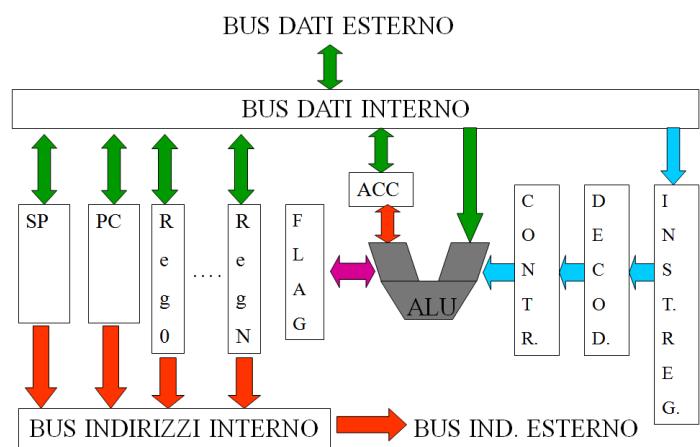
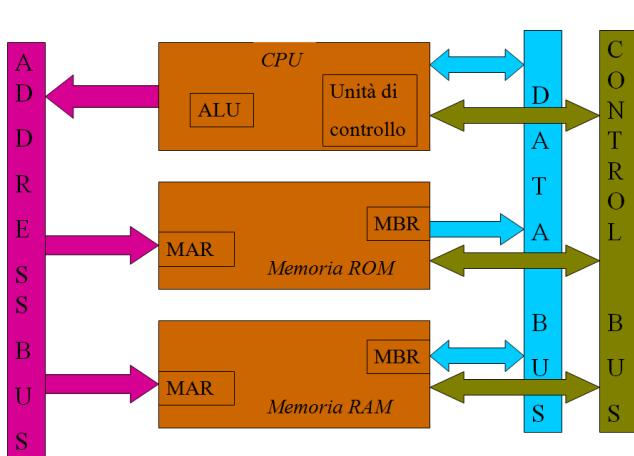


*Fondamenti di Informatica*

## Architettura di una CPU



Il bus dati (data bus) esprime la capacità di elaborazione del processore (quanti bit possono essere elaborati in parallelo)

Il bus indirizzi (address bus) esprime la capacità di memorizzazione del processore ( $2^m$  celle di memoria, se  $m$  è il numero dei bit del bus)

La capacità di indirizzamento indica il numero di celle diverse cui si può accedere.

Stack: area di memoria gestita con logica LIFO (Last In First Out)

MAR = Memory Address Register; MBR = Memory Buffer Register

La memoria (ROM e RAM) contiene il programma e i dati sui quali opera la CPU. Il Program Counter (PC) contiene l'indirizzo della cella di memoria con la prossima istruzione da eseguire.

## Esecuzione delle istruzioni

Codice Operativo	Operando 1	Operando 2	
Codice Operativo	Sorgente	Destinazione	Mod. indirizzamento

**FETCH**: vengono letti i campi che costituiscono l'istruzione:

- 1) (PC) -> MAR
  - 2) ((MAR)) -> MBR; (PC)+1 -> PC
  - 3) (MBR) -> IR

I passi 1, 2, 3 permettono di caricare in IR (Instruction Register) il codice operativo (OP Code) dell'istruzione corrente. Passi analoghi permettono di caricare in opportuni registri della CPU gli operandi presenti nell'istruzione. In tal caso, nel passo 3 la destinazione del dato proveniente dalla memoria non è più IR, ma opportuni registri.

**DECODE:** viene identificata l'istruzione corrente sulla base dell'OP Code

**EXECUTE:** è diversa a seconda del tipo di istruzione. In pratica consiste nell'inviare comandi e dati alle unità interessate.

Esempi:

- 1) Somma tra il contenuto del registro R2 e il contenuto dell'accumulatore. Il risultato va nell'accumulatore

FORMATO: Codice Operativo

FETCH: (come sopra)

ESECUZIONE(R2) + (ACC) -> ACC

- 2) Somma tra il contenuto della cella di memoria il cui indirizzo è specificato nell'istruzione ed il contenuto dell'accumulatore; il risultato va nell'accumulatore

FORMATO: Codice Operativo + Operando

FETCH:	1) (PC) -> MAR	4) (PC) -> MAR
	2) ((MAR)) -> MBR; (PC)+1 -> PC	5) ((MAR)) -> MBR; (PC)+1 -> PC
	3) (MBR) -> IR	6) (MBR) -> Rn

EXECUTE:	1) (Rn) -> MAR	3) (MBR) -> Rn
	2) ((MAR)) -> MBR	4) (Rn)+(ACC) -> ACC

- 3) Saltare all'istruzione che è memorizzata nella cella il cui indirizzo è specificato all'interno dell'istruzione corrente:

FORMATO: Codice Operativo + Operando

FETCH:	1) (PC) -> MAR	4) (PC) -> MAR
	2) ((MAR)) -> MBR; (PC)+1 -> PC	5) ((MAR)) -> MBR; (PC)+1 -> PC
	3) (MBR) -> IR	6) (MBR) -> Rn

EXECUTE: 1) (Rn) -> PC

## ***Tipi di istruzioni della CPU***

- Trasferimento dati
- Aritmetiche su numeri interi
- Logiche
- Rotazioni e shift (scorrimento)
- Controllo programma
- Controllo macchina

Queste sei classi includono le istruzioni elementari essenziali per il funzionamento di una CPU.

Per indicare le istruzioni si usa una rappresentazione simbolica che usa codici mnemonici: si parla di assembly language. Assemblatore è il programma che traduce le istruzioni scritte in assembly language, nelle istruzioni corrispondenti in linguaggio macchina (rappresentazione binaria delle istruzioni).

### Istruzioni di trasferimento dati

Servono a trasferire (ricopiare) dati da memoria a registri e viceversa, da registro a registro, ecc.

LDA <indirizzo>	(INDIRIZZO) → A
STB <indirizzo>	(B) → INDIRIZZO
MOV A B	(A) → B
INP <PORTA> B	(PORTA) → B
OUT A <PORTA>	(A) → PORTA

### Istruzioni aritmetiche

Permettono di eseguire operazioni aritmetiche elementari su numeri interi (binari)

ADD <INDIRIZZO> A	(INDIRIZZO) + (A) → A
ADC <INDIRIZZO> A	(INDIRIZZO) + (A) + (C <sub>y</sub> ) → A
SUB A <INDIRIZZO>	(INDIRIZZO) - (A) → A
SBC B <INDIRIZZO>	(INDIRIZZO) - (B) - (C <sub>y</sub> ) → A

Le istruzioni aritmetiche sono eseguite dalla ALU e producono, oltre al risultato, come effetto collaterale, un vettore di bit scritto dalla ALU stessa nel registro dei flag (F). Tali bit hanno significato diverso uno dall'altro e costituiscono degli indicatori:

- C (carry): c'è stato/non c'è stato un riporto o prestito
- W (overflow): c'è stato/non c'è stato trabocco (complemento a 2)
- Z (zero): il risultato è /non è zero
- N (negative): il risultato è /non è negativo

In alcuni processori le operazioni di moltiplicazione e divisione non sono disponibili: in tal caso devono essere implementate con appositi programmi (routine software).

### Istruzioni logiche

Permettono di eseguire operazioni logiche su stringhe di bit memorizzate nei registri A e B.

AND <INDIRIZZO> A	(INDIRIZZO) • (A) → A
OR <INDIRIZZO> B	(INDIRIZZO) + (B) → B
XOR <INDIRIZZO> A	(INDIRIZZO) ⊕ (A) → A
NOT B	— (B) → A

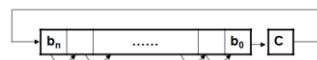
### Istruzioni di shift e rotazione

Operano sul contenuto di un registro

SRL A (shift logico a destra di (A))

SRA A (shift aritmetico a destra di (A))

RR A (rotazione a destra di (A))



SLL B (shift logico a sinistra di (B))

RL A (rotazione a sinistra di (A))

Se il numero che compare nel registro è espresso in complemento a 2, l'operazione di shift aritmetico verso destra equivale ad una divisione per una potenza di 2 (esempio: shift di una posizione = dividere per 2; shift di tre posizioni = dividere per  $2^3 = 8$ , ecc.)

Se il numero è espresso in valore assoluto, la divisione per potenze di 2 può essere effettuata con shift logici verso destra

Discorso duale per le moltiplicazioni per potenze di 2 che possono essere fatte con shift verso sinistra

### Istruzioni di controllo programma

Servono a modificare l'ordine di esecuzione sequenziale delle istruzioni di cui è composto un programma: prendono il nome di istruzione di salto.

I salti possono essere incondizionati (si salta comunque) o condizionati (si salta solo se la condizione del bit del registro dei flag specificato nell'istruzione è verificata); inoltre possono essere senza ritorno all'istruzione successiva a quella di salto oppure con ritorno.

#### Istruzioni di salto senza ritorno

Esempi di salti senza ritorno:

JP <INDIRIZZO> salto incondizionato

INDIRIZZO -> PC

JP <COND><INDIRIZZO> salto condizionato

JR <DISPLACEMENT> salto relativo alla posizione attuale del PC (PC) +  
DISPLACEMENT -> PC

JR <COND><DISPLACEMENT> salto relativo condizionato

DISPLACEMENT (= spiazzamento) rappresentato in complemento a 2: salti avanti o indietro rispetto alla posizione del PC.

Nel caso dei salti condizionati, in caso negativo il PC punta già all'istruzione successiva.

#### Istruzioni di salto con ritorno

Si tratta di chiamate a **subroutine (sottoprogrammi)**: è necessario non distruggere il contenuto che il PC ha alla fine della lettura dell'istruzione attuale caricando il valore dell'indirizzo dell'istruzione a cui si vuole saltare. Occorre salvare tale contenuto per rendere possibile, alla fine del sottoprogramma, l'operazione di ritorno e quindi l'esecuzione dell'istruzione successiva a quella che ha richiesto il salto.

Il salvataggio consiste nella memorizzazione del contenuto del PC nello **stack**, prima di caricare il nuovo indirizzo nel PC.

Tipicamente lo stack viene riempito a indirizzi via via decrescenti e svuotato ad indirizzi crescenti: l'indirizzo della cella di memoria riempita per ultima è contenuta nello Stack Pointer (SP), un registro della CPU.

Quando alla fine della subroutine viene richiesto il ritorno, il PC precedentemente salvato nello stack viene prelevato e salvato nel PC; lo SP viene aggiornato per puntare all'ultima cella scritta non ancora prelevata.

