Approfondimenti al Capitolo 15
Il linguaggio assembler

Aggiornato il 29 aprile 2009

Questo materiale integrativo ha lo scopo di concretizzare i contenuti del Capitolo 15 del testo in riferimento al linguaggio assembler per l'architettura $\times 86$. Per semplicità ci si riferisce all'assembler per la CPU 8086/88.[1]

L'assembler 8086 presenta alcune complicazioni derivanti dall'architettura stessa; in particolare dalla segmentazione della memoria.

Nel corso del capitolo ci si riferisce alla versione 4.0 dell'assemblatore Microsoft, designato con la sigla MASM (Macro assemblatore). Di esso si adottano la notazione e le convenzioni. Nell'illustrazione delle sue caratteristiche e delle modalità di impiego supporremo di operare sotto DOS, ovvero in una finestra di emulazione DOS.

Il Paragrafo14.8 riporta tre programmi di esempio, di crescente complessità. A essi si fa spesso riferimento per meglio illustrare i concetti presentati.

Sebbene nel testo ci si riferisca al MASM 4.0, per assemblare questi tre programmi si è fatto uso dell'assemblatore di Arrowsoft Systems, v1.00d, compatibile con MASM. Si tratta di un prodotto di pubblico dominio, con qualche limitazione rispetto alla versione 4.0 di MASM (i programmi devono stare entro 64K), ma perfettamente adeguato come strumento di esercitazione.[2]

Una vera miniera di materiali riguardanti il linguaggio assembler si trova al sito (*"The place on the Net to Learn Assembly Language"*) `http://webster.cs.ucr.edu`. Si suggerisce inoltre di procurarsi l'emulatore Emu8086, scaricabile da rete (`http://www.emu8086.com`). Attraverso una piacevole interfaccia grafica, questo programma permette di emulare l'esecuzione di un programma assembler 8086, mostrando passo-passo l'effetto dell'esecuzione delle singole istruzioni. Esso impone alcune limitazioni sulla sintassi del linguaggio assembler (per esempio, prevede che l'indirizzo di partenza del programma sia fisso), tuttavia può rivelarsi molto utile in una fase di approccio iniziale all'architettura 8086 e alla relativa programmazione assembler.

14.1 Generalità

Per qualunque traduttore il processo di scrittura-traduzione-esecuzione è quello schematizzato in Figura 6.1 del testo e ripetuto in maggior dettaglio in Figura 14.1. Esso si compone di questi passi:

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

In Figura 14.1 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. In questo caso il traduttore deve essere il relativo compilatore. Naturalmente compilatori e assemblatore devono produrre un codice congruente, rispettando alcuni standard di comunicazione fra moduli.

---

[1] Il linguaggio assembler dei modelli successivi è sostanzialmente identico, salvo il fatto che, a partire dal processore 80386, i registri a 32 bit vengono designati con un simbolo che inizia con la lettera E (per esempio EAX); mentre gli 8 registri di uso generale sono diventati praticamente del tutto equivalenti. Inoltre, con la crescita della capacità dei processori si sono aggiunte svariate istruzioni. Si tratta di differenze, che per un verso complicano e per un altro semplificano, ma che non cambiano la natura della programmazione.

[2] L'assemblatore e il suo manuale sono facilmente scaricabili da rete, da più parti.

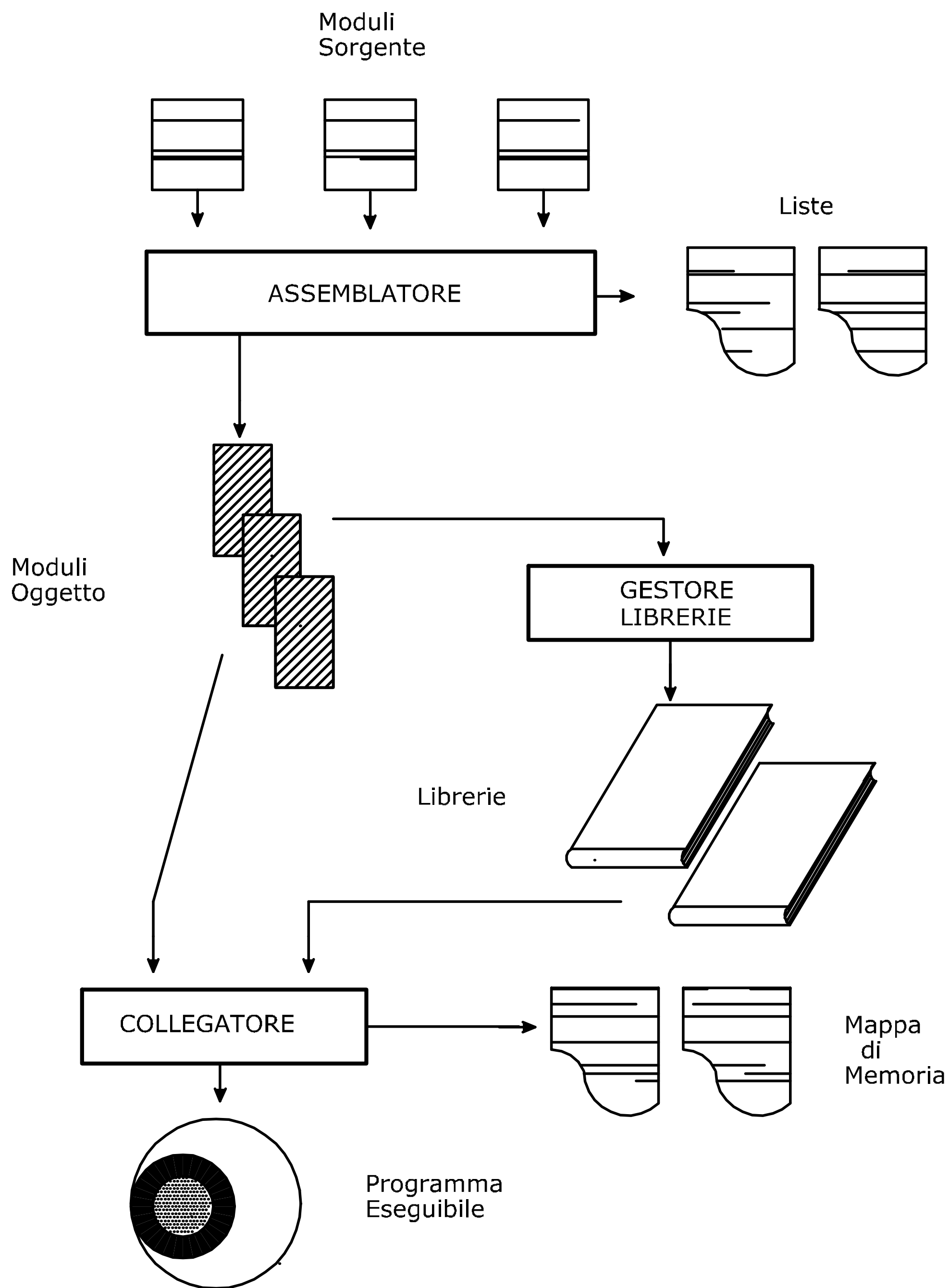

Figura 14.1 Generazione di un programma eseguibile a partire da più moduli sorgente. Nel caso del DOS del PC i file sorgente assembler hanno estensione standard `.ASM`; i file oggetto `.OBJ`; i file di libreria `.LIB`; i file eseguibili `.EXE` o `.COM`. Si assume che i moduli sorgente siano tutti scritti in assembler. È possibile che parte dei moduli sorgente siano scritti in un linguaggio di alto livello. In questo caso il traduttore deve essere il relativo compilatore e il programma assembler deve essere scritto in modo da conformarsi alle convenzioni del linguaggio di alto livello per quanto riguarda la comunicazione fra i moduli.

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 si dà l'estensione `ASM`. Ai fini della nostra discussione supponiamo che il file contenente il programma principale venga chiamato `MAIN.ASM` e che gli eventuali sottoprogrammi siano contenuti in moduli denominati `SUB1.ASM`, `SUB2.ASM` ecc.

Traduzione Ricordando che si assume di lavorare sotto (emulazione) DOS, per assemblare il modulo `MAIN` si può dare la seguente linea di comando

```
C:> MASM MAIN
```

Prima di entrare in esecuzione, l'assemblatore chiede il nome da assegnare al codice oggetto e che nome assegnare all'eventuale listato. Alternativamente il programmatore può fornire questo genere di informazioni nella linea di comando a seguito del nome del file. L'assemblatore produce il modulo (oggetto) `MAIN.OBJ` e l'eventuale listato `MAIN.LST`. Il modulo `.OBJ` costituisce il prodotto della traduzione del programma sorgente verso il codice di macchina. Il modulo `.LST` contiene il testo del programma e le informazioni corrispondenti al codice di macchina prodotto. Il processo di traduzione è descritto in maggior dettaglio al Paragrafo 14.5. Al termine dei Paragrafi 14.8.1 e 14.8.2 sono riportati due listati generati dall'assemblatore

Collegamento dei moduli I vari 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. Il linker produce il modulo eseguibile `MAIN.EXE`. Quest'ultimo è in una forma che corrisponde a quella che avrà in memoria al tempo di esecuzione, salvo alcune differenze, legate alla rilocazione degli indirizzi.

La Figura 14.1 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 e l'esecuzione del programma.

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, al suo punto di entrata (si veda il Paragrafo 14.4.2).

14.1.1 Sintassi

Un testo assembler è fatto di linee. Ogni linea è una stringa di caratteri `ASCII`, terminante con la combinazione *Carriage Return-Line Feed* (`CR-LF`). 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):

```
[Nome]  [Codice operazione]  [Operandi]  [;Commento]
```

Il campo del `Codice operazione`, normalmente indicato come campo `OP`, determina l'interpretazione degli altri campi.[3]

[3]Diversamente da molti assemblatori del passato, quello qui esaminato lascia una certa libertà nello scrivere il testo del programma. Per esempio: non è necessario che il `[Nome]` inizi nella prima colonna della riga. Nel seguito, per motivi di leggibilità, adotteremo la convenzione di scrivere il nome sempre a partire dalla prima colonna della riga. Se in prima colonna c'è un carattere di spazio si deve assumere che lo mnemonico che segue è un codice di operazione.

Commenti Prima di procedere è meglio parlare dei commenti.[4] Il campo del commento deve necessariamente iniziare con "`;`". Se il primo carattere di una linea – esclusi eventuali caratteri di spaziatura – è il carattere "`;`", allora tutta la linea viene considerata come un commento. C'è un altro modo per introdurre un commento: usare il codice di operazione `COMMENT`, con il quale è possibile aprire un commento che occupa un numero arbitrario di linee, secondo la seguente sintassi:

```
COMMENT  Delimitatore
Testo
Delimitatore [statement]
```

dove `Delimitatore` è un qualunque carattere `ASCII` che non appare all'interno di `Testo`. Esso funge da coppia di parentesi racchiudenti il commento.

Nomi Il campo `Nome`, quando è presente, deve contenere un identificatore scelto dal programmatore: una qualunque concatenazione di lettere, cifre e caratteri speciali, che inizi con una lettera o con uno dei caratteri "`_`", "`?`", "`$`" o "`@`". Non c'è differenza tra caratteri maiuscoli o minuscoli. Sono perciò nomi corretti: `TABLE, ON, Off, DiSpLaY, _JOB1, ARRAY_1`. Non lo sono `1ARRAY` e `M=2`; il primo perché inizia con una cifra ("1"), il secondo perché contiene un carattere non consentito. Posto che sia sintatticamente corretto, un nome non deve coincidere con alcune delle parole chiave del linguaggio (si veda più avanti).

Diversamente da altri assemblatori, dove il campo corrispondente a `Nome` viene considerato sempre e comunque un'etichetta, il `Nome` rappresenta l'identificatore di un "oggetto" o "simbolo", e serve per fare riferimento all'oggetto stesso. Un simbolo è un'entità costituita da un `Nome`, da un `Tipo` e da un insieme di `Attributi`. Esempi di simboli MASM sono le variabili, le etichette (*label*), le costanti numeriche e le stringhe, i nomi di procedura, di segmento, di macro ecc.

Il tipo del simbolo definisce l'uso che del simbolo si può fare all'interno del programma. La n-pla di valori assunti dagli attributi di un simbolo verrà indicata come `Valore` del simbolo stesso. Per esempio se un simbolo è una *variabile*, i valori assunti dai suoi attributi sono: (a) l'indirizzo della prima cella di memoria occupata dalla variabile; (b) il nome del segmento cui appartiene; (c) l'indicazione della dimensione della variabile (byte, parola, doppia parola ecc.).

Dei simboli definiti dal programmatore si parla nel Paragrafo 14.3.

Codice di operazione Il campo `OP` può contenere:

a) lo mnemonico di un'istruzione di macchina;
b) una direttiva per l'assemblatore (le direttive sono anche dette *pseudo-operazioni*).

Nel primo caso l'assemblatore genera il codice di macchina, tenendo conto del contenuto del campo degli operandi. A uno mnemonico corrisponde in genere più di una codifica binaria, in dipendenza dagli operandi specificati. Per esempio lo mnemonico `MOV` può dar luogo a una molteplicità di codici binari. Le due istruzioni:

```
MOV  AX,BX        ; Copia BX in AX
MOV  AX,VAR       ; Carica AX con il contenuto ..
                  ; .. della locazione VAR
```

producono differenti codici di macchina, inoltre la seconda istruzione viene codificata in un maggior numero di byte della prima.

Una direttiva costituisce un comando per l'assemblatore. Ci sono direttive per la definizione di dati, simboli e segmenti, e direttive di controllo dell'assemblatore. I seguenti sono esempi di definizione di dati:

```
        DB    ?       ;alloca spazio per un byte
VAR     DW     0      ;VAR inizializzata a 0
```

Il primo dei due statement riserva un byte di valore non definito; il secondo definisce una variabile, di una parola, di nome `VAR`, inizializzata a zero.

MASM prevede anche la definizione esplicita dei simboli. Per esempio, lo statement:

[4] Nella programmazione assembler è bene fare ampio ricorso ai commenti, poiché il livello di autodocumentazione del linguaggio è pressoché nullo.

```
FINE    LABEL   NEAR
        RET
```

equivale allo statement:

```
FINE:   RET
```

Ambedue possono essere utilizzate per assegnare l'etichetta FINE all'indirizzo nel codice binario in cui si trova l'istruzione RET. Nel primo caso è stata usata la definizione esplicita del simbolo FINE, usato come etichetta, mentre nel secondo quella implicita.

Il seguente statement mostra una direttiva per la definizione dei simboli:

```
 Num_25   EQU    25
```

esso comanda l'assemblatore a considerare la stringa Num_25 del tutto equivalente alla stringa 25. In altre parole definisce Num_25 come una costante numerica di valore 25.

Ci sono direttive che servono a scegliere il repertorio di istruzioni macchina da usare[5] o la base in cui vengono rappresentate le costanti numeriche. Per esempio:

```
        .286                 ; repertorio istruzioni 80286
        .RADIX      16       ; numerazione in base 16
```

Operandi Il contenuto del campo Operandi deve essere congruente con il contenuto del campo OP. In questo campo possono trovarsi espressioni contenenti simboli definiti dal programmatore, assieme a operatori del linguaggio stesso (Figura 14.2). Di questo si parla nel Paragrafo 14.3.

Parole riservate Le parole chiave del MASM possono essere divise in quattro classi:

- mnemonici delle istruzioni;
- mnemonici dei registri della CPU;
- mnemonici degli operatori;
- mnemonici delle direttive.

In Figura 14.2 vengono riportate le parole chiave corrispondenti alle direttive e agli operatori. Le direttive sono state divise in sei gruppi. A quelle dei primi tre gruppi si è accennato in precedenza e di esse si riparla nella parte seguente. Le direttive dei tre gruppi restanti servono a richiedere all'assemblatore l'esecuzione di specifiche azioni, quali la segnalazione di situazioni anomale o la lettura di file ausiliari, l'assemblaggio condizionato di frammenti di testo, il formato e le caratteristiche della lista generata.

14.2 Segmenti e moduli

L'architettura ×86 prevede un modello di memoria segmentata.[6] La segmentazione è la diretta traduzione del modello di elaborazione, secondo cui l'algoritmo è separato dagli oggetti manipolati e dalle entità temporanee. L'algoritmo (le istruzioni) si traduce nel segmento di codice, gli oggetti (le costanti e le variabili) nel segmento dati, le entità temporanee nello stack.

14.2.1 Direttive per la gestione dei segmenti

Nell'assembler 8086 si usano le direttive SEGMENT e ENDS per definire un (frammento di) segmento logico, secondo questa sintassi:

```
NomeSeg     SEGMENT [Allineamento][Combinabilità][‘NomeClasse’]
            BloccoStatement
NomeSeg     ENDS
```

[5] Il repertorio è andato estendendosi praticamente con ogni nuovo modello di CPU.

[6] A partire dal 386, la segmentazione potrebbe anche essere evitata con un artificio consistente nel prevedere un unico grosso segmento, corrispondente a tutto lo spazio di memoria indirizzabile dal programma, all'interno del quale la struttura degli indirizzi è lineare. In ogni caso, la segmentazione resta un caposaldo di questa architettura.

DIRETTIVE

DB	DD	DQ	DT	DW
DWORD	QWORD	RECORD	STRUC	TBYTE
=	ENDM	ENDP	EQU	EXITM
EXTRN	GROUP	LABEL	LOCAL	MACRO
NAME	PROC	PUBLIC	PURGE	
.186	.286	.286p	.287	.386
.386p	.8086	.8087	.RADIX	ASSUME
COMMENT	ENDS	EVEN	ORG	SEGMENT
.ERR	.ERR1	.ERR2	.ERRB	.ERRDEF
.ERRDIF	.ERRE	.ERRIDN	.ERRNB	.ERRNDEF
.ERRNZ	END	INCLUDE	IRP	IRPC
REPT				
ELSE	ENDIF	IF	IF1	IF2
IFB	IFDEF	IFDIF	IFE	IFIDN
IFNB	IFNDEF			
%OUT	.CREF	.LALL	.LFCOND	.LIST
.SALL	.SFCOND	.TFCOND	.XALL	.XCREF
.XLIST	PAGE	SUBTTL	TITLE	

OPERATORI

AND	BYTE	DUP	DWORD	EQ
FAR	GE	GT	HIGH	LE
LENGTH	LOW	LT	MASK	MOD
NE	NEAR	NOT	OFFSET	OR
QWORD	PTR	SEG	SHL	SHORT
SHR	SIZE	TBYTE	THIS	TYPE
.TYPE	WIDTH	WORD	XOR	

Figura 14.2 Direttive (pseudo-operazioni) e operatori del MASM.

Le direttive `SEGMENT` e `ENDS` (*end of segment*) costituiscono la coppia di delimitatori di apertura e chiusura della definizione del frammento di segmento rappresentato dalla sequenza di statement `BloccoStatement`.

`NomeSeg` è l'identificatore del segmento cui il frammento appartiene. Si parla di frammento di segmento perché in uno stesso modulo di programma lo stesso segmento può essere aperto e chiuso più volte. Le definizioni di segmenti possono anche essere annidate, ma non possono risultare sovrapposte. È perciò possibile dichiarare un segmento all'interno di un altro, come mostrato qui sotto. Non è invece consentito aprire un nuovo frammento del segmento corrente. In altri termini, non si può avere, nel campo di azione di una direttiva `SEGMENT`, un'altra `SEGMENT` con lo stesso `NomeSeg`.

```
CSEG     SEGMENT
         ...
CONST    SEGMENT        ;Inizio segmento annidato
         ...
CONST    ENDS           ;Fine segmento annidato
         ...            ;Continuazione segmento CSEG
CSEG     ENDS           ;Fine segmento CSEG
```

I parametri `Allineamento`, `Combinabilità` e `NomeClasse` servono per controllare il modo in cui il linker combina i frammenti di uno o più segmenti definiti in vari moduli, influenzando anche la disposizione con cui questi verranno caricati in memoria.

Allineamento L'`Allineamento` (*align type*) specifica da dove deve essere caricato il frammento di segmento. Esso può essere: `BYTE`, `WORD`, `PARA` o `PAGE`. Se non viene specificato, l'assemblato-

re assume un allineamento al paragrafo (`PARA`), cioè a un indirizzo multiplo di 16 (è questo il caso normale).

Combinabilità La `Combinabilità` (*combine type*) determina il modo in cui i vari segmenti possono essere combinati tra loro dal linker nel costruire il `.EXE`. Le alternative possibili sono: `PUBLIC`, `COMMON`, `STACK`, `MEMORY`, `AT`. Se la combinabilità non viene specificata, il segmento viene ritenuto non combinabile con altri segmenti, anche dello stesso nome, definiti in altri moduli.

Se la combinabilità è `PUBLIC` (o `MEMORY`) il segmento è pubblico. Al momento del collegamento, esso viene concatenato con tutti i segmenti pubblici di uguale nome, definiti negli altri moduli, in modo da formare un nuovo segmento avente lo stesso nome dei frammenti componenti e di lunghezza pari alla somma delle loro lunghezze.

Se la combinabilità è `COMMON`, i frammenti di uguale nome provenienti da moduli diversi vengono collegati per sovrapposizione, anziché giustapposizione, generando un unico segmento che ha lunghezza pari alla lunghezza del frammento di dimensione massima. L'opzione `COMMON` viene normalmente usata per definire aree dati contenenti variabili condivise sulla base della posizione, anziché del nome; analogamente al `COMMON` del Fortran.

La combinabilità `STACK` viene usata per i frammenti che definiscono l'area di memoria da usare per lo stack del programma. Il linker tratta i frammenti `STACK` in modo analogo a quelli `PUBLIC`, con la differenza che per il segmento risultante viene necessariamente assunto `SS` come registro di segmento, mentre `SP` viene inizializzato con la dimensione finale complessiva del segmento, pari alla somma di quella di tutti i frammenti componenti.

L'ultima opzione (`AT`) permette di specificare, mediante una espressione semplice posta come operando della `AT`, il segmento fisico in cui il segmento logico deve essere allocato. Tale opzione, insieme con la direttiva `ORG`, consente di avere il controllo diretto, a livello assembler, degli indirizzi fisici di allocazione degli oggetti generati.

Nome classe `NomeClasse`, definisce la "classe" a cui appartiene il segmento. La classe rappresenta un ulteriore strumento, dopo il nome del segmento, per raccogliere, al momento del collegamento, frammenti di segmento di tipo correlato. Tutti i segmenti di una classe vengono caricati in zone di memoria contigue. Se tale parametro viene omesso, il linker dispone i segmenti logici dell'eseguibile nello stesso ordine in cui li incontra nella scansione dei moduli rilocabili da collegare (Paragrafo 14.2.3). Non c'è alcun limite superiore alla dimensione dello spazio di memoria che una classe può complessivamente occupare (fermi restando i limiti alle dimensioni dei segmenti imposti dai differenti modelli di CPU; per esempio 64 KB per 8086/8088).

Gruppo Ragioni di efficienza possono consigliare la riduzione del numero dei segmenti fisici utilizzati da un programma. Quando due o più segmenti hanno dimensione complessiva inferiore alla dimensione massima possibile per un segmento, la direttiva `GROUP` consente di costruire una sorta di unico segmento logico con più segmenti di diverso nome. La sintassi è questa:

```
Nome    GROUP      NomeSeg[,NomeSeg, ... ]
```

`Nome`, viene assunto come identificatore del gruppo, ha le stesse caratteristiche degli identificatori di segmento e può apparire negli statement in tutti i luoghi in cui è ammesso uno di questi, salvo un'altra `GROUP`. Il gruppo risulta costituito dai segmenti i cui nomi appaiono nella lista che costituisce l'operando della direttiva.

La direttiva `GROUP` non assicura di per sé l'allocazione contigua dei segmenti del gruppo. Infatti essa riguarda esclusivamente la generazione degli scostamenti degli oggetti contenuti nel gruppo, per i quali si avranno due valori diversi, uno riferito alla base del gruppo e uno a quella del proprio segmento. Ne segue che altri segmenti potrebbero frapporsi tra quelli del gruppo, rendendo eventualmente impossibile la sua costruzione. La continuità dell'allocazione può essere assicurata raccogliendo in una stessa classe i segmenti del gruppo.

14.2.2 Generazione dei riferimenti: la direttiva ASSUME

Per consentire all'assemblatore di generare correttamente e automaticamente i riferimenti in memoria, occorre fornirgli una informazione di fondamentale importanza: quale sarà il contenuto dei registri di segmento al momento dell'esecuzione.

Per tale scopo è definita la direttiva `ASSUME` che ha la seguente sintassi:

```
ASSUME     RS:SpecificaRS[,RS:SpecificaRS, ... ]
```

dove RS può essere CS, DS, ES o SS, e SpecificaRS può essere il nome di un segmento o di un gruppo, ovvero la parola chiave NOTHING.[7]

La direttiva informa che (dal punto in cui essa si trova in poi) l'assemblatore deve assumere che il contenuto dei registri di segmento contengano gli indirizzi di segmento corrispondenti agli identificatori simbolici specificati.

