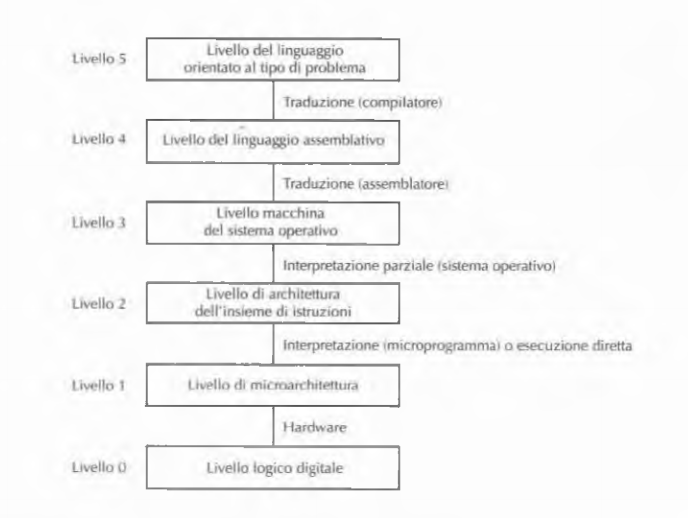
**Linguaggi, livelli e macchine virtuali**

- Traduzione: consiste nel sostituire, in una fase iniziale, ogni sua istruzione con un equivalente sequenza di istruzioni in LO. Il programma che ne risulta è costituito interamente da istruzioni di LO e può essere eseguito dal computer al posto del programma L1 originale.

- Interpretazione: consiste nello scrivere un programma in LO che accetta come dati d’ingresso programmi in L1; tale programma li esegue esaminando un’istruzione alla volta e sostituendola direttamente con l’equivalente sequenza di istruzioni LO. Il programma che la esegue è detto interprete.

La differenza è che, nel caso della traduzione, il programma L1 viene, all’inizio, convertito interamente in un programma LO. Nell’interpretazione ciascuna istruzione L1 viene esaminata e decodificata, e quindi eseguita direttamente senza generare alcun programma tradotto.

**Attuali macchine multilivello**



Livello 0:

• Hardware base: porte logiche e registri.

• Costituisce il fondamento fisico della macchina.

Livello 1 (Microarchitettura):

• Memoria locale con registri (8-32).

• Presenza dell'ALU (Arithmetic Logic Unit) per operazioni aritmetiche.

• Percorso dati: registri connessi all'ALU.

• Il microprogramma controlla le operazioni del percorso dati.

Livello 2:

• Esegue le istruzioni dell'architettura interpretate dal microprogramma.

Livello 3:

• Istruzioni simili a livello 2 ma con aggiunte: diverse organizzazioni della memoria, capacità di esecuzione concorrente.

• Servizi come interprete (storicamente il sistema operativo) per eseguire istruzioni identiche a livello 2.

• Un mix di istruzioni del livello ISA e nuove aggiunte.

Livello 4:

• Permette la scrittura più agevole dei programmi per i livelli inferiori (1, 2, 3).

• Traduce i programmi nei linguaggi dei livelli sottostanti tramite l'assemblatore.

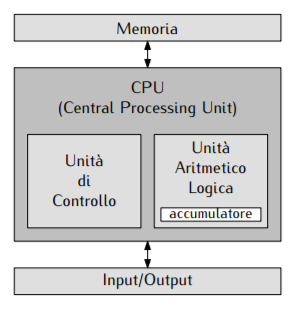
Livello 5:

• Definisce linguaggi di programmazione ad alto livello.

• I programmi vengono tradotti da un compilatore ai livelli 3 o 4.

In sintesi, dall'hardware base (livello 0) si passa attraverso i livelli di microarchitettura, istruzioni, e organizzazione della memoria fino ai linguaggi di programmazione ad alto livello, ognuno dei quali contribuisce a interpretare, eseguire e ottimizzare le istruzioni e i programmi in modi sempre più astratti e user-friendly.

**La macchina di Von Neumann**



Si compone delle seguenti parti:

- *memoria*: conserva sia il programma che i dati su cui deve lavorare il programma;

- *CPU*: è l’unità di elaborazione composta da tre elementi principali:

* *ALU (Arithmetic Logic Unit)*: esegue le istruzioni elementari come quelle aritmetiche e logiche;
* *CU (Control Unit)*: recupera le istruzioni in memoria secondo l’ordine stabilito dall’algoritmo e permette la loro esecuzione;
* *Accumulatore*: è una memoria interna della CPU che viene utilizzata per contenere gli operandi delle istruzioni eseguite dalla ALU.

- *Input/Output (I/O)*: costituisce l’interfacciamento del calcolatore verso l’esterno;

- *bus di comunicazione*: è il canale che permette la comunicazione tra le unità appena descritte.

**Processori**

La CPU è il cervello del computer, ed è composta dall’unità di controllo (CU), l’unità aritmetica e logica (ALU) e alcuni registri (piccole memorie ad altissima velocità).

Tra i registri sono molto importanti:  
- Program Counter (PC): punta alla prossima istruzione da prelevare per l’esecuzione;

- Instruction Register (IR): mantiene l’istruzione corrente in fase di esecuzione.

Le componenti di un computer sono collegate attraverso un bus: una collezione di cavi paralleli utilizzati per trasferire indirizzi, dati e segnali di controllo.

Organizzazione della CPU di von Neumann:

Registri: Composti da 1 a 32 registri, costituiscono il data path della CPU.

ALU (Arithmetic Logic Unit): Effettua operazioni aritmetiche e logiche su dati in ingresso provenienti dai registri. Gli input dell'ALU sono mantenuti in due registri dedicati mentre l'unità è impegnata in calcoli.

Bus: Collegamenti che permettono la comunicazione tra le varie parti della CPU.

Funzionamento:

ALU: Esegue operazioni come addizioni, sottrazioni e altre operazioni di base sui dati in input.

Registri di Output dell'ALU: Memorizzano i risultati delle operazioni eseguite dall'ALU.

Gestione dei dati: Il valore risultante può essere immagazzinato in un registro della CPU e successivamente trasferito in memoria, se necessario.

Tipologie di istruzioni:

Istruzioni Registro-Memoria: Coinvolgono il caricamento o lo scaricamento di parole di memoria nei registri. Permettono di prelevare dati dalla memoria e spostarli nei registri o viceversa. Questi dati possono essere utilizzati come input per le operazioni successive, come ad esempio l'ALU.

Istruzioni Registro-Registro: Coinvolgono operandi già presenti nei registri. Estraggono due operandi dai registri, li trasferiscono all'ALU e eseguono operazioni su di essi. Il risultato viene memorizzato in uno dei registri della CPU.

In sintesi, la CPU di von Neumann si basa su registri, ALU e bus per eseguire operazioni aritmetiche e logiche. Utilizza diverse istruzioni che coinvolgono trasferimenti di dati tra la memoria e i registri, o operazioni dirette tra i registri stessi con l'ausilio dell'ALU.

**Esecuzione dell’istruzione**

La CPU esegue ogni istruzione compiendo una serie di passi:

1. prelevare la successiva istruzione dalla memoria per portarla nell’IR;

2. aggiornare il PC per farlo puntare all’istruzione seguente;

3. determinare il tipo dell’istruzione appena prelevata (decodifica dell’istruzione);

4. se l’istruzione usa una parola in memoria, determinare dove si trova;

5. se necessario, prelevare la parola per portarla in un registro della CPU;

6. eseguire l’istruzione;

7. tornare al punto 1 per iniziare l’esecuzione dell’istruzione successiva.

Spesso ci si riferisce a questa sequenza di passi con il termine di ciclo esecutivo delle istruzioni, o ciclo di prelievo-decodifica-esecuzione (fetch-decode-execute).

**Strategie di progettazione delle CPU**

- CISC (Complex Instruction Set Computer): la CPU è in grado di comprendere molte istruzioni complesse nativamente (è il più alto livello di astrazione riconosciuto dalla macchina);

- RISC (Reduced Instruction Set Computer): si basa sull’idea che se le istruzioni sono semplici e poche, esse possono essere eseguite rapidamente (è necessario un solo ciclo nel datapath);

- Ibrido: a partire dal x486, le CPU intel contengono un sottoinsieme di istruzioni RISC (quelle più comuni) che possono essere eseguite in un singolo ciclo nel datapath, mentre le altre complesse sono interpretate secondo la classica modalità CISC.

Principi di progettazione dei calcolatori moderni (Principi RISC):

Esecuzione diretta delle istruzioni dall'hardware:

Evitare l'interpretazione delle istruzioni.

Le architetture CISC possono suddividere istruzioni complesse in microistruzioni.

Massimizzazione della frequenza di emissione delle istruzioni:

Il parallelismo è cruciale per le prestazioni.

Emissione di un gran numero di istruzioni più semplici contemporaneamente.

