ale-cci

Architettura dei calcolatori elettronici

May 29, 2020

Contents

Introduzione

Calcolatore Elettronico

Un calcolatore elettronico è un sistema gerarchico suddiviso in elaborazione, memorizzazione, trasmissione e di controllo. Queste funzioni corrispondono gli elementi: CPU, memoria, sistema I/O e Bus.

La CPU (unità di controllo) è ulteriormente divisa in 4 parti:

- ALU: esegue le operazioni aritmetiche e logiche.
- Control Unit: comanda le unità del processore.
- Registri: memorie interne al processore, utilizzate per tenere temporaneamente i dati che il processore deve elaborare.
- Bus: Interconnessione interna per il trasferimento dati nel processore.

Architettura di Von Neumann

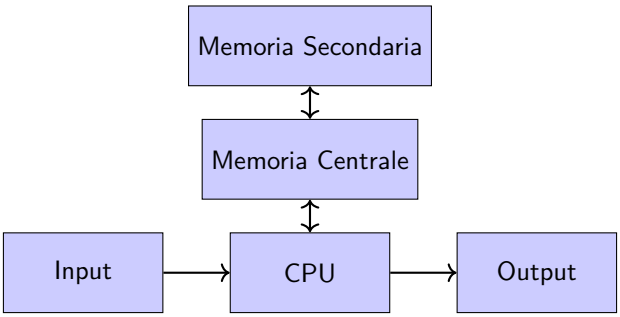
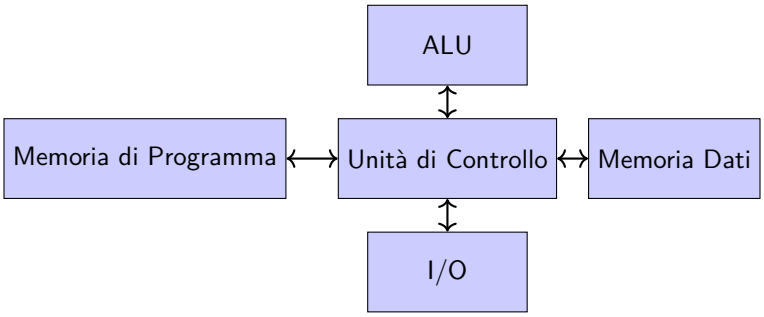


Figure 1: Computer secondo architettura di Von Neumann

La caratteristica principale dell'architettura è l'introduzione di una memoria interna. Precedentemente, i programmi erano salvati esternamente in schede perforate. La memoria centrale (RAM) è temporanea e fa da tramite alla memoria secondaria (HD), utilizzata per salvare permanentemente i dati.

Architettura di Harvard



Ideata ad Harvard, questa architettura è caratterizzata da una separazione tra la memoria del programma e la memoria dati. A causa di questa separazione, le istruzioni sono obbligate a passare attraverso la CPU. Questa architettura ha dato spunto a memorie separate a rapido accesso come la cache. L'idea di poter accedere separatamente a memoria del programma e memoria dati velocizza le prestazioni del calcolatore.

Leggi di Moore

1. Le prestazioni dei processori, e il numero di transistor ad esso relativo, raddoppiano ogni 18 mesi.
2. IL costo di una fabbrica di chip raddoppia da una generazione all'altra.

Legge di Amdahl

L'aumento progressivo della frequenza di clock e di transistor interni al processore non è sostenibile. Per questo si è iniziato a ragionare sul parallelismo.

Nel momento in cui si vuole parallelizzare un algoritmo, è possibile suddividere le istruzioni del programma in due gruppi: una componente sequenziale ed una parallelizzabile. Chiamata f la frazione di algoritmo parallelizzabile, ed N il numero di processori a disposizione, l'aumento di velocità di esecuzione S (*Speedup*) è calcolabile con la legge di Amdahl:

$$S = \frac{1}{(1-f) + \frac{f}{N}}$$

Da come si può evincere dalla formula, avere a disposizione un elevato numero di core per un algoritmo non parallelizzabile ($f = 0$), non porta alcun miglioramento:

$$\lim_{N \to +\infty} \frac{1}{1 + \frac{1}{N}} = 1$$

Il parallelismo è utilizzato in architetture pipeline, coprocessori paralleli (processori dedicati a specifiche operazioni) ed architetture multicore.

RISC vs CISC

ISA

L'ISA (*Instruction set Architecture*) di un processore, non è altro che la lista di istruzioni disponibili al programmatore, interpretabili dal processore. Un' istruzione dell'ISA inviata al processore, viene prima trasformata in comandi di microarchitettura (linguaggio macchina) e poi eseguita dall'hardware.

Un processore viene detto compatibile a livello di ISA con un altro processore, se tutte le sue istruzioni sono interpretabili da quest'ultimo.

La definizione di un ISA è la prima tra le diverse fasi della progettazione della CPU, ed in base a quanti comandi ne fanno parte, il processore si può definire di architettura CISC o RISC.

Architettura CISC

Un'ISA di tipo CISC (*Complex Instruction Set Computer*) è caratterizzata da un vasto numero di istruzioni a disposizione del programmatore, facilitandogli in questo modo la stesura del codice.

Il punto negativo di questa architettura è che la sua realizzazione, essendo ricca di features, risulta poco efficiente e dispendiosa dal punto di vista hardware. Inoltre le istruzioni utilizzate più di frequente dai programmatori sono un set ridotto (circa il 20%) di tutte le quelle a disposizione.

RISC

All'esatto opposto ci sono le ISA di tipo RISC (*Reduced Instruction Set Computer*).

Sono ISA che mettono a disposizione un numero ridotto e selezionato di istruzioni, portando il vantaggio di una realizzazione a livello di hardware più semplice e veloce. Come diretta conseguenza della semplificazione a livello di hardware si hanno tempi di esecuzione più rapidi rispetto alle architetture CISC.

Di contro, essendo il numero di istruzioni a disposizione ridotto, scrivere un programma solitamente risulta più tempo-dispendioso e complesso.

Confronto fra RISC e CISC

Altre differenze non ancora discusse su questi due tipi di architettura sono i seguenti:

RISC	CISC
Istruzioni di lunghezza fissa	Istruzioni di lunghezza variabile
Decodifica semplice	Decodifica complessa, a più cicli di clock
Unità di controllo cablata	Unità di controllo microprogrammata
Pochi metodi di indirizzamento	Svariati metodi di indirizzamento
Memoria allineata	Memoria non allineata
Molti registri di lunghezza fissa ed ortogonali	Pochi registri di varie lunghezze e non ortogonali
Processori load-store	

Memoria allineata

In una memoria allineata, i dati vengono disposti ad indirizzi multipli di n, portando il vantaggio di avere un rapido accesso alla memoria, dato che gli indirizzi sono semplici da calcolare.

Il difetto, come facilmente intuibile, si verifica nel caso di scrittura di dati di grandezza minore di n,

Ortogonalità registri

Registri e memoria sono collegati. Un'architettura RISC cerca di ridurre il più possibile gli accessi alla memoria, attraverso un numero ridotto di modalità di indirizzamento Un'architettura RISC ha poche modalità di indirizzamento e cerca di ridurre il più possibile gli accessi alla memoria. Per questo lavora con processori che comunicano con essa con le due sole operazioni `load` e `store`.

Per evitare ripetuti accessi, i dati vengono salvati temporaneamente su registri interni al processore stesso. Si definiscono ortogonali (caso RISC) se ogni registro può effettuare ogni operazione o non ortogonali nel caso in cui esistano registri specifici per specifiche operazioni (CISC).

Istruzioni

Il tempo impiegato al processore per eseguire un determinato programma è dipendente dal numero di cicli di clock che il processore impiega ad eseguire ogni istruzione. In altre parole, chiamato il tempo di clock T_{ck} e CPI_i il *clock per instruction* impiegato dall'istruzione N_i, il tempo di esecuzione è calcolabile come:

$$T_{CPU} = T_{ck} \sum_i (N_i \cdot CPI_i)$$

Le istruzioni a lunghezza fissa dell'architettura RISC, permettono di essere decodificate in un unico ciclo di clock. Inoltre, grazie alla logica hardware semplificata, tali architetture permettono di durata del tempo di clock T_{ck} minore.

Da come si può vedere in formula, entrambe queste caratteristiche portano una riduzione nel tempo di esecuzione complessivo di un programma.

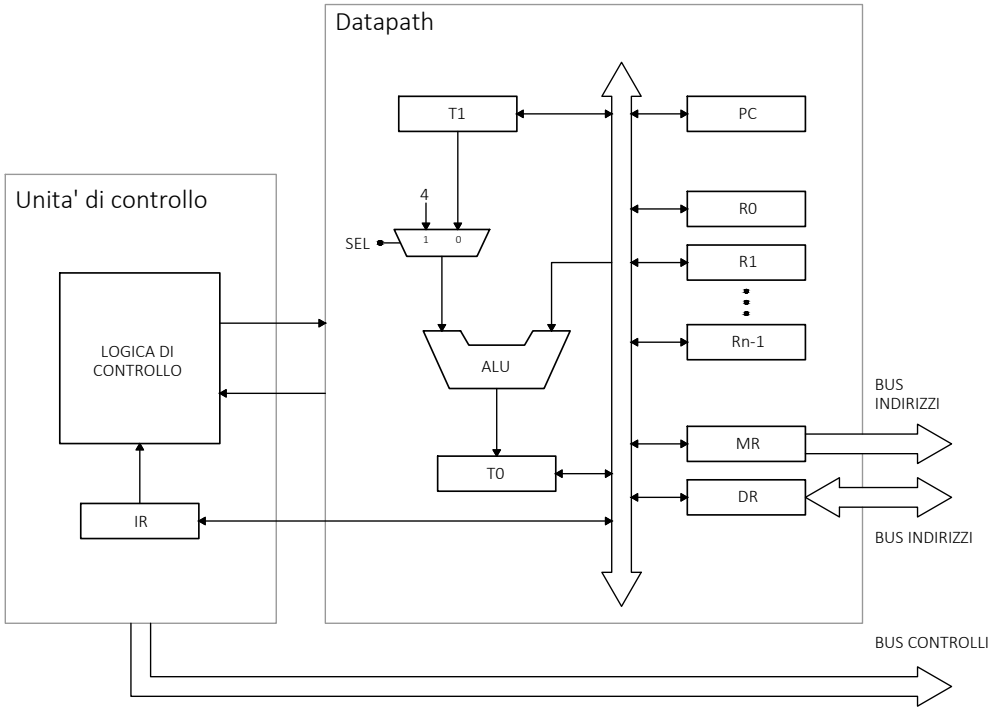
Microarchitettura CPU

La CPU

Una rete logica è definita **combinatoria** se dati degli ingressi, fornisce sempre gli stessi valori d'uscita, indipendenti dal tempo.
Diversamente una rete **sequenziale** è una macchina a stati finiti, dotata di memoria, quindi a valori in ingresso corrispondono uscite che dipendenti dallo stato attuale.

Dal punto di vista funzionale è possibile suddividere la CPU in due parti: data path e unità di controllo. Il data path, di cui componente fondamentale è l'ALU, è una rete logica che si limita ad eseguire istruzioni indicategli dall'unità di controllo, una rete sequenziale di stati: fetch, decode ed execute.

Architettura di riferimento RISC



Analizziamo adesso un esempio di architettura RISC.

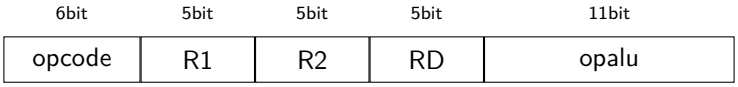
Un indirizzo indica la posizione di un byte in memoria, ad esempio l'indirizzo 2 indica il secondo byte, posizionato all'ottavo bit.
Le istruzioni sono di lunghezza fissa a 32 bit. Dato che istruzioni e dati sono salvati sempre ad indirizzi multipli di 4, il program counter è incrementato di 4 ad ogni istruzione. I registri, a 32bit, sono 32 e di uso generale. L'insieme dei registri prende il nome di register file.

In generale l'esecuzione all'interno di un processore passa attraverso 5 fasi: fetch, decode, execute, memory e writeback.
Nella prima fase (IF) il processore si occupa di leggere dalla memoria l'istruzione indicata dal program counter. Successivamente nella fase di decode (ID) viene decodificata e passata alla fase di execute

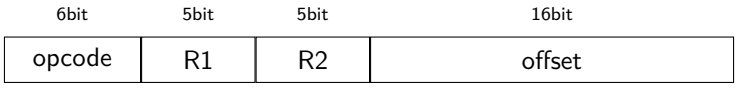
(EX). Una volta eseguita, in alcuni casi vengono scritti o letti dei dati dalla memoria (ME), ed infine dove necessario vi è la fase di writeback (WB) dove il risultato dell'operazione è scritta in un registro.

Ogni istruzione di questa ISA di riferimento può appartenere ad una delle tre seguenti categorie:

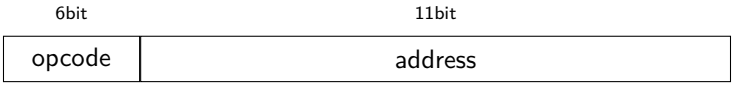
- Istruzione aritmetico/logica `add reg1, reg2, dest`



- accesso alla memoria/salto condizionato `je r2,r3,0045h`



- Salti incondizionati `jmp 0045h`



Oltre ai salti condizionati ed incondizionati esiste un terzo tipo di salto, la chiamata a funzione: un salto incondizionato che all'invocazione memorizza il valore corrente di `PC`, necessario per riprendere l'esecuzione una volta terminate le istruzioni della funzione.
Assumendo che in questa prima architettura non sia presente uno stack [1], chiamate a funzione salveranno il valore di `PC` nel registro `R31`.

Nel componente ALU, l'ingresso `OPALU` definisce l'operazione aritmetica da eseguire sui due segnali `ALUsorgA` e `ALUsorgB`. Oltre al risultato dell'operazione, in uscita sono presenti anche dei flag che ne descrivono le caratteristiche, come `ZF` (*Zero Flag*) e `SF` (*Sign Flag*) (vedi lista dei flag a pagina 18).

Load e Store

Load e store permettono rispettivamente lettura e scrittura in memoria. Entrambe prendono in ingresso un registro base, un registro destinazione ed un offset.
L'indirizzo di memoria in cui leggere o scrivere è calcolato sommando il contenuto del registro base con l'offset. Successivamente in caso di load il dato letto è scritto nel registro destinazione, ed in caso di store il dato è letto dal registro destinazione e scritto nell'indirizzo di memoria.

Per comunicare con la memoria si passa attraverso un unico bus, controllato da buffer 3state. Per leggere o scrivere è necessario abilitare i segnali `Mread` e `Mwrite`.

Register File

Prende come ingressi il primo registro (5bit), il secondo registro (5bit), il registro destinazione (5bit) e 32bit che identificano il dato in ingresso. In uscita ha due porte a 5bit che riportano i dati letti dal registro sorgente 1 e 2.

[1] Zona di memoria organizzata a pila LIFO, accessibile attraverso le istruzioni `push` e `pop`

CPU monociclo

In una CPU monociclo, tutte le istruzioni impiegano un unico ciclo di clock. È una struttura molto semplice, ma potenzialmente anche molto lenta, dato che il tempo di esecuzione di ogni istruzione necessita di essere pari al tempo di esecuzione di quella più lenta.

Facendo riferimento all'istruzione `st R6,R1, 20`, dato che in un unico ciclo di clock devo eseguire sia fetch dell'istruzione che store, sono necessarie due memorie separate. È inevitabile l'uso di un'architettura di Harvard (vedi pagina 1). Oltre alla doppia memoria introdotta dall'architettura è necessario duplicare anche altre risorse, come ad esempio un sommatore , richiesto per incrementare sia valore del program counter siccome l'ALU è già utilizzata in fase di execute.