È bene porre la direttiva immediatamente dopo l'apertura di ogni segmento codice, in modo che valga l'assunzione che CS individua il segmento in questione:

```
ASSUME     CS:NomeSC
```

dove NomeSC è il nome del segmento di codice corrente. È anche bene mettere una ulteriore ASSUME dopo ogni istruzione che alteri il contenuto di un registro di segmento.

È il caso di osservare che ASSUME, essendo una direttiva per l'assemblatore, non assicura in nessun modo che al momento dell'esecuzione i registri di segmento contengano quanto specificato con la direttiva. È compito del programmatore fare in modo che, al tempo di esecuzione, i registri di segmento non inizializzati automaticamente dal sistema operativo contengano quanto è stato fatto assumere all'assemblatore.

Nel caso del PC-DOS, il registro SS viene inizializzato dal DOS stesso[8], mentre il registro DS non viene invece inizializzato. Il programmatore deve provvedere caricando in DS la base del segmento dati di cui si è fatta assumere la presenza in DS, come nell'esempio qui riportato:

```
DATA    SEGMENT                  ;segmento dati
        ...........
DATA    ENDS
CSEG    SEGMENT   PUBLIC         ;segmento di codice
        ASSUME CS:CSEG,DS:DATA
        ASSUME SS:STACK,ES:nothing
;
ENTRY   LABEL   FAR
;le prossime 2 istruzioni prima di qualunque indirizzamento
;che usi DS
        mov     AX,DATA          ;Caricamento in DS...
        mov     DS,AX            ;..della base di DATA
        ......
```

dove ENTRY è il punto di entrata del programma (il DOS passa il controllo al corrispondente indirizzo. Si veda la direttiva END al Paragrafo 14.4.2). Le prime due istruzioni aggiornano DS in modo da puntare al segmento DATA.

Bisogna evidenziare che la precedente istruzione MOV AX,DATA viene interpretata in modo non usuale dall'assemblatore. Infatti, poiché l'operando è il nome di un segmento e non di una variabile o di una costante, l'istruzione non viene tradotta nell'operazione di macchina che carica in AX il contenuto della prima posizione del segmento stesso. Viene invece generata un'operazione di caricamento immediato incompleta: nel campo del codice di operazione l'assemblatore inserisce la codifica dell'operazione MOVE *immediate in* AX, mentre il campo dell'operando non viene riempito con uno scostamento. Viene invece inserita l'informazione che in tale campo il loader deve inserire l'indirizzo fisico – diviso per 16 – a cui viene caricato il segmento. Quando il segmento viene caricato in memoria, il loader ha l'informazione necessaria per inserire l'indirizzo nel campo dell'operando immediato, completando il codice di istruzione.[9]

14.2.3 Relazione tra segmenti e moduli

Un "modulo assembler" comprende sempre uno o più segmenti. La direttiva END chiude il modulo e può specificare l'etichetta, che corrisponde all'indirizzo iniziale, punto di entrata (*entry point*) dell'in-

[7]È possibile anche la forma ASSUME NOTHING.

[8]Inutile dire che CS e IP sono implicitamente inizializzati dal DOS nel momento in cui il controllo passa al programma.

[9]Un'annotazione marginale: vengono usate due istruzioni anziché un'ipotizzabile mov DS,DATA, perché non è consentito scrivere MOV DS,DATA in quanto l'istruzione di caricamento immediato in DS non è prevista nel repertorio 8086. A partire dal 386 questa limitazione è stata rimossa (CS è l'unico registro rimasto non caricabile con l'istruzione MOV.)

tero programma. Il modulo che contiene la specifica del punto di entrata prende il nome di "modulo principale" e gli altri moduli prendono il nome di "moduli secondari".

È stato detto che i segmenti che costituiscono un programma eseguibile possono essere classificati secondo tre differenti tipi: codice, dati e stack. Se un programma è costituito da più di un segmento di codice, a ogni istante solo uno di essi è attivo: quello il cui identificatore si trova nel registro `CS`. I segmenti dati si intendono per le costanti e le variabili statiche (quelle il cui spazio di memoria viene allocato al tempo di traduzione-collegamento); i segmenti di stack si intendono per costanti e variabili dinamiche (quelle il cui spazio di memoria viene allocato al tempo di esecuzione).[10]

In base al numero e al tipo dei segmenti che lo costituiscono, un programma viene classificato secondo diversi modelli di utilizzo della memoria, indicati con i nomi *Small*, *Compact*, *Medium*, *Large* e *Huge*. Il modello Small, che prevede un solo segmento nel quale sono allocati il codice, i dati e lo stack, con il vincolo che la somma delle dimensioni delle aree occupate non ecceda la dimensione massima di 64 KB. Nel modello Huge, non c'è limitazione al numero di segmenti, alla loro dimensione e alla dimensione delle singole variabili.

La suddivisione di un programma in moduli risulta ortogonale rispetto a quella in segmenti. Infatti un modulo può definire diversi segmenti e uno stesso segmento può derivare da parti definite in moduli differenti. Un modulo può definire uno o più frammenti dello stesso segmento. Attraverso le fasi con cui si passa dal programma simbolico a quello eseguibile, i frammenti vengono raccolti in modo da formare un unico segmento.

La Figura 14.3 mostra un esempio in cui tre moduli definiscono due segmenti codice (A e B), due segmenti dati (D1 e D2) e un segmento stack (S).

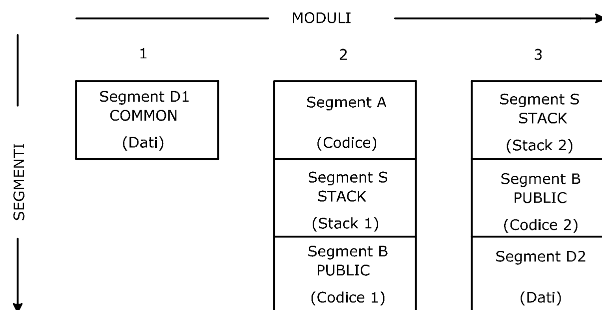

Figura 14.3 Segmenti e moduli.

L'esempio può essere interpretato secondo lo schema di un programma principale, corrispondente al segmento A (modulo 2) e da sottoprogrammi suddivisi in parte nel frammento di segmento B nel modulo 2 e in parte nel frammento di segmento B nel modulo 3; i frammenti di segmento B sono dichiarati pubblici. Il segmento D1 può essere riguardato come un'area dati in comune ai due segmenti di codice. Lo stack, come il segmento codice B, è diviso in due parti rappresentate dai frammenti definiti nei moduli 2 e 3. La frammentazione del segmento B corrisponde alla ripartizione del codice complessivo dei sottoprogrammi tra quello contenuto nel modulo principale (2) e quello compilato separatamente (3); mentre quella del segmento S è motivata dal fatto che i sottoprogrammi corrispondenti al modulo 3 possono richiedere l'uso di uno spazio di stack superiore a quello previsto in 2 (cautelativamente, il programmatore ha scelto di definire in 3 lo spazio di stack richiesto dal codice di questo modulo).[11]

La Figura 14.4 dà la mappa prodotta dal linker nel caso in cui si proceda al collegamento dei moduli nell'ordine in cui il essi appaiono nella Figura 14.3. Risultano 5 segmenti, due dei quali (B e S) sono stati ottenuti riunendo i rispettivi frammenti in base alla loro combinabilità.

L'allocazione in memoria mostrata in Figura 14.4 non può essere considerata particolarmente soddisfacente, in quanto difetta di ordine. Infatti i dati e il codice del programma si trovano dispersi in aree differenti e separate da zone contenenti oggetti di tipo diverso. La struttura della figura è inaccettabile

[10] È tuttavia possibile infrangere queste regole e allocare i dati anche nei segmenti di tipo codice. Ciò è sconsigliato e comunque da evitare per i programmi che operano in modo "protetto".

[11] Si noti che un modulo può anche contenere solo definizioni di dati (o solo codice, o solo stack) come accade nel caso del modulo 1 di Figura 14.3 .

Segmento D1 (Dati)
Segmento A (Codice)
Segmento S (Stack1)
Segmento S (Stack2)
Segmento B (Codice1)
Segmento B (Codice2)
Segmento D2 (Dati)

Figura 14.4 Dislocazione in memoria dei segmenti di Figura 14.3 in base ai soli nomi e alla combinabilità.

se il codice è destinato ad andare in una ROM (*Read Only Memory*) in cui, ovviamente, non può essere contenuta l'area di stack.

Per facilitare un'allocazione di memoria più logica e ordinata, ciascun segmento logico può essere associato a una classe, in modo che i segmenti appartenenti alla stessa classe, pur continuando a rimanere distinti, vengono allocati in modo contiguo in memoria. La Figura 14.5 mostra la disposizione che si otterrebbe nel caso precedente, definendo tre classi: codice, dati e stack. Si osservi che in ogni caso i due segmenti codice A e B e i due segmenti dati D1 e D2 sono rimasti distinti.

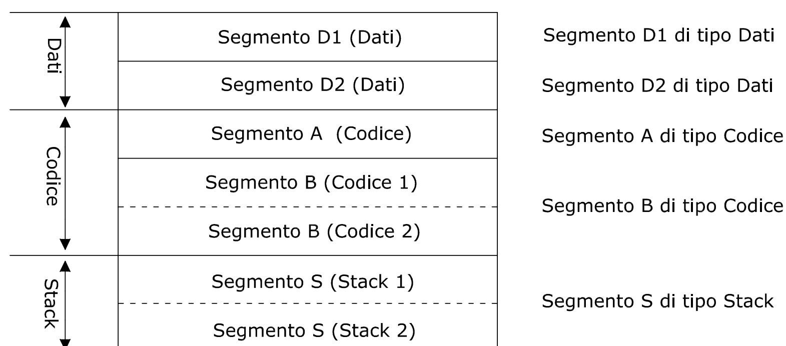

Figura 14.5 Effetto delle classi con riferimento ai segmenti di Figura 14.3.

Se si vogliono accorpare diversi segmenti logici di uguale tipo in uno stesso segmento fisico si può ricorrere al concetto di gruppo. La Figura 14.6 evidenzia quanto accade nel caso dell'esempio se vengono definiti 3 gruppi: Data, Text e Stack. La differenza rispetto al caso precedente sta nel fatto che gli indirizzi logici contenuti nell'eseguibile fanno riferimento a 3 soli segmenti, contro i 5 originari.[12]

L'accorpamento dei segmenti, sempre possibile se la somma delle dimensioni dei segmenti da raggruppare è inferiore alla dimensione massima possibile per un segmento (64 KB per l'8086), consente di aumentare l'efficienza del programma riducendo il numero dei caricamenti dei registri di segmento. D'altra parte non è opportuno che la compattazione sia effettuata tenendo conto della sola dimensione dei segmenti, viceversa è necessario fare in modo che i segmenti logici accorpati appartengano a oggetti logici affini o comunque correlati. La maniera in cui i segmenti vengono accorpati influisce direttamente sulla località del programma.

Sebbene la segmentazione della memoria spinga fortemente verso una programmazione modulare, è sempre possibile imbattersi in programmi che mescolano istruzioni, dati e stack all'interno di uno stesso segmento. Tali programmi possono operare correttamente alla stessa stregua di un programma modulare,

[12]È bene ricordare che l'appartenenza di due o più segmenti a uno stesso gruppo non implica la loro contiguità in memoria, che invece è una prerogativa delle classi.

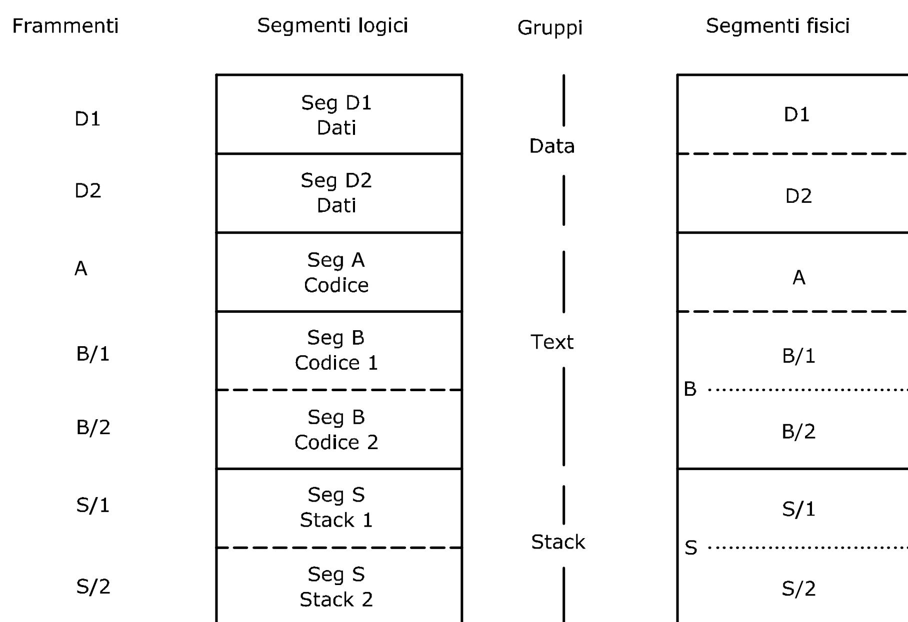

Figura 14.6 Effetto delle classi e dei gruppi.

ma di solito il costo della loro messa a punto e manutenzione risulta più elevato. In ogni caso occorre tener presente che, se un programma non è strutturato a segmenti, i meccanismi di protezione del modo virtuale protetto, disponibili nei processori più avanzati della famiglia, risultano inefficaci.

14.2.4 Una nota sulla segmentazione

La precedente discussione indica che la segmentazione è fonte di non poche complicazioni anche a livello di sintassi di programma. Tuttavia, ai fini dell'acquisizione di un minimo di pratica sperimentale, non è necessario essere in grado di mettere in atto tutte le raffinatezze che il MASM consente.

Ad esempio, per la costruzione di un programma, ottenuto collegando più moduli assemblati separatamente tra loro, è sufficiente: (a) non specificare l'allineamento (che verrà assunto al paragrafo); (2) dare ai segmenti di stack la combinabilità `STACK`; (3) dare eventualmente la combinabilità `PUBLIC` ai (frammenti) di segmento con lo stesso nome se questi stanno in moduli diversi.

14.3 I simboli definiti dal programmatore

La possibilità offerta al programmatore di definire simboli utilizzabili per riferire gli oggetti (segmenti, istruzioni, dati, costanti ecc.) presenti all'interno di un programma, costituisce la base di tutte le funzionalità fornite dal linguaggio assembler. Nel riferire un oggetto mediante un simbolo, il programmatore viene sollevato dalla necessità di specificare direttamente l'indirizzo, lasciando al sistema la responsabilità della definizione.

I nomi di segmento sono stati trattati nella sezione precedente. Nel seguito si considerano i seguenti tipi di simbolo: le etichette, le variabili e le costanti.

14.3.1 Etichette

Le etichette sono simboli utilizzati per identificare l'indirizzo di una istruzione. Esse sono normalmente usate nelle istruzioni di trasferimento del controllo, cioè salti e chiamate di procedura.

Gli attributi di una etichetta sono: (1) l'indirizzo; (2) la distanza e (3) la visibilità.

- Nell'architettura ×86 un indirizzo ha due componenti: segmento e scostamento. Per MASM, l'indirizzo equivale al nome del segmento in cui si trova l'etichetta e allo scostamento entro il segmento stesso.
- L'attributo distanza corrisponde, impropriamente, al "tipo" di etichetta. Esso può assumere solo valore `NEAR` o `FAR`. Nel primo caso l'etichetta può essere riferita solo dall'interno del segmento in cui è definita. Nel secondo può essere anche riferita da altri segmenti. Se non viene esplicitamente indicato il contrario, la distanza di una etichetta viene assunta `NEAR`. A ogni riferimento di un'etichetta `NEAR`, l'assemblatore pone nella codifica dell'indirizzo solo il valore dello scostamento. Se invece l'etichetta è definita `FAR`, viene generato un indirizzo formato dal valore dello scostamento e del segmento; ciò indipendentemente dal segmento in cui si trova il riferimento.
- L'attributo visibilità specifica se l'etichetta deve essere nota solo all'interno del modulo in cui è definita, se può essere riferita anche in altri moduli, ovvero se è definita in un modulo diverso da quello corrente. Il valore assunto nei tre casi è `INTERNAL, PUBLIC, EXTERN`. Se un'etichetta non è dichiarata globale o esterna, la sua visibilità si limita al modulo (cioè interna).

In conclusione l'insieme degli attributi di un'etichetta è dato dalla quadrupla:
`<segmento,scostamento,distanza,visibilità>`.

Il modo più semplice per dichiarare una etichetta è quello della dichiarazione implicita, che consiste nel porre il suo identificatore, seguito dall'operatore ":", nel campo `Nome` dello statement a cui essa deve essere associata. Per esempio lo statement seguente:

```
L1:      JMP     L2
```

dichiara l'etichetta `L1`. Trattandosi di una dichiarazione implicita `L1` è necessariamente `NEAR` e `INTERNAL` al modulo in cui appare.

Un'etichetta può essere definita anche esplicitamente mediante la direttiva `LABEL`, con la sintassi seguente:

```
NomeEtichetta      LABEL      Distanza
```

dove `Distanza` può essere `NEAR` oppure `FAR`. L'esempio precedente potrebbe essere così scritto in modo esplicito:

```
L1         LABEL   NEAR
           JMP     L2
```

La definizione implicita delle etichette consente di definire solo etichette `NEAR`. La definizione delle etichette di tipo `FAR` richiede necessariamente l'uso della direttiva `LABEL`. Se un'etichetta è riferita spesso all'interno di un programma, è bene che essa sia di tipo `NEAR`. D'altra parte se un punto del programma deve essere riferito anche da altri segmenti, allora allo stesso indirizzo può essere associata un'altra etichetta di nome diverso e di tipo `FAR`, come mostrato nell'esempio che segue:

```
L1_far   LABEL  FAR
L1:      JMP    L2
```

`L1` sarà usata per i riferimenti dall'interno dello stesso segmento, e `L1_far` dall'esterno.

14.3.2 Variabili

Le variabili sono i simboli con cui il programmatore fa riferimento ai dati.

Gli attributi di una variabile sono: (1) l'indirizzo; (2) il tipo e (3) la visibilità. L'indirizzo e la visibilità hanno significato identico a quello illustrato per le etichette, mentre il tipo ha l'interpretazione descritta in seguito.

Come già anticipato, l'assembler della famiglia ×86 risulta fortemente "tipizzato", nel senso che nel riferimento a un dato, costante o variabile, l'assemblatore verifica che il suo uso sia congruente rispetto al tipo dichiarato al momento della sua definizione. Per esempio, se la variabile `var` è stata definita di tipo byte tramite la direttiva `DB`, lo statement `mov var,AX` non è ammesso in quanto `AX` corrisponde a 2 byte.

Per definire le variabili l'assemblatore mette a disposizione del programmatore alcuni tipi di dato elementare, che vengono indicati come tipi base o primitivi, e corrispondono essenzialmente a quelli del processore. L'utente può definire egli stesso dei nuovi tipi, quasi come avviene nei moderni linguaggi di alto livello.

I tipi predefiniti in MASM sono: `B`yte, `W`ord, `D`oubleword, `Q`uadword e `T`enbyte, corrispondenti a dati che occupano 1, 2, 4, 8, 10 byte. Essi possono essere usati per definire le variabili attraverso la direttiva `Define`, la cui sintassi è la seguente:

```
[NomeVar]   Dx   ValoreIniziale[,ValoreIniziale, .... ]
```

dove x può essere una qualunque delle lettere B, W, D, Q, T, corrispondenti a ciascuno dei tipi di dato.

La definizione di una variabile causa l'allocazione in memoria del numero di byte corrispondente al tipo, e in più specifica, mediante ValoreIniziale, il valore con cui essa deve essere inizializzata staticamente al momento del caricamento.

La specifica di una lista di ValoreIniziale nella direttiva define causa l'allocazione di un insieme di oggetti dello stesso tipo, alla stregua di un array; l'indirizzo associato alla variabile è quello del primo elemento. La dichiarazione:

```
C       DB       10,11,12,?,14
```

definisce una variabile di nome C, di tipo byte, costituita da 5 elementi, di cui il quarto non ha un valore iniziale specificato, mentre i valori iniziali degli altri sono ordinatamente 10 ,11 ,12, e 14. I cinque elementi sono a indirizzi contigui crescenti. L'istruzione mov BL,C+2 carica in BL il contenuto della posizione C+2, cioè 12.

Se ValoreIniziale è una stringa di caratteri, allora il codice ASCII relativo a ciascun carattere è posto ordinatamente nella sequenza di byte che costituisce la variabile. Per esempio le due dichiarazioni:

```
A        DB       'STACK'
A        DB       83,84,65,67,75
```

sono perfettamente equivalenti da un punto di vista funzionale, anche se la prima ha un significato molto più chiaro della seconda.

L'operatore DUP permette di effettuare in modo conciso dichiarazioni di variabili variamente strutturate. Esso ha la seguente sintassi:

```
          Repliche  DUP  (ValoreIniziale[,ValoreIniziale, ..])
```

dove il valore di Repliche definisce il numero di volte che deve essere ripetuta la dichiarazione delle variabili i cui valori iniziali sono definiti entro le parentesi. Ad esempio, lo statement:

```
ARRAY_Mono_Dim  DB      100  DUP  (0)
```

alloca un'area di memoria di 100 byte inizializzati a 0, mentre

```
        DB       10   DUP  ('STACK')
```

causa la ripetizione per 10 volte della stringa "STACK".

Poiché il DUP è un operatore, può essere utilizzato in modo ricorsivo. Ciò consente di dichiarare variabili complesse usando DUP in modo annidato.

La tipizzazione forte dell'assembler 8086 risulta, in taluni casi, come un ostacolo ai fini della ottimizzazione dell'efficienza di un programma. Il programmatore può comunque eludere la tipizzazione, usando esplicitamente alcuni operatori. Per esempio, per la variabile C definita in precedenza è possibile scrivere:

```
        mov    AX, WORD PTR C
```

dove l'operatore PTR informa l'assemblatore di generare l'istruzione che carica in modo congruente a AX quanto sta all'indirizzo di C, determinando in tal modo una conversione tra i tipi. Il risultato dell'esecuzione dell'istruzione sarà il caricamento in AL di 10 e in AH di 11.

14.3.3 Variabili strutturate

Oltre ai tipi scalari e agli array, l'assembler 8086 consente di definire variabili con struttura a record, per mezzo delle direttive STRUC/ENDS e RECORD.

La direttiva STRUC/ENDS viene usata per definire le strutture MASM, analoghe ai record Pascal, i cui campi sono variabili di tipo qualsiasi tra quelli predefiniti.

La sintassi della direttiva è la seguente:

```
NomeStruttura  STRUC
        DefinizioneCampi
NomeStruttura  ENDS
```

Il suo effetto è quello di definire una struttura, chiamata NomeStruttura, contenente i campi specificati in DefinizioneCampi. La definizione di un campo è sintatticamente identica a quella di una variabile. Pertanto i campi sono essenzialmente variabili in cui l'attributo indirizzo corrisponde allo spiazzamento entro la struttura.

La direttiva STRUC/ENDS non causa allocazione di memoria. Questa verrà allocata al momento della definizione di ciascuna variabile. Una variabile di tipo struttura può essere definita con uno statement avente la seguente sintassi:

```
[NomeVar]  NomeStruttura    <[ValoreIniz.][,ValoreIniz., .. ]>
```

dove NomeStruttura deve essere il nome di una struttura definita precedentemente, e le parentesi acute sono obbligatorie.[13]

Per fare riferimento a un campo di una variabile di tipo struttura, in modo analogo ai linguaggi di programmazione di alto livello, l'assemblatore mette a disposizione una notazione estremamente comoda, basata sull'operatore spiazzamento (.), per esprimere tale somma.

14.3.4 Costanti

La direttiva EQU permette di associare un nome simbolico a un valore numerico od alfanumerico.

```
 NomeCostante  EQU  Espressione
```

A NomeCostante viene associato il tipo "costante" e il valore ottenuto dalla valutazione della Espressione. Il valore può essere di tipo intero, reale, stringa, alias (cioè sinonimo di altri simboli) o rilocabile. Per esempio sono valide le seguenti definizioni di costanti:

```
A        EQU    100        ;A e' un numero intero di valore 100 PI      EQU    3.14
;PI e' il numero reale 3,14 B       EQU    PI*A       ;B e' il numero reale 314,0.
ERROR2  EQU    'File Not Found'   ;ERROR2 e' una stringa ASCII ERROR   EQU    ERROR2
;Alias di ERROR2 Alcool  EQU    0F001H     ;Esadecimale F001 X       EQU    0101B
;Binario 101 (decimale 5) Y       EQU    0101Bh     ;Esadecimale 101B (decimale
4123)
```

A una costante è associato, come attributo, il tipo del risultato della espressione che figura nella sua definizione. Così alla costante A nell'esempio precedente è associato il tipo intero, a PI e a B il tipo reale, mentre alla costante ERROR2 e al suo alias ERROR è associato il tipo stringa. Si noti la definizione di numeri non decimali. Onde evitare possibili ambiguità con i nomi, il numerale esadecimale F001 è stato prefisso con uno 0. Un'altra possibile ambiguità nasce quando la base è 16 e il numerale termina con una cifra B o D. Per convenzione, se un numerale termina con una lettera che corrisponde a un identificatore di base, il MASM l'interpreta sempre in tale maniera; pertanto, nell'esempio, X è interpretato come binario, mentre Y come esadecimale.[14]

La definizione di una costante non causa allocazione di memoria e, quindi, può trovarsi al di fuori della definizione di qualsiasi frammento. Se la definizione si trova all'interno di un segmento, lo ILC attivo (si veda più avanti) non viene incrementato.

