Settimo Capitolo

Il livello del linguaggio assemblativo

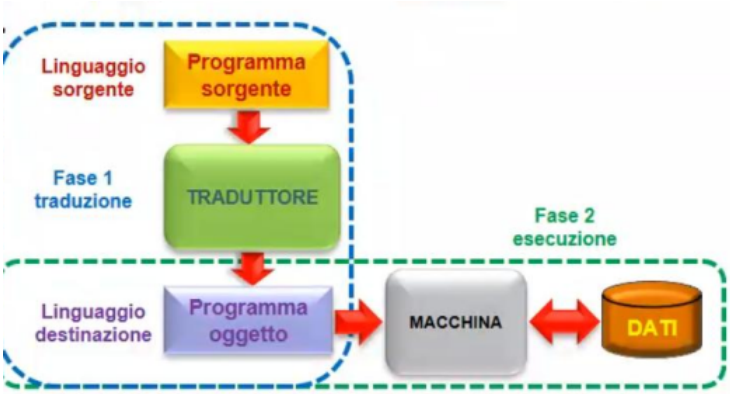
> Il **livello del linguaggio assemblativo** pone una differenza importante rispetto ai suoi livelli sottostanti: il livello assemblativo è implementato mediante **traduzione**, mentre i livelli visti finora sono *interpretati.*

Principalmente, osservando l'immagine, il livello del linguaggio assemblativo si occupa della prima fase: dato un **programma sorgente,** che può essere scritto in un linguaggio ad alto livello, come Python o Java, oppure in una rappresentazione simbolica di un linguaggio macchina numerico (qui penso intendiamo per Assembly), viene **tradotto** in un **programma oggetto equivalente,** il quale può essere eseguito dai livelli di microarchitettura, ISA e il livello macchina del Sistema Operativo. Il programma oggetto non è altro che un programma eseguibile scritto in binario.

La **traduzione** viene realizzata in due passi distinti:

- generazione di un programma equivalente nel linguaggio destinazione;
- esecuzione del programma appena generato.

Differentemente avviene nell'interpretazione, nella quale esiste un unico passo, ovvero l'esecuzione del programma sorgente originale.



Introduzione al linguaggio assemblativo

> Se il linguaggio sorgente è un linguaggio ad alto livello → Traduttore = **Compilatore;**
>
> Se il linguaggio sorgente è un linguaggio scritto in una rappresentazione simbolica di un linguaggio macchina numerico → Traduttore = **Assemblatore.**

Cos'è un linguaggio assemblativo

> Un linguaggio assemblativo è un linguaggio nel quale **ciascuna istruzione produce esattamente un'istruzione macchina,** quindi avremmo una corrispondenza uno-a-uno tra le istruzioni macchina e le istruzioni del programma assemblativo.

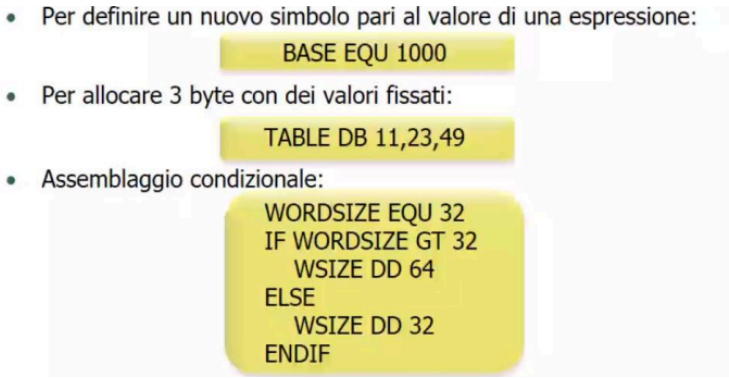
L'utilizzo del linguaggio assemblativo è dovuto alla sua semplicità rispetto ad utilizzare un linguaggio macchina; inoltre, il programmatore in linguaggio assemblativo ha accesso a tutte le funzionalità e a tutte le istruzioni disponibili nella macchina di destinazione, differentemente per il programmatore in linguaggio ad alto livello, il quale non ha le stesse libertà di accesso.

