1 Lezione del 25-09-24

1.1 Introduzione all'Assembler

1.1.1 Codifica macchina e codifica mnemonica

Possiamo adottare 2 metodi per codificare le istruzioni eseguite dal processore:

- **Codifica macchina:** la serie di zeri e di uni che codificano, nel linguaggio del processore, le operazioni che esegue. Il formato macchina è, nell'architettura che ci interessa, il seguente:

Segmento	Byte	Funzione
I Prefix (Instruction Prefix)	0/1 byte	Modifica l'istruzione.
O Prefix (Operand-size prefix)	0/1 byte	Modifica la dimensione degli operandi.
Opcode	1/2 byte	Specifica l'operazione.
Mode (ModR/M Byte)	0/1 byte	Specifica la modalità d'indirizzamento e i registri operandi.
SIB Byte	0/1 byte	Viene usato in congiunzione con il Mod/RM btye quando si usa l'indirizzamento complesso (scale-index-base).
Displacement	0/1/2/4 byte	Specifica un'offset in memoria, sempre nell'indirizzamento complesso.
Immediate	0/1/2/4 byte	Specifica le costanti ad indirizzamento immediato.

- **Codifica mnemonica:** un modo **simbolico** per riferirsi alle istruzioni. Un'istruzione può quindi essere semplicemente: `MOV \%EAX, 0x01F4E39`.

Il linguaggio assembler usa la codifica mnemonica delle istruzioni, e dispone di sovrastrutture sintattiche che lo rendono più comprensibile agli umani. Ad esempio, permette l'uso di nomi simbolici per locazioni di memoria: `MOV \%EAX, pippo`.

1.1.2 Istruzioni in codifica mnemonica

Un'istruzione ha 3 campi:

- **Codice operativo:** stabilisce quale operazione eseguire;
- **Suffisso di lunghezza:** stabilisce la lunghezza (che può variare) degli operandi;
- **Operandi:** gli operandi su cui si applica l'operazione. Possono essere contenuti in registri, in celle di memoria, nelle porte I/O o direttamente nell'istruzione (**costanti**).

Il suffisso di lunghezza può essere omesso quando è chiaro (essenzialmente quando si usa un registro).

Sintatticamente la struttura è `OPCODEsuffix source, dest`, che diventa qualcosa come `ADD \%BX, pluto`. Questa istruzione effettua l'operazione `ADD` (aggiungi), aggiungendo al registro `BX` ciò che è contenuto nel simbolo `pluto`.

Operandi di istruzioni

Le istruzioni ammettono 0, 1 o 2 operandi. Quando sono 2, il primo operando si chiama **sorgente** e il secondo **destinatario**, e solitamente hanno la stessa lunghezza. Quando è 1, l'operando può essere sia sorgente che destinatario a seconda dell'istruzione.

1.1.3 Primo esempio di programma

Si presenta un programma per contare il numero di uno trovati dalla locazione `0x00000100` a `0x0000010i3`e scriverlo nella locazione `0x00000104`.

```
MOVB $0x00, %CL          # sposta $0x00 in %CL
MOVL 0x00000100, %EAX    # sposta 32 bit da 0x00000100 a %EAX
CMPL $0x00000000, %EAX   # confronta 32 bit di 0 con il registro %EAX
JE   %EIP+$0x07          # salta se uguale a %EIP+$0x07,
                         # ergo 0x0000020C + 0x07 = 0x00000213
SHRL %EAX                # trasla a destra %EAX
ADCB $0x00, %CL          # aggiungi a %CL 0 + carry
JMP  %EIP-$0x0C          # salta incondizionato a %EIP-$0x0C,
                         # ergo 0x00000213 - 0x0C = 0x00000207
MOVB %CL, 0x00000104     # sposta byte da %CL a 0x00000104
```

Il programma svolge i seguenti passi:

Algoritmo 1 Conta 0

Inizializza il registro CL (Counter Low) a 0
Sposta i 32 bit da `0x00000000` a `0x00000103` in EAX
while true **do**
  if EAX è vuoto (tutti zeri) **then**
    Salta all'ultima istruzione
  end if
  Sposta EAX a destra
  Aggiungi il flag carry (che prende il valore rimosso da EAX) al registro CL
end while
Sposta il byte in CL nella locazione `0x00000104`

1.1.4 Istruzioni assembler

Le istruzioni assembler si dividono in:

- **Operative:** ovvero quelle che svolgono qualche operazione (ADD, SHR, MOV, CMP, ...);
- **Di controllo**: cioè che si occupano di altreare il flusso del programma (JMP, JE, ecc...).

