

**Michelangelo GROSSO, Paolo PRINETTO,
Maurizio REBAUDENGO, Matteo SONZA REORDA**

La programmazione in Assembler x86

2012

Indice

1. Introduzione	9
2. Architettura dei microprocessori Intel 8086 e 8088.....	11
2.1. Caratteristiche generali del processore 8086.....	11
2.2. L'8086 visto dal programmatore.....	13
2.3. Descrizione funzionale dell'8086	13
2.3.1. L'unità di interfaccia con il bus (BIU).....	14
2.3.2. L'unità di esecuzione (EU).....	15
2.3.3. Meccanismo di <i>pipeline</i>	15
2.3.4. Gestione delle parole di memoria da parte della BIU.....	16
2.4. Gestione della memoria	17
2.4.1. Segmentazione della memoria	17
2.4.2. Generazione degli indirizzi	18
2.4.3. Utilizzo della memoria.....	19
2.4.4. Lo stack	20
2.5. I registri	21
2.5.1. L' <i>Instruction Pointer</i>	21
2.5.2. I registri di dato	22
2.5.3. I registri puntatore ed i registri indice	22
2.5.4. Il registro di stato	23
2.6. Il microprocessore 8088.....	24
3. Dal codice sorgente al codice macchina	27
3.1. Il codice sorgente	27
3.2. L'assemblatore	27
3.3. Il linker.....	28
3.4. Il loader	28
4. Introduzione al linguaggio Assembler	31
4.1. Esempio 1: Scrittura di un valore in memoria	31
4.2. Esempio 2: Somma di due valori	32
4.3. Esempio 3: Somma degli elementi di un vettore (I versione).....	32
4.4. Esempio 4: Somma degli elementi di un vettore (II versione).....	33
4.5. Esempio 5: Input/Output.....	34
4.6. Esempio 6: Ricerca del carattere minore in una stringa.....	36
4.7. Esempio 7: Procedura (I versione).....	36
4.8. Esempio 8: Procedura (II versione).....	37
4.9. Esempio 9: Calcolo di un polinomio	38
5. Formato di un programma in linguaggio Assembler	39
5.1. Regole per scrivere un programma in linguaggio Assembler	39
5.2. Commenti.....	40
5.3. Codice operativo	40
5.4. Operandi.....	41

5.5. Identificatori	41
5.6. Costanti numeriche	42
6. Direttive per l'assemblatore.....	45
6.1. Direttive per la definizione dei dati.....	45
6.1.1. Direttiva DUP	46
6.1.2. Puntatori a dati	47
6.1.3. Direttiva DB.....	48
6.1.4. Direttiva DW.....	49
6.1.5. Direttiva DD.....	51
6.1.6. Direttive DQ e DT	53
6.2. Operatori	54
6.2.1. Operatori ed attributi di una variabile	54
6.2.2. Operatori aritmetici	56
6.2.3. Operatore PTR	57
6.2.4. Direttiva LABEL	59
6.3. Direttive per le definizioni di costanti.....	60
6.3.1. La direttiva EQU	60
6.3.2. La direttiva =.....	61
6.4. Il <i>location counter</i>	62
6.5. Le macro.....	63
6.5.1. Variabili locali all'interno di una macro	65
6.6. Le direttive SEGMENT e ENDS	66
6.7. La direttiva GROUP	67
6.8. La direttiva ASSUME.....	68
6.9. La direttiva END	70
6.10. Restituzione del controllo al Sistema Operativo MS-DOS	70
6.11. Scheletro di un programma (I versione).....	71
6.12. Direttive semplificate per la gestione dei segmenti	71
6.12.1. I modelli di memoria	71
6.12.2. Creazione dello stack	73
6.12.3. Creazione del segmento di dato	73
6.12.4. Creazione del segmento di codice.....	74
6.13. Interfaccia con i sistemi operativi MS-DOS e compatibili.....	74
6.14. Scheletro di un programma (II versione)	75
6.15. Direttive per la definizione di identificatori globali esterni.....	76
6.16. Uso dell'emulatore emu8086	77
6.16.1. Direttive proprie di emu8086.....	77
6.16.2. Operatori non supportati	78
6.16.3. Operatore aritmetico di modulo	78
6.16.4. Abbreviazione degli operatori WORD PTR e BYTE PTR.....	78
6.16.5. Direttiva = per la definizione di costanti	79
6.16.6. Direttive non supportate: GROUP e ASSUME	79
6.16.7. Istruzioni per il controllo del processore non supportate	79
6.16.8. Esempio di conversione di programma da emu8086 a MASM	79
7. I modi di indirizzamento	81
7.1. <i>Register Addressing</i>	81
7.2. <i>Immediate Addressing</i>	82

