Architettura degli elaboratori

UniVR - Dipartimento di Informatica

Fabio Irimie

2° Semestre 2023/2024

Indice

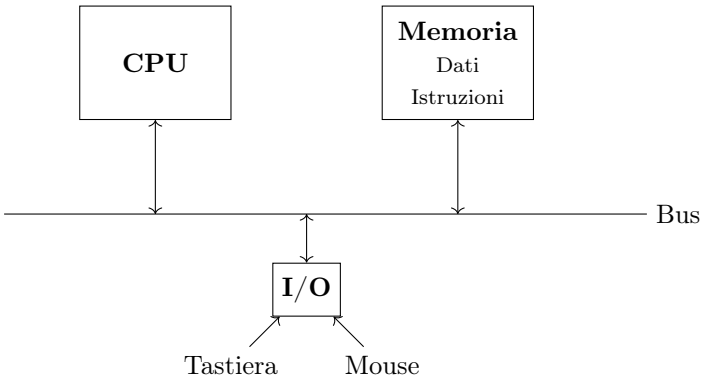
1 Architettura di Von Neumann

L'esigenza era quella di avere un'architettura che permettesse di eseguire programmi in modo automatico, senza dover cambiare il cablaggio del calcolatore, quindi il circuito deve essere abbastanza generale per poter eseguire programmi diversi.

1.1 Struttura

L'architettura di Von Neumann è composta da 5 parti principali:

- **Unità aritmetico-logica**: si occupa di eseguire le operazioni aritmetiche e logiche
- **Unità di controllo**: si occupa di controllare il flusso delle istruzioni
- **Memoria**: contiene i dati e le istruzioni
- **Input/Output**: permette di comunicare con l'esterno
- **Bus**: permette di trasferire i dati tra la memoria e l'unità aritmetico-logica (generalmente in oro)



1.2 Caratteristiche

Le istruzioni hanno bisogno di un'operazione che permetta di effettuare dei salti, in modo da poter implementare i cicli e le strutture di controllo. Inoltre le istruzioni devono essere eseugite in sequenza (un'istruzione alla volta).

1.3 CPU

Ogni processore ha un'set di istruzioni diverso in base all'architettura, questo set di istruzioni è chiamato **ISA** (Instruction Set Architecture). **Assembly** è un linguaggio di programmazione che permette di scrivere programmi in base all'ISA del processore e questo linguaggio viene tradotto in linguaggio binario attraverso un **assembler**. In questo corso viene usata l'architettura x86 (80x86).

1.3.1 Modello semplificato

Per rappresentare il funzionamento di un processore si può usare un modello semplificato rappresentato ad alto livello. Questo modello è composto da:

- **Central Processing Unit (CPU)**: esegue le istruzioni
- **Control Unit (CU)**: controlla il flusso delle istruzioni
- **Bus Dati (BD)**: trasferisce i dati alla CPU
- **Bus Istruzioni (BI)**: trasferisce le istruzioni alla CPU
- **Bus di Controllo (BC)**: trasferisce i segnali di controllo
- **Memory Address Register (MAR)**: contiene l'indirizzo di memoria da leggere
- **Memory Data Register (MDR)**: contiene i dati letti dalla memoria
- **Program Counter (PC)**: tiene conto dell'indirizzo dell'istruzione da eseguire
- **Instruction Register (IR)**: contiene l'istruzione corrente
- **Program Status Word (PSW)**: contiene i flag del processore (es. zero, carry, overflow). È come se fosse un array in cui ad ogni indice corrisponde un flag per ogni operazione.
- **Register File**: contiene i registri del processore (es. EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP)
- **Arithmetic Logic Unit (ALU)**: esegue le operazioni aritmetiche e logiche