Oltre a ciò, un programma in linguaggio assemblativo è più piccolo e veloce rispetto ad un programma in un linguaggio ad alto livello, pertanto possiamo constatare che sia migliore a quest'ultimo in termini di prestazioni.

Pseudocodici

> I comandi che l'assemblatore utilizza nel codice sono dette **pseudoistruzioni** o **direttive dell'assemblatore.**

Ad esempio:



Le macroistruzioni

> Una definizione di **macro** è un modo per assegnare un nome a una porzione di testo. Dopo che una macro è stata definita, il programmatore può scriverne il nome al posto della porzione di programma corrispondente.

Le macro nascono con lo scopo di ripetere linee di codice all'interno del programma, da **non confondere** con le **procedure,** spiegheremo poi la motivazione, intanto basta pensare che siano più efficienti rispetto a quest'ultime.

Esempio di macro

```
#definisco la macro
SWAP MACRO
   MOV EAX, P
   MOV EBX, Q
   MOV Q, EAX
   MOV P, EBX
   ENDM

#richiamo due volte la macro
SWAP
SWAP
```

```
#codice equivalente
MOV EAX, P
MOV EBX, Q
MOV Q, EAX
MOV P, EBX

MOV EAX, P
MOV EBX, Q
MOV Q, EAX
MOV P, EBX
```

Quando l'assemblatore incontra una definizione di macro, la salva nella *tabella delle definizioni di macro* per poterla utilizzare successivamente. Da quel momento, ogni volta che il nome della macro appare come codice operativo, l'assemblatore la **sostituisce con il corpo della macro**. Da tenere in considerazione che l'espansione della macro avviene durante il **processo assemblativo, e non durante l'esecuzione del programma.**

Abbiamo accennato in precedenza che le macroistruzioni si differenziano dalle procedure. La differenza cruciale è che una chiamata di macro è un'istruzione diretta all'assemblatore, il quale si occuperà di sostituire il nome con il suo corpo; la chiamata di procedura è un'istruzione macchina inserita nel programma oggetto e che sarà eseguità in un secondo momento per chiamare la procedura.

La tabella indica le differenze fra macro e procedura:

quesito	**macro**	**procedura**
Quando viene effettuata la chiamata?	durante l'assemblaggio	durante l'esecuzione
Il corpo è inserito nel programma oggetto in ogni punto in cui avviene una chiamata?	si	no
L'istruzione per la chiamata di procedura è inserita nel programma oggetto ed eseguita successivamente?	no	si
Occorre usare un'istruzione di ritorno dopo aver effettuato una chiamata?	no	si
Quante copie del corpo appaiono nel programma oggetto?	una per ogni chiamata di macro	una

Processo di assemblaggio

Un programma può contenere istruzioni che possono avere dei "salti" in avanti. In altre parole, potremmo avere, ad esempio, righe di codici che possono essere state definite successivamente. Il problema appena descritto è chiamato **problema dei riferimenti in avanti.**

Esistono due soluzioni:

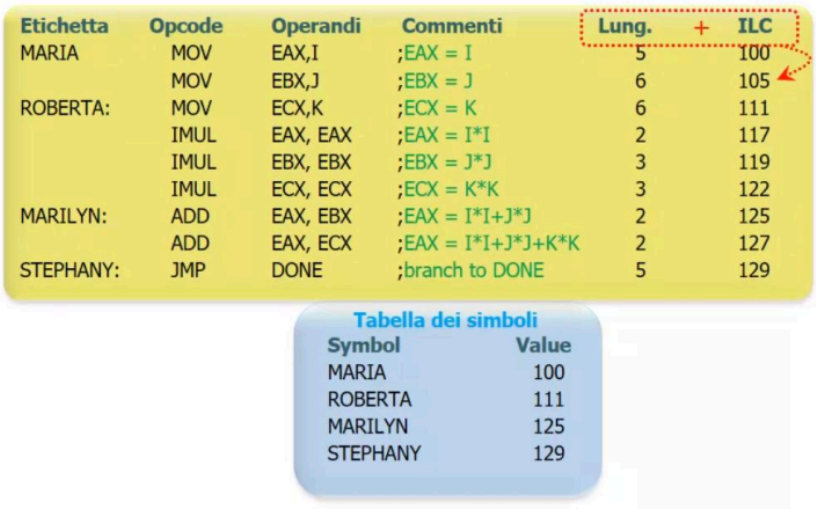
1. L'assemblatore legge il programma sorgente due volte; durante la prima lettura, l'assemblatore raccoglie le definizione dei simboli all'interno di una tabella. Prima della seconda lettura, si conoscono tutti i valori di tutti i simboli, in modo tale da non rimanere alcun riferimento in avanti ed è possibile leggere ogni istruzione, assemblarla e generarla;
2. Lettura del file di ingresso una sola volta, convertirlo in un formato intermedio e memorizzarla in una tabella. Successivamente, viene effettuata una seconda passata sulla tabella, invece che sul codice sorgente.

Primo passaggio

> La principale funzione della prima passata è quella di costruire la **tabella dei simboli**, contenente i valori di tutti i simboli. Un **simbolo** è un'etichetta o un valore al quale è stato assegnato un nome simbolico attraverso una pseudoistruzione.

Durante questa fase, l'assemblatore mantiene una variabile, chiamata **ILC** (*Instruction Location Counter),* per tenere traccia dell'indirizzo che l'istruzione che sta assemblando avrà a tempo di esecuzione. Ogni volta che viene elaborata un'istruzione, la variabile viene incrementata della sua lunghezza.

Nella maggior parte degli assemblatori la prima passata utilizza tre tabelle interne per i simboli, le pseudoistruzioni e i codici operativi. I simboli vengono definiti quando sono utilizzati come etichette oppure mediante una definizione esplicita.



Etichetta	Opcode	Operandi	Commenti	Lung.	+	ILC
MARIA	MOV	EAX,I	;EAX = I	5		100
	MOV	EBX,J	;EBX = J	6		105
ROBERTA:	MOV	ECX,K	;ECX = K	6		111
	IMUL	EAX, EAX	;EAX = I*I	2		117
	IMUL	EBX, EBX	;EBX = J*J	3		119
	IMUL	ECX, ECX	;ECX = K*K	3		122
MARILYN:	ADD	EAX, EBX	;EAX = I*I+J*J	2		125
	ADD	EAX, ECX	;EAX = I*I+J*J+K*K	2		127
STEPHANY:	JMP	DONE	;branch to DONE	5		129

Tabella dei simboli

Symbol	Value
MARIA	100
ROBERTA	111
MARILYN	125
STEPHANY	129

Seconda passata

> L'obiettivo della seconda passata è la **generazione del programma oggetto** e l'eventuale stampa del listato.

Le operazioni compiute dalla seconda passata sono più o meno simili a quelle di prima: le linee vengono lette ed elaborate una alla volta. Dato che all'inizio di ogni linea abbiamo scritto (su file temporaneo) il tipo, il codice operativo e la lunghezza, queste informazioni vengono lette in modo da risparmiare parte della fase di analisi dell'input.

Tabella dei simboli

Abbiamo visto che nella prima passata viene realizzata la **tabella dei simboli,** una tabella in cui sono presenti informazioni inerenti ai simboli e ai loro valori. Vedremo in questo paragrafo *come organizzarla.* In ogni caso, si cerca di simulare una memoria associativa, che non è altro che un insieme di coppie (simbolo, valore), nelle quali il simbolo funge da chiave.

Prima tecnica: ricerca lineare

Quando si vuole recuperare un simbolo, la routine della tabella dei simboli effettua semplicemente una ricerca lineare all'interno della tabella finché non trova l'elemento desiderato.