Decodifica facile delle istruzioni:

Limitare la complessità della decodifica per ogni istruzione.

Istruzioni regolari, di lunghezza fissa e con pochi campi.

Solo Load e Store fanno riferimento alla memoria:

Limitare l'accesso diretto della memoria solo per le operazioni Load e Store.

Consentire lo spostamento degli operandi dalla memoria ai registri con istruzioni dedicate.

Ampio numero di registri disponibili:

Garantire un gran numero di registri (almeno 32).

Mantenere i dati nei registri per evitare frequenti accessi lenti alla memoria.

Sintesi:

Esecuzione diretta delle istruzioni senza interpretazione.

Massimizzazione del parallelismo per migliorare le prestazioni.

Semplificazione della decodifica delle istruzioni.

Limitazione dell'accesso diretto alla memoria a operazioni specifiche (Load e Store).

Fornire un ampio numero di registri per minimizzare gli accessi alla memoria.

**Parallelismo a livello di istruzione**

Poiché l’incremento del clock del processore ha raggiunto un limite fisico, i progettisti di CPU guardano al parallelismo (più istruzioni nello stesso tempo) per incrementare le prestazioni.

Il parallelismo si può ottenere in due diversi modi:  
- Parallelismo a livello di istruzione: il parallelismo è sfruttato all’interno delle istruzioni per ottenere un maggior numero di istruzioni al secondo;

- Parallelismo a livello di processore: più CPU collaborano per risolvere lo stesso problema.

**Pipelining**

Una limitazione nella velocità di esecuzione delle istruzioni è rappresentato dal prelievo delle istruzioni dalla memoria.

Per alleviare questo problema, i calcolatori sono stati dotati della capacità di poter prelevare in anticipo le istruzioni dalla memoria, in modo da averle già a disposizione nel momento in cui dovessero rendersi necessarie. Le istruzioni venivano memorizzate in un insieme di registri chiamati buffer di prefetch, dai quali potevano essere prese nel momento in cui venivano richieste, senza dover attendere che si completasse una lettura della memoria.

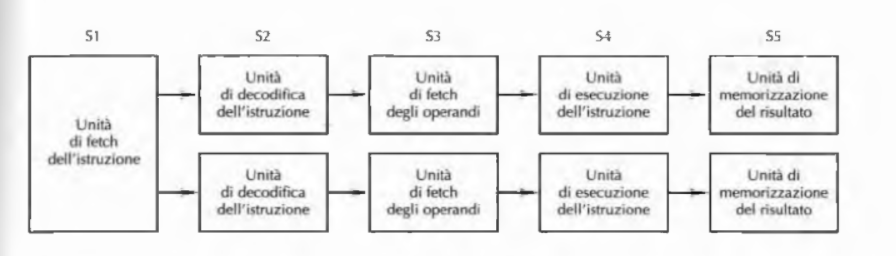
In pratica la tecnica di prefetching divide l’esecuzione dell’istruzione in due parti: il prelievo dell’istruzione e la sua esecuzione effettiva.

Il pipeline divide l’esecuzione di un’istruzione in un numero maggiore di parti che possono essere eseguite in parallelo; ciascuna di queste parti è gestita da componenti hardware dedicati.

**Processori con più pipeline**

Avere due pipeline è sicuramente meglio di averne una sola.

Questa architettura è stata utilizzata inizialmente dall’Intel x486.

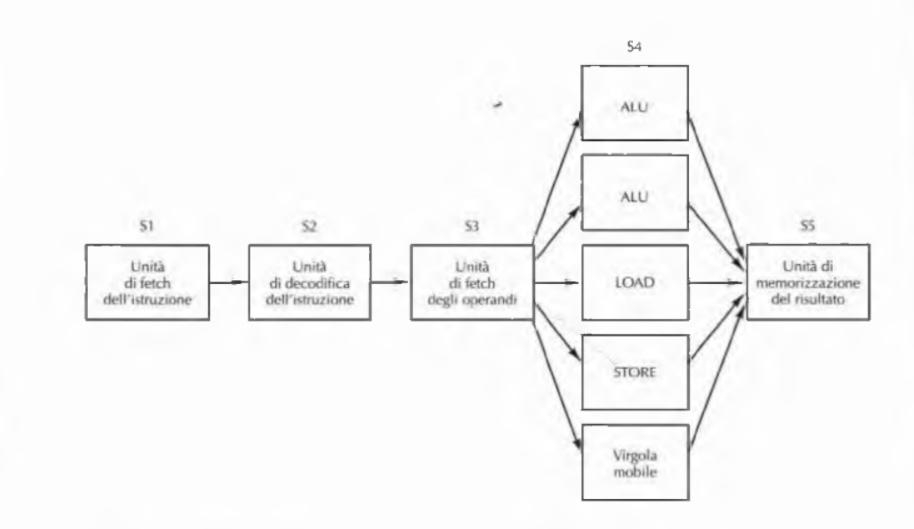


Affinché le due istruzioni possano essere eseguite in parallelo, non devono però esserci conflitti nell’uso delle risorse (cioè i registri) e nessuna delle due istruzioni deve dipendere dal risultato dell’altra.

Inoltre, non tutte le istruzioni possono essere svolte in parallelo (l’input di una istruzione può dipendere dal risultato della precedente) e sarebbero necessarie troppe componenti hardware per le varie unità che andrebbero poi sincronizzate.

**Architetture superscalari**

Viene utilizzata inizialmente da Intel Core. Il processore dispone di una sola pipeline con più unità funzionali in corrispondenza di alcuni stadi.



Affinché l’architettura abbia senso è necessario che la velocità di emissione della fase S3 sia più alta rispetto a quella della fase S4.

La fase S4 può avere delle unità ALU duplicate.

**Parallelismo a livello di processore**

Il parallelismo nel chip aiuta a migliorare le performance della CPU: con il pipelining e le architetture superscalari si può arrivare ad un fattore di miglioramento da 5 a 10.

Però per incrementare drasticamente le performance di un calcolatore occorre progettare sistemi con molte CPU, in questo caso si può arrivare ad ottenere un incremento di 50, 100, o anche più.

Esistono tre differenti approcci: computer con parallelismo sui dati, multiprocessori, multicomputer.

**Computer con parallelismo sui dati**

Ci sono due schemi differenti:

- Processori SIMD (Single Instruction-stream Multiple Data-stream): sono costituiti da un vasto numero di processori identici che eseguono la stessa sequenza di istruzioni su insieme differenti di dati;

- Processori vettoriali: un processore vettoriale esegue la stessa sequenza di operazioni su coppie di dati, ma tutte le addizioni sono svolte da un unico sommatore strutturato in pipeline.

Entrambe le architettura lavorano su array di dati, mentre il primo utilizza tanti sommatori quanti gli elementi del vettore, il secondo utilizza un solo sommatore e un unico registro vettoriale.

**Multiprocessori**

È un’architettura costituita da più CPU che condividono una memoria comune.

Poiché ciascuna CPU può leggere o scrivere qualsiasi zona della memoria comune, le CPU devono sincronizzarsi via software.

In questo caso le CPU hanno la necessità di interagire in modo così profondo che il sistema è detto fortemente accoppiato (tightly coupled).

**Multicomputer**

Multiprocessori con molte CPU sono difficili da realizzare, per via del problema delle connessioni di ciascuna CPU verso la memoria comune.

I progettisti hanno superato il problema abbandonando il concetto di memoria comune e realizzando un elevato numero di CPU interconnesse, ciascuna con la propria memoria privata.

Le CPU in un multicomputer sono accoppiate in modo lasco (loosely coupled) e comunicano attraverso scambi di messaggi.

In architetture grandi la completa interconnessione non è fattibile così sono utilizzate topologie differenti come la griglia, l’albero o l’anello.

**Memoria cache**

La memoria cache è un sistema che combina una piccola quantità di memoria veloce con una più ampia ma più lenta memoria principale al fine di migliorare le prestazioni complessive del sistema. I principi chiave della memoria cache includono:

Località spaziale e temporale: I programmi tendono ad accedere a porzioni specifiche della memoria in modo non casuale. La località spaziale suggerisce che i riferimenti successivi alla memoria saranno vicini agli attuali, mentre la località temporale indica che le stesse zone della memoria verranno frequentemente richiamate nel tempo.

Funzionamento della cache: Le parole di memoria frequentemente utilizzate vengono conservate nella cache. Quando la CPU richiede una parola, la cerca prima nella cache e, se non la trova, accede alla memoria principale. Questo riduce significativamente i tempi di accesso se le parole necessarie sono presenti nella cache.

Livelli di cache: I moderni sistemi di memoria possono avere più livelli di cache:

L1: Cache di istruzioni e cache di dati integrate direttamente nel chip della CPU, di dimensioni relativamente piccole (tra 16 e 64 KB).

L2: Cache di secondo livello, generalmente esterna al chip della CPU ma inclusa nel suo involucro, unificata (contenente sia dati che istruzioni) e di dimensioni variabili (da 512 KB a 1 MB).