Con la direttiva EQU si possono anche assegnare valori di tipo rilocabile. Per esempio, se il simbolo TOP_OF_STACK è stato definito nel segmento di stack, lo statement:

```
CIMA      EQU       TOP_OF_STACK
```

definisce il simbolo CIMA come sinonimo (alias) di TOP_OF_STACK. CIMA ha perciò tutti gli attributi di TOP_OF_STACK.

Un caso particolarmente interessante è dato dallo statement:

```
CUR_LOC  EQU       $
```

[13] La direttiva RECORD serve a definire tipi di record i cui campi sono costituiti da bit, anziché byte, per un totale di 8 o 16 bit.

[14] Se non diversamente specificato la base numerica è 10. Tramite la direttiva .RADIX, è possibile far assumere a MASM le basi 2, 8 e 16.

che assegna al simbolo CUR_LOC il valore, rilocabile, corrispondente alla locazione corrente entro il segmento corrente di assemblaggio (Paragrafo 14.5). Supponendo che sia ILC=0009 e SC=CSEG, il "valore" del simbolo CUR_LOC è dato dalla quadrupla

```
<CSEG,0009,NEAR,INTERNAL>
```

Un simbolo definito con la EQU non può essere ridefinito. Esiste però un'altra direttiva (rappresentata da = nel campo di operazione) che consente di ridefinire, da punto in cui appare in poi, una costante. Si può scrivere, ad esempio:

```
A       =       10
B       =       11
A       =       12 * B + A
```

In in sostanza, per il processo di assemblaggio, i nomi definiti con EQU costituiscono costanti mentre quelle definiti con = rappresentano variabili.

14.3.5 Procedure

Contrariamente a quanto accade nei linguaggi assemblativi della prima generazione, l'assembler 8086 consente di specificare il fatto che una sequenza di statement costituisce una procedura.

La dichiarazione di una procedura ha la seguente sintassi:

```
Nome    PROC    [Distanza]
        BloccoStatement
Nome    ENDP
```

dove BloccoStatement è una sequenza di statement che ne costituisce il corpo e Nome è un simbolo che verrà assunto come suo identificatore ed associato al suo entry point. Infatti un nome di procedura è a tutti gli effetti una etichetta, tanto che, al momento della definizione, Distanza gli associa una distanza, che viene considerata il tipo della procedura. Se tale parametro opzionale non è specificato, viene assunto NEAR.

La presenza delle parentesi PROC/ENDP è di fondamentale importanza, in quanto da esse dipende il modo in cui l'assemblatore codifica le istruzioni di ritorno da sottoprogramma (RET). Infatti, se la procedura è NEAR, l'istruzione di ritorno dovrà effettuare solo il prelievo di IP dallo stack (ritorno near), mentre se è FAR dovrà prelevare anche CS (ritorno far). Ovviamente alle due azioni corrispondono codifiche di macchina differenti.

È il caso di osservare che, mentre a una etichetta FAR è possibile fare un salto NEAR dall'interno del segmento in cui è definita, ciò è assolutamente impossibile per una chiamata di procedura, in quanto si avrebbe una inconsistenza tra il modo di chiamata e quello di ritorno (Paragrafo 14.3.1).

14.4 Direttive per il collegamento dei moduli

Alcune delle direttive dell'assemblatore sono in realtà significative anche per i programmi che operano a valle, e in particolare per il linker. Qui di seguito si considerano tre direttive: END, PUBLIC e EXTRN, le cui funzioni sono specificatamente legate alla fase del collegamento.

14.4.1 Direttive per la programmazione modulare

L'obiettivo della programmazione modulare è sostanzialmente quello di incapsulare, entro confini precisi, le funzioni richieste dal programma e le strutture di dati necessarie per realizzarle, consentendone l'utilizzo da ogni punto della applicazione, ma solo nei modi previsti. Nel caso della programmazione assembler, i confini sono sono dati dai moduli stessi.

Per rendere un simbolo disponibile anche in moduli diversi da quello in cui è definito occorre dichiararlo pubblico in quello che lo definisce. Con la direttiva PUBLIC (da non confondere con il termine omonimo usato per la combinabilità dei frammenti di segmento), l'assemblatore fornisce visibilità globale al simbolo. Essa ha la seguente sintassi:

```
PUBLIC    Nome[,Nome, ... ]
```

Per fare riferimento a un simbolo pubblico definito in un altro modulo, occorre dichiararlo mediante la direttiva EXTRN, secondo la sintassi:

```
EXTRN    Nome:Tipo[,Nome:Tipo, ... ]
```

che definisce uno o più simboli esterni al modulo. Il tipo può essere NEAR o FAR per le etichette, BYTE, WORD, DWORD, FWORD, QWORD, TBYTE per le variabili.

Ovviamente per un simbolo esterno non è definito l'indirizzo. Le istruzioni che fanno riferimento a un simbolo esterno vengono tradotte in modo incompleto. Spetta al linker completare i codici di istruzione, aggiustando gli indirizzi, in base alle informazioni della tavola dei riferimenti esterni (Paragrafo 14.5.1).

Va osservato che un simbolo dichiarato globale deve comunque essere definito con uno statement a parte, mentre la dichiarazione di un simbolo esterno costituisce a tutti gli effetti la sua definizione.

14.4.2 La direttiva END

La direttiva END chiude il modulo simbolico. Essa serve anche a specificare l'eventuale indirizzo di partenza del programma. Tale informazione viene inserita nel modulo oggetto e trasferita dal linker nell'eseguibile, in modo che il loader possa determinare a quale istruzione deve cedere il controllo dopo il caricamento del programma. La sintassi è:

```
END     [etichetta]
```

dove etichetta rappresenta l'indirizzo della prima istruzione da eseguire.

L'etichetta deve essere specificata solo nella END del modulo corrispondente al programma principale.

La necessità di specificare esplicitamente l'indirizzo di avvio del programma è dovuta al fatto che non esiste alcuna restrizione né convenzione che lo definisca implicitamente, come invece accade nei linguaggi ad alto livello.

14.4.3 La direttiva INCLUDE

Un problema che si presenta quando un programma è costituito da molti moduli è quello di mantenere la consistenza tra le definizioni dei vari oggetti presenti al loro interno. La direttiva INCLUDE costituisce uno strumento potente per affrontare questo (e non solo questo) problema. La sua sintassi ha la forma:

```
INCLUDE    NomeFile
```

dove NomeFile è un nome del file contenente il testo sorgente che si vuole importare. L'assemblatore sostituisce lo statement INCLUDE con la sequenza di statement contenuti nel file.

Se, per esempio, la definizione di tutti i simboli tramite EQU è fatta in un file, basta includere il file stesso in tutti i moduli che usano le relative definizioni. In altre parole la direttiva consente, oltre a evitare la riscrittura dei frammenti di codice che risultano di uso frequente, di centralizzare le definizioni. Un importante campo di impiego della direttiva si ha con l'impiego delle macro, cui si accenna qui di seguito.

14.4.4 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".

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 14.7.1) 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` (si veda il programma `SECONDO` (Paragrafo 14.8.2).

14.5 Il processo di traduzione

Il processo di traduzione[15] da modulo sorgente a modulo oggetto, o modulo rilocabile, è di norma in due passi, che si rendono necessari in quanto un generico statement può fare riferimento a oggetti non ancora definiti (riferimenti in avanti). Il processo di assemblaggio è schematizzato in Figura 14.7.

Il primo passo ha il compito di associare un indirizzo ai nomi di variabili e alle etichette che compaiono nel testo. Ovviamente un indirizzo è formato da due componenti, il *selettore* e lo *scostamento*. Il selettore è un simbolo, ed è dato dal nome del segmento in cui l'oggetto è dichiarato. Lo scostamento è relativo alla posizione dell'oggetto rispetto all'inizio del segmento stesso.

Per tener traccia dello scostamento, l'assemblatore utilizza una variabile designata come `ILC` (*Instruction Location Counter*). `ILC` viene azzerata all'inizio di un segmento e viene incrementata man mano che vengono esaminati i vari comandi. L'incremento è pari al numero di byte richiesti per la traduzione in linguaggio macchina dello statement in esame.

L'indirizzo assegnato dall'assemblatore non è definitivo e non corrisponde all'indirizzo fisico a cui l'oggetto verrà assegnato in memoria; si tratta di un indirizzo preso in un ipotetico spazio che parte, convenzionalmente, dalla posizione zero per tutti i segmenti. Il linker, nel generare il `.exe`, può modificare gli indirizzi assegnati dall'assemblatore, "rilocando" i vari segmenti che compaiono nei moduli oggetto. Il loader, nel sistemare il contenuto del `.exe` in memoria può di nuovo intervenire modificando gli indirizzi corrispondenti a oggetti che vengono rilocati.

Il risultato del primo passo è la tavola dei simboli, una tabella che raccoglie i nomi e gli attributi degli oggetti presenti nel modulo.[16] Un altro prodotto del primo passo è il codice intermedio, ovvero una

[15]A questo punto può essere utile saltare questo paragrafo e il Paragrafo 14.6, per tornarci in un secondo tempo, dopo aver acquisito dimestichezza con la programmazione assembler.

[16]In realtà, almeno nel caso del MASM, le tabelle sono più di una. Si tratta, evidentemente, di una scelta di convenienza di chi ha progettato l'assemblatore, ininfluente dal punto di vista concettuale.

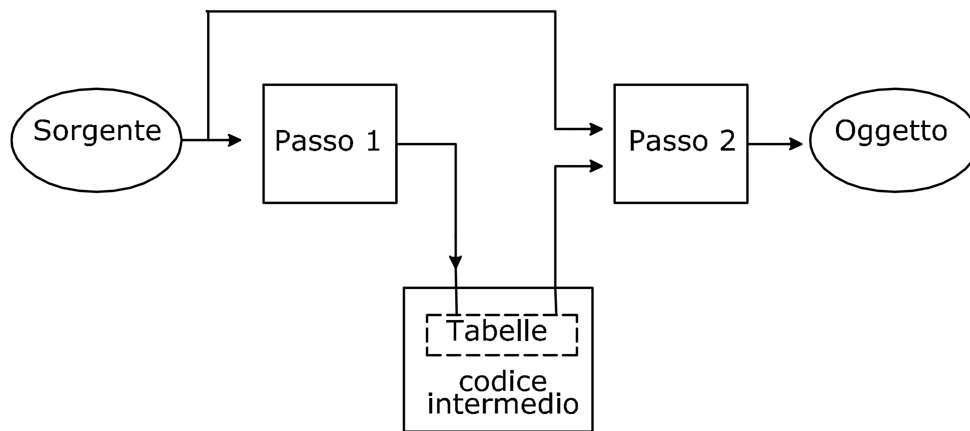

Figura 14.7 Assemblatore a due passi.

parziale traduzione del testo sorgente. La creazione del codice intermedio non è di per sé necessaria, in quanto il testo sorgente viene riletto al secondo passo, ma rappresenta un modo per rendere più efficiente il processo di traduzione. Il secondo passo ha il duplice compito di generare il modulo oggetto e di predisporre al suo interno le informazioni per il collegamento. Queste specificano le parole da modificare a cura del linker.

In Figura 14.8 viene illustrato il funzionamento complessivo di un assemblatore a due passi. La figura fa sostanzialmente riferimento al caso del MASM e mostra il dettaglio delle tabelle (dinamiche) costruite al primo passo. Si noti tra queste la tabella dei segmenti che, certamente, non viene costruita da assemblatori relativi a macchine con spazio degli indirizzi lineare.

In Figura 14.9 viene invece schematizzato il contenuto del modulo oggetto che, come si vede, contiene:

- `Nome del modulo`;
- `Nome dei frammenti di segmento` definiti nel modulo e relativa lunghezza;
- `Tavola dei simboli pubblici` (o globali, `public`), ossia la lista dei simboli che sono definiti all'interno nel modulo e ai quali può essere fatto riferimento da altri moduli; per ciascun simbolo è specificato il valore (il segmento di appartenenza e il suo scostamento entro il segmento);
- `Tavola dei riferimenti esterni` (`extrn`), cioè la lista dei simboli utilizzati, ma non definiti, all'interno del modulo e che perciò devono essere definiti in modo globale in un altro; a ciascun simbolo è associata la lista delle posizioni in cui esso è riferito all'interno del codice;
- `Codice binario` prodotto dall'assemblaggio, suddiviso in frammenti;
- `Tavola di rilocazione interna` dei frammenti, che specifica la posizione, all'interno del codice, degli oggetti che devono essere modificati qualora l'origine di ciascuno dei frammenti non coincida con quella del segmento, come ipotizzato al momento dell'assemblaggio;
- `Tavola di rilocazione esterna` dei frammenti, analoga alla precedente, ma che specifica gli oggetti da modificare in base alla posizione fisica dei segmenti nella memoria;
- `Informazioni ausiliarie`, quali per esempio l'indirizzo di avvio del programma, qualora si trovi all'interno del modulo, ovvero le informazioni usate dai *debugger* simbolici.

Il listato prodotto dall'assemblatore è un file a cui si da l'estensione `.LST`. Esso contiene il testo del programma sorgente (quello del modulo `.ASM`) e informazioni aggiuntive relative al programma in linguaggio macchina (si faccia riferimento ai listati dei Paragrafi 14.8.1 e 14.8.2).

14.5.1 Assemblatore: primo passo

Il primo passo ha lo scopo di associare scostamenti numerici a indirizzi simbolici (nomi di variabili ed etichette). In pratica tutto si riduce all'analisi sequenziale del testo, con avanzamento di `ILC` in base all'occupazione di memoria, e alla costruzione contestuale della tavola dei simboli (*symbol table* o `SYMTAB`). La variabile interna `ILC` viene impiegata per determinare la posizione occupata in memoria da oggetti come etichette e variabili. Per ciascun statement, `ILC` viene incrementata del numero di

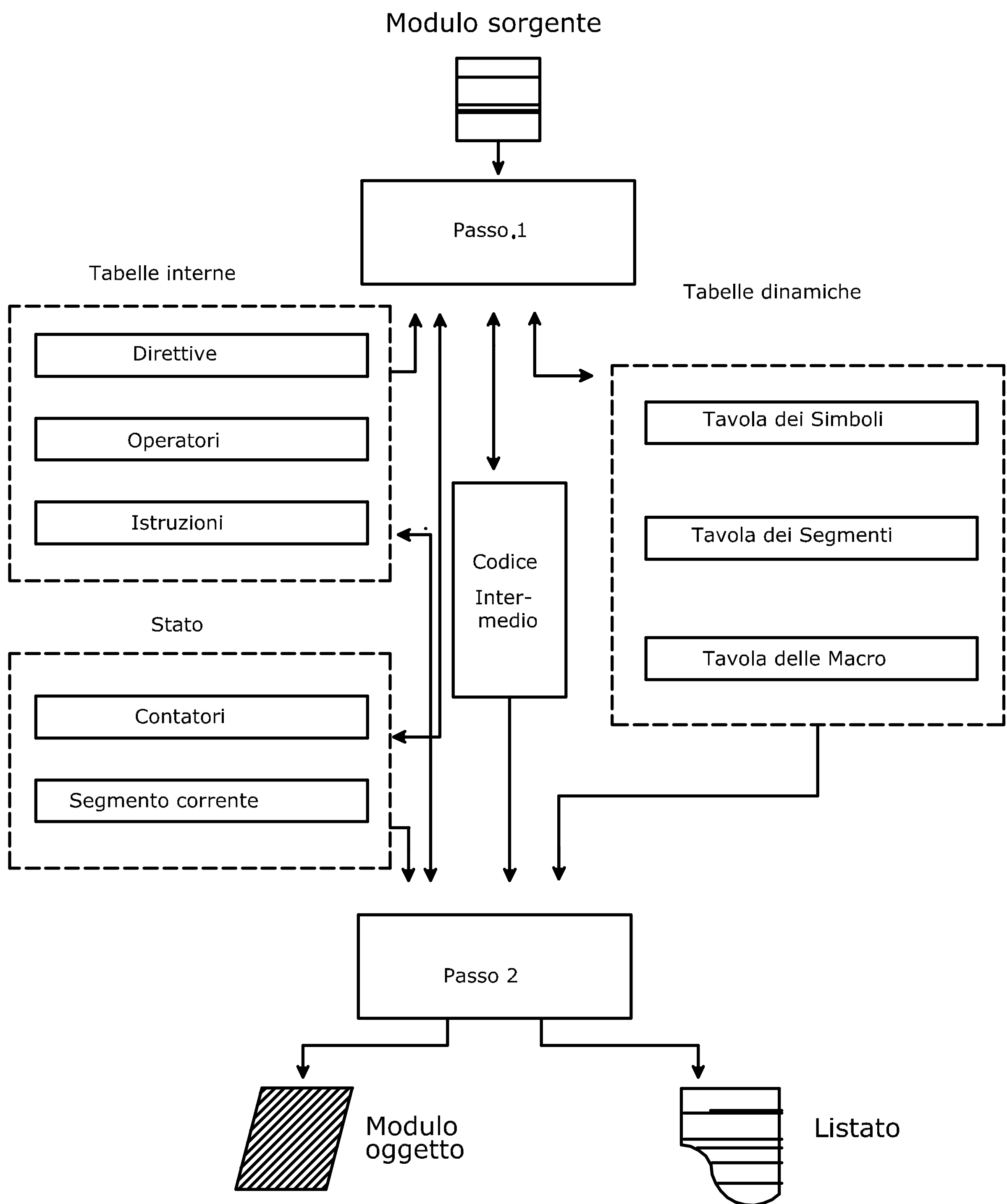

Figura 14.8 Schema di funzionamento di un assemblatore a due passi. Lo schema si riferisce sostanzialmente al MASM, ma, a parte alcuni dettagli specifici (per esempio, la tabella dei segmenti), ha valore generale. I prodotti del passo 1 sono: il codice intermedio e le tabelle dinamiche. I prodotti del passo 2 sono: il modulo oggetto e l'eventuale listato.

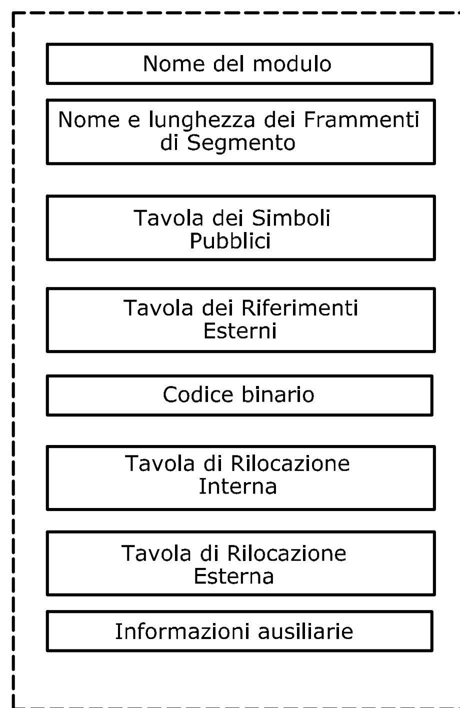

Figura 14.9 Contenuto del modulo oggetto.

locazioni occupate dal codice che esso ha generato. Ciò equivale ad assegnare sequenzialmente agli oggetti locazioni di indirizzo crescente, il che corrisponde alla sequenza esecutiva delle istruzioni ed è ovviamente corretto anche per i dati.

Schematicamente, l'algoritmo seguito dall'assemblatore consiste nel ripetere, fino a che non si incontra la direttiva `END`, questi passi:

a) lettura del prossimo statement;
b) esecuzione dell'azione determinata dal contenuto del campo `OP`, con eventuale calcolo dell'occupazione di memoria; inserimento dell'eventuale nome di variabile, di costante ecc. nella tavola dei simboli, con i suoi attributi. Per esempio: lo statement `ALFA DB ?,4,100` ha l'effetto di:
   - inserire `ALFA` nella tavola dei simboli, con il valore del corrente segmento e del corrente `ILC`;
   - quantificare in 3 byte l'occupazione di memoria;
c) incrementare ILC dell'occupazione di memoria.

Il meccanismo descritto consente di assegnare a tutti i simboli associati a locazioni di memoria il valore dell'attributo `Indirizzo`, costituito dalla coppia di attributi `<segmento,scostamento>`.

Tavola dei simboli Elemento essenziale del processo di traduzione è la tavola dei simboli. In essa vengono memorizzati, con il loro attributi, tutti i simboli introdotti dal programmatore, con l'esclusione quelli che corrispondono a nomi di macro e di segmenti, per i quali ci sono apposite tabelle (Figura 14.9). Per i programmi `PRIMO` e `SECONDO` (Paragrafi 14.8.1 e 14.8.2) sono state riportate le tavole dei simboli. Per ogni simbolo, a lato del nome che lo identifica, viene riportato il complesso degli attributi.[17] Si noti che per le costanti (numeri o testo) il valore attribuito corrisponde a quanto dichiarato, mentre per i simboli ai quali compete un indirizzo di memoria il valore attribuito è `ILC`, ovvero lo scostamento entro il segmento.

I simboli dichiarati globali (`public`) o esterni (`extrn`) entrano pure nella tavola dei simboli. Per i primi l'insieme degli attributi presenta anche l'indicazione di simbolo globale; per i secondi l'indicazione

[17]Nella terminologia di MASM, il termine `Attrib` viene impropriamente impiegato per denotare il segmento di appartenenza.

di simbolo esterno va a rimpiazzare il segmento di appartenenza. La parte della tavola dei simboli relativa a simboli globali o esterni va a costituire, previa opportuna trasformazione di rappresentazione, la tavola dei simboli pubblici e la tavola dei riferimenti esterni contenute nel formato rilocabile. Il resto della tabella viene perduto al termine dell'assemblaggio.

14.5.2 Assemblatore: secondo passo

Al secondo passo, l'assemblatore genera il modulo oggetto producendo, se richiesto dal programmatore, il listato del codice prodotto (si vedano i programmi `PRIMO` e `SECONDO` riportati più avanti). La traduzione delle istruzioni assembler avviene in accordo al formato delle istruzioni di macchina.

Il secondo passo è reso molto più efficiente del precedente se al passo uno viene prodotto tutto il codice che è possibile generare immediatamente, e si utilizza una tavola dei riferimenti in avanti per accedere alle parti che necessitano ulteriore elaborazione. In questo modo il completamento della generazione del codice può avvenire sfruttando un accesso diretto agli archivi interessati, evitando una nuova scansione sequenziale, con un notevole miglioramento nelle prestazioni dell'assemblatore.

14.6 Collegamento e caricamento

Il linker produce i programmi eseguibili (`.EXE`) a partire dai moduli oggetto generati dal programma assemblatore o dai compilatori.

Le funzioni che esso svolge sono essenzialmente due:

- raccogliere e contare i frammenti di segmento definiti nei vari moduli, in modo da costruire i segmenti logici (e gli eventuali gruppi che li contengono);
- risolvere i riferimenti incrociati tra i moduli.

Nella costruzione dei segmenti logici, i frammenti di segmento vengono trattati secondo quanto detto al Paragrafo 14.2.3. Le parti di codice che dipendono dalla posizione di ciascun frammento all'interno del segmento vengono modificate in base alla rilocazione. Per esempio, vengono modificate le istruzioni che contengono riferimenti a variabili definite nel frammento del segmento dati di un modulo giustapposto a un frammento dati di altro modulo. Non sono invece modificate le istruzioni di salto intrasegmento con indirizzamento relativo, proprio per le caratteristiche di tale modo di indirizzamento. Le tavole di rilocazione esterna dei frammenti vengono cumulate per costruire la tavola di rilocazione esterna di ciascun segmento. Questa tabella viene usata dal loader per aggiustare i riferimenti che dipendono dal segmento fisico in cui i segmenti logici verranno allocati utilizzando le informazioni di lunghezza contenute nella tabella dei segmenti di ciascun modulo.

Per quanto si riferisce alla risoluzione dei riferimenti incrociati, il linker riempie, a partire dalle tavole dei simboli pubblici di ciascun modulo, la sua tavola dei simboli, nella quale compaiono tutti gli identificatori pubblici presenti nel programma. In particolare, per ogni simbolo dichiarato esterno, il linker (in base alla tavola dei simboli pubblici) determina i valori numerici della base del segmento e dello scostamento a esso associati. Naturalmente il linker ricerca anche nelle eventuali librerie che sono state specificate nel comando che lo ha attivato.

Analogamente, dalle tavole dei riferimenti esterni viene costruita la tavola dei simboli esterni. Il normale processo di raccolta dei moduli ha perciò termine quando le due tavole dei simboli contengono gli stessi elementi, e quindi non si hanno più riferimenti esterni indefiniti. Il procedimento termina in errore se non tutti i riferimenti esterni vengono risolti.

