Esame Architettura

Introduzione al concetto di algoritmo

Un algoritmo é un insieme di istruzioni codificate in un determinato linguaggio, che va a stabilire il modo in cui dev'essere eseguito un determinato lavoro.

Gli algoritmi implementano operazioni di uso comune come, **conteggi**, **ricerche**, **copie di dati** oppure **ordinamenti** tra due array.

Gli algoritmi sono un concetto fondamentale nel mondo dell'informatica anzitutto perché é alla base della nozione teorica di **calcolabilitá**: un problema é calcolabile quando é risolvibile mediante un algoritmo. L'algoritmo, inoltre, é un concetto cardine anche nella fase di **programmazione**: preso un problema da automatizzare, la programmazione costituirà essenzialmente la traduzione o la codifica del l'algoritmo utile a risolvere il problema. Inoltre, questo modello ci aiuta a comprendere il funzionamento della **macchina di Turing**.

Ma andiamo nel dettaglio e vediamo le caratteristiche di un algoritmo:

Concetto di **Finitudine:** Un algoritmo deve avere un **numero limitato di passi**, quindi non é possibile che esso continui all'infinito, altrimenti non risolverebbe il nostro problema ma continuerebbe a lavorare senza mai arrivare all'obiettivo prestabilito.

Concetto di **Definitività:** Ogni istruzione dev'essere **chiara** e **non ambigua**, poiché le ambiguità porterebbero a dei risultati imprevisti.

Concetto di **Input e Output:** Gli algoritmi partono da un **Input iniziale** ovvero i dati iniziali che gli vengono forniti, ed ha il compito di produrre un **Output finale**, ovvero la soluzione al nostro problema.

Concetto di **Efficacia:** Lo scopo di un algoritmo oltre quello di trovare una soluzione al nostro problema é quello di farlo nel minor tempo possibile con il minimo utilizzo di risorse, infatti la qualitá di un algoritmo é misurata appunto dall'**efficienza**, ovvero la rapiditá e il risparmio di risorse con cui viene risolto il problema.

Concetto di **Generalità** Un buon algoritmo dovrebbe funzionare non solo per casi specifici ma per un intera classe di problemi simili. ****Per eseguire un algoritmo abbiamo bisogno di un elaboratore come ad esempio un computer.

****Per comprendere meglio possiamo fare un esempio in cui abbiamo un "**algoritmo"**, una **ricetta** la quale dice di "rompere le uova", quest'ultimo potrebbe essere considerato un passo elementare di un algoritmo di cucina, ma "aggiungere sale quanto basta", invece non é elementare in quanto non indica con precisione una quantità di sale ben precisa. ****Possiamo dire che un algoritmo é una **particolare macchina di Turing** o un **programma** della **macchina** di **Von** **Neumann.

Architettura della macchina di Von Neumann**

Una macchina di Von Neumann é una terna (N,IS,P) dove:

N={0,1,2,3,….} costituisce l'insieme dei numeri naturali ed é l'alfabeto della macchina.

IS={ZERO,INC,SOM,MOLT,DIV,UGUALE,MINORE,SALCOND,ALT} é **l'istruction set**, ovvero l'insieme delle istruzioni generiche della macchina.

P= {**I0, i1, i2, …, I|p| - 1**) é una sequenza finita e non vuota di istruzioni che vengono prese dall'insieme **IS** in cui vengono specificati i valori delle variabili, prende il nome di programma della macchina.

Adesso parliamo del suo funzionamento

Un programma eseguibile da una macchina di Von Neumann non é altro che una lista di istruzioni registrate in memoria centrale, che devono essere eseguite una per volta secondo l'ordine specificato nel programma e terminare solo quando incontra un istruzione di arresto.

La geniale idea trovata da Von Neumann fu quella di conservare in memoria sia le **istruzioni** che i **dati** ricavati dai calcoli, questo cambiamento rese piú semplice cambiare istruzione, adattare i programmi e realizzare operazioni complesse.

L'hardware della macchina di Von Neumann é costituito dalla **memoria** e da un **processore**.

Ad ogni locazione di memoria é associato un **indice** che rappresenta la sua posizione nella sequenza.

La memoria inoltre é strutturata per contenere:

Istruzioni del programma vengono memorizzate all'inizio, in locazioni con **indirizzi** che vanno da **0 fino a P-1** dove **P** indica il numero totale di istruzioni.

Dati del programma, che possono essere **letti** e **modificati** dalle istruzioni.

Il processore della macchina é costituito da 4 parti essenziali:

Program counter (Contatore di programma), é una locazione contenente l'indirizzo della prossima istruzione da eseguire. **Registro delle istruzioni:** ha il compito di conservare l'istruzione attualmente in fase di esecuzione. **ALU (Unità aritmetica e logica),** svolge operazioni aritmetiche e logiche necessarie per il calcolo. **CU o Control unit** (Unità di controllo): coordina il processo di esecuzione delle istruzione tramite una serie di segnali che indicano il cambiamento di stato, gestendo così l'intero processo.

Parliamo adesso del modo in cui questa macchina opera

La macchina di **Von Neumann** opera attraverso un modello di funzionamento che si basa sul ciclo **Fetch-decode-execute**, che fa si che la **CPU** possa eseguire le istruzioni di un programma in maniera sequenziale, e lo fa nel seguente modo:

Per prima cosa la **CPU** legge l'indirizzo dell'istruzione dal **Program counter**, dopodiché grazie all'indirizzo sappiamo dove si trova esattamente, quindi viene prelevata e la **CPU** la interpreta la tipologia di operazione (ad esempio somma,moltiplicazione,salto condizionato..etc), successivamente la **CPU** invia dei segnali all'unità aritmetico logica (**ALU**) per eseguire le operazioni indicate, se l'istruzione ha bisogno di operandi (**dati**) che sono in memoria allora la **CPU** li preleva utilizzando i relativi indirizzi di memoria, poi si passa alla fase di esecuzione dove l'**ALU** esegue l'operazione

richiesta, una volta calcolato il risultato viene registrato nella locazione di memoria, infine se l'istruzione non é un salto condizionato allora il **contatore verrà incrementato di 1** e si passa all'istruzione successiva, se invece é un salto condizionale il **contatore viene aggiornato** con il **nuovo indirizzo** modificando così la sequenza di esecuzione.

Questo ciclo appena descritto si **ripeterà** fin quando non si incontra l'istruzione **ALT** che indica la fine del programma.

Un'altro **modello** importante sino ai giorni d'oggi é la macchina di Turing che ci aiuta a capire a fondo il concetto di algoritmo.

Una **MdT** è un dispositivo di calcolo in grado di operare mediante una successione **finita di passi**. **Componenti di una MdT** (Macchina di Turing)

Una macchina di Turing é composta da un **nastro** nel quale sono contenute delle celle, ciascuna delle quali contiene un simbolo di un alfabeto finito spesso viene indicato come Σ (sigma), poi abbiamo la **Testina** che si muove lungo il nastro e posizionandosi su una cella ha il compito di leggere e scrivere il simbolo e può spostarsi a destra o a sinistra, un altro componente é l'**unita di memoria interna** ovvero una memoria costituita da un set finito di stati possibili, indicato con la lettera Q, e indica lo stato attuale della macchina, successivamente troviamo l'**unità di calcolo,** ha il compito di eseguire le operazioni logiche e aritmetiche, infine abbiamo l'**unità di controllo** quest'ultima legge il simbolo sotto la testina ed ha il compito di determinare quale sarà la prossima operazione della **MtD** come ad esempio, cambiare stato, scrivere un nuovo simbolo sulla cella corrente o spostare la testina a destra o sinistra.

Ma adesso parliamo di come funziona, una macchina di Turing é definita da un insieme di **regole** che ne definiscono il comportamento su un nastro di I/O (lettura/scrittura), possiamo immaginare questo **nastro** come un nastro di carta a lunghezza infinita, diviso in tanti quadratini che prendono il nome di **celle** l'insieme di queste celle definiscono una **sequenza lineare** di **celle,** ogni cella all'interno contiene un simbolo oppure é vuota, il suo contenuto può essere letto, scritto o eliminato dalla **Testina,** la macchina analizza il nastro, una cella per volta iniziando da quella non vuota più a sinistra. Ad ogni passo la macchina legge il simbolo contenuto nella cella, decide il suo **prossimo stato**, scrive un simbolo sul nastro e **sposta la testina** di una posizione. L'idea di base di Alan Turing era quello di simulare il funzionamento del cervello umano ad oggi abbiamo lo stesso obiettivo con l'intelligenza artificiale.

Come detto il comportamento della macchina viene stabilito da un insieme di regole, per esempio la quintupla **(0,A,1,B,-)** indica che la macchina si trova nello **stato** interno **0** e **legge** il simbolo sul nastro ovvero **A**, **passa** allo stato interno **1** e **scrive B** sul nastro dopodiché dato che c'é "**-**" la testina **rimarrà ferma**, se fosse stato "**>**" allora si sarebbe spostata a **destra**, se "**<**" allora si sposterà a **sinistra**.

Funzione parziale In una **MdT** una funzione parziale é una funzione che non da un risultato, se la MdT si ferma su un input allora produrrà un risultato, se invece entra in un ciclo infinito e non si ferma non arriveremo a nessun risultato per quell'input, quindi la funzione é parziale perché non sempre riesce a fornire un output.

Problema della fermata Il problema della fermata é un problema famoso nell'ambito della computazione ed indica l'impossibilità dello stabilire se un programma terminerà il suo lavoro oppure continuerà a funzionare all'infinito
Definizione del problema Dato un qualsiasi programma **P** e un input **I**, il problema della fermata sta nel determinare se **P** terminerà la sua esecuzione su **I** o se continuerà ad eseguire all'infinito.

Prova di indecidibilità Alan Turing, nel 1936 dimostrò che il problema della fermata é **Indecidibile**, poiché non esiste un algoritmo generale che, dato un programma e un input, sia in grado di determinare in modo corretto se quel programma terminerà o meno.

Memoria

La memoria é una delle principali componenti della macchina di **Von Neumann**, va immaginata come una piramide in cui in cima abbiamo le memorie **più veloci** (Memorie Primarie) e in basso quelle **meno veloci** (Memorie Secondarie) come, **Hard disk, SSD e nastri.** All'interno della memoria principale, un concetto fondamentale **spazio di indirizzamento,** indica il numero totale di indirizzi che possiamo utilizzare per accedere alle **celle di memoria**, esso viene determinato dall'**ampiezza dei bus di indirizzi** ovvero avremmo **2^n indirizzi**, quindi se il bus ha 4 linee possiamo avere $2^4 = 16$ indirizzi (da 0 a 15)

Cos'é un computer? Un computer é una macchina che computa, ovvero esegue un tipo di lavoro in maniera **automatica** attraverso appunto degli algoritmi, specificati in un **linguaggio binario** che può essere capito dalla macchina.

Il linguaggio binario é un sistema di numerazione che utilizza solo due simboli ovvero 1 e 0, ed é alla base di tutti gli elaboratori e dispositivi elettronici poiché é il modo in cui l'hardware rappresenta ed elabora i dati.

Il computer ha vari componenti tra cui:

Processore (CPU) Possiamo pensare al processore come al cervello del nostro elaboratore poiché si tratta di un componente di fondamentale importanza che ha il compito di svolgere calcoli e coordinare le operazioni, un processore é un singolo circuito integrato (**IC** integrated Circuit) in grado di effettuare **operazioni decisionali,** il processore può essere suddiviso in tre **Unità Funzionali** che sono, **CU** (Control Unit - Unità di Controllo) essa gestisce il **bus** controllando le linee **ABus,DBus** e **CBus,** leggendo e aggiornando i dati dalla memoria o dalle periferiche, poi abbiamo i **Registri,** ovvero particolari aree di memoria riservata che hanno il compito di far risparmiare tempo alla memoria **RAM,** conservando i dati letti o i risultati delle operazioni prendendo informazioni direttamente ****senza dover passare dal **bus** evitando così il cosiddetto collo di bottiglia (**bottleneck**), infine abbiamo L'**ALU** (Unità aritmetico logica) che esegue operazioni matematiche e logiche, essa funziona attraverso dei **microprogrammi** che vengono scritti in **microcodice,** un codice speciale che contiene le **microistruzioni** essenziali per svolgere le operazioni base. Ogni microistruzione è una singola operazione che deve essere eseguita dal processore, come caricare un valore in un registro, eseguire una somma o spostare dati tra memorie e registri, queste **microistruzioni** vengono organizzate in **microcodice,** ovvero delle sequenze di microistruzioni necessarie per eseguire operazioni **più complesse**, il registro **µPC** (**Microprogram Counter**) tiene traccia della prossima microistruzione da eseguire e quando una microroutine inizia il **µPC carica** la prima microistruzione, e ad ogni passo, si **incrementa** automaticamente per seguire le istruzioni in **ordine** fino al completamento dell'operazione.

Ogni processore dispone di un **set di istruzioni** specifico denominato **ISA** (Istruction Set Architetture o Istruction Set), ogni istruzione é contraddistinta da un numero specifico, **Op. Code** (**Operation Code)** ed é dotata di un numero preciso e definito di parametri che prendono il nome di **operandi** che insieme all'op code ne determinano la lunghezza dell'istruzione in byte, inoltre ad ogni op code viene associata una **descrizione mnemonica** che ci aiuta a ricordare cosa fa l'istruzione, ad esempio se l'istruzione é una somma (**ADD**) noi ogni volta che vedremo Add sappiamo che é un istruzione di somma nonostante il processore lo interpreterà come 01.

Il percorso che seguono i dati all'interno della CPU si chiama **Data path**, questo percorso coinvolge principalmente l'ALU, i registri e il Bus. Le istruzioni possono essere di due tipi **Registro→Registro** in cui si va a trasferire un dato da un **registro** ad **un'altro** oppure **Registro→Memoria** dove invece si trasferisce un dato dalla **memoria** esterna a un **registro** della **CPU** o viceversa, le istruzioni possono essere implementare in due modi, via **hardware** direttamente nella CPU come parte integrata del suo circuito oppure via **software** con un interprete, che si basa sulle istruzioni hardware per svolgere quelle piú complesse. Inizialmente esistevano le architetture **CISC** (Complex Istruction Set Computer) esse usavano molte **istruzioni complesse**, che venivano interpretate **lato software** ed erano supportate da grandi colossi come **IBM** e **Intel,** successivamente negli anni 80 nacquero le architetture **RISC** (Reduced Istruction Set Computer) come il **DEC Alpha** erano basate su poche istruzioni semplici eseguite in maniera diretta senza bisogno di un interprete, ad oggi anche i processori come il **Pentium (https://it.wikipedia.org/wiki/Pentium)** utilizzano l'architettura ibrida.

Un processore con architettura **CISC** é caratterizzato dalla presenza di **microprogrammi** e **microcodice,** é un processore ad **interprete** perché ogni istruzione dell'ISA richiede diversi passaggi per essere eseguita, come la **decodifica** per identificare l'operazione da eseguire, il caricamento degli **Operandi** e infine l'esecuzione tramite **microprogramma specifico** progettato appositamente per quell'istruzione; l'architettura **CISC** dispone di un ampio set di istruzioni, con formati di lunghezza variabile a causa del numero di **operandi** richiesti, questo

rende la fase di decodifica complessa, poiché il processore si trova a gestire istruzioni di lunghezza differente tra loro, un'altra caratteristica é la sua **potabilità,** il **microcodice** all'interno dispone di una sorta di **interprete,** questo microcodice può essere utilizzato anche sui nuovi modelli di processori senza bisogno di andare a riscriverlo.

Al contrario delle architetture **CISC** le architetture **RISC** hanno un numero limitato di istruzioni, ognuna con **lunghezza** e **numero di operandi** fisso, questo semplifica la fase di decodifica e fa si che le istruzioni vengano eseguite direttamente dall'hardware e non da microprogrammi, queste architetture come detto sono molto più veloci ed hanno tempi di esecuzione ridotti fino a 10 volte rispetto alle architetture **CISC,** l'unica pecca é che i programmi **RISC** sono più lunghi poiché ogni istruzione é molto semplice e richiede un singolo ciclo e per eseguire operazioni complesse si ha bisogno di piú istruzioni, per questo motivo i programmi tendono ad essere più lunghi.

Un'altra architettura é l'architettura **CRISC,** nata ****a causa delle difficoltá che si riscontrano in **CISC** e **RISC,** questa architettura fa uso della base **CISC** combinata ai componenti interni ottimizzati in stile **RISC** cosí da sfruttare i punti di forza di entrambe, si utilizza appunto il modello di **Von Neumann** come **base** che ha il compito di **gestire** l'ampia gamma di **istruzioni** e la **compatibilità** e l'unità RISC all'interno del processore, per eseguire rapidamente le istruzioni.

