S10L3

Traccia:

Nella lezione teorica del mattino, abbiamo visto i fondamenti del linguaggio Assembly. Dato il codice in Assembly per la CPU x86 allegato qui di seguito, identificare lo scopo di ogni istruzione, inserendo una descrizione per ogni riga di codice.

Ricordate che i numeri nel formato 0xYY sono numeri esadecimali. Per convertirli in numeri decimali utilizzate pure un convertitore online, oppure la calcolatrice del vostro computer (per programmatori).

```
0x00001141 <+8>:    mov   EAX,0x20
0x00001148 <+15>:   mov   EDX,0x38
0x00001155 <+28>:   add   EAX,EDX
0x00001157 <+30>:   mov   EBP, EAX
0x0000115a <+33>:   cmp   EBP,0xa
0x0000115e <+37>:   jge    0x1176 <main+61>
0x0000116a <+49>:   mov   eax,0x0
0x0000116f <+54>:   call   0x1030 <printf@plt>
```

Prima di procedere con l'esercizio assegnato nella traccia di oggi, ripassiamo quello imparato nella a lezione. Oggi abbiamo introdotto il linguaggio Assembly; questo, è un linguaggio di programmazione a basso livello strettamente legato all'architettura del processore di un computer. Ogni istruzione in Assembly corrisponde quasi direttamente a un'istruzione del set di istruzioni del processore, permettendo il controllo dettagliato dell'hardware, consentendo così ai programmatori di manipolare direttamente i registri del processore, la memoria e altri componenti hardware.

A differenza dei linguaggi di alto livello, il codice Assembly è specifico per l'architettura del processore e non è facilmente portabile. È anche meno leggibile per gli esseri umani rispetto ai linguaggi di alto livello a causa della sua sintassi tecnica.

Le istruzioni sono eseguite direttamente dal processore senza ulteriori livelli di interpretazione o compilazione, il che rende il codice Assembly molto efficiente in termini di velocità di esecuzione e utilizzo delle risorse.

Assembly è spesso utilizzato in situazioni dove è necessaria la massima efficienza, come nel firmware, nei sistemi embedded, nei driver di dispositivo, e in altre applicazioni critiche per le prestazioni.

Ogni tipo di processore ha il proprio set di istruzioni, il che significa che il codice Assembly scritto per un'architettura non funzionerà su un'altra senza modifiche significative.

Lo sviluppo in Assembly richiede spesso strumenti specializzati come assemblatori, debugger ed emulatori per scrivere, testare e ottimizzare il codice.

In sintesi, il linguaggio Assembly offre un controllo fine sull'hardware del computer, ma richiede una profonda comprensione dell'architettura del processore e del funzionamento interno del sistema.

Esercizio:

- **Analizziamo nel dettaglio la riga 1 del codice: 0x00001141 <+8>: mov EAX,0x20 e vediamo cosa rappresenta ogni sezione**

0x00001141 = Indirizzo di memoria

Questo è l'indirizzo di memoria in cui l'istruzione è memorizzata. Indica il punto preciso nel programma dove si trova questa istruzione.

<+8> = Offset

Indica la posizione dell'istruzione. In questo caso, l'istruzione si trova a 8 byte dall'inizio della funzione o del blocco di codice.

mov EAX, 0x20 = Istruzione Assembly

dove:

mov indica un'istruzione di trasferimento, ovvero copia un valore da una sorgente a una destinazione.

EAX è il registro di destinazione. (EAX è uno dei registri generali a 32 bit della CPU x86)

0x20 è l'operando sorgente, un valore immediato esadecimale (32 in decimale). Questo valore viene copiato nel registro EAX.

In sintesi: Carica il valore esadecimale 0x20 (32 in decimale) nel registro EAX.

- **Passiamo ora alla riga 2: 0x00001148 <+15>: mov EDX,0x38**

0x00001148 = Indirizzo di memoria

Questo è l'indirizzo di memoria in cui l'istruzione è memorizzata.

<+15> = Offset

Indica la posizione dell'istruzione. In questo caso, si trova a 15 byte dall'inizio della funzione o del blocco di codice.

mov EDX, 0x38 = Istruzione Assembly

dove:

mov indica un'istruzione di trasferimento ovvero copia un valore da una sorgente a una destinazione.

EDX è il registro di destinazione.

0x38 è l'operando sorgente che viene copiato nel registro EDX.

In sintesi: Carica il valore esadecimale 0x38 (56 in decimale) nel registro EDX.

- **Riga 3 del codice: 0x00001155 <+28>: add EAX,EDX**

0x00001155 = Indirizzo di memoria

Indica il punto preciso nel programma dove si trova questa istruzione.

<+28> = Offset

Indica la posizione dell'istruzione. In questo caso si trova a 28 byte dall'inizio della funzione o del blocco di codice.

add EAX, EDX = Istruzione Assembly

dove:

add somma il valore del secondo operando al primo operando e immagazzina il risultato nel primo operando.

