Appendice G

Il linguaggio Assembler

Giacomo Bucci

Revisione del 2 giugno 2017

Questo documento è una appendice al volume
Calcolatori Elettronici - Architettura e Organizzazione
IV edizione
McGraw-Hill Education (Italy), S.r.l.
Milano, 2017

Storia degli aggiornamenti

Giugno 2017: primo rilascio.

G

Il linguaggio assembler

OBIETTIVI

- Descrivere le caratteristiche essenziali dei linguaggi assembler
- Illustrare il processo di traduzione del codice sorgente e generazione del codice oggetto
- Mostrare esempi pratici con riferimento a differenti sistemi operativi.

CONCETTI CHIAVE

Linguaggio di programmazione, linguaggio assemblativo, pseudo operazioni, traduzione, generazione del codice, programma oggetto, tavola dei simboli, collegamento dei moduli, rilocazione. Passaggio dei parametri, chiamate al sistema operativo.

INTRODUZIONE

Il linguaggio *assembler*[1] è la forma più rudimentale di linguaggio di programmazione. Esso permette poco più della scrittura in forma mnemonica delle istruzioni di macchina e l'impiego di nomi simbolici in luogo di indirizzi, quantità numeriche e testuali (si faccia riferimento anche a quanto detto al Capitolo 3). Non si tratta di un vero linguaggio di programmazione, in quanto non ne possiede la caratteristica fondamentale: l'indipendenza dall'architettura della macchina. A marcare questa sostanziale differenza, il programma che traduce da testo assembler a codice di macchina viene detto assemblatore e non compilatore.

Poiché ogni architettura si riflette in un proprio linguaggio assembler, si dovrebbe parlare di "linguaggi" piuttosto che di "linguaggio" assembler. Tuttavia, tutti i linguaggi assembler sono basati più o meno sui medesimi concetti. La prima parte di questa appendice è dedicato alla loro illustrazione. Si fa riferimento a una plausibile sintassi, non dissimile da quella dell'assemblatore Intel/Microsoft (Masm) per la famiglia $\times$86, senza fare specificatamente riferimento al relativo repertorio di istruzioni.

Si noti che esiste anche uno standard IEEE (Std 694-1985, *IEEE Standard for Microprocessor Assembly Language*). Esso mira a specificare un insieme di mnemonici per i codici di operazione e a definire una sintassi standard. Ma non sempre tali regole vengono rispettate da chi realizza gli assemblatori.

Nella seconda parte si approfondisce la programmazione assembler $\times$86. Per i motivi che verranno esposti si preferisce utilizzare l'assemblatore Nasm (in luogo del Masm). Vengono

[1] Il termine anglosassone è *assembly language*. Nel gergo degli addetti ai lavori si parla normalmente di *linguaggio assembler*. Preferiamo questa dizione a quella linguisticamente più corretto di "linguaggio assemblativo".

mostrati e discussi diversi esempi di programmi assembler per due differenti ambienti: DOS e Linux (a 64 o 32 bit).

A pagina 47 sono riportati i riferimenti web a documentazione utile.

G.1 Generalità

Per qualunque traduttore il processo di scrittura-traduzione-esecuzione è quello schematizzato in Figura 3.1 del libro e ripetuto in maggior dettaglio in Figura G.1. Esso si compone di questi passi:

1) preparazione del testo del programma sorgente, diviso in più file detti anche moduli;
2) traduzione (assemblaggio o compilazione) dei moduli sorgente in moduli oggetto;
3) collegamento dei moduli e generazione del file eseguibile;
4) esecuzione.

In Figura G.1 si assume che i moduli sorgente siano tutti scritti in assembler. È possibile che parte dei moduli sorgente siano scritti in un linguaggio di alto livello e tradotti direttamente in linguaggio macchina dal compilatore. Talvolta il processo di compilazione avviene in più passi, non visibili all'utente A partire dal programma sorgente di alto livello viene generato un codice intermedio in forma di programma assembler, che, successivamente, viene tradotto in linguaggio macchina da un assemblatore facente parte del compilatore.

Per il collegamento (linking) e il corretto funzionamento dei moduli prodotti da differenti traduttori, i programmi devono essere scritti in modo da rispettare alcuni standard di comunicazione fra moduli (passggio dei parametri, ecc.).

Per semplificare l'esposizione, nella parte che segue si fa l'ipotesi l'intero processo di scrittura-assemblaggio-esecuzione di un programma avvenga sotto DOS (o un sistema similare, per esempio una shell di Linux), in modo che risulti chiaro il significato delle azioni svolte. Supporremo anche che l'assemblatore sia il programma di nome ASM, la cui sintassi sia sostanzialmente quella di Masm, ma non faremo riferimento ad alcun specifico repertorio di istruzioni.

Si fa invece l'ipotesi che il modello di memoria sia lineare.

Facendo riferimento al DOS, normalmente faremo impiego di lettere maiuscole, come d'uso. Ovviamente con altri sistemi (per esempio Linux maiuscole e minuscole sono differenziate)

Preparazione dei moduli sorgente

Per la preparazione dei testi basta un editor convenzionale che produca un puro file di caratteri ASCII (senza caratteri di controllo). Di solito ai file sorgente si dà l'estensione `ASM`.

Ai fini della nostra discussione supponiamo che il nostro programma sia costituito da un modulo principale denominato `MAIN.ASM` e da due sottoprogrammi, corrispondenti ai file `SUB1.ASM` e `SUB2.ASM`.

Traduzione

L'assemblatore elabora un file sorgente alla volta; quindi, per assemblare i tre moduli,

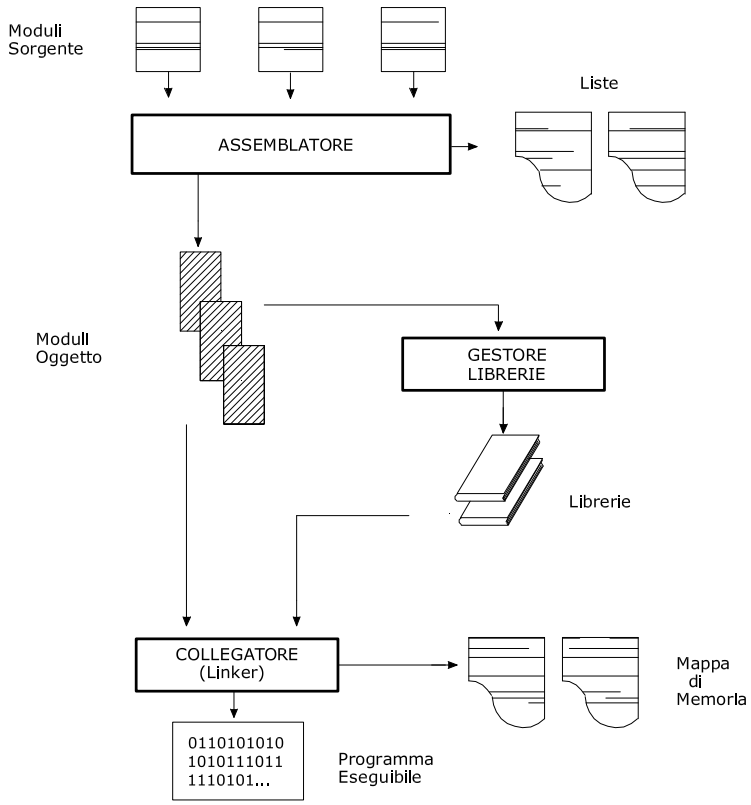

Figura G.1 Generazione di un programma eseguibile a partire da più moduli sorgente. Normalmente ai file sorgente di dà l'estensione `ASM`, anche se molti assemblatori non la richiedono. I file oggetto e gli eseguibili possono avere differenti estensioni a seconda del sistema considerato. Sotto DOS i file oggetto hanno estensione OBJ, mentre i programmi eseguibili hanno estensione EXE o COM. Sotto Linux non è predefinita nessuna estensione per file oggetto o eseguibili, potendo essere qualunque (si vedano gli esempi nella seconda parte dell'Appendice.

occorre fare tre passaggi. A seconda del particolare assemblatore, l'estensione `.ASM` può essere o meno omessa. Per assemblare il modulo `MAIN` si può dare la seguente linea di comando:

```
C:> ASM MAIN.ASM
```

che mette in esecuzione l'assemblatore ASM, dandogli come parametro d'ingresso il nome del file contenete il programma da assemblare. L'assemblatore produce il file (oggetto) e l'eventuale listato contenente il codice macchina equivalente.

Facendo riferimento al DOS, il modulo oggetto riceve automaticamente l'estensione `.OBJ` (cioè produce `MAIN.OBJ`). Ovviamente sotto altri sistemi possono valere altre convenzioni; in particolare, sotto Linux l'estensione data al modulo oggetto può essere qualunque a scelta del programmatore. Il modulo oggetto contiene il codice di macchina e altre informazioni che consentono il collegamento tra i moduli. Si faccia riferimento a quanto detto al Paragrafo 3.5.1 del testo. Per esempio, il nostro modulo `MAIN`, visto che chiamerà il sottoprogrammi SUB1 e SUB2, deve dichiarare SUB1 e SUB2 come simboli

esterni. Questo genere di informazioni vengono opportunamente trasferite nel modulo oggetto, in modo che esse possano guidare il linker nel collegare i moduli e formare il file eseguibile (vedere più avanti).

Il listato ha di solito estensione `.LST`, ma anche qui ogni sistema può avere una propria convenzione. Il listato contiene il testo del programma e le informazioni corrispondenti al codice di macchina prodotto.

Collegamento dei moduli

I moduli oggetto vengono collegati tramite il *linker* per formare un unico file eseguibile. Ad esempio, con la linea di comando:

```
C:> LINK MAIN SUB1 SUB2
```

dove `MAIN`, `SUB1` e `SUB2` sono i moduli oggetto ottenuti come traduzione dei rispettivi moduli sorgente. Si è fatta l'ipotesi che, come nel caso del DOS, il linker prenda i file `.OBJ` in mancanza di estensioni nella linea di comando. Il linker produce il modulo eseguibile `MAIN.EXE` (l'estensione `.exe` è tipica dei file eseguibili Microsoft). Quest'ultimo è in una forma che corrisponde a quella che sarà in memoria al tempo di esecuzione, salvo alcune differenze, legate alla rilocazione degli indirizzi (Paragrafo 3.5.1).

La Figura G.1 mostra che il collegamento può prevedere l'impiego di librerie. Queste possono essere di sistema, oppure costruite tramite uno speciale programma (*librarian*) di gestione delle librerie stesse a partire da moduli oggetto prodotti dal programmatore stesso.

Esecuzione

La linea di comando:

```
C:> MAIN
```

determina il caricamento in memoria del file `MAIN.EXE` e la sua esecuzione.

Il caricamento viene effettuato dal *loader*, un componente del sistema operativo, invocato indirettamente quando viene battuto il nome di un file eseguibile. Il loader provvede ad allocare in memoria il contenuto del file `MAIN.EXE`, assegnando un opportuno indirizzo fisico di partenza e apportando le modifiche che questo passaggio impone. Al termine del caricamento il controllo viene ceduto al programma attraverso una istruzione di salto al punto di entrata del programma. A seconda del sistema in uso il linker/loader aggiusta alcuni registri di macchina con i dovuti valori iniziali.

G.2 Sintassi

Un testo assembler è fatto di linee. Ogni linea è una stringa di caratteri di testo (di norma ASCII), terminante con la combinazione *Carriage Return-Line Feed* (`CR-LF`), oppure solo LF. Alcuni assemblatori non distinguono tra maiuscole e minuscole, per cui, ad esempio, non fa differenza tra scrivere `JMP DEST`, `jmp dest`, ovvero `jMP DeSt`.

Se si escludono i commenti multilinea, ogni linea rappresenta uno *statement*.

La sintassi di un generico statement prevede i seguenti 4 campi, separati da almeno una spaziatura (le parentesi quadre – qui e nel seguito – racchiudono elementi opzionali):

```
[Etichetta]   OPCODE   [Operandi]  [Commento]
```

Per semplicità abbiamo ipotizzato che il campo OPCODE sia obbligatorio; esso determina l'interpretazione degli altri campi.

Commenti

Ogni assemblatore ha le sue regole. Di norma i commenti iniziano con un carattere predefinito. Per

la maggior parte degli assemblatori il commento inizia col carattere "`;`", nel caso dell'assemblatore del MIPS con il carattere "`#`", altri assemblatori usano "`//`".

Codice di operazione

Il campo `OPCODE` è quello che determina il significato dello statement e, conseguentemente il codice generato dall'assemblatore. Il campo può contenere:

a) un mnemonico di operazione di macchina;
b) una direttiva per l'assemblatore, detta anche *pseudo-operazione* a rimarcare che non si tratta di una istruzione di macchina.

Esempio di mnemonici ARM sono `LR` (*Load Register*) o `BL` (*Branch with Link*); esempi di mnemonici ×86 sono `MOV` (*Move*) o `CALL`. Si tenga presente che il linguaggio assembler è poco più di una corrispondenza diretta con il codice di macchina, pertanto, scrivere nel campo OPCODE le mnemonico di una istruzione di macchina, significa far generare all'assemblatore il codice corrispondente (tenendo conto anche contributo degli operandi alla formazione dell'istruzione).

Tuttavia, se nel campo OPCODE fosse obbligatorio scrivere solo mnemonici di operazioni di macchina, sarebbe impossibile costruire un programma, in quanto non si avrebbe modo di definire, per esempio, variabili e costanti. A tale scopo servono le direttive. Una direttiva non rappresenta il nome simbolico di una operazione di macchina, ma, piuttosto, un ordine per l'assemblatore. Supponiamo che al programmatore occorra una costante di valore 127. scrivendo:

```
        DEC  127
```

viene generata, in corrispondenza dello statement in questione, la codifica binaria del numero decimale 127.

Se invece si scrive:

```
COST     EQU   127
```

Il simbolo COST è reso equivalente al numero 127; ovvero ogni volta che nel testo del programma comparirà il simbolo COST ad esso verrà sostituito 127. Ad esempio,

```
        LDI     R1, COST
```

avrebbe l'effetto di caricare (in modo immediato, cioè con il numero codificato nell'istruzione) il registro R1 con 127. Il vantaggio della definizione di un simbolo come COST si ha quando la costante deve essere usata più volte. Il programmatore può dimenticarsi quanto vale esattamente. Se poi capitasse che CONST deve avere un altro valore, basterà che venga cambiato solo lo statement in cui essa è definita, lasciando immutati tutti gli statement che la usano.

Alcuni assemblatori distinguono le direttive dai codici di operazione facendole precedere dal "."; ad esempio: la direttiva `.data` indica che la parte che segue contiene dati, mentre la direttiva `.text` indica che la parte che segue contiene istruzioni.

Operandi

L'esempio appena riportato ha messo in mostra cos'è un operando. Nel caso specifico è il numero (decimale) che segue lo mnemonico `DEC`. Lo statement

```
        ADD  R1,R2,R3
```

presenta tre operandi (nel caso specifico il nome simbolico di tre registri).

Il tipo e il numero di operandi possibili è sempre determinato dal codice di operazione. Nel caso dell'istruzione `ADD` precedente gli operandi sono tre; nel caso dell'istruzione `JAL` l'operando non può che essere uno solo: la destinazione del salto; nel caso della pseudo operazione DEC gli operandi possono essere in un numero a piacere:

```
        DEC  127,-32,8732,0,1
```

In questo caso l'assemblatore interpreterebbe la direttiva generando la rappresentazione binaria dei numeri riportati in posizioni contigue di memoria.