Esaurita la fase di costruzione dei segmenti, il linker passa a quella di preallocazione, nella quale questi vengono assegnati a ipotetici segmenti fisici, supponendo che tutta la memoria fisica sia disponibile. In altri termini il primo segmento viene allocato nel segmento fisico 0, il secondo nel segmento fisico che inizia subito dopo la fine dell'area occupata dal primo segmento logico e così via. Il linker compone quindi un unico programma eseguibile posizionando in sequenza i segmenti dei vari moduli: ciascun byte del programma risulta così individuato dal suo indirizzo, a partire dall'indirizzo 0000H.

Il programma risultante può essere eseguito senza modifiche solo a partire dalla base 0000H. Una diversa base iniziale richiede la rilocazione del programma stesso, in quanto tutti gli indirizzi che vi compaiono devono corrispondere alla posizione effettiva. La rilocazione è compito del loader. Nella rilocazione, le parole del programma contenenti i segmenti devono essere modificate, in quanto a esse viene sommato il selettore iniziale. Le parole di programma contenenti scostamenti non vanno invece modificate, in quanto l'indirizzo all'interno del segmento è invariante con la posizione.

Il linker produce quindi una tavola di rilocazione, che indica gli indirizzi delle parole contenenti le basi dei segmenti. Il programma collegato e la tabella di rilocazione fanno parte del file `.EXE` relativo al programma. Un esempio di tavola di rilocazione è riportato al termine del Paragrafo 14.8.3.

14.6.1 Caricamento in memoria ed esecuzione

Il loader calcola la "base di caricamento" dell'intero programma in funzione alla zona di memoria disponibile. Trasferisce quindi il programma in memoria a partire da quel punto. Le parole da rilocare hanno come base del segmento quello indicato nella tavola di rilocazione, sommato con la base di caricamento.

Il selettore di caricamento del programma viene aggiunto anche ai due selettori che rappresentano il valore iniziale di `CS` e `SS` nella tabella numerica di Inizializzazione, prima del loro trasferimento nei corrispondenti registri.

Prima di poter eseguire il programma rilocato, devono essere trasferiti nei registri i valori che compaiono nella tabella di inizializzazione. Il trasferimento dei valori iniziali di `CS` e `IP` viene tipicamente effettuato attraverso una preliminare immissione di tali valori nelle prime parole della pila e una successiva esecuzione di una istruzione di ritorno da sottoprogramma.

14.7 Esecuzione sotto DOS

La programmazione assembler è ormai confinata alle applicazioni, o alle parti di applicazioni, in cui è necessario "vedere" direttamente la macchina. Spesso si tratta di applicazioni *embedded* funzionanti o in modo *stand-alone*, cioè senza il supporto del sistema operativo, o sotto un sistema operativo di tempo reale, che normalmente non possiede la varietà di funzioni di un moderno sistema operativo di uso generale. Il DOS fornisce un supporto minimo al programmatore (solo le funzioni elementari sotto descritte), e perciò la programmazione per DOS ha molte similitudini con la programmazione per le applicazioni appena dette. È questo il motivo per cui si è optato per la programmazione per DOS.

14.7.1 Funzioni DOS

Il sistema operativo fornisce un insieme di funzionalità, o servizi, utilizzabili dai programmi. A tale scopo il DOS riserva le interruzioni[18] da 20H a 3F per i suoi usi. Ad esempio l'istruzione `INT 20H` può essere utilizzata per concludere un programma. L'interruzione 21H è particolarmente importante perché attraverso di essa il programmatore può richiedere differenti servizi (o funzioni) al DOS.

All'atto dell'esecuzione dell'istruzione `INT 21H`, il registro `AH` deve contenere il numero che identifica la funzionalità richiesta. Eventuali parametri vengono passati al DOS attraverso i registri di CPU. In Tabella 14.1 viene riportato un sottoinsieme delle quasi cento funzioni DOS disponibili.

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` = Return code	in

Tabella 14.1 Alcune funzioni DOS. La colonna centrale 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.

14.7.2 Il PSP

È stato detto che un programma `.EXE` è in forma rilocabile. Il DOS stabilisce la posizione da cui caricarlo in base alla conoscenza dello stato di occupazione della memoria e provvede ad aggiustare i

[18] Si ricordi che l'uso dell'istruzione INT disabilita il sistema di interruzione e quindi un'eventuale funzione DOS ha la possibilità di "correre" fino al termine senza dover subire interruzioni.

riferimenti agli indirizzi rilocati. In ogni caso, al programma viene fatto procedere un blocco di 100H posizioni denominato PSP *(Program Segment Prefix)*. Entro il PSP vengono inserite alcune informazioni che il programma può utilizzare. A tal fine, all'atto del passaggio del controllo al programma, i registri di segmento DS ed ES contengono la base di PSP. In Tabella 14.2 viene mostrata parte del contenuto di PSP.

Posizione	Descrizione
00	Istruzione di ritorno al DOS (INT 20H)
02	Indirizzo finale del programma
04	Riservato
05	Chiamata al *dispatcher* del DOS
0A	Indirizzo di terminazione (INT 22H)
0E	*Ctrl Break Handler* (INT 23H)
12	*Critical Error Handler* (INT 24H)
16	Area riservata al DOS
.....	.....
81-FF	Linea di comando del programma

Tabella 14.2 Parte del contenuto di PSP. La colonna di sinistra riporta in esadecimale l'offset rispetto alla base del PSP. Quella di destra riporta il contenuto corrispondente. Dalla locazione 81 in poi si trovano gli eventuali caratteri digitati nella linea di comando dopo il nome del programma.

14.7.3 Passaggio del controllo dal DOS

I programmi eseguibili possono essere in forma `.EXE` o `.COM`. I secondi sono ottenuti dai primi sottoponendoli al programma di utilità EXE2BIN. Un programma `.COM` non può eccedere complessivamente (tra codice, dati e stack) la dimensione di un segmento (64 KB). In ambedue i casi, il codice del programma viene caricato a partire da PSP:100H. La coppia CS:IP deve contenere l'indirizzo di partenza, pertanto CS viene portato a puntare al punto di ingresso. Se questo corrisponde alla prima posizione del codice, allora IP viene portato a 100H.

Programmi EXE La coppia CS:IP contiene l'indirizzo di entrata del programma CS, DS e ES puntano a PSP. I registri SS e SP puntano rispettivamente alla testa e alla base dello stack.

Programmi COM La differenza col caso precedente sta nel fatto che tutti i segmenti sono sovrapposti, per cui anche SS punta a PSP. SP punta alla testa dello stack (che, ovviamente, si trova entro il segmento).

14.7.4 Restituzione del controllo al DOS

Sono possibili differenti modalità:

1) Uso della funzione DOS 4CH;
2) Uso dell'istruzione di ritorno contenuta nella posizione PSP:0000.

Uso della funzione DOS 4CH La chiamata a questa funzione ha come effetto la conclusione del programma. Costituisce la forma più immediata per concludere un programma e tornare a quello che lo ha attivato. Tramite il registro AL può essere fornito un codice indicativo del risultato dell'esecuzione. Il programma attivante, che potrebbe essere diverso dal DOS, ha così modo di effettuare azioni conseguenti al risultato dell'esecuzione. Per convenzione `AL` = 0 sta per "programma concluso regolarmente". I programmi `PRIMO` (Paragrafo 14.8.1) e `TERZO` (Paragrafo 14.8.3) usano questa modalità.

Uso dell'istruzione di ritorno al DOS in PSP Per tornare al DOS è possibile utilizzare l'istruzione contenuta nella prima posizione del PSP (Tabella 14.2). A tale scopo occorre che il programma termini con un salto alla posizione PSP:0000. Il programma `SECONDO` (Paragrafo 14.8.2) usa questa modalità.

Questo metodo di ritorno al DOS è quello originale della versione 1.0 del DOS, mantenuto nelle versioni successive per compatibilità. Esso è divenuto obsoleto con la versione 2.0 che ha introdotto la funzione 4CH.

14.8 Appendice - Esempi di programmi assembler

In questa sezione si riportano tre programmi assembler di crescente complessità al fine di illustrare:

a) la strutturazione dei programmi assembler e le interazioni col sistema operativo;
b) l'uso delle macro;
c) l'impiego di moduli assemblati a parte.

Per tutti gli esempi mostrati si è fatto uso dell'assemblatore di Arrowsoft Systems v1.00d, di publico dominio, menzionato all'inizio del capitolo.

14.8.1 Il programma PRIMO

Scopo:

- mostrare la struttura generale di un programma assembler e i collegamenti con il sistema operativo.

Funzionalità:

- stampare un messaggio prefissato a video;
- stampare un secondo messaggio con una parte prefissata e una seconda parte corrispondente alla linea di comando (cioè ai caratteri battuti a seguito del nome del programma).

Esemplifichiamo il funzionamento del programma PRIMO riportando una possibile sequenza di comando-risposta come appare a video.

```
C:>
C:> primo Calcolatori Elettronici
Sono il programma Primo
Hai digitato:  Calcolatori Elettronici
C:>
C:> primo
Sono il programma Primo
Non hai digitato nulla dopo Primo
C:>
```

Tenuto conto che le prime istruzioni del programma devono necessariamente caricare in DS la base del segmento dei dati e considerato che abbiamo stabilito di tornare al DOS con la funzione 4CH, il programma assume questa struttura:

```
CODE        SEGMENT
            ASSUME CS:CODE,DS:DATA,SS:STACK,ES:NOTHING
PRIMO       LABEL       FAR
            MOV         AX,DATA     ;Inizializzazione
            MOV         DS,AX       ;del registro DS
            .......
            Corpo del programma
            .......
            MOV         AH,4CH      ;Restituzione del controllo
            INT         21H         ;al DOS
CODE        ENDS        END
            END         PRIMO
```

Notare che attraverso la direttiva `END` si stabilisce che il punto di entrata è `PRIMO`. Attraverso la direttiva `LABEL` si assegna a `PRIMO` la prima posizione del programma (con distanza `FAR`).

Quando il DOS cede il controllo al programma la coppia CS:IP punta a `PRIMO`, mentre in DS c'è la base di PSP (Paragrafo 14.7.2). Poichè DS deve puntare al segmento dati (più specificatamente alla posizione `DATA`), prima di tutto è necessario assegnare a DS il valore dovuto. A ciò provvedono le prime due istruzioni del programma. A questo proposito si veda quanto detto al Paragrafo 14.2.2.

Prima di illustrare il resto del programma conviene familiarizzare con il listato prodotto dall'assemblatore. A tale scopo si faccia riferimento al listato contenuto del file `PRIMO.LST` riportato fine di questo paragrafo.

Il listato presenta 4 colonne. La prima corrisponde al numero di riga del listato (non del testo sorgente[19]); la seconda corrisponde normalmente alla posizione assegnata a una variabile o costante o istruzione nel relativo segmento; la terza contiene il codice generato, la quarta riporta la riga come appare nel testo sorgente. Consideriamo, ad esempio, la riga 68:

```
68   002E  C6 85 0029 R 24       out2:   MOV     Coda[DI],'$'
```

Il secondo campo contiene lo scostamento (`002E`) dalla base del segmento di codice (`CODE`) dell'istruzione.

Il terzo campo riporta la traduzione dell'istruzione. Si noti che si tratta di una istruzione `MOV`, che ha come operando di destinazione una posizione di memoria, identificata tramite `Coda[DI]`, e come operando sorgente l'operando immediato "`$`" codificato nell'istruzione stessa. `C6 85` rappresenta la codifica di questo particolare `MOV`; `0029` è lo scostamento di `Coda` nel suo segmento dati (`DATA`); `24` è la codifica ASCII del carattere "`$`". Il simbolo `R` sta per "rilocabile" e indica che lo scostamento `0029` può essere modificato da linker. Ciò accadrebbe, per esempio, se questo frammento di segmento `DATA` venisse posto a seguire il frammento `DATA` di un precedente modulo: indicando con L è la lunghezza del frammento `DATA` che precede, l'istruzione verrebbe modificata dal linker, ponendo `0029`$+L$ nel campo dello scostamento del primo operando dell'istruzione.

A conclusione del listato, l'assemblatore produce anche la tavola dei segmenti e la tavola dei simboli. Per i segmenti fornisce la dimensione, l'allineamento, la combinabilità e la classe. Per i simboli fornisce il tipo, il valore e il segmento di appartenenza. Si noti che tutti i simboli sono stati convertiti in caratteri maiuscoli.

Torniamo ora alla logica del programma.

Il segmento `DATA` prende la forma mostrata nel listato riportato più avanti (linee 15-34). In `DATA` vengono definite tre aree contenenti caratteri ASCII con funzioni di "messaggi' (`Mess1`, `Mess2` e `Mess3`) e un blocco di 100 byte, a partire dalla posizione `Coda` (linea 24). L'area `Coda` viene impiegata per trasferivi gli eventuali caratteri battuti a seguito del nome del programma.

Si noti che per convenzione del DOS i messaggi di stampa a video devono concludersi con il carattere "`$`". Si noti anche l'uso della direttiva `EQU` per definire i codici del "ritorno carrello" ("`CR`", *Carriage Return*) e dell'"avanzamento linea" ("`LF`", *Line Feed*). Per evidenziare la natura di questa direttiva i due statement sono stati messi fuori da qualunque segmento. I caratteri "`LF`" e "`CR`" vengono aggiunti in coda al testo dei messaggi, prima del carattere "`$`" per concludere la riga.

La stampa dei messaggi viene effettuata tramite al funzione 09H del DOS (ad esempio, la stampa di `Mess1` è alle righe 45-47.

Il programma procede attraverso questi passi:

1) a video viene stampato il messaggio `Sono il programma Primo`;
2) vengono prelevati i caratteri digitati dall'utente a seguito del nome del programma. Essi si trovano in PSP a partire dalla posizione 81H (Paragrafo 14.7.2). ES punta a PSP. I caratteri vengono trasferiti nel vettore `Coda`;
3) viene stampato a video il messaggio `Hai digitato:` seguito dal contenuto del vettore `Coda`;
4) alternativamente viene detto che non è stato battuto niente a seguito di `PRIMO`.

Il contenuto del file `PRIMO.LST` è alla pagina seguente.

Contenuto del file PRIMO.LST

```
Arrowsoft Assembler
Public Domain v1.00d (64K Model)    Page    1-1
Il programma PRIMO                          06-06-:4


1                                           page 80,132
2                                           title Il programma PRIMO
3
4    = 000D                         CR      EQU     0DH ; Carriage Return
5    = 000A                         LF      EQU     0AH ; Line Feed
6
7    0000                           STACK   SEGMENT STACK
```

[19] Ad esempio, le linee 9-10 e 17-20 sono state prodotte dall'assemblatore e non hanno un corrispondente nel testo sorgente.

```
8    0000     64 [                    DW  100 DUP(?)
9                  ????
10                      ]
11
12   00C8                     STACK   ENDS
13
14                            ; Area Dati -----------------------------------------
15   0000                     DATA    SEGMENT
16   0000  53 6F 6E 6F 20 69  Mess1   DB   'Sono il  programma Primo',LF,CR,'$'
17         6C 20 20 70 72 6F
18         67 72 61 6D 6D 61
19         20 50 72 69 6D 6F
20         0A 0D 24
21   001B  48 61 69 20 64 69  Mess2   DB      'Hai digitato: '
22         67 69 74 61 74 6F
23         3A 20
24   0029     64 [            Coda    DB  100 DUP(?)
25                  ??
26                     ]
27
28   008D  4E 6F 6E 20 68 61  Mess3   DB 'Non hai digitato nulla dopo Primo',LF,CR,'$'
29         69 20 64 69 67 69
30         74 61 74 6F 20 6E
31         75 6C 6C 61 20 64
32         6F 70 6F 20 50 72
33         69 6D 6F 0A 0D 24
34   00B1                     DATA    ENDS
35
36                            ; Segmento di codice
37                            ;-----------------------------------------------------
38   0000                     CODE    SEGMENT
39                            ASSUME CS:CODE,DS:DATA,SS:STACK,ES:NOTHING
40
41   0000                     PRIMO   LABEL   FAR
42   0000  B8 ---- R                  MOV     AX,DATA     ;inizializ-
43   0003  8E D8                      MOV     DS,AX       ;-zazione DS
44
45   0005  8D 16 0000 R               LEA     DX,Mess1    ;stampa
46   0009  B4 09                      MOV     AH,09H      ;-a video di
47   000B  CD 21                      INT     21H         ;--Mess1
48
49   000D  BE 0081                    MOV     SI,81H
50   0010  BF 0000                    MOV     DI,0
51
52                            ;Trasferimento dei caratteri in Coda da PSP
53                            ;-----------------------------------------------------
54   0013  26: 8A 04          LOOP:   MOV     AL,ES:[SI]  ;ES -> PSP
55   0016  3C 0D                      CMP     AL,CR
56   0018  74 08                      JE      FINE
57   001A  88 85 0029 R               MOV     Coda[DI],AL
58   001E  47                         INC     DI
59   001F  46                         INC     SI
60   0020  EB F1                      JMP     LOOP
61
62                            ;Stampa a video di Mess2 e di Coda
63                            ;-----------------------------------------------------
64   0022  83 FF 00           FINE:   CMP     DI,0
65   0025  75 07                      JNE     out2
66   0027  8D 16 008D R               LEA     DX,Mess3    ;stampa Mess3
67   002B  EB 0A 90                   JMP     out
68   002E  C6 85 0029 R 24    out2:   MOV     Coda[DI],'$'
69   0033  8D 16 001B R               LEA     DX,Mess2    ;stampa Mess2
70   0037  B4 09              out:    MOV     AH,09H
71   0039  CD 21                      INT     21H
72
73   003B  B4 4C                      MOV     AH,4CH      ;ritorno a DOS
74   003D  CD 21                      INT     21H
75   003F                     CODE    ENDS
76                            END     PRIMO


Arrowsoft Assembler
Public Domain v1.00d (64K Model)    Page    Symbols-1
Il programma PRIMO                      06-06-:4


Segments and Groups:
```

```
       N a m e                    Size    Align   Combine Class

CODE . . . . . . . . . . . . . .  003F    PARA    NONE
DATA . . . . . . . . . . . . . .  00B1    PARA    NONE
STACK. . . . . . . . . . . . . .  00C8    PARA    STACK

Symbols:

       N a m e                    Type    Value   Attr

CODA . . . . . . . . . . . . . .  L BYTE  0029    DATA    Length =0064
CR . . . . . . . . . . . . . . .  Number  000D
FINE . . . . . . . . . . . . . .  L NEAR  0022    CODE
LF . . . . . . . . . . . . . . .  Number  000A
LOOP . . . . . . . . . . . . . .  L NEAR  0013    CODE
MESS1. . . . . . . . . . . . . .  L BYTE  0000    DATA
MESS2. . . . . . . . . . . . . .  L BYTE  001B    DATA
MESS3. . . . . . . . . . . . . .  L BYTE  008D    DATA
OUT. . . . . . . . . . . . . . .  L NEAR  0037    CODE
OUT2 . . . . . . . . . . . . . .  L NEAR  002E    CODE
PRIMO. . . . . . . . . . . . . .  L FAR   0000    CODE

49684 Bytes free

Warning Severe
Errors  Errors
0   0
```

14.8.2 Il programma SECONDO

Scopo:

- illustrare il ricorso alle macro.
- illustrare il ricorso ai sottoprogrammi.

Funzionalità:

- acquisire una stringa di caratteri numerici da tastiera;
- stampare a video la stringa immessa in ordine di caratteri invertito se è costituita da sole cifre numeriche;
- stampare a video il messaggio `Stringa non numerica` nel caso in cui la stringa contenga caratteri non numerici.

Le macro sono state descritte al Paragrafo 14.4.4. Nel nostro caso si tratta di ricorrere alla costruzione di due macro, una per l'ingresso da tastiera e una per la stampa dei caratteri sul video.

La prima macro, viene denominata `WRITELN` e prevede come "parametro" il nome della prima posizione dell'area di memoria che contiene la stringa da stampare a video. Per la stampa si usa la funzione 09H del DOS. Come già osservato in precedenza, la stringa deve terminare con il carattere "`$`".

La macro assume la seguente forma:

```
WRITELN     MACRO   MSG              ;parametro della macro
            MOV     DX,OFFSET MSG    ;DX: offset di MSG
            MOV     AH,09H           ;Chiamata alla funzione
            INT     21H              ;09H del DOS
            ENDM
```

La seconda macro viene denominata `READLN`; anche in questo caso il parametro è il nome della prima posizione dell'area di memoria utilizzata per caricare i caratteri introdotti da tastiera. La macro utilizza la funzione DOS 0AH, che impone, per l'area di memoria, il formato schematizzato in Figura 14.10, 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 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 vengono introdotti a partire dalla posizione `BUFF`+2 indicata come `car1` nello schema precedente. Ovviamente, `n` non supererà `NMAX-1`: nel caso che vengano battuti più caratteri, vengono caricati i primi `NMAX-1` e il "`CR`" finale.

La macro assume la seguente forma:

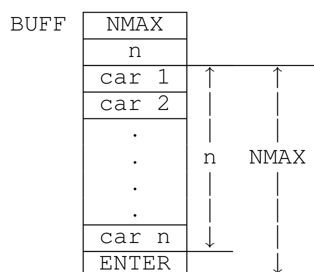

Figura 14.10 Formato dell'area di memoria per la funzione 0AH. La figura mostra il caso in cui l'area sia completamente piena.

```
READLN      MACRO   BUFF              ;parametro della macro
            MOV     DX,OFFSET BUFF    ;DX: offset di BUFF
            MOV     AH,0AH            ;Chiamata alla funzione
            INT     21H               ;DOS 0AH
            ENDM
```

Il listato del programma mostra come le macro vengono espanse: le linee corrispondenti all'espansione di una macro sono marcate con "+".

Nel caso di SECONDO si è deciso di utilizzare la modalità della versione 1.0 per il ritorno al DOS (Paragrafo 14.7.4. È dunque necessario prevedere il ritorno attraverso l'istruzione RET e salvare nello stack l'indirizzo del PSP. A ciò provvedono le prime tre istruzioni del codice (righe 63-65). Si noti che il punto di entrata (Main) viene dichiarato come procedura FAR, in modo che l'assemblatore generi il codice di ritorno compatibile per l'istruzione RET (si confronti il codice di questo RET, alla riga 96, con quello alla riga 124).

Il programma procede secondo questi passi:

1) all'atto dell'esecuzione, tramite la stampa a video di MESSAGGIO, viene richiesto l'inserimento da tastiera di una stringa di caratteri numerici;
2) successivamente viene chiamata la subroutine INVERTI, passando in SI l'offset dell'ultima posizione della stringa immessa, in DI l'offset della prima posizione di destinazione della stringa invertita e in CX il numero di caratteri della stringa. La subroutine inverte l'ordine dei caratteri inseriti se essi sono tutti numerici. Se ciò si verifica la subroutine restituisce AL = 0;
3) a seconda di AL, viene stampata a video la stringa invertita o il messaggio di errore `Stringa non numerica`;
4) il controllo è restituito al sistema operativo.