EAX è il registro di destinazione e anche il primo operando.

EDX è il secondo operando. EDX è un altro registro generale a 32 bit della CPU x86.

In sintesi: Somma il valore del registro EDX al valore del registro EAX e memorizza il risultato in EAX. (EAX = 32 + 56 = 88)

- **Passiamo ora alla riga 4 del codice: 0x00001157 <+30>: mov EBP, EAX**

0x00001157 = Indirizzo di memoria

Questo è l'indirizzo di memoria in cui l'istruzione è memorizzata.

<+30> = Offset

Indica la posizione dell'istruzione. In questo caso, l'istruzione si trova a 30 byte dall'inizio della funzione o del blocco di codice.

mov EBP, EAX = Istruzione Assembly

dove:

mov indica un'istruzione di trasferimento ovvero copia un valore da una sorgente a una destinazione.

EBP è il registro di destinazione.

EAX è l'operando sorgente.

In sintesi: Copia il valore del registro EAX nel registro EBP. (EBP = 88)

- **Descrizione della riga 5 del codice: 0x0000115a <+33>: cmp EBP,0xa**

0x0000115a = Indirizzo di memoria

Indica il punto preciso nel programma dove si trova questa istruzione.

<+33> = Offset

Indica la posizione dell'istruzione. In questo caso, l'istruzione si trova a 33 byte dall'inizio della funzione o del blocco di codice.

cmp EBP, 0xa = Istruzione Assembly

dove:

cmp è un'istruzione di confronto, ovvero confronta due operandi e imposta i flag nel registro dei flag della CPU in base al risultato del confronto.

EBP è il primo operando.

0xa è il secondo operando ed è un valore immediato esadecimale che rappresenta il numero 10 in decimale.

In sintesi: Confronta il valore del registro EBP con il valore esadecimale 0xa (10 in decimale).

- **Eccoci alla descrizione della riga 6: 0x0000115e <+37>: jge 0x1176 <main+61>**

0x0000115e = Indirizzo di memoria

Indica il punto preciso nel programma dove si trova questa istruzione.

<+37> = Offset

indica la posizione, in questo caso, l'istruzione si trova a 37 byte dall'inizio della funzione o del blocco di codice.

jge 0x1176 <main+61> = Istruzione Assembly

dove:

jge indica un'istruzione di salto condizionato. L'istruzione `jge` effettua un salto condizionato se il flag di "Greater than or Equal" è impostato; ovvero salta a un'altra parte del codice se il confronto ha dato esito positivo.

0x1176 è l'indirizzo di destinazione dell'istruzione di salto condizionato.

<main+61> Questa è una nota sulla destinazione del salto. Indica che l'indirizzo 0x1176 si riferisce alla posizione all'interno della funzione main, a 61 byte dall'inizio della funzione.

In sintesi: Salta all'indirizzo 0x1176 se il valore di EBP è maggiore o uguale a 10.

- **Riga 7 del codice nel dettaglio 0x0000116a <+49>: mov eax,0x0**

0x0000116a = Indirizzo di memoria

Indica il punto preciso nel programma dove si trova questa istruzione.

<+49> = Offset

Indica la posizione. In questo caso, l'istruzione si trova a 49 byte dall'inizio della funzione o del blocco di codice.

mov eax, 0x0 = Istruzione Assembly

dove:

mov indica un'istruzione di trasferimento, ovvero copia un valore da una sorgente a una destinazione.

eax è il registro di destinazione.

0x0 è l'operando sorgente, un valore esadecimale che corrisponde a zero in decimale.

In sintesi: Carica il valore esadecimale 0x0 (0 in decimale) nel registro EAX.

- **Passiamo ora all’ultima riga del codice: 0x0000116f <+54>: call 0x1030 <printf@plt>**

0x0000116f = Indirizzo di memoria

Indica il punto preciso nel programma dove si trova questa istruzione.

<+54> = Offset

Indica la posizione, in questo caso, l'istruzione si trova a 54 byte dall'inizio della funzione o del blocco di codice.

call 0x1030 <printf@plt> = Istruzione Assembly

dove:

call indica un'istruzione di chiamata a una procedura.

0x1030 è l'indirizzo di destinazione della chiamata alla procedura.

<printf@plt> questa è una nota sulla destinazione della chiamata, indica che l'indirizzo 0x1030 è associato alla funzione printf all'interno della procedura linkage table (PLT).

In sintesi: Chiama la funzione printf situata all'indirizzo 0x1030.

Conclusioni:

Ora che abbiamo visto nello specifico cosa rappresenta ogni voce del codice, possiamo sintetizzare affermando che il codice esegue una somma, verifica se il risultato è maggiore o uguale a 10, e in base a questa verifica decide se eseguire un salto condizionato o chiamare la funzione printf.