

# Architettura degli Elaboratori

## Lezione 1: Introduzione

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

# Organizzazione del corso

Sito del corso: [www.dsi.unive.it/~architet](http://www.dsi.unive.it/~architet)

Architettura degli elaboratori – primo modulo (primo semestre)

**Marta Simeoni** (orario: martedì 8.45 – 10.15 e  
giovedì 8.45 – 10.15)

orario di ricevimento: mercoledì 9.00 – 11.00

Architettura degli elaboratori – secondo modulo (secondo semestre)

**Salvatore Orlando** orario da definire

Ciascun modulo organizzato su 12 settimane  
(4 ore di lezione per settimana per un totale di 48 ore)

*L'esame è da 12 crediti: l'intero programma si svolge sui due semestri e viene registrato un unico voto.*

# Comunicazioni

- Comunicazione Studenti -> Docenti

[architettura@dsi.unive.it](mailto:architettura@dsi.unive.it)

- Comunicazione Docente -> Studenti

- Il mezzo privilegiato di comunicazione è il sito del corso

<http://www.dsi.unive.it/~architet>

# Libro di testo

David A. Patterson, John L. Hennessy

Struttura e Progetto dei Calcolatori

Terza Edizione

Zanichelli Editore 2010

# Modalità d'esame

- Compito finale scritto (maggio, giugno, settembre 2014 e gennaio 2015) sul programma del primo e secondo modulo
  - serie di **domande** teoriche ed **esercizi**
- Materiale didattico disponibile sul sito del corso
  - slide proiettate durante il corso
    - lezioni e esercizi
  - materiale di studio aggiuntivo
    - link a pagine utili
    - compiti svolti anni precedenti (con soluzioni)

# Modalità d'esame: compitini

Il corso prevede lo svolgimento di due prove intermedie (“compitini”) per facilitare gli studenti che seguono regolarmente il corso.

- Il primo compitino è relativo al programma del primo modulo e si svolge nella sessione invernale (appello di Gennaio 2014).
- il secondo compitino verte sul programma del secondo modulo e si svolge nella sessione estiva (Maggio-Giugno 2014), autunnale (Settembre 2014) e invernale (Gennaio 2015).

## IMPORTANTE:

- Il secondo compitino può essere svolto *solo dagli studenti che hanno superato positivamente il primo.*
- Chi non riesce a superare i due compitini con voto complessivo (media dei voti dei due compitini) sufficiente *deve rifare l'esame completo.*

# Primo modulo: ulteriore possibilità

Inoltre, per agevolare gli studenti che studiano man mano che il corso procede:

- Durante lo svolgimento del primo modulo verrà predisposta una **prima prova intermedia**, con domande ed esercizi sulla parte di programma già esposta in classe.
- A gennaio, in corrispondenza con l'appello d'esame, *gli studenti che hanno svolto la prima prova* potranno svolgere la **seconda prova intermedia** al posto del compitino. La seconda prova intermedia consisterà di domande ed esercizi relativi alla parte di programma svolta dopo la prima prova.

# Obiettivi

- I due moduli del corso si prefiggono di svelare i seguenti aspetti di un moderno computer:
  - organizzazione interna e il funzionamento
    - CPU
    - Memoria
    - I/O
  - concause HW/SW delle prestazioni
    - come si misurano le prestazioni
  - segreti della programmazione
    - scendendo rispetto al livello di astrazione *linguaggio ad alto livello*



# Obiettivi

- Perché è importante studiare questa materia?
  - per conoscere il calcolatore in tutte le sue componenti
  - per riuscire a costruire del software usabile e veloce
  - per essere in grado di prendere decisioni di acquisto HW

# Programma del primo modulo

- Organizzazione di base di un calcolatore (CPU, memoria, I/O) e livelli di astrazione.
- Rappresentazione dell'informazione e aritmetica dei calcolatori.
- Algebra booleana (Tabelle di verità, Forme canoniche di espr. booleane), Circuiti combinatori (multiplexer, decoder, PLA).
- Memoria: Latch, Clock, Flip-flop, Registri, RAM.
- Circuiti sequenziali sincroni.
- Principali istruzioni MIPS (aritmetico-logico, di controllo)
- Progetto della CPU:
  - Progettazione ALU e Register File.
  - Parte controllo e parte operativa.
  - Organizzazione a singolo e multiplo ciclo.
  - Progetto del controllo.
- Valutazione delle prestazioni: Tempo di CPU. Throughput. CPI. Misure di prestazioni e benchmarks.

# Introduzione

- Calcolatori elettronici
  - basati su tecnologie in rapidissima evoluzione
  - valvole -> transistor -> IC (VLSI)<sup>1</sup>
    - legge di Moore
      - ogni 1,5 anni osserviamo il raddoppio di:  
*capacità di memoria*  
*velocità del processore*  
*(dovuto a miglioramenti nella tecnologia e nell'organizzazione)*
      - abbattimento dei costi contemporaneo all'incremento di velocità
      - Considerando come base gli anni '40, se l'industria dei trasporti avesse seguito la stessa evoluzione dei computer:  
US coast-to-coast in 5 sec. per solo mezzo dollaro

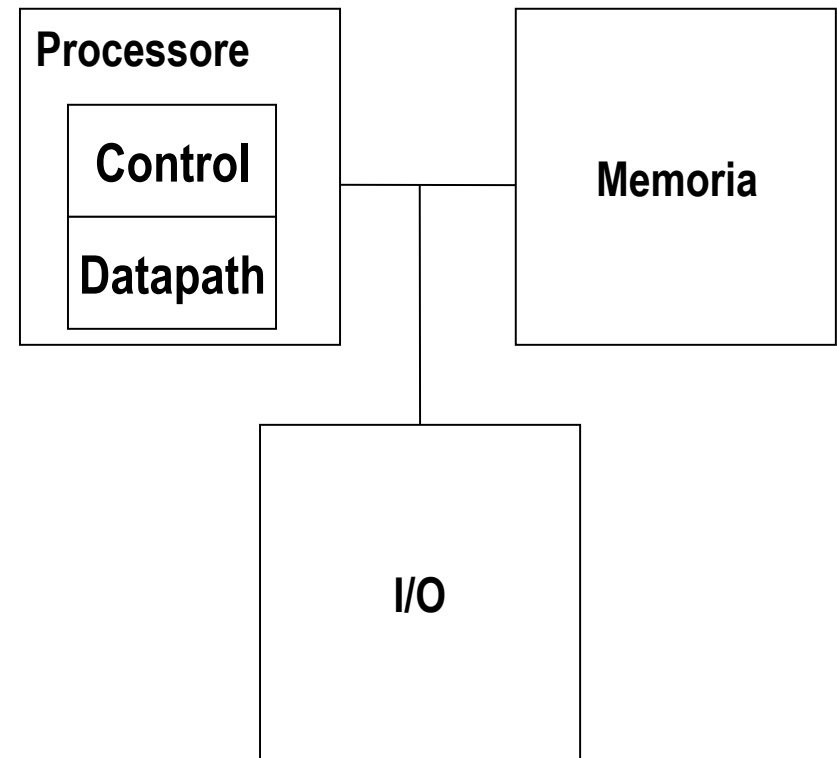
<sup>1</sup>(IC=Integrated Circuit    VLSI=Very Large Scale Integration)

# Introduzione

- Trend
  - incremento prestazioni e abbattimento costi
    - permette di affrontare e risolvere applicazioni sempre più complesse
  - integrazione con la rete
    - ancora nuove applicazioni informatiche (es. WEB)
  - integrazione con la rete telefonica e cellulare
    - nuovi hw e applicazioni (smartphone, ecc)

# Componenti principali di un computer

- *Input/Output (I/O)*
  - mouse, tastiera (I)
  - video, stampante (O)
  - dischi (I/O)
  - CD e DVD (I/O o I)
  - rete (I/O)
- *memoria principale*
  - DRAM, SRAM
- *processore (CPU)*
  - parte operativa (datapath)
  - parte controllo (control)
  - *bus*



# Com'è fatto un computer?

- I/O
  - serve per comunicare con l'esterno
  - include dispositivi di *memoria secondaria* (memoria non volatile), acceduti come dispositivi di I/O
- Memoria principale
  - usata per memorizzare programmi e dati durante l'esecuzione (concetto di *stored-program* introdotto da Von Neumann)
- Processore
  - è l'esecutore delle *istruzioni* appartenenti ad un'ISA
  - **ISA (Instruction Set Architecture)** definisce quindi il linguaggio (*povero*) comprensibile dal processore
  - le istruzioni sono lette dalla memoria, modificano dati in memoria o agiscono sull'I/O
  - decomponibile in
    - Parte Controllo* → mente
    - Parte Operativa* → braccio

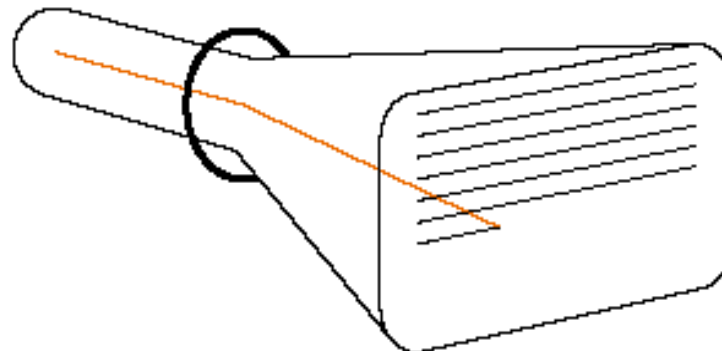
# Com'è fatto un Personal Computer ?

- Video: unità di uscita (O)
- Tastiera/mouse: unità di ingresso (I)
- Scatola: contiene
  - alimentatore
  - scheda madre (o motherboard o mainboard) e bus
  - processore
  - memoria volatile (RAM)
  - dischi (memoria stabile, I/O)
  - lettori CD/DVD
  - dispositivi di I/O per rete (LAN / MODEM)
  - dispositivi di I/O USB (Universal Serial Bus)

# Video

## CRT (Cathode Ray Tube)

- fascio di elettroni “spennellato” su una matrice di fosfori
  - necessario il refresh continuo dello schermo
  - pennello passa sullo schermo per righe, una riga alla volta
  - frequenza (di refresh) espressa in Hz
- Dati caratteristici
  - frequenza di refresh ( $> 70$  Hz per evitare sfarfallii)
  - numero di pixel (punti) dello schermo (es.: 1024x768)
  - numero di colori contemporaneamente visualizzati (RGB)

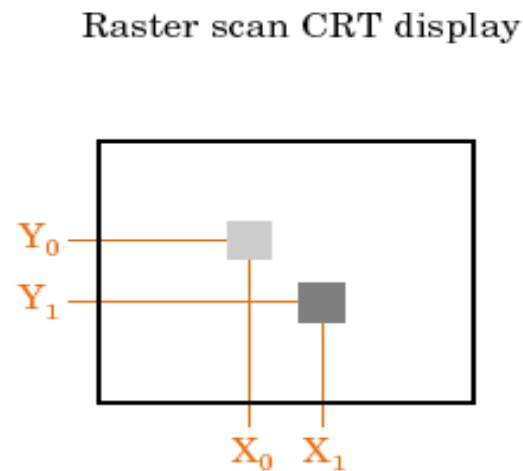
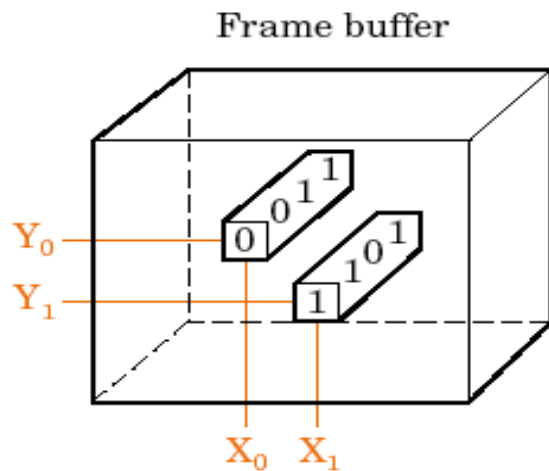




# Video

## ■ Frame buffer

- memoria RAM veloce che contiene la rappresentazione binaria dei vari pixel (ovvero dei colori corrispondenti)
- per visualizzare sullo schermo bisogna scrivere nel frame buffer
- un convertitore digitale/analogico, con la frequenza opportuna, rinfresca in continuazione i vari pixel dello schermo sulla base dei dati del buffer



# Video e Mouse

## LCD (Liquid Crystal Display)

- molecole organiche con struttura cristallina immersi in un liquido
- proprietà ottiche dipendono dall'allineamento delle molecole
- retro illuminati



## Mouse

- Il mouse è un dispositivo in grado di inviare un input ad un computer in modo tale che ad un suo movimento ne corrisponda uno analogo di un indicatore sullo schermo detto puntatore. È inoltre dotato di uno o più tasti ai quali possono essere assegnate varie funzioni.

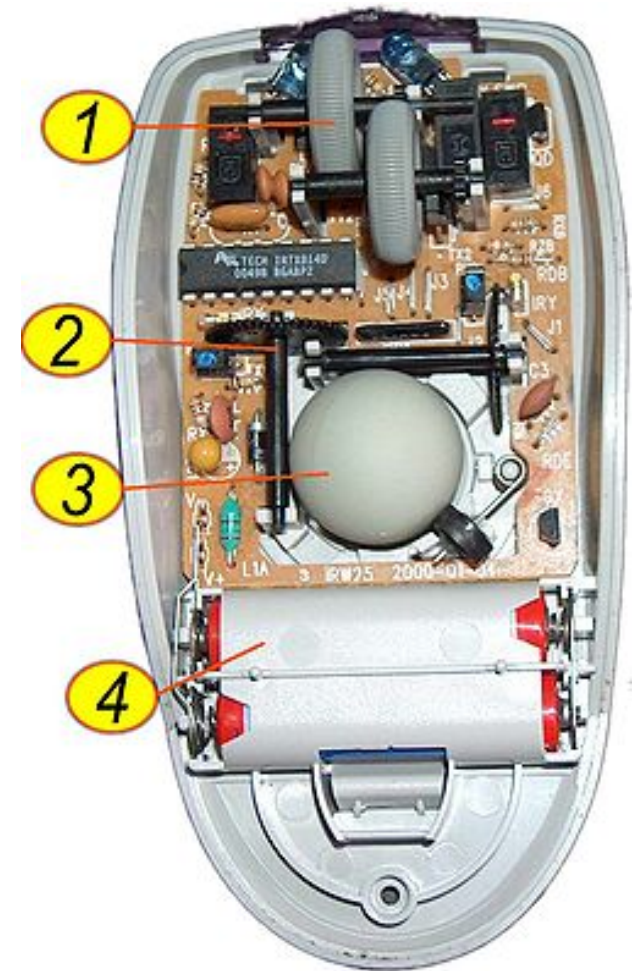
# Mouse

- **Mouse meccanici**

- Sfera (3) a contatto con due rotelle (2), una per l'asse delle X, l'altra per l'asse Y, che a loro volta sono connesse a contatori
- La rotazione della sfera muove le rotelle

- **Mouse ottici**

- composti da un led, un sensore ottico e un chip per l'acquisizione delle immagini
- processore più complesso di quello presente in un mouse tradizionale
- mouse laser: usano un laser al posto del led per l'illuminazione del piano d'appoggio



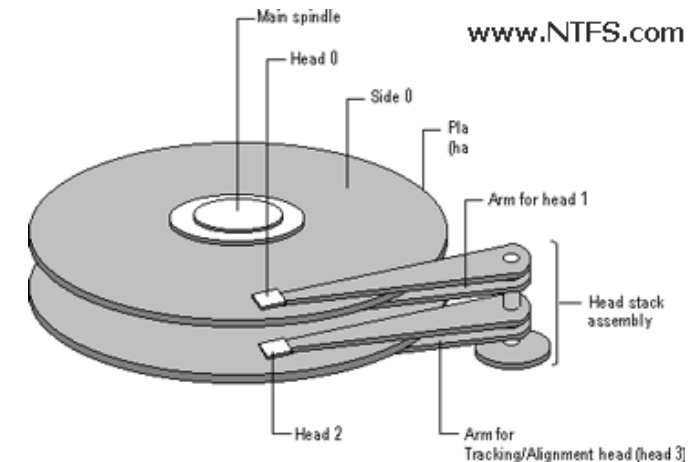
# Scatola

- Processore
  - chip che contiene parte controllo+parte operativa con registri
  - cache (buffer di memoria veloce)
- Memoria principale
  - DRAM, volatile, composta di vari chip
- Scheda madre: contiene diversi chip e bus, con alloggiamenti per
  - Processore
  - memoria
  - schede per gestire video, audio, rete, dischi, ecc. (I/O)

# Scatola

## Dischi (Memoria stabile secondaria)

- piatti girevoli ricoperti di materiale magnetico
- controller che ordina i movimenti della testina
- testina dotata di bobina elettromagnetica, che legge/scrive informazioni digitali (0/1)
- Floppy: lenti, 1.44 MB - 200 MB (Zip)
- Hard: piatti metallici, più veloci, con velocità di rotazione alta, diversi GB
- Differenza di tempo di accesso ai dispositivi di memoria
  - RAM (accesso da 5 a 100 nsec)
  - DISCHI (5-20 msec): 5-6 ordini di grandezza di differenza



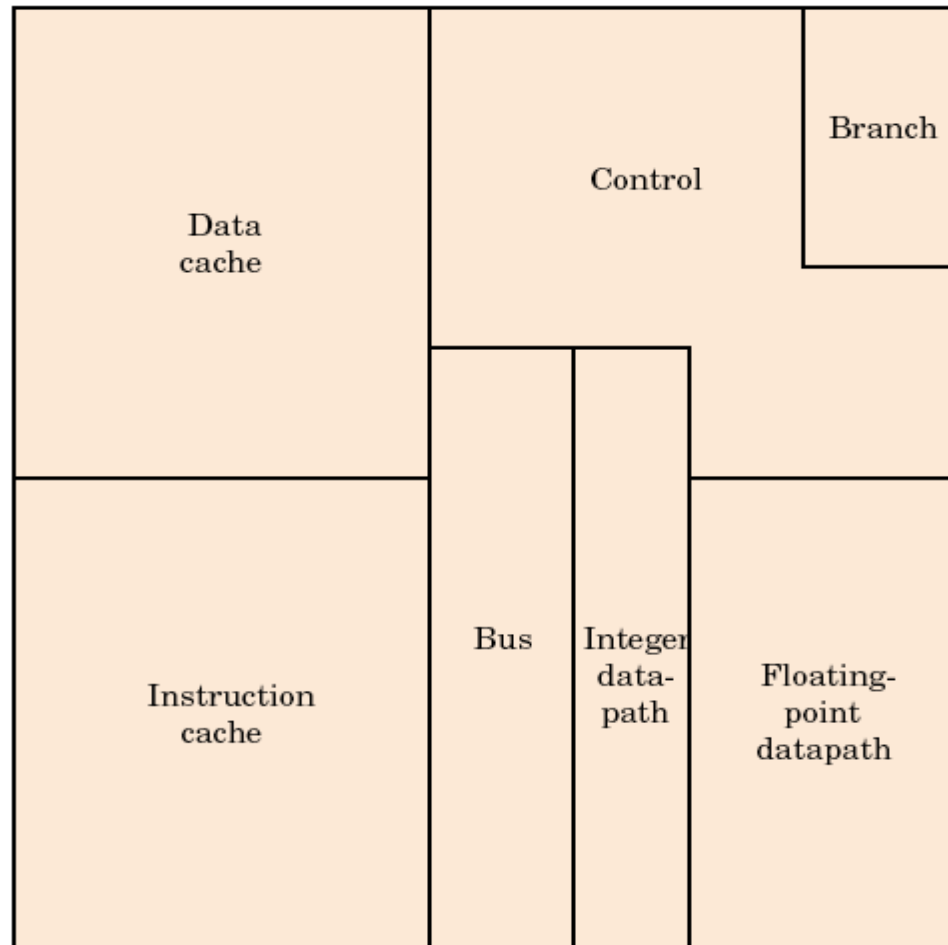
# Realizzazione

I componenti elettronici del computer sono realizzati con circuiti integrati (IC):

- fili + transistor realizzati con processo di integrazione larghissimo su frammenti di silicio (VLSI - Milioni di transistor su un singolo frammento)
- Silicio
  - presente nella sabbia
  - è un semiconduttore
  - aggiungendo materiali al silicio attraverso processo chimico, il silicio diventa
    - transistor, conduttore, o isolante

# Central Processing Unit

Pentium della Intel. Chip 91 mm<sup>2</sup>. 3,3 milioni di transistor

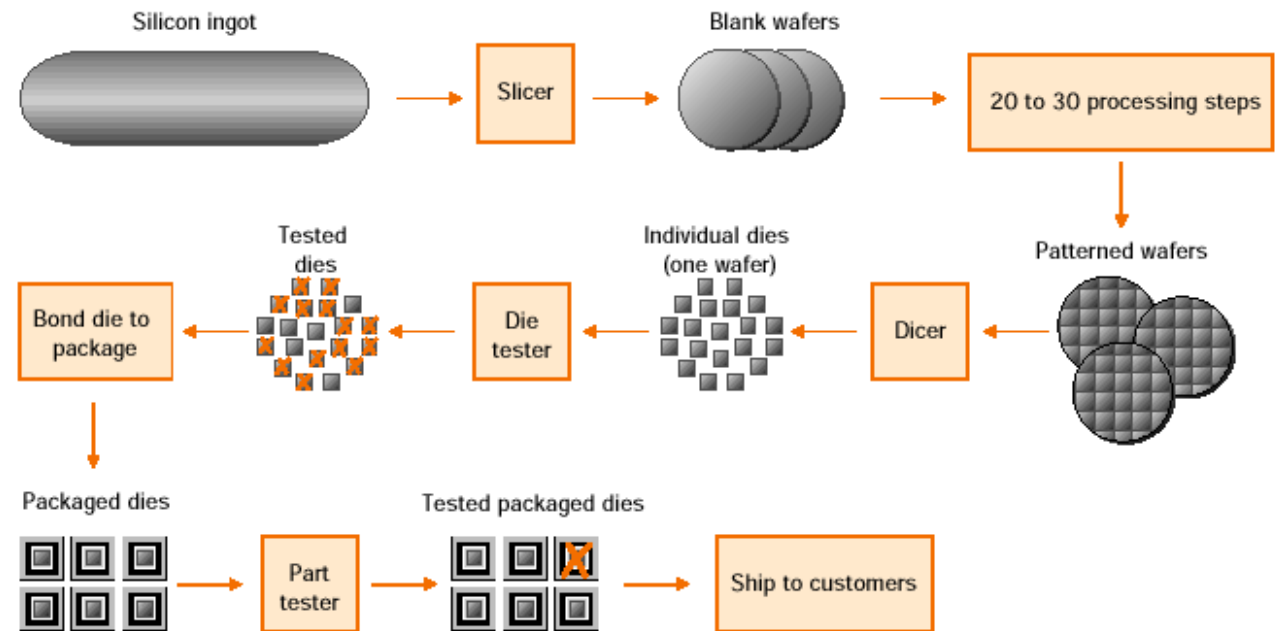


# Processo produttivo dei chip

Lingotto di silicio  
(15/30 cm diam.)

Wafer sottili ottenuti  
tagliando il lingotto

10/20 processi produttivi  
tramite pattern e processi  
chimici ⇒  
otteniamo diverse  
repliche dello stesso  
circuitto rettangolare



Separiamo i vari circuiti e otteniamo i *die*

Collaudiamo i *die*

Inseriamo nei package i *die* funzionanti, collegandoli ai piedini (*pin*) del package ⇒  
otteniamo i *chip* pronti per la consegna



# Livelli di astrazione

Per progettare o capire l'architettura di un sistema, oppure per programmare semplicemente un sistema, abbiamo bisogno di astrarre.

Es.: se cerchiamo di capire come funziona una CPU ... ci accorgiamo che la funzionalità della CPU è comprensibile se astraiano e guardiamo solo alla sua interfaccia di programmazione: ISA (Instruction Set Architecture)

Se scendiamo di livello, troviamo fili e milioni di transistor!! E diventa così impossibile comprenderne il funzionamento!!! In particolare, non si riesce a:

- capire come questo livello interpreta le istruzioni dell'ISA
- individuare i blocchi funzionali
- capire a cosa servono i blocchi funzionali stessi

# Livelli di astrazione

- In tutti i progetti di sistemi hw/sw
  - ritroviamo il concetto della strutturazione in livelli
- Un programmatore è solitamente interessato
  - al livello **n-esimo** del sistema e al relativo linguaggio (vista più astratta che guarda al livello più alto)
  - ai traduttori (compilatori o interpreti) che gli permettono di eseguire i programmi sui livelli sottostanti
- Un architetto deve invece conoscere i vari livelli e le relazioni tra di essi

# Livelli di astrazione

- I livelli più bassi rivelano più informazioni
- I livelli più alti astraggono omettendo dettagli
  - l'astrazione ci permette di affrontare la complessità
- I livelli più alti *virtualizzano*, ovvero offrono una vista virtuale dei livelli inferiori
  - *Macchina virtuale o astratta*: appare più potente e semplice da programmare della macchina sottostante

# Strutturazione in livelli

- Tradizionale vista a livelli dell'architettura hw/sw di un computer. Ogni livello mette a disposizione:
  - uno o più linguaggi riconosciuti
  - uno o più interpreti o compilatori per tradurre tra linguaggi

Livello 4: Linguaggio ad alto livello

Livello 3: Assembler

Livello 2: Sistema Operativo

Livello 1: Linguaggio Macchina (ISA)

Livello 0: Hardware/firmware
- Sistema Operativo (SO)
  - per ora pensiamo al S.O. come un livello il cui linguaggio è l'ISA estesa con nuove istruzioni ad alto livello per gestire risorse critiche (es. I/O)
  - offre nuove istruzioni oltre a quelle base dell'ISA

# Architettura degli Elaboratori

## Rappresentazione dell'informazione

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

# Rappresentazione dell'informazione

Differenza tra simbolo e significato

- la cifra (lettera) usata per scrivere è un simbolo che rappresenta l'informazione
- il concetto di numero (suono) corrisponde al significato dell'informazione

Per comunicare/rappresentare informazioni è quindi necessario usare dei simboli

- è necessaria una convenzione (rappresentazione, codifica o codice) per associare i simboli con il loro significato

# Codifica o codice

Per *codificare* l'informazione solitamente si usa un *alfabeto di simboli*

- **Alfabeto** = insieme finito di *simboli* adottati per rappresentare informazione
- Es: per rappresentare numeri nei calcolatori elettronici
  - Alfabeto binario: {0, 1}
  - Simboli associati con stati elettrici facilmente distinguibili

Una **codifica** (codice) fornisce una corrispondenza tra

- **sequenze** (stringhe, configurazioni) di simboli dell'alfabeto
- i **dati**

# Codifica o codice

Solitamente, i codici fanno riferimento a sequenze di simboli di lunghezza finita

- Alfabeto di  $N$  simboli e Sequenze di lunghezza  $K$ 
  - $N^K$  configurazioni possibili
- Rispetto ad un alfabeto binario ( $N=2$ )
  - numero totale di configurazioni:  $2^K$
  - $2^K \geq s$  (dove  $s$  è la cardinalità dell'insieme  $D$  dei dati)
  - Es.: se  $D$  comprende le 26 lettere dell'alfabeto inglese ( $s=26$ )
  - sono necessarie almeno sequenze di  $K$  simboli binari, con  $K \geq 5$



# Codifica e tipo di informazioni

La codifica delle informazioni non numeriche può essere effettuata in maniera semi arbitraria.