### *Esempi di salto con ritorno*

CALL <INDIRIZZO> salto con ritorno incondizionato

(SP) - 1 -> SP

(SP) -> MAR

(PC) -> MBR

(MBR) -> (MAR)

INDIRIZZO -> PC

CALL <COND><INDIRIZZO> salto con ritorno condizionato

Se la condizione è verificata, come sopra, altrimenti non si fa niente (PC punta già all'istruzione successiva a quella attuale)

### *Esempi di istruzioni di ritorno*

RET ritorno incondizionato

(SP) -> MAR

((MAR)) -> MBR

(MBR) -> PC

(SP) + 1 -> SP

RET <COND> ritorno condizionato

Se la condizione è verificata, come sopra, altrimenti non si fa niente (PC punta già all'istruzione successiva a quella attuale).

## Istruzioni di controllo macchina

Servono a modificare direttamente lo stato della CPU, senza operare quindi su dati o programmi.

Esempi:

HALT Arresta funzionamento

**RESET** Azzera lo stato della CPU

NOP No operation

EI Enable interrupt

DI      Disable interrupt

Dato che le caratteristiche architetturali di una CPU si ripercuotono direttamente sul suo Assembly Language, ne consegue che CPU diverse sono dotate di diversi Assembly Language che quindi sono basati su istruzioni diverse: NON PORTABILITÀ dei programmi.

## **Metodi di indirizzamento**

La memoria è organizzata in byte (8 bit): la locazione di memoria di indirizzo i corrisponde all'igesimo byte della memoria. In generale non è possibile indirizzare il singolo bit.

Il grado di parallelismo è il numero di bit che possono essere letti o scritti contemporaneamente.

Immediato: l'operando compare direttamente nell'istruzione come costante, ovviamente solo nel ruolo di "sorgente"

Absoluto: nell'istruzione compare l'indirizzo effettivo (fisico) di memoria dove si trova l'operando

Relativo: nell'istruzione compare un numero che rappresenta lo spostamento da attribuire al PC per ottenere l'indirizzo di memoria a cui saltare. Ciò consente programmi autorilocabili, che funzionano cioè senza modificare gli indirizzi in essi espressi, in qualsiasi posizione di memoria vengano caricati

Diretto a registro: l'operando è contenuto in un registro e nell'istruzione viene specificato l'identificatore del registro

Indiretto con registro: l'operando è in una cella di memoria il cui indirizzo di memoria è contenuto in un registro. Nell'istruzione viene specificato l'identificatore del registro

Con autoincremento (o postincremento): analogo all'indiretto con registro, ma il contenuto del registro viene prima utilizzato per indirizzare la memoria e poi incrementato della dimensione dell'operando

Con autodecremento (o predecremento): simmetrico rispetto all'autoincremento. ma il contenuto del registro viene prima decrementato di un numero pari alla dimensione in byte dell'operando e poi utilizzato per indirizzare la memoria

Indiretto con autoincremento: l'operando è in memoria. Il suo indirizzo si trova in un'altra posizione della memoria puntata dal contenuto di un registro. Nell'istruzione viene specificato l'identificatore del registro. Dopo essere stato utilizzato per indirizzare in modo indiretto l'operando, il contenuto del registro viene incrementato di un numero pari alla dimensione in byte di una cella di memoria atta a contenere un indirizzo

Con spiazzamento: nell'istruzione sono specificati un dato in complemento a 2 e l'identificatore di un registro. Il dato viene sommato al contenuto del registro per ottenere l'indirizzo dell'operando

Indiretto con spiazzamento: come nel precedente, ma l'indirizzo ottenuto dalla somma punta ad una posizione di memoria dove è contenuto l'indirizzo dell'operando

Implicito: è l'indirizzamento previsto da alcune istruzioni in cui il relativo codice operativo sottintende l'uso di registri. Tipico è il caso di istruzioni aritmetiche in cui l'accumulatore è coinvolto come sorgente di uno degli operandi e come destinazione del risultato

Con registri indice: utilizza due registri: uno contiene un indirizzo "base", l'altro un numero da moltiplicare per la dimensione dell'operando e da sommare poi alla base. Ciò che si ottiene è

l'indirizzo di memoria dell'operando. Questo tipo di indirizzamento è particolarmente efficiente per accedere a tabelle e matrici

Con lo stack pointer: SP punta alla sommità dello stack. Tale modalità permette di gestire le chiamate a subroutine. Alcune istruzioni permettono di inserire il contenuto di registri nello stack e, dualmente, di prelevare dati dallo stack da immagazzinare in registri (istruzioni PUSH e POP)

## Codici

In tutte le attività nelle quali si trattano informazioni, per codice si intende una modalità di rappresentazione, mediante un opportuno insieme di stringhe (o di simboli), di un insieme di oggetti materiali o un insieme di informazioni tendenzialmente più complesse delle stringhe (o dei simboli) che le codificano.

Una **codifica** mette in corrispondenza biunivoca ogni simbolo appartenente all'insieme più ampio con una stringa di simboli dell'insieme più ridotto secondo delle **regole di corrispondenza**, dette appunto *codifiche*.

Un codice binario è quindi la codifica dei simboli di un alfabeto  $\Sigma$  mediante stringhe di bit; se  $C$  è la cardinalità di  $\Sigma$  (ossia il suo numero di elementi) per il numero  $n$  di bit da utilizzare deve valere:

$$C \leq 2^n \quad n \geq \log_2 C$$

### Codici ridondanti e non ridondanti

Poniamo  $m = \lceil \log_2 C \rceil$  (parte intera superiore)

Un codice si dice **non ridondante** se utilizza un numero  $n = m$  di bit, ossia ne utilizza il numero minimo.

Un codice si dice **ridondante**, quando codifica gli  $M$  simboli distinti con  $n = m+r$  bit, cioè usando  $r$  bit aggiuntivi rispetto agli  $m$  bit strettamente richiesti dalla codifica binaria. L'aggiunta di bit di ridondanza permette di costruire codici che consentono di controllare eventuali errori di trasmissione. Si hanno due tipi di codici ridondanti:

- Codici a rivelazione di errore - Consentono di individuare la presenza di un errore;
- Codici a correzione di errore - Consentono non solo di individuare la presenza di un errore, ma anche di identificarne la posizione in modo da poterlo correggere.

La distanza di Hamming fra due parole in codice si ottiene contando il numero di bit diversi in posizioni corrispondenti. Se  $n = m$ , allora  $H = 1$ ; se invece c'è ridondanza,  $H \geq 1$ .

Si può dimostrare che per i codici a sola rilevazione:

- N° di bit errati rilevati:  $r = H - 1$

e che per i codici a correzione:

- N° di bit errati corretti:  $C \leq (H - 1)/2$
- N° di bit errati rilevati:  $R = H - C - 1$

## Codici ridondanti a rilevazione/correzione

Sono codici con distanza di Hamming  $H > 1$  e possono avere la caratteristica di rivelare o correggere uno o più bit errati.

### Parità:

È un codice con  $H = 2$ , che si ottiene dal codice non ridondante aggiungendo 1 bit in modo che il numero complessivo di bit uguali a “1” sia pari (parità pari) o dispari (parità dispari); avendo  $H=2$  può solo rilevare gli errori, e solo se questi accadono singolarmente o in un numero dispari nella stessa parola di codice.

### Codici di Hamming

Sono codici ridondanti con  $H > 2$  e perciò con capacità di auto correzione. Il più noto e semplice ha  $H=3$ : dato un codice non ridondante di  $n$  bit, vengono inseriti in particolari posizioni  $k$  bit di controllo.

Perché possa essere corretto un errore (singolo) deve valere:

$$n \leq 2^k - k - 1$$

Un codice di Hamming corregge un bit in posizione arbitraria, rivela la presenza di un solo errore e funziona bene se gli errori sono distribuiti casualmente.

Esempio: parola di 7 bit ( $n = 7$ )

Per creare un codice di Hamming con  $H=3$  bisogna rispettare la relazione precedente e quindi usare 4 bit di controllo aggiuntivi ( $k = 4$ ).

### Codici ciclici

Usati nella trasmissione a distanza su linee rumorose (soprattutto se sono possibili errori a raffica), sono codici rivelatori di errore, ma non auto correttori: in casi di errore rivelato il messaggio deve essere ritrasmesso.

Un messaggio  $M$  di  $k$  bit da trasmettere viene trattato come un polinomio di grado  $k-1$ , i cui coefficienti sono i bit del messaggio.

Preso un altro polinomio  $G$  di grado  $r$  si arriva, attraverso operazioni modulo 2 su  $M$  e  $G$ , ad un terzo polinomio  $T$  di grado  $t$  (con  $k < t \leq k+r$ ) divisibile per  $G$ : i coefficienti di  $T$  costituiscono la stringa di bit da trasmettere.

In ricezione un algoritmo analogo divide  $T$  per  $G$ . Se il resto della divisione è  $\neq 0$  il messaggio deve essere ritrasmesso.

## Codifica di cifre decimali

### Codice BCD

La codifica binary-coded decimal (BCD) è un modo comunemente utilizzato in informatica ed elettronica per rappresentare le cifre decimali in codice binario.

In questo formato ogni cifra di un numero è rappresentata da un codice binario di quattro bit, il cui valore è compreso tra 0 (0000) e 9 (1001); il BCD è un **codice pesato**, poiché il valore di ogni cifra

viene ottenuto eseguendo una somma pesata delle quattro cifre binarie che lo compongono. Le restanti sei combinazioni possono essere usate per rappresentare simboli. Per esempio il numero 127 è rappresentato in BCD come 0001, 0010, 0111. Durante la somma se il risultato supera 9 (1001) si somma 6(0110).

#### Codice a eccesso 3:

Analogo al codice BCD, ogni cifra è rappresentata dal corrispondente binario aumentato di 3: in questo modo dato un numero per ottenerne il suo complementare a 9 è sufficiente invertire 0 e 1, ossia basta complementare le cifre della sua codifica.

Esempio:       $9 - 2 = 7$                            $2 \text{ (0101)} ==> 7 \text{ (1010)}$

#### Codice 2421:

I numeri indicano i pesi di ogni bit: anche in questo caso le codifiche di una cifra e della cifra corrispondente al complemento a 9 sono fra loro complementari.

### **Codifica di informazioni alfanumeriche** (pag 39)

Per convenzione, si definisce alfabeto esterno di un calcolatore l'insieme dei caratteri che è in grado di leggere e stampare mediante i dispositivi di I/O. Questo alfabeto comprende almeno 64 caratteri (esempio codice Field data):

- le 26 lettere dell'alfabeto inglese maiuscole
- le 10 cifre decimali
- 28 caratteri vari (spazio, segni di punteggiatura, etc.)

