<https://nbasilicoae1.ariel.ctu.unimi.it>

Libri:

Strutture e progetto dei calcolatori

Computer organization and design: the hardware/software interface

“Progettazione digitale”

# Lezione 1 - 02/10/2023

## Introduzione

L’elaboratore è una macchina capace di rappresentare, memorizzare e manipolare delle informazioni date in input per produrne delle altre in output.

Esempio: una macchina che riceve in input 10 numeri interi, li ordina e salva il risultato in memoria.

Il concetto stesso di elaborazione è intimamento connesso a quello di macchina che la esegue. Il modello matematico più noto si chiama Macchina di Turing: è un elaboratore teorico in grado di eseguire qualsiasi algoritmo.

( L’elettronica cambiò tutto introducendo il transistor anziché la valvole: le informazioni ora sono rappresentate con valori di tensione elettrica ).

## Moore’s law

La storia dell’elaboratore presenta alcune caratteristiche generali:

* Miniaturizzazione: sono diventati sempre più piccoli (nello stesso spazio si riescono a mettere più componenti)
* Aumento della velocità
* Efficienza energetica: sono più efficienti (consumano meno energia),
* Diminuzione dei costi: sono più economici
* Differenzazione: sono più specializzati (ci sono più elaboratori specifici per fare certe cose, come per esempio CPU e GPU)

L’evoluzione di alcuni di questi principi fu intuita già nel 1965 da Gordon Moore.

Legge di Moore: la complessità di un circuito integrato raddoppia circa ogni 18 mesi.

Ha descritto lo sforzo ingegneristico nell’evoluzione degli elaboratori digitali negli ultimi 50 anni, che hanno aumentato in maniera esponenziale il numero di transistor nei processori renendoli più performanti. Si sta però raggiungendo un limite: c’è un impossibilità nello sostenere la crescita esponenziale delle performance poichè i transistor sono troppo stretti tra di loro e soppraggiungono dei limiti fisici.

[ Una soluzione a ciò sarebbe costruire processori più specifici ].

## Architettura di un elaboratore

È una descrizione di come è fatta una maccina in grado di svolgere elaborazione automatica dell’informazione.

Instruction set -> quali sono le operazioni elementari (istrzuioni) che la macchina può eseguire. )è l’inizio del software)

Hardware organization -> come i componenti sono connessi e operano tra loro.

Hardware design -> design logico dei circuiti che realizzano i vari componenti dell’elaboratore: ALU, control unit, bus, … (Dobbiamo creare dei componenti che svolgano delle funzioni elementari).

**Astrazione:** ogni livello si avvale degli elementi definiti nel livello sottostante trascurando come questi sono fatti all’interno. Io posso usare il software di un computer senza sapere come sia fatto il suo hardware.

## Hardware

È l’insieme dei componenti fisici che compongono un elaboratore, ciascuno adibito ad una particolare funzione (CPU, scheda madre (che serve per connettere i vari componenti), memoria, …)

### Organizzazione dell’hardware

L’organizzazione della maggior parte degli elaboratori oggi segue un modello chiamato Architettura di Von Neumann: ogni elaboratore deve seguire a livello generico il suo schema.

L’elaborazione avviene nella CPU (central processing unit) che ha il compito di eseguire delle istruzioni in sequenza (somme, accesso a memoria, ecc…). La CPU ha 3 sottocomponenti:

* Registri: è la memoria interna
* ALU: esecutore di calcoli logici o aritmetici
* UC: supervisore dell’esecuzione

La Memoria ha il compito di immagazzinare:

* Il programma, ossia la sequenza di istruzioni che deve eseguire la CPU.
* I dati, ossia i valori su cui le istruzioni lavorano o che rappresentano i loro risultati.

La CPU può accedervi in lettura o scrittura attraverso un canale di comunicazione detto Bus.

Un elaboratore deve poter comunicare con il mondo esterno, ad esempio per chiedere un dato all’utente o visualizzare il risultato di un’operazione su un terminale, e si avvale di:

* Periferiche di input: acquisizione di un dato (tastiera, mouse, …)
* Periferiche di output: visualizzare il dato (display, dischi, …)

Limiti di questo modello: bottleneck (collo di bottiglia) -> costruendo macchine di Von Neumann, la CPU ha avuto negli anni un’evoluzione descritta dalla legge di Moore. La memoria invece è migliorata negli anni dal punto di vista della velocità, ma non a questi livelli: la sua crescita è stata, rispetto alla CPU, molto più lenta -> si sono evolute infatti principalmente in densità e affidabilità.

La CPU è velocissima a eseguire le istruzioni, ma poi la memoria trascrive i dati lentamente -> la CPU, che poteva eseguire altre istruzioni, è costretta ad aspettare che la memoria abbia finito.

Bisogna quindi pensare ad un’architettura più complicata per ovviare a questo problema.

### Ciclo di esecuzione di una CPU

* I componenti della M. di Von Neumann svolgono ciascuno, in modo complementare, una particolare funzione.
* Interagendo e coordinandosi tra di loro, questi componenti collaborano all’esecuzione di un - programma, istruzione dopo istruzione.
* L’esecuzione di ciascuna istruzione attraversa 5 fasi che, ripetute per ogni istruzione, costituiscono il ciclo di esecuzione.

**Fetch-Decode-Execute**

**Fetch:** l’istruzione viene prelevata da memoria e caricata in un registro speciale (Instruction Register)

**Decode:** la UC riconosce di che istruzione si tratta e predispone la ALU per la sua esecuzione.

**Execute:** la ALU svolge le operazioni aritmetico/logiche necessarie all’esecuzione dell’istruzione.

**Memory:** viene effettuato un accesso a memoria se l’istruzione lo prevede (opzionale).

**Write Back:** vengono aggiornati i registri.

## Instruction Set

* L’instruction set è l’insieme delle istruzione che la macchina è in grado di eseguire
* Sono esepresse in linguaggio macchina: sequenze di bit (32 in genere) che la UC è in grado di riconoscere come istruzione (decodifica)
* Conoscere l’instruction set di una macchina ci permette di scrivere programmi per quella macchina senza dover considerare i dettagli hardware (astrazione). Ad esempio potremo scrivere un compilatore.

È il primo livello di astrazione sull’hardware: è l’inizio del software.

Macchine diverse possono implementare la stessa Instruction Set Architecture (ISA) (eseguono gli stessi programmi) ma con hardware diverso (e diverse prestazioni e costo)

* **ISA RISC** (Reduced IS Computer): istruzioni semplici e regolari, ciascuna richiede all’hardware poco lavoro per essere eseguita
* **ISA CISC** (Complex IS Computer): istruzioni complesse dove ciascuna può svolgere diversi task e richiede più lavoro all’hardware

### ISA RISC

Vantaggi:

* Semplicità: istruzioni semplici e regolari, più facili da progettare, ma anche decodificare ede seguire
* Prestazioni: istruzioni semplici possono essere eseguite rapidamente con alte prestazioni in task di calcolo.
* Efficienza: istruzioni semplici comportano un consumo energetico più basso e una minore emissione di calore. Sono quindi adatte per dispositivi mobili e sistemi embedded.
* Facilità per il compilatore: sono più facili da compilare.

Svantaggi:

* Aumento della quantità di codice: per fare cose complesse servono tantissime istruzioni.
* Accessi più frequenti alla memoria: e quindi più tempi d’attesa

### ISA CISC

Vantaggi:

* Codice compatto: una singola istruzione può corrispondere a più operazioni
* Set di istruzioni molto ricco: maggiore flessibilità
* Meno accessi a memoria: meno tempi di attesa

Svantaggi:

* Complessità: è richiesta un’architettura più complessa, più difficile da progettare e con consumi energetici più elevati
* Difficoltà per il compilatore: generare codice efficiente è una task molto più difficile
* Obsolescenza: la specificità di un’sitruzione può portare ad un suo abbandono nel tempo (con altri processori)

Le architetture RISC sono usate nei dispositivi embedded (stampanti, tv, router, …), devide mobili e nei componenti loT

* powerPC:usata in alcuni PC, ora per lo più in processori integrati
* ARM: smartphone (android), raspberry pi, …
* IRSC V
* MIPS
* Esempio istruzione RISC (x86): lw $9 8($6) #copia la parola di memoria all’indirizzo $6+8 nel registro $9

Le architetture CISC sono usate spesso nelle macchine general purpose (come i PC desktop) e i server

* Intel X86 (e sua estensione X64)
* AMD 64
* Esempio istruzione CISC (MIPS): Rep movsb #copia il byte all’indirizzo SI nel byte di indirizzo DI, incrementa SI e DI, decrementa CX, se CX non è 0 ripeti.

