

# **Appunti Programmazione 1 - Riccardi**

Giovanni Foletto, Enrico Carnelos, Stefano Viel

2020-11-21

Giovanni Foletto, Stefano Viel, Carnelos Enrico - Primo anno ICE

## PROGRAMMAZIONE 1 - RICCARDI

### Lezioni teoriche

#### 0.0) Obiettivi e Cos'è l'informatica:

**Obiettivo:** conoscenza di base dell'informatica.

**Informatica:** dal francese, informazione automatica. Termine coniato da Ph. Dreyfus nel 1962

Questo significa che è l'insieme degli aspetti scientifici e tecnici che sono specificatamente applicati alla raccolta e al trattamento dell'informazione, in particolare all'elaborazione automatica dei dati. Ma anche lo studio sistematico degli algoritmi che descrivono e trasformano l'informazione.

#### 1.1) Algoritmi e linguaggi di programmazione:

Un **algoritmo** è una sequenza precisa di operazioni, comprensibili da un esecutore, che definisce una sequenza finita di passi che portano alla realizzazione di un compito (*task o problema*).

L'algoritmo ha alcune caratteristiche: (2 e 3 sono le più importanti)

1. deve essere **comprensibile** al suo esecutore (linguaggi di programmazione nel caso di un calcolatore). L'algoritmo così codificato viene chiamato *programma*.
2. deve essere **corretto** (l'algoritmo ottiene la soluzione del compito cui è preposto, senza errori in nessun passo fondamentale).
3. deve essere **efficiente** (l'algoritmo ottiene la soluzione usando la minor quantità di risorse).

DA NOTARE CHE: il concetto di algoritmo non è prerogativa dei calcolatori, ma semplicemente il calcolatore (in questo caso un computer) ha capacità di calcolo tali da gestire e lavorare con quantità di dati altrimenti intrattabili.

Gli algoritmi sono il modo in cui, implicitamente o esplicitamente, affrontiamo ogni problema nella vita di tutti i giorni. Questo problema viene chiamato **task**, e l'algoritmo serve a risolverlo.

Il problema dell'algoritmo della scelta invece crea un problema, il cosiddetto **il problema della segretaria**, che contiene una semplificazione matematica dell'algoritmo della scelta ottimale dal punto di vista matematico.

Il concetto di questo problema è che dovendo scegliere di assumere una tra 100 segretarie, con l'unica remora che una volta assunta una, non si continua a cercare. L'algoritmo in questo caso direbbe di

scegliere solo passati i primi 37% elementi (in questo caso segretarie), dopodichè, passato questo step, il primo che si dimostra all'altezza del compito anche in base alle persone viste in precedenza viene assunto e si chiude la ricerca.

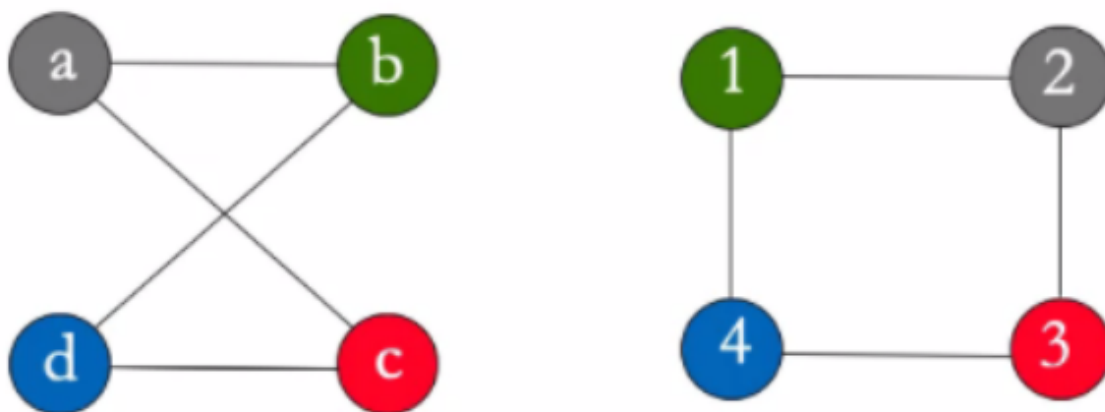
(Vedi: <https://www.smartworld.it/tecnologia/formula-matematica-decisioni-difficili-della-vita.html>).

Dal concetto di ottimizzazione e rendimento dell'algoritmo si incontra anche il concetto di **reward**: ho due scelte, una usuale (**exploit**), un'altra che cambia e esce dagli schemi (**explore**). La migliore dipende dal migliore risultato che le due rendono possibile.

**Il grafo** Per la rappresentazione di alcuni algoritmi è molto utile l'utilizzo del **grafo**. Il grafo è uno schema che collega tra loro le informazioni possedute, ottenute anche fonti diverse. Le informazioni così raccolte hanno il lato positivo di essere facilmente ricercabile, distinguibile e visibile. Inoltre rende evidenti i collegamenti che hanno tra loro.

Questo tipo di schematizzazione, essendo molto ottimizzata e mettendo subito in risalto le informazioni che hanno contatti tra di loro, vengono ampiamente utilizzati nell'intelligenza artificiale, permettendo appunto grandi elaborazioni. Il grafo quindi è un oggetto logico molto potente.

Un accortezza: con questa struttura è facile prendere due grafi uguali come diversi. I grafi si dicono **isomorfi** se hanno le stesse caratteristiche in termini di nodi e archi.



A questo punto però si crea un problema, infatti in alcuni casi, usando un dato algoritmo ottengo risultati diversi se non qualifco un qualche tipo di **parametro**. Proprio il concetto di parametro (inteso nel senso informatico, cioè elemento che caratterizza un determinato algoritmo e che l'algoritmo si aspetta per eseguire un compito). I parametri quindi possono **condizionare l'esecuzione del task**.

**Come descrivere un algoritmo:** L'algoritmo per definizione è una sequenza precisa di passi, di conseguenza una lista non ordinata di elementi non è l'oggetto giusto per la sua rappresentazione. Inoltre il fatto che attraverso un parametro si possa ottenere un risultato diverso rende la sua codifica difficile.

Innanzitutto è essenziale che sia descritto come un programma, cioè come una sequenza di istruzioni scritte in un linguaggio comprensibile al calcolatore. Il compito del programmatore è proprio quello di scrivere del codice che porti alla soluzione ottimale del problema.

Per iniziare bisogna inizialmente **definire esattamente il problema**, che può essere:

1. problema di **puro calcolo e conversione**, cioè risolvibile attraverso conoscenze matematiche.
2. problemi di **decisione**, cioè capire se una certa proprietà è propria di un certo elemento. (es. 15467 è primo?).
3. problemi di **ricerca**, cioè trovare un elemento con determinate caratteristiche in un determinato insieme.

Il passaggio da definizione del problema a esecuzione del task poi passa da altri step, che sono necessari alla codifica e alla comprensione dei vari step:

1. Problema: linguaggio **Lp**, ovvero linguaggio naturale (per descrivere il problema).
2. Algoritmo: linguaggio **La**, ovvero lo pseudocodice (lascia meno incomprensione ma più difficile da leggere).
3. Programma: linguaggio **Lt**, ovvero linguaggio macchina (es. C/C++).

Una volta eseguiti questi **macropassaggi** si può passare alla rifinitura del processo, attraverso dei **micro-algoritmi** (alcune volte il programma e l'algoritmo necessario al suo funzionamento sono così semplici da non necessitare la stesura di micro-algoritmi di supporto).

Su algoritmi molto complessi si può applicare la tecnica del **deconstruct**, ovvero decostruire il problema finché non si ottengono tanti problemi di facile soluzione.

Nel caso della stesura di un algoritmo si devono prendere in considerazione tutte le casistiche. Ad esempio se si scrive un algoritmo per cercare un libro bisogna anche tenere in considerazione la possibilità che questo libro non sia presente nell'indice della ricerca. Allo stesso modo non si può pensare di avere una differenza abissale di prestazioni se il libro è presente nella prima parte di indice o invece è in fondo. Ci sono di solito sempre più di un modo per risolvere problemi, quindi il problema principale diventa quello di trovare il modo più efficiente disponibile.

**La programmazione e il linguaggio di programmazione** Programmare innanzitutto significa analizzare un problema, progettare un algoritmo per ottenere una soluzione, esprimere l'algoritmo in un

linguaggio di programmazione, mettere la macchina nella situazione di eseguire il programma e infine correggere eventuali errori.

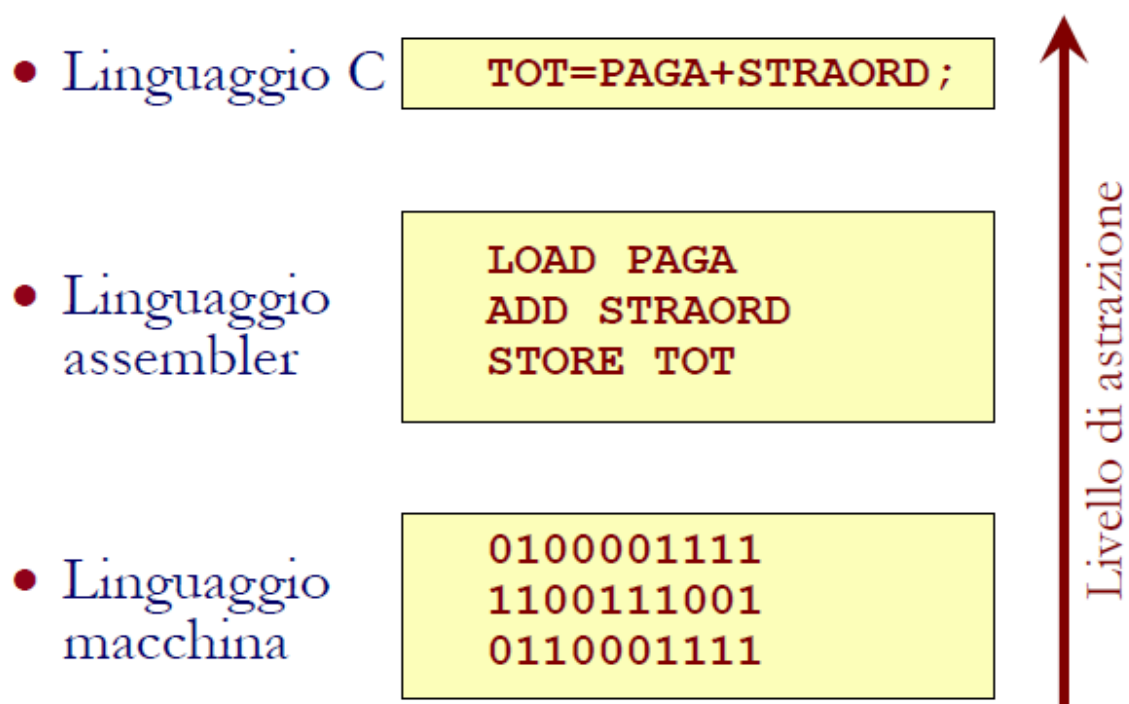
L'algoritmo deve essere tradotto da linguaggio formale e reso comprensibile all'esecutore. L'insieme dei comandi dati al calcolatore in un certo linguaggio viene chiamato **programma**.

Il linguaggio di programmazione deve essere **rigoroso e preciso** dal punto di vista di:

1. **sintassi**: le regole che descrivono le stringhe di parole riconosciute dal linguaggio.
2. **semantica**: regole per l'interpretazione delle stringhe e che descrivono i processi computazionali dell'esecutore (o più semplicemente un errore logico nella stesura del programma, cioè un errore nella comprensione logica del programma). Questo errore è più difficile da individuare perché il compilatore non dà errore, ma i risultati saranno incoerenti.