- Basta fissare una convenzione per permettere di riconoscere i dati
- Es. Codice ASCII - American Standard Code for Information Exchange - è una codifica di caratteri alfanumerici su sequenze di simboli binari di lunghezza  $k=8$

La codifica dei numeri

- deve essere accurata, perché è necessario effettuare operazioni (sommare, moltiplicare ecc.) usando la rappresentazione adottata
- di solito si adotta il sistema di numerazione arabica, o posizionale

Concentriamoci ora sulla rappresentazione dei numeri...

# Codifica dei numeri

Sistema di numerazione arabica in base 10 ( $B=10$ )

- cifre (simboli) appartenenti all'alfabeto di 10 simboli  $A=\{0,1,\dots,9\}$
- Il simbolo 0 corrisponde al numero zero, ..., il simbolo 9 corrisponde al numero nove
- simboli con valore diverso in base alla posizione nella stringa di simboli in A (unità, decine, centinaia, migliaia, ecc.)

Es:  $2543 = 2 \cdot 10^3 + 5 \cdot 10^2 + 4 \cdot 10^1 + 3 \cdot 10^0$

# Sistema di codifica posizionale

*Codificare i numeri naturali in una generica base B*

- fissare un alfabeto  $A$  di  $B$  simboli
- fissare una corrispondenza tra
  - i  $B$  simboli di  $A \Leftrightarrow$  i primi  $B$  numeri naturali  $\{0, 1, 2, \dots, B-1\}$
- numeri maggiori di  $B$  rappresentabili come stringhe di simboli  $d_i \in A$  :
  - $d_{n-1} \dots d_1 d_0$
- valore numerico della stringa, dove la *significatività* delle cifre è espressa in base alle varie potenze di  $B$ :
  - $B^{n-1} * d_{n-1} + \dots + B^1 * d_1 + B^0 * d_0$

Es:  $B = 8$   $A = \{0, 1, 2, 3, 4, 5, 6, 7\}$       $1056_8 = 1 * 8^3 + 0 * 8^2 + 5 * 8^1 + 6 * 8^0$   
 $= 512 + 40 + 6 = 558_{10}$

# Numeri naturali in base 2

Alfabeto binario  $A=\{0,1\}$ , dove i simboli sono detti bit, con 0 corrispondente al numero zero ed 1 al numero uno

- Nei calcolatori i numeri sono rappresentati come sequenze di bit di lunghezza finita
- numeri rappresentati in notazione arabica, con base  $B=2$  (numeri binari)
- $d_{n-1} \dots d_1 d_0$  dove  $d_i \in \{0,1\}$

$$\text{Es: } 10011 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 19$$

# Numeri naturali in base 2

Con stringhe di  $n$  bit, sono rappresentabili  $2^n$  dati (numeri diversi)

- dal numero 0 al numero  $2^n - 1$

Il valore numerico corrispondente, dove la significatività delle cifre è espressa sulla base di una potenza di  $B=2$ , è:

- $2^{n-1} * d_{n-1} + \dots + 2^1 * d_1 + 2^0 * d_0$

Es: il valore della stringa di simboli 1010 in base 2 è

- $1010_2 = 1*8 + 0*4 + 1*2 + 0*1 = 10_{10}$

# Esercizi

Determinare il valore decimale dei seguenti numeri:

- $140_5$
- $7436_8$
- $122_3$
- $122_7$
- $1011101_2$
- $1101_3$
- $328_9$

# Conversione inversa

Da base 10 a base B

Procedimento per *divisioni intere successive*

Sia dato un certo numero naturale  $N_{10}$  rappresentabile in base B come stringa di n simboli  $d_{n-1} \dots d_1 d_0$  il cui valore è:

$$N_{10} = B^{n-1} * d_{n-1} + \dots + B^1 * d_1 + B^0 * d_0$$

Se dividiamo per B

- otteniamo  $d_0$  come resto
  - *Quoziente*:  $B^{n-2} * d_{n-1} + \dots + B^0 * d_1$
  - *Resto*:  $d_0, \quad 0 \leq d_0 < B$
- possiamo iterare il procedimento, ottenendo  $d_1, d_2, d_3$  ecc. fino ad ottenere un Quoziente = 0

NOTA: il procedimento vale per conversioni da base B a base B' qualsiasi.

# Conversione inversa: algoritmo

```
Q=N; i=0;  
fintantoché è vero che (Q>0), ripeti:  
{  
  di = Q % B;    // Q mod B  
  stampa di;  
  Q = Q / B;      // Q div B  
  i = i+1;  
}
```

Q	R
4	0 d <sub>0</sub>
2	0 d <sub>1</sub>
1	1 d <sub>2</sub>
0	

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



# Esercizi

Qual è il valore decimale dei seguenti numeri:

1)  $1011101_2$

2)  $1111000111_2$

3)  $1201_3$

4)  $2301_4$

5)  $4230_4$

6)  $403421_6$

7)  $625_7$

8)  $723_8$

Tradurre i seguenti numeri:

$127_{10}$  in base 2, 3, 4, 5, 6, 7, 8

$127_8$  in base 2, 3, 4, 5, 6, 7, 10, 11

# Rappresentazioni ottale ed esadecimale

Ottale:  $B = 8$

Esadecimale:  $B = 16$

Usate per facilitare la comunicazione di numeri binari tra umani, o tra il computer e il programmatore

La codifica è più corta rispetto alla base 2

Esiste inoltre un metodo veloce per convertire tra base 8 (o base 16) e base 2, e viceversa

# Rappresentazione ottale

$B = 8, A = \{0,1,2,3,4,5,6,7\}$

Come convertire:

- Sia dato un numero binario di 10 cifre:  $d_9 \dots d_1 d_0$ , il cui valore è:

$$\sum_{i=0}^9 2^i \cdot d_i$$

Raggruppiamo le cifre: da destra, a 3 a 3

Poniamo in evidenza la più grande potenza di 2 comune possibile:

- $(2^0 d_9) 2^9 + (2^2 d_8 + 2^1 d_7 + 2^0 d_6) 2^6 +$   
 $(2^2 d_5 + 2^1 d_4 + 2^0 d_3) 2^3 + (2^2 d_2 + 2^1 d_1 + 2^0 d_0) 2^0$

# Rappresentazione ottale

$$(2^0 d_9) 2^9 + (2^2 d_8 + 2^1 d_7 + 2^0 d_6) 2^6 + \\ (2^2 d_5 + 2^1 d_4 + 2^0 d_3) 2^3 + (2^2 d_2 + 2^1 d_1 + 2^0 d_0) 2^0$$

I termini tra parentesi sono numeri compresi tra 0 e 7

- si possono far corrispondere ai simboli dell'alfabeto ottale
- I fattori messi in evidenza corrispondono alle potenze di  $B=8$ :

$$2^0=8^0 \quad 2^3=8^1 \quad 2^6=8^2 \quad 2^9=8^3$$

Da binario ad ottale:  $1001010111_2 = 1 \ 001 \ 010 \ 111 = 1127_8$

Da ottale a binario:  $267_8 = 010 \ 110 \ 111 = 10110111_2$

# Rappresentazione esadecimale

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

Come convertire:

- Sia dato un numero binario di 10 cifre:  $d_9 \dots d_1 d_0$ , il cui valore è:

$$\sum_{i=0}^9 2^i \cdot d_i$$

- *Raggruppiamo* le cifre: da destra, e a 4 a 4
- Poniamo in evidenza la più grande potenza di 2 possibile:  
 $(2^1 d_9 + 2^0 d_8) 2^8 + (2^3 d_7 + 2^2 d_6 + 2^1 d_5 + 2^0 d_4) 2^4 +$   
 $(2^3 d_3 + 2^2 d_2 + 2^1 d_1 + 2^0 d_0) 2^0$

# Rappresentazione esadecimale

$$(2^1 d_9 + 2^0 d_8) 2^8 + (2^3 d_7 + 2^2 d_6 + 2^1 d_5 + 2^0 d_4) 2^4 + (2^3 d_3 + 2^2 d_2 + 2^1 d_1 + 2^0 d_0) 2^0$$

I termini tra parentesi sono numeri compresi tra 0 e 15

- si possono far corrispondere ai simboli dell'alfabeto esadecimale

I fattori messi in evidenza corrispondono alle potenze di  $B=16$ :

- $2^0=16^0$     $2^4=16^1$     $2^8=16^2$

Da binario ad esadecimale:

$$1001011111_2 = 10 \ 0101 \ 1111 = 25F_{16}$$

Da esadecimale a binario:

$$A67_{16} = 1010 \ 0110 \ 0111 = 101001100111_2$$

# Numeri naturali binari

Il processore che studieremo (MIPS) rappresenta i numeri interi su 32 bit (32 bit = 1 word)

I numeri interi senza segno (unsigned) rappresentabili su 32 bit sono allora:

0000	0000	0000	0000	0000	0000	0000	0000	$_{\text{two}}$	=	$0_{\text{ten}}$
0000	0000	0000	0000	0000	0000	0000	0001	$_{\text{two}}$	=	$1_{\text{ten}}$
0000	0000	0000	0000	0000	0000	0000	0010	$_{\text{two}}$	=	$2_{\text{ten}}$
...										
1111	1111	1111	1111	1111	1111	1111	1101	$_{\text{two}}$	=	$4,294,967,293_{\text{ten}}$
1111	1111	1111	1111	1111	1111	1111	1110	$_{\text{two}}$	=	$4,294,967,294_{\text{ten}}$
1111	1111	1111	1111	1111	1111	1111	1111	$_{\text{two}}$	=	$4,294,967,295_{\text{ten}}$

Ora che abbiamo scelto una rappresentazione per i numeri binari vediamo come usarla per eseguire operazioni (tra numeri binari) in modo consistente

# Algoritmo di somma di numeri (naturali) binari

Per la somma di numeri rappresentati in binario possiamo adottare la stessa procedura usata per sommare numeri decimali

- sommare via via i numeri dello stesso peso, più l'eventuale riporto:
- La tabella per sommare 3 cifre binarie è la seguente:

di	di'	rip	RIS	RIP
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



# Esempio di somma

Sia  $A = 13_{\text{dieci}} = 01101_{\text{due}}$  e  $B = 11_{\text{dieci}} = 01011_{\text{due}}$

riporti: **1111**

A: 01101 +

B: 01011 =

-----

11000  $\equiv 24_{\text{dieci}}$

L'algoritmo impiegato dal calcolatore per effettuare la somma è simile a quello "carta e penna"

- le cifre sono prodotte una dopo l'altra, da quelle meno significative a quelle più significative

# Overflow

L'**overflow** si verifica quando il risultato è troppo *grande* per essere rappresentato nel *numero finito di bit* messo a disposizione dalle rappresentazioni dei numeri

⇒ **il riporto fluisce fuori**

Es: somma di due numeri rappresentati su 4 bit che produce un numero **non** rappresentabile su 4 bit

```
1111
 1111+
1001=
11000
```

# Sottrazione e numeri relativi

L'algoritmo impiegato nei calcolatori per **sottrarre** numeri binari

- è diverso da quello “carta e penna”, che usa la ben nota nozione di “prestito” delle cifre

*Non* viene impiegata l'ovvia rappresentazione in *modulo e segno* per rappresentare i *numeri relativi*

- si usa *invece* una **particolare rappresentazione dei numeri negativi**

Questa particolare rappresentazione permette di usare lo *stesso algoritmo efficiente* già impiegato per la somma

# Possibili rappresentazioni

Modulo e Segno

$$000 = + 0$$

$$001 = + 1$$

$$010 = + 2$$

$$011 = + 3$$

$$100 = - 0$$

$$101 = - 1$$

$$110 = - 2$$

$$111 = - 3$$

Complemento a uno

$$000 = + 0$$

$$001 = + 1$$

$$010 = + 2$$

$$011 = + 3$$

$$100 = - 3$$

$$101 = - 2$$

$$110 = - 1$$

$$111 = - 0$$

Complemento a due

$$000 = + 0$$

$$001 = + 1$$

$$010 = + 2$$

$$011 = + 3$$

$$100 = - 4$$

$$101 = - 3$$

$$110 = - 2$$

$$111 = - 1$$

# Caratteristiche delle rappresentazioni

**bilanciamento:** nel Complemento a Due, nessun numero positivo corrisponde al più piccolo valore negativo

**numero di zeri:** le rappresentazioni in Modulo e Segno, e quella in Complemento a Uno, hanno 2 rappresentazioni per lo zero

**semplicità delle operazioni:** per il Modulo e Segno bisogna prima guardare i segni e confrontare i moduli, per decidere sia il segno del risultato, e sia per decidere se bisogna sommare o sottrarre.

**Il Complemento a uno non permette di sommare numeri negativi.**

# Complemento a 2

La rappresentazione in *complemento a 2* è quella adottata dai calcolatori per i numeri interi con segno (signed).

Il bit più significativo corrisponde al segno (0 *positivo*, 1 *negativo*)

MIPS: Numeri relativi (signed) su 32 bit:

0000 0000 0000 0000 0000 0000 0000 0000<sub>two</sub> = 0<sub>ten</sub>

0000 0000 0000 0000 0000 0000 0000 0001<sub>two</sub> = + 1<sub>ten</sub>

0000 0000 0000 0000 0000 0000 0000 0010<sub>two</sub> = + 2<sub>ten</sub>

...

0111 1111 1111 1111 1111 1111 1111 1110<sub>two</sub> = + 2,147,483,646<sub>ten</sub>

0111 1111 1111 1111 1111 1111 1111 1111<sub>two</sub> = + 2,147,483,647<sub>ten</sub>

1000 0000 0000 0000 0000 0000 0000 0000<sub>two</sub> = - 2,147,483,648<sub>ten</sub>

1000 0000 0000 0000 0000 0000 0000 0001<sub>two</sub> = - 2,147,483,647<sub>ten</sub>

1000 0000 0000 0000 0000 0000 0000 0010<sub>two</sub> = - 2,147,483,646<sub>ten</sub>

...

1111 1111 1111 1111 1111 1111 1111 1101<sub>two</sub> = - 3<sub>ten</sub>

1111 1111 1111 1111 1111 1111 1111 1110<sub>two</sub> = - 2<sub>ten</sub>

1111 1111 1111 1111 1111 1111 1111 1111<sub>two</sub> = - 1<sub>ten</sub>

# Complemento a 2: rappresentazione

Rappresentazione di numeri in complemento a 2 su n bit dei numeri signed:

- 0: 0.....0
- $2^{n-1}-1$  numeri positivi:
  - 1 (0 .....01)
  - $2^{n-1}-1$  (massimo) (01.....11)
- $2^{n-1}$  numeri negativi
  - $-|N|$  rappresentato dal numero **unsigned** ottenuto tramite la seguente operazione:  
 $2^n - |N|$
  - -1:  $2^{n-1}$  (1.....1)
  - $-2^{n-1}$  (minimo):  $2^n - 2^{n-1} = 2^{n-1}$  (10.....0)

# Esercizi

Tradurre i seguenti numeri:

$11101010111_2$  in ottale ed esadecimale

$AF59_{16}$ ,  $77216_8$  in binario

Eseguire le seguenti somme tra numeri naturali espressi in binario su 8 bit e dire in quali casi si verifica overflow.

$10110111 + 00111111$

$11110010 + 01111111$

$00011111 + 11100000$

$10001001 + 11110001$

Scrivere i seguenti numeri in complemento a due su 8 bit:

$0_{10}$ ,  $100_{10}$ ,  $131_{10}$ ,  $-128_{10}$ ,  $-50_{10}$ ,  $-5_{10}$

Scrivere tutti i numeri in complemento a due rappresentabili su 5 bit



# Complemento a 2: valore

- Il valore corrispondente alla rappresentazione dei numeri positivi è quello solito
- Per quanto riguarda i numeri negativi, per ottenere **direttamente** il valore di un numero negativo su n posizioni, basta considerare
  - il bit di segno (=1) in posizione n-1 con peso:  $-2^{n-1}$
  - tutti gli altri bit in posizione i con peso  $2^i$

Dimostrazione:

$-|N|$  viene rappresentato in complemento a 2 dal numero  
**unsigned**  $2^n - |N|$

supponiamo che  $2^n - |N|$  corrisponda alla n-upla

$$d_{n-1}(=1) d_{n-2} \dots d_1 d_0 \Rightarrow$$

$$\begin{aligned} 2^n - |N| &= 2^{n-1} + 2^{n-2} * d_{n-2} + \dots + 2^1 * d_1 + 2^0 * d_0 \\ -|N| &= -2^{n-1} + 2^{n-2} * d_{n-2} + \dots + 2^1 * d_1 + 2^0 * d_0 \\ -|N| &= -2^{n-1} + 2^{n-2} * d_{n-2} + \dots + 2^1 * d_1 + 2^0 * d_0 \end{aligned}$$

# Complemento a 2: cambio di segno

Dato un numero positivo  $N$ , con bit di segno uguale a 0

Per ottenere la rappresentazione in complemento a 2 di  $-N$  è possibile impiegare equivalentemente

- Alg. 1: inverti tutti i bit (ovvero Complementa a uno) e somma 1
- Alg. 2: inverti tutti i bit a sinistra della cifra "1" meno significativa

Dato un numero negativo  $-|N|$  espresso in complemento a due, gli stessi algoritmi possono essere applicati per ottenere la rappresentazione di  $|N|$

# Regole per complementare a 2

- Esempio Alg. 1

00010101000



**Complementa a uno**

11101010111 +

---

1 =

11101011000

- Esempio Alg. 2

00010101000



**Complementa a uno fino alla cifra 1 meno significativa**

11101011000

# Regole per complementare a 2

## Alg. 1: inverti tutti i bit e somma 1 (dimostrazione)

- La rappresentazione in complemento a 2 del numero **negativo**  $-|N|$  è:

$$1 \ d_{n-2} \dots d_1 \ d_0,$$

- Il valore è:

$$-|N| = -2^{n-1} + 2^{n-2} * d_{n-2} + \dots + 2^1 * d_1 + 2^0 * d_0$$

- Allora:

$$\begin{aligned} |N| &= 2^{n-1} - 2^{n-2} * d_{n-2} - \dots - 2^1 * d_1 - 2^0 * d_0 = \\ &= (2^{n-1}-1)+1 - (2^{n-2} * d_{n-2} + \dots + 2^1 * d_1 + 2^0 * d_0) = \\ &\Rightarrow (2^{n-1} * 0 + 2^{n-2} * 1 + \dots + 2^1 * 1 + 2^0 * 1) - \\ &\quad (2^{n-2} * d_{n-2} + \dots + 2^1 * d_1 + 2^0 * d_0) + 1 = \\ &= (2^{n-1} * 0 + 2^{n-2} * (1 - d_{n-2}) + \dots + 2^0 * (1 - d_0)) + 1 \end{aligned}$$

Sommando e  
sottraendo 1

$$2^{n-1} - 1 = \sum_{i=0}^{n-2} 2^i$$

poiché (serie  
geometrica):

$$\sum_{i=0}^n q^i = \frac{1 - q^{n+1}}{1 - q}$$

**$\Rightarrow$  Invertendo tutti i bit della rappresentazione di  $-|N|$  otteniamo  $0(1 - d_{n-2}) \dots (1 - d_0)$**

**dove  $0 = 1 - d_{n-1}$**

**Il valore del numero *complementato* (positivo) è:**

$$2^{n-2} * (1 - d_{n-2}) + \dots + 2^0 * (1 - d_0)$$

**Sommando 1, otteniamo proprio il valore di  $|N|$  sopra derivato**

# Regole per complementare a 2

Alg. 1: inverti tutti i bit e somma 1 (dimostrazione - continuazione)

- Se N è un numero **positivo**, la rappresentazione di N sarà

$$0 d_{n-2} \dots d_1 d_0$$

il cui valore è:

$$2^{n-2} * d_{n-2} + \dots + 2^1 * d_1 + 2^0 * d_0$$

- Quindi il valore del numero negativo -N sarà uguale a

$$\begin{aligned} -N &= -2^{n-2} * d_{n-2} - \dots - 2^0 * d_0 = \\ &= (2^{n-1}-1) - (2^{n-1}-1) - 2^{n-2} * d_{n-2} - \dots - 2^0 * d_0 = \\ &= (2^{n-2} * 1 + \dots + 2^0 * 1) - (2^{n-1}-1) - \\ &\quad (2^{n-2} * d_{n-2} + \dots + 2^0 * d_0) = \\ &= -2^{n-1} + (2^{n-2} * (1-d_{n-2}) + \dots + 2^0 * (1-d_0)) + 1 \end{aligned}$$

Sommando e  
sottraendo  $(2^{n-1}-1)$

⇒ Invertendo tutti i bit della rappresentazione di N otteniamo  $1(1-d_{n-2})\dots(1-d_0)$

dove  $1=1-d_{n-1}$

Il valore del numero **complementato** (negativo) è:

$$-2^{n-1} + 2^{n-2} * (1-d_{n-2}) + \dots + 2^0 * (1-d_0)$$

Sommando 1, otteniamo proprio il valore di -N sopra derivato

# Estensione dei bit della rappresentazione

Regola: copiare il bit più significativo (bit di segno) negli altri bit

- 0010 -> 0000 0010
- 1010 -> 1111 1010

L'estensione del bit di segno funziona anche per i numeri negativi

il complemento a 2 del numero negativo 1010 è 110, indipendentemente dal numero di 1 iniziali (es. 1...1010)

Esempio di applicabilità dell'estensione del segno:

- un operando di una istruzione macchina può essere più corto di una word (32 bit)
- l'operando deve essere esteso nella corrispondente rappresentazione a 32 bit prima che i circuiti della CPU possano effettuare l'operazione aritmetica richiesta dall'istruzione

# Addizioni e sottrazioni

Le operazioni con i numeri binari in complemento a 2 sono *facili*

- sottraiamo usando semplicemente l'algoritmo dell'addizione
- il sottraendo (negativo) deve essere espresso in complemento a 2

Esempio:

Sottraz. dei valori assoluti      vs      Somma dei num. relativi in compl. 2

$$\begin{array}{r} 7- \\ \underline{6=} \\ 1 \end{array} \qquad \begin{array}{r} 0111- \\ \underline{0110=} \\ 0001 \end{array}$$

$$\begin{array}{r} 7 \quad + \\ \underline{(-6)=} \\ 1 \end{array} \qquad \begin{array}{r} 0111+ \\ \underline{1010=} \\ 0001 \end{array}$$

# Addizioni e sottrazioni

- Per sottrarre  $N1 - N2$  (numeri di  $n$ -bit),  $N1 > 0$  e  $N2 > 0$

- sommiamo  $(N1 + (2^n - N2)) \bmod 2^n$

- Perché questo tipo di *somma algebrica* funziona ?

Perché in questo caso non possiamo avere un **overflow** ?

- se  $N1 > N2$ , il risultato dovrà essere **positivo**.

$\Rightarrow (N1 + 2^n - N2) \bmod 2^n = N1 - N2$  poiché  $(N1 + 2^n - N2) > 2^n$

$$\begin{array}{r} 7- \quad 0111+ \\ 6= \quad 1010= \\ 1 \quad 10001 \end{array}$$

- se  $N1 < N2$ , il risultato dovrà essere **negativo**. Il modulo non avrà effetto, poiché  $(N1 + 2^n - N2) < 2^n$

$\Rightarrow (N1 + 2^n - N2) \bmod 2^n = 2^n - (N2 - N1)$

che corrisponde alla rappresentazione in complemento a due di  $-(N2 - N1)$

$$\begin{array}{r} 5- \quad 0101+ \\ 6= \quad 1010= \\ -1 \quad 1111 \end{array}$$



# Come scoprire gli Overflow

No overflow se **somma** di numeri con **segno discorde**

No overflow se **sottrazione** di numeri con **segno concorde**

**Overflow** se si ottiene un numero con segno diverso da quello aspettato, ovvero se si sommano algebricamente due numeri con segno concorde, e il segno del risultato è diverso. Quindi otteniamo **overflow**:

- se sommando due positivi si ottiene un negativo
- se sommando due negativi si ottiene un positivo
- se sottraendo un negativo da un positivo si ottiene un negativo
- se sottraendo un positivo da un negativo si ottiene un positivo

Considera le operazioni  $A + B$ , e  $A - B$

- Può verificarsi overflow se  $B$  è 0 ?

# Come scoprire gli overflow

Tabella riassuntiva delle possibilità di overflow:

segni degli operandi	tipo di operazione	
	somma	sottrazione
	pos. pos.	SI (neg.)
	neg. neg.	NO
	pos. neg.	SI (neg.)
	neg. pos.	SI (pos.)

# Come scoprire gli Overflow

**Somma** algebrica di due **numeri positivi** A e B la cui somma non può essere rappresentata su n bit in complemento a 2

- Overflow se  $A+B \geq 2^{n-1}$

A=01111 B=00001 (*OVERFLOW*  $\Rightarrow$  due ultimi riporti *discordi*)

A=01100 B=00001 (*NON OVERFLOW*  $\Rightarrow$  due ultimi riporti *concordi*)

01

```
01111+
00001=
-----
10000
```

00

```
01100+
00001=
-----
01101
```

# Come scoprire gli Overflow

**Somma** algebrica di due **numeri negativi** A e B la cui somma non può essere rappresentata su n-bit in complemento a 2

- Overflow se  $|A|+|B|>2^{n-1}$

A=10100 B=10101 (**OVERFLOW**  $\Rightarrow$  due ultimi riporti *discordi*)

A=10111 B=11101 (**NON OVERFLOW**  $\Rightarrow$  due ultimi riporti *concordi*)

10	11
10100+	10111+
<u>10101=</u>	<u>11101=</u>
01001	10100

Sembra quindi che la **regola dei riporti** funzioni sia per la somma di due numeri positivi che per la somma di due numeri negativi. Vedremo però che esiste un'eccezione a questa regola...

