# Relazione sul Linguaggio Assembly

## Introduzione

Il linguaggio assembly è un linguaggio di basso livello utilizzato per la programmazione delle unità di elaborazione dei computer. Si tratta di un linguaggio a basso livello in quanto è strettamente legato all'architettura del computer, utilizzando istruzioni che la CPU comprende direttamente. In questa relazione, esploreremo le caratteristiche, l'utilizzo e l'importanza del linguaggio assembly nella programmazione dei computer.

## Caratteristiche del Linguaggio Assembly

1. Istruzioni di Base: Il linguaggio assembly consiste in un insieme di istruzioni di base, ciascuna delle quali corrisponde a un'operazione specifica eseguita dalla CPU. Queste istruzioni includono operazioni aritmetiche, logiche, di controllo del flusso e di gestione della memoria.
2. Rappresentazione Diretta: Le istruzioni assembly sono una rappresentazione diretta delle istruzioni eseguite dalla CPU. Ogni istruzione assembly corrisponde a una singola istruzione del linguaggio macchina, rendendo più facile comprendere come il computer esegue il codice.
3. Basso Livello: Essendo a basso livello, il linguaggio assembly offre un controllo molto dettagliato sulle risorse del computer. Gli sviluppatori possono manipolare direttamente i registri della CPU e la memoria, ottenendo un alto grado di ottimizzazione del codice.
4. Specificità dell'Architettura: Il codice assembly è specifico per un'architettura di processore particolare. Il codice scritto per un processore x86 non può essere eseguito su un processore ARM senza modifiche significative.

## Utilizzo del Linguaggio Assembly

1. Programmazione di Sistema: Il linguaggio assembly è spesso utilizzato per scrivere il software di sistema, come sistemi operativi, driver di periferiche e bootloader. Questo perché consente un controllo dettagliato sull'hardware del sistema.
2. Programmazione Embedded: Nei sistemi embedded, dove le risorse sono limitate, l'ottimizzazione del codice è essenziale. Gli sviluppatori spesso utilizzano il linguaggio assembly per ottimizzare parti critiche del codice.
3. Reverse Engineering: Gli esperti di sicurezza informatica utilizzano il linguaggio assembly per analizzare e comprendere il funzionamento interno di software e firmware, aiutando nella scoperta di vulnerabilità di sicurezza.

## Importanza del Linguaggio Assembly

1. Ottimizzazione del Codice: In alcune situazioni, specialmente quando è necessaria un'alta efficienza computazionale, la scrittura di codice in assembly può portare a prestazioni notevolmente migliorate rispetto ai linguaggi di alto livello.
2. Comprendere l'Architettura del Computer: Studiare il linguaggio assembly aiuta gli sviluppatori a comprendere a fondo come funziona l'architettura del computer. Questo è essenziale per progettare algoritmi efficienti e risolvere problemi complessi.
3. Sicurezza Informatica: Gli esperti di sicurezza informatica devono avere una conoscenza approfondita del linguaggio assembly per comprendere e sfruttare le vulnerabilità dei software e dei sistemi.

Una qualsiasi persona si potrebbe chiedere “perché studiare l’assembly quando c’è Python, Java, C++ o [inserisci altro linguaggio]?”. In realtà è presto detto: l’assembly conferisce allo sviluppatore un accesso diretto e senza filtri delle risorse della macchina. Rispetto ad altri linguaggi di programmazione che si basano sulla compilazione (C++) o sono interpretati (Python o Java), assembly non ha bisogno di essere “pre-processato” in alcun modo.

Introduciamo ora il concetto di linguaggio a basso livello e un linguaggio ad alto livello. Dato un problema qualsiasi, lo sviluppatore vuole costruire un programma che possa eseguire alcuni particolari calcoli. Per strutturare il codice, inizia partendo da una specifica in linguaggio naturale (cioè un linguaggio comprensibile tra le persone) e incomincia a scrivere: variabili, funzioni e molto altro. La descrizione di questo codice può essere più o meno vicino al linguaggio naturale o meno, a seconda del linguaggio utilizzato.

Un linguaggio ad alto livello è un linguaggio che si avvicina molto al ragionamento umano ed è caratterizzato da una significativa astrazione dai dettagli di funzionamento di un calcolatore; ciò significa che uno sviluppatore non è tenuto a sapere come funziona una particolare parte dell’architettura, quando scrive il codice. Tre dei maggiori linguaggi che sono definiti ad alto livello sono il C, il C++ e Java.

La particolare semplicità di questi 3 linguaggi di programmazione fanno sì che vengano scelti per insegnare a chiunque come iniziare a programmare. C, C++ e Java non richiedono particolari conoscenze e i loro costrutti sono di facile comprensione, dato che si avvicinano molto al pensiero umano.

