Architettura degli elaboratori

UniVR - Dipartimento di Informatica

Fabio Irimie

2° Semestre 2023/2024

Indice

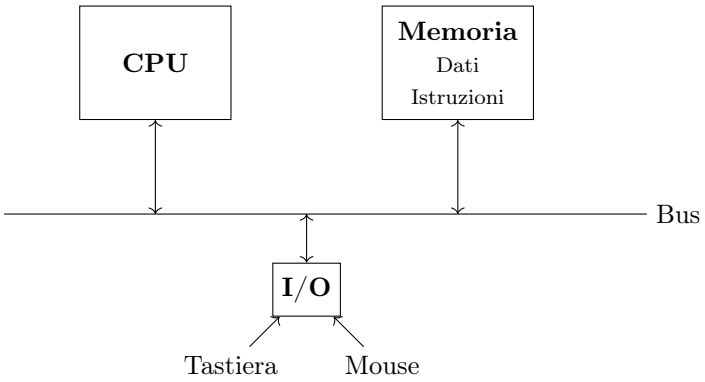
1 Architettura di Von Neumann

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

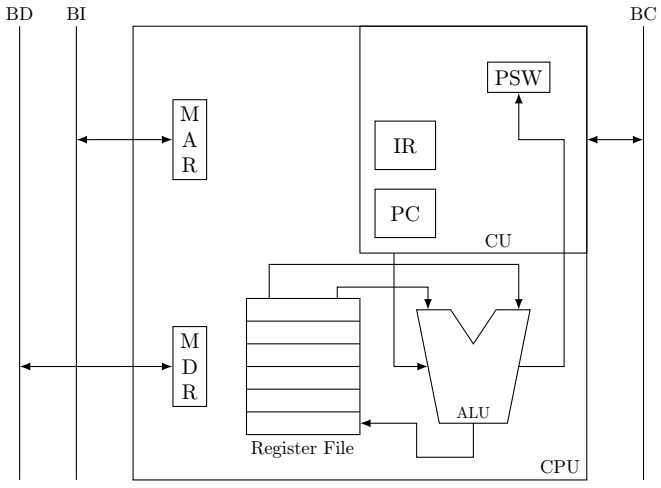


Figura 1: Struttura di un processore

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

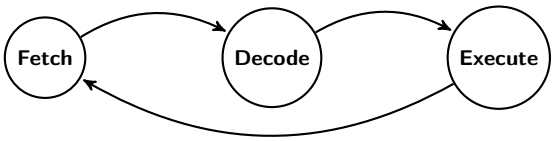
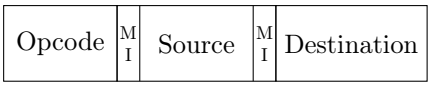


Figura 2: Flusso di esecuzione delle istruzioni

2 Assembly (Intel x86)

2.1 Codifica

Ogni istruzione è codificata nel seguente modo:



2.2 Istruzioni

2.2.1 Istruzioni di inizializzazione

- `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.

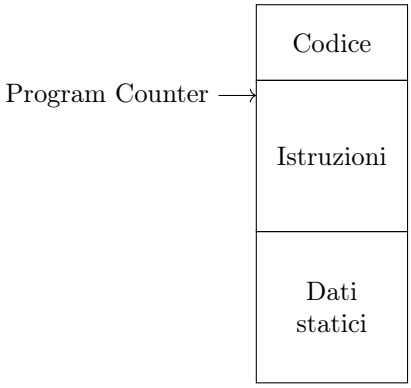


Figura 3: Struttura della memoria

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

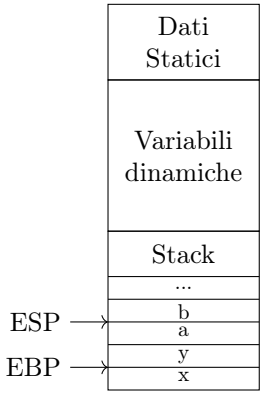


Figura 4: Struttura della memoria

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

3.3 Struttura dettagliata della CPU

Di seguito è riportato uno schema più dettagliato dei componenti della CPU in modo da capire come vengono eseguite le istruzioni.