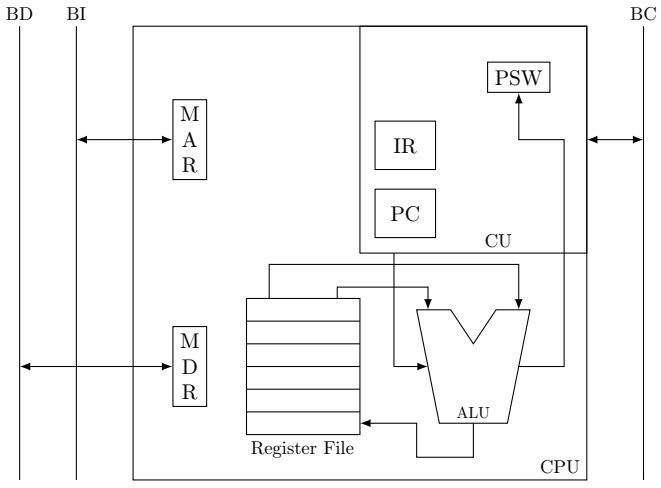


Figura 1: Struttura di un processore

Il flusso di esecuzione delle istruzioni è il seguente:

- **Fetch**: CU legge l'istruzione dalla memoria
- **Decode**: CU decodifica l'istruzione
- **Execute**: ALU esegue l'istruzione

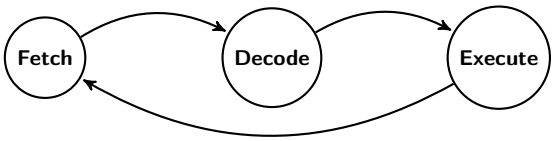
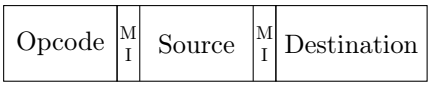


Figura 2: Flusso di esecuzione delle istruzioni

2 Assembly (Intel x86)

2.1 Codifica

Ogni istruzione è codificata nel seguente modo:



2.2 Istruzioni

2.2.1 Istruzioni base

- `MOVL <source> <destination>`: copia il contenuto di un registro (o costante) in un altro. Questa istruzione di solito viene utilizzata per spostare i valori dalla memoria ai registri e viceversa, in modo da poter effettuare operazioni solo su dati presenti nei registri e non direttamente in memoria, questo rende l'esecuzione più efficiente.
- `NOP`: non fa nulla e occupa solo un byte. La sua utilità è quella di "riempire i buchi", cioè delle zone di memoria non occupate da nessuna istruzione.

2.2.2 Istruzioni aritmetiche

- `ADDL <source> <destination>`: somma il contenuto di due registri (o costante). Siccome sono disponibili solo 2 parametri, il risultato viene salvato nel secondo parametro perchè viene visto sia come sorgente che destinazione per evitare di aggiungerne un terzo.
- `SUBL <source> <destination>`: sottrae il contenuto di due registri (o costante)
- `MULL <source> <destination>`: moltiplica il contenuto di due registri (o costante)
- `INC <source>`: incrementa il contenuto di un registro (o costante) di 1
- `DEC <source>`: decrementa il contenuto di un registro (o costante) di 1

2.2.3 Istruzioni logiche

- `CMPL <source> <destination>`: confronta il contenuto di due registri (o costanti) e modifica il flag `PSW` in base al risultato del confronto.

2.2.4 Istruzioni di salto

Se il salto è **assoluto** l'indirizzo fa riferimento alla memoria diretta, mentre se il salto è **relativo** l'indirizzo è relativo al Program Counter.