Etichette
Il campo dell'etichetta serve a dare un nome simbolico a ciò che appare alla sua destra. Per esempio:

```
C98    DEC  9876
```

assegna il nome simbolico alla posizione di memoria che conterrà il numero 9876. Il programmatore potrà fare riferimento alla posizione contenente il numero attraverso il suo nome simbolico. Ad esempio:

```
DEST    LW  R1,C98
```

Questo statement ha a sua volta un'etichetta, che serve ad identificare simbolicamente la sua posizione in memoria. Il programmatore potrà quindi, ad esempio, saltare a quella posizione da altra parte del programma scrivendo:

```
        JMP  DEST
```

Parole riservate
Alcuni parole (o, meglio, alcuni simboli testuali) sono riservate. Sostanzialmente le parole riservate possono essere divise in queste due classi:

- mnemonici delle istruzioni, delle direttive e di eventuali operatori;
- mnemonici delle entità che fanno parte del modello di programmazione (registri).

Naturalmente ogni assemblatore ha proprie regole per trattare le parole riservate. Per esempio, gli assemblatori ×86 usano per i registri nomignoli come, AL, DS, EBX, ecc.; l'assembler del PowerPC usa per i registri i simboli r0, r1, r2,.. ; alcuni assemblatori identificano i registri di macchina con il carattere "$", seguito dal numero d'ordine ($0, $1, $2, ..), ma i medesimi hanno anche un nomignolo riservato, per esempio il registro $4 è anche identificato come $a0.

Per gli assemblatori che distinguono tra maiuscole e minuscole, questa distinzione si applica solo ai simboli definiti dal programmatore, le parole riservate possono essere scritte in un modo o nell'altro o mischiando. Ad esempio SEGMENT, segment, SEgmenT.

Avvertenza
Le regole che abbiamo dato sono molto schematiche. Abbiamo supposto che lo statement `LD R1,C98` facesse riferimento alla posizione `C98`. Questa è la norma per molti assemblatori. Tuttavia essa tende a generare confusione, perché essa ha la stessa forma dell'istruzione `LD R1,COST`, sebbene nel primo caso si tratti di un riferimento alla memoria nel secondo caso del caricamento di un immediato. In Nasm si dovrebbe scrivere `LD R1,[C98]`, indicando esplicitamente che si tratta di un riferimento alla memoria. Molti assemblatori sono assai più progrediti di quanto abbiamo lasciato intravedere. Per esempio, è normale che venga fatta una distinzione tra etichette che rappresentano nomi di costanti o variabili e etichette (*label*) imposte alle istruzioni. Spesso i label devono terminare con il carattere "`:`".

G.2.1 Le macro

L'uso delle macro consente di ripetere in maniera semplice ed efficace la codifica di un insieme di istruzioni utilizzato più volte nel codice. Esse sono perciò un valido strumento per quelle funzioni, identificate nella fase di progetto, troppo minute per giustificare la realizzazione con una procedura, oppure per le quali i requisiti temporali siano incompatibili con l'overhead di chiamata. Una macro evita l'uso delle istruzioni per la chiamata e il passaggio dei parametri necessari nel

caso delle procedure. Per contro fa incorrere in un aumento delle dimensioni del codice, poiché le istruzioni che realizzano la funzione appaiono tante volte quante sono le "chiamate di macro".

In Masm una macro viene definita utilizzando la coppia di direttive `MACRO/ENDM`, secondo la sintassi:

```
NomeMacro  MACRO  [ListaParametriFormali]
           BloccoStatement
           ENDM
```

dove `NomeMacro` è un nome e la `ListaParametriFormali`, eventualmente vuota, contiene una sequenza di uno o più nomi, separati da virgole. I nomi assegnati ai parametri formali sono una classe speciale di simboli che si distingue da tutte le altre, pertanto è possibile usare gli stessi nomi nella definizione di più macro e, qualora un nome coincida con quello di un altro simbolo definito nel modulo, viene comunque considerato diverso da esso.

Il `BloccoStatement` è una sequenza di statement assembler di qualsiasi tipo, comprese le chiamate di macro e anche le direttive di definizione di altre macro, che così risulteranno annidate al suo interno.

L'impiego di una macro, detto impropriamente "chiamata", consiste nell'indicare il suo nome nel campo `OP` di uno statement, fornendo in quello `Operandi` gli eventuali parametri attuali, ciascuno dei quali verrà fatto corrispondere posizionalmente con uno formale della definizione. La sintassi di chiamata di una macro è:

```
        NomeMacro   [ListaParametriAttuali]
```

dove la `ListaParametriAttuali`, eventualmente vuota, contiene una sequenza di uno o più simboli separati da virgole, questi possono essere identificatori, nomi di registri, costanti numeriche e/o caratteri e altri. Durante l'assemblaggio, a ogni occorrenza del nome di una macro, viene sostituito il blocco di statement che ne costituisce il corpo, effettuando così l'"espansione" della macro. In questa fase i parametri attuali, specificati nella chiamata, vengono sostituiti ai corrispondenti parametri formali, operando una sostituzione di stringa. Il codice ottenuto viene poi normalmente assemblato.

Qui di seguito viene mostrato un esempio che dovrebbe chiarire l'utilità delle macro. Si supponga che da più parti di un programma si debbano presentare brevi messaggi di testo a video. A tal fine si può utilizzare le funzione DOS numero 9 (si veda il Paragrafo G.5.3) per definire una macro che prende come parametro il nome dell'area di memoria contenente i caratteri da presentare a video. Denominata `DISPLAY` la macro corrispondente, essa si definisce nel modo seguente:

```
DISPLAY  macro   msg              ;msg: parametro formale
         mov     DX,offset msg    ;DS:DX = indirizzo di msg
         mov     ah,9             ;AH = funzione DOS
         int     21H
         endm
```

Per presentare la stringa di nome `MESSAGGIO`, basta scrivere:

```
        DISPLAY    MESSAGGIO
```

questo statement viene espanso come da definizione della macro, sostituendo ogni occorrenza del simbolo `msg` col simbolo `MESSAGGIO`.

G.3 Il processo di traduzione

Torniamo per un attimo al tratto di codice assembler corrispondente allo statement C del Capitolo 3:

```
a = b+c;
```

che abbiamo supposto che esso fosse tradotto come:

```
LD    R2,B
LD    R3,C
ADD   R1,R2,R3
ST    A,R1
```

Ovviamente il programmatore non può limitarsi a scrivere solo le operazioni eseguite dal programma, deve anche prevedere gli oggetti da esse manipolati. Nel caso specifico deve prevedere tre posizioni di memoria per le tre variabili `a`, `b` e `c`. Supponendo di indicare con `A`, `B` e `C` le posizioni in questione, programma assembler potrebbe essere questo:

```
.text                          ;quel che segue è codice
      LD    R2,B
      LD    R3,C
      ADD   R1,R2,R3
      ST    A,R1
      :::
.data                          ;quel che segue sono dati
B     DEC   9999
C     DEC   777
A     DEC   0
```

(con "`:::`" si sono indicati eventuali ulteriori statement). Notare che nella Figura 2.7 del libro le tre posizioni `A`, `B` e `C` erano sparse, mentre nel precedente testo sono state rese contigue.

Il processo di assemblaggio consiste nel tradurre nella corrispondente "immagine di memoria" gli statement del programma.

Nello schema precedente è stato indicata solo la ripartizione tra codice e dati, lasciando che sia l'assemblatore (e successivamente il linker) ad assegnare gli indirizzi. Pertanto il processo di assemblaggio avviene con riferimento all'indirizzo di partenza convenzionale "0"[2]. Gli indirizzi possono essere poi elaborati dal linker quando collega i moduli e, infine, aggiustati dal loader all'atto del caricamento del codice in memoria (Paragrafo 3.5 del libro). Qui sotto si riporta il plausibile listato (`.LIST`) generato dall'assemblatore.

```
(0000:   LD  R2,x)       LD    R2,B
(0004:   LD  R3,x+4)     LD    R3,C
(0008:   ADD R1,R2,R3)   ADD   R1,R2,R3
(0012:   ST  x+8,R1)     ST    A,R1
                         :::
(x      9999)            B     DEC   9999
(x+4    777)             C     DEC   777
(x+8    0)               A     DEC   0
```

Rispetto al testo originale del programma ci sono due colonne in più. Quella a sinistra rappresenta l'indirizzo assegnato all'istruzione, la colonna accanto il codice corrispondente. Ovviamente l'assemblatore riporta il codice numerico anche per le operazioni, mentre noi ci siamo dovuti accontentare di riportare ancora i loro mnemonici.

Di norma il programmatore non dà l'indirizzo di partenza

Il succo del precedente ragionamento è che il processo di traduzione richiede che venga assegnato un "valore" ai simboli introdotti dal programmatore. In particolare devono essere individuati gli indirizzi corrispondenti alle etichette, in modo da poter formare le istruzioni come sopra. Di solito il processo si compie in due passi.

[2]Esistono delle direttive come `ORG` che consentono di fissare un preciso indirizzo di partenza.

Passo 1
Il primo passo ha il compito di associare un indirizzo ad ogni etichetta che compare nel testo.

A tale scopo, l'assemblatore utilizza una variabile designata come `ILC` (*Instruction Location Counter*). `ILC` viene inizializzata a 0 (o al valore dato dallo statement ORG) e viene incrementata man mano che vengono esaminati i vari statement. L'incremento è pari al numero di byte richiesti per la traduzione in linguaggio macchina di ogni statement.

Con riferimento al nostro esempio il listato precedente mostra come il programma calcola gli indirizzi da assegnare alle istruzioni. L'istruzione `LD R2,B` occupa la posizione 0 (quella iniziale), mentre la successiva occupa la posizione 4 –assumendo che le istruzioni occupino 4 byte. Si è supposto che la prima variabile venga allocata alla posizione x; se non ci fossero gli statement indicati con `:::`, x varrebbe 1016.

Per tenere traccia degli indirizzi, il passo 1 costruisce una *tavola dei simboli*, nella quale vengono memorizzate tutte le etichette e il corrispondente valore di ILC[3].

Passo 2
Il secondo passo ha il duplice compito di generare il codice oggetto nella forma che consenta di essere trattato dal linker.

Il passo 2 opera rileggendo il programma sorgente[4], ripartendo da `ILC`= 0 e incrementando ILC esattamente come al passo 1. Con riferimento all'istruzione alla posizione 104 (`LD R3,C`), il codice viene generato *assemblando* la traduzione dello mnemonico (`LD R3`) con l'indirizzo (`x+4`) assegnato all'etichetta C al passo 1.

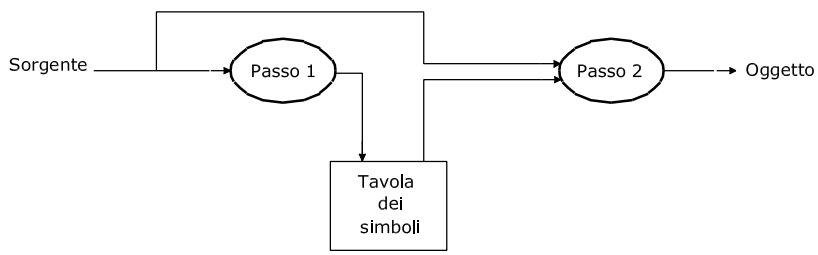

Figura G.2 Assemblatore a due passi.

In Figura G.2 viene illustrato il funzionamento complessivo di un assemblatore a due passi.

G.4 Collegamento dei moduli

Ai fini della nostra discussione abbiamo ipotizzato che il nostro programma fosse costituito da programma principale denominato `MAIN.ASM` e da due sottoprogrammi denominati `SUB1.ASM` e `SUB2.ASM`. Per convenienza abbiamo anche ipotizzato che ciascuno di essi fosse rappresentato

[3]Si noti però che non tutti i simboli hanno associato un ILC, ovvero non tutti i simboli introdotti dal programmatore sono etichette (ad esempio i simboli introdotti con la direttiva EQU). La tavola dei simboli serve a tenere traccia di tali valori e del loro *tipo*. Per esempio, i simboli A, B e C definiti nel tratto di codice a pagina 8 vengono immessi nella tavola dei simboli come "indirizzi" e il corrispondente valore è l'indirizzo ad essi assegnato; il simbolo COST di pagina 5 viene immesso nella tavola dei simboli come "numero" e il valore, ovviamente, è 127.

[4]Gli assemblatori più progrediti generano una *forma intermedia* al passo 1; questa viene letta al passo 2, evitando la completa rilettura del testo sorgente.

da un file di uguale nome. La traduzione rispettivamente in `MAIN.OBJ`, `SUB1.OBJ` e `SUB2.OBJ` richiede tre processi distinti di assemblaggio.

Per poter effettuare da `MAIN` la chiamata a `SUB1` o `SUB2`, è necessario che in esso i nomi delle sue subroutine siano dichiarati come "esterni", cioè attraverso una direttive che dice all'assemblatore che quel simbolo non è definito nel corrente modulo. Conseguentemente, l'assemblatore genererà un codice incompleto per l'istruzione di chiamata, aggiungerà nel modulo `.OBJ` le informazioni che serviranno a effettuare il collegamento da parte del linker. `MAIN.ASM` avrà questa forma:

```
       EXTRN  sub1,sub2       ;dichiarazione simboli esterni
main   :::                    ;punto di ingresso al Main
       CALL   sub1
       :::
       CALL   sub2
       :::
```

Nel modulo `MAIN.OBJ` ci sarà traccia che `sub1` e `sub2` sono simboli esterni, cui viene assegnato uno pseudo indirizzo esterno, in modo che le istruzioni di `CALL` possano essere assemblate rispetto ad esso. Schematicamente `MAIN.OBJ` avrà questa forma:

```
sub1   (1x)                   ;sub1: esterno n.1
sub2   (2x)                   ;sub2: esterno n.2
main   :::                    ;punto di ingresso al Main
       CALL   (1x)            ;CALL tradotto in codice, ..
       :::                    ; ...(1x) in forma simbolica
       CALL   (2x)            ; idem
       :::
```

Corrispondentemente nei file `SUB1.ASM` e `SUB2.ASM` i nomi dei sottoprogrammi devono essere dichiarati come "globali" (o "pubblici"), ovvero visibili dall'esterno. Ad esempio:

```
       GLOBAL sub1            ;sub1 visibile dall'esterno
sub1   :::                    ;punto di ingresso a sub1
```

Il fatto che `sub1` è un simbolo pubblico viene riportato nel modulo oggetto.

Il linker costruisce il modulo `.EXE`, mettendo assieme il codice dei tre moduli oggetto. Verosimilmente la struttura dell'eseguibile è questa (con I0, I1 e I2 si sono indicati gli indirizzi a cui vengono allocati i tre simboli):

```
(I0)   :::                    ;I0: indirizzo punto entrata
       CALL   I1              ;codice chiamata a sub1
       :::
       CALL   I2              ;codice chiamata a sub2
       :::
(I1)   :::                    ;I1: indirizzo sub1
(I2)   :::                    ;I2: indirizzo sub2
```

Si noti che di norma il codice `.EXE` è ancora rilocabile, nel senso che esso può essere posizionato ad un indirizzo qualunque in memoria all'atto del caricamento. Conseguentemente, il loader dovrà aggiustare gli indirizzi con il fattore di rilocazione.