Parlando di linguaggi di programmazione, bisogna dire anche qualcosa sull'astrazione del linguaggio:

1. se il linguaggio è di bassissimo livello, quindi troppo vicino alla macchina è difficile programmare.
2. se il linguaggio è troppo di alto livello, quindi molto vicino al linguaggio del programmatore, i programmi diventano inefficienti.

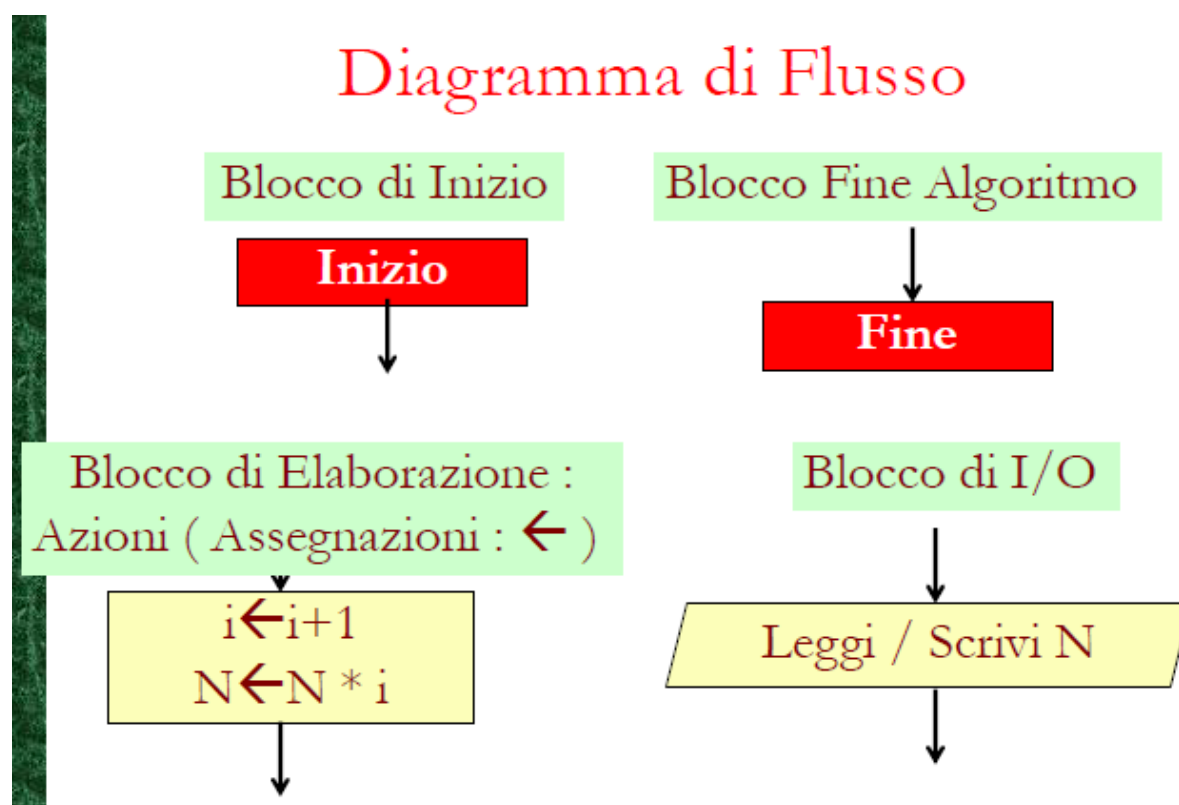


Agli albori dell'informatica si programmava in linguaggio macchina, cioè compreso direttamente dalla macchina.

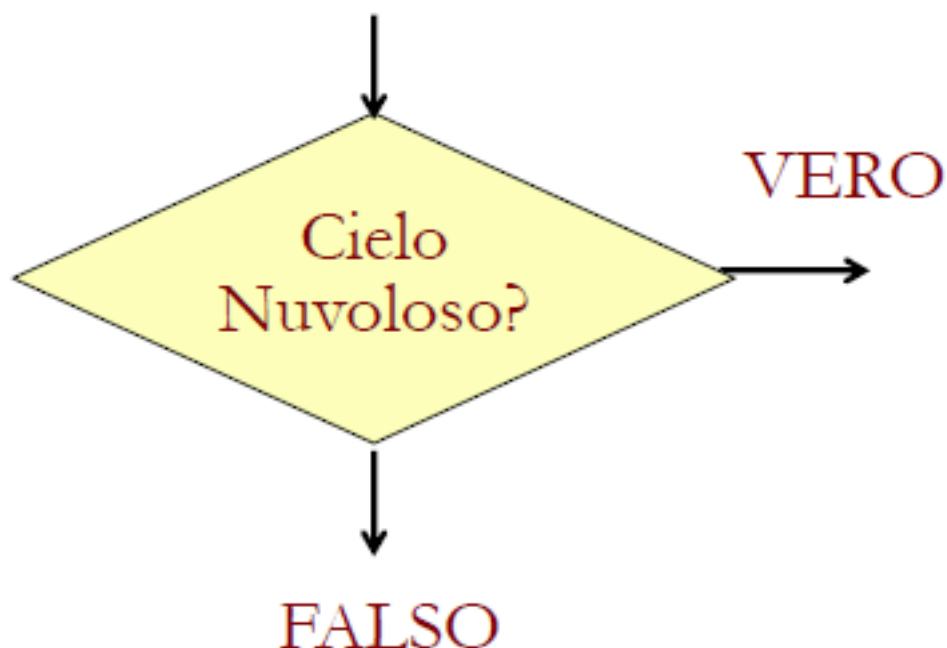
Dagli anni '50 iniziano a nascere i primi linguaggi di programmazione di alto livello, data la sempre più alta complessità dei programmi.

Ad oggi esistono tantissimi linguaggi di programmazione, ognuno con il suo scopo e inventato per esigenze diverse.

**Metodo grafico per creare algoritmi:** Gli algoritmi possono essere schematizzati anche attraverso diagrammi di flusso, cioè blocchi orientati che hanno un significato proprio.



## Blocco di Controllo



### CARATTERISTICHE:

- **BLOCCO INIZIO:** unico nodo del grafo da cui si può solo partire infatti ha una sola freccia uscente. Rettangolo rosso per convenzione. Di solito si definisce con '1' l'inizio.
- **BLOCCO FINE:** blocco in cui si può solo arrivare infatti ha una sola freccia entrante. Rettangoli rossi per convenzione. Di solito si definisce con '0' la fine.
- **BLOCCO ISTRUZIONE:**
  - **assegnazione:** dà il valore che c'è a sinistra alla variabile di destra.
  - **I/O:** (input/output), si sta leggendo o scrivendo
  - **controllo:** (con più uscite, in questo caso vero e falso)

### 1.2) Architettura hardware di un compilatore:

**Storia del Calcolatore:** Il primo calcolatore definito tale è ENIAC, creato nel 1945 in America con il principale scopo di calcolare la traiettoria dell'artiglieria. Poi donato al University of Pennsylvania,

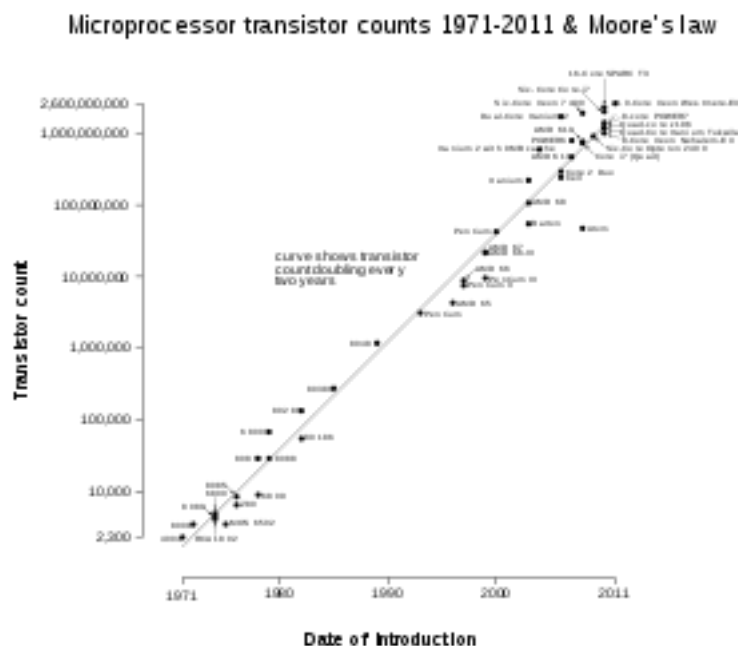
dove invece venne usato per il calcolo ingegneristico e scientifico. Pesava circa 27 tonnellate e per programmarlo erano necessarie diverse persone, infatti si basava su una *plugbboard* cioè un sistema di “prese” che, usate per fare il contatto giusto, fornivano risultati alle operazioni date come input. Non esisteva un linguaggio di programmazione per questo computer e i cosiddetti “programmatori” dell’epoca probabilmente non lo hanno mai visto in tutta la loro vita, dato che loro facevano solo i calcoli per farlo funzionare e poi qualcun altro si occupava di creare i giusti collegamenti. La sua potenza di calcolo era tale da permettere di calcolare quello che un uomo avrebbe fatto in 20 ore in soli 30 secondi. Aveva un clock di circa 100kHz, e garantiva una vasta quantità di calcoli possibili.

Dopo la seconda guerra mondiale inizia effettivamente lo sviluppo di una macchina più vicina all’immaginario comune. Sempre più comuni diventano:

1. MAINFRAME: computer di grandi dimensioni ed elevata potenza, solitamente condiviso fra più persone/uffici.
2. PERSONAL COMPUTER: computer dotati di tastiera e schermo separati, di memoria di massa interni o esterni alla memoria centrale e molto più economico.
3. LAPTOP: computer portatile, nonché evoluzione portatile del personal computer. Diventa molto famoso e utilizzato. Diventano così sofisticati che inizia il processo di miniaturizzazione dato che si poteva fare con le tecnologie disponibili ormai.
4. HANDHELD COMPUTER (SMARTPHONE): cellulari dotati di touchscreen, molto portatili e maneggevoli. Anche con questi come con i laptop avviene il processo di miniaturizzazione.
5. WEARABLE COMPUTER: sono computer così miniaturizzati da poter essere messi all’interno di un orologio, o un altro oggetto facilmente indossabile. Questi solamente sono caratterizzati da una gamma di sensori solitamente molto avanzati che possono servire per monitorare lo stato di salute o altri scopi.

La miniaturizzazione che è avvenuta è dovuta alla possibilità di dimezzare la grandezza del transistor per un certa quantità di volte, data dalla legge di Moore.



