

FONDAMENTI DI INFORMATICA

INFORMATICA: il termine informatica deriva dalla combinazione di INFORmazione autoMATICA ed è riferito alla scienza che studia il trattamento dell'informazione mediante processi automatizzabili.

È infatti intesa come informatica tutta quella disciplina non soltanto inerente ai pc e ai calcolatori general purpose, ma anche applicazioni in ambito di affari generali, banche, contabilità, simulazioni varie, pianificazioni....

Con **CALCOLATORE** infatti si intende tutto ciò che è concepito per l'elaborazione dei dati (non necessariamente una macchina calcolatrice o un pc, ma anche il microcontrollore di una lavatrice o di un frigorifero automatizzati)

HW: ciò che di un calcolatore si può vedere e toccare, la parte tangibile (e.g. schermo, processore, memorie etc etc)

SW: la parte immateriale di un calcolatore, come possono essere tutti i programmi che girano sul calcolatore o l'insieme di istruzioni che compongono le azioni che svolge il calcolatore.

I/O: dispositivi di ingresso e uscita verso/dal sistema che ne permettono il controllo o ne attuano i dati (e.g. schermo, mouse, tastiera, stampante etc etc)

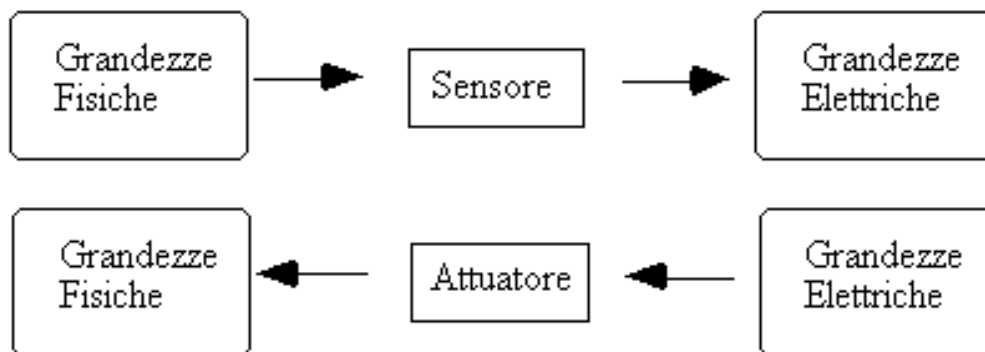
FIRMWARE: componenti hw pre-programmati

RETI: la comunicazione di più calcolatori mediante processi sw o componenti hw è una rete di calcolatori

ARCH. DI UN SISTEMA DI ELAB. Insieme dei moduli, delle soluzioni e delle tecniche che regola l'interazione tra hw, sw e l'interfaccia verso l'uomo.

ARCH. DI RETE: insieme dei moduli, delle soluzioni e delle metodologie tecniche che regolano l'interazioni tra calcolatori e la rete e tra i calcolatori stessi

Il calcolatore agisce soltanto con 0 e 1 e pertanto necessita di un qualche sistema che traduca l'informazione fisica in bit (*trasduttore*) e che, viceversa, traduca i bit in istruzione attuabile fisicamente (*attuatore*)



Il calcolatore inoltre è una macchina **sincrona**, ovvero lavora a istanti regolari. In ogni istante un calcolatore esegue delle operazioni; maggiori sono le operazioni che un processore riesce a svolgere in un secondo, maggiore è la potenza di calcolo associata al suddetto processore

Il periodo in cui viene eseguita un'operazione è detto **clock**.

Ciò che accade all'interno del calcolatore dipende dalla **frequenza**, ovvero dagli **Hz** (# di cicli al secondo)

obs: - se in ogni clock si modificano migliaia di bit basta che uno di questi sia errato per generare un errore ricorsivo all'interno del sistema

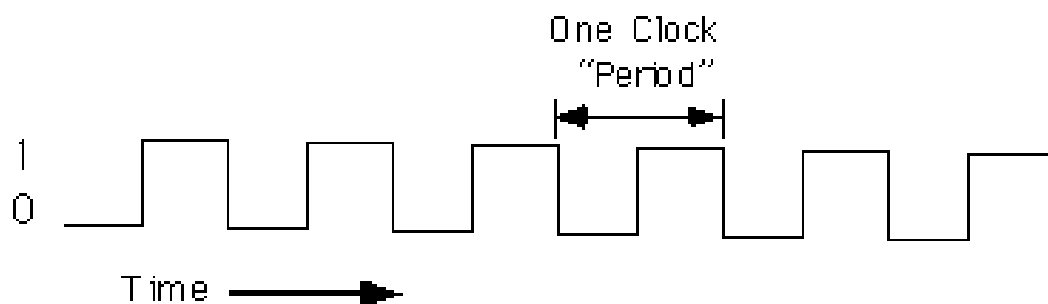
- se raddoppio il clock di una cpu essa raddoppierà i cicli al secondo, ma non necessariamente il resto del sistema le starà dietro. Dipende dall'aggiornamento di tutti i componenti o meno

MIPS: Mega Instructions Per Seconds ovvero il # di milioni di istruzioni al secondo (NB non confondere con Mhz)

MFLOPS: Mega Floating Point Operations Per Second ovvero milioni di operazioni in v. mobile per secondo

BAUD:: velocità di trasmissione dei dati, bit/sec

LINEA DI RESET: è una linea presente in ogni calcolatore che ha il compito, in caso di avaria, di chiudere un c.to che porta al reset della macchina. Può interfacciarsi sia con l'essere umano sia, come accade in ambienti ostili, con un **watch dog**, ovvero un dispositivo che quando rileva che il calcolatore a cui è associato non sta aggiornando i dati che si aspetta aggiorni procede a chiudere i c.to e resettare la macchina



Questo modo di eseguire operazioni a intervalli permette anche al processore di “capire” che cosa sta eseguendo. Lavorando esso infatti soltanto con 0 e 1, per evitare di avere un alfabeto di codifica troppo articolato, una sequenza come può essere 0110 può significare molte cose: - un numero - un comando - un pixel e il suo colore - un carattere etc etc

Dipendentemente dal contesto di esecuzione di un programma e da cosa ci si attende dal comando successivo il calcolatore sa dare il giusto significato a ciascuna sequenza di bit (e.g. se so che una data sequenza sta ad indicare il comando Add successivamente mi aspetto di ricevere un numero, non una lettera o un pixel)

In questo modo con un numero relativamente limitato di bit è possibile rappresentare un'infinità di concetti differenti, invece che dedicare a ciascun concetto una sequenza differente

SCHEMA A STRATI:

è un modo di procedere, all'interno dell'informatica, per il quale ogni utente conosce soltanto lo strato che interessa a lui e ha, al limite, una nozione sul prodotto finito

Tramite un processo di *astrazione* chi lavora in un determinato campo, e.g. lo sviluppo sw, non deve necessariamente avere competenze riguardo allo sviluppo hw del calcolatore. Ogni processo infatti è descritto tramite strumenti astratti, senza entrare nel dettaglio di ciò che lo compone

Applicazioni	Ms-word
Sw di base/ Sw comunicativo	Windows
Hw + Firmware	Pentium + Assembly

UNITÀ DI MISURA:

nel campo informatico l'unità fondamentale di misura è il BIT (ovvero un valore di tensione **alto** o **basso**) corrispondente a 1 o 0 (e, dal punto di vista logico, a *true* and *false*)

Proprio per questa loro corrispondenza con valori logici prima che numerici implica che si debba usare un procedimento di calcolo non convenzionale. L'analisi di V e F avviene attraverso l'**algebra di Boole**: un processo che associa alle varie combinazioni di 0 e 1 un output di 0 o 1 (And, Not, Xor)

- Byte 8 bit
- Kbyte: 2^{10} byte = 1024 byte
- Mbyte: 2^{20} byte = c.ca 1 mln byte
- Gbyte: 2^{30} byte = c.ca 1 mld byte

N.B.: i prefissi, a differenza del sistema di numerazione decimale, non indicano un passaggio di potenze dieci tra un prefisso e l'altro ma un passaggio di 2^{10} (trattandosi di valori in base 2)

Il fatto di avere soltanto due cifre per rappresentare l'informazione implica il fatto che non sempre l'informazione rappresentata sia corretta. Se ad esempio esegui una somma a 4 bit:

1111 +

0001 =

0000 → che sarebbe errato nel caso di calcolo in v. assoluto (avrei che sommando due numeri otterrei zero. Ma potrebbe essere anche esatto qualora fossi in complemento a 2)

È quindi necessario *combinare il minor numero di bit possibile con l'informazione rappresentata correttamente*

L'associazione simboli/caratteri ↔ bit è detto **ASCII** e, basandosi su rappresentazioni a 8 bit, da una possibilità di rappresentazione di 2^8 caratteri (256)

ESEMPI RAPPRESENTAZIONI

con la rappresentazione ASCII, in cui un byte rappresenta un carattere, per rappresentare una pagina di 2k caratteri necessiterà di 2Kbyte.

Similmente un libro di 500 pagine necessiterà di c.ca 1Mb

Supponendo ora di voler rappresentare la stessa pagina scannerizzandola con uno scanner da 300 ppi (2475 x 3450) avrei un'immagine che per una pagina (a toni di grigio) occuperebbe 8.5Mb. Similmente il libro occuperebbe c.ca 25Mb. Se decidessi di rappresentarla a colori (solitamente 8 bit per RGB) avrei ancora più occupazione di memoria

Una risoluzione di un monitor 1280x960 genera un'immagine di 1.2 Mb

(obs 1 pollice = 2.54 cm)

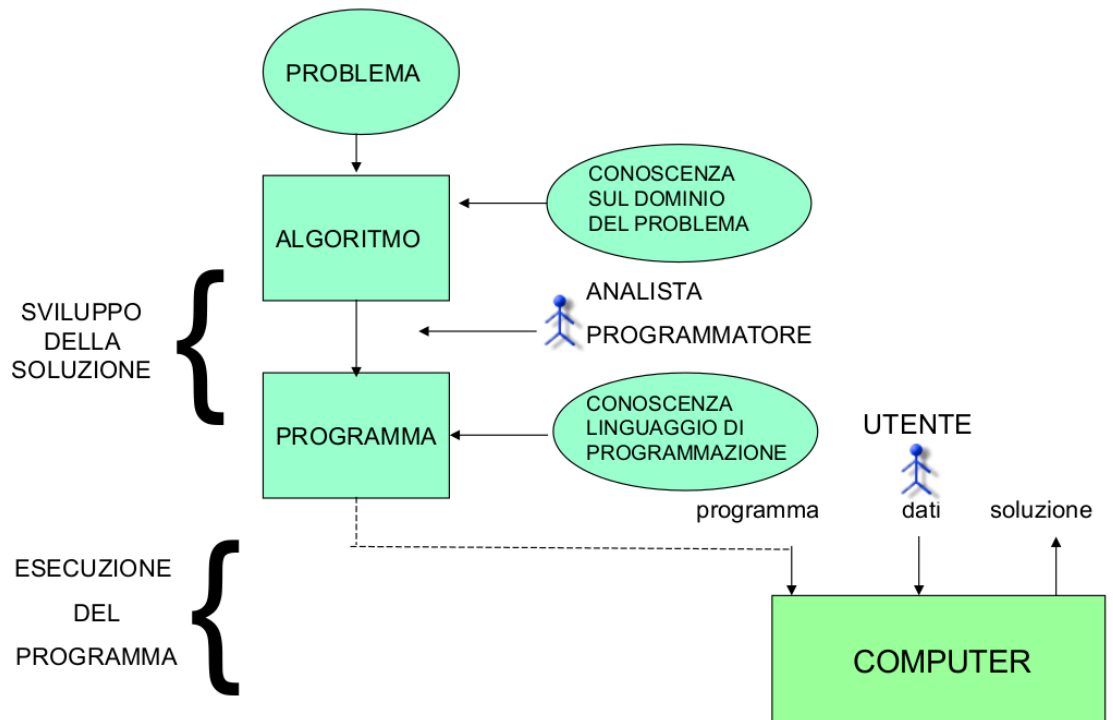
CLASSI DI CALCOLATORI:

- Personal computer
- Workstation
- Minicomputer (più utenti contemporaneamente)
- Mainframe (centinaia di utenti)

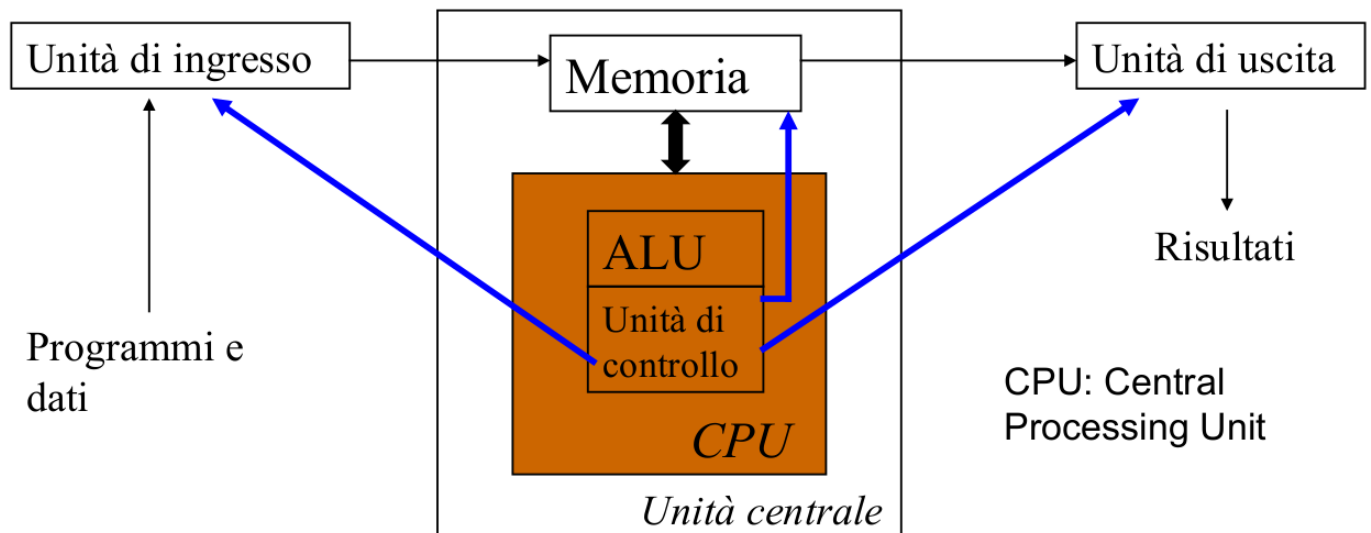
- Supercomputer
e architetture
parallele

RUOLO DEL PC E
DEGLI
ALGORITMI:

Un **algoritmo** è
uno schema di
calcolo che
risolve un
qualsivoglia
problema con un
numero finito di
passaggi
elementari



MACCHINA DI VON NEUMAN



È un'architettura hw per calcolatori digitali a programma memorizzato, in cui i dati del programma e le istruzioni del programma sono nello stesso spazio di memoria.

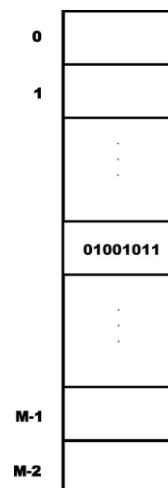
N.B. architettura ≠ tecnologia. Mentre la prima dipende dallo scopo della macchina la seconda dipende dalle innovazioni tecnologiche dei componenti che la compongono. (L'innovazione è continuata nei componenti, ma non nella struttura. Rimasta invariata dal suo concepimento ad oggi)

Essa si compone di:

- **Cpu:** è a sua volta composta da *unità di controllo* (ovvero quell'unità che monitora ogni processo eseguito all'interno dell'unità centrale) e da *ALU* (ovvero l'Arithmetic Logic Unit, l'unità logico-aritmetica che esegue effettivamente le operazioni sui bit)
 - **Memoria:** anche lei contenuta nell'unità centrale, immagazzina al suo interno tutte le informazioni sul programma e sulla sua esecuzione. (A differenza della macchina Harvard in cui istruzioni e dati sono divisi)
- (con memoria si intende la memoria di massa, ovvero la ROM con le istruzioni e la RAM. Le altre memorie sono considerate dispositivi i/o)
- **i/o:** ovvero i disp di input e output per l'immissione di dati e la lettura dei risultati

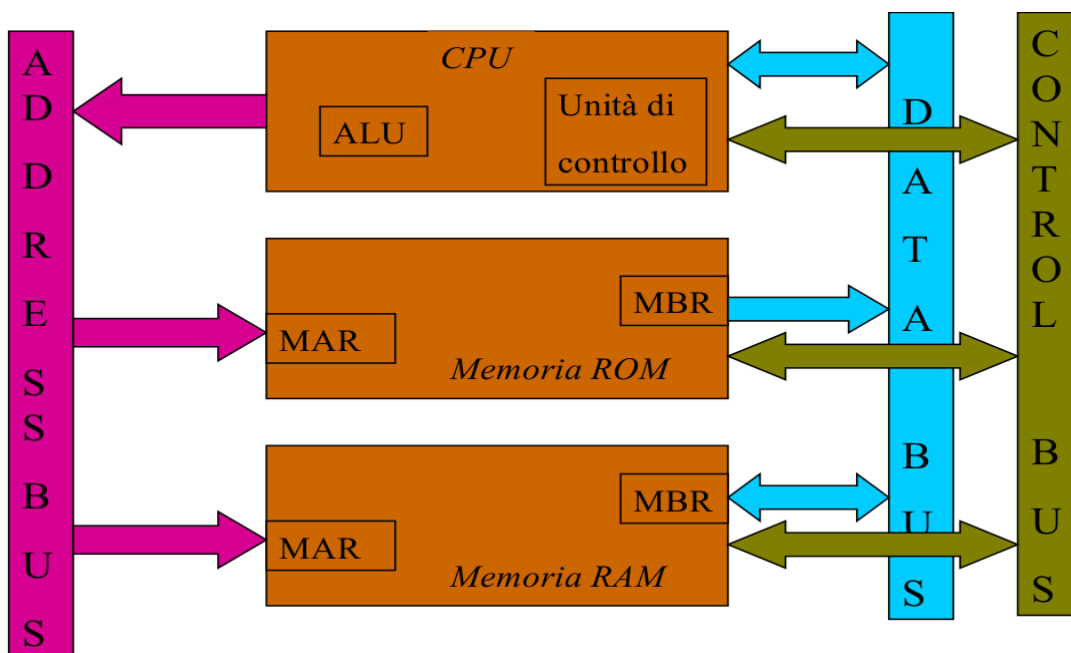
All'interno della ALU è presente un *accumulatore* che fa da ponte tra i/o grazie ad una speciale istruzione che carica una parola dalla memoria all'accumulatore e viceversa. E' un registro che contiene informazioni riguardo alle operazioni che vengono svolte all'interno della ALU

La potenza dell'architettura di Vn la si può notare dal fatto che essa sia ancora la base di molti calcolatori attuali nonostante sia stata pensata quasi 70anni fa. Ciò è dovuto al fatto che essa non regola soltanto gli insiemi coinvolti nel calcolatore, ma anche *l'architettura logica interna degli stessi* (disposizione delle porte logiche)



In una macchina di Vn la memoria è unica e contiene sia i programmi da eseguire sia l'istruzione per tali programmi. La *scrittura* attiva un segnale di invio di informazione mentre la *lettura* un segnale di ricezione dell'informazione

Il sistema della Vn è un master/slave in cui la CPU è il master che comanda i componenti che la circondano (slaves)



N.B. Il mondo esterno è fortemente *asincrono*, a differenza della CPU. E' pertanto necessario che trasduttori e attuatori lavorino al meglio per una conversione di dati reali asincroni in bit operati sincronicamente e viceversa

Esistono varie tipologie di memoria. Le due principali in un calcolatore sono:

-Rom: Read Only Memory (con significato di sola lettura per il processore) contiene le informazioni di funzionamento del sistema (stabilite dal progettista) e non può essere scritta (ha infatti soltanto frecce in uscita). E' inoltre importante che non venga persa l'informazione contenuta al suo interno, a differenza della Ram in cui, tolta l'alimentazione, perdo l'informazione.

-Ram: Random Access Memory (memoria ad accesso casuale) in cui vengono immagazzinati dati temporanei di un'operazione parziale che mi serviranno in futuro. Una volta tolta l'alimentazione i dati in ram vengono persi. E' molto più veloce di una Rom ma anche molto più costosa in termini produttivi.

N.B. Random non indica effettivamente la casualità dell'accesso, ma sottolinea come, a differenza delle memorie sequenziali, il tempo di lettura di una cella sia il medesimo a prescindere dalla posizione della cella. (scelta una cella a caso il tempo di lettura sarà il medesimo per ogni cella, nelle sequenziali più ci si allontana dalla prima cella più si alzano i tempi di lettura)

La quantità di Ram e Rom da inserire in un calcolatore dipendono dalla finalità del calcolatore stesso: Un computer che necessiterà di eseguire programmi molto complessi ma immagazzinare poca informazione finale avrà bisogno di molta Ram e poca Rom. Al contrario un controllore per lavatrici, ad esempio, non necessiterà di molte risorse in termini di Ram e Rom

Address Bus

E' il bus unidirezionale attraverso il quale la Cpu decide in quale indirizzo andare a leggere/scrivere le informazioni in memoria. Sono filamenti di rame (o materiale conduttore) con valore 0 o 1 disposti in parallelo che comunicano con le celle di memoria interessate. Serve *soltanto* per la cpu per comunicare con l'indirizzo corretto. E' visto in scrittura soltanto dalla Cpu e in lettura da tutti gli altri componenti (poiché in questo bus vengono definite le allocazioni di memoria decise dalla Cpu)

Solitamente è formato da condutture in rame di 0 e 1 della casella di interesse. Se ad esempio avessi 10 linee di bus avrei 2^{10} interlocutori

(il numero di linee dipende dal numero di interlocutori con cui si sta parlando)

Gli odierni sistemi hanno bus a 64 bit (2^{64} allocazioni possibili, molte maggiori della capienza delle memorie odierne) seguendo l'equazione $N \text{ bus} \rightarrow 2^N \text{ configurazioni possibili}$. Ovviamente maggiore è il numero di linee maggiore sono lo spazio occupato e i consumi dovuti al passaggio di corrente che ne conseguono. (è pertanto necessario ottimizzare il numero di linee a seconda dell'utilizzo che se ne vuole fare)

Il bus degli indirizzi indica **la capacità di memorizzazione del processore**, ovvero quante celle di memoria raggiungibili in relazione al numero di address bus

Data Bus

E' un bus per la sequenza di bit che rappresentano ciò su cui la Cpu deve lavorare, ovvero i dati da elaborare (intesi sia come dati veri e propri che come istruzioni per l'esecuzione degli stessi)

N.b la bidirezionalità delle frecce *non indica una simultaneità nella scrittura/lettura* del bus (impossibile avere scrittura e lettura simultanee sullo stesso bus) ma la possibilità di Cpu e Ram di leggere e scrivere sul detto bus in tempi alternati

I processori più ampiamente diffusi hanno data bus a 8 bit (sufficienti per la maggior parte degli utilizzi), permettendo una rappresentazione di $2^8 = 256$ valori in v.a

Può capitare di avere Cpu che lavorano a 128 bit con data bus a 64 bit max: ciò avviene non per una tecnologia scadente negli altri componenti ma per questioni di spazio e di interferenze → 128 data bus e address bus occupano qualche volta di più degli equivalenti a 64 con conseguente spreco di spazi, aumento dei consumi e delle interferenze tra fili adiacenti (ogni conduttura genera campo che può causare interferenze tra fili adiacenti). Si tende quindi ad ottimizzare il # di data bus e address bus in relazione alla finalità del calcolatore

Il bus dei dati indica **la capacità di elaborazione** del processore (ovvero quanti bit possono essere elaborati in parallelo)

e.g.

In una produzione automatizzata è necessario avere misure di precisione riguardo al peso del caffè impacchettato (1 kg) con una precisione sufficiente da non generare perdita nel profitto (richiesta 0,1g di

Numero di bit bus dati	4	8	16
Dati rappresentabili	$2^4=16$	$2^8=256$	$2^{16}=65536$
Precisione relativa			6.25%
~3.9%	~0.015%		
Precisione max.			62.5 gr
~3.9 gr	~0.015 gr		

Tenendo conto dell'influenza della quantizzazione e del suo passo si vede come sembri impossibile avere la precisione desiderata con 8 bit

→ → N.B la capacità del calcolatore e la precisione che posso raggiungere NON dipende dalla capacità di elaborazione. Se devo elaborare a 16 bit per ottenere la precisione desiderata, ma ho soltanto data bus da 8 bit non farò altro che mandare due tranches di dati al mio elaboratore in due momenti diversi. Cambia il tempo, in modo impercettibile, ma non il risultato finale

Quindi il *numero di bit dei data bus non influisce sulla precisione e sul tipo di calcolo che devo andare a fare, semplicemente differenzia la velocità con cui lo eseguo*. Se infatti devo eseguire un calcolo a 16 bit con data bus da 16 bit lo eseguirò in un unico periodo di clock, se invece ho data bus da 8 bit e devo eseguire un calcolo a 16 bit lo eseguirò in due periodi di clock differenti.

PRO: posso ridurre i costi per un determinato tipo di processo in cui la velocità al milionesimo non è fondamentale (e.g. va bene nei processi automatizzati, nell'Abs può fare la differenza)

CONTRO: vado a perdere qualche milionesimo di secondo nell'esecuzione dell'algoritmo e a dover utilizzare delle memorie leggermente più grandi (spesso in senso insignificante per l'utilizzo, si parla di Kb in più o in meno); un altro contro è la complessità dell'algoritmo risolutivo che necessita di essere più complicato poiché deve gestire il calcolo in due o più tempi differenti

CAPACITÀ DI INDIRIZZAMENTO: indica il numero di celle diverse a cui si può accedere:

> 2^{10} byte = 1 Kbyte = 1024 byte

> 2^{30} byte = 1 Gbyte = 1 mldib byte

Control bus

con chi (*address*) e cosa (*data*) usare non sono sufficienti per l'effettiva comunicazione delle parti. E' necessario che ci sia un regolatore che controlli effettivamente l'esecuzione dei comandi e la sua correttezza. Questo compito è dei *data bus*, che contengono informazioni riguardo le linee di Read e di Write e "decidono" a quale address bus accedere, quale data bus elaborare e come.