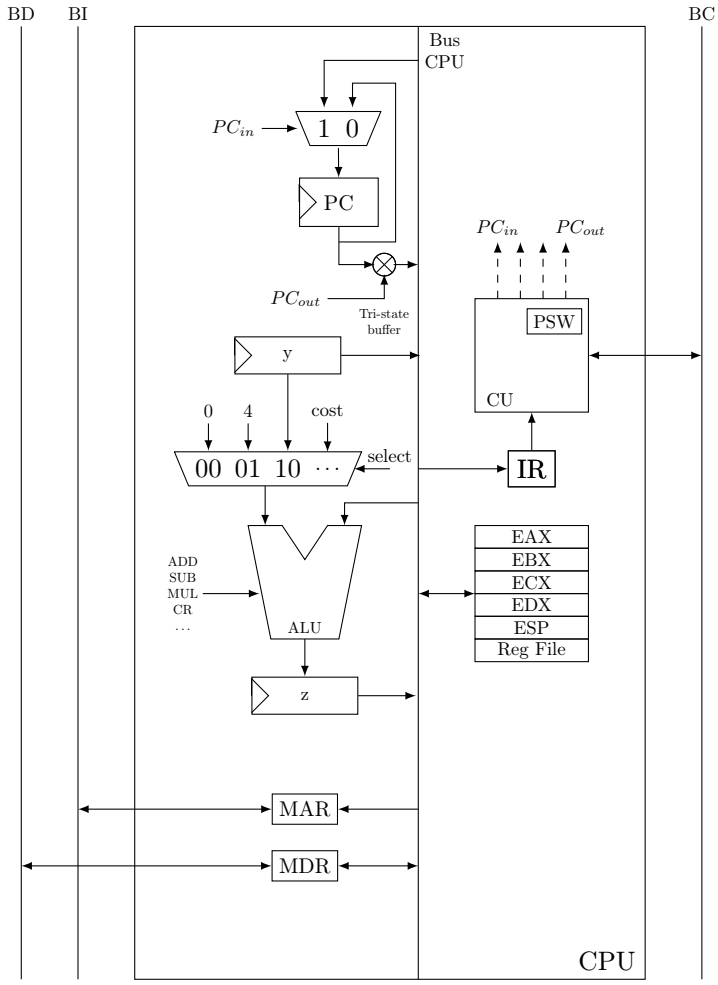


Figura 5: Schema della CPU

Da questo schema si possono notare le seguenti caratteristiche:

- Nella gestione del Program Counter il segnale passa attraverso un **Buffer Tri-State** che permette di disabilitare il segnale.
- Per mandare 2 valori alla ALU si utilizza un registro collegato ad un multiplexer che permette di selezionare quale valore mandare alla ALU. Nel multiplexer sono cablate delle costanti utili da passare alla ALU.
- Ogni indirizzo di memoria è gestito in **byte** indipendentemente. Quindi per accedere a parole da 32 bit bisogna andare avanti di 4 byte, mentre per accedere a parole da 64 bit bisogna andare avanti di 8 byte.

4 Micro operazioni

Le micro operazioni sono le operazioni elementari che la CPU esegue per eseguire un'istruzione.

Definizioni utili 4.1
CPI *(Clock Per Instruction): è il numero di cicli di clock necessari per eseguire un'istruzione. L'obiettivo è avere un CPI il più basso possibile.*

Esempio 4.1
Andiamo ad analizzare la sequenza di micro operazioni (Fetch, Decode, Execute) per la seguente istruzione:

```
MOVL %EAX, %EBX
```

F 1. PC_{out}, MAR_{in}, $READ$, $SELECT_4$, ADD, Z_{in}
Il Program Counter manda l'indirizzo di memoria in cui si trova l'istruzione da eseguire al Memory Address Register e manda un segnale di lettura.
Il segnale di selezione 4 manda un segnale al multiplexer per selezionare il valore da mandare alla ALU e il segnale di addizione manda un segnale alla ALU per sommare 4 all'indirizzo di memoria. Ciò vuol dire che l'indirizzo di memoria successivo è l'indirizzo di memoria corrente + 1 word. Tutto ciò per incrementare il Program Counter in modo da accedere all'istruzione successiva.