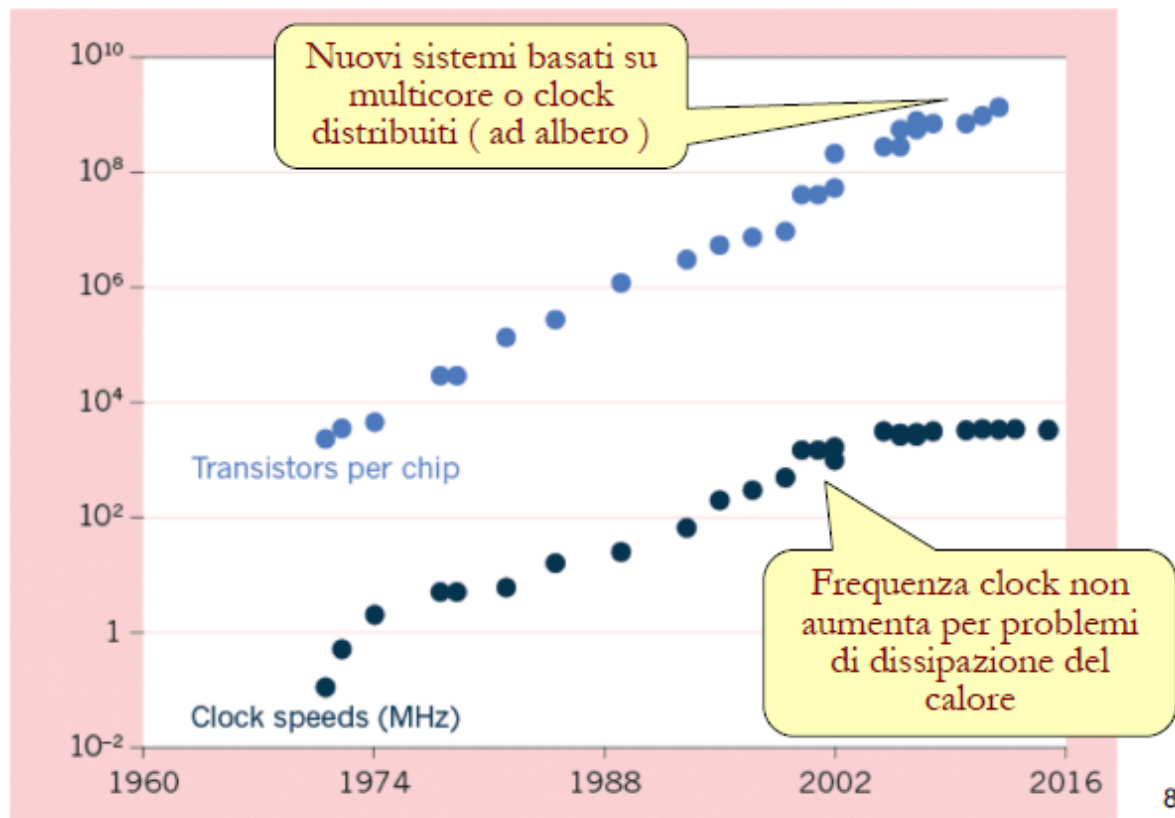


Moore infatti nel 1965 ipotizza che il numero di transistor inseribili nei microprocessori sarebbe raddoppiato ogni anno. La sua previsione si rivela corretta, finché verso la fine degli anni '80 viene riformulata: raddoppio del numero di transistori ogni 18 mesi.

Per aumentare ancora più le prestazioni si cerca di aumentare il clock dei processori stessi. Il **clock** di un processore è il dato che indica la quantità di calcoli può fare al secondo. Al momento il dato del clock è pressoché fermo intorno al dato di 3GHz. Questo stop è dovuto al problema fisico per cui aumentando troppo il clock, il calore emesso diventa esponenzialmente maggiore. Di conseguenza quando si sono provati clock più alti si ha avuto problemi di gestione delle temperature eccessivamente alte. Per ovviare questo problema si apre un'altra strada: il multithreading o clock distribuito ad albero. Questo permette di passare il clock in tutte le parti del processore e rendere quindi i processi sincronizzati, utilizzando la potenza di tutte le parti assieme e quindi garantendo una maggior potenza.

# Legge di Moore

## Transistors e Clock Speed

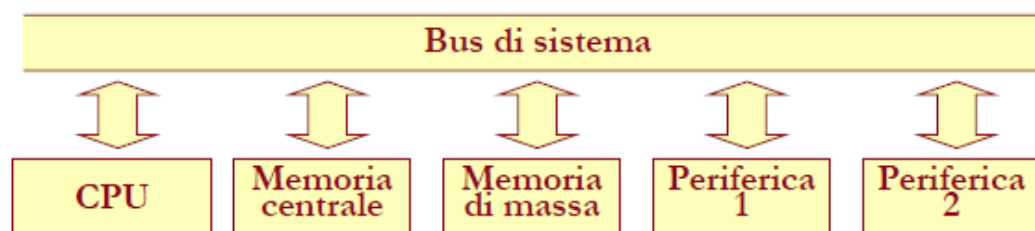


**Organizzare il software di un calcolatore** Il normale schema di un calcolatore è (generalmente) così costruito:



Per leggere questo schema bisogna capire che:

1. Tutto si basa su quello che ha sotto di sé e può comunicare solo con i blocchi a sé adiacenti. (ad es. il software di sistema non si interfaccia direttamente con l'hardware, passa attraverso il sistema operativo).
2. L'unico che può operare con l'hardware è il sistema operativo. Questo ne gestisce tutte le componenti.



#### Architettura Hardware:

L'architettura hardware sopra è nota con il nome di **Macchina di Von Neumann** ed è così costituita:

1. **CPU** (central processing unit): svolge l'elaborazione eseguendo i programmi logici.
2. **MEMORIA CENTRALE** : memoria utilizzata per memorizzare dati e istruzioni, volatile (usata quindi solo per l'esecuzione di programmi).

3. **MEMORIA DI MASSA** : usata per memorizzare grandi quantità di dati e programmi in maniera persistente, al contrario della memoria centrale.
4. **PERIFERICHE**: sono di vario tipo e servono a tutto quello che permette l'interazione con il calcolatore. (ad es. tastiera, monitor, stampante, ..).
5. **BUS DI SISTEMA**: elemento che interconnette gli altri componenti consentendo lo scambio di dati, quindi anche il collegamento fra periferiche e hardware.

### **Macchina di Von Neumann:** 1.2.1) CPU: unità di elaborazione

La CPU è l'unità di elaborazione del calcolatore, si occupa di caricare le istruzioni in memoria centrale, interpretarle e eseguirle.

E' altamente specializzata, perché è pensata per eseguire pochi tipi di operazioni ma molto velocemente.

Il suo lavoro è scandito dal **clock** (orologio interno). La potenza del calcolatore dipende in parte dal clock, infatti quanto più questo è alto, tante più sono le istruzioni che riesce ad eseguire al secondo. La sua frequenza infatti viene misurata in Hz, e  $1 \text{ Hz} = 1 \text{ ciclo/s}$ . Ad oggi le CPU riescono a lavorare in parallelo, il cosiddetto "lavoro condiviso", grazie al clock. Infatti il segnale del clock che arriva a tutti sincronizzato permette di eseguire azioni contemporaneamente.

### 1.2.2) Memoria centrale:

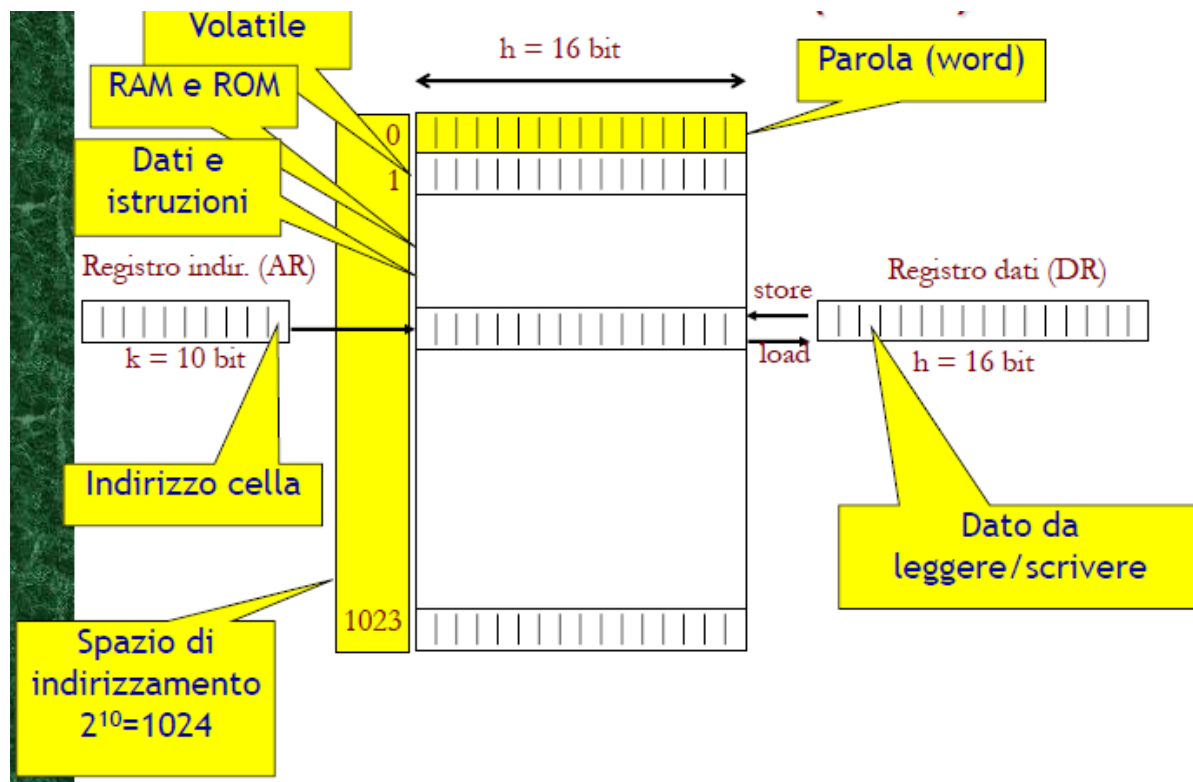
La memoria centrale è destinata a accogliere dati e programmi sui quali opera il calcolatore. (ad es. quando usiamo il computer qui vengono memorizzati i dati di programmi che stiamo usando, ...). La memoria centrale è velocissima ma volatile (cioè una volta che il programma finisce, tutti i dati di un programma vengono cancellati e lo spazio viene liberato). Questa memoria accoglie i dati necessari a far funzionare i programmi. Concettualmente è composta da una sequenza di celle ognuna delle quali contiene una parola (**word**). Ad ogni cella si può accedere direttamente specificandone l'**indirizzo**, e accedendone si può cambiare il suo contenuto (read/write). La quantità di bit da cui è composta la word dipende dalla macchina, infatti è caratteristico del microprocessore e attraverso questo si identifica lo spazio di indirizzamento.

La memoria centrale solitamente è realizzata da una **RAM** (random access memory). La sua caratteristica è che ogni cosa è accessibile e non devo scansionare tutti gli indirizzi per cercare la cella che ho bisogno. (per es. non è come il nastro magnetico che devono fisicamente essere spostate avanti e indietro per trovare l'informazione cercata). Questo ha ricadute dirette sul tempo di accesso, che è indipendente dall'indirizzo della word che si vuole accedere. Questa è una memoria volatile quindi i dati presenti qui sopra si perdono quando si spegne la macchina. Esistono SRAM (static RAM), molto veloce, può contenere pochi dati acceduti molto frequentemente, e DRAM (dynamic RAM) che contiene più dati, ma più lentamente.

Altri tipi di memoria presenti sono le **ROM** (read-only memory). Hanno caratteristiche simile alle RAM, hanno un accesso veloce ai dati (ma non come quello delle RAM) e si tratta di **memorie permanenti** su cui però non si può scrivere. Tipicamente sono utilizzate per memorizzare dati e programmi che servono prima del caricamento del Sistema Operativo (ad es. per il caricamento del BIOS = Basic Input-Output System).

La memoria quindi com'è fatta? Appare come una lista di word cioè un mattoncino composto da 16bit, di conseguenza posso memorizzare tanti dai quanti  $2^{16}$  valori binari. All'interno della memoria ci posso mettere tante word come gli indirizzi di memoria che sono 1024, e sono assegnati da 0 a 1023. Se questi fossero i limiti del tuo microprocessore allora devi fare in modo che all'interno di questo spazio di memoria ci stiano i dati e i programmi necessari al tuo scopo.

Per questo motivo in memoria non si può accedere a valori più grandi di  $2^{16}$  e non si può accedere all'indirizzo 1024.



Però poi a questa memoria si deve poter accedere, quindi nella CPU esistono dei registri, che sono word di memoria chiamate così per distinguerle da quelle in memoria centrale. Questi registri possono essere:

1. AR (Address Register): di almeno 10 bit, perché in questo caso posso indirizzare da 0 a 1023
2. DR (data Register): posto in cui i dati di cui la CPU ha bisogno per fare operazioni che vengono copiati in questa posizione dalla memoria centrale per rendere le informazioni direttamente

disponibili alla CPU.