In genere si utilizzano codici a 7 o 8 bit

- EBCDIC (Extended Binary Coded Decimal Interchange Code): 8 bit. Rappresenta caratteri alfanumerici e speciali. Personalizzato per le varie nazionalità (necessita di conversioni e ormai obsoleto)
- UNICODE: rappresenta con 16 bit tutti i caratteri della lingua parlata dall'uomo (usato da Java e adottato da HP, IBM, Microsoft, Oracle, ...)
- ASCII (American Standard Code for Information Interchange): ne esistono due versioni:
  - 7 bit → 128 simboli; in questo caso l'ottavo bit è usato a volte come ridondanza (H=2) per la rilevazione di un errore (bit di parità);

8 bit → 256 simboli (ASCII esteso); codifica anche lettere accentate e caratteri grafici; l'estensione non è standardizzata e quindi fra i 128 simboli aggiunti può succedere che alla stessa codifica corrispondano simboli diversi.

## Codifica di immagini (pag 42)

L'immagine è un insieme continuo di informazioni (luce, colore) in due dimensioni: quando si memorizzano immagini pittoriche o fotografiche si scomponte artificiosamente l'immagine in una sequenza di elementi di informazione codificati con sequenze binarie.

L'immagine viene suddivisa in un reticolo di punti detti pixel (picture element): ogni pixel viene codificato con una sequenza di bit. La qualità dell'immagine dipende dal numero di pixel per unità di lunghezza e dal numero di bit utilizzati per codificare ogni pixel.

Tuttavia non è possibile ingrandire a piacimento un'immagine perché non si aumenta il dettaglio, ma si ottiene un effetto sgranato, in cui si distinguono i singoli pixel.

Immagini in bianco e nero: raramente si usa un solo bit (es: fax). Ogni pixel di solito si codifica con 8 bit per rappresentare diversi livelli di grigio ( $2^8 = 256$  livelli)

Immagini a colori: i colori vengono ottenuti tramite la combinazioni di almeno 3 colori base, detti primari. La composizione può avvenire tramite la tecnica di sintesi sottrattiva o additiva in funzione del tipo di dispositivo utilizzato (stampante, monitor o televisore).

Per codificare un pixel si devono codificare i tre colori primari (es. RGB), la cui combinazione consente di ottenere il colore del pixel stesso: per ciascun colore primario spesso si usano 8 bit e quindi in totale per codificare ciascun pixel servono 24 bit. Ciò consente di codificare  $2^{24} \sim 16$  milioni di colori diversi.

Con *profondità di colore* si intende il numero di bit utilizzati per codificare i pixel.

L'occupazione di memoria di un'immagine dipende da:

- definizione o risoluzione dell'immagine (intesa come il numero di pixel per unità di lunghezza: dot per inch = DPI)
- numero dei colori (dipende a sua volta dal numero di bit usati per codificarli: 8, 16, 24 bit)

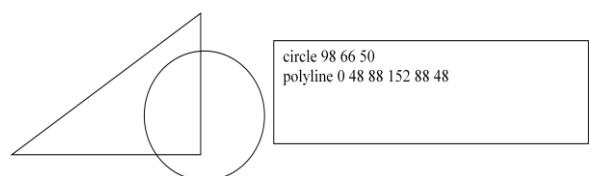
## Immagini vettoriali

Questo formato si usa in applicazioni di progettazione meccanica, architettonica, elettronica, e in tutte quelle applicazioni in cui l'immagine viene costruita con elementi di alto livello quali linee, cerchi, poligoni, archi, colori, ecc. Al contrario delle immagini bitmap in cui l'elemento di informazione è il pixel, in quelle vettoriali gli elementi dell'immagine sono gli oggetti grafici che la compongono.

Ogni oggetto è codificato attraverso un identificatore (es. polyline, circle, etc.) e alcuni parametri quali le coordinate del centro e la lunghezza del raggio (per la circonferenza) o le coordinate dei vertici (per il poligono).

Vantaggi:

- Indipendenza dal dispositivo di visualizzazione e dalla sua risoluzione
- Gli elementi grafici sono indipendenti l'uno dall'altro e si possono elaborare distintamente minore occupazione di memoria rispetto alla grafica bitmap



Svantaggi:

- Applicabilità limitata: un'immagine fotografica non si può scomporre in elementi primitivi
- Limiti nell'utilizzo in quanto si possono manipolare immagini vettoriali solo con il software utilizzato per crearle o compatibile

### Immagini in movimento

Le immagini in movimento vengono rappresentate attraverso sequenze di immagini fisse (frame) visualizzate ad una frequenza sufficientemente alta da consentire all'occhio umano di ricostruire il movimento (24, 25 o 30 immagini al secondo). Lo standard multimediale per le immagini in movimento, MPEG, codifica ciascun frame separatamente secondo lo standard JPEG.

Si utilizzano tecniche di compressione in quanto un minuto di filmato a 25 fotogrammi al secondo richiederebbe uno spazio di memorizzazione di 644 Mbyte.

## ***Dispositivi di memoria***

Due tecnologie di memorizzazione di informazioni digitali si sono affermate in base alle loro caratteristiche economiche e tecniche: le memorie **a semiconduttore** e le memorie **a supporto magnetico**.

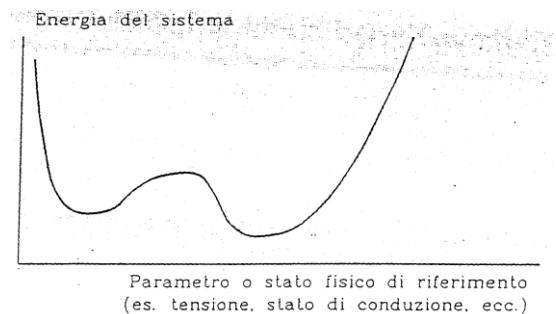
Entrambe queste memorie si basano su bistabili, cioè elementi caratterizzati da un diagramma energetico del tipo di quello rappresentato in figura.

Da un punto di vista matematico, il diagramma energetico di un bistabile deve presentare due minimi relativi separati da un massimo relativo di ampiezza significativa: in termini pratici, questo si traduce in un elemento che è in grado di permanere, per un tempo a priori indeterminato, in uno dei due stati stabili (cioè di minima energia) finché non interviene un fenomeno di entità sufficiente a far commutare il bistabile, cioè a farlo passare da uno stato a minima energia all'altro.

Associando convenzionalmente il valore logico ZERO ad uno dei due stati ed il valore logico UNO all'altro, è possibile memorizzare un'informazione binaria (bit) in ogni bistabile.

La classificazione delle memorie si basa su:

- modalità di accesso (a cui è legata la velocità di risposta)
- stabilità dell'informazione memorizzata



### **Memorie a semiconduttore**

#### Memorie RAM (Random Access Memory)

Ne esistono di due tipi:

- Statiche: Static RAM (SRAM)
- Dinamiche: Dynamic RAM (DRAM)

Nelle SRAM la cella elementare che memorizza un bit è costituita da un circuito contenente diversi transistor e l'informazione scritta si mantiene finché sono alimentate.

Nelle DRAM basta un solo transistor ma l'informazione tende a cancellarsi e va rinfrescata, riscrivendola, ogni pochi msec (refresh). Serve un controllore che gestisca il rinfresco (Dynamic RAM Controller).

Le SRAM sono più veloci delle DRAM, più costose e di minore capacità.

Esistono anche le RAM sincrone (SDRAM e SSRAM): offrono la possibilità di trasferire blocchi di dati presenti in memoria a indirizzi consecutivi, specificando un indirizzo di partenza e una lunghezza.

Il trasferimento è più veloce perché si genera solo il primo indirizzo e un segnale di clock sincronizza la sequenzializzazione dei dati.

### Memorie ROM

Sono memorie a sola lettura che contengono in genere solo programmi, tipicamente di funzionamento per applicazioni embedded o di inizializzazione dei calcolatori.

Mantengono l'informazione anche se vengono disalimentate.

Ne esistono di vari tipi:

- ROM (Read Only Memory): l'informazione viene scritta in fabbrica durante il processo di fabbricazione e non è più modificabile
- PROM (Programmable ROM): possono essere scritte una volta sola dall'utente attraverso una particolare apparecchiatura detta *programmatore di PROM*
- EPROM (Erasable Programmable ROM): come la PROM, può essere programmata dall'utente, ma può essere cancellata tramite esposizione ai raggi ultravioletti. Per questo motivo, i contenitori di EPROM hanno sempre una finestrella che lascia vedere il chip
- OTP ROM (One Time Programmable ROM): identiche alle EPROM, ma prive della finestrella trasparente (per produzioni in serie)
- EEPROM o E<sup>2</sup>PROM (Electrically Erasable Programmable ROM): come la EPROM, ma anche la cancellazione viene effettuata con segnali elettrici
- EAROM (Electrically Alterable ROM): alterabili elettricamente. Non è più necessaria la cancellazione di tutto il contenuto della memoria quando si vogliono modificare anche solo poche celle
- FLASH: offrono significativi vantaggi rispetto alle EPROM. Utilizzano una tecnica di cancellazione tramite impulsi elettrici, al posto della radiazione UV. Permettono cancellazioni parziali, direttamente sulla scheda e rapide.

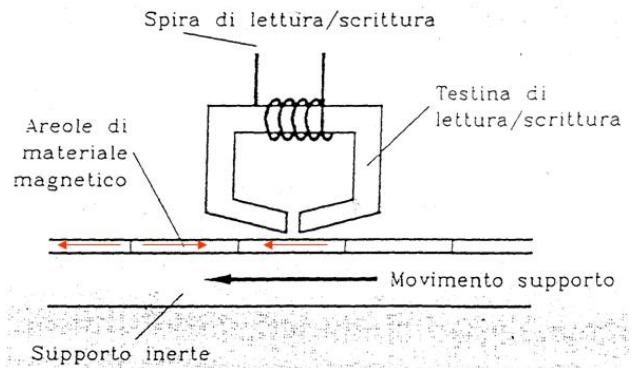
### **Memorie a supporto magnetico**

I bistabili sono costituiti da areole di materiale ferromagnetico (ossido di ferro), depositato su un supporto (plastico o ceramico) e portato in movimento sotto un dispositivo elettromagnetico (testina di lettura/scrittura).

I due stati stabili corrispondono ai due sensi di magnetizzazione di tale materiale, nella direzione di traslazione del supporto rispetto alla testina.

La magnetizzazione delle diverse areole si ottiene forzando una corrente positiva o negativa nella spira avvolta intorno alla testina, che le orienta nel verso desiderato.

La rivelazione dello stato di magnetizzazione (lettura) si effettua sfruttando la tensione che si manifesta nella spira a seguito del passaggio, sotto la testina, di areole polarizzate nei due sensi.



All'interno di questo filone tecnologico, il progresso è stato senza soste: si è passati dalla memorizzazione di decine di bit per millimetro quadro a svariate decine di migliaia. Ciò è stato ottenuto riducendo alcuni fattori geometrici (lo spessore dello strato magnetico, le dimensioni della testina di lettura/scrittura, la distanza fra testina e superficie magnetica), i cui valori sono ormai dell'ordine di frazioni di micron.