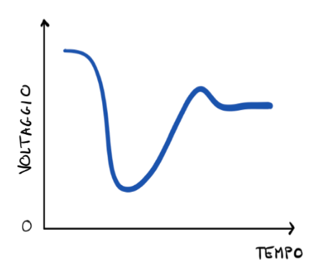
### MIPS

Il MIPS (Multiprocessor without Interlocked Pipeline Stages) fornisce un’ISA di tipo RISC. Nasce a metà degli anni 80 come architettura general purpose. Oggi è impiegata prevalentemente nell’ambito dei sistemi embedded.

# Lezione 2 – 05/10/2023

## Codifica dell’informazione

Per poter svolgere la sua funzione un elaboratore deve poter rappresentare l’informazione su cui lavora attraverso una grandezza fisica: valori di tensione elettrica.

Consideriamo un segnale di tempo che varia nel tempo (asse x il tempo, asse y il voltaggio). Noi mettiamo il valore della tensione in corrispondenza con un fenomeno fisico, come ad esempio la temperatura di una stanza. All’inizio la temperatura sarà alta, ma poi si abbasserà per poi rialzarsi nuovamente. Questo modo di rappresentare l’informazione è analogico: i valori che il segnale può assumere sono strettamente legati, o in analogia, con il fenomeno rappresentato (segnale analogico).

In contrapposizione al metodo anologico che rappresenta, l’intensità del fenomeno, l’approccio digitale decide invece di rappresentare le cifre del valore dell’intensità del fenomeno.

Il tipo di informazione più importante trattata da un elaboratore è quella numerica.  
Supponiamo di dover rappresentare il valore diciotto:

* Analogico: il valore del segnale (voltaggio) rappresenta il valore 18
* Digitale: il valore del segnale rappresenta una cifra del valore 18. Servono più segnali, uno per ogni cifra.

I computer moderni utilizzano rappresentazioni binarie, con due simboli che, per convenzione, chiamiamo 0 e 1. Il segnale è fisico, con una tensione magari di 4,8 V, ma ciò che rappresenta è logico, ossia uno 0 o un 1.

La memoria digitale è fatta da miliardi di componenti (basati su transistor) in grado di mantenere al loro interno un segnale che possono rappresentare le due cifre binarie, ossia dei bit.

Il numero 18 va quindi rappresentato in una sequenza di cifre binarie, e bisogna farlo in modo che sia possibile farci delle operazioni. Per farlo ci avvaliamo della teoria dei sistemi di numerazione.

## Sistema di Numerazione

Un insieme di regole che ci inventiamo per rappresentare il tipo di informazione numerica. Da sempre un sistema di numerazione è composto da due ingredienti fondamentali: la base e la notazione.

### Base

La base è un insieme di simboli, cifre, che possiamo usare per rappresentare un numero. Ognuno di questi simboli è associato ad una quantità numerica.

* B10 = {0, 1, 2, …, 9}; B8 = {0, 1, 2, …, 7} sistema ottale; B2 = {0, 1} sistema binario;   
  B16 = {0, 1, 2, …, 9, A, B, C, D, E, F} sistema esadecimale (0x9F3A op. 9F3A).
* I numeri romani usano simboli diversi.

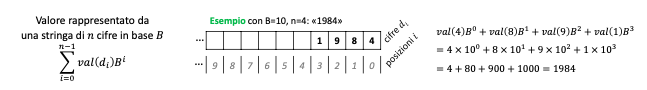
Avendo a disposizione 𝑛 cifre in base 𝐵 ogni cifra può assumere 𝐵 valori diversi e in totale si possono scrivere 𝐵𝑛 stringhe diverse.

* Con 4 cifre in base 16 posso scrivere 164 diversi numeri.

Ogni simbolo è associato ad un valore. Il simbolo “x” è associato, in tutte le basi in cui compare al valore x: ad esempio “9” rappresenta il valore 9 nelle basi decimale ed esadecimale. In base 16 “x” può corrispondere al simbolo E, ma il valore di “x” è 14. Il simbolo esiste in modo concreto, mentre il valore esiste in maniera astratta, è un concetto.

### Notazione

Serie di regole con cui si calcola il valore rappresentato a partire dalla sequenza di simboli che lo rappresenta.

Una notazione fondamentale è quella posizionale: si fa la somma pesata dei valori associati ai simboli; i pesi dipendono dalla base utilizzata e dalla posizione del simbolo nella sequenza.

Quando si passa da una posizione all’altra si cambia ordine di grandezza.

La cifra più a sinistra è detta Most Significant Digit, quella più a destra Least Significant Digit.

## Codifica dei Naturali N

Per convertire un numero da base B a base 10 bisogna fare il calcolo della somma pesata:

Per convertire un numero da base 10 a base B bisogna applicare l’algoritmo iterativo delle divisioni:

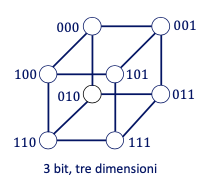
- Dato (𝑁)10 da convertire nella base B:

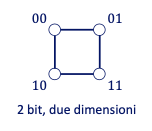
1. dividere 𝑁 per 𝐵 (con una divisione intera);
2. il resto della divisione diventa la prima cifra meno significativa che resta da calcolare del numero in base B;
3. se il quoziente è 0 abbiamo finito;
4. se il quoziente è diverso da 0 si torna al passo 1 considerando il quoziente come dividendo 𝑁;

- Da base 2 a base 16

Queste due basi sono una la potenza dell’altra. Ogni cifra esadecimale corrisponde a uno dei 16 possibili gruppi diversi di 4 cifre binarie.

### Rappresentazione grafica numeri binari

Un numero binario su 𝑛 bit può essere interpretato come un punto in uno spazio 𝑛-dimensionale.



In tutti i casi, numeri adiacenti (collegati) differiscono solo di un bit.



Dati due numeri binari 𝑛1 e 𝑛2 il numero di posizioni in cui un bit ha valore diverso tra un numero e l’altro si chiama distanza di Hamming tra 𝑛1 e 𝑛2.

Misura la distanza tra due codifiche, la differenza tra due rappresentazioni di due numeri che è diversa, in generale, dalla differenza tra i due valori rappresentati.

# Lezione 3 – 09/10/2023

## Codice di Grey

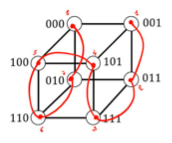
Supponendo di avere 3 bit, in base B2 possiamo scrivere Bn numeri naturali, ossia 23 = 8.

|  |  |  |
| --- | --- | --- |
| Numero Decimale | Codifica Binaria Posizionale | Codifica Binaria di Grey |
| 0 | 000 | 000 |
| 1 | 001 | 001 |
| 2 | 010 | 011 |
| 3 | 011 | 010 |
| 4 | 100 | 110 |
| 5 | 101 | 111 |
| 6 | 110 | 101 |
| 7 | 111 | 100 |

Nella codifica posizionale, le distanze di Hamming tra codifiche di numeri successivi sono: 1,2,1,3,1,2,1.

- Nel codice di grey, invece, le distanze di Hamming tra numeri successivi sono sempre pari ad 1.

* Diventa però più complicato calcolare il valore di un numero a partire dalla sua codifica
* Vantaggio: si può lavorare premendo un unico interruttore in quanto da un numero precedente a quello successivo cambia un solo bit.



I codici che hanno questa proprietà sono detti codici a distanza unitaria. Come trovarli? Percorso Hamiltoniano sulla rappresentazione grafica (percorso che visita tutti i punti una volta sola percorrendo i collegamenti).

### Operazioni aritmetiche

Si usano le stesse regole della base 10 con somme e riporti (differenze e prestiti), che valgono indipendentemente dalla base utilizzata.

