ale-cci

Architettura dei calcolatori elettronici

April 22, 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

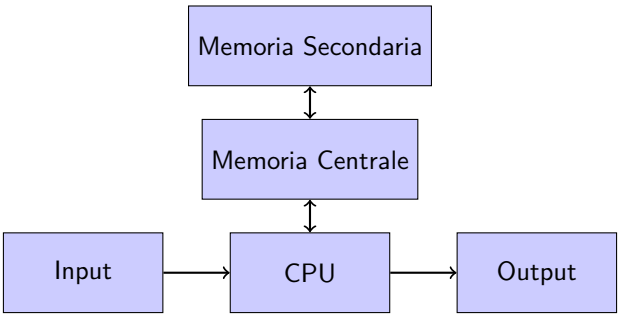
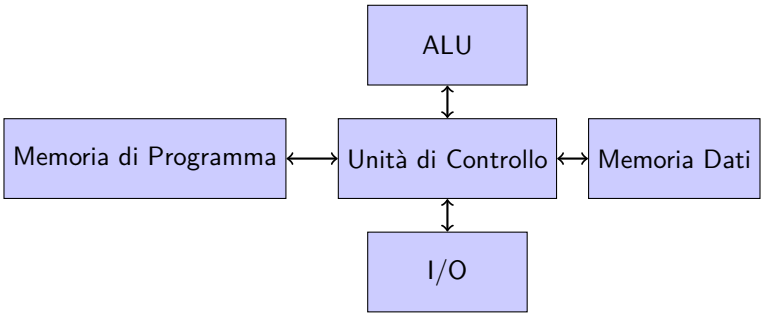


Figure 1: Computer secondo architettura di Von Neumann

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.

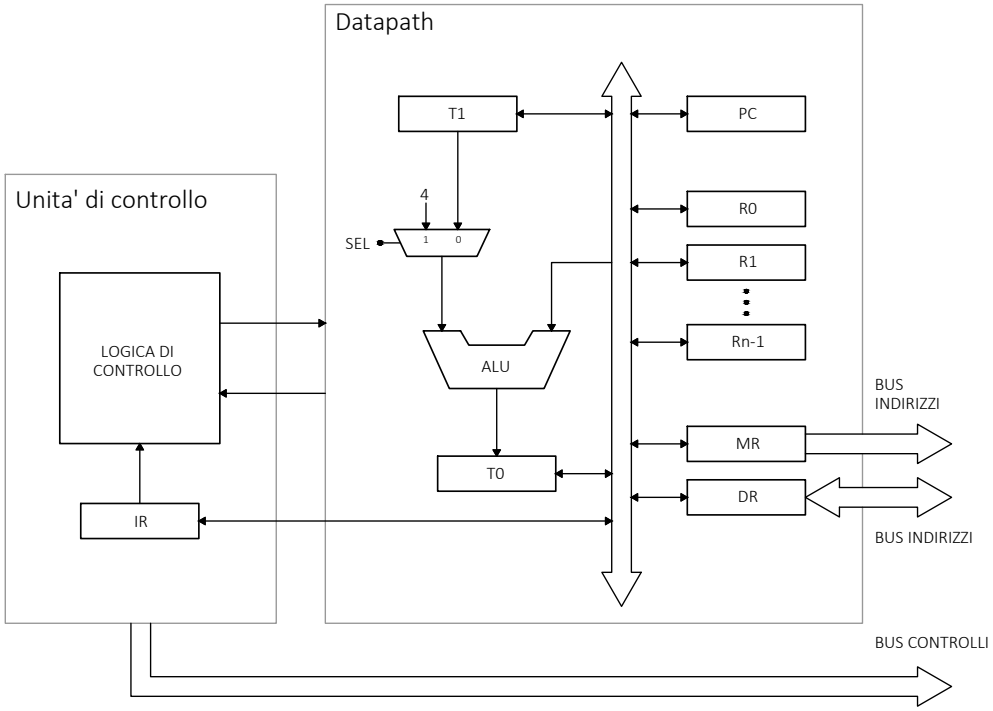
Microarchitettura CPU

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.

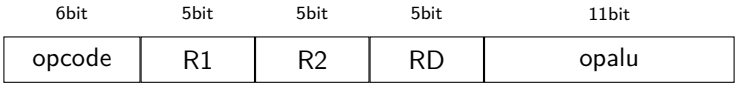
Un indirizzo indica la posizione di un byte in memoria, ad esempio l'indirizzo 2 indica il secondo byte, posizionato all'ottavo bit.
Le istruzioni sono di lunghezza fissa a 32 bit. Dato che istruzioni e dati sono salvati sempre ad indirizzi multipli di 4, il program counter è incrementato di 4 ad ogni 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

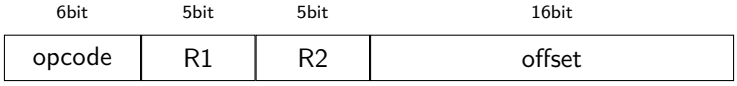
(EX). Una volta eseguita, in alcuni casi vengono scritti o letti dei dati dalla memoria (ME), ed infine dove necessario vi è la fase di writeback (WB) dove il risultato dell'operazione è scritta in un registro.