Concretamente il Control-Bus prende il comando dalla Cpu e lo restituisce sotto forma di segnali per i dispositivi (e viceversa)

N.B: sul data bus le linee non sono omogenee. (ovvero esistono linee di controllo non in relazione con le altre né per forma né per funzione)

Possono inoltre esserci linee soltanto in ingresso, soltanto in uscita o in ingresso e uscita; Alcune linee possono coinvolgere soltanto alcune parti mentre altre possono riguardare ogni parte del calcolatore.

Possono inoltre non essere soggette al classico schema in cui la Cpu è master e il resto dei componenti slave → e.g. *linea di reset* è un control bus in ingresso in cui la Cpu è slave (subisce il reset) mentre il master è l'utente o il watch dog, a seconda delle situazioni. Altre linee presenti sul bus di controllo sono la *read line* e la *write line*: ovvero le linee di lettura e scrittura su/verso le memorie e la Cpu.

Ci sono comunque delle linee base comuni a tutti i processori:

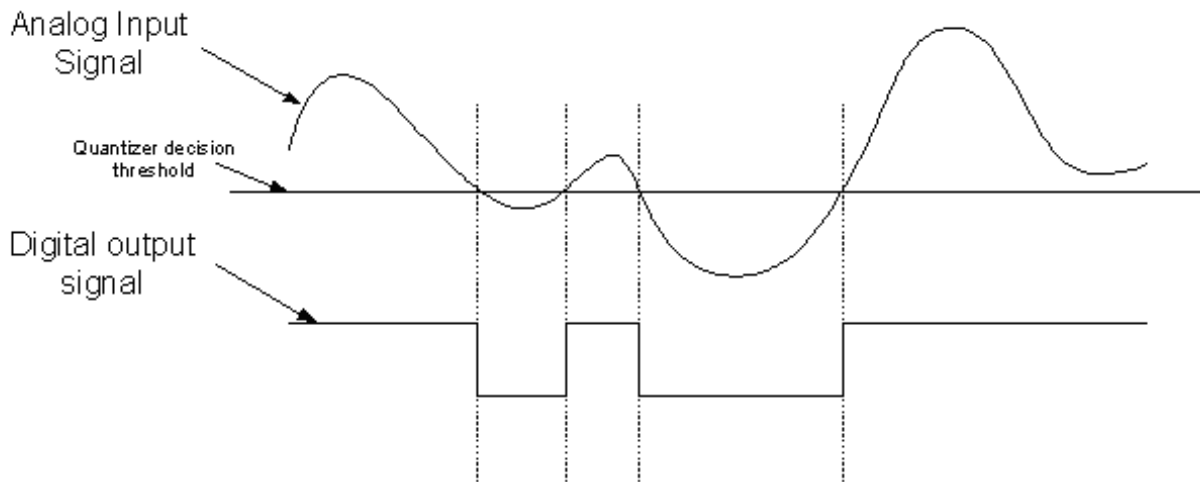
- > *Read*: singola linea che se attiva indica che il device è in *lettura* da parte del processore
- > *Write*: come sopra, ma in scrittura
- > *Byte—Enable*: gruppo di bus che indica la dimensione del dato

Obs: bus dati e bus indirizzi possono avere la stessa lunghezza o meno, al limite faccio passaggio di dati in due tranches con calcolo eseguito in due momenti successivi

MAR: Memory Address Register, ovvero un registro in cui un indirizzo viene incamerato e mantenuto fino a nuova sovrascrittura del registro stesso (il mantenere l'informazione è caratteristica dei registri, il fatto che l'informazione sia un indirizzo del MAR)

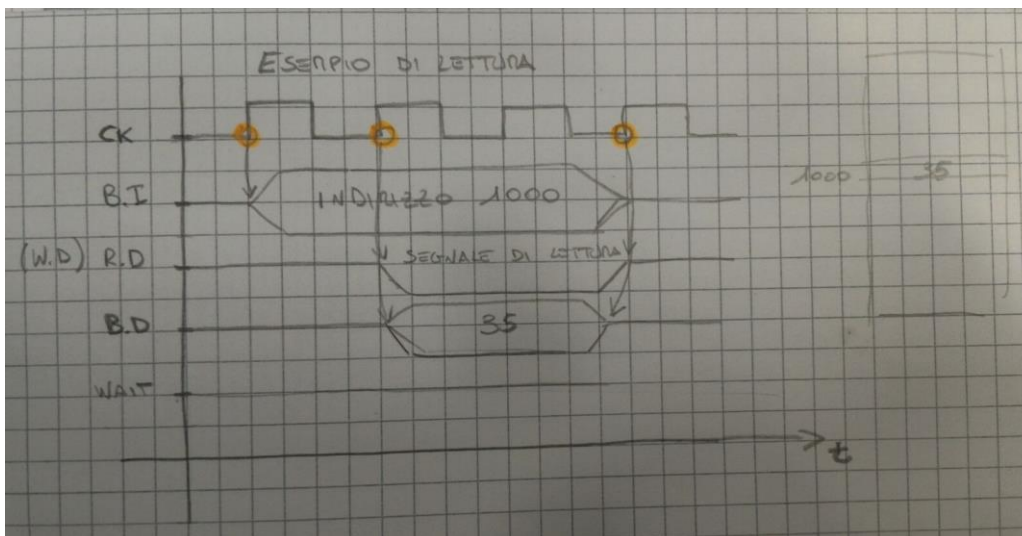
MBR: Memory Buffer Register, ovvero un registro in cui viene inserito il valore della variabile o dell'elemento presente nell'indirizzo selezionato per la lettura o la scrittura; è il registro che fornisce l'informazione alla Cpu e le passa la variabile.

CONVERSIONI: data la non sincronia del mondo reale fisico rispetto alla sincronia del calcolatore e dei suoi processi è necessario che si vada ad eseguire una conversione *Digital to Analog Converter* o, viceversa, una *Analog to Digital Converter*. Sono due dispositivi (DAC / ADC) che permettono di rendere i bit grandezze fisiche tangibili e viceversa



DAC: Il Digital (to) Analog Converter, in italiano Convertitore digitale-analogico, è un **componente elettronico** in grado di produrre sul suo terminale di uscita, un determinato livello di tensione o di corrente, in funzione di un valore numerico che viene presentato al suo ingresso; la risoluzione del DAC è estremamente importante e dipende dall'utilizzo che si deve fare del calcolatore: Si può passare da Dac a 16 bit fino a 24 o addirittura 64 bit. Tuttavia in questo caso aumentando la risoluzione aumento anche il carico di lavoro della cpu, portando ad un rallentamento della velocità di esecuzione del processo stesso.

Un esempio di Dac con quantizzazione decisa a priori è il SAC (Sample And Hold): in questo tipo di convertitore è previsto un delta di errore, che tuttavia sarà sempre minore di quello commesso non tenendo conto della variazione del valore in misurazione; se infatti esso cambia durante la misurazione è possibile riportare valori nettamente errati → liv 1 = 0111 e liv 2 = 1000. Se avviene movimento in misurazione è possibile che venga registrato un valore di 0000 che, ovviamente, è errato. Il Sac congela il movimento e arrotonda al valore quantizzato successivo; in questo modo l'errore presente è piccolo in rapporto alla registrazione completamente errata del valore stesso



Funzionamento di una lettura. Nel grafico si vedono il clock del processore, il bus degli indirizzi che apre la comunicazione, la linea di lettura, il data bus e la linea di wait

Stack: quando c'è una richiesta di interrupt o si deve eseguire una call è necessario che la cpu immagazzini i dati interrotti prima di eseguire la subroutine di servizio. La memoria di Stack è un'area di memoria predisposta a svolgere questo compito. Vengono memorizzati dati con la

tecnica LIFO; nel frattempo un apposito registro del Program-Counter viene aggiornato man mano che vengono aggiunti/tolti dati dalla memoria di stack
Il salvataggio nella stack avviene automaticamente subito prima di effettuare il salto alla subroutine di servizio di una interrupt o una call.

POLLING & INTERRUPT

Come già detto un calcolatore è una macchina perfettamente sincrona e ogni secondo esegue qualche milione di operazione. La velocità del processore tuttavia non sempre corrisponde a quella delle periferiche che deve comandare, sia interne che esterne al calcolatore (e.g. velocità memorie o schede audio/video.). E' pertanto necessario che vi sia sincronismo tra le parti e, per ottenere ciò, vengono introdotti appunto gli *interrupt* e il processo di *polling* → bloccano l'esecuzione di un'istruzione quando questa non può essere eseguita o quando, ad esempio, il processore sta eseguendo calcoli troppo velocemente per le periferiche.

INTERRUPT: gli interrupt sono interruzioni tramite segnale asincrono che indicano la necessità di una periferica di scambio di dati o un segnale sincrono che consente l'interruzione di un processo qualora si verificassero determinate condizioni (generalmente una richiesta al s.o. da parte di un processo in esecuzione)

Esistono due tipologie di interrupt:

> *Interrupt HW:* generati da dispositivi esterni alla CPU che hanno il compito di comunicare il verificarsi di eventi esterni come I/O. Un interrupt hw costringe il processore a memorizzare il suo stato di esecuzione fino all'arrivo dell'interrupt e ad iniziare l'esecuzione della subroutine (al termine della quale il processore riprende l'esecuzione del processo precedentemente interrotto)

obs: è un processo dispendioso per la CPU che può portare ad un rallentamento notevole del sistema con impiego anche totale dell'Unità Centrale

> *Interrupt SW:* istruzioni in linguaggio assembly (e.g. Assembly) che sfruttano il meccanismo delle interruzioni e la gestione dei processi per passare il controllo tra *programma chiamato* e *programma chiamante* e viceversa.

L'*interrupt* di un'istruzione può avvenire per diverse ragioni:

- esecuzione di un'istruzione non valida, come può essere una divisione per zero (che porterebbe ad un crash del sistema) → all'attivazione del flag corrispondente il processo viene interrotto per evitare danni al sistema.
- Richiesta di I/O da parte di un processo, specialmente quando derivante da periferiche interne
- un dispositivo di I/O informa la CPU che è disponibile a ricevere o fornire dati. (in questo caso viene avviata una procedura opportuna del s.o. che si occupa dell'evitare conflitti e perdita di informazione dovuta, ad esempio, alle diverse frequenze operative dei componenti che interagiscono tra loro). In questo caso è necessaria una gestione molto attenta da parte del *Programmable Interrupt Controller*, anche qui per evitare conflitti e perdita di informazioni durante l'esecuzione del processo.
- il TempoMax per l'esecuzione di un programma è stato raggiunto e lo *scheduler* deve riassegnare le priorità dei processi e riassegnare la CPU ad un altro processo in coda.
- Viene effettuato il *debugging* e alla fine di ogni esecuzione del programma andata a buon fine viene inserito un interrupt che consente di verificare se in tale punto è richiesta o meno un'interruzione

GESTIONE INTERRUPT: La prima cosa da fare all'arrivo di un comando di interrupt è capire quale parte del calcolatore o delle periferiche l'abbia generato. Questo avviene attraverso tre metodologie:

1) nelle linee multiple di interrupt ad ogni dispositivo è associato un proprio piedino di gestione degli interrupt; ipraticabile per la gestione di numerosi dispositivi e interrupt quasi simultanei; utile in tecnologie special pourpouse

2) il polling consiste nella scansione ripetuta di tutto ciò che genera I/O e, quando viene interrogato il dispositivo che ha generato l'interrupt, sarà l'unico a rispondere alla ALU.

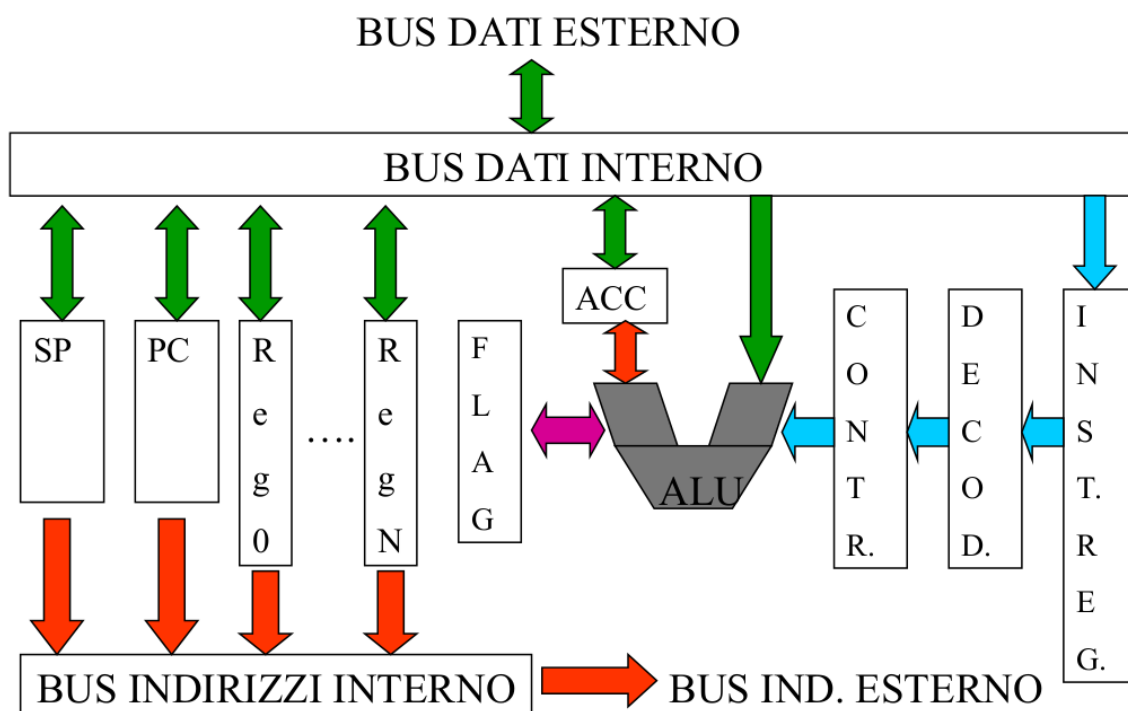
3) la vettorizzazione fa uso di un c.to integrato (il PIC già nominato) che ha in ingresso un numero di linee di *interrupt request* definite in partenza usate dai dispositivi per richiedere interruzioni. Il PIC quando riceve una richiesta di interrupt la "rigira" all'unità centrale e ricevuta la conferma si occupa di depositare nel *data-bus* l'indice del vettore di interrupt relativo all'ISR che gestisce l'interazione con il dispositivo (ogni vettore contiene le informazioni relative all'interrupt e a chi lo ha generato, permettendo alla CPU di riconoscere l'I/O richiedente.

POLLING: è la verifica ciclica di tutte le unità o periferiche di I/O da parte del s.o. di un calcolatore. Tramite il test dei bit dei bus associati ad ogni periferica, seguita da un'eventuale interazione (W/R). Il principale svantaggio è il tempo impiegato a checkare ogni periferica e il doverlo fare periodicamente. E' usato nell'identificazione degli interrupt

Vantaggi/Svantaggi: aldilà dell'unico vantaggio di riuscire a gestire le periferiche sia come hw che come sw, il polling presenta degli enormi svantaggi in termini di risorse sprecate dalla CPU per l'interrogazione (indirizzabili altrove), tempistiche di intervallo non costante ma dipendete dalle periferiche o difficile gestione delle interruzioni in caso di emergenza. Per questo le interruzioni tramite polling sono ormai obsolete e poco utilizzate

Cfr pg 56

ARCHITETTURA CPU



ALU: Arithmetic (and) Logic Unit, ovvero unità logico-aritmetica; è la vera responsabile dei calcoli della Cpu e colei che li esegue

Tipicamente utilizzata come parte di un processore monolitico, essa deve utilizzare la stessa notazione utilizzata dalle altre parti del calcolatore (binario, solitamente) e le operazioni di base che essa può eseguire sono:

- > operazioni matematiche sugli interi
- > operazioni logiche (And, Or, Xor)
- > operazioni di scorrimento binarie attraverso registri a scorrimento

I suoi input sono i dati da processare e il codice che attiva l'attività, contenente le istruzioni riguardo a cosa e come farlo. Gli output sono i risultati delle operazioni

Essa contiene due ingressi dal bus interno, di cui uno di questi derivante dall'accumulatore

Accumulatore: e' il registro più importante, che coinvolge tutte le operazioni logico-aritmetiche. E' fortemente legato alla ALU, di cui segue la larghezza (# di bit processabili) che è legato al processore considerato. Può immagazzinare un # di bit prestabilito che corrisponde alla capacità del processore; qualora gli operandi di un'operazione siano lo stesso numero/valore/variabile l'accumulatore provvederà (caso in cui abbia presente in memoria tale variabile) ad inviarlo su entrambi i lati: uno verso il data-bus per essere inviato alla ALU, e all'altro direttamente alla ALU

L'accumulatore ha due possibilità quando riceve un dato:

- immagazzinarlo se non deve essere utilizzato
- inviarlo alla ALU qualora esso debba essere usato in un calcolo

Obs: si noti come tutti gli elementi che comunicano tra l'interno della CPU e l'esterno (e viceversa) siano distinti appunto in componente interno/componente esterno. Questo perché solitamente interno ed esterno della Cpu (parte a contatto con le altre componenti/ ALU e comp. Interne) sono separate, con quest'ultima isolata; la causa di questo isolamento è principalmente da ricercarsi nel voler preservare l'unità logica in caso di corto c.to o problemi di varia natura che possono coinvolgere il nostro sistema

Flag: il termine flag indica un insieme di indicatori che sottolineano la corretta esecuzione o meno di una determinata istruzione all'interno della CPU. Sono variabili, solitamente booleane, che possono assumere solo due stati (1/0, True/False) e che segnala, con il suo valore, il verificarsi di un dato evento. L'impostare il flag a 1 *equivale a impostarlo a Vero*, ovvero indicare che si è verificato l'evento (viceversa con lo 0). Un esempio tipico è l'utilizzo di una variabile di flag booleana chiamata "trovato" all'interno di un algoritmo di ricerca. Questa variabile viene inizialmente impostata a "falso" e poi cambiata in "vero" solo quando viene trovato l'elemento ricercato.

Un altro esempio di flag sono le cosiddette "variabili semaforo" per sincronizzare l'accesso alle risorse all'interno dei sistemi operativi

Alcuni flag all'interno della CPU sono: *flag di carry, flag di zero, flag di segno, flag di trap, flag di overflow* (uno tra i più importanti, in quanto garantisce la corretta rilevazione di errori nell'esecuzione di operazioni)

ALU: la ALU può eseguire soltanto un determinato # di operazioni per ogni clock e, affinché sia pilotata correttamente insieme ai dati con cui eseguire le operazioni dovrebbero essere comprese anche le istruzioni per l'esecuzione vera e propria. Le istruzioni si eseguono in 3 momenti:

> **FETCH**: è la prima delle tre fasi e significa *prelevare*. Indica l'astrazione delle operazioni che caricano l'istruzione nel microproc. In questa fase la Control unit trasferisce una singola istruzione all'interno del *registro istruzioni* prelevandola da una risorsa (cache, Ram, i/o...) andando a modificare l'*execution flag*. La CPU carica l'istruzione dalla principale alla MDR, il cui valore è poi depositato nell'*instruction register*

Nella fase vera e propria di fetch viene prelevata dalla memoria l'istruzione da eseguire

> **DECODE**: è la parte più onerosa in termini di dispendio di risorse in quanto è quella che deve *decodificare* appunto l'istruzione giunta dalla fase di fetch, tradurre la sequenza di bit giuntagli in istruzione e inviarla alla Cpu facendole eseguire, tra tutte quelle disponibili, soltanto l'istruzione voluta.

> **EXECUTE**: è la fase di esecuzione vera e propria dell'istruzione. Dall'Instruction Register l'istruzione è mandata all'unità di controllo che la codifica. A questo punto se l'istruzione decodificata è tale da portare i dati memorizzati precedentemente in IR all'interno della ALU i dati possono essere manipolati da quest'ultima. (generalmente il dato nuovo viene scritto mediante *scrittura distruttiva*, ovvero cancellando il contenuto precedente del registro) Qualora l'istruzione non comprenda operazioni sui dati ma soltanto il loro spostamento da/verso una determinata cella di memoria la ALU non viene interpellata. Se l'azione termina con successo il Program Counter aumenta il suo valore di 1 all'interno dell'indirizzo di memoria, per indicare che l'operazione che verrà eseguita sarà quella contenuta nella cella successiva.

Program Counter: (detto anche *instruction pointer*) è un registro della Cpu la cui funzione è quella di conservare l'indirizzo di memoria della prossima istruzione, in linguaggio macchina, da eseguire. È un registro *puntatore*, ovvero punta ad un dato che si trova in memoria all'indirizzo corrispondente al valore contenuto nel registro stesso.

È utilizzato nel ciclo di *Fetch-Decode-Execute* e, normalmente, viene incrementato automaticamente alla fine di un'esecuzione, per segnare che l'esecuzione è conclusa con successo. Oltre alla modifica dopo l'esecuzione c'è anche la possibilità che il p.c. sia modificato esplicitamente da una o più istruzioni (e.g. se l'accumulatore ha tutti i bit a 0), consentendo così al programma di saltare a un'istruzione di programma che non sia quella immediatamente successiva a quella eseguita; forniscono quindi lo strumento fondamentale sul quale sono realizzate le strutture di controllo e i linguaggi di programmazione veri e propri

obs: nella maggior parte dei casi non ha una funzionalità informativa per la Cpu, in quanto essa sa che tipologia di dati sta per arrivarle; è semplicemente un contatore che in un registro apposito conta gli step di un programma per segnare la fine di un ciclo di f.d.e e ricominciare. È il "controllore" dell'esecuzione dei programmi, il componente che permette di mantenere l'ordine esecutivo di un'istruzione

Instruction Register (o *registro istruzione*) è un registro della CPU che immagazzina l'istruzione in fase di elaborazione. Ogni istruzione viene caricata in tale registro che la deposita mentre viene decodificata, la prepara per l'esecuzione e quindi la elabora.

Decodificatore (o *decoder*): è una tipologia di componente utilizzato per decodificare, appunto, le istruzioni che devono essere inviate alla ALU → a seconda della sequenza di bit in ingresso essi attivano una delle N linee in uscita e cui corrisponde una determinata funzione della ALU (somma, prodotto etc etc)

Il suo compito pertanto è quello di tradurre determinate sequenze di bit in altre sequenze

univoche che siano interpretabili dalla ALU. Ovviamente la lunghezza massima in uscita dell'encoder deve essere al massimo pari alla capacità della CPU (8/16/32...bit)

In generale avendo N linee di ingresso viene attivata esclusivamente una delle M linee di uscita con $M \leq 2^N$.

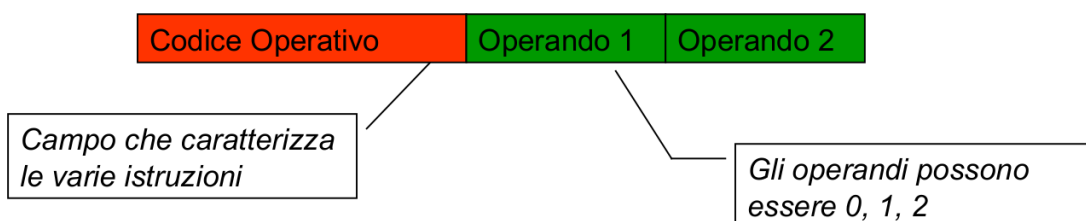
D	C	B	A		0	1	2	3	4	5	6	7	8	9
0	0	0	0		1	0	0	0	0	0	0	0	0	0
0	0	0	1		0	1	0	0	0	0	0	0	0	0
0	0	1	0		0	0	1	0	0	0	0	0	0	0
0	0	1	1		0	0	0	1	0	0	0	0	0	0
0	1	0	0		0	0	0	0	1	0	0	0	0	0
0	1	0	1		0	0	0	0	0	1	0	0	0	0
0	1	1	0		0	0	0	0	0	0	1	0	0	0
0	1	1	1		0	0	0	0	0	0	0	1	0	0
1	0	0	0		0	0	0	0	0	0	0	0	1	0
1	0	0	1		0	0	0	0	0	0	0	0	0	1
1	0	1	0		0	0	0	0	0	0	0	0	0	0
1	0	1	1		0	0	0	0	0	0	0	0	0	0
1	1	0	0		0	0	0	0	0	0	0	0	0	0
1	1	0	1		0	0	0	0	0	0	0	0	0	0
1	1	1	0		0	0	0	0	0	0	0	0	0	0
1	1	1	1		0	0	0	0	0	0	0	0	0	0

Esempio di encoder binario/decimale con condizione per cui $M < 2^N$. In questo caso le prime 10 delle 16 combinazioni sui 4 fili in ingresso danno luogo ad una corrispondente combinazione sui 10 fili in uscita, le 6 combinazioni successive non danno luogo ad uscita e sono ininfluenti

Program Counter: (ESP): è un registro dedicato alla CPU che contiene l'indirizzo della locazione di memoria occupata dal top dello *stack* per permettere le operazioni di push, che lo incrementerà, e di pop (che farà l'inverso, per i sistemi che eguono la logica LIFO) Per tenere traccia della "testa" dello stack esiste il registro dedicato *EBP* (o frame pointer) che punta per tutta la durata della procedura alla prima locazione in memoria dello stack in modo da tenerne traccia

CICLO DI ESECUZIONE DI UN'OPERAZIONE

FORMATO DELLE ISTRUZIONI



Esecuzione programma = ciclo di Fetch – Decode – Execute

Durante la parte di decode devono essere presenti all'interno del codice anche le istruzioni da eseguire e i dati sui quali eseguirle

Per questo vi è una parte, detta codice operativo (*op – code*) che identifica univocamente le procedure/operandi da eseguire

Oltre al codice operativo sono compresi anche gli operandi o le indicazioni per le allocazioni degli operandi non interni

e.g. il codica 0001 = ADD necessità di due operatori
> 0001, A, A → somma al valore dell'accumulatore il valore stesso, pertanto sarà composto esclusivamente da codice operativo (il valore è già presente nella CPU)
> 0001, A, 5 → somma al valore dell'accumulatore un valore che è sì in memoria, ma non presente all'interno dei registri della CPU:
L'istruzione pertanto si comporrà di coside operativo + operando

Il numero di letture di memoria necessarie per eseguire un'operazione dipende dalla presenza o meno degli operandi necessari nella CPU o meno (e lo vedo direttamente dall'analisi del codice operativo):

- qualora fossero già presenti nella CPU sarebbe necessaria soltanto una lettura di memoria, dato che letto il codice operativo gli operandi sono già disponibili
- qualora non fossero disponibili gli operandi sarebbero necessarie più letture di memoria, dipendentemente dal numero di operandi: maggiore è il loro numero, maggiori sono le celle di memoria da interrogare e pertanto le letture da fare

e.g. ADD A, R0 → ha già all'interno della CPU gli operandi necessari e pertanto richiederà soltanto un ciclo di fetch
ADD A, (REG_2) → necessità di più fasi di fetch dovendo recuperare il dato

Nel caso precedente durante la fase di fetch capisco se l'istruzione è completa o manca qualcosa (nel qual caso prelevo dato all'interno della memoria, al di fuori della CPU)
Il *prelievo dell'addendo* avviene tramite bus degli indirizzi prima e dei dati poi, con il contenuto di Reg_2 che viene bloccato nel MAR

((MAR)) → MBR

ovvero viene trasferito il contenuto a cui si è interessati all'interno dell'MBR che pilota il bus dei dati interno ed esterno

NOTA: il dato non viene mandato direttamente alla ALU per una questione di sincronismo, ovvero: se la ALU sta svolgendo un determinato processo e le invio il dato in modo non sincrono rischio di perdere parte del dato nel mentre che la ALU termina il processo. Per questo il dato viene salvato all'interno dei registri in attesa di essere usato

(MBR) → REGn (in cui mantengo dato a prescindere da stato memoria)

(REGn) + A → A (in cui viene eseguita l'istruzione di somma vera e propria)

Nota: nei processori più moderni e potenti esistono particolari tecniche di lettura unificata tra op.code e operandi (il che genera una notevole diminuzione dei tempi per un'operazione)

Esempio di Fetch

Il program counter indica dove andare (prima dal program counter interno, poi attraverso quello esterno) e raggiunge l'interlocutore selezionato; quest'ultimo invia l'informazione attraverso il processo inverso l'informazione raggiunge la CPU.