7.3.	<i>Direct Addressing</i>	82
7.3.1.	<i>Segment Override</i>	84
7.4.	<i>Register Indirect Addressing</i>	84
7.5.	<i>Base Relative Addressing</i>	85
7.6.	<i>Direct Indexed Addressing</i>	86
7.7.	<i>Base Indexed Addressing</i>	87
8.	Le istruzioni di trasferimento dati	91
8.1.	L'istruzione MOV.....	91
8.2.	L'istruzione XCHG.....	92
8.3.	L'istruzione LEA.....	94
8.4.	L'istruzione XLAT.....	96
8.5.	Le istruzioni LDS e LES	98
8.6.	Le istruzioni PUSH e POP	98
8.7.	Le istruzioni PUSHA e POPA	99
8.8.	Le istruzioni PUSHF e POPF	100
8.9.	Le istruzioni SAHF e LAHF	100
8.10.	Le istruzioni IN e OUT.....	101
8.11.	Le istruzioni che modificano i flag	101
9.	Le istruzioni di controllo del flusso	103
9.1.	L'istruzione di confronto: CMP	103
9.2.	Le istruzioni di salto.....	104
9.2.1.	Le istruzioni di salto condizionato	105
9.2.2.	L'istruzione di salto incondizionato: JMP	107
9.2.3.	Codifica degli indirizzi in istruzioni di salto.....	108
9.3.	Le istruzioni che gestiscono una sequenza.....	112
9.3.1.	L'istruzione LOOP.....	112
9.3.2.	Le istruzioni LOOPE, LOOPZ, LOOPNE e LOOPNZ	113
9.4.	I costrutti per il controllo del flusso	115
9.4.1.	Costrutto <i>IF-THEN</i>	115
9.4.2.	Costrutto <i>IF-THEN-ELSE</i>	117
9.4.3.	Costrutto <i>CASE</i>	118
9.4.4.	Costrutto <i>REPEAT-UNTIL</i>	121
9.4.5.	Costrutto <i>FOR</i>	122
9.4.6.	Costrutto <i>WHILE</i>	125
10.	Le istruzioni aritmetiche	127
10.1.	Le istruzioni ADD e SUB	127
10.2.	L'istruzione CBW.....	128
10.3.	L'istruzione ADC	129
10.4.	L'istruzione SBB.....	130
10.5.	Le istruzioni INC e DEC	132
10.6.	L'istruzione NEG.....	133
10.7.	Le istruzioni MUL e IMUL	134
10.7.1.	Moltiplicazione per una costante	135
10.7.2.	Moltiplicazione tra dati di tipo diverso	136
10.8.	Le istruzioni DIV e IDIV.....	136
10.8.1.	L'istruzione CWD.....	141

10.9. I numeri BCD	141
10.9.1. Le istruzioni AAA e DAA	142
10.9.2. Le istruzioni AAS e DAS	145
10.9.3. L'istruzione AAM	148
10.9.4. L'istruzione AAD	149
11. Le istruzioni per la manipolazione dei bit	151
11.1. Le istruzioni logiche	151
11.1.1. L'istruzione AND	151
11.1.2. L'istruzione OR	153
11.1.3. L'istruzione XOR	154
11.1.4. L'istruzione NOT	155
11.1.5. L'istruzione TEST	156
11.2. Le istruzioni di scorrimento	158
11.2.1. Le istruzioni di shift	158
11.2.2. Le istruzioni di rotazione	161
12. Le istruzioni per la manipolazione di stringhe	165
12.1. Introduzione	165
12.1.1. Preparazione dei registri	165
12.1.2. La ripetizione delle istruzioni per la manipolazione di stringhe ...	167
12.1.3. Aggiornamento del contenuto dei registri indice	168
12.1.4. Riassunto delle operazioni necessarie per manipolare le stringhe	170
12.2. Copiatura di una stringa	170
12.3. Confronto tra stringhe	172
12.4. Scansione di una stringa	175
12.5. Inizializzazione di una stringa	177
12.6. Elaborazione di una stringa	178
12.7. Istruzioni con operandi	182
13. Le procedure	183
13.1. Le procedure	183
13.2. Definizione di una procedura	183
13.3. Chiamata di una procedura	184
13.4. Ritorno da una procedura	185
13.5. Salvataggio dei registri	186
13.6. Punto di ingresso e di uscita di una procedura	186
13.7. Passaggio di parametri	187
13.7.1. Uso di variabili globali	188
13.7.2. Uso di registri	189
13.7.3. Uso dello stack	190
13.8. Condizione di errore	196
13.9. Tabelle di jump	200
14. Le istruzioni per il controllo del processore	203
14.1. Le istruzioni per la gestione delle interruzioni	203
14.1.1. Le interruzioni hardware	203
14.1.2. Le interruzioni software	204
14.1.3. L'istruzione INT	204