Tra le varie operazioni dell'ISA, il tempo di esecuzione maggiore è dovuto sicuramente alla `load` (vedi tabella 1), che oltre alle fasi di fetch e decode richiede: calcoli dall'ALU, writeback ed accesso alla memoria. Chiamato il suo tempo di esecuzione $T_{\text{mono}} = 82ns$ ed N il numero di istruzioni, il tempo di esecuzione del programma è $T_{\text{mono}} \cdot N$.

	Fetch	Decode	ALU	Memory	Writeback	Totale
Aritm.	30	5	12		5	52
Load	30	5	12	30	5	82
Store	30	5	12	30		77
jmp cond.	30	5	12			47
jmp	30	5				35
jal/call	30	5			5	40

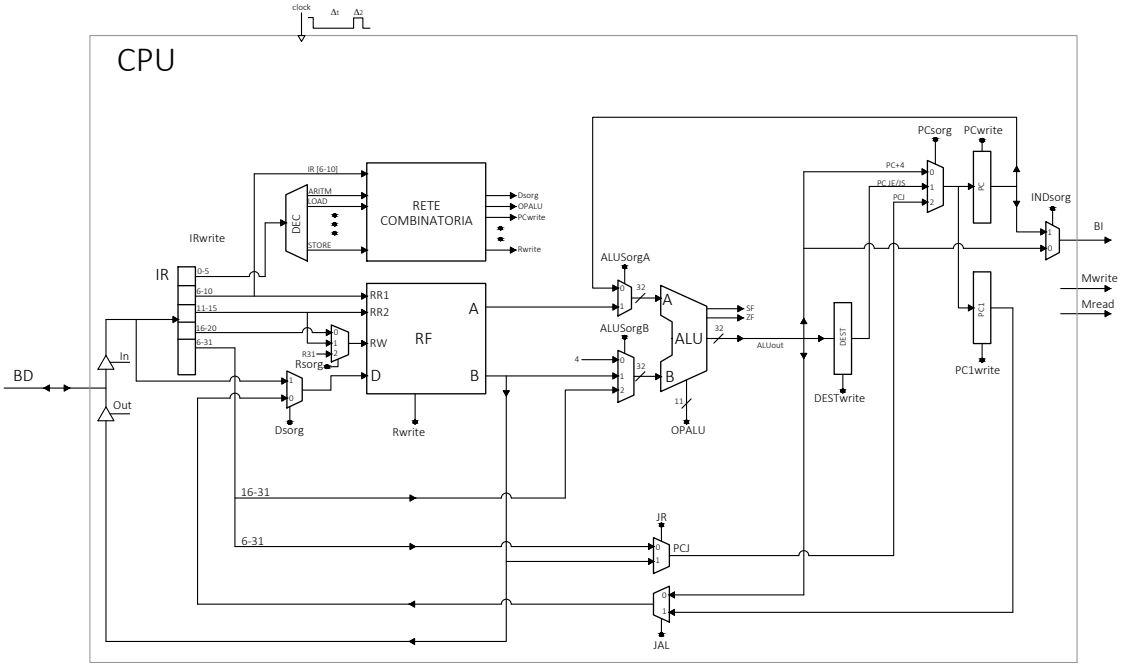
Table 1: Esempio tempi di esecuzione in architettura monociclo

CPU multiciclo

Ogni fase dell'istruzione è eseguita in un ciclo di clock differente. Alla base di questa architettura c'è il ragionamento che molte delle istruzioni dell'ISA richiedono un tempo di esecuzione notevolmente inferiore rispetto al resto delle istruzioni. In altre parole, chiamato $\sum T_{\text{multi}}$ il tempo di esecuzione di una singola istruzione il numero di istruzioni per cui $\sum T_{\text{multi}} > T_{\text{mono}}$ sarà inferiore al numero di istruzioni per cui $\sum T_{\text{multi}} < T_{\text{mono}}$. Ne consegue che l'architettura multiciclo è mediamente più veloce di un architettura monociclo.

Un vantaggio che porta quest'architettura è che non necessita più di componenti come sommatore e memoria secondaria, dato che le diverse fasi dell'istruzione sono divise in diversi cicli di clock evitando conflitti di risorse.

- Fetch: Dalla memoria viene letta l'istruzione indicata dal program counter. L'ALU incrementa il valore di quest'ultima per leggere la prossima istruzione al ciclo successivo.
- Decode: Il codice operativo indicato dell'istruzione entra in `IR`, viene assegnato il valore ai registri `R1` ed `R2`. L'ALU viene impiegata per calcolare il registro destinazione.
  In caso di salto condizionato, si calcola ugualmente il valore dell'indirizzo di destinazione, senza verificare le condizioni del salto.
- Operazione Aritmetica: Esegue l'istruzione utilizzando i dati calcolati nelle fasi precedenti.
- Memory attraverso le istruzioni `load` e `store`: vengono abilitati i segnali dei buffer 3state per lettura o scrittura dalla memoria all'indirizzo `Rd` calcolato nelle fasi precedenti.
- Writeback: In caso di operazione aritmetica prendo il valore in uscita dell'ALU e lo riporto nella porta dati del register file. In caso di load, il valore di uscita della memoria viene riportato nella porta dati del register file. In caso di JAL PC1 (valore precedente di PC) entra nel register file.



Interfacce con l'esterno: bus di dati attraverso buffer 3-state, in uscita al bus degli indirizzi arriva o alu-out o pc.

Segnali di controllo architettura multiciclo

La fase di instruction fetch, comune a tutte le istruzioni, richiede di leggere l'istruzione corrente dalla memoria ed incrementare il program counter di 4. Per queste operazioni necessita dei segnali:

`Mread`	Memory Read	`INDsorg = 1`	invia al buffer in uscita `PC`
`In`	abilitazione buffer 3state	`ALUsorgA = 0`	`PC` come ingresso `A` dell'ALU
`PCwrite`	program counter write	`ALUsorgB = 0`	`4` come ingresso `B` dell'ALU
`PC1write`	program counter 1 write	`OPALU = ADD`	operazione somma
`IRwrite`	instruction register write	`PCsorg = 0`	aggiorna `PC` in `PC + 4`

Nella fase di decode è richiesto di decodificare il codice operativo, inviare i due registri sorgente al register file e sfrutto l'ALU non utilizzata per calcolare in anticipo l'indirizzo di destinazione gli eventuali salti condizionati:

`DESTwrite`	abilitazione scrittura nel registro destinazione
`ALUsorgA=0`	`PC` in ingresso
`ALUsorbB=2`	`offset` dell'istruzione in ingresso
`OPALU = ADD`	operazione somma

Passate queste due fasi, comuni a tutte le operazioni, vi è una differenziazione a seconda del tipo di istruzione contenuta in `IR`.

In caso di istruzioni aritmetiche, composte da execute (T3) e writeback (T5), in prima fase, sono richiesti i segnali `ALUsorgA=1`, `ALUsorgB=2` ed `OPALU` per specificare le operazioni dell'ALU; in fase di writeback: `Rwrite` e `Rsorg = 0` per scrivere il contenuto del registro destinazione in `RF`, `DESTwrite = 0` per abilitare scrittura di `ALUout` in `dest`. L'istruzione passa anche attraverso una fase di memory (T4) ma vengono solamente mantenuti i valori dei segnali in T3, mantenendo così `ALUout` costante.

In caso di load, nella fase di execute, viene calcolato l'indirizzo del dato in memoria `Rb + offset`

(`ALUsorgA=1`, `ALUsorgB=2`, `OPALU=ADD`) e salvato in `Rdest`. In fase T4, sono tenuti costanti i segnali in T3, inoltre `INDsorg = 0` e `Mr =1` per leggere il valore del dato all'indirizzo di memoria calcolato precedentemente. A T5 viene salvato in `Rd` il dato letto dalla memoria `In=1` per mettere il bus dei dati in ingresso, viene abilitato `Rwrite` per abilitare la scrittura al register file. `Rsorg=1` e `Dsorg=1` per scrivere il registro `Rd` il dato proveniente dalla memoria.

In caso di store, i segnali nella fase di execute sono identici a quelli della load, ma eseguo `INDsorg=0` uno step in anticipo, per avere il segnale stabile alla fase successiva e `Out=1`. In T4 `Mw=1` per scrivere in memoria. Non ha una fase di writeback.

I salti condizionati richiedono solo un ciclo di completamento, in quanto l'indirizzo di destinazione è già calcolato nella fase precedente, è necessario solo da abilitare `PCwrite` in caso la condizione di salto sia verificata. (`ALUsorgA = 1`, `ALUsorgB=1`, `OPALU=SUB`) `PCsorg=1` è richiesto per selezionare `DEST` come ingresso a PC.

In caso di salti incondizionati è presente la sola fase T3 con `PCwrite=1` e `PCsorg=2`.

L'ultimo caso `jal`, è del tutto identico ad un salto condizionato, solo che viene seguito da una fase di writeback, in cui salvare `PC1` in `R31`.

Prestazioni dell'architettura multiciclo

Diversamente dall'architettura monociclo il cui T_{mono} corrisponde al tempo di esecuzione dell'istruzione più lenta, il tempo T_{multi} è pari al tempo impiegato dallo stadio più lento.
Prendendo come riferimento la tabella dei tempi di esecuzione dell'architettura monociclo (vedi pagina 7), ne consegue $T_{\text{multi}} = 30$
Confrontando le varie fasi ottengo chiaramente che i tempi di esecuzione delle singole istruzioni, ed il tempo di esecuzione medio è peggiore. Questo è dovuto ai $30ns$ della fase di fetch, che occupano più di $1/3$ del tempo di esecuzione di una singola istruzione.

Name	cck	Time	Usage
Aritm	5	150ns	40%
Load	5	150ns	25%
Store	4	120ns	10%
JE/JS	3	90ns	12%
JMP/JR	3	90ns	6%
JAL	5	150ns	2%

Miglioramenti architettura multiciclo

Il tempo di esecuzione medio dell'architettura multiciclo appena vista, $180ns$, è superiore agli $82ns$ dell'architettura monociclo. Per migliorarlo sono applicabili 3 metodi:

- Ridurre il periodo di clock, introducendo salti di attesa per le fasi più lunghe
- Sfruttare le componenti inutilizzate, calcolando in anticipo operazioni richieste in fasi successive
- Unire fasi distinte e modificare opportunamente il segnale di clock

Aumento granularità del clock

T_{multi} uguale al tempo d'accesso alla memoria risulta eccessivo per le fasi in cui essa è inutilizzata. Per questa architettura, il caso ideale sarebbe che tutti gli stadi richiedessero approssimativamente lo stesso tempo di esecuzione.

Prendendo $T_{\text{multi}} = 5ns$ (tempo di decodifica della fase di writeback) le fasi di accesso alla memoria saranno eseguite in 6 cicli di clock ($6 \times 5 \geq 30$). $T_{\text{multi}} = 5ns$ migliora le prestazioni dell'architettura, riducendo il tempo medio di esecuzione a $75.65ns$.

Anticipazione delle operazioni

Si può notare come l'indirizzo destinazione, calcolato in fase di decode, è utilizzato solo in caso di salti condizionati (istruzioni JE o JS). Possiamo quindi differenziare la fase T2 a seconda dell'istruzione da eseguire. Inoltre, dato che il periodo è sufficientemente lungo, possiamo eseguire brevi fasi vicine in un unico ciclo di clock:

- Le operazioni aritmetiche, dopo la fase di fetch, richiedono la decodifica ($5ns$), un'operazione di ALU ($12ns$) e il writeback ($5ns$), con un totale di $22ns$, sufficiente ad eseguirle in un unico ciclo di clock di $30ns$.
- Per l'istruzione load, il calcolo dell'indirizzo destinazione può essere effettuato in T2, la lettura dalla memoria in T3 ed il writeback sul registro in T4, rimuovendo la necessità di T5.
- L'operazione di store, esattamente come per load, è possibile anticipare il calcolo dell'indirizzo sorgente in T2, di conseguenza la scrittura in memoria è effettuabile immediatamente all'istante T3.
- Per i salti (`JMP` e `JS`) è possibile aggiornare il valore di `PC` direttamente in T2
- Mentre per la chiamata a funzione (`JAL`) è possibile aggiornare `PC` e la scrittura su `R31` nello stesso ciclo di clock, rendendo `PC1` non più necessario.

Il tempo di esecuzione medio di questa nuova architettura è $81.60ns$, nettamente migliore rispetto alla versione multiciclo originale.

Aumentando la granularità di clock a $T_{\text{multi}} = 6ns$ (tempo impiegato dalle nuove fasi di instruction fetch e memory), si ottiene un tempo medio di $64.76ns$: il 21% più veloce dell'architettura monociclo.

Architetture Avanzate

Prestazione dei calcolatori

Con benchmark si definisce un set di programmi differenti che rappresentano a grandi linee task frequenti eseguiti dal calcolatore. Per confrontare le prestazioni di diversi calcolatori, viene eseguito lo stesso benchmark e si paragonano i diversi tempi di esecuzione.

Prestazione della CPU

Sull'intero tempo di esecuzione di un programma, T_{cpu} (*Tempo di CPU*) è definito come l'effettivo tempo in cui la CPU è impiegata nell'esecuzione del task: $T_{\text{cpu}} = N_{cc} T_{ck}$, dove N_{cc} è il numero di cicli di clock.
Il metodo più semplice per calcolarlo è attraverso il CPI (*Clock Per Instruction*) , ovvero il numero di cicli di clock mediamente richiesti da un istruzione: $\textit{CPI} = N_{cc}/N$ dove N è il numero di istruzioni in un programma.

Per calcolare il CPI medio, occorre conoscere il CPI di ogni istruzione, e la frequenza con la quale l'istruzione i-esima viene eseguita F_i.

$$\textit{CPI} = \sum_i F_i \times \textit{CPI}_i = \sum_i \frac{N_i}{N} \textit{CPI}_i$$

Da cui:

$$T_{\text{cpu}} = N \times \textit{CPI} \times T_{ck}$$

Le architetture RISC, si concentrano nel ridurre il più possibile $\textit{CPI}_i$, riducendo il numero di cicli di clock richiesti da ogni istruzione. Il numero ridotto di istruzioni dell'architettura, si risente nella scrittura dei programmi, dove, per svolgere lo stesso compito, più istruzioni sono richieste, aumentando il CPI.

MIPS (*Mega Instruction Per Second*) e MFLOPS (*Mega FLoating Point Operation Per Second*) definite come $\textit{MIPS} = N/(\textit{CPU}_{\textit{time}} * 10^6) = f_{\textit{ck}}/\textit{CPI}$ sono utilizzate come unità per misurare le prestazioni. Dipendono entrambe dal *CPI* medio e quindi dal benchmark.

Il numero di MIPS non dipende da N, pertanto a parità MIPS e benchmark otteniamo valori diversi di $\textit{CPU}_{\textit{time}}$ se N cambia. Ne consegue che due CPU sono paragonabili con lo stesso benchmark solamente se hanno lo stesso set di istruzioni.

Architetture Pipeline

Questa architettura è la soluzione più efficace per aumentare la velocità di una CPU, aumentando il numero di istruzioni eseguite nell'unità di tempo (throughput). Diversamente dall'architettura multiciclo, tiene le istruzioni separate in più cicli di clock, rendendone possibile la loro esecuzione in parallelo. I segnali sono trasmessi di fase in fase attraverso registri di latch.

In generale, chiamato τ il tempo di clock, supponendo la divisione in k stadi, n istruzioni vengono elaborate in $T_k = (k + (n-1)) \cdot \tau$.
L'aumento di velocità (*Speedup*) è esprimibile come:

$$S_p = \frac{T_1}{T_k} = \frac{nk\tau}{(k+(n-1))\tau} = \frac{nk}{k+n-1}$$

È facilmente osservabile come come al crescere del numero di stadi k, S_p tende ad n: tutte le istruzioni sarebbero eseguite allo stesso istante. Purtroppo questo è possibile solo nel caso ideale.