Nonostante sia corretto, il costo di esecuzione è O(n). Pertanto, tale ricerca è lenta, poiché, in media, andremo ad esaminare metà della tabella

Seconda tecnica: ricerca dicotomica

La ricerca dicotomica è un algoritmo ricorsivo e richiede che la tabella sia ordinata (supponiamo alfabeticamente in base al simbolo). Funziona come segue: partendo dal simbolo centrale, lo confronta col simbolo che stiamo cercando. Se il simbolo precede alfabeticamente l'elemento centrale della tabella, richiamiamo l'algoritmo di ricerca nella prima metà della tabella, caso contrario nella seconda metà.

Nonostante il costo di esecuzione sia inferiore alla ricerca lineare (O(logn)), dobbiamo tenere conto del fatto che la tabella deve essere ordinata: riordinare una tabella avrà un costo di esecuzione, nel migliore dei casi, di O(nlogn), il ché non è vantaggioso.

Terza tecnica: utilizzo della codifica hash

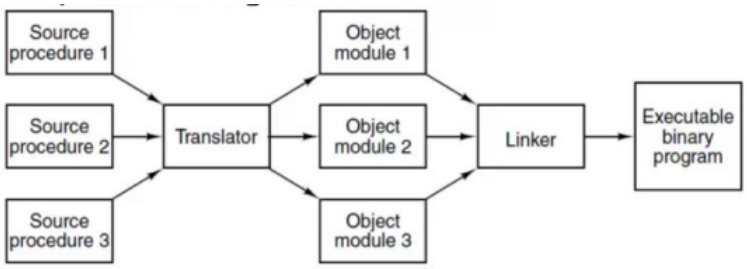
In questo caso, andiamo a definire una **funzione hash** che mappa i simboli nell'intervallo di interi compreso tra 0 e k-1. L'utilizzo della codifica hash è vantaggiosa, in quanto avremmo un costo computazionale pari a O(1), ma avremmo problematiche di collisione.

Linker e Loader

La maggior parte dei programmi è composta da più procedure. Generalmente, i compilatori e gli assemblatori traducono una procedura alla volta e memorizzano su disco il risultato della traduzione. Prima che il programma possa essere eseguito, è necessario recuperare tutte le procedure e collegarle fra loro in modo appropriato; inoltre, in assenza di memoria virtuale, occorre caricare in memoria centrare il programma ottenuto dal collegamento delle procedure, tramite il **loader**. Il programma che esegue questi passi è chiamato **linker**.

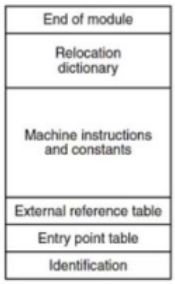
La traduzione completa avviene in due passi distinti:

- compilazione o assemblaggio dei file sorgente; eseguito dal traduttore
- collegamento dei moduli oggetto; eseguito dal linker



> La funzione del linker è quella di unire le procedure tradotte e di collegarle tra loro in modo da poterle eseguire come un'unica unità chiamata **programma eseguibile binario.**

Struttura di un modulo oggetto



Dal basso verso l'alto:

1. *Identificativo* → Contiene il nome del modulo e informazioni necessarie al linker;
2. *Tabella dei punti di ingresso* → Insieme dei punti di ingresso a cui possono fare riferimento altri moduli;
3. *Tabella dei riferimenti esterni* → Lista dei riferimenti utilizzati all'esterno del modulo.
4. *Istruzioni macchina e costanti* → Troviamo il codice assemblato e le costanti. Sarà l'unica parte che verrà caricata in memoria al momento dell'esecuzione;
5. *Dizionario di rilocazione* → Fornisce gli indirizzi che dovranno essere rilocati;
6. *Fine del modulo* → Troviamo l'identificativo di fine modulo, l'indirizzo ove iniziare l'esecuzione, un eventuale checksum per rilevare gli errori che possono avvenire durante la lettura del modulo.