Contenuto del file **SECONDO.LST**

```
Arrowsoft Assembler
Public Domain v1.00d (64K Model)    Page    1-1
Il programma SECONDO                        06-13-:4


1                               Page    80,132
2                               Title   Il programma SECONDO
3
4                               ;Definizioni di macro
5                               ;--------------------------------------
6                               READLN  MACRO   BUFF        ;lettura
7                                       LEA     DX,BUFF     ;-in BUFF di
8                                       MOV     AH,0AH      ;--una linea
9                                       INT     21H         ;---da tastiera
10                                      ENDM
11                              ;
12                              WRITELN MACRO   MSG         ;stampa
13                                      LEA     DX,MSG      ;-a video del
14                                      MOV     AH,09H      ;--contenuto
15                                      INT     21H         ;---di MSG
16                                      ENDM
```

```
17
18                              ; Segmento di stack e di dati
19                              ;------------------------------------
20   0000                       Stack   SEGMENT STACK
21   0000     0A [                      DB  10 DUP('stack')
22                  73 74 61 63
23                  6B
24                      ]
25
26   0032                       Stack   ENDS
27
28                              ;------------------------------------
29   0000                       Data    SEGMENT
30   = 000D                     CR  EQU     0DH
31   = 000A                     LF  EQU     0AH
32   0000  49 6E 73 65 72 69    Mess    DB     'Inserire una stringa numerica >', '$'
33         72 65 20 75 6E 61
34         20 73 74 72 69 6E
35         67 61 20 6E 75 6D
36         65 72 69 63 61 20
37         3E 24
38   0020  0A 0D 53 74 72 69    M_err   DB  LF,CR,'Stringa non numerica','$'
39         6E 67 61 20 6E 6F
40         6E 20 6E 75 6D 65
41         72 69 63 61 24
42   =                          Buffer  EQU     Max
43   0037  1E                   Max     DB      Dest-strng  ; Dimensione Max
44   0038  ??                   n       DB      ?           ; # caratteri letti
45   0039     1E [              strng   DB      30 DUP(?)   ; buffer effettivo
46                   ??
47                       ]
48
49   0057     32 [              Dest    DB  50 DUP(?)   ;
50                   ??
51                       ]
52
53   0089                       Data    ENDS
54
55
56   0000                       CSec    SEGMENT
57                                      ASSUME CS:CSec,DS:Data,SS:Stack
58   0000                       Main    PROC    FAR
59   0000  1E                           PUSH    DS          ;Per tornare
60   0001  2B C0                        SUB     AX,AX       ;- al DOS
61   0003  50                           PUSH    AX          ;-- con RET
62
63   0004  B8 ---- R                    MOV     AX,Data
64   0007  8E D8                        MOV     DS,AX       ;DS -> Data
65
66                                      WRITELN Mess        ;Stampa a video di Mess
67   0009  8D 16 0000 R           +     LEA     DX,Mess     ;-a video del
68   000D  B4 09                  +     MOV     AH,09H      ;--contenuto
69   000F  CD 21                  +     INT     21H         ;---di MSG
70                                      READLN  Buffer      ;Lettura stringa
71   0011  8D 16 0037 R           +     LEA     DX,Buffer   ;-in BUFF di
72   0015  B4 0A                  +     MOV     AH,0AH      ;--una linea
73   0017  CD 21                  +     INT     21H         ;---da tastiera
74
75   0019  B9 0000                      MOV     CX,0        ;
76   001C  8A 0E 0038 R                 MOV     CL,n        ;E' stato introdotto
77   0020  80 F9 00                     CMP     CL,0        ; almeno 1 car?
78   0023  74 21                        JE      Fine        ;no
79
80   0025  BE 0039 R                    MOV     SI,OFFSET Buffer+2
81   0028  03 F1                        ADD     SI,CX
82   002A  BF 0057 R                    MOV     DI,OFFSET Dest
83   002D  B0 0A                        MOV     AL,LF
84   002F  88 05                        MOV     [DI],AL
85   0031  47                           INC     DI
86   0032  B0 0D                        MOV     AL,CR
87   0034  88 05                        MOV     [DI],AL
88   0036  47                           INC     DI
89   0037  E8 0051 R                    CALL    Inverti
90   003A  3C 00                        CMP     AL,0
91   003C  75 09                        JNE     Errore      ;Numerica ?
92                                      WRITELN Dest        ;-Si: stampa stringa a video
93   003E  8D 16 0057 R           +     LEA     DX,Dest     ;-a video del
94   0042  B4 09                  +     MOV     AH,09H      ;--contenuto
```

```
95   0044  CD 21                  +     INT     21H         ;---di MSG
96   0046  CB                 Fine:    RET
97   0047                     Errore: WRITELN M_err       ;-No: stampa errore
98   0047  8D 16 0020 R           +     LEA DX,M_err        ;-a video del
99   004B  B4 09                  +     MOV AH,09H      ;--contenuto
100  004D  CD 21                  +     INT 21H     ;---di MSG
101  004F  EB F5                        JMP     Fine
102  0051                     Main     ENDP
103
104  0051             Inverti PROC     NEAR
105                 ;
106                 ;  In ingresso a Inverti
107                 ;  SI: offset della posizione dell'ultimo carattere introdotto
108                 ;  DI: offset della prima posizione libera di Dest
109                 ;  CX: numero di caratteri introdotti
110                 ;
111  0051  4E           Ciclo:  DEC     SI
112  0052  8A 04                MOV     AL,[SI]
113  0054  3C 30                CMP     AL,'0'
114  0056  7C 0F                JL      Exit        ;Non numerico
115  0058  3C 39                CMP     AL,'9'
116  005A  7F 0B                JG      Exit        ;Non numerico
117  005C  88 05                MOV     [DI],AL     ;Numerico
118  005E  47                   INC     DI
119  005F  E2 F0                LOOP    Ciclo
120
121  0061  B0 24                MOV     AL,'$'      ;Aggiunta del '$'
122  0063  88 05                MOV     [DI],AL     ;- a fine stringa
123  0065  B0 00                MOV     AL,0        ; OK!
124  0067  C3           Exit:   RET
125  0068               Inverti ENDP
126
127  0068               CSec    ENDS
128                     END     Main


Arrowsoft Assembler
Public Domain v1.00d (64K Model)    Page    Symbols-1
Il programma SECONDO                        06-13-:4

Macros:

        N a m e                     Length

READLN . . . . . . . . . . . . .    0003
WRITELN. . . . . . . . . . . . .    0003

Segments and Groups:

        N a m e                     Size    Align   Combine Class

CSEC . . . . . . . . . . . . . .    0068    PARA    NONE
DATA . . . . . . . . . . . . . .    0089    PARA    NONE
STACK. . . . . . . . . . . . . .    0032    PARA    STACK

Symbols:

        N a m e                     Type    Value   Attr

BUFFER . . . . . . . . . . . . .    Alias   MAX
CICLO. . . . . . . . . . . . . .    L NEAR  0051    CSEC
CR . . . . . . . . . . . . . . .    Number  000D
DEST . . . . . . . . . . . . . .    L BYTE  0057    DATA    Length =0032
ERRORE . . . . . . . . . . . . .    L NEAR  0047    CSEC
EXIT . . . . . . . . . . . . . .    L NEAR  0067    CSEC
FINE . . . . . . . . . . . . . .    L NEAR  0046    CSEC
INVERTI. . . . . . . . . . . . .    N PROC  0051    CSEC    Length =0017
LF . . . . . . . . . . . . . . .    Number  000A
MAIN . . . . . . . . . . . . . .    F PROC  0000    CSEC    Length =0051
MAX. . . . . . . . . . . . . . .    L BYTE  0037    DATA
MESS . . . . . . . . . . . . . .    L BYTE  0000    DATA
M_ERR. . . . . . . . . . . . . .    L BYTE  0020    DATA
N. . . . . . . . . . . . . . . .    L BYTE  0038    DATA
STRNG. . . . . . . . . . . . . .    L BYTE  0039    DATA    Length =001E

49312 Bytes free
```

```
Warning Severe
Errors  Errors
0   0
```

14.8.3 Il programma TERZO

Scopo:

- mostrare la suddivisione del programma in moduli e il loro collegamento.

Funzionalità:

- sommare due numeri inseriti da tastiera.

Stabiliamo anzitutto che il programma procederà secondo questi passi:

a) scrittura di un messaggio di invito a digitare un numero e lettura della corrispondente stringa;
b) conversione della stringa immessa nel corrispondente numero binario. Eventuale scrittura di errore e ritorno al punto a) se la stringa introdotta non è corretta;
c) ripetizione dei passi a) e b) per il secondo numero;
d) esecuzione della somma;
e) conversione del risultato nella stringa ASCII corrispondente e sua stampa a video.

Per quanto si riferisce al punto a) conviene definire una nuova macro che sfrutta le due macro `READLN` e `WRITELN` definite in precedenza. Alla nuova macro è stato dato il nome `PREAD`. Essa prende la seguente forma:

```
PREAD   MACRO       BUFF,MSG  ;BUFF: nome del buffer di ingresso
        WRITELN     MSG       ;MSG: nome del messaggio
        READLN      BUFF
        ENDM
```

Per convenienza tutte le macro sono state raggruppate nel file `MACRO.MAC` che viene incluso nel programma.

Per quanto si riferisce al punto b), stabiliamo di costruire una subroutine, sviluppata come modulo a parte, di nome `ASC2BIN` chiamata nel modo seguente:

```
        MOV     CX,<n>
        MOV     BX,OFFSET <str>
        CALL    ASC2BIN
```

dove `n` è il numero di caratteri della stringa `<str>` da convertire. La routine `ASC2BIN` restituisce in AX il risultato della conversione e in CL un codice di errore col seguente significato:

- CL = 0 : risultato della conversione corretto;
- CL = 1 : errore, la stringa contiene caratteri non numerici;
- CL = 2 : overflow.

Per quanto riguarda il punto e) conviene sviluppare una routine, di nome `BIN2ASC`, essa pure come modulo a parte, che effettua la conversione del risultato della somma nella corrispondente stringa di caratteri ASCII. La routine `BIN2ASC` sarà chiamata secondo questa convenzione:

```
        MOV     AX,<numero>
        MOV     BX,OFFSET <ris>
        CALL    BIN2ASC
```

Per assunzione `<numero>` è un numero non negativo, ovvero è il risultato della somma, `<ris>` è il nome della posizione a partire dalla quale si dovrà trovare la stringa ASCII corrispondente alla conversione di `<ris>`.

Il programma procede attraverso questi passi:

1) ogni termine viene letto attraverso la macro `PREAD` e convertito da stringa ASCII in rappresentazione binaria tramite `ASC2BIN`. Le stringhe lette devono contenere solo caratteri ASCII numerici e devono corrispondere a numeri positivi rappresentabili su 16 bit; non è consentito inserire spazi né all'inizio né alla fine; non è consentito neanche il segno;

2) viene effettuata la somma dei due numeri;
3) il risultato della somma viene convertito in stringa ASCII; a questa viene accodato il carattere “$” (per la convenzione DOS) prima di essere stampata.

Il testo del programma TERZO e dei due sottoprogrammi sono riportati di seguito.[20]
Si noti che in TERZO le due subroutine sono dichiarate esterne (e lontane). Corrispondentemente i nomi ASC2BIN e BIN2ASC sono dichiarati pubblici (e lontani).
ASC2BIN converte una sequenza di caratteri ASCII nel corrispondente numero binario solo se la stringa è composta di soli caratteri numerici. La routine implementa questo algoritmo:

$$\begin{array}{l} \texttt{Z} \leftarrow 0; \\ \texttt{for i = 1 to } n \\ \texttt{Z} \leftarrow \texttt{Z} * 10 + \texttt{num(V[i])}; \end{array}$$

dove: V è il vettore di caratteri ASCII; n il numero di caratteri e num(c) è la funzione che trasforma la codifica ASCII di un carattere numerico nel corrispondente valore.
Nella routine DX funziona da accumulatore dei risultati parziali e all’uscita dal ciclo, contiene il risultato finale della conversione. Poiché ASC2BIN non richiede nessuna variabile di appoggio, per essa non è dichiarato il segmento dati. Il testo della routine ASC2BIN è riportato più avanti.
BIN2ASC, attraverso divisioni successive per dieci, converte il risultato finale della somma contenuto in AX in forma binaria. I resti delle varie divisioni riportati a caratteri ASCII, in sequenza invertita rispetto a quella in cui si ottengono, costituiscono infatti le cifre del risultato. Il testo di BIN2ASC è riportato a seguire quello di TERZO.

Mappa prodotta dal linker In coda ai testi è stata riportata la mappa prodotta dal linker. Si noti che è stato costruito un unico segmento di stack (STACK) e un unico segmento dati (DATA); infatti nel testo questi nomi sono pubblici (la combinabilità STACK per lo stack equivale alla combinabilità PUBLIC per i dati). Il segmento di codice del programma principale MAINSEG non è stato accorpato al segmenti di codice costruito per i sottoprogrammi SUBSEG in quanto hanno nomi diversi.
Notare anche che i segmenti DATA e SUBSEG sono stati riuniti in classi, che però, almeno in questo caso non producono un effetto aggiuntivo rispetto alla combinabilità PUBLIC.
Si osservi che il linker mette i differenti segmenti uno di seguito all’altro e fa precedere lo stack al segmento MAINSEG in cui si trova il punto di entrata in posizione 000D:0000. All’atto del caricamento i vari segmenti verranno sistemati dal loader nella memoria libera e gli indirizzi di partenza dei segmenti verranno modificati in base alla rilocazione.

Il modulo TERZO.ASM

```
            Include     MACRO.MAC   ;importazione file di macro

STACK       SEGMENT     STACK
            DW          100 DUP(0)
STACK       ENDS
;---------------------------------------------------------------
DATA        SEGMENT PUBLIC     'DATA'
LF          EQU     0AH
CR          EQU     0DH
M_IN        DB      LF,CR,'Batti un mumero intero positivo (senza il segno) >','$'
M_ERR1      DB      LF,CR,'Stringa non numerica',LF,CR,'$'
M_ERR2      DB      LF,CR,'Overflow',LF,CR,'$'
M_RIS       DB      LF,CR,'Risultato della somma: '
RISULT      DB      10 DUP(?)
BUFFER      DB      6
n           DB      ?
STR         DB      6 DUP(?)
RP          DW      ?
DATA        ENDS

;Il programma principale
;---------------------------------------------------------------
MainSEG     SEGMENT
            ASSUME CS:MainSEG,DS:DATA,SS:STACK
            EXTRN   ASC2BIN:  FAR
```

[20]Questa volta sono stati riportati i testi sorgente (i file .ASM, non i .LST).

```
            EXTRN   BIN2ASC:  FAR

TERZO       LABEL   FAR
            MOV     AX,DATA
            MOV     DS,AX

P1:         PREAD   BUFFER,M_IN     ;lettura primo termine
;----------------------------------------------------------------
            MOV     CH,0
            MOV     CL,n            ;CX= #caratteri
            MOV     BX,OFFSET STR   ;DS:BX -> STR

            CALL    ASC2BIN         ; in CL il codice di errore
            CMP     CL,1
            JE      P1              ;non numerico?
            CMP     CL,2
            JE      OFLOW1          ;Superiore alla capacità?
            JMP     OK1
OFLOW1:     WRITELN M_ERR2          ;Troppo grosso
            JMP     P1              ; Rileggere!!

OK1:        MOV     RP,AX           ;salva primo termine

P2:         PREAD   BUFFER,M_IN     ;lettura secondo termine
;----------------------------------------------------------------
            MOV     CH,0            ;Tutto
            MOV     CL,n            ;-come
            MOV     BX,OFFSET STR   ;--sopra
            CALL    ASC2BIN
            CMP     CL,1
            JE      P2
            CMP     CL,2
            JE      OFLOW2
            JMP     OK2
OFLOW2:     WRITELN M_ERR2
            JMP     P2

OK2:        ADD     AX,RP       ;Somma i due termini
            JC      OFLOWR      ;Trabocco della somma?
            MOV     BX,OFFSET RISULT
            CALL    BIN2ASC

            WRITELN M_RIS
EXIT:       MOV     AH,4CH
            INT     21H
OFLOWR:     WRITELN M_ERR2
            JMP     EXIT
MainSEG     ENDS
            END TERZO
```

Il modulo BIN2ASC.ASM

```
DATA        SEGMENT PUBLIC 'DATA'
RESTI       DB      8 DUP(?)
DIECI       DW      10
DATA        ENDS

SubSEG      SEGMENT PUBLIC  'Sub'
            ASSUME CS:SubSEG,DS:DATA
            PUBLIC  BIN2ASC
BIN2ASC PROC    FAR
;In ingresso:
; AX: numero da convertire
; BX: offset della posizione in cui va la stringa risultato della conversione

            MOV     SI,0
            MOV     CX,0

CICLO1:     MOV DX,0        ;
            DIV     DIECI           ;AX= quoziente(AX/10); DX= resto(AX/10)
            ADD     DL,'0'          ;Resto trasformato in ASCII
            MOV     RESTI[SI],DL
            INC     SI
            INC     CX
            CMP     AX,0            ;Quoziente = 0?
            JNE     CICLO1
```

```
;Inversione della stringa dei resti e deposito nel buffer indicato in ingresso
            DEC     SI
CICLO2:     MOV     AL,RESTI[SI]     ;Prende
            MOV     [BX],AL          ;-e deposita
            DEC     SI
            INC     BX
            LOOP    CICLO2
            MOV     AL,'$'           ;Aggiunta '$' a fine stringa
            MOV     [BX],AL
            RET
BIN2ASC     ENDP
SubSEG      ENDS
            END
```

Il modulo ASC2BIN.ASM

```
;Non usa variabile o dati locali (non è definito il segmento dati)

SubSEG       SEGMENT PUBLIC  'Sub'
             ASSUME  CS:SubSEG
             PUBLIC  ASC2BIN     ; ASC2BIN visibile dall'esterno

ASC2BIN PROC    FAR
;In ingresso:
;   BX: offset della posizione del primo carattere della stringa da convertire
;   CX: numero di caratteri (n) della stringa (escluso CR)

;In uscita CL= codice di errore
;   CL= 0 OK: stringa numerica e corretta; AX: risultato della conversione
;   CL= 1 Errore: stringa in ingresso non numerica
;   CL= 2 Errore: trabocco
;Algoritmo seguito : ;   z=0; ;   for (i=1;n;i++) z=z*10+num(STR[i]);

             MOV     SI,BX        ;i=0  (equivale a )
             XOR     DX,DX        ;z=0
CICLO:       MOV     AX,10
             MOV     BL,[SI]      ;prende il prossimo
             CMP     BL,'0'       ;Test
             JL      NAN          ;-per
             CMP     BL,'9'       ;--carattere
             JG      NAN          ;---numerico
;
             SUB     BL,'0'       ;da carattere a numero
             MOV     BH,0         ;BX= numero
             MUL     DX           ;z= z*10
             CMP     DX,0         ;Test di
             JNE     OFLOW        ;-overflow
             ADD     AX,BX        ;z= z+num(STR[i]
             JC      OFLOW
             MOV     DX,AX        ;z in DX
             INC     SI
             LOOP    CICLO
             RET                  ;CL=0; AX=numero

NAN:         MOV     CL,1         ;Not A Number
             RET
OFLOW:       MOV     CL,2         ;Overflow
             RET
ASC2BIN      ENDP
SubSEG       ENDS
             END
```

La mappa prodotta dal linker

```
Start        Stop    Length       Name                    Class
 00000H      000C7H  000C8H       STACK
 000D0H      0015DH  0008EH       MAINSEG
 00160H      001F9H  0009AH       DATA                    DATA
 00200H      0025AH  0005BH       SUBSEG                  SUB

Program entry point at 000D:0000
```

Esercizi

1 Si considerino le seguenti definizioni:

```
A    DB      2,4
B    EQU     1
C    DB      B
D    EQU     A+B
E    EQU     A+(C-A)
```

Si considerino le istruzioni sotto riportate. Si ipotizzi che il codice oggetto corrispondente a ciascuna di esse sia formato da un primo byte contenente MOV AL e da uno o più byte successivi contenenti l'operando dell'istruzione. Si disegni a fianco di ciascuna istruzione il formato dell'istruzione stessa; si dica come deve essere interpretato il campo dell'operando e si indichi il risultato dell'esecuzione dell'istruzione (il valore che viene portato in AL). Si assuma che la posizione assegnata a A sia la 0.

```
MOV     AL,A
MOV     AL, B
MOV     AL, C
MOV     AL, D
MOV     Al, E
```

2 Si supponga di avere un programma con questo segmento dati:

```
DATA     SEGMENT PUBLIC 'DATA'
A        DW      0,1,2,3
B        EQU     A+3
C        EQU     3
D        DB      4,5,6
DATA     ENDS
```

Si indichi per quali delle righe seguenti di programma l'assemblatore dà errore. Per quelle che non danno errore si indichi cosa si troverebbe nel registro impiegato se l'istruzione venisse eseguita.

```
MOV     AL,A
MOV     AX,A+9
MOV     AX,B
MOV     AX,C
MOV     AL,B+C
MOV     AX,B+C
```

3 Si indichi l'effetto delle singole istruzioni sotto riportate; in particolare si indichi quale elemento di `MATRIX[1..100]` viene modificato.

```
MOV     AX,N
MOV     SI,N
MOV     BX,OFFSET(MATRIX)
MOV     20 (BX+SI),AX
```

Dove `N` e `MATRIX` sono definiti come:

```
N        EQU     10
MATRIX   DW      100 DUP(99)
```

4 Si considerino le seguenti definizioni:

```
DATA     SEGMENT PUBLIC 'DATA'
A        DB      'Questa e'' la lettera A'
B        EQU     'B'
C        EQU     B+2
D        EQU     A+B
E        EQU     A+C-D
F        DB      100 DUP ('B')
DATA     ENDS
```

Si richiede:
1) il tipo (*type*) e il valore inseriti nella tavola dei simboli per `A, B, ..., F`;
2) il contenuto del registro dopo l'esecuzione dell'istruzione `MOV AL,A`;
3) il contenuto del registro dopo l'esecuzione dell'istruzione `MOV AL,B`;
4) il contenuto del registro dopo l'esecuzione dell'istruzione `MOV AL,C`;
5) il contenuto del registro dopo l'esecuzione dell'istruzione `MOV AL,D`;
6) il contenuto del registro dopo l'esecuzione dell'istruzione `MOV AH,E`.

5 Si consideri il seguente tratto di codice:

```
DATA    SEGMENT
ALFA    DW    10,120000,0,1
```

In riferimento al processo di assemblaggio e allo statement che definisce ALFA, si richiede:
1) la sequenza delle azioni eseguite al primo passo;
2) il contenuto della tavola dei simboli al termine del primo passo;
3) la sequenza delle azioni eseguite al secondo passo.

6 Si considerino le seguenti istruzioni

```
mov     var[bx],ax
mov     cs:var[bx],ax
mov     ss:[bx],ax
mov     var[bp],ax
```

dove `var` è definita nel segmento `Data` come

```
var     dw      777
```

Si assuma che i registri DS, CS e SS contengano, rispettivamente, la base del segmento `Data`, del segmento di codice e dello stack. Qual è l'effetto delle tre istruzioni?

7 C'è differenza nel codice prodotto dall'assemblatore per le due istruzioni seguenti?

```
mov     al,100[bp]
mov     al,ss:100[bp]
```

8 Assumendo che il registro BX contenga 100, si illustri l'effetto delle seguenti istruzioni

```
mov     cs:[bx],ax
mov     cs:[bx+2],cx
jmp     cs:[bx]
```

9 Facendo riferimento alla mappa generata dal linker per il programma `TERZO` e alle considerazioni fatte a pagina 31, si supponga di assegnare al segmento MainSEG la classe 'Sub'. Si indichi quali possibili differenze si avrebbero nel collegamento tra i moduli e, conseguentemente nella mappa generata dal linker.

10 Si consideri il ciclo (di 100 passi) contenuto nel seguente tratto di codice.

```
            MOV CX,100
LOOP:       ADD AX,10
            PUSH    AX
            DEC CX
            JNZ LOOP
```

Con riferimento alla CPU 8086 si valuti di quanto si deteriorano le prestazioni quando lo stack è allineato agli indirizzi dispari anziché pari. Per rispondere alla domanda si facciano ragionevoli assunzioni sull'esecuzione delle istruzioni.

11 Qui sotto vengono riportate due versioni di un tratto di programma. Si richiede una valutazione quantitativa circa la velocità di esecuzione, determinando il rapporto tra i tempi impiegati entro il ciclo nei due casi. A tale scopo si assuma che la macchina sia l'8088 (bus a 8 bit). Si facciano inoltre le seguenti assunzioni: (a) durata delle operazioni di lettura e scrittura di un byte in memoria: 4 periodi di clock; (b) ciclo di fetch equivalente a un numero di letture pari al numero di byte di cui si compone l'istruzione; (c) durata della decodifica ed esecuzione dell'istruzione (con gli operandi disponibili in CPU): 1 periodo di clock. In mancanza di un manuale che dà l'esatto formato delle istruzioni, si facciano ragionevoli assunzioni sul formato del codice.
Versione A

```
;Segmento dati
    N       EQU     100
    COST    DB      2
    VAR     DB  ?
            ...
;Segmento di codice
            MOV     AL,VAR
            MOV     CX,N
    L:      ADD     AL,COST
            SUB     CX,1
            JNZ     L
;           ....
```

Versione B

```
;Segmento dati
    N       EQU     100
    COST    EQU      2
    VAR     DB  ?
            ...
;Segmento di codice
            MOV     AL,VAR
            MOV     CX,N
    L:      ADD     AL,COST
            SUB     CX,1
            JNZ     L
;           ....
```

12 Si costruisca un programma che legge da tastiera una stringa di caratteri ASCII (ad esempio il proprio nome) e riordina i caratteri in ordine alfabetico. La stringa risultante deve essere presentata a video, mostrando tutti i caratteri componenti la stringa originale. Maiuscole e minuscole sono indifferenziate, ma nella stringa finale devono comparire nell'ordine originale. Ad esempio, la stringa "`Maria Romei`" deve dare luogo a "` aaeiiMmorR`".

13 Si ripeta l'Esercizio 12 supponendo che (tutte) le maiuscole precedano (tutte) le minuscole e che nella stringa risultante non debbano comparire lettere uguali, ma considerando una minuscola diversa dalla corrispondente maiuscola. Ad esempio, la stringa "`Maria Romei`" deve dare luogo a "` MRaeimor`".

14 Scrivere un programma che costruisce la serie di Fibonacci per i primi n numeri. La serie di Fibonacci inizia con i due numeri 0 e 1 e ottiene ogni numero successivo sommando i due precedenti. Ad esempio, per $n = 10$ la serie è

$$0, 1, 1, 2, 3, 5, 8, 13, 21$$

Il numero n deve essere dato da tastiera. Il programma deve presentare a video la serie completa. Per le operazioni di ingresso uscita si faccia eventualmente uso di parti di programma del Paragrafo 14.8, adattandole convenientemente.

15 Si costruisca la serie di Fibonacci (Esercizio 14) (presentandola a video) fino al massimo valore consentiro per n, supponendo di rappresentare i numeri su 16 bit.