Pipeline non ideale

Esistono tre limiti all'aumento del numero di stadi nella pipeline:

- Alee strutturali: due fasi richiedono la stessa risorsa, ad esempio se fetch ed execute sono in esecuzione allo stesso tempo, la richiesta contemporanea dell'ALU crea un conflitto di risorse.
- Alee di dato: l'output di un'istruzione dipende da un risultato non ancora prodotto
- Alee di controllo: si verificano nel caso di jump, quando non è ancora stato determinato l'indirizzo di destinazione

Alee strutturali

Un'immediata soluzione per questo tipo di alee è la duplicazione di risorse richieste (es. due ALU) e l'utilizzo di un architettura di Harvard per supportare multipli accessi alla memoria.

Un'altra soluzione è ritardare l'esecuzione delle fasi che richiedono una risorsa già in uso, da evitare perché porta una riduzione del throughput.

Alee di dato

Chiamate A e B, due istruzioni, dove A precede B, esistono tre tipi di dipendenza:

- RAW (*read-after-write*) B legge un dato prima che sia scritto da A
- WAR (*write-after-read*) B scrive un dato prima che sia letto da A
- WAW (*write-after-write*) B tenta di scrivere un dato prima che A lo abbia scritto

Nel primo caso, prendendo come esempio le istruzioni in successione `add r1,r2,r3` e `sub r4,r1,r2`, il valore di r1 è richiesto dalla seconda istruzione nella fase di execute, ma è salvato dalla prima istruzione solo in fase di writeback.

Un metodo possibile per individuare questo tipo di alea è tenere marcato i registri in utilizzo ed controllare di volta in volta i registri richiesti dall'istruzione corrente.
Le possibili soluzioni sono le seguenti:

- Stallo: attendere che una delle due istruzioni termini
- Anticipazione: rendere immediatamente disponibile il dato, senza attendere la fase di WB. Ma risulta un metodo costoso e non banale da implementare.
- Sovrapposizione: Produco il risultato nel fronte di clock di salita e lo leggo nel fronte di discesa (half-clock). Non sempre raddoppiare la frequenza di clock risulta possibile.
- Riordinamento: Vengono eseguite delle istruzioni ortogonali [2], eseguendo cosi la seconda istruzione solo dopo che la prima abbia effettuato il WB

[2] non hanno conflitto con le altre istruzioni e non alterano l'output del programma

Alee di controllo

In caso di jump, il valore del program counter è noto solo in fase di decode, non sapendo a priori da dove continuare l'esecuzione si ha un alea di controllo. È risolvibile introducendo un delay di 1 ciclo di clock, dando il tempo alla CPU di calcolare il valore del program counter, o riordinando le istruzioni, anticipando la decodifica dell'indirizzo di un'istruzione.

Il problema maggiore delle alee di controllo è introdotto dai salti condizionali, nei quali è necessario controllare anche la condizione di salto. In questo caso, il semplice riordinamento delle istruzioni non è possibile.
Per minimizzare il numero di cicli "idle" sono utilizzate tecniche di predizione, per stimare in anticipo se la condizione di salto si verifica. Se queste euristiche funzionano più del 50% delle volte, si misura un aumento di prestazione.

Le tecniche possono essere statiche (i salti si verifichino sempre o che non si verifichino mai), o dinamiche: basate sul comportamento precedente del salto (vedi esempio in figura 2).

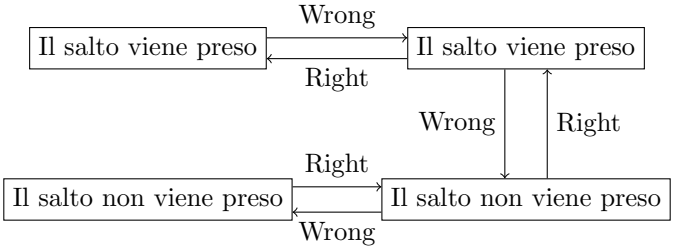


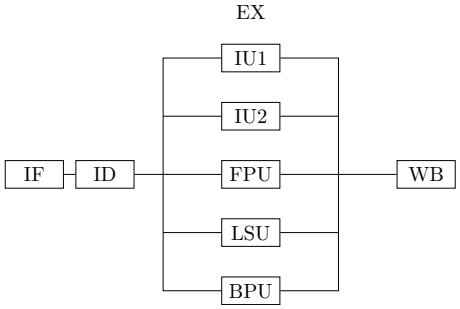
Figure 2: Esempio predizione dinamica

Queste predizioni sono salvate in una tabella associativa, dove ad ogni indirizzo di program counter del salto è associato il valore di relativa predizione.

Architetture superscalari

L'architettura appena vista prende il nome di pipeline lineare, non è più utilizzata perché, come visto, lo speedup teorico è ben diverso dal reale. Un altro problema dell'architettura è che ogni istruzione attraversa tutti gli stadi, imponendo un periodo di clock coincidente con il tempo dello stadio più lento (uguale alla multiciclo originale).

La soluzione è quindi introdurre non un parziale parallelismo ma un parallelismo totale, sovrapponendo specifiche fasi delle istruzioni.
Nell'architettura che utilizzeremo come riferimento, solo la fase di execute è in parallelo:



- IU1= ALU, per aritmetica intera (1 cck)
- IU2=ALU, per aritmetica intera, ma operazioni più complesse come moltiplicazione e divisione (2cck)
- FPU=ALU, per virgola mobile (4cck)
- BPU *Branch Prediction Unit*
- LSU *Load Store Unit*

Figure 3: Architettura parallelismo di riferimento

Ipotizzando che allo stadio ID (*Instruction Decode*) sia emessa una sola istruzione alla volta, appena messa in esecuzione è eseguita in parallelo con le altre fasi di execute già in esecuzione.

Problemi dell'architettura

1. Date due istruzioni i e j, se con i che precede j, se i impiega più cicli di clock di j ad essere eseguita, l'ordine di completamento può risultare invertito.
2. Può accadere che due istruzioni richiedano nello stesso istante di passare alla fase di writeback, generando un conflitto di risorse.

Per risolvere il problema 1 esistono tre metodi: il completamento in ordine, dove le istruzioni sono completate secondo l'ordine prestabilito; il buffer di riordinamento, dove le istruzioni una volta completate scrivono il loro output su un buffer in cui vengono successivamente riordinate; ed attraverso l'history buffer, dove lo stato coerente può essere ripristinato in presenza di conflitti.

Reservation Shift register

Il primo metodo, meno efficiente, è realizzabile attraverso un RSR (*Reservation Shift Register*). Per la scrittura in tale registro è utilizzato un metodo di prenotazione. L'RSR è una tabella di colonne:

- V, bit di validità che indica se la posizione corrente contiene informazioni,
- PC il program counter dell'istruzione, necessario per il ripristinare uno stato coerente in caso di predizione di salto errata
- UF (*Unità Funzionale*) componente che sta eseguendo l'istruzione
- R_d Registro destinazione del risultato

Il riordinamento richiede diversi cicli di clock, e non risolve il problema 2 nel caso di multiple richieste d'accesso alla memoria, in quanto posso avere una scrittura seguita da una lettura e riscontrare un alea di dato.

Per risolvere questo problema o non si emettono comandi di memorizzazione prima che le istruzioni emesse precedentemente siano completate, o si considera la memoria come un'unità funzionale quindi store occupa una posizione in RSR in modo che questa raggiunga la cima quando tutte le istruzioni precedenti sono completate.

Reordering Buffer

Il ROB (*ReOrdering Buffer*) non è in grado di risolvere i problemi d'accesso al buffer, per questo viene utilizzato un RSR in versione semplificata.

Nella tabella RSR, vengono tenute solo le colonne V ed UF ed aggiunto $pROB$, un puntatore alla riga della tabella ROB in cui è inserita l'unità funzionale.

Nella tabella ROB, vengono spostati PC e R_d rimossi dal RSR, in aggiunta ad un bit di completamento C (1=istruzione completata) e RIS risultato temporaneo della istruzione.

È gestito come un buffer circolare. Il writeback viene eseguito quando l'elemento puntato dalla testa del buffer circolare è segnato come completato.

History Buffer

È la soluzione più flessibile delle tre, permettendo il completamento delle operazioni fuori ordine. L' History Buffer tiene traccia di tutte le scritture sul register file, fornendo un modo di effettuare un rollback in caso di stato di incoerenza.

HB è gestito come il rob: un buffer circolare di cui ogni riga è costituita dai componenti C, PC, R_d e OLD che contiene il valore del registro di destinazione al momento dell'operazione.

Se alle componenti di HA si aggiunge un ulteriore campo EPR (*Errata Previsione*). Quando un'istruzione di salto arriva in testa ad HB, se EPR=1: Blocco l'emissione di nuovi valori, attendo operazioni ancora attive che vengano completate (svuotamento pipeline), eseguo il rollback fino ad arrivare alla prima istruzione sul percorso sbagliato. L'esecuzione riprende prelevando l'istruzione proveniente dal percorso corretto.

Considerazioni

Abbiamo assunto l'emissione ed il ritiro di un'unica istruzione alla volta. Se viene emessa una sola istruzione alla volta non accadrà mai che vengano eseguite più di un istruzione per clock. Occorre quindi ritirare, decodificare ed emettere più istruzioni in parallelo.

Introduzione ai linguaggi Assembly

Per evitare ai programmatori di ricordarsi a memoria il linguaggio macchina, ogni microprocessore ha un proprio linguaggio assembly, in grado di tradurre con una corrispondenza biunivoca le istruzioni a basso livello o indirizzi di memoria in codice macchina.

Con statement o pseudo-istruzione si intende una riga del programma assembly. A tale riga corrisponde una direttiva dell'assemblatore. Inoltre se a tale direttiva corrisponde un'istruzione in linguaggio macchina, prende il nome di istruzione.

Ogni istruzione è composta da un'etichetta (label) che rappresenta l'indirizzo di memoria in cui l'istruzione è memorizzata, un codice operativo (opcode) simbolo mnemonico per l'operazione e da nessuno, uno o più operandi.

```
label:      mov ax, bx
            jmp label
```

Le etichette sono sostituite automaticamente dall'assemblatore in indirizzi di memoria. Permettono di astrarre gli indirizzi fisici, semplificando la modifica e comprensione del programma.

I codici operativi (es. `mov`, `add` ...) sono gli alias dati alle istruzioni eseguibili dalla CPU.

Gli operandi funzionano come argomenti passati ai codici operativi (nel caso di assembly 8086 sono al massimo 2). Durante l'esecuzione del programma, la CPU provvede a reperire il valore degli operandi, che può essere passato direttamente all'istruzione per valore, tramite registro, contenuto in memoria o da una porta di I/O.

Le pseudo-istruzioni sono utilizzate durante il processo di assemblaggio: esempi sono i segmenti dati, commenti e le macro.

Linguaggio assembly 8086

Specifico per il processore general purpose a 16 bit Intel 8086. Ha 14 registri interni a 16 bit, 7 modi di indirizzamento con capacità di 1Mb.

n	parallelismo del processore	16
n_a	parallelismo della memoria	20
n_d	parallelismo del buffer di dati	16

Table 2: Parallelismi del processore Intel 8086

Tutti i processori Intel sono backward-compatible, per questo i concetti di base per questo processore sono gli stessi utilizzati da un processore moderno.

Il processore 8086 è diviso in due unità funzionali concettualmente separate: la BIU (*Bus Interface Unit*) ed EU (*Execution Unit*). Come dice il nome, la BIU si interfaccia con il bus dei dati attraverso un unico bus in ingresso controllato dal *Bus Control*. L'EU non ha un collegamento diretto con la memoria ed è dedicata ai calcoli.

Siccome il fetch di è in media media più veloce dell'esecuzione, le istruzioni, una volta prelevate dal bus, vengono salvate temporaneamente in una coda (*coda di prefetch*) in attesa di essere processate dalla EU.

Inoltre questa architettura la BIU ha un sommatore dedicato utilizzato per calcolare il valore dell'IP o l'indirizzo di accesso per i dati in memoria.

EU Control si occupa di decodificare le istruzioni e genera i segnali necessari al resto dei componenti nella EU: op-alu, abilitazioni dei registri (temporanei, generali e di flag). Flag register descrive lo stato dell'ultima operazione effettuata dall'ALU (vedi flag a pag. 18)

Scelte progettuali di un ISA

- dove sono memorizzati gli operandi nella CPU
- con che istruzioni si accede agli operandi
- modello di memoria
- formato delle istruzioni
- modello di indirizzamento (come indicare gli indirizzi di memoria)
- tipo e struttura degli operandi
- tipo di istruzioni previste

Dove sono memorizzati gli operandi nella CPU

Un metodo per memorizzare gli operandi è attraverso lo stack, in questo modo viene garantita un'indipendenza dal register set, ma con lo svantaggio di avere difficoltà di accesso agli operandi. Inoltre dato che il tempo di accesso allo stack è elevato rispetto al tempo di esecuzione si forma un bottleneck.

Un altro metodo è l'utilizzo di un unico registro accumulatore, da cui passano tutte le operazioni. La gestione diventa molto semplice ma è ovvia la formazione di bottleneck.

Il metodo più utilizzato è a set di registri, ovvero gli operandi vengono salvati direttamente nei registri del processore. In particolare, nella CPU 8086, i registri possono essere di tre categorie: general purpose (generici non ortogonali), segment o miscellaneous. I registri sono `ax`, `bx`, `cx`, `dx`, `si`, `di`, `bp`, `sp`, tutti a 16 bit. Per i primi 4 è possibile accedere agli 8 bit più e meno significativi, sostituendo alla `x` una `h` o una `l` rispettivamente (es. `ah` per accedere ai primi 8 del registro `ax`).

Registri generici	
`ax`	Accumulatore: registro di base per le operazioni
`bx`	Base: unico utilizzato per indirizzamenti di memoria
`cx`	Conteggio: utilizzato per cicli
`dx`	Dati: utilizzato per indirizzi di istruzioni i/o o overflow
`si` `di`	source e destination per operazioni con stringhe di byte
`bp`	Base Pointer accesso a parametri e variabili di funzioni.
`sp`	Stack Pointer: puntatore a top dello stack

Registri speciali	
IP	Instruction Pointer
FLAG	Register flag

Overflow flag	Operazione ha un risultato troppo grande
Direction	Indica se incrementare / decrementare per le istruzioni con stringhe
Interrupt Enable	Abilita/Disabilita mascheramento degli interrupt
Trap	Utilizzato dai debugger, genera un int 3 dopo ogni istruzione
Sign	1 quando il risultato è negativo
Zero	il risultato dell'operazione è 0
Auxiliary Carry	1 quando è presente un riporto tra la parte alta e la parte bassa del registro.
Parity Flag	1 quando il numero di bit è pari, 0 quando è dispari
Carry Flag	Riporto o prestito nella parte alta dell'ultimo risultato.

Table 3: Flag architettura

Modelli di Memoria

Accesso alla memoria

Per scegliere gli operandi con i quali accedere alla memoria, si dividono le ISA in base a due numeri: il **numero di riferimenti diretti in memoria** indicati nelle istruzioni dell ALU ed il **numero di operandi indicati in modo esplicito nelle istruzioni**. Entrambi possono assumere solo valori compresi tra 0 e 3 inclusi.

Ad esempio nelle architetture chiamate *register-register*, il numero di riferimenti diretti in memoria è 0, ed il numero di operandi che indica in modi indicati in modo esplicito è 3. In altre parole le uniche operazioni che possono accedere in memoria sono LOAD e STORE.
Il numero di operandi indicati in modo esplicito indica il numero massimo di operandi specificati in modo esplicito come parametri di una funzione.

Quindi per effettuare un operazione come `c = a + b` sono necessarie le operazioni:

```
load    r1, var1
load    r2, var2
add     r1, r2, r3
store   var3, r3
```

Utilizzando lo stesso esempio per una architettura *register-memory*, (1, 2) otteniamo:

```
mov     AX, var1
add     AX, var2            ; AX funziona sia da sorgente
                            ; che destinazione
mov     var3, AX
```

In questo caso posso avere al massimo solo un operando che fa riferimento alla memoria.

