1 Lezione del 26-11-24

1.1 Dall'assembler al linguaggio macchina

Abbiamo visto come un processore si programma attraverso il linguaggio **assembler**, che è più distante dai dettagli di implementazione e più vicino al linguaggio umano, quindi al programmatore. Vediamo quindi il processo di conversione dall'assembler al linguaggio macchina "*assemblaggio*".

1.1.1 Fetch degli operandi

Il problema principale del processore è indiduare gli operandi in base al tipo di indirizzamento dell'istruzione. Ad esempio:

- `MOV AH, AL` contiene operandi che si trovano già nei registri;
- `MOV $0x10, AL` contiene operandi che vanno letti in memoria assieme all'istruzione stessa;
- `MOV 0x00FF, AL` contiene operandi che vanno letti in memoria, in locazioni diverse da quella dell'istruzione stessa.

Durante la fase di fetch, quindi, il processore deve procurarsi gli operandi necessari ad eseguire la prossima istruzione, che questi siano nei registri o in memoria. Questa operazione sarà comune a ogni istruzione, cioè le modalità di indirizzamento degli operandi saranno identiche che si parli di una `ADD` come di una `MOV`, ecc... Dal punto di vista pratico, questo significa che la parte di **fetch** potrà essere messa *in comune*, mentre la fase di **esecuzione** sarà *specifica* ad ogni istruzione.

1.1.2 Formato delle istruzioni macchina

I primi 3 bit del codice operativo di un istruzione rappresentano il formato dell'istruzione:

- **Formato F0** (000): sono istruzioni per le quali il processore non deve compiere nessuna azione, in quanto accade alternativamente che:
  - Gli operandi sono registri;
  - Le istruzioni non hanno operandi.

  In questo caso il processore deve solo raccogliere l'indirizzo della prossima istruzione dall'instruction pointer ??? ;
- **Formato F2** (010): l'operando **sorgente** si trova in memoria, indirizzato attraverso DP. In questi casi tutta l'istruzione sta su un byte, ma l'operando sorgente va prelevato tramite un'ulteriore lettura in memoria di un byte;
- **Formato F3** (011): l'operando **destinatario** si trova in memoria, indirizzato attraverso DP. Notiamo che in questo caso l'operando destinatario non va *prelevato* dal processore, ma deve essere *sovrascritto*, cosa che accade durante la fase di esecuzione. Non ci sono quindi letture ulteriori in fase di fetch;

- **Formato F4** (100): l'operando **sorgente** è indirizzato in modo **immediato**, e sta su 8 bit. In questo caso l'istruzione è lunga 2 byte, e nella fase di fetch si dovranno leggere due byte in memoria consecutivi puntati dal registro IP (e non DP come nel caso precedente);
- **Formato F5** (101): l'operando **sorgente** si trova in memoria con indirizzamento **diretto**. Visto che lo spazio di memoria è a 24 bit, un istruzione sarà lunga 4 byte: 1 di OPCODE e 3 di indirizzo. In fase di fetch si dovrà quindi:
  - Leggere i 4 byte consecutivi dell'istruzione puntati dal registro IP per ottenere istruzione e indirizzo dell'operando sorgente;
  - Leggere l'operando sorgente stesso, in un'altra locazione di memoria.
- **Formato F6** (110): l'operando **destinatario** si trova in memoria con indirizzamento **diretto**. Di nuovo, il processore dovrà leggere 4 byte consecutivi puntati dall'IP, ma a questo punto non ci saranno *letture* in memoria, bensì *scritture* sull'indirizzo prelevato del destinatario, in fase di **esecuzione**;
- **Formato F7** (111): raggruppa le istruzioni di controllo seguite da indirizzo (quindi, ad esempio, non la `RET`, ma le varie `CALL`, `JMP`, salti condizionali ecc..). In questo caso, l'indirizzo viene specificato come prima su 3 byte, e bisogna nuovamente leggere 4 byte consecutivi puntati dall'indirizzo in IP.
- **Formato F1** (001): raggruppa le istruzioni non classificabili nei precedenti formati, fra cui:
  - Istruzioni I/O, con indirizzo a 16 bit sia sorgente (`IN`) che destinatario (`OUT`);
  - `MOV` con uno dei registri a 24 bit (DP o SP), sia sorgente che destinatario.

  Le operazioni in formato F1 si limitano in fase di fetch a prelevare l'OPCODE (fetch *scarna*). Gli operandi vengono gestiti successivamente in fase di esecuzione. Questa soluzione è sì poco elegante, ma risulta più agevole dal punto di vista dell'implementazione.

1.2 Architettura hardware del calcolatore

Vediamo quindi come viene implementato dal punto di vista fisico il calcolatore. Innanzitutto consideriamo un **bus**, formato da:

- **Linee di indirizzo**, nel nostro esempio 24 per 24 bit, di uscita per il processore e ingresso per gli spazi di memoria e I/O, notando che nel salto dal bus allo spazio di I/O si perdono gli 8 bit più significativi (si passa da 24 a 16). Non servono "forchette" con porte tri-state in quanto il processore è sempre l'unico a scrivere sulle linee;
- **Linee dati**, nel nostro esempio 8 per 8 bit, usate per leggere e scrivere byte di memoria. In questo caso il processore potrebbe leggere (dalla memoria o dallo spazio di I/O) o scrivere (sempre nella memoria o nello spazio di I/O), ergo potrebbero esserci conflitti di pilotaggio dei dati. Si adottano quindi le porte tri-state, disposte come abbiamo visto a forchetta.
- **Linee di controllo**, tutte attive basse, che sono nel nostro caso:

  - `/mr` e `mw`, cioe memory read e memory write per la memoria;
  - `ior` e `iow`, cioè I/O read e I/O write per lo spazio di I/O

- **Clock** e **reset**.