I registri sono elementi di supporto al calcolo.

### 1.2.3) Gerarchia di memoria

Ci sono a disposizione memorie con caratteristiche diverse in base al loro scopo:

1. Registri: presenti nella CPU, piccoli ma veloci, utili per contenere dati temporanei per le operazioni della CPU
2. Memoria cache (SRAM): veloce, contiene pochi dati usati/richiesti frequentemente
3. Memoria principale (DRAM): meno veloce ma contiene più dati
4. Memoria secondaria o di Massa: molti dati, indicativamente più lenti (anche se gli SSD consentono un vantaggio prestazionale rispetto agli HD o agli HHD).

### 1.2.4) Rappresentazione dell'informazione:

Tutti i dati vengono solitamente rappresentati in maniera **binaria**, ovvero il bit che può prendere il valore 0 o 1. Questo è molto utile perché si basano strutturalmente basati su dispositivi bistabili (corrente ON o corrente OFF). Per questo l'elaboratore elettronico può operare solo su sequenze di simboli binari.

Il BIT (derivato da Bynary digIT) è quindi l'unità elementare dell'informazione. Comandi e dati nel computer vengono quindi rappresentati con lunghe sequenze di numeri binari.

Essendo tutto codificabile attraverso il bit, allora sono state inventate strutture per contenere la rappresentazione figurata della realtà. Ad esempio la convenzione ASCII, poi diventata extended-ASCII, ha codificato i caratteri come sequenza di 8bit, il **byte**.

Carattere 'a' rappresentato da:

$$(97)_{10} = (01100001)_2$$



BYTE = 8 bit

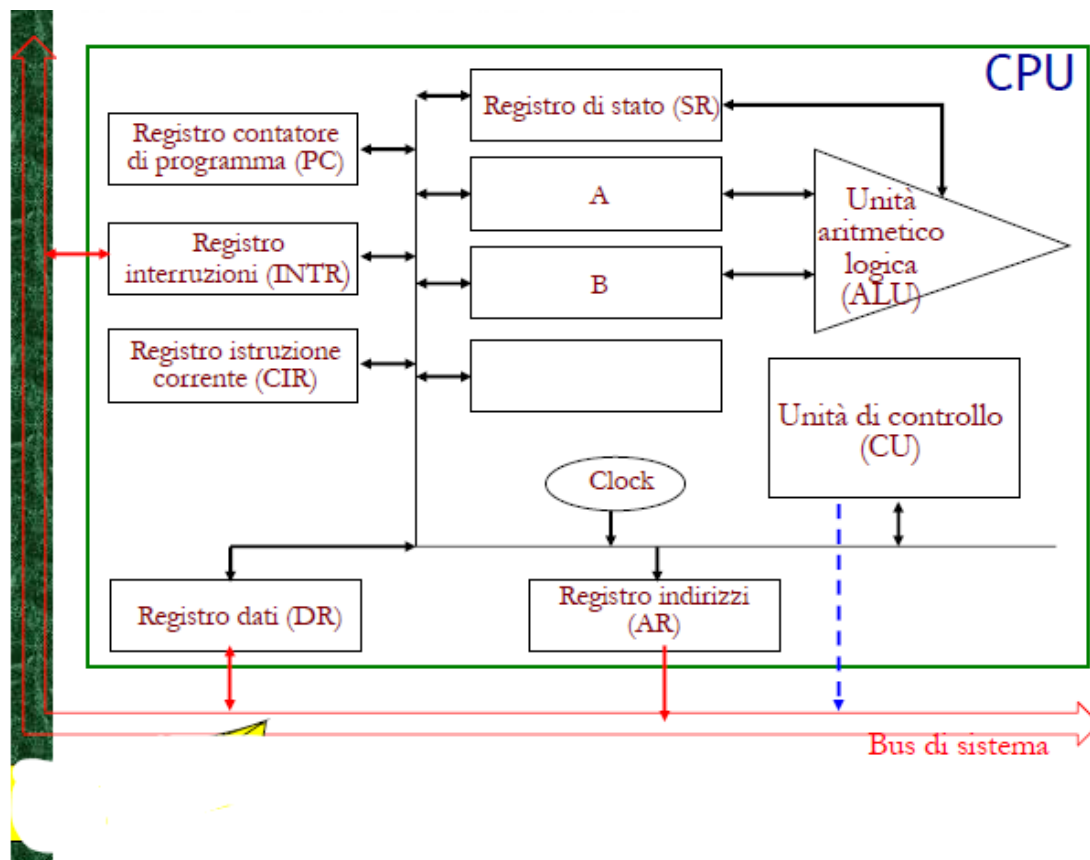
Per i multipli del byte si adottano gli stessi simboli del sistema decimale, ma visto che gli elementi sono in base 2, il fattore di scala è leggermente diverso:

KILOBYTE	KB	$2^{10} = 1024$ byte	$\sim 10^3$ byte
MEGABYTE	MB	$2^{20} = 1024^2 = 1024$ KB	$\sim 10^6$ byte
GIGABYTE	GB	$2^{30} = 1024^3 = 1024$ MB	$\sim 10^9$ byte
TERABYTE	TB	$2^{40} = 1024^4 = 1024$ GB	$\sim 10^{12}$ byte
PETABYTE	PB	$2^{50} = 1024^5 = 1024$ TB	$\sim 10^{15}$ byte
EXABYTE	EB	$2^{60} = 1024^6 = 1024$ PB	$\sim 10^{18}$ byte

#### 1.2.5) Funzionamento della CPU:

Innanzitutto bisogna dire che:

1. il trasferimento dei dati avviene con il bus di sistema.
2. Le fasi di elaborazione si susseguono in modo sincrono rispetto all'orologio di sistema (clock).
3. Durante ogni intervallo di tempo **l'unità di controllo** (che fa parte del processore) stabilisce la funzione da svolgere.
4. L'intera macchina opera in maniera sequenziale (però architetture più evolute prevedono l'esecuzione parallela delle istruzioni).



Come si vede dall'immagine la CPU ha a disposizione vari elementi:

1. i registri (PC, INTR, CIR, DR, AR, ...)
  - registro di stato della CPU (**SR**) (FLAG: C, Z, S, V), serve a far capire al compilatore lo stato delle periferiche e delle varie parti del compilatore stesso.
  - Il program counter (**PC**) indica gli indirizzi dove andare a prendere la prossima istruzione.
  - registro istruzioni (**INTR**) segnala lo stato di funzionamento delle periferiche.
2. unità di controllo: capacità di controllare quello che sta succedendo, in pratica manda segnali per impartire degli ordini, in particolare possono provenire da questa unità il segnale di prelievo, di decodifica e d'esecuzione dell'istruzione.
3. Unità ALO: oggetto che esegue le operazioni aritmetico-logiche.
4. si nota il BUS di SISTEMA, responsabile di collegarsi ai vari registri e alle varie periferiche. (sotto)

Il Ciclo base di funzionamento della CPU: (visione ad alto livello)

1. FETCH
  1. La CU manda un segnale affinché il PC sia spostato nel AR (cioè la prossima istruzione viene indicizzata).



2. segnale controllo (read) alla memoria centrale posto all'indirizzo in AR.
  3. il dato letto viene messo a disposizione nel DR (registro dei dati).
  4. la CU manda il segnale di controllo affinché il contenuto di DR sia spostato nel CIR (registro istruzione corrente).
2. INTERPRETAZIONE: le informazioni sul CIR vengono decodificate dalla CU.
  3. ESECUZIONE: la CU genera una sequenza di segnali di controllo necessari a eseguire l'istruzione.
  4. il PC viene incrementato per puntare alla prossima istruzione.

Durante l'esecuzione la CPU può eseguire 3 macro-tipologie di istruzioni:

1. Istruzioni aritmetiche.
2. istruzioni di controllo.
3. istruzione di trasferimento dei dati (sia registro-registro, sia memoria-memoria, sia memoria-registro, sia registro-memoria).

#### 1.2.6) Il Bus di Sistema

Il bus di sistema è l'elemento che interconnette le varie periferiche e i vari elementi del calcolatore. In ogni istante il bus è dedicato al collegare due unità, una che trasmette e una che riceve. Il processore esegue il *bus mastering*, ovvero seleziona le connessioni da attivare e indica l'operazione da svolgere. Il bus è suddiviso in tre insiemi di linee: *bus dati*, *bus indirizzi* e *linee di controllo* (quest'ultime trasportano informazioni relative alla modalità di trasferimento e alla temporizzazione).

Il Bus di sistema ha l'organizzazione della comunicazione cosiddetta *master/slave*, ovvero ci sono parti (*slave*) che devono sempre ascoltarne altre (*master*). Generalmente il master è l'elemento di controllo, nonché quello che manda i segnali di esecuzione.

#### 1.2.7) Le periferiche: memorie di massa

Con il termine memoria di massa ci si riferisce a un dispositivo di memorizzazione permanente capace di contenere grosse quantità di dati.

Possono essere: fissi o rimovibili, ad accesso sequenziale o casuale, dispositivi in sola lettura (RO), in lettura scrittura (RW), o WORM (Write Once Read Many), dispositivi magnetici, ottici o magnetico-ottici.

##### 1. HARD DISK

Sfruttano le proprietà magnetiche di alcuni materiali (sostanze ferromagnetiche) di poter assumere a comando una certa direzione di magnetizzazione. A ciascuna direzione associa un simbolo binario. Sono costituiti da micro-celle magnetizzabili indipendentemente. La magnetizzazione è semipermanente, cioè rimane anche in assenza di mancanza di corrente ma può essere modificata.

## 2. MEMORIE DI MASSA: HDD

Usa uno schema di memorizzazione creato in fase di formattazione di basso livello. Ogni superficie è divisa in tracce concentriche. I dati sono memorizzati in maniera sequenziale. Ogni traccia è divisa in settori. L'insieme delle tracce omologhe poste su diverse facce è detto cilindro.

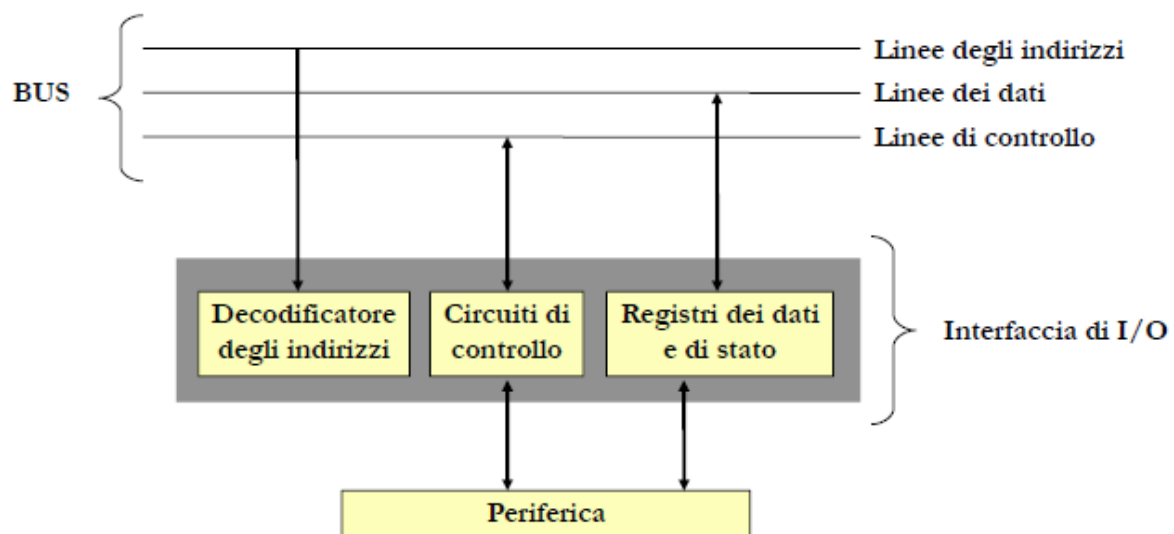
## 3. SSD: MEMORIE ALLO STATO SOLIDI

Accesso completamente elettronico (invece che elettromagnetico) ai dati. Al momento costi maggiori per unità di memoria. Permettono uno start-up immediato, latenza molto bassa e velocità di trasferimento circa un ordine di grandezza maggiore rispetto agli hard disk.

