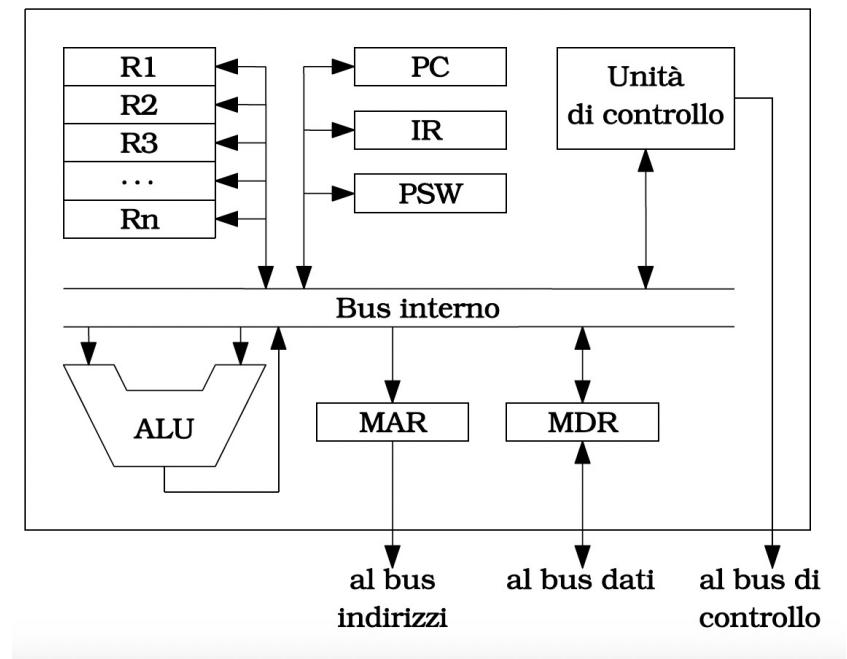
Architettura interna CPU: Registri

- **Memory Address Register (MAR)**: nelle operazioni di lettura/scrittura della memoria, memorizza l'indirizzo della memoria da leggere/scrivere
- **Memory Data Register (MDR)**: nelle operazioni di lettura/scrittura della memoria, memorizza il dato da scrivere in memoria o riceve il dato letto dalla memoria
- **Program Status Word (PSW)**: memorizza una serie di informazioni sull'ultima operazione eseguita dalla ALU



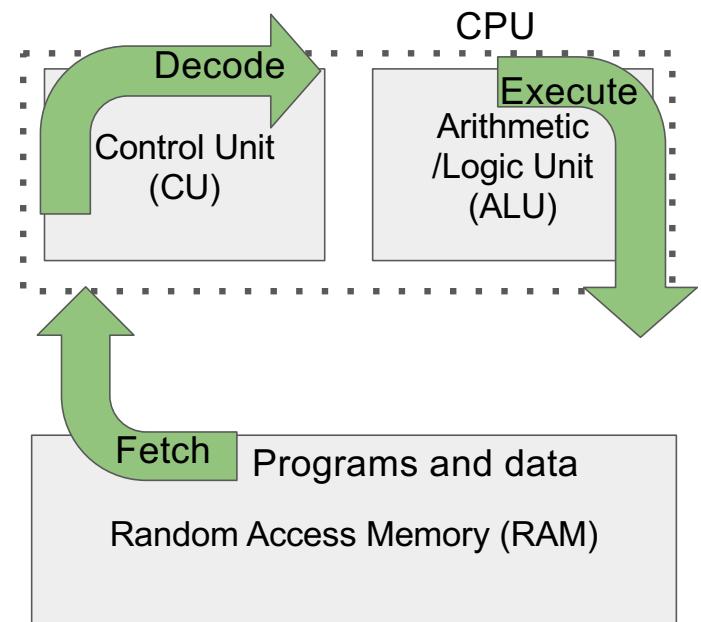
Architettura interna CPU: Bus interno e clock

- Tutti gli elementi interni della CPU comunicano tramite un **bus interno**
- Tutte le attività della CPU sono sincronizzate dall'orologio di sistema (**clock**):
 - ad **ogni ciclo di clock** la CPU esegue una qualche attività
 - la frequenza di clock(misurata in Hz - Hertz) è una misura della velocità del processore



Istruzioni e loro esecuzione

- Le **istruzioni del linguaggio macchina** sono, come i dati, **semplici sequenze di bit**
- Un'istruzione ha un **codice operativo (opcode)** che indica il tipo di operazione ed eventualmente degli **operandi**
- **L'esecuzione di un'istruzione avviene in tre fasi:**
 - fase di **fetch**: *caricamento dell'istruzione dalla memoria*
 - fase di **decode**: *decodifica dell'istruzione*
 - fase di **execute**: *esecuzione dell'istruzione*

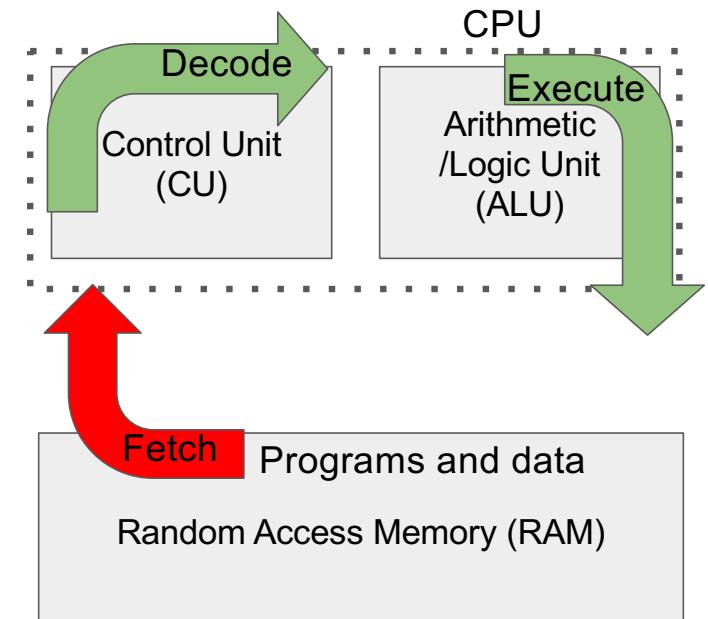


Ciclo Fetch-Decode-Execute

- **Fase di fetch:**

- il contenuto del PC viene scritto nel MAR
- l'istruzione viene letta dalla memoria e scritta nel MDR
- il contenuto del MDR viene trasferito nell'IR
- il PC viene incrementato

- **Memory Address Register (MAR):** memorizza l'indirizzo della cella da leggere/scrivere
- **Memory Data Register (MDR):** riceve il dato letto dalla memoria
- **Program Counter(PC):** memorizza l'indirizzo della prossima istruzione da eseguire
- **Instruction Register (IR):** memorizza l'istruzione in esecuzione

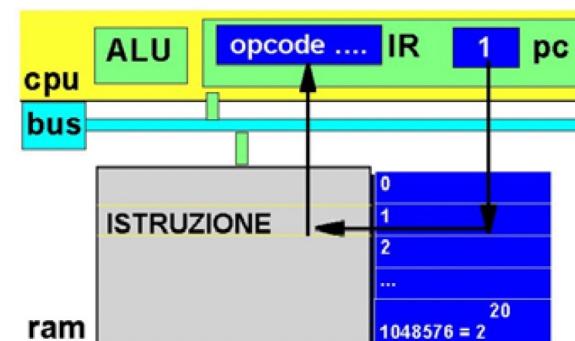
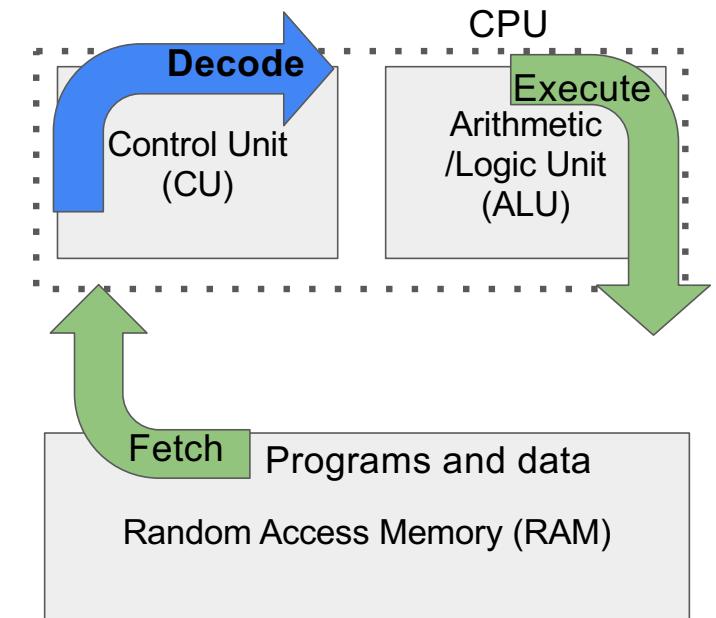


Ciclo Fetch-Decode-Execute

- **Fase di Decode:**
 - sulla base del codice operativo (opcode) dell'istruzione la CPU capisce il tipo di operazione da eseguire e come interpretare gli operandi

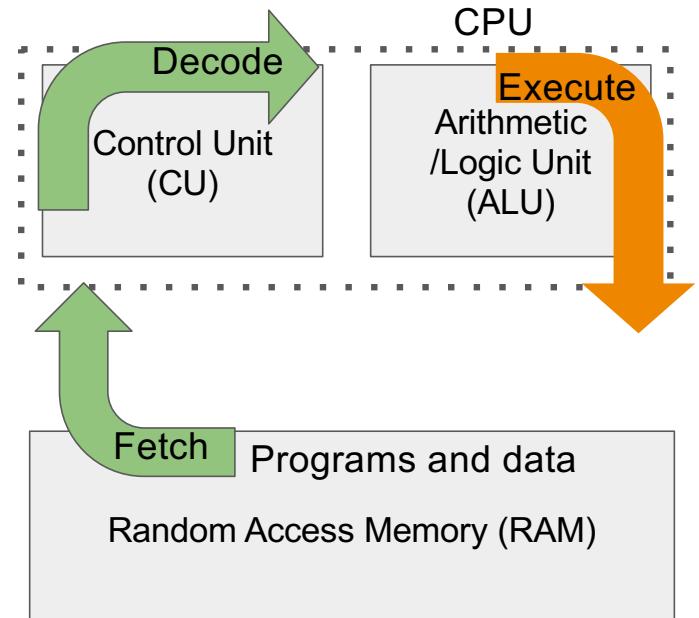
Opcode →

| OpCode | Operazione |
|--------|-------------|
| 1 | Somma |
| 2 | Sottrazione |
| ... | ... |



Ciclo Fetch-Decode-Execute

- **Fase di Execute:**
 - si individuano i dati usati dall'istruzione
 - si trasferiscono tali dati nei registri opportuni
 - si esegue l'istruzione
- Esempio ($R1 = R1 + R2$), cioè "scrivi in $R1$ la somma dei contenuti dei registri $R1$ e $R2$ ":
 - il contenuto del registro $R1$ viene trasferito in ingresso alla ALU
 - il contenuto del registro $R2$ viene trasferito in ingresso alla ALU
 - il risultato viene scritto nel registro $R1$ (sovrascrivendo il precedente valore)



Oltre la macchina di Von Neumann

- **Limite:** nella Macchina di Von Neumann le operazioni sono strettamente sequenziali.
- **Altre soluzioni** introducono forme di **parallelismo**:
 - processori dedicati (coprocessori) al calcolo numerico, alla gestione della grafica, all'I/O.
 - esecuzione in parallelo delle varie fasi di un'istruzione: mentre se ne esegue una, si acquisiscono e decodificano le istruzioni successive (pipeline)
 - architetture completamente diverse: *sistemi multi-processore, LISP machine, reti neurali.*

Architettura degli Elaboratori

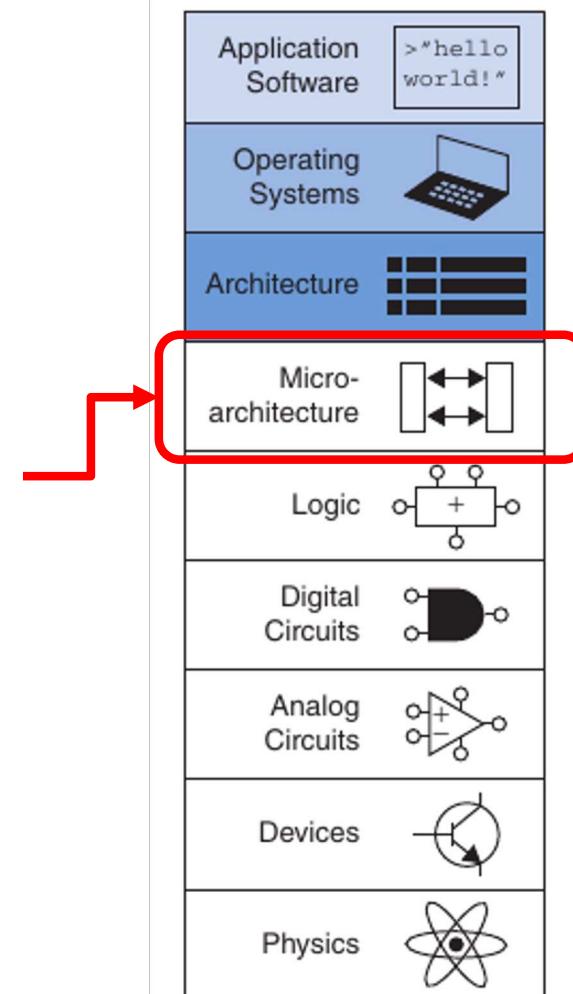
Lezione 28

Docente: R.Prevete
a.a. 2022/2023
22 maggio 2023

Architettura

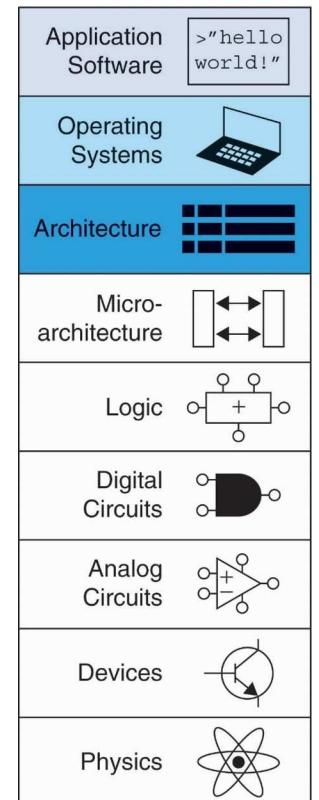
- Di cosa parleremo
 - Architettura ARM
 - registri
 - memoria

Abbiamo saltato questo livello della gerarchia

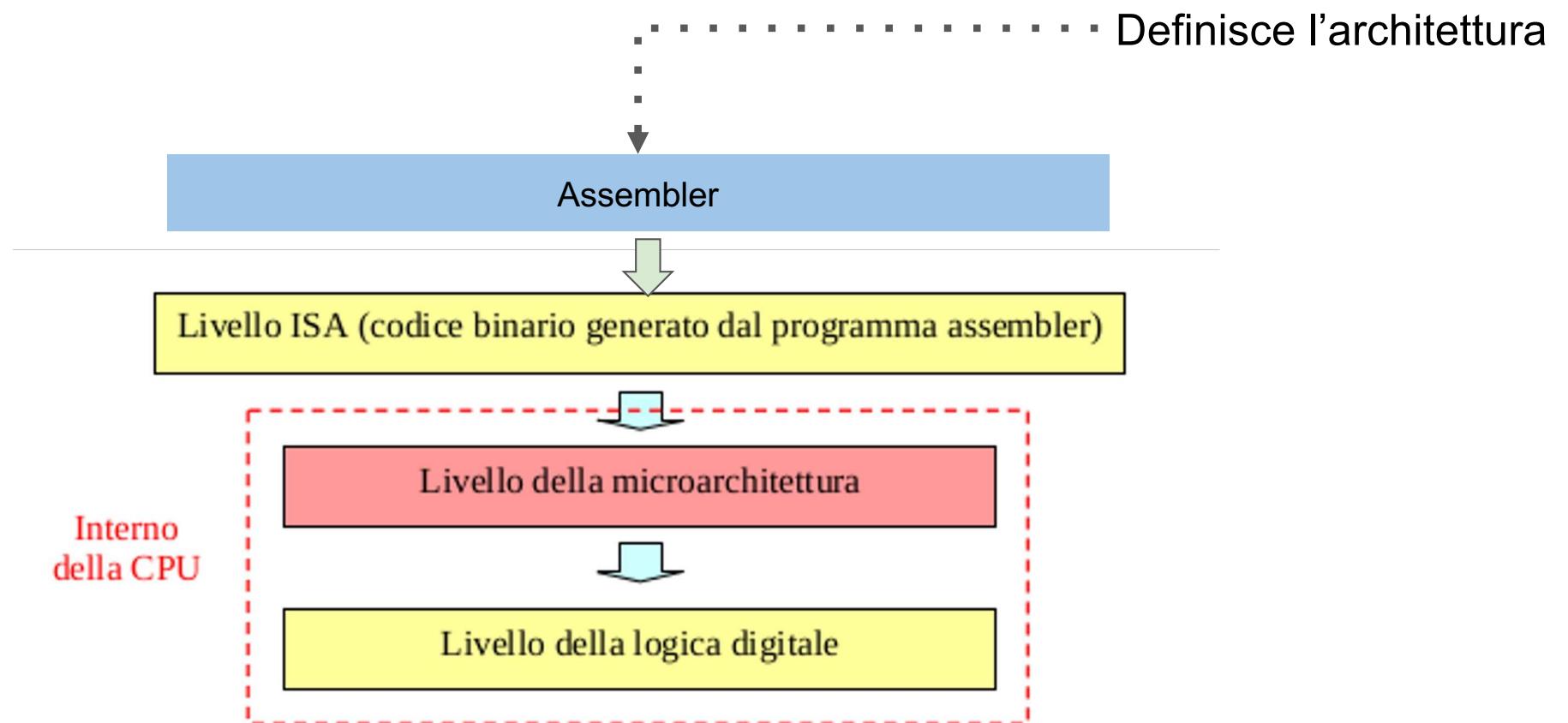


Introduzione

- **Architettura:** descrizione dal punto di vista delle operazioni (operazionale) di computer
 - Come un programmatore interagisce a basso livello con un computer
 - Definisce un insieme di:
 - istruzioni
 - registri che fungono da operandi per le istruzioni



