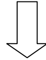
TRADUTTORI

Linguaggio sorgente



Traduzione

Linguaggio target

La traduzione è necessaria perché esiste un processore (h/w o interprete) per il linguaggio target, ma non per il linguaggio sorgente.

Passaggio di livello

Input e output non sono programmi per la stessa macchina virtuale.

L'esecuzione del programma tradotto nel linguaggio target dà lo stesso risultato che avrei ottenuto eseguendo il programma in linguaggio sorgente se fosse esistito un processore per quel linguaggio.

TRADUZIONE

Il programma nel linguaggio sorgente non è eseguito.

È completamente convertito in un programma equivalente in linguaggio target, programma oggetto o programma eseguibile ISA-Level (binario).

A questo punto il programma tradotto è eseguito.

Del sorgente non si fa alcun uso.

Solo il programma tradotto è caricato in memoria.

INTERPRETAZIONE

Un unico passo:

1. l'interprete esamina una istruzione per volta;
2. esegue l'istruzione equivalente (o le istruzioni equivalenti) in linguaggio ISA-Level;
3. l'interprete è caricato in memoria;
4. il programma sorgente è l'insieme dei dati su cui l'interprete lavora.

Non è generato alcun programma equivalente.
A volte, il sorgente può essere convertito in una forma intermedia (esempio Java byte code), per una più semplice interpretazione.

TRADUTTORI

1. Assembler (assemblatore)
   - il linguaggio sorgente (assembly language) è una rappresentazione simbolica del linguaggio ISA-Level (macchina).
2. Compilatore
   - il linguaggio sorgente è di alto livello (Java, C), e il linguaggio target è il linguaggio ISA_Level (macchina) o una sua rappresentazione simbolica.

ASSEMBLY LANGUAGE

Ogni statement corrisponde esattamente ad una istruzione ISA-Level.
Nomi e indirizzi simbolici, e non mi devo ricordare le codifiche esadecimali d'istruzioni e indirizzi.
L'assembler tradurrà i nomi simbolici nei loro valori esadecimali (codici macchina o indirizzi).

Svantaggi

Nessuna portabilità: il programma funziona solo su macchine della stessa famiglia.
Complicazione: programmare in assembly è:

- difficile (più probabile commettere errori);
- lungo (occorre molto più tempo per scrivere

Vantaggi

Accesso a tutte le funzionalità e a tutte le istruzioni della macchina target:
test di un bit di overflow, accesso a un registro, … tutto ciò che può essere fatto in ISA-Level può essere fatto in assembly;
parecchie funzionalità della macchina non sono visibili quando si programma usando un linguaggio di alto livello.
Migliori prestazioni.
Codice più piccolo e tempo di esecuzione minore.
Applicazioni "embedded" in telefoni cellulari, device driver, smart card.

APPROCCIO MISTO

Prima sviluppo del programma in codice di alto livello (meno tempo-uomo necessario, punto di vista globale sul problema).
Misure per individuare le parti più critiche:
esempio il 10% del programma richiede il 90% del tempo di esecuzione;
riscrittura delle parti critiche in assembler per migliorare le prestazioni (***tuning***).
Il rapporto costo/beneficio di questo approccio è favorevole:
costo: tempo-uomo nello sviluppo;
beneficio: riduzione nel tempo di esecuzione.

MOTIVI PER STUDIARE ASSEMBLY

Abilità di ridurre di 2 o 3 volte il tempo di esecuzione di alcune routine critiche è spesso fondamentale in grossi progetti.
Utile quando la memoria disponibile è poca: esempio applicazioni in smart card che contiene una CPU, ha meno di un 1 Mbyte di memoria (per ragioni di costi), e tuttavia bisogna eseguirvi algoritmi complessi, necessità di codice efficiente.
Un compilatore produce un output usato da un assembler, o esegue anche il processo di assemblaggio: utile lo studio per comprendere i compilatori.
Visione diretta sul funzionamento h/w della macchina.
Linguaggio che consente di comprendere meglio il funzionamento dell’architettura.