2. $WMFC$, Z_{out}, PC_{in}
WMFC*(Wait for Memory Fetch Complete): è un segnale che blocca il clock successivo della CPU finchè il dato viene messo nel bus dati.*
Siccome in questo ciclo di clock il bus non viene utilizzato viene sfruttato per mandare il segnale di incremento al Program Counter.
Il segnale di lettura manda un segnale di attesa finchè il dato non viene messo nel bus dati.

3. MDR_{out}, IR_{in}
Il Memory Data Register manda il dato letto dall'indirizzo di memoria all'Instruction Register.

DE 4. EAX_{out}, EBX_{in}, END
Il contenuto del registro EAX viene mandato in uscita e viene messo in ingresso al registro EBX. Successivamente viene messo a 1 il segnale di fine che fa ripartire il ciclo di Fetch-Decode-Execute.

Esempio 4.2
Istruzione:

```
MOVL (%EAX), %EBX
```

F
1. PC_{out}, MAR_{in}, $READ$, $SELECT_4$, ADD, Z_{in}
2. $WMFC$, Z_{out}, PC_{in}
3. MDR_{out}, IR_{in}

D
4. EAX_{out}, MAR_{in}, $READ$
5. $WMFC$

E
6. MDR_{out}, EBX_{in}, END

Esempio 4.3
Istruzione:

```
ADDL $4, %ECX
```

La costante 4 è già presente nell'Instruction Register, quindi non c'è bisogno di andare a leggerla dalla memoria.

F
1. PC_{out}, MAR_{in}, $READ$, $SELECT_4$, ADD, Z_{in}
2. $WMFC$, Z_{out}, PC_{in}
3. MDR_{out}, IR_{in}

DE
4. $OFFSET_{IR_{out}}$, y_{in}
   L'offset dell'Instruction Register serve per prendere il pezzo in cui è situata la costante 4
5. ECX_{out}, $SELECT_y$, ADD, Z_{in}
6. Z_{out}, ECX_{in}, END

Esempio 4.4
`JZ END` *(salto relativo)*
Il valore dell'etichetta `END` *è già calcolato dall'assembler e viene memorizzato nell'Instruction Register*

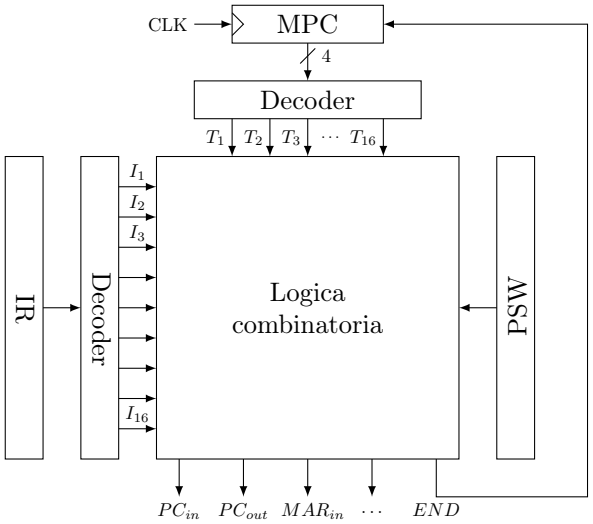
F
1. PC_{out}, MAR_{in}, $READ$, $SELECT_4$, ADD, Z_{in}
2. $WMFC$, Z_{out}, PC_{in}
3. MDR_{out}, IR_{in}