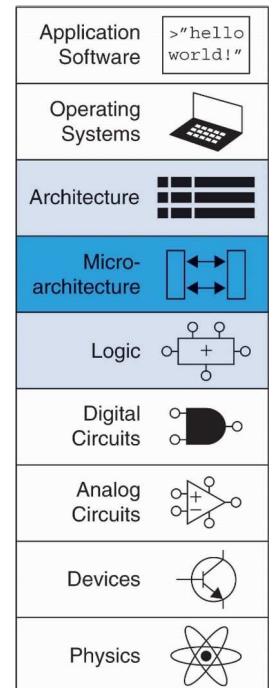
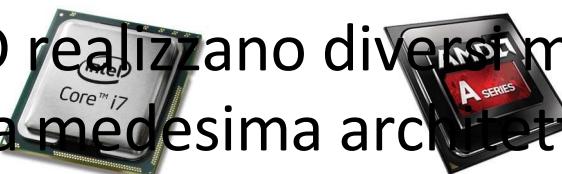
Livelli di Astrazione



Introduzione

Microarchitettura: definisce la disposizione specifica di registri, ALU, macchine a stati finiti, le memorie e altri blocchi logici necessari per implementare una architettura

- Come due algoritmi di ordinamento (bubble e quick sort) realizzano la stessa funzione con procedure differenti, così due microarchitetture possono realizzare la medesima architettura ma soluzioni tecnologiche e prestazioni molto differenti
- Ad esempio Intel e AMD realizzano diversi modelli (microarchitetture) della medesima architettura x86.

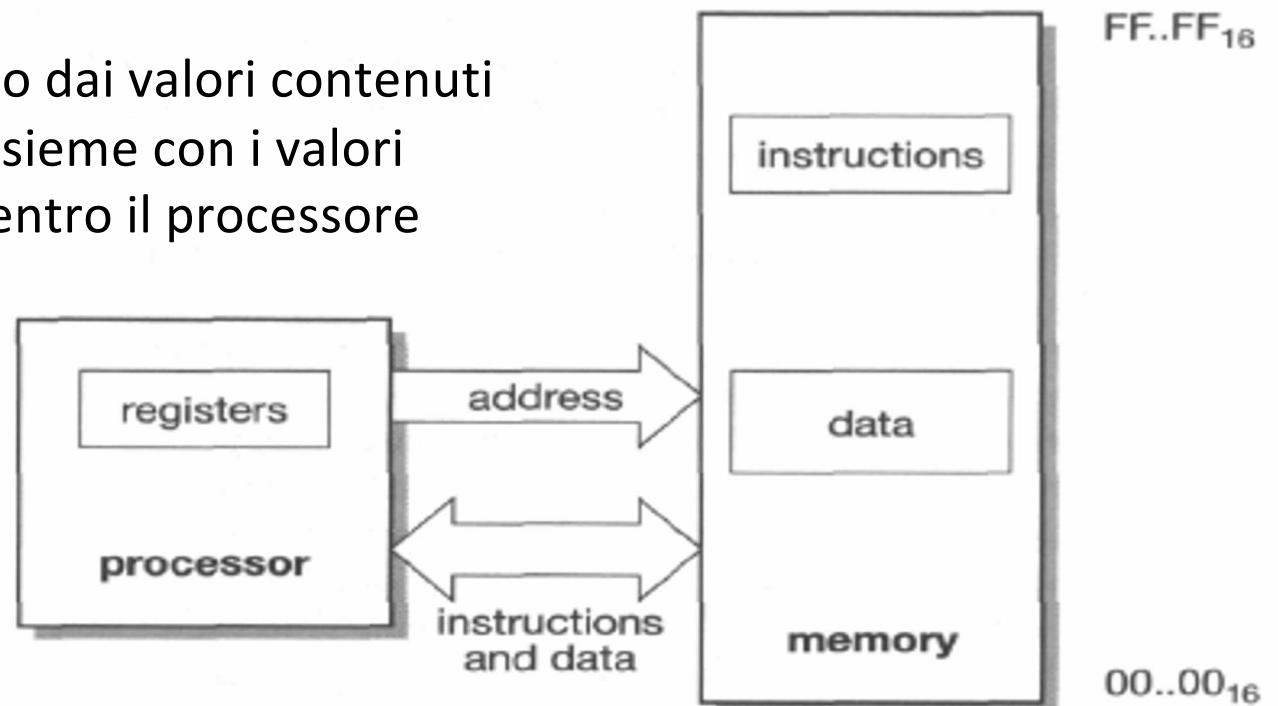


Cos'è un processore (CPU)

Un processore general-purpose è un **automa** a stati finiti, che esegue istruzioni rilocate in una memoria.

Lo **stato** del sistema è definito dai valori contenuti nelle locazioni di memoria insieme con i valori contenuti in alcuni **registri** dentro il processore stesso.

Ogni **istruzione** definisce in che modo lo stato deve cambiare e quale istruzione deve essere eseguita successivamente.



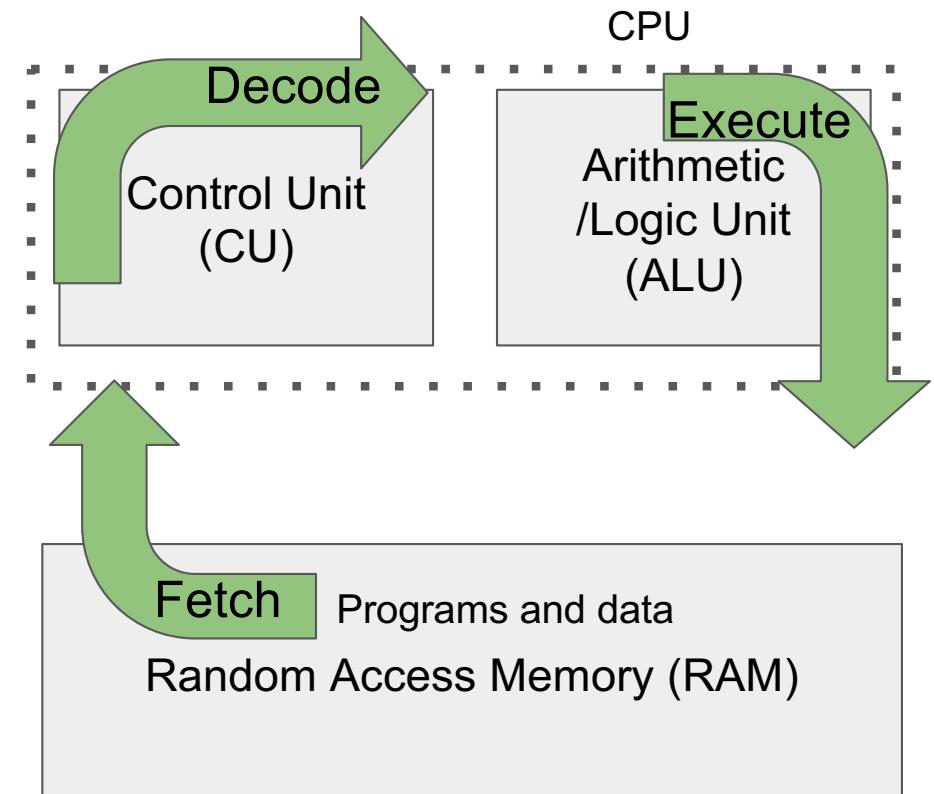
Schema generale di funzionamento

1. Valuta l'indirizzo di memoria presente nel **registro prossima istruzione** (*program counter*), e prendi da tale memoria l'istruzione seguente
2. Metti tale istruzione nel registro delle istruzioni
3. Cambia il contenuto del *registro prossima istruzione* per indicare l'istruzione seguente
4. Valuta il tipo dell'istruzione appena letta
 - a. Se l'istruzione usa delle parole in memoria, determina dove si trovano
 - b. Metti le parole, se necessario, in dei registri della CPU
5. Esegui l'istruzione
6. Torna al punto 1 e inizia a eseguire l'istruzione successiva

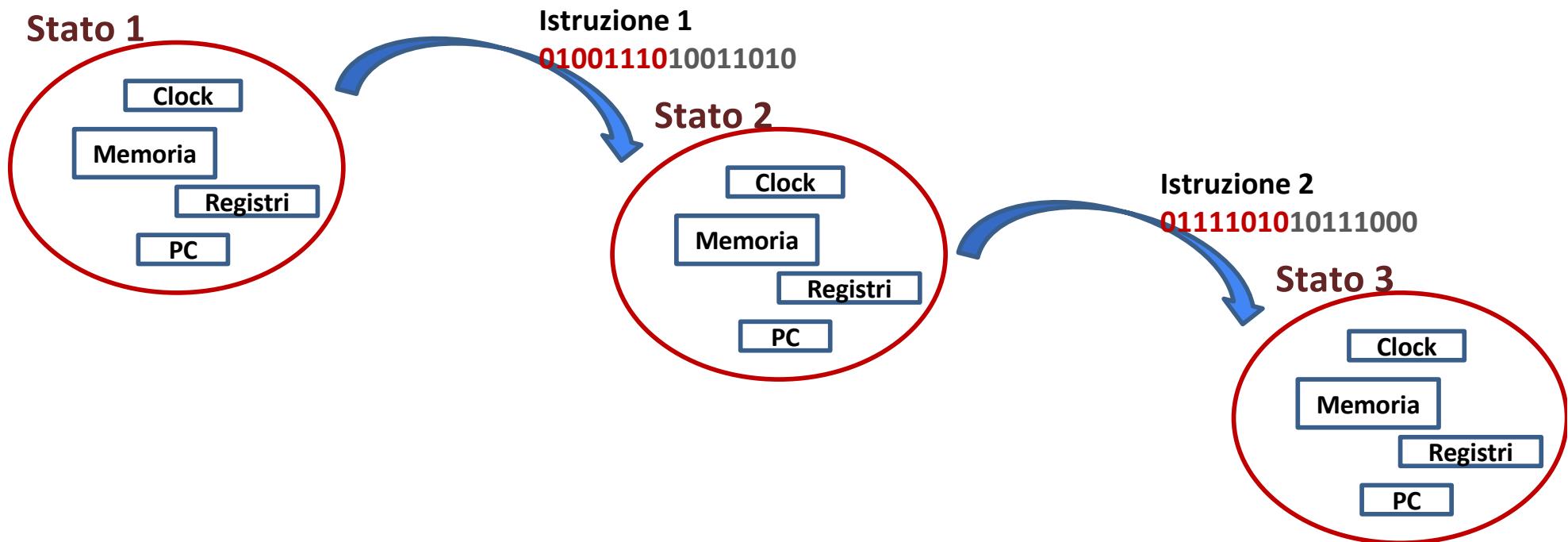
Schema generale

CPU esegue continuamente tale ciclo in tre momenti chiave:

1. **Fetch**: Prelevo l'istruzione dalla memoria RAM
2. **Decode**: Decodifico l'istruzione
3. **Execute**: L'istruzione è eseguita: i banchi di registro sono letti, il risultato ALU generato e scritto nel registro destinazione



CPU come Macchina a Stati Finiti



Da cosa è definita, quindi, una architettura?

- Dall'insieme di istruzioni di basso livello
- Dalla locazione degli operandi (registri e memoria)
- **Esistono differenti tipi di architetture:**
 - ARM, x86, MIPS, SPARC, PowerPC
 - Rappresentano più dei “dialetti” che dei veri linguaggi differenti

Famiglie di architetture

- **RISC** (Reduced Instruction Set Computer) **ARM**
 - Poche istruzioni di base e semplici.
 - Possono essere eseguite in un unico ciclo
 - Semplificano la CPU
- **CISC** (Complex Instruction Set Computers): **X86, pentium**
 - Insieme di istruzione di base più ampio e più complesso
 - Più possibilità quanto si scrive un programma a basso livello
 - CPU più complessa

CISC vs RISC

| CISC | RISC |
|--|--|
| Enfasi sull'hardware | Enfasi sul software |
| Istruzioni di diverse dimensioni e formati | Istruzioni della stessa dimensione con pochi formati |
| Meno registri | Usa più registri |
| Più modalità indirizzamento | Meno modalità indirizzamento |
| Uso esteso della microprogrammazione | Complessità nel compilatore |
| Le istruzioni richiedono un numero variabile di cicli di clock | Le istruzioni richiedono un solo ciclo di clock |
| Il pipelining è difficoltoso | Il pipelining è semplice |

Architettura ARM (RISC)

- Sviluppata negli anni 80 dalla Advanced RISC Machines – ora chiamata ARM Holdings
- Circa 10 miliardi di processori ARM venduti ogni anno
- Impiegata soprattutto nel settore degli smartphone e tablet
- Circa il 75% della popolazione umana usa prodotti equipaggiati da un processore ARM

ARM: Rende veloce i casi comuni

- L'architettura ARM comprende solo istruzioni semplice e di uso frequente
- Hardware necessario per decodificare ed eseguire le istruzioni è così *semplice* e veloce
- Istruzioni complesse (che sono meno comuni) corrispondono a più istruzioni ARM

C Code

```
a = b + c - d;
```

ARM assembly code

```
ADD t, b, c ; t = b + c  
SUB a, t, d ; a = t - d
```

ARM: Smaller is faster

I dati sono memorizzati:

- Direttamente all'interno di una istruzione.
 - Costanti (anche dette *immediates*)
- In registri (nella CPU)
- In memoria (RAM)

ARM: Smaller is faster

Dati in registri:

- ARM ha **16** registri principali
- I registri sono più veloci della memoria
- Ogni registro è costituito da 32 bits
- ARM chiamata una architettura a “32-bit” perché opera su dati di 32-bit

ARM: Smaller is faster

Dati in memoria:

- Più lenta dei registri ma più capiente
- A sua volta suddivisa in diversi livelli: **cache, centrale, di massa**
- **Le istruzioni ARM operano esclusivamente sui registri**, quindi i *dati residenti nella memoria devono essere spostati in un registro prima di poter essere elaborati.*

ARM: varie versione

Attualmente
...versione v9

| ARM Family | ARM Architecture | ARM Core | Feature |
|------------|------------------|----------|--|
| ARM1 | ARMv1 | ARM1 | First implementation |
| ARM2 | ARMv2 | ARM2 | ARMv2 added the MUL (multiply) instruction |
| | ARMv2a | ARM250 | Integrated MEMC (MMU), Graphics and IO processor. ARMv2a added the SWP and SWPB (swap) instructions. |
| ARM3 | ARMv2a | ARM3 | First integrated memory cache. |
| ARM6 | ARMv3 | ARM60 | ARMv3 first to support 32-bit memory address space (previously 26-bit) |
| | | ARM600 | As ARM60, cache and coprocessor bus (for FPA10 floating-point unit). |
| | | ARM610 | As ARM60, cache, no coprocessor bus. |

ARM: Come andremo a descrivere l'architettura

- Come è organizzata la memoria
- Chi sono e cosa fanno i 16 registri
- Il set di istruzioni e cosa fanno (*assembly language*)
- Dove sono i dati e le istruzioni
- Come sono codificate le istruzioni in memoria (*machine language*)
- Programmi in assembly