LE MACRO

Le macro, le routine e le librerie permettono al programmatore di disporre di codice già corretto e funzionante.
In questo modo, lo sviluppo di un programma procede più rapidamente, poiché il programmatore deve progettare e collaudare solo le parti effettivamente nuove, senza dover ogni volta ripartire da zero.
Le routine sono sezioni di codice che possono essere attivate, con istruzioni di chiamata (e di ritorno), da punti diversi del programma.
Ogni volta che una routine è chiamata, ha inizio l'esecuzione delle istruzioni che la compone ed, al termine, il controllo ritorna al programma chiamante.

Spesso bisogna ripetere sequenze d'istruzioni parecchie volte in un programma: scrivere le istruzioni richieste quando sono necessarie.
Se una sequenza è lunga, però, o deve essere usata molte volte, il riscriverle ripetutamente …
Soluzione: mettere la sequenza in una routine e chiamarla quando è necessario.
Svantaggio: richiede un'istruzione di chiamata ed un'istruzione di ritorno.
Se le sequenze sono brevi, ma usate frequentemente il tempo dovuto alla chiamata della routine può rallentare in maniera significativa il programma.
Macro
Soluzione semplice ed efficiente al problema di una richiesta ripetuta delle stesse (o quasi) sequenze d'istruzioni.

DEFINIZIONE, CHIAMATA ED ESPANSIONE DELLA MACRO

È un metodo per imporre un nome ad una parte di codice.
Dopo aver definito una macro, il programmatore può scrivere il nome della macro invece della parte di codice.
È, in effetti, un'abbreviazione di una parte di codice.
Può essere definita in un qualsiasi punto del programma, è sufficiente che la definizione preceda il primo richiamo che a lei è fatto.

```
conv_ascii    MACRO         ;input in AL output in AL
              AND AL,0FH    ;mascheramento bit 4..7
              ADD AL,30H    ;conversione ASCII 0..9
              CMP AL,39H    ;if (AL<=9) CONT1
              JBE CONT1
              ADD AL,7H     ;conversione ASCII A..F
       CONT1:
              ENDM
```


Per definire una macro occorrono le seguenti parti:

- un'intestazione della macro che impone il nome della macro in definizione;
- il testo comprendente il corpo della macro;
- una pseudoistruzione che marchi la fine della definizione.

Quando l'assembler incontra una definizione di macro la salva in una tabella di definizioni di macro per un uso successivo.
Da questo punto in poi, quando il nome della macro appare come codice operativo, l'assembler lo rimpiazza con il corpo della macro.
L'uso di un nome di macro come codice operativo è noto come ***chiamata*** della macro e la sua sostituzione con il corpo della macro è chiamata ***espansione*** della macro.

L’espansione della macro avviene durante il processo di assembling e non durante l’esecuzione del programma.
Guardando il programma ISA-Level è impossibile dire se nella sua generazione erano coinvolte o meno delle macro, poiché esse sono scartate quando il programma è assemblato.
La differenza è che una chiamata di macro è un segnale all’assembler perché sostituisca il nome della macro con il suo corpo.
Una chiamata alla routine è un’istruzione macchina inserita in un programma oggetto che sarà eseguita più tardi per chiamare la routine.

Il vantaggio dell’uso della macro è che si risparmiano chiamate a routine che a loro volta comportano la messa di parametri sullo stack, la modifica di CS:IP ed il loro successivo ripristino.
Lo svantaggio consiste nel fatto che il programma assume dimensioni tanto maggiori quanto più è fatto uso di macro, specialmente se queste sono lunghe e complesse.
In pratica conviene definire brevi gruppi d'istruzioni come macro e da blocchi più lunghi di programma formare routine.

Argomento	**Chiamata a macro**	**a routine**
Quando si fa la chiamata?	Assemblaggio	Esecuzione
Il corpo è inserito nel OBJ ovunque appariva il nome?	Sì	No
Un'istruzione di chiamata di routine è inserita nel OBJ poi eseguita in un secondo tempo?	No	Sì
Si deve usare un'istruzione di ritorno per restituire il controllo all'istruzione che seguiva la chiamata?	No	Sì
Quante copie del corpo appaiono nel programma oggetto?	Una per ogni chiamata di macro	1