14.1.4.	L'istruzione INTO.....	205
14.1.5.	L'istruzione IRET.....	205
14.2.	Le istruzioni per la sincronizzazione con l'esterno.....	205
14.2.1.	L'istruzione HLT.....	205
14.2.2.	L'istruzione WAIT.....	206
14.2.3.	L'istruzione ESC.....	206
14.2.4.	L'istruzione LOCK.....	206
14.3.	L'istruzione NOP.....	206
14.4.	Le funzioni di sistema del DOS.....	207
15.	Programmazione avanzata.....	209
15.1.	Procedure Assembler richiamabili da un programma C.....	209
15.1.1.	Regole per la procedura Assembler.....	209
15.1.2.	Regole per la procedura C chiamante.....	212
15.2.	Procedure C richiamabili da un programma Assembler.....	214
15.2.1.	Codice C di <i>startup</i>	214
15.2.2.	Programma principale in C.....	215
15.3.	Strutture dinamiche.....	215
15.3.1.	Allocazione e deallocazione.....	215
15.4.	La recursione.....	221
15.4.1.	Il passaggio di parametri.....	224
16.	Esercizi svolti.....	229
16.1.	Calcolo del numero di combinazioni semplici di elementi di un insieme.....	229
16.1.1.	Codice.....	229
16.2.	Riconoscimento degli anni bisestili.....	230
16.2.1.	Codice.....	230
16.3.	Generazione di segnali di <i>clock</i>	231
16.3.1.	Codice.....	232
16.4.	Calcolo dei prezzi scontati.....	233
16.4.1.	Codice.....	233
16.5.	Calcolo del valore di un insieme di monete.....	234
16.5.1.	Codice.....	234
16.6.	Media mobile.....	235
16.6.1.	Codice.....	235
16.7.	Identificazione di numeri primi.....	236
16.7.1.	Codice.....	237
16.8.	Gestione di un magazzino di tessuti.....	237
16.8.1.	Codice.....	238
16.9.	Filtro per indirizzi IP.....	239
16.9.1.	Codice.....	239
16.10.	Classificazione di caratteri.....	240
16.10.1.	Codice.....	241
16.11.	Allineamento di byte.....	242
16.11.1.	Codice.....	242
16.12.	Verifica della monotonia di una sequenza di interi.....	243
16.12.1.	Codice.....	243
16.13.	Rimozione di occorrenze multiple consecutive di caratteri.....	244
16.13.1.	Codice.....	245