DE
4. *(if ZERO == 0 END)* , $OFFSET_{IR_{out}}$, y_{in}
   L'if e il resto vengono eseguiti insieme, sempre
5. PC_{out}, $SELECT_y$, ADD, Z_{in}
6. Z_{out}, PC_{in}, END

4.1 Struttura della Control Unit

Per rappresentare i cicli di clock delle micro istruzioni si utilizza un registro chiamato **Micro Program Counter** che indica la prossima micro istruzione da eseguire. È un contatore con un segnale di reset che permette di far ripartire l'esecuzione delle micro istruzioni. Questo registro viene poi collegato ad un decoder con 16 uscite che indica il tempo del ciclo di clock. Se il segnale $T_2 = 1$ allora il segnale $PC_{in} = 1$, quindi $PC_{in} = T_2$. È presente poi un decoder che parte dall'Istruction Register e ha in uscita tutte le istruzioni I_n. Si possono così realizzare tutte le equazioni in logica combinatoria, ad esempio:

$$END = (I_1 + I_2 + I_3 + \ldots) \cdot T_6 + I_3 \cdot T_4 \cdot \overline{ZERO}$$



Esiste una memoria che contiene tutte le micro istruzioni chiamata **Firmware**, ma **CPU cablate** in questo modo non si realizzano più.

5 Dispositivi di input e output

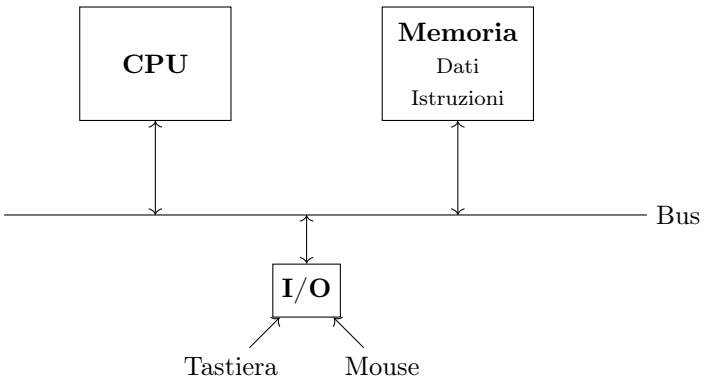


Figura 6: Schema di un sistema con dispositivi di input e output

Per poter ottenere un'interazione con l'utente è necessario avere dei dispositivi di input e output e a loro volta devono essere codificati per far corrispondere l'intenzione dell'utente con l'azione del computer. La strutture del microcontrollore input/output è composto da:

- **Dato**: contiene i dati da inviare o ricevere
- **Stato**: contiene lo stato del dispositivo
- **MC**: contiene il microcontrollore del dispositivo
- **IntA/D**: contiene l'interfaccia di analogico/digitale

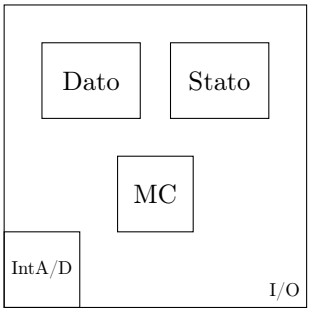


Figura 7: Struttura del microcontrollore input/output

La CPU accede ai valori dei registri di input/outpit tramite degli indirizzi che vengono riservati in un intervallo di memoria. Gli indirizzi di `Dato` e `Stato` sono:

- `IND DATA KEY` (Dato)
- `IND STATUS KEY` (Stato)

Questi indirizzi sono assegnati in fase di progettazione del sistema e sono fissi, ma si possono anche cambiare in certe architetture.

Ogni bit nel registro `status` ha un preciso significato, ad esempio se vale 0 significa che nessun tasto è stato premuto, se vale 1 significa che un tasto è stato premuto.