Calcolatori Elettronici - Architettura e Organizzazione

Appendice G

Seconda parte

Esempi di programmazione assembler

Revisione del 2 giugno 2017

G.5 Premessa

Questa parte mira a familiarizzare il lettore con la programmazione assembler. Ovviamente ci si riferisce all'architettura ×86.

Nel libro di testo, e nella prima parte di questa appendice, si è fatto ricorso alla sintassi del Masm, l'assemblatore di Intel e Microsoft. Il motivo per cui nel libro si è fatto riferimento a Masm è semplice: la documentazione Intel quando mostra le caratteristiche dei suoi dispositivi, all'occorrenza, ricorre a esempi scritti secondo la sintassi di questo assemblatore.

In questa seconda parte dell'Appendice G, si ricorre invece al Nasm (Netwide Assembler), un prodotto open source[5], distribuito sotto licenza BSD (semplificata).

Le ragioni che ci hanno fatto scegliere il Nasm sono queste:

- Nasm ha una sintassi semplice, e, da svariati punti di vista, più consistente di Masm. Si presta quindi meglio ad essere impiegato a scopo didattico.
- La sintassi di Nasm non è, tuttavia, dissimile da quella di Masm (stesso ordine degli elementi di uno statement, stessi nomi per i registri, ecc.), pertanto, se fino a questo punto si è fatto ricorso alla sintassi di Masm, il passaggio a Nasm sarà del tutto naturale.
- Sebbene il Nasm non abbia tutta la potenzialità del Masm, esso è un prodotto completo, con il quale si possono scrivere programmi di qualunque complessità.
- Alternativamente si poteva scegliere GAS (Gnu Assembler), l'assemblatore del progetto Gnu. Tuttavia la sua sintassi è anomala (ha l'operando di destinazione alla destra degli operandi sorgente); inoltre i registri vengono denotati in modo tale da rendere i testi di programma inguardabili.
- In Internet sono rintracciabili altri assemblatori, alcuni dotati anche di un sistema proprio di sviluppo o di simulazione, ma in genere si tratta di prodotti che non hanno le caratteristiche, né la diffusione di Nasm.
- Nasm può essere utilizzato su differenti piattaforme (Unix, Windows e DOS); ciò consentirà di proporre lo stesso esempio per differenti ambienti.

Invitiamo sen'altro il lettore a scaricare il manuale di Nasm e a sincerarsi delle sue caratteristiche. L'indirizzo del sito da cui può essere scaricato è al Paragrafo G.9.4. Qui di seguito vengono mostrati alcuni aspetti essenziali della sua sintassi[6].

G.5.1 Alcuni aspetti caratteristici dell'assemblatore Nasm

Riferimenti alla memoria

Un differenza fondamentale tra Nasm e Masm è il modo in cui Nasm fa riferimento alla memoria. In Nasm i riferimenti alla memoria si fanno mettendo tra parentesi quadre il nome (label) assegnato. Le seguenti sono due istruzioni Nasm

```
mov eax,[var]            ; eax <- M[var]
mov eax,[var+ebx]        ; eax <- M[var+ebx]
```

In Masm esse sarebbero state scritte come

```
mov eax,var             ; eax <- M[var]
mov eax,var[ebx]        ; eax <- M[var+ebx]
```

[5] C'è da dire che alla data di scrittura di queste righe, anche il Masm (ormai arrivato alla versione 8.0) è scaricabile liberamente da Internet. In passato era a pagamento.

[6] Per l'esattezza, si fa riferimento alla documentazione relativa alla versione 2.12.01.

La differenza tra le due sintassi è evidente se di considerano queste due definizioni:

```
uno     equ     1
due     dw      2
```

Nel caso di Masm le due istruzioni seguenti, sebbene esse abbiano identico aspetto, generano codici completamente diversi (la prima un caricamento immediato del numero 1 in ax, la seconda il caricamento del numero 2 contenuto in una cella di memoria).

```
        mov     ax,uno
        mov     ax,due
```

In Nasm i corrispondenti statement vengono scritti come segue, evidenziando il fatto che la seconda istruzione indirizza una posizione di memoria.

```
        mov     ax,uno
        mov     ax,[due]
```

Si noti che quanto sopra rende inutile la parola riservata OFFSET di Masm. Infatti, l'equivalente dello statement Masm `mov ax,offset var` diventa semplicemente `mov ax,var` se scritto in Nasm.

Inoltre Nasm tratta tutto ciò che si trova nel campo delle etichette come tali, cioè non distingue tra nomi di variabili e *label* (etichette che in Masm terminano con ":").

Tipi di variabili

Masm tiene traccia di come è definita una variabile (byte, parola, doppia parola parola quadrupla). Nasm ricorda soltanto l'indirizzo del primo byte della variabile; pertanto si deve tener conto della sua dimensione nell'indirizzarla. Ad esempio `mov word [var],2` trasferisce 16 bit nella parola di indirizzo `var`; se fosse stato scritto `mov [var],2` sarebbe stato trasferito un solo byte. Per contro, questa caratteristica del Nasm rende inutilizzabili certe istruzioni di movimento di stringhe.

Maiuscole/minuscole

Nasm differenzia tra maiuscole e minuscole (eccetto le parole riservate). Per esempio c'è differenza tra scrivere `VAR`, `Var` o `var`.

G.5.2 Linea di comando

Si assume che Nasm sia stato installato e che sia inserito nel PATH[7]. Per convenienza conviene portarsi nella cartella (*directory*) in cui si trova il file di testo del programma sorgente.

L'assemblatore si chiama con un comando del tipo:

```
    nasm -f <format>  <filename> [-o <output>]
```

Ad esempio, con la riga di comando

```
        nasm -f elf Fs.asm  -o Fo.o
```

viene assemblato il file Fs.asm, generando il file oggetto Fo.o in formato elf. Il formato elf è lo standard per i sistemi Linux a 32 bit (si veda il paragrafo 3.5.1 del libro). Invece, la linea di comando

```
        nasm -f elf Fxy.asm
```

produce il file oggetto Fxy.o, in formato elf.
Con la riga di comando

```
        nasm -f elf64 Fsorg.xx  -o Foggetto.zzzz
```

il file Fsorg.xx viene tradotto nel file Foggetto.zzzz in formato elf64.

[7] Ovviamente dovrà essere scaricata la versione corrispondente al sistema operativo sotto il quale l'assemblatore verrà impiegato.

Vale la pena di notare quanto segue:

- L'estensione `.asm` al file da assemblare è irrilevante, la si mette per convenienza per riconoscere i file sorgente; l'estensione può essere qualunque.
- L'opzione `-f` serve a specificare il formato del file oggetto. Il formato del programma oggetto deve essere scelto in modo da essere congruente con il sistema in uso (si veda più avanti).
- L'opzione `-o` serve a specificare il nome del file oggetto. Se esso non viene specificato, il file oggetto prende il nome del file sorgente con estensione.o. Al file oggetto può essere dato un qualunque nome, compatibilmente con i vincoli posti dal sistema. Nell'esempio appena mostrato esso è stato chiamato Foggetto.zzzz.
- Tralasciando altre opzioni (in numero molto copioso), per le quali si rinvia al manuale, ricordiamo l'opzione `-l` a seguito della quale è possibile specificare il nome del file "listing", ovvero il nome del file che contiene, oltre al testo, il codice assegnato ad ogni istruzione/dato e la relativa posizione.

G.5.3 Le funzioni del sistema operativo

Non di rado il programmatore assembler è chiamato a scrivere programmi che devono funzionare in modalità *stand-alone*, cioè a sé stanti, senza alcun supporto di sistema operativo. È facile che ciò accada per applicazioni *embedded*. In tal caso la verifica del funzionamento del programma può rivelarsi alquanto ostica e possono servire emulatori o apparati simili allo scopo di un controllo esaustivo. Alla fine, i programmi *stand-alone* finiscono normalmente in ROM e diventano parte dell'elettronica complessiva della macchina.

Nel seguito verranno, invece, mostrati alcuni esempi scritti per funzionare sotto differenti sistemi operativi. I sistemi operativi offrono un insieme di funzionalità che il programmatore può sfruttare per svolgere funzioni (per esempio di ingresso/uscita) che altrimenti risulterebbero molto gravose da sviluppare. In particolare sono di interesse le funzioni di ingresso/uscita (scrittura/lettura su terminale, apertura/chiusura scrittura/lettura file). Quindi, è del tutto naturale che per scopi didattici ci si debba appoggiare ai sistemi operativi.

Gli esempi proposti nel seguito sono stati sviluppati per seguenti ambienti:

DOS: FreeDOS 1.2, installato come macchina virtuale sotto Virtual Box 5.1.xx in ambiente Windows 10.

Linux: Ubuntu 16.04 LTS, installato come macchina virtuale sotto Virtual Box 5.1.xx o sotto VMware-player 12.5.x, ambedue in ambiente Windows 10.

Per quanto si riferisce all'uso di FreeDOS come macchina virtuale, si deve tener conto che il DOS originale non gira in ambiente Windows a 64 bit. FreDOS invece è del tutto compatibile con il DOS e può funzionare come macchina virtuale. Si è fatto ricorso a `FreeLink`, un linker scaricabile gratuitamente da Internet, funzionante sotto DOS.

Per quanto si riferisce alla scelta di usare Ubuntu come macchina virtuale si è trattato di un semplice motivo di convenienza: avere tutto nello stesso ambiente (Windows 10). Peraltro le macchine virtuali hanno un piccolo vantaggio, specialmente in fase di test dei programmi: se questi non funzionano correttamente, i danni che possono provocare restano comunque confinati alla macchina virtuale, senza impatti sul sistema ospitante.

Differenze tra Linux a 64 o 32 bit

Ubuntu 16.04 LTS è una versione Linux a 64 bit. Quindi esso fornisce le funzioni per la versione a 64 bit. Tuttavia, Ubuntu per i 64 bit, come pure altre implementazioni di Linux a 64 bit, è in grado di eseguire codice a 32 bit e in particolare di accettare le chiamate alle funzioni di Linux a 32 bit. Questo perché la versione a 64 bit è comprensiva delle librerie a 32 bit. Il lettore che voglia sincerarsi se il suo sistema è a 64 o 32 bit puo dare il comando[8]

```
dpkg --print-architecture
```

che, per sistemi a 64 bit, dovrebbe far vedere qualcosa come

```
amd64
```

Invece dando il comando

```
dpkg --print-foreign-architectures
```

se il sistema è disposto gestire l'architettura a 32 bit, dovrebbe apparire

```
i386
```

In caso contrario, occorrerà installare le dovute librerie.

Per poter far girare i programmi a 32 bit (sotto Linux a 32 bit) è necessario specificare alcune opzioni, sia per l'assemblatore, sia per il loader, come sarà mostrato più avanti nelle sedi opportune.

Convenzioni di chiamata delle funzioni del sistema operativo

Per chiamare le funzioni del sistema operativo occorre rispettare le convenzioni per esso stabilite.

- DOS: il sistema operativo viene chiamato con istruzione `int 21h`.
- Linux: nella versione a 32 bit il sistema operativo viene chiamato con l'istruzione `int 80h`; nella versione a 64 bit con l'istruzione `syscall`. In ambedue i casi esse corrispondono a chiamate in linguaggio C alle funzioni delle API.
- Windows: viene chiamato attraverso una API, corrispondente a un insieme di chiamate in linguaggio C. Facciamo a meno di presentare esempi sotto Windows.

Evidentemente, le chiamate attraverso l'istruzione `int` o `syscall`, come pure le chiamate alle funzioni C, devono passare i parametri in modo coerente rispetto alla corrispondente ABI/API. Qui di seguito riportiamo alcuni esempi di chiamata per DOS e Linux.

DOS

La chiamata al DOS richiede che il registro AH contenga il numero della funzione scelta e che gli eventuali parametri vengano passati con altri registri. Il DOS prevede circa un centinaio di funzioni; la Tabella G.1 viene riportato un sottoinsieme delle funzioni DOS disponibili. Un'ulteriore convenzione del DOS è che la stringa da presentare deve finire con il carattere "$" (i due byte che lo precedono sono rispettivamente il "ritorno carrello" e l'"avanzamento linea")

Ad esempio supponiamo che il nostro programma debba scrivere il messaggio "Hello World" a video. In Nasm occorrerebbero queste righe:

```
mess    db   'Hello World',0Dh,0Ah,'$'
       :::::
        mov  ah,1           ;funzione scelta
        mov  dx,mess        ;DS:DX indirizzo stringa
        int  21h
```

[8] Ci sono molteplici comandi che permettono di sapere se il sistema è a 64 o 32 bit.

Funzioni DOS	AH	Parametri	i/o
Ingresso di un carattere da tastiera	01	`AL` = Carattere digitato	out
Stampa di un carattere	02	`DL` = Carattere da stampare	in
Stampa di una stringa	09	`DS:DX` = Indirizzo stringa	in
Ingresso di una stringa da tastiera	0A	`DS:DX` = Indirizzo stringa	in
Terminazione del programma	4C	`AL` = Codice ritorno dal prog.	in

Tabella G.1 Alcune funzioni DOS. La seconda colonna riporta il numero (esadecimale) da caricare nel registro AH per richiedere la funzione corrispondente. La terza colonna riporta i registri da impiegare per il passaggio degli eventuali parametri della funzione chiamata; la quarta colonna dice se il parametro è di ingresso o di uscita alla funzione chiamata.

Linux 32

Il numero di funzione viene dato attraverso EAX. Eventuali parametri vengono passati nei registri EBX, ECX, EDX, SI, DI, ordinatamente. Il risultato dell'esecuzione della funzione viene restituito attraverso il registro EAX.

In Linux tutto viene visto come file. Un file è individuato attraverso il suo file descriptor (`fd`). Le funzioni di lettura e scrittura hanno come primo parametro l'`fd` del file. Nel caso di operazioni relative alla tastiera o al video fd corrisponde allo standard input `stdin` o allo standard output `stdout`. Si noti che la funzione di apertura di un file viene chiamata passando il nome del file; essa restituisce (in eax) lo `fd` (un numero) assegnatogli dal sistema; lo `fd` può essere utilizzato in eventuali, successive, operazioni di lettura/scrittura.

In tabella G.3 si elencano alcune chiamate e i relativi parametri.

Funzione	EAX	EBX	ECX	EDX
Ritorno a Linux	01	n. errore		
Read	03	fd	indirizzo stringa	n. max caratteri
Write	04	fd	indirizzo stringa	n. caratteri
Apertura file	05	indir. nome file	flag	modo
Chiusura file	06	fd		

Tabella G.2 Alcune funzioni Linux a 32 bit. La seconda colonna riporta il numero da caricare nel registro AH per selezionare la funzione; le colonne successive gli eventuali parametri. Il risultato della funzione sono restituiti al programma chiamante attraverso EAX.

Prendendo come riferimento l'esempio della scrittura di un messaggio, come in precedenza, il programma dovrà contenere queste righe

```
mess    db  'Hello World',0Ah
l       equ $-mess          ;numero di carattere da presentare
       :::::
        mov  eax,4          ;funzione scelta: scrittura
        mov  ebx,1          ;presentazione su stdout
        mov  ecx,mess       ;ECX<- indirizzo messaggio
        mov  edx,l          ;EDX<- numero caratteri
        int  80h
```

Notare che a differenza del DOS che riconosceva la fine della stringa dal carattere "$", qui il numero dei caratteri è un parametro di ingresso alla funzione.