Indirizzamento delle istruzioni operative

Le istruzioni operative si indirizzano attraverso l'**OPCODE** (codice operazione, ADD, MOV, ecc...), seguito da un suffisso (**B**, *byte* da 8 bit, **W**, *word* da 16 bit o **L**, *long* da 32 bit) che può essere omesso, e gli indirizzi sorgente e destinazione.

- Si possono **indirizzare i registri** sia come sorgenti che come destinatari, ovvero gli 8 registri generali da 32 bit, gli 8 registri generali da 16 bit, e gli 8 registri generali da 8 bit (disponibili solo sui registri A, B, C e D). Bisogna precedere i nomi dei registri con %.
- Si può avere **indirizzamento immediato**, ovvero di costanti preceduti da **$**, solo sull'operando sogente.

- Si può **indirizzare la memoria**, ma solo da sorgente o solo da destinatario, specificando un'indirizzo di memoria da 32 bit. Ergo non posso scrivere:

```
MOVB pippo, pluto
```

ma devo scrivere:

```
MOV pippo, %EAX % qua il suffisso di lunghezza e' implicito
MOVL %EAX, pluto
```

L'indirizzamento della memoria, nel caso più generale, è dato da:

$$\text{indirizzo} = \text{base} + \text{indice} \times \text{scala} \pm \text{displacement}$$

dove base e indice sono due registri generali da 32 bit, scala una costante dal valore 1 (default), 2, 4, 8, e displacement una costante intera.

La sintassi è `OPCODEsfx $\pm$disp(base,indice,scala)`.

Si distingue poi l'indirizzamento di tipo:

  - **Diretto**, dove si indica soltanto il displacement, che coincide con l'indirizzo. `OPCODEW 0x00002001` significa prendi la word a partire da `0x00002001`.
  - **Indiretto**, o con registro puntatore, dove si sfrutta un registro: `OPCODEL (\%EBX)` significa indirizzare il valore indirizzato da EBX. Si può specificare una scala: `OPCODEL (,\%EBX,4)` significa il valore nel registro EBX moltiplicato per 4. Si noti come a essere moltiplicato è l'indice e non la base.
  - **Displacement e registro di modifica**, ad esempio da `OPCODEW 0x002A3A2B (\%EDI )` si ottiene l'operando a 16 bit ottenuto sommando al displacement `0x002A3A2B` il contenuto di EDI, modulo 2^32.
  - **Bimodificato senza displacement**, ad esempio `OPCODEW (\%EBX, \%EDI)`, che dipende sia da EBX che da EDI. Si può anche includere una scala: `OPCODEW (\% EBX, \%EDI, 8)`.
  - **Bimodificato con displacement**, come prima ma con displacement: `OPCODEB 0x002F9000 (\%EBX, \%EDI)`, ovvero l'indirizzo dato da base in EBX + indice in EDI + l'offset modulo 2^32. Si può avere anche negativo: `OPCODEB -0x9000 (\%EBX , \%EDI)`, dove si sottrae l'offset invece di sommarlo.

Notare che senza il $ i numeri in formato esadecimale sono interpretati automaticamente come indirizzi.

- Si possono **indirizzare le porte I/O**, come prima in uno solo dei due operandi. Questo si fa con le istruzioni specifiche IN e OUT. In particolare si ha indirizzamento di tipo:

  - **Diretto**, solo per indirizzi < 256, in quanto nel formato macchina ci sono 8 bit. Ad esempio `IN 0x001A, \%AL` o `OUT \%AL, 0x003A`.
  - **Indiretto con registro puntatore**, usando come registro puntatore soltanto DX. Ad esempio `IN (\%DX), \%AX` o `OUT \%AL, (\%DX)`.

1.2 Panoramica sulle istruzioni

Abbiamo diviso le istruzioni in **operative** e **di controllo**. Possiamo fare ulteriori suddivisioni:

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

Conviene definire formato, funzionamento, comportamento sui flag e modalità di indirizzamento ammesse per gli operandi di ogni operazione, in quanto l'assembler non è **ortogonale**, ergo ci sono particolari restrizioni su *quali* operandi e modalità di indirizzamento possono essere combinate.