# Moltiplicazione e divisione tra interi

Oltre alle operazioni di somma e sottrazione tra interi, è necessario rendere disponibili anche le operazioni di **MOLTIPLICAZIONE** e **DIVISIONE** tra interi

- Più complicato che sommare/sottrarre
- Vengono realizzate tramite **shift e somme multiple**
  - più dispendioso in termini di tempo e di area di silicio occupata

Per semplicità posticipiamo la presentazione di queste operazioni...

# Numeri razionali (a virgola fissa)

Numeri con la **virgola** (o con il **punto**, secondo la convenzione anglosassone)

Nel sistema di numerazione posizionale in base  $B$ , con  $n$  cifre intere e  $m$  cifre frazionarie:

$$d_{n-1} \dots d_1 d_0, d_{-1} d_{-2} \dots d_{-m}$$

Il valore corrispondente è:

$$N = B^{n-1} * d_{n-1} + \dots + B^1 * d_1 + B^0 * d_0 + B^{-1} * d_{-1} + B^{-2} * d_{-2} + \dots + B^{-m} * d_{-m}$$

La notazione con  $n+m$  cifre è detta a **virgola fissa (fixed point)**

Conversione da base 10 a base 2

$$\blacksquare 10,5_{\text{dieci}} \quad 1010,1_{\text{due}}$$

# Conversione della parte frazionaria

Dato un numero frazionario  $N = 0,...$  espresso in base  $B$ , vogliamo trovare la sua rappresentazione in base 2. Procediamo per *moltiplicazioni successive*.

La rappresentazione di  $N$  in base 2 sarà

$0,d_{-1} d_{-2} \dots d_{-m}$

con  $d_{-i} \in \{0,1\}$  da determinare. Il valore è comunque:

$$N = 2^{-1} * d_{-1} + 2^{-2} * d_{-2} + \dots + 2^{-m} * d_{-m}$$

- se moltiplichiamo  $N$  per 2, la virgola si sposta a destra

$$2^0 * d_{-1} + 2^{-1} * d_{-2} + \dots + 2^{-m+1} * d_{-m}$$

- dopo aver moltiplicato per 2, la parte intera del numero diventa  $d_{-1}$

$$d_{-1}, d_{-2} \dots d_{-m}$$

Iterando il processo di moltiplicazione con la nuova parte frazionaria (fino a quando diventa nulla) si determina la sequenza di cifre binarie che corrisponde alla rappresentazione di  $N$ .

NOTA: il procedimento vale in realtà per conversioni da base  $B$  a base  $B'$  qualsiasi.

## Processo di conversione di $0,43_{\text{dieci}}$

	*2	Cifre frazionarie	
0,43	0,86	0	$d_{-1}$
0,86	1,72	1	$d_{-2}$
0,72	1,44	1	$d_{-3}$
0,44	0,88	0	$d_{-4}$
0,88	1,76	1	$d_{-5}$
0,76	1,52	1	$d_{-6}$
0,52	1,04	1	$d_{-7}$
0,04	0,08	0	$d_{-8}$
0,08	0,16	0	$d_{-9}$
0,16	...	...	...

$0,011011100\dots_{\text{due}}$

# Numeri razionali a virgola fissa: problemi

La notazione a virgola fissa (es.:  $n=8$  e  $m=8$ ) non permette di rappresentare numeri molto grandi o molto piccoli

Per numeri grandi:

- utile spostare la virgola a **destra** e usare la maggior parte dei bit della rappresentazione per la parte intera

$100000000000,0100$  parte intera non rappresentabile con  $n=8$  bit

Per numeri piccoli:

- utile spostare la virgola a **sinistra** e usare la maggior parte dei bit della rappresentazione per la parte frazionaria

$0,0000000000000001$  parte frazionaria non rappresentabile con  $n=8$  bit

Per ovviare a questi problemi si utilizza la rappresentazione dei numeri razionali a virgola mobile



# Numeri razionali (a virgola mobile)

Notazione in **virgola mobile**, o FP (Floating Point)

- si usa la *notazione scientifica*, con l'esponente per far fluttuare la virgola

- Segno, Esponente, Mantissa  $\Rightarrow$   $\pm 10^E * M$

$$0,121 \quad +10^0 * 0,121$$

$$14,1 \quad +10^2 * 0,141$$

$$-911 \quad -10^3 * 0,911$$

- Standard  $\Rightarrow$  Mantissa rappresentata come numero frazionario, con parte intera uguale a 0

# Esercizi

Considerare le somme già svolte per numeri naturali come somme di numeri interi in complemento a due e analizzare i possibili overflow.

Considerare la rappresentazione in virgola fissa con 8+8 bit e tradurre i numeri seguenti numeri decimali in numeri binari espressi in complemento a due:

$25,125_{10}$ ;  $52,375_{10}$ ;  $-35,625_{10}$ ;  $-63,75_{10}$

Scriverli poi in notazione scientifica

# Numeri razionali (a virgola mobile)

In base 2, l'esponente E si riferisce ad una potenza di 2

- Segno, Esponente, Mantissa  $\Rightarrow (-1)^S * 2^E * M$

Dati i bit disponibili per la rappresentazione FP, si suddividono in

- un bit per il segno
- gruppo di bit per E
- gruppo di bit per M



# Numeri razionali (a virgola mobile)

Una volta fissato il numero di bit totali per la rappresentazione dei numeri razionali a virgola mobile rimane da decidere:

- Quanti bit assegnare per la mantissa ?  
(maggiore è il numero di bit e maggiore è l'accuratezza con cui si riescono a rappresentare i numeri)
- Quanti bit assegnare per l'esponente ?  
(aumentando i bit si aumenta l'intervallo dei numeri rappresentabili)

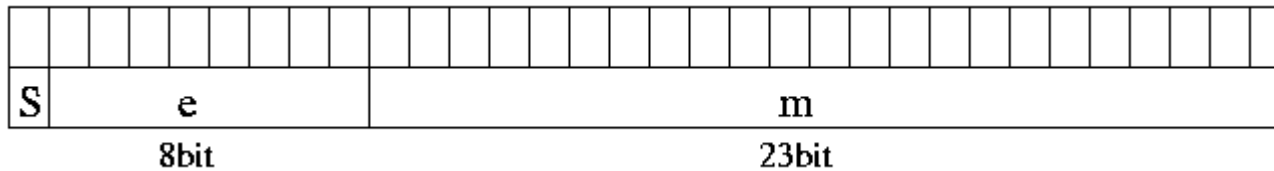
**OVERFLOW:** si ha quando l'esponente positivo è troppo grande per poter essere rappresentato con il numero di bit assegnato all'esponente

**UNDERFLOW:** si ha quando l'esponente negativo è troppo grande (in valore assoluto) per poter essere rappresentato con il numero di bit assegnato all'esponente

Il MIPS utilizza lo Standard IEEE754 per rappresentare i numeri a virgola mobile

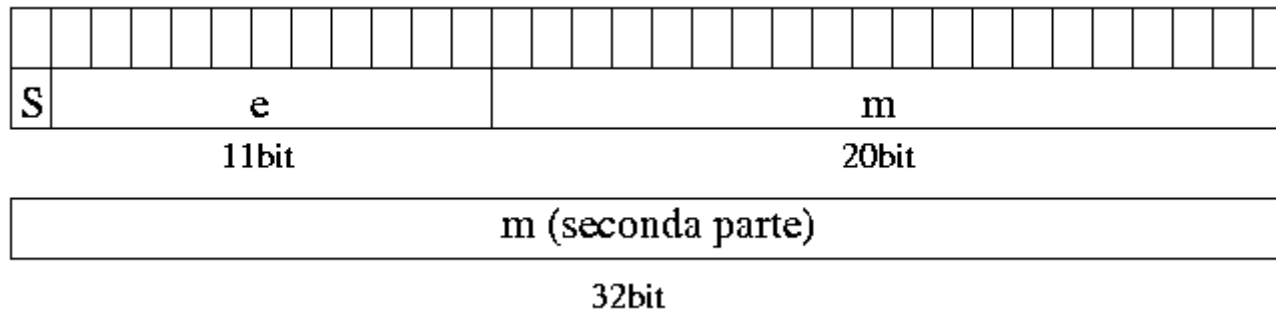
# Standard IEEE754

Standard IEEE754: Singola precisione (32 bit)



L'intervallo di valori (decimali) rappresentabili è  $\pm \sim 10^{-44.85} \text{ -- } \sim 10^{38.53}$

Standard IEEE754: Doppia precisione (64 bit)



L'intervallo di valori (decimali) rappresentabili è  $\pm \sim 10^{-323.3} \text{ -- } \sim 10^{308.3}$

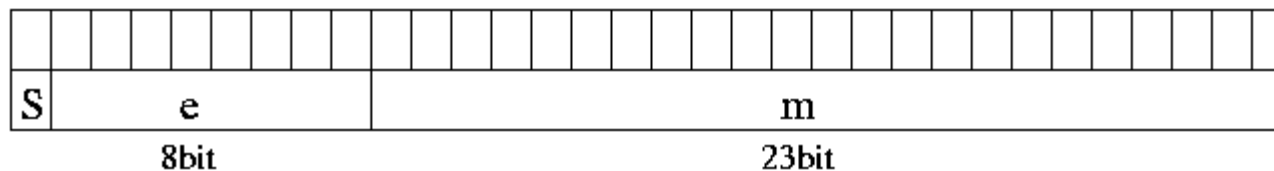
# Standard IEEE754

Mantissa =  $d_0 . d_{-1} \dots d_{-k}$

Lo standard IEEE754 utilizza la *notazione scientifica normalizzata*, che considera  $d_0 = 1$

- Poiché  $d_0 = 1$  è sempre uguale a 1, nella rappresentazione  $d_0$  è implicito
- si guadagna così un bit per la rappresentazione della mantissa e si aumenta, allo stesso tempo, l'accuratezza dei numeri razionali espressi

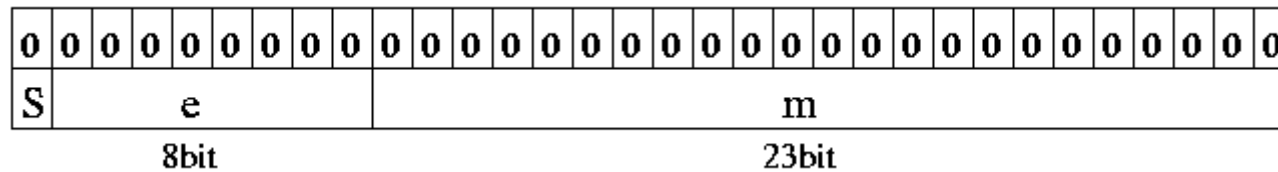
## Rappresentazione standard:



dove il numero rappresentato è:  $(-1)^s \cdot (1 + m) \cdot 2^e$

# Standard IEEE754

Per convenzione, lo *zero* viene rappresentato come:



Si noti che in generale, se  $m = 00\dots 0$ , il valore rappresentato  $1+m=1 \neq 0$

# Standard IEEE754

Come rappresentare i numeri FP in modo che sia facile realizzare il confronto?

- Vorremmo poterli confrontare come se fossero *interi*

Lo standard IEEE754 ha scelto:

- la posizione del segno in modo che sia facile il test  $\geq 0$  ,  $< 0$
- la posizione dell'esponente (in posizione più significativa) per semplificare l'ordinamento dei numeri rappresentati
  - Maggiore è l'esponente, maggiore è il valore assoluto del numero

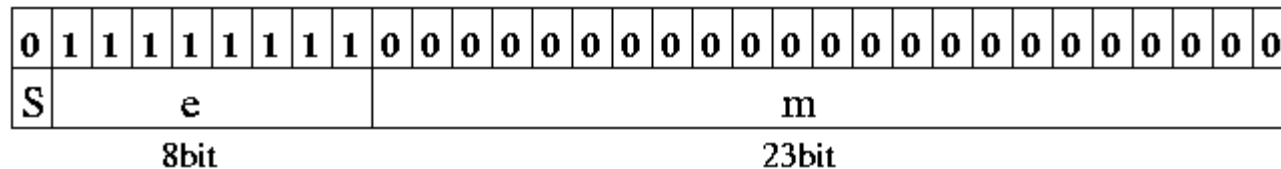


# Standard IEEE754

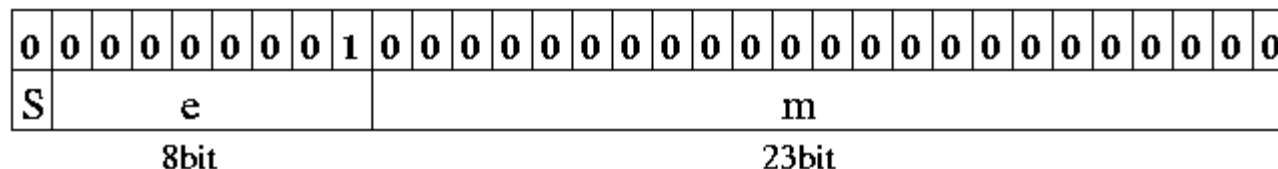
Gli esponenti negativi (in complemento a due) rappresentano però un problema per questo tipo di ordinamento. Infatti:

- **esponente negativo ==> secondo bit più “significativo” uguale a 1**
- un numero FP positivo, ma con un esponente negativo, appare come un numero intero grande

Esempio:  $1.0 \cdot 2^{-1} = 0,5$



Esempio:  $1.0 \cdot 2^1 = 2$



# Standard IEEE754

Una buona rappresentazione che semplifica il confronto dovrebbe denotare

- l'esponente "più negativo" come  $00...0_2$
- l'esponente "più positivo" come  $11...1_2$

Lo standard IEEE754 utilizza per questo motivo la **notazione polarizzata**

Singola precisione -> polarizzazione pari a  $127 = 01111111_2$

Doppia precisione -> polarizzazione pari a  $1023 = 011111111111_2$

Ad esempio:

- esponente -125 rappresentato come  $-125+127 = 00000010_2$
- esponente -1 rappresentato come  $-1+127 = 126 = 01111110_2$
- esponente 0 rappresentato come  $0+127 = 127 = 01111111_2$
- esponente +1 rappresentato come  $1+127 = 128 = 10000000_2$
- esponente +125 rappresentato come  $125+127=252=11111100_2$

# Standard IEEE754

Il valore di un numero in notazione polarizzata è quindi:

$$(-1)^S \cdot (1+m) \cdot 2^{(e - \text{polarizzazione})}$$

Con la **notazione polarizzata** gli esponenti variano

- da -126 a +127 [1-254] per la singola precisione
- da -1022 a +1023 [1-2046] per la doppia precisione

Si consideri che:

- l'esponente 00...0 è così riservato per lo *zero*
- l'esponente 11...1 è così riservato per casi particolari (numero fuori dall'insieme dei valori rappresentabili -- infinito)

# Standard IEEE754

**Esempio:** scrivere - 0. 25<sub>10</sub> in notazione FP IEEE754:

$$-0.25_{10} = -0.01_2 = -1.0 \cdot 2^{-2} \Rightarrow (-1)^1 \cdot (1 + 0.0) \cdot 2^{(-2+127)}$$

1	0	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S	e								m																												
8bit									23bit																												

**Esempio:** Quale numero decimale rappresenta la seguente sequenza di bit, letta secondo lo standard IEEE754?

1	0	0	0	0	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S	e								m																												
8bit									23bit																												

esponente:  $00001011_2 = 11_{10}$

mantissa:  $0.01_2 = 0.25_{10}$

Il numero rappresentato è:  $(-1)^1 \cdot (1 + 0.25) \cdot 2^{(11-127)} = -1.25 \cdot 2^{(-116)}$

# Somma di numeri FP

Algoritmo per sommare numeri FP

**Esempio:** eseguire  $(5_{10} + 3.625_{10})$   
assumendo una precisione di 4 bit

$$5_{10} = 101_2 = 1.01 \cdot 2^2$$

$$3.625_{10} = 11.101_2 \cdot 2^0 = 1.1101 \cdot 2^1$$

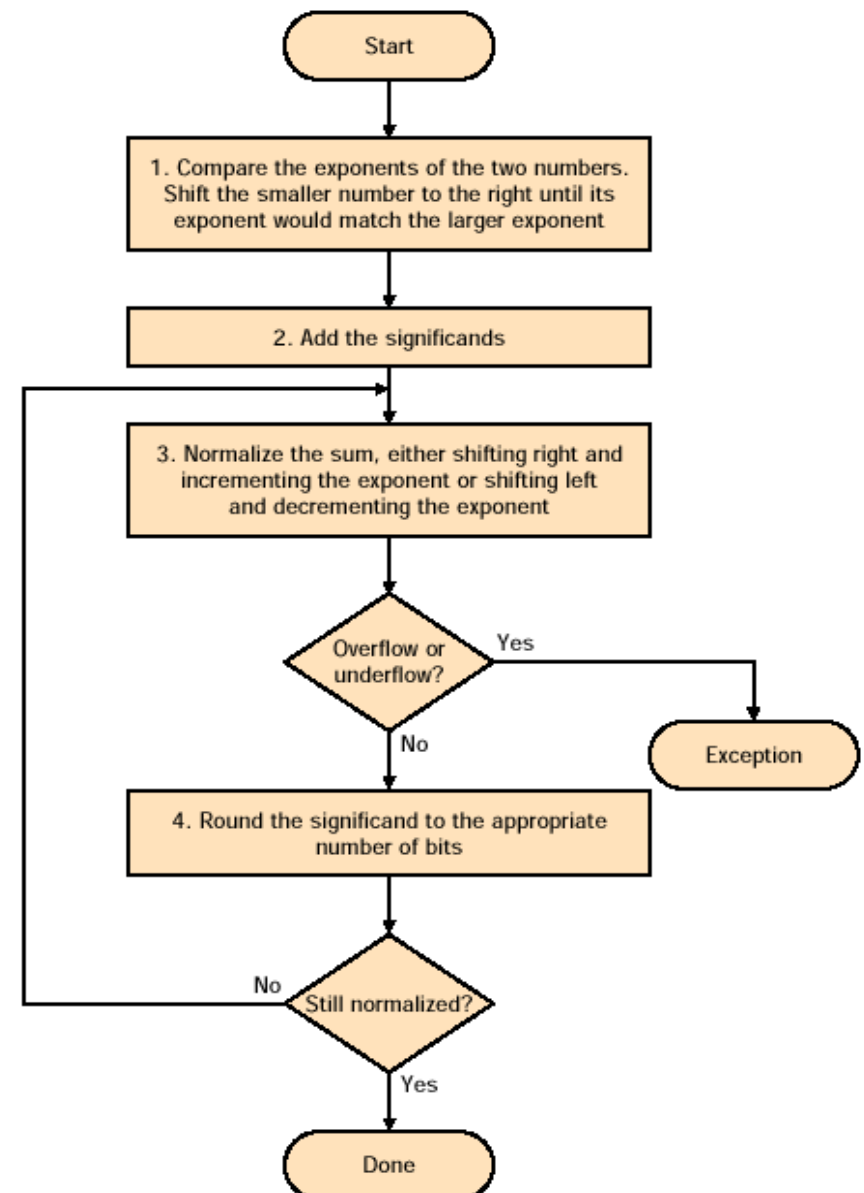
*Shifting:*  $3.625_{10} = 0.11101 \cdot 2^2$

*Somma:*

$$\begin{array}{r} 1.01000 + \\ 0.11101 = \\ \hline 10.00101 \cdot 2^2 \end{array}$$

*Normalizzazione:*  $1.000101 \cdot 2^3$

*Arrotondamento:*  $1.0001 \cdot 2^3$



# Somma di numeri FP (Esercizio)

Eseguire la somma algebrica di:

N1=

1	1	0	0	0	0	0	1	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

N2=

[illegible]

$N_1 < N_2$  con  $|N_1| > |N_2| \Rightarrow$  quindi segno negativo

Esponente:

$$N_1: 10000010_2 = 130_{10} \implies 130 - 127 = 3$$

$$N_2: 01111110_2 = 126_{10} \Rightarrow 126 - 127 = -1$$

**Allineamento mantissa** di  $N_2$  (shift a destra):

$$1.1111 \cdot 2^{-1} ==> 0.00011111 \cdot 2^3$$

# Somma di numeri FP (Esercizio)

Somma algebrica delle mantisse:

Esprimo le mantisse in complemento a due (aggiungo un bit per segno)

$N_1$ : 01.01010100 ma  $N_1$  è negativo  $\Rightarrow$  10.10101100

$N_2$ : 00.00011111 ma  $N_2$  è positivo  $\Rightarrow$  00.00011111

Somma:

10.10101100 +

00.00011111 =

-----

10.11001011 numero **negativo**!

Per ottenere il valore assoluto della mantissa è necessario trovare il suo opposto.

Risultato: 1.00110101

# Somma di numeri FP (Esercizio)

## Normalizzazione:

- La mantissa ottenuta è già normalizzata: 1.00110101
- per cui non è necessario modificare l'esponente del risultato

## Arrotondamento/Troncamento

- La mantissa ottenuta è rappresentabile su 23 bit. Quindi non dobbiamo introdurre errori nella rappresentazione (la rappresentazione del numero è in questo caso accurata)

**Numero FP risultato:**

**$s_{ris} = 1$**

**$e_{ris} = 130$**

**$m_{ris} = 001101010...0$**



# Somma di numeri FP (Esercizio)

## Esercizio : (continua)

**N<sub>1</sub>:**

1	1	0	0	0	0	0	1	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S	e								m																									
	8bit								23bit																									

+

**N<sub>2</sub>:**

0	0	1	1	1	1	1	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S	e								m																									
	8bit								23bit																									

=

-----

**RIS:**

1	1	0	0	0	0	0	1	0	0	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S	e								m																									
	8bit								23bit																									

Fare la riprova trasformando in decimale i due addendi della somma e il risultato

# Somma FP: errori

L'aritmetica FP, a causa dei limiti di rappresentazione (mantissa limitata), può introdurre **errori di accuratezza** nei risultati delle operazioni

Vediamo un esempio di calcolo erraneo, da cui possiamo desumere che **la somma in virgola mobile non è sempre associativa**:  
in generale, **non è quindi vero che  $x+(y+z) = (x+y)+z$**

Il fenomeno si può osservare quando dobbiamo sommare due numeri molto *grandi in valore assoluto*, ma di segno opposto, con un altro numero molto piccolo

$$x = -1.5_{10} \cdot 10^{38}$$

$$y = +1.5_{10} \cdot 10^{38} \quad \text{con } x,y,z \text{ espressi in singola precisione}$$

$$z = 1.0_{10}$$

$$\begin{aligned} x+(y+z) &= -1.5 \cdot 10^{38} + (1.5 \cdot 10^{38} + 1.0) \\ &= -1.5 \cdot 10^{38} + 1.5 \cdot 10^{38} = 0.0_{10} \end{aligned}$$

quando allineo l'esponente più piccolo al più grande la mantissa del più piccolo non è rappresentabile con il numero di bit a disposizione

$$\begin{aligned} (x+y)+z &= (-1.5 \cdot 10^{38} + 1.5 \cdot 10^{38}) + 1.0 \\ &= 0.0 + 1.0 = 1.0_{10} \end{aligned}$$

# Rappresentazione informazione alfanumerica

Per rappresentare le lettere (maiuscole, minuscole, punteggiature, ecc.) è necessario fissare uno standard.

L'esistenza di uno standard permette la comunicazione di documenti elettronici (testi, programmi, ecc.), anche tra computer con processori diversi.

ASCII (American Standard Code for Information)

- in origine ogni carattere era rappresentato da una stringa di 7 bit
- 128 caratteri, da 0 a 7F
- codici da 0 a 1F usati per caratteri non stampabili (caratteri di controllo)

<b>0A</b>	<b>(Line Feed)</b>
<b>0D</b>	<b>(Carriage Return)</b>
<b>1B</b>	<b>(Escape)</b>

<b>20</b>	<b>(Space)</b>
<b>2C</b>	<b>,</b>
<b>2E</b>	<b>.</b>

<b>30</b>	<b>0</b>
<b>...</b>	<b>.....</b>
<b>39</b>	<b>9</b>

<b>41</b>	<b>A</b>
<b>...</b>	<b>.....</b>
<b>5A</b>	<b>Z</b>

<b>61</b>	<b>a</b>
<b>...</b>	<b>.....</b>
<b>7A</b>	<b>z</b>

# Codice ASCII a 7 bit

MSB \ LSB	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
010	spc	!	“	#	\$	%	&	‘	(	)	*	+	,	-	.	/
011	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
100	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
101	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
110	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
111	p	q	r	s	t	u	v	w	x	y	z	{		}	~	del

Esempio: il carattere parentesi graffa “ { “ ha codice ASCII decimale **123**, equivalente alla parola ASCII binaria di 7 bit: **111 1011** (si verifichi che la codifica di questo carattere è esatta).

# ASCII

Codici ASCII esteso a 8 bit

256 codici diversi non bastano a coprire i set di caratteri usati, ad esempio, nelle lingue latine, slave, turche, ecc.

- ISO (International Organization for Standardization) con concetto di code page
- ISO 8859-1 è il codice ASCII a 8 bit per Latin-1 (esempio l'inglese o l'italiano con le lettere accentate ecc.)
- ISO 8859-2 è il codice ASCII a 8 bit per Latin-2 (lingue latine slave cecoslovacco, polacco, e ungherese)
- ecc.