Linux 64

Nel passare da Linux a 32 bit (quello che tradizionalmente veniva chiamato semplicemente `Linux`) a Linux a 64 bit sono stati cambiati i numeri delle funzioni (passati in RAX) e i registri usati per passare i parametri, che ora sono (ordinatamente) RDI, RSI, RDX, R10, R8 e R9.

Funzione	RAX	RDI	RSI	RDX
Read	00	fd	indirizzo stringa	n. max caratteri
Write	04	fd	indirizzo stringa	n. caratteri
Apertura file	05	indir. nome file	flag	modo
Chiusura file	06	fd		
Ritorno a Linux	60	n. errore		

Tabella G.3 Alcune funzioni Linux 64 bit. La seconda colonna riporta il numero (decimale) di funzione da passare nel registro RAX. Si noti che i registri usati per passare i parametri non sono i corrispondenti dei registri usati da Linux a 32 bit.

Ripetendo il nostro esempio si avrebbero queste righe di codice assembler:

```
mess    db      'Hello World!', 0Ah
l       equ     $-mess
       :::::
        mov     rax, 1                  ; rax <- N. funzione write
        mov     rdi, 1                  ; rd1 <- N. standard output
        mov     rsi, mess               ; indirizzo del messaggio
        mov     rdx, l                  ; numero di byte
        syscall                         ; chiamata al SO
```

Avvertenza

Nel seguito assumeremo che il lettore abbia una sufficiente conoscenza dei sistemi operativi. Inoltre faremo uso dei comandi nella forma più immediata, ma atta a illustrare gli argomenti trattati. Non ci soffermeremo su le possibili opzioni utilizzabili nei comandi. Per esempio, nell'impiego del compilatore `gcc`, trascureremo di dire che sono possibili opzioni come `-v` ("verbose"), che all'occorrenza fa generare svariati messaggi di spiegazione o di aiuto, o come `-Wall` con la quale si richiede di emettere tutti i "warning" che il compilatore ritiene di dare rispetto al testo del programma.

G.6 Hello World: il primo programma di esempio

In precedenza sono stati mostrati alcuni tratti di codice relativi alla presentazione a video del messaggio "Hello World". Qui di seguito si mostra la struttura completa del programma sviluppato per DOS e Linux.

G.6.1 Hello World sotto DOS

In Figura G.3 viene riportato il testo assembler della versione per DOS. Il file è stato denominato `HW.asm`. Le righe iniziali di commento illustrano i passi di assemblaggio, collegamento (linking) ed esecuzione. Come linker è stato usato FreeLink, liberamente scaricabile da Internet e di facile uso. Naturalmente sia `nasm.exe` sia `freelink.exe` devono trovarsi nel PATH.

```
; File HW.asm  versione DOS.   Scrive Hello World a video
;
; assemblare:    nasm -f obj hw.asm
; linking:       freelink hw         (.obj non  necessario)
; eseguire:      hw
;
CR    EQU       0DH  ; Carriage Return
LF    EQU       0AH  ; Line Feed

;Area Dati -------------------------------------------
SEGMENT .data
Mess    DB      'Hello World',LF,CR,'$'

;Stack ------------------------------------------------
SEGMENT stack64     STACK
        resb        64
testaSTK:

;Codice ----------------------------------------------
SECTION .txt

..start:
        MOV     AX,data     ;Inizializzazione
        MOV     DS,AX       ; del registro DS

        MOV     DX,Mess     ;DS:DX indirizzo Mess
        MOV     AH,09H      ;chiamata della funzione
        INT     21H         ; di scrittura del DOS

exit:   MOV     AH,4CH      ;uscita con ritorno a DOS
        INT     21H
```

Figura G.3 Programma "Hello World" in versione DOS.

Notare che:

- Dovendo funzionare sotto DOS, è stato scelto di generare un file oggetto in formato `obj`, il formato originale del DOS.
- A FreeLink viene passato il file `hw.obj`; si osservi che non è necessario riportare l'estensione, in quanto il linker sceglie comunque il file dello stesso nome con estensione obj.
- Sia il file oggetto generato da Nasm sia il file eseguibile `hw.exe`, generato dal linker, vengono prodotti nella stessa cartella in cui si trova `hw.asm`. Battendo `hw` (maiuscole o minuscole per il DOS sono irrilevanti) il programma passa in esecuzione.
- Per dare leggibilità al testo il codice del "Ritorno carrello" e "Avanzamento linea" sono stati definiti come `CR` e `LF`.
- Le parole riservate `segment` (tradizione di Masm) e `section` (preferita da Nasm e dai compilatori) sono equivalenti. Inoltre, essendo parole riservate, possono essere scritte con caratteri minuscoli o maiuscoli. Nasm "digerisce" praticamente tutto a seguito di queste due parole. Il simbolo che segue viene preso come nome del segmento/sezione; l'ulteriore simbolo viene preso come tipo di segmento/sezione. Nel caso specifico il segmento dati viene denominato `data`, il segmento di codice viene denominato come `txt`, il segmento di stack come `stack64` (si veda più avanti la mappa generata dal linker)[9].
- Il segmento di stack è stato aggiunto solo per far contento il linker, che, altrimenti, emetterebbe un *warning*, indicando la mancanza dello stack. Il programma funzionerebbe ugualmente senza la definizione dello stack, in quanto non ne fa uso. Nella definizione dello stack, l'etichetta `testaSTK` indica la testa dello stack (si ricordi che nell'architettura ×86 lo stack si espande verso gli indirizzi bassi).
- La sezione `text` contiene il codice. Si noti il label `..start`. Esso serve a indicare al linker il punto di entrata del programma (è una convenzione di Nasm per DOS).
- Le prime due istruzioni del segmento `txt` servono a caricare nel registro DS l'indirizzo di partenza del segmento dati. Si tratta di due statement necessari per DOS (macchine a 16 bit). Essi sono dovuti al fatto che, quando il DOS passa il controllo al programma, il contenuto del registro DS è indefinito (ovvero non contiene la base del segmento dati), diversamente da CS che, per forza di cose, contiene la base del segmento di codice. Il linker riloca gli indirizzi, assegnando al segmento `data` un indirizzo di base nello spazio degli indirizzi fisici). L'istruzione `MOV AX,data` ha l'effetto di portare in AX la base del segmento `data`. Il repertorio di istruzioni dell'8086 non presenta istruzioni che consentano di caricare direttamente in un registro di segmento il valore della base; occorre usare un registro generale (si è scelto di usare AX, ma andavano bene anche BX, CX, DX, SI e DI) e poi copiare nel registro di segmento il valore caricato. Volendo inizializzare anche lo stack sarebbero state necessarie queste istruzioni aggiuntive
```
mov     ax,stack64
mov     ss,ax
mov     sp,testaSTK
```
- Il terza istruzione della sezione `txt` (`MOV DX,Mess`) carica in DX l'offset di `Mess` rispetto alla sua base. Come abbiamo notato in precedenza, questa è una rilevante differenza sintattica rispetto al Masm, dove si dovrebbe scrivere `MOV AX, OFFSET Mess`.
- Seguono la chiamata della funzione di scrittura del DOS e l'uscita dal programma con ritorno al DOS.

[9] Il fatto che la mappa del linker riporti tutto in maiuscolo deriva dal fatto che esso appartiene all'epoca in cui era normale usare le lettere maiuscole nella scrittura dei programmi.

In Figura G.4 viene mostrata la mappa generata dal linker dando il comando `freelink hw.obj, -m hw`. Come si vede vengono riportati i nomi dei tre segmenti, le loro posizioni di inizio e fine, oltre alla dimensione.

```
   PROGRAM: HW.EXE
DATE:    05/05/17
TIME:    06:28 pm

Start  Stop   Length Name                   Class
00000H 0000DH 0000EH DATA
00010H 0004FH 00040H STACK64
00050H 00065H 00016H TXT

Program entry point at 0005:0000
```

Figura G.4 Mappa prodotta da FreeLink.

In Figura G.5 viene mostrato il listato prodotto dall'assemblatore facendo uso dell'opzione `-l <file>`; ad esempio dando il comando `nasm -f obj hw.asm -l hw.lst`. La colonna di sinistra mostra il numero di riga del programma. La seconda mostra la posizione assegnata dall'assemblatore; ovviamente ogni segmento parte dalla posizione convenzionale 0, penserà il linker ad effettuare la rilocazione. La terza colonna fornisce il contenuto delle posizioni della seconda colonna, ovvero la traduzione in linguaggio macchina del codice alla destra.

È istruttivo esaminare meglio il listato. Si noti che a `Mess` viene assegnato l'indirizzo 00000000, com'è ovvio essendo `Mess` il primo simbolo entro la sezione `data`. L'indirizzo va interpretato come "indirizzo entro la sezione `data`". Il messaggio occupa 14 byte, la cui codifica ASCII è riportata a seguire la posizione 0 e la posizione 9 (dalla posizione 9 ci sono i caratteri "`ld`", "`CR`", "`LF`" e "`$`").

Il codice inizia pure dall'indirizzo convenzionale 0. Penseranno il linker e il loader ad aggiustarlo. Alla posizione 0 si trova l'istruzione `MOV AX,data`; essa viene codificata come `B8[0000]`, dove `B8` è il codice di `MOV AX`, mentre [0000] rappresenta la componente indirizzo entro l'istruzione. Questa istruzione occupa 3 byte e quindi l'indirizzo assegnato alla prossima è 3. L'istruzione alla posizione 3 (`MOV DS,AX`) occupa invece due soli byte per cui la prossima istruzione è all'indirizzo 5. Notare che le istruzioni non sono allineate ai byte pari o a multipli di 4, in quanto l'architettura ×86 non richiede allineamento delle istruzioni.

L'istruzione alla posizione 5 ha codice `BA` e indirizza ancora la posizione 0, n quanto Mess si trova proprio in tale posizione.

Le istruzioni dei righi 28-29 e 31-32 non sono degne di particolare nota, esse sono solo la traduzione binaria della forma mnemonica.

In Figura G.6 si riporta l'immagine del video con i comandi per l'assemblatore, il linker e l'esecuzione.

```
1
2                                   ; File HW.asm  versione DOS.   Scrive Hello World
3                                   ;
4                                   ; assemblare:    nasm -f obj hw.asm
5                                   ; linking:  freelink hw   (.obj non  necessario)
6                                   ; eseguire:      hw
7                                   ;
8                                   CR    EQU       0DH  ; Carriage Return
9                                   LF    EQU       0AH  ; Line Feed
10
11                                   ;Area Dati--------------------------------
12                                   SEGMENT .data .DATA
13 00000000 48656C6C6F20576F72-      Mess DB   'Hello World',LF,CR,'$'
13 00000009 6C640A0D24
14
15                                   ;Stack-------------------------------------
16                                   SEGMENT stack64   STACK
17 00000000 <res 00000040>                   resb     64
18                                   testaSTK:
19
20                                   ; Codice-----------------------------------
21                                   SECTION .txt
22
23                                   ..start:
24 00000000 B8[0000]                         MOV AX,data     ;inizializzazione
25 00000003 8ED8                             MOV DS,AX       ; del registro DS
26
27 00000005 BA[0000]                         MOV DX,Mess  ;DS:DX indirizzo Mess
28 00000008 B409                             MOV AH,09H ;Chiamata della funzione
29 0000000A CD21                             INT 21H    ; di scrittura del DOS
30
31 0000000C B44C                     exit:   MOV AH,4CH ;uscita con ritorno a DOS
32 0000000E CD21                             INT 21H
```

Figura G.5 Listato prodotto dall'assemblatore dando il comando `nasm -f obj hw.asm -l hw.lst`.

```
D:\ASM\NASM\HELLO>nasm -f obj hw.asm   -l hw.lst
D:\ASM\NASM\HELLO>freelink hw.obj, /m hw

Freelink 2.50 Copyright 1989-92 Michael Devore and hyperkinetix, inc.
All rights reserved.


D:\ASM\NASM\HELLO>hw
Hello World
D:\ASM\NASM\HELLO>_
```

Figura G.6 Schermata sotto DOS relativa ai comandi di assemblaggio e linking e successiva esecuzione del programma.

G.6.2 Hello World sotto Linux 64 bit

Come è stato accennato al Paragrafo G.5.3 Nella versione a 64 bit di Linux, le funzioni del sistema operativo si chiamano attraverso l'istruzione `syscall`.

La funzione di scrittura (write) ha numero d'ordine 1. All'atto della chiamata si richiede che

a) il registro RDI contenga descrittore del file (fd) su cui si scrive;
b) il registro RSI contenga l'indirizzo del messaggio;
c) il registro RDX contenga il numero di byte da scrivere.

Schematicamente essa viene rappresentata come `write(fd, *messaggio, nbyte)`.

Per il dispositivo standard di uscita `fd` vale 1; ne consegue il codice Nasm riportato in Figura G.7 per l'esempio convenzionale di scrittura del messaggio "Hello World".

Il programma sorgente è stato denominato `HW64.asm`. Esso è in Figura G.7.

```
; Questo file:  HW64.asm
; Scrive a video "Hello World".
; Funziona su Linux-64 (usa syscall  --NON int 80h-- per chiamare il SO)
;
;Assemblare, collegare, eseguire:
;   nasm -f elf64 HW64.asm      (Nasm: produce HW64.o in formato elf64)
;   ld HW64.o                   (Linker: produce a.out   eseguibile)
;   ./a.out                     (Esecuzione)
;
;Piacendo, si può dare un nome diverso da a.out all'eseguibile:
;   ld HW64  -o HW64            (Produce HW64   eseguibile)
;   ./HW64                      (Esecuzione)
; ---------------------------------------------------------------------
        section .data
msg     db      'Hello World', 0Ah      ;0Ah è il "line feed"
L       equ     $-msg                   ;dimensione del messaggio


        section .text                   ; segmento di codice
        global  _start                  ; punto di entrata

_start:                                 ;WRITE(stdot,*msg,n)
        mov     rax, 1                  ; rax <- N. funzione write
        mov     rdi, 1                  ; rd1 <- N. standard output
        mov     rsi, msg                ; indirizzo del messaggio
        mov     rdx, L                  ; numero di byte
        syscall                         ; chiamata al SO

        mov     rax, 60                 ; uscita e ritorno al SO
        xor     rdi, rdi                ; con codice di non errore
        syscall                         ;  exit(0)
```

Figura G.7 Programma "Hello World" in versione Linux a 64 bit. (Nella pratica si è usato Ubuntu a 64 bit.)

Commenti

- Il file viene assemblato con la linea di comando `nasm -f elf64 HW64.asm`. L'estensione `.asm` è di per sé superflua. L'opzione `-f elf64` dice a Nasm di produrre codice in formato elf a 64 bit. Il file oggetto prodotto prende il nome `HW64.o` (l'estensione `o` è aggiunta automaticamente).
- Con il comando `ld HW64.o -o HW64` viene invocato il linker `ld` (il linker GNU). Il linker prende in ingresso il file `HW64.o` e produce il file eseguibile `HW64` avendo fatto ricorso all'opzione `-o`; se non si fosse esplicitamente indicato al linker il nome del file di uscita sarebbe stato prodotto il file eseguibile `a.out`.
  Si noti che usando il linker `ld` è essenziale che il programma contenga il simbolo `_start` definito come globale.
- Battendo `./HW64` il programma presenta a video la stringa `Hello World 64!`. Se non si fosse richiesto esplicitamente al linker di generare il file HW64, sarebbe stato generato il file eseguibile `a.out` e occorreva battere `./a.out`[10].