16 Si ripeta l'Esercizio 15 nel caso in cui i numeri siano rappresentati su 8, 32 o 64 bit.

17 Si consideri una matrice di interi $M[m, n]$. La matrice viene memorizzata in memoria per righe, dalla riga 0 alla riga $m - 1$. Costruire una routine che effettua la somma, elemento per elemento, della riga i con la riga j e deposita il risultato nella riga i.
Il programma deve prevedere la definizione della matrice con i suoi valori iniziali (usando ad esempio la direttiva `DW`). Si suggerisce di chiamare la routine passando in SI e DI gli indici delle due righe.

18 Con riferimento all'Esercizio 17 si realizzi la routine che costruisce una vettore di $R[n]$, in cui il generico elemento $R[i]$ è la somma di tutti gli elementi che compongono la colonna i ($0 \leq i \leq n-1$) di M.

19 Con riferimento all'Esercizio 17 si costruisca la routine che effettua la somma, elemento per elemento, della colonna i con la colonna j e deposita il risultato nella colonna j.

20 Si scriva un programma in grado di eseguire la criptatura di un messaggio immesso da tastiera, basandosi sul seguente algoritmo (facente uso di una "tabella delle sostituzioni").
Detto Γ l'alfabeto con il quale si costruisce il messaggio in chiaro e Ω quello con cui se ne esegue la criptatura e fissato un valore per k, la criptatura si ottiene sostituendo la lettera in posizione n in Γ con quella in posizione $n + k$ in Ω. Per esempio, posto:

Γ: "` ABCDEFGHIJKLMNOPQRSTUVWXYZ`"
Ω: "`GTHE QUICKBSOWNFXJMPDVRLAZY`"
$k = 1$

Se il messaggio da criptare è: "`ATTACK AT DAWN`", il testo cifrato risulta essere: "`HVVH OTHVTQHAF`".

Il programma deve leggere una riga da tastiera e presentare a video il risultato della criptatura.

21 Sia dato un numero intero positivo k. Si costruisca la successione $x_0, x_1, x_2, ..., x_n$, in base alle seguenti regole:

$x_0 = k;$
$x_{i+1} = x_i/2$ per x_i pari
$x_{i+1} = 3x_i + 1$ per x_i dispari

Si congettura che la successione converga a 1 per ogni k. Si tratta di una congettura verificata sperimentalmente, ma non provata in modo formale.

Si costruisca una subroutine assembler di nome `SUCC`, chiamata passando in AX il numero k e in BX l'indirizzo (l'offset) dell'area di memoria in cui si vuole costruire la successione.

22 Con riferimento all'Esercizio 21, ricorrendo eventualmente a parti dei programmi del Paragrafo 14.8, si costruisca il programma chiamante in modo che esso legga da tastiera il numero k e presenti a video la successione dei numeri trovati.

23 Con riferimento al problema dell'Esercizio 21 si costruisca la subroutine SUCC in modo che essa equivalga a

```
void SUCC(int, int[]);
```

e sia perciò chiamabile da un programma C nel modo seguente:

```
SUCC(k, v);
```

essendo `k` una variabile intera e `v` un vettore di interi in cui la routine deve depositare, da `v[0]` a `v[n]`) la successione.

Si costruisca il programma C che chiama `SUCC` e presenta a video il contenuto del vettore `v`. Si facciano delle prove con differenti valori di k (da leggere come ingresso).

24 Si indichi con $f(x)$ il fattoriale di x, con x intero ≥ 1. Il fattoriale può essere calcolato in modo ricorsivo come

$$f(x) = x \times f(x - 1)$$

concludendo il calcolo quando x diventa uguale a 1 e ponendo (come da definizione) $f(1) = 1$.
Si costruisca una routine assembler che calcola il fattoriale secondo tale modalità. Si costruiscano due versioni della routine: la prima chiamata passando il parametro nello stack, la seconda passando il parametro in AX.

Esercizio 1

Qui di seguito si riporta (parte del) file `LST` ottenuto assemblando il sorgente. In `DATA SEGMENT` sono state raccolte le definizioni dei dati, in `CODE SEGMENT` le istruzioni.

```
                                DATA   SEGMENT

0000  02 04                     A      DB       2,4
= 0001                          B      EQU      1
0002  01                        C      DB       B
= 0001                          D      EQU      A+B
=                               E      EQU      C

                                CODE   SEGMENT

0000  A0 0000 R                        MOV      AL,A
0003  B0 01                            MOV      AL,B
0005  A0 0002 R                        MOV      AL,C
0008  A0 0001 R                        MOV      AL,D
000B  A0 0002 R                        MOV      AL,E
```

Si osservi che la direttiva `equ` non alloca memoria; conseguentemente, di fianco al relativo offset l'assemblatore pone il simbolo "=". (La direttiva `equ` sta per "Equal").

Per quanto riguarda il formato e l'effetto delle istuzioni si consideri, ad esempio, `mov al,c`. Dal listato si osserva che il *location counter* dell'istruzione successiva è avanzato di tre posizioni, dunque l'istruzione occupa tre byte, di cui il primo (`A0`) corrisponde al codice dello specifico `mov`, il secondo e il terzo (`0002`) rappresentano lo scostamento di `c` rispetto alla base del suo segmento. Dunque l'effetto dell'istruzione `mov al,c` è il caricamento in `al` del numero 1.

L'istruzione `mov al,b` viene assemblata in due byte, di cui il primo è il codice del particolare `mov` (si noti che è diverso da quello del `mov` descritto in precedenza), e il secondo è l'operando in forma immediata.

Istruzione	Effetto	Formato
mov al,a	al← 2	3 byte
mov al,b	al← 1	2 byte
mov al,c	al← 1	3 byte
mov al,d	al← 4	3 byte
mov al,e	al← 1	3 byte

Esercizio 2

Si riporta la parte di interesse del file `LST`.

```
                                DATA   SEGMENT

0000  0000 0001 0002 0003       a      dw       0,1,2,3
= 0003                          b      equ      a+3
= 0003                          c      equ      3
0008  04 05 06                  d      db       4,5,6

                                CODE   SEGMENT

0000  A0 0000 R                        mov      al,a
```

```
error : Operand types must match
0003  A1 0009 R                        mov     ax,a+9
0006  A1 0003 R                        mov     ax,b
0009  B8 0003                          mov     ax,c
000C  A0 0006 R                        mov     al,b+c
error : Operand types must match
000F  A1 0006 R                        mov     ax,b+c
```

Vale la pena fare un'osservazione sul listato e, in particolare, sulla prima e quarta riga, quelle che definiscono le posizioni `a` e `d`. Si ricordi che gli indirizzi vengono assegnati in ordine crescente ai byte e che l'architettura Intel adotta l'ordinamento Little Endian che, assegna l'indirizzo della parola al byte meno significativo. Conseguentemente per il segmento dati la situazione in memoria rappresentata a parole è la seguente:

Offset(Hex)	Parola(Hex)
00 00	00 00
00 02	00 01
00 04	00 02
00 06	00 03
00 08	05 04
00 0A	-- 06

Nel caso in cui l'istruzione coinvolga un registro a 16 bit, nella parte bassa viene caricato il byte il cui offset è indicato nella codifica dell'istruzione, mentre nella parte alta viene caricato il contenuto della locazione di memoria avente offset immediatamente superiore. L'esecuzione delle istruzioni (quelle valide) produrrebbe questi risultati:

Istruzione	Effetto
mov ax,a+9	ah← 6 al← 5
mov ax,b	ah← 2 al← 0
mov ax,c	ah← 0 al← 3
mov ax,b+c	ah← 0 al← 3

Esercizio 3

L'effetto della prima istruzione è il caricamento in `ax` del valore 10. Anche la seconda istruzione ha lo stesso effetto ma su `si`. La terza carica in `bx` lo scostamento di `matrix` rispetto al registro `ds` ovvero lo scostamento del primo elemento di `matrix`. L'ultima istruzione modifica il 31-esimo elemento di `matrix` con il valore contenuto in `ax` ovvero con 10.

Esercizio 4

Al solito si riporta la parte di interesse del file `LST`.

```
                                       DATA SEGMENT
0000  51 75 65 73 74 61 20   a         db   'Questa e'' la lettera A'
      65 27 20 6C 61 20 6C
      65 74 74 65 72 61 20
      41
= 0042                       b         equ  'B'
= 0044                       c         equ   b+2
= 0042                       d         equ   a+b
= 0002                       e         equ   a+c-d
0016  0064[                  f         db    100 dup('F')
            42
                          ]

                                       CODE SEGMENT
0000  A0 0000 R                        mov   al,a
```

```
0003  B0 42                      mov   al,b
0005  B0 44                      mov   al,c
0007  A0 0042 R                  mov   al,d
000A  B4 02                      mov   ah,e
```

Ragionando poi nello stesso modo si ha:

Istruzione	Effetto
`mov al,a`	`al← 'Q'`
`mov al,b`	`al← 'B'`
`mov al,c`	`al← 'D'`
`mov al,d`	`al← 'F'`
`mov ah,e`	`al← 2`

Esercizio 10.5

La variabile ALFA è definita come un vettore di `Double Word` e contenente quattro elementi al suo interno.

1. al primo passo si associano scostamenti numerici a indirizzi simbolici (nomi di variabili ed etichette). Tutto si riduce all'analisi sequenziale del testo con avanzamento di ILC in base all'occupazione di memoria e alla costruzione contestuale della tavola dei simboli SYMTAB;
2. poiché ALFA è dichiarata come prima variabile all'interno del programma, essa verrà inserita all'interno della tavola dei simboli associandole l'indirizzo logico DS:0 (essendo 0 il valore di ILC);
3. al secondo passo l'assemblatore produce il codice oggetto dello statement sopra riportato ovvero genera i quattro numeri.

Esercizio 6

- La prima istruzione scrive il contenuto del registro AX dentro la locazione di memoria il cui indirizzo logico è DS:BX+777;
- La seconda scrive il contenuto del registro AX dentro la locazione di memoria il cui indirizzo logico è CS:BX+777;
- La terza istruzione muove il registro AX dentro la locazione di memoria il cui indirizzo logico è SS:BX;
- La quarta istruzione muove il contenuto di AX dentro la locazione di memoria all'indirizzo logico SS:BP+777.

Esercizio 7

Le due istruzioni hanno un effetto identico, entrambe collocano in `AL` il valore contenuto nello stack alla locazione `BP:100`. Il codice prodotto dall'assemblatore vale: `8A 46 64`. Nel quale:

- `8A` rappresenta l'opcode del `MOV`, quando viene utilizzato con una sorgente, registro o memoria, a 8 bit e con la destinazione, un registro, a 8 bit.
- `46` è la codifica che specifica di utilizzare i registri `AL` e `BP` (rispetto allo stack) per l'operazione richiesta.
- `64` è la codifica esadecimale dello scostamento scritto in forma decimale nell'istruzione, 100d = 64h.

Esercizio 8

L'effetto delle istruzioni è riportato in commento:

- `mov cs:[bx],ax          ;M[cs:100]←AX`
- `mov cs:[bx+2],cx        ;M[cs:102]←CX`
- `jmp cs:[bx]             ;IP←100`

Esercizio 9

Inserendo la classe 'Sub' al segmento MainSEG la mappa generata dal linker diventa la seguente:

```
Start  Stop   Length Name                    Class
00000H 000C7H 000C8H STACK
000D0H 00169H 0009AH DATA                    DATA
00170H 001FDH 0008EH MAINSEG                 SUB
00200H 0025EH 0005FH SUBSEG                  SUB

Program entry point at 0017:0000
```

La differenza fondamentale, rispetto alla mappa di pagina 397 del testo consiste nel fatto che, in questo caso, i segmenti non appaiono nella tabella secondo l'ordine in cui il collegatore li incontra nella scansione dei moduli rilocabili da collegare. Il linker associa a tutti i segmenti appartenenti alla classe SUB locazioni contigue di memoria. La funzione della classe è infatti proprio questa; associando un segmento ad una classe si ha la possibilità, in fase di collegamento, di raccogliere frammenti di segmento di tipo correlato.

Esercizio 10

Il tratto di codice è un ciclo ripetuto per 100 volte. Supponiamo che le istruzioni richiedano i seguenti clock per istruzione c_{pi}:

MOV	6
ADD	6
PUSH	11 (quando lo stack è allineato ai pari)
DEC	6
JNZ	6

La Tabella **??** riporta per ciascuna istruzione il numero di volte che viene eseguita n_i, la relativa frequenza x_i e il c_{pi}. Dalla Tabella si ricava $C_{PI} = \sum x_i c_{pi} = 7,2499$.

Istruzione	n_i	x_i	c_{pi}	$x_i \cdot c_{pi}$
MOV	1	0,24%	6	$0,0144$
ADD	100	24,95%	6	$1,497$
PUSH	100	24,95%	11	$2,7445$
DEC	100	24,95%	6	$1,497$
JNZ	100	24,95%	6	$1,497$
	401	100%		$7,2499$

Tabella 14.3 Prestazioni del programma dell'Esercizio 10 nel caso di stack allineato.

Se lo stack non è allineato, il numero di accessi alla memoria dell'istruzione PUSH risulta raddoppiato. Supponendo pari a 4 il numero di cicli richiesto per l' ulteriore accesso alla memoria, il c_{pi} dell' istruzione PUSH diventa 15. La Tabella 14.4 illustra il cambiamento. Dalla stessa si ha che $C_{PI} = \sum x_i c_{pi} = 8,2479$.

Istruzione	n_i	x_i	c_{pi}	$x_i \cdot c_{pi}$
MOV	1	0,24%	6	$0,0144$
ADD	100	24,95%	6	$1,497$
PUSH	100	24,95%	15	$3,7425$
DEC	100	24,95%	6	$1,497$
JNZ	100	24,95%	6	$1,497$
	401	100%		$8,2479$

Tabella 14.4 Prestazioni del programma dell' Esercizio 10 nel caso di stack non allineato.

Dunque le prestazioni degradano del rapporto $7.2499/8,2479$, ovvero del 12%.

Esercizio 11

Si osservi anzitutto che la differenza tra le due versioni del codice riguarda solo l'istruzione di somma, che nel secondo caso usa un operando immediato, mentre nel primo fa riferimento alla memoria. Si ipotizza che il codice oggetto corrispondente a ogni istruzione, sia formato da:

a) un byte contenente il codice operativo e la codifica del registro implicato;
b) un byte se l'operando è immediato o due se è in memoria.

Per valutare le prestazioni in velocità dei due programmi si calcola il numero totale di periodi di clock necessari per l'esecuzione. In base ai dati del testo dell' esercizio, il numero di periodi di clock richiesto dalle singole istruzioni per la loro esecuzione, è quello riportato nelle tabelle sottostanti.

Versione A:

	Istruzione		Formato	Periodi di clock	
	MOV	AL,VAR	3 byte	12+4+1	(n_1)
	MOV	CX,N	2 byte	8+1	(n_2)
L:	ADD	AL,COST	3 byte	12+4+1	(n_3)
	SUB	CX,1	2 byte	8+1	(n_4)
	JNZ	L	3 byte	12+1	(n_5)

Versione B:

	Istruzione		Formato	Periodi di clock	
	MOV	AL,VAR	3 byte	12+4+1	(n_1)
	MOV	CX,N	2 byte	8+1	(n_2)
L:	ADD	AL,COST	2 byte	8+1	(n_3)
	SUB	CX,1	2 byte	8+1	(n_4)
	JNZ	L	3 byte	12+1	(n_5)

Le prime due istruzioni nelle due differenti versioni richiedono lo stesso numero di periodi di clock. Per le tre istruzioni che compongono il ciclo la differenza è relativa solo alla prima istruzione (17 contro 9 periodi). Il numero di periodi di clock richiesti per il ciclo è $N \times (n_3 + n_4 + n_5)$, con N=100. Trascurando n_1 e n_2, il numero totale di periodi di clock necessari per l'esecuzione delle due versioni risulta:
Versione A: $100 \times 39 = 3900$
Versione B: $100 \times 31 = 3100$
da cui si deduce che la versione B è più veloce di circa il 26%.

Esercizio 12

L'esercizio può essere visto sotto due diversi aspetti, come un riordino di una stringa oppure come una creazione di una nuova stringa, a partire dalla sorgente, secondo certe regole.

Ordinamento: L'esercizio richiedere il riordinamento degli elementi di un vettore di caratteri (la stringa letta da tastiera), con la complicazione di considerare maiuscole e minuscole idempotenti. Qui di seguito viene dato l'algoritmo in C. A tale scopo si indica con `string[]` il vettore contente la stringa ASCII e `Lenght(V[])` una funzione che restituisce il numero dei caratteri contenuti in `V[]`. Con `M(c)` si indica una funzione che restituisce il valore numerico (la codifica ASCII) corrispondente alla forma maiuscola del carattere `c`. In altre parole, se `c='x'` o `c='X'` la funzione `M(c)` restituisce comunque `'X'`. Con `m(c)` si indica la funzione complementare di `M`, ovvero la funzione che restituisce la codifica della forma minuscola.

L'algoritmo di riordinamento opera in modo convenzionale, confrontando un carattere (a partire dal primo) con tutti i successivi e cambiandoli se il primo segue il secondo nell'ordine alfabetico. L'impiego delle due funzioni `M` e `m` garantisce il riordinamento nel modo richiesto. In conclusione l'algoritmo è il seguente:

```
for(i=0;i<Lenght(string[]);i++) {
    for(j=i+1;j<Lenght(string[]);j++)
    {
        if((M(string[i])>M(string[j])||(m(string[i])>m(string[j])))
        {
```

```
            temp=string[j];
            string[j]=string[i];
            string[i]=temp;
        }
    }
}
```

Questo algoritmo ha una complessità pari a

$$C = \frac{n(n-1)}{2}.$$

Soluzione alternativa: Una tecnica alternativa, che sarà utile per il prossimo esercizio, consiste nel confrontare i caratteri di `string[]` con tutte le lettere dell'alfabeto in ordine e trasferire al vettore `Dest[]` i caratteri di `string[]` quando uguagliano la lettera di confronto. Sfruttando la funzione `M(c)` vista in precedenza, il confronto può essere riferito alle sole lettere minuscole.

```
j=0;
    for(c="a";c<="z";c++) {
        for(i=0;i<Lenght(string[]);i++){
            if((string[i]==c)||(string[i]==M(c))){
                Dest[j]=string[i];
                j++;
            }
        }
    }
```

La complessità del seguente algoritmo è lineare, supponendo la stringa lunga n caratteri, si avrebbe:

$$C = 26 \cdot 2 \cdot n = 52n$$

Se n è maggiore di 26 questa sarebbe la soluzione migliore in quanto sarebbe una funzione che cresce linearmente con n, mentre la complessità della soluzione precedente sale nell'ordine di n^2. Nel caso però che n sia minore di 26 caratteri la prima soluzione è migliore.

Nella parte seguente si mostra una traduzione in linguaggio Assembler del secondo algoritmo. A tal fine faremo esplicito riferimento al programma SECONDO degli approfondimenti, usando le macro `WRITELN` e `READLN` definite in tale contesto. La traduzione richiede pochi accorgimenti:

- La definizione di due aree di memoria `String` e `Dest` corrispondenti ai due vettori, facendo attenzione al fato che `string` deve essere definito in modo tale da soddisfare le assunzioni delle macro `WRITELN` e `READLN`.
- L'impiego del registro `SI` come indice `i`.
- L'impiego del registro `DI` come indice `j`.
- L'impiego di `CX` come valore restituito da `Lenght(V[])`.
- L'uso di `DL` e `DH` come indice `c`.

Il file `Macro.mac` contiene le routine di lettura e scrittura, che sono spiegate in dettaglio a pagina 388 del testo e seguenti.

```
;Definizioni di macro
;--------------------------------------

INCLUDE Macro.Mac

; Segmento di stack e di dati
;--------------------------------------
Stack   SEGMENT STACK
    DB  10 DUP('stack')
Stack   ENDS
;--------------------------------------

Data    SEGMENT
CR      EQU     0DH
LF      EQU     0AH
Mess    DB      'Inserire una stringa>', '$'
```

```
Buffer  EQU     Max
Max     DB      Dest-strng                          ;Dimensione Max
n       DB      ?                                   ; # caratteri letti
strng   DB      30 DUP(?)                           ; buffer effettivo
Dest    DB      50 DUP(?)                           ;

Data    ENDS


CSec    SEGMENT
ASSUME CS:CSec,DS:Data,SS:Stack

Main    PROC    FAR

        MOV     AX,Data
        MOV     DS,AX              ;DS -> Data

        WRITELN Mess               ;Stampa a video di Mess
        READLN  Buffer             ;Lettura stringa

        MOV     CX,0
        MOV     CL,n               ;E' stato introdotto
        CMP     CX,0               ; almeno 1 car?
        JE      Fine               ;no

        MOV     SI,OFFSET Buffer+1 ;muove SI all'inizio della stringa inserita
        MOV     DI,OFFSET Dest     ;muove DI all'inizio della stringa di arrivo
        MOV     AL,LF              ;Sposta LF in AL
        MOV     [DI],AL            ;Scrive in cima all'arrivo il valore LF
        INC     DI                 ;punta al successivo carattere
        MOV     AL,CR              ;idem a prima ---
        MOV     [DI],AL            ;---con CR---
        INC     DI                 ;-----

        CALL    Ordina             ;chiama la funzione di ordinamento

        WRITELN Dest               ;Stampa l'arrivo a video

Fine:   MOV     AH,4CH
        INT     21H

Main    ENDP

Ordina PROC     NEAR
;  In ingresso a Ordina
;  SI: offset della posizione dell'ultimo carattere introdotto
;  DI: offset della prima posizione libera di Dest
;  CX: numero di caratteri introdotti

        PUSH    SI         ;salva nello stack il valore di SI (fine stringa ingresso)
        PUSH    CX         ;salva nello stack il valore di CX (lunghezza stringa)

        MOV     DL,41h     ;inizializza DL con la 'A'
        MOV     DH,61h     ;inizializza DH con la 'a'

Findc:  INC     SI
        MOV     AL,[SI]    ;sposta il carattere in AL
        CMP     AL,DH      ;controlla quale lettere minuscola sia
        JE      Xfer       ;nel caso di OK, passa alla stampa
        CMP     AL,DL      ;controlla se è una lettera maiuscola
        JE      Xfer       ;nel caso di OK, passa alla stampa

        JNE     Escape     ;dopo le prime 2 lettere (a,A) passa alle successive


Xfer:   MOV     [DI],AL    ;scrive la lettera trovata
        INC     DI

Escape: DEC     CX
        JNZ     Findc      ;Riesegue il ciclo di ricerca fino alla fine della stringa
                           ;non è detto infatti che la 'a' sia per forza
                           ;l'ultima lettera della stringa inserita....
                           ;Nel caso che la lettera in questione non sia stata trovata
        ADD     DL,1       ;passa al carattere maiuscolo successivo
        ADD     DH,1       ;passa al carattere minuscolo successivo
        CMP     DL,5Bh     ;controlla se si è arrivati dopo la 'Z'==5Ah

;---------------------- Nota la 'z' vale 7Ah  ----------------------
```

```
        JE      Exit        ;se sono uguali esce dalla procedura

        POP     CX          ;ripristina il valore di CX (per i loop)
        POP     SI          ;ripristina il valore di SI, per tornare in
                            ;fondo alla stringa

        PUSH    SI          ;risalvo il valore SI
        PUSH    CX          ;risalvo il valore CX

        JMP     Findc

Exit:   MOV     AL,'$'      ;Aggiunta del '$'
        MOV     [DI],AL     ;- a fine stringa
        MOV     AL,0        ; OK!RET
        POP     CX
        POP     SI
        RET

Ordina ENDP

CSec    ENDS
    END     Main
```

Esercizio 13

Questo esercizio potrebbe essere risolto come un problema di riordinamento, seguito dall'eliminazione dei caratteri duplicati. Indicando ancora con `string[]` il vettore contentente i caratteri ASCII letti da tastiera, il riordinamento risulta più semplice rispetto al caso dell'Esercizio 12, infatti vale la seguente relazione di ordinamento

$$z > .... > b > a > Z > ... > B > A.$$

L'algoritmo di riordinamento sarebbe quindi:

```
j=0;
for(c="a";c<="Z";c++)
     {   for(i=0;i<Lenght(string[]);i++)
         {
         if((string[i]==c))
              {
              Dest[j]=string[i];
              j++;
              }
         }
     }
```

Successivamente si dovrebbe prevedere l'eliminazione dei doppioni dalla stringa del risultato in accordo con le specifiche del testo dell'esercizio.

Nella codifica ASCII vale la relazione $z > \ldots > b > a > Z > \ldots > B > A$ (avendo indicato con A, ... Z, a, ... z, il valore numerico del codice della lettera), quindi otteniamo la soluzione dell'esercizio modificando l'algoritmo della soluzione alternativa dell'Esercizio 12 in modo da:

- Uscire dal ciclo di scansione della stringa, per passare alla lettera successiva, se la lettera in esame è stata trovata in `string[]`, (così da copiarne una sola copia in `Dest[]`)
- Utilizzare come limiti della ricerca sull'alfabeto la coppia di lettere a e Z.

