ASM x8086

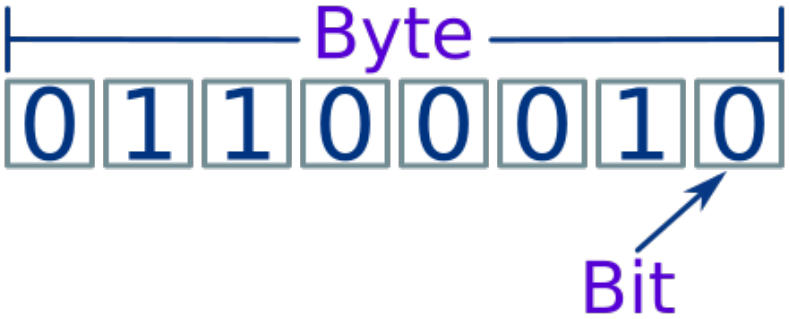
More` Riccardo

INTRODUZIONE

L’assembly e`, senza dubbio, il linguaggio di programmazione piu` a basso livello e piu` educativo rispetto al funzionamento dell’hardware su cui si sta lavorando. Per questo viene tuttora insegnato nelle scuole, nonostante sia ormai quasi in disuso nelle applicazioni aziendali.

MEMORIA

Uno dei motivi principali per cui l'assembly insegna a ragionare in modo vicino all'hardware e` l'accesso e utilizzo diretto alla memoria: il programmatore invece che gestire variabili che convenientemente gestiscono la memoria tramite un *garbage collector* e` costretto a maneggiare indirizzi, registri, bit, stack e altri concetti che non sono fondamentali nei linguaggi piu` high-level. (sento la necessita` di includere per completezza che alcuni linguaggi come il C e C++ consentono le operazioni *bit-wise*, l'uso di registri con la keyword *register* e in generale un accesso low-level all'hardware)



Per poter programmare in assembly e` necessario conoscere le unita` fondamentali della memoria, ovvero:

- **bit**, l'unita` piu` piccola presente in memoria: e` un singolo valore booleano.
- **nibble**, un insieme di 4 bit.
- **byte**, un insieme di 2 nibble, e` la grandezza di un carattere Extended ASCII. E` rappresentato nell'immagine soprastante.
- **word**, un insieme di 2 byte.
- **dword**, un insieme di 2 word.

Bisogna inoltre sapere che il bit piu` a sinistra in un dato si chiama **high bit**.

REGISTRI

I registri sono il nostro banco da lavoro ed uno strumento fondamentale in cui andremo a riporre i nostri dati da elaborare; Quelli fondamentali sono:

- **AX**, registro generale a 16 bit che viene utilizzato convenzionalmente per le istruzioni aritmetiche.
- **BX**, registro generale a 16 bit che viene utilizzato per l'utilizzo di pointer.
- **CX**, registro generale a 16 bit che viene utilizzato come indice nei loop.
- **DX**, registro generale a 16 bit che viene utilizzato per l'I/O (Input/Output).
- **SI**, registro dedicato all'indirizzo del primo elemento di un array.
- **DI**, registro dedicato all'indirizzo dell'elemento corrente di un array.
- **BP**, registro dedicato all'indirizzo del primo elemento dello stack.
- **SP**, registro dedicato all'indirizzo dell'ultimo elemento dello stack.
- **IP**, registro dedicato all'indirizzo dell'istruzione corrente.
- **CS**, registro dedicato all'indirizzo del segmento del codice.
- **DS**, registro dedicato all'indirizzo del segmento dei dati.
- **ES**, registro dedicato all'indirizzo del segmento extra.
- **SS**, registro dedicato all'indirizzo del segmento dello stack.
- **PSW**, registro dedicato alle flag.

Accedere ai registri a 8 bit

I registri generali (AX, BX, CX, DX) sono grandi 16 bit, una word.
Si puo` tuttavia accedere anche ai singoli byte che li compongono, l'**high byte** e il **low byte**. (da non confondere con il concetto di high bit)
Per accedervi e` sufficiente sostituire alla 'X' del registro una 'H' o una 'L', per accedere rispettivamente all'high e low byte.