L3: Cache di terzo livello, posizionata sulla scheda del processore e di dimensioni più ampie (alcuni MB).

Le cache sono di solito organizzate in modo gerarchico, dove il contenuto della cache di livello superiore è incluso nella cache di livello inferiore.

In sintesi, la memoria cache sfrutta la località dei riferimenti alla memoria per conservare dati frequentemente utilizzati, riducendo così i tempi di accesso e migliorando le prestazioni complessive del sistema, mentre i multipli livelli di cache offrono una gerarchia di velocità e capacità per ottimizzare ulteriormente le prestazioni.

**RAID**

Il RAID (Redundant Array of Inexpensive Disks) è una tecnologia che combina diversi dischi in un unico sistema di archiviazione per migliorare le prestazioni, la capacità e/o la ridondanza dei dati. Esistono diversi livelli di RAID, ognuno con specifiche modalità di organizzazione dei dati:

RAID livello 0:

Organizzazione basata sullo striping.

Distribuisce i dati tra i dischi senza ridondanza.

Migliore per richieste di grandi dimensioni.

Prestazioni elevate, ma non offre ridondanza dei dati.

RAID livello 1:

Organizzazione basata sulla duplicazione (mirroring).

Ogni dato viene replicato su un altro disco.

Prestazioni in scrittura simili a un singolo disco, ma prestazioni in lettura migliori.

Elevata tolleranza ai guasti grazie alla duplicazione dei dati.

RAID livello 2:

Organizzazione basata su parole o byte, con dischi sincronizzati.

Utilizza bit di correzione dell'errore (ECC - Error Correction Code) per la correzione degli errori.

Requisiti di sincronizzazione dei dischi, meno efficiente a meno che non vengano utilizzate molte unità.

RAID livello 3:

Utilizza la parità per il controllo degli errori, con un disco dedicato per i bit di parità.

Offre maggiore tolleranza ai guasti rispetto al livello 2, ma richiede la sincronizzazione dei dischi.

RAID livello 4:

Simile al livello 0, ma con una parità strip-per-strip su un disco separato.

Protegge dalla perdita di un disco, ma le prestazioni possono essere limitate durante l'aggiornamento di piccole quantità di dati.

RAID livello 5:

Distribuisce uniformemente i bit di parità su tutti i dischi.

Migliora la ridondanza e le prestazioni rispetto al livello 4, ma la ricostruzione dei dati dopo un guasto è complessa.

In sintesi, i livelli di RAID offrono diverse combinazioni di prestazioni, ridondanza e tolleranza ai guasti. La scelta del livello di RAID dipende dalle esigenze specifiche di prestazioni, ridondanza dei dati e costo.

Il livello di microarchitettura si trova sopra il livello logico digitale ed è responsabile dell'implementazione dell'Instruction Set Architecture (ISA), il quale può variare in base agli obiettivi di costo e prestazioni del computer. ISA come il RISC (Reduced Instruction Set Computing) si basano su istruzioni più semplici, eseguibili tipicamente in un singolo ciclo di clock, mentre ISA più complessi come il CISC (Complex Instruction Set Computing) possono richiedere più cicli per l'esecuzione di una singola istruzione.

**Esempio di microarchitettura**

Esaminiamo l'esempio pratico della Java Virtual Machine (JVM) semplificata, chiamata IJVM, operante esclusivamente su numeri interi, come rappresentazione di una microarchitettura. Questa IJVM sarà il nostro esempio per comprendere il controllo e l'ordinamento delle istruzioni.

Nel caso della microarchitettura per l'IJVM, questa conterrà un microprogramma registrato in una ROM, responsabile di prelevare, decodificare ed eseguire le istruzioni IJVM. Tuttavia, l'interprete JVM di Sun non può essere utilizzato per controllare l'hardware a un livello di dettaglio richiesto, essendo stato scritto in C per la portabilità.

Il modello convenzionale per progettare la microarchitettura è simile a un problema di programmazione, dove ogni istruzione ISA rappresenta una funzione richiamata dal programma principale. Il programma principale è un semplice ciclo senza fine che determina quale funzione (istruzione) invocare e poi riprende l'esecuzione.

Il microprogramma contiene variabili che costituiscono lo stato del calcolatore, tra cui il Program Counter (PC) che indica la locazione di memoria contenente la successiva istruzione da eseguire. Durante l'esecuzione di un'istruzione, il PC viene avanzato per puntare alla prossima istruzione.

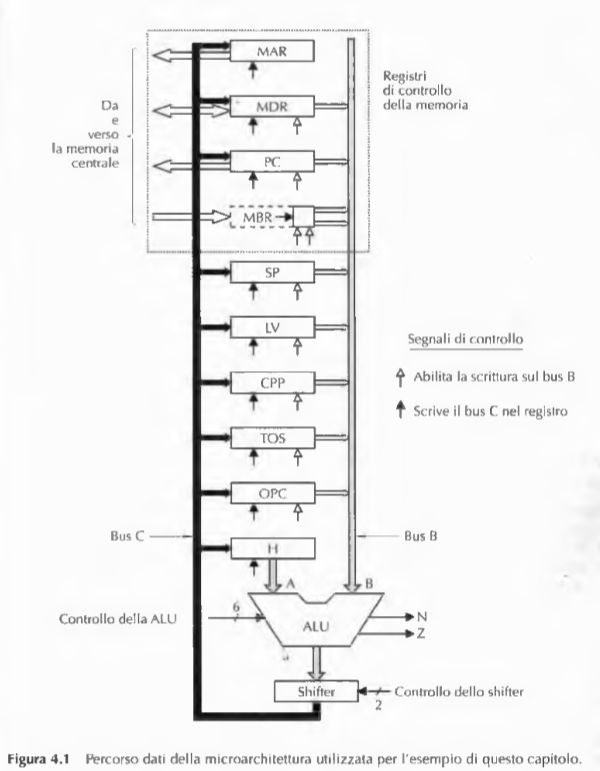
Ogni istruzione IJVM è composta da campi, di solito uno o due, con uno scopo specifico. Il primo campo è l'opcode che identifica il tipo di istruzione (ad esempio ADD, BRANCH) e può specificare l'operando (come indicare una variabile locale).

Il modello fetch-decode-execute è utilizzato per l'esecuzione delle istruzioni, fornendo un metodo astratto per l'implementazione di ISA complessi come l'IJVM.

Le microistruzioni compongono il microprogramma e controllano il percorso dati durante un ciclo, definendo il flusso di esecuzione.

**Percorso dati**

Il percorso dati è una parte essenziale dell'unità di elaborazione di un processore (CPU), comprendente l'Arithmetic Logic Unit (ALU), i suoi input e output, e una serie di registri a 32 bit come il PC, SP e MDR. Questi registri sono accessibili solo a livello di microarchitettura e memorizzano valori corrispondenti alle variabili dell'Instruction Set Architecture (ISA) a cui sono associati.



La funzione dell'ALU è determinata da sei linee di controllo che regolano le sue operazioni, ad esempio, F0 e F1 definiscono l'operazione, ENA e ENB abilitano gli input, INVA inverte l'input di sinistra e INC forza un riporto nel bit meno significativo per l'operazione di somma di 1.

La Figura 4.2 illustra alcune delle combinazioni più rilevanti per l'ALU, come somma, sottrazione, complemento a due, etc. Queste funzioni possono variare a seconda delle esigenze dell'ISA, e talvolta diverse combinazioni possono ottenere lo stesso risultato.

Nel percorso dati, l'ALU richiede due input: uno a sinistra (A) collegato al registro di mantenimento H e uno a destra (B) collegato al bus B. Il bus B può essere caricato da nove sorgenti diverse. È possibile caricare un valore in H utilizzando la somma di B con zero, ottenendo quindi lo stesso valore di B senza modifiche, e memorizzarlo in H attraverso lo shifter senza modificarlo ulteriormente.

Due linee di controllo aggiuntive, SLL8 (Shift Left Logical) e SRA1 (Shift Right Arithmetic), gestiscono operazioni di shift dei dati.

La lettura e la scrittura di un registro nello stesso ciclo avvengono in fasi diverse dello stesso ciclo di clock per evitare inconsistenze nei dati. Durante una fase iniziale del ciclo, quando viene selezionato un registro come input destro della ALU, i suoi valori vengono caricati sul bus B e mantenuti stabili per l'intera durata del ciclo. Le operazioni dell'ALU producono un risultato che viene inviato al bus C attraverso lo shifter. Verso la fine del ciclo, quando i valori di output sono stabili, un segnale di clock attiva la memorizzazione dei valori sul bus C nei registri, consentendo la lettura e la scrittura simultanea in un singolo ciclo.

La sequenza temporizzata del percorso dati permette la coerenza delle letture e scritture simultanee nello stesso registro, senza generare dati incoerenti.