Un esempio in assembly è il seguente:

```
TEST_KEY:
 MOVL IND_STATUS_KEY, %EAX ; Carica lo stato della tastiera nel
   registro EAX
 CMPL $0, %EAX ; Compara il valore di EAX con 0
 JE TEST_KEY ; Se EAX = 0 allora salta a TEST_KEY
 MOV IND_DATA_KEY, %EBX ; Carica il tasto premuto nel registro EBX
 MOV %EBX, INDC ; Carica il tasto premuto nel registro C
```

La verifica di un dispositivo di input/output è un'operazione che viene detta **polling**.

Ogni volta che si vuole leggere un dato da un dispositivo di input/output si deve effettuare una **SVC** (Supervisor Call) che permette di richiamare del pezzo di codice al livello del sistema operativo che permette di effettuare diverse operazioni.

5.1 Ottimizzazione

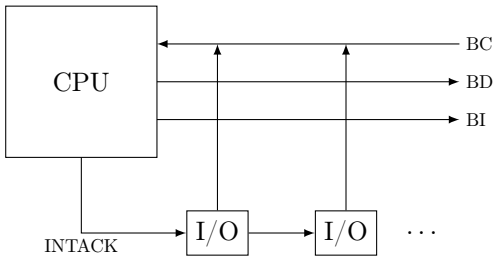
Ogni operazione di lettura effettuata con il bus richiede circa 10 cicli di clock.

```
TEST_KEY:
 MOVL IND_STATUS_KEY, %EAX ; 1 Read 1 Read Bus
 CMPL $0, %EAX ; 1 Read
 JE TEST_KEY ; 1 Read
 MOV IND_DATA_KEY, %EBX
 MOV %EBX, INDC
```

In totale si avranno $10 + 3$ cicli di clock ≈ 10. Con una frequenza di $10GHz$ si avranno $10^9 clock/sec$. Visto che un umano può premere un tasto al massimo 10 volte al secondo, si può dire che la frequenza di lettura è troppo alta, quindi si sprecano cicli di clock e non può essere gestito in polling.

5.2 Interrupt

Al posto di fare polling, cioè la CPU che controlla continuamente lo stato del dispositivo, si può utilizzare un **interrupt** che è un segnale hardware che interrompe il normale flusso di esecuzione del programma solo quando il dispositivo è pronto. La CPU **prima di ogni fetch** controlla se c'è qualche richiesta di interrupt.



Esiste un bit $\overline{INTERRUPT}$ che è sempre a 1 (negato a 0) che vale 1 solo quando c'è una **interrupt request**. Ogni interrupt ha un valore che contiene un pezzo di codice chiamato **Interrupt Service Routine (ISR)** che viene passato alla CPU attraverso il `INTACK` (Interrupt Acknowledge). Le ISR sono gestite dal sistema operativo e tutto l'insieme viene chiamato **Device Driver**. Ogni dispositivo input/output ha interrupt con valori diversi.

Siccome l'esecuzione di un interrupt può modificare i registri della CPU può esserci qualche conflitto con dei programmi già in esecuzione, quindi c'è bisogno di un meccanismo per eseguire l'ISR senza che vengano modificati i registri della CPU. Questo viene effettuato dal dispositivo input/output che salva il valore dei registri `PSW` e `PC` ed eventuali altri registri salvaldoli nello stack prima di eseguire l'ISR.

6 Laboratorio

6.1 Vantaggi e svantaggi di assembly

6.1.1 Vantaggi

Siccome assembly è un linguaggio di basso livello, è molto vicino all'hardware e quindi è possibile:

- accedere direttamente ai **registri** della CPU
- scrivere **codice ottimizzato** per una specifica architettura di CPU
- ottimizzare le **sezioni "critiche"** dei programmi

6.1.2 Svantaggi

I principali svantaggi sono:

- possono essere richieste **molte più righe** di codice
- è facile introdurre dei **bug** perchè la programmazione è più complessa
- il **debugging** è complesso
- **non è garantita la compatibilità** del codice per altri hardware

6.2 Utilità

Assembly permette di gestire direttamente il funzionamento della CPU, di conseguenza, i programmi Assembly, una volta compilati sono tipicamente più veloci e più piccoli dei programmi scritti in linguaggi di alto livello. Per questo motivo, l'assembly è utilizzato per scrivere codice che deve essere il più veloce possibile, come ad esempio i driver di hardware specifici.