Un' ultimo esempio di architettura è la *memory-memory* (3, 3) dove sia sorgente che destinazione sono completamente esplicitati.

Ordinamento della memoria

La memoria è sempre organizzata come un lungo array di celle a 8bit. Quando un dato di lunghezza più grande di 8bit deve essere salvato in memoria, può essere utilizzata sia la codifica **little endian** (memorizza l' Least Significant Bit all'indirizzo più basso), sia la **big endian** (all'indirizzo più basso viene salvato il Most Significant Bit).

Allineamento della memoria

Se la memoria è forzata a salvare i dati in modo allineato, allora riesco a leggere i dati di grandezza maggiore di 1 byte in un **singolo ciclo**. L'unico svantaggio è che per salvare dei dati di grandezza inferiore 4byte, avrò delle celle inutilizzate.
Se la memoria non è allineata, risparmio più spazio in memoria, ma l'accesso può richiedere più di un ciclo di CPU.

Memoria lineare e segmentata

Si definisce con **effective address**, l' indirizzo reale in memoria.
La **riallocazione della memoria** (RAM) si intende il riordinamento dei blocchi di memoria, in modo da raggruppare un unico blocco di memoria, tutti i blocchi non utilizzati, isolati dalla memoria in uso.

Il problema che porta con se la riallocazione di memoria, è che una volta che un blocco di memoria viene spostato, tutti gli effective address utilizzati nel codice contenuto al suo interno sono invalidati, e devono essere aggiornati uno ad uno dal processore con il nuovo effective address.
Per questo motivo alcune ISA preferiscono utilizzare un modello di memoria segmentato, in cui il codice utilizza **indirizzi relativi** anziché lavorare direttamente con gli effective address. Per accedere alla memoria attraverso un indirizzo relativo, vengono salvati in due registri (CS: Code Segment e DS: Data Segment) l'indirizzo di memoria da cui partono codice e dati del programma. Al momento di una riallocazione, per un modello di memoria segmentato, l'unica cosa invalidata sono i due registri segmento di ciascun programma.

Modello di memoria Intel 8086

Nel caso di Intel 8086, la memoria viene vista come un gruppo di paragrafi e segmenti: i **paragrafi** sono una zona di memoria a 16bit, i quali non si possono sovrapporre, mentre un **segmento** è un unita logica indipendente formata da locazioni continue di memoria, di dimensione massima 64k, ha inizio ad un indirizzo di memoria multiplo di 16, (in modo da essere allineato ad un paragrafo) ed a differenza dei paragrafi, sono sovrapponibili.

La dimensione massima di un segmento (64k) deriva direttamente dalla dimensione massima che può avere un indirizzo relativo. Dato che l'accesso ad un indirizzo avviene attraverso i registri, la dimensione massima è 2^n, e per Intel 8086: $n = 16 \Rightarrow 2^{16} = 64k$.

L'indirizzo di inizio di un segmento è salvato in un indirizzo di memoria a 20bit, ed è ottenuto da un registro a 16bit moltiplicato per 16.

La sovrapposizione dei segmenti era utilizzata in DOS nel tipo di eseguibile '.com', file che utilizzavano il modello di memoria 'tiny'. Prevedeva un unico segmento a cui corrispondevano DS SS (Stack Segment) e CS. I segmenti erano sovrapposti solo come indirizzo, non come dati. Siccome tutto era contenuto in un unico segmento, tra codice, dati e stack non era concesso di superare i 64k.

Modalità di Indirizzamento

Formato di Istruzione

Per definire un' istruzione all'interno del linguaggio, è necessario definire: il **codice operativo** (numero operandi espliciti), gli operandi ed il risultato e l'indirizzo della prossima istruzione. Per salvare tutte queste caratteristiche in memoria, diventa fondamentale definire un formato in cui codificare e decodificare l'istruzione.

Modalità di indirizzamento

La modalità di indirizzamento decide come indicare l'indirizzo in memoria in cui prendere i dati.

Indipendentemente dal tipo di memoria utilizzata (lineare, segmentata...) e dal tipo di operazione da effettuare, per indicare all'istruzione richiesta dove trovare l' operando, posso:

- passarlo attraverso un registro
- passarne il valore all'istruzione (modalità immediata)
- leggerlo dalla memoria

Quello che cambia tra le varie ISA sono quante e quanto complesse sono le operazioni per l' accesso alla memoria. Più modalità di indirizzamento ho più diventa facile l'accesso, ma allo stesso tempo aumenta la complessità della rete logica.

Esistono diverse opzioni per quanto riguarda alla lettura dell' operando dalla memoria. In caso di accesso **diretto** viene indicato l'indirizzo a cui prendere il dato in memoria. Diversamente se viene utilizzata una modalità di indirizzamento **indiretta** viene indicato l'indirizzo di memoria dell' operando in un registro.
Entrambi gli approcci diretto ed indiretto possono combinarsi nella modalità di indirizzamento **base** dove il registro base utilizzato come offset è unito ad un indirizzo diretto per trovare la posizione in memoria dell' operando.

Esiste un ulteriore versione non implementata in tutte le ISA chiamato **indiciato**: indico due registri e l'indirizzo in memoria dell' operando è la somma dei due registri. Viene chiamato in questo modo perché il primo registro funziona da registro base, mentre il secondo si comporta da indice (es. Accesso ai dati di un vettore).

Altre tipologie di indirizzamento, (meno frequentemente adottate dalle ISA) sono: L'accesso **indiretto** dove viene indicata la cella di memoria contenente l'indirizzo dell' operando, la modalità di indirizzamento **scalato** che si comporta esattamente come l' indiciato, ma viene specificato un ulteriore valore di offset, e per finire le modalità di **autoincremento** e **autodecremento**, le quali funzionano esattamente come la modalità d'accesso tramite registro, solo che dopo la letture il valore contenuto nel registro viene automaticamente incrementato o decrementato.

Le modalità di accesso possono essere combinate: `mov AX [12 + BX + SI]` .

Tramite registro	`mov BL AL`
Immediato	`mov BL, 12`
Diretto	`mov AX, [12]`
Indiretto tramite registro	`mov AX, [BX]`
Indiretto tramite indice	`mov AX, [SI]`

Figure 4: Rispettive istruzioni Assembly

> Se nella modalità di indirizzamento è presente il registro **BP**, il segmento di riferimento sarà **SS** (l'indirizzo è relativo allo stack), altrimenti il riferimento sarà sempre **DS** (segmento dati).

Se si vuole comunque accedere ad un altro segmento di memoria, è possibile effettuare un segment override, specificando il segmento di memoria dove si vuole accedere: `mov AX, [CX:BX + 5]`.

Modi di indirizzamento nel trasferimento di controllo

Con questo nome si intende come indicare il valore del program counter (o instruction pointer) al momento di un salto.
Normalmente viene indicato in modo diretto dalla istruzione di jump, e può essere espresso sia in modo assoluto, che in modo relativo. Nell' ISA Intel è disponibile anche gli indirizzamenti intrasegment ed intersegment, sia in modo diretto che indiretto.

Vengono chiamati intrasegment i salti che si trovano e terminano nello stesso code segment, mentre intersegment i salti che riguardano terminano in un code segment differente da quello di partenza. A seconda dei casi si può avere un indirizzamento diretto o indiretto, ovvero viene indicato direttamente il termine del salto o l'indirizzo di termine è contenuto in un registro.

Modi di indirizzamento I/O

Dal punto di vista dell' ISA esistono due metodi di indirizzamento: il **memory mapped I/O**, dove gli indirizzi per l'interazione con i dispositivi I/O sono contenuti in memoria, e **separated I/O**, dove gli spazi di indirizzamento I/O sono separati dalla memoria, per accedere ai dispositivi ho istruzioni diverse (`in` e `out`).

Per comunicare con i dispositivi I/O è possibile utilizzare un metodo ad indirizzamento diretto: `in AL, 100`, ma l'indirizzo massimo è limitato a 256. Se si vuole utilizzare indirizzi con valori maggiori a 256, è necessario utilizzare un metodo ad indirizzamento tramite registro. Il registro (a 16b) dedicato a queste operazioni è DX.

Tipi e struttura degli operandi

Sono i tipi di dato supportati dall' ISA. Possono essere di tipo intero (signed/unsigned), floating point (single, double o extended precision), caratteri (ascii/unicode), bool o multimediali.

Ed ovviamente è necessario scegliere quali operazioni sono previste dall' ISA per lavorare con i tipi di dato supportati.

Linguaggio Assembly 8086

In due parole

Non è case sensitive.
Ogni statement è terminato da `\n`, lo statement può proseguire alla riga successiva solo se questa comincia con il carattere `&`.
Gli identificatori hanno una lunghezza massima di 31 caratteri ed il nome non può iniziare con un numero.

Tipi di costante

```
mov ax, 13              ; decimale, anche 13D
mov ax, 13h             ; esadecimale (devono iniziare come un numero)
mov ax, 00100B          ; binario
mov ax, 13O             ; ottale

mov ax, 2.34            ; numeri reali
mov ax, 112E-3          ; rappresentabili anche in esponenziale

mov ax, 'T'             ; Costanti carattere
mov ax, 'test'          ; o anche stringa
```

Istruzioni per il trasferimento dati

```
mov dest, sorg          ; sposta il contenuto del secondo operando
                        ; nel primo
mov [bx], al            ; salva nell'indirizzo indicato da BX il
                        ; valore di al
xchg dest, sorg         ; scambia il contenuto dei due operandi
push w0rd               ; inserisce una word nello stack
pop  w0rd               ; estrae una word dallo stack
in   accum, porta       ; legge un dato dalla porta specificata
out  porta, accum       ; scrive un dato sulla porta specificata
```

Si ricorda che istruzioni come `mov [bx], [si]` non sono permesse perché siccome non stiamo utilizzando una macchina *memory-memory*, si può avere al più 1 riferimento alla memoria nella stessa istruzione.

Esistono altri trasferimenti non ammessi dalla `mov`:

- `mov ds, 100`, modificare direttamente il valore di un registro. Occorre utilizzare un registro general purpose:

  ```
  mov ax, 100
  mov ds, ax
  ```

- `mov dx, es`, trasferimento da segment register a segment register.
- `mov cs, 100`, qualsiasi trasferimento che abbia `cs` come destinazione. Ovvero cambiare il codice in esecuzione.

Lo stack pointer `sp` parte con valore iniziale 0xffff, ad ogni istruzione `push`, `sp` diminuisce di 2, mentre ad ogni `pop` aumenta di 2.

> Quando tolgo i dati dallo stack con pop, la cella di memoria non viene azzerata.

Istruzioni di aritmetica binaria

Operazioni di aritmetica binaria

```
; Operazioni ad 1 parametro
inc var1         ; Incrementa di 1 var1
dec var1         ; Decrementa di 1 var1

mul sorg     ; Moltiplicazione sorg * al oppure sorg * ax
div sorg     ; Divisione:      sorg / al oppure sorg / ax

imul sorg    ; mul con segno
idiv sorg    ; mul con segno

neg var1     ; nega il registro var1 (negato aritmetico, non binario)


; Operazioni a 2 parametri
add dest, sorg   ; Salvano entrambe il risultato dell'
sub dest, sorg   ; operazione in `dest`

cmp dest, sorg   ; uguale a sub ma non salva il
                 ; risultato in dest

adc dest, sorg   ; add with carry: dest = dest + sorg + carry_flag
sbb dest, sorg   ; sub with carry: dest = dest - sorg - carry_flag
```

NOTE

Nelle operazioni a due parametri, entrambi i registri devono avere stessa dimensione. Ad esempio `add AX, BL` non è permesso.

Nel caso in cui `div` sia troppo grande per essere contenuto nel registro destinazione, o il divisore sia 0, viene generato un `int 0h` (Divisione per zero).

Sono supportati i formati **signed**, **unsigned**, numeri decimali **packed** [3] e **unpacked** [4].
Nel caso di numeri decimali unpacked, i 4 bit superiori devono essere a 0 se il numero è usato in un'operazione di moltiplicazione o divisione.

Operazioni su 32 bit

Considerando come unico numero a 32bit i registri `bx` e `ax` (con 16bit più significativi salvati in `ax` e 16 meno significativi salvati in `bx`), ed un altro numero a 32bit salvato in analogo modo in `dx` `cx`, somma e sottrazione possono essere eseguite nel seguente modo

```
add ax, cx   ; Somma parti meno significative
adc bx, ds   ; Somma parti più significative con carry

sub ax, cx   ; Analogo per sottrazione
sbb bx, ds
```

[3] Ogni byte contiene due numeri decimali, la cifra più significativa è allocata nei 4 bit superiori. Es: 35=0011.0101
[4] Ogni byte contiene un solo numero decimale BCD nei 4 bit inferiori. Es: 35=0000.0011 0000.0101

Moltiplicazione e Divisione

Esistono due tipi, quelle che operano con segno (`imul` e `idiv`), e quelle che operano in modo unsigned (`mul` e `div`).
Prendono un solo operando, che può essere un registro generale o una variabile. Il secondo operando viene scelto dinamicamente in base alla dimensione del primo. Nel caso di moltiplicazione:

- se è di tipo **byte**: 8bit, il secondo è `al` , ed il risultato è salvato in `ax`
- se è di tipo **word**: 16bit, è `ax` ed il risultato è messo in `dx` : `ax` [5]
- se è di tipo **dword**: 32bit

Se viene preso dalla memoria è necessario specificare manualmente la dimensione attraverso le keyword elencate sopra (es: `mul word [0100]`)

In caso di divisione le operazioni di tipo byte utilizzano `ax` come secondo operando, e salvano risultato e resto in `al` ed `ah` rispettivamente. Per operazioni di tipo word, `dx` : `ax` è il secondo operando, resto e risultato sono salvati in `dx` e `ax` .

Esempio di divisione a 16 bit

```
mov dx, 0234h
mov ax, 5678h
mov cx, 1000h
div cx

; dx = 678
; ax = 2345
```

Se `dx` fosse maggiore di $1000h$, il risultato della divisione risulterebbe a 20byte e non sarebbe possibile salvarlo in `ax` . Quindi genera un interrupt.

Operazioni su numeri decimali

Esistono istruzioni che lavorano con i numeri salvati in formato packed ed unpacked, ma non prendono parametri, dato che lavorano solamente attraverso i registri AL

- AAA converte il risultato di una somma in decimale unpacked
- AAS converte il risultato di una sottrazione in decimale unpacked
- AAM converte il risultato di una moltiplicazione in decimale unpacked
- AAD converte il dividendo di una divisione da decimale unpacked a binario
- DAA converte il risultato di un addizione in decimale packed
- DAS converte il risultato di una sottrazione in decimale packed.

[5] Indico con `ax` : `bx` , un numero i cui bit più significativi sono salvati in `ax` , ed i bit meno significativi sono salvati in `bx` .

Trasferimento di controllo

Salti

Tutti i salti prendono come unico argomento l'indirizzo di destinazione. L'istruzione per il salto incondizionato (equivalente a goto in C) è `jmp`. Esistono anche i salti condizionati, i quali solitamente sono preceduti da un'istruzione `cmp`.

Instruction	Jump if
`JE`	`zf` $= 1$
`JNE`	`zf` $= 0$
`JA` o `JNBE`	`cf` $= 0$ e `zf` $= 0$
`JAE` o `JNB`	`cf` $= 0$
`JB` o `JNAE`	`cf` $= 1$
`JBE` o `JNA`	`cf` $= 1$ o `zf` $= 1$
`JG` o `JNLE`	`zf` $= 0$ e `sf` $=$ `of`
`JGE` o `JNL`	`sf` $=$ `of`
`JL` o `JNGE`	`sf` $\neq$ `of`
`JLE` o `JNG`	`zf` $= 1$ o `sf` $\neq$ `of`

Flag	
`JC` - `JNC`	Jump if Carry (Carry flag a 1)
`JO` - `JNO`	Jump overflow
`JS` - `JNS`	Jump Sign / Jump Not Sign
`JZ` - `JNZ`	Jump Zero (alias di `JE` e `JNE`)
`JP` o `JPE`	Jump Parity (Even). (bit di parità)
`JNP` o `JPO`	Jump Not Parity, o Jump Parity Odd
`JCXZ`	Jump if `cx` (registro contatore) Zero.