In Figura G.8 viene mostrato la sequenza precedente.

```
giacomo@giacomo-virtual-machine:~/Asm/HW64bit$ nasm -f elf64 HW64.asm
giacomo@giacomo-virtual-machine:~/Asm/HW64bit$ ld HW64.o -o HW64
giacomo@giacomo-virtual-machine:~/Asm/HW64bit$ ./HW64
Hello World 64!
```

Figura G.8 Sequenza di comandi relativa all'assemblaggio, al linking (tramite ld) e all'esecuzione del programma "Hello World" per la versione a 64 bit (sotto Ubuntu a 64 bit).

Osservazioni sul testo del programma HW64

Facendo riferimento a quanto è stato detto al Paragrafo "C.4.2 Modello di programmazione" dell'Appendice C, è interessante osservare quanto segue. Nella sequenza di chiamata per la funzione di uscita dal programma e ritorno al sistema operativo, la funzione è data col numero 60 in RAX. Se in luogo di RAX si fosse usato EAX, ovvero si fosse scritto

```
mov     eax, 60
syscall
```

Il programma avrebbe funzionato regolarmente, in quanto nel funzionamento a 64 bit, il caricamento di EAX pone tutti zeri nella parte di RAX alla sinistra di EAX stesso. Dunque, qualunque sia il contenuto di RAX, dopo `mov eax,60`, RAX contiene 60. Se invece si fosse scritto `mov ax, 60`, o anche `mov al, 60` e la parte a sinistra di RAX avesse contenuto un valore diverso da 0, il contenuto di RAX dopo il MOV non sarebbe 60 (nel caso del `mov ax` sarebbe il precedente contenuto dei bit RAX_{63-16} con il 60 in RAX_{7-0}); nel caso del `mov al` sarebbe il precedente contenuto dei bit RAX_{63-8}, con il 60 in RAX_{7-0}). La chiamata alla funzione darebbe quasi sicuramente luogo a errore.

[10] Il nome del programma eseguibile deve essere preceduto da "`./`" per indicare di cercare nella cartella corrente (assumendo che essa non sia nel PATH. È questa una convenzione di Linux.

Nella Figura G.7 i dati sono stati tenuti separati dal codice. Come abbiamo visto al Paragrafo 3.5.1 del libro, Linux ha un modello di memoria lineare, dunque la suddivisione non è di per sé necessaria. In Figura G.9 si mostra come il programma può essere semplicemente ristrutturato in modo da stare nella sola sezione di testo.

```
        section .text                   ; Codice
        global  _start                  ; Definisce il punto di entrata

_start: mov     rax, 1                  ; WRITE(stdot,*msg,n)
        mov     rdi, 1                  ;  standard output
        mov     rsi, msg                ;   indirizzo del messaggio
        mov     rdx, l                  ;    numero di byte
        syscall                         ; chiamata al SO

        mov     rax, 60                 ; uscita e ritorno al SO
        xor     rdi, rdi                ; con codice di non errore
        syscall                         ;  exit(0)

msg     db      'Hello World!', 10
l       equ     $-msg
```

Figura G.9 Il programma Hello World (versione Linux 64 bit), scritto in modo da comprendere tutto nella sola sezione TXT.

Il linker

Abbiamo usato il linker `ld` di GNU. Questo linker pretende che venga definito come punto di accesso al programma il simbolo globale `_start`.

In previsione del collegamento dei programmi scritti in assembler con quelli scritti in C, conviene tener conto del fatto che il compilatore `GCC` che useremo tra poco per compilare i programmi C, può anche essere impiegato in funzione di linker. L'uso di GCC come linker è molto conveniente. Anzitutto GCC ricorre a `ld` per la funzione di *linkage*, in modo invisibile al programmatore. Il vantaggio di usare GCC è che esso predispone la sua chiamata a `ld` in modo da fornire un insieme di opportune informazioni (che altrimenti dovrebbe dare il programmatore).

Nell'usare `ld` o GCC in funzione di linker c'è tuttavia una piccola complicazione. E' noto che in C il programma principale ha il punto di entrata definito come `main`. Supponiamo di modificare il programma di Figura G.7 sostituendo le righe

```
        global  _start                  ; punto di entrata
_start:
```

con queste

```
        global  main                    ; punto di entrata
main:
```

Apparentemente non dovrebbero esserci differenze, ma se si dà il comando `ld HW64.o` (dopo aver assemblato), si ottiene un messaggio di errore che informa della mancanza il simbolo `_start`, simbolo che definisce il punto a cui il sistema operativo passa il controllo.

Se invece torniamo alla versione originale di Figura G.7, ma usiamo GCC come linker, dando il comando `gcc -o HW64 HW64.o` (che dovrebbe produrre l'eseguibile `HW64` dal file oggetto `HW64.o`) si ottiene un messaggio di errore, che invece dice "definizione

multipla di _start".

Il motivo di questa apparente discordanza è che GCC (si ricordi che è fatto per il C!) assume come punto di accesso `main`, e quindi genera automaticamente un suo tratto di codice, a precedere quello del programma, entro il quel definisce `_start`. Per tale motivo, il simbolo `_start` finisce per risultare definito due volte . Per ovviare all'inconveniente, se si vuole usare GCC come linker con un programma dove il punto di ingresso è definito come `_start` è necessario usare l'opzione `-nostartfiles`, prima del nome dei file, in modo che gcc non generi la definizione di `_start`. Si veda la Figura G.10.

```
giacomo@giacomo-virtual-machine:~/Asm/HW64bit$ gcc -nostartfiles -o HW64  HW64.o
giacomo@giacomo-virtual-machine:~/Asm/HW64bit$ ./HW64
Hello World 64!
```

Figura G.10 Uso di GCC come linker con un programma oggetto in cui è definito il simbolo `_start` (come in Figura G.7 e G.9). L'opzione `-nostartfiles` dice a GCC di non generare il punto di entrata `_start`, essendo questo già definito nel programma. L'opzione deve precedere i nomi dei file nella linea di comando.

Se invece il programma definisse `main` come punto di entrata allora il comando di link `gcc -o HW64 HW64.o` sarebbe corretto. GCC genererebbe il codice necessario a definire `_start` (che provvede a passare il controllo a `main`) prima di chiamare al suo interno il linker `ld`.

Osservazione
Gli argomenti sul linker sono stati esposti solo al fine di semplificare la vita al lettore, mettendolo in grado di assemblare, collegare, ecc. Come al solito per una effettiva conoscenza dei sistemi operativi è necessario leggere la documentazione relativa e sperimentare.

G.6.3 Hello World sotto Linux, architettura a 32 bit

La differenza rispetto alla versione a 64 bit sta tutta nel modo in cui viene chiamato il sistema operativo. Le convenzioni di chiamata sono in Tabella G.3.

Si rimarca il fatto l'esperimento è stato condotto su una macchina a 64 bit. Ciò è possibile per due motivi: (a) l'architettura 1*time*86 a 64 bit è in grado di eseguire programmi a 32 bit in modo "compatibile"; (b) la versione Ubuntu di Linux a 64 bit, dispone delle librerie di Linux a 32 e pertanto consente le chiamate nella forma da esse previste. Ovviamente è necessario che vengano scelte le giuste opzioni nelle linee di comando.

La scrittura (Tabella G.3) ha numero d'ordine 4, i parametri si passano nei registri come indicato nel testo di Figura G.11. La chiamata si effettua attraverso l'istruzione `INT 80h`.

Il programma sorgente è stato denominato `HW64.asm`. I commenti riportati indicano i comandi da dare per assemblare, collegare ed eseguire. L'assemblatore Nasm per Linux assume come default che `elf` sia a 32 bit. In modo opposto, funzionando la macchina su un sistema a 64 bit, al linker deve essere esplicitamente detto che il formato del file oggetto è in versione 32 bit (`-m elf_i386`).

```
;File HW32.asm
;
; Assemblaggio: nasm -f elf HW32.asm
; Link (i sistemi a 64 bit richiedono l'opzione elf_i386):
;              ld -m elf_i386 HW32.o -o HW32
; Esecuzione: ./HW32

        SECTION .data
msg     db      'Hello World 32!', 0Ah
L       equ     $-msg


SECTION .text
        global  _start
_start:
        mov     edx, L      ;dimensione del messaggio
        mov     ecx, msg    ;puntatore al messaggio
        mov     ebx, 1      ;stdout
        mov     eax, 4      ;WRITE
        int     80h

        mov     ebx, 0      ; restituisce 0: 'Nessun errore'
        mov     eax, 1      ; Ritorno al SO
        int     80h
```

Figura G.11 Programma "Hello World" in versione Linux a 32 bit.

```
giacomo@giacomo-virtual-machine:~/Asm/HW32bit$ nasm -f elf HW32.asm
giacomo@giacomo-virtual-machine:~/Asm/HW32bit$ ld -m elf_i386 HW32.o -o HW32
giacomo@giacomo-virtual-machine:~/Asm/HW32bit$ ./HW32
Hello World 32!
```

Figura G.12 Comandi per la versione di "Hello Word" a 32 bit (su Ubuntu a 64 bit). Si osservi che Nasm assume che il formato `elf` del codice oggetto sia a 32 bit, mentre il linker richiede espressamente che venga indicato che il codice è a 32 bit (opzione `-m elf_i386`).

G.7 Inversione ordine: il secondo programma di esempio

Lo scopo di questo esempio è duplice

a) introdurre le chiamate ai sottoprogrammi e il passaggio dei parametri;
b) introdurre le macro.

Il programma dovrà esibire le seguenti funzionalità:

1) acquisire una stringa di caratteri numerici da tastiera (la stringa termina quando viene battuto il carattere di "Invio");
2) se la stringa immessa è costituita da sole cifre numeriche, allora presentare a video la stringa con l'ordine dei caratteri invertito;
3) se la stringa immessa non contiene esclusivamente cifre numeriche, allora presentare a video il messaggio `Stringa non numerica`;
4) si viene battuto solo il carattere di Invio, allora presentare a video il messaggio `Non è stato battuto alcun carattere`.

I sottoprogrammi

Il concetto di sottoprogramma è ben noto. In questo esempio si definisce un sottoprogramma all'interno del file che contiene il programma chiamante. In Nasm si tratta semplicemente di usare l'istruzione `CALL` seguita dal nome dato all'etichetta da cui inizia il sottoprogramma. Nell'esempio viene costruita la routine `_Inverti`.

Nel chiamare una subroutine occorre normalmente passare uno o più parametri di ingresso. Il modo più ovvio, specialmente quando si programma in assembler e non ci sono da rispettare regole di compatibilità con programmi scritti in linguaggi di alto livello, consiste nel passare i parametri attraverso i registri di CPU (si veda il testo del programma di Figura G.15 e della subroutine in Figura G.16).

Le macro

Le macro sono state descritte al Paragrafo G.2.1, con la sintassi di Masm.

In Nasm la sintassi è differente. Una macro viene definita utilizzando la coppia di direttive `%macro/%endmacro`, includendo tra le due la sequenza di istruzioni che si intende accorpare come macro e dando accanto al nome il numero dei parametri. La sintassi è questa:

```
%macro     Nome   [N. Parametri]
           BloccoStatement
%endmacro
```

Un esempio sarà utile. Consideriamo la seguente definizione della macro WRITELN, scritta in riferimento al DOS, per presentare a video un messaggio

```
\%macro  WRITELN 1
           MOV  DX,%1
           MOV  AH,09H
           INT  21H
\%endmacro
```

La macro ha un parametro formale. Lo statement `MOV DX,%1` stabilisce che al posto di `%1` (cioè del primo e unico parametro) verrà sostituito il simbolo che apparirà accanto a WRITELN nel testo del programma. Ad esempio:

```
        WRITELN Mess
```

verrebbe espanso con questa sequenza equivalente[11]:

```
MOV  DX,Mess
MOV  AH,09H
INT  21H
```

Quando nel programma si dovrà presentare un messaggio di testo a video, basterà scrivere all'occorrenza `WRITELN`, con accanto il nome dato alla posizione di partenza del messaggio.

Nel caso in cui i parametri sia più di 1, ad esempio 3, la sostituzione avviene ordinatamente. Ad esempio, nella macro definita come macro 3, ipotizzando che venga chiamata scrivendo `macro a, b, c`, quando nel corpo della macro compare %2 viene sostituito con `b`.

G.7.1 Inversione d'ordine: versione DOS

Poiché il programma prevede la scrittura e la lettura di messaggi dal terminale, cominciamo con la definizioni di due macro, una per scrivere e una leggere una linea, come in Figura G.13. Ambedue hanno un solo parametro: l'indirizzo dell'area di memoria il cui contenuto è, rispettivamente, da presentare o da riempire. La figura mostra anche una terza macro per tornare al DOS, questa senza parametri.

```
; Questo file:  MACRO.MAC
;

\%macro  WRITELN 1
           MOV  DX,\%1
           MOV  AH,09H
           INT  21H
\%endmacro

\%macro  READLN  1
           MOV  DX,\%1
           MOV  AH,0AH
           INT  21H
\%endmacro

\%macro  EXIT   0
           MOV  CAH
           INT  21H
\%endmacro
```

Figura G.13 Il file contenente la definizione di tre macro WRITELN, READLN e EXIT (versione DOS).

Per quanto si riferisce alla macro `WRITELN` basta rifarsi a quanto detto in precedenza circa la convenzione del DOS riguardo al formato dell'area contenente il messaggio da presentare.

Per quanto si riferisce alla macro `READLN`, occorre tener conto che la funzione DOS 0AH impone che l'area di memoria abbia il formato schematizzato in Figura G.14, dove `BUFF` è la prima posizione dell'area di memoria. All'atto della chiamata della funzione 0AH, questa posizione deve contenere un valore (`NMAX`) indicante il massimo numero di caratteri introducibili

[11] Abbiamo detto sequenza equivalente perché sul listato prodotto da Nasm la sostituzione non appare esplicitamente.

da tastiera (comprensivo del carattere CR, corrispondente a "Invio" sulla tastiera). Ciò implica che non possono essere introdotti più di NMAX-1 caratteri di testo effettivo. La seconda posizione viene impiegata dal DOS per scrivere il numero n di caratteri effettivamente introdotti. Questi si troveranno in BUFF a partire dalla posizione BUFF+2, indicata come car1 nello schema di Figura G.14. Ovviamente, n non supererà NMAX-1: nel caso che vengano battuti più caratteri, vengono caricati i primi NMAX-1 e il "CR" finale.

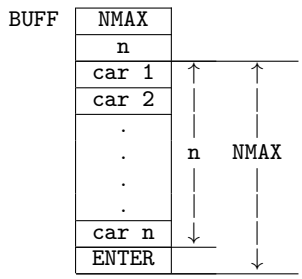

Figura G.14 Formato dell'area di memoria per la funzione di lettura (0AH) del DOS. Prima della chiamata, nella posizione iniziale dell'area in cui si troveranno, deve essere scritto il numero massimo (NMAX) di caratteri che possono essere messi nell'area stessa. La posizione successiva viene usata dalla funzione 0AH per dare il numero (n) dei tasti effettivamente battuti. La figura, tracciata per il caso in cui l'area sia completamente piena, fornisce la relazione tra NMAX e n. Nel caso in cui vengano battuti un numero di caratteri inferiore a NMAX-1, il carattere Enter si trova dopo l'ultimo carattere di testo battuto. Nel caso in cui vengano battuti più di NMAX caratteri, nel buffer si trovano i primi n-1, con in ultima posizione ENTER (CR).