In fine abbiamo l'architettura **SISD** (Single Istruction Single Data), come dice il nome stesso esegue una **singola istruzione** e un **singolo dato** alla volta, quindi ha dei limiti di prestazione, perché eseguire un'operazione per volta risulta troppo lento quindi ad oggi si cerca di **parallelizzare** le operazioni in modo tale da aumentare la velocità di calcolo.

Ogni processore agisce secondo una rigida sequenza di passi che si ripetono fino all'arresto della macchina, questa sequenza prende il nome di **Fetch-Decode-Execute-Store,** nella fase di **Fetch** la **CU** prende il valore del **Program Counter,** che contiene l'indirizzo della prossima istruzione da eseguire e lo invia sull'**ABus** per leggere dalla memoria l'**Op Code** della nuova istruzione da eseguire, poi si passa alla fase di **Decode** durante il quale la **CU** decodifica l'**Op Code** determinando il numero di passaggi (detti anche **Operandi)** ha bisogno, successivamente avvia la fase di **Operand-Fetch** caricando gli operandi dagli indirizzi di memoria per memorizzarli nei registri, si arriva poi alla fase di **Execute** ovvero la vera e propria fase di **esecuzione** del **microprogramma** associato all'**Op Code,** utilizzando i parametri caricati nei **registri,** il completamento di questa fase dipende dal **Clock della CPU,** poiché é lui che definisce

la velocità con il quale le istruzioni vengono eseguite,questo rallentamento é uno dei fattori chiave legato appunto al **collo di bottiglia** (Bottleneck), infine si arriva alla fase di **Store** appunto "Salvataggio" in cui i risultati ottenuti dai calcoli vengono inviati tramite il **BUS** della **CU** nella memoria oppure a dispositivi di Input Output.

Ma **quando termina** questo ciclo? Il ciclo prima descritto terminerà soltanto quando, durante la sua esecuzione incontrerà un segnale di **INTR** (Interrupt), questo segnale indica che una periferica ha richiesto l'attenzione del processore per eseguire un'altra determinata azione, quindi il ciclo verrà momentaneamente sospeso e il processore eseguirà la nuova richiesta.

Come accennato prima

Il **BUS** é un percorso di comunicazione tra le varie componenti della macchina, abbiamo piú tipologie di bus ovvero il **bus di indirizzi, bus di controllo** e **bus dati.**

Il **Bottleneck** si verifica quando la larghezza di banda del bus é insufficiente a gestire il volume di dati richiesti dai vari componenti quindi accade un vero e proprio collo di bottiglia in cui molti dati cercano di passare all'interno di un restringimento.

Memoria Centrale é il luogo nel quale sono contenute le istruzioni da eseguire, essa si suddivide in vari tipi:

RAM (Random Access Memory), una memoria di tipo volatile, ovvero mantiene le informazioni fino a quando essa é alimentata

ROM (Read Only Memory) é una memoria di sola lettura che contiene i dati di avvio

Memoria **CACHE,** una memoria di passaggio che ha il compito di memorizzare le informazioni che la **CPU** utilizza più frequentemente in modo da andare a ridurre il tempo che si impiega per recuperare queste informazioni dalla **RAM.** La memoria cache é **molto più veloce** rispetto alla **RAM** questo perché si tratta di tratta di una memoria **statica** quindi gli indirizzi al suo interno non cambiano, a differenza invece delle memorie **dinamiche** dove i dati sono distribuiti in maniera meno organizzata.

****Quando il processore **richiede** un **dato dalla memoria,** non va solo a caricare il dato richiesto ma prende anche quelli vicini, copiandoli così nella **memoria cache** questi gruppi di dati memorizzati assieme prendono il nome di **linee di cache.** La memoria cache sfrutta due principii importanti ovvero il principio della **località temporale** che si basa sul principio che i dati utilizzati di recente hanno un'alta probabilità che vengano riutilizzati nuovamente, quindi la cache **memorizza temporaneamente** i dati più recenti chiamati **MRU** (Most Recently Used) e fa si che se il **processore** ne avesse bisogno nuovamente può accedere direttamente nella **cache,** evitando così di accedere nuovamente alla RAM **risparmiando tempo,** un altro principio importante é il principio di **località spaziale** riguarda invece la **posizione dei dati**, per l'appunto il fatto che se utilizziamo un dato, é probabile che anche i **dati vicini** vengano utilizzati poco dopo, per far ciò la **cache** carica le **linee di cache** che contengono dati vicini la quale dimensione può variare dai **4-512** byte così facendo il processore avrà **accesso immediato** a tutti i dati della linea senza dover effettuare accessi alla **memoria principale.** Quando il **processore** ha bisogno di un dato, per prima cosa controlla nella **cache** se lo trova li, in quel caso si dice (**Hit**) e può usarlo subito senza dover accedere al **Bus** e quindi alla memoria **RAM,** se invece non é in cache si dice (**Miss**) e quindi il processore deve andare a recuperarla tramite **Bus** e una volta trovata la carica nella **cache** sostituendola alla linea meno usata di recente, i tempi di accesso alla memoria molto spesso di diversi **cicli di clock** soprattutto nel caso in cui il dato da leggere non si trovi nella **cache**, in questo caso si incorre in un caso di **cache miss,**

in questi casi l'elaboratore deve accedere alla memoria principale la DRAM per recuperare il dato richiesto, questo processo richiede molto più tempo rispetto una semplice lettura in cache, poichè il dato prima di arrivare deve attraversare vari stadi, tra cui il bus e il controller di memoria prima di arrivare dalla DRAM al processore, questo ritardo si traduce in un aumento significativo dei cicli di clock necessari per completare l'operazione. Il ritardo che si verifica prende il nome di **CAS Latency** (Column Address Strobe Latency) che indica il numero di cicli di clock necessari per ottenere una risposta dalla DRAM, poichè questo recupero di dati richiede tempo, tutte le istruzioni che seguono devono essere ritardate dello stesso numero di cicli, questo accade perché il processore deve attendere che i dati siano pronti prima di poter proseguire con l'esecuzione delle successive. All'interno di un computer solitamente troviamo **tre livelli di cache**, ovvero la cache di **livello 1 (L1)** é **la più veloce** e si trova **all'interno** del **processore**, vicino ai registri e le unità di calcolo ed ha dimensioni che variano tra i **16KB** e i **128KB** per core, questa cache viene usata per le **istruzioni** a cui il processore accede molto frequentemente, poi abbiamo la cache di **livello 2 (L2)** é leggermente più lontana rispetto ad L1, ha **dimensioni maggiori** che variano da **256KB** fino ad arrivare a qualche **MB** per core, qui vengono salvati i **dati** e le **istruzioni** che non si trovano in L1 ed infine abbiamo la cache di **livello 3 (L3)** é la più grande rispetto **L1** e **L2** e nonostante sia più **lenta** di quest'ultime é comunque **più veloce** della **RAM**, L3 funziona come una sorta di **backup** per i dati non trovati in **L1** e **L2.**

Per migliorare l'efficienza del processore oltre a fare uso della memoria cache, si incorre anche al concetto di **Parallelismo d'esecuzione**, ovvero una tecnica che consente al processore di eseguire più istruzioni contemporaneamente, in particolare si parla di **Prefetch,** in pratica il **prefetching** aiuta a mantenere un flusso continuo di dati per il processore riducendo i tempi di attesa, esistono diverse tipologie di prefetching, **prefetching hardware,** la CPU ha un'unità hardware chiamata prefetcher, che rileva automaticamente schemi ripetitivi di accesso ai dati e li carica in anticipo nella cache, un altro tipo di prefetch é il **Prefetching Software, inserito direttamente** dai programmatori nel **codice** come ad esempio delle variabili che e aiutano la CPU a sapere quali dati verranno utilizzati presto, sempre lato software alcuni **compilatori moderni analizzano** il **codice** e **caricano** in anticipo i **dati** più **utilizzati** nella cache, in fine abbiamo il **Prefetching mirato alla cache**, in questo caso vengono caricati direttamente nella cache **interi blocchi** di memoria di dimensioni precise circa **64 byte** per consentire un accesso più rapido. Presto alla coda di prefetch venne affiancato un sistema a **Pipeline,** un sistema a pipeline permette di iniziare ad eseguire una **nuova** operazione anche se quella precedente é **ancora in esecuzione**, aumentando così il numero di operazioni processate per unità di tempo. La **Pipeline** si suddivide in **cinque** fasi che sono, **IF** (Istruction Fetch) legge l'istruzione dalla memoria, **ID** (Istruction Decode) decodifica l'istruzione, identifica i registri e legge gli operandi necessari, **EX** (Execution) esegue l'operazione (che può essere somma, sottrazione ecc.) usando l'ALU **o la FPU, MEM** (Memory) interagisce con la memoria se necessario, per le operazioni di lettura e scrittura, infine **WB** (Write Back) salva il risultato nel registro corrispondente, ogni fase svolge una parte di **elaborazione differente**, così facendo possiamo avere fino a **cinque operazioni simultanee** in corso, con il sistema a **pipeline** lo stato di **EX** execution tende ad essere **più lento** creando cosi un **collo di bottiglia**, per tanto, per risolvere questo problema nei processori **superscalari** vengono aggiunte più **ALU,** permettendo così l'esecuzione parallela di più operazioni, grazie alla **pipeline** il processore può elaborare **più istruzioni** al **secondo**, riducendo il tempo di **inattività** tra un operazione e l'altra. Per far si che la **Pipeline** funzioni in maniera corretta bisogna **mantenere le informazioni** tra uno stato e l'altro, queste informazioni vengono salvate nei **Buffer Inerstadi,** questi buffer conengono i **Registri interstadio (RA,RB,PC_Temp)** e servono a trasferire i dati tra le fasi, **Registro IR,** che memorizza gli indirizzi sorgente e destinazione ed infine i **Segnali di controllo** che servono a regolare il comportamento degli stadi successivi, sono fondamentali per il funzionamento di un processore, servono a coordinare e gestire ogni operazione necessaria, esistono più tipologie di segnali di controllo come i **segnali di selezione** per i multiplexer servono a determinare quale ingresso del multiplexer deve essere attivato, **segnali di attivazione** per i registri**,** indica quando i registri possono leggere o scrivere un dato, questi segnali si assicurano che queste modiche

possano essere effettuate solo nei momenti opportuni, poi abbiamo i **segnali di condizione,** descrivono stati specifici del sistema o gli esiti delle operazioni, ad esempio possono indicare se un valore è pari a zero o se un confronto ha dato un esito positivo, **segnali per la gestione della memoria,** quest'ultimi coordinano le operazioni di lettura e scrittura in memoria e indicano se è necessario accedere alla memoria per prendere un dato, **operazioni dell'ALU,** per ogni istruzione, i segnali di controllo indicano all'ALU quale operazione eseguire , ad esempio una somma, una sottrazione o un'operazione logica. Questi segnali sono generati dal **circuito di controllo,** e possono essere prodotti principalmente in due modi, in modo **cablato** usando circuiti logici **fissi** per generare segnali di controllo, questo approccio si basa sulla **logica combinatoria,** in cui l'uscita dipende esclusivamente dagli ingressi correnti, è progettato per garantire la massima velocità poiché le decisioni vengono implementate direttamente tramite circuito fisico, oppure possono essere implementati tramite **microprogrammazione**, che invece fa uso di una sorta di programma interno memorizzato nella **ROM,** per generare questi segnali, in questo caso il processore si basa sulle istruzioni scritte nella **ROM** per decidere quali segnali inviare rendendo il sistema più flessibile ma leggermente meno veloce rispetto al controllo cablato. Il funzionamento del controllo cablato si basa su un **contatore modulo 5** (Un contatore che conta da 0 a 4 quindi può assumere cinque stadi distinti 0,1,2,3,4) , che va a suddividere l'esecuzione di un istruzione in fasi, scandite dal segnale di clock, ogni fase corrisponde ad una specifica azione come leggere, decodificare o eseguire, la parte essenziale del controllo cablato è il **decodificatore di istruzioni,** il quale legge le istruzioni e le memorizza in un registro speciale (**IR**) e la traduce in **codice binario**

Per migliorare l'efficienza della **pipeline** quando si eseguono salti condizionati (salta solo se la condizione è verificata) e incondizionati (salta anche se la condizione non è verificata) si utilizza una tecnica chiamata **predizione dei salti,** quest'ultima può essere di due tipi **Predizione Statica** che è una forma di predizione che si basa su un approccio predefinito, in cui il processore fa una scelta sul salto da effettuare senza tener conto delle esecuzioni precedenti questa tipologia di predizione non si adatta durante l'esecuzione ma si basa su regole generali qualche esempio di predizione statica é il **saltare sempre,** il processore predice che tutti i salti dovranno essere effettuati questo approccio è molto semplice ma può essere inefficiente quando il salto non viene eseguito poiché deve annullare le istruzioni errate e riprendere da quelle corrette, causando così un ritardo, un altro esempio di predizione **statica** é il **non saltare mai,** il processore predice che nessun salto verrà eseguito che può essere anch'essa inefficiente se i salti sono frequenti, un'altra strategia é quella di predire che il salto avvenga solo quando é destinato ad un indirizzo fisso, questo prende il nome di **salto al target,** la predizione **statica** è più **semplice** e richiede **meno** risorse **hardware** ma di conseguenza è molto **meno precisa** rispetto alla **predizione dinamica** che al contrario di quella statica **si adatta** in base al comportamento storico delle istruzioni di salto, il processore tiene conto delle decisioni prese in precedenza e si basa su quest'ultime per fare una previsione più precisa sui salti futuri, questa tipologia di predizione migliora nel tempo, diventando sempre più precisa man mano che il programma viene eseguito, tra i vari approcci alla tipologia dinamica troviamo le **macchine a stati**, le più semplici sono quelle a **due** stati che utilizzano due possibili predizioni, **probabilmente salta** (**PS**) il salto è previsto, **probabilmente non salta** (**PNS)** salto non previsto, il processore passa da uno stato all'altro basandosi come detto sui risultati delle predizioni passate, se la predizione è corretta lo stato rimane invariato se invece non è errata lo stato cambia e la predizione si inverte, nelle macchina a **quattro** stati che offrono una precisione maggiore, qui troviamo, **Molto probabilmente Salta** (**MPS**) massima fiducia che il salto avvenga, **Probabilmente Salta** (**PS**) fiducia moderata sul fatto che il salto avvenga, **Probabilmente Non Salta** (**PNS**) moderata fiducia che il salto non avvenga ed infine, **Molto Probabilmente Non Salta** (**MPNS**) massima fiducia che il salto non avvenga, il processore sposta gli stati gradualmente se la predizione è corretta allora **aumenterà** il livello di **fiducia** passando ad uno stato piú certo come ad esempio da **PS** a **MPS,** se invece è errata **diminuirà la fiducia verso** uno stato più **debole** come per esempio da **PS** a **PNS.** La predizione nella fase di fetch fa uso di una piccola unità chiamata **Buffer di destinazione di salto,** questo buffer aiuta il processore a prevedere dove andare quando incontra un

istruzione di salto, al suo interno troviamo, **l'indirizzo dell'istruzione** di salto che serve ad identificare lo stato specifico del programma, **bit di stato,** che indicano se il salto é probabile o meno che venga eseguito ed infine **l'indirizzo di destinazione,** ovvero il punto in cui andare se il salto si verifica. Il funzionamento é semplice, ogni volta che il processore preleva un'istruzione da eseguire verifica se quest'ultima é presente all'interno del **buffer di destinazione di salto,** se presente il processore utilizza le informazioni presenti per capire subito dove andare dopo, in caso contrario il processore continua normalmente eseguendo le istruzioni una dopo l'altra senza fare previsioni particolari, in caso di grandi programmi il buffer viene aggiornato in maniera dinamica durante l'esecuzione. Quando il processore esegue istruzioni può capitare che la stessa istruzione abbia bisogno della stessa risorsa hardware nello stesso momento, in questo caso la pipeline va in **stallo,** ad esempio ad ogni **ciclo del clock** il processore accede alla cache per prelevare la prossima istruzione da eseguire, ma se un istruzione come **Load** che carica i dati o **Store** che salva i dati, se hanno bisogno di accedere alla memoria nello stesso momento si crea un **conflitto** e in questa situazione il processore deve fermarsi e aspettare rallentando così **l'intero flusso** di esecuzione per evitare questo problema si può utilizzare una cache separata, una per le **istruzioni**, che verrà utilizzata per **prelevare** le **istruzioni** da eseguire e un'altra per i **dati**, utilizzata per **gestire** i **dati** necessari per le istruzioni **Load e Store.** Con le cache separate il processore può accedere contemporaneamente sia alle istruzioni che ai dati evitando così di andare in stallo e rendendo tutto molto più efficiente. Per valutare l'efficienza delle prestazioni di un processore è molto importante considerare diversi fattori come il numero di istruzioni che vengono eseguite, il tempo necessario per completarle e il tipo di architettura utilizzata, un processore senza **pipeline** esegue le istruzioni una alla volta, senza sovrapporre le fasi di fetch, decode, execute, il **tempo di esecuzione totale** si calcola con la formula: **T= N x S / R,** dove **N** indica il **numero** di **istruzioni** del **programma**, **S** è il **numero** di **cicli** di **clock** necessari per eseguire ogni istruzione, **R** indica la frequenza di clock del processore (quanti cicli esegue al secondo), per sapere quante operazioni vengono eseguite al secondo bisogna calcolare la frequenza di operazione ovvero il **throughput,** e si calcola con la formula **P = np x R / S,** np rappresenta il numero di programmi o operazioni eseguite in parallelo, un processore con pipeline invece, permette di sovrappore le fasi di esecuzione delle istruzioni migliorando le prestazioni, in questi casi si verifica il **throughput (caso ottimo)** dove il numero di operazioni completate è uguale al numero di **cicli di clock**, quindi ad ogni ciclo un istruzione verrà completata, la formula è **P = R**, quindi se il processore ha una frequenza di 2 Ghz (**2 miliardi** di cicli al secondo) e di conseguenza il **Throughput** ottimo è di **2 miliardi** di **istruzioni completate** al **secondo** In caso di conflitti, il numero medio di **cicli di clock per istruzione** (**S**) aumenta, questo accade perchè da questi conflitti vengono generati **stalli** o operazioni aggiuntive, facendo perdere tempo alla **CPU,** un caso comune è il **flushing** della pipeline che si verifica quando dobbiamo svuotarla in caso di errori nei salti condizionali o dipendenze tra istruzioni, ogni conflitto che comporta un aumento di medio nei cicli di stallo e si calcola con la formula **δ = p x c**, δ delta, p è la probabilità che il conflitto accada, c è il numero medio di cicli di clock per risolvere il conflitto.