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

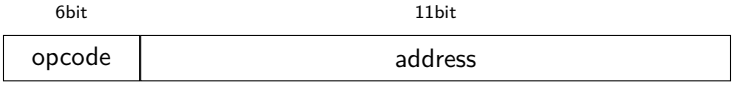
- Istruzione aritmetico/logica `add reg1, reg2, dest`



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



- Salti incondizionati `jmp 0045h`



Oltre a salto condizionato ed incondizionato esiste un terzo tipo di salto: la chiamata a funzione. È un tipo di salto incondizionato particolare, dato che quando eseguito è necessario salvare il valore del program counter per poter proseguire l'esecuzione una volta terminata la funzione.

In questa prima architettura assumiamo che non ci sia lo stack [1], quindi salviamo il program counter nell'indirizzo R_{31}.

Per selezionare le operazioni da far eseguire all'ALU è specificato un ingresso chiamato opalu. Oltre al risultato dell'operazione vengono tornati valori di flag aggiuntivi come zero flag (messo ad uno quando il risultato dell'ALU è zero) e sign flag(messo ad 1 quando il risultato è negativo).

Load e Store

Load e store permettono rispettivamente lettura e scrittura in memoria. Entrambe prendono in ingresso un registro base, uno destinazione ed un offset. L'indirizzo su cui leggere/scrivere il dato contenuto nel registro destinazione viene ottenuto sommando l'offset al registro base. 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

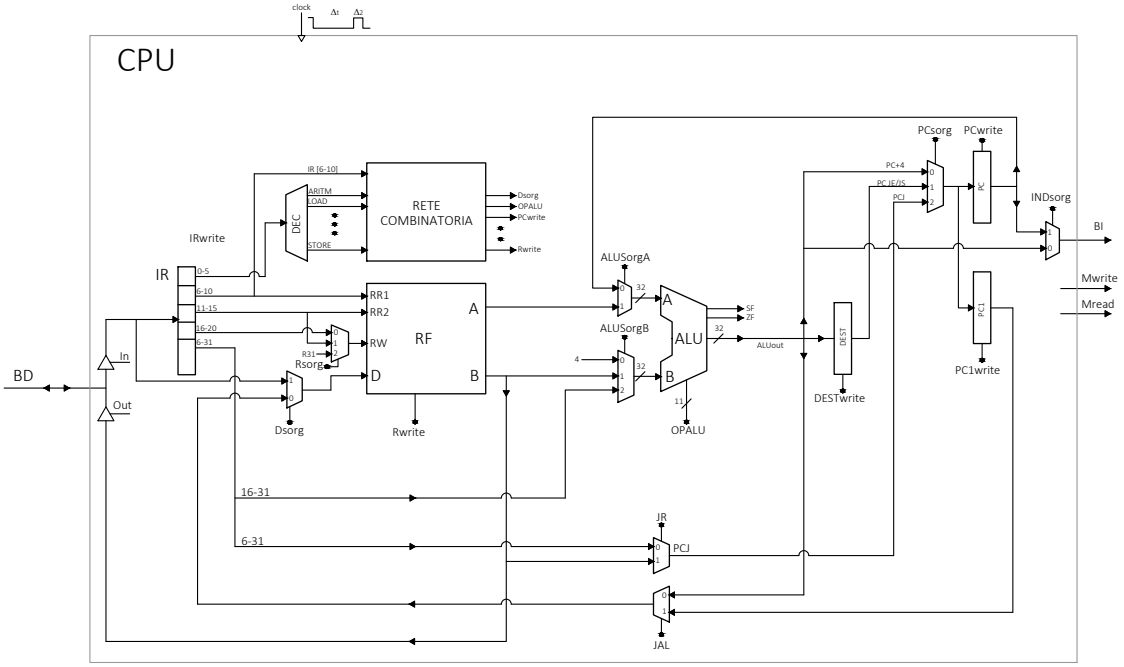
Table 1: Esempio tempi di esecuzione in architettura monociclo

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 in cui T_{mono} dipende dall'istruzione più lenta, il tempo T_{multi} dipende 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

L'architettura multiciclo per come vista sino ad ora è inefficiente, per migliorarla ci possono essere diversi modi:

- 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

Avendo preso T_{multi} uguale al tempo d'accesso alla memoria, nelle fasi in cui essa è inutilizzata, c'è spreco di tempo. L'ideale per questa architettura sarebbe che tutti gli stadi richiedano approssimativamente lo stesso tempo di esecuzione. Prendendo ad esempio $T_{\text{multi}} = 12$ (tempo dell'ALU), gli stadi di accesso alla memoria richiederanno 3 cicli di clock ($12 \cdot 3 > 30$).