**Microistruzioni**

Il controllo del percorso dati, come descritto nel testo, richiede la gestione di 29 segnali, suddivisi in cinque gruppi funzionali:

Scrittura dei dati dal bus C ai registri: 9 segnali

Abilitazione dei registri sul bus B per l'input della ALU: 9 segnali

Controllo delle funzioni della ALU e dello shifter: 8 segnali

Segnali per indicare la lettura/scrittura della memoria attraverso MAR (MDR): 2 segnali

Segnale per il prelievo dei dati dalla memoria attraverso PC o MBR: 1 segnale

Durante un ciclo del percorso dati, si caricano i valori dei registri sul bus B, si eseguono operazioni attraverso l'ALU e lo shifter, si guida il risultato sul bus C e si riscrivono i dati nei registri appropriati. Se viene eseguita un'operazione di lettura dalla memoria, essa inizia alla fine del ciclo di percorso dati, dopo il caricamento di MAR. I dati della memoria saranno disponibili solo nel ciclo successivo, quindi non possono essere utilizzati nel ciclo immediatamente successivo alla lettura.

Nel progetto del controllore, si utilizzano 24 segnali (24 bit) per controllare il percorso dati in un ciclo. La seconda parte del controllo riguarda le operazioni da eseguire nel ciclo successivo. Si adotta un formato che comprende i 24 bit di controllo, oltre a due campi aggiuntivi: NEXT\_ADDRESS e JAM.

Questi sei gruppi funzionali per il controllo del percorso dati comprendono:

Indirizzo (Addr): Contiene l'indirizzo di una potenziale successiva microistruzione.

JAM: Determina come viene selezionata la successiva microistruzione.

ALU: Seleziona le funzioni della ALU e dello shifter.

C: Seleziona quali registri sono scritti dal bus C.

Mem: Seleziona la funzione della memoria.

B: Seleziona la sorgente del bus B.

Questi segnali di controllo sono codificati in 36 bit e vengono utilizzati per descrivere le operazioni da eseguire, includendo le azioni per il ciclo successivo.

Il formato del controllore è stato attentamente strutturato per minimizzare l'incrocio delle linee, che in schemi simili corrispondono a collegamenti che si incrociano sui chip. Questi incroci possono complicare i progetti, quindi minimizzarli è una pratica consigliata per semplificare l'architettura

**Stack**

Il concetto di stack è fondamentale nella gestione delle variabili locali e nell'elaborazione delle espressioni in molti linguaggi di programmazione. In sostanza, il problema principale è determinare dove memorizzare le variabili locali, specialmente considerando la possibilità che una procedura possa richiamare se stessa.

La soluzione non può essere l'assegnazione di indirizzi assoluti di memoria alle variabili poiché ciò potrebbe generare conflitti se una procedura viene invocata più volte contemporaneamente.

Quindi, si utilizza lo stack, un'area di memoria flessibile, per memorizzare le variabili locali. Si definisce un registro, LV, che punta alla procedura corrente e un altro registro, SP, che punta all'ultimo elemento nello stack delle variabili locali.

Quando una procedura richiama un'altra procedura, lo spazio per le variabili locali della procedura chiamata viene allocato sopra nello stack. LV viene modificato per puntare alle nuove variabili locali, consentendo l'accesso tramite l'offset rispetto a LV.

Questo processo si ripete a ogni chiamata di procedura, con le variabili locali che vengono sovrapposte nello stack. Quando una procedura termina, lo spazio assegnato alle sue variabili locali viene liberato.

Lo stack può anche essere utilizzato per calcolare espressioni aritmetiche, dove gli operandi vengono inseriti nello stack e le operazioni vengono eseguite utilizzando la logica dello stack. Ad esempio, per eseguire a1 = a2 + a3, i valori di a2 e a3 vengono inseriti nello stack, l'addizione viene eseguita e il risultato viene memorizzato in a1.

In alcuni casi, come nel calcolo di espressioni che coinvolgono funzioni, sia le variabili locali che gli operandi possono essere memorizzati nello stack, consentendo un'elaborazione efficiente dell'espressione.

In sintesi, lo stack viene utilizzato per gestire le variabili locali delle procedure e come meccanismo per l'elaborazione efficiente delle espressioni, consentendo un'allocazione dinamica della memoria necessaria.

**ISA**   
L'ISA è fondamentale poiché costituisce un linguaggio comune comprensibile sia per i compilatori che per l'hardware. Sebbene sia teoricamente possibile eseguire direttamente codice di alto livello sull'hardware, ciò comporterebbe la perdita delle prestazioni ottimali ottenute tramite la compilazione. Inoltre, è preferibile che i computer siano in grado di eseguire programmi scritti in vari linguaggi anziché limitarsi a uno solo.

La prassi comune tra i progettisti è quella di tradurre vari linguaggi di alto livello in un formato intermedio comune, ovvero l'ISA, per poi costruire l'hardware che esegue direttamente i programmi in questo linguaggio. Tale livello definisce l'interfaccia tra compilatori e hardware. Durante la progettazione di una nuova macchina, sia i progettisti del compilatore che quelli dell'hardware collaborano per determinare le caratteristiche desiderate per l'ISA. Se le richieste del compilatore non sono realizzabili con costi accettabili, vengono escluse. Allo stesso modo, se l'hardware propone funzionalità che non possono essere utilizzate dai software esistenti, queste idee vengono scartate.

Tuttavia, nella pratica, la domanda principale dei potenziali utenti riguarda la compatibilità con i predecessori. Questo comporta la necessità di mantenere l'ISA costante o retrocompatibile con le generazioni precedenti, consentendo ai vecchi programmi di funzionare senza modifiche significative. Tuttavia, le nuove macchine possono offrire funzionalità innovative sfruttabili solo dai nuovi software.

Pertanto, sebbene gli ingegneri abbiano libertà nella progettazione dell'hardware, l'ISA deve rimanere retrocompatibile con le generazioni precedenti. Un buon ISA, oltre a garantire la retrocompatibilità, offre vantaggi in termini di prestazioni e può influenzare notevolmente la potenza di calcolo e l'efficienza del processore. Un buon ISA si distingue per la capacità di essere implementato efficientemente dalle attuali e future tecnologie, riducendo i costi di produzione e di esecuzione del software.

Inoltre, un ISA efficace favorisce una compilazione "pulita", offrendo al compilatore un insieme di istruzioni regolari e complete che consentono una scelta ottimale delle alternative di esecuzione. La regolarità e la completezza dell'ISA sono quindi importanti per consentire al compilatore di operare scelte migliori senza limitazioni che potrebbero compromettere l'efficienza del codice generato.

**Proprietà del livello ISA**

Il livello ISA, o Architettura dell'Insieme di Istruzioni, può essere definito come la rappresentazione della macchina dal punto di vista del programmatore in linguaggio macchina. Tuttavia, dato che ormai pochi programmatori scrivono direttamente in linguaggio macchina, possiamo considerare il codice a livello ISA come l'output di un compilatore.

Per produrre codice a livello ISA, i progettisti dei compilatori devono comprendere il modello di memoria, i registri disponibili, i tipi di dati e le istruzioni accessibili. Tutte queste informazioni insieme definiscono il livello ISA.

Inizialmente, si è affermato che aspetti come la microarchitettura, la microprogrammazione, la pipeline o la scalabilità non fanno parte del livello ISA perché non sono visibili direttamente al progettista del compilatore. Tuttavia, alcune di queste proprietà possono influenzare le prestazioni, il che è rilevante per il compilatore. Ad esempio, in una microarchitettura superscalare che emette istruzioni in successione se sono di tipo diverso, il compilatore potrebbe ottimizzare il codice per ottenere prestazioni migliori.

Spesso, il livello ISA è definito attraverso documenti formali, talvolta redatti da consorzi di aziende, che consentono ai produttori di costruire macchine in grado di eseguire lo stesso codice, garantendo gli stessi risultati.

Questi documenti di definizione hanno sezioni normative che stabiliscono requisiti obbligatori e informative che forniscono ulteriori dettagli e spiegazioni. Le sezioni normative utilizzano termini come "deve", "non deve" e "dovrebbe" per imporre, vietare o suggerire aspetti dell'architettura. Ciò significa che il compilatore non può fare affidamento su comportamenti prestabiliti e ciascun produttore ha la libertà di fare le proprie scelte.

Le specifiche architetturali sono accompagnate da test sperimentali che verificano la conformità dell'implementazione alla specifica corrispondente.

Un'altra proprietà importante del livello ISA è la presenza di almeno due modalità di esecuzione: modalità kernel e modalità utente. La modalità kernel è utilizzata per l'esecuzione del sistema operativo e consente l'accesso a tutte le istruzioni, mentre la modalità utente è destinata all'esecuzione di programmi applicativi e può limitare l'esecuzione di alcune istruzioni più "sensibili" che manipolano direttamente la cache, ad esempio.