Esistono più tipi di conflitti, **conflitti di dipendenza di dato,** questi accadono quando un istruzione dipende dal risultato di un altra, la probabilità che quest'ultimo si verifichi è del 10% (**P_dato= 0.1**), ad ogni conflitto viene introdotto uno stallo di 1 ciclo di clock (c_dato= 1) per calcolare i cicli aggiuntivi si utilizza la seguente formula :

$$\delta_{dato} = p_{dato} \cdot c_{dato}$$

Conflitti di salto, si verificano in caso di errore nella predizione di un salto condizionale, la probabilità che si verifichino è del **20%** (**P_salto = 0.2**), l'accuratezza del predittore di salto è del 90% quindi il tasso di errore è del 10% (1 - 0.9 = 0.1), ogni errore causa uno stallo di 1 ciclo di clock (C_salto = 1), per calcolare i cicli aggiuntivi si utilizza la seguente formula:

$$\delta_{salto} = p_{salto} \cdot c_{salto}$$

infine abbiamo i **conflitti per cache miss,** quest'ultimi si verificano quando i dati o le istruzioni richieste non sono disponibili nella cache e devono essere cercati in memoria principale, la probabilità che accada è del 5% (**P_miss = 0.05%**) quindi se (C_miss = 1) allora P_miss x C_miss = 0.05 x 1 = 0.05 , per calcolare i cicli aggiuntivi si utilizza la seguente formula :

$$\delta_{miss} = p_{miss} \cdot c_{miss}$$

Il 4 conflitto é il limite di risorse (da approfondire (forse)).

Quindi il **Throughtput reale** del processore, tenendo conto dei ritardi causati dai conflitti si calcola con la formula:

$$P_p = \frac{p \cdot R}{S + \delta + \delta_{dato} + \delta_{salto} + \delta_{miss}}$$

R è la frequenza di clock del processore

S indica il numero medio di cicli per istruzione senza conflitti

δ,δdato, δsalto,δmiss rappresentano i cicli di stallo aggiuntivi causati dai diversi tipi di conflitti

I processori superscalari sono progettati appunto per aumentare il **Throughput**, eseguendo più istruzioni in parallelo grazie a più unità di esecuzione indipendenti, questo fa si che si possa superare il limite di un'istruzione per ciclo di clock, le **caratteristiche** dei processori superscalari sono, **l'emisione multipla,** ovvero possono inviare **più istruzioni** alla volta alle unità di esecuzione, **unità di esecuzione** separate, contengono infatti due unità, **l'unita aritmetica** con il quale svolge calcoli con i numeri interi, numeri in virgola mobile o array, e un'**unità load/store** per le operazioni di lettura e scrittura in memoria. Essi sono composti da diversi componenti tra cui, il **Fetch unit,** che preleva le istruzioni dalla memoria e le inserisce in coda, **dispatch unit,** distribuisce le istruzioni alle **stazioni di prenotazione** una componente che ha il compito di inviare le istruzioni alle unità di esecuzione, il loro scopo è quello di coordinare e ottimizzare l'esecuzione parallela delle istruzioni, poi abbiamo la **commit unit** che riordina i risultati e aggiorna lo stato del processore ed infine **l'unità di esecuzione.** Inoltre esistono anche i processori superscalari a più unità di elaborazione che a differenza di quelli classici, essi dividono il carico di lavoro tra più **pipeline autonome** e **unità di elaborazione indipendenti,** durante l'esecuzione delle istruzioni segmentano il flusso del programma in blocchi e lo assegnano alle **pipeline parallele.** Nei processori superscalari la fase di **smistamento** è una fase **cruciale** che garantisce l'esecuzione corretta e parallela delle istruzioni, durante questa fase il processore assegna le istruzioni alle unità di esecuzione disponibili, assicurandosi che tutte le risorse siano disponibili per evitare conflitti o interruzioni, in questa fase avviene un controllo delle risorse durante il quale si verifica che ci siano abbastanza **registri temporanei** per salvare i risultati intermedi (dati generati durante l'esecuzione dell'istruzione), si controlla anche che ci sia spazio libero nelle **stazioni di prenotazione** e ci si assicura che ci sia una **locazione disponibile** nel **buffer di riordino** ovvero, una struttura fondamentale in grado di assicurarsi che i risultati della istruzioni vengano scritti nei registri o nella memoria **nell'ordine corretto**, sempre durante la fase di smistamento si ha una **gestione delle dipendenze** cioè se due istruzioni dipendono una dall'altra, l'unità di smistamento **blocca l'istruzione dipendente** fin quando i dati necessari non saranno **disponibili.**

Quando utilizziamo una **Pipeline** possiamo incorre in alcuni **rischi** che possono compromettere l'efficienza dell'elaborazione come ad esempio il **Data Hazard,** si verifica quando un'istruzione ha bisogno di un dato che non é ancora stato prodotto dall'istruzione precedente come ad esempio il risultato di un calcolo che ancora non é stato completato, **Control Hazard** causato dai salti condizionali o da chiamate a funzioni, quando il flusso del programma cambia la pipeline potrebbe aver già caricato le istruzioni successive al salto che di conseguenza potrebbero non essere

più corrette, ad esempio quando si ha un istruzione di salto si deve decidere a quale indirizzo andare, la pipeline potrebbe aver caricato istruzioni sbagliate e di conseguenza si incorre in un errore, **Structural Hazard,** si verifica quando più istruzioni cercano di accedere alla stesa risorsa hardware, come una singola **ALU** o una porta di memoria causando così un conflitto di accesso alla risorsa, questo problema potrebbe essere risolti utilizzando un hardware avanzato con più **ALU** o **FPU** multiple e utilizzare degli **stalli o interrupt.** Gli **Stalli** sono interruzioni momentanee della pipeline utilizzati per gestire questi tre problemi appena descritti, andando a bloccare la pipeline fin quando non si risolvono questi conflitti, per inserire questi creare questi stalli si utilizzano delle istruzioni **Nulle (NOP)** che non svolgono alcuna operazione occupano uno slot nel flusso di memoria, permettendo alle operazioni precedenti di completarsi senza causare errori o conflitti, e dunque allineare il codice. Una tecnica avanzata utile ad ottimizzare il flusso della pipeline, specialmente durante le operazioni di salto si chiama **esecuzione predicativa,** essa implementa dei moduli chiamati **unità di previsione dei salti** (Dynamic Branch Prediction), questa tecnica in breve cerca di prevenire **interruzioni** e **perdite** nella pipeline, esse possono essere gestite via **software** attraverso il compilatore oppure direttamente dall'**hardware** attraverso circuiti di controllo complessi. L'utilizzo delle **NOP** tuttavia comporta uno spreco di risorse pertanto quando possibile è meglio utilizzare l'**inoltro degli operandi** (Forwarding). Queste unità di previsione utilizzano algoritmi avanzati e tabelle simili a memorie cache per **prevedere il comportamento dei salti,** i salti possono essere considerati **presi** (taken) ad esempio, i salti **all'indietro** come nei **cicli** sono spess o considerati presi poiché ad ogni iterazione si effettua un salto **all'indietro** , oppure i salti **non presi** (not taken), tuttavia questa tecnica dispone di alcuni problemi come, nel caso che la previsione sia sbagliata il processore eseguirebbe le istruzioni inutili per poi eliminarle e tornare indietro perdendo tempo, questo prende il nome di esecuzione **speculativa** in cui il processore ****inizia a elaborare le istruzioni come se la previsione fosse corretta se poi l'istruzione é realmente corretta allora il processore sfrutterà il lavoro fatto durante questa esecuzione risparmiando così tempo e migliorando le prestazioni, se invece era sbagliata le istruzioni eseguite per errore vengono scartate e si riparte dall'istruzione corretta. Quando il processore rileva una **dipendenza** tra le istruzioni della **pipeline** invece di bloccarla completamente utilizza l'**esecuzione fuori ordine,** il processore salta l'istruzione bloccata (ovvero quella che dipende da un dato ancora non disponibile) e prosegue con l'esecuzione delle **istruzioni successive,** in questo modo la pipeline rimane attiva, tranne lo stato dove manca il dato per essere eseguito che verra completato successivamente una volta che il dato richiesto sarà disponibile. Un problema di questa tecnica si ha quando il processore dev'essere interrotto a causa di un **interrupt,** se il processore si trova in fase **fuori ordine,** lo stato del processore potrebbe essere incoerente perchè i dati della memoria non riflettono esattamente l'ordine del programma, in questi casi il processore è costretto ad annullare le istruzioni eseguite fuori ordine e riportare tutto allo stato originale e rielaborare le istruzioni nel modo corretto. Una delle dipendenze più comuni si verifica quando un'istruzione deve leggere un valore da un'altra calcolato precedente, questa dipendenza prende il nome di **RAW** (Read After Write) ad esempio se abbiamo un **istruzione 1: A = B + C** in cui il valore di A é uguale alla somma di B + C e un **istruzione 2: D = A + 1,** in questo caso la **seconda** istruzione **dipende** dal risultato della **prima**, se il processore tenta di eseguire la seconda istruzione prima che la prima sia stata completata si verifica un problema di **dipendenza RAW.** Un'altra dipendenza è **WAW** (Write After Write) si verifica nel momento in cui due istruzioni in **pipeline** cercano di **leggere** un registro o una locazione di memoria **contemporaneamente**, in questo caso succede che l'ultimo che scrive **sovrascrive** il primo. Abbiamo poi la dipendenza **WAR** (Write After Read) quest'ultima si verifica nel momento in cui si cerca di **scrivere** in una locazione di memoria o un registro prima che un'altra istruzione abbia già **completato** il suo lavoro. Esistono più metodi per risolvere queste dipendenze di stato, ad esempio il **forwarding degli operandi** sorvolando la fase di di store, quando un'unità richiede un dato si può inoltrare direttamente il dato all'unità che lo richiede senza effettuare uno store, un'altra tecnica è quella di utilizzare gli **Stalli** ovvero una tecnica che permette di bloccare temporaneamente un istruzione dipendente, fin quando il dato richiesto sarà disponibile, un'altra tecnica è il **riordino delle istruzioni,** riordinando le istruzioni in modo da eseguire **prima** quelle **indipendenti** e poi quelle **dipendenti,** un'altra tecnica è la **speculation,** dove il processore esegue in anticipo un'istruzione, ipotizzando che il

risultato sia corretto oppure utilizzare il **Prefetcing,** caricando le istruzioni o i dati in anticipo in previsione che vengano utilizzati. Per gestire correttamente il flusso di esecuzione anche in caso di dipendenze **RAW** i processori utilizzano dei **registri d'appoggio,** ovvero dei registri interni invisibili al programmatore che servono a memorizzare temporaneamente i risultati delle istruzioni eseguite fuori ordine, quando arriva il momento di riordinare le istruzioni e quindi ripristinare l'ordine corretto del programma, il processore usa una tecnica chiamata **register renaming** in cui i registri interni vengono rinominati con i nomi dei registri effettivi utilizzati dal programma risparmiando così tempo invece di andare a trasferire i valori dei registri temporanei. ****

Anche se non esplicitamente tutte queste innovazioni come pipeline e super-scalarità cercano di avvicinarsi al modello di esecuzione parallela **VLIW** (Very Long Instruction Word), questo modello cerca di migliorare il parallelismo esecutivo sfruttando l'hardware della CPU e progettando il codice in maniera tale da essere già ottimizzato per l'esecuzione parallela, nel **VLIW** i compilatori generano un **codice** in grado di **organizzare più operazioni indipendenti** su un'unica **istruzione lunga** e permette alla **CPU** di svolgere le operazioni in parallelo senza effettuare ulteriori controlli **sull'hardware,** esso però presenta dei limiti, il **VLIW** funziona bene solo quando le operazioni del codice sono **indipendenti** tra loro, se sono presenti troppe dipendenze tra le istruzioni questo modello risulta **inefficiente**, poiché non è possibile eseguire le istruzioni in parallelo e rimarrebbero **buchi** nei cicli di esecuzione sprecando così risorse, quando il codice non contiene abbastanza **istruzioni indipendenti**, la **CPU** non riesce a sfruttare appieno il **parallelismo**, per superare questo problema è necessario che il **compilatore** sia abbastanza **avanzato** in modo tale da analizzare il codice e andar a **ridurre** al minimo le **dipendenze** tra le istruzioni, anche se avvolte questa soluzione non è sufficiente e allora è compito del **programmatore** scrivere un **codice ottimizzato**, un altro svantaggio si verifica nei programmi dove le operazioni devono essere eseguite in stretta **sequenza** senza la possibilità di parallelizzare, in questo caso il **VLIW** risulta **meno efficiente** rispetto ad altre tecnologie come **l'esecuzione fuori ordine** che è progettata appositamente per adattarsi meglio alle dipendenze tra le istruzioni.

Adesso parliamo nello specifico delle tipologie di memoria

RAM Un'altro componente molto importante é la **RAM (Random Access Memory)** che, per non perdere il suo contenuto ha bisogno di essere sempre "Rinfrescata" con un **segnale elettrico** a frequenza costante per tutto il tempo in cui il sistema é alimentato. I **Bit** di questa memoria sono dei **microcondensatori** ed é per questo motivo che questa memoria é nota come **DRAM** (Dynamic RAM). Ogni tipo di memoria RAM dispone dei propri tempi di accessi, che in confronto ai tempi delle operazioni risultano essere molto più elevati, quest'ultimo rappresenta uno dei **colli di bottiglia** dell'architettura di **Von Neumann.** Per ridurre questo **Ritardo Strutturale** nell'accesso alla **DRAM,** é possibile utilizzare una RAM a tecnologia statica, **SRAM** dove i **microcondenzatori** sono sostituiti da micro **Flip-Flop,** tutta via questa memoria ha un **costo** più **elevato** e richiede più **spazio**, per questo motivo viene utilizzata solo in **memoria cache** per le comunicazioni tra **memoria principale** e **processore.**

ROM Nella memoria troviamo inoltre una zona di **memoria speciale** che non perde mai i valori dopo lo spegnimento. Quest'area è fondamentale poichè permette la corretta accensione del nostro elaboratore, quando clicchiamo il tasto di accensione il computer recupera le istruzioni base da questa memoria per avviare le sue periferiche (tastiera,monitor,mouse, etc) e verificare che tutto funzioni, questa verifica prende il nome di **POST** (Power On Self Test) dopo questa fase di verifica il sistema carica il sistema operativo per avviarlo e permetterci di utilizzarlo. Questa memoria **ROM** proprio per il fatto che il suo contenuto non viene cancellata, viene utilizzata dal **BIOS** (Basic Input Output System), il BIOS ha il compito di configurare l'hardware, ovvero assicurarsi che ogni componente funzioni correttamente, i programmi di questa memoria prendono il nome di **Firmware,** e non possono essere facilmente modificate, tuttavia esistono altre tipologie di memorie **ROM** chiamate **EPROM** o **EEPROM**, che permettono di aggiornare modificare o

eliminare il proprio contenuto attraverso delle tecniche specifiche, come ad esempio per le **EPROM** l'uso di luci ultraviolette per eliminare i dati delle celle di memoria e andare a riscriverne il contenuto, mentre le **EEPROM** fanno uso di segnali elettrici per andare ad eliminare e riscriverne il contenuto direttamente dal sistema in cui é collegata.