Svolgendo i calcoli otteniamo un tempo medio di $86.88ns$, migliore rispetto al precedente, ma ancora più lento rispetto agli $82ns$ dell'architettura monociclo.

Osservando che con $T_{\text{multi}} = 12$ nelle fasi di memoria rimane uno spreco di $36 - 3 \cdot 12 = 6ns$ riduciamo ancora la granularità di clock a $T_{\text{multi}} = 5ns$ (fase più veloce: decodifica del writeback) saranno richiesti quindi 6 cicli di clock per le fasi che fanno accesso alla memoria, e 3 per le fasi di ALU. Il tempo medio per istruzione si riduce a $75.65ns$, migliore rispetto al caso precedente e dell'architettura monociclo.

Anticipazione delle operazioni

Partiamo dall'osservazione che in T2 si esegue un passo che serve solo a JE/JS. Possiamo quindi differenziare la fase T2 a seconda dell'istruzione. Inoltre, dato che il periodo è sufficientemente lungo, possiamo unire più istruzioni in un unico ciclo di clock:

- Per operazioni aritmetiche, dopo il fetch, richiedono una decodifica ed un operazione dell'ALU, con un totale di $5 + 12 + 5 = 22ns$ sufficiente da eseguirle nell'unica fase T2
- Per load il calcolo dell'indirizzo può essere fatto in T2, la lettura della memoria in T3 e scrittura nel registro destinazione in T4, tagliando un ciclo di clock.
- L'operazione di store, esattamente come per la load, anticipiamo il calcolo dell'indirizzo sorgente in T2, quindi la scrittura in memoria è effettuabile immediatamente all'istante T3.
- In JMP/JS è possibile aggiornare PC direttamente in T2
- In JAL è possibile aggiornare PC e la scrittura in R31. PC1 non è più necessario.

Il tempo di esecuzione medio è $81.60ns$, migliore della versione multiciclo originale e circa la stessa velocità dell'architettura monociclo.

Aumentando la granularità di clock in quest'architettura, ipotizzando un $T_{\text{multi}} = 6ns$ (dato che IF e ME richiedono $6ns$), otteniamo un tempo medio di $64.76ns$ migliore del 21% rispetto alla monociclo.

Architetture Avanzate

Prestazione dei calcolatori

Con benchmark si intende un set di programmi differenti tra loro, che rappresentano a grandi linee task frequenti eseguiti dal calcolatore. Per comparare le prestazioni di diversi calcolatori, vengono paragonati i tempi di esecuzione del benchmark.

Prestazione della CPU

Considerato l'intero tempo di esecuzione di um programma, viene definito tempo di CPU, il tempo in cui effettivamente quest'ultima è impiegata nel task: $T_{\text{cpu}} = N_{cc}T_{ck}$, dove N_{cc} è il numero di cicli di clock.
Il modello più semplice per calcolarlo è il CPI (*Clock Per Instruction*): quanti cicli di clock servono in media per 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.

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

Da cui:

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

Il paradigma RISC è di ridurre il più possibile il *CPI*, portando il problema di aumentare il numero di istruzioni richiesto per formare un'operazione. CISC diversamente, riduce il numero di istruzioni richieste per programma, ma aumentando il *CPI*.

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

Il numero di MIPS non dipende da N, pertanto a parità di benchmark e MIPS otteniamo valori diversi di $\textit{CPU}_{\textit{time}}$ se N cambia. Quindi si possono confrontare tra loro due CPU rispetto ad un deterinato benchmark, solamente se hanno lo stesso set di istruzioni.

Architetture Pipeline

Questa architettura è la più significativa soluzione per aumentare la velocità di una CPU. Aumenta il numero di istruzioni eseguite nell'unità di tempo (throughput). Rispetto all'architettura multiciclo ha una necessità di disaccoppiare le varie fasi per renderle parallelizzabili. I segnali sono passati alle fasi successive attraverso registri di latch.

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

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

È facile anche calcolare come al crescere del numero di stadi k, S_p tende ad n, ma accade solo nel caso ideale

Pipeline non ideale

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

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

Una possibile soluzione per risolvere le alle strutturali sono la duplicazione delle risorse richieste (es. due ALU), e nel caso in cui siano necessari accessi multipli alla memoria, utilizzare un'architettura di Harvard. Diversamente per risolvere il problema è ritardare l'esecuzione delle fasi che richiedono una risorsa già in uso. Ovviamente sconsigliata perché riduce il throughput.