MACRO CON PARAMETRI

reg_xchg *MACRO* ***REG1****,* ***REG2***
MOV AX,REG1
MOV BX,REG2
MOV REG2,AX
MOV REG1,BX
ENDM

Un esempio di richiamo è il seguente:

reg_xchg ***SI****,* ***BP***

Problema

Macro con label, deve essere ***LOCAL***

ASSEMBLER

Non può semplicemente tradurre riga per riga in file sorgente:
Forward Reference Problem
Un simbolo (label che indica la riga cui saltare) è usato prima della sua definizione (la riga stessa è più avanti nel programma).
Soluzione.
Traduttore a due passate:
prima passata: costruzione di una tabella in cui sono collocati tutti i simboli, incluse le label;
seconda passata: ora tutte le label sono già note, il file può essere assemblato.
Conversione del file in una forma intermedia salvata in una tabella; la seconda passata non è più una lettura del sorgente, ma della tabella contenente il formato intermedio, occorre più memoria per memorizzare la tabella.

PRIMO PASSO

Lettura del file sorgente riga per riga.
Costruzione delle tabelle:
- la tabella delle pseudoistruzioni,
- la tabella degli opcode,
- la tabella delle costanti,
- la tabella dei simboli (***symbol table***), contenente i valori dei simboli.

Nell'assegnare un valore ad una label, l'assembler deve sapere quale indirizzo avrà quell'istruzione.
Si ferma quando trova l'istruzione di terminazione.
Per avere traccia di questo indirizzo, l'assembler mantiene una variabile durante l'assembling, nota come ***location counter***.

Questa variabile è messa a zero all'inizio del passo uno ed incrementata della lunghezza dell'istruzione per ogni istruzione elaborata.

Label	opcode	operandi	commento	L	LC
PIPPO:	**MOV AX,**	**I**	**;AX = I**	**5**	**100**
	MOV BX,	**J**	**;BX = J**	**6**	**105**
PLUTO:	**MOV CX,**	**K**	**;CX = K**	**6**	**111**
					117

SYMBOL TABLE

Symbol name	Address	Info
PIPPO	100	PUBLIC (EXTERN)
PLUTO	111	

TABELLA DEGLI OPCODE

Campi:
- opcode simbolico;
- due operandi;
- valore esadecimale dell'opcode;
- lunghezza istruzione (# byte);
- gruppo dell'opcode (a seconda del numero e del tipo di operandi).

TABELLA DEI SIMBOLI

1. Tre campi:
   - nome simbolo (o puntatore ad esso);
   - valore;
   - altre info (esempio lunghezza del tipo di dato associato a quel simbolo, se il simbolo è accessibile dall'esterno della procedura).
2. Memorizza il valore associato esplicitamente ad un simbolo all'interno del programma (esempio definizione di costante).
3. Memorizza l'indirizzo associato ad una etichetta (destinazione di una istruzione di salto).

SECONDO PASSO

Generazione del programma oggetto.
Stampa eventualmente il listing dell'assembling.
Inoltre deve emettere certe informazioni necessarie al linker per collegare routine assemblate in momenti diversi.
Di nuovo legge il file riga per riga.
Tipo di opcode e lunghezza delle istruzioni sono già note (dalla tabella, e magari anche da un file temporaneo).
Traduzione delle istruzioni in codice oggetto.
Rilevazione degli errori nel sorgente.
Generazione del file oggetto e suo salvataggio su disco.

MODULO OGGETTO

Struttura di un modulo oggetto:
1. Identificazione
2. Tabella dei simboli interni
3. Tabella dei simboli esterni
4. Istruzioni macchina e costanti
5. Dizionario di rilocazione
6. Fine del modulo

La <u>prima parte</u> contiene il nome del modulo, certe informazioni necessarie al linker, come le lunghezze delle varie parti del modulo oggetto e, a volte, la data dell'assembling.
La <u>seconda parte</u> è una lista dei simboli definiti nel modulo cui possono fare riferimento altri moduli, insieme con i loro valori; pseudoistruzioni (***PUBLIC***).
La <u>terza parte</u> consiste in una lista di simboli usati nel modulo, ma definiti in altri moduli, insieme con una lista delle istruzioni macchina che usano quei simboli. Il linker ha bisogno della seconda lista per inserire gli indirizzi giusti nelle istruzioni che usano i simboli esterni. Una routine può chiamarne altre, tradotte indipendentemente, dichiarando i nomi delle routine chiamate come esterne (***EXTERN***).