Essa arriva a tutti i dispositivi connessi all'interno della CPU, ma soltanto quello predisposto alla ricezione del dato si attiva (instruction register). Nel frattempo il program counter incrementa autonomamente.

Parallelamente alle istruzioni lavorano i *flag*

(PC) → MAR

((MAR)) → MBR; (PC)+1 → PC

nota: la parentesi indica il dover

prendere il contenuto di una cella

(MBR) → IR
fatto che il suo contenuto è un

Il fatto che MAR abbia due parentesi è dovuto al
indirizzo, non un'informazione

? = posso eseguire la ricerca mentre invio l'informazione. → NO, i processi 1 e 2 non possono essere contemporanei. Idem i processi 2 e 3, in cui non posso prendere un'allocazione che sta variando; essi devono essere sequenziali.

Similmente non posso incrementare il program counter in modo concomitante al suo utilizzo, ma posso in concomitanza con altre fasi, come il caricamento in MBR poiché il program counter non è in uso

NOTA: il fatto di incrementare il program counter alla seconda istruzione e non alla terza, ad esempio, dipende dal fatto che l'incremento del program counter, la prima e l'ultima istruzione hanno relativamente la stessa breve durata, mentre la seconda istruzione, quella di ricerca della cella ed effettiva lettura è la più lunga. Pertanto incrementando il program counter durante la seconda istruzione si è certi che, terminata l'istruzione, sia sicuramente terminato anche l'incremento; qualora si facesse il ultima istruzione la quasi contemporaneità delle due non permetterebbe margine sufficiente sull'effettiva esecuzione (c.ca 10 volte più lenta)

Registri adiacenti alla ALU:

Se l'istruzione comprende un operando che è già all'interno della memoria ma non all'interno dell'accumulatore, esso si trova nei registri ($t_0, t_1, t_2, \dots, t_n$) all'interno della CPU. Essi sono incameratori di dati e possono mantenere l'operando fino alla fine del ciclo di esecuzione, dopo il quale probabilmente vengono sovrascritti

Essi inoltre permettono di avere entità di facile e rapida consultazione (sfruttando il data bus interno e non interno + esterno + ricerca cella sono più rapidi)

La loro *dimensione* dipende dalla dimensione del bus dei dati e dal numero di linee presenti su essi

I registri sono connessi al bus degli indirizzi poiché se ho delle operazioni complesse che devono essere eseguite una dopo l'altra è necessario che sia pilotato il bus interno degli indirizzi per scrivere le informazioni parziali in determinate celle di memoria. Tuttavia quando vorrò scrivere in memoria sarà necessario generare un'istruzione che accompagni il dato da scrivere, affinché sia indicata la cella corretta di scrittura.

e.g. STORE A, (1011)

ESEMPI DI OPERAZIONI:

– ADD A, 100 in cui si chiede di sommare al valore dell'accumulatore il valore 100; si tratta di un'istruzione che contiene al suo interno solo il codice operativo, con istruzione e valore A da recuperare in fase di fetch

– Volendo sommare Accumulatore e contenuto di una cella di memoria la sequenza di fetch sarà:

(PC) → MAR

((MAR)) → MBR; (PC)+1 → (PC)

(MBR) → IR

(PC) → MAR

((MAR)) → MBR; (PC)+1 → (PC)

(MBR) → R_n

Dove si vede che il contenuto della memoria è prelevato in un secondo ciclo di fetch per essere portato alla ALU ed essere utilizzato.

N.B. → I due esempi, per quanto simili nelle finalità di somma, sono estremamente diversi per le modalità: mentre il primo legge l'operando dell'istruzione, il secondo nell'istruzione legge l'indirizzo a cui andare a prendere l'operando (si passa da solo codice operativo con dato disponibile a codice + operando con necessità di prelevamento). La varietà delle somme possibili dipende dalla complessità del calcolatore e dalle possibilità che possono presentarsi

Istruzione di SALTO:

La CPU a meno di essere spenta non smette mai di lavorare, background compreso. Pertanto è necessario che alla fine di un ciclo di istruzioni si torni ad un determinato punto del program counter. Per “saltare” all'inizio del program counter (e.g. durante l'esecuzione di un ciclo) si utilizza proprio l'istruzione di JUMP + L'indirizzo di destinazione del salto.

Nota: non necessariamente si salta all'inizio, anzi è un'usanza alquanto rara. Infatti saltare all'inizio implicherebbe tornare ad una situazione appena successiva all'avvio, dovendo così rieseguire tutte le inizializzazioni necessarie. Solitamente si salta all'inizio del fetch appena concluso o in un punto determinato del program counter che sia, appunto, *successivo a tali inizializzazioni*.

Il program counter può essere sovrascritto

Il fatto di andare a modificare l'ordine di esecuzione dei processi implica, necessariamente, la necessità di dover *riscrivere il program counter* cambiando le priorità esecutive. Dipendentemente dal caso l'istruzione precedente viene salvata prima di essere sovrascritta o viene eliminata

Una tipologia molto utilizzata sono i salti condizionati da risultati (sostanzialmente degli *if*)

Un'altra tipologia molto usata è quella dei salti con ritorno per l'esecuzione di *routine*

di sottoprogrammi per funzioni non disponibili da non riscrivere ogni volta (e.g. funzioni trigonometriche sviluppate a parte e richiamate alla necessità)

Address	Instruction
004937F7	MOV EAX,200
004937FC	MOV EDX,50
00493801	ADD EAX,67F0
00493806	MOV ECX,490AB3
0049380B	JMP 00497000
;Pretend there is a lot of code inbetween here.	
00497000	DEC EDX
00497001	MOV DWORD [49E6CC],EDX
00497007	MOV EAX,EDX

Jump to address 497000

then continue the code.

Se tuttavia per eseguire un salto basta l'istruzione di JUMP + ADDRESS, come ritorno al punto in cui stavo lavorando con salti annidati ? → Salvando al momento appena antecedente all'istruzione di jump il valore del program counter. Con gli annidati tuttavia è necessario avere un'allocazione precisa per il backup dei valori, poiché un registro è monovalore. La soluzione è scrivere in memoria gli indirizzi delle chiamate nello stack (pila), in cui vengono salvate le posizioni di ciascuna chiamata con la filosofia LIFO (Last In First Out)

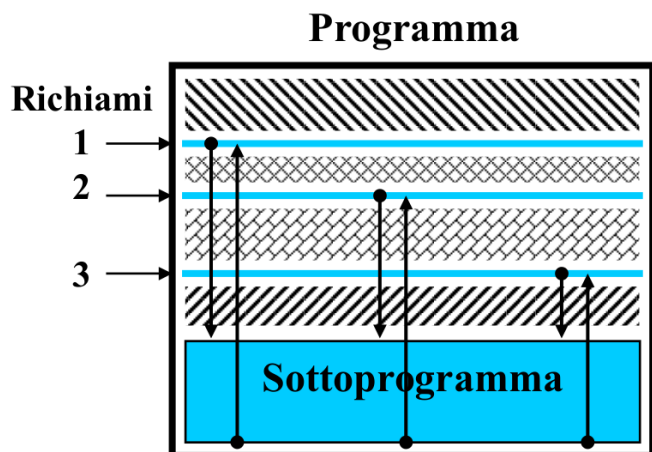
stack:: il termine stack, o *pila*, indica in informatica un tipo di dato astratto usato per riferirsi a strutture dati, in cui le modalità di accesso ai dati in essa sono della tipologia LIFO (Last In First Out) in ordine inverso rispetto alla scrittura.

Una tipologia di stack presente all'interno dei calcolatori moderni è il *call stack*, che tiene traccia dei punti in cui ogni subroutine attiva dovrebbe restituire il controllo quando termina l'esecuzione.

La ALU al suo interno, non sapendo dove andare a prendere ogni volta il valore del Program Counter (varia dipendentemente dal numero di *soubroutine* che vengono eseguite) utilizza lo **stack pointer**, ovvero un puntatore che tiene traccia, con la stessa filosofia, degli indirizzi di ciascuna call e del rispettivo valore del Program Counter

Nota: essendo lo stack allocato in memoria RAM e contenendo soltanto valori relativamente piccoli (count), il numero di *soubroutine* possibili è idealmente illimitato

Obs: le frecce nello schema della ALU vanno verso il data bus poiché è necessaria l'inizializzazione dello stack e della sua cima la momento dell'esecuzione della chiamata



Schema visivo delle chiamate ai sottoprogrammi in esecuzione. Dal main si passa al sottoprogramma, memorizzando nello stack l'indirizzo precedente alla chiamata per continuare, una volta interrotta la subroutine, da dove si era interrotto. Lo stack pointer punta sempre all'ultimo indirizzo non nullo non ancora prelevato

? = eseguito il *return* alla cella dove avevo eseguito la call, cosa trovo nella cella dove avevo lasciato il main per eseguire la subroutine appena

“superata” nell'elenco del P.c.?

Contiene comunque, a meno di sovrascritture, l'indirizzo dello stack corrispondente al *jump*. Quello che varia è il puntatore agli indirizzi della pila, non la pila.

1– esecuzione di CALL + INDIRIZZO (codice operativo + operando)

(PC)→ STACK

(SP)-1→(SP) *puntatore spostato a cella non vuota non prelevata*

(SP)→ MAR

(PC)→ MBR

(MBR)→(MAR)

(REG_n)→ PC

(verificare)

2– esecuzione di RETURN (solo codice operativo)

(SP)→ MAR

((MAR))→ MBR

(MBR)→ PC

(SP + 1)→ SP *come sopra, SP punta a ultima casella non vuota non prelevata*

RAPPRESENTAZIONI NUMERICHE

Ci sono numerosi modi di rappresentare i numeri, a partire dal conteggio tramite stanghette ai numeri romani. In informatica il più utilizzato è il binario, che ha il problema di non essere compatto → per rappresentare anche soltanto un numero di 3 cifre ne servono 8/9 di bit.

Inoltre se ho il limite di numeri da considerare e operazioni complesse c'è un elevato rischio di compiere errori (e.g. 999 + 1 con 3 cifre rischio di ottenere 000 se non sto attento al riporto)

Decimale: i numeri sono fattori pesati, in cui ogni coefficiente è moltiplicato per la potenza di 10 a seconda della posizione più o meno importante che occupa

Per una qualunque base di rappresentazione B necessito di B simboli \rightarrow ottengo il polinomio pesato corrispondente alla base A di partenza

In base 2 \rightarrow utilizzo 2 elementi: - 0 - 1

e.g. 9 in b10 diventerebbe $1001 = 1 \cdot 2^0 + 0 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 = 8 + 1 = 9$

? = quante cifre servono per rappresentare 10 cifre. L'importante è rispettare $2^N \geq 10$ con $N = \# \text{cifre}$

Pertanto per rappresentare 10 cifre necessito di 4 cifre. Infatti $2^4 = 16$ mentre $2^3 = 8$

(Per rappresentare M cifre devo avere $2^N \geq M$)

In base 8 ho otto elementi rappresentativi $\rightarrow 3715 = 5 \cdot 8^0 + 1 \cdot 8^1 + 7 \cdot 8^2 + 5 \cdot 8^3 = 1997$

In base 16 ho sedici elementi rappresentativi, le dieci cifre decimali più le prime sei lettere dell'alfabeto $\rightarrow 0, 1, 2, \dots, 9, A, B, C, D, E, F$

$7CD = 13 \cdot 16^0 + 12 \cdot 16^1 + 7 \cdot 16^2 = 1997$

CONVERSIONE DA B10 A B2, B8, B16...?

So che un numero nella base B di arrivo sarà rappresentato da termini con peso, ovvero dal polinomio

$$N_b = N_a = X_n \cdot B^n + X_{(n-1)} \cdot B^{(n-1)} + \dots + X \cdot B + X_0 = X_0 + B(X + B(X_2 + B) \dots)$$

Dividendo tutto per B ottengo

$$N_a/B = X_0/B + (X_1 + B(X_2 + B) \dots) \quad \text{Dove } X_0/B \text{ è resto; la parentesi il risultato intero della divisione}$$

Dividendo N volte (finché possibile) e considerando i resti in ordine inverso di ottenimento (per mantenere il peso delle cifre) ottengo il numero convertito

e.g. 1258 B10 \rightarrow in B2?

1258	:	2	0
629			1
314			0
157			1
78			0
39			1
19			1
9			1
4			0
2			1
1			

Obs: posso fare questo procedimento per ogni base A e B ? ipoteticamente e a livello teorico sì, la dimostrazione è sempre valida. In realtà non è applicabile se $A \neq 10$ per difficoltà computazionali: non siamo infatti in grado di eseguire divisioni e moltiplicazioni in basi diverse dalla 10

e.g. 10 in B16 come lo converto in B8? Passando dal decimale, altrimenti rischio errore in divisione

NUMERI FRAZIONARI:

E se i numeri da convertire sono frazionari/decimali?

Concettualmente non cambia nulla, semplicemente l'ordine posizionale dei resti delle divisioni è invertito e invece che considerare i resti considero gli interi

$$N_b = N_a = X \cdot B^{-n} + X_{(-n+1)} \cdot B^{(-n+1)} + \dots + X \cdot B + X_0 = X_0 + B(X + B(X_2 + B) \dots)$$

In cui dividendo ottengo $X_0 + B^{-1}*(X-1 + B^{-2} (...))$

e.g. $0,59375 * 2$ 1
1,18750 0
0,375 0
0,750 1
1,5 1
1

? = Quando mi fermo con le moltiplicazioni

Se ottengo decimale nullo o, nel caso di cifre periodiche o particolarmente lunghe, appena raggiungo la precisione desiderata

e.g. $0,1 * 2$ 0
0,2 0
0,4 0
0,8 1
1,6 1
1.2 0

PASSARE A BASE 'A' A BASE 'B' QUANDO $B=A^k$

Se devo passare da una base A ad una base B in cui la base di arrivo è potenza di quella di partenza, ad esempio da base 2 a base 8?

$N_2 = d_k d_{k-1} \dots d_5 d_4 d_3 d_2 d_1 d_0$

$= \dots(d_5 2^2 + d_4 2^1 + d_3 2^0) 2^3 + (d_2 2^2 + d_1 2^1 + d_0 2^0) 2^0$

Dove i termini in parentesi possono assumere, nel caso $2 \rightarrow 8$ valori tra 0 e 7 e i coefficienti estremi diventano i coefficienti posizione

e.g. 010010010110100 in cui devo semplicemente considerare i blocchi di k

elementi (dove k è la potenza tra la base A e la base B) e analizzarli come fossero i coefficienti posizionali dei numeri. Pertanto il numero precedente in binario, trasformato in ottale diventa

22264 mentre in esadecimale (dove i blocchi di numeri considerati sono 4) il numero diventa 24B4.

Similmente se dovessi eseguire il contrario come operazione potrei considerare numero per numero e trasformarlo nella base di arrivo (i.e. da base 16 a base 2)

Qualora avessi un numero non multiplo di cifre aggiungo uno zero in posizione t.c. non cambi il numero (sx per gli interi, dx per i decimali)

? = Come passo da base 16 a base 8 o viceversa ? \rightarrow NON passando dal decimale ma dal binario, essendo le basi di partenza e di arrivo entrambe potenze di 2.

Somma:

+	0	1
0	0	1
1	1	0 (1)

(la parentesi è il riporto)

e.g. 11001 +
 1101 =

 100110

Differenza:

-	0	1
0	0	1 (1)
1	1	0

e.g. 11001 -
 1101 =

 1100

Moltiplicazione

X	0	1
0	0	0
1	0	1

Il problema a livello circuitale è ridurre i tempi di tutte le operazioni che non siano la somma, infatti se prendo ad esempio la sottrazione ci sono più passaggi:

- > cfr tra le cifre da sommare
- > determinazione v.a. maggiore
- > esecuzione sottrazione
- > controllo segno

E se ciascuna operazione richiede un tempo T, alla fine ci vorrà un tempo $4 \cdot T$

NUMERO SENZA SEGNO:

La rappresentazione in v.a. del numero senza segno implica che con un byte io possa rappresentare 2^8 elementi, ovvero il range 0-255. Non è funzionale poiché arrivato a 255 + 1 torno a 000 con il numero di bit a disposizione

Inoltre non permette l'esecuzione di operazioni al di fuori dell'insieme N dei numeri naturali

MODULO E SEGNO (Binario Naturale)

Si utilizza un bit per la rappresentazione del segno ($0 \rightarrow +$; $1 \rightarrow -$)

Il problema di questa rappresentazione è duplice:

- > Perdo una rappresentazione per lo zero (che di fatto ha due rappresentazioni)
- > Con 8 bit posso rappresentare valori per 7 bit ovvero da 0 - 127
- > Un incremento binario corrisponde ad un aumento dei numeri positivi, ma un decremento dei negativi

(N.B il significato di una sequenza di bit non ha valore predeterminato, ma dipende dal contesto in cui mi trovo e dall'interpretazione che gli viene data)

Anche con questa notazione non risolvo il problema dell'unificazione dei passaggi per sottrazione e somma

Numero rappresentabili con questa notazione: $-2^{N-1}+1$ a $2^{N-1}-1$

COMPLEMENTO ALLA BASE DI UN NUMERO

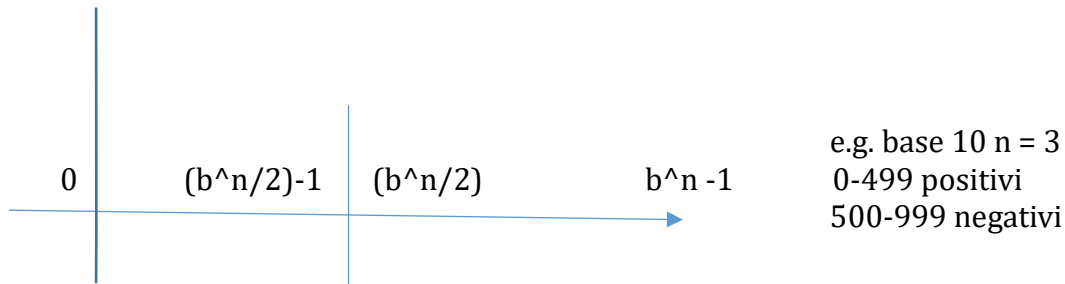
N cifre, base B $\rightarrow B^N$ configurazioni

Posso usare la stessa notazione per rappresentare sia positivi che negativi:

- Da 0 a $[B^N / 2] - 1$ rappresento i numeri positivi con la rapp. Posizionale

- Da $[B^N / 2]$ fino a $B^N - 1$ rappresento i negativi

(e.g. con 4 bit rappresento i positivi da 0 a $2^4 / 2 - 1$ ovvero da 0 a 7. Mentre da 8 a 15 rappresenterei i corrispettivi numeri negativi)



Per ottenere il negativo di un numero devo fare il complemento alla base di quel numero, ovvero devo fare $b^n - X$

(e.g. se con 2 cifre in base 10 se volessi rappresentare -40 rappresenterei $100 - 40 = 60$)

All'atto pratico questo si traduce con il mantenere tutti i bit uguali partendo dai più significativi fino al primo 1, dopodiché complementare ogni numero ($0 \rightarrow 1$; $1 \rightarrow 0$)

Il -1 è rappresentato sempre da tutti *uni*

COMPLEMENTO ALLA BASE - 1 DI UN NUMERO

Anche in questo caso utilizzo metà intervallo per i numeri maggiori di 0 e l'altra metà per i numeri minori di zero. La differenza sta nel minimo numero negativo rappresentabile.

Se infatti per la parte positiva non cambia assolutamente nulla, andando da 0 a $[b^n/2] - 1$, con i numeri negativi vado da $b^n/2$ a $b^n - 1$ in cui però il massimo dei numeri rappresentabili non corrisponde più a -1 ma a 0-

Pertanto per rappresentare un numero negativo a partire a un positivo X la formula sarà:

$$(b^n - 1) - X$$

A livello pratico la grossa differenza con la notazione precedente è che invece che mantenere invariati i bit fino al primo uno a partire dalle cifre più significative, qui per ottenere il corrispettivo negativo è sufficiente invertire i valori di 0 e 1 di tutto il numero

(e.g. $0110101 = 53$; se volessi rappresentare -53 basterebbe invertire e ottenere $\rightarrow 1001010$)

Pertanto riassumendo $C_b = b^n - 1$

$$C(b-1) = (b^n - 1) - X$$

$$C_b - C(b-1) = 1$$

Se volessi sapere le cifre rappresentabili in ogni metà (positiva e negativa) di una base 16?

Avrei che il numero massimo positivo rappresentabile è $(16^3)/2 - 1$, ovvero $2^7/2 - 1$, ovvero 2^6 , che in esadecimale sarebbe 800.

