1 Lezione del 24-09-24

1.1 Introduzione

Il corso di reti logiche tratta di:

1. **Linguaggio assembler:** come scrivere programmi semplici, come avviene la compilazione in linguaggio macchina;
2. **Reti logiche:** reti combinatorie, reti combinatorie per l'aritmetica, reti sequenziali asincrone e sincronizzate;
3. **Microprogammazione:** reti sequenziali sincronizzate, come realizzare una rete logica da specifiche. "Micro" qui sta per *hardware*;
4. **Il calcolatore:** processore, interfacce comuni e convertitori.

1.1.1 Introduzione alle reti logiche

Si parla di reti *logiche* in quanto si guarda all'hardware da una prospettiva funzionale, indipendente dalla sua tecnologia. Ad esempio, una porta NOR sarà implementata con determinati circuiti, ma tutto ciò che interessa a questo corso è come si comporta logicamente: $y = 1 \Leftrightarrow A = B = 0$.

1.2 Programmazione assembly

Il nome corretto del linguaggio sarebbe Assembly, ma noi lo chiameremo Assembler per ragioni storiche. L'assembler è il linguaggio con cui si scrivono le istruzioni eseguite dal processore. Il processore implementa effettivamente un ciclo fetch-execute dove preleva la prossima istruzione macchina (in assembler) dalla memoria e la esegue.

1.2.1 Linguaggio macchina

Il linguaggio macchina (LM) è dato dal contenuto effettivo della memoria che contiene le istruzioni, ergo una sequenza di zero e uno. Il linguaggio assembler adotta una sintassi simbolica per il linguaggio macchina: ad esempio, `MOV %AX, %BX`.

Il processo di trasformazione dall'assembler all'LM si chiama **assemblaggio**, mentre il processo di traasformazione da un linguaggio ad alto livello all'assembler si chiama **compilazione**.

1.2.2 Generalità sull'assembler

Si dice che assembler è un linguaggio a basso livello. Mancano i costrutti a cui siamo abituati da i linguaggi di alto livello:

1. Non esistono costrutti di flow control (for, if-else, ecc...), tutto si fa con istruzioni di salto.
2. Non esistono tipi variabile: gli operandi sono stringhe di bit che si riferiscono a locazioni in memoria.

Inoltre, l'assembler è strettamente legato all'hardware, ed è specifico per ogni processore. Noi vedremo l'assembler dei processori della famiglia Intel x86, che non è uguale all'assembler dei processori Arm Cortex, ecc... Questo rende il codice in assembler mai portatile. Fatta questa precisazione, possiamo dire che i principi generali restano comunque validi fra famiglie di processori diverse.

Esiste ancora oggi una nicchia di utilizzo del linguaggio assembler: quello dello sviluppo di sistemi embedded. Inoltre, il linguaggio ha un importante significato didattico e culturale.

1.3 Schema a blocchi del calcolatore

illustrazione modello funzionale

Un calcolatore è formato, in linea generale, da una rete di interconnessione (bus) che collega fra di loro:

- Interfacce che comunicano con dispositivi;
- La memoria principale che contiene dati e programmi;
- Il processore, che esegue il ciclo fetch-execute. Possiamo aggiungere che ogni processore, oggi, contiene almeno due blocchi:
  - L'**ALU**, Arithmetic Logic Unit, che si occupa di calcoli aritmetici su numeri interi (interpretando le stringhe di bit come numeri naturali o interi in complemento a 2) e operazioni logiche;
  - L'**FPU**, Floating Point Unit, che si occupa dei numeri a virgola mobile.

1.4 Riassunto di rappresentazione dell'informazione

1.4.1 Numeri naturali

N bit rappresentano 2^N naturali sull'intervallo $[0, 2^N - 1]$, ovvero:

$$b_{N-1}, b_{N-2}, ..., b_1, b_0 \Leftrightarrow X = \sum_{i=0}^{N-1} b_i \cdot 2^i$$

Il bit più a sinistra è il Most Significant Bit (MSB) (nell'esempio b_{N-1}), quello più a destra il Least Significant Bit (LSD) (nell'esempio b_0). Le cifre in base due a partire da un numero in un'altra base si trovano con l'algoritmo div-mod.

1.4.2 Numeri interi in complemento a due

N bit rappresentano 2^N interi sull'intervallo $[-2^{N-1}, 2^{N-1} - 1]$, ovvero:

$$X = \begin{cases} x & x \geq 0 \\ 2^N + x & x < 0 \end{cases}$$

oppure, usando l'operatore modulo:

$$|x|_2 N$$

poco chiaro, copiaci fondamenti

La legge inversa, che mi permette di trovare l'intero x dalla sua rappresentazione X, è:

$$x = \begin{cases} X & X_{N-1} = 0 \\ -(\bar{X} + 1) & X_{N-1} = 1 \end{cases}$$

dove la barra rappresenta l'operazione complemento.