Nel caso di alee di dato esistono, a loro volta, tre tipi di dipendenza tra istruzioni. Chiamate A e B, due istruzioni, dove A precede B:

- 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`, è ovvio che nella prima istruzione, il valore r1 è salvato nella fase di writeback, mentre durante nella seconda istruzione il valore di r1 è richiesto nella fase di execute. Prima di risolvere il problema è necessario che venga riconosciuto, ad esempio marcando quali registri richiede una particolare istruzione. Per risolverlo è possibile:

- 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

In caso di alee di controllo il program counter viene verificato nella fase di execute. È risolvibile sia eseguendo il salto con un delay di 1 ciclo di clock, o riordinando l'esecuzione delle istruzioni come nel caso precedente, calcolando prima l'indirizzo di destinazione.

Il problema principale delle alee di controllo è introdotto dai salti condizionati, dato che la condizione è nota solamente dopo la fase di execute. In questa situazione il semplice riordinamento delle istruzioni non è possibile, Infatti, per minimizzare i cicli "idle" sono utilizzate tecniche di predizione, in grado di stimare in anticipo se il salto viene preso o non viene preso. Se queste tecniche 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).

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

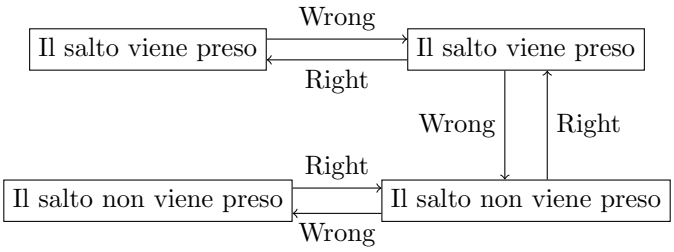


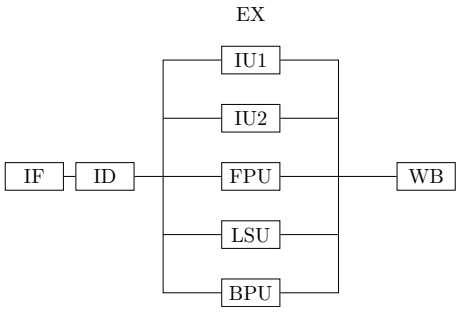
Figure 2: Esempio predizione dinamica

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

Architetture superscalari

Quello visto sino ad ora prende il nome di pipeline lineare, non viene più utilizzato perché come visto, lo speedup teorico è ben diverso dallo speedup reale. Un altro problema di questa architettura è che tutte le istruzioni attraversano tutti gli stadi, il periodo di clock è quindi determinato dallo stadio più lento (come nella multiciclo).

La soluzione è quindi introdurre non un parziale parallelismo, come nel caso della pipeline, 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

Ipotizziamo che allo stadio di execute venga emessa solo una istruzione alla volta (ID), una volta messe in esecuzione, è sono eseguite in parallelo.

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 essere invertito. Questo pone un problema di coerenza della macchina.
2. Può accadere che due istruzioni nello stesso istante richiedano di passare alla fase di writeback, generando un conflitto di risorse.

Per risolvere il primo dei due problemi 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 in un buffer in cui vengono successivamente riordinate; e attraverso un 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 un bit di validità che dice se la posizione corrente contiene informazioni,
- PC il program counter dell'istruzione, necessario per il ripristino dello stato coerente in caso di predizione di salto errata
- UF L'unità funzionale che sta eseguendo l'istruzione
- R_d Registro destinazione del risultato

Il completamento in ordine questo risulta molto dispendioso per quanto riguarda i cicli di clock. Rimane comunque vulnerabile nel caso di riferimenti alla memoria, in quanto posso avere una scrittura seguita da una lettura e riscontrare lo stesso problema che ho avuto con il register file.

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 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

Soluzione delle tre più flessibile, e permette il completamento in ordine anche del register file. Rappresenta una history di tutte le scritture nel register file, in modo da poter effettuare un rollback in caso di incoerenza, come previsioni di salto errate o interrupt.

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 al programmatore di ricordarsi specifiche sequenze di zeri ed uno, ogni microprocessore ha un proprio linguaggio assembly, in grado di tradurre con una corrispondenza 1:1 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	02h	Character Output	DL = ASCII Code
21h	03h	Keyboard Read and echo	AL = Read ASCII code
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 + 8]  ; primo valore pushato
        mov bx, [bp + 4]  ; secondo valore pushato

        pop bp
        ret
```