La complessità del seguente algoritmo è identica alla precedente, in quanto non conoscendo la composizione della stringa, di deve comunque controllare la presenza di tutti i caratteri test. Vale allora, con n lunghezza della stringa:

$$C = n \cdot (26 + 26).^{21}$$

[21] In realtà sarebbe maggiore, in quanto per sfruttare la condizione lineare $A < Z < a < z$, si deve passare da alcuni caratteri della tabella ASCII non convenzionali. Si invita a verificare il contenuto della tabella ASCII.

```
;Definizioni di macro

Include Macro.mac

; Segmento di stack e di dati
;-------------------------------------
Stack   SEGMENT STACK
    DB  10 DUP('stack')
Stack   ENDS

;-------------------------------------
Data    SEGMENT
CR      EQU     0DH
LF      EQU     0AH
Mess    DB      'Inserire una stringa>', '$'
Buffer  EQU     Max
Max     DB      Dest-strng                      ; Dimensione Max
n       DB      ?                               ; # caratteri letti
strng   DB      30 DUP(?)                       ; buffer effettivo
Dest    DB      50 DUP(?)

Data    ENDS


CSec    SEGMENT
ASSUME CS:CSec,DS:Data,SS:Stack

Main    PROC    FAR

        MOV     AX,Data
        MOV     DS,AX               ;DS -> Data

        WRITELN Mess                ;Stampa a video di Mess
        READLN  Buffer              ;Lettura stringa

        MOV     CX,0
        MOV     CL,n                ;E' stato introdotto
        CMP     CX,0                ; almeno 1 car?
        JE      Fine                ;no

        MOV     SI,OFFSET Buffer+1  ;muove SI all'inizio della stringa inserita
        MOV     DI,OFFSET Dest      ;muove DI all'inizio della stringa di arrivo
        MOV     AL,LF               ;Sposta LF in AL
        MOV     [DI],AL             ;Scrive in cima all'arrivo il valore LF
        INC     DI                  ;punta al successivo carattere
        MOV     AL,CR               ;idem a prima ---
        MOV     [DI],AL             ;---con CR---
        INC     DI                  ;-----

        CALL    Ordina             ;chiama la funzione di ordinamento

        WRITELN Dest                ;Stampa l'arrivo a video

Fine:   MOV     AH,4CH
        INT     21H

Main    ENDP

Ordina PROC     NEAR
;  In ingresso a Ordina
;  SI: offset della posizione dell'ultimo carattere introdotto
;  DI: offset della prima posizione libera di Dest
;  CX: numero di caratteri introdotti

        PUSH    SI          ;salva nello stack il valore di SI (fine stringa ingresso)
        PUSH    CX          ;salva nello stack il valore di CX (lunghezza stringa)

        MOV     DL,41h      ;inizializza DL con la 'A'

Findc:  INC     SI
        MOV     AL,[SI]     ;sposta il carattere in AL
        CMP     AL,DL       ;controlla quale lettere minuscola sia
        JE      Xfer        ;nel caso di OK, passa alla stampa

        JNE     Escape      ;dopo le prime 2 lettere (a,A) passa alle successive

Xfer:   MOV     [DI],AL     ;scrive la lettera trovata
        INC     DI
```

```
        JMP     Next

Escape: LOOP    Findc       ;Riesegue il ciclo di ricerca fino alla fine della stringa
                            ;non è detto infatti che la 'a' sia per forza
                            ;l'ultima lettera della stringa inserita....
                            ;Nel caso che la lettera in questione non sia stata trovata
Next:   ADD     DL,1        ;passa al carattere maiuscolo successivo
        CMP     DL,7Bh      ;controlla se si è arrivati dopo la 'Z'==5Ah

;-----------------------
;Nota la 'z' vale 7Ah
;-----------------------
        JE      Exit        ;se sono uguali esce dalla procedura

        POP     CX          ;ripristina il valore di CX (per i loop)
        POP     SI          ;ripristina il valore di SI, per tornare in
                            ;fondo alla stringa

        PUSH    SI          ;risalva il valore SI
        PUSH    CX          ;risalva il valore CX

        JMP     Findc

Exit:   MOV     AL,'$'      ;Aggiunta del '$'
        MOV     [DI],AL     ;- a fine stringa
        MOV     AL,0        ; OK!RET
        POP     CX
        POP     SI
        RET

Ordina ENDP

CSec    ENDS
    END     Main
```

Esercizio 14

La successione di Fibonacci si realizza impostando due valori di partenza: 0, 1 e sommando iterativamente. L'$n + 2_{esimo}$ termine è risultato della somma dei termini n_{esimo} e $n + 1_{esimo}$.

Il programma chiede a video il numero di iterazioni richieste. Dopo una fase d'inizializzazione dei registri, un ciclo provvede a sommare progressivamente i termini della successione e a stamparli a video.

Ai fini della semplificazione del testo del programma, l'ingresso di un numero da tastiera viene effettuato tramite la routine `INPUT`, riportata al termine.

```
        INCLUDE MACRO.MAC

STACK   SEGMENT STACK
        DW      50  DUP(0)
STACK   ENDS

DATA    SEGMENT PUBLIC  'DATA'

LF      EQU     0AH
CR      EQU     0DH
OFLOW   DB      LF,CR,'Overflow',LF,CR,'$'
M_RIS   DB      LF,CR,'Termine della successione: '
RISULT  DB      5   DUP(?)
DATA    ENDS

MainSEG SEGMENT
        ASSUME CS:MainSEG, DS:DATA, SS:STACK

        EXTRN   INPUT:  FAR
        EXTRN   BIN2ASC:FAR

FIBO    LABEL   FAR
        MOV     AX, DATA
        MOV     DS, AX

        CALL    INPUT               ; chiede un carattere a video

        PUSH    AX                  ; salva i registri
        PUSH    BX                  ; nello stack
        MOV     AL,'0'
```

```
        MOV     RISULT,AL
        MOV     AL,'$'
        MOV     RISULT+1,AL
        WRITELN M_RIS               ; stampa a video 0
        WRITELN RISULT
        POP     BX                  ; ripristina i
        POP     AX                  ; registri dallo stack

        MOV     CX, 0               ; muove i primi
        MOV     DX, 1               ; due valori in CX, DX

LOOP:   MOV     BX, CX              ; per far avanzare la successione
        ADD     CX, DX              ; somma tra i termini
        JC      OFLOWR              ; c'è overflow???

        PUSH    AX                  ; salva i registri per la stampa
        MOV     AX, CX              ; a video
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     BX, OFFSET RISULT
        CALL    BIN2ASC             ; prepara la stringa di uscita
        WRITELN M_RIS               ; stampa la stringa di uscita
        POP     DX
        POP     CX
        POP     BX                  ; ripristino della situazione
        POP     AX                  ; pre-stampa

        MOV     DX, BX              ; scrive il nuovo valore di BX
                                    ; fa avanzare la successione

        DEC     AX                  ; decrementa 'n'
        JNZ     LOOP                ; cicla il tutto fino a 'n=0'


EXIT:   MOV     AH, 4CH             ; uscita dal programma
        INT     21H

OFLOWR: WRITELN OFLOW               ; uscita per overflow
        JMP     EXIT

MainSEG ENDS
        END     FIBO
```

La routine `INPUT` corrisponde al codice seguente. La routine ritorna al chiamante solo quando viene battuto un numero intero inferiore al massimo rappresentabile.

```
Include MACRO.MAC

DATA    SEGMENT PUBLIC  'DATA'
CR      EQU     0DH
LF      EQU     0AH
M_ERR2  DB      LF,CR,'Overflow',LF,CR,'$'
M_ERR1  DB      LF,CR,'Stringa non numerica',LF,CR,'$'
TESTO1  DB      LF,CR,'Inserisci un numero positivo intero: ','$'
BUFFER  DB      50
n       DB      ?
STR     DB      50 DUP(?)
DATA    ENDS


SubSEG  SEGMENT PUBLIC      'Sub'
    ASSUME  CS:SubSEG,DS:DATA
    PUBLIC  INPUT

EXTRN ASC2BIN: FAR          ;la routine ASC2BIN presentata a pagina 396 del testo.

INPUT   PROC    FAR

;In uscita AX contiene (in forma binaria) il valore immesso dall'utente

P1:     PREAD   BUFFER,TESTO1
        MOV     CH, 0
        MOV     CL, n               ;CX = #caratteri
        MOV     BX, OFFSET STR      ;DS:BX -> STR

        CALL    ASC2BIN             ; in CL il codice di errore
```

```
        CMP     CL, 1
        JE      STRING              ;numerico ?
        CMP     CL, 2
        JE      OFLOW               ;Superiore alla capacità?
        JMP     OK
OFLOW:  WRITELN M_ERR2              ;Troppo grosso
        JMP     P1                  ; Rileggere!!

STRING: WRITELN M_ERR1              ;Troppo grosso
        JMP     P1                  ; Rileggere!!

OK:     RET

INPUT   ENDP
SubSEG  ENDS
        END
```

Esercizio 15

Il programma seguente, diversamente da quello precedente, non chiede il numero di termini della successione da stampare a video ma cicla fino al raggiungimento dell'overflow. Anche in questo caso i registri utilizzati sono a 16 bit.

```
        INCLUDE MACRO.MAC

STACK   SEGMENT STACK
        DW      50  DUP(0)
STACK   ENDS

DATA    SEGMENT PUBLIC  'DATA'
LF      EQU     0AH
CR      EQU     0DH
OFLOW   DB      LF,CR,'Overflow',LF,CR,'$'
M_RIS   DB      LF,CR,'Termine della successione: '
RISULT  DB      5   DUP(?)
DATA    ENDS

MainSEG SEGMENT
        ASSUME CS:MainSEG, DS:DATA, SS:STACK

EXTRN   BIN2ASC:FAR

FIBO    LABEL   FAR
        MOV     AX, DATA            ; mette in DS l'indirizzo
        MOV     DS, AX              ; del segmento DATA

        PUSH    AX                  ; salva i registri AX, BX
        PUSH    BX
        MOV     AL,'0'
        MOV     RISULT,AL
        MOV     AL,'$'
        MOV     RISULT+1,AL
        WRITELN M_RIS               ; stampa a video 0
        WRITELN RISULT
        POP     BX
        POP     AX                  ; ripristina AX, BX

        MOV     CX, 0               ; carica i primi due
        MOV     DX, 1               ; termini della successione

LOOP:   MOV     BX, CX              ; salva CX per far avanzare la successione
        ADD     CX, DX              ; somma dei registri
        JC      OFLOWR              ; overflow???
        MOV     AX, CX              ; prepara il risultato per la stampa
        PUSH    BX                  ; salva i registri
        PUSH    CX
        PUSH    DX
        MOV     BX, OFFSET RISULT   ; stampa a video
        CALL    BIN2ASC             ; del risultato
        WRITELN M_RIS
        POP     DX                  ; ripristina i registri
        POP     CX
        POP     BX
        MOV     DX, BX              ; aggiorna i termini della successione
        JMP     LOOP                ; cicla il tutto fino a overflow
```

```
EXIT:   MOV     AH, 4CH             ; esce dal programma
        INT     21H

OFLOWR: WRITELN OFLOW               ; stampa a video del messaggio di errore
        JMP     EXIT

MainSEG ENDS
        END     FIBO
```

Esercizio 16

Il programma che segue è una modifica del programma dell'Esercizio 15 per il caso di numeri a 8 bit.

```
        INCLUDE MACRO.MAC

STACK   SEGMENT STACK
        DW      50  DUP(0)
STACK   ENDS

DATA    SEGMENT PUBLIC  'DATA'
LF      EQU     0AH
CR      EQU     0DH
OFLOW   DB      LF,CR,'Overflow',LF,CR,'$'
M_RIS   DB      LF,CR,'Termine della successione: '
RISULT  DB      5   DUP(?)
DATA    ENDS

MainSEG SEGMENT
        ASSUME CS:MainSEG, DS:DATA, SS:STACK

EXTRN   BIN2ASC:FAR

FIBO    LABEL   FAR
        MOV     AX, DATA            ; mette l'indirizzo del
        MOV     DS, AX              ; segmento DATA in DS

        PUSH    AX                  ; salva i registri
        PUSH    BX
        MOV     AL,'0'
        MOV     RISULT,AL
        MOV     AL,'$'
        MOV     RISULT+1,AL
        WRITELN M_RIS               ; stampa a video 0
        WRITELN M_RIS
        POP     BX                  ; ripristino dei registri
        POP     AX

        MOV     CL, 0               ; muove i primi due
        MOV     DL, 1               ; termini della successione

LOOP:   MOV     BL, CL              ; salva il valore di CL
                                    ; fa avanzare la successione
        ADD     CL, DL              ; somma i due termini
        JC      OFLOWR              ; overflow???

        MOV     AH, 0               ; salvo i valori per la stampa
        MOV     AL, CL
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     BX, OFFSET RISULT   ; stampa a video del risultato
        CALL    BIN2ASC
        WRITELN M_RIS
        POP     DX                  ; ripristino i valori
        POP     CX
        POP     BX

        MOV     DL, BL              ; avanzamento della successione

        JMP     LOOP

EXIT:   MOV     AH, 4CH             ; uscita
        INT     21H

OFLOWR: WRITELN OFLOW               ; stampa errore overflow
        JMP     EXIT
```

```
MainSEG ENDS
        END     FIBO
```

La riscrittura dell'Esercizio 15 con una rappresentazione dei numeri su 32 e 64 bit non può essere fatta perché non si hanno le appropriate aritmetiche. Si potrebbe codificare un metodo, lungo, tedioso e di difficile comprensione, con il quale si può rappresentare un numero a 32 bit attraverso due registri a 16 bit. Una rappresentazione di questo tipo consente la somma di numeri di 32 bit (sommando prima i registri meno significativi e, successivamente, i registri più significativi con il riporto). Poichè però un registro a 16 bit rappresenta 2^{16} numeri (0...65535) il carry flag si attiverà con il trabocco del registro meno significativo. La conseguenza è che il numero ottenuto è in base 65536. La vera difficoltà è perciò quella di presentare a video un numero ricodificato in base 10. Di seguito i passi fondamentali dell'algoritmo di ricodifica:

- dividere la parola più significativa per 10;
- accodare da destra al resto ottenuto dalla divisione sopra la parola meno significativa;
- continuare la divisione e accodare il quoziente al precedente;
- ripetere fino a che i bit da accodare non sono finiti;
- prendere l'ultimo resto e conservarlo come prima cifra ASCII;
- ripartire con il quoziente e iterare il tutto fino a che il quoziente arriva ad annullarsi.

Esercizio 17

L'esercizio verrà svolto presentando non solo la routine richiesta, ma anche la parte di programma che introduce da tastiera le dimensioni della matrice e gli elementi costituenti la stessa. A tal fine il programma procede attraverso i seguenti passi:

a) Richiede e legge da tastiera il numero di righe *m*
b) Richiede e legge da tastiera il numero di colonne *n*
c) Richiede e legge da tastiera i valori dei singoli elementi. Presenta a video la matrice introdotta.
d) Richiede e legge gli indici delle due righe da sommare.
e) Effettua la somma, con le modalità richieste dal testo, e presenta a video il risultato.

La matrice viene definita in memoria come vettore `M DW 50 DUP(0)`. Con questa assunzione si limitano le dimensioni della matrice a 50 elementi. La Figura 14.11 mostra come viene interpretato il vettore `M`. L'inserimento dei valori avviene per righe.

Con riferimento alla Figura 14.11 l'elemento $M(r, c)$ ha uno scostamento rispetto alla prima posizione di `M` pari a:

$$(r-1) \cdot n + (c-1)$$

dove:

r: riga dell'elemento che di vuole trovare
c: colonna dell'elemento che di vuole trovare
n: numero delle colonne della matrice

Per esempio l'elemento $M(2, 3)$ in una matrice 3×3, cioè il terzo elemento della seconda riga, nel vettore `M` è alla posizione assoluta: $(2-1) \cdot 3 + (3-1) = 5$. Per effettuare agevolmente questo calcolo è stata creata la macro `LOCATE` che prende in ingresso r e c e restituisce il valore assoluto della posizione sul vettore `M`, mettendolo nel registro `AX`.

I passi dell'algoritmo che somma la riga i con la riga j sono i seguenti, la routine nel codice è chiamata `SOMMAR`:

1. Si calcola la posizione assoluta su `M` della riga j e si assegna a `SI`
2. Si deposita il valore puntato da `SI` in `DX`
3. Si calcola la posizione assoluta su `M` della riga i e si assegna a `SI`
4. Si deposita il valore puntato da `SI` in `AX`
5. Si esegue la somma di `AX` e `DX`
6. Si deposita il risultato nella locazione di `M` puntata da `SI`.
7. Si itera in base alle dimensioni della matrice, sommando 1 ad i e j.

Il controllo di questo ciclo è affidato ad un contatore, il registro `CX`. Il ciclo termina quando `CX` è uguale a N, ovvero al numero delle colonne.

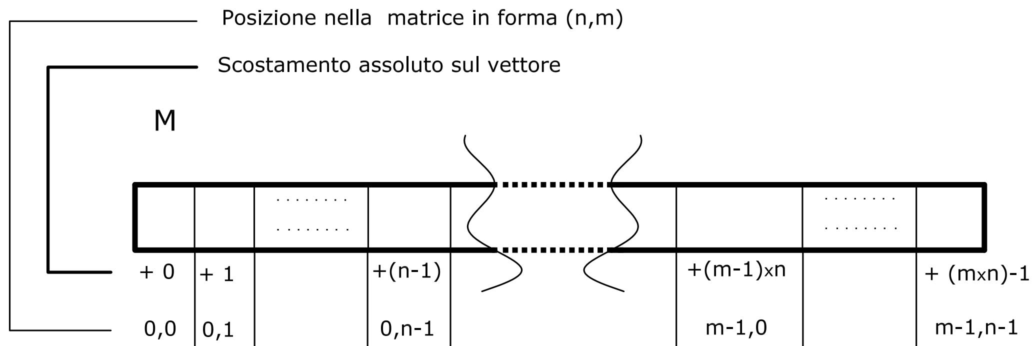

Figura 14.11 Interpretazione del vettore M.

Nel seguito si illustra il codice in modo da attenersi strettamente ai codici di riferimento presentati sul testo, senza eseguire alcun tipo di ottimizzazione o strutturazione dello stesso ad esclusione della routine `INPUT` introdotta nell' Esercizio 14.

```
;Definizioni di macro
;--------------------------------------

Include Macro.Mac

;la macro locate prende i valori dalla memoria, I,J per i valori totali riga e colonna
;in SI e DI rispettivamente gli indici di RIGA e COLONNA
;le righe e le colonne iniziano da 0,0 a (i-1),(j-1)
;esempio l'elemento alla posizione (r,c), si ottiene come: r*(J-1) + (c-1)

LOCATE  MACRO   SI,DI
        PUSH    SI
        PUSH    DI
        PUSH    BX
        MOV     BX,OFFSET J
        MOV     AX,[BX]
        DEC SI
        MUL SI
        DEC DI
        ADD AX,DI       ;IN AX C'E' IL VALORE VOLUTO
        POP BX      ;BX VIENE RIPRISTINATO
        POP DI
        POP SI
    ENDM

;Segmento Dati e Stack
;------------------------------------------------------------------
STACK           SEGMENT STACK
   DB   100 DUP(?)

STACK       ENDS

;--------------------------------------------------------------
```

```
DATA    SEGMENT PUBLIC  'DATA'

CR      EQU     0DH
LF      EQU     0AH
I       DW      0
I_STR   DB      LF,CR,'Le righe sono:  ','$'
J       DW      0
J_STR   DB      LF,CR,'Le colonne sono:  ','$'
DIM     DW      2 DUP(?)
DIM_STR DB      LF,CR,'La dimensione della matrice vale: ','$'
X_ASS   DW      0
Y_ASS   DW      0
TESTO0  DB      'Inserisci la matrice,  riga per riga, separando gli elementi con Enter',LF,CR,'$'
TESTO2  DB      LF,CR,LF,CR,'INSERIMENTO ELEMENTI MATRICE',LF,CR,'$'
TESTO3  DB      LF,CR,'INSERIMENTO NUMERO RIGHE','$'
TESTO4  DB      LF,CR,'INSERIMENTO NUMERO COLONNE','$'
TESTO5  DB      LF,CR,'La somma è stata eseguita, ecco la matrice: ',LF,CR,'$'
TESTO6  DB      LF,CR,LF,CR,'SCELTA PRIMA RIGA DELLA SOMMA(ovvero locazione del risultato)',LF,CR,'$'
TESTO7  DB      LF,CR,'SCELTA SECONDA RIGA DELLA SOMMA',LF,CR,'$'
X       DW      0
Y       DW      0
M_STR   DB      LF,CR,'Il risultato vale:  '
M_RES   DB      2 DUP(?)
RES     DB      ?
RISULT  DB      20 DUP(?)
M       DW      50 DUP(0)

DATA        ENDS
;---------------------------------------------------------------

CSec    SEGMENT

ASSUME CS:CSec,DS:Data,SS:Stack

    EXTRN INPUT:    FAR
    EXTRN BIN2ASC:  FAR

MATRICE PROC    FAR

        MOV     AX,Data
        MOV     DS,AX       ;DS -> Data

        WRITELN TESTO0      ;stampa del messaggio di info


;chiede le dimensioni preventive della matrice, prima le righe, poi le colonne

        WRITELN TESTO3      ;Lettura numero di righe
        CALL    INPUT
        MOV     I,AX        ;I è il numero di righe

        WRITELN TESTO4      ;Lettura numero di colonne
        CALL    INPUT
        MOV     J,AX

        MOV     CX,I
        MUL     CX          ;trova le dimensioni della matrice e la mette dentro AX

        MOV     AH,0
        MOV     DIM,AX

;STAMPA A VIDEO DELLE DIMENSIONI. RIGHE I, COLONNE J E DIMENSIONE.

        MOV     AX,I
        MOV     BX,OFFSET RES
        CALL    BIN2ASC
        WRITELN I_STR
        WRITELN RES

        MOV     AX,J
        MOV     BX,OFFSET RES
        CALL    BIN2ASC
        WRITELN J_STR
        WRITELN RES

        MOV     AX,DIM
        MOV     BX,OFFSET RES
        CALL    BIN2ASC
```

```
        WRITELN DIM_STR
        WRITELN RES

;Inserimento valori matrice:

;In AX e dentro lo stack in posizione TOP c'è la dimensione della matrice

        MOV     BX,OFFSET M     ;fa puntare SI all'inizio di M
        MOV     SI,0
        MOV     DX,0
        PUSH    BX
        PUSH    SI
        WRITELN TESTO2

Elemento: CALL  INPUT

        POP     SI
        POP     BX
        MOV     AH,0
        MOV     [BX],AL         ;inserisce il valore
        INC     SI              ;punta alla posizione successiva
        ADD     BX,1
        PUSH    BX
        PUSH    SI
        CMP     SI,DIM          ;Inseriti tutti i valori della matrice? SI -> continua

        JNE     Elemento        ;altrimenti andiamo a inserire il numero successivo

        CALL    SOMMAR          ;chiamata della funzione di richiesta e somma delle due righe

;STAMPA A VIDEO DEL RISULTATO DELLA SOMMA

Stampa:
        MOV     BX,OFFSET M    ;fa puntare SI all'inizio di M
        MOV     SI,0
        PUSH    BX
        PUSH    SI

LOOP:   MOV     AH,0                ;azzera il valore di AX
        MOV     AL,[BX]
        MOV     BX,OFFSET M_RES     ;utilizza la funzione BIN2ASC per presentare
        CALL    BIN2ASC             ;a schermo il valore.
        WRITELN M_STR
        POP     SI
        POP     BX
        INC     BX
        INC     SI
        PUSH    BX
        PUSH    SI
        CMP SI,DIM                  ;stampa tutti gli elementi
        JNE LOOP

USCITA: MOV     AH,4CH
        INT     21H

MATRICE ENDP

;***************************************************
;INIZIO ROUTINE DI RICHIESTA E SOMMA DELLE 2 RIGHE

SOMMAR PROC     NEAR

        WRITELN TESTO6          ;richiesta prima riga
        CALL    INPUT
        MOV     AH,0
        MOV     X,AX            ;X è la prima riga

        WRITELN TESTO7          ;richiesta seconda riga
        CALL    INPUT
        MOV     AH,0
        MOV     Y,AX            ;Y è al seconda riga da sommare

        MOV     SI,X
        MOV     DI,1
        LOCATE  SI,DI           ;Viene chiamata la macro locate
        MOV     X_ASS,AX        ;sulla prima riga da sommare

        MOV     SI,Y            ;Viene chiamata la macro locate
```

```
        MOV     DI,1            ;sulla seconda riga da sommare
        LOCATE  SI,DI
        MOV     Y_ASS,AX

        MOV     DI,0            ;DI è il contatore dei passi

Somma:
        MOV     BX,OFFSET M
        ADD     BX,Y_ASS        ;somma a BX lo scostamento assoluto della seconda riga
        MOV     DH,0
        MOV     DL,[BX]         ;mette il valore della seconda riga in DX

        MOV     BX,OFFSET M
        ADD     BX,X_ASS        ;somma a Bx la posizione assoluta della prima riga
        MOV     AH,0
        MOV     AL,[BX]         ;mette il valore della prima riga in AX
        ADD     AL,DL           ;esegue la somma tra i due valori
        MOV     [BX],AL         ;sostituisce il precedente valore nella prima riga

        ADD     X_ASS,1         ;incrementa tutti gli indici e esegue il controllo
        ADD     Y_ASS,1         ;per vedere se siamo arrivati al numero
        INC     DI              ;di colonne (ovvero la dimensione delle righe)
        CMP     DI,J
        JNE     SOMMA

        WRITELN TESTO5

        RET

SOMMAR ENDP

;FINE ROUTINE
;*********************************************************

CSec    ENDS
    END     MATRICE
```

Esercizio 18

Per la soluzione di questo esercizio si deve necessariamente far riferimento all' Esercizio 17 del quale viene mantenuto tutto il corpo del programma e vengono effettuate solo due modifiche:

1. Definizione nel segmento Dati del vettore R, che con una matrice composta al massimo da 50 elementi, deve essere almeno 25 (questo per una matrice di dimensioni 2x25).
2. Sostituzione della routine `SOMMAR` con la routine `CREAV` e della relativa chiamata.