Legenda	
A	Above
B	Below
G	Greater
L	Less
E	Equal
N	Not

Esempio di utilizzo di salti condizionati

```
init:   mov ax, 10
        mov bx, 5

check:  cmp ax, bx
        ja halt          ; jump to halt only if ax > bx

        inc ax
        jmp check

halt:   mov  ax, 4c00h
        int 21h
```

CALL e RET

Una procedura è una label, la cui chiamata corrisponde ad un salto incondizionato, i parametri sono passati via stack. La differenza da un normale salto incondizionato è che al momento di una call, è salvato l'instruction pointer nello stack.

Una procedura, nel caso sia all'interno di uno stesso segmento di codice (inter-segment) è detta di tipo **NEAR**, mentre se può esser chiamata all'interno di un segmento di codice qualsiasi (intra-segment) è detta di tipo **FAR**.

Nel momento in cui effettuo una `call` di tipo NEAR, l'unica cosa che cambia è l'instruction pointer, dato che non cambia il segment. Diversamente se effettuo una `call` FAR, siccome cambia anche il code segment, viene anch'esso pushato all'interno dello stack.

```
start:      call function

halt:       mov ax, 4c00h
            int 21h

function:   mov ax, 10h
            ret
```

> jmp e call hanno la stessa sintassi. Per questo se confuse il compilatore non dà errore. Se una funzione è invocata con jmp l'istruzione ret fa comunque il pop di un valore dallo stack e cambiato l'instruction pointer.

LOOP

L'istruzione `loop etichetta` o `loope etichetta` è equivalente ad effettuare le operazioni:

```
dec  cx
cmp  cx, 0
je  etichetta
```

Esistono anche le varianti: `loopz` e `loopne` che controllano inoltre lo zero flag.
Esempio di utilizzo di `loop`:

```
start:  mov ax, 0h
        mov cx, 10h
cycle:  add ax, 10h     ; Eseguita 10h = 16 volte
        loop cycle
```

INT ed IRET

Gli Interrupt interrompono l'esecuzione normale del programma. Possono essere di tipo hardware o invocati via software (es, tramite istruzione `int`) Il programma, una volta fermato, passa il controllo ad una procedura di tipo FAR, chiamata RRI (*Inserire acronimo*). Al termine dell'esecuzione di questa procedura è eseguita l'istruzione `iret`.

Dato che il programma è interrotto e deve riprendere la sua normale esecuzione, al momento di un'interrupt vengono eseguite in ordine le operazioni di:

- Salvare nello stack il register flag (`pushf`)
- Trap Flag = 0 (disabilita esecuzione step by step per ragioni di sicurezza), e IF = 0 (Interrupt Flag = 0, per evitare l'interruzione di altri interrupt mascherabili).
- Salvare nello stack CS e carica CS della RRI
- Salvare nello stack IP e carica IP della RRI

L'istruzione duale `iret` , recupera le istruzioni di IP, CS e register flag precedentemente salvate nello stack.

Esistono due possibili categorie di interrupt:

- Interrupt BIOS, che dal nome agiscono direttamente a livello di BIOS. Esempi sono la 10h per l'output su video e la 16h per l'input da tastiera.
- Interrupt DOS, che agiscono a livello di sistema operativo. Esempio è 21h, utilizzata sempre per I/O da tastiera e terminazione processo.

Ogni interrupt ha un elenco di funzioni, ed il registro `ah` specifica quale utilizzare.

Code	Function	Description	Info
10h	0Eh	Write chatacter on TTY	AL = Character ASCII code BH = page number (0 current page) BL = foreground color (only gui mode)
16h	00h	Keyboard Read	AL = Read ASCII code AH = scan code (specifies input source)
21h	01h	Keyboard Read and echo	AL = Read ASCII code
21h	02h	Character Output	DL = ASCII Code
21h	09h	Print string ($ = end)	DX = String Addr
21h	4Ch	Terminate Process and EXIT	AL = Exit Code

Esempio di utilizzo interrupt

```
CPU 8086              ; direttiva per il processore, indica che si
                      ; scrive codice asm 8086 non necessaria

Start:  mov ah, 00h
        int 16h       ; lettura da testiera

        cmp al,1bh    ; check if input == esc
        je Exit

        mov ah, 0eh
        mov bx, 00h ; on page 0
        int 10h       ; Print to video
```

```
        jmp Start


Exit:   mov ax, 4C00h
        int 21h      ; Return 0
```

Definizione Dati

La sintassi utilizzata per definire un dato in assembly è necessaria una label, non richiesta ma utile per avere un riferimento per accedere al dato (facoltativo) ; il tipo del dato, ed i valori per l'inizializzazione, che possono essere uno o più separati da virgola.

I tipi di dato a disposizione sono `DB` (*Define Byte* 8bit), `DW` (*Define Word* 16bit) e `DD` (*Define doubleword* 32bbit).
Per dichiarare dei dati evitando l'inizializzazione è possibile utilizzare rispettivamente `RESB`, `RESW` e `RESW`.

```
ByteVar:    DB 0                ; Byte inizializzato a 0
ByteArray:  DB 1,2,3,4          ; Array di 4 byte
String:     DB '8086',0dh,0ah   ; Array di 6 caratteri (4 + CR/LF)
FiveTh:     DW 100*50           ; Assegnamento risultato operazione
Zeros:      times 256 DB 0      ; Array di 256 0

Table:      RESB 50             ; Array di 50 byte non inizializzati

NearPtr:    DW String           ; Contiene l'offset di String (NEAR)
FarPtr:     DD String           ; Contiene offset ed indirizzo del segmento
                                ; di String (FAR)
```

Esempio utilizzo Variabili

```
CPU 8086
; Program to check if the same character is pressed twice

SECTION data ; definisce il segmento dati
UserChr:  RESB   1
IntroMsg: DB     "Press two keys",0ah,0dh,0
SameMsg:  DB     "Same Character inserted twice",0
DiffMsg:  DB     "The Two characters are different",0

SECTION text
..start:    ; definiisce inizio del main
            mov ax, data
            mov ds, ax

            mov si, IntroMsg
            call PrintMsg

            call Read
            mov [UserChr], al

            call Read
            cmp al, [UserChr]
            jne DiffChr
```

```
SameChr:    mov si, SameMsg
            call PrintMsg
            jmp End

DiffChr:    mov si, DiffMsg
            call PrintMsg

End:        mov ax, 4c00h
            int 21h


; Read char with int16
Read:       mov ah, 00h
            int 16h
            ret

; Print Message pointed by si
PrintMsg:   mov ah, 0eh
            mov bx, 00h
p_loop:     mov al, [si]
            int 10h

            inc si
            cmp al, 0
            jne p_loop

            ret
```

All'inizio del programma è richiesto che `DS` punti alla sezione dei dati per accedere alle variabili dichiarate

Istruzioni di logica binaria

```
and dest,sorg
not dest
or dest,sorg
test dest,sorg
xor dest,sorg
```

`and`, `not`, `or`, `xor` eseguono l'operazione logica sui bit dei registri forniti come parametro. `test` funziona come `and` modifica i flag ma non salva il risultato. È spesso utilizzato per controllare se determinati bit siano ad 1 es: `test al, 0010000b`.

Shift e rotate

```
shl dest, count
shr dest, count
rol dest, count
ror dest, count
```

```
sal dest, count
sar dest, count
rcl dest, count
rcr dest, count
```

Gli shift aritmetici `sar` e `sal` si differenziano dagli shift logici `shl` e `shr` perché il bit di segno non viene shiftato, rimanendo fisso di posizione.
Nelle istruzioni `rcl` ed `rcr` il bit shiftato viene prima inserito nel flag di carry ed alla rotazione successiva inserito nell'estremo opposto del byte. Le istruzioni `rol` e `ror` il bit shiftato viene inserito immediatamente nell'estremo opposto, inoltre viene salvato anche nel carry flag.

> Se viene specificato `CPU 8086`, nel caso in cui `dest` sia diverso da 1, è richiesto passarne il valore attraverso il registro `CX`.

Esempio

```
SECTION data
number: DB 10010111b

SECTION code
..start:
        mov ax, data
        mov ds, ax

        mov dl, [number]
```

```
        call DlRepr          ; dl=10010111b

        mov dl, [number]
        shl dl, 1
        call DlRepr          ; dl=00101110b

        mov dl, [number]
        ror dl, 1
        call DlRepr          ; dl=11001011b

End:    mov ax, 4c00h
        int 21h


; Print binary value of dl
DlRepr: mov cx, 8
        rol dl, 1    ; Start printing from MSB, equivalent of ror dl, 7

        mov ah, 0eh
        xor bx, bx

r_loop: mov al, dl
        and al, 1    ; Get LSB
        add al, '0'  ; LSB to Ascii

        int 10h
        rol dl, 1    ; Rotate number to next digit

        loop r_loop

        mov al, 0dh ; Print New line (CR+LF)
        int 10h
        mov al, 0ah
        int 10h

        ror dl, 1   ; Restore dl to orignal value
        ret
```

Operazioni su stringhe di dati

Ogni tipo di dati, composto da più di un valore è trattato come una stringa di dati. Considerando quindi due stringhe `str1` e `str2` ho a disposizione le operazioni:

`cmps`	Compare String
`movs`	Move string
`lods`	Load string, carica primo elemento della stringa in `al` / `ax`
`stos`	Store `al` / `ax` nell'indice indicato dalla stringa
`scas`	Scan string, confronta `al` / `ax` con una stringa

Per tutte queste operazioni, l'indirizzo della stringa sorgente è sempre `[ds:si]`, mentre quella destinazione `[es:di]`. Gli indirizzamenti `ds` e `es` sono indicati da due registri differenti per permettere spostamento di dati tra due segmenti di memoria differenti.

Inoltre ogni operazione ha una sua forma con `b` o `w` indicati al termine, per esplicitare operazioni su byte o word.

Le operazioni operano su singoli elementi della stringa, ad esempio se eseguita `movsb` solo un byte è spostato da sorgente a destinazione, ed il registro `cx` è incrementato/decrementato di 1. Per questo motivo esistono le funzioni di utilità che prendono come unico operando le istruzioni precedenti:

`rep`, `repe`, `repz`	Ripetono l'operazione fino a quando `cx != 0` e `zf=1` decrementando `cx`
`repne`, `repnz`	Ripetono l'operazione fino a quando `cx != 0` e `zf=0` decrementando `cx`

Le doppie condizioni d'uscita servono per uscire dalla ripetizione quando termino l'operazione o quando trovo un confronto positivo nel caso di `cmps` / `scas`. In altre parole nel caso di `cmps` tra due stringhe utilizzo `repe` quando voglio trovare il primo elemento diverso, e `repne` quando voglio trovare il primo elemento uguale.

Di default tutte queste operazioni incrementano l'indirizzo `cx`, scorrendo le stringhe da sinistra a destra. Per invertire il senso di scorrimento è necessario mettere ad 1 `DF` (*Direction Flag*).

Esempio con lodsb

```
CPU 8086

SECTION data
Msg:        DB "Test String",0

SECTION code
..start:
            mov ax, data
            mov ds, ax

            xor bx, bx
```

```
            mov ah, 0eh

            mov si, Msg

l_print:    lodsb
            int 10h

            cmp al, 0
            jne l_print

l_end:      mov ax, 4c00h
            int 21h
```

Istruzioni per il controllo dei flag

`clc` / `stc`	Clear/Set Carry Flag
`cld` / `std`	Clear/Set Direction Flag
`cli` / `sti`	Clear/Set Interrupt Flag
`cmc`	Complement Carry Flag: (Toggle `cf`)

`lahf`	Load flag in `ah`
`sahf`	Store `ah` into flag
`popf`	Pop flag from stack
`pushf`	Push flag into stack

Le istruzioni `lahf` e `sahf` funzionano operano solo sui flag di stato: `SF`, `ZF`, `AF`, `CF` e `PF`. Istruzioni come `clc`, `stc` e `cmc` sono usate per operazioni aritmetiche a più byte.

Passaggio di parametri con stack

Per effettuare passaggio di parametri ad una funzione, oltre ad utilizzare registri o variabili in memoria, è possibili utilizzare lo stack.

```
    push ax
    push bx
    call funct
    add sp, 4
```

Una volta caricato un parametro nello stack con `push` è necessario, una volta invocata la funzione, ritornare il valore dello stack pointer al valore originario, presente prima di `push`. Per effettuare l'operazione viene solitamente utilizzata un `add` e non `pop` per evitare di "sprecare registri".

Si che viene **sommato** e non sottratto 2 per ogni numero di parametri passati alla funzione, perché il valore dello stack pointer parte inizialmente dal valore `0xFFFF`, crescendo verso `0x0000`.

Per recuperare i parametri dallo stack, viene utilizzato il base pointer `bp`, dato che `sp` può essere soggetto a variazioni all'interno della funzione. Inoltre, l' operazioni come `[bp]` fanno riferimento direttamente allo stack segment, diversamente da quelle come `[bx]` che fanno riferimento al data segment.

Recupero di valori dallo stack

```
funct:  push bp          ; per tenere salvato il valore di bp
        mov bp, sp

        mov ax, [bp + 6]  ; primo valore pushato 2 + 2*2
        mov bx, [bp + 4]  ; secondo valore pushato

        pop bp
        ret
```

Memorie

Distinzione delle memorie

Una memoria è un'unità logica, dedicata alla memorizzazione dei dati nel calcolatore. Diversi tipi di memoria, sono classificate in base a 5 parametri:

- Capacità: numero delle parole memorizzabili Chiamato L il numero di linee, ed N il numero di parole $L = \log_2 N$. Ogni parola è un dato ad M bit.
  Dato che di solito $M = 8$, l'unità di misura la capacità di memoria si misura in byte.
- Caratteristiche fisiche: Di cui tipo (semiconduttore come RAM, a superficie magnetica HD, ottica DVD), il consumo, l'affidabilità misurata come MTBF (*Mean Time Between Failure*), alterabilità (solo lettura o lettura/scrittura) e durevolezza (volatile o non-volatile)

- Organizzazione: memorie interne al calcolatore sono organizzate in una gerarchia, parallelismo ed interlacciamento (come si interfacciano con memorie di livello superiore)
- Modalità d'accesso: in quale modi si accede ai dati in memoria,
  - Sequenziale: per accedere ad un dato in una posizione fissa è necessario leggere tutti i dati precedenti (cassette mangianastri)
  - Diretto: è possibile accedere direttamente alla posizione in memoria, in molti casi però il tempo d'accesso è dipendente dalla posizione del dato rispetto alla posizione appena letta o scritta (cd)
  - Casuale: un accesso diretto con tempo d'accesso costante (RAM, ROM)
  - Associativo: si accedono agli indirizzi di memoria attraverso delle chiavi hash (tag), per controllare se un dato è presente vengono controllate tutte le locazioni esistenti.
- Prestazioni: Per misurare le prestazioni di una memoria, esistono diversi parametri, uno di questi, come citato al punto precedente è il tempo d'accesso, ovvero il tempo impiegato dalla memoria per raggiungere l'indirizzo da quando è fornito.

  Il tempo di ciclo T_{rc} è definito come il tempo di accesso più il tempo necessario per terminare l'operazione prima di poter compiere un altro accesso.

  Mentre la velocità di trasferimento (bit rate) è l'inverso del tempo d'accesso, misurando il numero di bit trasferiti.

  Solitamente sono tutti dati imposti dalla CPU, a cui la memoria si adegua o introduce ritardi.