# UNICODE

Lo standard UNICODE è stato creato da un consorzio di gruppi industriali

- ulteriore estensione (ISO 10646) con codici a 16 bit (65536 codici diversi)
- i codici che vanno da 0000 a 00FF corrispondono a ISO 8859-1
  - per rendere più facile la conversione di documenti da ASCII a UNICODE

# Istruzioni macchina e codifica binaria

Le istruzioni macchina, ovvero il linguaggio che la macchina (processore) comprende, hanno bisogno anch'esse di essere codificate in binario

- devono essere *rappresentate* in binario in accordo ad un **formato** ben definito

Il linguaggio macchina è molto restrittivo

- il processore che studieremo sarà il **MIPS**, usato da Nintendo, Silicon Graphics, Sony...
- l'ISA del MIPS è simile ad altre architetture RISC sviluppate dal 1980
- le istruzioni aritmetiche del MIPS permettono solo operazioni elementari (add, sub, mult, div) tra coppie di operandi a 32 bit
- le istruzioni MIPS operano su particolari supporti di memoria denominati **registri**, la cui lunghezza è di 32 bit = 4 Byte

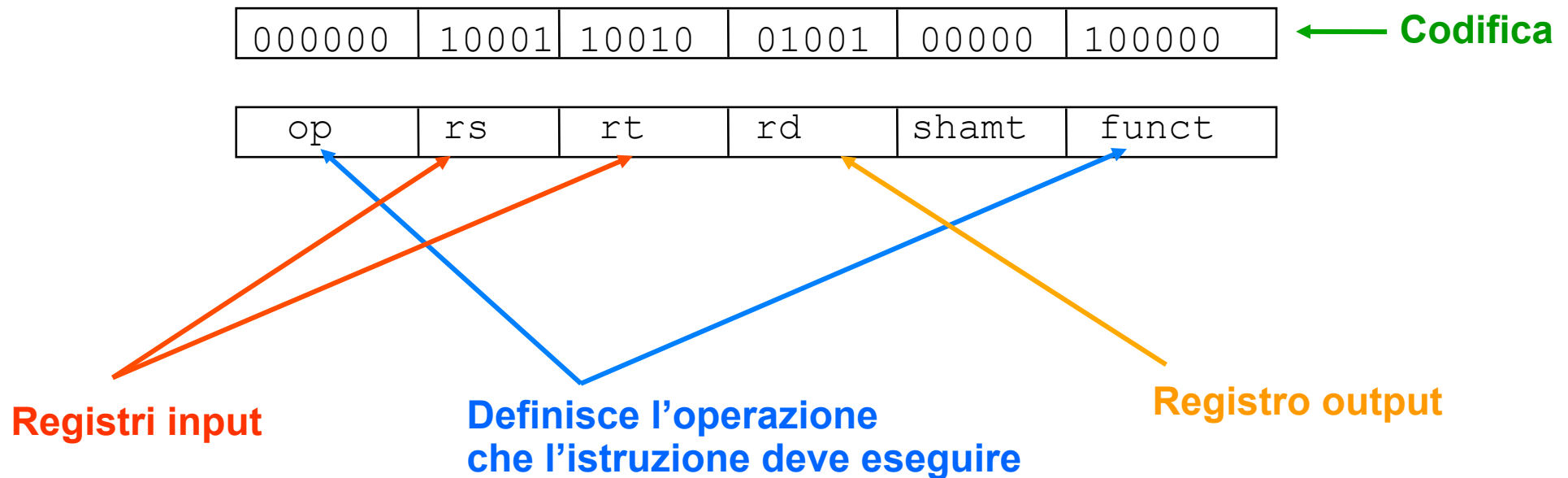
# Formato (codifica) delle istruzioni

Esempio:

`add $9, $17, $18` (semantica:  $\$9 = \$17 + \$18$ )

dove i registri sono identificati dai numeri 9, 17, 18

Formato delle istruzioni:





# Informazione e memoria

L'informazione, opportunamente codificata, ha bisogno di essere memorizzata nel calcolatore per essere utilizzata.

In particolare i programmi (e i dati) devono essere trasferiti nella **memoria principale** del computer per l'esecuzione.

Organizzazione della memoria:

- sequenza di **celle** (o **locazioni**) di lunghezza prefissata
- ogni cella è associata con un numero (chiamato **indirizzo**)
- se un indirizzo è espresso come numero binario di  $m$  bit
  - sono *indirizzabili*  $2^m$  celle diverse (da 0 a  $2^m - 1$ )
- indirizzi consecutivi  $\Rightarrow$  celle contigue
- nelle memorie attuali, ogni cella di memoria è lunga
  - $2^3 = 8 \text{ bit} = 1 \text{ Byte}$  (**memoria indirizzabile al Byte**)

# Informazione e memoria

I Byte consecutivi sono organizzati in gruppi

- ogni gruppo è una **Word**
- processori a **64 bit** (Word di 8 Byte) e a **32 bit** (Word di 4 Byte)
- le istruzioni aritmetiche operano su Word
  - la dimensione della Word stabilisce qual è il massimo intero rappresentabile

# Numeri binari magici

$2^3 = 8$  (8 bit = 1 Byte **B**)  
 $2^5 = 32$  (32 bit = 1 Word)

NOTA: la dimensione della Word dipende dal processore. Esistono processori dove la Word è di  $2^6 = 64$  bit, oppure di  $2^4 = 16$  bit

$2^{10} = 1024$  (**K** Kilo Migliaia - **KB** (kilobytes) - Kb (kilobits))  
 $2^{20}$  (**M** Mega Milioni - **MB**)  
 $2^{30}$  (**G** Giga Miliardi - **GB**)  
 $2^{40}$  (**T** Tera Migliaia di Miliardi - **TB**)  
 $2^{50}$  (**P** Peta Milioni di Miliardi - **PB**)

8 bit (1 B) è un'unità fondamentale:

- è l'unità di allocazione della memoria
- codici ASCII e UNICODE hanno dimensione, rispettivamente, 1B e 2B

Nome	Abbr	Fattore	dimensione
Kilo	K	$2^{10} = 1.024$	$10^3 = 1.000$
Mega	M	$2^{20} = 1.048.576$	$10^6 = 1.000.000$
Giga	G	$2^{30} = 1.073.741.824$	$10^9 = 1.000.000.000$
Tera	T	$2^{40} = 1.099.511.627.776$	$10^{12} = 1.000.000.000.000$
Peta	P	$2^{50} = 1.125.899.906.842.624$	$10^{15} = 1.000.000.000.000.000$

# Codici per scoprire o correggere errori

Le memorie elettroniche (o magnetiche come i dischi) memorizzano bit usando meccanismi che possono occasionalmente generare errori

- es.: un bit settato ad 1 viene poi letto uguale a 0, o viceversa

Formalizziamo il concetto di errore in una codifica a  $n$  bit

- C codifica corretta, C' codifica letta
- **Distanza di Hamming** tra le codifiche

$H(C, C')$  : **numero di cifre binarie differenti a parità di posizione**

- Possibili situazioni
  - $H(C, C')=0$  : significa che C e C' sono uguali (**OK**)
  - $H(C, C')=1$  : significa che C e C' differiscono per 1 solo bit
  - $H(C, C')=2$  : significa che C e C' differiscono per 2 soli bit
  - ecc.

Es: C = 11010011

C' = 11110010 allora  $H(C, C') = 2$

# Codici per scoprire e correggere errori

E' possibile definire codifiche che permettono di scoprire e anche correggere errori (codici correttori)

Vediamo un esempio di codifica che permette di scoprire errori singoli

# Parità

Per **scoprire** gli errori *singoli*, ovvero per accorgersi se  $H(C, C')=1$

- **aggiungiamo** un bit di *parità* alla codifica
- bit aggiuntivo posto a 0 (se il numero di bit a 1 è pari) o a 1 (se il numero di bit a 1 è dispari)
  - quindi il numero di bit a 1 è sempre pari in una codifica corretta
- se si verifica un errore *singolo* (un numero di errori *dispari*) in  $C'$ , allora il numero di bit 1 non sarà più pari
- purtroppo, con un singolo bit di parità, non scopriremo mai un numero di errori *doppio*, o in generale *pari*

# Parità

In verità, usare un bit di parità significa usare una *codifica non minimale* nella rappresentazione dell'informazione

- una *codifica minimale* usa tutte stringhe possibili
- in questo si usano solo la metà delle stringhe permesse su  $n+1$  bit
- la distanza “minima” di Hamming tra coppie di codifiche permesse è 2
  - es.:  $n=2$  con 1 bit di parità (no. bit pari)
  - Stringhe (codifiche) permesse
    - 000      011    101    110
  - Stringhe (codifiche) non permesse
    - 001      010    100    111

---

# Esercitazioni su rappresentazione dei numeri e aritmetica dei calcolatori

slide a cura di Salvatore Orlando & Marta Simeoni



## Interi *unsigned* in base 2

Si utilizza un alfabeto binario  $A = \{0,1\}$ , dove 0 corrisponde al *numero zero*, e 1 corrisponde al *numero uno*

$$d_{n-1} \dots d_1 d_0 \quad \text{con } d_i \in \{0,1\}$$

Qual è il numero rappresentato ?  $N = d_{n-1} \cdot 2^{n-1} + \dots + d_1 \cdot 2^1 + d_0 \cdot 2^0$

Quanti numeri sono rappresentabili su  $n$  bit?

00....00	=	0
00....01	=	1
00....10	=	2
...		
01....11	=	$2^{n-1} - 1$
10....00	=	$2^{n-1}$
...		
11....11	=	$2^n - 1$
100....00	=	$2^n$

Con sequenze di  $n$  bit  
sono rappresentabili  
 $2^n$  numeri naturali  
(da 0 a  $2^n - 1$ )

# Interi *unsigned* in base 2

---

I seguenti numeri naturali sono rappresentabili usando il numero di bit specificato ?

$20_{10}$  su 5 bit ? **SI**

$64_{10}$  su 6 bit ? **NO**

$500_{10}$  su 9 bit ? **SI**

$1025_{10}$  su 10 bit ? **NO**

Ricorda che:

$$2^0 = 1$$

$$2^1 = 2$$

$$2^2 = 4$$

$$2^3 = 8$$

$$2^4 = 16$$

$$2^5 = 32$$

$$2^6 = 64$$

$$2^7 = 128$$

$$2^8 = 256$$

$$2^9 = 512$$

$$2^{10} = 1024$$

....

# Conversione binario-decimale

---

**Esercizio:**  $1110101_2 = ???_{10}$

$$1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 =$$

$$64 + 32 + 16 + 0 + 4 + 0 + 1$$

**Soluzione:**  $1110101_2 = 117_{10}$

# Conversione decimale-binario

---

**Esercizio:**  $100_{10} = ???_2$

$100 : 2 = 50$	resto 0
$50 : 2 = 25$	resto 0
$25 : 2 = 12$	resto 1
$12 : 2 = 6$	resto 0
$6 : 2 = 3$	resto 0
$3 : 2 = 1$	resto 1
$1 : 2 = 0$	resto 1



**Soluzione:**  $100_{10} = 1100100_2$

# Conversione dec-bin: metodo più pratico

Scriviamo direttamente il numero decimale come somma di potenze di 2. Per far questo, sottraiamo via via le potenze di 2, a partire dalle più significative.

**Esercizio:**  $103_{10} = ???_2$

103	-	64	=	39	==>	$2^6$
39	-	32	=	7	==>	$2^5$
7	-	4	=	3	==>	$2^2$
3	-	2	=	1	==>	$2^1$
1	-	1	=	0	==>	$2^0$

**Allora**  $103_{10} = 2^6 + 2^5 + 2^2 + 2^1 + 2^0$

**Soluzione:**  $103_{10} = 1100111_2$

Ricorda che:

$$2^0 = 1$$

$$2^1 = 2$$

$$2^2 = 4$$

$$2^3 = 8$$

$$2^4 = 16$$

$$2^5 = 32$$

$$2^6 = 64$$

$$2^7 = 128$$

$$2^8 = 256$$

$$2^9 = 512$$

$$2^{10} = 1024$$

....

# Conversione binario-ottale e viceversa

---

**Esercizio:**  $10101111_2 = ???_8$

<u>10</u>	<u>101</u>	<u>111</u>
2	5	7

**Soluzione:**  $10101111_2 = 257_8$

**Esercizio:**  $635_8 = ???_2$

6	3	5
<u>110</u>	<u>011</u>	<u>101</u>

**Soluzione:**  $635_8 = 110011101_2$

# Base 16

---

Quali dei seguenti numeri esadecimali sono numeri sono corretti?

~~BED~~

~~CAR~~

938

DEAD

BEBE

A129

~~ACI~~

DECADE

~~BAG~~

DAD

~~4H3~~

# Conversione binario-esadecimale e viceversa

**Esercizio:**  $101111101101_2 = ???_{16}$

1011   1110   1101  
**B**   **E**   **D**

**Soluzione:**  $101111101101_2 = BED_{16} = 3053_{10}$

**Esercizio:**  $A3C9_{16} = ???_2$

**A**   **3**   **C**   **9**  
1010   0011   1100   1001

**Soluzione:**  $A3C9_{16} = 1010001111001001_2 = 41929_{10}$

Ricorda che:

$1_{10} = 1_{16} = 0001_2$   
 $2_{10} = 2_{16} = 0010_2$

.....

$9_{10} = 9_{16} = 1001_2$   
 $10_{10} = A_{16} = 1010_2$   
 $11_{10} = B_{16} = 1011_2$   
 $12_{10} = C_{16} = 1100_2$   
 $13_{10} = D_{16} = 1101_2$   
 $14_{10} = E_{16} = 1110_2$   
 $15_{10} = F_{16} = 1111_2$



# Interi *signed* in complemento a 2

Come si riconosce un numero **positivo** da uno **negativo**?

- Positivo  $\Rightarrow$  bit più significativo 0
- Negativo  $\Rightarrow$  bit più significativo 1

Su  **$n$  bit** sono rappresentabili  **$2^n$  interi unsigned** (da 0 a  $2^n-1$ )

Sempre su  **$n$  bit**, quanti **interi signed** in complemento a 2 ?

$$0\ldots\ldots 00 = 0$$

$$0\ldots\ldots 01 = 1$$

...

$$01\ldots\ldots 11 = 2^{n-1}-1 \text{ (massimo pos.)}$$

$$10\ldots\ldots 00 = -2^{n-1} \text{ (minimo neg.)}$$

...

$$11\ldots\ldots 11 = -1$$

in totale sempre  
 $2^n$  numeri

Dato  $N > 0$ , il numero  $-N$  si rappresenta su  **$n$  bit** con il numero unsigned  $2^n - N$

$$-1 \Rightarrow 2^n - 1 \quad (1\ldots\ldots 1)$$

$$-2^{n-1} \Rightarrow 2^n - 2^{n-1} = 2^{n-1} \quad (10\ldots\ldots 0)$$

# Complemento a 2

**Esercizio:** Rappresentare  $-35_{10}$  in complemento a 2 su 8 bit

$$00100011_2 = +35_{10}$$



Complemento a uno

$$\begin{array}{r} 11011100 \\ + \\ 1 \\ \hline \end{array}$$

$$\begin{array}{r} 11011101 \end{array}$$

**Soluzione:**  $-35_{10} = 11011101_2$

# Complemento a 2

---

**Esercizio:** Rappresentare -35 in complemento a 2 su 8 bit

$$00100011_2 = +35_{10}$$



$$11011101_2$$

Inverti (complementa a 1) tutti i bit a sinistra del bit "1" meno significativo

# Complemento a 2

---

**Esercizio:** Quale numero decimale rappresenta il seguente numero binario in complemento a due?

1111 1111 1111 1111 1111 1110 0000 1100<sub>2</sub>



0000 0000 0000 0000 0000 0001 1111 0100<sub>2</sub> =

$$2^2 + 2^4 + 2^5 + 2^6 + 2^7 + 2^8 = 500_{10}$$

**Soluzione:** il numero è  $-500_{10}$

# Complemento a 2: somma e sottrazione

**Esercizio:** eseguire  $53_{10} - 35_{10}$  in complemento a due su 8 bit

$$35_{10} = 00100011_2 \quad \text{complementando: } -35_{10} = 11011101_2$$

$$\begin{array}{r} 53_{10} - \\ 35_{10} = \\ \hline 18_{10} \end{array}$$

$\Rightarrow$

$$\begin{array}{r} 53_{10} + \\ (-35)_{10} = \\ \hline 18_{10} \end{array}$$

$\Rightarrow$

$$\begin{array}{r} 11111101 \\ 00110101_2 + \\ 11011101_2 = \end{array}$$

$$\hline (100010010_2) \bmod 2^8$$



$$00010010_2 = 18_{10}$$

# Complemento a 2: somma e sottrazione

**Esercizio:** eseguire  $15_{10} - 38_{10}$  in complemento a due su 8 bit

$$38_{10} = 00100110_2 \quad \text{complementando: } -38_{10} = 11011010_2$$

$$\begin{array}{r} 15_{10} - \\ 38_{10} = \\ \hline -23_{10} \end{array}$$

$\Rightarrow$

$$\begin{array}{r} 15_{10} + \\ (-38)_{10} = \\ \hline -23_{10} \end{array}$$

$\Rightarrow$

$$\begin{array}{r} 00011110 \\ 00001111_2 + \\ 11011010_2 = \end{array}$$

$$\hline (011101001_2) \bmod 2^8$$



$$00010111_2 = 23_{10}$$

# Overflow

---

In quali dei seguenti casi si può ottenere overflow?

- somma di due numeri con segno concorde? SI
- somma di due numeri con segno discorde? NO
- sottrazione di due numeri con segno concorde? NO
- sottrazione di due numeri con segno discorde? SI

# Come scoprire l'overflow

---

Primo caso:

**somma** algebrica di due **numeri positivi** A e B. Si ha overflow se  $A+B \geq 2^n - 1$ , ovvero se  $A+B$  non può essere rappresentata su n bit in complemento a due.

Esempi:

A=01111 B=00001 (**OVERFLOW**  $\Rightarrow$  due ultimi riporti discordi)

A=01100 B=00001 (**NON OVERFLOW**  $\Rightarrow$  due ultimi riporti concordi)

01  
01111+  
00001=  
            
10000

00  
01100+  
00001=  
            
01101



# Come scoprire l'overflow

---

Secondo caso:

**somma** algebrica di due **numeri negativi** A e B. Si ha overflow se  $|A| + |B| > 2^n - 1$ , ovvero se  $|A| + |B|$  non può essere rappresentata su n bit in complemento a due.

Esempi:

A=10100 B=10101 (**OVERFLOW**  $\Rightarrow$  due ultimi riporti *discordi*)

A=10111 B=11101 (**NON OVERFLOW**  $\Rightarrow$  due ultimi riporti *concordi*)

10

```
10100+
10101=
-----
01001
```

11

```
10111+
11101=
-----
10100
```

# Esercizio

Considerate i numeri esadecimali  $x_1 = 7A$   $x_2 = 13$   $x_3 = FF$   
 $x_4 = C1$   $x_5 = 84$

Scrivere i cinque numeri in codice binario a 8 bit

$$x_1 = \underline{0111} \ \underline{1010} \quad x_2 = \underline{0001} \ \underline{0011} \quad x_3 = \underline{1111} \ \underline{1111}$$

$$x_4 = \underline{1100} \ \underline{0001} \quad x_5 = \underline{1000} \ \underline{0100}$$

Interpretare il codice binario in complemento a due ed eseguire le operazioni

$$x_1 - x_2; \quad x_3 + x_4; \quad x_4 + x_5; \quad x_4 - x_1$$

$$\begin{array}{r} 11111000 \\ x_1 \ 01111010 + \\ -x_2 \ 11101101 = \end{array}$$

$$\hline (101100111) \bmod 2^8 = 01100111 \quad \text{overflow?}$$

## Esercizio (continua)

$$\begin{array}{r} 11111111 \\ x_3 \quad 11111111 + \\ x_4 \quad 11000001 = \\ \hline (111000000) \bmod 2^8 = 11000000 \quad \text{overflow?} \end{array}$$

$$\begin{array}{r} 10000000 \\ x_4 \quad 11000001 + \\ x_5 \quad 10000100 = \\ \hline (101000101) \bmod 2^8 = 01000101 \quad \text{overflow?} \end{array}$$

$$\begin{array}{r} 10000000 \\ x_4 \quad 11000001 + \\ -x_1 \quad 10000110 = \\ \hline (101000111) \bmod 2^8 = 01000111 \quad \text{overflow?} \end{array}$$

# Esercizio - caso particolare

Si ricorda che l'opposto del numero negativo più piccolo su *n bit* non può essere rappresentato in complemento a due

- codifica non simmetrica

Supponiamo quindi:

- di lavorare con rappresentazioni in complemento a due su **3 bit**
- di dover effettuare la sottrazione  $x-y$   
dove  $y=100_2$  è il minimo numero rappresentabile ( $y=-4_{10}$ )

**Esercizio:** calcolare  $x-y$  usando il solito algoritmo, dove  $x=001_2$

000	
X 001 +	
-Y 100	← Il complemento a due non ha effetto
<hr/>	
(0101) mod $2^3 = 101$	<b>OVERFLOW</b> , anche se riporti concordi !!

Abbiamo infatti sottratto un numero negativo da un numero  $\geq 0$   
⇒ **il segno atteso è positivo**

## Esercizio - caso particolare (continua)

**Esercizio:** calcolare  $x-y$ , dove  $x=111_2$

- poiché  $x=111_2$  allora  $x = -1_{10}$
- non dovremmo avere overflow, poiché vogliamo effettuare la somma algebrica di due numeri con segno discorde
- il risultato da ottenere è  $x-y = -1_{10} - (-4_{10}) = 3_{10}$

100

X 111 +

-Y 100

(1011) mod  $2^3 = 011 = 3_{10}$

Il complemento a due  
non ha effetto

corretto

**NO OVERFLOW** , anche  
se riporti discordi !!

Abbiamo infatti sottratto un numero negativo da un numero negativo  $\Rightarrow$  **in questo caso non si può verificare overflow**

# Procedura generale per determinare l'OVERFLOW

Alla luce dell'esempio precedente, guardare solo ai due ultimi riporti per controllare l'OVERFLOW potrebbe quindi portare a risultati erranei

Operazione	Segno 1° operando	Segno 2° operando	Segno atteso
Somma	+	+	+
Somma	+	-	qualsiasi
Somma	-	+	qualsiasi
Somma	-	-	-
Sottrazione	+	+	qualsiasi
Sottrazione	+	-	+
Sottrazione	-	+	-
Sottrazione	-	-	qualsiasi

Solo in questi casi si può verificare un **OVERFLOW**.

Possiamo controllarlo confrontando il bit di segno del risultato con il segno atteso.

# Numeri con la virgola (virgola fissa)

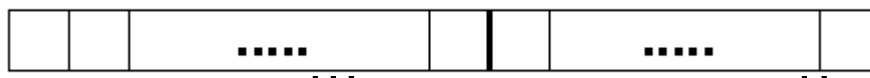
---

Data una base **B**, si assegnano:

**n** cifre per rappresentare la **parte intera**

**m** cifre per rappresentare la **parte frazionaria**

In base  $B=2$ , abbiamo quindi **m+n** bit per parte intera e frazionaria



Esempio:

$$d_{n-1} \dots d_1 d_0 . d_{-1} \dots d_{-m}$$

Qual è il numero rappresentato in base **B**?

$$N = d_{n-1} \cdot B^{n-1} + \dots + d_1 \cdot B^1 + d_0 \cdot B^0 + d_{-1} \cdot B^{-1} + \dots + d_{-m} \cdot B^{-m}$$


# Virgola fissa

**Esercizio:**  $23.625_{10} = ???_2$

(usare la rappresentazione in virgola fissa con  $n=8$ ,  $m=8$ )


Conversione parte intera:

$23 : 2 = 11$	resto 1
$11 : 2 = 5$	resto 1
$5 : 2 = 2$	resto 1
$2 : 2 = 1$	resto 0
$1 : 2 = 0$	resto 1



Conversione parte frazionaria:

$0.625 \times 2 = 1.25$	parte intera 1
$0.25 \times 2 = 0.50$	parte intera 0
$0.50 \times 2 = 1$	parte intera 1



**Soluzione:**  $23.625_{10} = 00010111.10100000_2$



# Numeri con virgola mobile

---

Un numero reale  $R$  può essere scritto in base  $B$  come

$$R = \pm m \cdot B^e$$

$m$  = mantissa

$e$  = esponente

$B$  = base

Esempi con  $B = 10$

- $R1 = 3.1569 \times 10^3$
- $R2 = 2054.00035 \times 10^{-6}$
- $R3 = 0.1635 \times 10^2$
- $R4 = 0.0091 \times 10^{-12}$

Notazione scientifica:

$$m = 0 . d_{-1} \dots d_{-k}$$

Notazione scientifica normalizzata:

$$m = d_0 . d_{-1} \dots d_{-k} \text{ con } d_0 \neq 0$$

# Numeri binari in virgola mobile

---

Rappresentando mantissa ed esponente in binario in **notazione scientifica normalizzata** si ottengono numeri del tipo:

$$\pm 1.xx...x \cdot 2^{yy...y}$$

Si osservi che:

Spostare la virgola (punto) a **destra** di  $n$  bit significa **decrementare** di  $n$  l'esponente

$$\begin{array}{lll} \text{es:} & 0.01 \cdot 2^3 & = 1.0 \cdot 2^1 \\ \text{Infatti} & 1 \cdot 2^{-2} \cdot 2^3 & = 1 \cdot 2^1 \end{array}$$

Spostare la virgola (punto) a **sinistra** di  $n$  bit significa **incrementare** di  $n$  l'esponente

$$\begin{array}{lll} \text{es:} & 100.011 \cdot 2^3 & = 1.00011 \cdot 2^5 \\ \text{Infatti} & (1 \cdot 2^2 + 1 \cdot 2^{-2} + 1 \cdot 2^{-3}) \cdot 2^3 & = (1 \cdot 2^0 + 1 \cdot 2^{-4} + 1 \cdot 2^{-5}) \cdot 2^5 \\ & 2^5 + 2^1 + 2^0 & = 2^5 + 2^1 + 2^0 \end{array}$$

# Numeri FP

**Esercizio:**  $10_{10} = ???_2$  FP

$$10_{10} = 1010_2 = 1010.0_2 \cdot 2^0 = 1.01 \cdot 2^3$$

**Esercizio:**  $151.25_{10} = ???_2$  FP

$$151_{10} = 128 + 16 + 4 + 2 + 1 = 10010111_2$$

$$0.25_{10} \times 2 = 0.50_{10} \quad \text{parte intera } 0$$

$$0.50_{10} \times 2 = 1_{10} \quad \text{parte intera } 1$$

$$0.25_{10} = 0.01_2$$

$$\text{Quindi } = 151.25_{10} = 10010111.01_2 = 1.001011101_2 \cdot 2^7$$

# Numeri FP

---

Una volta fissato il numero di bit totali per la rappresentazione dei numeri razionali rimane da decidere:

- Quanti bit assegnare per la mantissa ?

(maggiore è il numero di bit e maggiore è l'accuratezza con cui si riescono a rappresentare i numeri)

- Quanti bit assegnare per l'esponente ?