**Modelli di memoria**

I computer organizzano la memoria in celle consecutive, comunemente di 8 bit, chiamate byte. La preferenza per i byte deriva dal fatto che i caratteri nella tabella ASCII occupano 7 bit, permettendo di utilizzare un byte intero. In futuro, se lo standard UNICODE diventerà predominante, potrebbe portare a unità consecutive di 16 bit.

Di solito, i byte vengono raggruppati in parole di 4 byte (32 bit) o 8 byte (64 bit). Molte architetture richiedono che le parole siano allineate lungo i loro limiti. Ad esempio, una parola di 4 byte può iniziare solo agli indirizzi 0, 4, 8, ecc., mentre una parola di 8 byte può iniziare agli indirizzi 0, 8, 16, ecc. Questo allineamento ottimizza l'efficienza della memoria, ma può creare complicazioni.

La capacità di leggere parole da indirizzi arbitrari richiede hardware più complesso, rendendo il chip più grande e costoso. Gli ingegneri preferirebbero evitare questo, ma impongono spesso che ogni riferimento in memoria sia allineato. Tuttavia, la richiesta di retrocompatibilità e di poter eseguire codice vecchio porta spesso a compromessi.

Molti processori hanno uno spazio lineare degli indirizzi che va da 0 fino a 2^32 o 2^64 byte. Alcune macchine hanno spazi separati per istruzioni e dati, il che consente l'esecuzione di programmi e il recupero di istruzioni da spazi diversi. Questo schema offre vantaggi di sicurezza e permette una più ampia selezione di indirizzi.

I modelli di memoria separata per dati e istruzioni differiscono dalla cache di primo livello separata. Nel primo caso, gli accessi ai diversi spazi degli indirizzi portano a risultati diversi, mentre nella cache separata, un singolo spazio degli indirizzi è diviso tra cache differenti.

Un aspetto cruciale nel modello di memoria è la semantica della memoria stessa. Ad esempio, l'ordine di esecuzione di un'istruzione di caricamento (LOAD) dopo un'istruzione di memorizzazione (STORE) allo stesso indirizzo. In molte architetture, il riordinamento delle microistruzioni può creare comportamenti inaspettati.

Per risolvere questo problema, i progettisti di sistema possono adottare vari approcci. Ad esempio, la serializzazione di tutte le richieste di accesso alla memoria assicura che ogni operazione sia completata prima dell'emissione della successiva, garantendo l'ordine di esecuzione specificato dal programma. Al contrario, permettere all'hardware di gestire l'ordine richiede istruzioni aggiuntive nel programma per sincronizzare gli accessi alla memoria. Esistono anche modelli di memoria intermedi che bloccano solo alcuni accessi, lasciando altri liberi.

Sebbene queste peculiarità dell'architettura a livello ISA possano creare problematiche per i programmatori, l'abitudine a tali implementazioni è diffusa, poiché molte dipendono da caratteristiche sottostanti come il riordinamento delle istruzioni, le pipeline complesse e i sistemi di cache multipli.

**Registri**

I registri sono componenti essenziali di un computer e possono essere divisi in due categorie principali a livello ISA: registri specializzati e registri d'uso generale.

Registri specializzati: Questi includono registri come il Program Counter (PC) e il puntatore allo stack, dedicati a funzioni specifiche e visibili sia a livello ISA che microarchitetturale.

Registri d'uso generale: Questi registri contengono variabili locali e risultati temporanei del calcolo. In alcune architetture, i registri d'uso generale sono simmetrici e intercambiabili, mentre in altre possono essere specializzati. Ad esempio, il Pentium 4 ha un registro come EDX che può essere utilizzato come registro d'uso generale, ma è destinato anche a ricevere metà del prodotto in una moltiplicazione o metà del dividendo in una divisione.

Solitamente, sistemi operativi o compilatori seguono convenzioni nell'uso di questi registri. Se ci sono convenzioni su come utilizzare i registri e queste non vengono rispettate, possono verificarsi problemi con la gestione dei dati in memoria.

Oltre ai registri visibili ai programmi dell'utente, ci sono registri specializzati visibili solo in modalità kernel che gestiscono cache, memoria, dispositivi di I/O e altre funzionalità hardware. Questi sono utilizzati solo dal sistema operativo.

Il registro di flag o PSW (Program Status Word) è un registro di controllo che contiene vari bit di varia natura, tra cui i codici di condizione. Questi codici di condizione riflettono lo stato del risultato dell'operazione più recente, come N per indicare un risultato negativo, Z per un risultato uguale a zero, V per indicare un overflow, C per un riporto oltre l'ultimo bit più significativo, A per un riporto oltre il terzo bit (riporto ausiliario) e P per un risultato pari (parità nulla).

Questi codici di condizione sono utilizzati da istruzioni come CMP (comparazione) e dai salti condizionati. Ad esempio, un'istruzione CMP imposta il codice di condizione in base alla differenza tra due operandi, e un'istruzione successiva come BEQ (branch on equal, "salto se uguali") controlla il bit di condizione Z per decidere se eseguire il salto.

Oltre ai codici di condizione, il PSW contiene altri campi specifici della macchina, come la modalità di macchina (utente o kernel), i bit di traccia per il debugging, il livello di priorità della CPU e lo stato di attivazione degli interrupt. Alcuni di questi campi potrebbero essere scrivibili solo in modalità kernel per ragioni di sicurezza e controllo dell'hardware.

**Indirizzamento**

Indirizzamento Immediato: In questa modalità, l'operando è incorporato direttamente nell'istruzione stessa. L'operando è immediatamente disponibile e non richiede un riferimento in memoria separato. Tuttavia, questa modalità presenta limitazioni nella dimensione dell'operando e nel numero di operandi forniti.

Indirizzamento Diretto: Qui, l'indirizzo completo dell'operando in memoria è specificato nell'istruzione. Questa modalità è adatta per accedere a variabili globali il cui indirizzo è noto in fase di compilazione, ma limita l'accesso a una posizione di memoria fissa.

Indirizzamento a Registro: Si specifica un registro piuttosto che un indirizzo di memoria. Questa è la modalità più utilizzata in quanto i registri sono veloci in accesso e hanno indirizzi brevi.

Indirizzamento a Registro Indiretto: L'indirizzo dell'operando in memoria è contenuto in un registro. Questo metodo permette di referenziare la memoria senza incorporare un intero indirizzo nell'istruzione e consente l'accesso a diverse posizioni di memoria in diverse esecuzioni.

Indirizzamento Indicizzato: Questa modalità permette di referenziare una parola di memoria che si trova a uno specifico spiazzamento (offset) dal contenuto di un registro. È utile per accedere alle variabili locali o agli elementi di un array.

Indirizzamento Indicizzato Esteso: Alcune macchine consentono la somma del contenuto di due registri più un offset per calcolare l'indirizzo di memoria. Questo metodo è più efficiente della modalità con offset in quanto richiede meno spazio per l'istruzione.

Indirizzamento a Stack: Le istruzioni senza indirizzi possono essere eseguite in associazione con uno stack. Gli operandi sono posti sullo stack e le operazioni vengono eseguite utilizzando gli elementi nello stack.

Notazione Polacca Inversa: Si tratta di una notazione per esprimere le espressioni matematiche in cui gli operatori vengono posti dopo gli operandi. Questa notazione è ideale per la valutazione delle formule da parte di un computer dotato di stack.

Le diverse modalità di indirizzamento hanno vantaggi e limitazioni, e la scelta di una modalità rispetto a un'altra dipende dalle esigenze specifiche della programmazione e dall'architettura del computer utilizzato.

**Trap e Interrupt**

Trap: Le trap sono chiamate di procedura automatiche che vengono attivate quando si verificano determinate condizioni eccezionali causate da un programma. Ad esempio, un'overflow (quando un risultato aritmetico supera la capacità di rappresentazione) può causare una trap. Le trap sono gestite da un gestore di trap, che si occupa di affrontare l'eccezione, come la stampa di un messaggio di errore.

Interrupt: Gli interrupt sono cambiamenti nel flusso esecutivo generati da eventi esterni al programma in esecuzione. Al contrario delle trap, gli interrupt non sono sincroni e possono essere causati da eventi esterni, come la pressione di un tasto o l'arrivo di dati da un dispositivo di input/output. Quando si verifica un interrupt, il controllo viene trasferito a un gestore di interrupt che prende in carico l'evento. Una volta completato il gestore di interrupt, il controllo ritorna al programma interrotto esattamente da dove è stato interrotto.

Alcuni punti importanti riguardo alle trap e agli interrupt includono:

Gestione Hardware e Software: Le trap sono generate da condizioni interne al programma, mentre gli interrupt sono causati da eventi esterni all'esecuzione del programma.

Trasparenza: Dopo l'interruzione, sia le trap che gli interrupt devono far tornare il sistema allo stato precedente, garantendo una corretta ripresa dell'esecuzione del programma.