La memoria

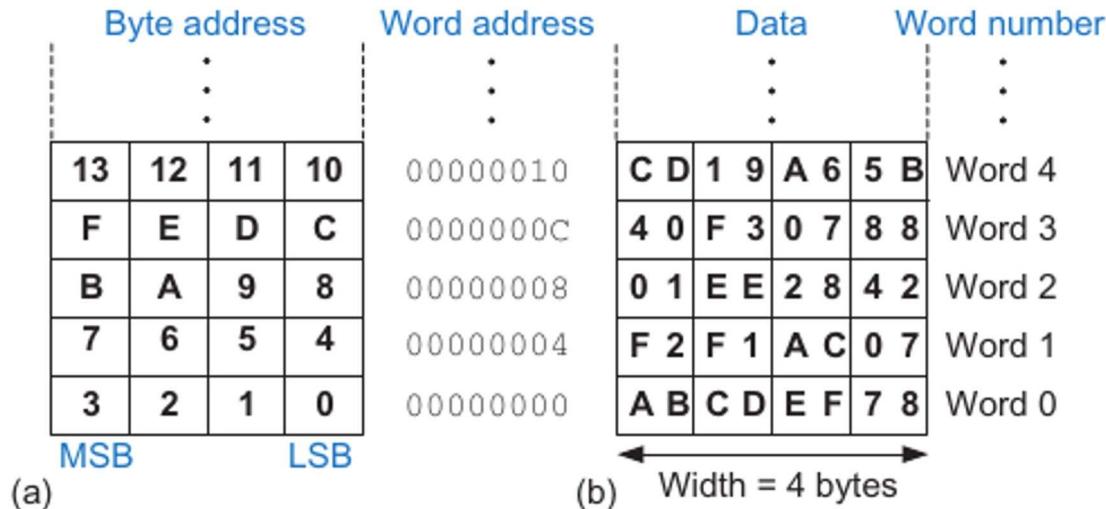
- Nell'architettura ARM, le **istruzioni agiscono/operano esclusivamente su registri**
- I **dati** archiviati in memoria devono essere **spostati in un registro** e poi possono essere elaborati.
- Utilizzando una **combinazione di memoria e registri**, un programma può accedere a una grande quantità di dati abbastanza rapidamente.
- **Ricorda:** La memoria è un array di “registri”, data-words

La memoria

- L'architettura ARM usa una memoria con 32-bit per gli indirizzi e 32-bit per le data-words (parole)
- Tuttavia, ARM usa una memoria byte-addressable (byte-indirizzabile)
 - Ogni byte ha un unico indirizzo
- Un parola può essere vista, allora, come composta da 4 byte (ognuno con un suo indirizzo)
- Ogni indirizzo di una parola, quindi, è un multiplo di 4

Byte-Addressable Memory

- Ad ogni **byte** corrisponde un unico indirizzo



- Indirizzo della prima parola:
 - $0 \times 4 = 0 = 0x...000$
- Indirizzo delle seconda parola:
 - $1 \times 4 = 4 = 0x...004$
- Indirizzo della terza parola:
 - $2 \times 4 = 8 = 0x...008$
- Indirizzo della quarta parola:
 - $2 \times 4 = 16 = 0x...00C$

- Una parola è di 32-bit = 4 bytes, quindi gli indirizzi delle parole incrementano di 4 in 4

Byte-Addressable Memory

- Il byte più significativo (The most significant byte - MSB) è sulla sinistra
 - Il byte meno significativo (The least significant byte - LSB) è sulla destra
 - Per convenzione la memoria è rappresentata con gli indirizzi di memoria più piccoli verso il basso, mentre gli indirizzi di memoria più grandi verso l'alto

| Byte address | | | | Word address | | | |
|--------------|--|--|--|--------------|--|--|--|
| : | | | | : | | | |
| 13 | | | | 00000010 | | | |
| F | | | | 0000000C | | | |
| B | | | | 00000008 | | | |
| 7 | | | | 00000004 | | | |
| 3 | | | | 00000000 | | | |
| | | | | | | | |
| MSB | | | | LSB | | | |

Registri:

I registri dei processori ARM sono 16: da R0 a R15, e si distinguono in base al loro utilizzo:

- **R0 - R12** sono i registri di uso generale;
 - Can be used during common operations to store temporary values, pointers (locations to memory), etc. R0, for example, can be referred as accumulator during the arithmetic operations or for storing the result of a previously called function.
- **R13** (o SP) è generalmente il registro dello **Stack Pointer**, cioè l'indirizzo di memoria dello stack (memoria per una funzione). Non è però obbligatorio il suo utilizzo come Stack pointer; viene usato solo per convenzione;

Registri:

I registri dei processori ARM sono 16: da R0 a R15, e si distinguono in base al loro utilizzo:

- **R13** (o SP) è generalmente il registro dello **Stack Pointer**, cioè l'indirizzo di memoria dello stack (memoria per una funzione). Non è però obbligatorio il suo utilizzo come Stack pointer; viene usato solo per convenzione;

Registri:

I principali registri dei processori ARM sono 16: da R0 a R15, e si distinguono in base al loro utilizzo:

- **R14 (o LR)** è il registro del **Link Register**, nel quale viene salvato l'indirizzo di ritorno nel momento in cui viene chiamata una procedura;
 - When a function call is made, the Link Register gets updated with a memory address referencing the next instruction where the function was initiated from. Doing this allows the program return to the “parent” function that initiated the “child” function call after the “child” function is finished.

Registri:

I principali registri dei processori ARM sono 16: da R0 a R15, e si distinguono in base al loro utilizzo:

- **R15 (o PC)** è il **Program Counter**, il registro contiene l'indirizzo di memoria della prossima istruzione da eseguire

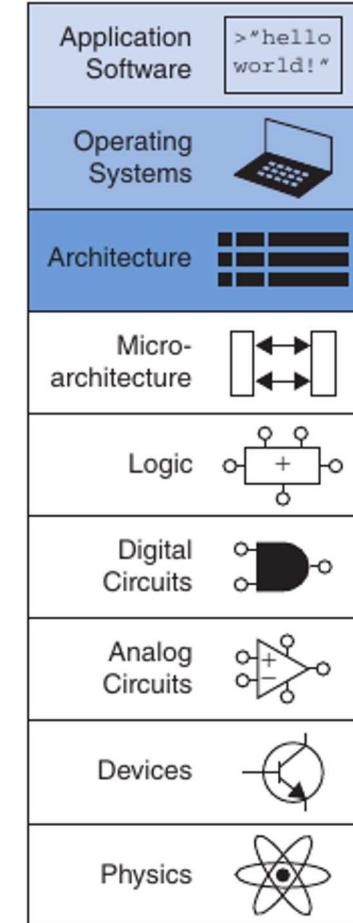
Architettura degli Elaboratori

Lezione 29

Docente: R.Prevete
a.a. 2022/2023
2023 05 24

Architettura

- Di cosa parleremo
 - Architettura ARM
 - Registri
 - Interazione con la memoria
 - Tipi di dati
 - Istruzioni di tipo data processing



Registri: Overview

Variabili che si vogliono mantenere per più lungo termine

Table 6.1 ARM register set

| Name | Use |
|----------|--|
| R0 | Argument / return value / temporary variable |
| R1–R3 | Argument / temporary variables |
| R4–R11 | Saved variables |
| R12 | Temporary variable |
| R13 (SP) | Stack Pointer |
| R14 (LR) | Link Register |
| R15 (PC) | Program Counter |

Memoria di una funzione

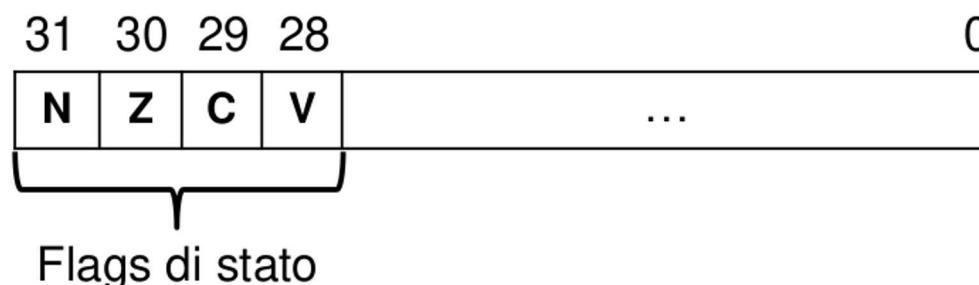
Dopo chiamata funzione

Prossima istruzione

Registro di stato

- Un altro registro molto importante è il registro di stato CPSR (Current Program Status Register) il quale tiene traccia dello stato del programma.
- Molto utili sono i 4 bit di stato (detti **flags di stato**) i quali esprimono le seguenti condizioni aritmetiche (Ricordate l'uscita del ALU??):
 - Bit N negativo;
 - Bit Z zero;
 - Bit C carry/overflow;
 - Bit V overflow.

Registro CPSR



Linguaggio Assembler

- Il linguaggio assembler è un linguaggio di basso livello ma comprensibile per un essere umano

High-Level Code

```
a = b + c;
```

ARM Assembly Code

```
ADD a, b, c
```

Esempio:

- L'operazione di ADD è eseguita su b e c, il risultato è posto in a
- ADD è l'istruzione da eseguire (the *mnemonic*)
- b, c sono gli *operandi sorgenti*
- a è l'*operando destinazione*

Linguaggio Assembler

High-Level Code

a = b + c;

ARM Assembly Code

ADD a, b, c

High-Level Code

a = b - c;

ARM Assembly Code

SUB a, b, c

- Tutte le istruzioni hanno lo stesso formato
 - Principio di regolarità: Regolarità significa semplicità

Inizializzare un registro

- MOV:

L'istruzione MOV permette di inizializzare il valore di un registro con un immediate, una costante

ESEMPI:

MOV R4, #64

MOV R5, #0xFF0

Inizializzare un registro

- MOV:

L'istruzione MOV può anche avere come secondo operando un registro

ESEMPIO:

MOV R1, R7

copia il contenuto del registro R7 in R1.

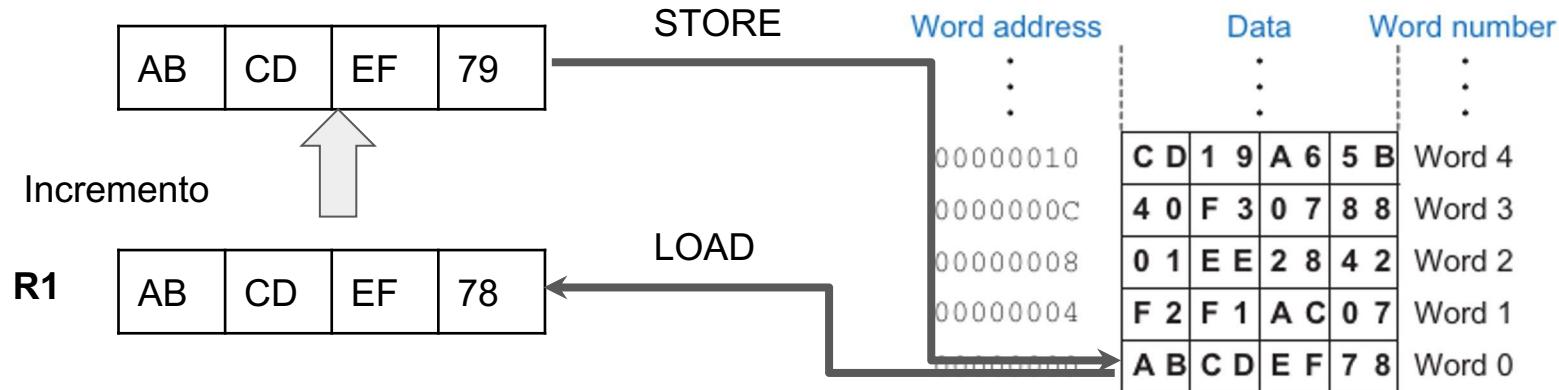
Istruzioni relative alla memoria: load/store model

- ARM utilizza un modello load/store per l'accesso alla memoria:
 - solo le istruzioni di Load/store possono accedere alla memoria.

Mentre su x86 la maggior parte delle istruzioni può operare direttamente sui dati in memoria, su ARM i dati devono essere spostati dalla memoria nei registri prima di essere utilizzati.

Istruzioni relative alla memoria: load/store model

- Ad esempio: l'incremento di un valore a 32 bit in un particolare indirizzo di memoria su ARM richiederebbe tre tipi di istruzioni:
 - Caricare prima il valore in un determinato indirizzo in un registro,
 - Incrementarlo all'interno del registro
 - Salvarlo di nuovo in memoria nel registro di memoria.



Istruzioni relative alla memoria: load/store model

Due forme principali:

- LOAD: LDR
 - LDR Ra, [Rb, imm] imm è l'offset ed è un immediate
 - LDR Ra, [Rb, Rc] offset è il valore presente nel register Rc
- STORE: STR
 - STR Ra, [Rb, imm] imm è l'offset ed è un immediate
 - STR Ra, [Rb, Rc] offset è un register

Rb+ imm oppure Rb+ Rc sono indirizzi di memoria

Istruzioni relative alla memoria: load/store model

- LDR Ra, [Rc, imm] offset è un immediate
 - Possiamo omettere il valore dell'immmediate (caso semplice)

ESEMPIO:

MOV R5, #0

LDR R7, [R5]

; indirizzo base = 0

; R7 <= i dati contenuti nell'indirizzo di memoria (R5)

Istruzioni relative alla memoria: load/store model

- LDR Ra, [Rc, Rb] l'offset è un *register offset*
- In questo caso l'indirizzo memorizzato in Rc è addizionato con il valore presente in Rb. Il valore risultante è l'indirizzo da cui prendere il contenuto e porlo in Ra

ESEMPIO:

MOV R5, #0

LDR R7, [R5, #8]

; indirizzo base = 0

; R7 <= dati contenuti nell'indirizzo di memoria (R5+8)

Istruzioni relative alla memoria: load/store model

- STR Ra, [Rc, imm] offset è un immediate
- STR Ra, [Rb, Rc] offset è un register

In maniera analoga ai casi visti prima

ESEMPIO:

MOV R5, #12 ; #0xc

MOV R7, #4

STR R7, [R5, #8]

; indirizzo base = 12

; R7 => i dati contenuti nell'indirizzo di memoria R7 sono posti
nell'indirizzo di memoria ($R5+8=12+8=20$)

Istruzioni relative alla memoria: load/store con offset scalato (shiftato = moltiplicato/diviso)

- LDR Ra, [Rb, Rc, <shifter>]
 - STR Ra, [Rb, Rc, <shifter>]
-
- Abbiamo un terza forma di offset
 - In questo cas Rb è il registro base, Rc è una costante (o un registro contenente una costante) il cui valore è shiftato a destra o a sinistra secondo quanto specificato da <shifter>

Istruzioni relative alla memoria: load/store con offset scalato (moltiplicato/diviso)

- LDR Ra, [Rb, Rc, <shifter>]
- STR Ra, [Rb, Rc, <shifter>]

STR R3, [R0, R1, LSL #2] ; Store the value found in R3 to the memory address found in R0 with the offset R1 left-shifted by 2. Base register (R0) unmodified. Thus, the memory address is $R0 + (R1 \times 4)$.

LDR R3, [R0, R1, LSL #2] ; R1 is scaled (shifted left by two) then added to the base address (R0). Base register (R0) unmodified. Thus, the memory address is $R0 + (R1 \times 4)$.

Istruzioni relative alla memoria: address mode

Tre diversi modi di usare base e offset per individuare un indirizzo di memoria

- address mode: offset addressing (indirizzamento)
 - Come già abbiamo visto, calcola l'indirizzo come $ind=base + offset$ e il valore della base non cambia
- address mode: pre-index addressing
 - Calcola l'indirizzo come $ind=base + offset$, poi è cambiato il valore della base a ind .
- address mode: post-index
 - Calcola l'indirizzo come $ind=base$, poi è cambiato il valore della base a $base + offset$.