Il numero di pin in ingresso del componente della memoria non coincide necessariamente con n_a ed n_d del bus dei dati. In un processore moderno, normalmente $n_a = 36$ quindi può indirizzare al massimo $64G$ di memoria RAM. Le dimensioni dell'unità di memoria può essere inferiore a $64G$, magari si hanno 4 memorie da $16G$, con $L = 34$. n_a rappresenta la massima memoria indirizzabile del processore, L rappresenta la memoria effettiva di un cip specifico di memoria.

Le modalità d'accesso sono

Gerarchie di memoria

La memoria ideale sarebbe di capacità infinita, con tempo d'accesso, costo e consumo nullo.

L'insieme delle memorie presenti in un calcolatore mira a combinare le caratteristiche migliori di ogni tipo di memoria per raggiungere l'obbiettivo di memoria ideale.

Salendo la gerarchia diminuisce il tempo d'accesso e la capacità delle memorie e cresce il costo per bit.

Al livello più alto della gerarchia sono presenti i registri interni alla CPU, segue la cache, poi la memoria centrale o principale, ed in fondo alla gerarchia le memorie secondarie, con alte capacità ma bassi costi e permanenti.

Principio di località

Un programma in un certo istante t necessita di determinati dati Se un programma ad un istante t accede ad un certo dato, c'è alta probabilità che siano richiesti anche i dati adiacenti (proprietà di località spaziale), e che all'istante successivo si acceda nuovamente alla memoria (proprietà di località temporale).

Gli algoritmi della MMU (*Memory Management Unit*) si concentrano a minimizzare il numero di accessi in memoria a livelli più bassi.

Blocco di memoria

Con blocco si fa riferimento all'unità di informazione minima scambiata fra i livelli di memoria. Quando un particolare dato viene richiesto ad una memoria, l'evento che il dato non sia presente prende il nome di "miss" mentre se il dato è già presente si chiama "hit".
Lo scopo di una buona gerarchia di memoria è massimizzare il rapporto tra hit e miss. La frequenza di accessi trovati direttamente al livello superiore prende il nome di hit-rate (h). La miss rate è quindi complementare a quest'ultimo.
Il tempo per recuperare il dato al livello successivo in caso di miss prende il nome di "Miss Penalty", mentre se il dato è già disponibile, il tempo per recuperarlo è pari al tempo di hit.

Il miss penalty T_{mp} può essere visto come la somma tra hit time ed il tempo richiesto per il trasferimento dei dati dai blocchi inferiori (T_{miss}).
Ottenendo la relazione:

$$T_{acc} = h \cdot T_h + (1-h)T_{mp} = T_h + (1-h)T_{miss}$$

Dalla formula, minimizzando la miss rate, si tende alla memoria ideale, portando il tempo d'accesso uguale al tempo di hit.

Gerarchie di memoria

Una gerarchia di memoria è definita dai suoi livelli di memoria: il loro numero, dimensione, velocità e componenti.
Il piazzamento del blocco, chiamato anche funzione di traduzione o di mapping, sceglie dove allocare il blocco nel livello di memoria corrente. Nel caso della cache ad esempio è necessario calcolare un hash per determinarne la locazione.

L'identificazione come del blocco indica come risalire alla posizione del blocco di memoria.

Il rimpiazzamento si occupa di inserire i dati provenienti dai livelli inferiori della gerarchia, scegliendo in quale posizione del livello corrente inserirlo.

I dati, una volta modificati, vengono scritti sui livelli inferiori utilizzando strategie di scrittura.

Ad esempio, nel caso dei registri, l'identificazione è nominale, ed il nome viene indicato nel codice sorgente, mentre identificazione e rimpiazzamento sono definiti dal compilatore.

In memoria centrale, il piazzamento del blocco dipende dall'istruzizone (specificato dall'indirizzo di memoria), i dati sono identificati dall'indirizzo e le scritture sono scielte a livello di codice.

Le memorie centrali nei calcolatori sono le RAM, volatili e di lettura e scrittura). Di memorie RAM ci sono due categorie: SRAM (*Static RAM*) utilizzate per memorie cache e DRAM (*Dynamic RAM*) utilizzate come memoria centrale.

Le memorie SRAM sono molto veloci e realizzate attraverso flip flop D, che permettono di memorizzare il dato senza la perdita. La tecnologia richiesta per realizzare uno di questi flip flop è molto costosa, per questo al giorno d'oggi non è possibile utilizzare queste memorie per immagazzinare grandi quantità di dati.

Oltre ai flip flop in queste memorie sono presenti logiche di indrizzo e la selezione di lettura scrittura.

Le memoria DRAM ogni bit è rappresentato da un condensatore / transistor (nel caso di tecnologia mos). Il loro costo è molto inferiore quindi rispetto alle precedenti.
Siccome i condensatori col tempo si scaricano, necessitano di circuiti automatici di refresh per ricaricare il condensatore.

Oltre a memorie RAM sono presenti le memorie ROM, a sola lettura. Esse sono divise in: le ROM originali erano scrivibili un'unica volta e non sono volatili. PROM (*Programmable ROM*), scrivibili un'unica volta, EPROM (*Erasable PROM*), cancellabili attraverso la luce ultravioletta, EEPROM (*Electrically Erasable PROM*), cancellabili con segnali elettrici. FLASH leggibili e scrivibili con dati non volatili (come SSD).

Memorie SRAM

Le SRAM, memorie ram statiche, sono RAM accedute come se fossero una matrice di dati. Normalmente il singolo dato è rappresentato da flip flop D.

Per studiare un ciclo di lettura di una memoria statica, prendiamo come esempio una ram con $N = 1024$ e $M = 4bit$. Il numero di bit necessari ad indirizzare i valori è quindi $L = \log_2 N = 10$. La RAM ha quindi ingressi e 2 segnali di controllo CS (*Chip Select*) per abilitare il dispositivo e WE (*Write enable*).

Una volta caricato l'indirizzo nel bus degli indirizzi, si abilita CS e dopo un tempo T_{acc} il dato viene reso disponibile dalla memoria sul bus dei dati.

Per scrittura viene abilitato il write enable e caricato il dato da scrivere nel bus dei dati, ed al momento dell'abilitazione di CS, dopo un certo tempo il dato viene scritto in memoria.

Al termine di ogni operazione CS torna a 0, disabilitando il chip.

Memorie DRAM

Nel caso di ram dinamiche, tipicamente con capacità maggiore es $8Gb$ il numero di bit necessari per indirizzare i dati dovrebbe essere $L = 33$, portando un numero di ingressi esagerato per un chip di memoria.
La soluzione per accedere a questo tipi di memoria è dividere la capacità come una matrice bidimensionale, es una matrice di $2K$ righe e $4M$ colonne, ottenendo numero di bit di indirizzamento pari alla dimensione maggiore.
Quindi le DRAM sono più lente rispetto alle SRAM, anche per la necessità di avere un secondo indirizzo per identificare il dato n memoria.

I due segnali per identificare gli indirizzi di una DRAM prendono il nome di RAS (*Row Address Strobe*) e CAS (*Column Address Strobe*).

RAS viene utilizzato per l'ingresso di un decoder, RAS viene dato in ingresso ad un decoder, identificando con 1 la riga da leggere/scrivere. In un secondo momento il segnale CAS viene utilizzato come ingresso di un demultiplexer, selezionando la colonna in ingresso del relativo indirizzo.

Nel caso di lettura il segnale passa attraverso un buffer DOUT, inviando i dati al processore, nel caso di scrittura i dati vengono letti dal processore scrivendo il dato nella riga e colonna selezionati.

Qua sotto se mi ricordo di aggiungerla sarà presente lo schema funzionale di una memoria DRAM.

Contatore di refresh e circuito di refresh si sostituiscono agli indirizzi di riga per leggere gli indirizzi di memoria e ricaricare i condensatori. OE è equivalente a CS.

Lettura e scrittura DRAM

La differenza fondamentale è la necessità di inserire un secondo indirizzo per identificare i dati. Prima si inserisce il valore di RAS nel bus dei dati, abilitandone la lettura con RAS. Successivamente si inserisce

CAS abilitandone la lettura con CAS. Al momento dell'abilitazione di CAS, RAS non viene disabilitato, (spiegazione successiva).
Successivamente dopo un tempo di accesso, nel caso di lettura viene fornito sul bus dei dati il valore contenuto in memoria.

Quando entrambi i valori RAS e CAS sono disabilitati l'output è in 3-state, quindi non più modificabile.

Per effettuare le write è necessario che il segnale WE sia abilitato prima di abilitare il CAS, altrimenti il valore contenuto in memoria verrebbe copiato nel bus.

SDRAM

Sono sempre delle DRAM, realizzate con condensatori, con la differenza che sono sincrone. Infatti se si guarda lo schema logico di una DRAM, non è presente un segnale di clock, effettuando ogni operazione sul fronte di salita/discesa del CAS.

Il vantaggio principale di questo tipo di memoria è la certezza di avere il dato in memoria dopo un numero preciso di cicli di clock. Cosa non sempre vera nel caso di memoria asincrona, dove il tempo richiesto è variabile.

Lo svantaggio che portano è visibile nel caso in cui i dati siano disponibili prima che il ciclo di clock sia completato. Questo è compensato dal **trasferimento a burst**.

La CPU è infatti in grado di scegliere il numero di dati da fare ritornare alla memoria. Nel caso di trasferimento singolo, quando la memoria è interrogata torna un solo indirizzo. In caso di trasferimento a burst la memoria restituisce ad ogni ciclo di clock, il dato all'indirizzo di memoria successivo a quello specificato.

Ad esempio nel caso di accesso ad un file, il primo valore viene letto dopo 4 cicli di clock, mentre i successivi direttamente al ciclo di clock seguente. Questa lettura a bus è fondamentale per riempire la cache interna.

DDR

Le DDR (*Double Data Rate*) nascono come un ulteriore miglioramento delle SDRAM. Come dice il nome, leggono due dati in un unico ciclo di clock, campionando sia sul fronte di salita che sul fronte di discesa.

Il principio di funzionamento è identico.

FPM-DRAM

Fast Page Mode DRAM permettono di tenere il segnale di RAS# attivo, variando solo il segnale CAS# in modo da velocizzare tutti gli accessi successivi al valore in memoria, ottenendo trasferimenti di tipo 6-3-3-3 O 5-3-3-3.

EDO-DRAM

Extended Data Out DRAM, permettono di risparmiare cicli di clock per trasferimenti successivi, perché l'uscita non richiedevano di disabilitare l'uscita (WE) ad ogni lettura.

RDRAM

Rambus DRAM (Rambus è il nome di un consorzio), sviluppate da Intel. Permettevano di avere chip con fino a 320 pin con velocità di trasferimento di $1.6Gb/s$, trasferimento a blocchi stile Fast. Chip con pin su un unico lato per ridurre il consumo di energia. Utilizzate in schede video ad alte prestazioni.

Memorie Permanenti

Sono memorie a semiconduttore, non volatili. Le memorie FLASH (realizzate in tecnologia NAND o NOR) sono utilizzate per chiavette USB e SSD. NVSRAM (*Non Voltaile SRAM*) non sono molto diffuse per via dell'elevato costo di produzione.

FeRAM

Le *Ferroelectirc RAM*, chiamate anche FRAM o F-RAM, rientrano nelle tecnologie di memorie non volatili emergenti, ed attraverso uno strato di materiale ferroelettrico, permettendo la permanenza ai dati anche in caso di mancanza di corrente.

Rispetto alle memorie flash consumano meno energia, hanno una maggiore velocità di scrittura ed un numero di scrittura/cancellazione maggiore.

PCM

Le *Phase Change memory* sono composte da materiale in grado di cambiare fase (cristallina o amorfa). In particolare una regione amorfa presenta bassa riflettività e una regione cristallina presenta alta riflettività, permettendo attraverso punte o laser di leggerne il contenuto (funzionamento simile a CD/DVD).

MRAM

Le *Magnetoresisteive RAM*, sfruttano l'effetto magnetoresistivo, non memorizzando le informazioni come quantità di carica elettrica (Come le RAM), ma come un campo magnetico.
La lettura dell'informazione è ottenuta attraverso la misura della resistenza elettrica della cella.

RRAM/CBRAM

La *Conductive Bridging RAM* è basata sulle proprietà di un elettrolita solido (tipicamente solfuro di germanio drogato con rame) che posto tra un elettrodo relativamente inerte (esempio tungsteno, materiale delle lampadine a filo) ed uno elettrochimicamente attivo come argento o rame, fa si che quando si applica un campo elettrico viene provocato uno spostamento di ioni metallici nell'elettrolita, formando dei "nano-fili" conduttivi.

I principali vantaggi di questa tecnologia sono il basso consumo, l'elevata velocità di scrittura e la lunga durata.

Interfaccia tra CPU e Memoria

Tramite il bus degli indirizzi di dimensione na, unidirezionale verso la memoria, e tramite il bus dei dati di dimensione nd bidirezionale il processore scambia informazioni con la memoria.

Oltre a bus degli indirizzi e bus dei dati sono presenti bus di controllo, con segnali di read, write, ads (*Address Strobe*) indica che il bus degli indirizzi è campionabile e presenta un'indirizzo valido (unidirezionale verso la memoria). Nel caso di memorie sincrone, è presente anche Rdy (*Ready*) unidirezionale verso il processore, ed indica se il dato della memoria è pronto.

È presente anche un segnale M/IO (*Memory I/O*) che indica se si vuole accedere a memoria o input/output. Infatti esistono due politiche per accedere alla memoria: le memory mapped IO, dove gli indirizzi di memoria sono considerati dal processore insieme agli indirizzi di input/output, accedere ad IO in questo caso vuole dire accedere alla memoria; nelle macchine intel come l'8086 gli indirizzi di memoria ed indirizzi per direttive ad input/output sono separati, quindi è necessario specificare se il dato nel bus degli indirizzi è destinato per la memoria o ad un dispositivo esterno.

Architettura specifica 8086

Sono presenti due ingressi di terra in modo da distribuire il carico Il ciclo di lettura è diviso in quattro fasi, ed a fasi diverse, i 20 piedini per gli indirizzi possono essere considerati come ingressi per il bus degli indirizzi o ingressi per il bus dei dati. Questo comportamento viene indicato dicendo che il bus dei dati ed indirizzi è multiplexato.

NMI, INT e INTA servono per la gestione delle interruzioni. Il segnale d'ingresso INT indica la richiesta di interrupt al processore, il segnale INTA è il segnale in uscita del processore ed indica che il processore è in risposta all'interrupt. Infatti il processore è in grado di ignorare gli interrupt nel caso stia effettuando calcoli importanti.
Questo non è possibile se il segnale di interrupt che riceve in ingresso è NMI (*Non Maskerable Interrupt*), dove il processore entro tempo prestabilito è obbligato a rispondere.

In uscita ha il segnale M/IO che indica se si sta lavorando con memoria o segnale IO. Ha in ingresso il segnale di READY, proveniente dalla memoria. Il segnale ALE (*Address latch Enable*), equivalente all'AdS di prima. Segnale di Read e Write sono separati.

Segnale di test non utilizzato nel funzionamento normale, e i segnali di minmax specificavano la modalità di funzionamento dell'8086 (min modalità ridotta che consuma di meno). Segnali di HOLD e HOLDA, sono utilizzati per permettere ad altri componenti di fornire i dati e controllare l'esecuzione, permettendo al processore di effettuare altri calcoli.

DTR *Data transition / Receive*, dove quando è a 1 si ha un trasferimento dati da processore a memoria, e quando è a 0 il trasferimento è tra memoria e processore.

DEN *Data Enable* indica che quello che sta leggendo in questo momento il processore sono dei dati. DEN e DTR funzionano sui transceiver bidirezionali, mentre l'ALE funziona sui latch.

BHE *Bus High Enable* serve per l'interfacciamento con i chip fisici, perché il bus dei dati essendo a 16 bit e le memorie, avendo un parallelismo a 8 bit, per leggere due byte alla volta BHE permette di mettere due memorie in parallelo, leggendo nella parte alta dei 16 bit il valore dalla seconda memoria.

Multiplexing nel dettaglio

Nel transceiver Quando viene abilitato il segnale DTR, il segnale da input viene messo in output, diversamente il segnale passa da output ad input.