Pertanto il massimo numero positivo rappresentabile in esadecimale sarà 7FF, il successivo, 800, rappresenterà il corrispettivo negativo

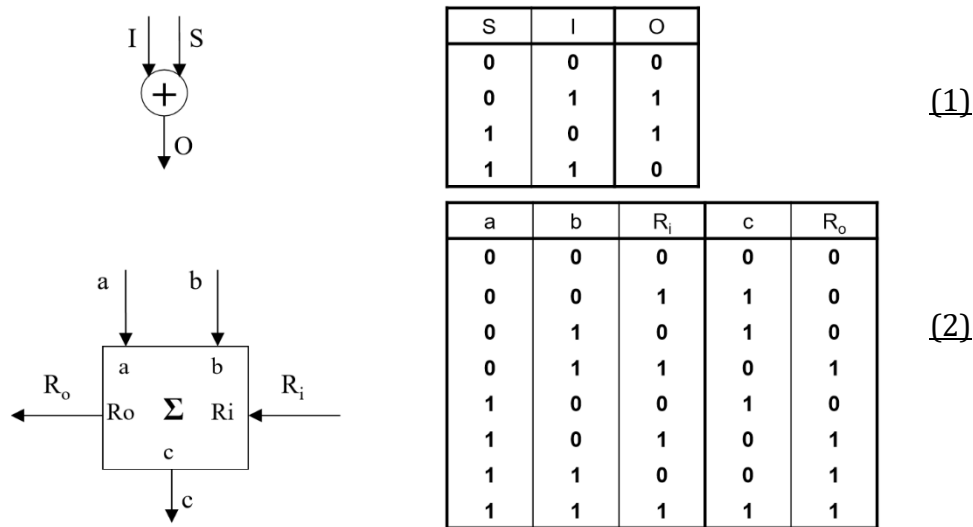
Obs: ammettendo di essere in complemento vale la regola

$$A - B = A + (2^k - B) = A + (2^k - B - 1) + 1$$

Dove gli elementi all'interno delle parentesi sono esattamente i corrispettivi negativi del numero B. Ciò permette di eseguire, almeno in binario, una notevole semplificazione di quella

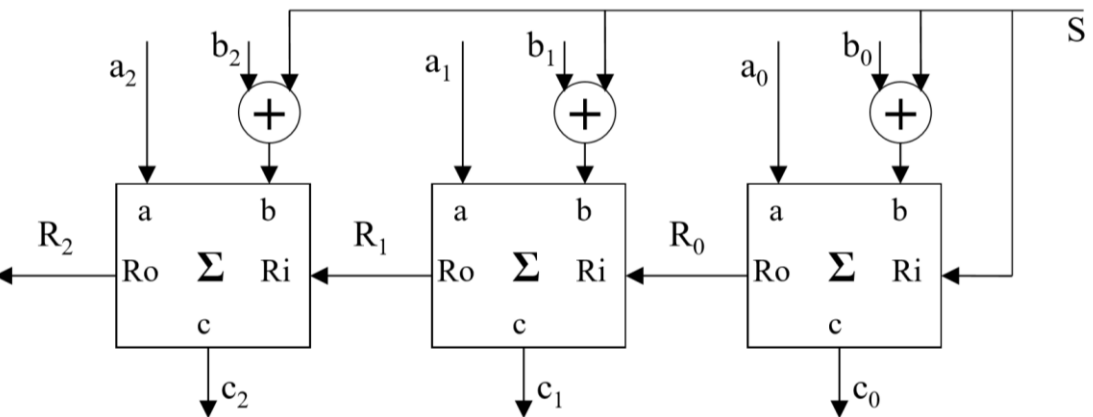
che è la differenza in una serie di somme, senza i problemi di controllo di segno e valore assoluto che si porta dietro la prima operazione

Sommatore:



Nella figura (1) si vede un exor in cui, in sostanza, viene fatta/non fatta un'inversione sull'ingresso I → Come si nota dalle tabelle di verità se ho 0 su S riporto esattamente I in uscita; se su S ho 1 invertito il valore di I

In (2) ho un altro dispositivo, ovvero un sommatore in cui tengo conto anche del resto sia in ingresso che in uscita (rispettivamente Ri e Ro) e dove comunque eseguo una somma $a+b=c$



Ovvero un dispositivo che esegue la somma posizionale di due numeri. In cui S rappresenta l'eventuale resto di ingresso, bo l'exor e le sommatorie gli effettivi sommatore

- $S=0 \rightarrow C_0=a_0+b_0 \quad C_1=a_1+b_1+R_0 \quad C_2=a_2+b_2+R_1$
ovvero la somma posizionale di $a_2a_1a_0 + b_2b_1b_0$
- $S=1 \rightarrow C_0=a_0+(-b_0)+1 \quad C_1=a_1+(-b_1)+R_0 \quad C_2=a_2+(-b_2)+R_1$
ovvero la differenza posizionale di $a_2a_1a_0 - b_2b_1b_0 \rightarrow a - b$

Nota: nella sottrazione l'eliminazione del riporto già in partenza è mantenuta dall'ingresso nella prima operazione dell'1 come variabile, oltre che come indicatore di sottrazione

OVERFLOW & RIPORTO

010011 +
010001 = E' giusto? Dipende da che rappresentazione sto usando

100100

- Se stessi lavorando in valore assoluto sarebbe giusto
- Se stessi lavorando in complemento avrei che sommando due numeri positivi ottengo un numero negativo → (19+17=-28)

Questo poiché usando 6 bit posso rappresentare da -32 a +31, pertanto 36, ovvero la somma corretta, non è rappresentabile

In questo caso si dice che ho avuto *overflow*: ovvero il superamento della capacità di rappresentazione per il numero in complemento (nel grafico dei numeri rappresentabili finirei nel lato dei numeri negativi); questo genera l'attivazione di un flag nei registri appositi

e.g. 01111000 +
01101001 =

11100001

RIPORTO:

Supponendo ora di voler fare 11111011 +
11110000 =
11101011

Avrei un risultato che in valore assoluto sarebbe errato in quanto mancherebbe l'ultimo riporto a causa dell'insufficienza dei bit per rappresentare il numero ottenuto

N.B: overflow != riporto. Il primo indica in complemento il superamento dei numeri rappresentabili con quella combinazione, il secondo invece sottolinea il generare riporto da parte dell'operazione

$0 \leq A \leq 2^{N-1} - 1$ $0 \leq B \leq 2^{N-1} - 1$	$-2^{N-1} \leq A < 0$ $0 \leq B \leq 2^{N-1} - 1$	$0 \leq A \leq 2^{N-1} - 1$ $-2^{N-1} \leq B < 0$	$-2^{N-1} \leq A < 0$ $-2^{N-1} \leq B < 0$
$0 \leq S \leq 2^N - 2$	$-2^{N-1} \leq S < 2^{N-1} - 1$	$-2^{N-1} \leq S < 2^{N-1} - 1$	$-2^N \leq S < 0$
Sommando due numeri positivi si ha overflow se si ottiene un numero negativo. S potrebbe non essere rappresentabile in N bit ma lo è sempre in N+1 bit.	Non ci sono mai problemi di overflow.	Non ci sono mai problemi di overflow.	Sommando due numeri negativi si ha overflow se si ottiene un numero positivo. S potrebbe non essere rappresentabile in N bit ma lo è sempre in N+1 bit.

TECNICA DELL'ECCESO

In valore assoluto ho, a colpo d'occhio, una visione sugli ordini di grandezza dei numeri; in complemento invece non così rapido il discorso

La tecnica dell'eccesso nasce dall'esigenza di avere una rapida comprensione delle dimensioni di stringhe di bit a cfr senza dover svolgere necessariamente tutti i calcoli.

Viene traslato il grafico del valore assoluto di una quantità +q (la cui rappresentazione diventerà la rappresentazione dello 0)

Ho infatti una curva crescente con continuità

La traslazione solitamente avviene di un fattore 2^{N-1} (e.g. con 3 bit sposto di 4)

- Come trovo il numero corrispondente in eccesso?

Semplicemente ho la formula per cui dato un numero, per avere il corrispondente in eccesso devo rappresentarlo nella base desiderata e successivamente togliere +q, ovvero il fattore di traslazione

e.g. 1110 in eccesso \rightarrow lo trasformo in decimale (14) e gli tolgo la traslazione (8) ottenendo il risultato \rightarrow 6

- Di quanto mi sposto? Idealmente nella metà del massimo raggiungibile. Se fermarmi a 2^{N-1} o a $2^{N-1}-1$ dipende da quanto spazio voglio dedicare ai positivi e quanto ai negativi

e.g. -10 eccesso base 16? $\rightarrow -10 + 127 = 117 \rightarrow$ lo tramuto in b16 \rightarrow 60B

VIRGOLA FISSA:

Come rappresento numeri più grandi di quelli rappresentabili al massimo?

Come rappresento la virgola se voglio rappresentare un numero con presenza di parte decimale e parte intera?

La virgola fissa consiste nel decidere dove posizionare la virgola e, a priori, quanti bit dedicare alla parte intera e quanta a quella frazionaria.

e.g. 72,6 con 12 bit (8 int. 4 fraz.) \rightarrow 10110111 0111

La virgola fissa non è funzionale poiché perdo un notevole intervallo di rappresentazione e oltre ad avere un numero limitato di bit, lo abbasso ancora di più. Ha inoltre una scarsa precisione rappresentativa per quanto riguarda le cifre decimali: fissando a priori infatti il numero di cifre da dedicare alle due parti rischio la perdita di informazione per arrotondamento

VIRGOLA MOBILE

A livello decimale per ovviare al problema dei numeri grossi di difficile rappresentazione si è ricorso alla notazione scientifica, moltiplicando un numero normalizzato per una potenza di dieci;

$$\alpha = \text{sign}(\alpha) \cdot b^\beta \sum_{i=1}^{\infty} d_i \cdot b^{-i}$$

Normalizzazione \rightarrow

$$\text{con } \beta \in \mathbb{Z}, d_i \neq 0, \text{sign}(\alpha) = \begin{cases} +1 & \alpha > 0 \\ -1 & \alpha < 0 \end{cases}$$

In cui qualunque $a \neq 0$ può essere rappresentato in base $b > 1$ attraverso la normalizzazione; ho 3 casi:

- $\pm d_{b-1} d_{b-2} \dots d_0, d_{-1} d_{-2} \dots$ per Beta > 0
- $\pm 0, d_{-1} d_{-2} \dots$ per Beta = 0
- $\pm 0, 0 \dots 0 d_{-b-1} d_{-b-2} \dots$ per Beta < 0

In cui in pratica l'unica variabilità che si ha è sull'indice i

? = Da cosa dipende il numero rappresentabile? Dalla variazione di Beta

Un qualunque numero N viene rappresentato nella forma

$N = \pm 0, \dots \times 10^{\text{exp}}$ dove i numeri dopo la virgola sono le cifre significative (**mantissa**) del numero che voglio rappresentare, mentre il numero elevato a exp è la mia base b

Nota: la variazione di rappresentazione delle cifre significative è da $1/b^N < M < 1$

N.b. il valore minimo non è tutti 0 e 1 in fondo in quanto in quel caso la normalizzazione sarebbe errata

Da cui $\rightarrow A = S \quad 0, M \times b^{\text{exp}}$

e.g. decimale: 1259411000000...0 \rightarrow 0 0.125941 $\times 10^{20}$

Ho tuttavia degli elementi ridondanti:

- Virgola
- Base di elevamento, in quanto già parte della convenzione
- 0, in quanto so già che la mantissa è normalizzata

Eliminando il superfluo ottengo la rappresentazione in virgola mobile di un numero in una base b

? = in che ordine vado a disporre segno, mantissa ed esponente della base?

L'ordine esatto, soprattutto per facilitare il cfr tra due numeri, è :

° segno

° esponente

° mantissa

L'esponente (E) a cui bisogna elevare la base B per ottenere il fattore per cui moltiplicare la mantissa per ottenere A

$$A = S \cdot M \times B^E$$

E' inoltre necessario utilizzare per l'esponente una rappresentazione che consenta la di raffigurare sia i numeri positivi che negativi (complemento base o eccesso)

e.g.

S	E	E	M	M	M	M	M	M	M	M
---	---	---	---	---	---	---	---	---	---	---

1758.37	$0.175837 \cdot 10^4$	0 04 17583700
-0.001	$-0.1 \cdot 10^{-2}$	1 98 10000000
1	$0.1 \cdot 10^1$	0 01 10000000
5000	$0.5 \cdot 10^4$	0 04 50000000
-63517.8	$-0.635178 \cdot 10^5$	1 05 63517800
-0.0000635178	$-0.635178 \cdot 10^{-4}$	1 96 63517800

In base 2:

utilizzo sempre un bit per il segno (0/1), 8 bit per l'esponente e 23 per la mantissa

$$1001100011111000.0111011 \quad 0.10011000111110000111011 \cdot 2^{16}$$

Nota: in virgola mobile cambia notevolmente il range di rappresentazione dei miei numeri, con il problema della minor precisione. Ho infatti infiniti numeri che vengono rappresentati in modo approssimato a causa del numero precedentemente deciso di bit per la rappresentazione

In virgola mobile ogni intero è un punto di accumulazione per i punti della linea dei valori Rappresentabili rispetto ai valori che non posso rappresentare esattamente (inserire grafici Range 4.11.16)

L'esponente varia da -128 a +127 (**n.b. è in complemento**)

Si parla di dinamica per descrivere l'intervallo dei numeri rappresentabili $-2^{127} < N < 2^{127}$ (circa $-10^{38} < N < 10^{38}$)

Il *massimo numero negativo rappresentabile* è $-2^{-128} \times 0.1 = -2^{-129}$ (dove 0.1 è la mantissa minima rappresentabile)

Il *minimo numero positivo rappresentabile* è 2^{-129} .

Underflow: quando il mio numero da rappresentare cade all'interno dei due numeri precedentemente descritti: ovvero è minore del minimo positivo rappresentabile o maggiore del massimo negativo

Overflow: come nel caso del complemento alla base se esco a livello di valore assoluto dai valori rappresentabili

Nel caso di over/underflow devo cambiare base e passare alla DOPPIA PRECISIONE: 64 bit (1 per S, 11 per E, 52 per M)

Note:

- Con i periodici inevitabilmente perdo informazione riguardo la periodicità a causa delle approssimazioni necessarie
- Per la conversione da v.m. a decimale e viceversa devo assicurarmi di avere tutto nella stessa base prima di muovere la virgola e normalizzare, altrimenti vado a cambiare radicalmente il numero in questione
- Al variare del numero di bit per la mantissa e per l'esponente vario il range di rappresentazione e, di conseguenza, la precisione

OPERAZIONI IN VIRGOLA MOBILE:

- Moltiplicazione/Divisione: sono le due operazioni più semplici, in cui le mantisse vengono moltiplicate/divise, gli esponenti sommati/sottratti e, se necessario, il tutto viene rinormalizzato alla fine aggiustando il grado dell'esponente
- Somma/Sottrazione: è necessario prima di poter eseguire tali operazioni andare a portare l'esponente minore allo stesso grado del maggiore spostando le cifre significative a destra di un numero di posti pari *alla differenza dei due esponenti*; a questo punto posso eseguire le operazioni sulle mantisse, mantenere l'esponente com'è e al limite normalizzarlo ad operazione eseguite

NB: nello shift c'è il rischio di perdere informazione dalle cifre meno significative per motivi di dimensioni di mantissa. Potrebbe trattarsi di un errore di poco conto in numeri particolarmente elevati, ma è necessario tenerne conto; infatti non vale:

$$(A+B)+C = A+(B+C)$$

Poiché la perdita di informazione prima o dopo di un'operazione fa la differenza sul risultato finale

IEEE754

? = quanto vale X in 0,X nella rappresentazione in virgola mobile? *Sempre 1 per l'ipotesi della normalizzazione*

Posso per questo ometterlo (hidden bit) e avere così una mantissa che virtualmente è di 24 bit

La mantissa M è inoltre variabile tra $1 \leq M < 2$; infatti se prendessi tutti zeri avrei semplicemente l'1 precedente la virgola, se avessi tutti 1 avrei una rappresentazione che tende a 2

La precisione che permette di avere questa rappresentazione è di

$$2 - \text{lunghezza} (m - 1) \leq x \leq 2^{\text{lunghezza} (m)}$$

Con m = mantissa

Nota: come rappresento lo 0? Concretamente non ho una rappresentazione vera dello zero, pertanto devo stabilire a priori una combinazione che lo rappresenti:

0 00000000 000000000000000000000000 → ovvero tutti 0, a prescindere dal significato secondo la formula

e.g. $0.5 = 1 \cdot 2^{-1} \rightarrow 0 \quad 01111110 \quad 00000000 \quad 00000000 \quad 00000000$
 $3.141593 \rightarrow 0 \quad 10000000 \quad 1001001 \quad 00001111 \quad 11011100$

Per facilitare il confronto avrei bisogno di una crescita uniforme degli esponenti, in modo da riconoscerli ad occhio, e non una crescita spezzata come quella della rappresentazione in complemento. Per questo l'esponente è *rappresentato in eccesso alla base*.

In questo modo l'intervallo rappresentabile dal punto di vista dei valori varia da 2^{-127} a 2^{128}

RAPPRESENTAZIONI NON NUMERICHE:

I calcolatori lavorano soltanto con i numeri, per questo è necessario introdurre un codice, ovvero una corrispondenza biunivoca tra un simbolo e la sua rappresentazione (nel caso dei calcolatori in bit)

e.g. 01=A 02=B.....25=Y 26=Z

Alfabeto: insieme finito di simboli e caratteri distinti

Parola/Stringa: sequenza finita di simboli o caratteri

Codifiche: dato un insieme I di elementi qualsiasi, si dice codifica di I mediante parole di un alfabeto A un procedimento che permette di stabilire una corrispondenza biunivoca tra gli elementi di I e le parole di un sottoinsieme Q dell'insieme delle parole di A

Un *codice binario* è pertanto la codifica dei simboli dell'alfabeto Σ mediante stringhe di bit.

Cardinalità: se C è la cardinalità di Σ per il numero n di bit da usare deve valere $C \leq 2^n$

Ovvero: $N \geq \lceil \log_2(C) \rceil = M$ (ovvero l'intero approssimato per eccesso del log)

Nota: meglio utilizzare più bit del dovuto per un discorso di ampliamento del linguaggio utilizzato e di eventuale possibile correzione di errori durante l'invio/la ricezione di messaggi codificati.

Ridondanza: un codice si dice *ridondante* se usa più bit del minor numero **necessario** per rappresentare i simboli di Σ

(NB: si parla di numero minimo necessario per rappresentare l'alfabeto interessato, non di combinazioni rimaste libere. Se voglio rappresentare 100 elementi con 7 bit $\rightarrow 2^7 = 128$, ma con 6 non posso rappresentarne 100 pertanto NON è ridondante)

Distanza di Hamming: all'interno di un alfabeto è il numero minimo di bit che differenziano una parola dell'alfabeto da tutte le altre

e.g. 00100 110100 00110 11110 \rightarrow distanza $H = 1$

Ridondanza $\rightarrow H \geq 1$

NON ridondanza $\rightarrow H = 1$

Nota: nel caso di processo inverso posso solo dire se ho ridondanza con $H > 1$, con $H = 1$ NON posso concludere nulla

CORREZIONE INFORMAZIONE:

Come posso correggere una modifica dell'informazione durante il transito dall'emettitore al ricevitore?

- Parità: aggiungo al termine della mia informazione un bit che sia 1 o 0 e verifico che nel complesso il numero di 1 dell'informazione sia dispari. In questo caso, sotto l'ipotesi di un solo bit modificato, posso capire se e in quale stringa si è modificato. Il bit di parità (o $H=2$) può soltanto rilevare se è stata alterata l'informazione, non capire anche quale parte di essa sia stata danneggiata

Una variante del bit di parità potrebbe essere imporre anche un controllo in verticale ogni tot di informazione inviata, in modo da avere un controllo incrociato che, in caso di modifica, mi permetta di capire esattamente quale bit è stato modificato

- Codice di Hamming: (o $H=3$) in cui dato un codice non ridondante vengono inseriti k bit di controllo. Perché possa essere corretto un errore è necessario che valga $N \leq 2^k - k - 1$

Rivela la presenza di un solo errore e può correggere un bit in posizione arbitraria:

 1001 vengono eseguiti controlli di disparità su gruppi scelti di bit, in modo che con-

frontando i risultati di ciascun controllo si abbia esattamente il bit modificato (e.g. sulla relazione precedente, con una parola di 7 bit sarà necessario utilizzare 4 bit per il controllo affinché la formula sia verificata)

Nonostante sia più dispendioso del semplice controllo di parità a differenza di quest'ultimo è anche correttivo

Affinchè un codice sia correttivo è necessario che la distanza di Hamming $H > 2$, altrimenti è semplicemente rivelativo di errore

CODICI CICLICI:

Sono codici CRC non correttivi a rilevazione di errore utilizzati su lunghi tratti di trasmissione su linee rumorose, in particolare su interferenza a raffica. In caso di errore il messaggio deve essere ritrasmesso.

Un messaggio M di k bit da trasmettere viene trattato come un polinomio di grado $k-1$, i cui coeff. Sono i bit del messaggio. Preso un altro G di grado r si attiva attraverso operazioni modulo 2 (ovvero somme senza riporto del resto) da cui ottengo un polinomio T di grado $k < t \leq k+r$ divisibile per G . Al ricevente se il polinomio non è divisibile esattamente per G ciò corrisponderà ad un errore in trasmissione

CODIFICA DECIMALI:

Codice BCD (Binary Coded Decimal): ogni cifra decimale viene codificata con 4 bit. È un codice pesato perché il valore di ogni cifra viene ottenuto eseguendo una somma pesata delle quattro cifre binarie che lo compongono

Codice ad eccesso tre: è basato sul codice BCD. Ogni codifica si ottiene sommando 3 alla codifica BCD. Non è un codice pesato. Per passare da una cifra alla cifra corrispondente al complemento a 9 basta complementare le cifre binarie della sua codifica

Codice 2421: i numeri indicano ordinatamente i pesi delle 4 cifre binarie. Come per l'eccesso tre, le codifiche di una cifra e della cifra corrispondente al complemento a 9 sono fra loro complementari

	BCD	Eccesso 3	2421
0	0000	0011	0000
1	0001	0100	0001
2	0010	0101	0010
3	0011	0110	0011
4	0100	0111	0100
5	0101	1000	1011
6	0110	1001	1100
7	0111	1010	1101
8	1000	1011	1110
9	1001	1100	1111

CODICE DI GRAY


Supponendo di dover eseguire una misurazione di un valore, se questo valore viene misurato nel passaggio tra 7 e 8 (0111 → 1000) senza sapere quando questo cambiamento avverrà e quanto durerà (ovvero senza poter programmare temporalmente il cambiamento) ci sarebbe il rischio di andare a misurare un valore errato poiché viene preso un numero nel mentre del passaggio. Ovvero potrei leggere 1111 o 0000, o anche 1100 dipendentemente dalla velocità di cambio dei bit (i transistor sono tutti diversi, le velocità variano)

Nota: il calcolatore è sincrono soltanto internamente, gli eventi che gli arrivano dall'esterno e che necessitano di misurazione sono asincroni e non prevedibili talvolta (e.g. ABS e inizio frenata sono casuali e non prevedibili)

Per ovviare al problema di misura e sua lettura si usa il **codice di gray**, ovvero un codice per cui al passaggio da una cifra all'altra il cambiamento è sempre di una cifra soltanto, indipendentemente dal valore di partenza e arrivo: così facendo al limite *l'errore che commetto sarà di un'unità*

È un codice riflesso poiché specchia le configurazioni di bit al raggiungimento del massimo numero di cifre.

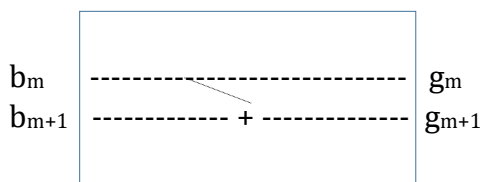
b2	b1	b0	g2	g1	g0
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	0	0



Per passare *da binario a grey* è necessario:

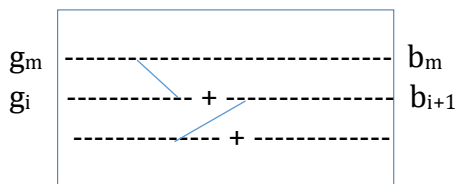
$g_m = b_m$ cifra più significativa è uguale

$g_i = b_i \oplus b_{i+1}$ somma modulo due (senza riporti) di i con il precedente $i-1$



Se invece volessi passare *da grey a binario*:

$$\begin{cases} b_m = g_m \\ b_i = g_i + b_{i+1} \end{cases}$$



In cui risulta evidente come il ciclo più veloce sia quello di passaggio *da binario a grey*, in quanto in un unico momento riesco ad effettuare la conversione (ho già gli elementi necessari)

Per passare *da grey a binario* invece è necessario che per calcolare un elemento io abbia il risultato della somma precedente, ovvero l'i-esimo elemento -1 in binario

? = E' un codice ridondante

Dalla distanza H di Hamming non posso dire niente ($H=1$ può esserlo o non esserlo). Non lo è in quanto, come il codice d'origine, rappresento con il minimo numero di elementi necessari i numeri di cui ho bisogno. Potrei renderlo ridondante rappresentando con tanti bit pochi elementi

RAPPRESENTAZIONI ALFANUMERICHE

Alfabeto esterno: insieme dei caratteri che è in grado di leggere e stampare mediante i dispositivi di I/O

Generalmente per la rappresentazione dei caratteri esterni si usano 7 o 8 bit

EBCDIC (Extended Binary Coded Decimal Interchange Code): 8 bit, rappresenta caratteri alfanumerici e speciali. E' ormai obsoleto

UNICODE: rappresenta con 16 bit (32 nella versione più recente): comprendono anche caratteri speciali e rappresentazioni estere. Rappresenta tutti i caratteri della lingua parlata dell'uomo ed è uno standard dei maggiori produttori nell'ambito dell'IT

ASCII:

American Standard Code for Information Interchange.

Ne esistono due varianti:

- 7 bit: permettono di rappresentare 128 elementi differenti. Non essendo 7 un numero tondo spesso viene arrotondato a 8 bit con l'ottavo che funziona da bit di parità, con ridondanza e $H=2$
- 8 bit: permette di rappresentare 256 simboli ed è detto ASCII esteso. Codifica anche lettere accentate e caratteri grafici; l'estensione di 128 elementi non è standardizzata, pertanto i caratteri in più possono rappresentare alfabeti o elementi differenti a seconda della nazione o dell'alfabeto esterno che si sta utilizzando

Entrambi i codici rappresentano, oltre ai caratteri, anche le 'funzioni' del calcolatore, come spostare il cursore, lo spazio, il tabulatore ecc.

L'effettiva rappresentazione di un numero, come può essere 8072, *dipende dalla convenzione usata e dallo scopo*

REPPRESENTAZIONE DI IMMAGINI

In un calcolatore ogni elemento è *discreto*, mentre la realtà è un elemento continuo (e lo sono anche le immagini)

Per essere rappresentata in un calcolatore un'immagine deve essere *discretizzata*

La discretizzazione avviene attraverso un reticolo di punti detti **pixel**: tramite questo reticolo si può vedere l'immagine come fosse un puzzle di punti, in cui ogni punto rappresenta un colore

Ogni pixel viene codificato da una sequenza di bit.

La *qualità* dell'immagine dipende da due fattori:

- Numero di pixel che la rappresenta
- Numero di bit per pixel (cambia la gamma cromatica e la profondità del colore che posso rappresentare)

Tipologie di rappresentazione:

- *Bianco e nero*: ogni pixel solitamente viene codificato con 8 bit per rappresentare 256 livelli di grigio, partendo dal nero ad arrivare al bianco
- *Colori*: vengono ottenuti tramite la combinazione di almeno 3 colori (le combinazioni RGB, in cui vengono combinati insieme il rosso R, il blu B e il verde G). La composizione può avvenire sia in modo additivo che sottrattivo dipendentemente dalla periferica utilizzata. Per ciascun colore solitamente si usano 8 bit, per un totale di 24 bit a pixel. In questo modo posso rappresentare 2^{24} colori, ovvero c.ca 16 mln

L'occupazione in memoria di un'immagine dipende da *definizione, numero di colori, numero di bit per colore e di conseguenza per pixel*

Tipo immagine	Definizione	Colori	Occupazione
Televisiva	720 x 625	256 (8 bit)	~ 440 kByte
Monitor di PC	1024 x 768	65.536 (16 bit)	1,5 MByte
Fotografica	15.000 x 10.000	16.000.000 (24 bit)	~ 430 MByte

Problema: non è possibile ingrandire a piacimento un'immagine poiché oltre una certa soglia si iniziano a vedere i pixel con l'effetto della frammentazione dell'immagine e dei suoi bordi; aumentare il numero di pixel non sempre è possibile in quanto occuperei una quantità di memoria considerevolmente maggiore.

E' necessaria pertanto una *compressione*:

sostanzialmente la compressione riduce il numero di colori per pixel e aggrega i pixel vicini dello stesso colore in un unico pixel

una di queste tecniche di compressione è il *JPEG*: funziona iniziando una compressione 2x2 di coppie di pixel in cui viene fatta una media della crominanza. Successivamente lo stesso procedimento viene eseguito su blocchi 8x8 di pixel, in cui ciascun blocco viene trattato come un'unità.

