Ragionando su *Assembly*

Salvatore Capolupo

Panoramica

In questo documento si fa riferimento ad esempio puramente didattici, che potrebbero non funzionare in un'ottica di un'applicazione reale o in contesto professionale.

L'Assembly 8086 è un linguaggio di programmazione di basso livello progettato per il microprocessore Intel 8086. Questo linguaggio consente di operare direttamente sull'hardware attraverso l'uso di istruzioni semplici e veloci. È particolarmente utilizzato per la programmazione di sistema e l'ottimizzazione delle prestazioni in contesti a basso livello.

Registri Principali

Il processore 8086 dispone di registri a 16 bit suddivisi in varie categorie:

- **Registri di uso generale**: AX, BX, CX, DX.
- **Registri di indice**: SI (Source Index), DI (Destination Index).
- **Registri di puntatore**: SP (Stack Pointer), BP (Base Pointer).
- **Registri di segmento**: CS (Code Segment), DS (Data Segment), SS (Stack Segment), ES (Extra Segment).

Istruzioni di Base

Alcune delle istruzioni più comuni del linguaggio Assembly 8086 includono:

- **MOV** – Trasferisce dati tra registri o tra memoria e registri.
- **ADD**, **SUB** – Eseguono operazioni aritmetiche di somma e sottrazione.

- **MUL**, **DIV** – Moltiplicazione e divisione.
- **AND**, **OR**, **XOR**, **NOT** – Operazioni logiche bit a bit.
- **JMP**, **JE**, **JNE**, **JG**, **JL** – Istruzioni di salto condizionato e incondizionato.
- **CALL**, **RET** – Chiamata e ritorno da sottoprogrammi.
- **INT** – Genera un'interruzione software.

Esempio di Codice

Un semplice esempio che mostra la somma di due numeri:

```
MOV AX, 5       ; Carica 5 in AX
MOV BX, 3       ; Carica 3 in BX
ADD AX, BX      ; Somma BX ad AX
; AX ora contiene 8
```

Calcolo delle percentuali

Un mercato ha un valore complessivo iniziale di **1000** euro: in un primo momento, perde il **5%** del proprio valore. In seguito guadagna di nuovo il **5%**. Il mercato, pertanto, sarà:

- A. Tornato al valore iniziale (1000)
- B. Diminuito
- C. Aumentato

Spiegazione

1. Valore iniziale: 1000
2. Dopo la perdita del 5%:

$$1000 - \frac{5}{100} \cdot 1000 = 1000 - 50 = 950$$

3. Ora calcoliamo il guadagno del 5% su 950:

$$950 + \frac{5}{100} \cdot 950 = 950 + 47.5 = 997.5$$

4. Risultato finale: 997.5

Conclusione: Il mercato ha **perso** $1000 - 997.5 = \boxed{2.5}$

Che c'entra con Assembly 8086?!

C'entra! Supponiamo di voler fare una simulazione semplice di questi calcoli in Assembly, usando il registro `AX` per il valore iniziale. Questo ci aiuterà a testare quanto abbiamo visto e renderci conto di come si possa scrivere in questo linguaggio.

Codice Assembly 8086 (semplificato)

```
ORG 100H

; AX contiene il valore iniziale: 1000
MOV AX, 1000        ; AX = 1000
MOV BX, AX          ; Salviamo il valore iniziale in BX

; Calcolo del 5% di perdita: AX * 5 / 100
MOV CX, 5
MUL CX              ; DX:AX = AX * CX = 1000 * 5 = 5000
MOV CX, 100
DIV CX              ; AX = 5000 / 100 = 50 (perdita)

; Sottraiamo la perdita da BX
SUB BX, AX          ; BX = 1000 - 50 = 950

; Ora calcoliamo il 5% di guadagno su 950
MOV AX, BX          ; AX = 950
MOV CX, 5
MUL CX              ; AX = 950 * 5 = 4750
MOV CX, 100
DIV CX              ; AX = 47.5     risultato intero = 47

ADD BX, AX          ; BX = 950 + 47 = 997
; Risultato finale in BX     997
```

Nota: Questo esempio semplifica il calcolo troncando le frazioni decimali, come tipico nei registri interi.

Crescita delle dimensioni di un file

Un file presenta dimensione iniziale di **1 KB**: ogni minuto, viene duplicato e salvato nella stessa cartella, raddoppiando quindi lo spazio occupato. Dopo **10 minuti**, quanta memoria occupa?