Gestione Priorità: Gli interrupt possono essere gestiti con priorità, consentendo di dare precedenza a determinati eventi o dispositivi rispetto ad altri.

Ripristino e Gestione: Durante un interrupt, vengono salvati i registri di stato per poi ripristinarli alla fine del gestore di interrupt, garantendo il ritorno al flusso precedente.

Inoltre, il capitolo menziona l'implementazione hardware e software di trap e interrupt, discutendo le azioni che avvengono a livello di hardware e software quando si verificano queste eccezioni o interruzioni. Infine, viene esplorato il concetto di trasparenza degli interrupt, che garantisce che più interruzioni possano essere gestite senza compromettere l'integrità del sistema.

**linguaggio assemblativo**

Traduzione e esecuzione: Il linguaggio assemblativo coinvolge la traduzione del codice sorgente in un programma oggetto, che viene eseguito solo dopo la traduzione. Durante l'esecuzione del programma oggetto, si interagisce con i livelli di micro e macro architettura e con il livello macchina del sistema operativo.

Linguaggio Assemblativo: È un linguaggio in cui ogni istruzione corrisponde direttamente a un'istruzione macchina. Utilizza simboli per i nomi e gli indirizzi, rendendolo più leggibile rispetto alla rappresentazione binaria o ottale. Consente un accesso completo alle funzionalità e alle istruzioni della macchina di destinazione.

Vantaggi del linguaggio assemblativo: Nonostante la complessità, viene utilizzato per le prestazioni e la piena gestione delle funzionalità della macchina. In alcune applicazioni dove le dimensioni del programma, la velocità e il controllo completo della macchina sono cruciali, il linguaggio assemblativo rimane l'unica scelta.

Formato delle istruzioni: Le istruzioni nel linguaggio assemblativo sono composte da quattro parti: un campo etichetta (opzionale), un campo operazione (codice operativo), un campo operandi e un campo commenti. Le etichette forniscono nomi agli indirizzi e sono utilizzate per le destinazioni delle istruzioni di salto. Il campo del codice operativo contiene l'abbreviazione simbolica del codice o il comando per l'assemblatore. Il campo operandi specifica gli indirizzi o i registri utilizzati come operandi dall'istruzione. I commenti forniscono spiegazioni utili sul funzionamento del programma.

Esempio di linguaggio assemblativo: Viene mostrato un esempio di codice assemblativo che esegue un'istruzione di somma di interi su architettura x86, spiegando l'utilizzo delle etichette, dei codici operativi, degli operandi e dei commenti.

L'obiettivo principale del linguaggio assemblativo è offrire un controllo dettagliato sul funzionamento della macchina, consentendo ai programmatori di ottimizzare le prestazioni e le funzionalità del programma, a discapito della complessità e della portabilità del codice su diverse macchine.

Pseudoistruzioni:

Le pseudoistruzioni o direttive dell'assemblatore sono comandi indirizzati all'assemblatore stesso.

Esempi includono EQU, DB (define byte), condizionali come IF e ENDIF.

Macroistruzioni:

Le macroistruzioni consentono ai programmatori di definire sequenze di istruzioni per un uso ripetuto.

Definizione, chiamata ed espansione delle macro sono parti fondamentali.

Durante l'assemblaggio, l'assemblatore sostituisce il nome della macro con il suo corpo (espansione di macro).

Macro VS Procedure:

Le chiamate di macro sono istruzioni dirette all'assemblatore per sostituire il nome della macro con il suo corpo.

Le chiamate di procedura sono istruzioni macchina che richiamano una procedura e comportano un salto nel codice.

Macro con Parametri:

Le macro possono avere parametri formali che vengono sostituiti con valori attuali durante l'espansione della macro.

Processo di Assemblaggio:

L'assemblatore opera in due passate per risolvere riferimenti in avanti, costruire la tabella dei simboli e generare il programma oggetto.

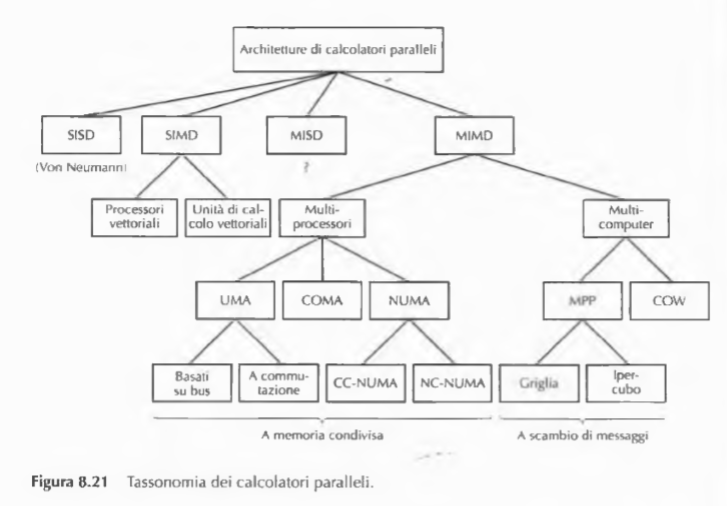
Collegamento e Caricamento:

Il linker unisce i moduli oggetto generati separatamente in un eseguibile, risolvendo riferimenti a memoria e ad altre procedure.

Compiti del Linker:

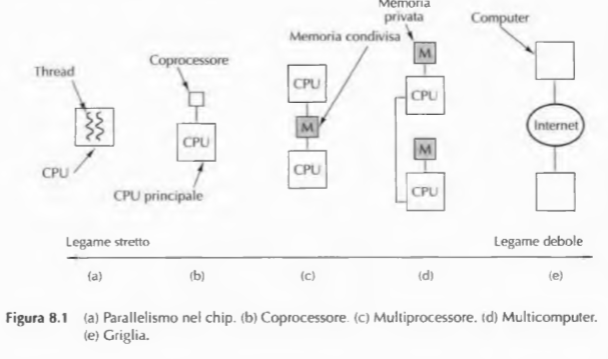
Il linker crea un'immagine dello spazio di indirizzamento virtuale, assegna gli indirizzi di partenza ai moduli e risolve riferimenti a memoria e procedure esterne.

# Classificazione di Flynn



# Parallelismo nel chip

Obiettivo: far svolgere al chip più compiti alla volta.

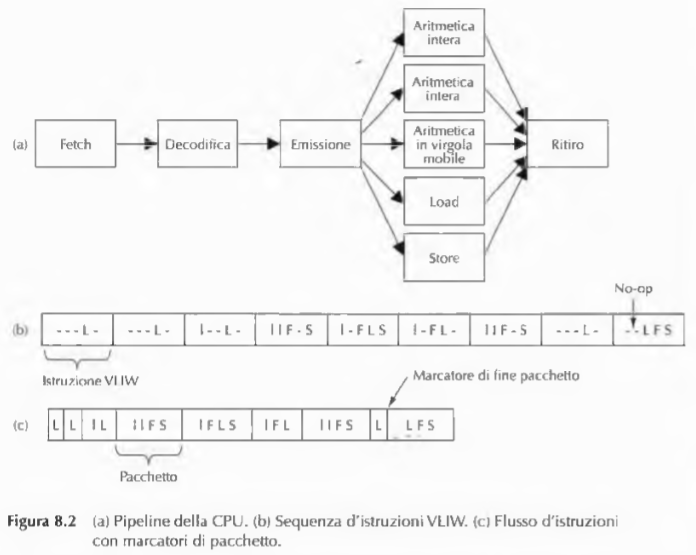


## **Parallelismo a livello delle istruzioni**

L’idea è di emettere più istruzioni per ciclo di clock. Ci sono due tipi di CPU a emissione multipla: processori superscalari e processori VLIW (“Very Long Instruction Word”, con parole di istruzione molto lunghe).

Le CPU superscalari sono composte da pipeline e più unità funzionali (Figura 8.2(a)).

I processori VLIW sono in grado di indirizzare le diverse unità funzionali con una sola linea di pipeline.



**MULTITHREADING**

Multithreading a grana fine:

Consente alla CPU di gestire più thread contemporaneamente, commutando tra di essi ad ogni ciclo.

Ogni thread ha il proprio set di registri e l'identificatore del thread viene associato a ciascuna operazione.

Nasconde gli stalli nell'esecuzione delle istruzioni, consentendo ad altri thread di continuare l'esecuzione.

Multithreading a grana grossa:

Un thread continua ad emettere istruzioni fino a quando non va in stallo, momento in cui la CPU passa immediatamente a un altro thread.

Potenzialmente meno efficiente del multithreading a grana fine ma richiede meno thread per mantenere la CPU occupata.

Varianti del multithreading a grana fine:

Un'implementazione "anticipa" gli stalli per approssimare il multithreading a grana fine, riducendo i cicli persi.

L'identificatore del thread viene inserito in ogni operazione nel caso del multithreading a grana fine, mentre nel multithreading a grana grossa si svuota la pipeline a ogni commutazione di thread.