Lo standard JPEG può effettuare una compressione che parte da un fattore 10 per arrivare ad un massimo di fattore 30

IMMAGINI VETTORIALI:

Spesso vi sono applicazioni, in particolare quelle meccaniche, elettroniche ecc. in cui è necessaria precisione assoluta e non è possibile ricorrere ad alcuna approssimazione per l'immagine. Per questo si utilizzano le immagini vettoriali, in cui ogni elemento di base (cerchio, linea, triangolo ecc.) rappresenta quelli più complessi ed è descritto soltanto dalle coordinate necessarie, non dai pixel che codificano la linea

Così un cerchio è codificato da tipo di forma, coordinate del centro e ampiezza del raggio

Un triangolo è descritto dalle coordinate dei punti uniti dalla poligonale

...

...

Vantaggi: è una rappresentazione indipendente dalla piattaforma di rappresentazione e dalla sua risoluzione; gli elementi grafici sono indipendenti uno dall'altro e il loro movimento è molto più semplice. L'occupazione in memoria inoltre è molto minore di una codifica bitmap.

Svantaggi: hanno una limitata applicabilità a causa del non poter scomporre in unità fondamentali elementi come una fotografia e vi sono dei limiti imposti dal fatto che per manipolarle sia necessario avere il software che le ha generate o comunque uno compatibile

Esistono diversi formati di file vettoriali:

- PostScript: rappresenta immagini vettoriali e bitmap anche a colori e con diversi formati di compressione.
- DXF: usato da molti programmi di disegno, memorizza i vettori in formato testuale.

IMMAGINI IN MOVIMENTO

Per rappresentare immagini in movimento devo rappresentare una sovrapposizione di immagini in un determinato intervallo di tempo (l'occhio umano vede al massimo a c.ca 30fps)

È tuttavia necessario adottare forti tecniche di compressione in quanto dovendo esserci almeno 30 immagini al secondo, ciascuna del peso di qualche MB, si fa presto a creare filmati di centinaia di MegaByte (con 25 fotogrammi al secondo un minuto di filmato non compresso richiederebbe uno spazio di c.ca 680 MB)

Il formato più diffuso è l'MPEG

ALGEBRA DI BOOLE

$:=$ significa 'è definito da'

$|$ significa oppure

L'algebra di Boole è definita su due elementi, 0 e 1, equivalenti a Falso e Vero rispettivamente.

Dalla combinazione di zeri e uni ottengo come risultati sempre zeri e uni. Nonostante si tratti di un'algebra come può esserla quella euclidea, non valgono le stesse proprietà e quando si parla di 'somma e prodotto' sono sempre intesi in senso logico e non algebrico

Le due costanti sono 0 e 1 e i tre operatori fondamentali sono:

➤ NOT (simbolo \neg)  *talvolta indicato anche senza triangolo, solo con il cerchio

x	\overline{x}
0	1
1	0

➤ AND (prodotto logico, simbolo *) 

x	y	x*y
0	0	0
0	1	0
1	0	0
1	1	1

È vera solo se sono entrambe vere

➤ OR (somma logica, +)



x	y	x+y
0	0	0
0	1	1
1	0	1
1	1	1

È vera solo se è vera almeno una delle due entrate

Funzioni Booleane: sulle variabili booleane può essere definita una funzione booleana logica, in particolare $F(x_1, x_2, \dots, x_n)$ che per ogni n-upla x_i assume valori 0 o 1

Una particolare funzione booleana è definita quando ad essa è associata una *tavola della verità*, ovvero una tabella in cui a determinati input (n-variabili) si fanno corrispondere i corrispettivi output (2^n righe, ovvero 2^n possibili combinazioni)

Teorema di dualità: ogni identità e ogni relazione resta valida se si scambiano gli 0 e gli 1 e si cambiano gli AND con gli OR. L'espressione ottenuta si dice *duale*

Proprietà:

		Duale
	$x \cdot 0 = 0$	$x + 1 = 1$
	$x \cdot 1 = x$	$x + 0 = x$
Idempotenza	$x \cdot x = x$	$x + x = x$
Complementazione	$x \cdot \bar{x} = 0$	$x + \bar{x} = 1$
Prop. commutativa	$x \cdot y = y \cdot x$	$x + y = y + x$
Prop. associativa	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	$(x + y) + z = x + (y + z)$
Prop. Distributiva	$x \cdot (y + z) = x \cdot y + x \cdot z$	$x + (y \cdot z) = (x + y) \cdot (x + z)$
Prop. assorbimento	$\begin{cases} x + x \cdot y = x \\ x \cdot (\bar{x} + y) = x \cdot y \end{cases}$	$\begin{cases} x \cdot (x + y) = x \\ x + (\bar{x} \cdot y) = x + y \end{cases}$
De Morgan	$\begin{aligned} \overline{x \cdot y} &= \bar{x} + \bar{y} \\ &= x = x \end{aligned}$	$\begin{aligned} \overline{x + y} &= \bar{x} \cdot \bar{y} \\ &= x = x \end{aligned}$

Nota: per dimostrare una proprietà posso ricorrere a due metodologie:

- Creare la tabella delle verità della sequenza rappresentata
- Ricorrere a proprietà note tra quelle elementari e utilizzarle per la dimostrazione

(sempre più conveniente la tabella, in quanto con le formule potrei non arrivare a conclusione o dilungarmi troppo in rapporto all'uguaglianza da dimostrare)

Gli operatori AND e OR sono utilizzabili come operatori di mascheratura:

- AND: se voglio azzerare una parte di una stringa e mantenere immutata l'altra pongo un And a 0 della parte da eliminare e a 1 della parte da non modificare
- OR: se voglio azzerare tutto e tenere/alzare soltanto un bit pongo un OR tutto a zero tranne il bit interessato a 1

OPERATORI UNIVERSALI

Sono quegli operatori che, come si evince dal nome, possono essere combinati per formare tutti gli operatori dell'algebra booleana.

NAND ($/$) = negazione dell'AND

$$1 / 1 = 0$$

$$1 / 0 = 0 / 1 = 0 / 0 = 1$$

simbolo grafico 

x	y	x / y
0	0	1
0	1	1
1	0	1
1	1	0

NOR (\downarrow) = negazione dell'OR

$$0 \downarrow 0 = 1$$

$$1 \downarrow 0 = 0 \downarrow 1 = 1 \downarrow 1 = 0$$

simbolo grafico 

x	y	x \downarrow y
0	0	1
0	1	0
1	0	0
1	1	0

Proprietà:

$$x \downarrow 1 = 0$$

$$x \downarrow 0 = \bar{x}$$

$$x \downarrow (y \downarrow z) \neq (x \downarrow y) \downarrow z$$

$$x \downarrow x = \bar{x}$$

$$\bar{x} \downarrow \bar{y} = x \cdot y$$

$$\overline{x \downarrow y} = x + y$$

$$x / 0 = 1$$

$$x / 1 = \bar{x}$$

$$x / (y / z) \neq (x / y) / z$$

$$x / x = \bar{x}$$

$$\bar{x} / \bar{y} = x + y$$

$$\overline{x / y} = x \cdot y$$

Solo NOR

Not $\overline{x+0} = \bar{x} = \overline{x+x}$

Or $\overline{\overline{x+y+0}} = \overline{\overline{x+y+x+y}} = x+y$

And $\overline{\overline{x \cdot y}} = \overline{\overline{x+y}} \rightarrow \overline{x \cdot y} = \overline{x+y}$

Solo NAND

Not $\overline{\overline{x \cdot 1}} = \bar{x} = \overline{x \cdot x}$

And $\overline{\overline{\overline{x \cdot y \cdot 1}}} = \overline{\overline{\overline{x \cdot y \cdot x \cdot y}}} = x \cdot y$

Or $\overline{\overline{x+y}} = \overline{\overline{\overline{x \cdot y}}} \rightarrow x+y = \overline{\overline{x \cdot y}}$

Porte con le Universali:

Realizzazione tramite

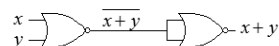
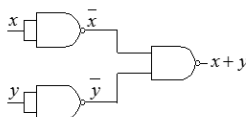
NAND

NOR

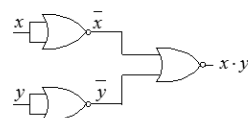
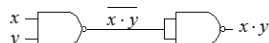
NOT



OR



AND



EXOR: è detto anche sommatore modulo 2, già visto in codici ciclici, nel c. gray, nella somma della ALU e come invertitore di segno nella rappresentazione in complemento n-1)

È inoltre utilizzato come bit di parità pari: se infatti preso un qualunque numero binario eseguo l'exor su di esso a due a due partendo dalla cifra più significativa ottengo 1 se il numero di 1 è dispari, 0 se è pari

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

Il simbolo grafico è il \oplus

$$f(x, y) = \bar{x} \cdot y + x \cdot \bar{y} = x \oplus y$$

$$x \oplus 0 = x$$

$$x \oplus 1 = \bar{x}$$

$$x \oplus x = 0$$

$$x \oplus \bar{x} = 1$$

$$x \oplus y = y \oplus x \quad \text{commutativa}$$

$$(x \oplus y) \oplus z = x \oplus (y \oplus z) \quad \text{associativa}$$

Se volessi ad esempio riportare a 0 l'accumulatore, quale sarebbe più veloce come istruzione?

- LD A,0
- XOR A,A

La migliore delle due è la seconda, poiché utilizza elementi già presenti all'interno dei registri e nell'istruzione di fetch, unica, contiene anche l'operando. La prima istruzione invece non consente di essere eseguita in un solo ciclo di fetch ma ne prevede due: uno per l'istruzione ed uno per il prelevamento dell'operando

ISTRUZIONI CPU:

La CPU può eseguire soltanto alcune tipologie di istruzioni:

- *Trasferimento dati*: è il compito di copiatura di file dalla memoria ai registri e viceversa, da registro a registro, da registro a unità logica e viceversa. Supponendo A e B nomi di registri esempi di comandi possono essere

$$\text{LDA <indirizzo>} \quad (\text{indirizzo}) \rightarrow A \quad \text{STB<indirizzo>}$$

$$\text{MOV A,B} \quad \text{INP<porta>, B} \quad (\text{porta}) \rightarrow B$$

$$\text{OUT A <porta>} \quad (A) \rightarrow \text{porta}$$
- *Istruzioni aritmetiche*: somma (o sottrazione) con relativi registri e flag (riporto, over/underflow)

Esempi di queste istruzioni possono essere:

ADD <indirizzo>, A ADC<indirizzo>, A SUB<indirizzo>
(indirizzo)+(A)→A <indirizzo>+(A)+(Cy)→A*

*Cy indica la somma con riporto/differenza con prestito, utile ad esempio quando sto eseguendo operazioni che superano la capacità del mio processore e richiedono pertanto più passaggi per essere eseguite

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

- *Istruzioni logiche*: permettono di eseguire le istruzioni logiche su stringhe di bit memorizzate nei registri. Le operazioni AND, OR, NOT, EXOR operano su coppie di bit corrispondenti.

(indirizzo) * A (indirizzo)+A NOT B XOR<indirizzo> A

- *Rotazione e shift*: spostano i bit all'interno di una parola.

Shift: il bit più significativo viene spostato verso sx o dx a seconda della direzione dello shift e il bit più/meno significativo diventa lo zero, con il carry flag che salva il bit d'avanzo dalla traslazione

E' utile ad esempio se devo eseguire una traslazione per far combaciare filo di trasmissione e parola o se devo eseguire, equivalentemente alla divisione per 10 dei numeri decimali esatti, una divisione esatta per due (*shift a dx*).

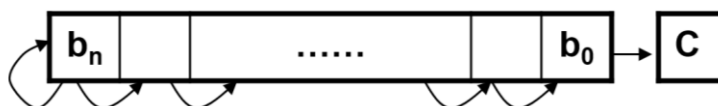
(Nota: quest'ultimo caso va bene se sono in v.a., infatti se ad esempio 1111 1110 lo shifto

ottengo 0111 1111, che in v.a è la divisione per due, ma in complemento passo da -2 a 127)

esistono due tipologie di shift:

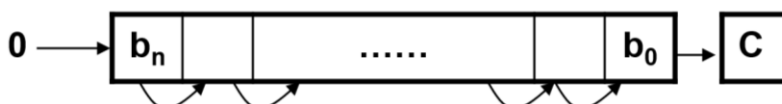
> shift aritmetico: in cui viene shiftata la parola ma il bit più (o meno, a seconda della direzione di shift) significativo viene mantenuto nella posizione in cui è, per eseguire divisioni/moltiplicazioni in complemento

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



> shift logico: in cui viene shiftata la parola senza mantenere il bit più/meno significativo, usato per le operazioni su numeri in v.a.

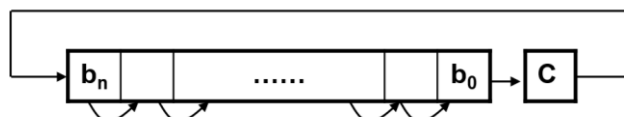
SRL A (shift logico a destra di (A))



Il **bit del carry** nel caso di divisione mi da l'eventuale resto della divisione (ovvero la parità/disparità del numero di partenza), mentre nel caso delle moltiplicazioni mi da il bit più significativo del numero di partenza

Rotazione: operazione di shift in cui il bit 'in uscita' viene riportato in ingresso nel posto lasciato libero dalla traslazione. Utile ad esempio se deve eseguire un'operazione in due momenti, dove nel primo eseguo una parte dell'operazione la cui uscita è l'ingresso della seconda

RR A (rotazione a destra di (A))



ISTRUZIONI CONTROLLO PROGRAMMA:

Sono istruzioni che modificano l'ordine sequenziale di esecuzione del programma e delle istruzioni che lo compongono. Sono le cosiddette istruzioni di salto.

Esistono due tipologie di salto:

- Condizionato: se l'esecuzione o meno dell'istruzione di salto dipende dal verificarsi o meno di una condizione del registro di flag specifico
- Incondizionato: se il salto viene eseguito a prescindere dal verificarsi o meno di condizioni particolari (e.g. se non è implementata la funzione seno e devo andare a pescarla in un sottoprogramma)

A loro volta ciascuna di queste due categorie può essere suddivisa in altre due categorie:

- Senza ritorno: in cui si continua dalla subroutine senza tornare all'istruzione successiva a quella di salto

I salti senza ritorno necessitano di più parametri: - istruzione - indirizzo - condizione (se condizionato) - spostamento (se salto relativo)

e.g:

* JP <INDIRIZZO>	incondizionato assoluto	(1)
* JP <COND><INDIRIZZO>	condizionato assoluto	(2)
* JR<SPOSTAMENTO>	incondizionato relativo (al PC)**	(3)
* JR<CONDIZIONE><SPOSTAMENTO>	condizionato relativo (al PC)	(4)

** nel salto relativo non vado a definire un indirizzo di arrivo del salto ma di quando spostarmi relativamente alla posizione le program counter

Le 4 operazioni sono, rispettivamente:

INDIRIZZO → PC (1)

Se la condizione è verificata, INDIRIZZO → PC, altrimenti non si fa niente (nella fase di fetch il PC è automaticamente aggiornato con l'indirizzo dell'istruzione successiva a quella attuale) (2)

(PC) + DISPLACEMENT → PC (3)

Displacement rappresentato in complemento a 2: salti avanti o indietro rispetto alla posizione del PC

Se la condizione è verificata, (PC) + DISPLACEMENT → PC, altrimenti non si fa niente (PC punta già all'istruzione successiva a quella attuale) (4)

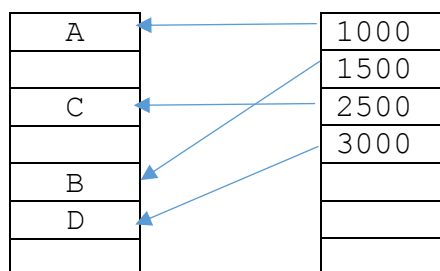
- Con ritorno: sono chiamate di subroutine in cui è necessario non distruggere il contenuto che il PC ha prima dell'esecuzione del salto (alla fine della lettura dell'istruzione attuale, caricando il valore dell'indirizzo dell'istruzione a cui si vuole saltare)

Grado di parallelismo: # di bit che possono essere letti o scritti contemporaneamente in una memoria

Spazio di indirizzamento: è il numero di byte che possono essere indirizzati: 2^N dove $N = \# \text{ bit usati}$

Esistono varie tipologie di indirizzamento, dipendentemente dallo scopo e dalla memoria utilizzata:

- **Immediato**: l'operando compare direttamente nell'istruzione come costante nel ruolo di "sorgente" numerica per l'operazione. Equivale ad un'unica fase di fetch;
e.g. `ADD A, 152` `LD A, 7`
- **Assoluto**: nel comando trovo al posto dell'operando l'indirizzo di memoria in cui andare a reperire l'informazione di cui ho bisogno.
e.g. `LD A, (1000)` `JP (2400)`
- **Relativo**: nel programma compare un numero che rappresenta lo spostamento da attribuire al program counter per ottenere l'indirizzo di memoria desiderato. Consente di avere programmi *autorilocabili*, che funzionano senza modificare gli indirizzi in essi espressi al variare della disposizione in memoria di comandi e pc
e.g. `JP +10`
- **Diretto a registro**: l'operando invece che trovarsi in una cella di memoria è contenuto in un registro e all'interno dell'istruzione è contenuto il registro di riferimento in cui trovare il valore.
- **Indiretto a registro**: come il precedente, con la differenza che questa volta il registro non contiene l'operando in se ma la cella di memoria alla quale reperire il valore cercato
- **Con autoincremento**: il contenuto del registro viene prima utilizzato per indirizzare la memoria e poi incrementato della dimensione dell'operando (n-byte); molto usato per le letture sequenziali di dati, in cui le celle sono consequenziali
- **Con autodecremento**: è il duale del precedente; inizia da un'allocazione in memoria e decrementa ogni volta il contenuto del registro che indica l'allocazione in memoria.
- **Indiretto con autoincremento**: l'operando è in memoria con l'indirizzo che si trova in un'altra posizione della memoria, puntata dal contenuto di un registro. Nell'istruzione viene specificato l'identificatore del registro. L'operando dopo essere stato utilizzato viene incrementato d un numero pari alla dimensione in byte di una cella di memoria per contenere un indirizzo.



- **Spiazzamento**: nell'istruzione sono specificati un dato in complemento a 2 e l'identificatore di un registro di memoria. Il dato viene sommato al contenuto del registro e si ottiene l'indirizzo dell'operando da tale somma
- **Indiretto con spiazzamento**: come sopra con la differenza che l'indirizzo ottenuto dalla somma punta ad una posizione di memoria dove è contenuto l'indirizzo dell'operando, e non l'operando stesso
- **Implicito**: è l'indirizzamento previsto in alcune tipologie di processori dove il relativo codice operativo sottintende l'utilizzo dei registri. Il caso più tipico è l'utilizzo dell'accumulatore sottinteso quando espresso come sorgente di uno degli operandi.
e.g. `SUB B`

- **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. Il numero ottenuto è l'indirizzo di memoria dell'operando. Molto efficiente come indirizzamento per l'analisi di tabelle e matrici
- **Con lo stack pointer:** è la modalità attraverso la quale posso gestire le subroutine. Infatti quando avviene una sottochiamata o un interrupt devo interrompere ciò che sta facendo la CPU, eseguire l'interrupt e successivamente riprendere da dove avevo interrotto senza perdita dell'informazione fino a lì elaborata. Il problema sorge dal fatto che i registri utilizzati sono gli stessi, sia per l'interrupt che per le operazioni di routine della CPU.

Devo pertanto mettere da parte gli indirizzi usati e a interrupt finito riprenderli. Il problema maggiore in questa soluzione sarebbe la non velocità, in quanto nell'istruzione avrei *soltanto indirizzi oltre all'istruzione*, con una conseguente ripetitività dei cicli di fetch. Per velocizzare tutto il processo posso scrivere senza specificare, sottintendendo lo stack (che per costruzione si autoaggiorna)

Sono le cosiddette istruzioni di push e pop:

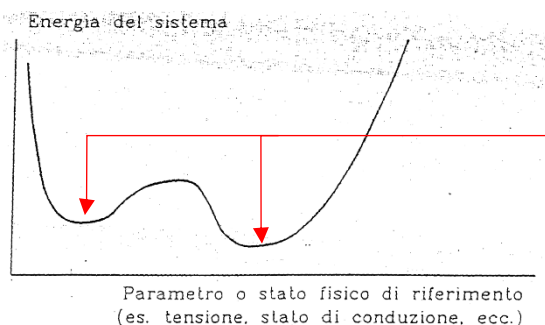
- *push* aggiunge un dato in cima alla lista
- *pop* preleva il dato in cima allo stack

Così facendo lo SP punta alla sommità dello stack e l'istruzione di salvataggio della routine interrotta invece che contenere tutti gli indirizzi contiene soltanto l'istruzione di scrittura/prelevamento. Sottintendendo il resto.

DISPOSITIVI DI MEMORIA

Le tecnologie attualmente utilizzate per la memorizzazione di informazioni digitali sono:

- Memorie a semiconduttore



- Memorie a supporto magnetico

Entrambe le memorie si basano sui c.ti bistabili, ovvero circuiti caratterizzati da due condizioni di stabilità a cui corrispondono i due valori logici fondamentali, 0 e 1

In cui si vedono due minimi assoluti e un massimo relativo a separarli *

*il massimo relativo deve essere abbastanza elevato da non consentire il passaggio da uno all'altro stadio per errore, ma non eccessivamente elevato da determinare un consumo non necessario di energia per il passaggio

Dal punto di vista pratico si tratta di un dispositivo in grado di rimanere in una delle due condizioni stabili finché non interviene una forza esterna che lo tramuta nella condizione duale (i due stati stabili sono le condizioni *a minima energia*)

Associando convenzionalmente ai due stati un 1 e uno 0 è possibile immagazzinare e leggere stringhe di dati, associati appunto a questi due valori dei bistabili

Due elementi importanti delle memorie sono:

- Modalità di accesso
- Stabilità dell'informazione memorizzata e suo mantenimento

RAM (Random Access Memory)

I due tipi esistenti sono le ram statiche e le ram dinamiche.

SRAM: la cella elementare è composta da un circuito bistabile simmetrico composto da

transistor disposti in modo che se da un lato della cella elementare alzo la tensione dall'altra si abbassa e viceversa. In questo modo associando da una o l'altra condizione i due valori logici 1 e 0 posso rappresentare l'informazione. Hanno il vantaggio di essere molto stabili nel mantenere l'alimentazione (finché sono alimentate), ma per avere una capienza notevole occupano molto spazio e utilizzano numerosi dispositivi attivi

DRAM: le ram dinamiche invece sono costituite soltanto da un dispositivo attivo che sfruttando le capacità parassite mantengono l'intera cella elementare alta/bassa in tensione. Hanno il vantaggio di occupare molto meno spazio per grandi capacità, ma perdono in stabilità dell'informazione in quanto non essendo c.ti ideali tendono a scaricare la capacità, con conseguente perdita di informazione. E' pertanto necessario che siano *refreshate* a ΔT predefiniti (nell'ordine dei 10^{-3} secondi).

Ovviamente non può essere la CPU a occuparsi del refresh poiché ciò implicherebbe un consumo di risorse inutile, è compito del *Dynamic Ram Controller*.

Tendenzialmente le più veloci sono le SRAM, le più economiche e di maggior capacità le DRAM

RAM SINCRONE: offrono la possibilità di trasferire blocchi interi di informazione invece che lavorare per celle elementari. Solitamente si specifica l'indirizzo di partenza e la lunghezza del blocco che viene mosso. Il trasferimento è il più rapido delle 3 Ram poiché si genera solo il primo indirizzo e un segnale di clock sincronizza la sequenzializzazione dei dati. Hanno tuttavia un funzionamento non corrispondente a quello a-sequenziale della CPU, pertanto sono utili in determinati contesti specifici

Le celle sono disposte a matrice e l'accesso avviene per riga/colonna. Sono utilizzate solitamente per le memorie Cache

ROM (Read Only Memory)

Sono memorie a sola lettura (nome ereditato dalle prime memorie di questo tipo, in cui una volta prodotte con lo schema dati deciso non potevano essere più riscritte)

ROM: l'informazione veniva scritta in fabbrica e non è più modificabile post-produzione

PROM: sono Rom programmabili. Vengono vendute con tutti i transistor che compongono le celle a 0 collegati a dei fusibili. Per programmarle porto a 1, rompendo i fusibili, le celle elementari che voglio a tensione alta e lascio a zero le celle che voglio a 0.

Hanno il grosso difetto di non essere sovrascrivibili, pertanto una volta rotto un fusibile non sono più cambiabili di stato

EPROM: come la Prom può essere programmata dall'utente; tuttavia questa può essere anche sovrascritta. E' necessario prima, attraverso una finestrella sulla cella base, azzerarne lo stato con la luce ultravioletta. Il difetto grosso delle Eprom è che capitava spesso che venissero cancellate per sbaglio a causa di esposizioni involontarie a luce UV

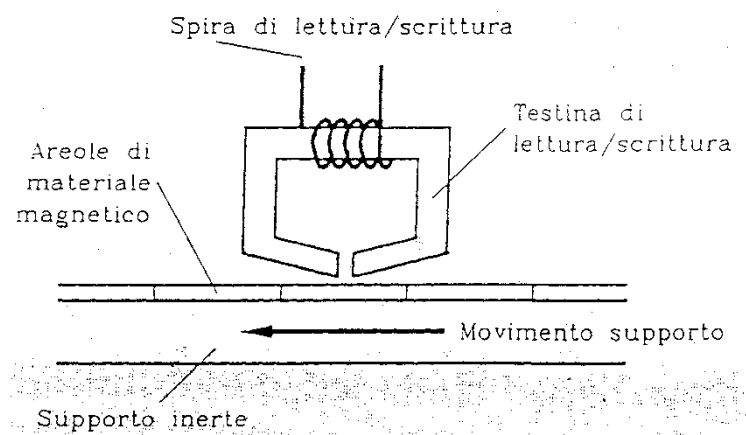
Per rimediare sono state introdotte le OTP ROM in cui l'utente poteva programmare la Rom ma, non essendoci finestrelle, non poteva resettarne lo stato. Venivano prodotte per la produzione in scala delle memorie testate su Eprom a raggi Uv.

E²PROM: sono l'equivalente delle Eprom con la differenza che la loro cancellazione avviene elettricamente, pertanto non necessitano di luce Uv e sono molto più controllabili.