Questi componenti comunicano tra loro attraverso un canale di interconnessione chiamato **BUS**

BUS Il bus é un sistema di collegamento che permette ai vari componenti del nostro elaboratore come (memoria,cpu ed i dispositivi di input/output) di comunicare tra loro, é composto da un insieme di linee, ogni linea può trasportare solo un singolo bit ovvero (0 o 1) per rappresentare un segnale alto si utilizza 1 mentre per quelli bassi 0. Tuttavia nonostante sia composto da diverse linee, solo un dispositivo alla volta può trasferire dati, la porta logica che permette a un dispositivo di inviare i dati su una delle linee del bus prende il nome di **bus driver** e si occupa di attivare la linea del bus per permettere il trasferimento dati, i vari componenti sono collegati al bus tramite delle porte **tri-state,** queste porte possono essere in tre stati, **attivo,** quando un dispositivo deve inviare dati, la porta è attiva ed il trasferimento è consentito, **disattivo,** la porta è disattivata e non interferisce con il traffico sul bus, questo perché il componente non ne richiede l'uso oppure in **alta impedenza,** in questo stato la porta non ha alcun impatto sul bus, come se fosse spenta.

Abbiamo più tipologie di **BUS** ovvero, **Address Bus (ABus)** ha il compito di trasportare gli indirizzi di memoria esso viene utilizzato dal processore per **identificare** la locazione di memoria a cui accedere, **Data Bus (DBus)** trasporta i dati che devono essere **letti** o **scritti** dalla memoria o dai dispositivi I/O, **Controll Bus (CBus)** trasporta i segnali di comando, ovvero specifica cosa fare con i dati e gli indirizzi, ad esempio se l'operazione é completata o se deve leggere o scrivere.

È importante sapere che il numero di linee **dell'address bus** o del **data bus** non é sempre strettamente legato al numero del processore, ad esempio per l'architettura a **32 bit** abbiamo un **ABus** a **32 linee** che può indirizzare fino a **4gb** di memoria e un **DBus** a **32 linee** che permettono di trasferire fino a **32 bit** ovvero **4 byte,** mentre nelle architetture a **64 bit,** l'address bus **ABus** ha **64 linee** che permettono di traferire fino ad un massimo circa di **16 exabyte** di memoria che equivalgono **16 miliardi di gigabyte** e il data bus **DBus** ha **64 linee** e permette di trasferire **64 bit** ovvero **8 byte** di dati per singola operazione.

Per garantire che tutti i trasferimenti avvengano nel modo corretto e privi di errori, il **CBus** utilizza alcune linee speciali che svolgono funzioni specifiche, una linea indica chi é il **destinatario** dell'operazione ovvero se i dati da trasferire partono dalla **memoria** ai dispositivi **I/O** o viceversa, un'altra linea specifica se l'operazione é di **lettura** o **scrittura** indicando se il processore sta **inviando (0)** o **prelevando (1)** i dati, infine un'altra linea segnala se il trasferimento dei dati é **concluso** evitando così di passare ad una nuova operazione senza prima aver completato quella attuale. Un metodo per migliorare le prestazioni del bus e ridurre i rallentamenti è quello di utilizzare tre bus per gestire la comunicazione tra le diverse componenti, l'idea base è qurlla di separare i flussi di dati in modo da evitare congestioni e garantire un flusso ordinato, supponiamo di utilizzare un **bus A ,** un **bus B** e un **bus C, bus A** e **bus B** per i dati delle sorgenti delle operazioni ovvero i dati che provengono dalla memoria o dai registri, necessari per eseguire le operazioni, il **bus C,** invece è dedicato solamente ai risultati delle operazioni, quindi i dati che devono essere inviati ai registri o alla memoria, separando così il flusso dei dati in ingresso (A,B) da quello dei dati in uscita (C), in questo modo il sistema riuscirà a gestire meglio i trasferimenti, evitando così i conflitti tra i diversi flussi di dati. Un altro elemento importante è il **generatore di indirizzo di istruzione,** il quale è direttamente connesso al **Program Counter,** il program counter è un registro tiene traccia dell'indirizzo dell'**istruzione** che il processore deve eseguire successivamente, il generatore di indirizzo si occupa di **prelevare** l'indirizzo fornito dal **program counter** e usarlo per localizzare le istruzioni nella memoria, questo processo è essenziale affinchè il processore carichi la giusta istruzione al momento giusto, infine il **blocco di controllo** è il componente che **gestisce l'esecuzione delle istruzioni,** esso legge l'istruzione corrente dall'

Istruction Register (**IR**) e decide quale istruzione deve essere eseguita, coordinando i vari componenti del processore, in base alla complessità dell'istruzione, il blocco di controllo determinerà il numero di **passi** necessari per completare l'esecuzione, ovviamente maggiore è la complessità maggiore risulterà il numero di passi da effettuare

Problema dei filosofi a cena Il **Problema dei filosofi a cena** rappresenta una situazione tipica in cui piú processi cercano di condividere una risorsa limitata (come la **CPU** o la **Memoria**). Per capirlo a fondo possiamo immaginare quattro filosofi seduti a tavola, davanti a loro ci sono dei piatti di cibo, ma hanno solo una forchetta. I filosofi hanno solo tre alternative ovvero, parlano, mangiano e dormono quindi solo chi ha la forchetta può mangiare gli altri parlano o dormono, se un filosofo tra di loro é egoista e tiene la forchetta per sé, gli altri filosofi finiranno per morire di (starvation) fame, questo é un problema che può verificarsi facilmente nei computer quando alcuni processori bloccano l'accesso delle risorse impedendo così ad altri processi di accedervi.

Per risolvere questo problema é importante gestire correttamente le risorse del sistema ed organizzare le priorità tra i processi, per far ciò il Bus utilizza "orologio" detto Clock che ha il compito di regolare la velocità delle operazioni.

All'interno della Motherboard di un elaboratore non é semplice definire con certezza la posizione del bus di sistema poiché esso é contenuto all'interno di un insieme **Chipset**. Questo chipset é costruito in base alle caratteristiche specifiche del processore, e gestisce la comunicazione tra i diversi componenti garantendo che il tutto funzioni correttamente.

Con il passare degli anni le tecnologie con cui viene realizzato il bus sul sistema classico intel si sono evolute siamo passati dallo **Standard ISA** (DBus a 8 bit e clock a 8.33 MHz) allo **Standard EISA** (DBus a 16 bit e clock 8.33 MHz) per poi passare allo **Standard PCI** (Peripheral Component Interconnect, DBus a 32 bit e clock a 33 MHz) fino ad arrivare al **PCI Express** (DBus 32-64 bit e clock a 66 MHz), con il passare del tempo sono stati sviluppati BUS interni in dedicati appositamente al trasferimento di dati delle **GPU** e **CPU,** partendo dallo **Standard Vesa** fino ad arrivare al più moderno **AGP,** il BUS **PCI** è fisicamente più lungo rispetto al BUS **AGP** poiché ha un architettura che gli permette di gestire una quantità di dati maggiore.

Parallelismo Asincrono e Sincrono Questo miglioramento é utile in particolare nelle applicazioni ad alte prestazioni che fanno uso di molta potenza di calcolo, in particolare sfruttano la **GPU,** le moderne gpu infatti sfruttano il **Parallelismo Asincrono** che permette di elaborare grandi quantità di informazioni in maniera efficiente e a differenza del **Parallelismo Sincrono** in cui i processi devono coordinarsi e scambiarsi informazioni continuamente, nel **Parallelismo Asincrono** ogni processo può lavorare in maniera indipendente senza dover attendere il completamento degli altri processi, inoltre le **GPU** con più **core** possono suddividere il lavoro come ad esempio il rendering di immagini in molteplici micro-operazioni parallele senza che debbano necessariamente sincronizzate tra loro.

Finora abbiamo parlato dei dati e delle istruzioni come elementi che possiamo manipolare sotto forma di input e output, adesso invece, entriamo nel concetto in maniera più specifica analizzando nel dettaglio il loro ruolo all'interno del sistema e le carratteristiche.

I/O La sezione Input/Output ha il compito di acquisire dati e della loro rappresentazione in diverse forme, come Output, che può essere visivo, ovvero su monitor o su carta in caso di stampanti, oppure su sistemi di memorizzazione secondari (**SSD,Hardisk,Nas).** Concettualmente la sezione di **I/O** può essere vista come un **insieme di celle** simile alla memoria principale, ma ha uno spazio di indirizzamento ridotto riservato solo agli indirizzi **I/O**, ogni dispositivo dispone di un proprio **range** di indirizzi chiamati anche registri di **I/O** o porte di **I/O**, avvolte se si necessita di grandi quantità di memoria si utilizzano gli indirizzi di memoria al posto di quelli di **I/O** in questo caso si parla di **Mappatura in memoria.** Oltre a

questa modalità, la sezione **I/O** dispone anche della **Gestione delle interruzioni,** questo grazie al **CBus** che implementa due segnali principali che sono, **INTR** che richiede l'interruzione e **INTA** che conferma l'avvenuta presa in carico, quando viene generata un interruzione viene richiesto al processore di **sospendere momentaneamente** la sua **esecuzione** per gestire l'esecuzione di una parte di programma che lo riguarda sotto forma di procedura associata a quell'interruzione, questo prende il nome di **ISR** (Interrupt Service Routine), questo accade quando avviene un **interruzione senza** alcun **preavviso** come ad esempio il movimento improvviso del mouse o l'input da tastiera. Per evitare di sovraccaricare il **processore,** alcuni sistemi fanno si che si possano trasferire i dati direttamente dalla memoria al dispositivo e viceversa, senza coinvolgere il processore, questo processo prende il nome di **DMA** (Direct Memory Access) e aiuta a trasferire grandi quantità di dati velocemente. Ogni dispositivo che si collega ad un computer ha una **scheda controller**, ovvero una parte che ha il compito di gestire le comunicazioni con il sistema in particolare appunto l'invio e la ricezione dei dati, prima dell'introduzione della tecnologia **PCI,** ogni dispositivo doveva essere configurato manualmente tramite lo spostamento di **ponticelli modificabili** sulla scheda madre, per impostare **valori specifici** come ad esempio, la configurazione degli indirizzi di I/O, il numero di interruzione in modo da evitare che due dispositivi possano comunicare contemporaneamente con la CPU e impostare i valori del **DMA** poiché ogni dispositivo deve avere un canale **DMA** separato per evitare **conflitti di attribuzione**. Inoltre, il Bus **PCI** ci permette di usare la tecnologia **Plug&Play (PnP)** che permette al **bios**, al **sistema operativo** e al **firmware,** di assegnare linee e indirizzi in maniera automatica a ciascun dispositivo, quando il computer si avvia oppure nel momento in cui un dispositivo viene installato così facendo l'utente non deve effettuare **configurazioni manuali. Plug&Play**,appunto significa collega e usa ed indica dispositivi che non hanno bisogno di configurazioni, al giorno d'oggi ormai quasi tutte le periferiche sono Plug&Play. Le schede di rete aggiuntive invece si collegano al bus tramite appositi slot come gli **slot PCI** nel quale si possono inserire le schede di espansione ma solamente a computer spento, mentre le connessioni moderne come **USB 2.0** che trasferisce dati fino a 480 Mbit/s ****e **Firewire** 400 Mbit/s, supportano il collegamento a caldo (**Hot Swap)** permettendo di collegare e scollegare i dispositivi mentre il computer é acceso e inoltre vengono alimentati dalla porta stesso quindi non richiedono un alimentatore esterno. Queste nuove tecnologie fanno si che le vecchie porte seriali **RS232 e PS/2** stanno pian piano scomparendo, sostituite da USB e altre tecnologie più moderne.

Fino ad ora abbiamo parlato di linguaggi di programmazione in generale. Ora, entrando più nello specifico, possiamo approfondire i concetti di **assembly** e **assembler**.

Quando si parla di **Assembly** ci riferiamo ad un linguaggio di programmazione più vicino al computer. perchè lavora quasi direttamente con le proprie istruzioni proprio per questo motivo non è molto semplice da utilizzare. Il programmatore scrive le istruzioni in assembly ma queste da sole non sono ancora pronte per essere eseguite dalla macchina, infatti si ha bisogno di un programma specifico ovvero l'**assembler** che si occupa di tradurre il codice assembly in **sequenze binarie** ovvero 0 e 1, infatti queste sequenza sono il linguaggio nativo del computer, l'unico che riesce a comprendere ed eseguire.

Per capire meglio è importante comprendere la differenza tra linguaggi **compilati** e linguaggi **interpretati,** nei linguaggi **compilati** il programma viene tradotto interamente in linguaggio macchina da un programma chiamato **compilatore,** questo lo rende più **veloce** poiché il computer lavora in maniera diretta sul codice tradotto come ad esempio **C**, **C++** e **Visual Basic,** nei linguaggi **interpretrati** invece, il programma viene tradotto riga per riga durante l'esecuzione da un programma chiamato **interprete,** questo li rende più **portabili**, perché per ogni piattaforma non hanno bisogno di essere ricompilati, tra questi troviamo **JavaScript, PHP** e **Python.** Esiste però un caso speciale ovvero **Java** che è un linguaggio **ibrido** cioè combina le caratteristiche sia dei linguaggi **compilati** che di quelli **interpretati.** La traduzione del codice avviene in **tre** passaggi, il programma inizialmente scritto in un linguaggio di alto livello come (**C,C++**) viene tradotto dal compilatore in **assembly,** a questo punto, il file **assembly** viene passato all'**assemblatore** che lo converte in

file **oggetto** questo file é scritto in codice binario, quindi leggibile dal computer ma ancora non può essere eseguito da solo, infine il **linker** collega tutti i **file oggetto** prodotti dal **compilatore** e **assemblatore,** insieme ad eventuali **librerie** esterne necessarie, creando un unico **file eseguibile** pronto per essere avviato. L'assemblatore é fondamentale per tradurre il linguaggio **Assembly** in **codice binario** eseguibile, può capitare però di trovarsi davanti a situazioni come ad esempio un etichetta (l'etichetta é un nome associato a un indirizzo di memoria) che viene utilizzata prima di essere **dichiarata**, in questo caso si utilizza il metodo **dell'assemblatore** a **due passi**, ovvero al primo passo l'assemblatore **analizza l'intero codice sorgente** e **identifica** i **simboli** e li **inserisce** in una **tabella** dove associa a ciascun nome il valore numerico corrispondente (ad esempio gli **indirizzi** delle **etichette**), una volta completata la tabella, l'assemblatore sostituisce i simboli presenti nel codice con i rispettivi valori numerici e genera il **codice oggetto definitivo** pronto per essere eseguito. I **vantaggi** sono, la **gestione accurata dei simboli** come detto prima grazie alla tabella dei simboli**,** la **modularità** questo metodo rende il processo di assemblaggio più **modulare**, separando la parte di analisi (creazione della tabella) con quella di sostituzione e generazione del codice oggetto, rendendo così il processo più strutturato e gestibile,un altro vantaggio é appunto il fatto che leggendo due volte il codice é possibile rilevare con **maggiore precisione** gli errori, questo processo facilita il debug e la risoluzione di problemi, gli **svantaggi** sono, la sua **lentezza** perché leggendo il codice in due fasi l'assemblatore deve eseguire due letture del codice, il che lo rende piú lento rispetto ad un assemblatore che non usa il metodo a due passi, un altro svantaggio é che **richiede più memoria,** a causa della tabella di simboli che contiene tutte le informazioni come nomi, valori numeri e indirizzi che vanno a occupare una quantità maggiore di memoria, oltre ad essere più lento durante la fase di assemblaggio, l'uso della tabella dei simboli e delle due letture **richiede più tempo** durante la fase di **esecuzione** soprattutto se il codice é molto **grande** e **complesso.** Per completare il processo di traduzione di un programma in un file eseguibile entra in scena il **Linker,** dopo che il codice sorgente è stato compilato e assemblato in file oggetto quest'ultimo spesso non è ancora autonomo questo accade perchè possono essere presenti riferimenti a funzioni o variabili dichiarate in altri file o librerie esterne. Il compito del **Linker** è quello di risolvere questi riferimenti e combinare i diversi file oggetto in un unico file eseguibile completo ad esempio se il programma utilizza librerie standard come **printf** in **C**, il **Linker** si assicura che il file

****Dopo che l'assemblatore completa il suo lavoro, genera il **codice oggetto,** un file binario che contiene il programma tradotto in istruzioni comprensibili dalla macchina, quest'ultimo da solo non può essere eseguito ed entra in gioco il **Loader,** che ha il compito di trasferire il **programma oggetto** dalla memoria secondaria ad esempio il disco, alla memoria **centrale RAM,** dove potrà essere eseguito dal processore, il **loader legge** il programma per determinarne la **lunghezza** e i **l'indirizzo** di caricamento in memoria, una volta ottenute queste informazioni si passa alla vera e propria fase di **caricamento,** in questa fase le istruzioni e i dati del programma vengono trasferiti dal **disco** alla memoria **RAM rispettando** i **vincoli** di allocazione indicati nel file oggetto, infine il **loader** conclude il suo compito saltando alla **prima istruzione del programma,** iniziando così la fase di esecuzione.