Analizziamo per esempio il seguente codice

int somma\_due\_numeri(int a, int b){

int somma;

somma = a + b;

return somma;

}

Dichiariamo una variabile somma e salviamo all’interno della variabile il risultato dell’operazione di somma tra a e b. Noi sviluppatori non sappiamo come in modo pratico il programma gestisca le due variabili a e b, né tanto meno dove va a salvare le tre variabili. In questo caso, come è possibile notare, è il linguaggio “stesso” che fornisce delle astrazioni molto semplici per implementare la somma. Alcuni esempi di astrazioni semplici possono essere int somma per creare una variabile, il segno + per addizionare e anche il fatto di ritornare un valore, senza minimamente preoccuparsi di salvare, cancellare, scrivere in memoria.

Discorso ben diverso per i linguaggi a basso livello che sono più vicini al modo di funzionare di un calcolatore e operano direttamente con le risorse del calcolatore. Per programmare, il programmatore deve necessariamente conoscere la struttura hardware del computer, il funzionamento e l’architettura del processore. In particolar modo, gli indirizzi della memoria e i registri della CPU. Cercheremo quindi di andare a capire in profondità come funziona un elaboratore per comprendere effettivamente il grande vantaggio dell’assembly: impartire istruzioni “grezze” alla CPU.

Confrontando le potenzialità del linguaggio Assembly con quelle di un linguaggio di programmazione ad alto livello, si possono individuare i seguenti vantaggi/svantaggi:

Pro:

* Propedeuticità: aiuta a far capire come funziona veramente un elaboratore;
* Aiuta a scrivere meglio nel linguaggio di più alto livello (ad esempio, C), perché si capisce come poi viene eseguito;
* È il linguaggio di programmazione a basso livello per eccellenza perché consente il più vasto accesso a tutte le risorse del computer;
* Per i punti precedenti, consente di eseguire ottimizzazioni delle prestazioni;

Contro:

* La stesura di un programma Assembly è molto complessa e richede conoscenze non banali;
* Ogni architettura ha uno specifico set di istruzioni, quindi il codice in assembly non è portabile su diverse piattaforme;
* Maggiore lunghezza e minor leggibilità del codice;

## Compilare vs Assemblare

Una persona qualunque potrebbe chiedersi: ma se esistono sia i linguaggi ad alto livello che i linguaggi a basso livello, cosa interpreta un calcolatore? Per poter rispondere a questa domanda, è necessario prima fare un passo indietro.

Come ben si sa, un calcolatore è molto utile nello svolgimento di problemi. Si ha un algoritmo (cioè una serie di istruzioni elementari) la cui esecuzione permette lo svolgimento di un problema. Esempi di algoritmi possono essere dal trovare i primi 10 numeri primi all’algoritmo per ordinare una serie di numeri. Questo algoritmo viene descritto da alcuni particolari linguaggi di programmazione, linguaggi creati a tavolino che permettono di esprimere le istruzioni in un formato comprensibile dal calcolatore.

Data la natura elettronica dell’esecutore, è possibile descrivere un programma solo attraverso una sequenza di segnali elettrici 0 o 1 che vengono interpretati fisicamente da una parte. L’obiettivo dei linguaggi di programmazione è quella quindi di trasformare il codice in linguaggio macchina (cioè una sequenza di 0 e 1). Questa trasformazione è effettuata da due programmi esterni chiamati “compilatore” ed “interprete” (a seconda del linguaggio utilizzato verranno usati entrambi o solo uno dei due).

Un interprete si occupa di valutare il programma: segue il flusso di esecuzione del codice ed esegue contemporaneamente la traduzione in linguaggio macchina dei comandi del programma e la loro esecuzione. Quello che restituisce l’interprete è il risultato dell’esecuzione del programma. Gli interpreti vengono utilizzati per esempio in linguaggi come Python, Ruby, Perl, PHP (vengono chiamati per questo motivo linguaggi interpretati).

Un compilatore invece si occupa di creare un codice oggetto (un binario) partendo dal linguaggio sorgente. L’esecuzione del programma ottenuto è più veloce in quanto la fase di traduzione è già avvenuta. I compilatori vengono utilizzati nei linguaggi di programmazione C, C++,

Per compensare i punti deboli delle due soluzioni , esiste il cosiddetto compilatore just-in-time o a real-time (in inglese “tempestivo”). Questo particolare tipo di compilatore, che a volte è chiamato anche compreter (da compilator e interpreter), traduce il codice del programma come l’interprete, cioè solo durante l’esecuzione. In questo modo, all’alta velocità di esecuzione (grazie al compilatore) si aggiunge un processo di sviluppo semplificato.