- `JMP <etichetta>`: salta all'istruzione con etichetta
- `JE <etichetta>`: salta all'istruzione con etichetta se i flag sono settati in modo che i due operandi siano uguali
- `JNE <etichetta>`: salta all'istruzione con etichetta se i flag sono settati in modo che i due operandi siano diversi
- `JG <etichetta>`: salta all'istruzione con etichetta se i flag sono settati in modo che il primo operando sia maggiore del secondo
- `JGE <etichetta>`: salta all'istruzione con etichetta se i flag sono settati in modo che il primo operando sia maggiore o uguale del secondo
- `JL <etichetta>`: salta all'istruzione con etichetta se i flag sono settati in modo che il primo operando sia minore del secondo
- `JLE <etichetta>`: salta all'istruzione con etichetta se i flag sono settati in modo che il primo operando sia minore o uguale del secondo

Comparando e poi utilizzando i salti si possono implementare le strutture di controllo come i cicli e le condizioni. Nell'etichetta si può inserire un'indirizzo di memoria assoluto che permette di saltare a quell'indirizzo, questo però non è molto utile perchè il programma potrebbe essere caricato in un'area diversa della memoria.

2.2.5 Istruzioni di gestione dello Stack

- `PUSHL <source>`: inserisce il contenuto di un registro (o costante) nello stack
- `POPL <destination>`: estrae il contenuto dello stack e lo mette in un registro (o costante)
- `CALL <etichetta>`: salva l'indirizzo successivo al Program Counter nello stack e salta all'istruzione con etichetta

2.2.6 Metodi di indirizzamento

I metodi di indirizzamento (MI) sono diversi modi per accedere ai dati in memoria, i più comuni sono:

- **Registro**: Un'istruzione può accedere direttamente ai registri ad esempio: `MOVL %EAX, %EBX`

- **Immediato**: Un'istruzione può accedere direttamente ai dati ad esempio: `MOVL $10, %EBX`
- **Assoluto**: Un'istruzione può accedere direttamente ai dati ad esempio: `MOVL DATO, %EBX`
  dove `DATO` è un'etichetta che punta ad un'indirizzo di memoria
- **Indiretto Registro**: Un'istruzione può contenere un registro che punta ad un altro registro ad esempio: `MOVL (%EAX), %EBX`
- **Indiretto Registro con Spiazzamento**: Un'istruzione può mettere un offset rispetto al registro contenuto nell'istruzione ad esempio: `MOVL $8(%EAX), %EBX`

Non tutte le istruzioni ammettono tutti i metodi di indirizzamento e alcuni metodi di indirizzamento possono essere usati solo con alcune istruzioni.

2.3 Esempi

Un esempio di codice in C è il seguente:

```
...
int a; // INDA (etichetta che punta ad un indirizzo di memoria con
  valore intero)
int b; // INDB
int c; // INDB
...
a = 5; // %EAX
b = 10; // %EBX

if (a > b) {
  c = a - b; // %ECX
} else { // ELSE
  c = a + b;
}
```

La traduzione in assembly è la seguente:

```
MOVL INDA, %EAX // Ridondante
MOVL $5, %EAX
MOVL INDB, %EBX // Ridondante
MOVL $10, %EBX
MOVL %EAX, %ECX
COMPL %EAX, %EBX
JLE ELSE
SUBL %ECX, %EAX
JMP ENDIF
ELSE:
ADDL %EBX, %ECX
ENDIF:
```

Un altro esempio di un for loop in C:

```
for (int i = 0; i < 10; i++) { // int i; %EDX
  ...
}
```

La traduzione in assembly è la seguente:

```
MOVL $0, %EDX
FOR:
COMPL $10, %EDX
JE ENDFOR
...
INC %EDX
JMP FOR
ENDFOR:
```

2.4 File assembly

Un file assembly ha estensione `.s` e può contenere diverse sezioni:

- **.section .data**: contiene le variabili globali e le costanti

```
.section .data
hello:
  .ascii "Hello, world!\n" ; Dichiarazione di una stringa
  costante

hello_len:
  .long . - hello ; Lunghezza della stringa (posizione corrente
  (.) - posizione iniziale)
```

- **.section .text**: contiene il codice assembly composto da istruzioni, etichette e sottoprogrammi

```
.section .text
.global _start ; Nome convenzionale del punto di inizio del
  programma
_start:
  movl ...
  ...
```

- **.section .bss**: contiene le variabili globali non inizializzate (spazio da riservare)

2.5 Compilazione

Per compilare un file assembly si compiono i seguenti passi:

1. **Compilazione**: si compila il file assembly con il comando `as` che crea un file binario (`.o`) contenente l'implementazione di ogni singolo file.