Il testo del programma (Figura G.15) inizia con lo statement `%include "macro.mac"` che fa includere il file `macro.mac`, come se fosse una parte del programma stesso. Il carattere "%" è diretto al preprocessore, interno a Nasm, responsabile dell'espansione delle macro. Le righe successive, fino alle prime righe della sezione TXT, sono simili a quelle del programma Hello World.

Nel segmento Data vengono definiti due messaggi in più (Mess2, e M_err). Viene definita l'area Buff entro la quale il DOS dovrà depositare i caratteri introdotti da tastiera e l'area Dest nella quale vengono spostati in caratteri in modo che siano in ordine inverso.
Si noti che Nmax e n sono definiti come scostamenti (posizione) entro Buff. Nmax andrà inizializzato prima della chiamata alla READLN, mentre in n si troverà il numero dei caratteri battuti[12].

Il blocco di statement che inizia con WRITELN Mess e termina con JMP Fine ha lo scopo di verificare se è stato immesso almeno un carattere di testo.

Il tratto di codice che inizia da Go: chiama la routine _Inverti. Prima della chiamata vengono caricati nei registri i parametri di ingresso alla routine (in SI l'offset della posizione dell'ultimo carattere introdotto; in DI l'offset della prima posizione di Dest; in CX il numero di caratteri introdotti). Al ritorno da _Inverti viene effettuata la verifica circa il fatto che _Inverti abbia trovato solo caratteri numerici (AL=0), ovvero abbia trovato almeno un carattere non numerico (AL≠0). Nel caso che non ci sia errore, viene scritto il contenuto di Dest,predisposto da _Inverti.

Per quanto riguarda la routine _Inverti essa viene riportata a parte nella Figura G.16 (per motivi tipografici, ma essa fa parte del file invdos.asm). Si faccia attenzione a cosa contengono i registri in ingresso e al fatto che in uscita la routine indica se ha trovato solo caratteri numerici

[12]Saprebbe il lettore definire Buff in modo che la posizione iniziale contenga NMAX senza dover inizializzare?

```
;Questo file  invdos.asm        (Inversione DOS)
%include "macro.mac"

CR EQU  0DH
LF EQU  0AH
; Dati ---------------------------------------------------------------
            segment Data
Mess        DB     'Inserire una stringa numerica >', '$'
Mess2       DB LF,CR,'Non  stato battuto alcun carattere','$'
M_err       DB 'Stringa non numerica',CR,LF,'$'
Buff        resb    30      ;Qui vengono letti                      ;
Nmax        equ     0
n           equ     1
Dest        resb    30      ;Qui vengono messi in ordine inverso
; Stack  -------------------------------------------------------------
            segment     STACK STACK
            resb 10
topSTK:
; Codice -------------------------------------------------------------
            section .txt
..start:
            MOV  AX,Data
            MOV  DS,AX              ;DS -> Data

            WRITELN Mess                ;Stampa a video di Mess
            mov     byte [Buff],28      ;Inizializ. Nmax
            READLN  Buff                ;Lettura stringa

            mov     CX,0                ;CH<-0; CL<-0
            MOV     CL,[Buff+n]         ;CX<- n
            CMP     CL,0                ;almeno 1 car?
            JNE     Go                  ; se si continua
            WRITELN  Mess2              ;Presenta 'Non  stato battuto...'
            JMP Fine
Go:
            MOV  SI,Buff+n
            ADD  SI,CX                  ;SI->ultimo introdotto
            MOV  DI,Dest
            CALL  _Inverti              ;Chiamata sottoprogramma
            CMP  AL,0                   ;Stringa
            JNE  Errore                 ; numerica ?

            WRITELN Dest                ;  sì: presentazione
Fine:       EXIT                        ;Uscita

Errore:     WRITELN M_err               ;   no: stampa 'Stringa non numerica',
           JMP  Fine
```

Figura G.15 Programma per l'inversione dell'ordine di una stringa (versione sotto DOS). Questa è la parte costituente il programma principale. La parte corrispondente alla routine `_Inverti` è riportata in Figura G.16 per ragioni tipografiche, ma essa fa parte del file `INVDOS.ASM`.

```
;Sottoprogramma _Inverti    (parte di invdos.asm)
;----------------------------------------------------
_Inverti:
;
;In ingresso a Inverti
;  SI: offset della posizione dell'ultimo carattere introdotto
;  DI: offset della prima posizione di Dest
;  CX: numero di caratteri introdotti
;

            MOV     AL,LF
            MOV     [DI],AL
            INC     DI
            MOV     AL,CR
            MOV     [DI],AL
            INC     DI

Ciclo:      MOV     AL,[SI]
            CMP     AL,'0'
            JL      Exit            ;Non numerico
            CMP     AL,'9'
            JG      Exit            ;Non numerico
            MOV     [DI],AL         ;Numerico
            INC     DI
            DEC     si
            LOOP    Ciclo

            MOV     al,CR
            mov     [di],al
            inc     DI
            mov     al,LF
            mov     [di],al
            inc     DI
            MOV     AL,'$'          ;Aggiunta del '$'
            MOV     [DI],AL         ; a fine stringa
            MOV     AL,0            ;restituisce AL=0 :  OK!
Exit:       RET
;fine del file invdos.asm
```

Figura G.16 Sottoprogramma che effettua l'inversione della stringa. Questa subroutine restituisce 0 in AL se l'inversione è stata eseguita. Se la stringa contiene caratteri non numerici il sottoprogramma restituisce AL$\neq$0.

o anche caratteri non numerici; nel secondo caso il programma principale presenta il messaggio "Stringa non numerica".

In Figura G.17 viene riportata la schermata comprensiva dei comandi di assemblaggio, linking ed esecuzione. Il programma è stato fatto correre per tre volte distinte, provando i tre possibili casi.

```
D:\ASM\NASM\INV>nasm -f obj invdos.asm -l l.lst
D:\ASM\NASM\INV>freelink invdos

Freelink 2.50 Copyright 1989-92 Michael Devore and hyperkinetix, inc.
All rights reserved.


D:\ASM\NASM\INV>invdos
Inserire una stringa numerica >1234567890
0987654321
D:\ASM\NASM\INV>
D:\ASM\NASM\INV>invdos
Non  stato battuto alcun carattere
D:\ASM\NASM\INV>invdos
Inserire una stringa numerica >12345y67890
Stringa non numerica
D:\ASM\NASM\INV>
```

Figura G.17 Sequenza dei comandi di assemblaggio, linking ed esecuzione (tre possibili casi). Versione DOS.

Al fine di mostrare come le macro vengono espanse, in Figura G.18 vengono mostrati tre tratti del listato (file `l.lst`) prodotto dall'assemblaggio.

Lo statement `%include` determina l'inclusione del file delle macro che viene mostrato nelle prime 12 righe. La righa 28 (ripetuta) corrisponde all'espansione della macro `WRITELN`. Analogamente le righe 30, 36 e 37 sono espansioni di macro.

Vale la pena di fare alcune osservazini sul codice prodotto dall'assemblatore.

Notare che all'indirizzo 5 si trova l'istruzione `MOV DX,Mess` codificata esattamente come in Figura G.18, essendo `Mess` nella medesima posizione.

All'indirizzo 22 si trova l'istruzione `JNE Go`, codificata come `7509`, dove 75 è il codice di `JNE`, mentre 09 è lo scostamento tra `2D` (posizione di `JNE Go`) e `24` posizione dell'istruzione successiva a `JNE`. Si tratta di un salto relativo rispetto a IP. Al tempo di esecuzione la CUP somma l'IP corrente (che è già aggiornato a quello dell'istruzione successiva (24), con lo scostamento contenuto nell'istruzione (9), per ottenere l'indirizzo di destinazione (2D).

Al rigo 37 c'è l'istruzione `JMP Fine`, in posizione `2B`, codificata come `EB16`. `EB` è il codice del salto incondizionato relativo; `16` è la distanza rispetto alla destinazione (43-2D = 16, in esadecimale).

```
     1                                  ;Questo file  invdos.asm
     2                                  %include "macro.mac"
     1                              <1>
     2                              <1> %macro  READLN  1
     3                              <1>     MOV     DX,%1
     4                              <1>     MOV     AH,0AH
     5                              <1>     INT     21H
     6                              <1> %endmacro
     7                              <1>
     8                              <1> %macro  WRITELN 1
     9                              <1>     MOV     DX,%1
    10                              <1>     MOV     AH,09H
    11                              <1>     INT     21H
    12                              <1> %endmacro
 :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
    28                                      WRITELN Mess ;Stampa a video di Mess
    28 00000005 BA[0000]            <1>   MOV   DX,%1
    28 00000008 B409                <1>   MOV   AH,09H
    28 0000000A CD21                <1>   INT   21H
    29 0000000C C606[5D00]1C                mov     byte [Buff],28
    30                                      READLN  Buff ;Lettura stringa
    30 00000011 BA[5D00]            <1>   MOV   DX,%1
    30 00000014 B40A                <1>   MOV   AH,0AH
    30 00000016 CD21                <1>   INT   21H
    31
    32 00000018 B90000                      mov     CX,0
    33 0000001B 8A0E[5E00]                  MOV     CL,[Buff+n]
;E' stato introdotto
    34 0000001F 80F900                      CMP     CL,0 ;almeno 1 car?
    35 00000022 7509                        JNE     Go ; se si continua
    36                                      WRITELN  Mess2
    36 00000024 BA[2000]            <1>   MOV   DX,%1
    36 00000027 B409                <1>   MOV   AH,09H
    36 00000029 CD21                <1>   INT   21H
    37 0000002B EB16                        JMP     Fine
    38                                  Go:
    39 0000002D BE[5E00]                    MOV     SI,Buff+n
  :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
    47                                  Fine: EXIT
    47                              <1> Fine:
    47 00000043 B44C                <1>     mov ah,4ch
    47 00000045 CD21                <1>     int 21h
  :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

Figura G.18 Parti del listato che mostrano l'espansione della macro (versione DOS).

G.7.2 Inversione d'ordine: versione Linux a 32 bit

Qui di seguito si mostra la versione Linux del programma per invertire una stringa numerica già mostrato in versione DOS al Paragrafo G.7.1.

Com'è ovvio le differenze sono dovute essenzialmente al modo in cui vengono chiamate le funzioni del sistema operativo. Cominciamo con il presentare le macro relative alla versione Linux a 32 bit (Figura G.19).

```
;;; questo file: macro32.mac            (versione 32 bit Linux)
;
%macro  READLN  2                       ;READ(*Buff,nMax)
        mov     ecx, %1                 ; ecx <- indirizzo del messaggio
        mov     edx, %2                 ; edx <- numero Max di caratteri
        mov     ebx, 1                  ; ebx <- stdin
        mov     eax, 3                  ; read
        int     80h
%endmacro

%macro  WRITELN 2                       ;WRITE(*buff,n)
        mov     ecx, %1
        mov     edx, %2
        mov     ebx, 1
        mov     eax, 4                  ;write
        int     80h
%endmacro


%macro  EXIT    0
        mov     eax, 1                 ; system ritorno al dos
        xor     ebx, ebx         ; nessun errore
        int 80h
%endmacro
```

Figura G.19 Macro per la chiamata di funzioni del SO Linux, versione a 32 bit.

Si evidenzia che la funzione di lettura (read=3) prende come parametro in EDX il numero massimo (nMax) di caratteri che possono essere introdotti. Al ritorno in EAX c'è il numero dei caratteri effettivamente battuto su tastiera, questo numero è comprensivo del carattere "Invio". In sostanza:

- se viene battuto il solo Invio, in EAX ritorna 1;
- se vengono battuti nMax caratteri, in EAX torna nMax+1; i caratteri (compreso Invio) sono in Buff;
- se vengono battuti n<nMax caratteri, in EAX torna n+1; i caratteri (compreso Invio) sono in Buff;
- se vengono battuti n>nMax caratteri, in EAX torna nMax+1; in Buff vengono caricati i primi nMax+1 caratteri (compreso Invio). I caratteri battuti in più rispetto a nMax vengono tralasciati.

La versione Linux del programma di inversione a 32 bit, specificato come all'inizio del Paragrafo G.7 è in Figura G.20 (la parte corrispondente al programma principale) e in Figura G.21 (per la parte corrispondente alla routine `_Inverti`, che, si tenga a mente, è comunque parte del file `Inv32.asm`).

```
;Questo file:  Inv32.asm     (inversione stringa Linux 32)
;
%include "macro32.mac"      ;Inclusione macro

CR      EQU  0DH
LF      EQU  0AH
        section .data    ;-----------------------------------------
Mess    DB     'Inserire una stringa numerica >'
Mess2   DB     'Non è stato battuto alcun carattere', LF
LMess   equ     Mess2-Mess
LMess2  equ     $-Mess2
M_err   DB      'Stringa non numerica', 0Ah
LM_err  equ     $-M_err

        section .bss     ;-----------------------------------------
Buff    resb    30
Nmax    equ     30
Dest    resb    32

        section .text ;-----------------------------------------
        global  _start ; Definisce il punto di entrata
_start:
        WRITELN Mess, LMess    ;scrive 'Inserire una stringa numerica >'
        READLN  Buff,Nmax      ;lettura
        MOV     ECX,EAX        ;EAX= n. car introdotti
        CMP     EAX,1          ;Solo "enter" ?
        JNE     Go             ; se si: continuare
        WRITELN Mess2,LMess2   ;scrive 'Non è stato battuto alcun carattere'
        JMP     Fine
Go:
        MOV     ESI,Buff
        Add     ESI,ecx
        sub     ESI,2           ;SI->ultimo introdotto
        sub     ecx,1
        MOV     EDI,Dest

        CALL  _Inverti
        CMP     AL,0
        JNE     Errore          ;Numerica ?
        WRITELN Dest,30         ; sì: scrivi stringa riordinata
Fine:   EXIT

Errore: WRITELN M_err,LM_err    ;Stampa 'Stringa non numerica'
        JMP     Fine
```

Figura G.20 Programma per l'inversione di una stringa numerica in versione Linux 32bit

```
_Inverti:                   ; sottoprogramma versione Linux 32
;
;  In ingresso:
;  ESI: offset della posizione dell'ultimo carattere introdotto escluso LF
;  EDI: offset della prima posizione di Dest
;  ECX: numero di caratteri introdotti
;

Ciclo:
        MOV     AL,byte [ESI]
        CMP     AL,'0'
        JL      Exit                    ;Non numerico
        CMP     AL,'9'
        JG      Exit                    ;Non numerico
        MOV     byte [EDI],AL           ; in Dest

        INC     EDI
        DEC     Esi
        dec     ECX
        jne     Ciclo

        mov     al,LF
        mov     byte [EDI],al
        MOV     AL,0                    ; OK!
Exit:   RET

;fine del file Inv32.asm
;
```

Figura G.21 Routine `_Inverti` (Linux a 32 bit).