- Tutte le operazioni che producono un risultato a partire da uno o più operandi possono causare un overflow

* Vuol dire che il risultato dell’operazione è troppo grande per essere rappresentato su 𝑛 bit, ne servono almeno 𝑛 + 1
* Il numero numero naturale più grande che posso scrivere con n bit è 2n – 1

## Codifica degli Interi Z

Numeri interi Z = {..., − 3, −2, 0, 1, 2, 3, ...}

I numeri naturali sono sufficienti se dobbiamo solo contare (o ordinare), ma limitarci a quelli sarebbe ovviamente fortemente limitante (es: non posso svolgere differenze arbitrarie tra naturali).

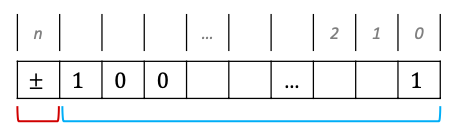
Consideriamo i numeri interi che presentano una caratteristica in più: il segno, che dobbiamo rappresentare in qualche modo.

Ci sono diverse soluzioni che presentano vantaggi e svantaggi per un elaboratore:

* Modulo e segno (molto semplice, ma non molto vantaggiosa)
* Complemento a 2 (più complicata, ma vantaggiosa e quindi molto usata)

### Modulo e Segno

Supponendo di avere a disposizione n bit:



**MSD** indica il segno:

- 0 per positivo (+)

- 1 per negativo (-)

I restanti **𝑛 − 1** bit indicano il modulo con la stessa notazione dei numeri naturali.

Per l’elaboratore questa rappresentazione, che replica il procedimento umano, è inefficiente:

* Ridondanza: lo zero ha due codifiche + 0 e – 0.
* Complessità: certe operazioni risultano laboriose, ad esempio la somma algebrica richiede di (a) controllare il segno degli operandi, (b) sottrarre il maggiore al minore se i segni sono diversi o (c) sommare i valori se i segni sono uguali e (d) calcolare il segno del risultato. Queste operazioni non possono essere svolte contemporaneamente, c’è una sequenzialità, e il circuito che le realizza diventa complesso.

### Complemento a 2

È un modo di rappresentare i numeri interi che permette di fare la somma algebrica facilmente.

Supponendo di avere a disposizione n bit e un numero intero N da codificare:

* Se N è positivo o nullo lo si codifica come il numero naturale di valore N su n – 1 bit e si pone il MSD a 0.
* Se N è negativo lo si codifica come il naturale 2n - |N| su n bit, ossia il valore che mancherebbe a |𝑁| per arrivare a 2𝑛 (il suo complemento a 2𝑛). Il MSD vale sempre 1 dopo aver applicato questa formula.

In entrambi i casi bisogna stare attenti al problema della rappresentabilità: il numero naturale N deve poter essere rappresentabile nei bit a disposizione (per es: nei numeri positivi entro n – 1 bit).

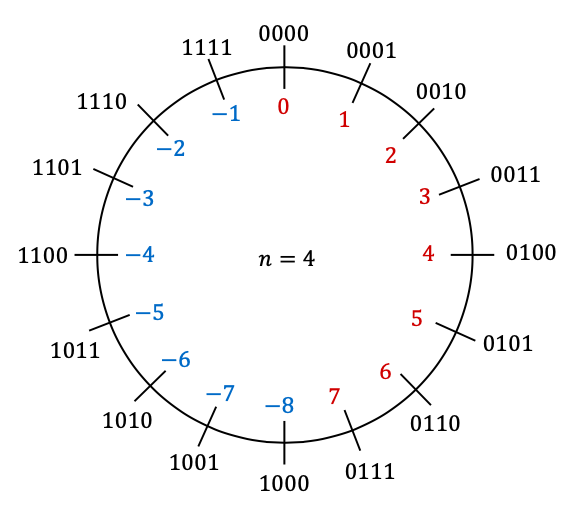
Esempi:

* 𝑁 = 5 -> converto 5 su 3 bit e aggiungo uno 0 a sinistra -> 0101
* 𝑁 = 8 -> converto 8 su 3 bit -> overflow. 4 bit non bastano (sarebbe 01000 = 5 bit minimi)
* 𝑁 = −8 -> converto 24 – 8 = 8 su 4 bit -> 1000
* 𝑁= −5 -> converto 24 – 5 = 16 – 5 = 11 su 4 bit -> 1011 (= 11 in cod. binaria)
* 𝑁 = −11 -> converto 24 − 11 = 5 su 4 bit -> 0101

Ma quindi è 5 o −11? Bisogna fare attenzione all’intervallo rappresentabilità.

(-11 è sbagliato perché non si può rappresentare in 4 bit, ma su almeno 5 bit)

- Con 𝑛 bit posso scrivere 2𝑛 stringhe binarie, le rappresento su una ruota in ordine orario crescente secondo il valore che avrebbero se fossero naturali.



- Per i positivi devo usare n – 1 bit e aggiungere 0 a sx:

1. L’N più grande rappresentabile è 2n-1 – 1 (uno spazio è occupato dallo zero).
2. Tutti i positivi e lo zero iniziano con 0.

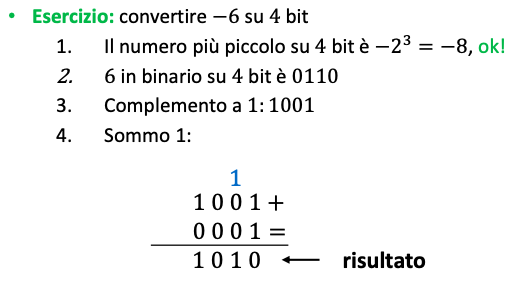
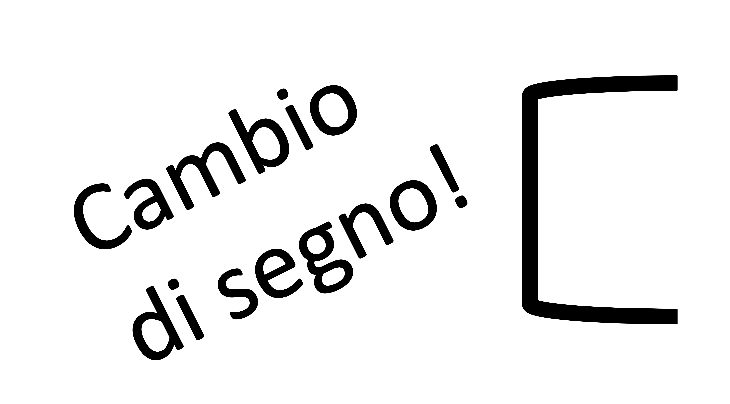
- Le stringhe restanti sono assegnate ai negativi che si dispongono in senso inverso seguendo la regola del complemento: il numero – 𝑁 è assegnato alla stringa che avrebbe valore 2n − 𝑁 nei naturali.

1. L’N più piccolo è -2n – 1
2. Tutti i negativi iniziano con 1

- I numeri che stanno fuori dall’intervallo [−2n - 1, 2n – 1 −1] non possono essere rappresentati su n bit in complemento a 2.

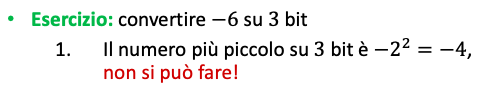
- Primo vantaggio: lo zero ha ora una sola codifica.

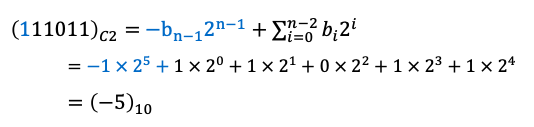
Metodo alternativo per convertire – 𝑁 in C2 su 𝑛 bit:

1. Verifico la rappresentabilità su 𝑛 bit (va sempre fatto)
2. Converto 𝑁 in binario
3. Faccio il complemento a 1

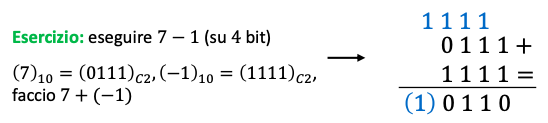
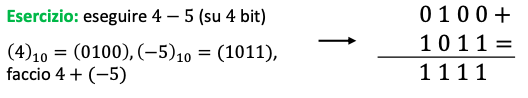
(inverto tutti i bit)

1. So­mmo 1 in binario

(regole dei naturali)

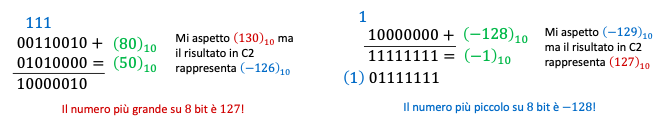
Per convertire da C2 a base 10 basta usare la regola posizionale dando al MSD un peso negativo:

- Il bit più significativo è un bit di segno a tutti gli effetti: può essere esteso senza modificare il valore assoluto: (111111111111111111011)C2 è sempre (−5)10

- Altro vantaggio: le somme algebriche si fanno con lo stesso procedimento dei naturali ignorando l’ultimo riporto.