Istruzioni relative alla memoria: address mode

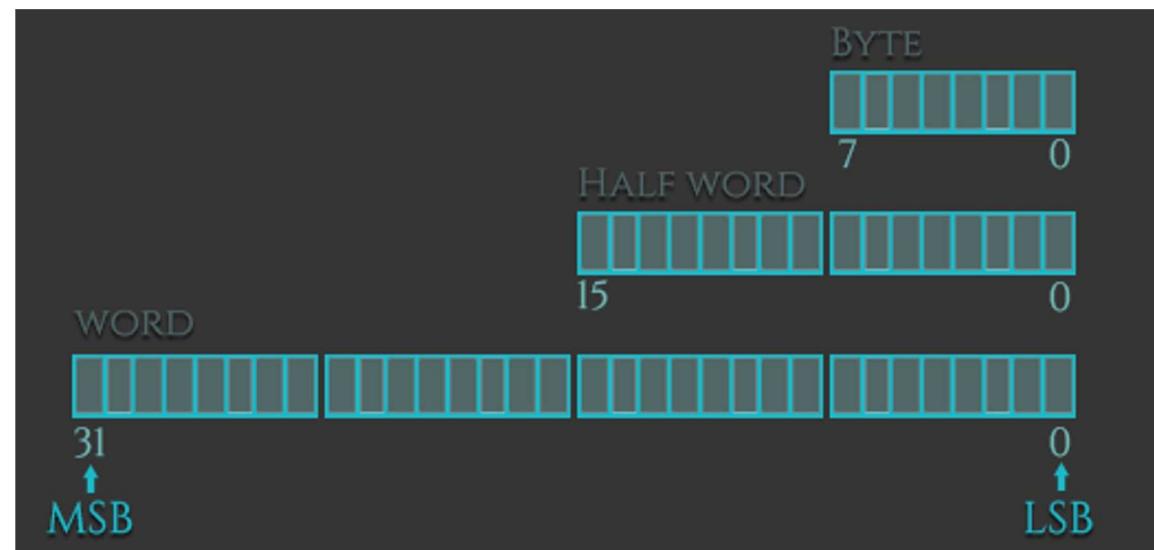
ESEMPI

| Mode | ARM Assembly | Address | Base Register |
|------------|-------------------|---------|---------------|
| Offset | LDR R0, [R1, R2] | R1 + R2 | Unchanged |
| Pre-index | LDR R0, [R1, R2]! | R1 + R2 | R1 = R1 + R2 |
| Post-index | LDR R0, [R1], R2 | R1 | R1 = R1 + R2 |

Linguaggio Assembler: Tipi di dati

Abbiamo fondamentalmente 6 tipi di dati

- **Signed e unsigned**
 - **words**
 - **Half words**
 - **Bytes**



Linguaggio Assembler: Tipi di dati

Abbiamo fondamentalmente 6 tipi di dati

- **Signed** significa che io prendo sia numeri positivi che negativi (però range più piccolo)
- **Unsigned** significa che io prendo solo numeri positivi che negativi (però range più grande)

Linguaggio Assembler: Tipi di dati e interazione con la memoria

| Name | Description | Operation |
|-----------------------|------------------|--|
| STR Rd, [Rn, ±Src2] | Store Register | $\text{Mem}[\text{Adr}] \leftarrow \text{Rd}$ |
| LDR Rd, [Rn, ±Src2] | Load Register | $\text{Rd} \leftarrow \text{Mem}[\text{Adr}]$ |
| STRB Rd, [Rn, ±Src2] | Store Byte | $\text{Mem}[\text{Adr}] \leftarrow \text{Rd}_{7:0}$ |
| LDRB Rd, [Rn, ±Src2] | Load Byte | $\text{Rd} \leftarrow \text{Mem}[\text{Adr}]_{7:0}$ |
| STRH Rd, [Rn, ±Src2] | Store Halfword | $\text{Mem}[\text{Adr}] \leftarrow \text{Rd}_{15:0}$ |
| LDRH Rd, [Rn, ±Src2] | Load Halfword | $\text{Rd} \leftarrow \text{Mem}[\text{Adr}]_{15:0}$ |
| LDRSB Rd, [Rn, ±Src2] | Load Signed Byte | $\text{Rd} \leftarrow \text{Mem}[\text{Adr}]_{7:0}$ |
| LDRSH Rd, [Rn, ±Src2] | Load Signed Half | $\text{Rd} \leftarrow \text{Mem}[\text{Adr}]_{15:0}$ |

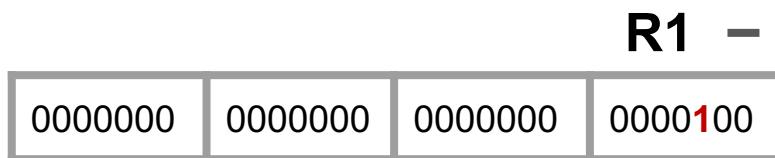
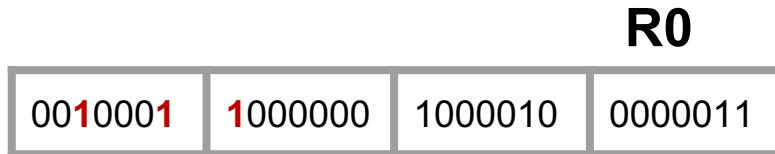
Linguaggio Assembler: Tipi di dati e interazione con la memoria

Si tenga conto che:

- “LDR, STR, LDRB, and STRB are the most common memory instructions
- The immediate offset is only 8 bits and the register offset cannot be shifted.
- LDRB and LDRH zero-extend the bits to fill a word, while
- LDRSB and LDRSH sign-extend the bits.

Esempio LDRH

LDRH R0, [R1, #2]



| | | | | |
|----------|----------|----------|----------|-----|
| | | | | 0xC |
| 00000000 | 11111111 | 01100001 | 01100000 | 0x8 |
| | | | | 0x4 |
| | | | | 0x0 |



Memory address

zero extended



| | | | | |
|----------|----------|----------|----------|-----|
| | | | | 0xC |
| 00000000 | 11111111 | 01100001 | 01100000 | 0x8 |
| | | | | 0x4 |
| | | | | 0x0 |



Memory address

Linguaggio Assembler: Tipi di istruzione

Le istruzioni dell'architettura ARM possono essere suddivise in **tre tipologie**:

- **Data-processing**
- **Branch**
- **Memory**

Si distinguono per il formato dell'istruzione

Linguaggio Assembler: Tipi di istruzione

- **Data processing:** Istruzioni aritmetico-logiche che implementano le operazioni matematiche. Queste possono essere di tipo aritmetiche (somma, differenza, prodotto), logiche (operazioni booleane) o relazionali (comparazioni tra due valori)
- **Branch:** Istruzioni che cambiano il controllo del flusso delle istruzioni, modificando il contenuto del Program Counter (R15). Le istruzioni di branch sono necessarie per l'implementazione di istruzioni di condizione (if-then- else), cicli e chiamate di funzioni.
- **Memory:** istruzioni di Load/Store muovono dati da (load) e verso (store) la memoria principale.

Architettura degli Elaboratori

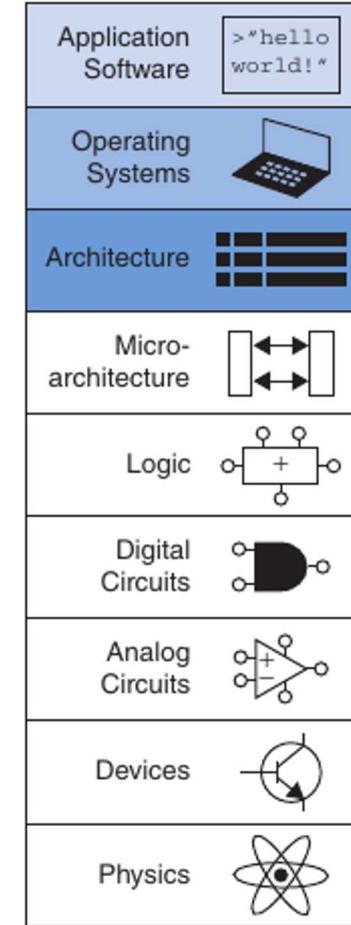
Lezione 30

Docente: R.Prevete
a.a. 2022/2023
26 mag 2023

Architettura

- Di cosa parleremo

- Architettura ARM
 - Istruzioni su dati
 - Istruzioni di shift



Istruzioni aritmetico-logiche

In linguaggio assembly ARM le istruzioni aritmetico-logiche hanno una sintassi del tipo

opcode{S}{condition} dest, op1, op2

- Per **opcode** si intende l'istruzione che deve essere eseguita.
- Il campo {S} è un suffisso opzionale che, se specificato, carica nel registro di stato CPSR il risultato dell'operazione compresi i flag di stato.

Istruzioni aritmetico-logiche

In linguaggio assembly ARM le istruzioni aritmetico-logiche hanno una sintassi del tipo

opcode{S}{condition} dest, op1, op2

- Anche il campo {condition} è un suffisso opzionale il quale definisce la condizione logica necessaria all'esecuzione dell'istruzione; se la condizione è verificata l'istruzione verrà eseguita altrimenti verrà ignorata.
- Il campo dest indica il registro destinatario nel quale verrà salvato il risultato dell'operazione.

Istruzioni aritmetico-logiche: ricordo

In linguaggio assembly ARM le istruzioni aritmetico-logiche hanno una sintassi del tipo

opcode{S}{condition} dest, op1, op2

- In fine **op1** e **op2** sono gli operandi dell'operazione che si desidera eseguire.

Bisogna specificare però che op1 fa riferimento all'operando soltanto mediante indirizzamento a registro, mentre op2 può indirizzare sia in **maniera immediata** che a registro.

Istruzioni aritmetico-logiche: Esempi

| Codice ad alto livello | ARM Assembly code |
|---|--|
| a = b + c; Commento con ';' Solo una riga | <pre>;R0 = a, R1 = b, R2 = c ADD R0, R1, R2 ;a = b + c</pre> |

- R0 argomenti, valore restituito, variabili temporanee
- R1, R2 argomenti, variabili temporanee

Sommo il contenuto dei dati nel registro R1 con quello del registro 2, il tutto è assegnato come contenuto del registro R0

Istruzioni aritmetico-logiche: Esempi

| Codice ad alto livello | ARM Assembly code |
|------------------------|--|
| $a = b + c - d;$ | <pre>; R0 = a, R1 = b, R2 = c, R3 = d ; R4 = t ADD R4, R1, R2 ; t = b + c SUB R0, R4, R3 ; a = t - d</pre> |

- Non posso “fare” direttamente tutta l’espressione aritmetica
- Prima sommo R1 (a) e R2(b), e il risultato lo pongo in R4 (temporaneo)
- Poi sottraggo ad R4 il valore contenuto in R3 (d), il risultato lo pongo in R0
 - R0 argomenti, valore restituito, variabili temporanee
 - R1, R2 argomenti, variabili temporanee

Istruzioni aritmetico-logiche: costanti

Oltre ad operare su dati contenuti in registri posso usare anche **costanti**:

- Sono valori non contenuti in registri o in memoria RAM ma tali valori sono *immediatamente* disponibili
- Per tale motivo sono anche chiamati: **immediate**
- Abbiamo allora due tipi di operandi:
 - operandi con *indirizzamento a registro* (visti precedentemente)
 - operandi con indirizzamento immediato

Costanti: esempi

| Codice ad alto livello | ARM Assembly code |
|------------------------|--------------------------------------|
| a = b + 15; | ; R0 = a, R1 = b, ADD R0, R1, #15 |

| Codice ad alto livello | ARM Assembly code |
|------------------------|---------------------------------------|
| a = b + 15; | ; R0 = a, R1 = b, ADD R0, R1, #0xF |

Costanti: esempi

| Codice ad alto livello | ARM Assembly code |
|------------------------|--|
| $a = b + c - 15;$ | ; R0 = a, R1 = b, R2 = c, ; R4 = t ADD R4, R1, R2 ; t = b + c SUB R0, R4, #0xF ; a = t - 15 |

Istruzioni aritmetiche e costanti: Esempi

| Codice ad alto livello | ARM Assembly code |
|-----------------------------------|--|
| b=10; c=8; d=12 a = b + c - d; | ; R0 = a, R1 = b, R2 = c, R3 = d ; R4 = t MOV R1, #10 MOV R2, #8 MOV R3, #12 ADD R4, R1, R2 ; t = b + c SUB R0, R4, R3 ; a = t - d |

ARM Assembly: un nota

Etichette (Label)

- Il codice Assembly usa le etichette per indicare la locazione di istruzioni all'interno del programma
- Quando il codice Assembly è “traslato” in codice macchina le etichette sono trasformate nell'indirizzo di memoria della relativa istruzione a cui facevano riferimento.
- In ARM assembly le etichette non possono essere parole riservate del linguaggio Assembly.
- E' convenzione identare le istruzioni e NON identare le label

ARM Assembly: un nota

Esempio erichetta

```
MOV R2,#10 ; R2 = 10  
ADD R1, R2, #17 ; R1 = R2 + 17
```

.

.

.

TARGET

```
SUB R1, R1, #78 ; R1 = R1 - 78
```

ARM Assembly: altro esempio

```
.data.  
var1: .word 3  
var2: .word 4  
  
.text  
.global _start
```

_start:

```
ldr r0, adr_var1 ; load the memory address of var1 via label adr_var1 to R0  
ldr r1, adr_var2 ; load the memory address of var2 via label adr_var2 to R1  
ldr r2, [r0]      ;load the value at memory address found in R0 to R2  
str r2, [r1]      ;store the value found in R2 to the memory address in R1
```

```
adr_var1: .word var1  
adr_var2: .word var2
```

- In rosso “direttive” per il compilatore Assembly (non fanno parte del linguaggio).
- In blu delle etichette
- NOTA: I colori sono solo presenti in questa slide per evidenziare le differenti parti ma non sono parte del linguaggio!

ARM Assembly: altro esempio

.data

var1: .word 3

var2: .word 4

- Direttiva per mettere quelle linee di codice nella parte di memoria dedicata ai dati

.text

.global _start

- Direttiva per mettere quelle linee di codice nella parte di memoria dedicata alle istruzioni

_start:

```
ldr r0, adr_var1 ;load the memory address of var1 via label adr_var1 to R0  
ldr r1, adr_var2 ;load the memory address of var2 via label adr_var2 to R1  
ldr r2, [r0]      ;load the value (0x03) at memory address found in R0 to R2  
str r2, [r1]      ;store the value found in R2 (0x03) to the memory address in R1
```

adr_var1: .word var1

adr_var2: .word var2

ARM Assembly: altro esempio

Ricordo che in LDR il secondo operando è un indirizzo di memoria di una word il cui contenuto andrà nel primo operando (r0)

| R0 | R1 | R2 |
|---------|---------|-----|
| 0x10090 | 0x10094 | 0x3 |

1. ldr r0, adr_var1
2. ldr r1, adr_var2
3. ldr r2, [r0]
4. str r2, [r1] ; qui sovrascrivo il valore 0x4 con 0x3 (non è fatto vedere)

| memory address | Value | |
|----------------|---------|-----------|
| 0x10106 | 0x10090 | addr_var1 |
| 0x10102 | 0x10094 | addr_var2 |
| ... | | |
| 0x10094 | 0x4 | var2 |
| 0x10090 | 0x3 | var1 |
| ... | | |

Overview principali istruzioni aritmetico/logiche (1)

| Descrizione | Opcode | Sintassi | Semantica |
|--|--------|---|--|
| Addizione | ADD | ADD R _d , R ₁ , R ₂ /#imm | R _d = R ₁ + R ₂ /#imm |
| Sottrazione | SUB | SUB R _d , R ₁ , R ₂ /#imm | R _d = R ₁ - R ₂ /#imm |
| Moltiplicazione | MUL | MUL R _d , R ₁ , R ₂ /#imm | R _d = R ₁ x R ₂ /#imm |
| Carica nel registro | MOV | MOV R _d , R ₁ /#imm | R _d ← R ₁ /#imm |
| Carica negato nel registro | MVN | MVN R _d , R ₁ /#imm | R _d ← -(R ₁ /#imm) |
| AND logico | AND | AND R _d , R ₁ , R ₂ /#imm | R _d = R ₁ \wedge R ₂ /#imm |
| OR logico | ORR | ORR R _d , R ₁ , R ₂ /#imm | R _d = R ₁ \vee R ₂ /#imm |
| OR esclusivo | EOR | EOR R _d , R ₁ , R ₂ /#imm | R _d = R ₁ \oplus R ₂ /#imm |
| AND con complemento del secondo operando | BIC | BIC R _d , R ₁ , R ₂ /#imm | R _d = R ₁ \wedge \neg R ₂ /#imm |

Overview principali istruzioni aritmetico/logiche (2)

| Descrizione | Opcode | Sintassi | Semantica |
|-----------------------|--------|---|--|
| Shift logico sinistro | LSL | LSL R _d , R ₁ , R ₂ /#imm | R _d = R ₁ << R ₂ /#imm |
| Shift logico destro | LSR | LSR R _d , R ₁ , R ₂ /#imm | R _d = R ₁ >> R ₂ /#imm |
| Rotazione destra | ROR | ROR R _d , R ₁ , R ₂ /#imm | R _d = R ₁ \circlearrowright R ₂ /#imm |
| Confronto | CMP | CMP R ₁ , R ₂ | NZCV \leftarrow R ₁ - R ₂ |
| Negato del confronto | CMN | CMN R ₁ , R ₂ | NZCV \leftarrow R ₁ + R ₂ |

Istruzioni di shift

Le istruzioni di shift

- LSL (logical shift left), *(ricorda: entrano zeri)*
- LSR (logical shift right), *(ricorda: entrano zeri)*
- ASR (arithmetic shift right), *(ricorda, entra il MSB)*
- ROR (rotate right) *(ricorda: come pac-man)*

Non c'è un ROL perchè rotazioni a sinistra possono essere realizzate con rotazioni a destra con una quantità complementare:

- Consideriamo 5 bit
- **10100 Rotazione a destra di 5-3=2** → **01010** → **00101**
- **10100 Rotazione a sinistra di 5-2=3** → **01001** → **10010** → **00101**

Istruzioni di shift: con costanti (immediate)

| Source register | | | | |
|-----------------|-----------|-----------|-----------|-----------|
| R5 | 1111 1111 | 0001 1100 | 0001 0000 | 1110 0111 |

| Assembly Code | Result |
|-----------------|---|
| LSL R0, R5, #7 | R0 1000 1110 0000 1000 0111 0011 1000 0000 |
| LSR R1, R5, #17 | R1 0000 0000 0000 0000 0111 1111 1000 1110 |
| ASR R2, R5, #3 | R2 1111 1111 1110 0011 1000 0010 0001 1100 |
| ROR R3, R5, #21 | R3 1110 0000 1000 0111 0011 1111 1111 1000 |

Istruzioni di shift: con registro

Source registers

| | | | | |
|----|-----------|-----------|-----------|-----------|
| R8 | 0000 1000 | 0001 1100 | 0001 0110 | 1110 0111 |
| R6 | 0000 0000 | 0000 0000 | 0000 0000 | 0001 0100 |

Assembly code

LSL R4, R8, R6
ROR R5, R8, R6

Result

| | | | | |
|----|-----------|-----------|-----------|-----------|
| R4 | 0110 1110 | 0111 0000 | 0000 0000 | 0000 0000 |
| R5 | 1100 0001 | 0110 1110 | 0111 0000 | 1000 0001 |

Istruzioni di shift: moltiplicazioni/divisioni

Ricordiamo che:

- Shifting a value left by N is equivalent to multiplying it by 2^N .
- Likewise, arithmetically shifting a value right by N is equivalent to dividing it by 2^N ,

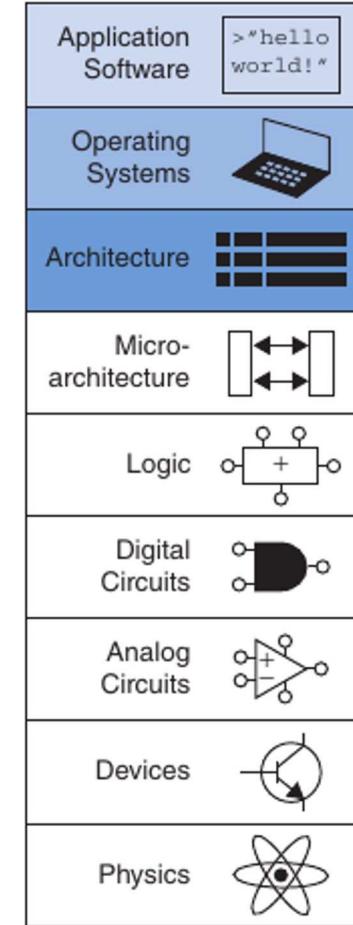
Architettura degli Elaboratori

Lezione 31

Docente: R.Prevete
a.a. 2022/2023
29 maggio 2023

Architettura

- Di cosa parleremo
 - Architettura ARM
 - ancora qualcosa sulle istruzioni su dati
 - Istruzioni logiche
 - Istruzioni condizionali e branching



Istruzioni di moltiplicazione: Attenzione!

- La moltiplicazione è differente dalle altre operazioni aritmetiche
 - Moltiplicare due numeri a 32-bit può produrre numeri a 64-bit
- L'architettura ARM fornisce istruzioni che producono un risultato a 32-bit o a 64-bit

Istruzioni di moltiplicazione: MUL, UMUL, SMUL

- MUL (Prodotto a 32-bit):
 - **MUL R1, R2, R3** moltiplica il valore di R2 con quello di R3 ed i 32-bit meno significativi del risultato sono posti in R1
 - I 32-bit più significativi del prodotto sono persi
- UMUL (Prodotto a 64-bit, su valori senza segno):
 - **UMULL R1, R2, R3, R4** realizza una moltiplicazione tra R3, ed R4 (unsigned) ed il risultato è messo in R1 e R2. R1 i 32-bit meno significativi, R2 i 32-bit più significativi
- SMUL (Prodotto a 64-bit, su valori con segno):
 - **SMULL R1, R2, R3, R4** realizza una moltiplicazione tra R3, ed R4 (signed) ed il risultato è messo in R1 e R2. R1 i 32-bit meno significativi, R2 i 32-bit più significativi

Istruzioni logiche

Le istruzioni logiche includono

- AND,
- ORR (OR) ,
- EOR (XOR),
- BIC (bit clear).

Tali operazione agiscono bitwise (bit per bit) su due operandi e pongono il risultato in un registro di destinazione

Istruzioni logiche

| Source registers | | | | |
|------------------|-----------|-----------|-----------|-----------|
| R1 | 0100 0110 | 1010 0001 | 1111 0001 | 1011 0111 |
| R2 | 1111 1111 | 1111 1111 | 0000 0000 | 0000 0000 |

Assembly code

AND R3, R1, R2

R3

ORR R4, R1, R2

R4

EOR R5, R1, R2

R5

BIC R6, R1, R2

R6

MVN R7, R2

R7

Result

| | | | |
|-----------|-----------|-----------|-----------|
| 0100 0110 | 1010 0001 | 0000 0000 | 0000 0000 |
| 1111 1111 | 1111 1111 | 1111 0001 | 1011 0111 |
| 1011 1001 | 0101 1110 | 1111 0001 | 1011 0111 |
| 0000 0000 | 0000 0000 | 1111 0001 | 1011 0111 |
| 0000 0000 | 0000 0000 | 1111 1111 | 1111 1111 |

- Operazione bit-wise
- Il primo operando (R1) è sempre un registro, il secondo può anche essere un immediate

Istruzioni logiche

| Source registers | | | | | |
|------------------|----|-----------|-----------|-----------|-----------|
| | R1 | 0100 0110 | 1010 0001 | 1111 0001 | 1011 0111 |
| | R2 | 1111 1111 | 1111 1111 | 0000 0000 | 0000 0000 |

Assembly code

AND R3, R1, R2
ORR R4, R1, R2
EOR R5, R1, R2
BIC R6, R1, R2
MVN R7, R2

Result

| | | | | | |
|----------------|----|-----------|-----------|-----------|-----------|
| AND R3, R1, R2 | R3 | 0100 0110 | 1010 0001 | 0000 0000 | 0000 0000 |
| ORR R4, R1, R2 | R4 | 1111 1111 | 1111 1111 | 1111 0001 | 1011 0111 |
| EOR R5, R1, R2 | R5 | 1011 1001 | 0101 1110 | 1111 0001 | 1011 0111 |
| BIC R6, R1, R2 | R6 | 0000 0000 | 0000 0000 | 1111 0001 | 1011 0111 |
| MVN R7, R2 | R7 | 0000 0000 | 0000 0000 | 1111 1111 | 1111 1111 |

- **MVN (MoVeNot):**

- Realizza un NOT bit-wise sul secondo operando (R2, registro o costante) e muove il risultato sul primo operando (R7)

Istruzioni logiche: alcune note

BIC

- The bit clear (BIC) instruction is useful for masking bits (i.e., forcing unwanted bits to 0).
- BIC R6, R1, R2 computes R1 AND (NOT R2). In other words, BIC clears the bits that are asserted in R2.
- Any subset of register bits can be masked.
- In the example, the most significant two bytes of R1 are cleared or masked, and the unmasked bottom two bytes of R1, 0xF1B7, are placed in R6.

| Source registers | | | |
|------------------|-----------|-----------|-----------|
| R1 | 0100 0110 | 1010 0001 | 1111 0001 |
| R2 | 1111 1111 | 1111 1111 | 0000 0000 |
| R6 | 0000 0000 | 0000 0000 | 1011 0111 |

BIC R6, R1, R2

Istruzioni logiche: alcune note

ORR

The ORR instruction is useful for combining bitfields from two registers. For example, 0x347A0000 ORR 0x000072FC = 0x347A72FC.

| Source registers | | | | | |
|------------------|-----------|-----------|-----------|-----------|-----------|
| R1 | 0100 0110 | 1010 0001 | 1111 0001 | 1011 0111 | |
| R2 | 1111 1111 | 1111 1111 | 0000 0000 | 0000 0000 | |
| ORR R4, R1, R2 | R4 | 1111 1111 | 1111 1111 | 1111 0001 | 1011 0111 |

Istruzioni logiche: alcune note

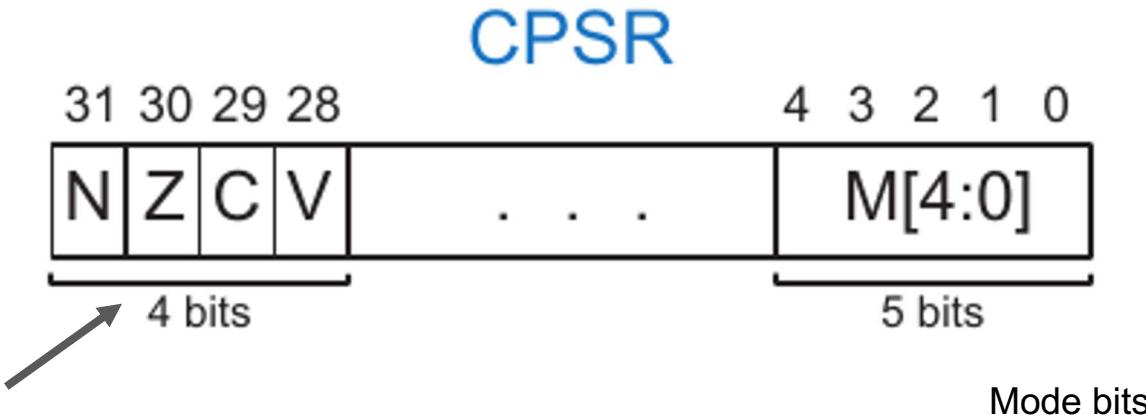
ORR

The ORR instruction is useful for combining bitfields from two registers. For example, 0x347A0000 ORR 0x000072FC = 0x347A72FC.

| Source registers | | | | | |
|------------------|-----------|-----------|-----------|-----------|-----------|
| R1 | 0100 0110 | 1010 0001 | 1111 0001 | 1011 0111 | |
| R2 | 1111 1111 | 1111 1111 | 0000 0000 | 0000 0000 | |
| ORR R4, R1, R2 | R4 | 1111 1111 | 1111 1111 | 1111 0001 | 1011 0111 |

Istruzioni condizionali e branching

Ricordiamo: Current Program Status Register (CPSR)



- negative (N),
- zero (Z),
- carry (C),
- overflow (V),

Istruzioni condizionali e branching

Ricordiamo: Current Program Status Register (CPSR)



- Il modo più comune di assegnare un valore ai bit di stato è usare l'istruzione CMP:
 - CMP R2, R3 Sottraggo il secondo operando dal primo e il risultato assegna dei valori ai flag
 - Se R2 e R3 sono uguali Z=1

Istruzioni condizionali e branching

- **Istruzione CMP:**

- CMP R2, R3
- Differenti condizioni

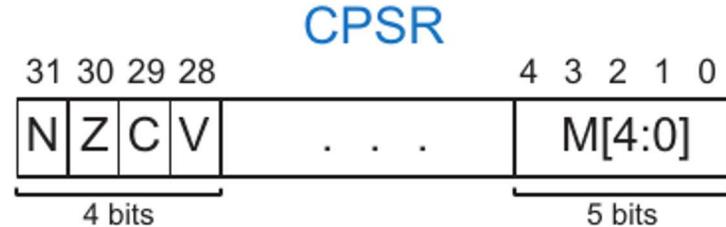
Ad esempio:



| | |
|--|---------------------|
| Equal | $Z==1$ |
| Not Equal | $Z==0$ |
| Signed Greater Than | $(Z==0) \&& (N==V)$ |
| Signed Less Than | $N!=V$ |
| Signed Greater Than or Equal | $N==V$ |
| Signed Less Than or Equal | $(Z==1) (N!=V)$ |
| Unsigned Higher or Same (or Carry Set) | $C==1$ |
| Unsigned Lower (or Carry Clear) | $C==0$ |

CMP: Analizziamo i vari casi

- **Istruzione CMP:**
 - CMP R2, R3
 - Sottraggo il secondo operando dal primo: R2-R32
- Se R2 e R3 sono senza segno
 - Devo sottrarre
- Vediamo cosa significa sottrarre due unsigned...



- negative (N),
- zero (Z),
- carry (C),
- overflow (V),

CMP: sottrazione tra due unsigned

$$Y_i = A_i - B_i$$

Borrow (prestito richiesto)

A_i

B_i

Y_i

| | | | |
|--|---|---|---|
| | 1 | 1 | 0 |
| | 1 | 0 | 0 |
| | 0 | 1 | 1 |
| | 0 | 0 | 1 |

$$4-3=1$$

| | | | |
|--|---|---|---|
| | 1 | 1 | 0 |
| | 1 | 1 | 0 |
| | 0 | 1 | 1 |
| | 0 | 1 | 1 |

$$6-3=3$$

| | | | |
|--|---|---|---|
| | 1 | 1 | 0 |
| | 1 | 0 | 0 |
| | 0 | 0 | 1 |
| | 0 | 1 | 1 |

$$4-1=3$$

Borrow (prestito richiesto)

A_i

B_i

Y_i

| | | | |
|---|---|---|---|
| 1 | 1 | 0 | 0 |
| 1 | 0 | | 1 |
| 1 | 1 | 1 | |
| 1 | 1 | 0 | |

$$5-7=-2$$

| | | | |
|---|---|---|---|
| 1 | 1 | 0 | 0 |
| 0 | 0 | | 1 |
| 0 | 1 | 0 | |
| 1 | 1 | 1 | |

 Borrow -out

$$2^n - 2^{(n-1)} = 2^1(2^{(n-1)}) - 2^{(n-1)} = 2^{(n-1)}$$

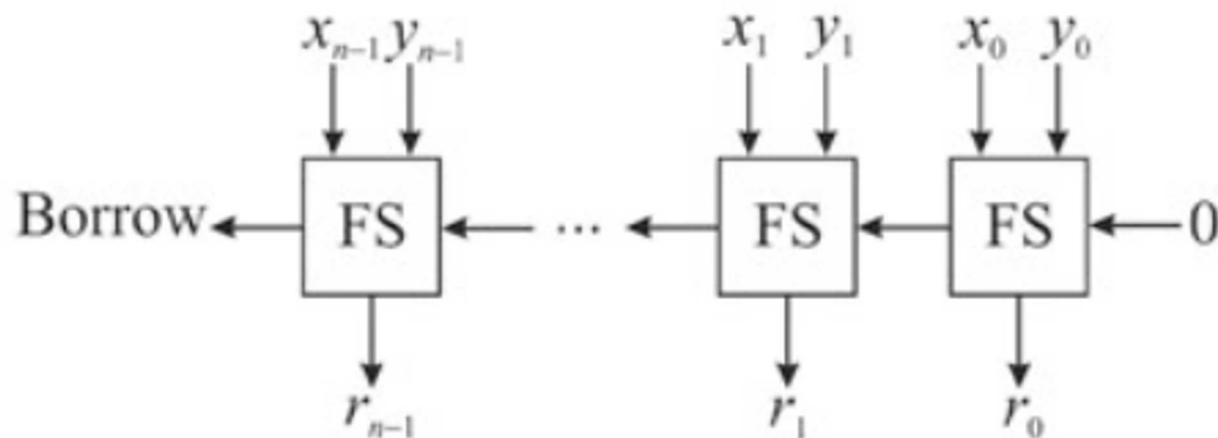
$$1-2=-1$$

CMP: sottrazione tra due unsigned

| A | B | Prestito effettuato | Y | Prestito da richiedere |
|----------|---|---------------------|----------|------------------------|
| Borrow - | | | Borrow + | |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

CMP: sottrazione tra due unsigned

- Puo' essere realizzato in maniera analoga al ripple-adder



Quando c'e' un Borrow out sto sottraendo con R3< R2, e diventa
C=1

| CPSR | | | |
|------|----|----|----|
| 31 | 30 | 29 | 28 |
| N | Z | C | V |

4 bits

| | | | |
|---|---|---|--------|
| . | . | . | M[4:0] |
| 4 | 3 | 2 | 1 |

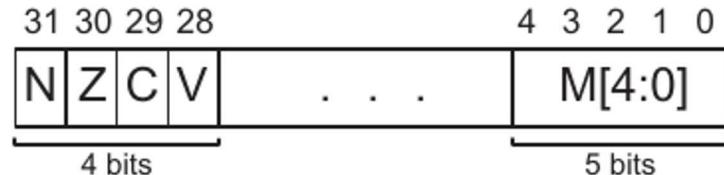
5 bits

CMP: con unsigned

- **Istruzione CMP:**

- CMP R2, R3
- Sottraggo il secondo operando dal primo: R2-R3

CPSR



- Se R2 e R3 sono senza segno
 - Se $R3=R2$ ($R2-R3$) $N=0, Z=1, C=0, V=0$
 - Se $R2 > R3$ ($R2-R3$) $N=0, Z=0, C=0, V=0$
 - Se $R2 < R3$ ($R2-R3$) $N=0, Z=0, C=1, V=0$
- Vediamo cosa accade quando sono con segno...

