

# 1 Lezione del 24-02-25

## 1.1 Introduzione al corso

Continuiamo lo studio di una particolare architettura per calcolatori, a partire da quanto detto riguardo alle reti logiche, introducendo i concetti di **interruzione**, **protezione** e **memoria virtuale**. Questi 3 strumenti ci permetteranno di realizzare il paradigma della **multiprogrammazione**, cioè di far eseguire ad una macchina con un singolo processore più programmi contemporaneamente. Non si pensi questo significhi avere più processori, in quanto il corso riguarda esclusivamente processori *single-threading*.

## 1.2 Architettura

L'architettura di riferimento è quella classica, composta da **CPU**, **memoria** e **I/O** interconnessi da un **bus**.

Durante lo studio di un'architettura è opportuno porsi la domanda "*chi fa cosa?*", che fornisce determinati *chi* ai determinati *cosa* forniti da un opportuno livello di astrazione (transistor, porte logiche, diagrammi funzionali, ecc...).

La domanda che potremo porci adesso è "*chi comanda?*" all'interno dell'architettura vista. La risposta più giusta è quella del **software**: l'architettura è fatta per *eseguire* software.

Per convincerci di questo possiamo sostituire la domanda "*chi fa cosa?*" con la domanda "*chi sa cosa?*".

- La **CPU** conosce lo stato corrente dei registri e l'istruzione in esecuzione. Fra un'istruzione e l'altra non c'è alcun bisogno di sapere cosa è accaduto finora, e cosa accadrà in futuro, ma solamente l'istruzione corrente. Quindi si può pensare che la CPU non *sa* qual'è l'obiettivo della computazione, ma si limita a portarla avanti.
- La **memoria** è un oggetto passivo, che contiene il programma, ma si limita a restituire i dati richiesti quando sono richiesti. Notiamo che le memorie che usiamo sono ad **accesso casuale**, ergo nessuno scorre alla ricerca di indirizzi, ma si può leggere e scrivere in posizioni arbitrarie in tempo pressoché costante. La memoria contiene **sempre** qualcosa, che questo sia significativo o meno, e la sua tipizzazione dipende solamente dalle intenzioni del programmatore.
- L'**I/O** è il componente più variegato dell'architettura. L'unica costante che rende la comunicazione con le periferiche più facile è la presenza di un interfaccia, che riduce tale comunicazione ad una semplice lettura o scrittura nello spazio di I/O. La differenza fra le letture e scritture nello spazio di I/O e lo spazio di memoria è la possibile presenza di **effetti collaterali**, cioè effetti non riconducibili alla sola variazione di stato di una locazione di memoria. Inoltre la CPU non è l'unica a scrivere nello spazio di I/O, in quanto questo può essere fatto anche dalle periferiche stesse.
- Il **bus** è un insieme di linee (*fili*), che trasportano ciò che ogni componente sta comunicando in un dato momento. Ogni componente vede ciò che viene scritto sul bus in qualsiasi momento, e l'indirizzamento di locazioni specifiche nello spazio di memoria o nello spazio di I/O viene fatto attraverso **maschere** di indirizzo.

### 1.2.1 Flusso di controllo

Abbiamo visto come la CPU si limita a prelevare ed eseguire istruzioni nel ciclo di **fetch-execute**. L'istruzione successiva alla corrente, il cui indirizzo viene scritto nell'**instruction pointer**, viene decisa dall'istruzione corrente stessa (si pensi alle istruzioni di salto). Il **flusso di controllo** è quindi deciso dall'istruzioni stesse, cioè dal programma.

### 1.2.2 Bootstrap

Il **bootstrap** è un processo secondo il quale si porta il sistema in un certo stato di esecuzione, apparentemente impossibile, o comunque molto difficile, da raggiungere. Ad esempio, il compilatore del linguaggio C è scritto esso stesso in linguaggio C. La domanda naturale è "*come è stato compilato il compilatore?*". La risposta è un processo di bootstrap, usando o un compilatore presistente, magari che implementa un sottoinsieme parziale del C, o scrivendo l'intero compilatore in linguaggio macchina, cioè assemblando codice assembler.