- Il problema dell’overflow si ripropone anche per le somme in C2.

- Il risultato di una somma di due numeri in C2 su 𝑛 bit potrebbe cadere fuori dall’intervallo di

rappresentabilità [−2n - 1, 2n - 1 −1]

- L’overflow può accadere solo quando si sommano numeri dello stesso segno. Si risconosce quindi molto facilmente:

1. Sommo due numeri positivi (bit di segno 0) e ho un risultato negativo (bit di segno 1).
2. Sommo due numeri negativi (bit di segno 1) e ho un risultato positivo (bit di segno 0).

* Alternativa: controllare gli ultimi due riporti generati, se sono diversi c’è stato overflow.

# Lezione 4 – 12/10/2023

## Codifica dei Reali R

- Sono i numeri che descrivono i fenomeni del mondo.

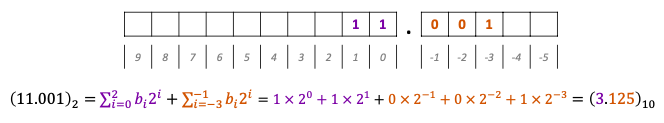
- R presenta due fondamentali differenze con N e Z:

1. Non li usiamo per contare ma per misurare, un processo che di solito richiede di poter rappresentare numeri piccoli che tendono allo zero e numeri molto grandi.
2. A differenza di naturali e degli interi non sono numerabili, non si possono contare: tra due reali qualsiasi ci sono infiniti reali

- Pertanto non si possono rappresentare i numeri reali, ma si possono solo approssimare con dei numeri razionali a precisione finita.

- I numeri razionali sono il risultato di una divisione tra interi, lo sviluppo decimale è infinito ma periodico. Nei numeri reali invece lo sviluppo decimale può essere non periodico.

### Rappresentazione dei numeri R

Per poter rappresentare un frazione di un numero si introduce la virgola e si associa alle cifre decimali degli indici negativi -> la formula della somma pesata si estende di conseguenza.

In questo modo è possibile convertire da base 2 a base 10 un numero con parte frazionaria.

* Assumiamo per semplicità che le parti intere siano sempre naturali, anche se possiamo facilmente generalizzare agli interi pensandole codificate in C2.

Per convertire da base 10 a base 2 si usa l’algoritmo iterativo delle moltiplicazioni:

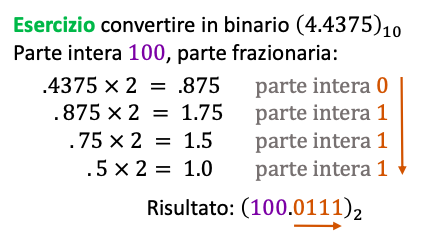
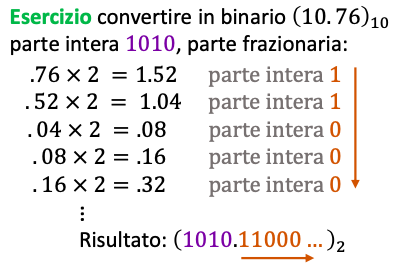
- Dato (*I.F*)10 da convertire nella base B = 2:

1. *I* si converte in binario naturale;
2. Si moltiplica *.F* per 2;
3. La parte intera del risultato diventa la prima cifra più significativa che resta da calcolare;
4. Tornare a 2 considerando la parte frazionaria del risultato al posto di *.F*

- Si termina quando:

1. la parte frazionaria del risultato è 0

- In questo caso il numero frazionario può essere rappresentato con un numero finito di cifre senza ricorrere all’approssimazione (Numeri del tipo 𝑝(2- q ) con 𝑝, 𝑞 ∈ N)

1. Abbiamo finito i bit: si ricorre al troncamento o all’arrotondamento.

**Troncamento:** genero cifre fino a quando esaurisco i bit e lascio così.

(10.76)10 -> (1010.11000010100...)2; se in totale avessi 6 bit -> 1010.11~~00001010…~~

In base 10: 𝜋 = 3.1415~~926535897932384…~~ -> 3.1415

- Il troncamento è sempre un arrotondamento verso lo zero (per difetto sui positivi, per eccesso sui negativi).

**Arrotondamento:** scarto le cifre come nel troncamento, ma scelgo se arrotondare per eccesso o per difetto cercando di minimizzare l’errore di approssimazione.

(101.1101~~1010100~~)2: per eccesso 101.110 + 000.001 = 101.111

(0.001010~~011010100~~)2: per difetto 0.00101 (come nel troncamento)

## Rappresentazioni in virgola fissa e virgola mobile

- Bisogna capire ora come ripartire gli n-bit tra la parte intera e quella frazionaria.

- I due metodi principali per organizzare i bit nella rappresentazione dei reali sono:

1. Rappresentazione in virgola fissa
2. Rappresentazione in virgola mobile

### Virgola fissa

Assegno un numero di bit 𝑛I alla parte intera, i restanti 𝑛 – 𝑛I = 𝑛F a quella frazionaria e mantengo questa ripartizione sempre.

* Il numero 11.001 presenta 𝑛 = 5 bit, 𝑛*I* = 2 bit per la parte intera, 𝑛*F* = 3 bit per la parte frazionaria.

La virgola non cambia mai posizione! Ha sempre 𝑛*I* cifre alla sua sinistra e 𝑛𝐹 alla sua destra, essendo implicita può essere omessa (non uso bit per indicare la sua posizione nel numero).

* Il numero massimo rappresentabile è 2𝑛I  - 1 + ~1 ≅ 2𝑛I.
* Il numero più vicino allo 0 rappresentabile è 2-𝑛F

2-𝑛F è il contributo più piccolo possibile dato da un bit nella nostra codifica, è anche la differenza di valore tra due rappresentazioni numeriche successive (due tacche): viene anche chiamato **precisione**.

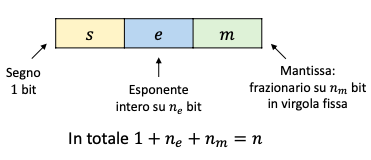
* La metà di questo valore rappresenta l’errore di approssimazione se si applica l’arrotondamento. Errore = 2-𝑛F / 2 = 2-𝑛F-1
* Più cifre dedichiamo alla parte frazionaria, più piccola è la precisione e più fitte sono le tacche sulla retta orientata di R: si ottiene un’approssimazione migliore.

### Virgola mobile

Nella rappresentazione in virgola mobile la virgola non ha un posizione prefissata, ci permette di rappresentare nella stessa codifica i numeri: 11.011 e − 0.0001111. In virgola fissa queste due scritture non possono coesistere.

* La posizione della virgola non è più implicita, dobbiamo usare bit per dire dove sta.
* La precisione diventa ora variabile.

Dati n-bit, la notazione del numero è suddivisa in tre campi:

1. 1 bit per il **segno** s del numero (0 per dire positivo, 1 per dire negativo)
2. ne bit che codificano un numero intero con segno detto **esponente** e
3. nm bit che codificano un numero frazionario positivo in virgola fissa detto **mantissa** m

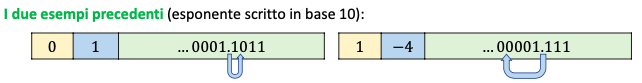
Il valore che questi tre campi rappresentano si

calcola con la formula:

(-1s ) · m · 2e

L’esponente fa quindi muovere la virgola.

Questa notazione è anche detta scientifica.



- La notazione conveniente è detta **forma normalizzata**.

- Il numero in virgola mobile è normalizzato se la parte intera della mantissa ha una sola cifra significativa.