2. **Linking**: si uniscono i file binari con il comando `ld` che crea un file eseguibile a partire dai file binari.
3. **Esecuzione**: si rende eseguibile il file e si esegue con il comando `./<nomefile>`

3 Memoria

La memoria è una lista indicizzata di celle, a cui ognuna è associata un indirizzo. La memoria è composta da due parti principali:

- **Codice**: contiene le istruzioni
- **Dati statici**: contiene i dati

Non si può sapere a priori dove verrà caricato il programma in memoria, quindi è necessario utilizzare lo spostamento relativo per accedere ai dati e alle istruzioni.

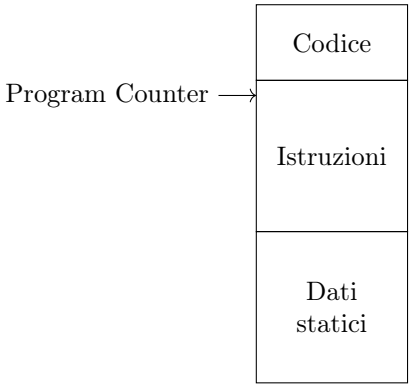


Figura 3: Struttura della memoria

Definizioni utili 3.1
***Footprint**: è l'area di memoria occupata da un programma:*

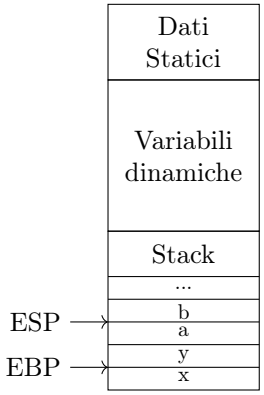
- ***L**: 32 bit*
- ***V**: 16 bit*
- ***B**: 8 bit*

3.1 Memoria dinamica

La memoria dinamica è composta da due parti principali:

- **Heap**: contiene le variabili allocate dinamicamente e ha una dimensione variabile

- **Stack**: contiene le variabili locali e i parametri delle funzioni. Ha una dimensione fissa e limitata. Lo stack cresce con la modalità **LIFO** (Last In First Out), cioè l'ultimo elemento inserito è il primo ad essere estratto e nell'architettura x86 cresce verso l'alto. Lo stack è composta anche da 2 puntatori:
  - **ESP** (Extended Stack Pointer): punta all'ultimo elemento inserito nello stack
  - **EBP** (Extended Base Pointer): punta alla base dello stack



Per gestire i dati nello stack si utilizzano le seguenti istruzioni:

- `PUSHL <source>`: inserisce il contenuto di un registro (o costante) nello stack
- `POPL <destination>`: estrae il contenuto dello stack e lo mette in un registro (o costante)

3.2 Richiamare una funzione

Per richiamare uan funzione bisogna far saltare il Program Counter all'indirizzo della funzione e poi salvare l'indirizzo successivo nello stack. Per fare ciò si utilizza l'istruzione `CALL`.

- `CALL <etichetta>`: salva l'indirizzo successivo al Program Counter nello stack e salta all'istruzione con etichetta. Quando la funzione termina, per tornare al punto di chiamata si utilizza l'istruzione `RET`.
- `RET`: estrae l'indirizzo successivo al Program Counter dallo stack e salta a quell'indirizzo (torna al punto di chiamata).

Per recuperare i dati in memoria si utilizza l'istruzione `LEAL` (Load Effective Address):

- `LEAL <source>, <destination>`: prende l'indirizzo di memoria in cui è stato salvato qualcosa e lo mette in un registro