6.3 Registri

Tutti i processori della famiglia x86 hanno i seguenti registri: AX, BX, CX, DX, CS, DS, ES, SS, SP, BP, SI, DI, IP, FLAGS.

Originariamente i registri AX, BX, CX, DX, SP, BP, SI, DI, IP e FLAGS avevano una dimensione di 16 bit. A partire dal 80386, la loro dimensione è stata estesa a 32 bit e al loro nome è stato aggiunto il prefisso E (Extended). Per ragioni di retrocompatibilità, i registri di 16 bit possono essere utilizzati anche nei processori a 32 bit utilizzando il loro nome originale.

6.3.1 Registri general purpose

I seguenti registri sono generici, pertanto è possibile assegnargli qualunque valore. Tuttavia, durante l'esecuzione di alcune istruzioni i registri generici vengono utilizzati per memorizzare valori ben determinati:

- **EAX** (Accumulator register): è usato come accumulatore per operazioni aritmetiche e contiene il risultato dell'operazione
- **EBX** (Base register): è usato per operazioni di indirizzamento della memoria
- **ECX** (Counter register): è usato per "contare", ad esempio nelle operazioni di loop
- **EDX** (Data register): è usato nelle operazioni di input/output, nelle divisioni e nelle moltiplicazioni

6.3.2 Registri di segmento

CS, DS, ES e SS sono i **registri di segmento** (segment registers) e devono essere utilizzati con cautela:

- **CS** (Code Segment): punta alla zona di memoria che contiene il codice. Durante l'esecuzione del programma, assieme al registro IP, serve per accedere alla prossima istruzione da eseguire (attenzione: non può essere modificato!)
- **DS** (Data Segment): punta alla zona di memoria che contiene i dati
- **ES** (Extra Segment): può essere usato come registro di segmento ausiliario
- **SS** (Stack Segment): punta alla zona di memoria in cui risiede lo stack

6.3.3 Registri puntatore

ESP, EBP, EIP sono i registri puntatore (pointer registers):

- **ESP** (Stack Pointer): punta alla cima dello stack. Viene modificato dalle operazioni di PUSH (inserimento di un dato nello stack) e POP (estrazioni di un dato dallo stack). Si ricordi che lo stack è una struttura di tipo LIFO (Last In First Out - l'ultimo che entra è il primo che esce). È possibile modificarlo manualmente ma occorre cautela!
- **EBP** (Base Pointer): punta alla base della porzione di stack gestita in quel punto del codice. È possibile modificarlo manualmente ma occorre cautela!
- **EIP** (Instruction Pointer): punta alla prossima istruzione da eseguire. Non può essere modificato!

6.3.4 Registri indice

ESI e EDI sono i registri indice (index registers) e vengono utilizzati per operazioni con stringhe e vettori:

- **ESI** (Source Index): punta alla stringa/vettore sorgente
- **EDI** (Destination Index): punta alla stringa/vettore destinazione
- **EFLAGS**: è utilizzato per memorizzare lo stato corrente del processore. Ciascuna flag (bit) del registro fornisce una particolare informazione. Ad esempio, la flag in prima posizione (carry flag) viene posta a 1 quando c;è stato un riporto o un prestito durante un'operazione aritmetica; la flag in seconda posizione (parity flag) viene usata come bit di parità e viene posta a 1 quando il risultato dell'ultima operazione ha un numero pari di 1

6.3.5 Composizione dei registri

I registri sono composti da 32 bit e possono essere divisi in registri più piccoli:

- **EAX**: AX, AH, AL
- **EBX**: BX, BH, BL
- **ECX**: CX, CH, CL
- **EDX**: DX, DH, DL
- **ESP**: SP
- **EBP**: BP
- **ESI**: SI
- **EDI**: DI