* In base 2 significa che la mantissa è sempre fatta così 1.10110... (“0.” non è ammesso per definizione): “1.” diventa quindi implicito.
* I confronti tra due numeri diventano più semplici: si confrontano gli esponenti (che rappresentano gli ordini di grandezza) e, se uguali, si confrontano le mantisse.
  + 1101 × 2-1 e 10.11 × 2-1 (non normalizzati)
  + 1.101 × 22 e 1.011 × 22 (normalizzati)

### Confronto virgola fissa e mobile in base 10

- Si suppone di avere a disposizione n = 6 cifre decimali e un bit di segno trascurabile.

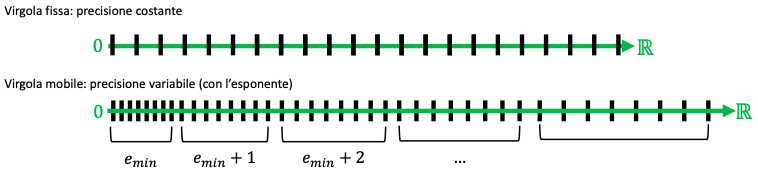
* Per la virgola fissa assegniamo 3 cifre alla parte intera e 3 alla parte frazionaria: 123.447; 003.012
* Per la virgola mobile assegniamo 4 cifre alla mantissa (di cui una per la parte intera) e 2 all’esponente con segno: 1.122 x 10-21; 0.043 x 10 04

- Il numero massimo rappresentabile è:

* Per la virgola fissa: 999.999 ≅ 103
* Per la virgola mobile 9.999 × 1099 ≅ 10100,
* In entrambe le notazioni oltre ai rispettivi valori si va in **overflow**.

Il numero più vicino allo zero rappresentabile è:

* Per la virgola fissa: 000.001 = 10-3,
* Per la virgola mobile 0.001 × 10-99 = 10-102
* In entrambe le notazioni per i numeri che sono ancora più vicini allo zero di ciò che è possibile rappresentare si va in **underflow** (non è rappresentabile perché è troppo piccolo).

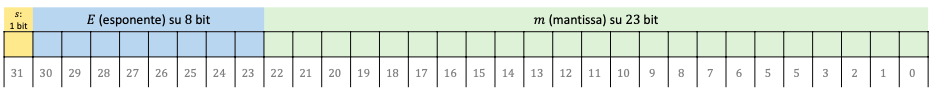
- La rappresentazione in virgola mobile, a differenza di quella in virgola fissa, ci consente di scrivere numeri molto grandi e molto piccoli poiché la precisione è variabile: le tacche sono distribuite diversamente sulla linea.

# Lezione 5 – 16/10/2023

## Lo standard IEEE 754

- L’implementazione della rappresentazione in virgola mobile dentro i calcolatori moderni è regolata da uno standard: l’IEEE-SA Standard n. 754 for Floating-Point Arithmetic (in breve, IEEE 754)

- Vengono proposti due standard differenti (i calcolatori moderni gli adottano entrambi):

1. Formato a precisione singola (float) su 32 bit
2. Formato a precisione doppia (double) su 64 bit

- Con questi 32 bit si possono rappresentare:

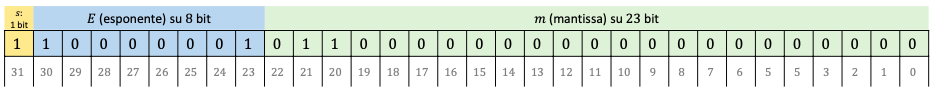
1. Numeri in virgola mobile normalizzati
2. Numeri in virgola mobile non normalizzati (detti sub-normalizzati o de-normalizzati)
3. Codici speciali

### Numeri normalizzati

Se 0 < E (un binario naturale) < 255 allora i 32 bit stanno codificando un numero normalizzato. Di conseguenza dobbiamo interpretare i 3 campi in questo modo:

* *s* è il segno del numero (0 positivo, 1 negativo)
* *m* è la parte frazionaria della mantissa in forma normalizzata: (1.m)2 dove “1.” è implicito
* *E* è il valore dell’esponente a cui è sommato 127 (“in eccesso” 127): il vero esponente è *e* = *E* – 127

Il numero rappresentato è (-1)s x 1.m x 2e

**Esercizio**: cosa rappresenta il seguente numero in formato IEEE 754?

Il segno è negativo.

*E* = 1 x 27 + 1 x 20 = 128 + 1 = 129 -> numero normalizzato

Esponente *e* = *E* - 127 = 129 – 127 = 2, mantissa = 1.011

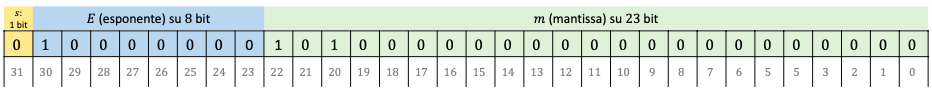
Spostamento della virgola: 1.011 x 22 = 101.1

In base 10: 1 x 22 + 1 x 20 + 1 x 2 -1 = 4 + 1 + ½ = 5.5

**Esercizio**: rappresentare il valore 3.25 in formato IEEE 754 a precisione singola

Converto in binario la parte intera in 11 e la parte frazionaria in .01

Normalizzo 11.01 = 1.101 x 21

s = 0, E = 1 + 127 = 128 = (10000000)2, *m* = 101000 …

### Numeri sub-normalizzati

Se *E* = 0 e *m* ≠ 0 allora i 32 bit stanno codificando un numero sub-normalizzato. Di conseguenza dobbiamo interpretare i 3 campi in questo modo:

1. *s* è il segno del numero (0 positivo, 1 negativo)
2. *m* è la parte frazionaria del numero la cui parte intera è assunta essere 0: (0.*m*)2, dove “0.” È implicito
3. *E* viene scartato e si assume *e* = -126

Il numero rappresentato è (-1)s x 0.*m* x 2-126

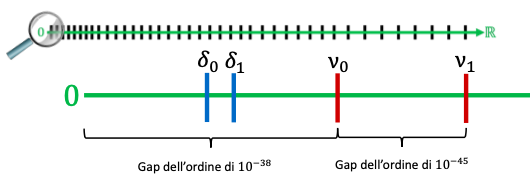
I numeri sub-normalizzati ci aiutano a infoltire la rappresentazione nelle vicinanze dello 0, ma non solamente.

- Il numero normalizzato più vicino allo 0 è 1.000… x 2-126 = 2-126 ≅ 1.17 x 10-38

Il numero successivo è 1.00… x 2-126 = (1 + 2-23) x 2-126 = 2-126 + 2-149 ≅ 1.17 x 10-38 + 1.4 x 10-45

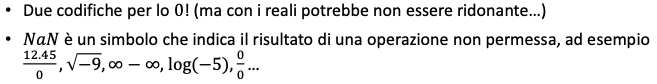
- Il numero sub-normalizzato più vicino allo 0 è 0.000…1 x 2-126 = 2-149 ≅ 1.4 x 10-45

Il numero successivo è 0.00…010 x 2-126 = 2 x (1.4 x 10-45) = 2-148 ≅ 2.8 x 10-45



Nelle vicinanze dello 0 otteniamo una risoluzione maggiore e localmente costante

### Codici speciali



**Considerazioni (Da rivedere tutte)**

* Perché quel formato? Perché usare l’eccesso 127 per l’esponente?
* La risposta sta nella complessità dell’hardware che deve manipolare questi numeri, il formato scelto facilita molte operazioni
* Esempio: confronto fra due numeri IEEE 759, ad esempio stabilire chi è il maggiore
  + Se i segni sono diversi è immediato
  + Se i segni sono uguali basta confrontare i restanti bit come si faceva con i naturali! L’esponente, in posizione più alta, domina i bit della mantissa e il segno non va gestito perché non è codificato esplicitamente (eccesso 127)
* In generale le operazioni in virgola mobile richiedono hardware più complesso delle corrispettive svolte su numeri naturali e interi
* Sono però anche le più frequenti che ci serve svolgere, visto che molti fenomeni del mondo sono descritti da numeri Reali (che noi approssimiamo)
* Negli elaboratori di solito c’è una unità dedicata a queste operazioni (floating-point unit, o co-processore in virgola mobile) e il numero di operazioni in virgola mobile che una CPU può svolgere in un intervallo di tempo è una metrica della sua potenza di calcolo: FLOPS (Floating-point Operations per Second)
* Abbiamo visto modi di rappresentare l’informazione numeri utilizzando simboli binari, 1 e 0, che gli elaboratori sanno rappresentare e manipolare in hardware
* In diversi linguaggi di programmazione, il programmatore ha accesso alla scelta della rappresentazione da usare mediante la specifica del tipo.
* Ad es. in C int significa di solito 32 bit in C2, unsigned int 32 bit in naturale, float IEEE 754 precisione singola, double IEEE 754 precisione doppia. In Go è analogo, in JavaScript i «Number» sono IEEE 754 in doppia precisione

# Lezione 6 – 19/10/2023

## Funzioni e porte logiche elementari

Bisogna progettare un circuito in grado di rappresentare ed elaborare l’informazione numerica attraverso un sistema di numerazione binario.