(aumentando i bit si aumenta l'intervallo dei numeri rappresentabili)

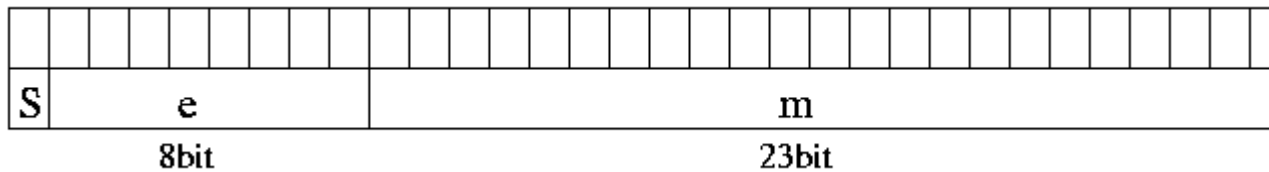
**OVERFLOW:** si ha quando l'esponente positivo è troppo grande per poter essere rappresentato con il numero di bit assegnato all'esponente

**UNDERFLOW:** si ha quando l'esponente negativo è troppo grande (in valore assoluto) per poter essere rappresentato con il numero di bit assegnato all'esponente

# Numeri FP in standard IEEE754

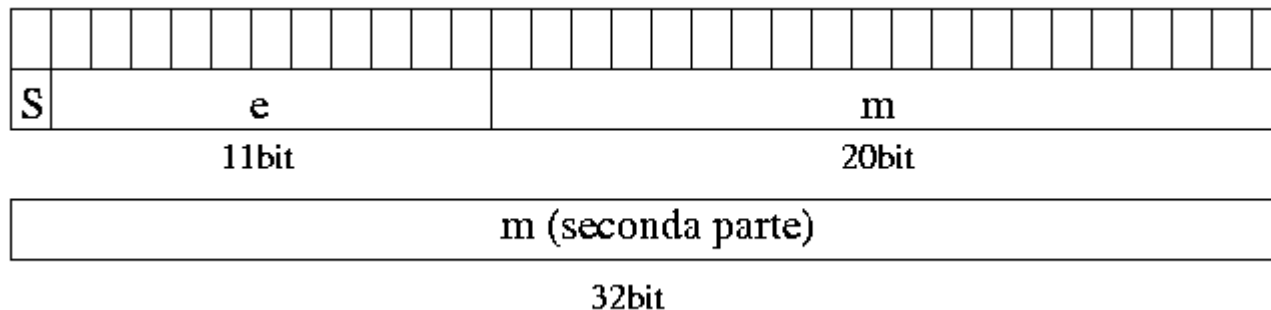
---

## Standard IEEE754: Singola precisione (32 bit)



L'intervallo di valori (decimali) rappresentabili è  $\pm \sim 10^{-44.85}$  --  $\sim 10^{38.53}$

## Standard IEEE754: Doppia precisione (64 bit)



L'intervallo di valori (decimali) rappresentabili è  $\pm \sim 10^{-323.3}$  --  $\sim 10^{308.3}$

# Standard IEEE754: Esempio

**Esempio:** scrivere  $-10.625_{10}$  in notazione floating point, usando lo standard IEEE754 in singola precisione

$$\begin{aligned} -10.625_{10} &= -(10 + 0.5 + 0.125) = -(2^3 + 2^1 + 1/2 + 1/8) = \\ &= -1010.101_2 = -1.010101 \cdot 2^3 = \\ &= (-1)^1 \cdot (1 + 0.010101) \cdot 2^3 \end{aligned}$$

Aggiungendo all'esponente la polarizzazione il numero diventa:

$$(-1)^1 \cdot (1 + 0.010101) \cdot 2^{(3+127)}$$

[illegible]

# Standard IEEE754: Esempio

**Esempio:** Quale numero decimale rappresenta la seguente sequenza di bit, letta secondo lo standard IEEE754?

0	1	0	0	0	1	1	0	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

esponente:  $10001101_2 = 128+8+4+1=141_{10}$

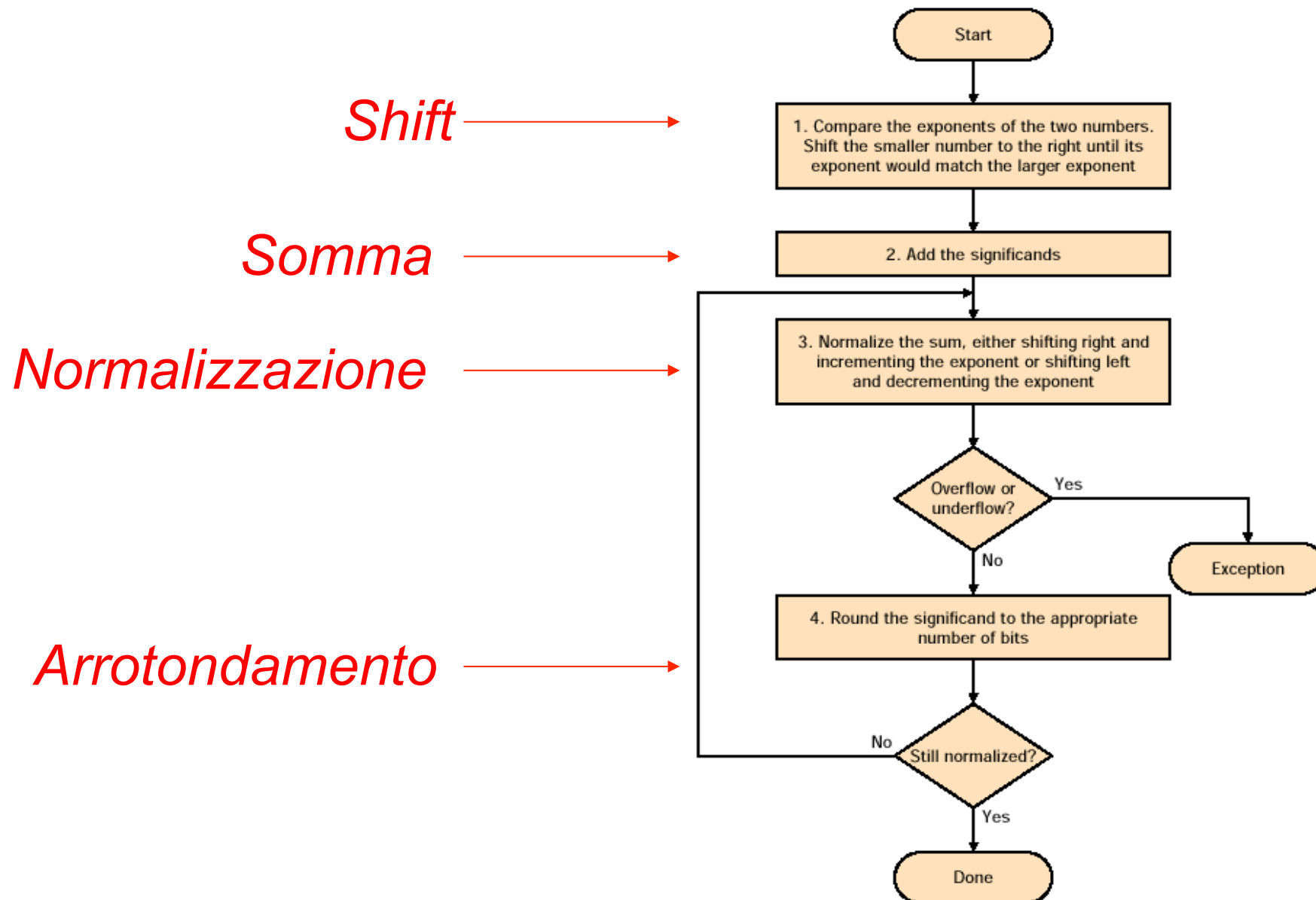
mantissa:  $1 + 0.10011_2 = 1 + 0.5 + 0.0625 + 0.03125 = 1.59375_{10}$

Quindi il numero è:  $1.59375_{10} \cdot 2^{(141 - 127)} = 1.59375_{10} \cdot 2^{14}$

oppure:

$$1,10011 * 2^{14} = 110011000000000 = 16384 + 8192 + 1024 + 512 = 26112$$

# Somma di numeri FP





# Somma di numeri FP con standard IEEE754

**Esercizio:** Dati i due numeri esadecimali A=C3160000 e B=42F80000

1. tradurre i numeri in binario
2. interpretare le sequenze di bit ottenuti come numeri FP espressi secondo lo standard IEEE754 in singola precisione
3. eseguirne poi la somma specificando tutti i passaggi
4. rappresentare il risultato ottenuto in esadecimale

**Soluzione:**

## 1. Traduzione in binario

A = 1100 0011 0001 0110 0000 0000 0000 0000

B = 0100 0010 1111 1000 0000 0000 0000 0000

## 2. Interpretazione di A e B come numeri FP:

A = 1 10000110 001011000000000000000000

B = 0 10000101 111100000000000000000000

cioè:

$$A = (-1)^1 \cdot 1.001011 \cdot 2^{10000110} = (-1)^1 \cdot 1.001011 \cdot 2^{01111111 + 111}$$

$$B = (-1)^0 \cdot 1.1111 \cdot 2^{10000101} = (-1)^0 \cdot 1.1111 \cdot 2^{01111111 + 110}$$

# Somma di numeri FP con standard IEEE754

---

## 3. Somma

- l'esponente di A è  $2^7$  mentre quello di B è  $2^6$ . Allineando B si ottiene:

$$A = 1.001011 \quad B = 0.111110$$

- A è negativo e quindi eseguo il complemento a due aggiungendo un bit per il segno

$$A = 10.110101 \quad B = 00.111110$$

- somma delle mantisse

$$A = 10.110101 +$$

$$B = 00.111110$$

-----

C = 11.110011 è un numero negativo il cui valore assoluto è 0.001101

- arrotondamento: non necessario

# Somma di numeri FP con standard IEEE754

---

- normalizzazione

$$C = (-1)^1 \cdot 0.001101 \cdot 2^{10000110} = (-1)^1 \cdot 1.101 \cdot 2^{01111111 + 100}$$

e quindi

$$\begin{aligned} C &= 1 \ 10000011 \ 101000000000000000000000 \\ &= 1100 \ 0001 \ 1101 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \end{aligned}$$

## 4. Rappresentazione di C in esadecimale

$$C = C1D00000$$

# Architettura degli Elaboratori

## Algebra booleana e circuiti logici

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

# Algebra & Circuiti Elettronici

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

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

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

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

# Algebra & Circuiti Elettronici

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

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

# Blocco logico

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



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

# Circuiti combinatori/sequenziali

## Circuito *combinatorio*

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

## Circuito *sequenziale*

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

Per ora ci concentriamo sui *circuiti combinatori*



# Funzioni Logiche

Una **funzione logica** è completamente specificata da

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

Vediamo in dettaglio...

# Tabelle di Verità

Funzione logica completamente specificata tramite *Tabella di Verità*

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

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

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

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

# Algebra Booleana

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

Esempio:  $E = \sim A \sim B C + A B \sim C$

- bit in *input* e *output* rappresentati tramite *variabili logiche* (con valori 0 o 1)
- input *combinati* tramite le operazioni di *somma* (OR), *prodotto* (AND) e *inversione* (NOT) *logica* dell'algebra di Boole
  - **OR ( $A+B$ ):** risultato uguale ad 1 (true) se almeno un input è 1 (true)
  - **AND ( $A \cdot B$ ):** risultato uguale ad 1 (true) solo se tutti gli input sono 1 (true)
  - **NOT ( $\sim A$ ):** risultato uguale all'inverso dell'input ( $0 \rightarrow 1$  oppure  $1 \rightarrow 0$ )

# Algebra Booleana

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

A	X
0	1
1	0

$$X = \sim A$$

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

$$X = A \cdot B$$

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

$$X = A + B$$

# Proprietà dell'algebra di Boole

## PROPRIETÀ

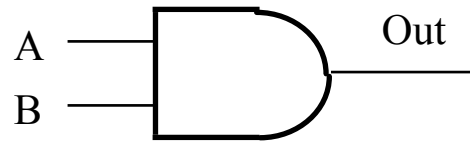
Identità:	$A+0=A$	$A \cdot 1=A$
Nullò:	$A+1=1$	$A \cdot 0=0$
Idempotente:	$A+A=A$	$A \cdot A=A$
Inverso:	$A+(\sim A)=1$	$A \cdot (\sim A)=0$
Commutativa:	$A+B=B+A$	$A \cdot B=B \cdot A$
Associativa:	$A+(B+C)=(A+B)+C$	$A \cdot (B \cdot C)=(A \cdot B) \cdot C$
Distributiva:	$A \cdot (B+C)=(A \cdot B)+(A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

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

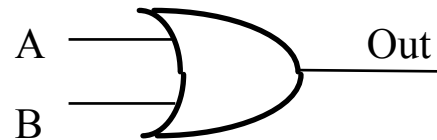
# Dalle equazioni logiche ai circuiti combinatori

## Porte logiche

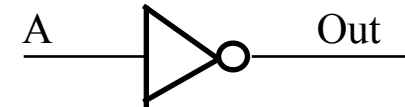
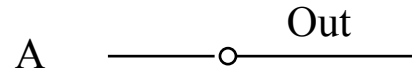
■ *AND*:       $\text{Out} = A \cdot B$



■ *OR*:       $\text{Out} = A + B$

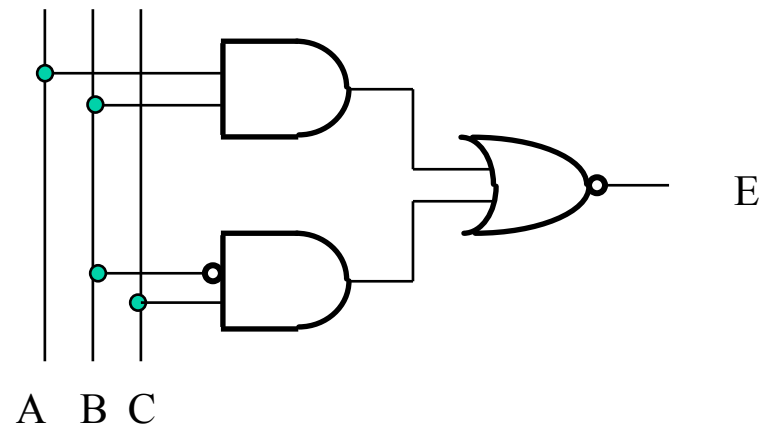


■ *NOT*:       $\text{Out} = \sim A$



Esempio di equazione e corrispondente circuito:

■  $E = \sim((AB) + (\sim BC))$



# NAND e NOR

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

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

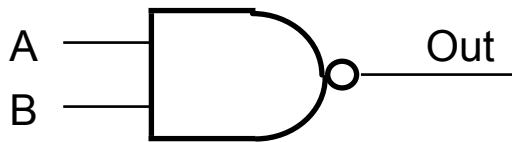
Si può dimostrare che le operazioni NAND o NOR (e le corrispondenti porte) sono sufficienti per implementare qualsiasi funzione logica

# NAND e NOR

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

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

**NAND: porta e tabella di verità**



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

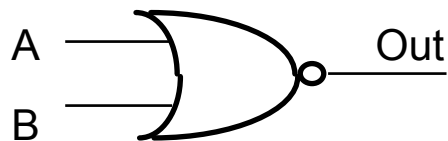


# NAND e NOR

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

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

**NOR: porta e tabella di verità**

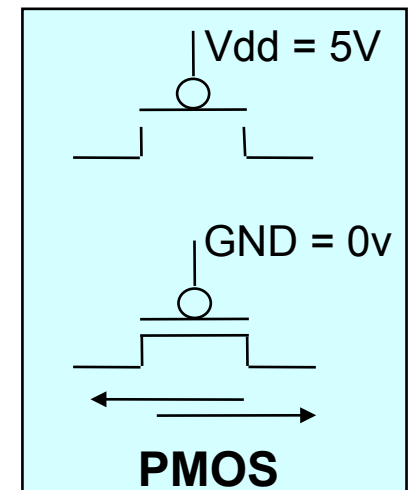
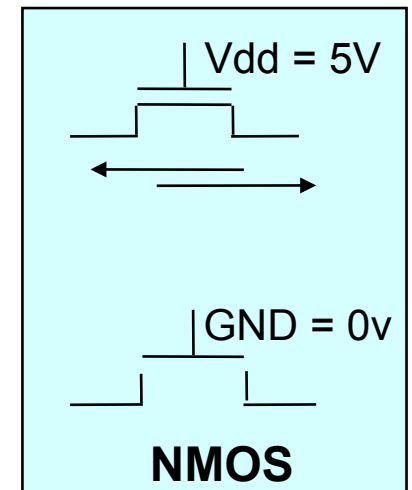


A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0

# Porte logiche e transistor

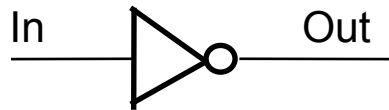
Tecnologia CMOS (Complementary Metal Oxide Semiconductor) per realizzare transistor sul silicio

- I transistor sono degli interruttori velocissimi che lasciano (o meno) passare la corrente, e sono comandati da un segnale elettrico
- NMOS (N-Type Metal Oxide Semiconductor) transistor
  - Se applichi un ALTO voltaggio ( $V_{dd}$ ), il transistor diventa un “conduttore”
  - Se applichi un BASSO voltaggio (GND), il transistor interrompe la conduzione (resistenza infinita)
- PMOS (P-Type Metal Oxide Semiconductor) transistor
  - Se applichi un ALTO voltaggio ( $V_{dd}$ ), il transistor interrompe la conduzione (resistenza infinita)
  - Se applichi un BASSO voltaggio (GND), il transistor diventa un “conduttore”

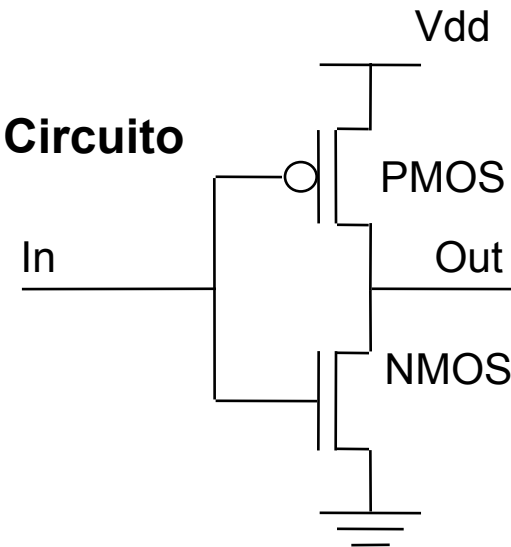


# Componenti base: Inverter CMOS

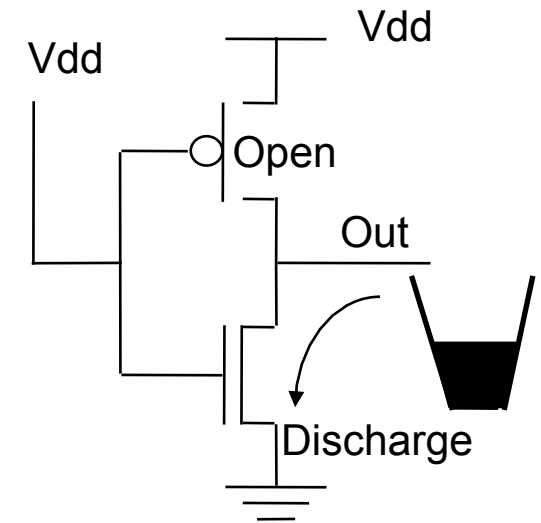
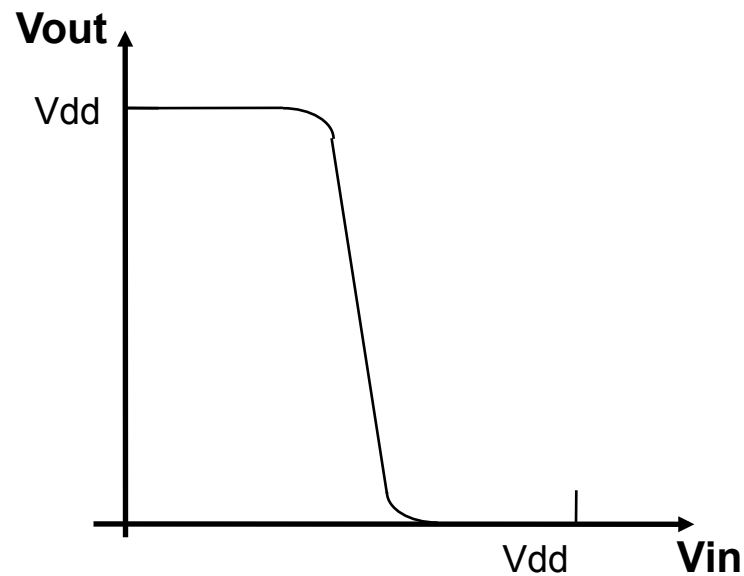
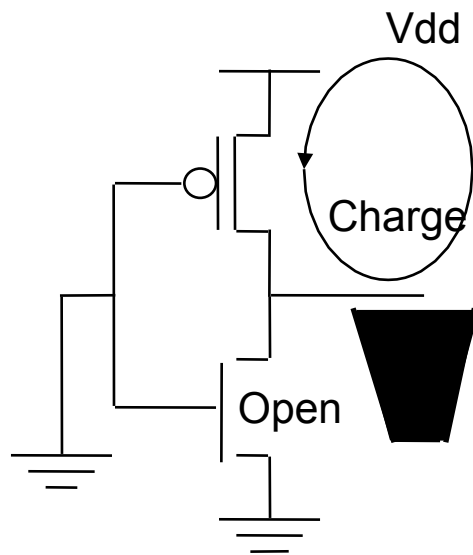
**Simbolo**



**Circuito**

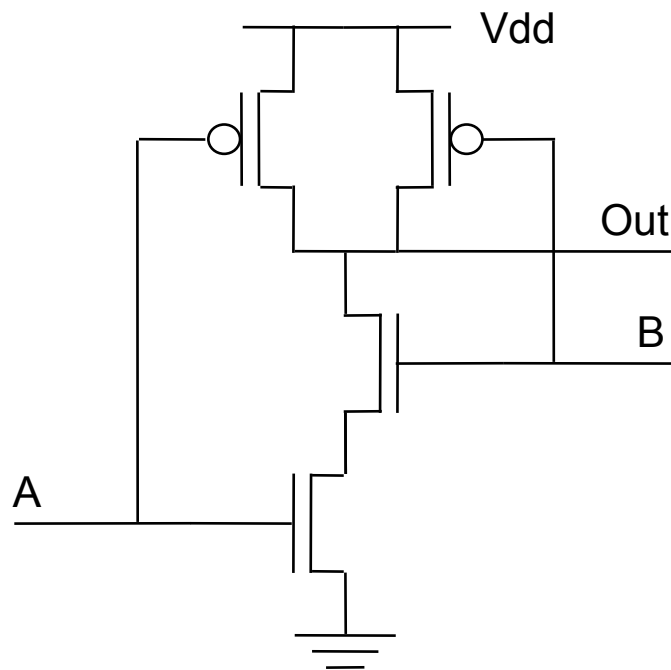
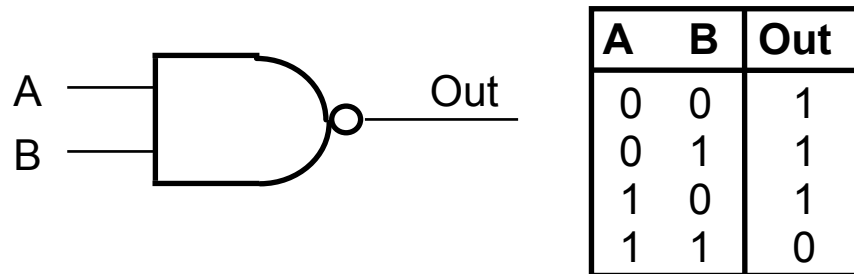


**Operazione d'inversione**

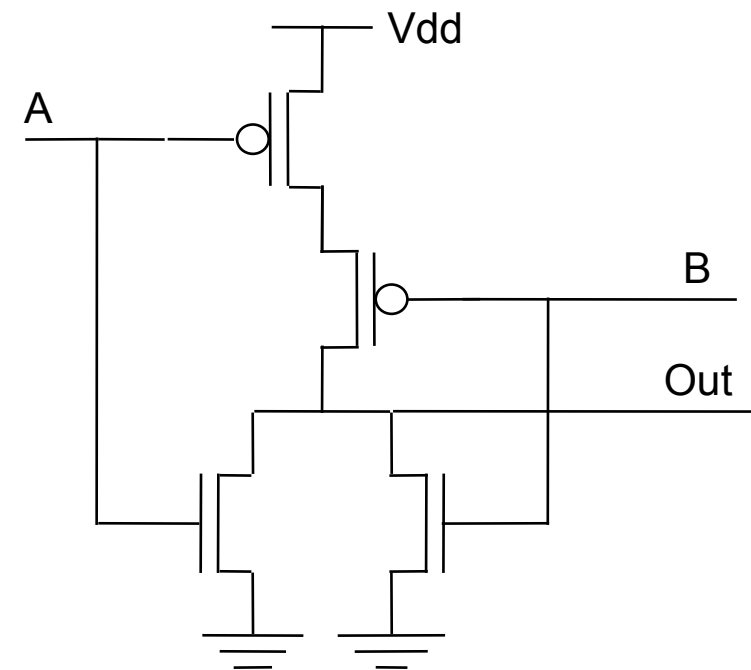
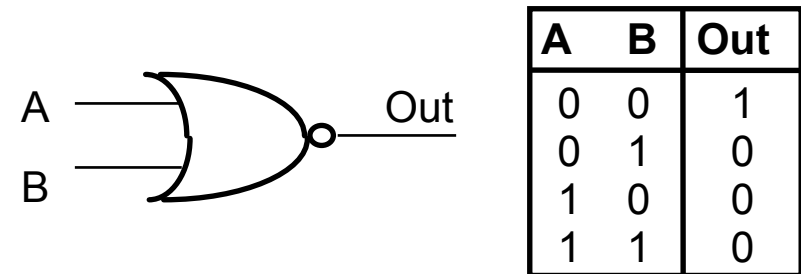


# Componenti base: Porte Logiche NOR e NAND

**Porta NAND**



**Porta NOR**



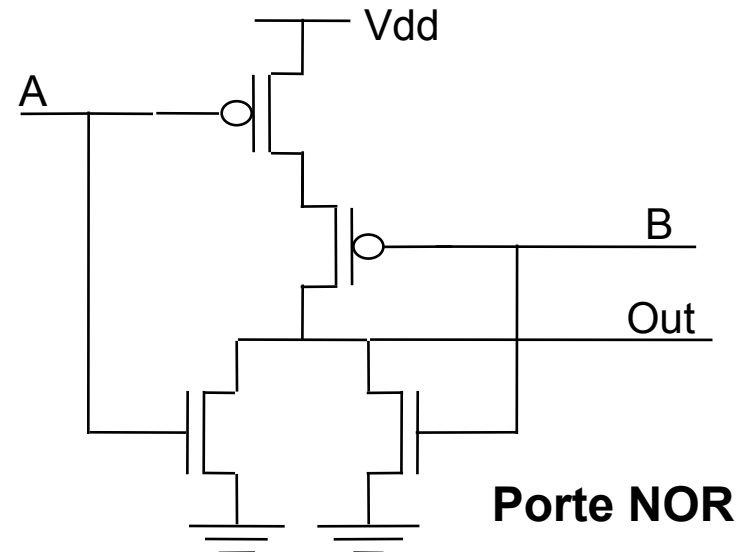
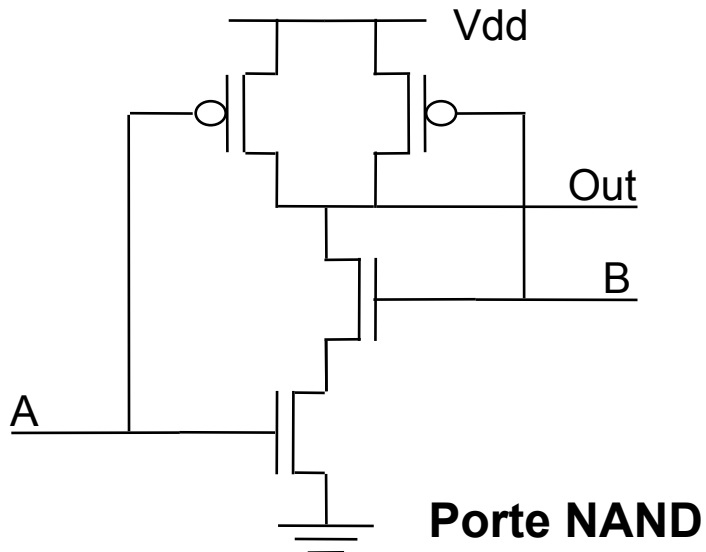
# Confronto tra Porte