La tecnologia attuale può essere ulteriormente migliorata, ma si è ormai vicini ai limiti intrinseci di fattibilità: per questa ragione c'è un forte stimolo a realizzare nuove memorie basate su principi fisici nuovi.

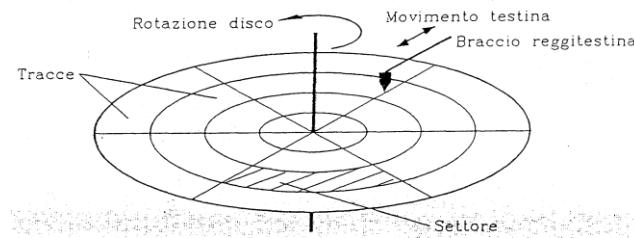
Caratteristiche comuni dei dispositivi di memorizzazione a supporto magnetico sono:

- l'elevata quantità di informazione immagazzinabile (da MByte a diverse centinaia di GByte)
- la permanenza dell'informazione anche in assenza di alimentazione

Ciò che differenzia i diversi dispositivi è essenzialmente la forma della struttura sulla quale viene depositato il materiale ferromagnetico; esistono infatti dischi flessibili (floppy disk), dischi rigidi (hard disk) e nastri.

### Hard Disk

Il materiale (2-3 micron di spessore) è deposto sulle superfici di un disco (realizzato in leghe di alluminio o in materiali compositi in vetro), tenuto in rotazione a velocità costante attorno al proprio asse: ogni superficie è suddivisa in tracce costituite da corone circolari concentriche, che possono condividere un'unica testina, o disporre di proprie testine dedicate. Le testine sono mantenute a pochi micron di distanza dal disco.



I dischi a testine fisse sono più veloci, ma sono molto costosi ed il disco non può essere asportato ed essere sostituito: sono più diffusi i dischi fissi a testine mobili e sigillati insieme alle testine in un unico contenitore ermetico (Winchester).

Un'unità a disco può essere costituita da un solo disco o da più dischi coassiali.

#### *Memorie RAID (Redundant Array of Independent Disks):*

Insieme di dischi rigidi a basso costo collegati tra di loro: servono per proteggere i dati in caso di

malfunzionamento di un disco rigido. L'insieme dei dischi viene visto come un unico disco

I dati sono distribuiti su più di un disco (ridondanza) e possono essere recuperati in caso di guasti.

#### Floppy Disk

Analoghi ai dischi rigidi, ma il supporto in tal caso è flessibile e possono essere estratti dal dispositivo di memorizzazione (drive) in modo da poter trasferire le informazioni tra diversi calcolatori.

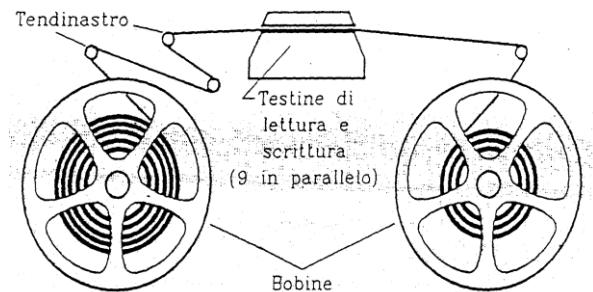
I floppy disk sono normalmente fermi e quindi il tempo di accesso alle informazioni (circa 0.1 s) è più lungo rispetto a quello dei dischi rigidi (dato dal tempo di posizionamento della testina più il tempo necessario a raggiungere l'informazione). I più diffusi sono da 3.5 pollici (capacità di 720 KByte o 1.44 MByte).

#### Nastri

Il materiale ferromagnetico è depositato su nastri di plastica avvolti su opportune bobine; sono suddivisi generalmente, in senso trasversale, in 9 strisce (piste) parallele, ciascuna assegnata ad una testina che consentono la memorizzazione di un byte dotato del bit di parità.

In senso longitudinale, invece, le informazioni sono organizzate in blocchi, suddivisi in record, intercalati da zone non magnetizzate (interrecord gap).

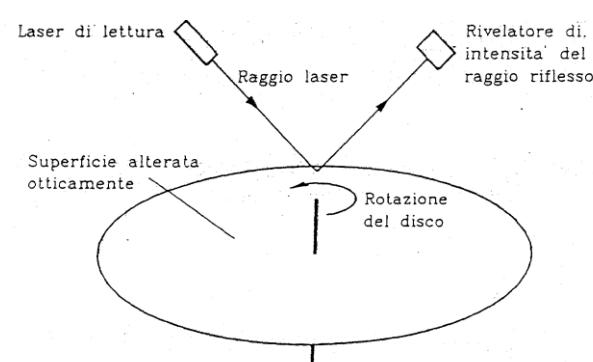
Il nastro è normalmente fermo e trasferisce un blocco per volta



#### Memorie a dischi ottici

Sono particolarmente interessanti per le caratteristiche di affidabilità, economicità e capacità.

Derivate dai CD (compact Disk) per riproduzioni audio, sono basate su bistabili costituiti da deformazioni permanenti ("buchi" o "pit") apportate, durante la fase di scrittura, alla struttura meccanica di supporto (un disco di materiale plastico).



Un raggio di luce, generato da un laser per garantire la dimensione limitata richiesta, colpisce la superficie del disco: in assenza di deformazioni della superficie, una percentuale considerevole

dell'energia incidente viene riflessa verso il fotorivelatore, mentre in presenza di una deformazione si ha una dispersione di energia luminosa che solo in piccola parte lo raggiunge. Le variazioni di tensione di tale dispositivo consentono di ricostruire l'informazione presente sul disco.

Come le ROM sono memorie a sola lettura che mantengono l'informazione senza possibilità di cancellarla.

Sono chiamate WORM (Write Once Read Many), dato che le unità sono dotate della possibilità di alterare la superficie dei dischi (mediante un laser di potenza molto superiore a quello di lettura) e consentire quindi all'utente di memorizzare in modo permanente informazioni destinate a successive consultazioni.

Sulla superficie del disco è presente un'unica traccia, avvolta a spirale.

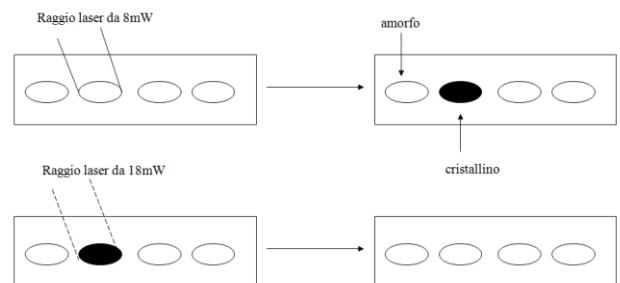
È un dispositivo ad elevata capacità (oltre 1/2 Gbyte su un disco di 5 pollici di diametro), ma con tempi di accesso relativamente lunghi (fino a 1 secondo), che può permettere di archiviare periodicamente tutte le informazioni disponibili sul calcolatore.

Mette a disposizione archivi enormi se si ha una configurazione a "juke box" in cui più dischi possono essere prelevati ed inseriti in modo automatico.

### *CD riscrivibili: Magneto-Ottici*

Magneto-Optical: Sandwich di policarbonato e materiale magnetico.

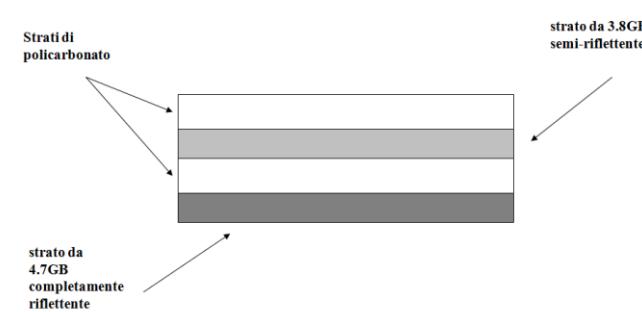
Per la memorizzazione usa un raggio laser di circa 40mW ed un campo magnetico: il punto riscaldato dal laser assume polarità uguale a quella del campo magnetico applicato. Prima di registrare si deve cancellare.



### *CD riscrivibili: Phase Change*

Dischi a variazione di fase (phase-change):

- Sandwich di policarbonato e tellurio o selenio
- Bit scritto come spot usando un laser da 8mW oppure da 18mW
- Lettura tramite un laser di minore potenza
- Riscrivibile direttamente
- Ogni spot può essere in stato amorfo o cristallino



### *DVD-ROM*

Digital Video Disk (Digital Versatile Disk)

Stesso diametro del CD-ROM ma molto più capiente: da 4.7GB (singola faccia singolo strato) a 17 GB (doppia faccia doppio strato).

### Tipi di accesso

*Uniforme o casuale*: tipico della memoria centrale. L'accesso ai dati avviene in modo diretto tramite il loro indirizzo e in un tempo costante che non dipende dalla loro posizione

*Sequenziale*: tipico dei nastri. I dati vengono scritti e letti uno dopo l'altro in posizione contigua sul supporto

*Diretto o misto*: tipico dei dischi. L'accesso ai dati avviene in modo diretto, ma il tempo di accesso varia a seconda della corrente posizione relativa tra dato e testina di lettura/scrittura

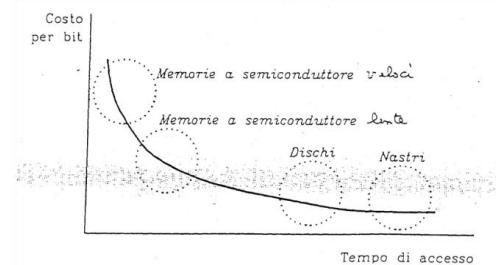
## Memoria centrale e di massa

Due parametri molto utili per qualificare i diversi tipi di dispositivi sono il costo per bit memorizzato e il tempo di accesso (la velocità di risposta alla richiesta di trasferimento di informazioni).

Una considerazione che discende dall'esame del grafico è il costo relativamente elevato dei dispositivi a semiconduttore, tale da rendere troppo onerosa e quindi improponibile la realizzazione di calcolatori con grandi quantità di memoria di lavoro; un problema ulteriore è relativo alla volatilità delle informazioni, che vengono perse non appena il dispositivo viene disalimentato.

Un'ampia disponibilità di memoria è particolarmente utile perché:

- uno spazio di indirizzamento ampio consente di utilizzare strutture dati complesse ed estese;
- i calcolatori usati da più utenti devono soddisfare la presenza contemporanea di vari programmi e dei relativi dati in memoria



Una soluzione economicamente valida sarebbe quella di utilizzare dispositivi a supporto magnetico (ad esempio dischi) per realizzare, a parità di costo, memorie di lavoro molto estese, ma i tempi di accesso necessari per ottenere l'informazione ricercata sono tali da impedirne l'uso come supporto di memoria di lavoro.