Uno degli esempi più noti di linguaggio basato sulla compilazione just-in-time è Java: come componente del Java Runtime Environment (JRE), tale compilatore JIT migliora le prestazioni delle applicazioni Java convertendo il codice byte precedentemente generato in linguaggio macchina durante l’esecuzione.

Andiamo quindi a visualizzare in dettaglio come viene costruito un programma partendo da un sorgente scritto in linguaggio C e la sua effettiva esecuzione.

### Compilare ed eseguire un programma

Il passaggio dal codice sorgente alla esecuzione del programma passa per 3 fasi: compilazione, linking, caricamento ed esecuzione.

Durante la fase di compilazione, il codice viene analizzato e per ogni istruzione viene generata una porzione di linguaggio macchina che la implementa. Anche le istruzioni che riguardano dichiarazioni/allocazioni dei dati sono tradotte in modo opportuno. L’output è un file oggetto in cui vengono mantenuti i simboli usati nel codice (come ad esempio etichette mnemoniche associate ai dati).

Un compilatore esegue quattro passaggi principali:

* Scansione: lo scanner o parser legge un carattere alla volta dal codice sorgente e tiene traccia di quale carattere è presente in quale riga.
* Analisi lessicale: il compilatore converte la sequenza di caratteri che appaiono nel codice sorgente in una serie di stringhe di caratteri (note come token ), che sono associate da una regola specifica da un programma chiamato analizzatore lessicale. Una tabella dei simboli viene utilizzata dall’analizzatore lessicale per memorizzare le parole nel codice sorgente che corrispondono al token generato.
* Analisi sintattica: in questa fase viene eseguita l’analisi della sintassi, che implica la preelaborazione per determinare se i token creati durante l’analisi lessicale sono nell’ordine corretto in base al loro utilizzo. L’ordine corretto di un insieme di parole chiave, che può produrre un risultato desiderato, è chiamato sintassi. Il compilatore deve controllare il codice sorgente per garantire l’accuratezza sintattica.
* Analisi semantica: questo passaggio consiste in diversi passaggi intermedi. Innanzitutto, viene verificata la struttura dei token, insieme al loro ordine rispetto alla grammatica in una data lingua. Il significato della struttura del token viene interpretato dal parser e dall’analizzatore per generare finalmente un codice intermedio, chiamato codice oggetto.

Il codice oggetto include istruzioni che rappresentano l’azione del processore per un token corrispondente quando viene rilevato nel programma. Infine, l’intero codice viene analizzato e interpretato per verificare se sono possibili ottimizzazioni. Una volta eseguite le ottimizzazioni, gli opportuni token modificati vengono inseriti nel codice oggetto per generare il codice oggetto finale, che viene salvato all’interno di un file.

I file oggetto (tradotti in linguaggio macchina) sono collegati assieme, risolvendo i riferimenti ai simboli esterni usati nei vari file oggetto, utilizzando un programma denominato linker. Tra i simboli esterni impiegati, abbiamo le invocazioni delle funzioni di libreria (esempio: printf()) esterna, cioè non definita all’interno del codice sorgente. Il risultato è un file eseguibile, cioè un file in cui oltre al codice c’è informazione riguardo alla posizione in memoria in cui va caricato il programma, nonché eventuali simboli ancora non “risolti”.

Anche se si utilizza il compilatore solo nella prima “fase”, spesso con il termine compilazione si indica l’intero processo di traduzione da linguaggio ad alto livello a linguaggio macchina.

Per l’esecuzione, il Loader, che è un componente del sistema operativo, carica il programma in memoria (gerarchia) e poi passa il controllo della CPU alla prima istruzione del programma (in sistemi con con librerie dinamiche come Windows, invoca il linker dinamico per risolvere i simboli mancanti). Effettua altre procedure in altri meccanimi più complessi. Nel momento dell’effettiva esecuzione della prima istruzione del programma, c’è una locazione di memoria speciale, memorizzato in un registro della CPU, che contiene un indirizzo di memoria molto utile, cioè la prossima istruzione in linguaggio macchina da eseguire.

Il linguaggio assembly svolge un ruolo fondamentale nella programmazione dei computer, specialmente in contesti dove è necessaria un'ottimizzazione estrema del codice o una comprensione dettagliata dell'architettura del computer. Benché sia più complesso da scrivere e comprendere rispetto ai linguaggi di alto livello, il linguaggio assembly rimane uno strumento potente nelle mani degli sviluppatori, consentendo loro di raggiungere livelli di controllo e prestazioni che altrimenti sarebbero difficili da ottenere.