**Introduzione**

Il livello del linguaggio assemblativo è implementato mediante traduzione invece che interpretazione. Nella traduzione il programma originale non viene eseguito direttamente, ma viene invece convertito in un programma equivalente chiamato programma oggetto la cui esecuzione è portata avanti solo dopo che la traduzione è stata completata.

Mentre si esegue il programma oggetto sono coinvolti i livelli di micro e macro architettura ed il livello macchina del sistema operativo.

**Introduzione al linguaggio assemblativo**

**Che cos’è un linguaggio assemblativo**

Un puro linguaggio assemblativo è un linguaggio nel quale ciascuna istruzione produce esattamente un’istruzione macchina. In altre parole esiste una corrispondenza uno-a-uno tra le istruzioni macchina e le istruzioni del programma assemblativo.

La ragione dell’utilizzo del linguaggio assemblativo al posto del linguaggio macchina (in esadecimale) è che è molto più facile programmare utilizzando il linguaggio assemblativo. L’uso di una forma simbolica per i nomi e gli indirizzi, al posto della rappresentazione binaria oppure ottale, costituisce un’enorme differenza. La maggior parte dei programmatori è in grado di ricordare che ADD, SUB, MUL e DIV sono le abbreviazioni di aggiungi, sottrai, moltiplica e dividi, mentre pochi riescono a ricordarsi i corrispondenti valori numerici utilizzati dalla macchina.

Il linguaggio assemblativo gode anche di un’altra proprietà che lo contraddistingue dai linguaggi ad alto livello. Il programmatore in linguaggio assemblativo ha accesso a tutte le funzionalità e a tutte le istruzioni disponibili nella macchina di destinazione, mentre il programmatore in linguaggio ad alto livello non ha la stessa libertà di accesso (per esempio non può accedere ai registri).

Un’ultima differenza è che i programmi in linguaggio assemblativo vengono scritti per una specifica famiglia di macchine, mentre un programma scritto in un linguaggio ad alto livello può essere eseguito su molte macchine diverse.

**Perché usare il linguaggio assemblativo**

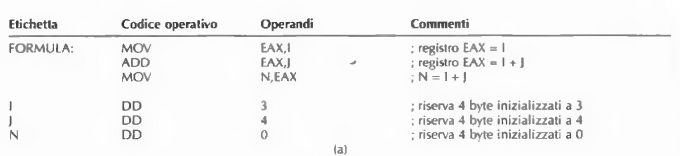
Programmare in linguaggio assemblativo è difficile. Ma allora, perché mai si dovrebbe programmare in linguaggio assemblativo? Esistono due motivi: le prestazioni e le possibilità di accesso alla macchina. Per alcune categorie di applicazioni ove servono ridotte dimensioni del programma, velocità e completo sfruttamento della macchina non esiste alternativa.

**Formato delle istruzioni del linguaggio assemblativo**

La struttura delle istruzioni di un linguaggio assemblativo rispecchia la struttura delle istruzioni macchina, delle quali le prime forniscono una rappresentazione simbolica.

La Figura mostra alcuni frammenti del linguaggio assemblativo che eseguono l’istruzione

N = I + J eseguito su architettura x86.



Esistono differenti assemblatori per la famiglia Intel, ciascuno dei quali ha una propria sintassi. Noi analizzeremo il linguaggio assemblativo Microsoft MASM x86.

Le istruzioni del linguaggio assemblativo sono composte da quattro parti: un campo etichetta, un campo per l’operazione (codice operativo), un campo per gli operandi e un campo per i commenti.

Le etichette sono utilizzate per fornire nomi simbolici agli indirizzi, e sono necessarie per definire le destinazioni alle quali portano le istruzioni che effettuano salti e per poter accedere alle parole di dati memorizzate mediante nomi simbolici. Se un’istruzione è etichettata, l’etichetta inizia (di solito) nella colonna 1.

Il campo Codice operativo contiene un’abbreviazione simbolica del codice operativo oppure un comando per l’assemblatore stesso.