Tuttavia la distribuzione degli indirizzi generati durante l'esecuzione di programmi non è di fatto casuale: esiste un'elevata probabilità che, a partire dalla generazione di un certo indirizzo di memoria, ne venga generato uno uguale o simile entro breve tempo, cioè a distanza di pochi accessi in memoria. Questo comportamento è noto come **principio di località degli accessi**.

Il principio di località degli accessi deriva da:

- una *località temporale*, dovuta al fatto che ogni programma ha un'elevata probabilità di riutilizzare a breve le informazioni appena acquisite
- una *località sequenziale*, dovuta al fatto che l'esecuzione di un'istruzione ha un'elevata probabilità di essere seguita dall'istruzione immediatamente successiva nel programma; il discorso analogo vale per i dati

Questo comportamento fa sì che diventi fattibile un'organizzazione *virtuale* della memoria di lavoro, nella quale lo spazio utilizzabile da ogni singolo programma è largamente superiore alle dimensioni fisiche della memoria di lavoro effettivamente presente nel calcolatore (memoria centrale) nella quale

vengono temporaneamente ricopiate dalla memoria più ampia e lenta (memoria di massa) le informazioni correntemente utilizzate dall'unità centrale.

Il termine *virtuale* deriva dal fatto che in questa organizzazione di memoria lo spazio di indirizzamento dell'unità centrale non ha un riscontro fisico nella memoria centrale, che è solo una parte della memoria reale del sistema, ma fa riferimento ad una memoria di lavoro virtuale (corrispondente in pratica alla memoria di massa). Questo richiede una politica di gestione della memoria virtuale, che consiste in un metodo di conversione dell'indirizzo virtuale, emesso dall'unità centrale, nell'indirizzo fisico della cella di memoria di lavoro nella quale è stato ricopiato il valore della locazione virtuale cercata: tale compito è demandato ad un'unità dedicata, l'unità di gestione della memoria (MMU: Memory Management Unit).

Quando l'unità centrale richiede l'accesso ad una locazione di memoria virtuale non presente in una cella di memoria reale, si rende necessario effettuare una ricopiatura della memoria virtuale desiderata da memoria di massa a memoria centrale, eventualmente preceduta da una ricopiatura in senso inverso per liberare spazio nella memoria centrale (swapping). Per ottimizzare le prestazioni, si sfrutta in queste situazioni il principio della località degli accessi e si trasferisce da memoria di massa un intero blocco di locazioni adiacenti, contando su un'elevata probabilità di usarle a breve.

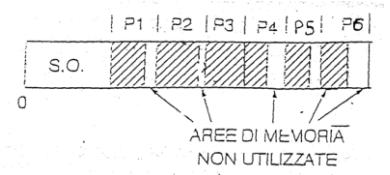
## Paginazione

Si basa sul concetto di pagina, cioè di blocco di parole consecutive, di dimensione prefissata (pochi KB) che costituisce l'unità minima di informazione trasferita dalla memoria di massa alla memoria centrale e viceversa durante le operazioni di swapping richieste: in questa modalità di gestione della memoria, sia la memoria virtuale, sia la memoria fisica vengono suddivise in pagine di uguali dimensioni e ogni programma occupa in genere più pagine la cui dislocazione è libera dal vincolo di consecutività.

L'operazione di conversione da indirizzo virtuale a indirizzo fisico consiste nella ricerca della posizione di memoria fisica nella quale la pagina virtuale referenziata è stata inserita: questo si fa costruendo una tabella che fornisce la corrispondenza fra pagine virtuali e quelle fisiche.

Se tutte le pagine sono occupate, la politica di scelta più usata prevede di eliminare la pagina non utilizzata da più tempo (LRU: Least Recently Used).

Il problema principale della gestione a pagine è l'eccessiva dimensione della tabella quando lo spazio di indirizzamento virtuale è molto esteso: questo problema non può essere risolto aumentando la dimensione delle singole pagine, se si vuole evitare un eccessivo spreco di memoria dovuto al fatto che l'ultima pagina di un programma è utilizzata solo in parte.



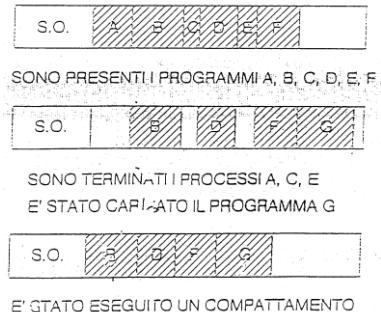
## Segmentazione

La suddivisione non è in blocchi di uguale dimensione, ma in unità logicamente separate, i *segmenti* (moduli di cui è composto un programma, strutture dati usate, ecc); la conversione di indirizzo richiede che ad ogni programma venga associata una tabella che, per ogni segmento del programma,

riporta il bit di presenza, l'indirizzo di tale segmento in memoria centrale e la lunghezza del segmento.

La conversione comporta quindi la lettura della tabella:

- se il segmento è presente, si ottiene dalla tabella l'indirizzo di memoria centrale in cui il segmento interessato è stato caricato;
- se il segmento è assente, si rende necessario il caricamento dalla memoria di massa: la differente lunghezza dei segmenti comporta una suddivisione molto più articolata della memoria fisica, non più partizionata in pagine di uguale lunghezza, ma assegnata in base alla richieste dei programmi in esecuzione;



Tuttavia ogni caricamento di un nuovo segmento va ad occupare lo spazio lasciato libero da un segmento di dimensioni maggiori di quello che si sta caricando, con il risultato di lasciare inutilizzata la parte di memoria corrispondente alla differenza fra segmenti vecchio e nuovo: dopo un certo numero di operazioni di swapping, la memoria appare frammentata, cioè piena di spazi vuoti residui dei vari caricamenti, che riducono in modo inaccettabile lo spazio di memoria centrale utilizzabile. Una compattazione periodica della memoria implica una ricopertura dell'intero contenuto della memoria ed è quindi un'operazione estremamente lenta.

### Segmentazione paginata

Tale metodo è una soluzione intermedia tra paginazione e segmentazione in grado di sfruttare le caratteristiche migliori di entrambe: la suddivisione della memoria virtuale è ancora realizzata in segmenti di memoria variabili, che sono però suddivisi in pagine di lunghezza fissa, senza vincolo di consecutività fisica.

La conversione di indirizzi richiede l'uso di due livelli di tabella, il primo destinato ad identificare la posizione del segmento, il secondo ad individuare la pagina all'interno del segmento: il risultato è una gestione particolarmente efficiente che elimina i problemi di frammentazione e relativa compattazione mantenendo una visione segmentata della memoria virtuale.

La gestione virtuale dello spazio di indirizzamento della CPU di un calcolatore, sfruttando il principio di località degli accessi, consente di costruire strutture di memorizzazione con dimensioni tipiche del dispositivo di memoria più grande e lento, ma con tempi di accesso di poco superiori a quelli tipici del dispositivo più ristretto e veloce. Si ha quindi una gerarchia di memoria nella quale:

- le informazioni a disposizione del calcolatore sono memorizzate nel dispositivo più capiente e lento;
- una copia della parte di tali informazioni di uso frequente è presente nel dispositivo più veloce e meno capiente, il disco;
- la parte delle informazioni quotidiane usate dai programmi in esecuzione è ricopiata anche nel successivo gradino della gerarchia, la memoria a semiconduttore lenta ed ampia;
- di quest'ultima parte, le informazioni di uso corrente (come le istruzioni di un ciclo di programma) sono ricopiate anche nell'ultimo gradino: la memoria a semiconduttore veloce, ma piccola.

Tutte le informazioni ragionevolmente necessarie nell'uso del calcolatore sono presenti in unità di memoria a disco e lo scaricamento periodico di dati verso CD/nastro o il caricamento periodico dei dati da CD/nastro possono essere effettuati da operatori umani, senza necessità di prevedere meccanismi automatici

Completamente diverso è il discorso per la coppia di memorie a semiconduttore che risultano comunque rallentate dalla necessità di accedere ad una memoria di lavoro lenta e richiedono pertanto dispositivi veloci per funzionare a pieno ritmo; d'altro canto, le limitate capacità dei dispositivi veloci li rendono inadatti a costituire l'unico supporto l'unico supporto di memoria centrale.

Si introducono nell'architettura del calcolatore entrambi i tipi di memorie a semiconduttori, realizzando strutture di memoria centrale con velocità paragonabili a quelle delle memorie veloci e capacità tipiche delle memorie lente. Si parla di **memoria cache** (nascosta).

### Memoria cache

L'organizzazione delle due memorie a semiconduttore è simile alla gestione della memoria virtuale: i due tipi di memorie sono considerati suddivisi in blocchi di uguale dimensione, che vengono temporaneamente ricoppiati dalla memoria grande e lenta a quella piccola e veloce per fornire tempi di risposta che siano nella maggior parte dei casi quelli della memoria piccola e veloce, sfruttando le proprietà di località degli accessi.

Per velocizzare ulteriormente le operazioni, l'accesso a memoria grande e lenta viene iniziato in parallelo alla ricerca nella cache, prima ancora di sapere se la parola cercata esiste o meno in memoria piccola e veloce: in caso negativo, infatti, si raggiunge comunque la cella desiderata nel tempo di accesso alla memoria grande e lenta, senza rallentamenti dovuti alla presenza della cache. Questa gestione della memoria cache sia un modo per inserire parallelismo a livello di accesso alla memoria di lavoro.

La presenza della circuiteria di gestione consente di avere disponibile, anche nel caso peggiore, l'informazione desiderata nel tempo che sarebbe necessario per accedere alla memoria di lavoro usuale; fra un accesso e l'altro, in parallelo alle attività della CPU, tale circuiteria si preoccupa anche di portare nella memoria più veloce le informazioni che più probabilmente saranno necessarie in un prossimo futuro alla CPU stessa, effettuando una sorta di trasferimento a pacchetti, invece che a parole singole, con la memoria di lavoro.

## Evoluzione della macchina di Von Neumann

### Problemi

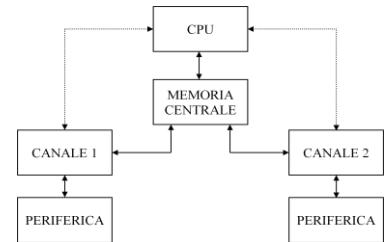
Nonostante il fatto che con il progresso tecnologico aumentasse la velocità di elaborazione, con CPU più rapide e parallelismo di dati e indirizzamento maggiori, la macchina di Von Neumann aveva un limite intrinseco che ne impediva un drastico miglioramento delle prestazioni, ossia la stretta sequenzialità delle operazioni: singolarmente, l'una dopo l'altra, ogni istruzione macchina veniva acquisita, decodificata ed eseguita, il che comporta un tempo totale di elaborazione pari alla somma

dei tempi richiesti da ogni singola attività. Riassumendo: mancanza di parallelismo, lentezza di calcolo di operazioni su matrici (stessa istruzione per dati diversi), sequenzialità nell'accesso alla memoria centrale o a un dispositivo di I/O.