1.4.3 Notazione esadecimale

Scrivere lunghe stringhe binarie diventa velocemente complicato. Per questo si adotta una notazione esadecimale per stringhe di 4 bit ($[0, 15]$):

riporta tabella stringhe esadecimali

A questo punto, possiamo denotare qualsiasi stringa binaria come una lista di numeri esadecimali prefissi da `0x` (che serve ad indicare la rappresentazione esadecimale stessa), ad esempio `0xC1`.

1.5 Struttura del calcolatore

1.5.1 Spazio di memoria

La memoria del calcolatore, vista dal programmatore assembler, è uno spazio lineare di 2^{32} (su calcolatori a 32 bit) locazioni (celle) di memoria, dalla capacità di un btye ciascuna. Ogni cella è quindi identificata da un numero di 32 bit, detto **indirizzo**.

Lo spazio di memoria è in larga parte implementato attraverso Random Access Memory (RAM), ovvero memoria volatile. Solo una piccola parte dello spazio è implementata attraverso Read Only Memory (ROM), ovvero memoria permanente, che contiene le istruzioni da eseguire al reset.

1.5.2 Accesso allo spazio di memoria

Il processore può accedere (leggere/scrivere) a:

- Singole locazioni (byte) da 8 bit;
- Doppie locazioni (word) da 16 bit;
- Quadruple locazioni (double word) da 32 bit.

Per gli accessi 16/32 bit si usa l'indirizzo più piccolo delle 2/4 locazioni. Si ricorda che l'indirizzo più grande contiene i bit più significativi.

Gli indirizzi di memoria assembler sono solo simbolici, e vengono tradotti dall'assemblatore, e in parte runtime. Questo significa che non si può accedere a memoria appartenente al sistema operativo, o memoria fuori dai limiti fisici del sistema, ecc...

1.5.3 Spazio di Input/Output

Lo spazio di Input/Output è formato da 2^{16}, ovvero 64k, locazioni o **porte**. Ogni porta ha una capacità di un byte ed è indirizzata da un numero di 16 bit.

Il processore accede alle porte attraverso operazioni particolari di lettura o scrittura (in o out). Spesso le porte sono configurate per un solo tipo di operazione: sola lettura o sola scrittura.

Le locazioni di memoria sono solitamente identifiche fra di loro, le porte di I/O no. Indirizzi diversi significano dispositivi diversi, e si rende quindi necessario conoscere fisicamente gli indirizzi.

1.5.4 Processore

Il processore è dotato di una memoria interna formata da locazioni di memoria da 32 bit (**registri**). Questi si dividono in registri **generali**, riservati alle elaborazioni, e **di stato**, riservati a compiti speciali.

sii piu chiaro

I 16 bit bassi dei registri sono riferibili autonomamente (retro-compatibili). DI alcuni registri si possono riferire parti ad 8 bit.

1.5.5 Registri generali

Alcuni registri vengono utilizzati per particolari funzioni, per motivi storici.

- EAX (AX, AH od AL) è utilizzato da alcune istruzioni aritmetiche per contenere operandi e risultati. Viene detto **accumulatore**.
- ESI, EDI, EBX e EBP sono a volte utilizzati come registri puntatore, base (B) e indice (I).
  - ESI
  - EDI
  - EBX veniva usato come indirizzo di base per l'accesso in memoria. Viene solitamente detto **base**.
  - EBP guarda slide!
- ECX è utilizzato come contatore nei cicli. Vienedetto **contatore**.
- EDX è utilizzato come operando di operazioni aritmetiche. Viene detto **data**.
- ESP è utilizzato per indirizzare la **pila** o **stack**, ovvero una parte di memoria con disciplina LIFO che serve a gestire sottoprogrammi.

1.5.6 Registri di stato

L'EIP viene detto 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 (in memoria ROM).

Possiamo quindi dire che il ciclo fetch-loop si svolge come segue:

- Il processore preleva dalla memoria, all'indirizzo EIP, una nuova istruzione;
- Incrementa EIP del numero di byte dell'istruzione prelevata;
- Esegue l'istruzione e ripete.

Da questo si ha che le istruzioni in memoria vengono eseguite sequenzialmente nell'ordine in cui sono incontrate, a meno che non si definiscano salti attraverso altre determinate istruzioni.