Multithreading simultaneo:

Ogni thread emette istruzioni fino a quando possibile, ma se uno va in stallo, la CPU inizia immediatamente l'esecuzione di un'istruzione di un altro thread per mantenere la CPU pienamente impegnata.

Le moderne CPU utilizzano queste tecniche di multithreading per gestire gli stalli nell'esecuzione delle istruzioni e mascherare i tempi di latenza della cache, ottimizzando l'uso dell'hardware e migliorando le prestazioni complessive della CPU.

**Tipologie di multiprocessori in un solo chip:**

Multiprocessori omogenei:

Le CPU condividono le stesse cache di primo e secondo livello oltre alla memoria principale.

Due tipi predominanti:

* Pipeline doppia: Un singolo chip con una duplice pipeline che può raddoppiare il throughput, condividendo alcune risorse come le unità funzionali.
* Multiprocessori separati: Più core separati, ognuno contenente una CPU completa e con la propria pipeline. Questi core sono meno accoppiati, comportando interazioni più complesse tra CPU.

Multiprocessori eterogenei:

* Ogni core ha un compito specifico, come decoder audio/video, crittoprocessore o interfacce di rete. Queste architetture integrano un calcolatore completo in un unico chip, noto come "system on a chip".

Condivisione delle risorse:

* Anche se le CPU possono condividere cache o risorse, condividono sempre la memoria principale.
* Il processo di snooping garantisce la coerenza dei dati nelle cache condivise, rimuovendo automaticamente le copie obsolete.

Sistemi di comunicazione tra core:

* Nei sistemi più piccoli, un unico bus potrebbe essere sufficiente, ma nei sistemi più grandi potrebbe diventare un collo di bottiglia.
* Soluzioni comuni includono l'uso di più bus o la configurazione di un anello attraverso tutti i core.
* In un anello, l'arbitraggio avviene attraverso un token che circola nell'anello: un core deve catturare il token per trasmettere e, una volta terminata la comunicazione, può rimettere il token nell'anello per continuare la circolazione. Questo protocollo previene le collisioni sull'anello.

Infine, viene sottolineato che, sebbene siano disponibili chip multicore, le applicazioni e gli algoritmi attuali devono ancora sfruttare pienamente le potenzialità di questi sistemi paralleli. La scrittura di algoritmi paralleli efficaci e la gestione delle risorse condivise rimangono sfide per molti programmatori.

**MULTIPROCESSORI E MULTICOMPUTER**

**Multiprocessori**:

* Memoria Condivisa: Le CPU condividono un unico spazio di memoria comune.
* Cooperazione tra Processi: Processi diversi possono condividere lo stesso spazio degli indirizzi virtuali mappati nella memoria comune, permettendo una comunicazione semplice attraverso le istruzioni LOAD e STORE.
* SMP (Symmetric MultiProcessor): Le CPU hanno uguale accesso a tutti i moduli di memoria e dispositivi di I/O, gestiti in modo intercambiabile dal sistema operativo.
* Tipologie di Multiprocessori:
  + UMA (Uniform Memory Access): Ogni parola di memoria può essere letta alla stessa velocità.
  + NUMA (NonUniform Memory Access): Non tutte le parole di memoria possono essere lette alla stessa velocità.

**Multicomputer**:

* Memoria Privata: Ogni CPU ha la propria memoria privata accessibile solo da essa.
* Comunicazione tra CPU: Non c'è accesso diretto alla memoria delle altre CPU, richiedendo uno scambio di messaggi attraverso la rete di interconnessione.
* Complessità di Programmazione: Programmare un multicomputer è più complesso rispetto a un multiprocessore a causa della comunicazione attraverso messaggi software (send e receive).
* Costruzione e Costi: I multicomputer grandi sono più semplici ed economici da costruire rispetto ai multiprocessori con lo stesso numero di CPU.

**Implementazioni Ibride della Memoria Condivisa**:

* Varianti di Implementazione della Memoria Condivisa: L'implementazione della memoria condivisa può avvenire a livelli diversi, presentando vantaggi e svantaggi specifici.
* Approcci Ibridi: Sforzi per costruire sistemi ibridi che combinano caratteristiche di memoria condivisa e accesso distribuito, con diverse soluzioni per l'implementazione della memoria condivisa.

In sintesi, il testo evidenzia le differenze fondamentali tra multiprocessori e multicomputer, sottolineando le caratteristiche distintive di ciascun approccio, la gestione della memoria e la complessità di programmazione associata a ciascuno di essi. Inoltre, evidenzia gli sforzi per creare sistemi ibridi che possano combinare vantaggi di entrambi gli approcci.

**Timesharing:**

* Vettore di Liste di Processi: Utilizza un vettore di liste di processi a varie priorità di esecuzione.
* Condivisione di Risorse e Bilanciamento del Carico: Assegna il tempo tra le CPU come se fossero in un sistema monoprocessore, bilanciando automaticamente il carico.

**Problemi Potenziali:**

* Dispute per l'Accesso alla Struttura di Scheduling: Con l'aumentare delle CPU, potrebbero verificarsi conflitti nell'accesso a questa struttura dati condivisa.
* Sovraccarico nello Scambio di Contesto per I/O: Lo scambio di contesto quando un processo bloccato per I/O potrebbe causare sovraccarico.

**Approcci per Mitigare i Problemi**:

* Flag Speciale per Spin Locks: Quando un processo termina il suo quanto di tempo mantenendo uno spin lock, può attivare un flag speciale per ottenere un extra-quanto e non bloccare completamente il processo successivo.
* Scheduling per Affinità: Assegna lo stesso processo alla stessa CPU per sfruttare la cache della CPU già utilizzata dal processo.

**Scheduling a Condivisione dello Spazio (Space Sharing):**

* Partizionamento Statico delle CPU: Assegna gruppi di CPU a gruppi di processi correlati, permettendo l'esecuzione simultanea di processi collegati.
* Eliminazione della Multiprogrammazione: Elimina l’overhead degli scambi di contesto ma può portare a tempo sprecato se alcune CPU non hanno lavoro da fare.

**Schedulazione Gang:**

* Esecuzione Simultanea dei Thread Appartenenti ad un Gruppo (Gang): Schedula i thread correlati come un'unità, eseguendoli contemporaneamente in diverse CPU.
* Sincronizzazione dei Tempi di Esecuzione dei Thread della Gang: Tutti i membri della gang iniziano e terminano la loro porzione di tempo contemporaneamente, in intervalli discreti.

**Vantaggi della Schedulazione Gang:**

* Comunicazione più Rapida tra Thread di un Processo: Tutti i thread di un processo in esecuzione simultaneamente possono comunicare tra loro in modo più efficiente e rapido.

In sintesi, le tecniche di scheduling analizzate mirano a ottimizzare l'utilizzo delle risorse di CPU, affrontando sfide come il bilanciamento del carico, l'efficienza della comunicazione tra processi e l'ottimizzazione degli scambi di contesto. Ogni approccio ha i propri vantaggi e svantaggi, e la scelta dell'algoritmo dipende dalle esigenze specifiche del sistema e delle applicazioni da gestire.

**Scheduling su Multiprocessori e Multicomputer**:

* Multiprocessori: Tutti i processi condividono la stessa memoria, quindi le CPU possono scegliere liberamente i processi da eseguire. La schedulazione richiede una lista centralizzata dei processi pronti.
* Multicomputer: Ogni nodo ha memoria e processi propri. La schedulazione è più facile, ma l'allocazione dei processi ai nodi diventa cruciale. L'uso della schedulazione gang può essere adottato per consentire la coordinazione dell'esecuzione dei processi tra i nodi.

**Bilanciamento del Carico:**

Assegnazione dei Processi: Gli algoritmi di allocazione del processore cercano di assegnare i processi ai nodi in base a vari fattori, come l'utilizzo della CPU, della memoria e la quantità di comunicazione tra processi. L'obiettivo può essere minimizzare l'uso della CPU non sfruttata, ridurre la larghezza di banda di comunicazione o garantire condizioni eque per gli utenti e i processi.

**Virtualizzazione:**

**Macchine Virtuali**: La virtualizzazione permette a un singolo computer "reale" di ospitare più computer "virtuali". Ogni macchina virtuale può eseguire il proprio sistema operativo e applicazioni.

**Vantaggi della Virtualizzazione**: Isolamento dei malfunzionamenti, risparmio di spazio, minor consumo energetico, maggiore manutenibilità e possibilità di eseguire applicazioni legacy.

**Requisiti di Virtualizzazione**: La CPU deve supportare un insieme di istruzioni sensibili che sono un sottoinsieme di quelle privilegiate per garantire la virtualizzazione. L'Intel 386 e i suoi successori, ad esempio, presentavano istruzioni sensibili non virtualizzabili.

**Sviluppi nelle Tecnologie di Virtualizzazione**: Intel e AMD nel 2005 introducono la virtualizzazione sulle loro CPU (VT per Intel, SVM per AMD) creando contenitori per eseguire le macchine virtuali.