Se i transistor PMOS sono più veloci:

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

Se i transistor NMOS sono più veloci:

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



# Forme canoniche

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

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

- equazione in forma canonica derivabile dalla corrispondente tabella

Forma canonica     *SP (somma di prodotti)*

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

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

# Forme canoniche

Forma canonica *PS (prodotto di somme)*

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

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

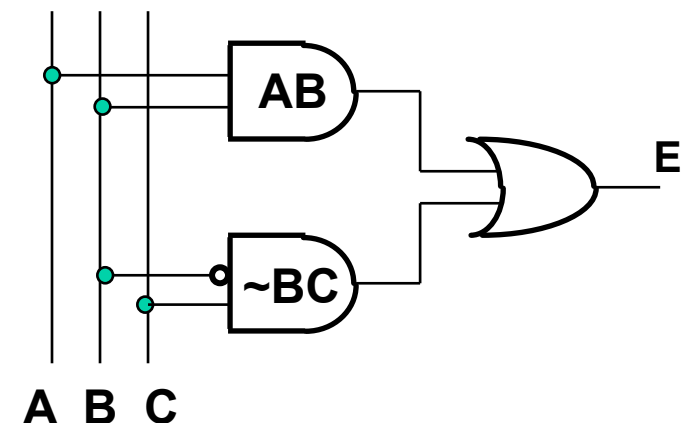
$$E = (A+B+C) \cdot (A+\sim B+C) \cdot (A+\sim B+\sim C) \cdot (\sim A+B+C) \\ (\sim A+B+\sim C) \cdot (\sim A+\sim B+\sim C)$$

# Dalle forme canoniche ai circuiti (2-level logic)

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

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

Esempio di forma SP:  $E = (AB) + (\sim BC)$



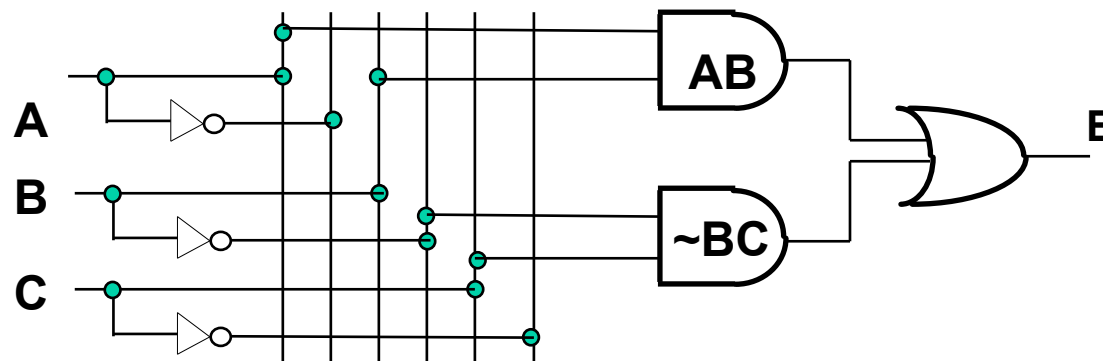


## Rappresentazione alternativa (circuito a 2 livelli)

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

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

Esempio di forma SP:  $E = (AB) + (\sim BC)$



# Minimizzazione circuiti

*Scopo della **minimizzazione***

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

# Esempio di minimizzazione

Esempio di minimizzazione usando le proprietà dell'algebra di Boole

– Funzione F che assume valori **indipendentemente dal valore di A**:

$$\begin{aligned} F &= \sim AB + AB = && \text{(distributiva)} \\ &= B (\sim A + A) = && \text{(inverso)} \\ &= B \cdot 1 = B && \text{(nullo)} \end{aligned}$$

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

# Esempio di minimizzazione

$$f = \sim A \sim B C \sim D + \sim A \sim B C D \\ + \sim A B C \sim D + \sim A B C D$$

$\sim A C$  compare in tutti i prodotti,  
combinato con tutti i possibili  
valori di  $B$  e  $D$

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

$$f = \sim A C$$

Infatti:

$$f = \sim A C (\sim B \sim D + \sim B D \\ + B \sim D + B D) = \\ = \sim A C (1) = \sim A C$$

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



# Tecniche di minimizzazione

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

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

# Tecniche di minimizzazione (...continua)

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

# Mappe di Karnaugh

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

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

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

B \ A	0	1
	0	1
0		
1		1

**2 variabili**

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

**3 variabili**

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

**4 variabili**

# Mappe di Karnaugh

*Scopo mappe:*

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



# Esempi di p-sottocubi

1-sottocubo

		AB			
		00	01	11	10
C	0		1		
	1		1		

$$f = \sim AB$$

1-sottocubo

		AB			
		00	01	11	10
C	0				
	1	1			1

$$f = C \sim B$$

2-sottocubo

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

$$f = \sim B \sim D$$

3-sottocubo

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

$$f = \sim D$$

3-sottocubo

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

$$f = B$$

2-sottocubo

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

$$f = BD$$

# Rappresentazione differente per p-sottocubi

1-sottocubo

		AB			
		00	01	11	10
C	0		1		
	1		1		

$$f = \sim AB$$

1-sottocubo

		AB			
		00	01	11	10
C	0				
	1	1			1

$$f = C \sim B$$

2-sottocubo

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

$$f = \sim B \sim D$$

3-sottocubo

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

$$f = \sim D$$

3-sottocubo

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

$$f = B$$

2-sottocubo

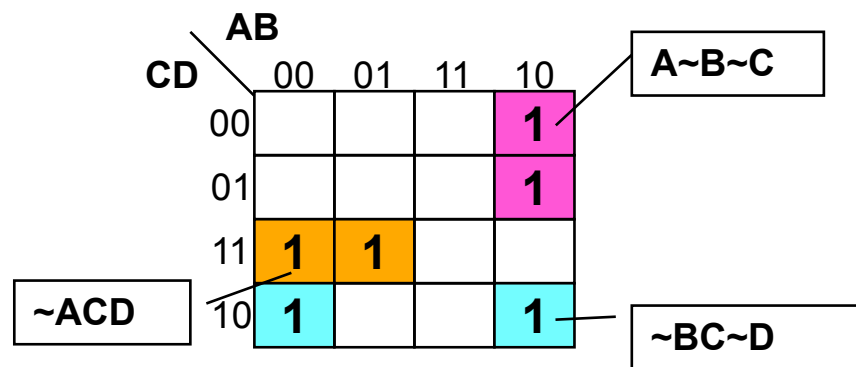
		AB			
		00	01	11	10
CD	00				
	01		1	1	
	11		1	1	
	10				

$$f = BD$$

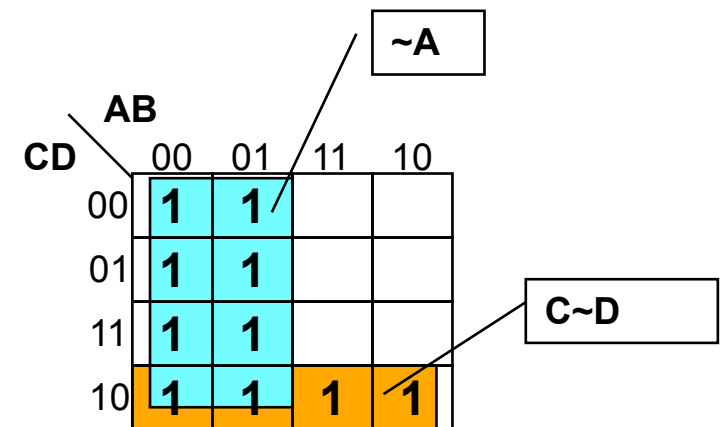
# Minimizzazione con mappe di Karnaugh

Intuitivamente

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

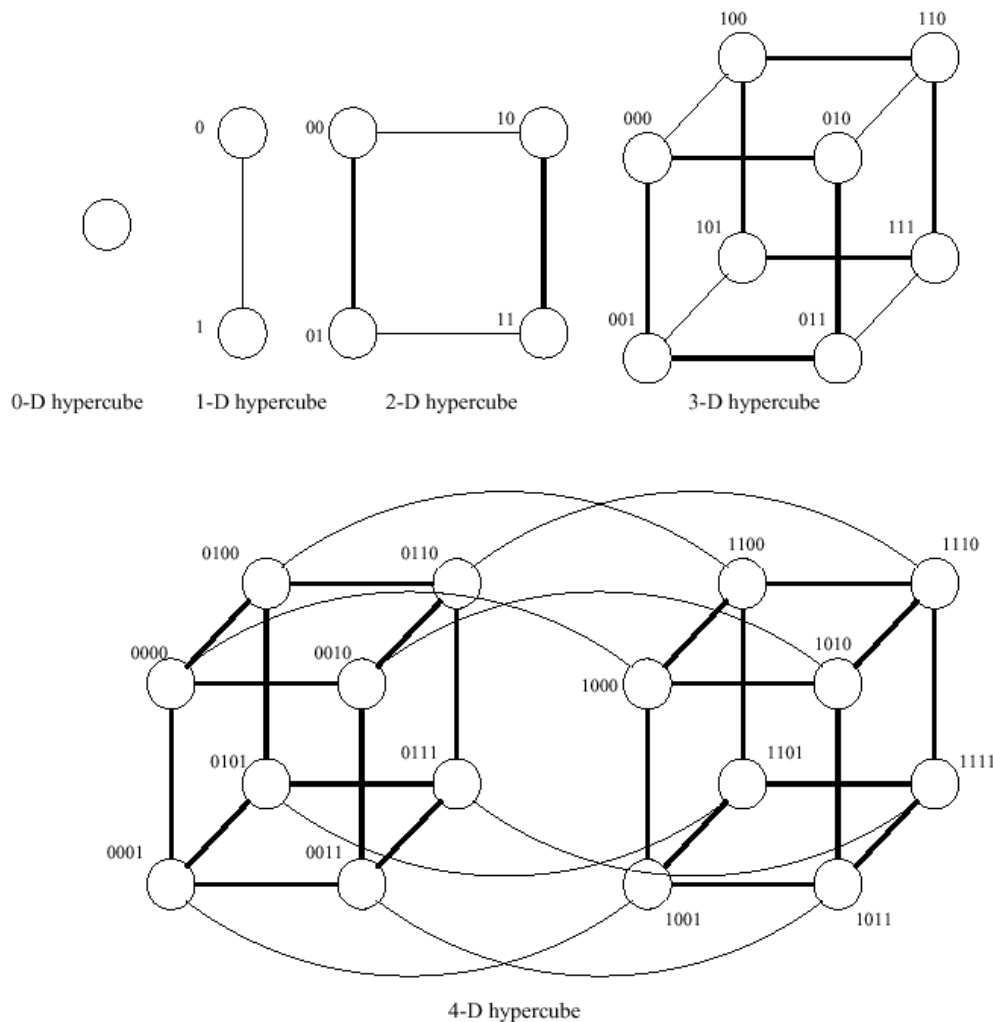


$$f = \sim ACD + A\sim B\sim C + \sim BC\sim D$$



$$f = \sim A + C\sim D$$

# Ipercubi e Mappe di Karnaugh

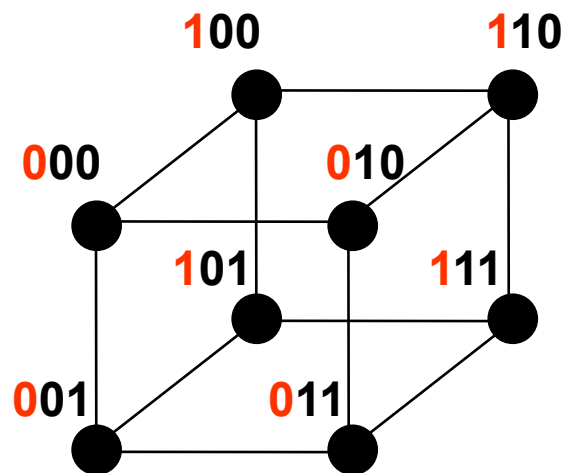


Abbiamo definito alcuni gruppi di  $2^p$  celle delle mappe come *p-sottocubi*

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

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

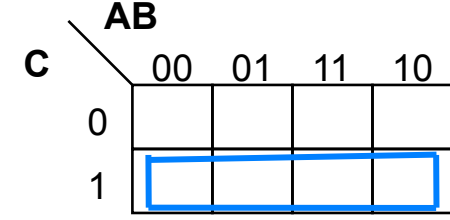
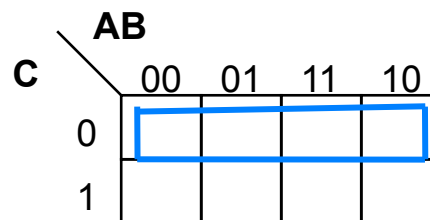
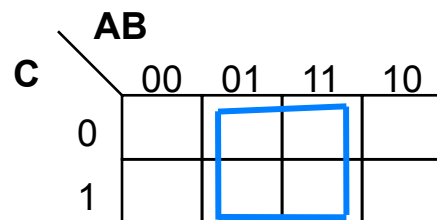
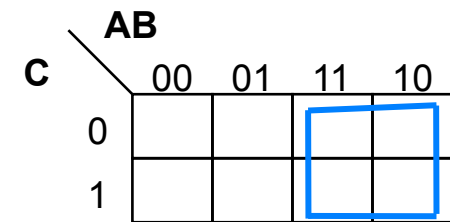
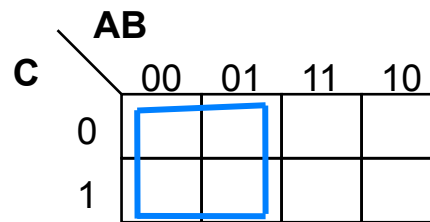
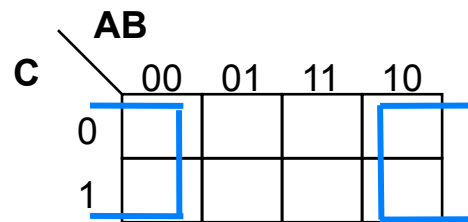
# Ipercubi e Mappe di Karnaugh



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

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

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



# Funzioni incomplete

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

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

## Problema:

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

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

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

$$f = \sim A + C \sim D$$

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

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

$$f = \sim A \sim B + \sim A \sim C + \sim A D + A C \sim D$$

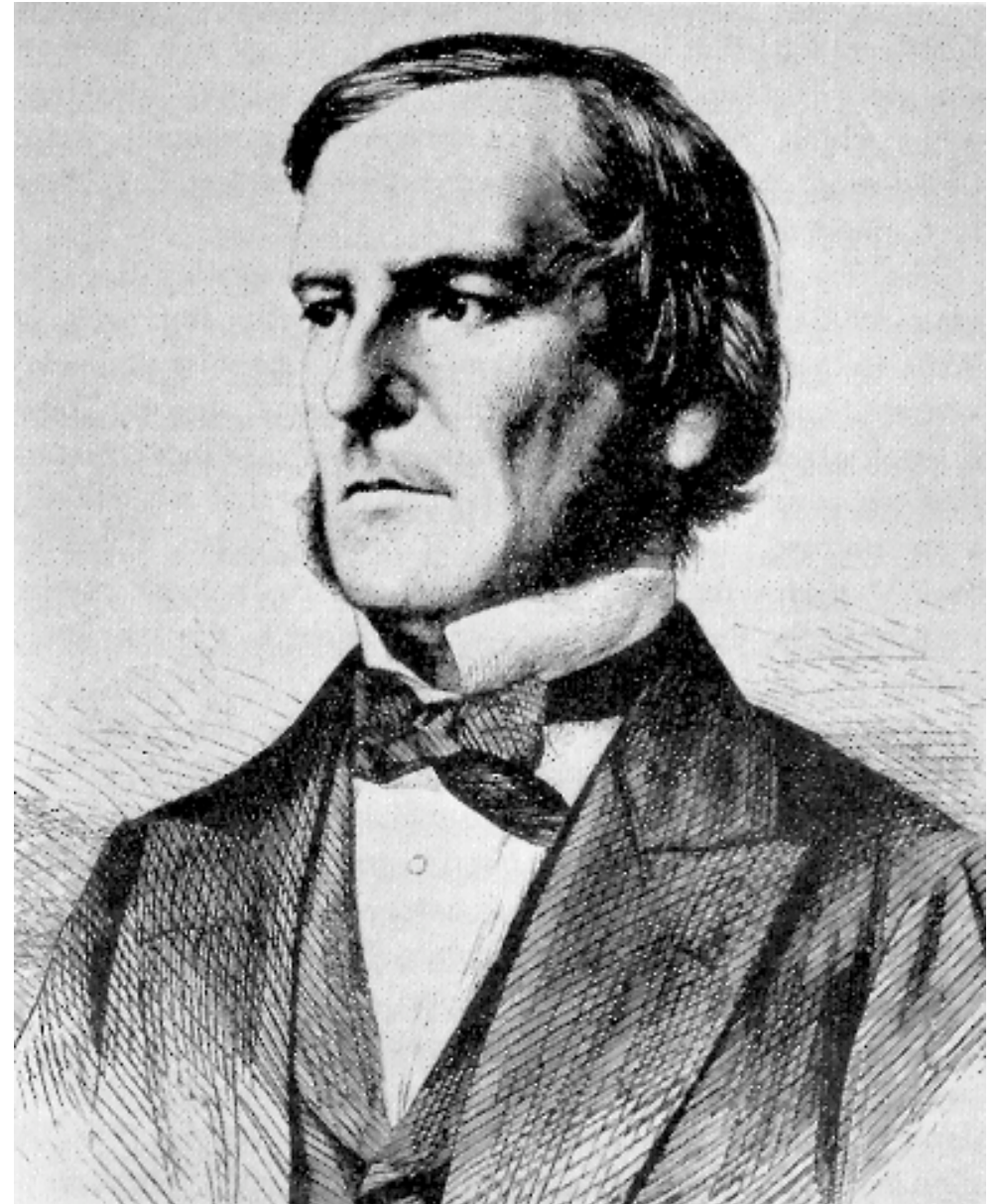
# Algoritmo di Quine McCluskey

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

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

# George Boole (1815 – 1864)

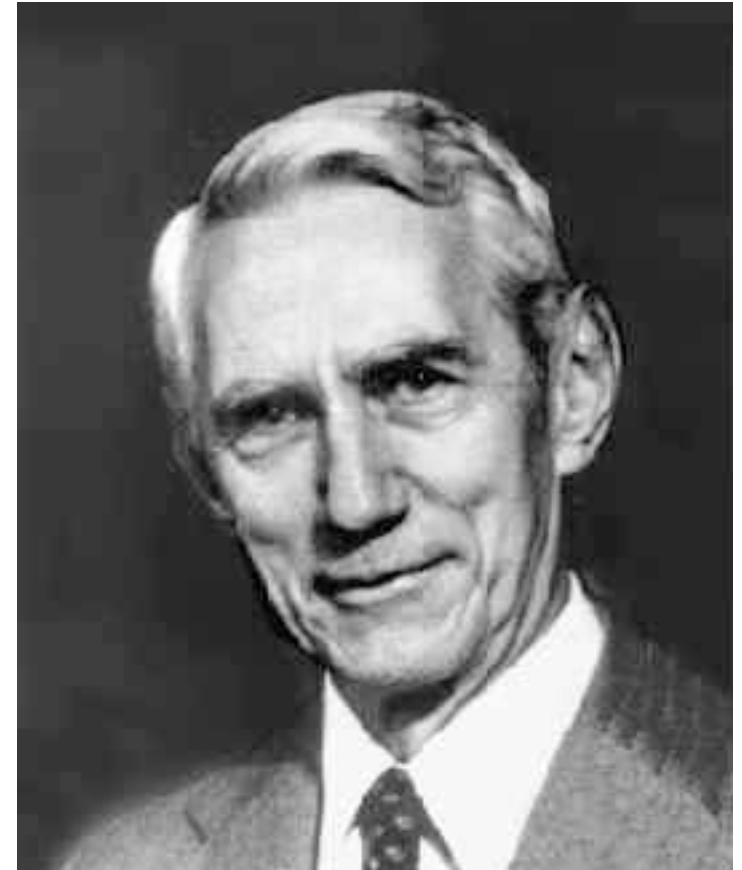
- Definisce lo strumento concettuale che sta alla base del funzionamento del calcolatore elettronico: l'algebra booleana.
- Nel suo libro del 1854 crea il legame tra logica e algebra: enunciati logici espressi mediante operazioni algebriche.
- Il suo lavoro in vita fu considerato pura matematica.
- Nel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





# Claude Shannon (1916 – 2001)

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



---

# Esercitazioni su circuiti combinatori

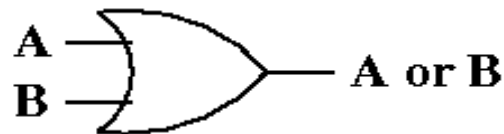
slide a cura di Salvatore Orlando e Marta Simeoni

# Algebra Booleana: funzioni logiche di base

---

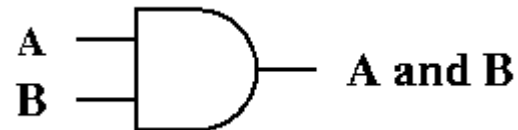
**OR** (somma): l'uscita è 1 se almeno uno degli ingressi è 1

A	B	(A + B)
0	0	0
0	1	1
1	0	1
1	1	1



**AND** (prodotto): l'uscita è 1 se tutti gli ingressi sono 1

A	B	(A · B)
0	0	0
0	1	0
1	0	0
1	1	1



# Algebra Booleana: funzioni logiche di base

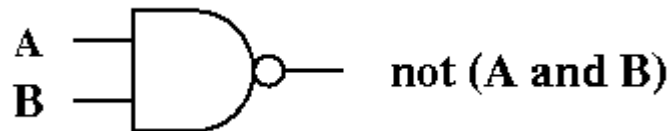
**NOT** (complemento): l'uscita è il complemento dell'ingresso

A	$\sim A$
0	1
1	0



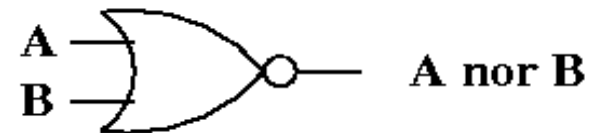
**NAND**

A	B	$\sim(A \cdot B)$
0	0	1
0	1	1
1	0	1
1	1	0



**NOR**

A	B	$\sim(A + B)$
0	0	1
0	1	0
1	0	0
1	1	0



# Algebra booleana: equazioni

---

## Come si dimostra che due funzioni logiche sono uguali?

Ci sono due metodi:

- **Costruire la tabella di verità** delle due funzioni e verificare che, per gli stessi valori dei segnali di ingresso, siano prodotti gli stessi valori dei segnali di uscita
- **Sfruttare le proprietà dell'algebra booleana** per ricavare una funzione dall'altra (tramite sequenze di equazioni)

# Algebra booleana: equazioni

---

Come si dimostra che due funzioni logiche sono uguali?

Esempio: considerare le leggi di De Morgan

$$\sim(A \cdot B) = (\sim A) + (\sim B)$$

A	B	(A·B)	$\sim(A \cdot B)$	$\sim A$	$\sim B$	$(\sim A) + (\sim B)$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

# Algebra booleana: equazioni

Come si dimostra che due funzioni logiche sono uguali?