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 depositato su un supporto plastico o ceramico (deve essere abbastanza rigido da non flettere in lettura ma altrettanto leggero e neutro elettricamente)

I due stati stabili corrispondono ai due stati logici e all'orientamento fisico in senso magnetico del supporto.

SCRITTURA: il solenoide avvolto alla testina viene attraversato da corrente elettrica in una direzione dipendente dal dato che voglio scrivere (in base alla direzione avrò infatti un diverso campo magnetico con conseguente orientamento differente degli elementi ferromagnetici sul supporto, facendo la differenza tra 1 o 0)

LETTURA: avviene il procedimento inverso, con la spira scarica e il supporto già polarizzato in precedenza. Questo orientamento altera il potenziale della spira, inducendo un campo elettrico al suo interno direzionato in uno o nell'altro senso a seconda della direzione di orientamento dell'informazione nell'areola di memoria.

Per *migliorare la velocità* di un supporto magnetico si è ricorsi alla riduzione delle misure geometriche, portando a notevoli vantaggi dal punto di vista della velocità di r/w e di capacità di memorizzazione. Tuttavia vi sono dei limiti non valicabili, quali la dimensione delle areole senza generare interferenze o l'eccessiva vicinanza della testina al supporto, che potrebbe causare un contatto dei due con conseguente distruzione del supporto stesso.

Svantaggi:

- L'essere dipendente da movimenti fisici implica una certa fragilità dal punto di vista strutturale e di sopportazione delle sollecitazioni rispetto alle memorie flash.
- Forti campi magnetici possono alterare o addirittura annullare l'informazione presente all'interno della mia unità di memoria.

*come risolvo il fatto che i tempi di lettura siano nell'ordine massimo di qualche millesimo di secondo mentre la Cpu lavora nell'ordine di 10^{-9} ? Oltre alla situazione in cui ho i dati in fila come ottengo una velocità di lettura/scrittura sufficiente a non perdere rapidità di calcolo della Cpu? **Il danny forse risponderà, o forse no***

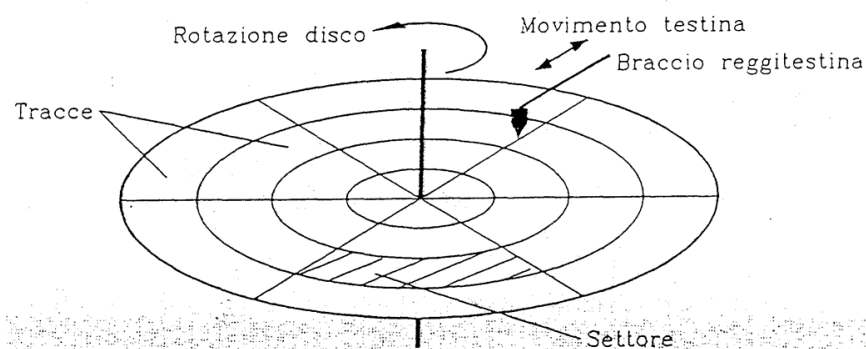
Caratteristiche comuni dei dispositivi di memorizzazione a supporto magnetico sono:

- l'elevata quantità di informazione immagazzinabile (da alcune centinaia di Gbyte a qualche Tbyte)
- 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)
- 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

Possono esistere, al fine di migliorare le prestazioni, dischi a testine fisse (molto costosi) o dischi che sfruttano entrambe le facce per la lettura/scrittura, entrambe con la loro testina e più di una alla volta, visti dalla CPU come unica allocazione di memoria (*dischi coassiali*)

Parametri tipici:

- densità lineare di memorizzazione: > 500.000 bit/pollice
- numero di tracce per pollice: > 10.000
- velocità di rotazione: da 7200 a 15.000 rpm (giri/minuto)
- velocità di spostamento del braccio: 30 ms attraverso tutte le tracce
- velocità di trasferimento: 10 - 60 MByte/secondo
- capacità di memorizzazione: 256 Gbyte ÷ 4 Terabyte

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, essi sono visti come un unico disco con i dati distribuiti su più di un disco (ridondanza) al fine di recuperare i dati in caso di guasti

FLOPPY DISK

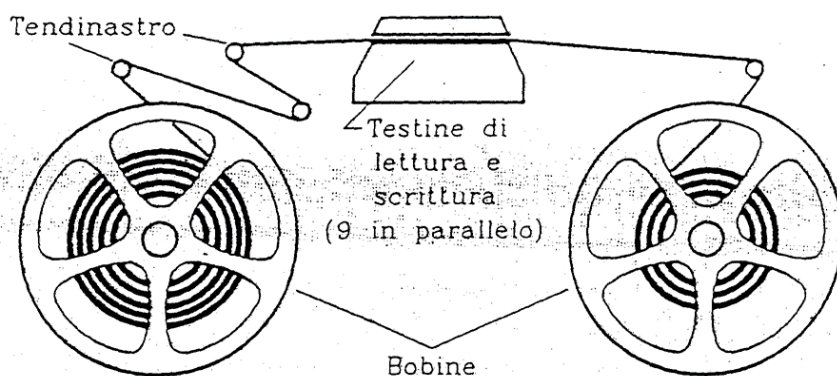
I floppy disk sono normalmente fermi e quindi il tempo di accesso alle informazioni (circa 0.1 s) è più lungo rispetto a quello dei dischi rigidi (dato dal tempo di posizionamento della testina più il tempo necessario a raggiungere l'informazione)

I più diffusi sono da 3.5 pollici (capacità di 720 KByte o 1.44 MByte). Anni orsono si usavano anche da 5.25 pollici (capacità di 360 KByte o 1.2 MByte) o da 8 pollici (capacità da 128 o 256 Kbyte)

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 mediante trasporto di dischi magnetizzati

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*)

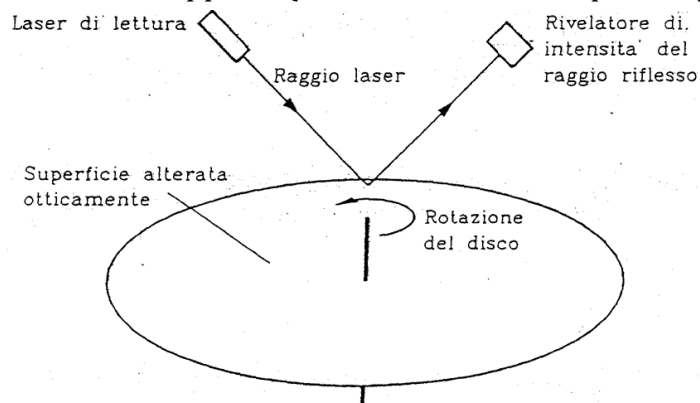
Il nastro è normalmente fermo e trasferisce un blocco per volta

Parametri tipici:

- velocità di trascinamento: 100 – 200 pollici/secondo
- larghezza del nastro: $\frac{1}{4}$ – $\frac{1}{2}$ pollice
- lunghezza del nastro: 350 – 1.000 m
- densità di memorizzazione: 25 MByte/pollice
- capacità: 100 GByte – 1 TByte

DISCHI OTTICI:

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 la presenza di una deformazione provoca una dispersione di energia luminosa.

Le variazioni di tensione di tale dispositivo consentono di ricostruire l'informazione presente sul disco

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

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

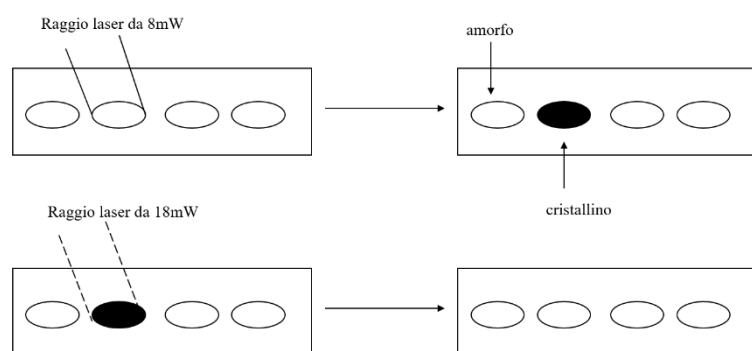
Mette a disposizione archivi enormi se si ha una configurazione a "juke box" in cui più dischi possono essere prelevati ed inseriti in modo automatico

CD RISCIVIBILI:

CD-ReWritable detto anche CD-E (Erasable) sono cancellabili e riscrivibili. Usano la tecnologia del phase-change.

Sandwich di polycarbonato e tellurio o selenio in cui il bit è scritto come spot usando un laser da 8mW oppure da 18mW. La lettura avviene tramite un laser di minore potenza

Riscrivibile direttamente con lo stesso laser di lettura, a seconda del wattaggio utilizzato



Ogni spot può essere in stato amorfo o cristallino

DVD

Digital Video Disk, hanno lo stesso diametro dei CD-rom ma sono molto più capienti. Utilizzano strati differenti di scrittura/lettura a seconda della riflettanza e, di conseguenza, dell'intensità della luce che viene loro proiettata contro. Pertanto sulla stessa faccia posso avere una capacità di memorizzazione molto superiore ai CD-rom

ACCESSO ALLA MEMORIA

Come è possibile rendere compatibili tempi nell'ordine di 10^{-8} della CPU con i tempi di lettura dei supporti fisici nell'ordine dei 10^{-4} ? Variando la modalità di accesso in modo da ottenere il massimo dai supporti fisici per determinati compiti (e.g. accesso sequenziale) e dai supporti a semiconduttore (e.g. accesso casuale)

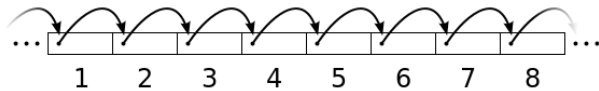
Tipi di accesso:

- **uniforme/casuale:** accesso casuale inteso come "uniforme come richiesta di tempo indipendentemente dal settore richiesto". L'esempio di accesso casuale è la Ram o le memorie a semiconduttore, in cui a qualunque zona io acceda i tempi richiesti sono i

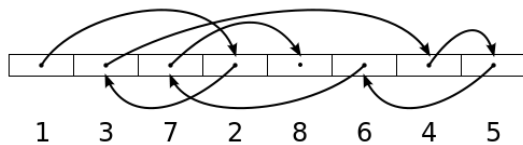
medesimi. Non sono ad accesso casuale ad esempio i nastri delle videocassette, in cui per accedere alla locazione X devo saltare tutte le locazioni precedenti a essa

- **sequenziale:** significa che l'accesso alla memoria avviene in una sequenza preordinata, come può avvenire nel nastro magnetico. Può talvolta essere l'unico modo di accedere ai dati, come può accadere sui nastri delle cassette, o può essere un metodo scelto come avviene per i supporti magnetici, in cui l'accesso sequenziale limita al minimo gli spostamenti della testina magnetica velocizzando l'operazione
- **diretto/misto:** equivalente all'accesso diretto casuale, con la differenza che avviene sui supporti fisici; ciò implica il dover contare nei tempi d'accesso anche la posizione relativa di lasero o testina

Accesso sequenziale



Accesso casuale



A livello funzionale ogni dispositivo di immagazzinamento è un contenitore di sequenze di bit equivalenti a parole (di bit) che a loro volta rappresentano l'informazione salvata. Tuttavia esistono delle differenze sostanziali tra i tempi di accesso di una memoria piuttosto che un'altra, dipendentemente anche dalla tecnica di accesso utilizzata. È necessario trovare il giusto compromesso tra dimensione della memoria al minor costo (a vantaggio delle memorie magnetiche, più lente ma molto più capienti) e la velocità di lettura/scrittura necessarie per stare al passo con i tempi dei calcolatori (preferendo in questo caso memorie a semiconduttore per la loro velocità pressoché identica a quella della CPU)

Per la gestione delle memorie all'interno di un calcolatore è stata introdotta la *gerarchia delle memorie*.

Potrei costruire sia calcolatori solo a semiconduttori che calcolatori solo a supporti magnetici, ma nel primo caso avrei dei costi elevatissimi non necessari nel caso di utilizzo sequenziale della memoria e un'usura maggiore rispetto ai supporti fisici, nel secondo avrei una lentezza elevatissima nel caso di utilizzo casuale della memoria. Per questo si combinano le due tipologie di memorie per avere il meglio di entrambe in un calcolatore

Principio di località degli accessi: se la CPU sta eseguendo una data istruzione presente in memoria, vuol dire che con molta probabilità le prossime istruzioni da eseguire saranno ubicate nelle vicinanze di quella in corso; vale a dire che, durante la normale esecuzione dei programmi, la CPU passa molto tempo accedendo a zone di memoria ristrette e solo occasionalmente accede a locazioni molto lontane; questo permette di velocizzare molto l'accesso alla memoria tramite lo spostamento preventivo di grosse zone della memoria stessa, non del singolo dato, all'interno di memorie più veloci (e.g. cache)

Questo principio si basa su due località:

- *Località temporale:* ogni programma ha un'elevata probabilità di riutilizzare a breve un'informazione appena utilizzata
- *Località sequenziale:* ogni programma ha un'elevata probabilità di utilizzare un'informazione adiacente a quella appena utilizzata (idem per i dati)

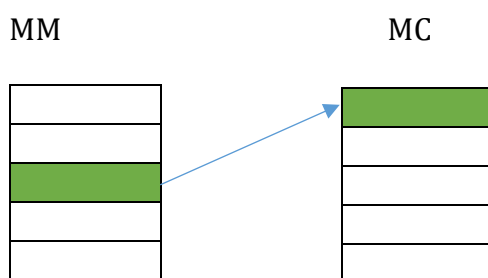
Questo processo permette di *virtualizzare* la memoria: ossia di rendere la memoria più capiente e meno rapida come memoria di storage mentre la più veloce carica parte dell'informazione contenuta nella prima (selezionandola secondo le due località) e "simulando" per la CPU una

memoria unificata.

Agli occhi della CPU infatti la memoria risulterà essere unica, quando in realtà essa è divisa tra:

- Memoria di massa: ovvero la memoria più capiente ma più lenta, utilizzata effettivamente per immagazzinare dati e programmi, viene copiata parzialmente in una più veloce.
- Memoria centrale: ovvero la memoria più rapida, più piccola di quella di massa, utilizzata per copiare porzioni di dati e programmi e visualizzata dalla CPU come fosse la memoria di massa (avviene traduzione dell'indirizzo fisico della MM in virtuale della MC)

In questa gestione della memoria intere porzioni della MM sono copiate nella MC dipendentemente dalla probabilità e dalla frequenza di utilizzo (ciò che è utilizzato meno viene eliminato dalla MC) Se un indirizzo necessario alla CPU si trova all'interno della MC un apposito dispositivo "traduce" l'indirizzo fisico della MM nel corrispondente virtuale della MC. Qualora non fosse presente il blocco necessario all'interno della MC si provvederà a prelevare il suddetto dalla MM e a copiarlo nella MC



MMU: (Memory Manager Unit) è il gestore della virtualizzazione della memoria, controlla che il blocco richiesto sia presente nella memoria centrale. Se è presente *traduce l'indirizzo fisico della MM nel corrispondente virtuale della MC*. Se invece non è presente *provvede a indirizzare la CPU nella MM e a copiare il blocco interessato nella MC*.

E' anche il responsabile dell'eliminazione di una cella di memoria dopo troppo tempo che questa non viene utilizzata e della gestione delle frequenze di uso dei vari dati (lo spostamento di dati in blocchi multipli spiega l'utilità delle Ram sincrone)

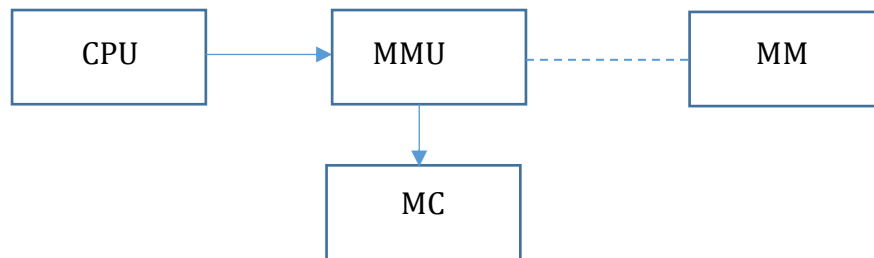
SWAPPING: è la procedura che precede la copiatura in MC della porzione desiderata di MM, per la quale è necessaria una copiatura inversa al fine di liberare spazio sulla MC. Per il principio di località sopracitato non viene spostato il singolo indirizzo di memoria ma l'intero blocco che lo contiene.

Svantaggi: devo cercare di avere soltanto qualcosa che effettivamente mi serva, per evitare di avere, in un settore copiato, la necessità soltanto di pochi byte; altrimenti si produce una percentuale di utilizzo scarsa sul settore e questo rischia di essere eliminato dalla MC. Inoltre devo trovare un modo efficiente di controllare nell'immediato la presenza o meno del dato e del settore all'interno della MC

Le **informazioni eliminate** vengono:

- *Eliminate* se non ne ho modificato il contenuto e la copia è uguale all'originale nella MM, pertanto copiarla nuovamente porterebbe ad una perdita di tempo
- *Copiate nuovamente* nella MM se le ho modificate e ho bisogno di salvare i cambiamenti che ho apportato

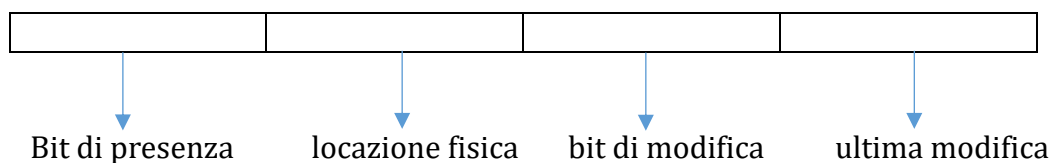
Nota: la copia non avviene in modo alternato soltanto per le zone di memoria modificate ma si copia nuovamente l'intero settore in cui è presente il dato modificato; questo al fine di velocizzare la riscrittura in quanto su supporti più lenti come può essere la MM la scrittura in sequenza è molto più veloce della casuale



Nella MM il numero di bit per la memorizzazione dipende dalla capacità di memorizzazione della memoria (e.g. con 1TB necessita di 40 bit). Nella MC invece il numero di bit necessari per ogni indirizzo è dato semplicemente dalla capacità di rappresentazione degli indirizzi della MC. L'mmu lavora come traduttore tra indirizzo fisico e indirizzo virtuale

Paginazione: si basa sul concetto di pagina, ovvero di blocco di parole consecutive di dimensione pesata (pochi KB) che costituisce l'unità minima di informazione trasferita dalla MC alla MM e viceversa. In questa gestione le memorie vengono divise in settori (*pagine*) delle medesime dimensioni e ogni programma occupa più pagine la cui dislocazione è libera dal vincolo di consecutività. La conversione da indirizzo virtuale a fisico consiste nella ricerca della posizione di memoria fisica nella quale la virtuale referenziata verrà inserita. Ciò avviene con una corrispondenza tra tabella che fa una corrispondenza tra pagine virtuali e fisiche

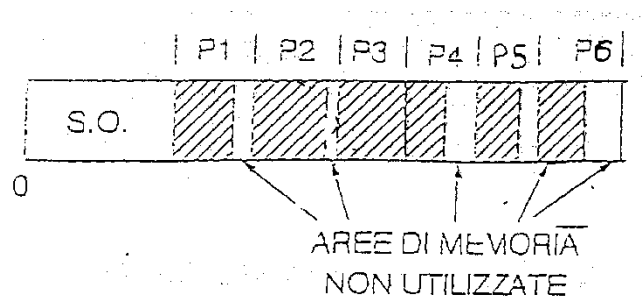
Per **controllare la presenza del dato** all'interno della MC è necessario avere un check immediato e rapido. Pertanto si tabulano i settori della MM, in ogni riga un settore, e si aggiunge un bit indicatore di presenza in caso di informazione nella MC oltre alla locazione fisica dell'informazione nell'MC. Tale tabella si trova all'interno del MMU, poiché è lui l'incaricato a verificare presenza o meno del dato e comandare un eventuale copiatura.



Il *bit di ultima modifica* è fondamentale in quanto permette di stabilire una gerarchia di eliminazione delle informazioni all'interno della MC quando è necessario fare spazio alla MM per una nuova scrittura

Il *bit di presenza* definisce se l'elemento cercato è presente o meno all'interno della MC, in caso contrario l'MMU invierà richiesta di copiatura alla MM nella MC

Questa paginazione è più lenta nell'organizzazione iniziale e richiede un costo di memoria maggiore, ma è molto più rapida in consultazione di una soluzione in cui si rappresentano solo gli indirizzi presenti



Problema: gli spazi non utilizzati in figura accadono perché la paginazione prevede blocchi di memoria di dimensione fissata. Tuttavia se un programma utilizza un blocco di memoria e una parte di un altro, la parte non utilizzata dell'altro rimarrà, appunto, inutilizzata e occuperà memoria inutilmente.

Lo stesso tipo di frammentazione si verifica quando la quantità di memoria da salvare è minore delle dimensioni di una pagina. In questo caso si parla di frammentazione interna.

E' necessario trovare la giusta via di mezzo tra la dimensione di blocchi e il loro numero; se infatti prendessi blocchi troppo grandi avrei un tempo migliore di utilizzo, ma una minore efficienza dovuta al crescere degli spazi inutilizzati. Viceversa con blocchi più piccoli ma più numerosi avrei un utilizzo più efficiente dei blocchi ma la necessità di leggere/scrivere in memoria molte più volte del necessario.

e.g. con pagine di 4KB

Poiché vi sono pagine lunghe 4096 byte ($4096-1 = 4095$ può essere rappresentato in forma binaria da 12 bit), la MMU usa per indirizzarle i primi 4 bit come numero di pagina, ed i successivi 12 per l'indirizzo relativo all'interno della pagina. Se le pagine fossero lunghe 2048 byte, la MMU utilizzerebbe i primi 5 bit per il numero di pagina, ed i successivi 11 bit per l'indirizzo relativo. Ne consegue che quando le dimensioni della pagina sono minori, con il paging è possibile indirizzare un numero maggiore di pagine

La soluzione risiede nel *copiare blocchi di elementi diversi con dimensioni diverse*: ciò permette di risolvere sia il problema della memoria inutilizzata sia quello delle cancellazioni superflue (ovvero se ad esempio in un blocco ho poca informazione utile quel blocco rischia di essere usato poco ed essere eliminato dalla MC). Guardando inizio e fine dell'elemento di interesse posso decidere la dimensione del blocco da dedicargli.

SVANTAGGI PAGINAZIONE: lo svantaggio maggiore della paginazione è l'obbligo di avere dei blocchi di dimensione prestabilita, il che porta ad avere della memoria inutilizzata. Ciò può portare ad eliminare interi blocchi contenenti informazioni magari importanti poiché poco utilizzati

e.g. MM = 1TB MC = 8GB dimensione pagina = 1KB

* quanti bit per l'indirizzo virtuale? 40 bit, ovvero il numero necessario rappresentare 1TB

* quanti bit per l'indirizzo fisico? 33 bit, ovvero il numero necessario a rappresentare 8GB, ovvero $2^3 \cdot 2^{30}$

* quante pagine virtuali? $2^{40}/2^{10} = 2^{30}$ parole. Ovvero il rapporto tra spazio disponibile e dimensione di ciascuna pagina

* quante pagine fisiche? $2^{33}/2^{10} = 2^{23}$. Ovvero il rapporto tra spazio disponibile e dimensione di ciascuna pagina.

* quale dimensione ha la tabella?

Ha 2^{30} righe, ovvero una per ogni pagina virtuale disponibile

Ha i 4 campi delle tabelle di conversione: presenza/assenza, modifica/non modifica, n bit per l'ultima modifica e 23 bit per l'indirizzo. Il fatto di avere 23 bit e non 33 dipende dal fatto che l'indirizzo effettivo a cui punto è nella memoria fisica (che ha 2^{23} pagine), anche se queste rappresentano la memoria virtuale che ha dimensioni maggiori. Più di quelle della MC non posso allocare, pertanto se dovessi avere MC piena avrei righe della tabella che indicano pagine non allocate.

Non mi interessa dove è stato copiato il dato, l'importante è sapere dove si trova all'interno della MC

EX: se devo prendere in MM la pagine 12343 e il suo indirizzo 127, quando copio nella MC ho

che l'indirizzo del dato che mi interessa diventa il corrispettivo fisico in MC, ex 1235, mentre l'ordine di allocazione interno rimane 127

Bit di differenza: tra la dimensione dell'indirizzo fisico e quella virtuale vi sono tot bit di differenza. Questo avviene poiché l'MMU mantiene i bit di indirizzo interno, modificando quelli di rappresentazione esterno e traducendolo nell'indirizzo di pagina della MC

Esempio concreto: se ho un'enciclopedia con 145663 pagine in plichi da 100, 1456 è

l'indirizzo del plico, 63 è l'indirizzo all'interno del plico. Idem con la CPU e l'MMU:

se ho indirizzo generato di 40 bit, i 10 meno significativi rimangono e sono i bit che indicano l'indirizzo all'interno della pagina di interesse, i 30 precedenti vengono convertiti in 23 e indicano la locazione della pagina nella memoria fisica (ovvero **cambio il numero d'ordine da virtuale a fisico**)



?= se avessi 8 KB di parola invece che 1 nell'esempio precedente?

Cambiarebbe il numero delle righe e delle colonne, ma il tempo di consultazione sarebbe pressoché identico, poiché comunque le MC sono memorie ad accesso diretto che puntano direttamente all'indirizzo di interesse, senza dovervi accedere in modo sequenziale

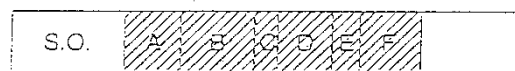
SEGMENTAZIONE

Come evito il problema dello spreco di memoria dovuta alla paginazione? Tramite la segmentazione e la divisione che non è più in blocchi di uguale dimensione ma in unità logiche separate, i *segmenti*. Essi sono moduli di cui è composto un programma, una struttura dati etc etc

Nella segmentazione si identifica l'elemento di interesse e copio soltanto la parte che mi interessa, trasferendo in memoria solo quella; così facendo se anche una porzione dei dati che mi interessa è a cavallo di due o più pagine non mi interessa poiché copio dall'inizio alla fine della parte che mi interessa.

