Appunti Calcolatori Elettronici

Luca Seggiani
2025

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 permetterano 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 archiettura è oppurtuno 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 restiture 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 boostrap, usando o un compilatore presesistente, magari che implementa un sottoinsieme parziale del C, o scrivendo l'intero compilatore in linguaggio macchina, cioè assemblando codice assembler.

Il boostrap 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 boostrap. 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 e 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, cioò 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.

2 Lezione del 25-02-25

2.1 Interazione fra CPU e memoria

Nell'architettura Intel x86 la CPU interroga la RAM in due situazioni:

- Durante la lettura di un istruzione;
- Durante la lettura di *eventuali* operandi in memoria richiesti dall'istruzione. Notiamo che per ogni istruzione è previsto un solo indirizzo esplicito di un operando in memoria (non è permesso scrivere qualcosa come `MOV (%RBP), (%RDI)`). indirizzo Alcune istruzioni possono però avere comunque più di un operando in memoria (ad esempio le istruzioni di stringa, `MOVS`, ecc... o la stessa istruzione di pila `POP`).

Dal punto di vista pratico, il collegamento fra CPU e RAM è rappresentato da:

- Un **bus dati** a 64 bit;
- Un certo numero di linee per il **numero di riga**. Questo non corrisponde all'indirizzo del primo byte contenuto in ogni riga, ma l'indice proprio di ogni regione (intesa come riga) da 64 bit all'interno della RAM. Si noti inoltre che queste non sono necessariamente 2^{64}, o 2^{57} (il massimo spazio indirizzabile secondo l'architettura x86), ma più spesso intorno alle 2^{36}-2^{37};
- Determinate **linee di controllo** che segnalano l'operazione in corso da parte del processore.
- 8 linee di **byte enable**, attive basse, che rappresentano i byte di interesse all'interno di ogni locazione da 64 bit della RAM. Dal punto di vista della lettura, queste linee non sono particolarmente utili in quanto tutta la locazione verrà comunque riportata sul bus dati, o comunque le locazioni non selezionate potranno essere invalide o in alta impedenza, senza avere effetto sulla CPU (che non le leggerà). Per quanto riguarda la scrittura, invece, la RAM lascerà inalterati i byte con byte enable alto.

2.1.1 Struttura della RAM

Modelliziamo un modulo di RAM come una rete provvista di:

- Una linea di **select**, attiva bassa;
- Le **linee di indirizzo**;
- Una linea di *memory read* e una linea di *memory write*, o comunque un certo numero di **linee di controllo** necessarie all'accesso in scrittura e lettura;
- Un **bus dati** di ingresso/uscita.

Dalla CPU arriveranno, come abbiamo detto, i **numeri di riga**, i **byte enable**, il **bus dati** e le **linee di controllo**.

I numeri di riga si collegano direttamente alle linee di indirizzo di ogni modulo, che rappresenterà un certo byte della locazione (avremo quindi, nell'architettura descritta, 8 moduli per 8 byte, quindi 64 bit). I byte enable dovranno quindi smistarsi nelle linee di select di ogni modulo di RAM, a selezionare il modulo corrispondente. Il bus dati verrà composto, analogamente, concatenando le linee di uscita da 8 bit di ogni modulo di RAM. Notiamo che avevamo chiamato questo montaggio **parallelo**.

Vorremo poter estendere la memoria disponibile oltre il numero di locazioni da un byte fornite da ogni modulo di RAM. Pensiamo di fare questo attraverso più banchi di memoria con locazioni da 64 bit. In questo caso avremo bisogno di montaggio in **serie**, e quindi di generare un segnale di select a partire non solo dalle line di byte enable, ma anche da una **maschera** generata a partire dal numero di riga. Questo si potrà fare agevolemnte mettendo il segnale di uscita della maschera in OR (ricordiamo segnali attivi bassi, quindi si applica De Morgan) con il byte enable di ogni modulo di RAM compreso nel banco di memoria associato a tale maschera.

2.1.2 Allineamento e RAM

Quanto discusso finora rende più chiaro l'importanza del corretto allineamento degli oggetti in memoria. Leggere un oggetto da 8 byte non allineato nel montaggio di RAM descritto, infatti, richiederà necessariamente 2 accessi, contro il singolo accesso necessario per un oggetto allineato. Inoltre, alcuni dei byte più significativi risulteranno invertiti di posto rispetto ai byte meno significativi, cioè si richiede un operazione di shift interna al processore.

Questa combinazione di operazioni, eseguite in **hardware**, rende gli accessi in memoria non allineati molto poco performanti, e quindi sconsigliati (anche se l'architettura Intel x86 li permette comunque).

Un problema che potrebbe interessarci è, data una regione di memoria $[x, y)]$ di dimensione b uguale a un singolo banco di RAM, ottenere gli indici della prima regione in cui cade l'intervallo, e la prima in cui non cade più.

Vediamo come calcolare la prima regione di appartenenza. In **hardware**, questo può essere calcolato semplicemente prendendo gli $n - b$ bit più significativi dei numeri di riga x e y.

In **software**, questo equivarrà ad uno shift a destra che conservi i soli $n - b$ bit più significativi.