Esistono più tipologie di **loader,** come il **loader statico,** esso è il più semplice, questo tipo di loader va a caricare l'intero programma, comprese tutte le istruzioni e dati, direttamente in memoria prima che inizi l'esecuzione, esso però richiede una quantità significativa di memoria e non offre flessibilità, il **loader dinamico** invece, è più sofisticato, invece di caricare tutto il programma in una volta, trasferisce solo i **moduli** e le **librerie** necessarie al momento del loro utilizzo, durante l'esecuzione del programma, questa strategia consente un uso più efficiente della memoria e permette di gestire programmi più complessi su sistemi con risorse limitate, infine abbiamo il **bootstrap loader,** un programma speciale memorizzato nella **ROM** che entra in azione all'avvio del computer, il suo compito è quello di avviare il sistema operativo preparandolo per essere caricato in memoria, esso è essenziale poiché è il primo programma ad essere eseguito permettendo il corretto funzionamento del nostro sistema.

Le **Librerie** sono insiemi di file che contengono funzioni e sottoprogrammi pronti per l'uso, possono essere utilizzati da altri programmi, come una sotta di kit che rende più semplice la costruzione del nostro programma evitando di scrivere tutto da 0. Per la creazione di queste librerie viene utilizzato un programma chiamato **Archiver,** che raccoglie i vari file necessari e li organizza in un unico file libreria, questo file contiene le informazioni utili al **Linker** per collegare le parti del programma con le funzioni e i sottoprogrammi della libreria.

Il **Debugger** è uno strumento fondamentale per identificare e correggere gli errori nei programmi, permettendo di eseguire il programma in modo controllato e fermarlo quando vogliamo per analizzare cosa sta succedendo e se tutto funzioni come previsto, in particolare ci aiuta a identificare **Bug,** cioè errori di logica o funzionamento che il **compilatore** non può rilevare, con il debugger possiamo osservare lo stato della memoria, dei registri e delle variabili del programma durante l'esecuzione e può essere utilizzato in due modalità, **Trace Mode,** in questa modalità il programma verrà eseguito **un istruzione per volta** dopo ogni passo l'istruzione si **interrompe** e possiamo analizzare lo stato del programma in quel esatto momento, una volta effettuati i controlli necessari possiamo decidere di procedere con l'esecuzione indicando la nostra intenzione al debugger che allora rientrerà dall'istruzione e passerà all'istruzione successiva, un altro tipo di debugger è il **Breakpoint,** in questa modalità possiamo definire **dei punti specifici** del codice chiamati appunto **Breakpoint** dove vogliamo che l'esecuzione si **fermi**, infatti quando un programma raggiunge uno di questi punti si **interromperà automaticamente** sostituendo temporaneamente l'istruzione con un **comando** di **interruzione** chiamato **trap** , in modo tale che possiamo controllare cosa sta accadendo fino a quel esatto momento, quando comunichiamo l'intenzione di voler riprendere la normale esecuzione, il **debugger** sostituirà la **trap** con il contenuto originale dell'istruzione.

Il **Sistema operativo (SO)** è il software più importante del computer, ha il compito di coordinare e gestire tutte le sue attività, è come il cervello del computer, si assicura che tutto funzioni in maniera corretta, ad esempio fa si che più programmi possano essere eseguiti contemporaneamente, gestisce i dispositivi di I/O (tastiera, mouse, stampante, ecc), controlla lo stato della memoria e permette all'utente di poter interagire con il sistema attraverso l'interfaccia utente (**GUI-Graphical User Interface**). Il **SO** è composto da due parti principali, **routine essenziali,** presenti in memoria centrale e sono fondamentali perchè permettono al sistema operativo di funzionare senza interruzioni, queste routine prendono il nome di **Kernel,** tra queste troviamo **gestione della memoria,** questa funzione si occupa di amministrare la memoria centrale del computer assicurandosi che il programma utilizzi solo la memoria necessaria e che una volta chiuso la memoria venga liberata per altri programmi, evitando così **conflitti** e **crash,** un altra funzione è la **gestione dei processi** che ha il compito di gestire l'esecuzione dei programmi, dando a ciascun programma il tempo necessario per lavorare senza interferenze, esistono anche altre routine come quella per la **gestione degli I/O , gestione degli errori e gestione dei file system e gestione dei programmi,** tutto ciò compone la prima parte del **SO,** la seconda invece contiene i **programmi di utilità** ovvero degli strumenti speciali che non sono sempre attivi ma vengono avviati solo quando ne abbiamo bisogno per svolgere compiti specifici, ad esempio la **gestione di file** o la **deframmentazione del disco,** ovvero, con il **tempo** i file sul nostro elaboratore tendono a diventare **sparsi** in **piccole parti** divise sul disco, questo rende il sistema più lento, i programmi di deframmentazione **gestiscono** queste situazioni **riorganizzando** questi file in maniera che siano tutti vicini e **facili** da **raggiungere,** esistono anche altri programmi di utilità come gli **antivirus** per la protezione del sistema e programmi di **backup dei dati** che ci permette di recuperare i dati in caso di mal funzionamento del nostro sistema effettuandone una copia.

Nel caso della gestione di un programma, quando un programma deve leggere i dati dal disco per poi stamparli, il sistema operativo si occupa di coordinare il processore e le operazioni di I/O in modo tale che il programma possa eseguire le sue operazioni senza bloccare il funzionamento del sistema, ad esempio se il programma richiede una lettura del disco il sistema operativo può decidere se effettuare un salto ed eseguire un altra parte di programma in modo tale

che il sistema non rimanga bloccato in operazioni lente come la lettura, questo si chiama **multitasking** ovvero un meccanismo che permette a più programmi di lavorare nello stesso momento per far ciò sfrutta un sistema di interruzioni che possono essere **Hardware,** provengono da dispositivi fisici come tastiere o dischi che richiamano l'attenzione del processore oppure interruzioni **software,** quest'ultime vengono generate dai prorammi stessi o dal sistema operativo. Ad ogni interruzione è associato una **routine di servizio** un piccolo programma progettato per rispondere alle interruzioni, per esempio se l'interruzione è generata dalla tastiera avvierà una routine di servizio che legge il tasto premuto, questo avviene per ogni tipo di interruzione e per sapere quale interruzione avviare il sistema utilizza una struttura denominata **vettore di interruzione,** una tabella che associa ogni tipo di interruzione a un indirizzo di memoria, come se fosse un elenco che collega un determinato tipo di interruzione alla propria risposta appropriata. Per gestire più programmi contemporaneamente, il sistema utilizza una tecnica chiamata **time slicing,** in cui l'esecuzione di ogni programma viene suddivisa in **quanti di tempo** indicati con il simbolo τ (tau), un contatore di sistema (timer) tiene conto di questi quanti di tempo, quando il tempo assegnato all'esecuzione di un determinato programma termina, il timer genera un interruzione che avvia la routine **scheduler** il quale si occuperà di quale programma eseguire successivamente, basandosi su criteri come la priorità. I programmi in **multitasking** possono trovarsi in tre stati diversi che sono, **Running** quando il programma è attualmente in fase di esecuzione sul processore,**Runnable** se il programma è pronto per essere eseguito ma sta aspettando il proprio turno perchè il processore è occupato con un altro programma ed in fine **Blocked** il programma sta attendendo un evento esterno come ad esempio il completamento di un'operazione I/O (come ad esempio il completamento di una stampa)

L'**algebra booleana della commutazione** è un sistema matematico che si occupa di variabili che possono assumere solo due stati 1 (Vero) o 0 (Falso), quest'ultima é utilizzata in informatica ed elettronica per rappresentare e risolvere problemi logici e progettare circuiti digitali, le **operazioni fondamentali** sono, **AND** (Prodotto Logico es. A x B), Questa operazione restituisce **1** solo quando **tutte** le **variabili** in **ingresso** sono **1** se anche solo una di queste variabili è **0** allora il risultato sarà **0, OR** (Somma Logica es A + B) questa operazione restituisce **1** solo quando almeno una delle due variabili è **1** in caso contrario il risultato sarà **0, NOT** (Complementazione A) questo operatore inverte l'ordine della variabile in ingresso se è **1** l'uscita sarà **0,** e viceversa.

L'operazione logica **OR** gode delle seguenti proprietà, proprietà commutativa, proprietà associativa, può essere applicata a più variabili ed ha 0 come elemento neutro, la funzione viene scritta nel seguente modo:

$$ f(x_1, x_2) = x_1 + x_2 = x_1 \vee x_2 $$

L'operazione logica **AND** gode delle seguenti proprietà, proprietà commutativa, proprietà associativa, può essere applicata a più variabili ed ha **1** come **elemento neutro,** la funzione è scritta nel seguente modo:

$$ f(x_1, x_2) = x_1 \cdot x_2 = x_1 \land x_2 $$

L'operazione logica **NOT** gode della seguente proprietà, **inversione del valore logico,** quest'ultima non si applica a più variabili ma agisce su una singola variabile invertendone il valore, la funzione viene scritta nel seguente modo:

$$ f(x) = \overline{x} \\\text{Dove:} \\\text{Se } x = 1, \text{ allora } \overline{x} = 0. \\\text{Se } x = 0, \text{ allora } \overline{x} = 1. $$

Il **minitermine** é un concetto fondamentale nell'algebra booleana, che descrive una **combinazione unica** di variabili che porta ad una logica a valere **1** solo per quella determinata configurazione, quindi varrà **1** solo se tutte le variabili di ingresso sono in una determinata posizione, in tutti gli tutti gli altri casi la funzione varrà **0**

Per comprendere meglio vediamo un **espressione generale,** considerando una funzione booleana con **n** variabili (x1,x2,…,x**n**), ogni possibile combinazione di valori delle variabili rappresenta un **minitermine** ad esempio se abbiamo due variabili x1 e x2 le possibili combinazioni sarebbero:

\[\begin{aligned} m_0 &= \overline{x_1} \, \overline{x_2} \quad & \text{(quando } x_1 = 0, x_2 = 0) \\ m_1 &= \overline{x_1} \, x_2 \quad & \text{(quando } x_1 = 0, x_2 = 1) \\ m_2 &= x_1 \, \overline{x_2} \quad & \text{(quando } x_1 = 1, x_2 = 0) \\ m_3 &= x_1 \, x_2 \quad & \text{(quando } x_1 = 1, x_2 = 1) \end{aligned} \]

La somma logica di tutti i mintermini corrispondenti ai casi in cui la funzione assume il valore **1** fornisce la funzione desiderata.

Il **maxtermine** è una funzione booleana che assume il valore **0** per una specifica configurazione delle variabili e **1** in tutti gli altri casi. È possibile rappresentare ogni maxtermine tramite una combinazione logica delle variabili binarie.

Per una funzione booleana, i maxtermini che valgono **0** possono essere combinati per ottenere la funzione desiderata mediante il **prodotto logico (AND)** dei maxtermini.

Ad esempio, se la funzione vale **0** per le configurazioni \(M_2,M_4,M_5,M_6\) si ha:

\[f_1 = \=M_2∧M_4∧M_5∧M_6 \]

I **maxtermini** sono utili per rappresentare una funzione nella **Forma Normale Congiuntiva (CNF o POS)**. I **mintermini** rappresentano configurazioni per cui la funzione assume il valore **1**, e la loro combinazione avviene mediante **somma logica (OR)**: $f_ 1

=m_ 0

∨m_ 1

∨m_ 3

∨m_ 7$

I **maxtermini** rappresentano configurazioni per cui la funzione assume il valore **0**, e la loro combinazione avviene mediante **prodotto logico (AND)**: **$f_ 1

=M_ 2

∧M_ 4

∧M_ 5

∧M_ 6$**

Questi strumenti permettono di rappresentare una funzione in due forme equivalenti:

- **Disjunctive Normal Form (DNF)** o **SOP (Sum of Products)** per i **mintermini**.
- **Conjunctive Normal Form (CNF)** o **POS (Product of Sums)** per i **maxtermini**.

Una funzione booleana si dice essere in **forma minima** quando non esiste un'espressione equivalente che abbia un **costo inferiore**, ovvero con un numero minore di **letterali** (variabili o loro negazioni). Le espressioni in forma minima sono vantaggiose poiché risultano più concise, facilitando la loro comprensione e interpretazione. Inoltre, sono più efficienti da realizzare come circuiti digitali, poiché riducono il numero di porte logiche necessarie, rendendo così l'implementazione più economica e ottimizzata. Per questi motivi, le espressioni in forma minima sono particolarmente utili per ottimizzare circuiti e risolvere problemi logici in modo efficiente.

Le operazioni logiche di base, come AND, OR, NOT e XOR, sono fondamentali nell'algebra booleana e trovano ampio impiego nell'informatica e nell'elettronica digitale.

L'operazione **AND** restituisce un valore vero (1) solo quando entrambi gli ingressi sono veri. Viene rappresentata dalla simbolizzazione booleana $f=x1 \cdot x2$, dove l'operazione è realizzata solo quando entrambe le variabili in ingresso sono 1.

L'operazione **OR**, invece, restituisce un valore vero se almeno uno degli ingressi è vero. La sua espressione booleana è $f=x1+x2$, indicando che la funzione è vera anche quando uno solo dei due ingressi è 1.

L'operazione **NOT** inverte il valore dell'ingresso: se l'ingresso è 1, l'uscita sarà 0, e viceversa. La sua rappresentazione booleana è $f=\neg x1f$, e agisce su una singola variabile, modificandone il valore.

Un'altra operazione logica importante è l'**XOR** (Exclusive OR), che restituisce 1 solo se i due ingressi sono diversi. In altre parole, l'XOR è vero quando uno e solo uno degli ingressi è vero. La sua espressione booleana può essere scritta come $f=x1 \oplus x2$, ed è equivalente a $f=x1' \cdot x2+x1 \cdot x2'$, dove il simbolo $\oplus$ denota l'operazione XOR e l'apice indica la negazione. La tabella di verità dell'XOR evidenzia il suo comportamento, che restituisce 1 quando gli ingressi sono 01 o 10, e 0 quando entrambi sono 00 o 11.

Le **porte NAND** e **NOR** sono varianti delle operazioni AND e OR. In particolare, la porta **NAND** è la negazione della porta AND e restituisce 0 solo quando entrambi gli ingressi sono 1, mentre la porta **NOR** è la negazione della porta OR e restituisce 1 solo quando entrambi gli ingressi sono 0. Le loro espressioni booleane sono:

- **NAND**: $f=x1 \uparrow x2=\neg(x1 \cdot x2)$
- **NOR**: $f=x1 \downarrow x2=\neg(x1+x2)$

Le porte NAND e NOR sono considerate "porte universali" in elettronica, poiché possono essere utilizzate per realizzare qualsiasi altra funzione logica, compreso l'XOR. Questo è possibile grazie alle leggi dell'algebra booleana, come le leggi di De Morgan, che permettono di esprimere qualsiasi funzione booleana tramite combinazioni di porte NAND o NOR.

Le **leggi di De Morgan** e la **legge dell'involuzione** sono fondamentali per comprendere come trasformare qualsiasi espressione booleana in una forma che utilizzi solo le operazioni NAND o NOR. Ad esempio, a partire da una forma normale disgiuntiva (DNF), che rappresenta una funzione booleana come somma di prodotti, si può applicare la legge di De Morgan per ottenere una rete di porte NAND.

La porta **NAND** a n ingressi è un'estensione della porta NAND a due ingressi e viene progettata per gestire un numero arbitrario di variabili di input. Il suo funzionamento segue una regola semplice: l'uscita sarà 0 solo quando tutti gli ingressi sono pari a 1. In tutti gli altri casi, l'uscita sarà 1. Questa caratteristica la rende particolarmente utile in situazioni che richiedono l'elaborazione simultanea di più segnali di input.

Un **addizionatore a un bit** è un circuito logico progettato per sommare due bit, a e b, tenendo conto del riporto in ingresso. Esso produce due risultati: il valore della somma, che rappresenta il bit meno significativo, e il bit di riporto in uscita. Quest'ultimo, posizionato a sinistra, viene trasferito come riporto in ingresso all'addizionatore successivo. Questo processo di somma continua fino a completare tutte le operazioni di somma necessarie.

