1 Lezione del 25-09-25

1.1 Richiami architetturali

Riprendiamo alcuni aspetti architetturali di un sistema di elaborazione. L'architettura che consideriamo è quella di *Von Neumann*, modello ancora oggi in uso e composto da:

- La **CPU** (*Central Processing Unit*) o come abbiamo già detto *processore*. Rappresenta un circuito piuttosto complesso che ha però l'unica funzione di *esecutore di istruzioni*.

  Le istruzioni che questa esegue possono essere di tipo **CISC** (*Complex Instruction Set*), come ad esempio nell'architettura x86, o di tipo **RISC** (*Reduced Instruction Set*), come ad esempio nell'architettura ARM. Ricordiamo comunque che nelle moderne implementazioni dell'x86 si traduce comunque in un instruction set RISC a livello architetturale per questioni di ottimizzazione.

  Si può infatti dire che è inutile avere molte e complesse istruzioni (CISC) che richiedono molti cicli di clock, quandi si possono avere poche e semplici istruzioni (RISC) che ne richiedono pochi: eventuali istruzioni più complesse potranno essere implementate come *subroutine* che usano più istruzioni semplici.

  Ricordiamo quindi che la CPU si limita ad eseguire istruzioni, e non conosce (non memorizza) il programma. La poca memoria che ha a disposizione (sotto forma di *registri*) viene usata per mantenere i dati che sta elaborando;

- La **RAM** o *memoria centrale*, o ancora come abbiamo visto *memoria principale*. Questa ha il compito di memorizzare *dati* e *programma* (questo il fulcro dell'architettura di Von Neumann) e di renderli disponibili alla CPU e, come vedremo, anche ad altri dispositivi.

  Abbiamo visto che è una memoria *volatile*, quindi che si mantiene solo finché il calcolatore è acceso, e che è una memoria ad *accesso diretto*, cioè si può accedere a qualsiasi locazione in tempo costante (a differenza di memorie di tipo *sequenziale*, ecc...).

  Le operazioni che possiamo svolgere sulla memoria sono *letture* e *scritture* su locazioni di memoria. Nelle memorie moderne le letture sono *non distruttive*, mentre le scritture (chiaramente) lo sono.

- Qualche tipo di complesso di **I/O**. Questo comprende periferiche come *tastiera*, *porte seriali/parallele*, *interfacce di rete*, ecc...

  Un dispositivo particolare che si trova nello spazio di I/O è il **disco** o *memoria secondaria*, a differenza della principale *persistente*, e usata per l'archiviazione di dati a lungo termine. Chiaramente, il tradeoff in questo caso è in termini di tempo (i dischi, anche allo stato solido, sono molto più lenti in tempo di accesso della RAM).

- Un **bus**, o *rete di interconnessione*, che permette a questi componenti di comunicare fra di loro.

  Questa comunicazione dovrà essere **bidirezionale**, in quanto ad esempio la CPU deve sia leggere che scrivere dalla RAM: abbiamo visto come bus di questo tipo possono essere implementati sfruttando la logica a 3 stati.

Sperabilmente un bus dovrà contenere un numero consistente di linee. Torniamo all'esempio della CPU che legge in memoria: avremo bisogno di specificare l'*indirizzo* della locazione che vogliamo leggere, e vorremo vederci tornare una o più *parole* (cioè i dati che ci interessano) dalla memoria. Il modo più veloce per effettuare questa operazione è fornirsi di abbastanza linee per specificare sia gli indirizzi che i dati in **parallelo**: un bus *seriale* si dimostrerebbe molto più lento.

A livello logico dobbiamo dire anche che c'è bisogno di un **protocollo**, o comunque una qualche *politica* di gestione del bus.

- Ad esempio, la politica più semplice è quella dove la CPU è l'unica che può iniziare una transazione sul bus: questa è la classica configurazione *master-slave* dove la CPU rappresenta il *master* e memoria e I/O rappresentano gli *slave*;
- Esistono però situazioni dove potremmo volere che i dispositivi (ad esempio il disco) scrivano in memoria, o viceversa sia la memoria a scrivere sui dispositivi. Questo è effettivamente il caso del *DMA*. Avere un bus che supporta più iniziatori di transazioni richiede necessariamente un protocollo che stabilisca chiaramente chi può iniziare in quale momento una data transazione.

Le transazioni avvengono chiaramente in fasi, di cui ne individuiamo almeno 3 nel caso più semplice (singolo master, più slave):

1. Una prima fase di richiesta della transazione da parte dell'*iniziatore*;
2. Una fase di attesa da parte dell'iniziatore del responso dell'*obiettivo*;
3. Una fase dove l'operazione viene effettivamente eseguita, in un determinato lasso di tempo.

Nel caso di più master, abbiamo bisogno di meccanismi più sofisticati che implementino **mutua esclusione** e **sincronizzazione** delle risorse a cui i più iniziatori potrebbero voler accedere. Questo è vero sia a livello *logico* (su risorse logiche o comunque gestite dal S/O) che *elettrico* (2 o più componenti non pilotino mai le stesse linee contemporaneamente, pena fili bruciati).