Vediamo come calcolare l'offset di x o y all'interno delle rispettive regioni. Mascherando gli stessi bit, invece, si può ottenere l'indirizzo all'interno del banco del confine

della regione. Per la precisione, vogliamo una maschera fatta da $n - b$ 0 e b 1. Questa si può ricavare agevolmente prendendo 2^b come `1UL << b` e sottrandogli 1, ottenendo la maschera desiderata (si avranno borrow propagati dal bit in b fino al LSB).

Infine, vediamo come calcolare la prima regione di non appartenenza. In questo caso potremo calcolare la regione in cui cade $y-1$, e aggiungervi 1 (tenendo conto di eventuali *wrap-around*). Il -1 è richiesto dal fatto che y potrebbe cadere sul confine. In questo caso avremo `((y - 1) >> b) + 1`, considerata somma modulo $n - b$. Alternativamente, si può prendere $y + b$ e calcolarne la regione di appartenenza.

2.2 Spazio di I/O

Veniamo quindi alla trattazione dello spazio di I/O e delle interfacce ivi connesse. L'accesso alle periferiche viene fatto attraverso le istruzioni `IN` e `OUT`, ammesso che non ci sia nessun sistema operativo in esecuzione, ma solo il nostro programma, e appositi sottoprogrammi di ingresso/uscita, la cui struttura non è al momento importante.

Le periferiche che studieremo, per semplicità di trattazione, derivano in parte da quelle disponibili sui PC **IBM AT** (famiglia *IBM 5170*). I PC di questa categoria (compresi tutti i vari *IBM compatible*) si basavano sullo standard per periferiche **ISA** (*Industry Standard Architecture*). Visto che i PC moderni derivano dai vecchi IBM compatible, anche oggi si cerca di emulare (almeno in parte) questo standard.

Le periferiche, nello specifico saranno:

- La **tastiera**;
- Il **video** su VGA;
- Il **timer**;
- Gli **hard disk**.

2.2.1 Tastiera

Dal punto di vista funzionale, la tastiera deve solo scoprire quali tasti sono premuti e comunicarlo al calcolatore. In particolare, noi studieremo tastiere IBM che trasmettono secondo lo standard PS/2.

Nei PC IBM il tasto non restituisce il carattere ASCII del carattere premuto, ma un codice associato ad ogni tasto che va convertito in software. Questo codice viene ottenuto per *scansione* dell'intero piano della tastiera. Dal punto di vista meccanico, ci sono **tracce** orizzontali e verticali disposte, rispettivamente, su ogni riga o colonna di tasti. La pressione di un tasto comporta una deformazione delle tracce che chiude un circuito fra la riga e la colonna del tasto corrispondente. Un **microcontrollore** (originariamente un Intel 8042) collegato sia alle tracce orizzontali che alle tracce verticali scansiona ciclicamente, con impulsi, o le righe leggendo le colonne, o le colonne leggendo rige, cercando un circuito chiuso. Un cortocircuito viene quindi rilevato dal microcontrollore, che aggiorna una (piccola) memoria interna con il tasto premuto. Di conseguenza, invia al calcolatore un segnale che codifica quali tasti sono stati premuti rispetto al precedente istante temporale, e quali tasti sono stati rilasciati rispetto al precedente istante temporale.

La tastiera non restituisce solo pressioni di tasti, ma anche i loro rilasci, cosa che può essere utile per ottenere combinazioni di tasti, pressioni estese nel tempo, ecc... I codici di pressione si dicono **make code**, mentre i codici di rilascio si dicono **break code** La

stessa pressione ripetuta di un tasto quando l'utente lo tiene premuto per un certo istante temporale era, nei PC IBM, realizzata direttamente nella tastiera (tecnologia *type-matic*), tra l'altro con periodo configurabile. Tramite il *type-matic*, su appositi tasti abilitati, si ha infatti una ripetizione dell'evento di *pressione* (non rilascio) di un tasto a frequenza costante dopo un intervallo di pressione continua.

Lato calcolatore, il segnale prodotto dal microcontrollore della tastiera viene letto da un interfaccia provvista dei seguenti registri:

- **RBR**, *Receive Buffer Register*;
- **TBR**, *Transmit Buffer Register*;
- **STR**, *Status Register*;
- **CMR**, *Command register*

RBR e TBR, come STR e CMR, condividono gli indirizzi, rispettivamente `0x60` e `0x64`. Il RBR conterrà i make e break code, mentre l'STR conterrà i flag di stato sia per RBR che per TBR (rispettivamente ai bit 0 e 1).

Potremmo chiederci il significato di un registro di trasmissione TBR. Questo serve, ad esempio, a governare i led di stato per funzioni speciali quali Caps-Lock, Num-Lock, Scroll-Lock ecc... nonché a modificare le impostazioni del type-matic e, in maniera completamente slegata alla tastiera, a provocare il reset del PC, scrivendo `0xFE` in CMR.

Vediamo quindi del codice C++ per l'interazione con l'interfaccia di tastiera:

```
const ioaddr iSTR = 0x64; // indirizzo di STR e CMR
const ioaddr iRBR = 0x60; // indirizzo di RBR e TBR

natb get_code() {
  natb c;
  do {
    c = inputb(iSTR); // inputb e' una funzione che sfrutta la IN dell'asm
  }
  while (!(c & 0x01)); // controlla il LSB di STR
  return inputb(iRBR);
}
```