Notiamo come nel bus non figura la linea di selezione `/s` per gli spazi di memoria, in quanto questo viene generato attraverso una maschera dalla memoria stessa sulla base degli indirizzo di lettura, cioè avrà il solo scopo di selezionare diversi banchi di memoria, a *livello* di memoria.

1.2.1 Spazio di memoria

Abbiamo quindi che lo spazio di memoria è implementato, su 16 MB, in parte con tecnologia RAM, in parte con EPROM (che contiene il bootstrap), e in parte con memoria video dedicata. Diciamo di avere 64 KB di memoria EPROM e 64 KB di memoria video.

Possiamo combinare queste memorie attraverso, come abbiamo visto prima, linee di select generate attraverso opportune maschere. In particolare, finisci quando ci arrivi vai

1.2.2 Spazio di I/O

Lo spazio di I/O è realizzato fisicamente attraverso **interfacce**, che sono elementi di raccordo tra il bus e i dispositivi I/O. Dal lato dispositivo, queste sono implementate in una maniera che "risponde" al dispositivo. Dal lato bus, invece, sono tutte uguali, cioè presentano le entrate di selezione, I/O read e I/O write, eventuali **indirizzi interni**, che servono a discriminare più **porte**, e le linee di entrata/uscita di un byte di dati, cioè, tranne che per gli indirizzi interni, le stesse linee fornite da una RAM.

Notiamo che in un interfaccia una porta può operare o in **sola lettura**, o in **sola scrittura**. Ad esempio, non potremo scrivere nell'interfaccia di una tastiera, e non potremo leggere nell'interfaccia di un monitor.

Potremmo chiederci come mai implementare interfacce per ogni dispositivo, e non connetterli direttamente al bus. Ci sono principalmente due ragioni:

- Diversi dispositivi hanno **diverse velocità**: spesso di molti ordini di grandezza, e comunque molto più lente del processore;
- Diversi dispositivi hanno **diverse modalità di trasferimento**: a volta un bit alla volta (**seriale**), a volte in gruppi di bit (**parallelo**).

Implementare interfaccie ci permette quindi di **standardizzare** l'input e l'output del calcolatore, rendendo le temporizzazioni e le modaità di trasferimento **omogenee**.

1.2.3 Processore

Possiamo quindi individuare, nel processore, tutti i registri effettivamente necessari:

- Registri visibili al programmatore, cioè:
  - i soliti
- Registri di supporto a uscite, fra cui:
  - i soliti

Dove notiamo la particolarità dei registri che supportano le linee dati, che saranno innanzitutto *forchettati*, cioè fatti passare attraverso una porta tri-state controllata da un enabler generato dal registro DIR (per *direzione*). DIR sarà a 0 di default, cioè scollegherà il registro dati di uscita dal bus, e lo porremo a 1 solo nel caso in cui dovremmo scrivere, attraverso tale registro, sul bus.

- I registri STAR e MJR;
- Registri di OPCODE, SOURCE e DEST_ADDR necessari alla fase di fetch;
- Registri di appoggio per operazioni, che saranno APP3, APP2, APP1, APP0 e NUMLOC.

1.2.4 Ciclo di fetch-execute

Vediamo quindi i dettagli del ciclo fetch-execute.

- La fase iniziale è quella di **reset**, dove si inizializzano i registri:
  - F verrà inizializzato a 0;
  - IP otterrà il valore del primo indirizzo dello spazio di memoria dove si trova il bootstrapper;
  - STAR sarà inizializzato al primo stato;
  - DIR verrà inizializzato a 0;
  - /MR, /MW, /IOR e /IOW verranno inizializzati a 1 (attivi bassi);
- Poi si passa alla fase di **fetch**, dove si prelevano istruzioni e operandi. In ordine:
  - Il processore preleva un byte dalla memoria, all'indirizzo indicato in IP;
  - Incrementa IP, modulo 2^24;
  - Controlla che il byte prelevato corrisponda appunto all'OPCODE di una delle istruzioni che conosce. In caso contrario, si va in stato di blocco;
  - Carica il byte prelevato in OPCODE;
  - Controlla il formato dell'OPCODE in modo da definire le modalità di indirizzamento. A questo punto si ramifica, effettuando le letture necessarie in memoria come specificato qualche paragrafo fa.
- Dopo la fase di fetch, viene la fase **execute**, dove si eseguono effettivamente operazioni sugli operandi;
- Nel caso di un errore in fase di fetch, o dell'incontro dell'istruzione HLT in fase execute, si dovrà andare in **stato di blocco**, cioè il processore dovrà smettere di rispondere agli ingressi e mantenere ferme le sue uscite.