Un addizionatore può essere realizzato utilizzando porte logiche, in particolare due porte XOR. La prima porta calcola a⊕b, mentre la seconda determina la somma finale sss, combinando (a⊕b) con cin, ossia il riporto in ingresso proveniente dalla somma precedente. Questo meccanismo è fondamentale nella costruzione di circuiti complessi, come le ALU (Unità Logiche e Aritmetiche), che costituiscono componenti essenziali dei processori.

L'addizionatore completo, o **full adder**, è un circuito in grado di sommare due bit aaa e bbb, insieme a un eventuale riporto in ingresso cin. Il circuito produce due uscite: la **somma**, che rappresenta il risultato dell'operazione, e il **riporto in uscita**, che può essere trasferito come ingresso all'addizionatore successivo.

L'addizionatore a propagazione di riporto, noto come **carry lookahead adder**, è un circuito avanzato progettato per aumentare la velocità delle operazioni di somma. Questo risultato viene ottenuto riducendo i tempi di propagazione del riporto, migliorando così le prestazioni complessive del calcolo.

L'addizionatore tradizionale calcola la somma in modo sequenziale, elaborando i riporti uno alla volta e passando da un bit all'altro. Questo approccio diventa sempre più lento all'aumentare del numero di bit, come 16, 32, 64 e oltre.

Un **addizionatore a propagazione di riporto** adotta un metodo più efficiente rispetto ai circuiti di somma tradizionali, sfruttando due concetti fondamentali: la **generazione del riporto** e la **propagazione del riporto**. La generazione del riporto avviene direttamente quando entrambi gli ingressi sono pari a 1, mentre la propagazione del riporto si verifica quando il riporto deve essere sommato ai bit successivi. In questo caso, se almeno uno degli ingressi è 1 e il riporto è anch'esso 1, si genera un ulteriore riporto. Questo approccio consente di calcolare i riporti in parallelo, migliorando così le performance complessive rispetto ai metodi tradizionali, dove i riporti venivano calcolati in sequenza.

L'idea alla base dell'addizionatore a propagazione di riporto è calcolare i riporti in anticipo utilizzando i valori di **generazione** (G) e **propagazione** (P). Questo consente di evitare il ritardo dovuto al passaggio sequenziale dei riporti attraverso le porte logiche.

La realizzazione di questo circuito sfrutta una rete di porte logiche, come AND, OR e XOR, che calcolano i segnali di P e G, permettendo di determinare il riporto per ogni bit della somma in modo rapido ed efficiente. Questo tipo di addizionatore è comunemente utilizzato nei circuiti ad alte prestazioni, come le ALU dei processori più avanzati.

Trabocco: si verifica un trabocco quando il riporto supera la capacità massima di rappresentazione del numero, andando oltre la larghezza del registro. Ad esempio, in un'architettura a 64 bit, un riporto generato al 64º bit causa un trabocco, poiché non può essere gestito dal registro.

La gestione del trabocco viene effettuata utilizzando un addizionatore algebrico a n-bit, progettato per rilevare e trattare questi casi.

L'addizionatore algebrico a n-bit è un circuito logico progettato per sommare due numeri binari a n bit, gestendo anche il segno dei numeri. Questo consente l'addizione di numeri interi con segno, rappresentati in complemento a 2.

In questa rappresentazione, i numeri positivi sono espressi come numeri binari standard, mentre i numeri negativi vengono ottenuti tramite il complemento a 2. Questo processo consiste nell'invertire tutti i bit del numero e aggiungere 1 al risultato. Il bit più significativo (MSB) è utilizzato per indicare il segno del numero: se è 0, il numero è positivo; se è 1, il numero è negativo. Dopo aver convertito i numeri in complemento a 2, la somma viene eseguita utilizzando un circuito simile a un addizionatore a n-bit, ma con logica aggiuntiva per gestire il trabocco.

L'addizionatore algebrico a n-bit è progettato per gestire non solo la somma dei numeri, ma anche il riporto, il segno e il trabocco. Le sue uscite principali sono la somma, che fornisce n bit corrispondenti ai n bit del risultato della somma, e il carry-out, che rappresenta il riporto finale. Il carry-out è fondamentale per determinare se si è verificato un overflow. Questo tipo di addizionatore è essenziale nei sistemi digitali, poiché consente di sommare correttamente numeri con segno, garantendo l'affidabilità e l'efficienza dei calcoli.

Il ritardo in un circuito è causato dal percorso più lento, determinato da un componente che opera a velocità inferiori rispetto agli altri. Per gestire il ritardo totale, viene utilizzato un **Ripple Carry Adder (RCA)**.

L'RCA è un tipo di addizionatore che concatena addizionatori a 1 bit (o completi) per eseguire somme bit per bit. Il termine *ripple* si riferisce alla propagazione del riporto attraverso i vari bit, che avviene in modo simile a un'onda che si diffonde nel circuito. Questo schema, sebbene semplice e facile da implementare, può risultare lento per operazioni su numeri con un alto numero di bit, a causa della propagazione sequenziale del riporto.

I componenti principali dell'**RCA** (Ripple Carry Adder) sono gli **addizionatori a 1 bit** e gli **addizionatori completi**. Gli addizionatori a 1 bit sommano due bit insieme al riporto in ingresso, generando un riporto in uscita. Gli addizionatori completi, invece, passano i riporti agli addizionatori a 1 bit successivi, ripetendo il processo bit per bit fino a ottenere la somma finale.

Il ritardo dell'RCA è dovuto alla **propagazione del riporto**, poiché ogni riporto deve essere calcolato e propagato al bit successivo. Questo crea una **dipendenza di dato**, in cui ogni bit dipende dal riporto calcolato precedentemente, rallentando l'intero processo di somma.

Il risultato di ciascun Full Adder dipende dal riporto calcolato dal Full Adder nella posizione anteriore. Fattorizzando l'equazione del riporto, si ottengono le funzioni di generazione (G) e di propagazione (P), che dipendono solo dagli ingressi e possono essere calcolate in parallelo in un ritardo di porta. La parallelizzazione del calcolo del riporto consente di ridurre il ritardo causato dalla propagazione sequenziale.

Ogni riporto può essere calcolato in funzione degli addendi e del riporto in ingresso, espandendo iterativamente fino a ottenere un'equazione in cui il riporto dipende solo dai vari ingressi e dai riporti precedenti. In questo modo, ciascun riporto può essere calcolato in parallelo dopo due ritardi di porta, migliorando l'efficienza del calcolo.

La cella di un sommatore a 1 bit, o Full Adder bit cell, può essere modificata per fornire anche le funzioni di generazione e propagazione. Queste funzioni vengono ottenute rispettivamente da una porta AND per G e una porta XOR per P. Inoltre, il bit di somma (s) è ottenuto da due porte XOR annidate.

Un addizionatore con anticipo di riporto a 4 bit è progettato per ridurre il ritardo causato dall'operazione di somma. L'anticipo del riporto permette di calcolare i riporti in anticipo per tutti i bit, evitando che ogni bit dipenda dal dato del riporto precedente. La logica di anticipo del riporto genera i riporti in parallelo, riducendo il tempo necessario per ottenere il risultato finale.

Esistono anche addizionatori con anticipo di riporto a due livelli, che riducono ulteriormente il tempo di calcolo rispetto ai CLA tradizionali. In questi addizionatori, i calcoli di generazione e propagazione vengono eseguiti su due livelli logici distinti. Tuttavia, nei casi in cui il numero di celle supera le quattro, si può incorrere in valori di fan-in troppo elevati nelle porte di generazione dei riporti, il che rallenta il circuito. Per risolvere questo problema, si può replicare l'idea di anticipo di riporto su più livelli, migliorando l'efficienza. Ad esempio, per ottenere un addizionatore a 16 bit con anticipo di riporto, si possono collegare quattro addizionatori a 4 bit con un blocco di anticipo di riporto che genera i riporti in parallelo.

Nel caso della moltiplicazione di numeri senza segno, il processo è simile a quello che si apprende a scuola, con la moltiplicazione di ciascuna cifra del moltiplicatore e la somma dei risultati, spostati di una posizione. La moltiplicazione binaria coinvolge solo l'uso di 1 e 0, con il prodotto che dipende dalle singole cifre del moltiplicando e del moltiplicatore.

Il circuito moltiplicatore sequenziale sfrutta un processo algoritmico che ottiene risultati parziali, che vengono successivamente salvati nei registri. Questo tipo di moltiplicatore ha il vantaggio di ridurre la complessità hardware, ma richiede più tempo per ottenere il risultato rispetto ai moltiplicatori paralleli.

L’algoritmo funziona così:

A = 0 Q = Moltiplicatore M = Moltiplicando

Per n cicli: se q0 = 1: A = A + M altrimenti: A = A + 0

c = riporto c || A || Q scorrono verso destra di una posizione

Dopo n cicli i due registri A || Q concatenati conterranno il prodotto finale.

La moltiplicazione sequenziale di numeri senza segno può essere realizzata utilizzando un addizionatore a bit e due registri di scorrimento. Ad ogni ciclo, l'addizionatore somma il moltiplicando (o un array di zeri) con un prodotto parziale fatto scorrere a destra.

Per la moltiplicazione di numeri con segno in complemento a due, l'algoritmo di moltiplicazione deve essere modificato. Nel caso di moltiplicatori e moltiplicandi negativi, si applicano le tecniche di estensione del segno e complemento a due. Un approccio più generale ed efficiente per la moltiplicazione con segno è l'algoritmo di Booth, che ricodifica il moltiplicatore come somma e sottrazione di potenze di 2.

La ricodifica Bit-pair, una tecnica utilizzata per comprimere la sequenza di bit, può essere applicata per ottenere una versione più efficiente dell'algoritmo di Booth. Raggruppando i bit in coppie (bit-pairs), si riduce la lunghezza della sequenza senza perdere informazioni, ottimizzando il numero di addizioni necessarie durante la moltiplicazione.

Infine, la divisione tra interi segue un processo simile a quello della moltiplicazione. Si allinea il divisore con il dividendo a partire da sinistra, si calcola la divisione parziale tra le cifre allineate e si trascrive il resto. Questo processo continua fino a completare la divisione.

La *memory word* rappresenta la dimensione dell'unità di dato che un sistema di elaborazione può gestire e memorizzare in un singolo ciclo di clock della CPU. Un vantaggio principale di una *memory word* più grande è che consente una gestione più rapida dei dati, permettendo alla CPU di elaborare più informazioni contemporaneamente, migliorando le prestazioni, soprattutto in applicazioni complesse. Inoltre, semplifica l'elaborazione di strutture dati complesse, riducendo la necessità di operazioni aggiuntive.

Tuttavia, un svantaggio di una *memory word* più grande è il maggiore consumo di energia, poiché la CPU potrebbe richiedere più potenza per gestire operazioni più complesse in un singolo ciclo. Inoltre, se la memoria o la larghezza di banda non sono adeguate a supportare una parola di memoria più grande, si potrebbero verificare colli di bottiglia che limitano le prestazioni del sistema. La scelta della dimensione ideale dipende quindi dal bilanciamento tra prestazioni, consumo energetico e costi.

L'organizzazione della memoria nei computer si basa su un modello strutturato in cui l'informazione viene immagazzinata sotto forma di un vettore di *word*. Ogni *word* rappresenta un'unità di dati, a cui è associato un indirizzo binario univoco che ne consente l'identificazione. La capacità di rappresentare gli indirizzi dipende dal numero di bit del sistema, poiché un numero binario con un determinato numero di bit può definire un certo intervallo di indirizzi.

Le *word* successive nel vettore sono mappate su indirizzi consecutivi, e l'insieme degli indirizzi associati a tutte le *word* disponibili costituisce lo spazio di indirizzamento. Questa struttura organizzativa è essenziale nell'architettura di un computer, poiché permette di trattare la memoria come una sequenza lineare di unità di dati, facilitando l'accesso e la gestione delle informazioni.

L'indirizzamento e l'ordinamento dei byte costituiscono elementi centrali nell'organizzazione della memoria nei sistemi informatici. Nei computer, l'unità minima di informazione che può essere indirizzata è generalmente il byte, e a ciascun byte contenuto in una parola vengono assegnati indirizzi consecutivi. Questo implica che gli indirizzi delle parole risultano essere multipli della loro lunghezza in byte, garantendo una gestione strutturata e coerente dei dati all'interno della memoria.

Un aspetto importante legato a questa organizzazione è l'ordinamento dei byte, conosciuto come *endianness*. Questo termine descrive il modo in cui i byte all'interno di una parola sono disposti e indirizzati. Vi sono due modalità principali di *endianness*: il formato big-endian e quello little-endian. Nel big-endian, l'indirizzo associato ai byte cresce man mano che si passa dal byte più significativo, cioè quello con il peso aritmetico maggiore, a quelli meno significativi. Al contrario, nel formato little-endian, l'indirizzo aumenta seguendo l'ordine opposto, cioè dal byte meno significativo al più significativo.

Queste differenze nell'organizzazione dei byte influenzano il modo in cui i dati vengono interpretati e trasferiti tra sistemi diversi, rendendo l'endianness una scelta architetturale cruciale per la compatibilità e il funzionamento corretto dei programmi e delle interfacce hardware.

L'Instruction Set Architecture (ISA) rappresenta l'insieme di istruzioni che un processore è in grado di eseguire e le modalità con cui tali istruzioni vengono utilizzate. Questo insieme costituisce il linguaggio macchina specifico per ogni processore e funge da punto di interfaccia tra il software e l'hardware. Ogni processore commerciale dispone del proprio ISA, che ne determina le capacità operative e il modo in cui interagisce con il sistema.

All'interno di un ISA, ogni istruzione è identificata da un codice univoco chiamato *Operation Code* (Op. Code), che consente al processore di riconoscere e interpretare l'operazione da eseguire. Ogni istruzione è inoltre accompagnata da un numero definito di parametri, detti operandi, che insieme all'Op. Code ne determinano la lunghezza complessiva in byte. Per rendere più comprensibile la programmazione a livello umano, a ciascun Op. Code viene associata una descrizione mnemonica che semplifica la comprensione dell'istruzione. Ad esempio, l'istruzione di somma può essere rappresentata dal mnemonico "ADD", permettendo ai programmatori di riconoscerla immediatamente come un'operazione di somma, anche se il processore la interpreta come un codice binario specifico, ad esempio "01".

L'ISA non solo definisce le istruzioni, ma anche l'approccio architetturale adottato dai processori. Esistono due principali tipi di ISA: CISC (Complex Instruction Set Computing) e RISC (Reduced Instruction Set Computing). I processori CISC sono progettati per supportare un ampio set di istruzioni, spesso complesse, capaci di eseguire operazioni articolate con una sola istruzione. Al contrario, i processori RISC adottano un approccio più essenziale, con un set di istruzioni ridotto ma ottimizzato per l'efficienza e la velocità.

Per facilitare l'interazione con il linguaggio macchina, si utilizza il linguaggio assemblativo, una rappresentazione simbolica più leggibile per gli esseri umani. Questo linguaggio traduce i codici binari in simboli mnemonici comprensibili, che vengono poi convertiti in linguaggio macchina tramite un assemblatore. Poiché ogni ISA è progettata in base a caratteristiche specifiche del processore, esistono assemblatori differenti che riflettono le particolarità architetturali di ciascun tipo di processore, garantendo un funzionamento corretto e una gestione ottimale delle istruzioni.

Nel linguaggio assemblativo generico, è importante disporre di una notazione chiara per identificare registri e locazioni di memoria, elementi fondamentali per il funzionamento del processore. I registri, che si trovano all'interno del processore, vengono identificati attraverso nomi specifici. Esistono registri generici, come R0, R1, fino a Rn, utilizzati per operazioni di base; registri speciali, come il Program Counter (PC) e l'Instruction Register (IR), che gestiscono il flusso delle istruzioni; e registri di input/output, come INGRESSO_DATO e USCITA_DATO, che facilitano lo scambio di informazioni tra il processore e i dispositivi esterni.

Le locazioni di memoria, invece, vengono identificate attraverso indirizzi. Questi possono essere indicati come costanti numeriche o simboliche, ad esempio VAR1, IND o CICLO, definite in precedenza per facilitare la programmazione.

Un aspetto spesso discusso è la differenza tra registri e cache. I registri vengono utilizzati per memorizzare dati temporanei, come gli operandi di una somma o i risultati di operazioni aritmetiche e logiche. La cache, invece, è una memoria ad alta velocità posizionata tra il processore e la memoria principale. Il suo scopo è ridurre i tempi di accesso ai dati utilizzati più frequentemente, evitando il ricorso continuo alla DRAM, che è più lenta. Sebbene i registri siano più veloci, la cache ha una capacità maggiore ed è suddivisa in tre livelli: L1, L2 e L3. La cache L1 è la più veloce ma meno capiente, mentre la L3 offre maggiore capacità a scapito della velocità.