## 4. RAID

Per aumentare le prestazioni dei sistemi a disco è possibile raggruppare più dischi in un sistema RAID (Redundant Array of Inexpensive Disk). Questo sistema suddivide i file in blocchi registrati su dischi diversi per aumentare le prestazioni (data striping). Questo sistema è anche utilizzato per incrementare l'affidabilità dei sistemi a disco attraverso un meccanismo di ridondanza.

### 1.2.8) L'interfaccia delle Periferiche



Comunicare con le periferiche può essere complicato, infatti se si dovesse avere del codice per parlare con ogni singola periferica il lavoro del programmatore diventerebbe impossibile. Per questo esiste un'interfaccia, che fa come da traduttore tra il linguaggio della periferica e la macchina.

E' possibile avere un'interfaccia diversa per ogni periferica, ma è più logico avere delle interfacce standard per periferiche simili. (Ad es. USB: Universal Serial Bus).

Questa interfaccia si occupa concettualmente della gestione e dello scambio di dati tra il processore e le periferiche. In generale contiene:

1. un registro dati della periferica
2. un registro di comando della periferica
3. un registro di stato (questo talvolta collegato al registro delle interruzioni del processore)

A seconda del processore e dei registri delle periferiche, le interfacce possono:

1. condividere lo spazio di indirizzi della memoria (memory mapped I/O)
2. adottare uno spazio di istruzioni/indirizzi distinti (port mapped I/O)

Le periferiche attraverso le interfacce possono essere gestite con due metodi principalmente:

1. POLLING: il processore invia sul bus il comando di lettura e si mette in attesa che il dato sia disponibile sul registro della periferica attraverso continui cicli. (PRO: facile implementazione e gestione; CONTRO: tiene sospeso il processore nel ciclo di attesa del dato).
2. INTERRUPT: il processore invia il comando di lettura alla periferica e poi continua le sue operazioni. Quando il dato è disponibile sul registro della periferica, la periferica stessa “solleva” un *interrupt*, un'interruzione. Il processore così interrompe le sue operazioni, salva il proprio stato ed esegue una opportuna routine per la gestione delle interruzione (compito del sistema operativo). Questa routine serve a verificare la presenza del dato sulla periferica del dato e a iniziare il trasferimento nel registro interni al processore, fino ad arrivare in memoria. Alla fine dell'*interrupt* il processore ritorna alle sue operazioni normali. (PRO: lascia più libero il processore di operare; CONTRO: gestione, implementazione e controllo più complicati).

Tra le periferiche sono presenti anche i terminali, cioè qualunque dispositivo di puntamento (tastiera, mouse, video ...), e le stampanti.

### 1.3) Architettura software di un compilatore:

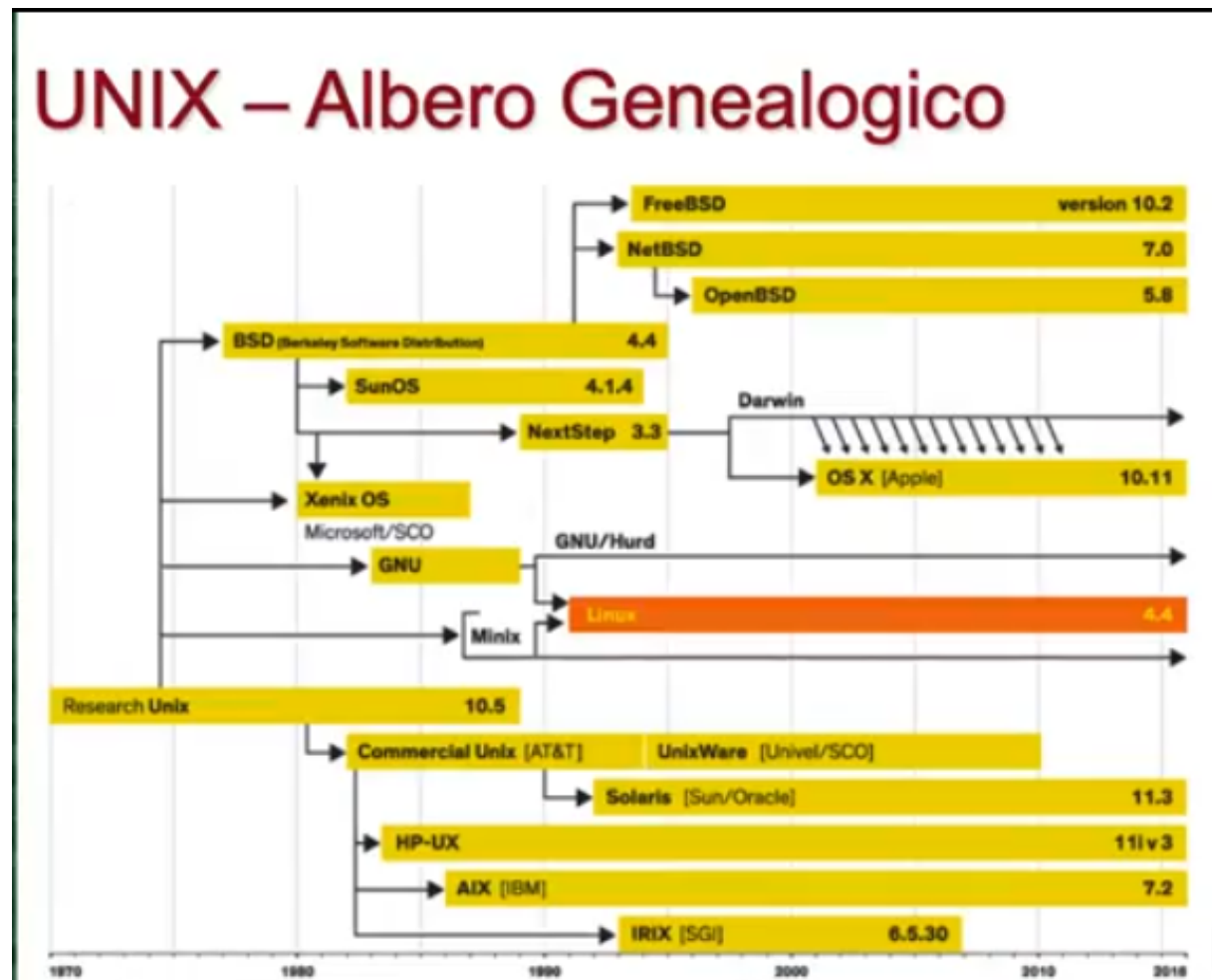


Con il termine *sistema operativo* intendiamo l'insieme di programmi che opera direttamente sulla macchina fisica, fornendo interfacce di alto livello e mascherandone le caratteristiche specifiche delle periferiche e del processore. Questo ad esempio è molto utile nella gestione delle periferiche, perché permette di fornire un metodo *consistente* delle periferiche, ovvero una modalità standard di interfacciarsi con le periferiche disponibili senza dover eseguire comandi di basso livello. Importante notare inoltre che il sistema operativo è l'unico elemento con accesso diretto alle risorse hardware, e si riesce ad accedervi in altro modo direttamente probabilmente abbiamo un problema.

Il sistema operativo è il modo che consente ai programmi di ottenere risorse dal calcolatore. La traduzione da programma a calcolatore avviene ad alto livello, in modo che non si debba scontrarsi con la gestione degli indirizzi o dei registri. Il sistema operativo quindi, dovendo gestire tutte le funzionalità di basso livello attraverso dei controlli di alto livello, opera un alto livello di astrazione del linguaggio macchina.

**Storia dei sistemi operativi:** Nel 1982 Kernigham introduce Unix, prodotto dai laboratori AT&T, che al tempo avevano grandi esigenze di un ecosistema stabile e uguale su cui sviluppare applicazioni. Caratteristica principale: poteva servire più utenti contemporaneamente, usando il multitasking. Tutti gli utenti infatti hanno un ordine di esecuzione al processore, ma sono tutti nell'ordine del

proprio utente. Da Unix poi si svilupperanno tutti i sistemi operativi che tuttora conosciamo (Come MS-DOS o Linux).



Architettura di un SO, che ad oggi è organizzato secondo un architettura a *strati* (anche detta *onion skin architecture*). Ogni strato fornisce un'astrazione dello strato su cui si appoggia e permette una chiara separazione tra interfaccia e implementazione delle diverse funzionalità, oltre che a fornire l'insieme di programmi e librerie.



Il *kernel* del sistema operativo, ovvero il cuore dell'OS si occupa di:

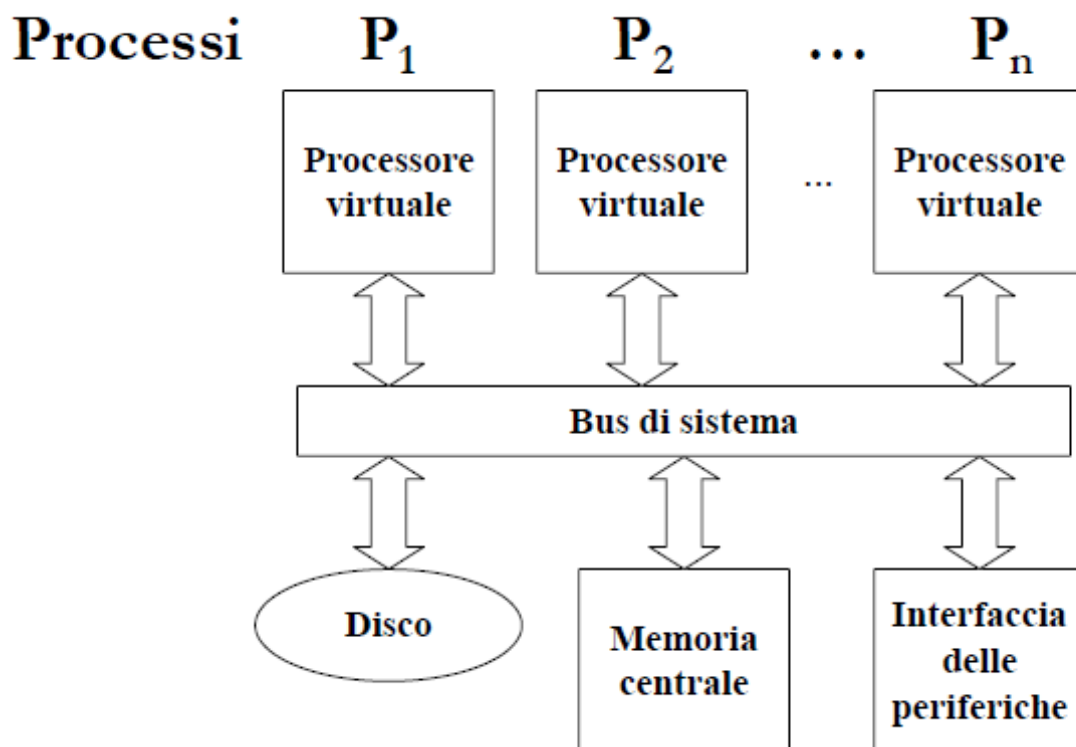
- gestione periferiche (I/O informazioni)
- gestione memoria (Fornisce/non fornisce alla memoria in base al carico del momento, stabilisce il tipo di memoria da fornire)
- gestione dei processi

Un processo è un'entità dinamica, contrariamente al programma, infatti viene causato dal codice in esecuzione (*programma*) e dal suo stato di esecuzione (es. valore delle sue variabili).