```
giacomo@giacomo-virtual-machine:~/Asm/Riordino$
giacomo@giacomo-virtual-machine:~/Asm/Riordino$ nasm -f elf Inv32.asm
giacomo@giacomo-virtual-machine:~/Asm/Riordino$ ld -m elf_i386 Inv32.o
giacomo@giacomo-virtual-machine:~/Asm/Riordino$ ./a.out
Inserire una stringa numerica >123456
654321
giacomo@giacomo-virtual-machine:~/Asm/Riordino$ ./a.out
Inserire una stringa numerica >
Non è stato battuto alcun carattere
giacomo@giacomo-virtual-machine:~/Asm/Riordino$ ./a.out
Inserire una stringa numerica >7654t012
Stringa non numerica
giacomo@giacomo-virtual-machine:~/Asm/Riordino$
```

Figura G.22 Esempio di assemblaggio, linking, esecuzione (versione a 32 bit, su Linux a 64 bit).

G.8 Moduli di programma separati

Negli esempi precedenti la routine `_Inverti` era parte dello stesso programma principale. Nella pratica della programmazione, quando il testo del programma diventa troppo lungo e quindi ingestibile, è buona regola suddividerlo in moduli, in modo da poter assemblare/compilare ciascun modulo in maniera indipendente. A volte si parla di programmazione modulare. Una volta che un modulo è stato realizzato e verificato esso può essere riusato anche in più applicazioni che richiedano le funzioni che esso implementa.

Assumiamo di voler sviluppare la routine `_Inverti` come modulo di programma distinto, soggetto ad assemblaggio indipendente. La routine verrà chiamata dal programma principale, dal quale essa viene separata. Per rendere possibile la chiamata è necessario che nel modulo del programma principale la routine venga definita come "esterna", mentre nel modulo della routine `_Inverti` stessa dovrà essere definita come "globale" per poter essere vista dall'esterno. Mostriamo come in riferimento Linux. Chiamiamo

- `Inv32X.asm` il file contenente nuovo programma principale. Esso dovrà contenere lo statement `extern _Inverti` per dichiarare che `_Inverti` è un simbolo definito esternamente a `Inv32X.asm` (Figura G.23).
- `SubInv.asm` il file contenente la routine `_Inverti`. Esso dovrà contenere lo statement `global _Inverti` che rende il simbolo `_Inverti` globale, ovvero visibile dall'esterno al modulo `SubInv.asm` (Figura G.24);

Il programma principale presenta la sola variazione dovuta alla dichiarazione di `_Inverti` come simbolo esterno. Per tale motivo, in Figura G.23, si è evitato di ripeterne il testo. Lo stesso criterio si è adottato in Figura G.24 per quanto riguarda la subroutine `_Inverti`.

```
;;Questo file:  Inv32X.asm
;
        extern      _Inverti             ;_Inverti simbolo esterno
;
; ;;;; questo tratto del tutto identico al precedente Inv32.asm
; ;;;; ovviamente con l'esclusione di _Inverti
;
;Fine del file Inv32X.asm
```

Figura G.23 Programma principale. A parte lo statement che dichiara `_Invert`, il testo del programma è del tutto identico a quello di Figura G.20. Per tale motivo non è stato riportato.

La Figura G.25 mostra che i due moduli `Inv32X.asm` e `SubInv.asm` sono stati soggetti ad assemblaggio separato. Al Linker sono stati dati i due moduli oggetto corrispondenti (`Inv32X.o` e `SubInv.o`). Al linker è stato ordinato di produrre il file eseguibile `invx`. Le righe successive mostrano l'invocazione di `invx` e la risposta con una corretta stringa di caratteri numerici.

```
;-----------------------------------------------------------------------
;
;;;Questo file SubInv.asm
;
        global      _Inverti            ;_Inverti simbolo globale
LF      equ         0AH
        section .txt
_Inverti:
;
; Il tratto da qui alla fine è del tutto identico
; a quello contenuto nel file Inv32.asm
;
;;;Fine del file SubInv.asm.
```

Figura G.24 Parte variata del sottoprogramma `_Inverti`, organizzato come modulo distinto. `_Inverti` è definito come simbolo globale. Notare che è stato necessario aggiungere la definizione del simbolo LF usato verso la fine del testo, affinché essa fosse presente nel modulo.

```
giacomo@giacomo-virtual-machine:~/Asm/Inversione$
giacomo@giacomo-virtual-machine:~/Asm/Inversione$
giacomo@giacomo-virtual-machine:~/Asm/Inversione$
giacomo@giacomo-virtual-machine:~/Asm/Inversione$ nasm -f elf Inv32X.asm
giacomo@giacomo-virtual-machine:~/Asm/Inversione$ nasm -f elf SubInv.asm
giacomo@giacomo-virtual-machine:~/Asm/Inversione$ ld -m elf_i386 Inv32X.o SubInv.o -o invx
giacomo@giacomo-virtual-machine:~/Asm/Inversione$ ./invx
Inserire una stringa numerica >456789
987654
giacomo@giacomo-virtual-machine:~/Asm/Inversione$
```

Figura G.25 Sequenza di comandi che mostra l'assemblaggio di due moduli distinti e il loro collegamento (versione a 32 bit). Il Linker collega i due moduli `Inv32X.o` e `SubInv.o`, producendo come uscita (opzione `-o`) il file eseguibile `invx`.

G.9 Collegamento con linguaggi di alto livello

Nella pratica professionale si programma con linguaggi di alto livello. La programmazione in assembler è di norma limitata allo sviluppo di tratti specifici di codice, quando ragioni speciali, ad esempio motivi di efficienza, la fanno preferire. Si pone quindi il problema di come costruire sistemi software composti da tratti di codice ottenuti mettendo assieme parti di programma compilate con parti assemblate.
Mostreremo questi aspetti in riferimento al linguaggio C, supponendo di operare su macchina Linux a 64 bit. A tal fine, faremo prima un semplice programma C che scrive "Hello World", senza passaggio dei parametri; il programma sarà suddiviso in programma principale e routine di stampa, che poi verranno trasformati in assembler in modo da chiamarsi l'un l'altro. Successivamente, mostreremo con un esempio il passaggio dei parametri.

G.9.1 Collegamento in assenza di parametri

Il programma che scrive "Hello World" in C è banale; tutto si riduce all'uso della funzione `printf` facente parte della libreria standard di ingresso/uscita del C. Il compilatore GCC produce direttamente il codice eseguibile.

```
/* file hw0.c        compilazione:  gcc hw0.c -o hw0   */
#include <stdio.h>
int main ()
{
  printf ("Hello World C!\n");
  return 0;
}
```

L'esecuzione di `hw0` fa apparire la stringa "Hello World" a video.

Suddivisione

Se si deve arrivare a collegare parti di programma in C con parti in assembler è ovvio che si devono avere compilazioni/assemblaggi separati. Per questo motivo modifichiamo il precedente programma suddividendolo nel programma principale (`main.c`, nel file `hw2.c`) e nel sottoprogramma `print.c` (file `print.c`). Il programma principale e il sottoprogramma prendono la forma di Figura G.26. Si noti che è necessario prevedere anche il file "`header`" con il quale si dà il prototipo della funzione `print`.

Essendo `hw2.c` e `print.c` moduli distinti, da collegare dopo la loro compilazione, occorre dare al compilatore GCC l'opzione `-c`, con la quale viene richiesta la sola compilazione senza generazione dell'eseguibile.
Con la linea di comando

```
gcc -c hw2.c
```

viene generato il modulo oggetto `hw2.o`. Con la linea di comando

```
gcc -c print.c
```

viene generato il modulo oggetto `print.o`. A questo punto i due moduli possono essere collegati con questa linea di comando

```
gcc -o hw2  hw2.o print.o
```

Il risultato è la generazione del modulo eseguibile `hw2`.
Battendo `./hw2` sul video appare `Hello World C!`

```
/* file print.h */
void print();

/* file hw2.c */
#include <stdio.h>
#include "print.h"
int main()
{
  print();
  return 0;
}


/* file print.c */
#include <stdio.h>
void print ()
{
  printf ("Hello World C!\n");
  return;
}
```

Figura G.26 Suddivisione in programma principale e sottoprogramma. Il file print.h si rende necessario per dare al programma principale il prototipo della funzione `print`.

Chiamata da assembler di sottoprogramma C

Mantenendo ferma la funzione `print` di Figura G.26, costruiamo in sostituzione di `main.c` C, la corrispondente versione assembler (`hw2.asm`) che chiama la `print` di Figura G.26. Il programma `hw2.asm` è in Figura G.27 (tra i commenti le linee di comando).

```
;file hw2.asm
;      Assemblare:   nasm -f elf64 hw2.asm  -o hw2.o
;      Collegare:    gcc hw2.o  print.o  -o hw2a
section .text
        global  main         ; Definisce il punto di entrata
        extern  print        ; print è esterna
main:
        call    print
        mov     rax, 60       ; system call 60: EXIT
        xor     rdi, rdi      ; codice in uscita: 0
        syscall               ; ritorno al SO
```

Figura G.27 Programma principale in versione assembler per la chiamata della funzione `print` scritta in C. Notare la dichiarazione di `print` come simbolo esterno.

All'assemblatore viene fatto generare il modulo oggetto `hw2a.o`, in formato `elf64` (ricordarsi che il programma viene fatto girare sotto Ubuntu a 64 bit); GCC viene impiegato come linker, facendogli generare `hw2a`.

Battendo `./hw2a` sul video appare `Hello World C!`

Chiamata da C di sottoprogramma assembler

Questa volta sviluppiamo il sottoprogramma in assembler e lo chiamiamo dal il programma principale C di Figura G.26. Chiamiamo `printa` la routine di stampa in assembler. Il programma principale di Figura G.26 richiede una piccola modifica in modo da chiamare `printa` (invece di `print`); esso viene denominato `hw3.c`.

```
/* file hw3.c */
#include <stdio.h>
#include "printa.h"
int main() {
  printa();
  return 0;
}
```

Il prototipo della funzione di stampa `printa` è questo:

```
  /* file print.h */
void printa();
```

Ovviamente il comando di compilazione è `gcc -c hw3.c`

La funzione di stampa in assembler (`printa`) prende la forma di Figura G.28 (tra i commenti le linee di comando).

```
;file printa.asm   sottoprogramma chiamato da hw3.c
;      Assemblare:   nasm -f elf64 printa.asm
;      Collegare:    gcc hw3.o  printa.o  -o hw3

        section .text
        global  printa      ; Definisce il punto di entrata

printa:
mov     rax, 1                  ; WRITE(stdot,*msg,n)
        mov     rdi, 1                  ;  standard output
        mov     rsi, msg                ;   indirizzo del messaggio
        mov     rdx, l                  ;    numero di byte
        syscall                         ; chiamata al SO

        mov     rax, 60                 ; uscita e ritorno al SO
        xor     rdi, rdi                ; con codice di non errore
        syscall                         ;  exit(0)

msg     db      'Hello World!', 10
l       equ     $-msg
```

Figura G.28 La funzione `printa`, corrispondente assembler della `print.c` di Figura G.26.

Il collegamento si ottiene con il comando `gcc hw3.o printa.o -o hw3`.
Battendo `./hw3` appare `Hello World!` a video.

Osservazione

Vale la pena di osservare che nella pratica professionale è estremamente improbabile che i sottoprogrammi siano in C e il programma principale sia in assembler, ovvero che il codice proveniente dall'assembler chiami codice proveniente dal C. La situazione usuale è che programmi scritti in C chiamino sottoprogrammi scritti in assembler.

G.9.2 Passaggio dei parametri

Nei precedenti esempi non c'era passaggio dei parametri. Dovendo scambiare parametri tra C e assembler è ovvio che il programmatore deve rispettare le convenzioni del C. Purtroppo queste non sono le stesse a seconda della versione del sistema operativo e dell'architettura.

Tradizionalmente, sulle macchine dotate di stack, il parametri vengono passati tramite questo componente architetturale, come esposto al Paragrafo 3.7.1 del testo. Per un attimo facciamo riferimento alla architettura ×86 a 32 bit e consideriamo la chiamata alla funzione `f`