Esempio: considerare le leggi di De Morgan

$$\sim(A+B) = (\sim A) \cdot (\sim B)$$

$$\begin{aligned}\sim A \sim B &= \sim A \sim B + 0 = \sim A \sim B + [\sim(A+B) \cdot (A+B)] = \\ &[\sim A \sim B + \sim(A+B)] \cdot [\sim A \sim B + (A+B)] = \\ &[\sim A \sim B + \sim(A+B)] \cdot [(\sim A + A) \cdot (\sim B + A) + B] = \\ &[\sim A \sim B + \sim(A+B)] \cdot [\sim B + A + B] = (\sim A \sim B) + \sim(A+B) \\ \sim A \sim B &= \sim A \sim B \cdot 1 = \sim A \sim B \cdot [\sim(A+B) + (A+B)] = \\ &(\sim A \sim B) \cdot \sim(A+B) + (\sim A \sim B) \cdot (A+B) = \\ &(\sim A \sim B) \cdot \sim(A+B) + [\sim A \sim B A + \sim A \sim B B] = (\sim A \sim B) \cdot \sim(A+B) \\ \sim A \sim B &= \sim A \sim B + \sim(A+B) = ((\sim A \sim B) \cdot \sim(A+B)) + \sim(A+B) = \\ &\sim(A+B) \cdot [(\sim A \sim B) + 1] = \sim(A+B)\end{aligned}$$

# Realizzazione di circuiti combinatori

---

**Esercizio:** Dati tre ingressi A, B, C realizzare un circuito che fornisca in uscita tre segnali

D è vera se almeno uno degli ingressi è vero

E è vera se esattamente due input sono veri

F è vera se tutti e tre gli input sono veri

Intuitivamente le equazioni sono:

$$D = A + B + C$$

$$F = ABC$$

$$E = (AB + BC + AC) \cdot \sim(ABC)$$



# Realizzazione di circuiti combinatori

**Esercizio:** Dati tre ingressi A, B, C realizzare un circuito che fornisca in uscita tre segnali

D è vera se almeno uno degli ingressi è vero

E è vera se esattamente due input sono veri

F è vera se tutti e tre gli input sono veri

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

Tabella di verità

# Realizzazione di circuiti combinatori

**Esercizio:** Dati tre ingressi A, B, C realizzare un circuito che fornisca in uscita tre segnali

D è vera se almeno uno degli ingressi è vero

E è vera se esattamente due input sono veri

F è vera se tutti e tre gli input sono veri

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

Prodotti di somme (PS):

$$D = A+B+C$$

# Realizzazione di circuiti combinatori

**Esercizio:** Dati tre ingressi A, B, C realizzare un circuito che fornisca in uscita tre segnali

D è vera se almeno uno degli ingressi è vero

E è vera se esattamente due input sono veri

F è vera se tutti e tre gli input sono veri

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

Prodotti di somme (PS):

$$D = A+B+C$$

$$E = (A+B+C) (A+B+\sim C) (A+\sim B+C) \\ (\sim A+B+C) (\sim A+\sim B+\sim C)$$

# Realizzazione di circuiti combinatori

**Esercizio:** Dati tre ingressi A, B, C realizzare un circuito che fornisca in uscita tre segnali

D è vera se almeno uno degli ingressi è vero

E è vera se esattamente due input sono veri

F è vera se tutti e tre gli input sono veri

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

Prodotti di somme (PS):

$$D = A+B+C$$

$$E = (A+B+C) (A+B+\sim C) (A+\sim B+C) \\ (\sim A+B+C) (\sim A+\sim B+\sim C)$$

$$F = (A+B+C) (A+B+\sim C) (A+\sim B+C) \\ (A+\sim B+\sim C)(\sim A+B+C) \\ (\sim A+B+\sim C)(\sim A+\sim B+C)$$

# Realizzazione di circuiti combinatori

**Esercizio:** Dati tre ingressi A, B, C realizzare un circuito che fornisca in uscita tre segnali

D è vera se almeno uno degli ingressi è vero

E è vera se esattamente due input sono veri

F è vera se tutti e tre gli input sono veri

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

Somme di Prodotti (SP):

$$D = (\sim A \sim B C) + (\sim A B \sim C) + (\sim A B C) + (A \sim B \sim C) + (A \sim B C) + (A B \sim C) + (A B C)$$

$$E = (\sim A B C) + (A \sim B C) + (A B \sim C)$$

$$F = A B C$$

# Realizzazione di circuiti combinatori

---

**Esercizio:** Minimizzare la funzione D dell'esercizio precedente

$$D = (\sim A \sim B C) + (\sim A B \sim C) + (\sim A B C) + (A \sim B \sim C) + (A \sim B C) + (A B \sim C) + (A B C)$$

B C					
A		00	01	11	10
0			1	1	1
1		1	1	1	1

# Realizzazione di circuiti combinatori

**Esercizio:** Minimizzare la funzione D dell'esercizio precedente

$$D = (\sim A \sim B C) + (\sim A B \sim C) + (\sim A B C) + \\ (A \sim B \sim C) + (A \sim B C) + (A B \sim C) + (A B C)$$

B C					
A		00	01	11	10
0			1	1	1
1		1	1	1	1

Si può considerare un rettangolo più grande di quello a sinistra, che include anche quello selezionato

# Realizzazione di circuiti combinatori

**Esercizio:** Minimizzare la funzione D dell'esercizio precedente

$$D = (\sim A \sim B C) + (\sim A B \sim C) + (\sim A B C) + (A \sim B \sim C) + (A \sim B C) + (A B \sim C) + (A B C)$$

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



# Realizzazione di circuiti combinatori

**Esercizio:** Minimizzare la funzione D dell'esercizio precedente

$$D = (\sim A \sim B C) + (\sim A B \sim C) + (\sim A B C) + \\ (A \sim B \sim C) + (A \sim B C) + (A B \sim C) + (A B C)$$

B C					
A		00	01	11	10
0			1	1	1
1		1	1	1	1

**Errore!**

si deve raccogliere un  
p-sottocubo (rettangolo  
di celle adiacenti)  
di  $2^p$  celle

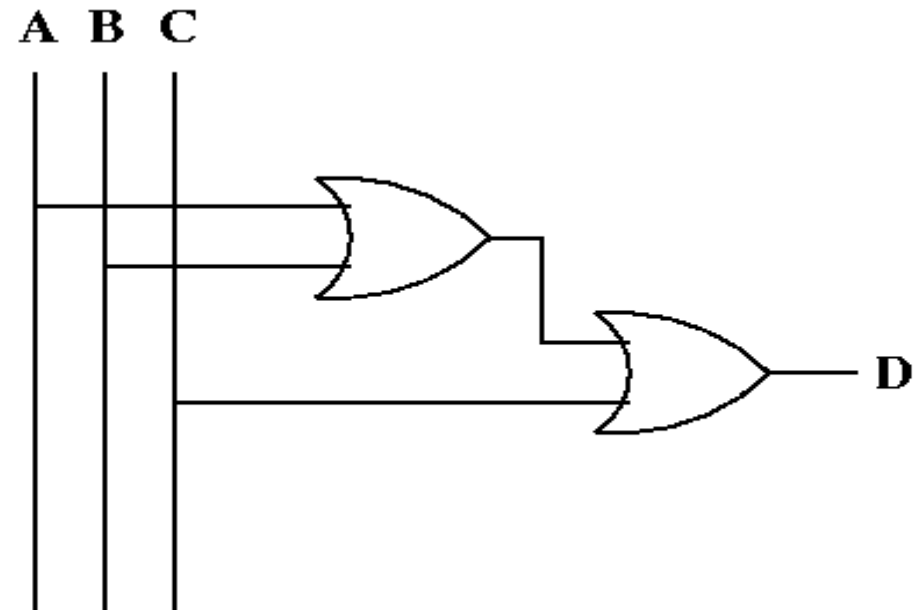
# Realizzazione di circuiti combinatori

**Esercizio:** Minimizzare la funzione D dell'esercizio precedente

$$D = (\sim A \sim B C) + (\sim A B \sim C) + (\sim A B C) + (A \sim B \sim C) + (A \sim B C) + (A B \sim C) + (A B C)$$

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

$$D = A + B + C$$



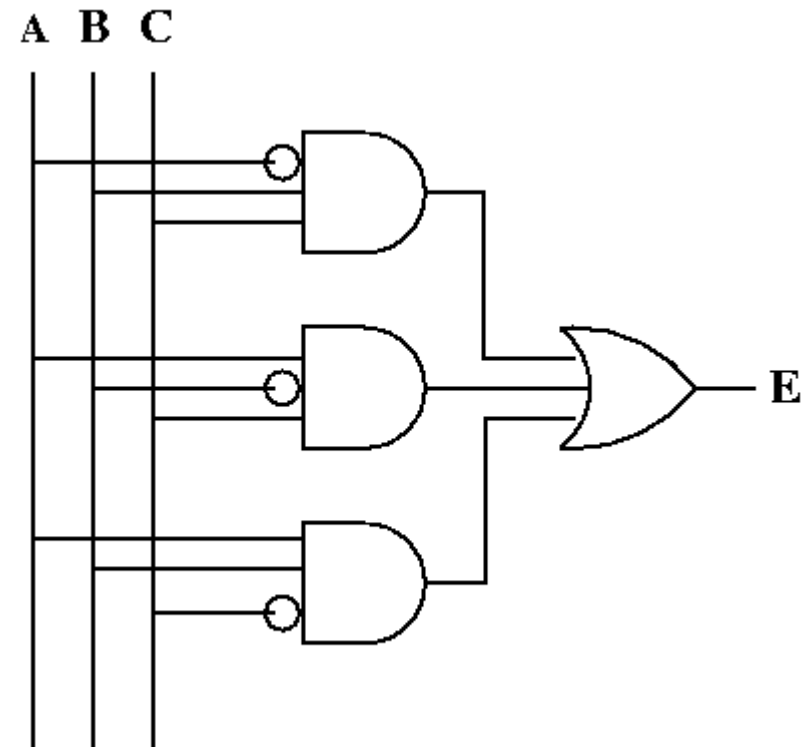
# Realizzazione di circuiti combinatori

**Esercizio:** Minimizzare la funzione E dell'esercizio precedente

$$E = (\sim ABC) + (A \sim BC) + (AB \sim C)$$

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

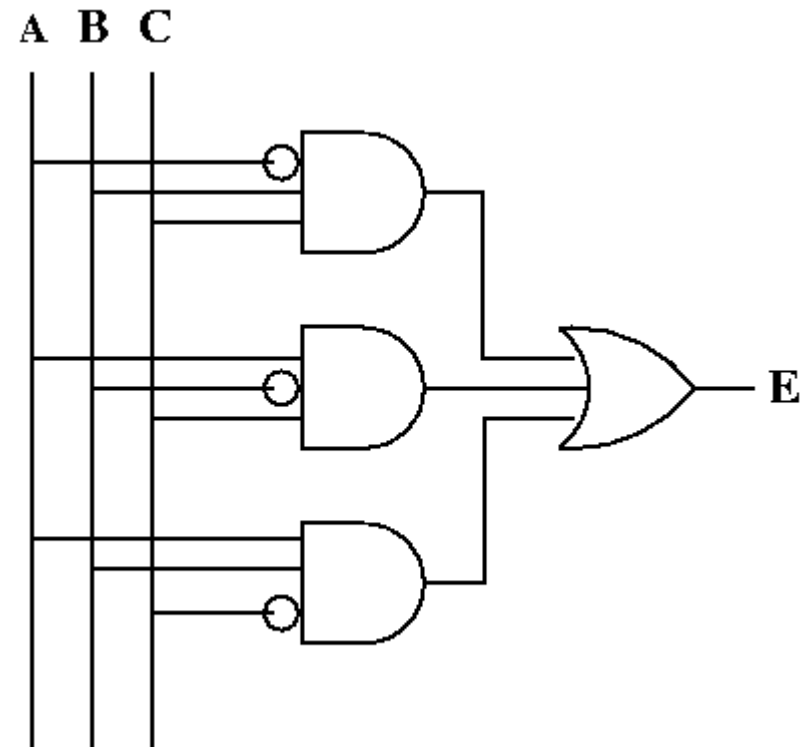
$$E = (\sim ABC) + (A \sim BC) + (AB \sim C)$$



# Realizzazione di circuiti combinatori

**Esercizio:** Realizzare il circuito precedente (riportato qui in figura) nei seguenti casi:

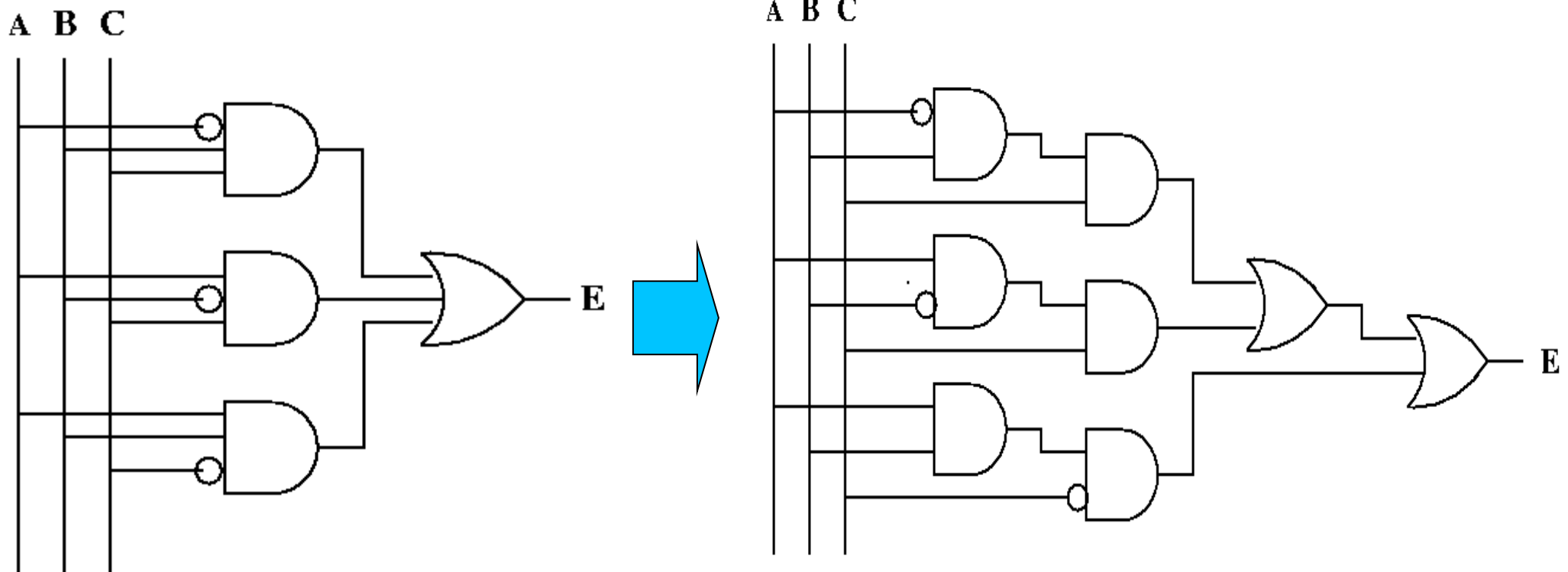
1. utilizzando porte AND e OR a due ingressi
2. utilizzando porte NAND a tre ingressi (ed eventualmente invertitori)



# Realizzazione di circuiti combinatori

Esercizio: (continua)

Realizzazione utilizzando porte AND e OR a due ingressi



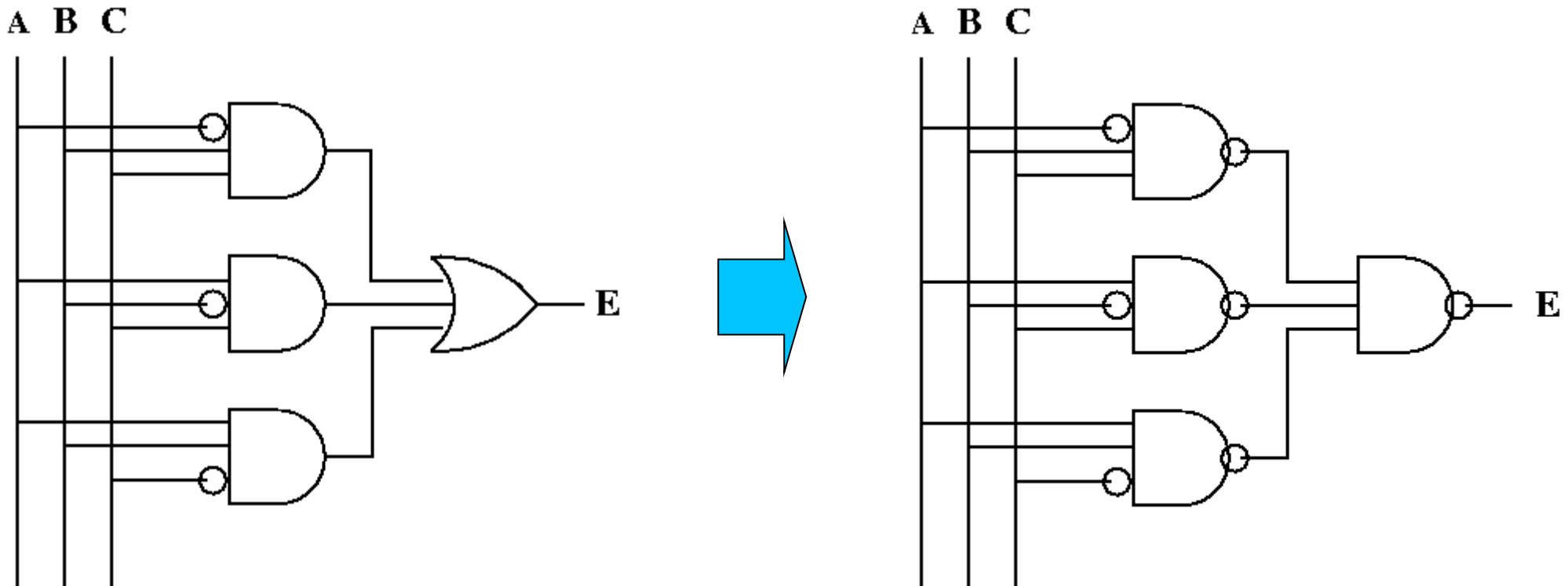
# Realizzazione di circuiti combinatori

Esercizio: (continua)

Realizzazione utilizzando porte NAND a tre ingressi e NOT

$$E = (\sim ABC) + (A \sim BC) + (AB \sim C) = [\text{applico De Morgan}]$$
$$\sim [ \sim(\sim ABC) \cdot \sim(A \sim BC) \cdot \sim(AB \sim C) ]$$

NB: Dimostrare l'equivalenza con l'equazione "intuitiva"



# Realizzazione di circuiti combinatori

**Esercizio:** Minimizzare la funzione F dell'esercizio precedente espressa come prodotto di somme (PS)

$$F = (A+B+C) (A+B+\sim C) (A+\sim B+C) (A+\sim B+\sim C)(\sim A+B+C) (\sim A+B+\sim C)(\sim A+\sim B+C)$$

B C					
A		00	01	11	10
0	0	0	0	0	0
1	0	0	0		0

$$F = B \cdot A \cdot C$$

**p-sottocubi composti da zeri.** Per ottenere le varie somme (PS), in ogni somma devono apparire solo le variabili che rimangono invariate in ogni p-sottocubo. Le variabili appaiono negate quando hanno valori uguali ad 1.

# Realizzazione di circuiti combinatori

**Esercizio:** Dati quattro ingressi A, B, C, D realizzare un circuito che fornisca in uscita il segnale E definito come segue:

- il valore di E è indifferente se gli ingressi sono tutti 0 o tutti 1
- E è 1 se gli ingressi contengono un numero dispari di 1
- E è 0 se gli ingressi contengono un numero pari di 1

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

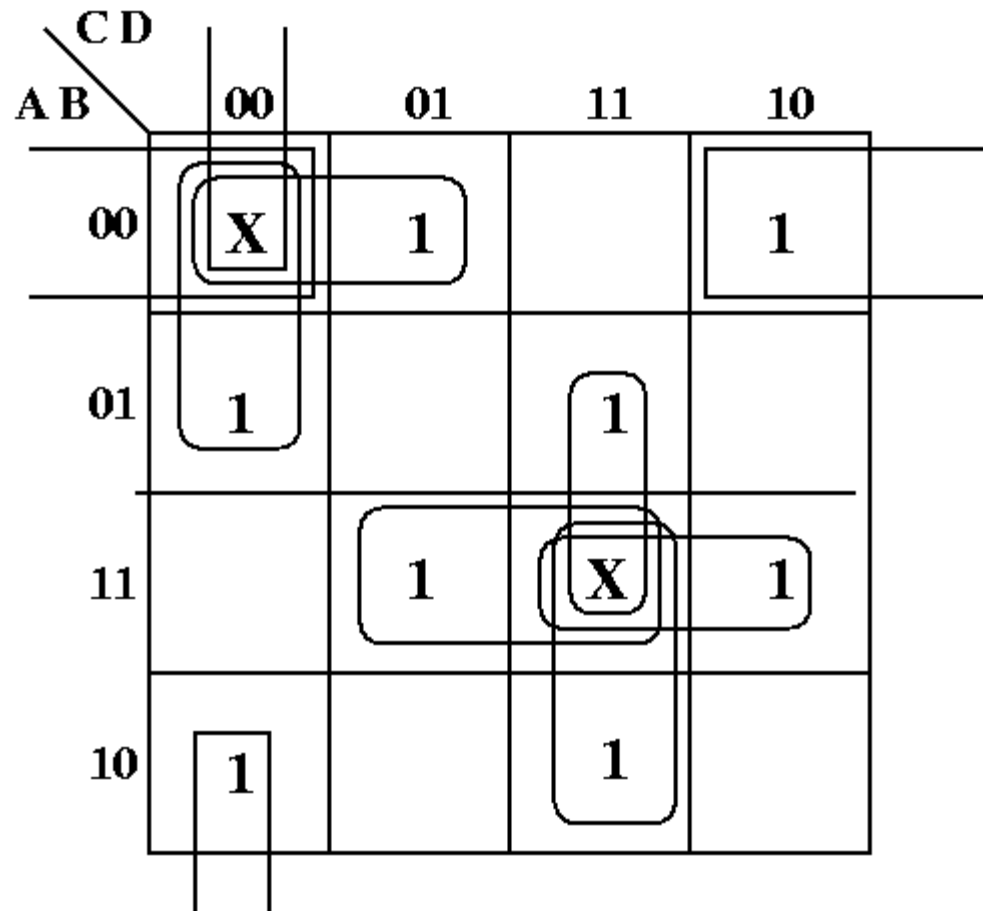
Tabella di verità



# Realizzazione di circuiti combinatori

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

Tabella di verità



Mappa di Karnaugh

# Realizzazione di circuiti combinatori

C D A B					
		00	01	11	10
00	<div>X</div>	1			1
01	1			1	
11		1	<div>X</div>	1	
10	1			1	

Realizzare il circuito usando porte AND e OR a due soli ingressi

$$E = \sim A \sim B \sim C + \sim A \sim C \sim D + \sim B \sim C \sim D + \sim A \sim B \sim D + BCD + ABC + ABD + BCD$$

# Architettura degli Elaboratori

## Circuiti combinatori

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

# Circuiti integrati

I circuiti logici sono realizzati come IC (circuiti integrati)

- realizzati su chip di silicio (piastrina)
- Porte (gate) e fili depositati su chip di silicio, inseriti in un package e collegati all'esterno con un certo insieme di pin (piedini)
- gli IC si distinguono per grado di integrazione
  - da singole porte indipendenti, a circuiti più complessi

# Circuiti integrati

## Integrazione

- SSI (Small Scale Integrated): 1-10 porte
- MSI (Medium Scale Integrated): 10-100 porte
- LSI (Large Scale Integrated): 100-100.000 porte
- VLSI (Very Large Scale Integrated): > 100.000 porte

Con tecnologia SSI, gli IC contenevano poche porte, direttamente collegate ai pin esterni

Con tecnologia MSI, gli IC contenevano alcuni componenti base

- circuiti comunemente usati nel progetto di un computer

Con tecnologia **VLSI**, un IC può oggi contenere una CPU completa (o più)

- microprocessore

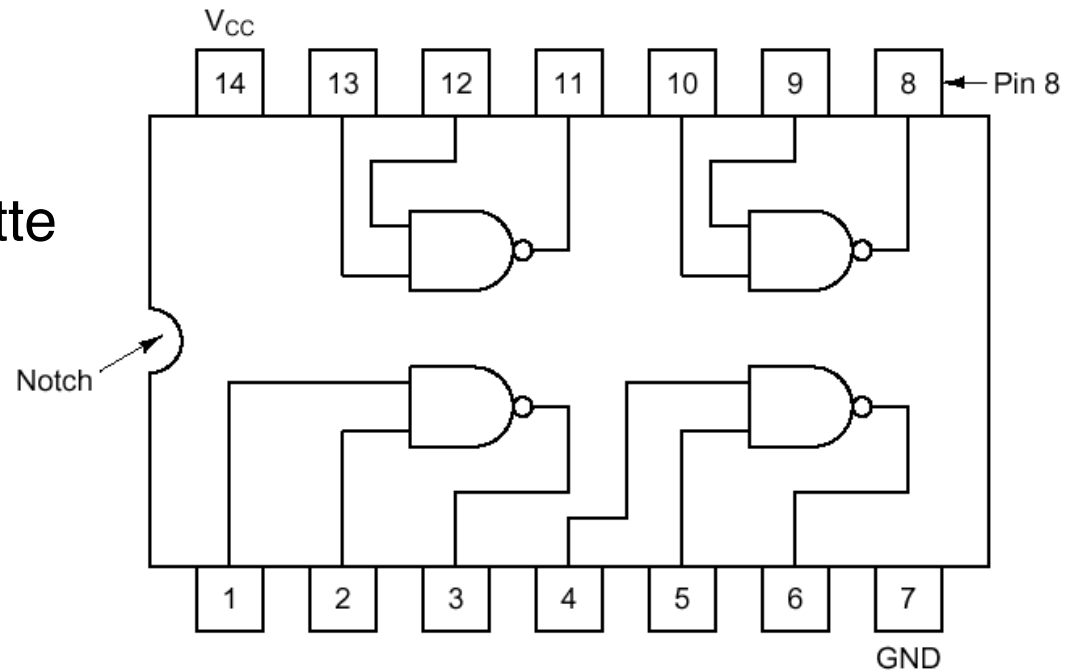
# Esempio di chip SSI

*Texas Instruments 7400*

- DIP (Dual Inline Package)
- Tensione e terra condivisi da tutte le porte
- Tacca per individuare l'orientamento del chip

SSI: Rapporto **pin/gate** (piedini/porte) grande

- Con l'aumento del grado di integrazione il rapporto **pin/gate** diminuisce (molti gate rispetto ai pin)
- **chip più specializzati**
- **es. chip che implementano particolari circuiti combinatori**



# Circuiti combinatori

Circuiti combinatori usati quindi come blocchi base per costruire circuiti più complessi

- spesso realizzati direttamente come componenti MSI

Tratteremo:

- Multiplexer e Demultiplexer
- Decoder
- ALU

Per implementare circuiti combinatori useremo

- PLA
- ROM

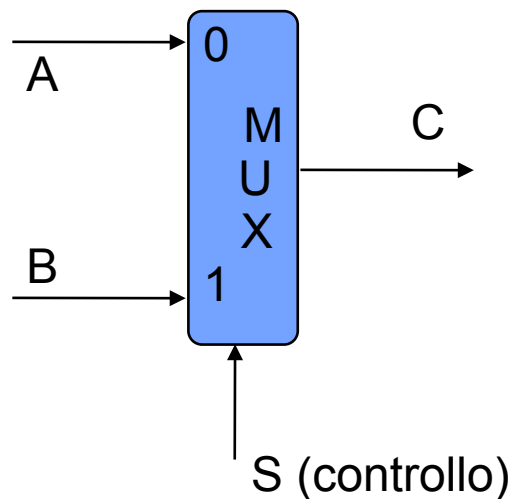
# Multiplexer (1)

$n$  input ed  $1$  output

$\log_2 n$  segnali di *controllo* (da considerare come ulteriori *input* del circuito)

Il multiplexer, sulla base dei *segnali di controllo*, seleziona quale tra gli  $n$  input verrà presentato come output del circuito

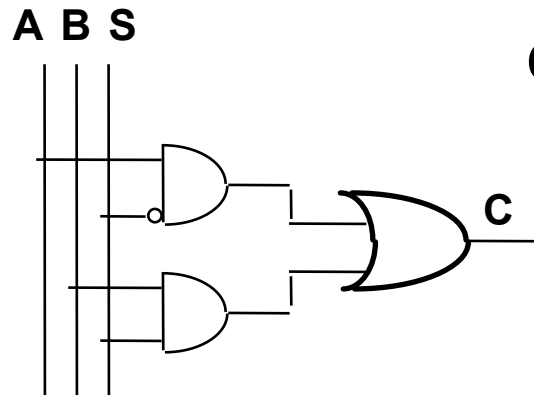
Caso semplice: **Multiplexer 2:1**, con un solo bit di controllo