I programmi hanno bisogno di riservare spazio per le variabili: i progettisti del linguaggio MASM hanno scelto di usare DD (Define Double), perché nell’8088 la parola era di 16 bit.

Nelle istruzioni del linguaggio assemblativo il campo Operandi viene utilizzato per specificare gli indirizzi e i registri utilizzati come operandi dall’istruzione macchina.

Nel caso di un’istruzione per la somma di interi il campo operandi indica quali elementi devono essere sommati fra loro.

Nel caso di un’istruzione di salto indica invece a quale indirizzo deve portare la diramazione.

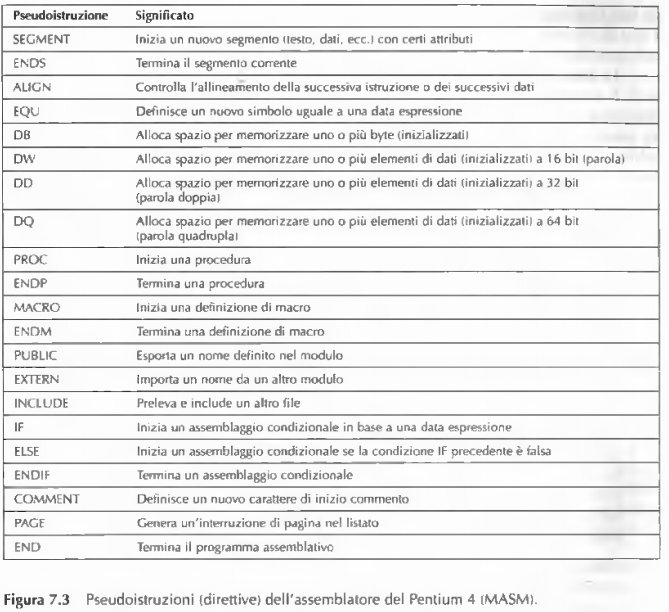
Gli operandi possono essere registri, costanti, locazioni di memoria, e così via.

II campo Commenti è il luogo in cui i programmatori possono inserire utili spiegazioni sul funzionamento del programma a beneficio di altri programmatori che in seguito lo potrebbero utilizzare o modificare.

**Pseudoistruzioni**

Un linguaggio assemblativo può anche contenere dei comandi indirizzati all’assemblatore. Questi comandi sono chiamati pseudoistruzioni o anche direttive dell’assemblatore.

Abbiamo già visto un esempio (“DD”) nella figura precedente. La Figura 7.3 elenca alcune pseudoistruzioni dell’assemblatore MASM x86.



Esempi:

- *BASE EQU 1000*

BASE può essere usato in qualsiasi punto al posto di 1000.

- *TABLE DB 11,23,49*  
Alloca spazio per 3 byte e li inizializza rispettivamente ai valore 11, 23, 49.

*- WORDSIZE EQU 32*

*IF WORDSIZE GT 32 ;la parola di indirizzo simbolico WSIZE è inizializzata a 64 oppure a 32 in*

*WSIZE: DD 64 ;base al valore di WORDSIZE*

*ELSE*

*WSIZE: DD 32*

*ENDIF*

Assemblaggio condizionale.

**Macroistruzioni**

Spesso i programmatori di linguaggi assemblativi hanno bisogno di ripetere più volte all’interno di un programma alcune sequenze di istruzioni. Si trasforma la sequenza di istruzioni in una procedura e viene richiamata quando è richiesta.

**Definizione, chiamata ed espansione di macro**

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.

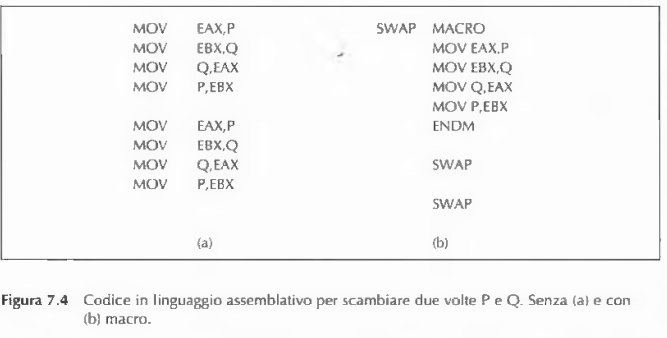
La Figura 7.4(a) mostra un programma in linguaggio assemblativo per x86 che scambia due volte il contenuto delle variabili *p* e *q*.