La quarta parte è il codice assemblato e le costanti. Questa parte del modulo oggetto è la sola che sarà caricata nella memoria per essere eseguita. Le altre cinque parti saranno usate dal linker e poi scartate prima che l'esecuzione cominci.
La quinta parte è il dizionario di rilocazione, alle istruzioni che contengono indirizzi di memoria si deve aggiungere una costante di rilocazione.
La sesta parte contiene un'indicazione di fine modulo, a volte un totale di controllo per rilevare errori fatti durante la lettura del modulo, e l'indirizzo da cui iniziare l'esecuzione.

ERRORI TIPICI

Uso di un simbolo che non è stato definito.
Duplicazione della definizione di un simbolo.
Opcode scritto in modo sbagliato.
Il numero di operandi dopo un opcode è sbagliato.
Un numero in ottale contiene un 8 o un 9.
Uso illegale dei registri.
Manca l'istruzione di terminazione (esempio manca .end-method).

RICERCA IN UNA TABELLA

L'organizzazione della symbol table, concettualmente, è un insieme di coppie (simbolo, valore), in pratica una ***memoria associativa*** che dato il simbolo, deve produrre il valore.
Il metodo d'implementazione più semplice è d'implementare la symbol table come un array di coppie, il primo elemento del quale è il simbolo ed il secondo è il valore (search, codifica hash).

RICERCA LINEARE

RICERCA BINARIA

Le righe sono ordinate (ad esempio, ordinamento alfabetico sul nome del simbolo).
Confronto il mio simbolo con la riga centrale.
Se è uguale, ho finito.
Se alfabeticamente il mio simbolo è prima della riga centrale, restringo la ricerca alla mezza tabella superiore.
Se viene dopo, restringo la ricerca alla mezza tabella inferiore.
Ripeto ricorsivamente.
Tempo medio di ricerca $\log_2 n$, se n è il numero di righe.
Occorre mantenere le righe ordinate.

ALGORITMO DI HASHING

Funzione di hashing che mappa ogni simbolo (n) ad un intero da 0 a k-1.
Esempio
Moltiplico i codici ASCII del simbolo fra loro e prendo il risultato modulo k.
Memorizzo una hash table con k righe; tutte le righe della tabella originaria che corrispondono allo stesso valore di k (esempio i) sono memorizzate in una lista, puntata dalla i-esima riga della hash table.

Dato il mio simbolo, calcolo l’output della funzione di hashing, accedo alla corrispondente riga della hash table, e scorro linearmente la lista da essa puntata.
Lunghezza media della lista n/k.
Posso ridurre l’occupazione di memoria da parte della tabella (k piccolo), o il tempo di lookup (k grande).

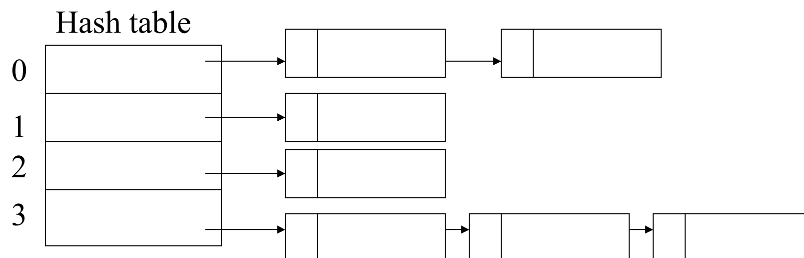
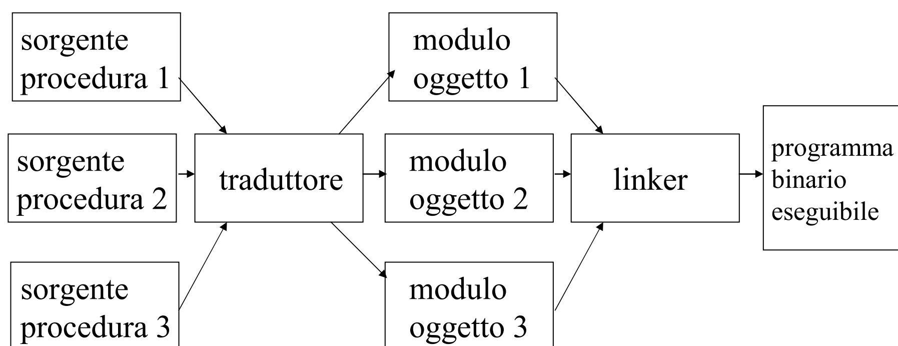