Il bootstrap si rende necessario anche all'avvio del calcolatore, per il caricamento del programma all'interno della memoria e l'inizio dell'esecuzione. Nei calcolatori moderni questo viene fatto attraverso la **ROM**, cioè una memoria a sola lettura che contiene un programma di bootstrap. All'avvio il processore è impostato in modo che al reset prenda come indirizzo proprio quello della ROM, e quindi inizi ad eseguire il programma di bootstrap. All'interno della ROM si trova, nei calcolatori moderni, il **BIOS** (o **UEFI**, nei sistemi moderni), che ha il solo compito di impostare alcune periferiche di base e caricare il sistema operativo.

Iniziamo quindi ad approfondire, uno per uno, i moduli dell'architettura.

## 1.3 Memoria

La memoria è un insieme contiguo di locazioni di memoria, che nelle architetture moderne sono rappresentate da byte. Storicamente, la memoria era indirizzata a *parole*, cioè insiemi di bit coincidenti in dimensioni coi registri del processore. Una parola poteva essere di più byte, mentre oggi le memorie sono accessibili ai singoli byte. Ad esempio, le memorie usate nell'architettura Intel x86 sono accessibili ad 1 byte (**MOVB**), 2 byte (**MOVW**), 4 byte (**MOVL**), e 8 byte (**MOVQ**).

### 1.3.1 Endianess

Notiamo che la posizione in memoria del byte più significativo di una parola (in questo caso consideriamo una "parola" da 8 byte, da cui si ricavano tutte le altre misure) determina l'*endianess* dell'architettura. In particolare, se l'ultimo byte sta in fondo nella memoria, si dice **big-endian**, mentre se viceversa l'ultimo byte viene per primo nella memoria, si dice **little-endian**.

L'architettura Intel x86 che andiamo a considerare è little-endian, come lo sono la maggior parte delle architetture moderne. Un esempio di utilizzo del big-endian è nella trasmissione di dati attraverso il protocollo IP, usato nelle comunicazioni Internet.

### 1.3.2 Allineamento

Indicheremo con **offset** la distanza in byte fra due locazioni di memoria, inteso come il numero di locazioni che vanno saltate per raggiungere un indirizzo a partire dall'altro. In questo ha senso parlare anche di offset *negativi*.

Visto che lo spazio di memoria è effettivamente ciclico, cioè si ha *wrap-around* ai suoi capi, si ha che gli offset rimangono validi **modulo** la dimensione dello spazio di memoria, che è sempre  $2^n$ , con  $n$  nel nostro caso uguale a 64.

Il *wrap-around* si comporta bene con gli offset, ma lo stesso non si può dire per quanto riguarda **intervalli** di byte. Preso un certo intervallo  $[x, y)$ , quindi, si ha che questo contiene gli indirizzi  $\{n \mid x \leq n < y\}$ , ammesso che  $x < y$ , cosa che risulta falsa nel caso di intervalli che hanno *wrap-around*. Decidiamo di non considerare intervalli di questo tipo. Questo rende necessaria un'eccezione per intervalli che comprendono l'ultimo byte: in questo caso è concesso  $[x, 0)$ , con 0 che indica il fondo dello spazio di memoria.

Veniamo quindi all'**allineamento**. Dire che un indirizzo è allineato ad un numero  $n$  significa dire che quell'indirizzo è un multiplo di  $n$ . Chiaramente, conviene scegliere  $n$  potenze di 2. In questo caso, per riconoscere se un indirizzo è allineato a  $2^k$ , basta guardare i suoi primi  $k$  bit.

Si dice spesso che oggetti sono *allineati alla parola*, ecc... Questo significa che sono allineati alla *dimensione* della parola specificata. Altrimenti, si può dire che un oggetto è allineato *naturalmente*, nel caso in cui sia allineato alla dimensione di stesso.

Infine, il **confine** di un oggetto è l'indirizzo che lo delimita dal resto dello spazio di memoria.