La *conversione* di indirizzo richiede che ad ogni programma venga associata una tabella che per ogni segmento riporti il bit di presenza, l'indirizzo di tale segmento e la **lunghezza** del segmento: se pertanto il segmento non è presente è necessario caricare dalla memoria di massa segmenti di dimensione differente, con la conseguenza di una memoria che viene assegnata dipendentemente dalle richieste di esecuzione

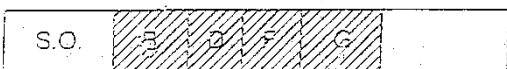
Problema: ogni caricamento va ad occupare lo spazio lasciato libero da un segmento che pertanto deve essere \geq del segmento che lo sostituisce. Questo implica una selettività delle pagine di memoria da eliminare non decisa soltanto da tempo ma anche da necessità di dimensioni. Inoltre dopo tot swap si genera frammentazione nel disco:



SONO PRESENTI I PROGRAMMI A, B, C, D, E, F



SONO TERMINATI I PROCESSI A, C, E
E' STATO CARICATO IL PROGRAMMA G



E' STATO ESEGUITO UN COMPATTAMENTO

Si ha pertanto una memoria piena di 'buchi' residui dei vari caricamenti, che singolarmente sono piccoli ma sommati creano una buona porzione di disco sprecata. Si potrebbe *compattare periodicamente* la memoria, ma è una soluzione temporanea e inefficiente in quanto richiede molto tempo e la copiatura di TB interi di dati

SEGMENTAZIONE PAGINATA: la soluzione a questo problema è la segmentazione paginata, che unisce i vantaggi di paginazione e segmentazione. In questa tecnica di gestione della memoria la CPU genera un indirizzo virtuale composto dal numero di segmento virtuale, il numero della pagina e l'offset. L'MMU preleva il numero del segmento, accede in tabella e nel caso sia assente provvede a scaricarla da MM a MC.

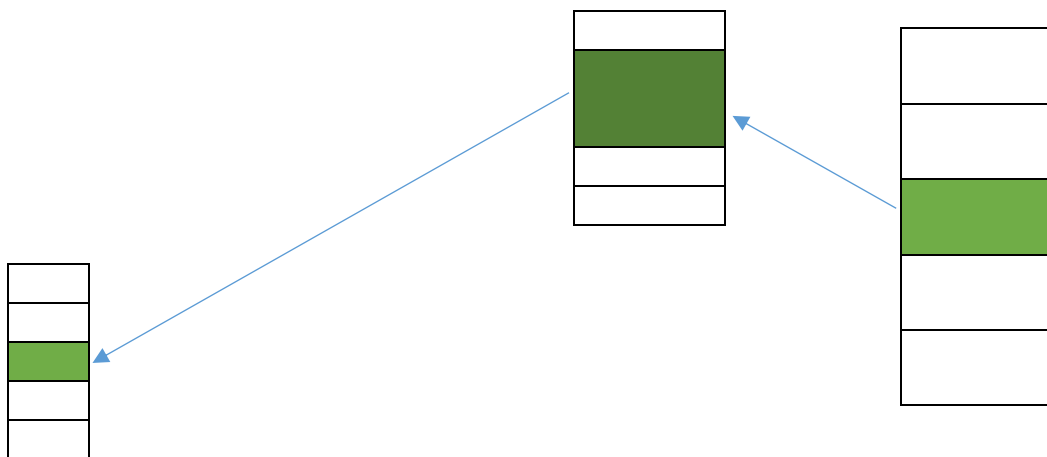
La divisione della memoria virtuale è ancora realizzata a segmenti di memoria di dimensioni variabili, che sono però suddivisi in pagine di lunghezza fissa senza vincolo di consecutività. La *conversione* di indirizzi richiede due livelli di tabella, uno per identificare la posizione del segmento, l'altra per identificare la pagina all'interno del segmento. Infatti gli **spazi vuoti** lasciati dalla frammentazione e dalle **celle occupate** vengono considerati **logicamente** come unici, per quanto **fisicamente** non lo siano.

L'unico svantaggio di questa tecnica è la gestione della memoria, che tuttavia viene ripagata da un'efficienza e uno sfruttamento a pieno delle possibilità della memoria stessa



GERARCHIA DI MEMORIA

La gestione virtuale della memoria permette di gestire una *gerarchia della memoria* con memorie molto veloci (statiche) per l'utilizzo immediato di programmi, memorie rapide ma non rapidissime (semicond. dinamiche) e memorie dalle grandi capacità ma non molto rapide (supporti magnetici)



Le informazioni a disposizione vengono memorizzate nel supporto più lento ma più capiente, una copia di tali informazioni di uso quotidiano è salvata all'interno delle memorie ad accesso più rapido, ovvero la memoria a semiconduttore lenta e di quest'ultima parte le informazioni di uso corrente sono a loro volta riportate anche nell'ultimo grado della gerarchia: la memoria a semiconduttore veloce, che è anche la più piccola

I problemi di gestione dei primi due livelli della gerarchia sono piuttosto semplici a causa della grande differenza di lettura/scrittura dei dispositivi, ma negli ultimi livelli, in cui la differenza è di poche volte più veloce, è necessario utilizzare una tecnica che esegua *la ricerca in contemporanea su entrambe le memorie*; così facendo se ci fosse riscontro positivo nella più veloce avrei un effettivo guadagno delle prestazioni. Altrimenti fare una ricerca alla volta annullerebbe il vantaggio

MEMORIA CACHE (Nascosta)

I due tipi di memoria a semiconduttore sono divisi in blocchi come le memorie centrale e di massa. Sfruttano anche queste il principio di località degli accessi per decidere cosa e dove copiare. Per velocizzare effettivamente le operazioni la ricerca inizia in parallelo in ogni

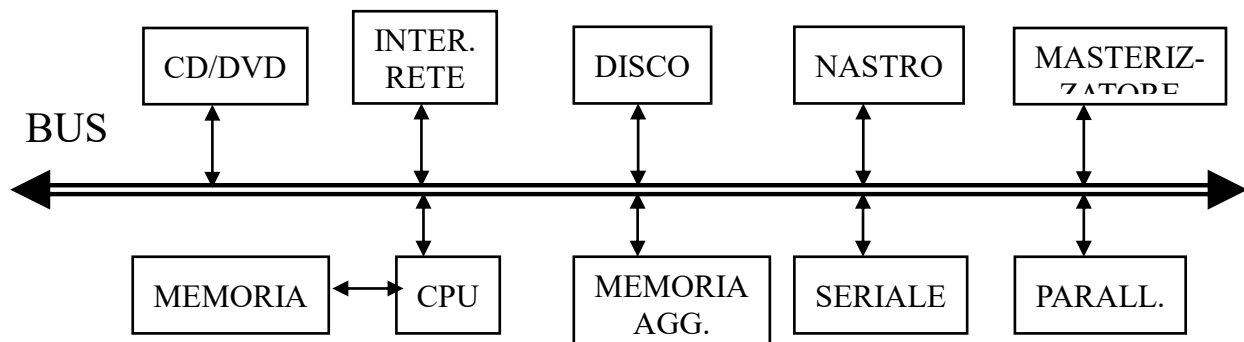
locazione di memoria, raggiungendo comunque la cella desiderata nel minor tempo possibile di accesso alla memoria, senza rallentamenti dovuto al numero di ricerche

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. Tale circuiteria in contemporanea si occupa di portare nella memoria più veloce le informazioni che presumibilmente saranno necessarie in un prossimo futuro alla CPU stessa, effettuando una sorta di trasferimento a pacchetti invece che a parole singole

EVOLUZIONE ARCHITETTURE

Con l'evoluzione tecnologica si è evoluto anche la forma della macchina di VN e del modo di interfacciarsi dei dispositivi tra loro e con l'esterno. I *bus* ad esempio sono stati unificati in un unico bus che permetta comunicazione tra tutte le periferiche e che garantisca accesso a tutte ai dati. In quest'architettura la CPU è la sola a parlare con la memoria, si interfaccia con il resto delle periferiche e queste ultime comunicano tra di loro.

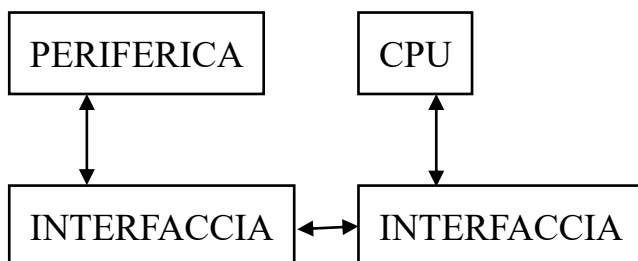
Ogni periferica è vista dalla CPU come un elemento di memoria a indirizzo prefissato



Problema: > le periferiche di I/O possono essere lontane dalla CPU e in generale dal bus dei dati? No, poiché un'eccessiva distanza coperta da bus implicherebbe un'interferenza elevatissima a causa degli effetti elettromagnetici dovuti al passaggio di corrente nei bus, con conseguente disturbo reciproco dei bus e perdita di informazione

> c'è inoltre il problema dell'interfacciare un elemento sincrono come la CPU con la realtà del mondo esterno, tipicamente asincrona

Per risolvere questi problemi si introduce la cosiddetta interfaccia di comunicazione, in cui si frappone tra CPU e periferica un'interfaccia che si occupi di compensare i problemi di sincronismo tra gli elementi e di gestire le comunicazioni tra periferiche e calcolatore

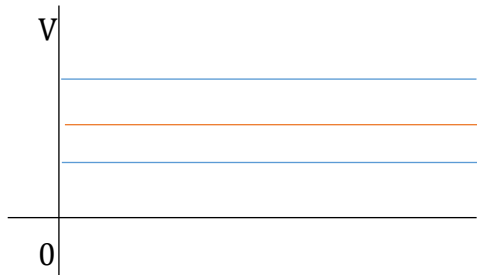


Esistono due tipi di interfacce:

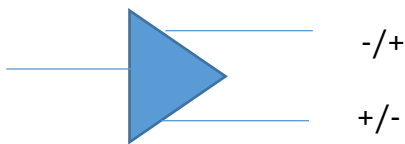
- Parallela: in cui vengono utilizzati cavi paralleli per la trasmissione di più bit per volta, implicando l'utilizzo di cablature relativamente ridotte per non introdurre interferenze. E' inoltre più veloce di base rispetto ad una seriale anche se più costosa a causa delle schermature necessarie sui vari fili che la compongono (std di questo tipo sono IEEE 488 o CENTRONICS)

- Seriale: Sono interfacce tendenzialmente più lente che però utilizzano pochi fili e permettono la comunicazione a notevole distanza; sostanzialmente serializza tutta l'informazione e invia un bit per volta alternati e sulla stessa linea.
(std di questo tipo sono CurrentLoop, RS232, USB)

Come riduco i disturbi nelle interfacce? Introducendo una banda limite e identificando gli elementi al di sopra come 1 e al di sotto come zero. Più è ampia la banda e largo il DeltaV più ammortizzo le interferenze



OBS: nel caso di disturbi additivi? La soluzione è avere un'uscita doppia in cui l'identificazione di 1 e 0 non avviene più secondo il valore effettivo della tensione ma dipendentemente da quale elemento è più alto in tensione e quale più basso.



EVOLUZIONE VON NEUMANN

L'evoluzione tecnologica ha portato ad un miglioramento delle prestazioni dei calcolatori ma è stato necessario risolvere un problema intrinseco della macchina di VN. Nell'architettura fondamentale le operazioni vengono eseguite necessariamente con sequenzialità, pertanto si deve trovare un modo di far fruttare al meglio l'hw a disposizione

Cosa succede durante le varie fasi? Che vengono utilizzate, a seconda della fase, diverse risorse hw e con diversi compiti

Processo	<i>Fetch</i>	<i>Decode</i>	<i>Execute</i>
Risorse	PC IR	Dec	Alu Acc Flag

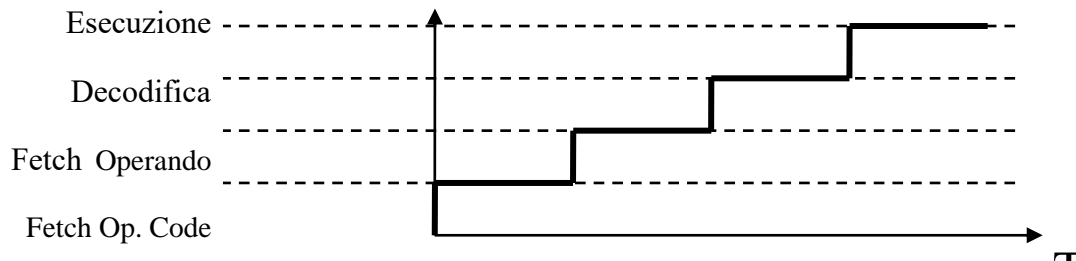
Si nota come due unità funzionalmente autonome, la CPU e un dispositivo di I/O, sono costrette a sequenzializzare le proprie attività impiegando un tempo pari alla somma dei singoli tempi di lavoro. Similmente con accessi a memoria centrale: rigorosa sequenzialità di accesso senza prevedere alcun meccanismo per anticipare le richieste della CPU e per effettuare accessi a memoria in parallelo con le operazioni della CPU stessa. O anche con programmi di calcolo, che una volta prelevato il dato per la ALU fermano la memoria o le altre componenti hw con conseguente spreco di tempo

OBS: nella fase di fetch, ad esempio, decoder o ALU non sono impegnate; pertanto si può pensare di utilizzare le fasi di un programma per eseguirne altre su un secondo programma, andando così a guadagnare tempo nell'esecuzione complessiva

Con le istruzioni standardizzate posso vedere l'esecuzione di un programma o una richiesta come la catena di montaggio di una fabbrica in cui ogni operaio è addetto ad una piccola parte

della produzione completa (ogni risorsa esegue solo una fase dell'istruzione) e finito un pezzo lavora sul successivo (così mentre un programma è in fase di execute PR e IR stanno eseguendo la fase di fetch di un altro programma, dimezzando i tempi di esecuzione del complesso)

Esecuzione sequenziale semplice



Esecuzione sequenziale secondo pipeline

	<i>i-3</i>	<i>i-2</i>	<i>i-1</i>	<i>i</i>
Esecuzione				
Decodifica	<i>i-2</i>	<i>i-1</i>	<i>i</i>	<i>i+1</i>
Fetch Operando	<i>i-1</i>	<i>i</i>	<i>i+1</i>	<i>i+2</i>
Fetch Op. Code	<i>i</i>	<i>i+1</i>	<i>i+2</i>	<i>i+3</i>

PIPELINE:

Una CPU con pipeline è composta da cinque stadi specializzati, capaci di eseguire ciascuno una operazione elementare di quelle sopra descritte. La CPU lavora come in una catena di montaggio cioè ad ogni stadio provvede a svolgere in maniera sequenziale un solo compito specifico per l'elaborazione di una certa istruzione. Quando la catena è a regime, ad ogni ciclo di clock dall'ultimo stadio esce un'istruzione completata. Nello stesso istante ogni unità sta però elaborando in parallelo i diversi stadi di successive altre istruzioni. In sostanza quindi si guadagna una maggior velocità di esecuzione a prezzo di una maggior complessità circuitale del microprocessore, che non deve essere più composto da una sola unità, ma da cinque unità che devono collaborare tra loro.

Si ottiene così la possibilità di lavorare in parallelo su più istruzioni alla volta, andando a compensare l'intrinseco limite di VON Neumann

Svantaggi: sono necessarie delle condizioni particolari affinché la pipeline sia attuabile:

- Non posso avere istruzioni che richiedono le stesse risorse in contemporanea, ma devono essere utilizzate risorse in modo mutualmente esclusivo (altrimenti torno alla sequenzialità e perdo il vantaggio effettivo della pipeline)
- Le istruzioni devono avere le stesse fasi che le compongono, altrimenti la pipeline, benché funzionante, non porterebbe alcun vantaggio poiché avrei casella della pipeline vuota e risorsa hw non utilizzata
- Gestione dei salti: se infatti ho un salto mi ritrovo, dopo il return del jump, a dover ripetere le operazioni delle istruzioni al di sotto della diagonale dell'istruzione che ha chiamato il salto. La pipeline continua a funzionare, ma ho perdita del vantaggio da essa derivante

La pipeline può essere utilizzata anche per altri motivi, come ad esempio l'esecuzione di calcoli multipli simultaneamente (e.g. calcolo del discriminante di una equazione, in cui si divide il calcolo intero in sottounità di calcolo)

MODALITA' DI ACCESSO ALLE PERIFERICHE

Esistono due tipologie di accesso alle periferiche, dato che la CPU non può sapere quando una periferica avrà bisogno di accedere alla memoria o a qualunque parte hw del calcolatore:

- **POLLING:** è la modalità meno efficiente ma più semplice e di semplice realizzazione delle due; prevede un'interrogazione continua delle periferiche affinché, quando si renda necessario un processo da queste richiesto, la CPU sia pronta. E' tuttavia dispendioso poiché se ho molte periferiche e molte sono inutilizzate per molto tempo diventa uno spreco di risorse della CPU e di tempo (potrebbe essere utile in contesti semplici, e.g. l'abs dell'automobile per controllare se il guidatore sta frenando, attraverso una continua interrogazione e rilevamento della frenata)
- **INTERRUPT:** è il metodo più efficiente di interfaccia tra periferiche e CPU, benchè sia più complicato nella realizzazione. Prevede che siano le periferiche a richiedere alla CPU l'attenzione necessaria per l'esecuzione dell'operazione

Problema: come gestisco, da interrupt, una possibile richiesta poco frequente ma molto fitta quando presenta (i.e. scrittura disco da parte di una periferica)?

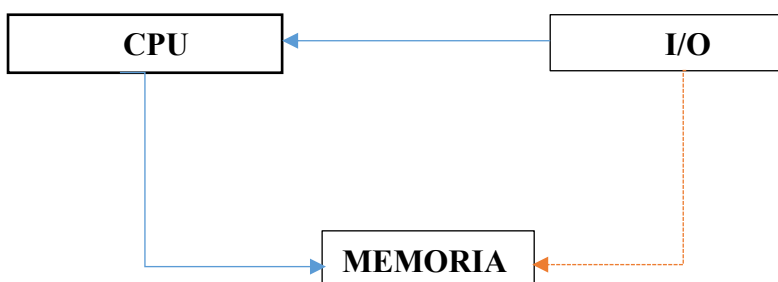
Se i dati sono pochi il tutto può essere gestito dalla CPU che riceve i dati dalle periferiche, ad esempio la misurazione della temperatura dell'acqua da parte di una lavatrice, e la salva nei registri. Ma se i registri non sono sufficienti alla memorizzazione dei dati ed è necessario scrivere in memoria?

Devo introdurre la DMA (Direct Memory Access), divisa in due fasi:

- DMA REQ: richiesta di DMA da parte della periferica alla CPU
- DMA ACK: accettazione della DMA da parte della CPU ed esecuzione della r/w da parte della periferica.

Con questa tecnica le periferiche hanno accesso direttamente alla memoria in cui ipoteticamente dovrebbe scrivere la CPU. In questo modo la CPU non lavora, non viene continuamente interrogata e non vi è spreco temporale. Inoltre la CPU ha la possibilità di lavorare su dati eventualmente presenti in cache: se non sono presenti dati in cache non esegue compiti e al limite il guadagno è della scrittura in memoria, se invece sono presenti dati in cache posso eseguire operazioni su quei dati con ulteriore guadagno rispetto alla sola miglioria di r/w del DMA

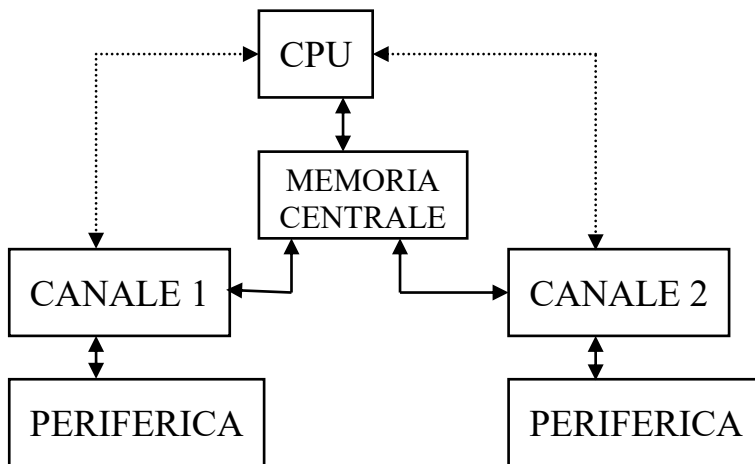
Quanto guadagno tempisticamente? Senza la DMA per ogni dato avevo due trasferimenti, uno per la CPU e uno effettivamente per la scrittura/lettura. Ora invece l'interrupt è uno solo e pertanto i tempi di esecuzione effettiva dimezzano.



Problema: se dovessi avere comunicazioni criptate o trasferimenti satellitari in cui è necessario aggiungere ridondanza come faccio, una volta ricevuto il dato, a eliminare la ridondanza o crittografia dei dati?

Aggiungo un'entità che si occupi di eseguire queste operazioni in modo che quando la periferica di I/O deve accedere alla memoria i dati sono già pronti all'uso per la CPU

Canale di input/output



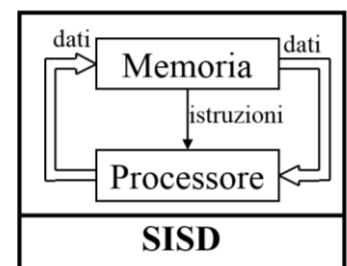
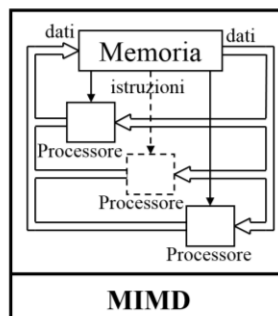
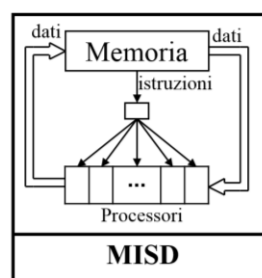
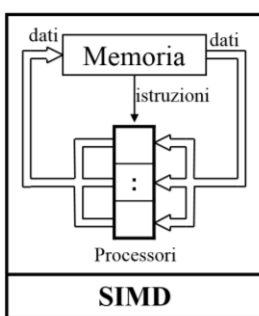
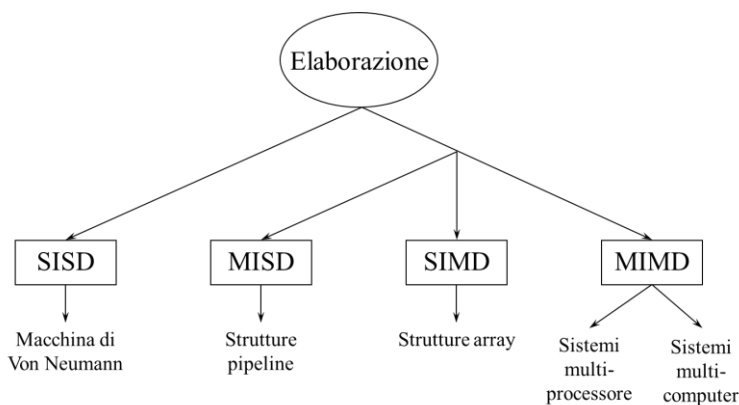
Questa velocizzazione dei processi tramite pipeline, scrittura diretta in memoria e canale di I/O viene meno con esecuzione di calcoli ripetuti ma su dati diversi. Se infatti devo eseguire un calcolo matriciale, in cui *devo eseguire calcoli identici su dati differenti*, ho perdita di tempo dovuta ad acquisizione ed interpretazione delle stesse istruzioni a causa del fatto che la ALU opera soltanto su coppie di valori

?=come fanno le GPU ad eseguire calcoli così rapidi sulle immagini: tale processo avviene poiché invece di avere un'istruzione applicata ad un solo dato hanno la stessa istruzione applicata a molteplici dati in cui la fase di fetch e di decodifica rimane unica, mentre la fase di execute viene suddivisa sulle innumerevoli ALU che compongono una scheda grafica moderna. Avviene pertanto un parallelismo tra i dati; (che si affianca al parallelismo tra le istruzioni) Vengono così a definirsi, dipendentemente dalla presenza o meno di uno dei due parallelismi definiti precedentemente, diverse architetture di gestione dei parallelismi:

- SISD: tipico della macchina di VN, nell'architettura SISD è possibile eseguire un'unica istruzione per volta su un unico dato, eseguendo le istruzioni una dopo l'altra. Si genera così un forte collo di bottiglia dovuto al fatto che ogni componente viene utilizzata soltanto poco per volta, mai in contemporanea ad altre per istruzioni differenti. E' un problema che si può parzialmente risolvere con il pipelining, che tuttavia viene ridotta dalle istruzioni di jump e dalla non predittività delle istruzioni
- SIMD: tipico delle GPU, vi sono centinaia/migliaia di ALU che si dividono i dati da processare. Concettualmente c'è l'elaborazione contemporanea dei dati; Il modello SIMD è composto da un'unica unità di controllo che esegue una istruzione alla volta controllando più ALU che operano in maniera sincrona. Ad ogni passo, tutti gli elementi eseguono la stessa istruzione scalare, ma ciascuno su un dato differente. Un elaboratore basato su questo modello è anche detto *Array Processor*. Gli svantaggi delle SIMD sono dovute all'aumento esponenziale di registri e bus dovuti all'aumento di ALU. Questo porta a costi progettuali notevolmente più elevati di una architettura SISD; inoltre il guadagno dell'architettura SISD viene meno quando l'istruzione da eseguire non è la stessa per tutti i pixel: in questo caso ogni pixel necessita di processo a se stante, con collo di bottiglia

- MIMD:** Nel modello di calcolo MIMD ci sono più processori che operano in parallelo in modo asincrono. Ciascun processore esegue un programma (o istruzione) diverso su dati diversi. In un singolo (ma grande) problema, ogni processore risolve un sotto-problema. La comunicazione tra i vari processori avviene sia mediante la memoria condivisa (si parla di processori "tightly coupled" altamente legati), sia per mezzo di una rete di interconnessione (multi computer debolmente legati).
 Prevedono la replicazione dell'intera struttura della macchina di Von Neumann per ottenere architetture multiprocessore
 Vi è quindi sia un parallelismo di dati che di istruzioni
 Anche in questo modo sorge il problema di gestire connessioni di bus e tra ALU molteplici, con conseguente aumento dei costi di progettazione e delle dimensioni occupate
- MISD:** Nel modello di calcolo MISD esistono più processori, ognuno con una propria memoria (registri), la quale a sua volta avrà un proprio flusso di istruzioni. Queste istruzioni verranno eseguite sullo stesso flusso di dati. Un esempio di un possibile utilizzo è quello nell'ambito della crittografia. Ad oggi non sono ancora state costruite macchine da commercializzare con questa architettura.
 Ovvero un dato a cui applico più istruzioni, *struttura pipeline* (NB != pipeline istruzioni)

Riassumendo:

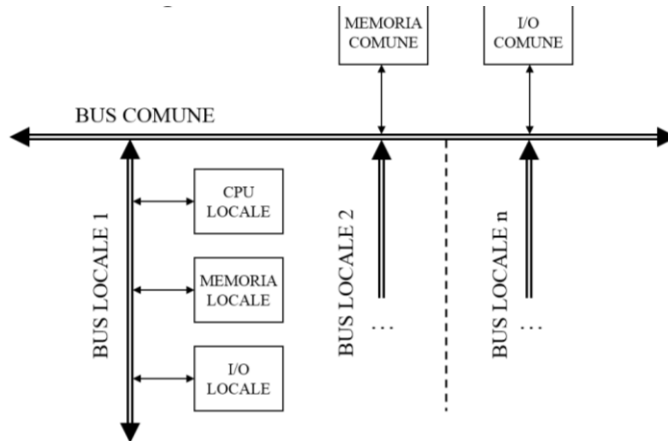


GESTIONE DEL MULTI-ALU:

come gestisco la presenza di più ALU sullo stesso elemento hw, con conseguente aumento dei canali di comunicazione, dei bus di ciascuna ALU e dei registri?