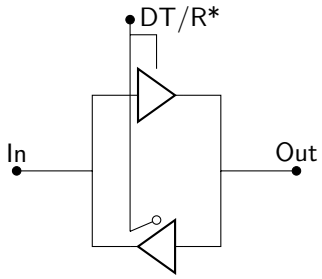


Figure 5: Transceiver

Nel caso di dati infatti, deve essere specificato in quale direzione i dati devono passare. Nel transceiver Quando viene abilitato il segnale DTR, il segnale da input viene messo in output, diversamente il segnale passa da output ad input.

CS disabilita completamente il transceiver, non facendo passare il segnale

Dato che lo stesso numero di uscite della CPU deve guidare il bus dei dati ed il bus degli indirizzi, attraverso dei latch e dei transceiver bidirezionali, viene deciso dove instradare i segnali.

Nell istante T1, avremo quindi `DEN*=1` disattivando la scrittura del bus dei dati e `ALE = 1` campionando i bit nel bus degli indirizzi.
In t3 `DEN*=0` abilitando i 3state, `ALE=0` indicando ai latch di non campionare (tenendo come valore presente nel bus degli indirizzi il valore precedente), `DTR = 0` abilitando scrittura nel 3state.

In T1, quindi si passa l'indirizzo di memoria, in T2 è una fase di bufferizzazione, dando il tempo alla memoria di campionare il buffer degli indirizzi, in T3 abilita il buffer dei dati ed in T4 è il ciclo di completamento.

Lettura e Scrittura

In fase di lettura, l'indirizzo di memoria è contenuto inizialmente in tutti e 20 i bit. Il segnale di `ALE` viene abilitato, permettendo al flip flop D di campionare i risultati, il data enable non è attivo.
Il `DTR` rimane tutto il tempo a 0, siccome si tratta di una fase di lettura.

In fase T2 si porta il valore di `DEN` a 0, indicando la fine della fase in cui il bus multiplexato ha la funzione di indirizzo, abilito anche `RD*=0`

In fase T3 vengono caricati i valori dei 16 bit del dato di memoria nel bus degli indirizzi, essendo abilitato il valore di `RD` alla fase precedente e `DEN`.

Nella parte finale il `DTR` viene portato ad 1, abilitando la scrittura da processore a memoria.

Bus sincroni ed asincroni

Le fasi T1, T2, T3 e T4 sono scandite dai diversi cicli di clock. Il vantaggio di un'architettura sincrona è la certezza di sapere lo stato in cui mi trovo al momento, e la certezza dei tempi di esecuzione. L'alternativa è una struttura asincrona, gestita tramite un'protocollo che indica quando inizia e termina ogni fase.

Il metodo più comune per gestire approcci asincroni, è attraverso segnali di handshake: ARDY (*Address Ready*),equivalente ad ADS e ALE, e DRDY sono i segnali utilizzati dal protocollo di handshake. Il segnale ARDY indica di iniziare la fase di indirizzo. Sono presenti anche i segnali di acknowledge, i quali dal nome riconoscono di aver ricevuto il segnale.

Prendendo come esempio il caso di un ciclo di lettura, il processore manda gli indirizzi alla memoria con ARDY. La memoria dopo un certo tempo la memoria risponde con un segnale di acknowledge `ack`. Il processore ricevendo il segnale `ack` capisce che la memoria ha effettuato l'elaborazione dell'indirizzo e disabilita il segnale ADRY, comunicando alla memoria che ha ricevuto l'acknowledgement. Quando la memoria ha copiato tutti i dati, porta anch'essa il segnale `ack=0`.

Si esegue un'analogo procedimento per il trasferimento dei dati: il processore attende il segnale `drdy` dalla memoria, il quale indica quando i dati letti sono presenti sul bus. Appena riceve il segnale risponde con `ack` indicando alla memoria che ha ricevuto i dati.
Una volta ricevuti questi dati, la memoria toglie il segnale `drdy`.

Interfaccia I/o

Per i dispositivi di input/output è utilizzata comunque la logica sincrona. Dove il campionamento è effettuato sempre attraverso flip-flop.

Indirizzamento e accesso in moduli

Quello visto fino ad ora è l'interfacciamento tra processore e memoria logica, un'unica entità con un bus degli indirizzi ed un bus di dati con dimensioni pari a quelle del sistema.

Nel caso le memorie a disposizione abbiano una dimensione inferiore a quella del bus di dati ($L < n_a$), si può escludere gli ultimi k bit dell'indirizzo meno significativi, (con $k = n_a - L$), quindi per leggere un'elemento di 32 bit all'indirizzo 100, con a disposizione quattro memorie in parallelo, ottengo il dato leggendolo da un'unica riga, indicata dai bit più significativi rimanenti: ovvero 1000 1001 1010 1011 se $k = 2$ la riga è 10.
Il problema nasce quando è necessario leggere una parola di 32 bit all'indirizzo 1001, dove la cella degli ultimi 8 bit sarebbe 1100, indicando una riga differente. È necessario effettuare la lettura con accessi in memoria differenza.

Ricordiamo che al capitolo sulle architetture risc e cisc (pg. ???) è stato detto che le architetture RISC preferiscono indirizzi di memoria allineati, garantendo letture più veloci pur consumando spazio in memoria.

Nel caso di letture da righe differenti, per realizzare tale modello di memoria, sono necessari CS separati, in modo da abilitare e disabilitare la scrittura sul bus delle celle con righe differenti. I segnali di CS vengono chiamati di byte enable.

In caso di parole allineate, l'accesso è possibile in un unico ciclo, abilitando il valore dei byte enable a tutte le memorie.

Nel caso dell'8086, è presente un solo piedino di byte enable, utilizzando `a0` come abilitazione delle

memorie inferiori e BHE come abilitatore delle memorie superiori.

Memorie Cache

Le memorie cache non sono visibili dal programmatore, il processore, attraverso l'MMU cerca di accedere prima agli indirizzi attraverso la cache, e, se non sono presenti, attraverso il system bus li preleva dalla memoria.

Essa è una memoria associativa, divisa in due parti: la cache directory, che contiene un tag, un indice richiesto per risalire al dato presente nella memoria, e la cache memory, dove sono presenti i dati, organizzata in blocchi di k parole.

Ciascuna riga (blocco) della cache, quindi è composta da multiple parole. La copia da ram a cache avviene a blocchi, favorendo i principi di località spaziale e temporale.

Classificazione delle miss

Le miss si classificano principalmente in tre categorie: le inevitabili per qualsiasi tipo di architettura (es. primo accesso). Di capacità, si richiede di accedere ad un dato che è appena stato sostituito per via dell'insufficienza di memoria, e le miss di conflitto: che dipendono dalle politiche di piazzamento.

Associatività

Tutte le architetture di cache, funzionano dividendo l'indirizzo del processore in index, offset e tag. Preso d'esempio l'indirizzo `3f70H`, l'index identifica il numero di riga dove controllare il dato richiesto dal processore, e per identificare un dato contenuto in una riga si utilizza il tag, composto da t bit.

Nel caso in cui il tag corrisponda, si ha una hit. Come l'index è l'indice di riga, l'offset, definito dagli o bit meno significativi, è l'indice di colonna.

In poche parole, l'index ed offset identificano riga e colonna, mentre il tag, se corrisponde, indica se il dato cercato è presente nella cache.

Il rimpiazzamento, avviene quando tag non corrisponde, in caso carica il dato richiesto dalla ram. Questo tipo di cache, viene chiamato direct mapping, ovvero, ad un indirizzo, associa una ed una sola cella nella memoria. Il caso opposto sono le cache fully associative, dove un dato può andare in una riga qualsiasi della cache.

Questo tipo di cache, viene suddivisa solamente in tag ed offset. Per controllare se un'oggetto è in memoria, non avendo più un index è necessario controllare tutti i tag per vedere se uno corrisponde. In caso di miss, se c'è ancora spazio nella cache, si introduce il dato in righe vuote, altrimenti si procede con una politica di rimpiazzamento.

Vengono anche chiamate CAM (*Content Addressable Memory*), per via che il tag è utilizzato come indice per controllare se un dato è contenuto nella cache.

Pro e contro delle due architetture

La fully associative, è più costosa, in quanto richiede un numero di comparatori almeno pari a 2^t. Ma ha un numero notevolmente inferiore di miss di conflitto.

La direct map invece è più economica, ma molto rigida, generando molte miss di conflitto.

Una via di mezzo è la cache set-associative, dove le linee sono organizzate set, ovvero un'insieme di linee dove un dato può essere salvato in qualsiasi riga. Nel caso della fully associative, tutte le righe sono raggruppate in un unico set, nel caso della direct mapping, ad ogni riga corrisponde un set.

Il numero di set è quindi $N_S = N_L/n$, con N_L numero di righe e n la grandezza del set. La selezione del set è direct mapped, mentre la selezione della linea su cui salvare i dati è effettuato come la fully associative.

Al cambiare dell'associatività, cambia anche la dimensione del tag di index, $i = \log_2 N_s = \log_2 N_L/n$, infatti con n, crescendo sempre per potenze di 2, ha valore massimo di N_L, corrispondendo al caso di fully associative.

Cache Suddivise

Ormai tutti i processori utilizzano due cache divise per i dati e le istruzioni, mentre la ram è unica.

Politiche di rimpiazzamento

Queste politiche hanno senso solamente in caso di fully associative o n-way associative. Esistono due principali strategie, una random, dove il rimpiazzamento è casuale o pseudo-casuale, ed una LRU (*least Recently Used*) la quale si basa sul principio di località temporale.

Politiche di scrittura

Diversamente dalla lettura, dove i dati vengono sempre allocati in cache quando vengono letti da un livello inferiore, nel caso di scrittura, è possibile modificare il dato solamente nella cache e successivamente modificare il dato (writeback), oppure applicare la stessa modifica nei blocchi di memoria inferiori (writethrough).

Una politica writethrough offre una coerenza dei dati, garantendo che i cambiamenti siano sempre applicati, ma risulta molto lenta, in quanto somma al tempo di scrittura della cache, il tempo d'accesso e di scrittura del livello inferiore.

Diversamente, per la politica writeback, il dato viene scritto solamente quando deve essere rimpiazzato.

Nel caso di miss, in caso sia stata adottata una politica di writeback è frequente allocare i dati in cache e poi modificarli (write-allocate). La miss in caso di writethrough normalmente è gestita scrivendo il dato direttamente ai blocchi inferiori, senza caricarli in cache.

Protocollo MESI

Per ogni linea di cache, sono associati due bit, necessari per identificare uno dei quattro stati: Motified, Exclusive, Shared, Invalid Utilizzato soprattutto per i sistemi multiprocessore.

- M - modified: La linea è disponibile in una sola cache, senza essere stata scritta in memoria.
- E - exclusive La linea è disponibile in una sola cache ma non è stata modificata.
- S - shared: La linea è potenzialmente presente in più caches. Una scrittura sulla cache locale porta un write-through, invalidando le altre caches.
- I - invalid: La linea non è disponibile nella cache e l'accesso in lettura ne causa l'allocazione

Questi stati sono utilizzati solo per la gestione dei dati, nel caso di codice non sono utilizzati, in quanto è condiviso tra tutti i processori.

Memoria Virtuale

Nel caso in cui si abbia un processore con un bus di indirizzamento a 36bit (64Gb), se si ha a disposizione un' unico banco di ram da 8Gb è comunque simulare il comportamento di 64Gb di memoria attraverso la memoria virtuale: quando si eccedono gli 8G di memoria disponibili, i restanti vengono salvati in uno spazio riservato del disco, chiamato swap file.

Si occupa il sistema operativo a gestire i Gb di dati disponibili attraverso i principi di vicinanza spaziale e temporale.

Tecnica di paginazione: Lo spazio di indirizzamento della memoria centrale viene diviso in blocchi (pagine) di dimensione fissa e continugi. Un inidirizzo è identificabile quindi attraverso un'indice di pagina ed un offset rispetto alla pagina.

Nel caso di un acceso ad indirizzo, il sistema operativo, attraverso l' address mapper, l'indirizzo virtuale viene trasformato in indirizzo fisico.

Supponendo un rimpiazzamento associativo, è presente una tabella delle pagine, dove inserito l'indirizzo, viene ritornato il valore di pagina corrispondente. Internamente alla tabella delle pagine sono presenti anche dei bit di stato, i quali indicano se la riga è presente in memoria fisica o virtuale, se la pagina è modificabile, un dirty bit ad indicare se la pagina è stata modificata ed altri.

Spesso la tabella delle pagine, essendo molto grande, è associata ad una cache, chiamata TLB (*Translation Lookaside Buffer*) che tiene traccia delle corrispondenze pagina virtuale e pagina fisica recenti. Per il rimpiazzamento è quasi sempre utilizzata la politica di Least Recently Used. Per la scrittura invece è utilizzata una strategia di write-back, registrando la presenza di cambiamenti attraverso il dirty bit.

Al momento della traduzione da indirizzo virtuale ad indirizzo fisico, se quest'ultimo non è presente si ha un "page fault" da parte del sistema operativo, ed al momento di rimpiazzamento, scrive i dati della pagina da sovrascrivere in memoria.

Nel caso in cui la dimensione delle pagine sia troppo grande, per evitare sprechi di memoria, si può ricorrere ad una tabella di indirizzamento che punta alle tabelle delle pagine, ma in questo caso sarebbero richiesti tre accessi in memoria, ed i tempi sarebbero troppo lenti.
Per questo è utile la TLB, che salvando gli ultimi indirizzamenti richiesti in memoria velocizza gli accessi ripetuti alle stesse pagine.

Normalmente un indirizzo, oltre ad avere un offset ed un numero di pagina, ha anche un numero di segmento, che nella tabella dei segmenti è utilizzato come 'tag'.

Dispositivi IO

Non è possibile pensare che i dispositivi di input/output siano sempre compatibili con il bus di sistema (AGP *Accelerated graphic protocol*, dedicato alle schede grafiche)
I dispositivi IO, nel loro bus di sistema, mettono a disposizione l'interfaccia con uno o più bus PCI semi-standard. Questo permette di avere dispositivi che funzionano a diverse velocità.

Per interfacciarsi con I/O vengono utilizzati solitamente dei registri, definiti da un certo spazio di indirizzamento I/O. I registri possono essere di tre tipologie:

- (DREG) Registro dati: trasferimento dei dati buffer, necessario per il trasferimento asincrono dei dati, registri di comando e di stato.
- (CREG) Registri di comando: dove il processore scrive i dati per inviare comandi alla periferica (es. stampa fronte retro)
- (SREG) Registri di stato: indica lo stato della periferica (es. manca carta)

I registri sono lettura e scrittura, e spesso sono necessari solo pochi bit per descrivere sia stato che comando. L'accesso avviene tramite registro.

L'accesso alle periferiche avviene attraverso indirizzi. Nel trasferimento dati col bus dipende dal parallelismo del bus del dispositivo, dalla possibilità di trasferimento a blocchi (burst), può avvenire a transazioni suddivise, necessarie per supportare l'accesso a multipli dispositivi, e dal livello di astrazione fornito dal sistema operativo.

Polling

Nel Polling o controllo di programma, il master della comunicazione tra i due dispositivi (CPU) decide come comunicare con l'IO: la CPU verifica lo stato delle periferiche fino a quando non ne trova una che richiede dati in ingresso o in uscita. Appena terminato il trasferimento dati con la periferica riprende a leggere gli stati delle periferiche.

Difetti del Polling

Parecchio tempo del processore è impiegato per controllare lo stato delle periferiche, inoltre se alcune periferiche sono lente, rallentano l'intero processo.

Nel caso di mancanza di un meccanismo di timeout per interrompere il controllo di stato delle periferiche, in caso una di esse risulti non funzionante, l'intero sistema si blocca.

La procedura è anche non scalabile, se sono connesse parecchie periferiche il processo rallenta notevolmente.