- negative (N),
- zero (Z),
- carry (C),
- overflow (V),

CMP: Con segno, analizziamo i vari casi

- **Istruzione CMP:**
 - CMP R2, R3
 - Sottraggo il secondo operando dal primo: $R2 - R3$
- Se R2 e R3 sono con segno
 - Se R2 e R3 sono entrambi positivi non posso andare in overflow
 - Se $R2 > R3$ ($R2 - R3$) $N=0, Z=0, C=0, V=0$
 - Se $R2 < R3$ ($R2 - R3$) $\textcolor{red}{N=1}, Z=0, C=0, V=0$
 - Se R2 e R3 sono entrambi negativi non posso andare in overflow
 - Se $R2 > R3$ ($R2 - R3$) $N=0, Z=0, C=0, V=0$
 - Es. $R3=-6, R2=-3$ ($R2 - R3) = -3 + 6 > 0$
 - Se $R2 < R3$ ($R2 - R3$) $\textcolor{red}{N=1}, Z=0, C=0, V=0$
 - Es. $R3=-3, R2=-6$ ($R2 - R3) = -6 + 3 = -3 < 0$



- negative (N),
- zero (Z),
- carry (C),
- overflow (V),

CMP: Analizziamo i vari casi

- **Istruzione CMP:**

- CMP R2, R3
- Sottraggo il secondo operando dal primo: R2-R3



- Se R2 e R3 sono con segno
- Se R2 è positivo ed R3 negativo:
 - Sicuramente $R2 > R3$
 - $R2 - R3$ equivale a sommare due positivi,
 - O vado in overflow ed il risultato è negativo **N=1, V=1**
 - oppure ho un risultato positivo **N=0, V=0**
- Esempio overflow: $R2=0111$ $R3=1000$ ($R2-R3$) deve essere un numero positivo perchè R2 è positivo ed R3 negativo
- $R2-R3 \rightarrow 0111 - 1000 \rightarrow 0111 + (0111+0001) =$
- $0111 + 1000 = 1111$ ottengo un risultato negativo!!! **OVERFLOW!**

- negative (N),
- zero (Z),
- carry (C),
- overflow (V),

CMP: Analizziamo i vari casi

- **Istruzione CMP:**

- CMP R2, R3
- Sottraggo il secondo operando dal primo: R2-R3



- Se R2 e R3 sono con segno
- Se R2 è negativo ed R3 positivo:
 - Sicuramente $R3 > R2$
 - $R2 - R3$ equivale a sommare due negativi,
 - O vado in overflow ed il risultato è positivo **N=0, V=1**
 - oppure ho un risultato negativo **N=1, V=0**
- Esempio overflow: $R2=1000$ $R3=0111$ ($R2-R3$) deve essere un numero negativo perché R2 è negativo ed R3 positivo
- $R2-R3 \rightarrow 1000 - 0111 \rightarrow 1000 + (1000+0001) =$
- $1000 + 1001 = 0001$ ottengo un risultato negativo!!! **OVERFLOW!**

- negative (N),
- zero (Z),
- carry (C),
- overflow (V),

CMP: Analizziamo i vari casi

RIASSUMENDO: SENZA SEGNO

R3 = R2

- N=0, **Z=1**, C=0, V=0

R2 > R3

- N=0, Z=0, C=0, V=0

R2 < R3

- N=0, Z=0, **C=1**, V=0



RIASSUMENDO: CON SEGNO

R3 = R2

- N=0, **Z=1**, C=0, V=0
- N=0, **Z=1, C=1**, V=0

R2 > R3

- N=0, Z=0, C=0, V=0
- **N=1**, Z=0, C=0, **V=1**

R2 < R3

- **N=1**, Z=0, C=0, V=0
- **N=0**, Z=0, C=0, **V=1**

- negative (N),
- zero (Z),
- carry (C),
- overflow (V),

| | UNSIGNED | SIGNED |
|---------|-------------------|-----------------|
| R3 = R2 | Z | Z |
| R2 > R3 | $\bar{Z} \bar{C}$ | \bar{Z} (N=V) |
| R2 < R3 | C | $\bar{N}=V$ |

Istruzioni condizionali e branching



- Altre istruzioni su dati (mnemonic) possono assegnare dei valori ai flag di stato quando sono seguite dal simbolo “S”.
- Ad esempio:
 - SUBS R2, R3, R7 Sottrae il valore di R7 da R3, il risultato è messo in R2, e i flag di stato sono aggiornati

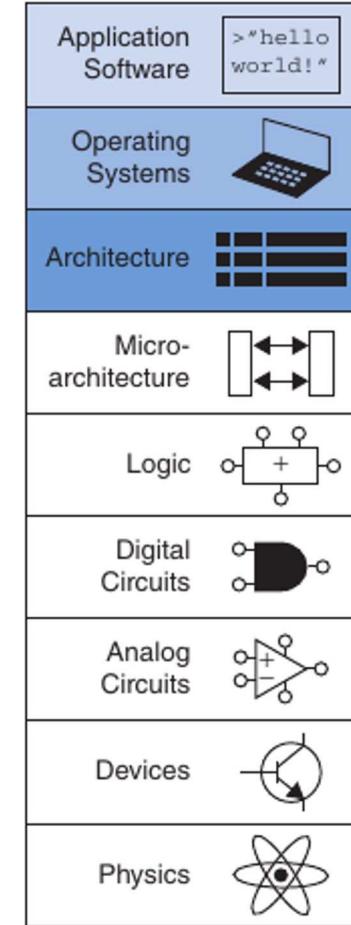
Architettura degli Elaboratori

Lezione 32

Docente: R.Prevete
a.a. 2022/2023
31 mag 2023

Architettura

- Di cosa parleremo
 - Architettura ARM
 - Istruzioni con condizione
 - Istruzioni di Branching
 - Condizionali
 - Iterazioni



Istruzioni condizionali e branching



- Ci sono altre istruzione capace di modificare i flag di stato come CMP
- Ad esempio: SUBS
 - SUBS R2, R3, R7
 - Sottrae R7 da R3, poi mette il risultato in R2 ($R2=R3-R7$)
 - Infine modifica i condition flags N, Z, C, V (ad esempio se il risultato è negativo **N=1**)

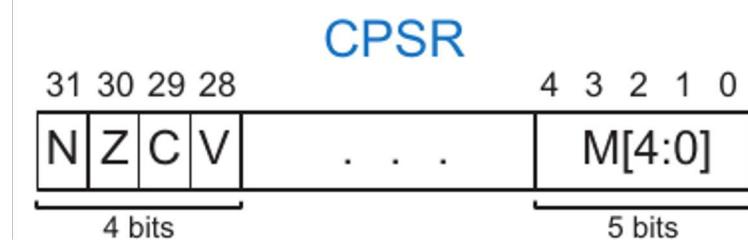
Istruzioni condizionali e branching

- In generale istruzioni con suffisso S permettono di modificare i flag di condizione



| Type | Instructions | Condition Flags |
|----------|--|-----------------|
| Add | ADDS, ADCS | N, Z, C, V |
| Subtract | SUBS, SBCS, RSBS, RSCS | N, Z, C, V |
| Compare | CMP, CMN | N, Z, C, V |
| Shifts | ASRS, LSLS, LSRS, RORS, RRXS | N, Z, C |
| Logical | ANDS, ORRS, EORS, BICS | N, Z, C |
| Test | TEQ, TST | N, Z, C |
| Move | MOVS, MVNS | N, Z, C |
| Multiply | MULS, MLAS, SMLALS, SMULLS, UMLALS, UMULLS | N, Z |

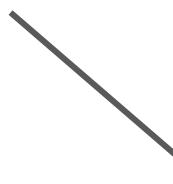
Istruzioni condizionali e branching



- Un'istruzione capace di modificare i flag di stato come CMP e SUBS è in genere seguita da una *istruzione condizionale*
- Una istruzione condizionale è costruita come una istruzione seguita da una condition mnemonic (*messa come suffisso*). Ad esempio:
 - $ADD + EQ \rightarrow ADDEQ$
 - $ADD + LT \rightarrow ADDLT$
 - $ADD + NE \rightarrow ADDNE$

Istruzioni condizionali e branching

SUFFISSI (alcuni esempi)



| Condition Code | Meaning (for cmp or subs) | Status of Flags |
|----------------|--|---------------------------|
| EQ | Equal | $Z==1$ |
| NE | Not Equal | $Z==0$ |
| GT | Signed Greater Than | $(Z==0) \&\& (N==V)$ |
| LT | Signed Less Than | $N!=V$ |
| GE | Signed Greater Than or Equal | $N==V$ |
| LE | Signed Less Than or Equal | $(Z==1) \parallel (N!=V)$ |
| CS or HS | Unsigned Higher or Same (or Carry Set) | $C==1$ |
| CC or LO | Unsigned Lower (or Carry Clear) | $C==0$ |

Istruzioni condizionali

ESEMPIO

| Condition Code | Meaning (for cmp or subs) | Status of Flags |
|----------------|--|----------------------|
| EQ | Equal | $Z==1$ |
| NE | Not Equal | $Z==0$ |
| GT | Signed Greater Than | $(Z==0) \&\& (N==V)$ |
| LT | Signed Less Than | $N!=V$ |
| GE | Signed Greater Than or Equal | $N==V$ |
| LE | Signed Less Than or Equal | $(Z==1) (N!=V)$ |
| CS or HS | Unsigned Higher or Same (or Carry Set) | $C==1$ |
| CC or LO | Unsigned Lower (or Carry Clear) | $C==0$ |

```
CMP      R2 , R3  
ADDEQ   R4 , R5 , #78  
ANDHS   R7 , R8 , R9  
ORRMI   R10, R11, R12  
EORLT   R12, R7 , R10
```

Istruzioni condizionali

- Compare: R2-R3 (signed)
 - $R3=R2 \rightarrow Z$
 - $R2 > R3 \rightarrow \bar{Z} (N=V)$
 - $R2 < R3 \rightarrow \bar{N}=V$
- Compare: R2-R3 (unsigned)
 - $R3=R2 \rightarrow Z$
 - $R2 > R3 \rightarrow \bar{Z} \bar{C}$
 - $R2 < R3 \rightarrow C$

| | |
|-------|-----------------|
| CMP | R2 , R3 |
| ADDEQ | R4 , R5 , #78 |
| ANDHS | R7 , R8 , R9 |
| ORRMI | R10 , R11 , R12 |
| EORLT | R12 , R7 , R10 |

Istruzioni condizionali

- ADDEQ è eseguita solo quando
 - $Z=1$ ($R3=R2$)

| | |
|-------|-----------------|
| CMP | R2 , R3 |
| ADDEQ | R4 , R5 , #78 |
| ANDHS | R7 , R8 , R9 |
| ORRMI | R10 , R11 , R12 |
| EORLT | R12 , R7 , R10 |

- Compare: R2-R3 (signed)
 - $R3=R2 \rightarrow Z$
 - $R2>R3 \rightarrow \bar{Z} (N=V)$
 - $R2 < R3 \rightarrow \bar{N}=V$
- Compare: R2-R3 (unsigned)
 - $R3=R2 \rightarrow Z$
 - $R2>R3 \rightarrow \bar{Z} \bar{C}$
 - $R2 < R3 \rightarrow C$

Istruzioni condizionali

- ANDHS è eseguita solo quando
 - C=1 (R2 < R3 unsigned)



```
CMP      R2 , R3
ADDEQ   R4 , R5 , #78
ANDHS   R7 , R8 , R9
ORRMI   R10, R11, R12
EORLT   R12, R7 , R10
```

- Compare: R2-R3 (signed)
 - R3=R2 → Z
 - R2>R3 → \bar{Z} (N=V)
 - R2 < R3 → $\bar{N}=V$
- Compare: R2-R3 (unsigned)
 - R3=R2 → Z
 - R2>R3 → $\bar{Z} \bar{C}$
 - R2 < R3 → C

Istruzioni condizionali

- ORRMI è eseguita solo quando
 - $N=1$ ($R2 - R3$ negativo)



```
CMP      R2 , R3
ADDEQ   R4 , R5 , #78
ANDHS   R7 , R8 , R9
ORRMI   R10, R11, R12
EORLT   R12, R7 , R10
```

- Compare: $R2-R3$ (signed)
 - $R3=R2 \rightarrow Z$
 - $R2>R3 \rightarrow \bar{Z} (N=V)$
 - $R2 < R3 \rightarrow \bar{N}=V$

- Compare: $R2-R3$ (unsigned)
 - $R3=R2 \rightarrow Z$
 - $R2>R3 \rightarrow \bar{Z} \bar{C}$
 - $R2 < R3 \rightarrow C$

Istruzioni condizionali

- EORLT (XOR CONDIZIONATO)
è eseguito solo quando
 - N=1 V=0, oppure N=0, V=1
(R2 < R3)



```
CMP      R2 , R3
ADDEQ   R4 , R5 , #78
ANDHS   R7 , R8 , R9
ORRMI   R10, R11, R12
EORLT   R12, R7, R10
```

- Compare: R2-R3 (signed)
 - R3=R2 → Z
 - R2>R3 → \bar{Z} (N=V)
 - R2 < R3 → $\bar{N}=V$
- Compare: R2-R3 (unsigned)
 - R3=R2 → Z
 - R2>R3 → $\bar{Z} \bar{C}$
 - R2 < R3 → C

Istruzioni di branching

Ricordo che:

- Un programma di solito viene eseguito in sequenza, con il registro **prossima istruzione (PC)** **incrementando di 4 dopo ogni istruzione per puntare all'istruzione successiva** (Ricorda che le istruzioni sono lunghe 4 byte e ARM è un byte-architettura indirizzata.)
- Le istruzioni Branch cambiano il registro prossima istruzione

Istruzioni di branching

Abbiamo tre tipologie di istruzioni di branching

- B <label> (incondizionata)
- B[condizione] <label> (condizionate)
 - BEQ <label>
 - BNE <label>
- BL branch e link. Usata per le chiamate di funzioni (vedremo in seguito)

Istruzioni di branching: incondizionata

Indirizzi di memoria

| |
|----|
| 0 |
| 4 |
| 8 |
| 12 |
| |
| 16 |

```
ADD R1, R2, #17      ; R1 = R2 + 17
B    TARGET          ; branch to TARGET
ORR R1, R1, R3       ; not executed
AND R3, R1, #0xFF    ; not executed
TARGET
SUB R1, R1, #78      ; R1 = R1 - 78
```

PC=0 → 4 → 16

Istruzioni di branching: condizionata

```
MOV R0, #4          ; R0 = 4
ADD R1, R0, R0      ; R1 = R0 + R0 = 8
CMP R0, R1          ; set flags based on R0-R1 = -4. NZCV = 1000
BEQ THERE           ; branch not taken (Z != 1)
ORR R1, R1, #1       ; R1 = R1 OR 1 = 9
```

THERE

```
    ADD R1, R1, #78   ; R1 = R1 + 78 = 87
```

```
If (R0!=R1)
    R1 = R1 | 1;
R1=R1+78;
```

Istruzioni di branching: condizionata

All'istruzione di branching **B** può essere aggiunto un suffisso di condizione come per le altre istruzioni

Table 6.3 Condition mnemonics

| Mnemonic | Name | CondEx |
|--------------|-------------------------------------|-----------------------|
| EQ | Equal | Z |
| NE | Not equal | \bar{Z} |
| CS/HS | Carry set / unsigned higher or same | C |
| CC/LO | Carry clear / unsigned lower | \bar{C} |
| MI | Minus / negative | N |
| PL | Plus / positive or zero | \bar{N} |
| VS | Overflow / overflow set | V |
| VC | No overflow / overflow clear | \bar{V} |
| HI | Unsigned higher | ZC |
| LS | Unsigned lower or same | Z OR \bar{C} |
| GE | Signed greater than or equal | $\bar{N} \oplus V$ |
| LT | Signed less than | $N \oplus V$ |
| GT | Signed greater than | $Z(\bar{N} \oplus V)$ |
| LE | Signed less than or equal | Z OR ($N \oplus V$) |
| AL (or none) | Always / unconditional | Ignored |

Istruzioni di branching: IF/ELSE

High-Level Code

```
if (apples == oranges)
    f = i + 1;

f = f - i;
```

ARM Assembly Code

```
; R0 = apples, R1 = oranges, R2 = f, R3 = i
CMP R0, R1          ; apples == oranges ?
BNE L1              ; if not equal, skip if block
ADD R2, R3, #1       ; if block: f = i + 1
L1
SUB R2, R2, R3       ; f = f - i
```

Semplice IF

NE NOT EQUAL

Istruzioni di branching: IF/ELSE

High-Level Code

```
if (apples == oranges)
    f = i + 1;

else
    f = f - i;
```

ARM Assembly Code

```
; R0 = apples, R1 = oranges, R2 = f, R3 = i
CMP R0, R1          ; apples == oranges?
BNE L1              ; if not equal, skip if block
ADD R2, R3, #1       ; if block: f = i + 1
B   L2               ; skip else block
L1
SUB R2, R2, R3       ; else block: f = f - i
L2
```

BNE L1
BEQ L2

L1

qualscosa 1
B fine

L2

qualscosa 2
fine

IF ...ELSE...