Ho due tecniche:

- La prima (**sistema multiprocessore**) consiste nel dirigere tutte le comunicazioni su un unico Bus comune condiviso tra gli interlocutori a cui si somma solo il bus e i registri personali delle varie componenti. Ha il grosso difetto che deve essere comandato da qualche componente affinché non si verifichino interferenze



- La seconda consiste nella gestione simil-cubo dei bus con il maggior numero di collegamenti senza però avere necessità di connettere ogni elemento con ogni altro. Immaginando un cubo si connette ogni spigolo lasciando libere le connessioni diagonali. In questo modo ogni comunicazione tra nodi non adiacenti implica il passare attraverso nodi intermedi.
 2^N processori ai vertici di un N-cubo. Sono necessari al più N passaggi per portare da un processore all'altro

SISTEMI OPERATIVI

Insieme di programmi che rendono facilmente disponibile all'utente le potenzialità della macchina in modo trasparente e rendere disponibile tale interazione indipendente da hw che c'è dietro e dalle conoscenze dell'utente. Si occupa della gestione delle risorse e del loro utilizzo

Fondamentalmente è composto da due parti: → Kernel → sw di base

Il **kernel** del sistema operativo va in esecuzione all'accensione della macchina (**bootstrap ***) e svolge le seguenti funzioni:

- Controllo dell'esecuzione dei programmi dell'utente
- Amministratore delle risorse hw, soprattutto nei sistemi multiutenti

* viene innanzitutto eseguita una diagnostica del sistema e delle periferiche, per assicurarsi che non ci siano errori o mancanze fondamentali; dopodiché vengono caricati in memoria centrale i programmi fondamentali per l'esecuzione del s.o., kernel in primis

Overhead: tempo sottratto all'utilizzatore per operazioni interne, eg bootstrap o check diagnostico, che viene sottratto all'utente

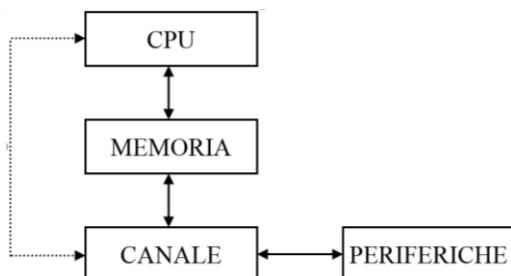
L'altra parte del s.o. è il **software di base**, che si occupa di tutte quelle operazioni che vengono eseguite direttamente dall'utente come compilare un file, cancellarlo, editarlo...



Questa completa trasparenza del s.o. implica che le conoscenze necessarie all'utilizzo del sw di base siano limitate al sistema operativo, senza necessità di sapere cosa sia o come agisca il kernel. Inoltre la virtualizzazione a cui si ricorre per eseguire sw e kernel porta ad avere hw molto diversi che supportano e permettono l'esecuzione di numerosi s.o. differenti

Tipologie di s.o.:

- **DEDICATI**: sono stati i primi sistemi operativi, tornati in uso con i primi personal computer. La macchina viene utilizzata da un utente per volta che può eseguire soltanto un programma per volta; il più grosso svantaggio dei sistemi dedicati è la perdita di tempo durante la fase di *block* del processo, con risorse non sfruttate al meglio e tempi morti per la ALU. Hanno tendenzialmente un kernel molto semplice
- **A LOTTI(BATCH)**: vengono accorpati insieme di lavori in un unico lotto (*batch*) e trasferito poi sull'unità di ingresso veloce (*disco*)
I lavori sono impostati in modo che non sia necessario in alcun caso l'intervento dell'uomo. Tutti i parametri sono predefiniti e specificati attraverso appositi script, comandi o file di controllo (i file batch) che si occupano di controllare che la procedura avvenga come previsto. Una tale modalità operativa si oppone al funzionamento in linea (*inline*) dei programmi, che richiede l'interazione diretta e continua tra una macchina e un operatore
Hanno il difetto di lasciare dei tempi morti alla CPU tra un'esecuzione e l'altra e pertanto di non utilizzare al meglio la CPU
- **MULTIPROGRAMMAZIONE**: supponendo di avere più programmi, li carico tutti in memoria e arrivati ad un istante in cui un programma ha necessità di andare a prendere dei dati dall'esterno della unità di elaborazione lo metto in stato di *block*, iniziando l'esecuzione di un altro programma che nel frattempo ha recuperato i dati necessari alla sua esecuzione. Al termine di questo riprendo il precedente interrotto, che nel frattempo ha recuperato i dati necessari pre interruzione e così via...



Il più grosso svantaggio del multiprogramma è che non è equo, ovvero non ripartisce le chiamate e l'utilizzo della CPU dipendentemente dal task ma soltanto dal dover prendere o meno dati dalla memoria. Questo nel caso di programmi con frequente ricorso a I/O, ad esempio, porta ad avere un programma interrotto moltissime volte; inoltre se viene interrotto da un programma che occupa notevolmente la CPU per il calcolo può generare un tempo di *ready* per il primo eccessivamente lungo in rapporto al tempo effettivo di esecuzione del

programma qualora fosse stata gerarchizzata la gestione dei processi

Si dice che il multiprogrammato non è in grado di distinguere tra i processi I/O

bounded e quelli *CPU bounded*: ovvero non organizza a dovere le risorse al fine di evitare che uno della seconda classe rubi tempo ad uno della prima mentre questi è in

ready da diversi cicli di clock.

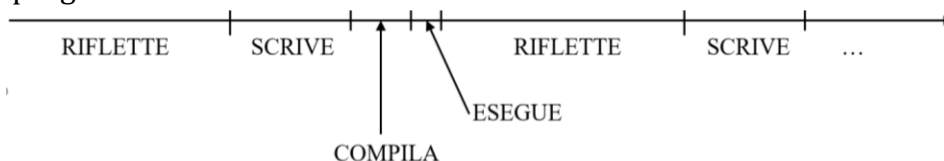
(come si vede dall'immagine del cfr tra le tipologie di s.o, il multiprogrammato genera delle attese molto lunghe per programmi in ready qualora l'esecuzione di altri programmi da parte della CPU richieda tempi lunghi, come può essere un calcolo complesso)

- **TIME-SHARING:** è una tecnica in cui l'esecuzione o il clock della CPU vengono divisi in intervalli temporali (*quanti*) grazie ai quali ogni programma ha un tempo massimo di esecuzione assegnato, uguale per tutti i programmi. Così facendo ogni programma ha a disposizione intervalli equivalenti di risorse e non vi è il problema che si presentava con il multiprogrammato della CPU occupata per troppo tempo da programmi richiedenti una grande mole di calcolo a scapito di quelli che necessitavano di un forte cfr con le periferiche I/O.

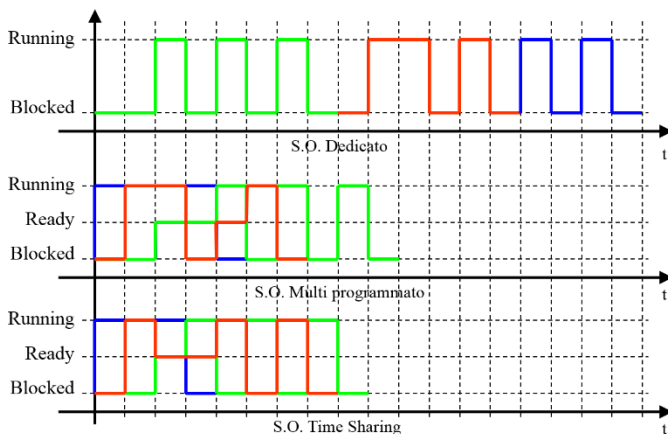
Così facendo ho che *anche se un programma non ha completato le sue operazioni viene comunque interrotto* e il suo p.c. viene salvato insieme a tutti gli altri dati relativi alla posizione momentanea dell'esecuzione (p.c, registri, flag...), suddividendo la CPU e il suo utilizzo in modo equo per tutti i programmi.

(particolarmente utilizzato nei sistemi multiutente, in cui ad ogni utente è assegnato un tempo specifico di utilizzo della CPU principale)

Interrupt: non sono un problema nel timesharing in quanto tutto il timing della gestione è affidato ad un **temporizzatore**, il quale si occupa di gestire le priorità e quale programma deve essere eseguito, quale fermato etc. e un evento eccezionale che genera un interrupt non fa altro che "dire" al temporizzatore di bloccare tutti i programmi che non servono. *



Problema: il passaggio da un processo all'altro è istantaneo? NO. Infatti è necessario un tempo affinché avvenga il **salvataggio del contesto**, ovvero quel tempo necessario affinché il programma che viene mandato in stop/ready non venga perso. Devono essere salvati il p.c, i registri, i flag eventuali...e questo porta ad avere, ogni X ms un tempo t corrispondente a questo salvataggio (o nel caso di ripresa del processo interrotto di ripresa dei dati salvati pre-cambio) → e' quindi necessario un tempo di **overhead** (ovvero quel tempo "rubato" all'utente e in cui non eseguo nessuna operazione)



- **REAL TIME:** se ad esempio ho un impianto d'allarma non posso permettermi questi ritardi. Vengono pertanto utilizzati sistemi operativi real time. Sono sistemi per una specifica operazione che vengono chiamati così in virtù dei vincoli dei tempi di risposta

che si prefiggono (ovvero un tempo massimo entro il quale mandare in esecuzione un programma a seguito di una richiesta in tal senso)

Per definizione di ha un sistema real time *quando il tempo che passa dalla richiesta di esecuzione di un processo al completamento della stessa è minore del tempo fissato*

SISTEMI SU PIU' CPU: come distribuisco il sistema operativo quando ho più CPU? Ho due possibilità: *

- *Una cpu principale:* il s.o. gira su una CPU master e altre CPU che si dividono le risorse e le periferiche
- *S.o. diviso su più CPU:* il s.o. viene eseguito su più CPU

Mentre il primo permette una miglior gestione delle risorse e dei processi, il secondo è molto utile qualora ci sia necessità di avere un calcolatore sempre attivo che permetta modifiche in tempo reale della sua architettura, ad esempio sostituzione delle memorie o di un componente hw

PROGRAMMI

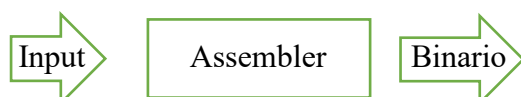
Quando vengono scritti dei programmi essi vengono digitati dall'utente programmatore in linguaggio naturale/codice; affinché siano eseguiti è necessario che essi siano convertiti in linguaggio binario, ovvero nella 'lingua' della CPU.

Affinchè questo sia possibile si ricorre a vari elementi che permettono questa conversione, con approcci differenti dipendentemente da cosa si vuole ottenere e dal linguaggio utilizzato (differenza *tra compilati e interpretati*)

- Assemblatore: è come fosse una scatola nera che converte il linguaggio dell'utente (assembly, linguaggio di basso livello) in informazioni di tipo binario per il calcolatore. Ovvero passo da linguaggio naturale/simboli a binario.

Il nome deriva dal fatto che le istruzioni vengono convertite e montate una accanto all'altra come fossero in fila

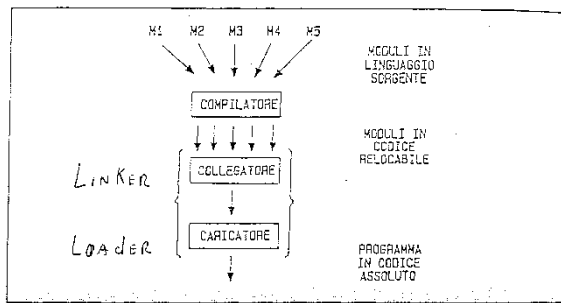
Esso funziona con una logica di corrispondenza tra ciò che viene digitato in input e il corrispondente risultato in output binario per la CPU, ovvero ha una corrispondenza 1:1 dei comandi (**statement**) con le istruzioni corrispondenti



Il compito degli assembler avviene solitamente in due fasi: in una prima fase viene effettuata l'analisi delle istruzioni con controllo del corrispettivo nelle tabelle di corrispondenza, una seconda in cui viene generato effettivamente il codice in linguaggio macchina per il calcolatore.

Questo approccio con un unico linguaggio eseguito in un unico momento porta alla necessità, per programmi particolarmente lunghi, di avere la possibilità di dividere l'intero programma in blocchi più piccoli con funzioni specifiche; questo mi permette, in caso di errore del programma, di restringere subito la ricerca dell'errore al blocco che lo ha generato, generando anche una maggiore intercambiabilità in caso di modifica ad una delle componenti (non devo riscrivere l'intero codice, ma solo la parte relativa a quel componente)

- Linker: è un programma incaricato di *fondere in un unico modulo eseguibile oggetti separatamente compilati*, permettendo l'utilizzo di librerie preconfezionate.

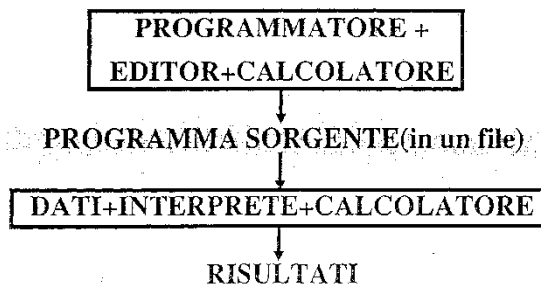


Grazie al linker è possibile anche che più persone lavorino allo stesso progetto, dividendosi i compiti, e unendo grazie ad esso ciascuna parte di codice con le altre.

Consente anche l'unione di moduli creati con linguaggi differenti (e.g. la gestione degli interrupt in C può essere scritta in assembly)

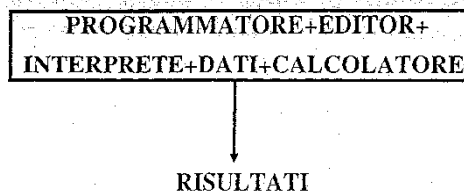
- **Interprete:** ci sono dei programmi che permettono di essere eseguiti direttamente da un programma detto *interprete*. È un approccio differente rispetto alla compilazione, e lo si nota analizzandone l'approccio al codice scritto dall'utente; supponendo un parallelismo tra compilati e interpretati rispetto alla traduzione di una frase dall'inglese all'italiano:
 - **Interpretato:** traduco ogni singola parola una alla volta, con un risultato che prescinde dal linguaggio e dalle forme fatte o dal significato dipendente dal contesto. Alterna la lettura di un comando (1), la sua interpretazione (2) e l'esecuzione (3). Non ragiona a blocchi ma a parole
 - **Compilato:** legge prima i blocchi semantici e poi traduce dipendentemente dal contesto. Esegue anche un'analisi lessicale, sintattica e semantica (per garantire congruenza, esistenza e logica nei comandi). Genera inoltre un albero più una tavola dei simboli per l'esecuzione ordinata dei comandi

Interpretazione:

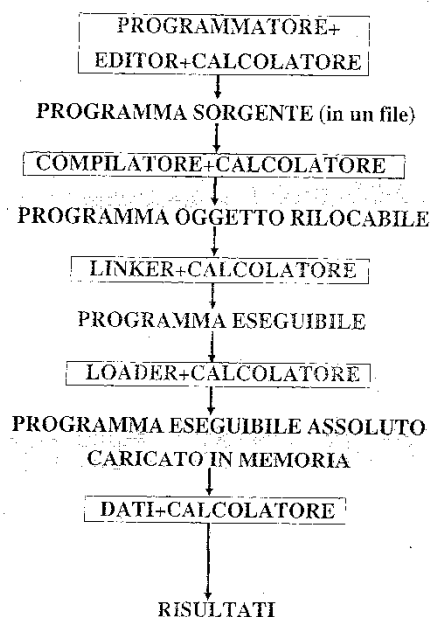


o anche:

Interpretazione: Ambiente di Sviluppo unico



Compilazione



Questo implica che il linguaggio interpretato abbia una velocità esecutiva maggiore del compilato, che tuttavia permette una maggiore versatilità dovuta all'analisi lessicale dei costrutti del codice. Il linguaggio compilato ha una corrispondenza 1:N tra gli statement e le istruzioni

- **Editor:** sono programmi che permettono di interagire con il terminale, creare e modificare testi (file in codifica ASCII)

LINGUAGGI:

- *Linguaggio macchina (1° generazione):* sono i primi linguaggi nati, utilizzavano direttamente il linguaggio macchina per programmare i calcolatori (alle origini il binario). Le istruzioni venivano eseguite dalla CPU senza necessità di nessun intermediario, con lo svantaggio che ogni CPU aveva il proprio linguaggio macchina, pertanto non era trasportabile come linguaggio.

La programmazione era notevolmente difficoltosa e la leggibilità del codice praticamente nulla (con conseguente facilità di errori)

Il grosso vantaggio di un linguaggio di questo tipo era l'immediatezza esecutiva e l'alta efficienza rispetto ai linguaggi compilati/programmati delle generazioni successive.

- *Linguaggi assemblativi (2° generazione):* sono stati la naturale evoluzione del linguaggio macchina; a partire da una tabella di corrispondenza a istruzioni in linguaggio più simile al naturale e più intuitive si fa corrispondere un'istruzione per la CPU. Il compito di traduzione è affidato all'assemblatore. Aggiunge la possibilità di definire e utilizzare macrofunzioni, ma mantiene ancora una notevole difficoltà per quanto riguarda la leggibilità.

E' poco portatile e vincolante; tuttavia non vincola al singolo processore, ma alla famiglia affine di processori, come possono essere i MIPS

Anche in questo caso la programmazione e la leggibilità sono difficoltose e scarse, ma l'efficienza è molto elevata.

- *Linguaggi di alto livello (3° generazione):* le istruzioni sono espresse in un linguaggio significativo al programmatore e distanti dal linguaggio macchina. Questo comporta la necessità di avere un compilatore o un interprete, ma anche il vantaggio di non dipendere dal calcolatore: basta infatti cambiare compilatore per avere portabilità del linguaggio.

La presenza di compilatore e interprete riduce drasticamente l'efficienza del linguaggio di alto livello rispetto agli assemblativi o al linguaggio macchina, ma ne facilita notevolmente la leggibilità e l'apprendimento

La scelta del linguaggio avviene dipendentemente da più parametri, quali la portabilità ricercata, l'efficienza necessaria, le conoscenze pregresse del programmatore, l'architettura che si sta utilizzando....

- *Linguaggi procedurali:* è la categoria dei linguaggi di alto livello che si differenzia in quanto basata sul modello computazionale di Von Neumann. Il programma è visto come sequenza di istruzioni che tende a modificare il contenuto della memoria. I linguaggi procedurali sono indipendenti dalla macchina, essendo di alto livello, compatibili con l'utilizzo di simboli e nomi per i dati, con una sintassi vicina a quella naturale; essi sono basati su istruzioni e assegnazioni e permettono l'utilizzo dei dati definendo tipo di dati e operazioni sui tipi. Permettono inoltre l'iterazione e i cicli sulle strutture di controllo e permettono l'utilizzo di sottoprogrammi; sono caratterizzati da una buona leggibilità.

Le categorie in cui si dividono sono:

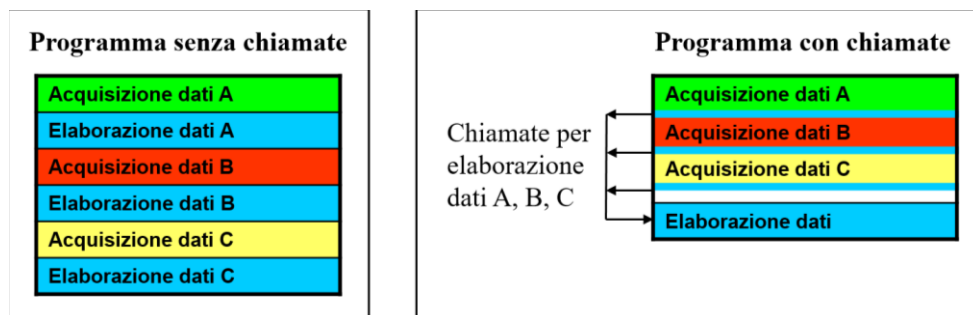
- Uso generale
- Uso speciale
- Applicazioni web

SOTTOPROGRAMMI E MACROISTRUZIONI

Sottoprogrammi:

è una sequenza di istruzioni che vengono eseguite a seguito di una chiamata, spesso per integrare funzioni esterne al codice principale (e.g. sin, cos etc). L'insieme di più sottoprogrammi compone le librerie.

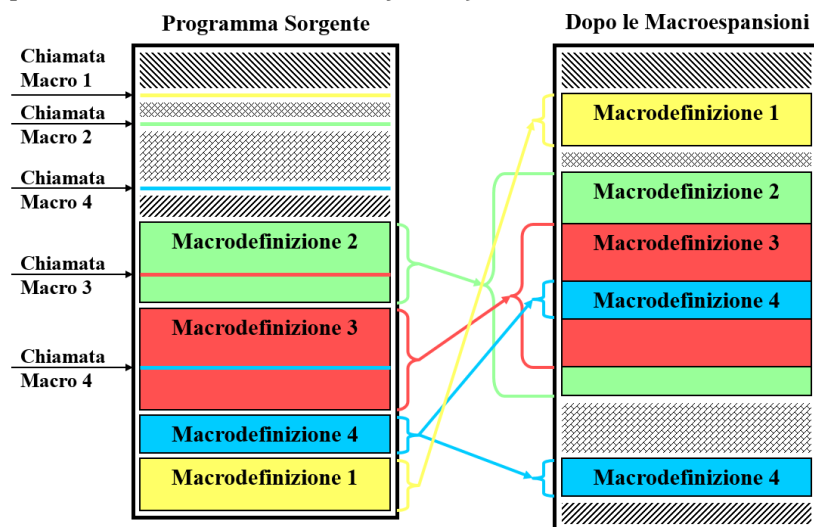
L'avere un sottoprogramma unico consente di richiamare quest'ultimo quando necessario, scrivendo soltanto una volta la parte di codice corrispondente (aumentando notevolmente la leggibilità del codice)



Parametri formali e parametri attuali/effettivi

I primi sono i simboli che rappresentano i dati, su cui il sottoprogramma opera e che vengono specificati nella **definizione** del sottoprogramma; i secondi sono i dati effettivi, corrispondenti ai parametri formali su cui il sottoprogramma effettivamente opera, specificati nella chiamata del sottoprogramma. (e.g. se sin è il sottoprogramma, theta è il nome dell'angolo, parametro formale, 3.14 è il parametro attuale di theta parametro formale)

Oltre al sottoprogramma chiuso (semplice richiamo al sottoprogramma) esiste anche il **sottoprogramma aperto**, in cui ad ogni richiamo del sottoprogramma viene sostituito il sottoprogramma stesso. Il sottoprogramma viene chiamato macrodefinizione, la sua chiamata macrochiamata. Il meccanismo che porta ad avere ad ogni sottoprogramma il corrispettivo 'aperto' è detto macroespansione, che sostituisce i singoli richiami con le istruzioni del sottoprogramma. (qualora una macroistruzione contenga richiami ad altre macroistruzioni si parla di macroistruzioni nidificate)



L'apertura dei sottoprogrammi porta ad avere un'occupazione di memoria maggiore (infatti prima dell'esecuzione ad ogni chiamata si sostituisce l'intero sottoprogramma), tuttavia questo rende sequenziale l'esecuzione ed elimina i salti, rendendo *efficace la pipeline*. Esiste, di contro, il **sottoprogramma chiuso** in cui il sottoprogramma compare una volta sola e, all'interno del codice, sono presenti dei richiami; possono esistere programmi ricorsivi.

ovvero che richiamano sé stessi. Questa soluzione rende il programma meno avaro di memoria, ma lo rende *inadatto alla pipeline* (anticipa operazioni che non vengono eseguite, diventando inutile). E' infatti necessario determinare l'indirizzo di rientro del programma chiamante, passare i parametri di ingresso e di uscita, e gestire le chiamate/sottochiamate (seppur in modo infinitesimale, è più lento un sottoprogramma chiuso rispetto ad un aperto)

GESTIONE CHIAMATE

Quando un sottoprogramma viene chiamato, l'indirizzo di rientro, cioè quello dell'istruzione successiva a quella che ha effettuato la chiamata, viene memorizzato nello STACK (zona di memoria gestita in logica LIFO: pila).

Quando il sottoprogramma termina l'esecuzione, viene letto dalla pila l'ultimo indirizzo caricato che non è stato ancora prelevato e viene trasferito nel Program Counter.

La gestione LIFO dello STACK permette di nidificare le chiamate ai sottoprogrammi.