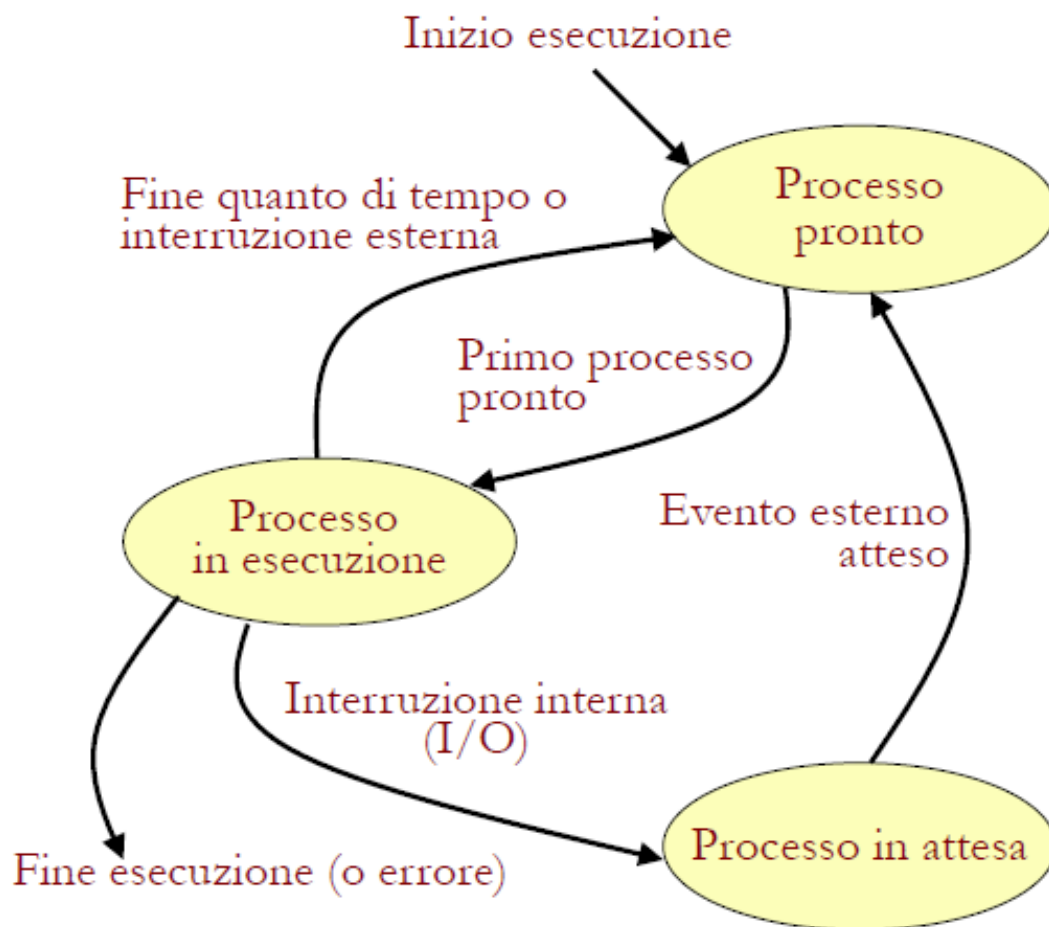
Un processo è quindi un insieme di due elementi:

- E: il codice eseguibile del programma (questo fa partire il fetch per creare un processo)
- S: lo stato del processo

Logicamente il processo non viene gestito direttamente dal processore reale, se no verrebbe meno il concetto di multitasking. Il processore crea dei processori virtuali che si comportano similmente e che possono essere assegnati a un processo.



Nel processore è possibile che ci sia un solo processo in esecuzione in ogni istante, mentre gli altri processi sono pronti o in attesa. Ad ogni processo viene assegnato un valore massimo di tempo di esecuzione, scaduto tale viene revocato il processore virtuale e assegnato a un altro processo. Questa tecnica è detta di time-sharing, viene eseguito dal nucleo che decide da quali processi devono andare in esecuzione determinando lo *scheduling*, che è sequenziale. La soluzione tipica per la gestione del tempo di esecuzione di processi è a turno (*round-robin*, tutti i processi in questo hanno la stessa priorità ad essere eseguiti, nella realtà non funziona perché alcuni processi hanno la priorità). La CPU viene rilasciata anche quando un processo sta aspettando un I/O da/verso una periferica.



x

In questa complicata gestione delle risorse prende parte anche la gestione della memoria, che pensa al partizionamento della memoria tra i vari processi che la richiedono garantendo la protezione/separazione fra le diverse zone allocate. Il gestore memoria gestisce anche la memoria che va assegnata ai processi, e quando i processi finiscono la memoria fisica si occupa di creare una memoria virtuale (più lenta di solito, chiamata SWAP o semplicemente memoria virtuale) e assegnare quella memoria.

### THE C PROGRAMMING LANGUAGE:

Ci sono principalmente 3 tipi di programmazione:

- programmazione per **HARDWARE**, ovvero la programmazione di dispositivi fisico/logici il cui il linguaggio di programmazione coincide con il linguaggio macchina.
- programmazione per **FIRMWARE**, ovvero la programmazione che usa un'insieme comune di istruzioni mediante assembly, ovvero linguaggio di programmazione molto basso



- programmazione per SOFTWARE, ovvero programmare in un linguaggio intermedio che simula le funzionalità della macchina fisica, che per questo permette maggiore flessibilità (nel senso che posso realizzare un gran numero di programmi diversi e con funzioni diverse), ma che ha un utilizzo delle risorse meno efficiente.

Il sistema operativo prende parte molto nell'ultimo caso, infatti è suo il compito di fornirci le periferiche necessarie a fornire la traduzione da linguaggio macchina (che usano ad esempio nella programmazione firmware) e il codice. L'OS si occupa anche dell'astrazione degli oggetti e/o delle istruzioni complesse di alto livello.

### **Diffusione dei linguaggi e Perché il C**

Ci sono tantissimi linguaggi di programmazione creati o utilizzati per specifici utilizzi in cui sono molto apprezzati. Linguaggi più standard di medio-basso livello come il C o il C++ permettono una maggior comprensione del funzionamento della maggior parte degli altri linguaggi, che in alcuni casi forniscono astrazioni alle strutture di basso livello ancora presenti in questi.

Il C presenta una serie di elementi che lo rendono importante da imparare:

- permette l'allocazione dinamica e altri aspetti normalmente di alto livello, a un livello basso.
- gestione della memoria molto manuale
- linguaggio apprezzato/richiesto dalle aziende
- ha un astrazione che è posizionata tra il medio e il basso livello, motivo per cui è molto utile per ad. es. embedded system.

Il linguaggio C è stato creato nel 1972 da Kernighan e Ritchie ai Bell Tel. Labs.

#### **1.1) Operazioni Logiche (algebra di Boole)**

L'algebra di Boole è basata su tre operatori logici (NOT, AND, OR). Gli operandi possono assumere due valori: VERO e FALSO.

Gli operatori godono della proprietà

- commutativa (es.  $A \text{ OR } B = B \text{ OR } A$ )
- distributiva (es.  $A \text{ AND } (B \text{ OR } C) = (A \text{ AND } B) \text{ OR } (A \text{ AND } C)$ )

Le tabelle di verità associano a tutti i possibili valori degli operandi il risultato

# NOT(A) !A

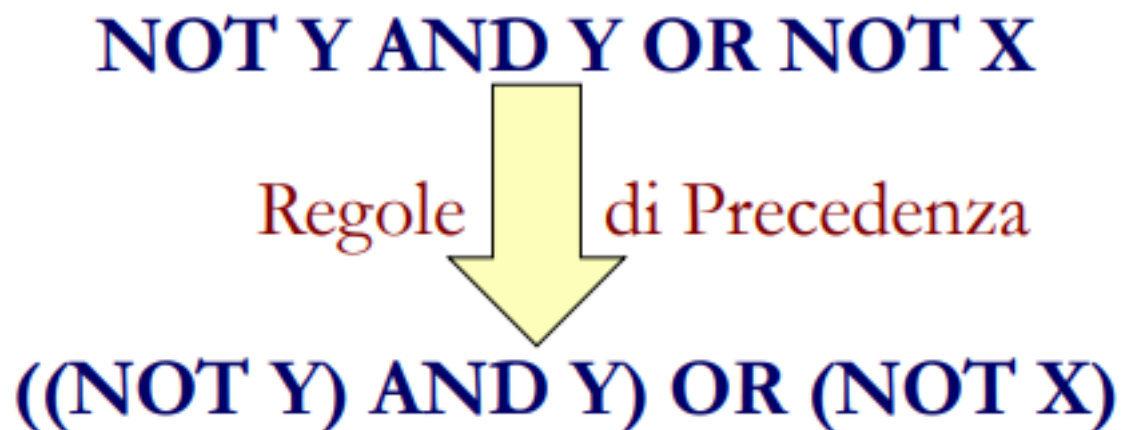
<u>A</u>	<u>NOT A</u>
0	1
1	0

<u>A</u>	<u>B</u>	<u>A AND B</u>
0	0	0
0	1	0
1	0	0
1	1	1

<u>A</u>	<u>B</u>	<u>A OR B</u>
0	0	0
0	1	1
1	0	1
1	1	1

solitamente NOT viene rappresentata con !, AND con && e OR con ||.

quando si valuta un'espressione l'ordine è il seguente (NOT, AND, OR), per esempio.

**Figure 0.1:** image-20201207220627893

Per calcolare il risultato di un'espressione (per esempio NOT Y AND (Y OR NOT X)) si crea una tabella di verità, prima con i valori singoli per poi raggrupparli fino ad arrivare alla formula di partenza.

X	Y	NOT X	NOT Y	Y OR NOT X	NOT Y AND (Y OR NOT X)
1	0	0	1	0	0
0	1	1	0	1	0
0	0	1	1	1	1
1	1	0	0	1	0

### Leggi di De Morgan

- $A \text{ AND } B = \text{NOT } ((\text{NOT } A) \text{ OR } (\text{NOT } B))$
- $A \text{ OR } B = \text{NOT } ((\text{NOT } A) \text{ AND } (\text{NOT } B))$

si possono dimostrare compilando la tabella di verità e osservando che le tabelle delle espressioni ai due lati dell'uguale hanno stessi risultati a parità di input.

**Tautologia:** espressione che è sempre vera

**Contraddizione:** espressione sempre falsa

## 1.2) Codifica Semplici Algoritmi in C

istruzione di assegnamento:

```
1 x = 23;  
2 w = 'a';  
3 y = z;
```

**Costruttore if-else** diagramma di flusso:

L'espressione tra parentesi viene valutata, se vera viene eseguito il primo blocco, se falsa l'altro. Naturalmente si può utilizzare anche l' if da solo senza else. All'interno dell'espressione posso utilizzare gli operatori logici (&&, ||).

è sempre meglio utilizzare indentazione e parentesi graffe per una migliore leggibilità e per non commettere errori.

**Operatore ternario ?** è un altro modo di scrivere if-else, la sintassi è la seguente

```
1 espressione1 ? espressione2 : espressione3; //questo equivale al  
   seguente if-else  
2  
3 if (espressione1)  
4     { espressione2; }  
5 else  
6     { espressione3; }
```

dopo le parentesi graffe il ; non è necessario, ma se viene messo non c'è errore.

**Precedenza degli operatori** In un'espressione vengono eseguiti prima gli operatori con precedenza superiore, se gli operatori sono dello stesso gruppo si usano le regole di associatività (da destra o da sinistra), le parentesi possono essere usate per modificare la precedenza.