Interrupt

I dispositivi di IO inviano segnali di interrupt al processore, indicando che necessitano di trasmettere dei dati. Esistono due tipologie di interrupt: hardware veri e propri segnali elettrici che possono essere mascherabili o non mascherabili, e software, generati dalla cpu.

Gestione delle interruzioni

Passa attraverso sei fasi:

1. Notifica delle interruzioni, viene segnalato l'interrupt (software attraverso bit di flag o hardware attraverso segnali)
2. Accettazione: quelle software vengono sempre accettate, per quelle hardware se il flag di mascheramento degli interrupt è disabilitato, vengono tutte accettate, altrimenti vengono accettate solo se non mascherabili.
3. Identificazione della sorgente che ha inviato l'interrupt attraverso l'RRI, per determinare come gestire l'interrupt. Nel caso in cui più sorgenti abbiano lo stesso segnale di interrupt, per identificare quella che ha inviato il segnale si ricorre al polling.

   Diversamente in caso di interrupt vettorizzati internamente è presente un "arbitro di priorità" decide quale interruzione servire.

   Se sono interrupt vettorizzati con un controllore esterno, è presente un componente esterno PIC (*Programmable Interrupt Controller*) al processore, che riceve tutti gli interrupt e segnala alla CPU quali, quando eseguirli e la sorgente dell'interrupt attraverso un identificativo.

   In memoria è presente una tabella delle interruzioni, generata all'avvio della macchina, dove ad ogni interrupt è associato un indirizzo far, della routine di risposta all'interruzione.

   L' interrupt type ricevuto dalla periferica, viene quindi moltiplicato per 4, ottenendo l'indirizzo di memoria della rispettiva routine di risposta.
4. Salvataggio dello stato della cpu, nelle macchine intel viene caricato sullo stack, e modifica del program counter.
5. Esecuzione delle RRI
6. Ritorno al programma e ripristino dello stato della CPU precedente.

Nell' 8086, l'interfaccia di interruzioni, è un normale ciclo di bus, in cui il segnale INTA sostituisce il segnale READ.

Intel 8259

Ha 8 pin di dato dove riceve le parole di programmazione e invia l'interrupt type (ID dispositivo) al processore. Ha altri 8 pin di interrupt per le periferiche o un'altro controllore PIC slave, raggiungendo un massimo di 64 gestioni di interrupt.

Il riconoscimento della interruzione viene usato il segnale INTA (*INTerrupt Acknowledge*).

Internamente ha un registro IRR (*Interrupt Request Register*) ad 8 bit dove ogniuno di essi, se ad 1, indica che la corrispondente periferica richiede un interrupt. Un registro IMR *Interrupt Map Register*), ad 8 bit, in cui un bit ad 1 indica che l'interrupt corrispondente è mascerato.

Le informazioni ricevute, insieme all'elenco delle richieste ricevute, vengono trasmesse al priority resolver, con lo scopo di decidere quale tra gli interrupt ricevuti abbia una maggiore priorità, ed inviarlo al ISR (*In Service Register*) e successivamente trasmetterlo attraverso il bus dei dati.

L'ISR memorizza le istruzioni in esecuizione, dove il bit n viene settato se la richiesta IRQn è stata accettata dalla CPU.

Ha un blocco di logica di controllo per gestire i segnali di INTA ricevuti dal processore.

Ed un'ultimo blocco per la gestione del PIC in cascata.

Una volta inviati il vettore di interruzione, il PIC attende che la CPU invii un comando di EOI (*End Of Interrupt*) che notifichi fine dell'interruzione.

All'arrivo di EOI, PIC resetta il bit di ISR e controlla se sono presenti richieste di interrupt precedenti (rimaste in sospeso perchè con priorità minore), da inviare.

Nel caso si riceva un interrupt con priorità maggiore di quello in esecuzione, viene data la precedenza a quest'ultimo, interrompendo momentaneamente l'esecuzione di quello corrente. Questo metodo prende il nome di Fully Nested Interrupt.

Connesioni a cascata

Tutti i PIC sono collegati al bus dei dati che porta al processore. Hanno un piedino in ingresso per specificare se comportarsi come master o slave.

INTA e CS è ricevuto da tutti, gli interrupt degli slave vanno al master, mentre l'interrupt del master va al processore.

La priorità che il master associa agli slave è uguale per tutti. Durante la programmazione, viene associato un id agli slave. Al momento della gestione di un'interrupt, il master comunica attraverso il CAS, quale slave far caricare i dati sul bus.

DMA controller

Nel caso di interruzioni è possibile che il processore risulti impegnato, e non sempre pronto a gestirle. Il *Direct Memory Access* è il meccanismo per cui, sotto il controllo hardware del DMAC (*DMA Controller*) una periferica si interfaccia direttamente con la memoria senza passare dalla CPU.

Il processore vede il DMAC come una normale periferica, ed è in grado di programmarlo: il processore può mandare richieste di lettura o scrittura, indicare indirizzo di IO e memoria ed il numero di parole da trasferire.

Questo è utile per ottimizzare il trasporto dei dati, soprattutto nel caso di memorie ad accesso lento. Delegando il trasferimento dei dati al DMAC, la CPU è in grado di volgere altre operazioni che non richiedono il bus di sistema.

Nel DMAC sono presenti una serie di piedini per gli ingressi di dato, un data register e data count, che indica il numero di parole da trasferire. Contiene anche un bus degli indirizzi, utilizzato per indicarli alla memoria ed una logica di controllo, con una coppia di segnali `DMA Request` e `DMA Acknowledge`, utilizzati rispettivamente per chiedere al processore di diventare il master del bus, ed il segnale di risposta del processore.

Infine ha anche un segnale di interrupt per notificare il processore.

- La cpu programma il DMAC, indicando lettura/scrittura numero di parole, indirizzo di memoria e dispositivi IO.
- Dispositivo di IO richiede di iniziare il trasferimento
- Tramite i segnali di HOLD e HOLDA, si mette in comunicazione con il processore per prendere il controllo del bus di sistema. Il segnale HOLD non può essere mascherato. Il processore risponde con HOLDA, tenendolo attivo fino a quando il segnale HOLD non diventa basso, indicando che DMAC ha finito il trasferimento.

- La CPU non lavora sul bus ed il DMAC inizia i trasferimenti.
- DMAC rilascia il controllo del bus al processore attraverso un segnale di interrupt.

I trasferimenti possono essere *fly-by*, dove i dati vanno direttamente da dispositivo IO a memoria, senza passare dal DMAC. Non possibile nel caso di trasferimento da memoria a memoria (es. deframmentazione), dove si dovrebbe indicare nel bus sia l'indirizzo del dato sorgente che del dato destinazione. In questi casi viene utilizzata la modalità *flow-through*, memorizzando temporaneamente i dati nel DMAC.

Trasferimenti

Oltre alla modalità di trasferimento a blocchi, dagli DMAC può essere supportato il trasferimento singolo: trasferisco una parola alla volta, e per trasferire la successiva si necessita la riprogrammazione ed il protocollo di handshake, e trasferimento on-demand dove vengono trasferiti i dati fino a quando c'è richiesta.

Il DMA controller è visto come una periferica, e, al giorno d'oggi è integrato direttamente negli HD e dispositivi di IO.

Il DMA può essere anche collegato ad un bus di IO, dove sono presenti anche le altre periferiche di IO.

Memorie Esterne

le memorie secondarie, sono caratterizzate da un'alta capacità, basti costi di produzione, e la non volatilità dei dati.

Esempio disco magnetico

È un unità di memoria che può essere costituita da più dischi magnetici che ruotano a velocità costante, connessi ad un unico asse.

Ogni superficie viene chiamata faccia, ed una serie di testine, leggono i dati da ognuna di esse, muovendosi all'unisono. Le tracce su cui sono posizionate le testine, hanno tutte la stessa distanza dal centro, formando un cilindro.

Ciascuna traccia in ogni faccia, è divisa in settori, di varia densità per tenere costante la quantità di bit letti al secondo.

Le tre informazioni per identificare un dato sono: la traccia (cilindro e faccia) e settore.

Il piatto circolare è formato da materiale magnetizzabile, recentemente composto da materiale vetroso, garantendo una minore distanza della testina dal disco per una minor tempo d'accesso al dato.

La testina è fatta da materiale ferromagnetico, a forma di ferro di cavallo, attraverso una bobina genera un campo magnetico per leggere il dato. Per la scrittura la bobina induce un campo magnetico che agisce sullo strato magnetizzabile, per la lettura lo strato magnetizzabile in movimento induce corrente su bobina. Spesso sono utilizzate due testine separate per lettura e scrittura.

La testina non tocca la faccia del disco ma è separata da un cuscinetto d'aria per evitare di rovinare i dischi.

Più la testina è vicina al disco più può essere piccola, garantendo una maggiore densità dati, ma un maggior rischio di contatto.

Accesso a dato

Il tempo d'accesso t_a è determinato da t_s, il tempo per posizionare la testina sulla traccia, dipendente dalla distanza tra il cilindro corrente e quello contente il dato. t_l tempo impiegato per posizionare la testina sul dato e t_d, il tempo per leggere serialmente i dati.

SSD

Sono più veloci rispetto agli HDD, dato che non hanno tempi di latenza, con una tecnologia solo elettronica, si comportano come memorie ad accesso casuale.

È più robusto ai danni meccanici, non avendo dischi interni che si possono rovinare. Consumano e dissipano meno energia.

Hanno il difetto di avere ancora un costo per bit superiore rispetto agli hard disk. Per questo esistono tecnologie ibride, chiamate SSHD composte da una componente meccanica ed una a stato solido. Il firmware si occupa della gestione dati, mentre la componente a stato solido si comporta come una

cache.

Un'altro problema degli SSH è che le prestazioni decadono col tempo: alla lettura è necessario leggere un'intero blocco alla volta, e le operazioni di scrittura richiedono una cancellazione completa di un blocco, ed il suo completo aggiornamento.
Inoltre dopo un certo numero di utilizzi è inutilizzabile.

Vengono quindi utilizzate tecnologie per prolungarne la durata, utilizzando una cache per raggruppare funzioni di scrittura, algoritmi che dividono le scritture su differenti blocchi, e la gestione dei bad blocks [6].

Dischi ottici

Esistono diverse versioni:

- CD: Compact disk, non cancellabile, per memorizzare informazioni, tipicamente audio
- CD-ROM: Non riscrivibili, per portare dati fino a $650Mb$
- CD-R: CD Scrivibile un'unica volta
- CD-RW: CD Rewritable
- DVD: Digital Versatile Disk, versione "più grossa" del CD, contiene fino a $17G$
- DVD-R
- DVD-RW
- Blu-Ray DVD, presenta una maggiore capacità rispetto ad DVD, raggiungendo una capacità massima di $25G$

La tecnologia di base è policarbonato, e l'informazione viene codificata con dei pit (buche) su uno strato metallico riflettente. Le parti interne si dividono quindi in pit, che riflettono male la luce, e le parti land, che la riflettono bene.

In fase di lettura, attraverso un laser, si riesce a leggere le informazioni codificate sul disco.

[6] Dopo un ripetuto numero di scritture su un blocco, il dato viene scritto su un blocco differente

Strutture di interconnessione

I bus nascono dalla necessità di comunicazione tra uno o più moduli (es. Modello di Von Neumann) Internamente al bus, quando un dispositivo trasmette, tutti gli altri collegati riescono a leggere i dati. Non è possibile però che due dispositivi trasmettano in contemporanea.

Utilizzare il bus come unico metodo di comunicazione, è che si comporta come collo di bottiglia.

Le linee di comunicazione sono a singolo bit, e possono essere seriali o parallele.

Caratteristiche del bus di dati sono il tipo: dedicato o generico, l'ampiezza del bus (n_a), il metodo di arbitraggio, centralizzato (master-slave) o distribuito (dispositivi rispettano un protocollo di scrittura), temporizzazione (sincrona o asincrona) ed i tipi di trasferimento dati.

Ogni scelta è un compromesso tra prestazione e costo di produzione.

Tipi di bus

Nei bus dedicati sono assegnati tipicamente ad un insieme fisico di componenti, rendendo possibile la separazione trasmissione dati-indirizzi, utilizzabile tipicamente attraverso una linea di controllo aggiuntiva.

I bus multiplexati, necessitano di un controllo per distinguere le informazioni trasmesse (DEN, indirizzo, dati, ...). Vincola ad avere un tempo limitato per la lettura dell'indirizzo, dato che richiede di trasmettere successivamente i dati.

Un bus multiplexato ha costi inferiori, in quanto richiede meno collegamenti ma più logica fisica, riducendo quindi le prestazioni.

Metodo di arbitraggio

Può essere centralizzato, dove il processore o un'altro dispositivo dedicato gestisce il controllo del bus, o distribuito, dove attraverso un'algoritmo i dispositivi cooperano per l'accesso.

L'algoritmo utilizzato per la gestione distribuita del bus deve evitare le collisioni.

Temporizzazione

Gli eventi possono essere coordinati nel bus in modo sincrono o asincrono. Quando il bus è sincrono, è presente un clock che scandisce tutte le operazioni, campionando i dati sui fronti di salita o discesa. Sono necessari anche segnali di stabilizzazione, per ridurre i disturbi generati da segnali paralleli.

I pro sono la semplicità di progetto e controllo, il contro è lo spreco di tempo, dovuto alla richiesta di dividere ogni operazione in un numero intero di cicli di clock.

In caso di bus asincrono, ogni operazione è innescata dalla precedente, garantendo una maggior efficenza nell'uso di cicli, ma una maggiore complessità di progetto e controllo.

Ampiezza del bus

Aumentare l'ampiezza del bus vuole dire aumentare il numero di bit trasferiti, quindi il bit rate. Aumentare il bus degli indirizzi implica un aumento nello spazio di indirizzamento: il massimo intervallo di locazioni indirizzabili.

Inoltre aumentare il parallelismo porta una maggiore velocità, tutti i bus moderni sono seriali. Un suo svantaggio è l'aumento della complessità di realizzazione. I bus seriali permettono di fare linee più lunghe, diminuendo l'interferenza tra le diverse linee.

Tipi di trasferimento

Tutti i bus permettono lettura e scrittura, ma esistono anche particolari tipi di funzionamento:

- Read modify write: lettura seguita da una scrittura allo stesso indirizzo, in un solo ciclo d'accesso il dato viene letto, modificato e riscritto.
- Read after write, operazioni di controllo, indivisibili. Utilizzate nei bus per comunicare con i dispositivi IO, garantendo la corretta scrittura del dato.
- Trasferimento a blocchi, leggendo un indirizzo, prosegue la lettura fino ad n indirizzi successivi.

Bus di sistema

Il bus di sistema connette i principali elementi di un calcolatori: CPU, memorie e IO. I bus hanno lo stesso problema delle memorie, vorrebbero un'elevata capacità e prestazioni con un basso costo di produzione, ma non è possibile. Per questo esiste una gerarchia di bus interna.
Elencati dal più veloce al più lento:

- Processor Bus, utilizzato per comunicare internamente al processore, collegando i diversi registri
- Cache Bus, dedicato alla cache interna al processore
- System Bus, utilizzato per legare il processore alle memorie
- Local IO Bus, bus ad alta velocità utilizzato per collegare le periferiche critiche, come HDD e scheda video
- Standard IO Bus, utilizzato per le periferiche più lente.

Bus Unico PDP

Un unico bus, creato dalla PDP, utilizzato per fare tutte le operazioni di IO. Ha il vantaggio di essere modulare, e facilmente standardizzato.

Lo svantaggio di questo bus, è che tutti i dispositivi sono collegati allo stesso bus, indipendentemente dalla loro velocità. Per questo la velocità di clock può risultare troppo veloce / lenta.
Inoltre all'aumentare dei dispositivi connessi, la lunghezza del bus aumenta, implicando una lentezza di propagazione dei segnali.
Il bus è anche limitato in ampiezza, dato che tutti i dispositivi connessi hanno lo stesso parallelismo dei dati.

Bus PCI - Peripheral Component Interconnect