- A. 10 KB
- B. 1024 KB
- C. 2047 KB
- D. 2^{10} KB

Spiegazione semplice

Ogni minuto, il numero di file raddoppia:

- Minuto 0: 1 file = 1 KB
- Minuto 1: 2 file = 2 KB
- Minuto 2: 4 file = 4 KB
- ...
- Minuto 10: 2^{10} file = 1024 KB

Per sapere quanto spazio occupano tutti i file generati nel tempo, sommiamo tutte le dimensioni precedenti:

$$1 + 2 + 4 + \cdots + 2^9 = 2^{10} - 1 = 1023 \text{ KB}$$

Alla fine del decimo minuto, viene creato anche il file 2^{10}:

$$1023 + 1024 = \boxed{2047 \text{ KB}}$$

Conclusione: Risposta corretta = **C. 2047 KB**

Calcolo in Assembly

Questo è un tipico esempio di **crescita esponenziale**, comune in algoritmi ricorsivi, alberi binari, gestione cache o backup duplicati.

Esempio pratico

Supponiamo di voler calcolare $2^{10} - 1$ usando operazioni bitwise.

```
; Calcolo 2^10 - 1 = 1024 - 1 = 1023
MOV AX, 1
SHL AX, 10      ; AX = 2^10 = 1024
DEC AX          ; AX = 1023
; AX contiene il risultato finale
```

Nota: Lo shift sinistro (SHL) moltiplica per 2 alla potenza indicata. È molto usato per ottimizzare moltiplicazioni.

Le stringhe palindrome

Una stringa contiene una parola palindroma, ovvero una parola che si legge uguale da sinistra a destra e da destra a sinistra, come `radar` o `level`. Considera la seguente stringa:

`r1a2d3a4r`

- A. Non è palindroma
- B. È palindroma solo se rimuoviamo i numeri
- C. È palindroma solo se invertiamo i numeri
- D. Non è verificabile

Spiegazione semplice

Analizziamo la stringa: `r1a2d3a4r`

- Se eliminiamo i numeri otteniamo: `radar`
- `radar` è un palindromo
- I numeri sono irrilevanti per la simmetria delle lettere

Conclusione: la stringa `r1a2d3a4r` è palindroma se ignoriamo i caratteri non alfabetici.

Risposta corretta: B. È palindroma solo se rimuoviamo i numeri

Nei linguaggi di programmazione, per verificare se una stringa è palindroma, spesso si:

- filtrano i caratteri (es. si rimuovono numeri, spazi, punteggiatura)
- si confrontano la stringa con la sua versione invertita
- Come si fa in Assembly?

Esempio in Assembly 8086 (semplificato)

Supponiamo di voler copiare solo i caratteri alfabetici in un buffer per controllare se sono palindromi:

```
; SI punta all'inizio della stringa 'r1a2d3a4r'
; DI punta a un buffer vuoto

NEXT_CHAR:
    LODSB               ; AL = [SI], carica prossimo
       carattere
    CMP AL, 'A'
    JB SKIP_CHAR        ; Salta se < 'A'
    CMP AL, 'z'
    JA SKIP_CHAR        ; Salta se > 'z'
    STOSB               ; Se    alfabetico, salvalo nel
       buffer

SKIP_CHAR:
    CMP AL, 0
    JNZ NEXT_CHAR       ; Continua finch  non trovi il
       terminatore 0
```

Nota: Questo codice filtra i caratteri non alfabetici. Dopo si può confrontare il buffer con la sua versione invertita.

Le funzioni in Assembly

Supponiamo di avere una funzione in Assembly che raddoppia il valore contenuto nel registro `AX`.

Il codice principale inizializza `AX` con il valore `3`, poi chiama la funzione due volte di seguito.

Qual è il valore di `AX` alla fine?

- A. 6
- B. 12
- C. 9
- D. 3

Spiegazione passo-passo

- **Inizio:** AX = 3
- **Prima chiamata:** AX = $3 \times 2 = 6$
- **Seconda chiamata:** AX = $6 \times 2 = 12$

Per cui: Risposta corretta = **B. 12**

Questo esercizio introduce il concetto di **funzione** in Assembly. Le funzioni sono blocchi riutilizzabili di codice che:

- Vengono richiamati con `CALL`
- Ritornano al punto successivo con `RET`
- Possono usare i registri per passare e restituire valori

Esempio

```
; Inizio del programma
MOV AX, 3          ; Valore iniziale
CALL RADDOPPIA     ; Prima chiamata -> AX = 6
CALL RADDOPPIA     ; Seconda chiamata -> AX = 12
; Ora AX contiene 12

; Definizione della funzione
RADDOPPIA:
    SHL AX, 1      ; AX = AX * 2
    RET
```

Nota: La funzione modifica direttamente AX usando uno shift a sinistra di 1 bit (moltiplicazione per 2). Nessun parametro esplicito: si lavora direttamente sul registro.