Per quanto riguarda le istruzioni di base per l'accesso alla memoria, esistono comandi fondamentali che permettono di gestire dati tra registri e memoria. L'istruzione **Load** consente di caricare un dato dalla memoria in un registro, specificando il registro come destinazione e la locazione di memoria come sorgente. Al contrario, l'istruzione **Store** viene usata per salvare un dato da un registro in una locazione di memoria, definendo il registro come sorgente e la memoria come destinazione.

Le operazioni aritmetiche di base includono comandi come **Add** e **Subtract**. L'istruzione **Add** permette di sommare il contenuto di due registri e memorizzare il risultato in un terzo registro specificato come destinazione. Analogamente, l'istruzione **Subtract** esegue la sottrazione tra i valori di due registri e salva il risultato in un registro di destinazione.

Un esempio semplice per comprendere la somma in linguaggio Assembly potrebbe essere il seguente: supponiamo di voler sommare due numeri, ad esempio 5 e 3, e salvare il risultato in un registro. Ecco come potrebbe essere scritto il codice Assembly:

```
LOAD R1, 5       ; Carica il numero 5 nel registro R1
LOAD R2, 3       ; Carica il numero 3 nel registro R2
ADD R3, R1, R2   ; Somma il contenuto di R1 e R2, salva il risultato in R3
STORE R3, RISULTATO ; Salva il contenuto di R3 nella locazione di memoria "RISULTATO"
```

Nel linguaggio assemblativo, operandi e risultati possono essere specificati attraverso diversi metodi, chiamati **modi di indirizzamento**. Questi modi rappresentano le diverse modalità con cui si indicano gli indirizzi delle informazioni necessarie per eseguire un'istruzione. Le architetture RISC, ad esempio, supportano vari modi di indirizzamento, ognuno dei quali è progettato per specifiche esigenze operative.

Uno dei più comuni è il **modo di registro**, in cui il nome o l'indirizzo di un registro del processore che contiene l'operando o il risultato è specificato direttamente nell'istruzione. Questo metodo è veloce e utilizzato per operazioni che coinvolgono dati già presenti nei registri. Al contrario, nel **modo assoluto** (o diretto), l'indirizzo di una locazione di memoria contenente l'operando o il risultato è dato esplicitamente nell'istruzione stessa. Ad esempio, l'istruzione `Load R2, NUM1` carica il valore memorizzato all'indirizzo "NUM1" nella memoria nel registro R2.

Un'altra modalità interessante è il **modo immediato**, in cui l'operando non è memorizzato in un registro o in una locazione di memoria, ma è indicato direttamente nell'istruzione come un valore costante preceduto dal simbolo cancelletto (**#**). Ad esempio, un'operazione di somma potrebbe aggiungere il valore costante 200 al contenuto di un registro, come nell'istruzione `Add R4, #200`.

Nel **modo indiretto da registro**, invece, l'istruzione non contiene direttamente l'indirizzo dell'operando, ma il nome di un registro che, a sua volta, contiene l'indirizzo in memoria. Questo metodo, rappresentato con il nome del registro tra parentesi tonde (ad esempio, `(R6)`), è particolarmente utile quando si desidera riutilizzare la stessa istruzione cambiando dinamicamente gli operandi.

Un ulteriore metodo è il **modo con indice e spiazzamento**, in cui l'indirizzo effettivo dell'operando o del risultato è calcolato sommando un valore costante, detto spiazzamento, al contenuto di un registro. Questo è indicato con una sintassi del tipo `X(Ri)`, dove `X` è lo spiazzamento e `Ri` è il registro contenente l'indirizzo base. Questo modo di indirizzamento si dimostra particolarmente utile nella gestione di strutture dati come vettori o liste, in cui è necessario accedere a elementi consecutivi o in posizioni relative.

L'istruzione di salto condizionato, chiamata **Branch_if**, è una funzione essenziale per il controllo del flusso in un programma. In pratica, consente al programma di eseguire un'operazione in un punto diverso del codice, ma solo se una determinata condizione risulta vera. Questa condizione può essere basata sui valori contenuti nei registri del processore, indicati con una notazione come `[Ri]`, oppure su valori esplicitamente definiti nel codice.

Quando la condizione è soddisfatta, il salto porta il programma a una destinazione specifica, che corrisponde a un'istruzione memorizzata in un'altra posizione della memoria. Ad esempio, si può utilizzare un'istruzione di salto per tornare a un punto precedente in un ciclo, come nel caso in cui si voglia ripetere un'operazione finché il contenuto di un registro, come `R2`, è maggiore di zero. In questo scenario, il programma salta a un'etichetta denominata, ad esempio, `CICLO`, ogni volta che la condizione è verificata, continuando invece in modo lineare quando la condizione non è più valida.

L'assemblatore è uno strumento fondamentale per la traduzione del codice sorgente scritto in linguaggio assemblativo in codice macchina binario, noto come programma oggetto. Oltre alla semplice traduzione delle istruzioni, l'assemblatore deve risolvere alcuni problemi essenziali per il corretto funzionamento del programma. In particolare, deve assegnare valori numerici a nomi e simboli definiti nel codice, determinare la posizione in memoria delle istruzioni macchina e allocare lo spazio necessario per gli operandi e i risultati del programma.

Per assistere l'assemblatore in questi compiti, il linguaggio assemblativo include non solo istruzioni eseguibili, ma anche comandi specifici chiamati **direttive di assemblatore**. Queste direttive non vengono tradotte in codice macchina, ma forniscono indicazioni utili per la gestione del programma durante il processo di assemblaggio.

Una delle direttive più comuni è la **dichiarazione di eguaglianza**, utilizzata per associare un valore numerico a un nome simbolico nel programma sorgente. La sua sintassi è:

```
NOME EQU Valore_numerico
```

Ad esempio, questa direttiva consente di utilizzare nomi significativi al posto di valori numerici, migliorando la leggibilità del codice.

Un'altra direttiva importante è la **ORIGIN**, che indica all'assemblatore l'indirizzo di memoria in cui iniziare a collocare le istruzioni e i dati definiti nelle righe successive del programma. La sintassi è:

```
ORIGIN Indirizzo_di_memoria
```

Per riservare spazio in memoria, si utilizza la direttiva **RESERVE**, che specifica all'assemblatore di allocare una quantità definita di spazio, espressa in byte. La sintassi è:

```
RESERVE Spazio_in_byte
```

Questa direttiva è utile, ad esempio, per allocare aree di memoria destinate a dati che verranno definiti o utilizzati durante l'esecuzione del programma.

Infine, la direttiva **DATAWORD** consente di riservare una word in memoria e di assegnarle un contenuto specifico. La sua sintassi è:

```
DATAWORD Contenuto_da_assegnare
```

Questa direttiva è spesso impiegata per inizializzare valori che il programma utilizzerà durante la sua esecuzione.

Una linea di codice assemblativo generalmente si compone di quattro campi principali: etichetta, operazione, operandi e commento.

L'**etichetta** è un nome associato all'indirizzo di memoria a cui è assegnata un'istruzione o un blocco di memoria riservato. Sebbene l'etichetta sia facoltativa, viene utilizzata per identificare facilmente una particolare posizione in memoria, rendendo il codice più leggibile e organizzato.

L'**operazione** rappresenta il nome dell'istruzione che deve essere eseguita dal processore o una direttiva di assemblatore. Le operazioni possono essere istruzioni di base, come `ADD` o `MOV`, oppure direttive che influenzano il processo di assemblaggio, come `EQU` o `ORIGIN`.

Gli **operandi** forniscono informazioni di indirizzamento che indicano come accedere agli operandi coinvolti nell'istruzione. Gli operandi possono riferirsi a registri, locazioni di memoria o valori immediati, a seconda del tipo di istruzione e del modo di indirizzamento utilizzato.

Infine, il **commento** è una parte della linea di codice che viene ignorata dall'assemblatore, ma che serve per fornire spiegazioni o chiarimenti sul funzionamento del codice. I commenti sono essenziali per la comprensione del codice da parte degli sviluppatori, ma non influiscono sull'esecuzione del programma.

Per quanto riguarda la **notazione dei numeri**, l'assemblatore supporta diverse rappresentazioni numeriche: binaria, decimale ed esadecimale. Per distinguere tra questi formati, vengono utilizzati specifici prefissi. Il prefisso `%` indica un numero in **binario**, i numeri **decimali** non hanno prefisso (ad esempio `25`), mentre il prefisso `0X` è utilizzato per i numeri in **esadecimale** (ad esempio `0X1A`). Questi prefissi aiutano a chiarire il formato numerico e a evitare ambiguità nel codice.

Un esempio che mostra come usare il **modo immediato** in linguaggio assemblativo con numeri in **binario**, **decimale** ed **esadecimale**:

- **Decimale**: `ADD R2, R1, #5`
- **Binario**: `ADD R2, R1, %101`
- **Esadecimale**: `ADD R2, R1, 0xF`

La **stack** è una struttura dati che segue il principio **LIFO** (Last In, First Out), dove l'ultimo elemento inserito è il primo a essere rimosso. Un esempio pratico di questo comportamento potrebbe essere la pila di piatti in un ristorante: l'ultimo piatto che viene messo in cima alla pila sarà il primo a essere tolto. Un concetto fondamentale legato alla gestione della stack è lo **Stack Pointer (SP)**, un registro che punta sempre alla cima della stack, ovvero all'indirizzo di memoria in cui si trova l'elemento più recentemente aggiunto. La stack è strutturata in modo che gli indirizzi degli elementi siano disposti in ordine decrescente, partendo dalla base della stack.

Quando si eseguono operazioni sulla stack, come l'aggiunta o la rimozione di elementi, si interagisce direttamente con lo Stack Pointer. Quando un nuovo elemento viene aggiunto, il valore dello Stack Pointer diminuisce, poiché il nuovo elemento viene posto in una locazione di memoria inferiore, più vicina alla base della stack. Al contrario, quando un elemento viene rimosso, il valore dello Stack Pointer aumenta, portando la cima della stack a puntare a una locazione di memoria superiore.

Per aggiungere un elemento alla stack, in un'architettura **RISC**, si utilizza una sequenza di istruzioni. La prima operazione consiste nel **decrementare** il valore dello Stack Pointer per "spostare" la cima della stack verso un indirizzo di memoria inferiore, mentre la seconda operazione consiste nello **scrivere** il valore desiderato nella locazione di memoria appena indicata dallo Stack Pointer aggiornato. Ad esempio, se vogliamo aggiungere il valore contenuto nel registro Rj, la sequenza di istruzioni potrebbe essere la seguente: prima diminuiamo il valore di SP di 4 byte (equivalente a una parola di memoria) con l'istruzione `Subtract SP, SP, #4`, quindi memorizziamo il contenuto del registro Rj nella locazione di memoria puntata da SP con `Store Rj, SP`.

Per rimuovere un elemento dalla stack, si utilizza l'operazione di **Pop**, che prevede due fasi. La prima consiste nel **caricare** il valore dalla locazione di memoria puntata dallo Stack Pointer in un registro del processore, mentre la seconda operazione consiste nel **incrementare** il valore dello Stack Pointer, spostando la cima della stack verso un indirizzo di

memoria superiore. In pratica, se vogliamo prelevare il valore dalla cima della stack e memorizzarlo nel registro Rj, la sequenza di istruzioni sarà: prima carichiamo il valore da SP con `Load Rj, SP`, quindi aumentiamo il valore di SP di 4 byte con `Add SP, SP, #4` per "spostare" la cima della stack al nuovo indirizzo.

Un sottoprogramma, noto anche come routine, è una sequenza di istruzioni che svolge un compito specifico e può essere richiamato in qualsiasi momento durante l'esecuzione di un programma. Per eseguire una routine, un programma utilizza un'istruzione di chiamata, o **call instruction**, che permette di saltare al blocco di codice del sottoprogramma. Quando la routine ha completato la sua esecuzione, è necessario tornare al punto successivo nel programma principale, e questo avviene mediante l'istruzione di ritorno, o **return instruction**, che permette di riprendere l'esecuzione dal punto in cui il sottoprogramma è stato chiamato.

Il meccanismo di chiamata a un sottoprogramma prevede che, prima di saltare all'indirizzo della routine, il programma salvi l'indirizzo dell'istruzione successiva (che è l'indirizzo a cui il programma deve tornare una volta terminato il sottoprogramma) nel **link register**, un registro speciale. Questo permette di memorizzare l'indirizzo di ritorno in modo sicuro.

Quando viene eseguita l'istruzione di chiamata, il programma esegue due passi: prima salva l'indirizzo dell'istruzione successiva (contenuto nel registro **PC**, Program Counter) nel link register, e poi salta all'indirizzo specificato nell'istruzione di chiamata, che corrisponde all'inizio del sottoprogramma. Una volta che la routine ha completato il suo compito, l'istruzione di ritorno interviene, ripristinando il valore del Program Counter dal link register, consentendo così al programma di riprendere l'esecuzione dal punto in cui era stato interrotto prima della chiamata al sottoprogramma. Questo meccanismo di collegamento e rientro consente di riutilizzare blocchi di codice, facilitando il riuso e la modularità all'interno di un programma.

Nei circuiti elettronici, i valori binari 0 e 1 sono rappresentati utilizzando la tensione elettrica, una grandezza continua che deve essere discretizzata per garantire una chiara distinzione tra i due stati logici. Questo avviene attraverso l'uso di una soglia di separazione, un valore di riferimento che divide i livelli di tensione in due categorie.

Tutte le tensioni superiori alla soglia sono interpretate come valore logico 1, mentre quelle inferiori sono associate al valore logico 0. Tuttavia, per evitare problemi legati al rumore del circuito, che potrebbe introdurre incertezze nei valori prossimi alla soglia, viene definita una "banda vietata". Questa zona intorno alla soglia rappresenta un intervallo di tensioni che non viene preso in considerazione, eliminando così le possibili ambiguità nella lettura dei dati.

Questo approccio garantisce l'affidabilità del sistema, riducendo al minimo gli errori dovuti a fluttuazioni o interferenze nei segnali elettrici.

I transistor sono dispositivi semiconduttori fondamentali nei circuiti integrati, impiegati per amplificare segnali elettronici o per commutare correnti, ossia attivare e disattivare il flusso di elettricità in un circuito. Essi sono realizzati con materiali semiconduttori come il silicio, che viene trattato attraverso un processo di drogaggio, ovvero l'introduzione di impurità selezionate. Questo trattamento crea regioni con caratteristiche elettriche differenti: regioni a carica positiva, dette di tipo *p*, e regioni a carica negativa, dette di tipo *n*.

Le due principali tipologie di transistor si distinguono per il loro principio di funzionamento. Il transistor bipolare, conosciuto anche come BJT (Bipolar Junction Transistor), regola il flusso di corrente tra due terminali, chiamati collettore ed emettitore, grazie a una terza corrente di controllo nota come corrente di base. Questo tipo di transistor è particolarmente efficiente nel controllare correnti di alta intensità ed è ampiamente utilizzato nei circuiti di amplificazione.

Il transistor a effetto di campo, o FET (Field Effect Transistor), opera invece secondo un principio differente: la corrente che fluisce tra due terminali, denominati *drain* e *source*, è regolata dalla tensione applicata a un terzo terminale chiamato *gate*. Questo controllo basato sulla tensione rende il FET particolarmente adatto per applicazioni a basso consumo energetico, come quelle nei dispositivi elettronici portatili o nei microprocessori.

I transistor possono essere impiegati in due modalità principali: come interruttori, per permettere o interrompere il flusso di corrente in un circuito, o come amplificatori, per incrementare l'ampiezza di un segnale elettrico. Grazie a queste proprietà versatili, i transistor rappresentano il cuore della microelettronica e sono presenti in una vasta gamma di applicazioni, dai dispositivi più semplici ai sistemi elettronici complessi.

Nella tecnologia dei transistor, in particolare quella a metallo-ossido-semiconduttore (MOS), il funzionamento del dispositivo dipende dalla tensione applicata ai terminali. In base al valore della tensione di ingresso, il transistor può trovarsi in uno stato di conduzione, in cui permette il passaggio di corrente, oppure in uno stato di interdizione, in cui blocca il flusso di corrente.

I valori tipici di tensione utilizzati nella tecnologia MOS variano a seconda della progettazione del circuito. Ad esempio, in alcuni casi, la tensione di alimentazione (*Vcc*) è pari a 5 Volt, con una tensione di soglia (*Vsoglia*) fissata a 2,5 Volt. In altri contesti, la tensione di alimentazione può essere ridotta a 3,3 Volt, con una corrispondente tensione di soglia pari a 1,5 Volt.