## Parallelismo nelle operazioni di I/O

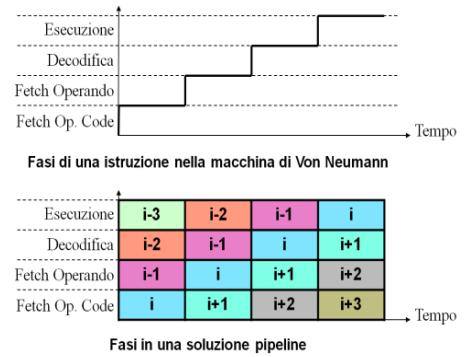
Le periferiche sono collegate ai componenti interni tramite un'interfaccia (o controller), che si occupa di tradurre i dati in ingresso e in uscita in un modo comprensibile al destinatario. I controller risultano collegati allo stesso bus che collega la CPU alla memoria principale. Nell'interazione con le periferiche, possono essere adottati tre metodi con tre diversi livelli di parallelismo:

- **Interruzione:** la CPU continua a lavorare e quando le operazioni di I/O sono terminate l'interfaccia invia un segnale di interrupt, in modo da permettere la lettura dei dati riducendo al minimo il tempo di sospensione dell'esecuzione del programma. Questa tecnica diventa inefficace nel caso in cui la quantità di dati da elaborare sia significativa; inoltre, deve essere previsto il caso in cui si presentino più interrupt requests in contemporanea, per cui è necessario stabilire delle priorità.
- **DMA (Direct Memory Access):** l'interfaccia invia una richiesta alla CPU per occuparsi personalmente del trasferimento dei dati alla memoria centrale, senza interpellare il processore, che cede l'utilizzo del bus mentre continua l'esecuzione di altri programmi. Ciò permette di trarre enorme vantaggio nel caso in cui si lavori su grandi moli di dati, dato che il tempo necessario per il trasferimento è relativamente lungo; tuttavia il DMA aumenta il traffico sul bus, che necessita di un accurato coordinamento degli accessi, un lavoro di non semplice realizzazione che può portare al *collo di bottiglia di Von Neumann*, che si verifica quando la CPU e i controller entrano in competizione.
- **Canali di I/O:** sono dispositivi dedicati a funzioni di scambio di informazioni con le periferiche che consentono di svincolare la CPU da operazioni di sincronizzazione, transcodifica, formattazione dei dati ecc. Una volta ricevuta l'autorizzazione, i canali lavorano in completa autonomia e in parallelo con la CPU: il vantaggio che se ne ricava è elevato, ed è per questo motivo che sono largamente utilizzati per esempio nell'elaborazione grafica o nella ricezione di segnali analogici.



## Parallelismo nella CPU

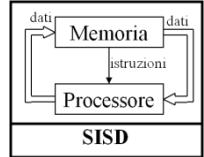
Per ottenere una parallelizzazione del funzionamento della CPU si può utilizzare il concetto di **pipeline** con le diverse sottounità dedicate alle fasi di fetch, decode ed execute: invece di lasciare immobile la circuiteria dedicata alle altre fasi, essa viene impiegata costantemente facendo in modo che al termine dell'analisi di una determinata istruzione, lo stesso elemento passi subito a processare l'istruzione successiva, analogamente a quanto avviene nella *catena di montaggio*. L'incremento di efficienza che ne deriva è notevole, ma



presenta un difetto: nel caso in cui vi sia un'istruzione di salto, il vantaggio che si presentava con istruzioni contigue svanisce, in quanto le operazioni prelevate non sono quelle che dovranno andare effettivamente eseguite.

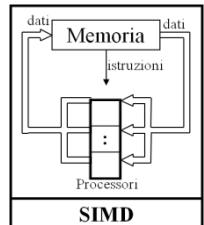
## Sistemi multiprocessore

Affinché si abbia una vera elaborazione parallela, è necessario che vi siano più unità di elaborazione.

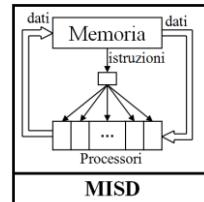


Nella macchina di Von Neumann si ha un singolo flusso di dati e un singolo flusso di istruzioni: per questo motivo è classificata come **SISD** (Single Instruction stream, Single Data stream).

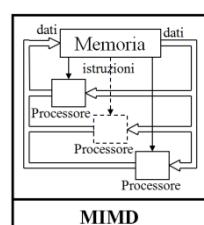
Una macchina dotata di una sola unità di controllo e più unità aritmetiche indipendenti viene definita come **SIMD** (Single Instruction stream, Multiple Data stream): questi tipi di architetture permettono di eseguire le stesse operazioni su dati diversi, rivelandosi particolarmente utili, ad esempio, nelle elaborazioni matriciali che richiedano la valutazione di uguali espressioni matematiche ma combinazioni numeriche differenti. Il processore principale invia le istruzioni alle unità di elaborazione (o Processing Element), che le eseguono nello stesso istante: ogni unità è dotata di una memoria privata e può eventualmente scambiare informazioni con le altre.



Nelle macchine **MISD**, invece, esistono più processori, ognuno con una propria memoria (registri), la quale a sua volta ha un proprio flusso di istruzioni che vengono eseguite sullo stesso flusso di dati. Un possibile utilizzo di questa tipologia di architetture si trova nell'ambito della crittografia, anche se nella realtà non si è mai avuta la necessità di realizzarle o metterle in commercio.



Le macchine **MIMD** prevedono la replicazione dell'intera struttura della macchina di Von Neumann per ottenere architetture multiprocessore: si tratta di un calcolatore costituito da più unità di controllo e unità di calcolo che operano in parallelo su flussi di dati diversi effettuando elaborazioni anch'esse diverse in modo asincrono. Nella risoluzione di un singolo problema, i vari processori si dividono i compiti da svolgere per velocizzare il procedimento: nel caso in cui la memoria sia condivisa è necessario che vi siano dei meccanismi di controllo che evitino conflitti; se invece ogni processore ha memoria propria, vi è comunque la possibilità di scambio di informazioni tra i diversi processori.



## Sistemi operativi

### Caratteristiche generali

Un sistema operativo è un software che fornisce all'utente una serie di programmi per usufruire al meglio della potenza di calcolo della macchina: i sistemi operativi nascondono tutti i dettagli tecnici legati allo specifico hardware e architettura rappresentando le informazioni ad un alto livello, meglio comprensibile dall'uomo.



Esso garantisce l'operatività di base di un calcolatore, coordinando e gestendo le risorse del processore e della memoria, le periferiche, le attività software (processi) e facendo da interfaccia con l'utente, senza il quale quindi non sarebbe possibile l'utilizzo del computer stesso e dei programmi specifici. Saper utilizzare una macchina significa quindi saper utilizzare il suo sistema operativo: per questa ragione spesso macchine diverse supportano lo stesso sistema operativo. Il sistema operativo è costituito dal *kernel* e dal *software di base*.

Il **kernel** è un programma che va in esecuzione all'avvio della macchina e che assume un ruolo di vitale importanza nell'utilizzo della macchina: esso gestisce i processi per l'esecuzione dei programmi, l'allocazione della memoria e dei canali di I/O del computer, i componenti hardware e delle periferiche. Al momento dell'esecuzione dei programmi utente, il kernel impiega la CPU per un certo lasso di tempo (*tempo di overhead*), che va quindi aggiunto al tempo reale di completamento delle operazioni.

Il **software di base**, invece, è un insieme di programmi che servono a facilitare l'utilizzo della macchina, permettendo ad esempio la creazione di altri programmi, la modifica, la gestione (copia, eliminazione, trasferimento) e la visualizzazione dei file, l'utilizzo delle periferiche, il recupero delle situazioni di errore, ecc.

### Classificazione dei sistemi operativi

Cronologicamente parlando, è possibile identificare vari tipi di sistemi operativi:

- **Dedicati:** la macchina è dedicata all'uso da parte di un singolo utente che può eseguire solo un programma alla volta. Il S.O. fornisce un interprete di comandi e la gestione di file e dati, ed è caratterizzato da un basso sfruttamento durante le interazioni con l'utente e da un kernel molto semplice. I sistemi dedicati erano usati non solo agli esordi dell'informatica, ma anche con i PC.
- **A lotti (batch):** via via più utilizzati per sfruttare meglio la crescente velocità delle macchine: un insieme di lavori (*jobs*) veniva accorpato in un lotto (*batch*) e venivano trasferiti sulla memoria di massa, per poi essere caricati ed eseguiti senza interruzioni fino al termine secondo la **coda di job**, che segue la logica FIFO (First In First Out). Eseguire una serie di jobs in sequenza permetteva di eliminare i tempi morti tra la terminazione del programma di un utente e il caricamento del programma dell'utente successivo: tuttavia la CPU si ritrovava a sottostare alle ridotte velocità dei dispositivi di I/O, il che poteva comportare tempi di elaborazione comunque molto lunghi.
- **Multiprogrammazione:** utilizzati per migliorare l'efficienza delle macchine sfruttando bene la CPU: nei sistemi *multitasking* più programmi sono caricati in memoria allo stesso tempo e le operazioni di I/O sono sovrapposte temporalmente all'esecuzione di altri programmi grazie all'uso dei canali. Quando un processo in esecuzione (*running*) necessita di informazioni da un dispositivo di I/O, il sistema operativo avvia l'operazione di I/O, sospende temporaneamente il programma in corso e prosegue il suo lavoro con un altro processo pronto ad essere eseguito (*ready*): in questo modo si possono venire a creare una coda di processi ordinati secondo la priorità assegnata a ciascuno di essi (*scheduling*). Tuttavia questi sistemi non erano in grado

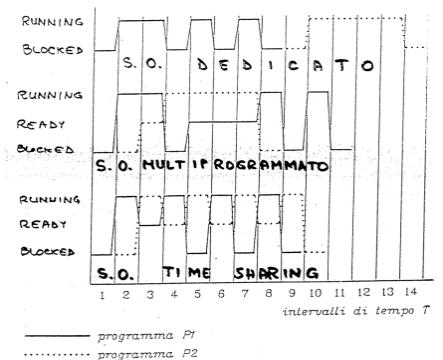
di riconoscere eventuali processi che richiedevano l'utilizzo della CPU per un prolungato periodo di tempo (ad esempio, programmi di elaborazione matematica con dati ricevuti in precedenza): in questi casi la CPU continuava a operare su questi processi, negando quindi l'esecuzione agli altri programmi fino al termine di tutte le operazioni correnti.