La pila (stack)

Supponi di avere una pila (stack) inizialmente vuota. Inseriamo, nell'ordine, i valori seguenti:

```
PUSH 5, PUSH 8, PUSH 2
```

Successivamente, eseguiamo due `POP`.
Quali valori verranno estratti, e in che ordine?

- A. 5 e 8
- B. 2 e 8
- C. 8 e 5
- D. 2 e 5

Spiegazione semplice

Lo stack è una struttura dati **LIFO** (Last In, First Out) → *"Ultimo che entra, primo che esce"*.

- `PUSH 5` → stack: [5]
- `PUSH 8` → stack: [8, 5]
- `PUSH 2` → stack: [2, 8, 5]

Eseguendo:

- `POP` → esce 2
- `POP` → esce 8

Risposta corretta: B. 2 e 8

Come si usa in Assembly

Lo stack viene usato per:

- Salvare valori temporanei
- Gestire chiamate a funzione (ritorni, parametri)
- Lavorare con variabili locali

Ogni `PUSH` decrementa lo stack pointer (`SP`) e salva un valore. Ogni `POP` legge il valore più in alto e incrementa `SP`.

Esempio in Assembly 8086

```
MOV AX, 5
PUSH AX         ; Stack: [5]
MOV AX, 8
PUSH AX         ; Stack: [8, 5]
MOV AX, 2
PUSH AX         ; Stack: [2, 8, 5]

POP BX          ; BX = 2
POP CX          ; CX = 8
```

Nota: Il valore 2 viene estratto per primo, poi l'8. Il 5 rimane nello stack.

Domanda di Logica

Una funzione ricorsiva calcola il fattoriale (`n!`) di un numero intero positivo. Supponiamo di voler calcolare `3!` con questa funzione.

Quante chiamate verranno inserite nello **stack** prima che la funzione cominci a "tornare indietro"?

- A. 1
- B. 2
- C. 3
- D. 4

Spiegazione

Il fattoriale è definito così:

$$n! = n \times (n-1) \times (n-2) \cdots \times 1$$

In forma ricorsiva:

$$\texttt{fattoriale(n)} = \begin{cases} 1, & \text{se } n = 1 \\ n \times \texttt{fattoriale(n-1)}, & \text{se } n > 1 \end{cases}$$

Per `fattoriale(3)` si hanno:

- Chiamata a `fattoriale(3)` → salvata nello stack
- Chiamata a `fattoriale(2)` → salvata nello stack
- Chiamata a `fattoriale(1)` → salvata nello stack

Alla chiamata `fattoriale(1)`, la funzione restituisce 1 e comincia a tornare indietro.

Risposta corretta: C. 3

Collegamento con Assembly

Ogni volta che una funzione viene chiamata, il processore:

- Salva l'indirizzo di ritorno sullo **stack**

- Salva eventualmente parametri e variabili locali
- Alla fine della funzione, esegue `RET`, che preleva dallo stack l'indirizzo per tornare

Nel caso della ricorsione, ogni chiamata è "sospesa" finché non viene completata la successiva. Per questo lo stack cresce.

Esempio in Assembly 8086 semplificato

```
; Supponiamo AX contenga il valore di N (es. 3)
; Fattoriale ricorsivo: semplificato per spiegare lo
   stack

FATTORIALE:
    CMP AX, 1
    JE BASE_CASE        ; Se AX = 1, ritorna

    PUSH AX             ; Salva AX sullo stack
    DEC AX              ; AX = AX - 1
    CALL FATTORIALE     ; Chiamata ricorsiva
    POP BX              ; Riprendi AX originale da BX
    MUL BX              ; AX = AX * BX (risultato finale)
    RET

BASE_CASE:
    MOV AX, 1           ; Caso base: fattoriale(1) = 1
    RET
```

Nota: Ogni chiamata ricorsiva aggiunge un livello allo stack. Quando si arriva al caso base, le `RET` iniziano a svuotare lo stack.