Istruzioni di branching: IF/ELSE

Notiamo che:

- A volte le istruzioni di branching possono essere sostituite direttamente con istruzioni con condizione
- In genere in questo modo si ottiene un codice più efficiente
- Vediamo degli esempi....

IF/ELSE: istruzioni con condizione

High-Level Code

```
if (apples == oranges)
    f = i + 1;

f = f - i;
```

; R3=i, R2=f

| | |
|------------------|---------------------------------------|
| CMP R0, R1 | ; apples == oranges ? |
| ADDEQ R2, R3, #1 | ; f = i + 1 on equality (i.e., Z = 1) |
| SUB R2, R2, R3 | ; f = f - i |

Semplice IF

IF/ELSE: istruzioni con condizione

High-Level Code

```
if (apples == oranges)
    f = i + 1;

else
    f = f - i;
```

; R3=i, R2=f

| | |
|------------------|--|
| CMP R0, R1 | ; apples == oranges? |
| ADDEQ R2, R3, #1 | ; f = i + 1 on equality (i.e., Z = 1) |
| SUBNE R2, R2, R3 | ; f = f - i on not equal (i.e., Z = 0) |

IF/ELSE

Istruzioni di branching: switch/case

High-Level Code

```
switch (button) {  
    case 1: amt = 20; break;  
  
    case 2: amt = 50; break;  
  
    case 3: amt = 100; break;  
  
    default: amt = 0;  
}
```

ARM Assembly Code

```
; R0 = button, R1 = amt  
CMP R0, #1           ; is button 1 ?  
MOVEQ R1, #20         ; amt = 20 if button is 1  
BEQ DONE             ; break  
  
CMP R0, #2           ; is button 2 ?  
MOVEQ R1, #50         ; amt = 50 if button is 2  
BEQ DONE             ; break  
  
CMP R0, #3           ; is button 3?  
MOVEQ R1, #100        ; amt = 100 if button is 3  
BEQ DONE             ; break  
  
MOV R1, #0            ; default amt = 0  
DONE
```

Istruzioni di branching: iterazioni

High-Level Code

```
int pow = 1;  
int x = 0;  
  
while (pow != 128) {  
    pow = pow * 2;  
    x = x + 1;  
}
```

Iterazione con pre-condizione (WHILE)

ARM Assembly Code

```
; R0 = pow, R1 = x  
MOV R0, #1          ; pow = 1  
MOV R1, #0          ; x = 0  
  
WHILE  
    CMP R0, #128     ; pow != 128 ?  
    BEQ DONE         ; if pow == 128, exit loop  
    LSL R0, R0, #1    ; pow = pow * 2  
    ADD R1, R1, #1    ; x = x + 1  
    B    WHILE        ; repeat loop  
  
DONE
```

Istruzioni di branching: iterazioni

High-Level Code

```
int i;  
int sum = 0;  
  
for (i = 0; i < 10; i = i + 1) {  
    sum = sum + i;  
}
```

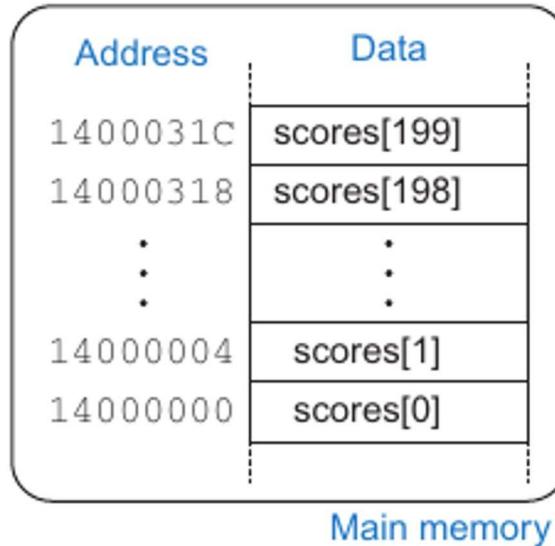
Numero fissato di
iterazioni

ARM Assembly Code

```
; R0 = i, R1 = sum  
MOV R1, #0          ; sum = 0  
MOV R0, #0          ; i = 0          loop initialization  
  
FOR  
    CMP R0, #10      ; i < 10 ?      check condition  
    BGE DONE         ; if (i >= 10) exit loop  
    ADD R1, R1, R0    ; sum = sum + i  loop body  
    ADD R0, R0, #1    ; i = i + 1      loop operation  
    B   FOR          ; repeat loop  
DONE
```

Memoria ed iterazioni

Supponiamo che 200 valori siano memorizzati in memoria in maniera consecutiva (array):



Vogliamo accedere a tutti i 200 valori. Come fare?

Memoria ed iterazioni

Supponiamo che 200 valori siano memorizzati in memoria in maniera consecutiva (array):



Ricordiamo:

| Mode | ARM Assembly | Address | Base Register |
|------------|-------------------|---------|---------------|
| Offset | LDR R0, [R1, R2] | R1 + R2 | Unchanged |
| Pre-index | LDR R0, [R1, R2]! | R1 + R2 | R1 = R1 + R2 |
| Post-index | LDR R0, [R1], R2 | R1 | R1 = R1 + R2 |

Memoria ed iterazioni

High-Level Code

```
int i;  
int scores[200];  
...  
  
for (i = 0; i < 200; i = i + 1)  
    scores[i] = scores[i] + 10;
```

| Address | Data |
|----------|-------------|
| 1400031C | scores[199] |
| 14000318 | scores[198] |
| . | . |
| 14000004 | scores[1] |
| 14000000 | scores[0] |

Main memory

Branch con
Signed
Greater Than

Con post-index

ARM Assembly Code

```
; R0 = array base address  
; initialization code ...  
MOV R0, #0x14000000 ; R0 = base address  
ADD R1, R0, #800    ; R1 = base address + (200*4)  
  
LOOP:  
    CMP R0, R1          ; reached end of array?  
    BGE L3              ; if yes, exit loop  
    LDR R2, [R0]          ; R2 = scores[i]  
    ADD R2, R2, #10       ; R2 = scores[i] + 10  
    STR R2, [R0], #4      ; scores[i] = scores[i] + 10  
    ; then R0 = R0 + 4  
    B    LOOP             ; repeat loop  
L3
```

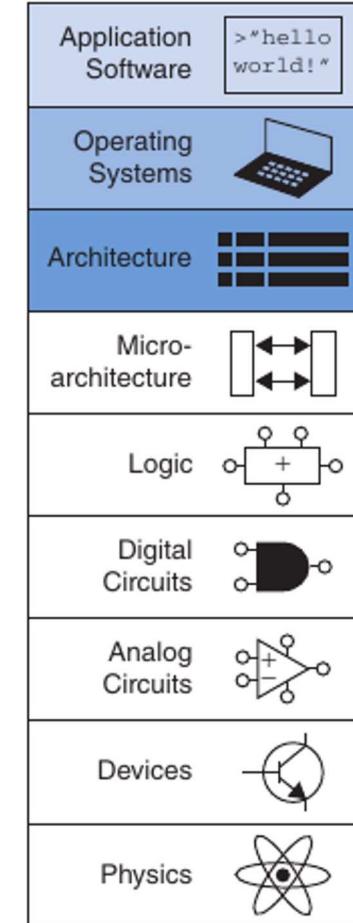
Architettura degli Elaboratori

Lezione 29

Docente: R.Prevete
a.a. 2021/2022

Architettura

- Di cosa parleremo
 - Architettura ARM
 - Ancora memoria ed iterazioni
 - Funzioni
 - Chiamata di funzione
 - Valore restituito
 - Stack
 - Esempi



Memoria ed iterazioni: altro esempio

Convertire un array di caratteri di lunghezza 10 da lowercase ad upper case

| Decimal | Hexadecimal | Binary | Octal | Char |
|---------|-------------|---------|-------|------|
| 65 | 41 | 1000001 | 101 | A |
| 66 | 42 | 1000010 | 102 | B |
| 67 | 43 | 1000011 | 103 | C |
| 68 | 44 | 1000100 | 104 | D |
| 69 | 45 | 1000101 | 105 | E |
| 70 | 46 | 1000110 | 106 | F |
| 71 | 47 | 1000111 | 107 | G |
| 72 | 48 | 1001000 | 110 | H |
| 73 | 49 | 1001001 | 111 | I |
| 74 | 4A | 1001010 | 112 | J |
| 75 | 4B | 1001011 | 113 | K |
| 76 | 4C | 1001100 | 114 | L |
| 77 | 4D | 1001101 | 115 | M |
| 78 | 4E | 1001110 | 116 | N |
| 79 | 4F | 1001111 | 117 | O |
| 80 | 50 | 1010000 | 120 | P |
| 81 | 51 | 1010001 | 121 | Q |
| 82 | 52 | 1010010 | 122 | R |
| 83 | 53 | 1010011 | 123 | S |
| 84 | 54 | 1010100 | 124 | T |
| 85 | 55 | 1010101 | 125 | U |
| 86 | 56 | 1010110 | 126 | V |
| 87 | 57 | 1010111 | 127 | W |
| 88 | 58 | 1011000 | 130 | X |
| 89 | 59 | 1011001 | 131 | Y |
| 90 | 5A | 1011010 | 132 | Z |

RICORDO:

Tabella ASCII è una associazione tra numeri e simboli alfanumerici

| Decimal | Hexadecimal | Binary | Octal | Char |
|---------|-------------|---------|-------|------|
| 97 | 61 | 1100001 | 141 | a |
| 98 | 62 | 1100010 | 142 | b |
| 99 | 63 | 1100011 | 143 | c |
| 100 | 64 | 1100100 | 144 | d |
| 101 | 65 | 1100101 | 145 | e |
| 102 | 66 | 1100110 | 146 | f |
| 103 | 67 | 1100111 | 147 | g |
| 104 | 68 | 1101000 | 150 | h |
| 105 | 69 | 1101001 | 151 | i |
| 106 | 6A | 1101010 | 152 | j |
| 107 | 6B | 1101011 | 153 | k |
| 108 | 6C | 1101100 | 154 | l |
| 109 | 6D | 1101101 | 155 | m |
| 110 | 6E | 1101110 | 156 | n |
| 111 | 6F | 1101111 | 157 | o |
| 112 | 70 | 1110000 | 160 | p |
| 113 | 71 | 1110001 | 161 | q |
| 114 | 72 | 1110010 | 162 | r |
| 115 | 73 | 1110011 | 163 | s |
| 116 | 74 | 1110100 | 164 | t |
| 117 | 75 | 1110101 | 165 | u |
| 118 | 76 | 1110110 | 166 | v |
| 119 | 77 | 1110111 | 167 | w |
| 120 | 78 | 1111000 | 170 | x |
| 121 | 79 | 1111001 | 171 | y |
| 122 | 7A | 1111010 | 172 | z |

Memoria ed iterazioni: altro esempio

Convertire un array di caratteri di lunghezza 10 da lowercase ad upper case

High-Level Code

```
// chararray[10] declared and initialized earlier  
int i;  
  
for (i = 0; i < 10; i = i + 1)  
    chararray[i] = chararray[i] - 32;
```

Memoria ed iterazioni: altro esempio

Convertire un array di caratteri di lunghezza 10 da lowercase ad upper case

ARM Assembly Code

```
; ARM assembly code
; R0 = base address of chararray (initialized earlier), R1 = i
        MOV    R1, #0          ; i = 0
LOOP      CMP    R1, #10         ; i < 10 ?
          BGE    DONE          ; if (i >=10), exit loop
          LDRB   R2, [R0, R1]    ; R2 = mem[R0+R1] = chararray[i]
          SUB    R2, R2, #32       ; R2 = chararray[i] - 32
          STRB   R2, [R0, R1]    ; chararray[i] = R2
          ADD    R1, R1, #1          ; i = i + 1
          B      LOOP          ; repeat loop
DONE
```

1 Byte alla volta!

[R0,R1] corrisponde all'indirizzo di memoria R0+R1

Funzioni

Cose a cui stare attenti:

- Come chiamare una funzione e cosa accade
- Ricordarsi il “punto” in cui la funzione è stata chiamata
- Salvare ciò che restituisce una funzione

Funzioni

Chiamata di funzione (prime considerazioni):

- Quando una funzione ne chiama un'altra, la **funzione chiamante, caller**, e la **funzione chiamata, callee**, devono condividere dove inserire gli argomenti e il valore restituito.
 - In ARM, la funzione chiamante, **caller**, usa convenzionalmente quattro registri **R0–R3** prima di effettuare la chiamata alla funzione, **callee**, e la **funzione chiamata** inserisce il valore restituito nel registro **R0** prima di terminare

Funzioni

Chiamata di funzione (prime considerazioni):

- L'architettura ARM usa:
 - L'istruzione di branch è «Braching Link», BL, per chiamare una funzione, tale istruzione:
 - 1) ci fa “saltare” all'istruzione indicata dalla label (cioè PC = all'indirizzo di memoria della label) e
 - 2) memorizza l'indirizzo di memoria della istruzione che segue BL nel Link Register (LR)
 - Al termine della chiamata di funzione il valore del Link Register (LR) è copiato nel registro prossima istruzione PC:
 - MOV PC, LR

Funzioni: esempio funzione vuota!

High-Level Code

```
int main() {  
    simple();  
    ...  
}  
  
// void means the function returns no value  
void simple() {  
    return;  
}
```

Funzioni: esempio funzione vuota!

ARM Assembly Code

0x00008000

...

0x00008020

...

0x0000902C

MAIN

...

...

BL SIMPLE

; call the simple function

SIMPLE MOV PC, LR

; return

IN LR= 0x0008024

IN PC= 0x0008024



Indirizzi di memoria delle istruzioni

Funzioni: un passo in più

High-Level Code

```
int somma();  
int main() {  
    int a=2, b=5, r=0;  
    r=somma(a,b);  
    return 0;  
}  
  
int somma (int a, int b) {  
    int s= a+b;  
    return s;  
}
```

ARM Assembly Code

```
main ;label  
    MOV R1, #2  
    MOV R2, #5  
    MOV R0, #0  
    BL somma  
    B fine  
somma ;label  
    ADD R3, R1, R2  
    MOV R0, R3  
    MOV PC, LR  
fine ;label
```

- R1 e R2 si comportano come argomenti della funzione
- R0 è dove memorizzo il valore restituito

- prossima istruzione nel main memorizzata in LR

Attenzione! Modifico registri che sono accessibili anche nella funzione chiamante

Funzioni: STACK

High-Level Code

```
int somma();  
int main() {  
    int a=2, b=5, r=0;  
    r=somma(a,b);  
    return 0;  
}  
  
int somma (int a, int b) {  
    int s= a+b;  
    return s;  
}
```

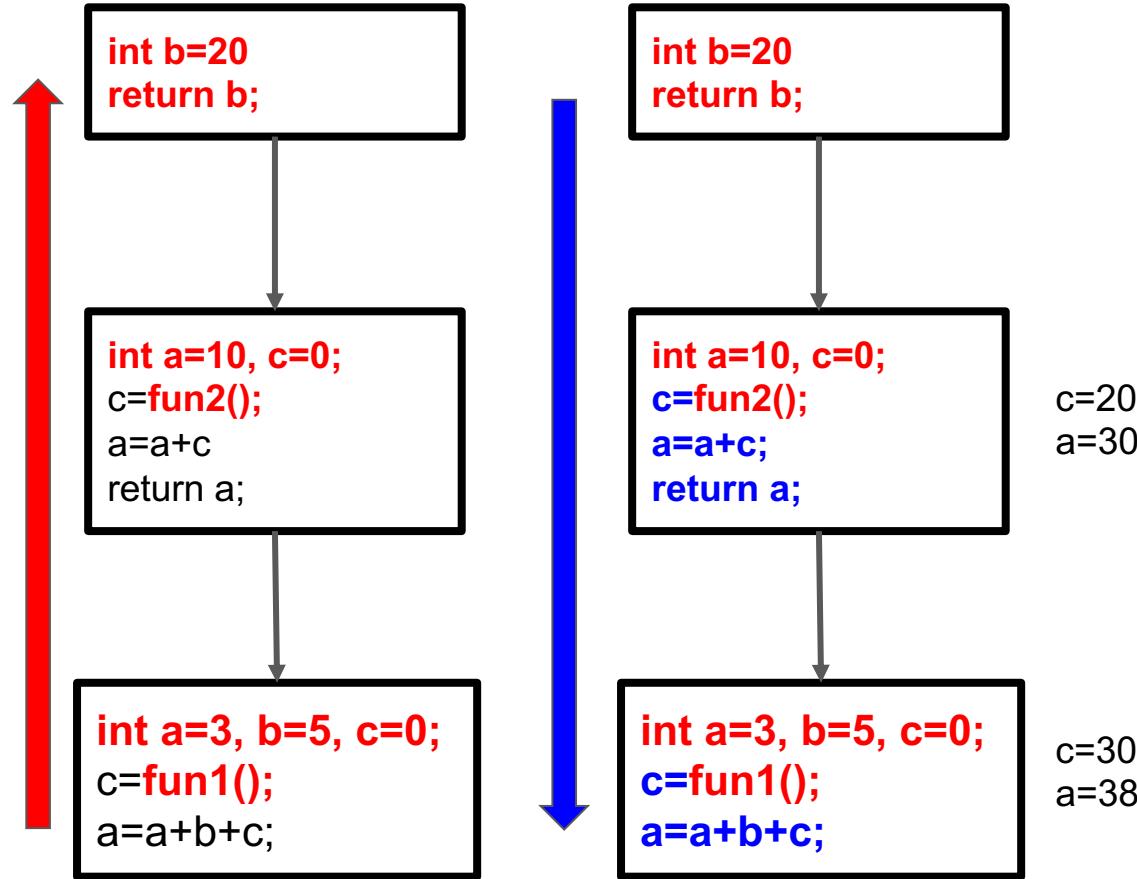
- Quando chiamo una funzione all'interno della funzione **NON** devo modificare le variabili della funzione (main ad esempio) in cui la funzione è chiamata
 - Nell'esempio di High-level code le variabili a, b e r non devono poter essere modificate all'interno della funzione *somma*.
- In ARM assembly questo significa che i devo preservare il contenuto dei registri
- Come lo faccio, salvando il contenuto in uno **stack**.