Un circuito digitale è composto da tanti elementi, ognuno dei quali compie un’elaborazione logica elementare. Connettendo tra loro questi elementi si ottengono circuiti in grado di svolgere elaborazioni più complesse.

### Algebra di Boole

L’algebra di Boole comprende simboli e valori binari su cui possiamo svolgere operazioni logiche:

* Variabili x1, x2, … che possono assumere il valore True (1) o False (0)
* I valori 1 e 0 e le variabili possono essere usate come operandi di tre operatori logici elementari: Not, And e Or.

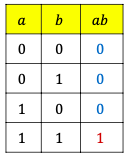
Combinando valori e simboli binari con gli operatori logici elementari possiamo definire espressioni booleane che rappresentano funzioni logiche.

* Insieme dei valori binari *B* = {0, 1}
* Variabile binaria *a* ∈ *B*
* Funzione logica su *n* variabili binarie *f*: *Bn →B*

### Operatore logico NOT

Esprime la negazione logica di un’espressione booleana.

Si indica con NOT(a): se a ha valore 1, la sua negazione ha valore 0 e viceversa



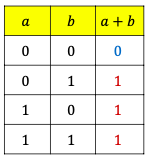
**Operatore logico AND**

Esprime la congiunzione logica tra due espressioni *a* e *b*.

- Si indica con *ab* o con *a* ∧ *b.*

- Se entrambe le espressioni *a* e *b* hanno valore 1 la loro congiunzione vale 1, altrimenti ha valore 0.

- È anche chiamato prodotto logico, e conviene pensarlo come un min.

**Operatore logico OR**

Esprime la disgiunzione logica tra due espressioni *a* e *b*.

- Si indica con *a* + *b* o con *a* ∨ *b*.

- Se almeno una delle due espressioni *a* o *b* ha valore 1, la loro disgiunzione vale 1 altrimenti ha valore 0.

- È anche chiamato somma logica, e conviene pensarlo come un max

**Precedenza tra operatori**

Quando scriviamo espressioni booleane usando questi operatori logici bisogna tenere in considerazione le regole di precedenza con cui si svolgono le operazioni:

* NOT ha la precedenza su AND e OR
* AND ha la precedenza su OR

**Il principio di dualità**

Data un’espressione booleana, la sua duale i ottiene scambiando:

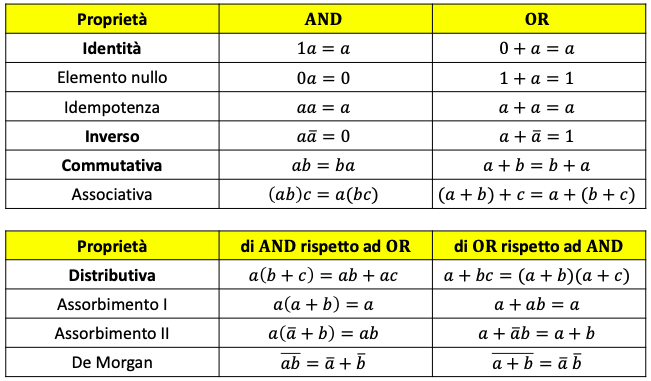
* Gli AND con gli OR e viceversa
* Gli 1 con gli 0 e viceversa

Principio di dualità: se un’uguaglianza booleana è valida allora lo è anche la sua duale.

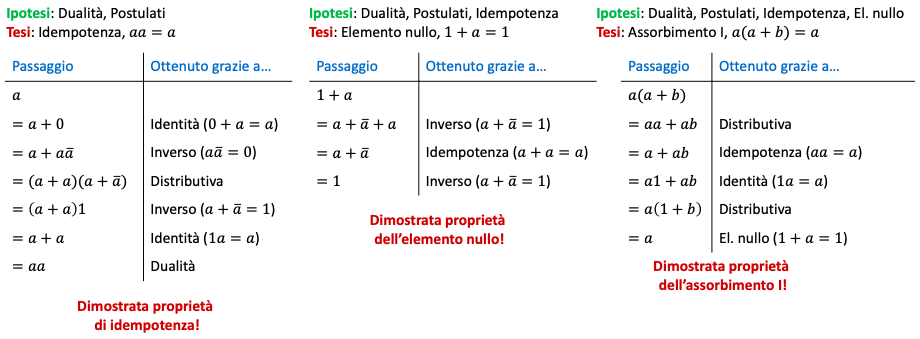
Esempio:

*a* + *ā* = 1, la sua duale *aā* = 0

(*ā* + *b*) + 1 = 1, la sua duale ( *a* + *b̄* ) 0 = 0

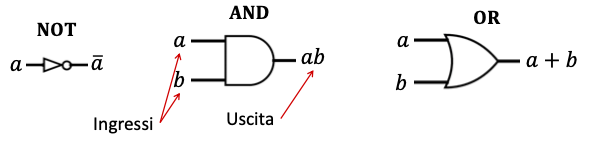
**Proprietà degli operatori logici**

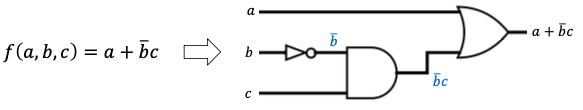
- Identità, Inverso, Commutativa e Distributiva sono postulati: si assumono essere vere.

- Le altre si dimostrano a partire dai postulati applicando le regole dell’algebra

## Porte logiche

Oltre alla loro espressione Booleana, i tre operatori logici possiedono delle “controparti hardware”: un circuito digitale elementare che svolge con i segnali di tensione la stessa funzione che l’operatore logico svolge nell’Algebra di Boole.

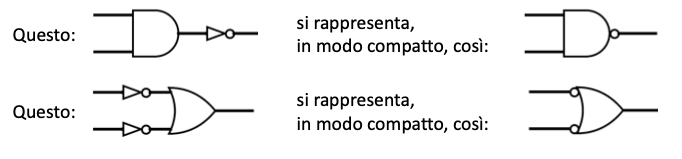
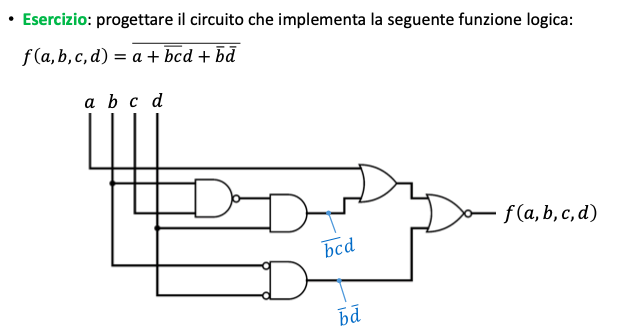
Questi elementi si chiamano porte logiche e si indicano graficamente con questi simboli:

Così come possiamo usare i tre operatori logici per creare funzioni, possiamo analogamente combinare le porte logiche connettendole tra di loro, collegando uscite con ingressi.

Quello sopra rappresentato è un circuito combinatorio: «combina» gli input (che rappresentano le variabili binarie 𝑎,𝑏, e 𝑐) per ottenere un output (che rappresenta la funzione logica 𝑓(𝑎, 𝑏, 𝑐)).