Anche se assemblatori diversi utilizzano notazioni leggermente differenti, in tutti i casi una definizione di macro è composta dalle seguenti parti principali:

1) un’intestazione che indica il nome della macro che si sta definendo;

2) il testo che costituisce il corpo della macro;

3) una pseudoistruzione che segna la fine della definizione (per esempio, ENDM).



Quando l’assemblatore incontra una definizione di macro, la salva nella tabella delle definizioni di macro per poterla utilizzare successivamente. Da quel momento in poi ogni volta che il nome della macro appare come codice operativo, l’assemblatore la sostituisce con il corpo della macro.

L’uso del nome di una macro come codice operativo viene detto chiamata di macro, mentre la sua sostituzione con il corpo della macro è chiamata espansione di macro.

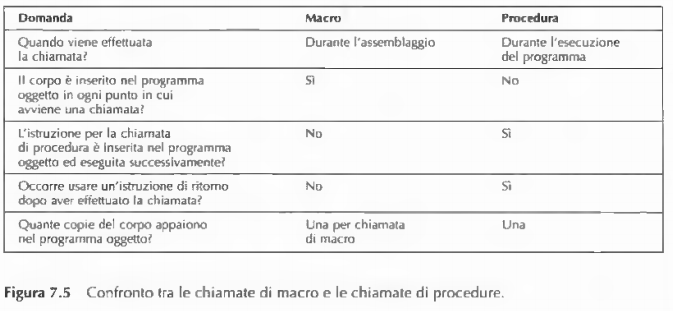
L’espansione della macro viene effettuata durante il processo assemblativo e non durante l’esecuzione del programma. Questo è un punto importante. Il programma della Figura 7.4(a) e quello della Figura 7.4(b) produrranno esattamente lo stesso codice in linguaggio macchina.

**Macro VS procedure**

Le chiamate di macro non devono essere confuse con le chiamate di procedure.

La differenza principale è che una chiamata di macro è un’istruzione diretta all’assemblatore, affinché sostituisca il nome della macro con il suo corpo. Una chiamata di procedura è invece un’istruzione macchina inserita nel programma oggetto e che sarà eseguita in un secondo momento per chiamare la procedura.

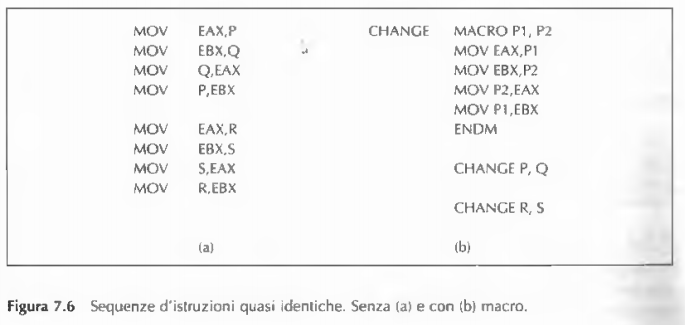
La Figura 7.5 confronta le chiamate di macro con le chiamate di procedura. Quindi, in pratica, nell’esecuzione di codice con delle macro non viene effettuato alcun salto.



**Macro con parametri**

La definizione di una macro può includere dei parametri formali che verranno poi sostituiti dai corrispondenti valori attuali. I parametri attuali sono specificati nel campo operandi della chiamata di macro.

La Figura 7.6(b) mostra il programma della Figura 7.6(a) riscritto utilizzando una macro con due parametri. P1 e P2 sono i parametri formali. Nel momento in cui la macro viene espansa, ogni occorrenza di P1 nel corpo della macro viene sostituita dal secondo parametro attuale. Nella chiamata di macro “*CHANGE P, Q”* P è il primo parametro attuale e Q è il secondo.