Tabella originaria: n = 7 Hash table: k = 4

LINKER

La maggior parte dei programmi sono costituiti da più di una procedura.
Il compilatore o l'assembler traducono una procedura per volta.
Tutti i moduli oggetto devono poi essere linkati (collegati) in modo appropriato.
Una modifica in un'unica procedura implica la traduzione di questa soltanto, ed un nuovo processo di linking (più rapido del processo di traduzione dell'intero codice).
Input e output si collocano allo stesso livello: sono programmi per la stessa macchina virtuale.
Creazione di un unico programma binario eseguibile.

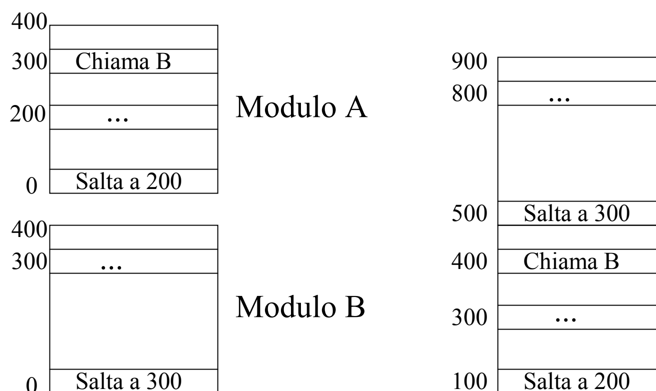
TRADUZIONE E LINKING



COMPITI DEL LINKER

All'inizio del passo uno del processo di assembling, l'assembler inizializza la variabile Location Counter a LC = 0.
Tutti gli altri indirizzi (comprese le locazioni cui saltare) sono calcolati a partire da questo indirizzo iniziale virtuale.
Questo è equivalente ad assumere che il modulo oggetto sarà collocato all'indirizzo zero (virtuale) durante l'esecuzione.
Lo stesso avviene in tutti i moduli.
Per eseguire il programma, il ***loader*** (caricatore) porta il modulo oggetto nella memoria centrale.

SITUAZIONE DOPO IL CARICAMENTO IN MEMORIA



L'indirizzo di partenza reale è 100 e non 0.
Perciò il salto all'interno del modulo A (e quello all'interno del modulo B) punta alla locazione errata.
Questo è il ***problema della rilocazione***, accade perché il modulo oggetto rappresenta uno spazio d'indirizzamento separato.
Analogamente va calcolato l'indirizzo cui saltare quando il modulo A chiama il modulo B, che non è noto perché ogni modulo ha un suo address space separato: ***problema del riferimento esterno***.
Entrambi questi problemi si possono risolvere con il linker.

COSTRUZIONE DI UN UNICO ADDRESS SPACE

Il linker fonde gli spazi d'indirizzamento separati dei moduli oggetto in un unico indirizzo lineare con i seguenti passi:

1. costruisce una tabella di tutti i moduli oggetto e delle loro lunghezze;
2. basandosi su questa tabella, assegna un indirizzo di caricamento ad ogni modulo oggetto;
3. trova tutte le istruzioni che contengono un indirizzo di memoria, a tutte aggiunge una ***costante di rilocazione*** uguale all'indirizzo di partenza del modulo in cui sono contenute;
4. trova tutte le istruzioni che fanno riferimento ad altre routine e v'inserisce l'indirizzo di queste routine.

LA MEMORIA DOPO LA COSTRUZIONE DELL'UNICO ADDRESS SPACE

900	
800	…
500	Salta a 800
400	Chiama 500
300	…
100	Salta a 300

Tabella dei moduli oggetto:

modulo	lunghezza	indirizzo di partenza
A	400	100
B	400	500

LINKING

Processo in due passi:

1. lettura dei moduli, costruzione della tabella dei moduli, costruzione di una tabella dei simboli globale contenente entry points e external references di tutti i moduli;
2. lettura moduli, rilocazione e linking di un modulo per volta.