Questi valori rappresentano configurazioni standard nella progettazione dei circuiti digitali e garantiscono il corretto funzionamento del transistor, permettendo una chiara distinzione tra gli stati logici 0 e 1. La scelta delle tensioni dipende dalle esigenze del sistema e dal livello di efficienza energetica richiesto, oltre che dalla compatibilità con altri componenti del circuito.

I transistor MOS (Metal-Oxide-Semiconductor) sono dispositivi a tre terminali: il gate (base), il sink (pozzo) e il source (sorgente). Il loro funzionamento dipende dalla tensione applicata al gate, che controlla la conduzione tra il pozzo e la sorgente.

Quando viene applicata una tensione al gate, essa determina se il transistor sarà in stato di conduzione o meno. Se la tensione al gate è sufficiente per superare una certa soglia, il transistor entra in conduzione, permettendo così il passaggio della corrente dal pozzo alla sorgente. In questa condizione, la tensione nel pozzo diventa praticamente uguale alla tensione nella sorgente, poiché il transistor "collega" direttamente i due terminali. Se la tensione applicata al gate è inferiore alla soglia, il transistor non entra in conduzione e la connessione tra pozzo e sorgente rimane interrotta.

Questo comportamento consente al transistor MOS di agire come un interruttore elettronico, controllando il flusso di corrente in base alla tensione applicata al gate.

Esistono due principali tipi di transistor MOS: NMOS e PMOS, che si differenziano principalmente per il comportamento elettrico e la loro configurazione.

Nei transistor **NMOS**, la tensione applicata al gate determina il loro stato di conduzione. Quando la tensione al gate è alta, il transistor entra in conduzione, permettendo il flusso di corrente tra la sorgente e il pozzo. Al contrario, quando la tensione al gate è bassa, il transistor entra in stato di interdizione, interrompendo il flusso di corrente. Nei transistor NMOS, la sorgente è collegata alla massa, e queste configurazioni sono vantaggiose per la loro maggiore velocità di commutazione e per la resistenza relativamente bassa, il che le rende ideali per applicazioni ad alta velocità.

Nei transistor **PMOS**, il comportamento si inverte rispetto ai NMOS. Quando la tensione al gate è alta, il transistor si trova in stato di interdizione e non consente il passaggio della corrente. Quando la tensione al gate è bassa, il transistor entra in conduzione, permettendo il flusso di corrente. Nei transistor PMOS, la sorgente è collegata all'alimentazione. Questi transistor sono particolarmente utili per le applicazioni in cui è necessario un basso consumo energetico, poiché consumano poca energia quando sono in stato di riposo o inattivi.

Entrambi i transistor vengono utilizzati per il controllo del flusso di corrente, succesivamente, entrambi i sistemi formano il CMOS.

Un circuito **NOT**, o **inverter**, può essere realizzato utilizzando un transistor **NMOS**. La configurazione di base prevede il collegamento della sorgente del transistor alla massa, mentre il pozzo è collegato all'alimentazione tramite una resistenza.

Nel funzionamento di questo circuito, la tensione applicata al gate (base) del transistor determina lo stato di conduzione del dispositivo. Quando la tensione di ingresso al gate è alta (logico "1"), il transistor entra in conduzione e collega direttamente il pozzo alla massa, riducendo la tensione di uscita nel pozzo a zero (logico "0"). Al contrario, quando la tensione al gate è bassa (logico "0"), il transistor entra in stato di interdizione, interrompendo il flusso di corrente tra il pozzo e la massa. In questa condizione, la resistenza al collegamento tra il pozzo e l'alimentazione permette di far salire la tensione di uscita nel pozzo a un livello alto (logico "1").

Questo comportamento, dove l'uscita è l'opposto dell'ingresso, è il principio di funzionamento di una porta **NOT**, che viene anche chiamata **inverter**. La sua funzione è quindi quella di invertire il valore logico del segnale di ingresso, fornendo l'opposto come uscita.

Un circuito **NOR** può essere ottenuto collegando due transistor **NMOS** in parallelo. In questo caso, la tensione di uscita dipende dallo stato di conduzione di entrambi i transistor. Quando i segnali di ingresso ai gate dei transistor sono bassi (logico "0"), entrambi i transistor si trovano in stato di interdizione, ovvero non permettono il passaggio di corrente. In questa condizione, la tensione di uscita sarà alta (logico "1"), poiché la resistenza collegata al pozzo consente alla tensione di salire fino al livello dell'alimentazione.

Tuttavia, se uno dei due transistor è in conduzione (cioè se uno dei segnali di ingresso è alto, logico "1"), il circuito si chiude, e la corrente fluisce dal pozzo verso la massa, portando la tensione di uscita a basso livello (logico "0").

Questo tipo di circuito è molto utilizzato nelle memorie digitali, poiché la sua logica può essere applicata a vari tipi di architetture per la memorizzazione e il recupero di dati.

Un circuito **NAND** può essere realizzato collegando due transistor **NMOS** in serie. Il funzionamento di questa porta dipende dalla tensione applicata ai segnali di ingresso e dal comportamento di ciascun transistor.

Nel circuito NAND, la tensione di uscita sarà bassa (logico "0") solo quando entrambi i transistor sono in conduzione, cioè quando entrambe le tensioni di ingresso sono alte (logico "1"). In questa condizione, la corrente può fluire attraverso entrambi i transistor, creando una connessione tra il pozzo e la massa, e portando la tensione di uscita a livello basso.

Al contrario, in tutti gli altri casi, almeno uno dei transistor sarà in stato di interdizione (quando uno degli ingressi è basso, logico "0"), impedendo alla corrente di fluire attraverso il circuito. In queste situazioni, la tensione di uscita sarà alta (logico "1"), grazie alla resistenza che collega il pozzo all'alimentazione.

La tecnologia **CMOS** (Complementary Metal-Oxide-Semiconductor) è stata sviluppata per risolvere il problema del consumo energetico elevato dei transistor **NMOS** quando sono in stato di conduzione. In un transistor NMOS, la corrente che scorre tra il pozzo e la sorgente provoca dissipazione di potenza a causa della resistenza del dispositivo, soprattutto quando il transistor è acceso. Questo problema viene superato dalla combinazione dei transistor **NMOS** e **PMOS** in un circuito CMOS.

In un circuito CMOS, i transistor NMOS e PMOS sono configurati in due rami separati, che sono collegati in modo complementare. Quando uno dei transistor è in conduzione, l'altro è in interdizione, il che significa che non c'è mai un percorso diretto di corrente tra la massa e l'alimentazione, evitando così il consumo continuo di energia. Questo comportamento garantisce che la potenza venga dissipata solo durante le fasi di commutazione, quando i transistori passano da uno stato all'altro.

I vantaggi principali della tecnologia CMOS sono molteplici. In primo luogo, il **consumo di potenza** è molto ridotto, poiché la corrente viene consumata solo durante la commutazione tra gli stati di "1" e "0". La **potenza dissipata** è proporzionale alla **frequenza di commutazione**, il che significa che il consumo energetico cresce solo quando il circuito opera a frequenze elevate.

Inoltre, i **transistor MOS** utilizzati nella tecnologia CMOS sono molto **piccoli**, permettendo di integrare miliardi di transistor in un singolo chip. Questa miniaturizzazione consente una maggiore densità di circuiti, portando a dispositivi più potenti e compatti. Le piccole dimensioni dei transistor consentono anche un'**alta frequenza di commutazione**, che può raggiungere valori nell'ordine dei gigahertz (GHz), rendendo la tecnologia CMOS particolarmente adatta per applicazioni ad alte prestazioni, come i microprocessori e le memorie.

Una **porta NOT** basata sulla tecnologia **CMOS** è realizzata collegando un transistor **NMOS** in serie con un transistor **PMOS**, con entrambi i transistor che condividono la stessa tensione di ingresso al gate (base). Questo tipo di configurazione sfrutta il comportamento complementare dei due transistor, che garantisce un basso consumo di energia e un funzionamento efficiente.

Nel funzionamento di questa porta, quando la tensione di ingresso è alta (logico "1"), il transistor **NMOS** entra in conduzione, mentre il transistor **PMOS** entra in stato di interdizione. In questa condizione, la corrente scorre attraverso il transistor NMOS, e la tensione di uscita sarà bassa (logico "0").

Al contrario, quando la tensione di ingresso è bassa (logico "0"), il transistor **NMOS** è in stato di interdizione, mentre il transistor **PMOS** entra in conduzione. In questa condizione, la corrente scorre attraverso il transistor PMOS, e la tensione di uscita sarà alta (logico "1").

Grazie a questa configurazione, in ogni momento uno dei due transistor è in conduzione e l'altro è in interdizione, evitando la continuità tra l'alimentazione e la massa, questo comportamento riduce il consumo di potenza, poiché la corrente fluisce solo durante le transizioni di stato, ovvero quando il circuito cambia tra un valore logico "1" e un valore logico "0".

Una **porta NAND** basata sulla tecnologia **CMOS** è realizzata utilizzando una combinazione di transistor **NMOS** e **PMOS**, ma con configurazioni differenti per i due tipi di transistor. Nel ramo **NMOS**, i due transistor sono collegati **in serie**, mentre nel ramo **PMOS** i transistor sono collegati **in parallelo**.

Nel ramo **NMOS**, i due transistor in serie garantiscono che la corrente possa fluire verso la massa (ground) solo quando entrambi i segnali di ingresso sono alti (logico "1"). In questo caso, entrambi i transistor **NMOS** sono in conduzione, creando un percorso diretto tra l'alimentazione e la massa, il che fa abbassare la tensione di uscita a "0". In tutti gli altri casi, almeno uno dei due transistor **NMOS** è in stato di interdizione, e la corrente non può fluire verso la massa.

Nel ramo **PMOS**, invece, i due transistor sono collegati in parallelo. Ciò significa che la corrente può fluire attraverso uno dei transistor **PMOS** se uno dei segnali di ingresso è basso (logico "0"). In questo caso, la tensione di uscita sarà alta (logico "1"). Se entrambi i segnali di ingresso sono alti, entrambi i transistor **PMOS** sono in stato di interdizione, impedendo la conduzione e facendo scendere la tensione di uscita a "0".

La logica della porta **NAND** è tale che l'uscita sarà bassa (logico "0") solo quando entrambi gli ingressi sono alti (logico "1"). In tutti gli altri casi, l'uscita sarà alta (logico "1"). Questa combinazione di transistor **NMOS** e **PMOS** in configurazione CMOS consente di ottenere un'alta efficienza energetica e una bassa dissipazione di potenza, in quanto la corrente fluisce solo durante la commutazione degli stati logici.

Una **porta NOR** basata sulla tecnologia **CMOS** è realizzata con una configurazione in cui il ramo **NMOS** presenta due transistor collegati **in parallelo**, mentre il ramo **PMOS** presenta due transistor collegati **in serie**. In questa configurazione, la logica della porta NOR prevede che l'uscita sia alta (logico "1") solo quando entrambi gli ingressi sono bassi (logico "0"). In tutti gli altri casi, quando almeno uno degli ingressi è alto (logico "1"), l'uscita sarà bassa (logico "0"). La disposizione dei transistor in parallelo per il ramo **NMOS** e in serie per il ramo **PMOS** permette di ottenere questa logica, riducendo anche il consumo di potenza grazie al comportamento complementare dei transistor.

Per quanto riguarda la **porta AND** basata sulla tecnologia CMOS, essa può essere ottenuta collegando una porta **NOT** all'uscita di una porta **NAND**. Questo si traduce in un circuito che richiede un totale di sei transistor. Infatti, la porta **NAND** da sola richiede quattro transistor, e l'aggiunta della porta **NOT** (che è costituita da due transistor) aumenta il totale a sei. La porta **AND** così realizzata implementa la logica in cui l'uscita è alta (logico "1") solo quando entrambi gli ingressi sono alti (logico "1"). La combinazione di queste porte CMOS garantisce un'efficienza energetica elevata, in quanto la corrente fluisce principalmente durante le transizioni di stato, riducendo il consumo di potenza.

Il **tempo di transizione** in un circuito è il periodo che un segnale impiega per cambiare da un livello logico a un altro, ad esempio, da "0" a "1" o viceversa. Questo tempo è importante perché influisce sulla velocità con cui un circuito può rispondere ai cambiamenti nei segnali di ingresso.

Il **ritardo di propagazione**, invece, rappresenta il tempo necessario affinché un'uscita di un circuito si adatti ai nuovi valori di ingresso. In altre parole, è il tempo che intercorre tra l'applicazione di un cambiamento all'ingresso e la risposta dell'uscita del circuito. Il **ritardo di propagazione critico** si riferisce al ritardo del percorso più lento tra ingresso e uscita, che determina il tempo di risposta complessivo del circuito. Questo percorso critico è fondamentale per stabilire la frequenza massima di funzionamento del circuito, cioè quante volte il circuito può commutare in un dato periodo di tempo.

Il **fan-in** di una porta logica è il numero di segnali di ingresso che può ricevere, mentre il **fan-out** è il numero di ingressi paralleli a cui l'uscita della porta può essere collegata. Entrambi questi fattori influenzano negativamente il **ritardo di propagazione** e il **margine di rumore** del circuito. Un **fan-in** e un **fan-out** elevati aumentano il carico sui segnali, causando ritardi più lunghi e rendendo il circuito più suscettibile al rumore. Tipicamente, si cerca di limitare entrambi questi valori a dieci per porta per mantenere un buon equilibrio tra performance e affidabilità del circuito.

La **porta tri-state** è un tipo di porta logica che può assumere tre stati distinti: alto (logico "1"), basso (logico "0") e alta impedenza (Z). Lo stato di alta impedenza è simile a un circuito aperto, il che significa che la porta non ha alcuna influenza sul circuito a cui è connessa, come se fosse scollegata. Questo stato è utile in applicazioni dove più dispositivi devono condividere lo stesso bus di dati, poiché consente di disattivare temporaneamente una porta senza interferire con il funzionamento degli altri componenti del sistema.

I **circuiti integrati (IC)** sono dispositivi elettronici che combinano vari componenti elettronici, come resistenze, condensatori e diodi, su un singolo chip di materiale semiconduttore, generalmente silicio. Questo approccio consente di ottenere circuiti più compatti ed economici. Gli IC possono essere digitali, analogici o **mixed signal**, a seconda del tipo di segnali che gestiscono. Un **circuito digitale** lavora con segnali discreti, che assumono solo i valori binari "0" o "1". Un **circuito analogico**, invece, gestisce segnali continui, che possono variare tra un'infinità di valori. Infine, un **circuito mixed signal** integra sia componenti digitali che analogici, permettendo di trattare sia segnali discreti che continui.

Esistono diverse tipologie di circuiti integrati, che si distinguono in base alla **scala di integrazione**, ovvero il numero di componenti, come porte logiche o transistor, integrati su un singolo chip. I circuiti a **piccola scala di integrazione (SSI)** contengono un numero ridotto di componenti, come poche porte logiche. I circuiti a **media scala di integrazione (MSI)** sono in grado di gestire funzioni più complesse, come addizionatori, sottrattori, registri o multiplexer. I circuiti a **grande scala di integrazione (LSI)** possono integrare unità logiche aritmetiche (ALU), banchi di registri o piccoli processori. Infine, i circuiti a **molto grande scala di integrazione (VLSI)** possono contenere milioni di transistor e sono utilizzati per applicazioni avanzate, come memorie di grande capacità e processori potenti.

Un **decoder** è un circuito logico che prende un input codificato e lo converte in una specifica uscita, in base alla combinazione dei segnali in ingresso. Il funzionamento di un decoder si basa sulla sua capacità di attivare una delle sue linee di uscita, a seconda del numero binario ricevuto in ingresso. Ogni combinazione di segnali in ingresso corrisponde a una linea di uscita specifica, che viene attivata, mentre le altre rimangono disattivate. Il decoder è quindi utilizzato per operazioni come la selezione di indirizzi in sistemi di memoria o per altre applicazioni che richiedono la decodifica di segnali binari.

Il **multiplexer** (spesso abbreviato in MUX) è un circuito logico che permette di selezionare uno tra i suoi diversi ingressi e inviarlo all'uscita. In sostanza, il multiplexer agisce come uno "switch" che convoglia un solo ingresso tra quelli disponibili sulla sua uscita, in base a una configurazione di segnali di selezione. Gli ingressi di selezione determinano quale ingresso "dato" venga inviato all'uscita. La realizzazione di un multiplexer può essere descritta come una somma dei prodotti degli ingressi, il che implica che il circuito è in grado di indirizzare dinamicamente uno degli ingressi all'uscita, a seconda della configurazione degli ingressi di selezione. Questo tipo di circuito è utile in molte applicazioni, come la gestione di flussi di dati in sistemi di comunicazione e nella costruzione di sistemi logici complessi.