**Processo di assemblaggio**

Nei paragrafi successivi descriveremo brevemente come funziona un assemblatore. Anche se ogni macchina ha il proprio linguaggio assemblativo, i diversi processi di assemblaggio sono tra loro sufficientemente simili da poterci permettere una descrizione generale.

**Assemblatori a due passate**

Dato che ogni programma in linguaggio assemblativo consiste in una serie di istruzioni che possono avere dei “salti” in avanti, l’assemblatore non può conoscere in anticipo la posizione dell’istruzione richiamata (problema del riferimento in avanti).

È possibile gestire i riferimenti in avanti in due modi:

- l’assemblatore può leggere il programma sorgente due volte. Ciascuna lettura del programma sorgente è chiamata passata. Durante la prima lettura questi assemblatori raccolgono in una tabella tutte le definizioni dei simboli, comprese le etichette delle istruzioni. Prima di iniziare la seconda passata si conoscono quindi i valori di tutti i simboli. In tal modo non rimane alcun riferimento in avanti ed è possibile leggere ogni istruzione, assemblarla e generare il codice corrispondente.

- leggere il programma sorgente una sola volta, convertirlo in un formato intermedio e memorizzare in una tabella della memoria questa forma intermedia. In seguito viene effettuata una seconda passata sulla tabella.

In entrambi gli approcci, la prima passata ha anche il compito di salvare tutte le definizioni di macro e di espanderle quando si incontrano le loro chiamate.

**Prima passata**

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

Per assegnare nel campo etichetta di un’istruzione un valore a un simbolo, l’assemblatore deve conoscere quale indirizzo avrà tale istruzione durante l’esecuzione del programma. Durante il processo di assemblaggio l’assemblatore mantiene una variabile, chiamata ILC (Instruction Location Counter, “contatore di locazioni delle istruzioni”), per tener traccia dell’indirizzo che l’istruzione che sta assemblando avrà a tempo di esecuzione.

Nella maggior parte degli assemblatori la prima passata utilizza tre tabelle interne per i simboli, le pseudoistruzioni e i codici operativi.

**Seconda passata**

La funzione della seconda passata è la generazione del programma oggetto.

Oltre a ciò la seconda passata deve generare delle informazioni richieste dal linker per collegare in un unico file eseguibile le procedure assemblate in momenti distinti.

**Tabella dei simboli**

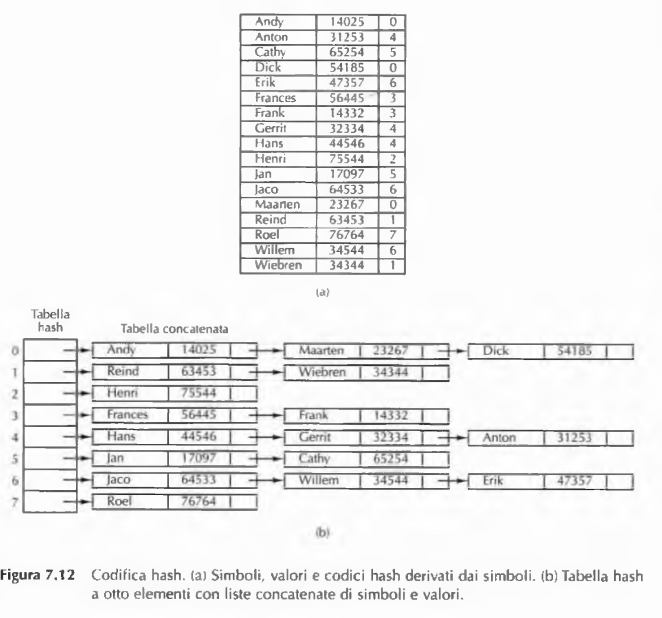
Durante la prima passata l’assemblatore accumula informazioni sui simboli e i loro valori. Tali informazioni devono essere memorizzate nella tabella dei simboli per essere utilizzate durante la seconda passata. Può essere descritta come una tabella associativa, costituita da un insieme di coppie <simbolo, valore> accessibili tramite il simbolo.