In sintesi, l'efficace gestione dei processi su sistemi multiprocessore e multicomputer richiede approcci specifici di schedulazione e bilanciamento del carico, mentre la virtualizzazione offre vantaggi significativi ma richiede supporto hardware specifico per essere implementata in modo efficiente.

Gli hypervisor, o VMM (Virtual Machine Monitor), sono software che gestiscono macchine virtuali consentendo l'esecuzione simultanea di più sistemi operativi su una singola macchina fisica. Questi hypervisor possono essere classificati in diverse categorie, come tipo 1, tipo 2 e paravirtualizzazione, ognuno con differenti approcci e performance.

**Hypervisor di Tipo 1:**

* Posizione e Modalità di Esecuzione: Si trova nel sistema operativo reale (SO host) e gira in modalità kernel.
* Supporto alle Macchine Virtuali: Gestisce le macchine virtuali (SO guest).
* Gestione delle Istruzioni Sensibili: Quando una VM esegue un'istruzione sensibile, avviene una trap nel kernel dell'hypervisor. Quest'ultimo può esaminare l'istruzione e, se è stata inviata dalla VM del SO guest, la esegue; altrimenti, simula il comportamento dell'hardware reale.
* Esempio di Implementazione: I processori con tecnologia VT consentono a un hypervisor di tipo 1 di gestire le VM in modo efficiente.

**Hypervisor di Tipo 2:**

* Posizione e Modalità di Esecuzione: Si tratta di un programma utente eseguito su un sistema operativo host.
* Traduzione Binaria: Utilizza la tecnica della traduzione binaria per eseguire le istruzioni delle VM. Questo processo coinvolge la scansione del codice, la traduzione delle istruzioni sensibili in procedure VMware e l'esecuzione dei blocchi tradotti nella cache di VMware.
* Esempio di Implementazione: VMware è un esempio di hypervisor di tipo 2 che si esegue come programma utente su un SO host.

**Confronto tra Hypervisor:**

* Approccio Trap-and-Emulate (Tipo 1) vs Traduzione delle Istruzioni (Tipo 2): Tipo 1 adotta l'approccio "trap-and-emulate" che genera trap e un overhead gestionale, mentre il tipo 2 utilizza la traduzione delle istruzioni sensibili, risultando più efficiente.
* Istruzioni Sensibili e Gestione delle VM: Nel tipo 2, le istruzioni sensibili sono sostituite da chiamate a procedure che ne emulano il comportamento, mentre il tipo 1 può implementare anche la traduzione binaria, come il tipo 2.

**Paravirtualizzazione:**

* Approccio Modificato del SO Guest: Prevede la modifica del codice sorgente del SO guest, introducendo chiamate di procedure definite dall'hypervisor.
* Interfaccia e Performance: L'hypervisor definisce un'interfaccia (API) per i sistemi operativi guest, trasformandosi in un microkernel. Questo tipo di VM, chiamato paravirtualizzato, migliora le performance in quanto le trap vengono trasformate in system call.

In sintesi, le diverse tipologie di hypervisor si distinguono per l'approccio alla gestione delle istruzioni sensibili, la loro posizione e modalità di esecuzione e l'efficienza nell'operare con le VM, offrendo soluzioni e performance differenziate.

**ARM**

l'architettura ARM (Advanced RISC Machine) è una famiglia di microprocessori RISC (Reduced Instruction Set Computer) a 32-bit e 64-bit sviluppata da ARM Ltd. Questa architettura è ampiamente diffusa in dispositivi come cellulari, tablet, console per videogiochi, portatili e una vasta gamma di apparecchi elettronici, tra cui automobili, elettrodomestici e televisori.

La forza principale dell'architettura ARM risiede nella sua semplicità circuitale, che richiede un minor numero di transistor e di silicio, rendendolo più economico e con un basso consumo energetico rispetto alle sue prestazioni. Questo bilanciamento tra potenza di calcolo e risparmio energetico ha reso gli ARM molto adatti per dispositivi mobili.

Un altro fattore chiave del successo di ARM è il suo modello di business. ARM Ltd concede licenze del design del microprocessore ad altre aziende, permettendo loro di adattare il progetto alle proprie esigenze, modificandolo e producendo chip personalizzati. Questo approccio ha alimentato un'evoluzione rapida e l'ampia diffusione degli ARM in molteplici settori e applicazioni.

L'architettura ARM condivide le caratteristiche fondamentali di una tipica architettura RISC, progettata per ottimizzare l'emissione delle istruzioni. Per massimizzare il throughput delle istruzioni, un processore ARM dispone di un elevato numero di registri, tutti con la stessa dimensione, e esegue l'elaborazione dei dati direttamente nel processore senza l'intervento della memoria, accedendo ad essa solo per le operazioni di load/store.

La decodifica rapida delle istruzioni è favorita da un formato regolare e una lunghezza fissa delle istruzioni stesse, indipendentemente dal tipo di istruzione. Inoltre, l'indirizzamento efficiente richiede che il valore dell'indirizzo sia contenuto direttamente nell'istruzione (immediato) o in un registro del processore, evitando passaggi eccessivi.

In sintesi, l'architettura ARM si distingue per la sua efficienza, semplicità circuitale, basso consumo energetico e la capacità di adattamento attraverso la concessione di licenze, fattori che l'hanno resa dominante nei dispositivi mobili e diffusa in numerosi altri settori e applicazioni.

**Aspetti innovativi**:

* Un datapath specifico che permette di coniugare congiuntamente la funzione aritmetica e logica con quella di scorrimento su più bit in una sola istruzione;
* La modalità indirizzamento con auto incremento e auto decremento per ottimizzare i cicli nei programmi;
* La possibilitá di eseguire istruzione di load/store multiple per massimizzare il throughtput dei dati;
* Quasi tutte le istruzioni possono prevedere una pre-condizione che vincola l’esecuzione dell’istruzione stessa(esecuzione condizionale) che permette al processore ARM di compiere in un solo passaggio quello che per altre CPU richiede l’esecuzione di piú istruzioni.

**ARM Cortex – Ambiti**:

* **Applicazione** :architetture tradizionali con sistema di memoria virtuale MMU (memory management Unit)
* **Real-Time** : Architetture destinate a contesti real-time e un sistema di memoria protetta MPU (Memory protected Unit)
* **Microcontrollore**: Modello di programmazione per l’elaborazione veloce degli interrupt.

**ARMv6**:

Introduce definitivamente l’estensione Jazelle che permette di eseguire direttamente codice byte code trasformando di fatto il processo in una Java Virtual Machine. Il tutto in aggiunta allo stato Thumb del processore in cui utilizza un set di istruzioni a 16bit favorendone una cosi maggiore densità nelle applicazioni.

**Processore ARM**:

Il processore ARM utilizza un’architettura RISC a 32 bit, dimensione che coincide con quella dei registri, delle istruzioni e dei dati su cui operano tali istruzioni.  
Lo spazio di indirizzamento utilizzato dall’ARM è di 32 bit e la memoria è indirizzabile al singolo byte, a 16 (halfword) o 32 bit (word), in modalità allineata al byte. Quindi l’indirizzo di un dato a 16 bit deve essere multiplo di 2 byte, mentre di un dato a 32 bit multiplo di 4 byte.

Il processore ARM può indirizzare quindi fino a 2^32 locazioni di memoria sia nel formato little-endian (default) sia Big-endian, mentre accede al codice sempre in formato little-endian. Per selezionare la modalità Big-endian occorre asserire il pin BIGEND che verrà letto dalla CPU in fase di avvio della macchina. Tuttavia, la maggior parte dei dispositivi funziona in modalità little-endian ed è per questa ragione che è il valore predefinito di funzionamento del processore ARM.

Una caratteristica pregnante dell’architettura ARM è la lunghezza fissa a 32 bit delle istruzioni, questo permette di definire un set di istruzioni ortogonali cioè dotate di forma regolare facilmente decodificabile.

L’altro aspetto interessante e peculiare è la possibilità di condizionare l’esecuzione delle istruzioni in maniera preventiva in base al valore dei flag di stato. A differenza di altri linguaggi assembly, nel set delle istruzioni ARM non esistono istruzioni di shift o rotazione i bit se non innestate all’interno di altre istruzioni di elaborazione (aritmetiche, logiche, …) e, comunque, possono essere applicate ad un solo operando. Tuttavia, i processori ARM riescono ad eseguire una operazione aritmetica o logica e una di shift o di rotazione in una sola istruzione. Differentemente da altri linguaggi assembly, non esiste una istruzione per eseguire l’operazione di divisione, i progettisti dell’architettura hanno pensato di delegare questo compito al coprocessore che, avendo un ambito di applicazione specifico, puó risolvere al meglio il compito. In alternativa, si può sempre scrivere con una procedura assembly in grado di eseguire la divisione.