**se  $S=0$ : passa A**  
**se  $S=1$ : passa B**

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

$$C = A \sim S + BS$$





# Multiplexer (2)

Multiplexer 8:1

$8=2^3$  input e 3 segnali di controllo

$8=2^3$  porte AND e 1 porta OR

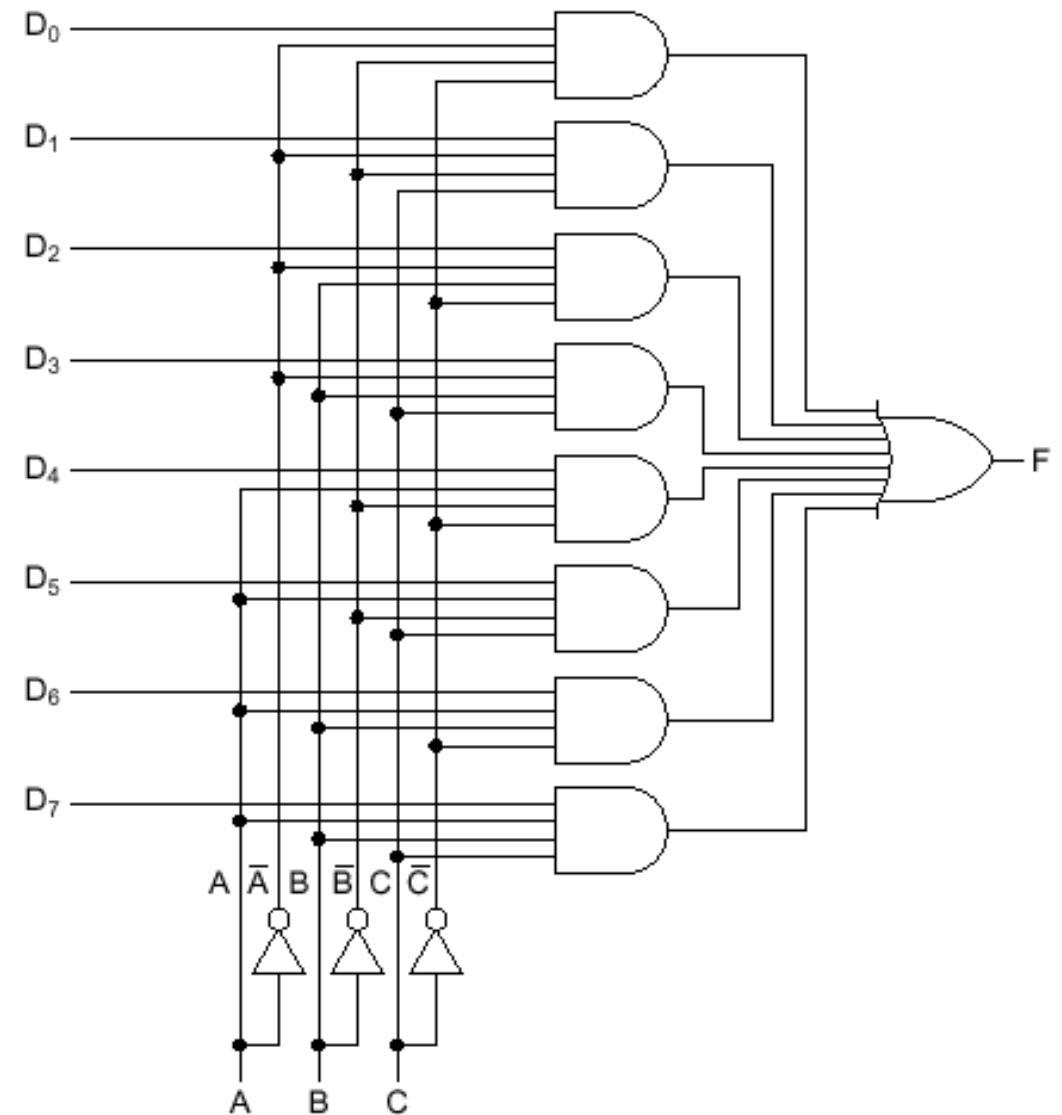
In ogni porta AND entra una combinazione diversa dei segnali di controllo A, B, C

- $8=2^3$  combinazioni possibili

in dipendenza dei valori assunti da A, B e C, tutte le porte AND (eccetto una) avranno sicuramente output uguale a 0

solo una delle porte produrrà eventualmente un valore 1

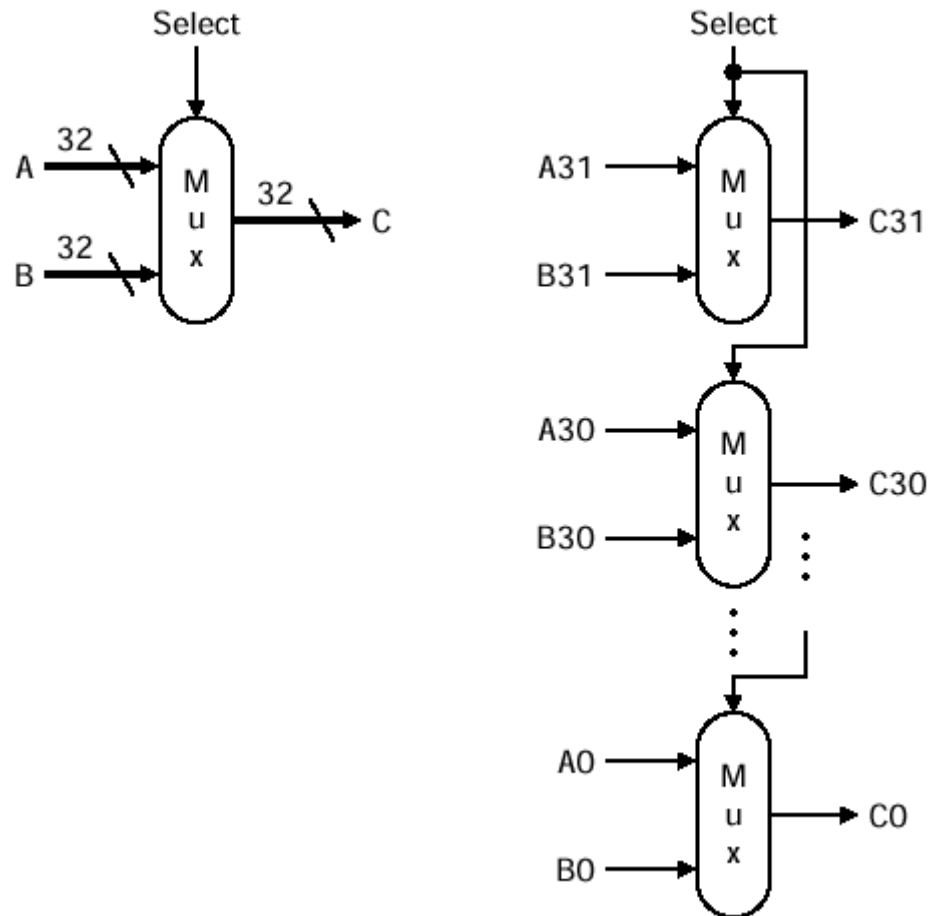
- es.  $A \sim B C = 1 \Rightarrow$  passa  $D_5$



# Multiplexer

Multiplexer 2:1 a 32-bit (con *fili larghi* di 32 bit)

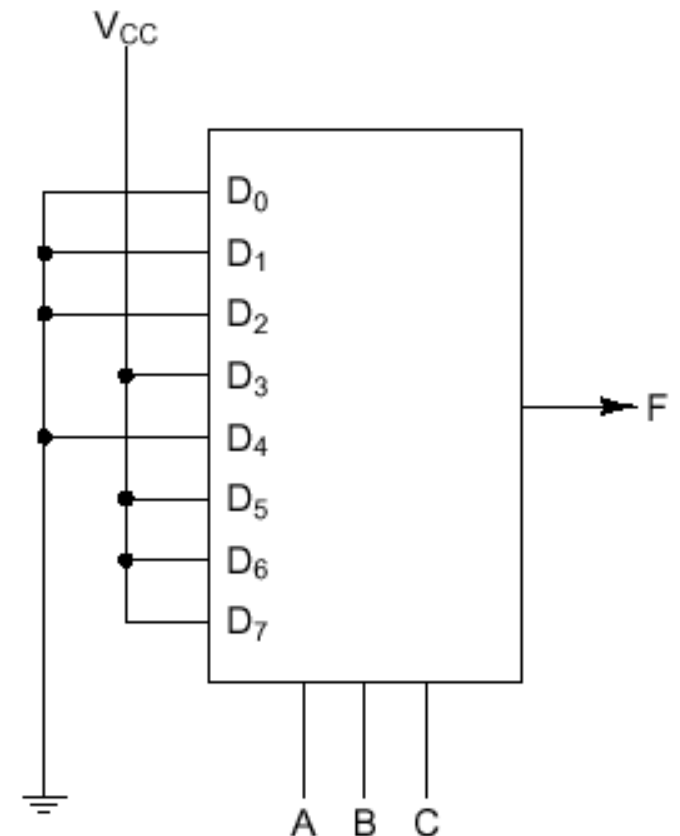
- costruito usando 32 **1-bit Multiplexer 2:1** con un segnale di controllo distribuito ai vari Multiplexer



# Multiplexer e funzioni logiche

## Multiplexer $n:1$

- possono essere usati per definire una qualsiasi funzione logica in  $\log_2 n$  variabili
  - funzione definita da una **tabella di verità** con  $n$  righe
  - le  $\log_2 n$  variabili in input della funzione logica *diventano* i segnali di controllo del multiplexer
- ogni riga della tabella di verità
  - corrisponde ad uno degli  $n$  **input del multiplexer**, collegati ad un generatore di tensione (o alla terra)
    - se l'output, associato alla riga della tabella di verità, è 1 (o 0)
- grande spreco di porte
  - circuito *fully encoded*
  - porte AND con arietà maggiore del necessario (+1)



Componente MSI che realizza un multiplexer 8:1

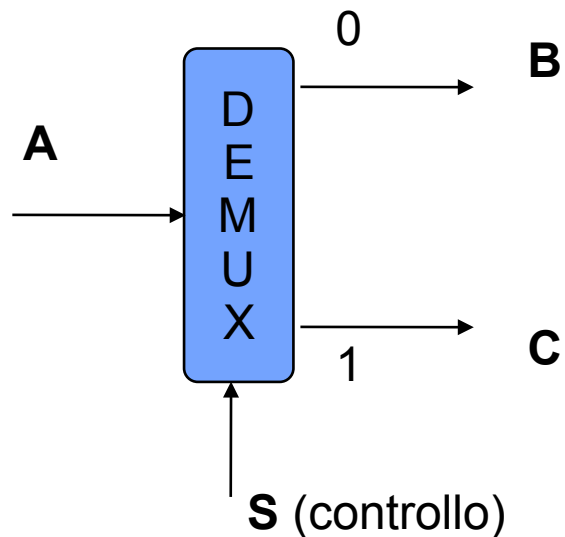
Collegamento per ottenere la funzione:

$$F = \sim ABC + A \sim BC + AB \sim C + ABC$$

# Demultiplexer

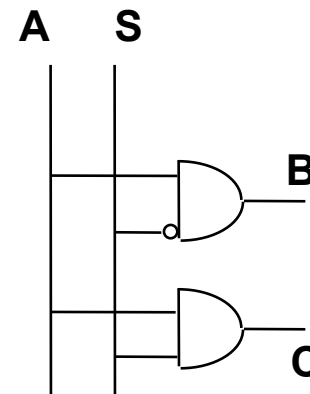
Da 1 singola linea in **input**, a **n** linee in **output**

- $\log_2 n$  segnali di controllo (S)
- se la linea in input è uguale a 0
  - tutti gli output dovranno essere uguali a 0, indipendentemente da S
- se la linea in input è uguale a 1
  - un solo output dovrà essere uguale a 1, tutti gli altri saranno 0
  - l'output da affermare dipende da S



$$B = A \sim S$$

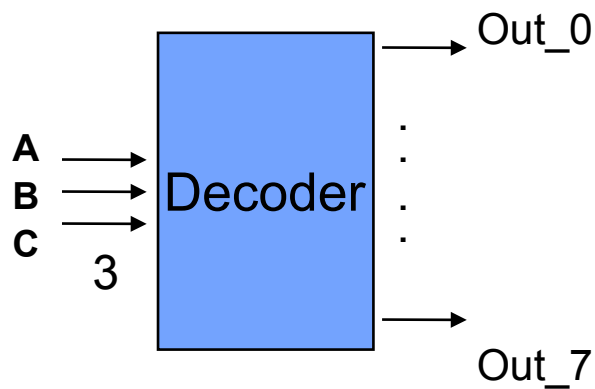
$$C = AS$$



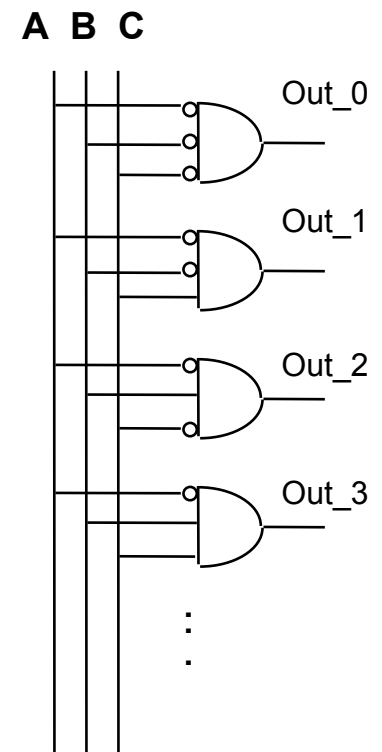
# Decoder

Componente con  $n$  input e  $2^n$  output

- gli  $n$  input sono interpretati come un numero unsigned
- se questo numero rappresenta il numero  $i$ , allora
  - solo il bit in output di indice  $i$  ( $i=0,1,\dots,2^n-1$ ) verrà posto ad 1
  - tutti gli altri verranno posti a 0



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

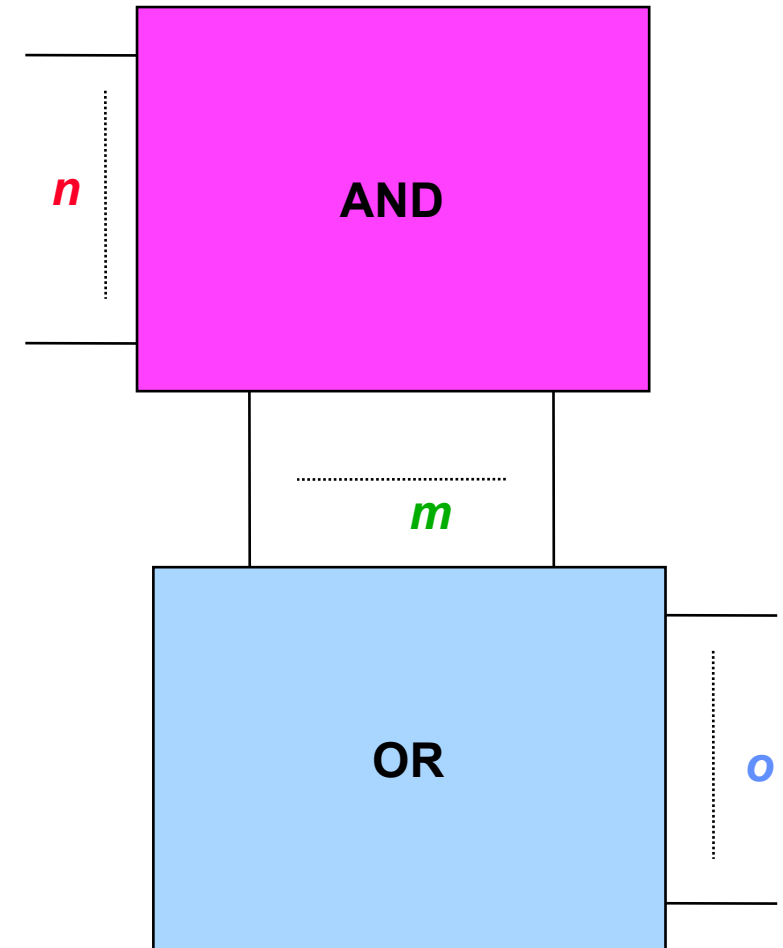


# PLA

## Programming Logic Array (PLA)

Componente per costruire funzioni logiche arbitrarie

- permette di costruire funzioni in forma SP
  - porte AND al primo livello, e porte OR al secondo livello
- $n$  input e  $o$  output
  - $m$  porte AND
  - $o$  porte OR
- $m$  fissa un limite al numero di **mintermini** esprimibili
- $o$  fissa un limite al numero di funzioni differenti in forma canonica SP



# PLA

Esempio:

- $n=12$
- $o=6$
- $m=50$

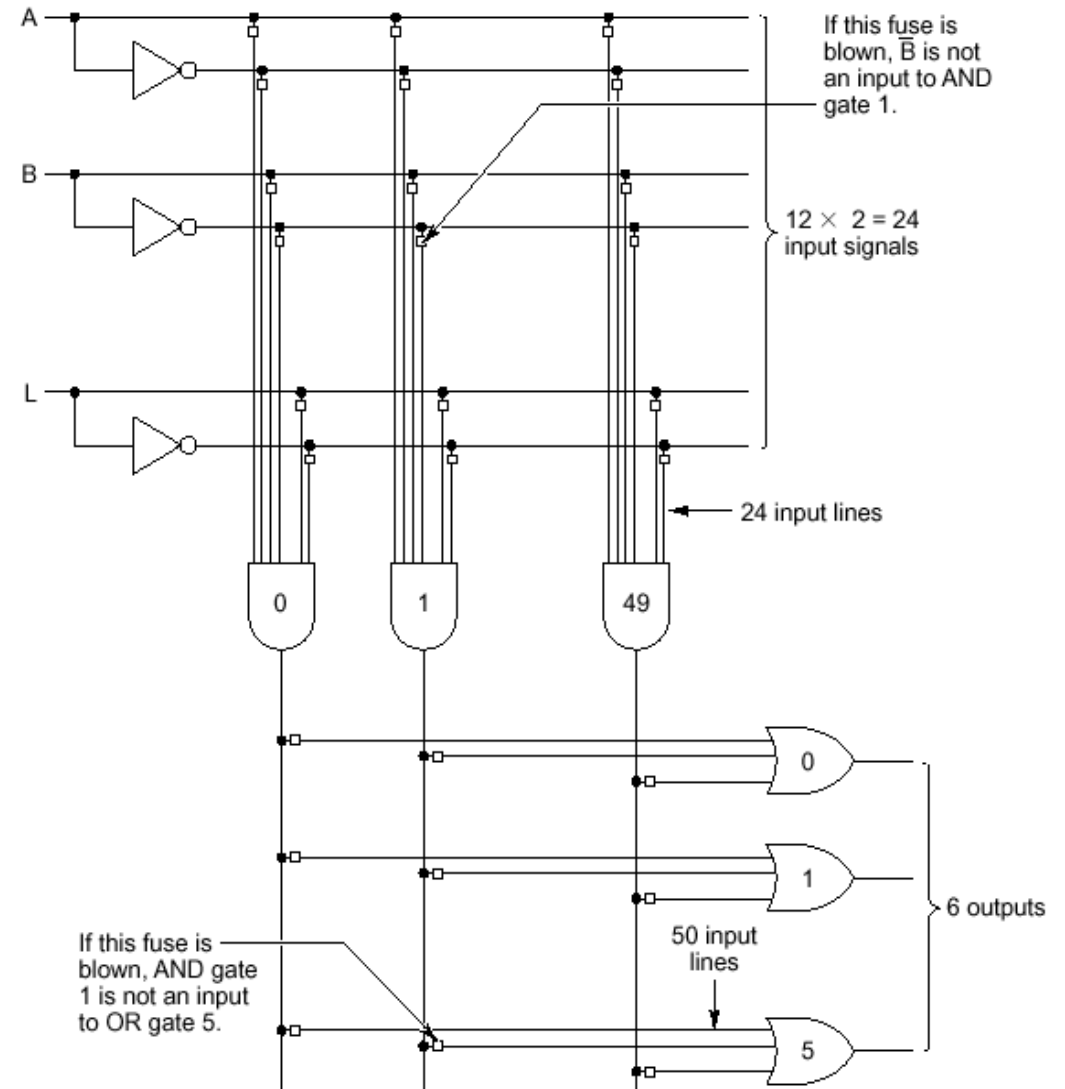
In ogni porta AND entrano  $2n=24$  input

- normali e invertiti

In ogni porta OR entrano  $m=50$  input

**Fusibili** da bruciare per decidere

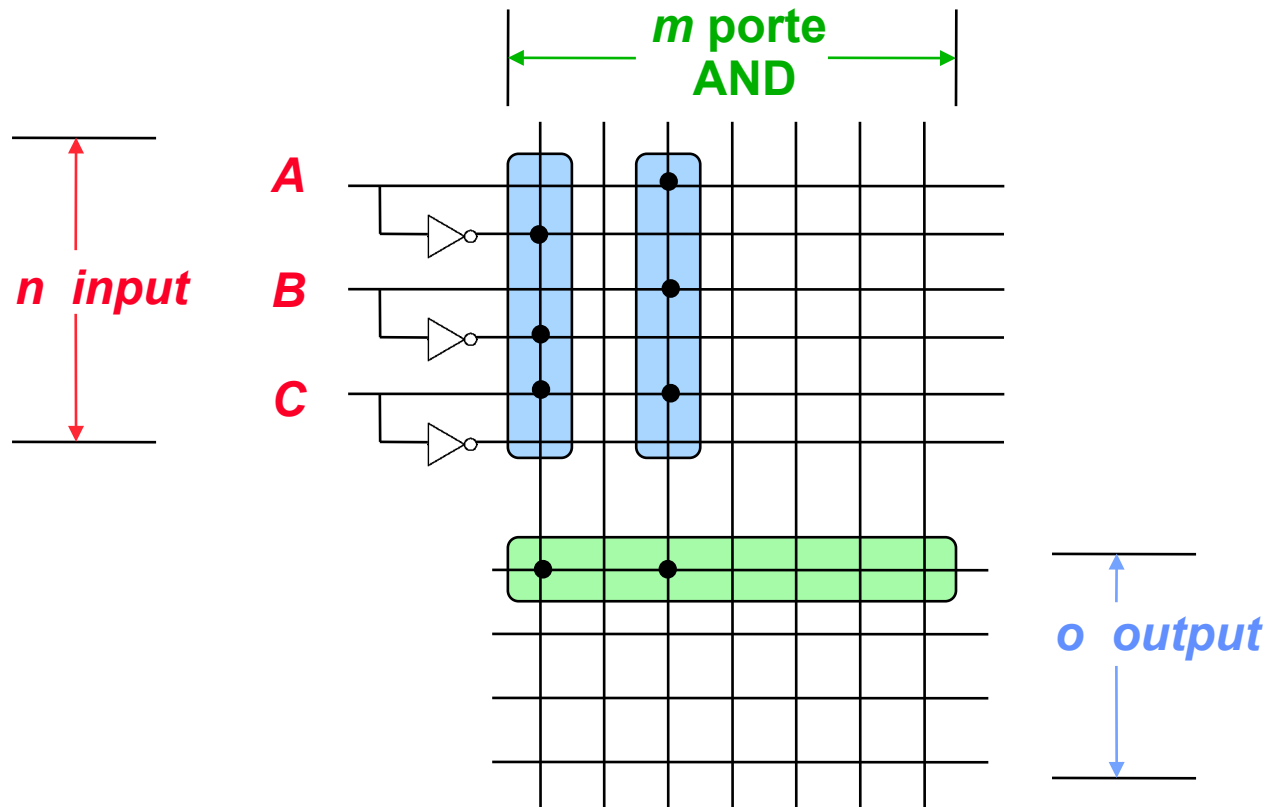
- quali sono gli input di ogni porta AND
- quali sono gli input delle varie porte OR



# PLA

Esempio di funzione:

$$O = \sim A \sim B C + A B C$$

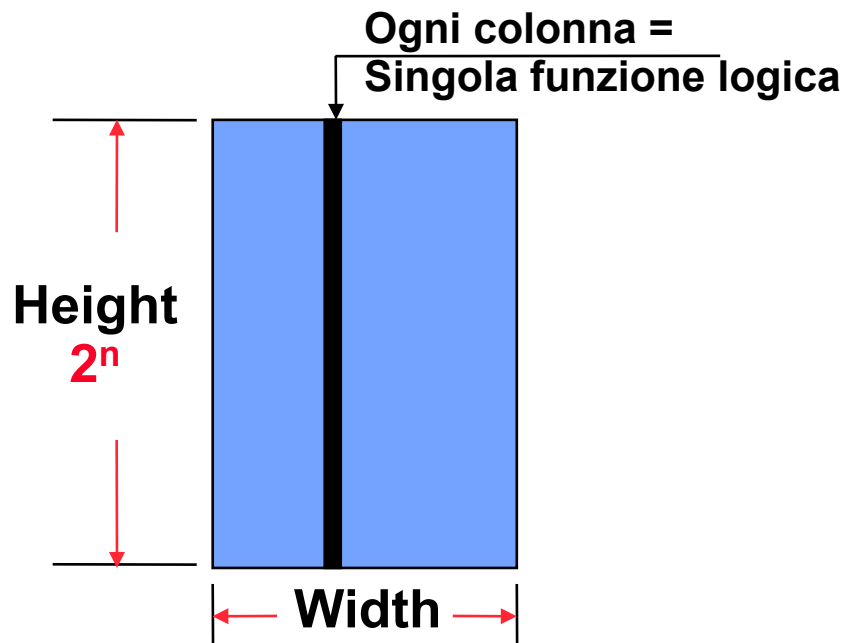




# ROM

Memorie usabili anche per implementare, in maniera non minima, funzioni logiche arbitrarie

- ROM (Read Only Memory)
  - PROM (Programmable ROM)
    - scrivibili solo una volta
  - EPROM (Erasable PROM)
    - cancellabile con luce ultravioletta



- In pratica
  - data una **Tabella di Verità**, le ROM sono usate per memorizzare direttamente le diverse funzioni logiche (corrispondenti a colonne distinte nella Tabella)
  - Indirizzo a  $n$  bit
    - individua una specifica combinazione delle  $n$  variabili logiche in input
    - individua una cella di **Width** bit della ROM
- Ogni funzione:
  - Singola colonna della ROM
  - Funzioni **fully encoded**
  - **PLA** più efficiente