Associatività da sinistra a destra significa che a parità di priorità l'espressione viene eseguita partendo da sinistra a destra.

```
1 if (a + b - 4 <= 9 && x < tot -1 ) // questa espressione é equivalente  
   a quella sotto  
2  
3 if (((a + b - 4) <= 9) && (x < tot -1) )
```

**Istruzione Iterativa ( ciclo )** il diagramma di flusso è il seguente (il ciclo si chiama while). Il blocco istruzioni viene ripetuto fino a quando l'espressione non diventa falsa.

(da pagina 38 a 56 un po' di esercizi noiosi)

#### Getchar e Putchar

```
1 //getchar legge il prossimo carattere inserito da tastiera
2 c = getchar();
3 //putchar stampa il carattere nello standard output
4 putchar(c)
```

**Esercizio scale:** Sia data una scala di N gradini. Si supponga di salire l'intera scala con passi da 1, 2 o 3 scalini. In quanti possibili modi si può salire l'intera scala ?

#### risoluzione in modo semplice:

Devo capire in quanti modi possibili posso salire una scala con n gradini. Posso compiere passi da 1, 2 o 3 gradini.

Mi calcolo in quanti modi posso salire una scala formata da 1 2 o 3 gradini e basta.

Un gradino = 1 modo

Due gradino = 2 modo

Tre gradino = 4 modo

Per arrivare al quarto gradino ho solo tre possibilità: fare un passo da uno, da due o da tre, quindi devo arrivare al gradino 4-1, 4-2, 4-3, siccome so già quanti passi ci vogliono per arrivare in questi posti, basta sommarli assieme per capire il numero di passi per arrivare al quarto.

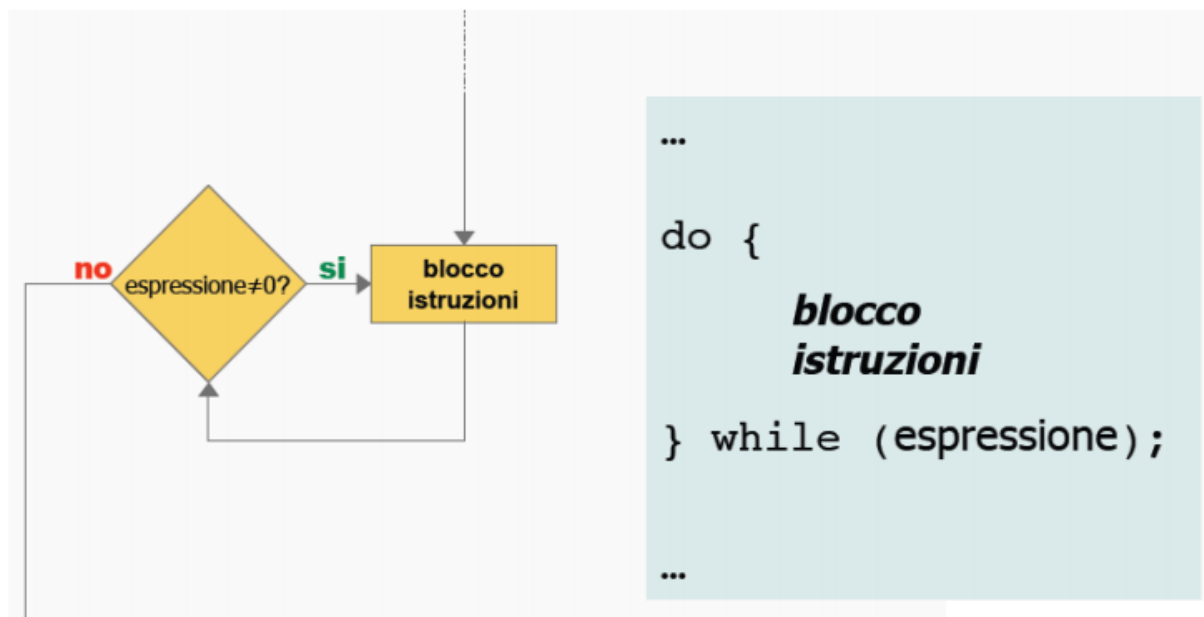
Aggiorno il numero di passi per il gradino n-1, n-2, n-3 e vado avanti. (se non si capisce chiedetemi che vi spiego meglio).

### 1.3) Strutture di controllo

**Istruzione di ciclo: FOR** schema a blocchi e sintassi:

questo operatore è utile quando so a priori quante operazioni devo fare, in quei casi è più compatto rispetto ad un while.

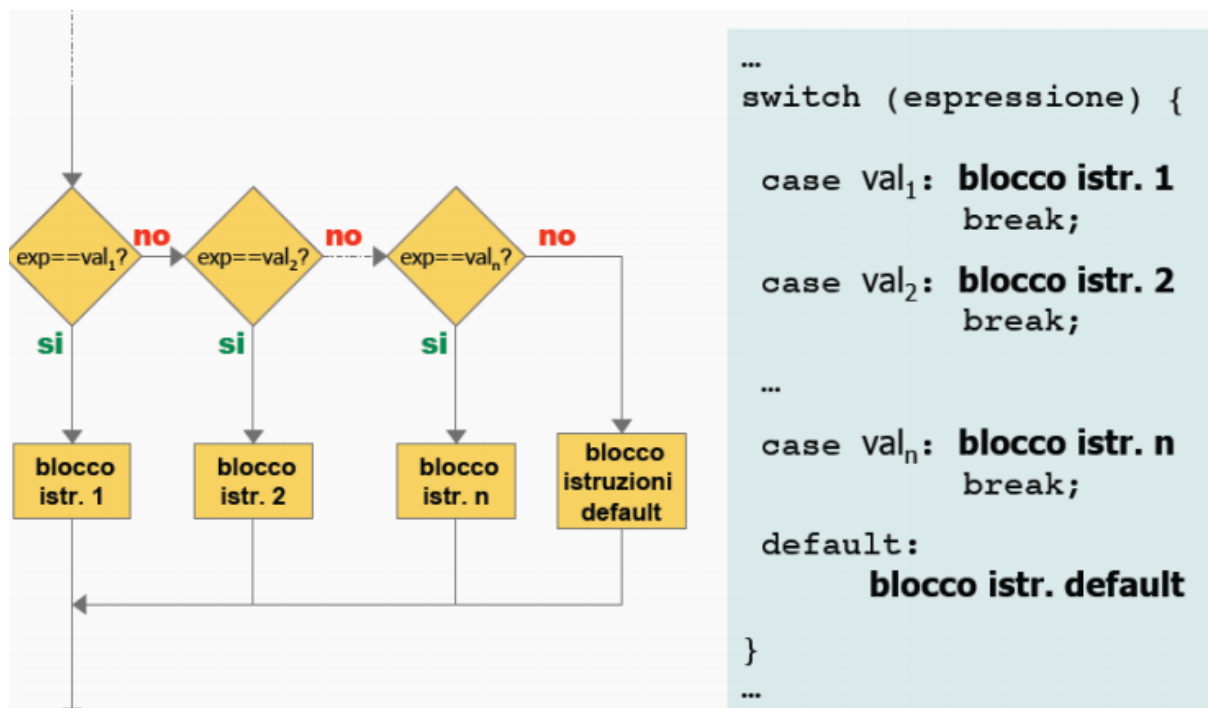
**Il costrutto DO-WHILE** schema a blocchi e sintassi:



```
1 // un esempio  
2 Contatore = 0;  
3 do  
4 {  
5     scanf (" %c", &Dato);  
6     Contatore ++;  
7 } while (Dato != '%');
```

si utilizza quando voglio eseguire un blocco di istruzioni almeno una volta, può essere utile quando devo fare un controllo sull'input da tastiera.

**Il costrutto SWITH** Va a sostituire un if-else multiplo, schema a blocchi e sintassi:



Il **default** (le istruzioni che vengono eseguite in caso che nessuna delle altre sia vera) è opzionale. I singoli case vengono eseguiti quando il valore dell'espressione è uguale a quello scritto appena dopo l'istruzione case. ATTENZIONE: bisogna mettere il break dopo il blocco istruzioni altrimenti si rimane all'interno dello switch (verranno valutati i case seguenti ed eseguito il default se presente).

Valuta solo variabili di tipo INT, quindi l'espressione deve avere come risultato un int.

**Break:** quando viene eseguito all'interno di un while, for, do, switch provoca l'uscita dall'istruzione

**Continue:** quando viene eseguito all'interno di un ciclo passa alla iterazione successiva

un linguaggio di programmazione può codificare qualsiasi algoritmo se ha: sequenza di istruzioni, if-else e while.

## 2.1) Array in C

Gli array possono essere paragonati a vettori e matrici in matematica. Da un punto di vista più concreto sono una sequenza di celle di memoria consecutive e omogenee. L'array è quindi un contenitore per *variabili dello stesso tipo*.

A ciascun elemento dell'array si accede tramite indice (esempio `a[i]` è l'elemento alla posizione i-esima). Le parentesi quadre sono operatori ad alta precedenza (sono al primo livello della tabella).


Il primo elemento dell'array è quello in posizione 0, la macchina astratta prende l'indice e lo somma all'indirizzo della prima cella dell'array.

Prima di utilizzare le array bisogna dichiararle:

```
1 int a[100]; // dichiara un contenitore a (array) che potrà contenere
              100 elementi di tipo int, il primo elemento lo si trova in a[0], l'
              ultimo in a[99]
```

il compilatore va a riservare la memoria per tutti questi elementi

Elemento	a[0]	a[1]	a[2]	a[3]	...
Indirizzo	1000	1001	1002	1003	...



**Indici che spaziano da 0 a 99**

In generale l'ultimo elemento dell'array è nella posizione n-1, dove n è la lunghezza dell'array stessa. All'interno delle parentesi si possono mettere delle espressioni. Un veloce esercizio:

SOLUZIONE: a[0]= indeterminato, a[1] = 6

ATTENZIONE: se vado oltre l'indice massimo dell'array accedo a celle di memoria che non appartengono all'array e il cui valore è indeterminato.

L'array in C non è un tipo, ma un costruttore di tipo.

**Inizializzazione e stampa** si può inizializzare direttamente al momento della dichiarazione

```
1 int a[5] = {5, 2, -5, 10, 234};
2 int b[4] = {5, 2, -5}; //un elemento non é inizializzato
3 int c[2] = {5, 2, -5}; // ERRORE: inizializzato un elemento che non
                          appartiene all'array
```

per array grandi questo metodo diventa scomodo, quindi si usano i cicli per inizializzare.

anche per stampare un array devo utilizzare un ciclo

```
1 printf("%d", a); // errato perché a é un array
2
3 int i=0; // questo é il procedimento corretto
4 while (i<5){
5     printf("%d",a[i]);
6     i++;
7 }
```



esercizi sulle array dalla slide 31 a 41.

**Array dinamici:** il C permette inizializzare la dimensione di un array durante l'esecuzione di un programma (per esempio chiedendo la dimensione da tastiera).