6.3.6 Composizione del registro EFLAGS

Il registro EFLAGS è composto da 32 bit e ogni bit corrisponde ad un flag:

- **ZF** (Zero flag): impostato a 1 se il risultato dell'operazione è 0
- **SF** (Sign flag): impostato a 1 se il risultato dell'operazione è un numero negativo, a 0 se è positivo (rappresentazione in complemento a 2)
- **OF** (Overflow flag): impostato a 1 nel caso di overflow di un'operazione
- **TF** (Trap flag): impostato a 1 genera un'interruzione ad ogni istruzione. Utilizzato per l'esecuzione passo-passo dei programmi
- **IF** (Interrupt flag): impostato a 1 abilita gli interrupt esterni, con 0 li disabilita
- **DF** (Direction flag): impostato a 1 indica che nelle operazioni di spostamento di stringhe i registri DI e SI si autodecrementano (con 0 tali registri si auto incrementano)

6.4 Modalità di indirizzamento

Si rifersce al modo in cui un'istruzione assembly accede ai dati in memoria e può essere:

- **Indirizzamento a registro**: l'operando è contenuto in un registro ed il nome del registro è specificato nell'istruzione. Ad esempio:

  `%Ri`

- **Indirizzamento diretto** (o assoluto): l'operando è contenuto in una locazione di memoria, e l'indirizzo della locazione viene specificato nell'istruzione. Ad esempio:

  `(IND)`

- **Indirizzamento immediato** (o di costante): l'operando è un valore costante ed è definito esplicitamente nell'istruzione. Ad esempio:

  `$VAL`

- **Indirizzamento indiretto**: l'indirizzo di un operando è contenuto in un registro o in una locazione di memoria. L'indirizzo della locazione o il registro viene specificato nell'istruzione. Ad esempio:

  `(%Ri)` o `($VAL)`

- **Indirizzamento indicizzato** (Base e spiazzamento): l'indirizzo effettivo dell'operando è calcolato sommando un valore costante al contenuto di un registro. Ad esempio:

  `SPI(%Ri)`

- **Indirizzamento con autoincremento**: l'indirizzo effettivo dell'operando è il contenuto di un registro specificato nell'istruzione. Dopo l'accesso, il contenuto del registro viene incrementato per puntare all'elemento successivo
- **Indirizzamento con autodecremento**: il contenuto di un registro specificato nell'istruzione viene decrementato. Il nuovo contenuto viene usato come indirizzo effettivo dell'operando

6.5 Istruzioni

6.5.1 Istruzioni di inizializzazione

- `mov src, dst` **Move**: consente l'inizializzazione di un registro o di un'area di memoria. Accetta i modificatori `l`, `w` e `b` per indicare la dimensione dell'operando `s rc`
- `lea src, dst` **Load Effective Address**: trasferisce l'indirizzo effettivo dell'operando `src` nel registro `dst`