I circuiti combinatori hanno due proprietà fondamentali (che sono in realtà legate tra loro):

1. L’elaborazione procede in un senso solo: da sinistra a destra
2. L’uscita dipende solo dagli input: a parità di input l’uscita è sempre la stessa (è un circuito senza memoria, come vedesse sempre gli input per la prima volta)

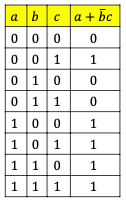
Notazione grafica compatta: quando il NOT (l’inverter) si trova su un ingresso o su una uscita di una porta logica può essere denotato con un pallino su quell’ingresso o uscita:

# Lezione 7 – 23/10/2023

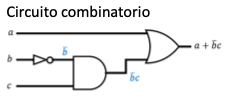
## Metodi per rappresentare una funzione logica

In generale data una funzione logica 𝑓 definita nell’algebra Booleana, abbiamo tre modi di rappresentarla:

1. Espressione Booleana
2. Circuito combinatorio
3. La tabella di verità: una tabella che ha una riga per ogni possibile combinazione di valori di input e che specifica, su ogni riga, il valore della funzione nella configurazione corrispondente



- Funzione *f(a, b, c)* = *a* + *b̄c*

1. Espressione Booleana: *a* + *b̄c*
2. Circuito combinatorio:
3. Tabella di verità:

Data una funzione *f*: *Bn → B*

* La tabella di verità è descrizione esaustiva, ha 2n righe e n + 1 colonne. Ogni una funzione ne ammette una sola.
* L’espressione booleana è una descrizione formale che possiamo leggere facilmente e manipolare con le regole dell’Algebra di Boole. Una funzione ne ammette infinite.
* Il circuito combinatorio è l’implementazione hardware della funzione: una macchina che rappresenta input e output con dei segnali di tensione e che associa a un dato input un output che rappresenta il valore della funzione in quell’input. Una funzione ne ammette infiniti.

## Operatori composti

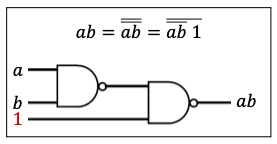
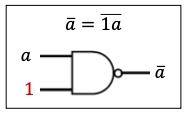
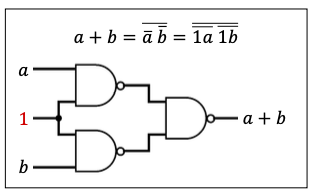
Gli operatori composti sono più complessi degli operatori elementari NOT, AND, OR.

### NAND

“Not AND” è un AND con un NOT all’uscita

- È l’opposto di AND e vale 0 quando entrambi gli input sono pari a 1

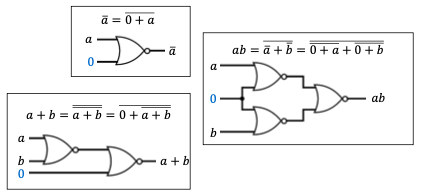
- **Completezza funzionale**: NOT, AND e OR possono essere implementati con la sola porta NAND, e per questo motivo è anche detta *porta universale*.



### NOR

“Not OR” è un or con un NOT all’uscita

- È l’opposto di OR e vale 1 solo quando entrambi gli input sono pari a 0

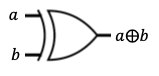
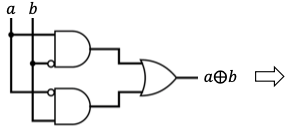
- **Completezza funzionale**: NOT, AND e OR possono essere implementati con la sola porta NOR, e per questo motivo è anche detta porta universale.

### XOR

L’ “Exclusive OR“ è un OR esclusivo, corrisponde all’operatore Booleano ⨁

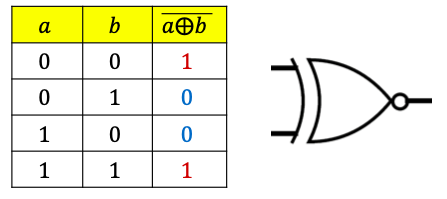
- Vale 1 solo quando uno e uno solo degli input è pari a 1

- Si esprime con l’espressione booleana *ab̄* + *āb*



XOR ha 3 interpretazioni:

* **Funzione di diversità**: vale 1 quando i bit sono diversi
* Può essere usato per fare il complemento a 1 di un bit: a è il dato in input mentre b è un segnale di controllo. Se b=0 il dato a passa verso l’uscita inalterato, altrimenti viene fatto il complemento a 1 -> è un NOT a comando.
* Funzione di parità (pagina 27 appunti)



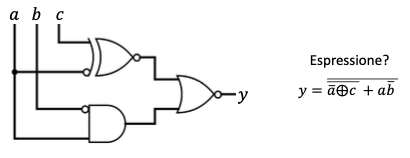
### XNOR

“Not XOR” è uno XOR negato.

- Vale 1 solo quando gli input sono uguali: è una **funzione di uguaglianza**.

## Analisi e sintesi di un circuito: forme canoniche

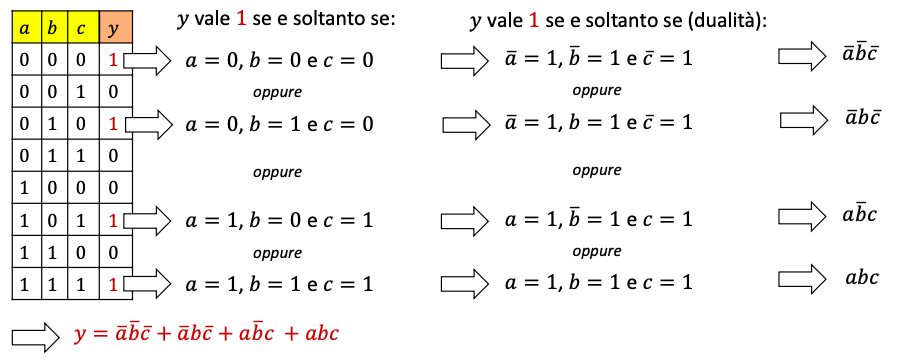
### Analisi e sintesi di un circuito combinatorio

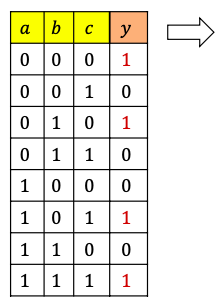
**Analisi di un circuito:** a partire dal circuito o dalla espressione della funzione logica si costruisce la tabella di verità, determinando il valore dell’uscita (o dell’espressione) a fronte di ogni possibile configurazione di input.

**Sintesi di un circuito:** a partire dalla tabella di verità o dall’espressione Booleana costruiamo il circuito che la implementa.

### Prima forma canonica

Si descrive la funzione partendo dalla sua tabella e indicando tutti i punti in cui ha valore 1.

È una descrizione completa per l’ipotesi del mondo chiuso: ciò che non ha valore 1 ha valore 0.

È un modo talmente naturale di descrivere una funzione che prende il nome di prima forma canonica. È anche un procedimento meccanico che possiamo applicare a qualsiasi funzione.

**Mintermini:**

- Definiti come AND tra tutte le variabili (input) della funzione, ogni variabile compare una volta sola nella sua forma naturale o negata

- Ne abbiamo uno per ogni configurazione di input in cui la funzione vale 1

- Se nella configurazione di input corrispondente la variabile vale 0, nei mintermini comprarirà negata, altrimenti in forma naturale.

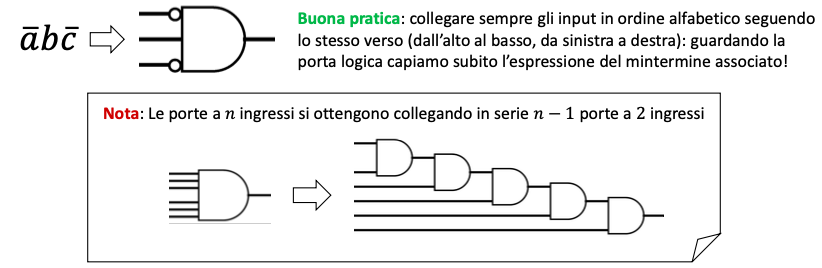
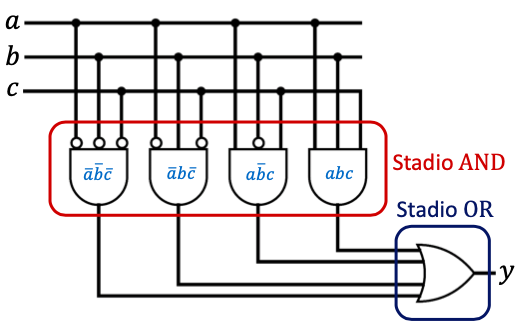
- Chiamando *mi* l­’*i*-esimo mintermine, una funzione *y* può essere sempre espressa come l’OR tra tutti i suoi *n* mintermini:

Prima forma canonica: **somma di prodotti** (mintermini) -> sum of products (SOP)

**Scrittura dell’espressione booleana**

1. Identificare i mintermini della funzione
2. Combinarli con un OR

**Sintesi del circuito**

- Ogni mintermine corrisponde ad un AND a più ingressi (tanti quante le variabili)

- Schema a due stadi

* 1. Stadio AND: una porta AND per ogni mintermine
  2. Stadio OR: un OR tra tutte le uscite dello stadio AND

- Anche la sintesi del circuito, come la scrittura dell’espressione Booleana, è un procedimento meccanico: lo stesso per ogni funzione logica.