- **Interattivi (Time-Sharing):** mentre i sistemi *multitasking* erano pensati generalmente per gestire più operazioni allo stesso tempo, i sistemi *time-sharing* erano utilizzati soprattutto in ambienti multiutente, con più persone che lavorano contemporaneamente alla stessa macchina: il tempo di utilizzo della CP viene suddiviso dal sistema operativo in fette (*time slices*) di circa 100-800 ms, in modo tale che ogni processo in memoria ricevesse a turno l'uso della CPU per quel certo lasso di tempo. Così facendo, l'utente che lavora al terminale ha l'impressione di avere la macchina a sua completa disposizione (come in un sistema dedicato): tuttavia nella durata complessiva dell'esecuzione per ogni utente assumeva rilevanza significativa il *tempo di overhead*, che diventava particolarmente in gente nel caso in cui ci fossero molti processi attivi. Fu la crescita della velocità delle macchine a contribuire alla riduzione di questo lasso di tempo.

- **Transazionali (Real Time):** sono sistemi al servizio di una specifica applicazione con vincoli precisi nei tempi di risposta: vengono utilizzati tipicamente in ambito industriale (controllo di processo, gestione di strumentazioni, allarmi, transazioni bancarie, ecc) o comunque dove sia necessario ottenere una risposta dal sistema entro un tempo prefissato. Un sistema operativo *real time* non deve essere necessariamente veloce, l'importante è che il tempo che passa dalla richiesta di esecuzione di un processo al completamento della stessa sia minore del tempo stabilito.

- **Multiprocessore:** nei sistemi ad architetture parallele (o concorrenti) è necessario che il sistema operativo gestisca le diverse CPU e le risorse tra loro in comune (memoria condivisa, bus, periferiche).

Affinché vi sia un'adeguata suddivisione dei lavori e nessun processore rimanga inutilizzato, il sistema operativo può essere eseguito contemporaneamente da più CPU (*sistema multiprocessore simmetrico*) oppure da una sola CPU (che funge da *master*), mentre le altre (*slave*) svolgono compiti differenti (*sistema multiprocessore asimmetrico*).



## Creazione di un programma compilato

Per la realizzazione di programmi compilati sono necessari alcuni software, di cui alcuni solitamente inclusi nel software di base; certi sistemi operativi dispongono inoltre strumenti di ausilio della programmazione molto utili. L'insieme minimo necessario per lo sviluppo di programmi include le seguenti funzioni: *editing* dei sorgenti dei programmi, *compilazione* dei sorgenti per la produzione di file oggetto, *linking* dei file oggetto da cui si ottiene l'eseguibile. Una volta caricato il programma in

memoria, è possibile eseguirlo e, in seguito, ripetere le operazioni sopra descritte per un eventuale *debugging*.

I sorgenti, i file oggetto e gli eseguibili vengono memorizzati sul supporto di memoria magnetico (il disco rigido) e vengono trattati dal sistema operativo allo stesso modo di tutti gli altri: sono dotati ad esempio di estensione, diritti di accesso e possono essere copiati, spostati, cancellati, ecc.

Esistono tuttavia alcuni linguaggi di programmazione (come il BASIC e PROLOG) che possono essere direttamente eseguiti tramite un **programma interprete**: questo software traduce ogni istruzione del linguaggio riportata nel file sorgente e ne comanda l'esecuzione, il che se gli si aggiunge un editor di testi permette di creare un *ambiente di sviluppo unico*. Tuttavia un programma interpretato, in esecuzione, richiede più memoria ed è meno veloce, a causa dell'*overhead* introdotto dall'interprete stesso: l'interprete deve infatti analizzare le istruzioni a partire dal livello sintattico, identificare le azioni da eseguire (eventualmente trasformando i nomi simbolici delle variabili coinvolte nei corrispondenti indirizzi di memoria) ed eseguirle, mentre le istruzioni di un programma compilato, già in linguaggio macchina, vengono caricate e istantaneamente eseguite dal processore. In compenso, la maggiore rapidità dell'interpretazione di un programma rispetto al ciclo compilazione ed esecuzione può costituire un vantaggio durante lo sviluppo, specialmente durante il debugging: la maggior parte degli interpreti consentono all'utente di agire sul programma in esecuzione sospendendolo, ispezionando o modificando i contenuti delle sue variabili, e così via, in modo spesso più flessibile e potente di quanto si possa ottenere da un debugger.

## Editor

Gli editor sono programmi che interagiscono con l'utente e gli permettono di creare e modificare testi, visualizzandoli e consentendo l'inserimento, la cancellazione, la ricerca, la sostituzione, la copiatura di caratteri e altro. Alcuni editor pensati per i programmatori suggeriscono direttamente i costrutti corretti per il linguaggio che si sta utilizzando con un unico comando: in questo modo è più semplice garantire la correttezza sintattica del codice sorgente che dovrà essere in seguito compilato.

## Compilatore

Affinché possa essere generato il codice oggetto, il compilatore esegue delle operazioni sul sorgente per verificare l'assenza di errori o, in caso contrario, segnalarli all'utente. Queste operazioni si possono suddividere in:

- **Analisi lessicale:** il testo viene scansionato per individuare gli elementi di base del linguaggio (parole chiave, variabili, costanti, delimitatori ecc); dopodiché il file sorgente viene suddiviso in stringhe di simboli (detti *token*) e viene creata una *tabella dei simboli* per le variabili.
- **Analisi sintattica:** sulla base della grammatica del linguaggio di programmazione, le stringhe prodotte in precedenza vengono elaborate per capire come è strutturato il programma: vengono così generate una *forma intermedia* (generalmente una *struttura ad*

*albero oppure a matrice*) corrispondente alla sintassi del programma e delle *tavole dei simboli* con tutti gli identificatori.

- **Analisi semantica:** sulla base della forma intermedia generata nel passaggio precedente, viene verificato il corretto impiego del linguaggio, ossia la *compatibilità di tipo* delle espressioni e delle conversioni e la *dichiarazione degli identificatori*. Dall'analisi semantica è possibile individuare lo scopo del programma.
- **Generazione del codice e ottimizzazione:** dalla precedente forma intermedia viene generato un codice di basso livello, ma ancora lontano dal codice macchina necessario per l'esecuzione da parte della CPU: a questo punto è possibile che sia presente una fase di ottimizzazione del codice in grado di rendere più efficiente il programma, agendo sulla sua dimensione e quindi il relativo tempo di calcolo (per esempio, sostituendo ad alcune espressioni altre di più immediata realizzazione mantenendo la stessa funzionalità). Infine può essere quindi tradotto nelle istruzioni della particolare CPU: la sequenza delle varie fasi della compilazione e il loro peso dipendono dal tipo di compilatore.

Così come i compilatori traducono in codice eseguibile programmi scritti in un linguaggio scritto ad alto livello, gli **assemblatori** traducono programmi scritti nel linguaggio *assembly* (di basso livello) proprio del processore specifico. Mentre per i linguaggi ad alto livello ad ogni *statement* corrispondono più istruzioni eseguibili da parte del microprocessore, per quelli a basso livello si ha una *corrispondenza 1:1* fra statement del file sorgente e istruzioni. Gli assemblatori sono tipicamente a due passate: nella prima viene effettuata l'analisi delle istruzioni e creata la tavola dei simboli, nella seconda viene prodotto il codice in linguaggio macchina per il processore.

I macroassemblatori ammettono *macroistruzioni*, ossia istruzioni definite dal programmatore a cui vengono fatti corrispondere più statement del linguaggio assembly.

I **precompilatori** (o **preprocessori**) sono programmi che, una volta verificata la correttezza lessicale, sintattica e semantica, operano una traduzione delle istruzioni scritte in un linguaggio di alto livello (codificato quindi con una propria grammatica) a istruzioni di un altro linguaggio, generalmente di livello inferiore, per cui esiste un compilatore. Un esempio di utilizzo massiccio del preprocessore si ha nel linguaggio C, dove le *direttive del preprocessore* (facilmente identificabili dall'`#` iniziale) si dimostrano assai utili per dichiarare le funzioni di libreria utilizzate, per definire macro e anche per definire piccole routine espandibili all'occorrenza.

I **metacompilatori**, invece, sono strumenti che, a partire dalla definizione formale di un linguaggio (la sua grammatica), sono in grado di creare un compilatore per il linguaggio stesso (più frequentemente un precompilatore); un esempio di metacompilatore è YACC per Unix, che genera un analizzatore sintattico per un codice scritto con una notazione BNF.

## Linker

Il *linker* è il programma che si occupa di unire più file oggetto compilati separatamente in un singolo modulo eseguibile, con l'aggiunta eventuale di librerie software preconfezionate; grazie ad esso è possibile accoppare parti diverse di uno stesso lavoro, suddivise in fase di progettazione o realizzate

da persone diverse. Uno dei possibili utilizzi del linker consiste nel mettere insieme file oggetto prodotti utilizzando linguaggi di programmazione differenti, in modo da sfruttare al meglio i vantaggi di ciascuno di essi.

### Loader

Il *loader* è il programma che si occupa di trasferire in memoria l'eseguibile, caricarne l'indirizzo della prima istruzione nel Program Counter e farne partire l'esecuzione.

L'attività del loader tuttavia dipende anche dal tipo di codice contenuto nell'eseguibile, che può essere:

- *codice binario assoluto*: l'indirizzo in cui verrà posto il programma in memoria è noto al linker, quindi tutti gli indirizzi dei dati necessari al suo funzionamento vengono adattati di conseguenza; questo caso può verificarsi ad esempio in alcuni sistemi dedicati uniprogrammati;
- *codice binario rilocato*: tutti gli indirizzi sono riferiti all'indirizzo di origine del programma (indirizzo 0) cioè sarà poi il loader a riaggiornarli in seguito tenendo conto dell'indirizzo reale oppure utilizzando tecniche di indirizzamento indicizzato o con spostamento rispetto al valore attuale del PC.

### Debugger

Un *debugger* è un programma progettato per l'analisi dell'esecuzione dei programmi, affinché possa esserne verificata o meno la correttezza del funzionamento ed eventualmente facilitarne il risolvimento da parte del programmatore: per questa ragione è possibile per esempio permettere l'esecuzione passo-passo inserendo dei punti di arresto (*breakpoint*) o la lettura e la modifica dei contenuti dei registri della CPU e della memoria, oppure segnalare il verificarsi di eventi e, nel caso di *debugger simbolici*, provvedere immediatamente alla risoluzione del problema consentendo di operare sul codice sorgente. Il funzionamento del debugger è strettamente legato alle informazioni attinte dalle tabelle generate dal compilatore e dal linker, senza le quali non sarebbe possibile il suo utilizzo.

## Linguaggi di programmazione

### Classificazione

I linguaggi di programmazione possono essere suddivisi in:

- **Prima generazione (linguaggio macchina)**: è il linguaggio le cui istruzioni sono direttamente eseguibili dalla CPU, utilizzando la codifica binaria di codice operativo e operandi: tuttavia i programmi non sono trasportabili, in quanto eseguibili solo sugli elaboratori equipaggiati con la CPU corrispondente al linguaggio. Nonostante si dimostri, se ben utilizzato, il più efficiente in assoluto, un programma scritto in questo linguaggio è di difficile realizzazione per via della pessima leggibilità, che può anche portare facilmente a commettere errori.