```
int f(a, b, c)
```

dove a, b e c sono tre parametri (su 32 bit). Il meccanismo è il seguente.

- Il programma chiamante esegue tre istruzioni di push: `push c, push b, push a`; cioè inserisce i tre parametri sullo stack, partendo dall'ultimo fino al primo[13].
- Viene eseguita l'istruzione `call f`.
- All'entrata in `f` vengono effettuate queste due operazioni:
  - `push epb`
  - `mov ebp, esp`

  La prima "salva l'EPB del chiamante", la seconda aggiorna EPB a puntare alla testa dello stack, in modo che EPB diventi il registro di riferimento sia per indirizzare i parametri (che già si trovano nello stack), sia per indirizzare le eventuali variabili/dati interni della routine (per i quali vengono presi posti a seguire sempre sullo stack).
- Al termine della routine, il risultato viene messo in EAX; EBP viene aggiornato con il valore salvato; lo stack viene svuotato dei parametri e viene effettuato il ritorno al chiamante (facendo in modo che lo stack sia nella stessa situazione in cui si trovava prima della chiamata).

Il C di Linux a 32 bit, ma anche sui sistemi a 16 bit, come pure molti altri compilatori adottano la convenzione precedente.

Convenzione di chiamata per l'architettura a 64 bit

Con l'architettura a 64 bit le cose sono cambiate. Qui di seguito facciamo riferimento al sistema Linux a 64 bit (per essere precisi a Ubuntu a 64 bit) e prendiamo il considerazione il compilatore GCC[14].

La convenzione di chiamata prevede che

a) i primi 6 parametri vengano passati tramite i registri `RDI, RSI, RDX, RCX, R8 e R9`, ordinatamente a partire dal primo;
b) gli eventuali ulteriori parametri vengano invece passati attraverso lo stack, ordinatamente a partire dall'ultimo.

In Figura G.29 viene data una schematizzazione dello stack in riferimento alla chiamata di una funzione (`f`), alla quale vengono passati 9 parametri (a, b,..., i). Si ricordi che lo stack dell'architettura ×86 si sviluppa verso gli indirizzi bassi. La parte che nello stack precede l'indirizzo di ritorno, ne dà lo stato stack prima dell'esecuzione della chiamata (`call f(..)`).
Le prime istruzioni di `f`

1) salvano RPB (`push rbp`);
2) portano RBP a puntare alla posizione corrente della testa dello stack (`mov rbp, rsp`);

[13] Questo è l'ordine del C. Esso si giustifica con l'intento di fare in modo che la funzione f prelevi il primo parametro con il primo pop (se usa questa istruzione per prelevare i parametri).

[14] La relativa ABI è specificata nel documento "System V Application Binary Interface - AMD64 Architecture Processor Supplement", scaricabile all'indirizzo
`https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf`.

3) riservano lo spazio per le variabili locali, incremendando RSP del numero pari a quello dei byte da esse occupato. Se per esempio si ha a che fare con un intero vengono presi 4 byte, mentre per un `double` vengono presi 8 byte.

Ci sarebbe da aggiungere che la specifica della ABI prevede che dopo lo spazio per le variabili locali venga riservata una ulteriore "zona rossa" di 128 byte, che il compilatore può usare a suo piacimento. GCC utilizza le prime posizioni di tale zona per copiarci il contenuto dei registri impiegati per passare i parametri.

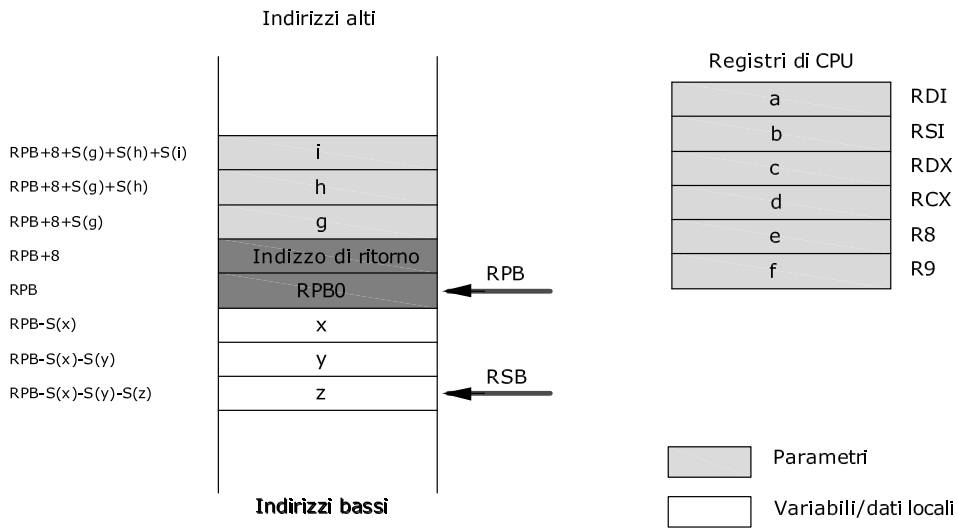

Figura G.29 Stato dello stack per la chiamata: `f(a, b, c, d, e, f, g, h, i)`. I primi 6 parametri vengono passati nei registri come indicato (a destra in alto). I restanti parametri vengono messi nello stack, ordinatamente a partire dall'ultimo. Si è assunto che la funzione `f` abbia tre variabili interne `x`, `y` e `z`. Con S(e) si è indicato lo spazio richiesto (numero di byte) dall'elemento e. In altre parole, mentre per i primi 6 parametri si impiegano sempre 64 bit (misura dei registri) per i parametri passati tramite stack e per le variabili locali lo spazio occupato deriva al loro formato (int occupa 4 byte, double 8).

G.9.3 Esempio di chiamata da C di funzione scritta in assembler

L'esempio consiste nel costruire un sottoprogramma assembler che effettui il quadrato di due numeri interi. Lasceremo il programma principale le incombenze relative alla stampa dei messaggi e alla lettura del numero di cui si vuole il quadrato. Iniziamo sviluppando in C sia il programma principale che la funzione. Chiamiamo `q.c` il programma principale e `quad.c` la funzione che effettua il calcolo del quadrato. Ambedue sono in Figura G.30. Si noti che, sia in `q.c`, sia in `quad.c`, presentano una certa ridondanza (non è strettamente necessario definire le variabili `y` e `v`).

Costruiamo ora la funzione `quad` in assembler, in modo da ricalcare la versione C di Figura G.30. Nel caso specifico, essendoci un solo parametro questo viene passato tramite RDI. Inoltre, siccome deve essere assegnato il valore del risultato alla variabile locale `v`, è necessario prendere per essa un posto sullo stack; ciò impone di salvare RBP e aggiornarlo in modo da poter fare riferimento alla posizione di `v`. Dunque tutto si riduce a:

```
/* file q.c */
#include <stdio.h>
#include "quad.h"
int main() {
int x;
printf ("Batti un numero intero: ");
scanf("%d",&x);
int y = quad(x);
printf ("Quadrato di %d = %d\n",x,y);
return 0;    }

/* file quad.h */
int quad(int x);

/* file quad.c */
int quad(int x) {
int v;
v= x*x;
return v;   }
```

Figura G.30 Il programma `q.c` e la funzione `quad.c`.

- Salvare RBP e aggiornarlo tramite RSP;
- Effettuare il prodotto tra EAX e EDI, avendo copiato prima in EAX il numero passato in EDI (come previsto dall'istruzione `IMUL`);
- Salvare nella posizione presa per la variabile locale `y` il risultato del prodotto. Con riferimento alla Figura G.29, la posizione di `y` è la prossima sullo stack, ovvero quella alla posizione RPB-4;
- Ripristinare RBP e tornare al chiamate (con in EAX il risultato).

Risulta il testo assembler riportato in Figura G.31. Viene calcolato il quadrato del parametro passato in RDI. Trattandosi di intero (32 bit) il numero sta tutto in EDI. Si noti che la funzione calcola il quadrato di interi e quindi anche di numeri negativi (a tale scopo è stata usata l'istruzione `imul` che moltiplica numeri con segno, diversamente dalla mul, usata in precedenza che invece moltiplica numeri senza segno.

```
;file  quad.asm
;  Assemblare:   nasm -f elf64 q.asm
;  Collegare:    gcc q.o  quad.o  -o q
section .text
        global  quad
quad:   push    rbp             ;salva rbp del chiamante
        mov     rbp,rsp         ;rbp-> testa stack
        mov     eax, edi
        imul    eax, edi        ; prodotto interi
        mov     [rbp-4], eax    ;rpb-4 punta a y
        pop     rbp
        ret
```

Figura G.31 Versione assembler della funzione quad.

La Figura G.32 mostra la sequenza di comandi e l'esecuzione a video.

```
giacomo@giacomo-virtual-machine:~/Asm/C$ gcc -c q.c
giacomo@giacomo-virtual-machine:~/Asm/C$ nasm -f elf64 quad.asm
giacomo@giacomo-virtual-machine:~/Asm/C$ gcc -o q  q.o quad.o
giacomo@giacomo-virtual-machine:~/Asm/C$ ./q
Batti un numero intero: -9
Quadrato di -9 = 81
giacomo@giacomo-virtual-machine:~/Asm/C$ ./q
Batti un numero intero: 1234
Quadrato di 1234 = 1522756
giacomo@giacomo-virtual-machine:~/Asm/C$
```

Figura G.32 Video della sequenza di comandi e dell'esecuzione del programma q (scritto in C), chiamante la routine quad (scritta in assembler), per la versione a 64 bit.

Ottimizzazione della versione assembler

Tenuto conto che è del tutto irrilevante che alla variabile `v` venga assegnato un valore, la funzione può essere semplificata come in Figura G.33. Si noti che, non prendendo posto per `v`, non c'è bisogno di toccare RPB. Basta restituire il valore del prodotto. Il tutto si riduce a 3 istruzioni.

```
;file  quad.asm
;  Assemblare:   nasm -f elf64 q.asm
;  Collegare:    gcc q.o  quad.o  -o q

section .text
        global  quad
quad:
        mov     eax, edi
        imul    eax, edi       ; prodotto interi
        ret
```

Figura G.33 Funzione `quad.asm` ottimizzata.

Confronto tra la versione C e la versione assembler

È istruttivo fare il confronto tra il codice della `quad.asm` (Figura G.31) e l'equivalente assembler della `quad.c` prodotto del compilatore GCC. Il testo generato dal compilatore è in Figura G.34. Si rimarca che tale testo è in assembler GAS (Gnu Assembler).

Tralasciando di esaminare il consistente numero di direttive, e a parte la difficoltà di lettura del GAS (operando di destinazione a destra e non a sinistra del/dei sorgente/sorgenti, registri denotati con un % iniziale), si nota che il numero di istruzioni derivanti dalla traduzione da C ad assembler è superiore a quello della corrispondente funzione di Figura G.31: 9 istruzioni contro 7. La differenza è dovuta al fatto che la traduzione standard di GCC prevede che i parametri passati nei registri vengano copiati sullo stack (nella zona rossa, cioè oltre le variabili locali). Se poi si confronta il testo di Figura G.34 con la versione assembler ottimizzata si hanno 9 istruzioni contro 3.

Il confronto parrebbe indicare un marcata superiorità, quanto a efficienza e occupazione di memoria, dell'assembler rispetto al C.

Questa è una considerazione che si sente spesso fare e che è vera in termini generali. Tuttavia i moderni compilatori prevedono la possibilità di ottimizzare il codice. GCC prevede l'opzione `-O`. Con essa si possono richiedere differenti livello di ottimizzazione; in particolare con `-O3` si

```
    .file "quad.c"
    .text
    .globl  quad
    .type   quad, @function
quad:
.LFB0:
    .cfi_startproc
            pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
            movq    %rsp, %rbp
    .cfi_def_cfa_register 6
            movl    %edi, -20(%rbp)
            movl    -20(%rbp), %eax
            imull   -20(%rbp), %eax
            movl    %eax, -4(%rbp)
            movl    -4(%rbp), %eax
            popq    %rbp
    .cfi_def_cfa 7, 8
            ret
    .cfi_endproc
.LFE0:
    .size   quad, .-quad
    .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.4) 5.4.0 20160609"
    .section .note.GNU-stack,"",@progbits
```

Figura G.34 Testo assembler equivalente alla quad.c, prodotto dal compilatore GCC. Il testo viene generato con il comando gcc -S quad.c che genera il file quad.s.

richiede l'ottimizzazione sia dell'occupazione di memoria sia del tempo di esecuzione. Il risultato della compilazione ottimizzata (`gcc -O3 -S quad.c`) è in Figura G.35.

Tralasciando anche questa volta di considerare le direttive, si scopre che anche il compilatore produce un codice con sole 3 istruzioni.

Ma attenzione! Questo risultato è valido per la versione a 64 bit, che usa i registri per passare i parametri. Su un sistema a 32 bit il compilatore avrebbe comunque salvato EBP e avrebbe passato il parametro tramite lo stack, aggiungendo almeno 3 istruzioni in più.

Si può concludere che usualmente i programmi scritti in assembler sono più efficienti di quelli compilati, ma molto dipende dal sistema in uso.

Considerazioni conclusive

È stato presentato un certo numero di programmi assembler scritti in Nasm, per DOS e per Linux. L'attenzione è stata diretta a mettere in evidenza i seguenti aspetti:

- Chiamate delle funzioni del sistema operativo.
- Impiego delle macro.
- Chiamate ai sottoprogrammi e passaggio dei parametri.
- Compilazione/assemblaggio di moduli distinti e loro successivo collegamento per ottenere il file eseguibile.
- Convenzioni di chiamata e collegamento di moduli prodotti dal compilatore C e dall'assemblatore.

Il lettore è invitato a realizzare in assembler (anche parzialmente) qualche programma di suo interesse.

```
   .file "quad.c"
.section .text.unlikely,"ax",@progbits
.LCOLDB0:
.text
.LHOTB0:
.p2align 4,,15
.globl quad
.type quad, @function
quad:
.LFB0:
.cfi_startproc
        movl %edi, %eax
        imull %edi, %eax
        ret
.cfi_endproc
.LFE0:
.size quad, .-quad
.section .text.unlikely
.LCOLDE0:
.text
.LHOTE0:
.ident "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.4) 5.4.0 20160609"
.section .note.GNU-stack,"",@progbits
```

Figura G.35 Testo assembler equivalente alla quad.c, prodotto dal compilatore GCC con opzione di ottimizzazione -O3. (Il testo è stato generato generato con il comando gcc -O3 -S quad.c.)

Infine, ci sia consentito dare questo **suggerimento**: un modo per capire come vanno le cose consiste nel far generare dal compilatore l'equivalente assembler, ispirandosi ad esso nello scrivere il proprio testo assembler.

G.9.4 Siti web

Per comodità del lettore, qui di seguito viene riportato l'indirizzo dei siti da cui possono essere scaricati i principali programmi menzionati nel corso della seconda parte dell'Appendice. Il lettore è invitato a scaricarli e installarli. Ovviamente, se per esempio il suo sistema operativo è Ubuntu e desidera sperimentare solo con esso, non ci sarà che da scaricare la sola versione di Nasm corrispondente, il compilatore GCC fa parte di per sé del sistema Linux. Il Paragrafo 1.3 della documentazione Nasm ("Installation") spiega come si installa l'assemblatore.

Indirizzi utili
Nasm: `www.nasm.us/`
FreeLink: `sourceforge.net/projects/freelinkbr/`
Ubuntu: `www.ubuntu-it.org/`
FreeDOS: `www.freedos.org/`
Virtual Box: `https://www.virtualbox.org/`
VMware-player: `https://vmware-player.it.softonic.com/`

Altre informazioni si trovano facendo ricerche con Google. Ad esempio se si ricercano le funzioni del DOS si arriva alla pagina `http://spike.scu.edu.au/ barry/interrupts.html#ah09` dove esse sono elencate. Con maggiore facilità si trovano manuali, specifiche, ecc. relative a Linux e al GCC.

Indice analitico

Assembler
- etichetta, 6
- OPCODE, 5
- operandi, 5
- sintassi, 4

Assembler (8086), *1*
- commenti, 4
- macroistruzioni, 6

Chiamata da assembler a C, 40
Chiamata da C ad assembler, 41
- esempio, 43

Collegamento al C, 39
Collegamento dei moduli, 9
Confronto C-assembler, 45

DOS, 2

File oggetto, 3
File sorgente, 3
Formato file oggetto, 13
FreeDOS, 14
Funzioni del sistema operativo, 14
- DOS, 15
- Linux 32 bit, 16
- Linux 64 bit, 17

Hello World
- sotto DOS, 18
- sotto Linux 64 bit, 22
- sotto Linux, 32 bit, 25

Inversione ordine, 27
- versione DOS, 28
- versione Linux 32 bit, 34

Linguaggio assembler, *vedi* assembler (8086)
Linker, 4
Linux, 2

Macro, 7
- espansione, 7
- parametri formali, 7

Macro in Nasm, 27
Masm, 1, 12
Moduli di programma separati, 37

Nasm, 1, 12
- aspetti caratteristici, 12
- linea di comando, 13

Passaggio dei parametri, 42
- ABI 64 bit, 42

Processo di traduzione, 7

Simboli esterni, 10
Simboli globali (pubbblici), 10

Ubuntu, 14

Virtual Box, 14
VMware-player, 14

Windows 10, 14

Indice

G Il linguaggio assembler **1**
G.1 Generalità . . . 2
G.2 Sintassi . . . 4
G.2.1 Le macro . . . 6
G.3 Il processo di traduzione . . . 7
G.4 Collegamento dei moduli . . . 9
G.5 Premessa . . . 12
G.5.1 Alcuni aspetti caratteristici dell'assemblatore Nasm . . . 12
G.5.2 Linea di comando . . . 13
G.5.3 Le funzioni del sistema operativo . . . 14
G.6 Hello World: il primo programma di esempio . . . 18
G.6.1 Hello World sotto DOS . . . 18
G.6.2 Hello World sotto Linux 64 bit . . . 22
G.6.3 Hello World sotto Linux, architettura a 32 bit . . . 25
G.7 Inversione ordine: il secondo programma di esempio . . . 27
G.7.1 Inversione d'ordine: versione DOS . . . 28
G.7.2 Inversione d'ordine: versione Linux a 32 bit . . . 34
G.8 Moduli di programma separati . . . 37
G.9 Collegamento con linguaggi di alto livello . . . 39
G.9.1 Collegamento in assenza di parametri . . . 39
G.9.2 Passaggio dei parametri . . . 42
G.9.3 Esempio di chiamata da C di funzione scritta in assembler . . . 43
G.9.4 Siti web . . . 47

Indice analitico **49**