6.5.2 Istruzioni aritmetiche e logiche

- `sar op1, op2` **Shift Arithmetic Right**: esegue lo shift a destra sul registro `op2` di tanti bit quanti specificati in `op1`. Il bit più significativo viene replicato (così da funzionare anche con numeri negativi in complemento 2) e il bit scartato viene messo nel **Carry Flag**. `op1` può essere un registro o un valore immediato, `op2` deve essere un registro
- `sal op1, op2` **Shift Arithmetic Left**: esegue lo shift a sinistra sul registro `op2` di tanti bit quanti specificati in `op1`. Il bit meno significativo viene messo a 0 e il bit scartato viene messo nel **Carry Flag**. `op1` può essere un registro o un valore immediato, `op2` deve essere un registro
- `inc op` **Increment**: incrementa di 1 il valore memorizzato in `op`. `op` può essere un registro o una locazione di memoria
- `dec op` **Decrement**: decrementa di 1 il valore memorizzato in `op`. `op` può essere un registro o una locazione di memoria
- `add src, dst` **Add**: somma a `dst` il valore di `src` e memorizza il risultato in `dst`
- `sub src, dst` **Subtract**: sottrae da `dst` il valore di `src` e memorizza il risultato in `dst`
- `mul moltipl` **Unsigned Multiplication**: esegue la moltiplicazione senza segno. `moltipl` deve essere un registro o una variabile. Se `moltipl` è un byte il registro `AL` viene moltiplicato per l'operando e il risultato viene memorizzato in `AX`. Se `moltipl` è una word il contenuto del registro `AX` viene moltiplicato per l'operando e il risultato viene memorizzato nella coppia di registri `DX:AX` (`DX` conterrà i 16 bit più significativi del risultato). Se `moltipl` è un long il contenuto del registro `EAX` viene moltiplicato per l'operando e il risultato viene memorizzato nella coppia di registri `EDX:EAX` (`EDX` conterrà i 32 bit più significativi del risultato).

- `imul moltipl` moltiplicazione con segno
- `div divisore` **Unsigned Division**: esegue la divisione senza segno. `divisore` deve essere un registro o una variabile. Se `divisore` è un byte il registro `AX` viene diviso per l'operando, il quoziente viene memorizzato in `AL`, e il resto in `AH`. Se `divisore` è una word, il valore ottenuto concatenando il contenuto di `DX` e `AX` viene diviso per l'operando (i 16 bit più significativi del dividendo devono essere memorizzati nel registro `DX`), il quoziente viene memorizzato nel registro `AX` e il resto in `DX`. Se `divisore` è un long, il valore ottenuto concatenando il contenuto di `EDX` e `EAX` viene diviso per l'operando (i 32 bit più significativi del dividendo sono nel registro `EDX`), il quoziente viene memorizzato nel registro `EAX` e il resto in `EDX`
- `xor src, dst` **Logical Exclusive OR**: calcola l'OR esclusivo bit a bit dei due operandi e lo memorizza nell'operando `dst`. Spesso si utilizza per azzerare un registro, utilizzandolo sia come `src` che come `dst`)
- `or src, dst` **Logical OR**: calcola l'OR logico bit a bit dei due operandi e lo memorizza nell'operando `dst`
- `and src, dst` **Logical AND**: calcola l'AND bit a bit dei due operandi e lo memorizza nell'operando `dst`
- `not op` **Logical NOT**: inverte ogni singolo bit dell'operando op

6.6 AT&T vs Intel

Le principali differenze tra la sintassi AT&T e Intel sono:

- In AT&T i nomi dei registri hanno il carattere `%` come prefisso
- In AT&T l'ordine degli operandi è `<sorgente>, <destinazione>`, opposto rispetto alla sintassi Intel
- In AT&T la lunghezza dell'operando è specificata tramite un suffisso al nome dell'istruzione. `b` per **byte** (8 bit), `w` per **word** (16 bit), `l` per **double word** (32 bit)
- Gli operandi immediati in AT&T sono preceduti dal simbolo `$`
- la presenza di prefisso in un operando indica che si tratta di un indirizzo di memoria. Ad esempio:

  `movl $pippo, %eax` è diverso da `movl pippo, %eax`

- l'indicizzazione o l'indirezione è ottenuta racchiudendo tra parentesi l'indirizzo di base espresso tramite un registro o un valore immediato. Ad esempio:

```
movl 5, 17(%ebp)
```

6.7 Compilazione

Prima di tutto bisogna creare un file con estensione `.s` e scrivere il codice assembly. Dopo aver scritto il codice assembly, si compila il file con il comando `as` che crea un file binario (`.o`) contenente l'implementazione di ogni singolo file. Infine si uniscono i file binari con il comando `ld` che crea un file eseguibile a partire dai file binari.

```
$  as -o <nomefile>.o <nomefile>.s
$  ld -o <nomefile> <nomefile>.o
$  ./<nomefile>
```