**Array multidimensionali** Le array di due dimensioni corrispondono alle matrici in matematica. Si dichiarano nel seguente modo:

```
1 int a[N][M]; //N numero righe M numero colonne
2
3 //é anche possibile definire più dimensioni
4 int a[10][5][20];
```

Come per gli array ad 1 dimensione, anche questi possono essere inizializzati nella fase di dichiarazione:

```
1 int a[4][5]= { {2, 5, -8, 7, 6},
2               {3, 10, 7, 6, 1},
3               {-1,8, -8, 5, 3},
4               {2, 5, 8, 4, 2}
5               };
```

Per semplicità possiamo immaginare l'array in due o più dimensioni, ma la macchina astratta del C memorizza gli elementi uno dietro l'altro. Per esempio l'array creato sopra verrà memorizzata nel seguente modo:

a <sub>00</sub>	a <sub>01</sub>	a <sub>02</sub>	a <sub>03</sub>	a <sub>04</sub>	a <sub>10</sub>	a <sub>11</sub>	a <sub>12</sub>	a <sub>13</sub>	a <sub>14</sub>	a <sub>20</sub>	a <sub>21</sub>	a <sub>22</sub>	a <sub>23</sub>	a <sub>24</sub>	a <sub>30</sub>	a <sub>31</sub>	a <sub>32</sub>	a <sub>33</sub>	a <sub>34</sub>
2	5	-8	7	6	3	10	7	6	1	-1	8	-8	5	3	2	5	8	4	2

altri esempi di inizializzazione corretta e sbagliata:

```
1 int D[][]={1,2,3,4}; // errata
2 int E[2][]={1,2,3,4}; //errore non viene specificato il numero di
   colonne
3
4 int F[][4]={1,2,3,4}; //va bene
5 // in c nella dichiarazione di un array bisogna valorizzare tutte le
   dimensioni, si può fare a meno di quella più a sinistra
```

le array possono anche essere inizializzate con dei cicli

```
1 int main(int argc, char *argv[]){
2     int matrice[10][5];
3     int i=0,j=0;
4     while (i<10)
```

```

5      {
6          j=0;
7          while (j<5)
8          {
9              printf("%d ",matrice[i][j]);
10             j++;
11         }
12         printf("\n");
13         i++;
14     }
15 }

```

esercizi da pagina 61 in poi

## 2.2) Stringhe in C

Un array di char può essere rappresentata con una stringa (per esempio "hello"). L'ultimo carattere deve essere il carattere nullo '\0'. Questo carattere serve alle varie funzioni per capire dove terminerà la stringa. Quindi quando vado a creare una stringa per memorizzare n caratteri ne serviranno n+1 (uno lo uso per il carattere nullo).

Esiste un modo semplificato per inizializzare un'array di caratteri come stringa:

```
1 char mia_stringa[] = "Ciao a tutti!";
```

Questo mi memorizza automaticamente lo spazio per i miei caratteri più il carattere terminatore. Quindi il risultato sarà:

Codice ASCII	C	i	a	o		a		t	u	t	t	i	!	
	67	105	97	111	32	97	32	116	117	116	116	105	33	0

**Figure 0.2:** image-20201212103116117

L'inizializzazione vista sopra è molto più veloce ed è equivalente ad inizializzare nel seguente modo:

```
1 char mia_stringa[] = {'C','i','a','o', ' ','a', ' ','t','u','t','t','i','i',
    ' ','!', '\0'};
```

Se non specifico il numero all'interno delle parentesi quadre quando dichiaro l'array il compilatore va a riservare uno spazio pari al numero degli elementi con cui l'array viene inizializzato. Nel caso delle stringhe posso anche dichiarare esplicitamente la dimensione di memoria da riservare:

```
1 char frase[20]="Ciao a tutti!";
```

Bisogna stare attenti che un elemento (dei 20 messi a disposizione per l'array) sarà occupato da '\0' e poi gli elementi in più saranno lasciati vuoti.

ATTENZIONE: se non specifico né il numero di caratteri (all'interno delle parentesi quadre) né assegno alla stringa un valore, il compilatore dà un errore, perché non sa quanta memoria riservare.

```
1 char parola[]; // ERRORE
```

per stampa le stringhe si usa %s:

```
1 printf("%s", mia_stringa); //questo non sarebbe possibile se non ci
   fosse il carattere terminatore perché non saprei dove fermarmi
```

quando faccio scanf non bisogna mettere la & perché la stringa è un array e la variabile con il suo nome è già un indirizzo.

```
1 scanf("%s", parola);
```

## 2.4) Rappresentazione di informazioni

in un calcolatore le informazioni vengono rappresentate sotto forma di dati, codificati in un linguaggio comprensibile al calcolatore. Per permetterci di interpretare le informazioni i dati devono essere decodificati. Quindi ci sono diversi livelli di decodifica che partono dall'hardware fino ad arrivare ad informazioni interpretabili a noi umani.

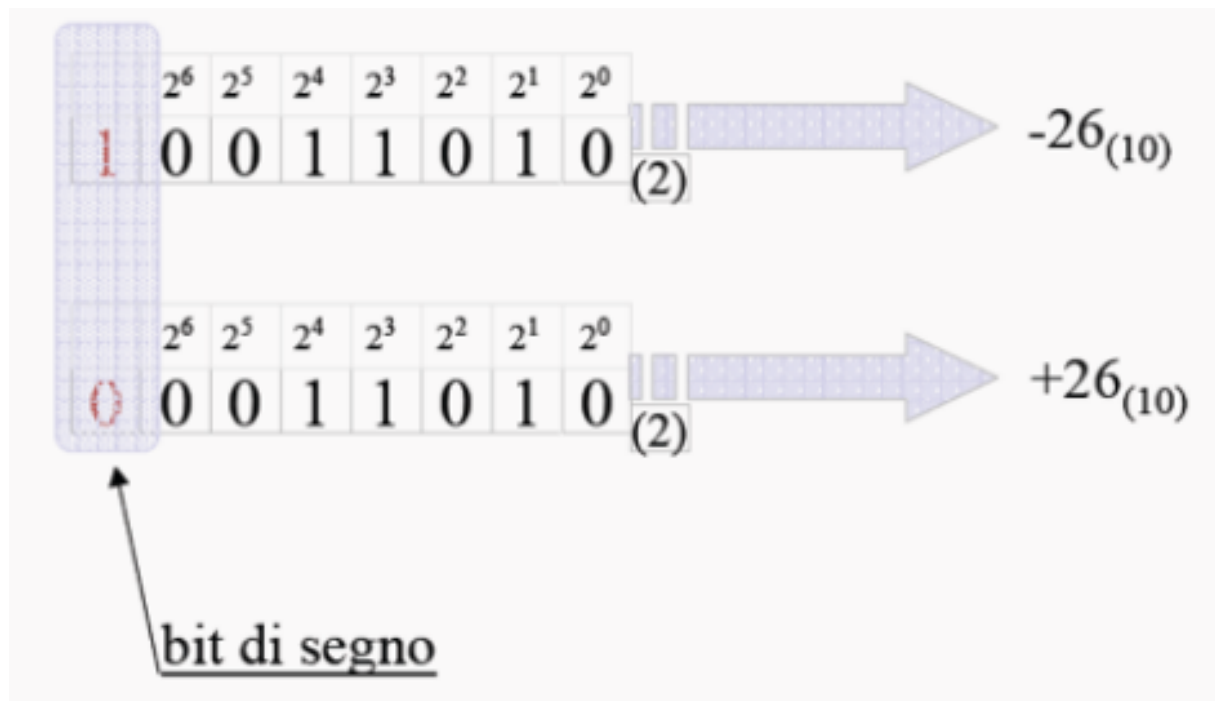
I tipi di dato che il calcolatore può interpretare direttamente sono :

- booleani
- numeri interi
- numeri frazionari
- caratteri

Per questi dati la codifica è gestita direttamente dall'HW, per tipi di dato più complessi si usa una rappresentazione di tipo software.

**Interi** Sono rappresentati da una sequenza finita di bit. 8 bit (un byte) permettono di rappresentare i valori da 0 a 255. Solitamente per gli interi positivi si usano 4 byte (32 bit), quindi i numeri vanno da 0 a 4.294.967.295. (questo implica che all'interno dei calcolatori i numeri sono finiti).

Per rappresentare anche i numeri negativi, si utilizza il primo bit come bit di segno (0 per i numeri negativi, 1 per i positivi)



In realtà nei calcolatori non si usa questa rappresentazione ma quella in complemento a due con i seguenti vantaggi: non c'è un doppio zero, non c'è bisogno di una circuiteria specifica. Esempio:

dec	binario
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

Per decodificare i valori positivi si procede nel modo normale, per quelli negativi si decodifica e poi si sottrae  $2^N - 1$ . Per invertire i numeri si invertono gli zeri con gli uno e si somma uno.

**Numeri frazionari** I dati con numeri dopo la virgola vanno rappresentati in maniera opportuna, ci sono due tecniche:

- **virgola fissa:** si dividono i bit che rappresentano i valori interi da quello per i valori dopo la virgola
- **virgola mobile:** la maggior parte dei bit viene usata per le cifre rappresentative del numero, gli altri per sapere dove mettere la virgola.

esempio codifica virgola fissa:

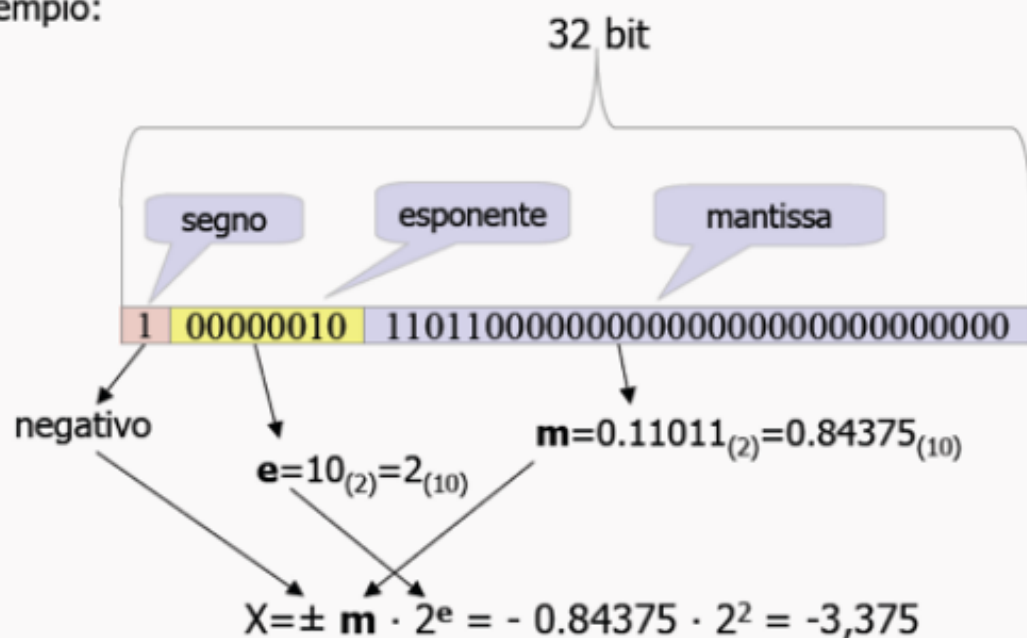
Si ha ad esempio  $1001.1010_{(2)} = 9.625_{(10)}$  secondo la codifica:

$2^3$	$2^2$	$2^1$	$2^0$	$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$
8	4	2	1	0.5	0.25	0.125	0.0625
1	0	0	1	1	0	1	0

(2)

Per la virgola mobile solitamente vengono utilizzati 32 bit, 1 per il segno, 8 per l'esponente e il resto per la mantissa

Ad esempio:



Quindi la mantissa rappresenta numeri da 0 a 1, che verranno moltiplicati per  $2^e$  in modo da ottenere il numero desiderato.

$$m = 0.11011_{(2)} = 2^{-1} + 2^{-2} + 2^{-4} + 2^{-5} = 0.84375$$

un numero si dice normalizzato se l'esponente è diverso da 0, la mantissa è compresa tra 1 e 2 l'intervallo dei numeri è

$$(-2^{128}, -2^{-126}] [2^{-126}, 2^{128})$$

**Caratteri** per codificare i caratteri si utilizza la tabella ASCII, i primi 128 valori sono fissi i successivi rappresentano la tabella ASCII estesa con caratteri più specifici (per esempio c'è una tabella ASCII estesa con i caratteri è, ò, à...).

attualmente si utilizza l'UNICODE che utilizza 2 bytes per ogni carattere e permette di non avere tabelle diverse per ogni regione del mondo.