Esistono varie tecniche per realizzarla:

- implementare la tabella dei simboli come un vettore di coppie, in cui il primo elemento è il simbolo (oppure un puntatore a esso) e il secondo è il valore (oppure un puntatore a esso). Quando si vuole recuperare un simbolo, la routine della tabella dei simboli effettua semplicemente una ricerca lineare all’interno dell’array finché non trova l’elemento desiderato. Questo metodo è facile da programmare, ma allo stesso tempo è lento, dato che per ciascuna ricerca occorre esaminare, in media, metà della tabella.

- ordinare la tabella rispetto ai simboli e utilizzare un algoritmo di ricerca dicotomica per cercare il simbolo desiderato. Questa tecnica richiede che sia conservato l’ordine degli elementi della tabella, e ciò è costoso dal punto di vista computazionale. (O(log n)).

- usare una codifica che mappa i simboli nell’intervallo di interi compreso tra 0 e k-1. Questa tecnica richiede però che la funzione hash fornisca una distribuzione uniforme dei valori hash e la gestione delle collisioni. (O(1)). La codifica hash è illustrata nella Figura 7.12.



**Collegamento e caricamento**

La maggior parte dei programmi è composta da più di una procedura. 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 centrale il programma ottenuto dal collegamento delle procedure. I programmi che eseguono questi passi sono chiamati in vari modi, tra cui linker, linking loader e linkage editor.

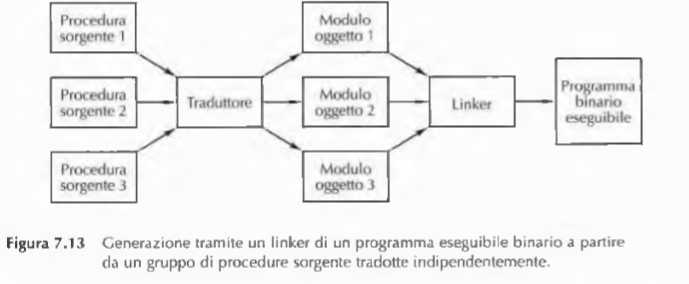
La traduzione completa di un programma sorgente richiede due passi distinti, com’è mostrato nella Figura 7.13:

1) compilazione o assemblaggio delle procedure sorgenti;

2) collegamento dei moduli oggetto.

Il primo passo viene eseguito dal compilatore o dall’assemblatore, e il secondo dal linker.

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



Esiste un buon motivo per cui i compilatori e gli assemblatori traducono ciascuna procedura sorgente come un’entità separata. Se un compilatore, o un assemblatore, dovesse leggere una serie di procedure sorgente e produrre direttamente un programma in linguaggio macchina pronto per essere eseguito, allora la modifica anche solo di un’istruzione in una delle procedure sorgente richiederebbe una nuova traduzione di tutte le altre procedure. Se si usa la tecnica dei moduli oggetto separati (mostrata nella Figura 7.13) è necessario ritradurre soltanto la procedura modificata e non quelle rimaste invariate.

**Compiti del linker**

L’idea è di creare all’interno del linker un’immagine esatta dello spazio di indirizzamento virtuale del programma eseguibile e di posizionare tutti i moduli oggetto nelle locazioni corrette.

Il linker unisce gli spazi degli indirizzi separati dei diversi moduli oggetto all’interno di un unico spazio lineare degli indirizzi seguendo questi quattro passi:

1) costruisce una tabella di tutti i moduli oggetto e delle loro lunghezze;

2) in base a questa tabella assegna un indirizzo di partenza per ciascun modulo;

3) cerca tutte le istruzioni che fanno riferimento alla memoria e aggiunge a ciascuna di loro una costante di rilocazione uguale all’indirizzo di partenza del suo modulo;

4) cerca tutte le istruzioni che fanno riferimento ad altre procedure e inserisce in quei punti gli indirizzi delle procedure corrispondenti.