Funzioni: Esempio di STACK

Ciascuna variabile è locale, visibile, accessibile solo all'interno del box

```
int main() {  
    int a=3,b=5,c=0;  
    c=fun1();  
    a=a+b+c;  
    ...  
  
    return 0;  
}  
  
int fun2(){  
    int b=20;  
    return b;  
}  
  
int fun1(){  
    int a=10,c=0;  
    c=fun2();  
    a=a+c;  
  
    return a;  
}
```

High-Level Code



Funzioni: STACK

- Lo stack è la memoria utilizzata per salvare le informazioni all'interno di una funzione.
- Lo stack :
 - si **espande** (utilizza più memoria) quando il processore ha bisogno di più spazio in memoria
 - si **contrae** (utilizza meno memoria) quando il processore non ha più bisogno delle variabili lì memorizzate.
- Rappresenta una coda LIFO (Last-In-First-out), quindi una pila
- Ciascuna funzione alloca memoria (push) nello stack per rappresentare variabili locali, ma la dealloca (pop) quando la funzione termina
- L'ultima memoria allocata nello stack è il TOP

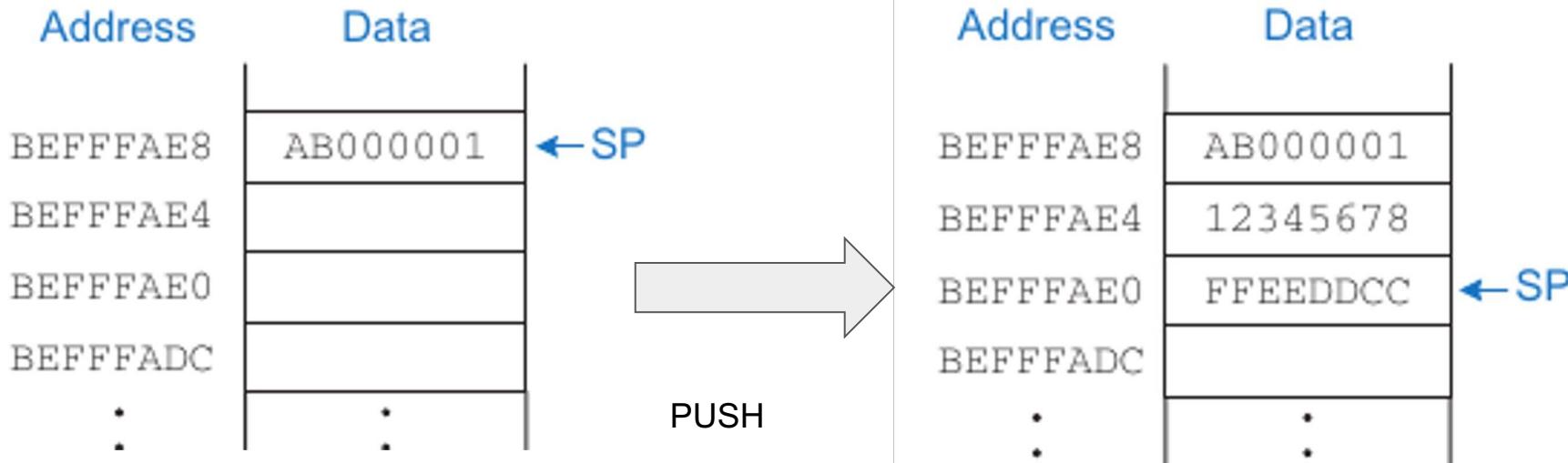
Funzioni: STACK



In ARM:

- C'è un registro Stack Pointer (SP o R13) che contiene l'indirizzo dell'ultima locazione di memoria immessa nello STACK
 - Nell'esempio SP contiene 0xBEFFFAE8
- Lo stack si espande verso il basso
 - Nell'esempio la memoria disponibile (che può ancora essere usata) è agli indirizzi
 - $0xBEFFFAE8 - 4 = 0xBEFFFAE4$
 - $0xBEFFFAE8 - 8 = 0xBEFFFAE0$
 - $0xBEFFFAE8 - 12 = 0xBEFFFADC$
 - ...e così via

Funzioni: STACK



SONO STATE INSERITE TWO WORD NELLO STACK

ESEMPIO:

```
MOV R4, #0x12345678  
MOV R5, #0xFFEEDDCC  
SUB SP, SP, #4  
STR R4, [SP]  
SUB SP, SP, #4  
STR R5, [SP]
```

Funzioni: STACK

Un uso fondamentale dello stack è di salvare e recuperare registri che sono usati in una funzione

Passi da seguire:

1. Creare spazio di memoria nello stack per memorizzare i valori dei registri usati nella funzione
2. Salvare nello stack i valori dei registri utilizzati nella funzione
3. Eseguire la funzione usando i registri salvati
4. Recuperare i valori originali dallo stack e opportunamente rimetterli nei registri
5. Deallocare la memoria dallo stack

Funzioni: STACK

Esempio

ARM Assembly Code

High-Level Code

```
int main() {  
    int y;  
    ...  
    y = diffofsums(2, 3, 4, 5);  
    ...  
}
```

```
int diffofsums(int f, int g, int h, int i) {  
    int result;  
  
    result = (f + g) - (h + i);  
    return result;  
}
```

```
; R4 = y  
MAIN  
    ...  
    MOV R0, #2      ; argument 0 = 2  
    MOV R1, #3      ; argument 1 = 3  
    MOV R2, #4      ; argument 2 = 4  
    MOV R3, #5      ; argument 3 = 5  
    BL  DIFFOFSUMS ; call function  
    MOV R4, R0      ; y = returned value
```

Quando eseguo BL DIFFOFSUMS in LR sarà messo l'indirizzo della prossima istruzione, in questo caso l'indirizzo di MOV R4,R0

Funzioni: STACK

Esempio

LR è l'indirizzo della prossima istruzione,
in questo caso l'indirizzo di MOV R4,R0

ARM Assembly Code

```
; R4 = y
MAIN
    .
    .
    .
    MOV R0, #2      ; argument 0 = 2
    MOV R1, #3      ; argument 1 = 3
    MOV R2, #4      ; argument 2 = 4
    MOV R3, #5      ; argument 3 = 5
    BL  DIFFOFSUMS ; call function
    MOV R4, R0      ; y = returned value
```

| Address | Data |
|----------|------|
| BEF0F0FC | ? |
| BEF0F0F8 | |
| BEF0F0F4 | : |
| BEF0F0F0 | : |
| . | . |
| . | . |

SP ←

```
;R4 = result
DIFFOFSUMS
    SUB  SP, SP, #12      ; make space on stack for 3 registers
    STR  R9, [SP, #8]      ; save R9 on stack
    STR  R8, [SP, #4]      ; save R8 on stack
    STR  R4, [SP]          ; save R4 on stack

    ADD   R8, R0, R1       ; R8 = f + g
    ADD   R9, R2, R3       ; R9 = h + i
    SUB   R4, R8, R9       ; result = (f + g) - (h + i)
    MOV   R0, R4            ; put return value in R0

    LDR   R4, [SP]          ; restore R4 from stack
    LDR   R8, [SP, #4]      ; restore R8 from stack
    LDR   R9, [SP, #8]      ; restore R9 from stack
    ADD   SP, SP, #12      ; deallocate stack space

    MOV   PC, LR            ; return to caller
```

Funzioni: STACK

Esempio

ARM Assembly Code

| Address | Data |
|----------|------|
| BEF0F0FC | ? |
| BEF0F0F8 | R9 |
| BEF0F0F4 | R8 |
| BEF0F0F0 | R4 |
| • | • |
| • | • |

| Address | Data |
|----------|------|
| BEF0F0FC | ? |
| BEF0F0F8 | |
| BEF0F0F4 | |
| BEF0F0F0 | |
| • | • |
| • | • |

```
;R4 = result
DIFFOFSUMS
    SUB SP, SP, #12          ; make space on stack for 3 registers
    STR R9, [SP, #8]          ; save R9 on stack
    STR R8, [SP, #4]          ; save R8 on stack
    STR R4, [SP]              ; save R4 on stack

    ADD R8, R0, R1            ; R8 = f + g
    ADD R9, R2, R3            ; R9 = h + i
    SUB R4, R8, R9            ; result = (f + g) - (h + i)
    MOV R0, R4                ; put return value in R0

    LDR R4, [SP]              ; restore R4 from stack
    LDR R8, [SP, #4]           ; restore R8 from stack
    LDR R9, [SP, #8]           ; restore R9 from stack
    ADD SP, SP, #12            ; deallocate stack space

    MOV PC, LR                ; return to caller
```

Funzioni

- Ci sono funzioni in ARM che possono salvare e recuperare automaticamente più registri
- Per tale motivo ARM divide i registri in: **preservati** e **non preservati**
- I **registri preservati** sono R4–R11 e SP and LR (R13 e R14)
- I **non preservati**: R0-R3 e R12
- In questo modo tali operazioni automaticamente salvano e recuperano solo i preservati
- **Inoltre quando scriviamo il codice possiamo mantenere tale convenzione**

Funzioni

- Riassumendo

Table 6.6 Preserved and nonpreserved registers

| Preserved | Nonpreserved |
|-------------------------------|---------------------------------|
| Saved registers: R4–R11 | Temporary register: R12 |
| Stack pointer: SP (R13) | Argument registers: R0–R3 |
| Return address: LR (R14) | Current Program Status Register |
| Stack above the stack pointer | Stack below the stack pointer |

Funzioni: esempio

ARM Assembly Code

```
; R4 = result  
DIFFOFSUMS  
    PUSH  (R4)                      ; save R4 on stack  
    ADD   R1, R0, R1                  ; R1 = f + g  
    ADD   R3, R2, R3                  ; R3 = h + i  
    SUB   R4, R1, R3                  ; result = (f + g) - (h + i)  
    MOV   R0, R4                      ; put return value in R0  
    POP   (R4)                      ; pop R4 off stack  
    MOV   PC, LR                      ; return to caller
```

Preservati: R4–R11,
SP e LR (R13 e R14)

Non preservati: R0-
R3 e R12

PUSH (and **POP**) save (and restore) registers on the stack in order of register number from low to high, with the lowest numbered register placed at the lowest memory address, regardless of the order listed in the assembly instruction. For example, **PUSH {R8, R1, R3}** will store R1 at the lowest memory address, then R3 and finally R8 at the next higher.

Funzioni

- **E' importante notare che:**
 - il link Register (LR o R14) può non essere preservato se ho una singola chiamata a funzione senza che la stessi chiami un'altra funzione (funzione LEAF)
 - Invece deve essere preservato se una funzione chiama un'altra funzione (FUNZIONE NON LEAF)

Funzioni: esempio non LEAF

High-Level Code

```
int f1(int a, int b) {  
    int i, x;  
  
    x = (a + b)*(a - b);  
    for (i=0; i<a; i++)  
        x = x + f2(b+i);  
    return x;  
}
```

High-Level Code

```
int f2(int p) {  
    int r;  
    r = p + 5;  
    return r + p;  
}
```

Funzioni: esempio non LEAF

High-Level Code

```
int f2(int p) {  
    int r;  
    r = p + 5;  
    return r + p;  
}
```

Questa è una funzione leaf, non
salvo il LR

; R0 = p, R4 = r

F2

```
PUSH {R4}          ; save preserved registers used by f2  
ADD R4, R0, 5      ; r = p + 5  
ADD R0, R4, R0      ; return value is r + p  
POP {R4}          ; restore preserved registers  
MOV PC, LR         ; return from f2
```

Funzioni: esempio non LEAF

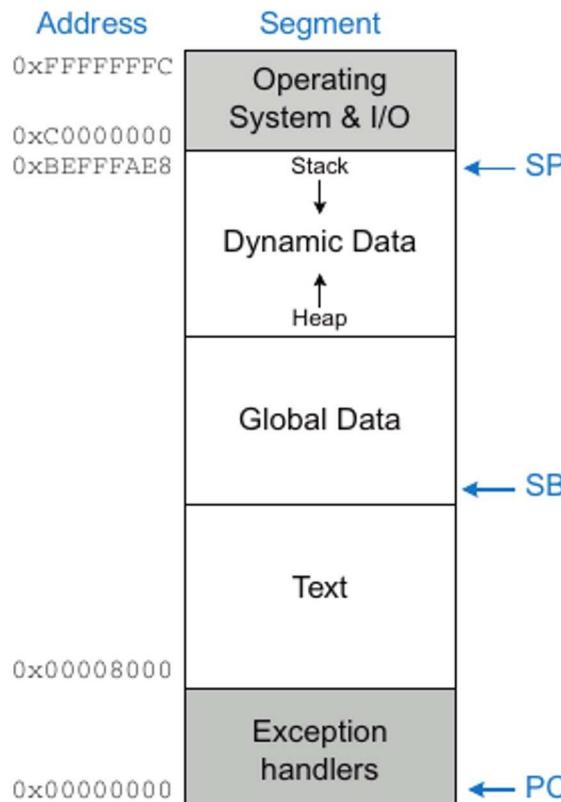
Questa è una funzione non leaf, salvo il LR

High-Level Code

```
int f1(int a, int b) {  
    int i, x;  
  
    x = (a + b)*(a - b);  
    for (i=0; i<a; i++)  
        x = x + f2(b+i);  
    return x;  
}
```

```
; R0 = a, R1 = b, R4 = i, R5 = x  
F1  
    PUSH {R4, R5, LR} ; save preserved registers used by f1  
    ADD R5, R0, R1 ; x = (a + b)  
    SUB R12, R0, R1 ; temp = (a - b)  
    MUL R5, R5, R12 ; x = x * temp = (a + b) * (a - b)  
    MOV R4, #0 ; i = 0  
FOR  
    CMP R4, R0 ; i < a?  
    BGE RETURN ; no: exit loop  
    PUSH {R0, R1} ; save nonpreserved registers  
    ADD R0, R1, R4 ; argument is b + i  
    BL F2 ; call f2(b+i)  
    ADD R5, R5, R0 ; x = x + f2(b+i)  
    POP {R0, R1} ; restore nonpreserved registers  
    ADD R4, R4, #1 ; i++  
    B FOR ; continue for loop  
RETURN  
    MOV R0, R5 ; return value is x  
    POP {R4, R5, LR} ; restore preserved registers  
    MOV PC, LR ; return from f1
```

Esempio suddivisione memoria



The Text Segment

The text segment stores the machine language program. ARM also calls this the read-only (RO) segment. In addition to code, it may include literals (constants) and read-only data.

The Global Data Segment

The global data segment stores global variables that, in contrast to local variables, can be accessed by all functions in a program. Global variables are allocated in memory before the program begins executing. Global variables are typically accessed using a static base register that points to the start of the global segment. ARM conventionally uses R9 as the static base pointer (SB).

The Dynamic Data Segment

The dynamic data segment holds the stack and the heap

The Exception Handler, OS, and I/O Segments

The lowest part of the ARM memory map is reserved for the exception vector table and exception handlers, starting at address 0x0 (see Section 6.6.3). The highest part of the memory map is reserved for the operating system and memory-mapped I/O (see Section 9.2).