- **Seconda generazione (linguaggi assemblativi):** sono linguaggi che, tramite brevi termini mnemonici, rappresentano codici operativi e operandi (precedentemente binari) in forma simbolica: le singole istruzioni sono nuovamente quelle eseguibili dalla CPU ed è stata aggiunta la possibilità di fare riferimento alle variabili tramite il loro nome e definire e utilizzare macroistruzioni, pur mantenendo il più elevato livello di efficienza caratteristico di quelli di prima generazione. Tuttavia, per la traduzione in linguaggio macchina è necessario un assemblatore e i programmi sono utilizzabili solo su calcolatori dotati della specifica CPU corrispondente al linguaggio (non è possibile utilizzarli su altre macchine); la programmazione resta comunque difficile, di scarsa leggibilità e con ampio margine di errore.
- **Terza generazione (linguaggi di alto livello):** sono linguaggi progettati per essere facilmente comprensibili dagli esseri umani tramite l'utilizzo di costrutti logici, che fanno della facilità d'uso e di leggibilità il loro punto di forza: le istruzioni dunque esprimono operazioni significative nella logica del programmatore ma assai lontane da quelle realmente eseguite dalla CPU. Per consentire l'esecuzione di programmi scritti con questi linguaggi è necessario che un compilatore o un interprete traduca ogni istruzione in una sequenza di istruzioni macchina: un altro vantaggio dei linguaggi di alto livello è la possibilità di utilizzare lo stesso programma su tutti gli elaboratori che dispongono di un traduttore per tale linguaggio. Una delle poche note di demerito di questo tipo di linguaggi è il ridotto livello di efficienza rispetto ai linguaggi assemblativi, anche se è possibile ricorrere a ottimizzazioni in grado di sopperirvi in buona misura.

## Scelta e utilizzo dei linguaggi

La scelta del linguaggio di programmazione va operata sulla base di diversi parametri, quali l'adattabilità al problema che si è intenti a risolvere, la facilità di uso e di apprendimento dello stesso, la comprensibilità dei programmi realizzabili (in vista di una eventuale modifica successiva), la portabilità dei programmi su altre macchine, l'efficienza dei programmi stessi e l'esistenza di librerie e software di supporto per il programmatore.

Per le loro caratteristiche, i linguaggi di alto livello possono essere essenzialmente classificati in:

- **Procedurali:** a differenza degli altri tipi di linguaggi di terza generazione, i *linguaggi procedurali* sono basati sul modello computazionale di Von Neumann, in cui il programma viene interpretato come una sequenza di istruzioni tendenti a modificare i dati contenuti nella memoria. Essendo di alto livello, sono indipendenti dalla macchina su cui si lavora e dispongono di una sintassi vicina al linguaggio naturale, che gli conferiscono un buon grado di leggibilità: sono basati su istruzioni e assegnazioni e permettono inoltre di utilizzare dei nomi simbolici per i dati e di fare astrazioni sugli stessi, definendo tipi di dati ed eseguendo operazioni sui tipi. È possibile inoltre determinare l'ordine di esecuzione delle istruzioni mediante strutture di controllo (come cicli *for*, *while*, *do*) e sottoprogrammi (*funzioni* e *procedure*). Questi linguaggi possono essere ulteriormente suddivisi in tre categorie: di uso generale (BASIC, C, COBOL, ecc), dotati di grande versatilità nella risoluzione di problemi;

di uso speciale (CHILL, SIMULA, APT, ecc), in grado di svolgere compiti specifici; per applicazioni Web (HTML, PHP).

- **Funzionali:** sono linguaggi che si approccia alla computazione come allo svolgimento di una serie di funzioni matematiche: rispetto ai procedurali, che invece prediligono la specifica di una sequenza di comandi da eseguire e i cui valori vengono calcolati attraverso delle assegnazioni, in un programma funzionale i valori vengono trovati elaborando nuovi dati a partire dai precedenti. Il punto di forza principale di questi linguaggi è la mancanza di effetti collaterali delle funzioni, il che comporta una più facile verifica della correttezza e della mancanza di bug del programma e la possibilità di una maggiore ottimizzazione dello stesso. Un altro loro particolare utilizzo per l'ottimizzazione dei programmi potrebbe consistere nel trasformare gli stessi per utilizzarli nella programmazione parallela.
- **Logici:** sono linguaggi che, come suggerisce il nome, si basano sulla logica formale e si concentrano sulla descrizione di ciò che si sa, piuttosto che uno specifico procedimento di risoluzione: i programmi scritti con questi linguaggi di programmazione, infatti, sono costituiti da una sequenza di istruzioni logiche che esprimono una serie di *fatti* (che descrivono situazioni sempre vere) e *regole* (che permettono di dedurre nuove situazioni vere sulla base dei fatti a disposizione) per quanto riguarda il dominio del problema. Fornendo al programma una serie di informazioni vere e aggiungendo un'altra serie di regole che descrivono il modo in cui le informazioni si combinano tra di loro, è possibile giungere alla soluzione del problema. Il linguaggio logico di maggior impiego è il PROLOG.
- **Orientati agli oggetti:** permettono di definire oggetti software in grado di interagire gli uni con gli altri attraverso lo scambio di informazioni: le dichiarazioni delle strutture dati e delle procedure che operano su di esse sono raggruppate in zone circoscritte del codice sorgente (chiamate *classi*), che vengono utilizzate per creare *oggetti* definiti da *attributi* (dati) e *metodi* (procedure). I linguaggi di programmazione definiti ad oggetti permettono di implementare tre meccanismi usando la sintassi nativa del linguaggio: l'*incapsulamento*, che consiste nella separazione della cosiddetta *interfaccia* di una classe (insieme dei dati e dei metodi visibili dall'esterno) dalla corrispondente implementazione; l'*ereditarietà*, che permette essenzialmente di definire delle classi a partire da altre già definite; il *polimorfismo*, che permette di scrivere una parte di codice che fa uso di un oggetto (*client*) in grado di servirsi di oggetti di classi diverse ma dotati di una stessa interfaccia comune. Esempi di linguaggi orientati ad oggetti sono il C++, Java, C#, ecc.

I **sottoprogrammi** sono costrutti sintattici dei linguaggi di programmazione costituiti da sequenze di istruzioni che vengono eseguite ogni volta che avviene una *chiamata* al sottoprogramma specifico, identificato con il proprio nome: in questo modo è possibile scomporre un problema complesso in problemi più semplici e di più facile comprensione. Il grande vantaggio dei sottoprogrammi consiste nel dover scrivere una volta sola una parte di codice che potrà essere utilizzata più volte, ad esempio al variare dei parametri: grazie ai sottoprogrammi, inoltre, è possibile creare programmi meglio strutturati e più aperti a eventuali modifiche e raffinamenti successivi. Raggruppando un insieme di

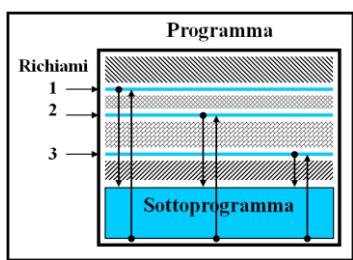
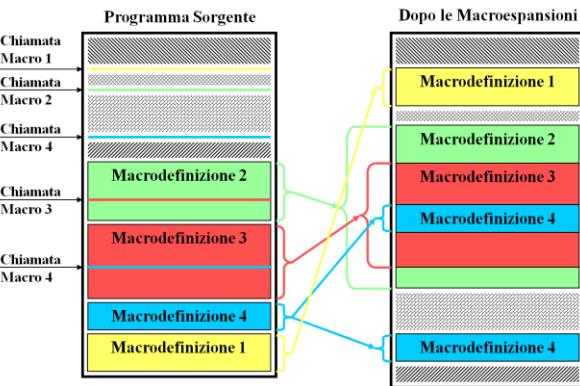
sottoprogrammi in librerie preconfezionate è possibile semplificare ulteriormente le operazioni di sviluppo e manutenzione, evitando al programmatore di dover riscrivere ogni volta le stesse funzioni o strutture dati.

Con *parametri formali* si intende l'insieme dei simboli che rappresentano i dati con cui il sottoprogramma opera e di solito si trovano specificati nella definizione del sottoprogramma stesso; i *parametri attuali*, invece, sono i dati, corrispondenti ai parametri formali, che effettivamente il programma usa e sono specificati nella chiamata al sottoprogramma.

Possono essere definiti due tipi di sottoprogrammi: *aperti* e *chiusi*.

I **sottoprogrammi aperti**, anche detti *macroistruzioni*, sono presenti in tutti i punti in cui vengono chiamati con tutte le istruzioni che definiscono il sottoprogramma stesso: questo avviene per via del meccanismo della *macroespansione*, che provvede a sostituire ai singoli richiami la serie di istruzioni che costituiscono il sottoprogramma, mentre ai parametri formali vengono sostituiti i dati che effettivamente rappresentano, cioè i parametri attuali. La definizione di questi sottoprogrammi e la loro chiamata sono detti *macrodefinizione* e *macrorichiamo*; se una macroistruzione contiene richiami ad altre macroistruzioni, si ha a che fare con *macroistruzioni nidificate*.

I **sottoprogrammi chiusi**, invece, compaiono una volta sola con tutte le loro istruzioni, ma vengono adoperati solo nei punti del programma in cui vengono chiamati. Anche in questo caso si possono avere *sottoprogrammi nidificati*, in cui un sottoprogramma viene chiamato da un altro: esiste inoltre la possibilità, in alcuni linguaggi di programmazione che lo consentono, di avere *chiamate ricorsive*, cioè in cui un sottoprogramma richiama se stesso. I sottoprogrammi chiusi presentano tuttavia alcuni problemi che portano a preferire quelli aperti dal punto di vista dell'efficienza: la gestione delle chiamate e il passaggio dei parametri sia in ingresso (da chiamante al sottoprogramma chiamato) che in uscita (i risultati ottenuti dal sottoprogramma vengono restituiti alla funzione chiamante). Ogniqualvolta un sottoprogramma viene chiamato, l'indirizzo di rientro (cioè l'indirizzo dell'istruzione successiva a quella che ha effettuato la chiamata) deve essere memorizzato nello *stack* (o pila), cioè la zona di memoria gestita con la logica LIFO (Last In First Out); quando il sottoprogramma termina l'esecuzione, viene letto dallo stack l'ultimo indirizzo caricato in quella porzione di memoria che non è stato ancora prelevato per essere poi trasferito nel Program Counter, affinché possa essere ripresa l'esecuzione dove vi era stato il salto.



Questo metodo di gestione LIFO della pila è fondamentale per permettere di nidificare le chiamate ai sottoprogrammi, anche se tuttavia richiede un passaggio ulteriore rispetto alle macroistruzioni.