La routine `CREAV` inizializza `SI` al contatore per del numero di righe, ovvero gli elementi di R, `BX` alla prima posizione della matrice M, infine `DI` alla prima posizione di R. Sfruttando la memorizzazione di M, come in Figura 14.11, si può notare che la creazione del vettore R, consiste nell'eseguire la somma ogni J elementi (con J numero di colonne) e salvarne il risultato in R per I volte (con I numero di righe).

Alla fine si stampa a video il vettore `R`.

```
;************************************
;INIZIO ROUTINE CREAZIONE VETTORE R(n)

CREAV PROC  NEAR

        MOV     SI,0                    ;inizializza SI come contatore delle righe
        MOV     BX,OFFSET M
        MOV     DI,OFFSET R

INIZ:   MOV     CX,J                    ;CX è il numero di colonne
        MOV     AX,0                    ;AX l'accumulatore della somma

SOMM1:  MOV     DX,[BX]
        INC     BX
        ADD     AX,DX
        LOOP    SOMM1                   ;si somma tutta la riga

        MOV     [DI],AX                 ; si trasferisce il risultato in R
        INC     DI
        INC     SI
        CMP     SI,I                    ;si controlla SI, se è uguale al numero di righe si esce
```

```
        JE      EXIT
        JNE     INIZ

EXIT:   RET

CREAV ENDP

;FINE ROUTINE CREAZIONE VETTORE R(n)
;**************************************
```

Esercizio 19

Rispetto alla soluzione dell'esercizio 17 cambia la modalità di accesso agli elementi della matrice in quanto si tratta di sommare colonne con colonne, invece di righe con righe.
In conseguenza della rappresentazione in memoria di M, Figura 14.11, risulta variato solo il punto 7 dei passi della routine SOMMAR. Si riportano i passi della routine SOMMAC che effettua l'operazione richiesta. La matrice ha dimensioni M(r,c).

1. Si calcola la posizione assoluta su M della colonna j e si assegna a SI
2. Si deposita il valore puntato da SI in DX
3. Si calcola la posizione assoluta su M della colonna i e si assegna a SI
4. Si deposita il valore puntato da SI in AX
5. Si esegue la somma di AX e DX
6. Si deposita il risultato nella locazione di M puntata da SI.
7. Si itera in base alle dimensioni della matrice, sommando il numero di righe r, agli indici delle colonne.

Il controllo di questo ciclo è affidato ad un contatore, il registro CX, che interrompe il ciclo quando è uguale a c, ovvero al numero delle colonne.

Rispetto all' Esercizio 17 veniva richiesto che la somma fosse depositata nella seconda colonna, e per fare ciò è stato invertito l'ordine di estrazione dei valori. L'aggiornamento, come si può notare, procede per righe (r), non di una unità, in quanto il valore successivo di una colonna, si trova r-locazioni più avanti.

```
;**********************************************
;INIZIO ROUTINE RICHIESTA E SOMMA DI DUE COLONNE

SOMMAC PROC NEAR
        WRITELN TESTO6
        CALL    INPUT
        MOV     AH,0
        MOV     X,AX            ;X è la prima colonna


        WRITELN TESTO7
        CALL    INPUT
        MOV     AH,0
        MOV     Y,AX            ;Y è al seconda colonna da sommare

SOMMMA_COLONNE:
        MOV     SI,1
        MOV     DI,X
        LOCATE  SI,DI           ;carica la posizione assoluta della prima colonna
        MOV     X_ASS,AX

        MOV     SI,1
        MOV     DI,Y
        LOCATE  SI,DI           ;carica la posizione assoluta della seconda colonna
        MOV     Y_ASS,AX

        MOV     DI,0
        MOV     SI,J

Somma:  MOV     BX,OFFSET M
        ADD     BX,X_ASS        ;somma a BX lo scostamento assoluto della prima colonna
        MOV     DH,0
        MOV     DL,[BX]         ;mette il valore della prima colonna in DX

        MOV     BX,OFFSET MATRIX
        ADD     BX,Y_ASS
        MOV     AH,0
```

```
        MOV     AL,[BX]          ;mette il valore della seconda colonna in AX

        ADD     AL,DL            ;esegue la somma tra i due valori

        MOV     [BX],AL          ;sostituisce il precedente valore nella seconda colonna

        ADD     X_ASS,SI        ;incrementa il valore assoluto su MATRIX della prima
        ADD     Y_ASS,SI        ;e della seconda colonna del valore SI, che è il numero
        INC     DI              ;colonne della matrice
        CMP     DI,I
        JNE     SOMMA

        WRITELN TESTO5

        RET

SOMMAC  ENDP

;FINE ROUTINE RICHIESTA E SOMMA DI DUE COLONNE
;**********************************************
```

Esercizio 20

L'algoritmo è molto semplice. Per ogni carattere della stringa inserita dall'utente, viene eseguita una ricerca su Γ, per determinarne la posizione p. Questo valore viene quindi sommato alla chiave k e viene scritto nella stringa di uscita il valore alla posizione $p + k$ in Ω. Il valore $p + k$ è da intendersi modulo 27. In uscita viene presentata la stringa criptata.

```
;Definizioni di macro

Include Macro.mac

;Segmento Dati e Stack
;-----------------------------------------------------------------
STACK   SEGMENT STACK

     DB 10 DUP(0)

STACK       ENDS
;--------------------------------------------------------------

DATA        SEGMENT
CR          EQU     0DH
LF          EQU     0AH
TESTO       DB      'Immetti il testo da criptare > ','$'
GAMMA       DB      ' abcdefghijklmnopqrstuvwxyz',CR,LF,LF,'$'
OMEGA       DB      'gthe quickbsownfxjmpdvrlazy',CR,LF,LF,'$'
K           EQU     1
Buffer      EQU     Max
Max         DB      FINE_MESS-MESSAGGIO
n           DB      ?
MESSAGGIO   DB      80 DUP(?)
FINE_MESS   EQU     $
DATA        ENDS
;--------------------------------------------------------------

CODESec SEGMENT ASSUME CS:CODESec,DS:Data,SS:Stack

Main    PROC    FAR

        MOV     AX,Data
        MOV     DS,AX                ;DS -> Data

        WRITELN TESTO                ;Stampa a video di Mess
        READLN  Buffer               ;Lettura stringa

        MOV     CX,0
        MOV     CL,n                 ;E' stato introdotto
        CMP     CX,0                 ; almeno 1 car?
        JE      Fine                 ;no

        MOV     SI,OFFSET Buffer+1  ;muove SI all'inizio della stringa inserita
        MOV     DI,OFFSET MESSAGGIO ;muove DI all'inizio della stringa di arrivo
        MOV     AL,LF                ;Sposta LF in AL
        MOV     [DI],AL              ;Scrive in cima all'arrivo il valore LF
        INC     DI                   ;punta al successivo carattere
```

```
        MOV     AL,CR               ;idem a prima ---
        MOV     [DI],AL             ;---con CR---
        INC     DI                  ;-----

        CALL    CriptMe             ;chiama la funzione di ordinamento

        WRITELN MESSAGGIO           ;Stampa il messaggio criptato a video

Fine:   MOV     AH,4CH
        INT     21H

Main    ENDP

CriptMe PROC    NEAR
;  In ingresso a CriptME:
;  SI: offset della posizione del primo carattere introdotto;
;  DI: offset della prima posizione libera di MESSAGGIO
;  CX: numero di caratteri introdotti


Crip:   SUB     BX,BX           ;azzera il registro BX, che sarà
        INC     BX              ;l'indice di riferimento posizione (p)
        INC     SI
        MOV     DL,[SI]         ;carica in DL il valore puntato da SI

Comp:   CMP     DL,GAMMA[BX]    ;compara il valore della stringa
                                ;messa dall'utente con il
                                ;primo carattere dell' alfabeto
        JE      Prin            ;se viene trovato il carattere viene memorizzato
        INC     BX
        JMP     Comp            ; non esiste ciclo di uscita xche la lettera deve essere
                                ;in tutti i modi trovata dentro l'alfabeto

Print:  ADD     BX,K            ;Somma la chiava di criptatura a p.
        CMP     BX,27           ;Esegue un controllo sulla posizione
        JNBE    Sottrai         ;in quanto dobbiamo simulare una coda
        JNAE    Stampa          ;circolare

Sottrai: SUB    BX,27           ;riparte il conteggio dall'inizio,in quanto
                                ;la somma p+k è un numero maggiore a 27

Stampa:  MOV    AL,OMEGA[BX]
         MOV    [DI],AL         ;mette in coda al messaggio di uscita il carattere criptato
         INC     DI             ;passa in uscita al carattere successivo
         LOOP    Crip

Exit:   MOV     AL,'$'          ;Aggiunta del '$'
        MOV     [DI],AL         ;- a fine stringa
        MOV     AL,0            ; OK!RET
        RET

CriptMe ENDP

CODESec ENDS
    END     Main
```

Esercizio 21

Di seguito si mostra la routine che calcola la successione (si veda l'Esercizio 22 per il programma chiamante). Il test della parità del termine contenuto in AX viene effettuato con una divisione per 2; se il resto (contenuto in DX) è diverso da zero il numero è dispari. La routine si ferma quando si raggiunge il numero 1. A tale scopo viene controllato che il quoziente della divisione sia pari a 0; se risulta essere 0 allora la routine esce rendendo il controllo al programma chiamante.

```
; In ingresso: ;   AX: contiene il numero 'k' ;   BX: è l'offset
di memoria dell'area dove costruire la successione ;   CX:
contiene il numero di iterazioni compiute dalla routine


SubSEG  SEGMENT PUBLIC
    ASSUME CS:SubSEG

PUBLIC  SUCC
```

```
SUCC    PROC    FAR

        MOV     CX, 0        ; azzera il contatore delle iterazioni

START:  MOV     DX, 0        ; azzera il registro DX
        INC     CX           ; CX conta le iterazioni eseguite

        PUSH    AX           ; salvataggio di AX

        MOV     [BX], AX    ; primo termine della successione in memoria

        ADD     BX, 2       ; aumenta di 2 l'indice BX
        PUSH    BX          ; salva BX, dopo serve alla divisione
        MOV     BX, 2       ; carica il divisore
        DIV     BX          ; procede alla divisione
        POP     BX          ; ripristina BX

        CMP     AX, 0       ; verifica se la successione
        JE      FINE        ; è arrivata alla fine

        CMP     DX, 0       ; verifica se il termine
        JE      PARI        ; corrente è pari
        JNE     DISP        ; o dispari

PARI:   POP     DX          ; svuota lo stack dal valore obsoleto di AX
        JMP     START       ; torna su START

DISP:   POP     AX          ; ripristino del valore di AX
        PUSH    BX          ; salvataggio di BX per la divisione
        MOV     BX, 3       ; se è dispari si moltiplica per 3
        MUL     BX
        POP     BX          ; ripristino del valore di BX
        INC     AX          ; aggiunge 1, poi si torna a START per
        JMP     START       ; la memorizzazione

FINE:   POP     DX          ; svuota lo stack dal valore obsoleto di AX
        RET                 ; rende il controllo al chiamante

SUCC    ENDP
    SubSEG  ENDS
    END
```

Esercizio 22

Di seguito il programma chiamante la routine SUCC che chiede il numero *k* da tastiera e stampa a video la successione dei numeri trovati.

```
    INCLUDE MACRO.MAC

STACK   SEGMENT STACK
    DW  50  DUP(0)
STACK   ENDS

DATA    SEGMENT PUBLIC  'DATA'

LF      EQU     0AH
CR      EQU     0DH
OFLOW   DB      LF,CR,'Overflow',LF,CR,'$'
M_RIS   DB      LF,CR,'Termine della successione: '
VIDEO   DB      5   DUP(?)
RISULT  DW      500 DUP(?)

DATA    ENDS

MainSEG SEGMENT
    ASSUME CS:MainSEG, DS:DATA, SS:STACK

    EXTRN   BIN2ASC:FAR
    EXTRN   INPUT:  FAR
    EXTRN   SUCC:   FAR

MAIN    LABEL   FAR
        MOV     AX, DATA            ; mette l'indirizzo del segmento
        MOV     DS, AX              ; DATA in DS
```

```
        CALL    INPUT               ; in AX c'è il carattere da
                                    ; tastiera 'k'

        MOV     BX,OFFSET RISULT    ; mette in BX l'offset dell'area di memoria
        CALL    SUCC                ; chiama la routine della successione

        MOV SI, 0                   ; inizializza i registri
        MOV DX, 0                   ; per la stampa a video

LOOP:   MOV AX, WORD PTR RISULT[SI]     ; carica il numero da stampare in AX

        PUSH    SI              ; salva i registri
        PUSH    CX              ; prima della stampa
        PUSH    DX

        MOV     BX, OFFSET VIDEO    ; codifica  ASCII
        CALL    BIN2ASC             ; del risultato e
        WRITELN M_RIS               ; stampa a video

        POP     DX                  ; ripristino dei registri
        POP     CX
        POP     SI

        ADD     SI, 2                ; incremento dell'indice
        INC     DX                   ; incremento del contatore della stampa
        CMP     DX, CX               ; test di fine iterazione: si devono stampare
                                     ; tanti elementi quanti sono quelli calcolati
        JE      EXIT
        JNE     LOOP

EXIT:   MOV     AH, 4CH              ; esce dal programma
        INT     21H

OFLOWR: WRITELN OFLOW           ; stampa il messaggio di
        JMP     EXIT            ; errore e esce

MainSEG ENDS
    END MAIN
```

Esercizio 23

Per la soluzione di questo esercizio è stato utilizzato come compilatore il Microsoft Visual C++, quindi l'assembler utilizzato è obbligatoriamente IA32. Le differenze, in questo preciso caso, si limitano a:

- il nome dei registri, es: `BX->EBX`;
- alcune istruzioni come la `IMUL` (vedi Esercizio 9.2) hanno una diversa modalità d'uso, mantenendo invariato l'effetto.

Riportiamo di seguito il listato C della routine e del programma chiamante, secondo le specifiche dell'Esercizio, dove però la funzione `succ` restituisce un intero al posto di void; questo per permettere al chiamante una più semplice gestione della stampa a video dei valori.

```
//file main.c

#include <stdio.h>

int succ{int x, int v[]};

void main(void)
{
     int k;
     int i;
     int numero_elementi;
     int v[100];

     printf("Battere il numero k : ");
     scanf("%d", &k );

     numero_elementi = succ(k, v);
```

```
    for(i=0; i<numero_elementi; i++)
        printf("\nEcco il risultato : %d", v[i]);

    printf("\nFatti %d cicli.\n",i);

}


/////////////////////////////////////////////////////////////

//file succ.c

int succ(int k, int v[])
{
    int temp;
    int i;
    i=0;

    v[0] = k;

    while (k!=1)
    {
    temp= k%2;  //resto divisione intera (modulo 2)

    if(temp==0)
         {
              k=k/2;
              v[i]=k;
         }
    else
         {
              k= (3*k) +1;
              v[i]=k;
         }
         i++;
    }

    return  i;
}
```

Viene ora presentata la routine in assembler IA32 che corrisponde al codice in formato C del file `succ.c`. Semplicemente basta sostituire tutto il contenuto del file `succ.c` con il seguente codice e procedere con la compilazione. Si noti l'uso dello stack come nell'Esercizio 9.14. Le variabili interne della funzione vengono salvate nel verso degli indirizzi decrescenti rispetto alla posizione puntata da EBP, mentre l'indirizzo di ritorno e i parametri si trovano nel verso degli indirizzi crescenti. Da ciò derivano i seguenti scostamenti rispetto a EBP (in esadecimale):

C	=	*v (indirizzo di v)
8	=	k
4	=	Ind. ritorno
0	=	EBP0
-4	=	temp
-8	=	i

L'header della funzione è rimasto uguale a quello che si sarebbe usato in caso di puro codice C, in quanto il Visual Studio non consente la chiamata diretta a funzioni scritte in assembler. Richiede invece che il codice assembler sia scritto internamente alla funzione C, ma la chiamata della stessa

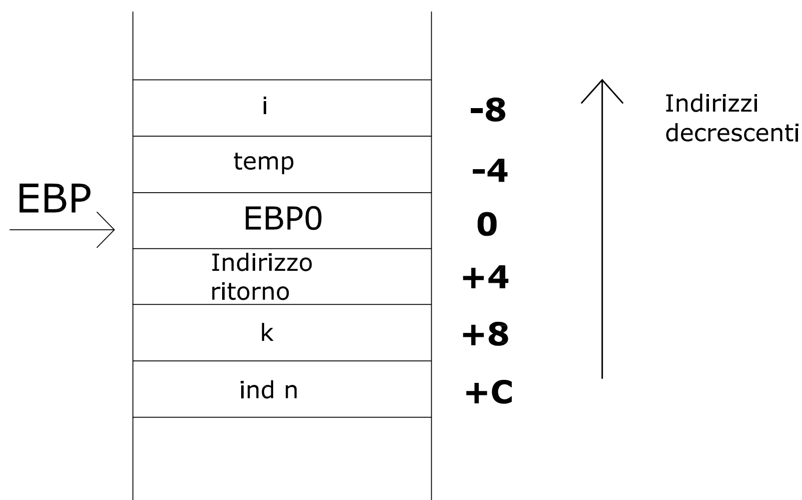

Figura 14.12 Memorizzazione delle variabili dell'Esercizio 23 sullo stack all'atto della chiamata della routine `succ`.

debba essere gestita dal linker come di consueto. Il codice assembler si inserisce nel modo seguente: `__asm{... codice ASM ...}`.

Per ulteriori chiarimenti si invita a visionare le direttive della Microsoft riguardo a questo problema all'indirizzo:
`http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclang/html/_core_writing_functions_with_inline_assembly.asp`

Infine, per semplicità di lettura, è stato inserito prima delle istruzioni assembler il codice C corrispondente.

```
//FILE succ.c (con codice interno in assembler)

int succ(int k, int v[])
{
__asm {

;INIZIO del codice assembler

;5:         int temp;
;6:         int i;
;7:         i=0;
             mov         dword ptr [ebp-8],0            ;i è in posizione -8

;8:
;9:         v[0] = k;
             mov         eax,dword ptr [ebp+0Ch]
             mov         ecx,dword ptr [ebp+8]
             mov         dword ptr [eax],ecx            ;k è in posizione +8


;11:        while (k!=1)
    ciclo:   cmp         dword ptr [ebp+8],1
             je          esci

;12:        {
;13:        temp= k%2;
             mov         edx,dword ptr [ebp+8]
             and         edx,80000001h
             jns         modulo
             dec         edx
             or          edx,0FEh
             inc         edx
    modulo:  mov         dword ptr [ebp-4],edx          ;temp è in posizione -4


;15:        if(temp==0)
             cmp         dword ptr [ebp-4],0
             jne         dispari
```

```
;16:          {
;17:              k=k/2;
            mov        eax,dword ptr [ebp+8]
            cdq
            sub        eax,edx
            sar        eax,1
            mov        dword ptr [ebp+8],eax

;18:              v[i]=k;
            mov        eax,dword ptr [ebp-8]
            mov        ecx,dword ptr [ebp+0Ch]
            mov        edx,dword ptr [ebp+8]
            mov        dword ptr [ecx+eax*4],edx

;19:          }
;20:        else
            jmp        aggiorna

;21:          {
;22:              k= (3*k) +1;
    dispari: mov        eax,dword ptr [ebp+8]
            imul       eax,eax,3
            add        eax,1
            mov        dword ptr [ebp+8],eax

;23:              v[i]=k;
            mov        ecx,dword ptr [ebp-8]
            mov        edx,dword ptr [ebp+0Ch]
            mov        eax,dword ptr [ebp+8]
            mov        dword ptr [edx+ecx*4],eax

;24:          }
;25:          i++;
    aggiorna: mov        ecx,dword ptr [ebp-8]
            add        ecx,1
            mov        dword ptr [ebp-8],ecx

;26:        }
            jmp        ciclo


;28:        return  i;
    esci:   mov        eax,dword ptr [ebp-8]

;29:   }
            pop        edi
            pop        esi
            pop        ebx
            mov        esp,ebp
            pop        ebp
            ret

    } //chiusura codice assembler

}   //ritorno al programma chiamante
```

Si noti che con la traduzione data dallo statement 29 ("}"), il ritorno viene effettuato direttamente dal codice assembler e che quindi il controllo non passa mai dal ritorno del C al programma chiamante (ultimo "}"), per il quale il compilatore genera il suo codice.

Esercizio 24

Mostriamo prima il caso del passaggio del parametro tramite stack. Il programma principale è quello sotto riportato. Esso non merita particolari commenti, se non l'affermazione che esso impiega la routine `INPUT`, tradotta all'Esercizio 14, per la lettura del valore di cui di vuole il fattoriale.

```
    INCLUDE MACRO.MAC

STACK   SEGMENT STACK
        DB      500 DUP(0)
STACK   ENDS

DATA    SEGMENT PUBLIC  'DATA'
LF      EQU     0AH
CR      EQU     0DH
```

```
M_RIS   DB      LF,CR,'Ecco il fattoriale : '
RISULT  DB      5   DUP(?)
DATA    ENDS

MainSEG SEGMENT
        ASSUME CS:MainSEG, DS:DATA, SS:STACK

EXTRN   BIN2ASC: FAR
EXTRN   INPUT: FAR
EXTRN   FATT: FAR

FATTO   LABEL   FAR
        MOV     AX, DATA            ; mette l'indirizzo del segmento
        MOV     DS, AX              ; DATA in DS

        CALL    INPUT               ; chiede un carattere da tastiera
                                    ; in AX il carattere inserito

        PUSH    AX
        CALL    FATT                ; lancia la routine del fattoriale
        ADD     SP,2                ; ripristina SP alla posizione precedente alla chiamata

        MOV     BX, OFFSET RISULT   ; stampa del risultato
        CALL    BIN2ASC
        WRITELN M_RIS

EXIT:   MOV     AH, 4CH             ; esce dal programma
        INT     21H


MainSEG ENDS
        END FATTO
```

La routine `FATT` sotto riportata fa uso dello stack per trattare la ricorsività. Per verificare la fine della discesa ricorsiva si effettua un controllo sul valore del parametro. Quando quest'ultimo è arrivato ad 1 si può risalire moltiplicando la successione di termini precedentemente memorizzati sullo stack.

```
; In ingresso:
;   AX: numero per cui si richiede il fattoriale
; In uscita:
;   AX: risultato

SubSEG  SEGMENT PUBLIC
        ASSUME CS:SubSEG

PUBLIC  FATT

FATT    PROC    FAR
        MOV     BP, SP              ; recupero del valore di
        ADD     BP, 4               ; AX nello stack
        MOV     AX, [BP]
        CMP     AX, 1               ; fine della discesa ricorsiva?
        JG      RECURS              ; no
        JMP     FINE                ; si

RECURS: DEC     AX                  ; diminuzione del valore
        PUSH    AX
        CALL    FATT                ; chiamata ricorsiva

        MOV     BP, SP              ; recupero del precedente valore
        ADD     BP, 6               ; la moltiplicazione avviene
        MOV     BX, [BP]            ; sulla risalita
        IMUL    BX                  ; moltiplicazione dei termini
        ADD     SP, 2               ; aggiornamento di SP

FINE:   RET

FATT    ENDP
SubSEG  ENDS
        END
```

Per quanto si riferisce al passaggio del parametro tramite AX, si può ancora usare il precedente programma chiamante, anche se lo statement `PUSH AX` e il corrispondente `ADD SP,2` sono del tutto superflui.

```
; In ingresso:
;   AX: numero per cui si richiede il fattoriale
```

```
; In uscita:
;   AX: risultato

SubSEG  SEGMENT PUBLIC
        ASSUME CS:SubSEG

PUBLIC  FATT

FATT    PROC    FAR
        CMP     AX, 1               ; verifica se AX è
        JNE     RECURS              ; alla fine della discesa
        RET                         ; ricorsiva

RECURS: PUSH    AX                  ; se non alla fine:
        DEC     AX                  ; mette AX nello stack
        CALL    FATT                ; decrementa AX e chiama FATT

        MOV     BP, SP              ; copia dallo stack il termine
        MOV     BX, [BP]            ; da moltiplicare
        IMUL    BX                  ; moltiplicazione dei termini
        ADD     SP, 2               ; aggiorna SP prima del RET
        RET                         ; rende il controllo al chiamante

FATT    ENDP
SubSEG  ENDS
        END
```