- Non essendo la forma più compatta, si semplifica l’espressione per sintetizzare un circuito più semplice: questo perché la SOP implementa molte porte AND e OR con n input, che vengono implementate ciascuna come n – 1 porte da 2 input collegate in serie.

- La forma semplificata non rappresenta però una forma canonica.

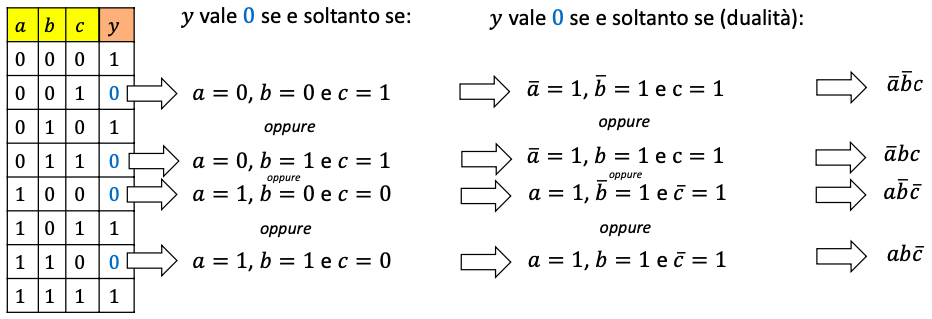
**Implicanti**

- Se una funzione è una somma di prodotti, quei prodotti che non includono tutte le variabili si chiamano implicanti.

- Gli implicanti «sintetizzano» somme di mintermini perché è come se dicessero che le variabili non specificate possono assumere qualsiasi valore.

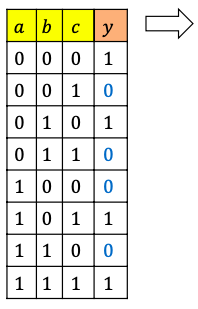
# Lezione 8 – 26/10/2023

### Seconda forma canonica

- Grazie all’ipotesi del mondo chiuso posso fornire una descrizione completa della funzione logica indicando tutti i punti in cui ha valore 0, ciò che non indico avrà implicitamente valore 1.

- È la seconda forma canonica, ottenuto con un ragionamento duale rispetto alla prima.

- Anche questo è un procedimento meccanico che possiamo applicare a qualsiasi funzione.



**Maxtermini**:

- Definiti come OR tra tutte le variabili (input) della funzione, ogni variabile compare una volta sola nella sua forma naturale o negata

- Ne abbiamo uno per ogni configurazione di input in cui la funzione vale 0

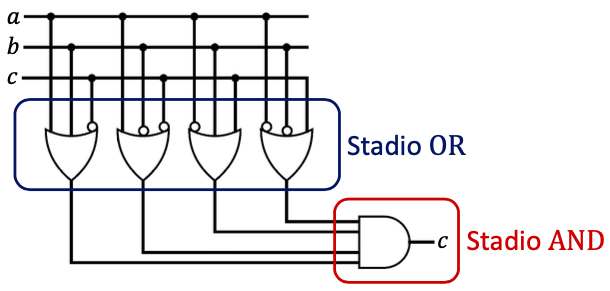
- Se nella configurazione di input corrispondente la variabile vale 1, nel maxtermini comparirà negata, altrimenti in forma naturale

- Chiamando *Mi* l’*i*-esimo maxtermine, una funzione *y* può essere sempre espressa come l’AND tra tutti i suoi *n* maxtermini:

Seconda forma canonica: **prodotto di somme** (maxtermini) -> product of sums (POS)

**Scrittura dell’espressione booleana**

1. Identificare i maxtermini della funzione
2. Combinarli con un AND

**Sintesi del circuito**

- Ogni maxtermine corrisponde ad un OR a più ingressi (tanti quante sono le variabili)

- Valgono le stesse considerazioni che abbiamo per la SOP

## Valutare costi e prestazioni

### Limiti dei circuiti logici

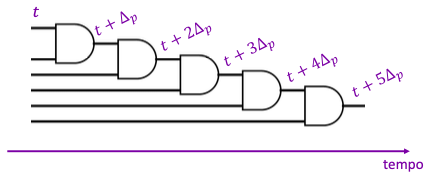
- Le due forme canoniche sono il metodo più semplice con cui sintetizzare un circuito combinatorio a partire dalla tabella di verità

- La semplicità ha però un costo: influisce su aspetti come la lunghezza dell’espressione, la grandezza del circuito e il numero di porte.

Per caratterizzare in modo quantitativo e rigoroso questo costo dobbiamo considerare il fatto che i circuiti sono componenti hardware con annessi limiti fisici:

1. **Propagation delay**: in ogni porta logica se al tempo 𝑡 gli input cambiano, l’uscita non commuta (passa da 0 a 1 o v.v.) in modo istantaneo, l’output sarà stabile dal tempo 𝑡 + Δ𝑝
2. **Fan-out limitato**: il numero di ingressi a cui posso collegare una uscita (pilotaggio) è limitato; in generale collegando un’uscita a un numero maggiore di ingressi il tempo di commutazione aumenta.

### Cammino critico

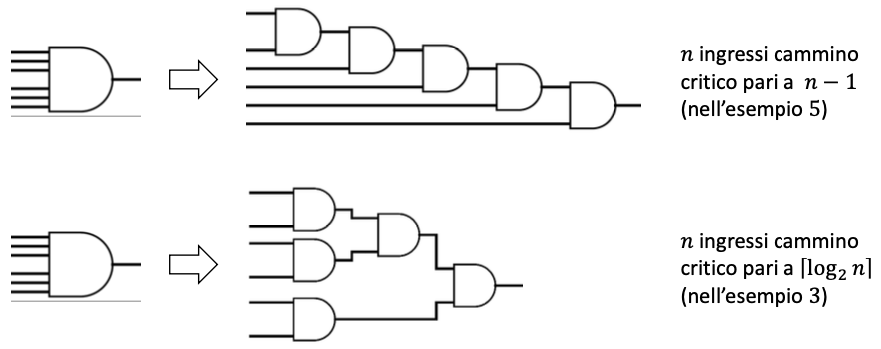
Ogni porta logica che un segnale deve attraversare introduce un ritardo additivo sul tempo di commutazione dell’uscita

- Nel suo percorso dall’ingresso all’uscita il segnale paga un ritardo Δ𝑝 ogni volta che viene attraversata una porta logica

- Dato il percorso da un ingresso ad un uscita, il numero porte logiche attraversate si chiama lunghezza del cammino

- La lunghezza massima di tutti i percorsi presenti in un circuito si chiama **cammino critico**

**-** È una metrica con cui valutare le performance di un circuito, più grande è il cammino critico più lento sarà il circuito

Se consideriamo il cammino critico come metrica di performance possiamo proporre una implementazione migliore delle porte logiche ad n ingressi:

- Altri criteri che possiamo usare: area occupata, numero totali di porte, energia dissipata, facilità di interpretazione

- In generale avere una forma semplificata migliora queste metriche, semplificare è più difficile ma porta a dei vantaggi

## Parità e maggioranza

### Funzione di parità

Fino a qui

È la terza interpretazione dello XOR.

### Funzione di maggioranza

# Lezione 9 – 30/10/2023

## Mappe di Karnaugh

### Semplificazione di circuiti

### Rappresentazione grafica

### Rappresentazione piana in 2D

## Blocchi funzionali combinatori

### Decoder

### Encoder

### Multiplex

### Comparatore

# Lezione 10 – 02/11/2023

## Circuiti aritmetici

### Half Adder

### Full adder

### Sommatore a propagazione di riporto

### Anticipazione di riporto

# Lezione 11 – 06/11/2023

110

101