BINDING TIME

Multiprogrammazione
Un programma può essere scaricato e ricaricato in memoria più volte prima di completare la sua esecuzione.
Non è detto che l'indirizzo iniziale sia ogni volta lo stesso.
I riferimenti agli indirizzi di memoria potrebbero perciò risultare errati.
Quando un programma è scritto, contiene nomi simbolici per indicare gli indirizzi di memoria.
Il momento in cui si determina l'effettivo indirizzo di memoria corrispondente al nome simbolico, è chiamato ***tempo di bind***.

Esistono almeno sei possibilità per il tempo di bind:

1. Quando si scrive un programma.
2. Quando si traduce il programma.
3. Quando si linka il programma, ma prima di caricarlo.
4. Quando si carica il programma.
5. Quando si carica un registro di base usato per l'indirizzamento.
6. Quando si esegue l'istruzione contenente l'indirizzo.

PIÙ PRECISAMENTE

Quando i simboli sono convertiti in indirizzi virtuali?
Quando gli indirizzi virtuali sono convertiti in indirizzi reali?
In realtà la costruzione di un unico address space corrisponde alla prima conversione (linking).
Il binding è completo solo quando è avvenuta anche la seconda conversione.

Il vero problema
Mappaggio indirizzo virtuale - indirizzo reale.
Per consentire il caricamento del programma in memoria in posizioni diverse, anche se il linking è già stato completato.
Tre metodi.

MAPPAGGIO INDIRIZZO VIRTUALE INDIRIZZO REALE

1. Uso del PC: quando sposto un programma in memoria, modifico solo il PC.
2. Uso di un registro di rilocazione:
   - tutti gli indirizzi sono calcolati come somma con il contenuto di un registro, che punta all'indirizzo fisico di caricamento della prima istruzione del programma, CS nelle CPU Intel;
   - il contenuto del registro deve essere modificato quando il programma è spostato;
   - tutto il programma in un blocco unico.

3. Paginazione (memoria virtuale):
- programma diviso in ***pagine*** (sezioni di lunghezza fissa);
- ***page table***: mappa il numero della pagina al primo indirizzo fisico di memoria che essa occupa quando è caricata;
- binding: modifico solo la page table (il valore del mappaggio);
- più generale del metodo precedente: il programma non è in un unico blocco.

LINK DINAMICO

Su un computer con memoria virtuale.
Non linko tutte le procedure prima d'iniziare l'esecuzione, ma linko una procedura la prima volta che è invocata.
Questo processo è noto come ***link dinamico***.
Ad ogni programma è associato un segmento chiamato ***segmento di link***; esso contiene un blocco d'informazioni per ogni routine chiamabile.
Questo blocco d'informazioni comincia con una parola riservata all'indirizzo virtuale della routine ed è seguita dal nome della routine, memorizzato come una stringa di caratteri.
Per esempio, le DLL (Dynamic Link Library), le DRV (DRiVer) e le FON (FONt) di Windows.

DLL IN WINDOWS

Contiene procedure e/o dati.
Di solito: una libreria composta da più procedure, che possono essere caricate in memoria e accedute da due o più processi contemporaneamente.
Risparmio spazio: non devo caricare in memoria la stessa procedura più volte se è usata da più applicazioni.
Costruita dal linker a partire da un insieme di file (procedimento simile alla costruzione di un eseguibile ISA-Level).
Non può essere eseguita autonomamente.

1. Implicit linking
   - L'applicazione è linkata staticamente ad un file import library, che consente l'uso della DLL;
   - quando l'applicazione è caricata, se la DLL che serve non è già in memoria, è caricata subito.
2. Explicit linking
   - la DLL è caricata runtime, nel momento in cui serve all'applicazione, e dopo il suo utilizzo è scaricata.

UBERTINI MASSIMO
http://www.ubertini.it
massimo@ubertini.it

Dip. Informatica Industriale
I.T.I.S. "Giacomo Fauser"
Via Ricci, 14
28100 Novara Italy

tel. +39 0321482411
fax +39 0321482444

http://www.fauser.edu
http://www.fauser.edu/fau/sistem.htm
massimo@fauser.edu

Massimo Ubertini