ISTRUZIONI FONDAMENTALI

- **ADD operando_1 operando_2**, aggiunge operando_2 a operando_1 salvando il risultato in operando_1.
- **SUB operando_1 operando_2**, sottrae operando_2 da operando_1 salvando il risultato in operando_1.
- ***IMUL** / **MUL operando**, moltiplica l'operando per AL e salva il risultato in AL.
- ***IDIV** / **DIV operando**, divide il valore contenuto in AL per l'operando e salva il risultato in AL.
- **CMP operando_1 operando_2**, sottrae il secondo operando dal primo senza salvare il risultato ma settando gli appositi flag.
- **JMP label**, esegue un salto verso una *label*(ettichetta).
- **INC / DEC operando**, incrementa o decrementa l'operando specificato.
- **MOV operando_1 operando_2**, copia operando_2 dentro ad operando_1.
- **INT codice**, invia un interrupt all'*OS* (Sistema Operativo, in questo caso il DOS) corrispondente al codice specificato.
- **XOR / OR / AND operando_1 operando_2**, esegue l'operazione *bit-wise* fra gli operandi.
- **JX / JXX label**, esegue un salto condizionale verso una label, una lista esaustiva dei jump condizionali e` disponibile attraverso questo link.
- **Push/Pop operando**, sposta un elemento dallo/allo stack ad/da un registro. Utilizza il principio *LIFO*(Last In First Out, l'ultimo elemento aggiunto e` il primo ad essere rimosso).

*La differenza tra le operazioni base e la loro variante con la "i" (e.g. mul / imul) risiede nel tipo di operando: le operazioni base utilizzano *unsigned integer*, mentre le varianti con la i adoperano *signed integer*.

AMBIENTE DI SVILUPPO

L’ambiente di sviluppo e` il DOSBOX, un emulatore che simula l’hardware di un *Personal Computer* e offre un interfaccia CLI del MS-DOS.
Gli strumenti utilizzati per la compilazione dei programmi in assembly sono il Turbo Debugger, il Turbo Linker e il Turbo Compiler.
I comandi essenziali per compilare un eseguibile usando la suite di TASM sono:

```
E:\>TASM.EXE FILE.ASM
```

Questo comando esegue il *compiling* partendo dal *source code*.

```
E:\>TLINK.EXE FILE.OBJ
```

Questo comando esegue il *linking* sul nostro file oggetto e ci restituisce il nostro eseguibile a 16 bit.

Consiglio:
Per evitare di montare ogni volta la partizione contenente i file TASM e` possibile modificare il file delle impostazioni di default di DOSBOX: basta aggiungere i comandi che si vogliono eseguire allo start-up nel file menzionato dalla **DOSBox Status Window**.

SCHELETRO DI UN PROGRAMMA

Per rendere un programma in assembly compilabile e funzionante sono necessari vari elementi fondamentali.
Potete trovarli riportati in seguito sotto forma di esempio.

```
;direttiva per definire il modello di segmentazione
.model tiny

;apertura della direttiva .code
.code

        ;set-up segmento dati
        mov ax,@data
        mov ds,ax

        ;codice del programma
        mov ah,1
        int 21h
        sub al,48
        mov bl,al
        ...
        dec cl
        jnz mylabel
        dec bl
        jnz mylabel2

        ;interrupt per ritornare il controllo al DOS
        mov ah,4ch
        int 21h

;chiusura della direttiva .code
end
```

OTTIMIZZAZIONE E *BEST PRACTICES*

Nonostante l'ottimizzazione sia pressoche` inutile dato che DOSBOX non e` *cycle-accurate*(non emula il numero di cicli dell'hardware originale bensi` usa al 100% di capacita` una thread del processore) rimane un esercizio interessante e divertente per approfondire il linguaggio.
La seguente e` una lista delle ottimizzazioni e best practice piu` rilevanti che io ricordi:

- E` best practice usare **MOV registro,0** per azzerare un registro, e` pero` ottimizzabile con **XOR registro,registro**.
- E` possibile sfruttare il fatto che molte istruzioni(e.g. mul, a dd, inc, etc. etc.) settano i flag per saltare l'utilizzo di **CMP** all'interno di un loop.
- E` best practice usare l'istruzione **LOOP** per i cicli determinati, e` tuttavia piu` veloce **CMP** + **JMP**.

E` inoltre da notare che classici trucchi per l'ottimizzazione applicabili in C/C++ sono riproducibili in assembly, come lo shift a sinistra per le potenze di 2.