Facciamo quindi una considerazione su come organizzare lo spazio di memoria e lo spazio dedicato ai registri delle periferiche. Esistono due configurazioni principali:

- **Memory-mapped I/O**: disponiamo i registri di I/O direttamente nello spazio di memoria, usando gli stessi indirizzi per indirizzare sia la memoria che i dispositivi;
- **Port-mapped I/O**: sfruttiamo due spazi, lo *spazio di memoria* e lo *spazio di I/O*, che mantengono separati i due tipi di informazione. Questo può essere fatto agilmente includendo un bit di selezione di spazio nel bus, ed è la soluzione adottata dall'architettura x86.

1.2 CPU

Vediamo nel dettaglio il primo componente, cioè la CPU.

1.2.1 Cicli CPU

Il funzionamento della CPU avviene in maniera **ciclica**: cogliamo più fasi che si ripetono nel tempo da quando questa viene accesa (reset) fino a quando viene spenta.

1. **Prelievo** o *fetch*: si legge la prossima istruzione in memoria, puntata dall'**IP** o **PC** (*Instruction Pointer* o *Program Counter*), e la si porta in un qualche registro interno al processore, pronta ad essere eseguita;
2. **Decodifica** o *decode*: si interpreta il significato dell'istruzione, cioè si individua qual'è effettivamente l'istruzione che dobbiamo eseguire, e si portano all'interno di registri gli eventuali *operandi sorgente* o gli indirizzi degli *operandi destinazione*;
3. **Esecuzione** o *execute*: si esegue effettivamente l'istruzione, direttamente attraverso la rete di controllo della CPU o sfruttando una o più **ALU** (*Arithmetic and Logic Unit*).

   Successivamente, il risultato viene (se necessario) riscritto in memoria attraverso un'operazione di *write-back*. Questa fase viene a volte considerata come a sé stante (ad esempio nelle pipeline delle architetture RISC).

1.2.2 Registri CPU

La CPU è dotata di una sua memoria interna formata da locazioni di memoria dette **registri**. Questi si dividono in registri **generali**, riservati alle elaborazioni, e **di stato**, riservati a compiti speciali.

Registri generali

Consideriamo un set estremamente generico di registri:

- **AX**, **BX**, **CX** e **DX** sono i classici registri programmatore a uso generale;
- **ESP** è utilizzato per indirizzare la **pila** o **stack**, ovvero una parte di memoria con disciplina LIFO che serve a gestire sottoprogrammi.

Registri di stato

Ricordiamo due registri di stato:

- L'**IP** o **PC** (*instruction pointer* o *program counter*). Viene usato per contenere l'indirizzo della locazione dalla quale sarà prelevata la prossima istruzione da eseguire. Il contenuto dell'EIP è fissato al reset iniziale, e impostato sulla prima istruzione da eseguire.
- L'**F** (registro dei *flag*). Consiste di una serie di elementi binari detti **flag**, fra cui ricordiamo:
  - **OF:** flag di overflow (traboccamento) delle operazioni aritmetiche, si imposta se l'ultima operazioni, presi gli operandi come interi, ha prodotto un risultato non rappresentabile su n bit;
  - **SF:** flag di segno, impostato quando l'ultima operazione restituisce un complemento a 2 con MSB $= 1$ (ergo negativo);
  - **ZF:** flag zero, che viene impostato quando l'ultima operazione restituisce qualcosa di nullo;

- **CF:** flag di carry (riporto), che viene impostato quando l'ultima operazione richiede un riporto o un prestito, ergo presi gli operandi come naturali il risultato non è rappresentabile su n bit;
- **IF:** flag di interruzioni attivate, quando è attivo il processore risponde alle interruzioni (che approfondiremo in seguito).

Al reset i flag visti finora sono impostati a 0.

1.2.3 Instruction set

Consideriamo un set di istruzioni estremamente basilare. Innanzitutto, possiamo dividere le istruzioni in **operative** e **di controllo**. Possiamo quindi fare ulteriori suddivisioni al'interno di queste categorie:

- **Operative:**
  - Di trasferimento;
  - Aritmetiche;
  - Di traslazione/rotazione:
  - Logiche.
- **Di controllo:**
  - Di salto;
  - Di gestione di sottoprogrammi.

discorso CISC

Per informazioni più approfondite sulla struttura generale del processore considerato si rimanda ai testi specializzati o agli appunti in `https://raw.githubusercontent.com/seggiani-luca/appunti-rl/34228f66db395637bd1824d04f3130b977cc0ce4/master/master.pdf`.

1.3 RAM

Approfondiamo quindi il discorso della RAM. Nel sistema considerato inseriremo un elemento di **cache**, nello specifico fra la CPU e il bus (da cui si accede alla RAM). Il funzionamento della cache è dettagliato in `https://raw.githubusercontent.com/seggiani-luca/appunti-ce/638d3abf2e1d473632b575401582203c3b113c82/master/master.pdf`, e per quanto ci riguarda possiamo dire che funge da unità di "*memoizzazion*" dei dati (quando vengono richiesti), più veloce della RAM.