16.14.	Conversione ASCII-binario	246
16.14.1.	Codice	246
16.15.	Conversione binario-ASCII	246
16.15.1.	Implementazione	246
16.15.2.	Codice	247
16.16.	Buffer circolare	247
16.16.1.	Implementazione	248
16.16.2.	Codice	249
16.17.	Ricerca di una sottomatrice	250
16.17.1.	Implementazione	251
16.17.2.	Codice	251
16.18.	Mappa geografica digitalizzata	253
16.18.1.	Implementazione	254
16.18.2.	Codice	255
16.19.	Ricerca degli anagrammi	256
16.19.1.	Soluzione proposta	256
16.19.2.	La procedura di ordinamento	256
16.19.3.	Codice	257
16.20.	Elaborazione di una matrice	259
16.20.1.	Soluzione proposta	259
16.20.2.	Codice	260
16.21.	Interpolazione di una funzione	263
16.21.1.	Soluzione proposta	263
16.21.2.	Codice	264
17.	Esercizi proposti	267
17.1.	Buffer FIFO	267
17.2.	Sostituzione dei caratteri di tabulazione	267
17.3.	Sottrazione tra insiemi	268
17.4.	Triangolo di Floyd	268
17.5.	Formattazione di una stringa	269
17.6.	Filtraggio di una sequenza	269
17.7.	Compressione di una stringa	269
17.8.	Compattamento di valori	270
17.9.	Trasmissione seriale	270
17.10.	Compattamento di un segnale	271
17.11.	Ricerca in un dizionario	271
17.12.	Analisi delle ore di entrata/uscita	272
17.13.	Analisi delle presenze	273
17.14.	Visita in ampiezza di un grafo	273
17.15.	Analisi di connettività di un grafo	273
17.16.	Calcolo del ciclo hamiltoniano di lunghezza minima	274
17.17.	Livellamento di un grafo	275
17.18.	Calcolo della massima <i>clique</i> in un grafo	275
17.19.	Calcolo della sottosequenza di costo massimo	276
17.20.	Torre di Hanoi	276
17.21.	Memorizzazione di una data in forma compatta	277
17.22.	Conversione da data a numero d'ordine del giorno	278
17.23.	Conversione da data a numero d'ordine della settimana	278

17.24.	Inserimento di un elemento in un albero binario di ricerca.....	278
17.25.	Shell sort	279
17.26.	Selection sort.....	279
17.27.	Analisi di una matrice sparsa	280
17.28.	Ordinamento delle colonne di una matrice	280
17.29.	Somma delle diagonali	281
17.30.	Prodotto di polinomi	281
17.31.	Moltiplicazione tra numeri in floating-point (I versione)	282
17.32.	Moltiplicazione tra numeri in floating-point (II versione)	282
17.33.	Determinazione della temperatura a regime di una piastra.....	282
17.34.	Prenotazione posti	283
17.35.	Gestione di un ospedale	283
17.36.	Sistema <i>client-server</i>	284
17.37.	Classifica di un gara automobilistica	284
17.38.	Sistema di avvistamento radar	285
17.39.	Gara ciclistica.....	286
17.40.	Istogramma orizzontale	286
17.41.	Analisi di un circuito.....	287
17.42.	Crittografia di un testo	288
17.43.	Elaborazione di un'immagine	289
17.44.	Correzione di un questionario	289
17.45.	Calcolo della media dei voti.....	290
18.	Bibliografia	291

1. Introduzione

Gli ultimi decenni hanno visto l'affermarsi di numerosi linguaggi di programmazione di alto livello quali ad esempio C, C++, Java. L'introduzione di tali linguaggi non ha tuttavia eliminato la necessità di conoscere ed utilizzare i linguaggi Assembler.

Le ragioni che ancora giustificano l'uso di un linguaggio Assembler sono così sintetizzabili:

- *efficienza*: in taluni casi esso permette la realizzazione di programmi eseguibili più efficienti (in termini di velocità di esecuzione o di memoria richiesta) di quelli ottenibili da un programma scritto in un linguaggio di alto livello; situazioni di questo tipo tendono a diventare sempre più rare, in quanto l'evoluzione dei compilatori fa sì che il codice eseguibile generato sia molto spesso più efficiente di quello scritto da un bravo programmatore Assembler. In ogni caso, l'uso del linguaggio Assembler viene limitato alle parti più critiche dal punto di vista computazionale.
- *accesso all'hardware*: i linguaggi di alto livello tendono a "virtualizzare" l'hardware "nascondendolo" al programmatore, soprattutto al fine di incrementare la portabilità dei programmi; questa "virtualizzazione" può rappresentare una limitazione allorquando sia necessario accedere a determinate componenti specifiche dell'architettura del sistema (ad esempio i registri del processore o delle periferiche).
- *reverse engineering*: a partire da un file eseguibile è possibile ricostruire il programma Assembler ad esso corrispondente, mentre è praticamente impossibile risalire in modo univoco ad un programma equivalente in un linguaggio di alto livello; ne consegue che quando si desidera modificare o anche semplicemente analizzare un file eseguibile di cui non si possiede il corrispondente programma sorgente, è necessario lavorare a livello di codice Assembler.

Infine, è evidente che la conoscenza di almeno un linguaggio Assembler costituisca una condizione necessaria per comprendere il funzionamento di un processore, e rappresenti quindi un obiettivo didattico imprescindibile all'interno di un curriculum di perito o ingegnere informatico.

Scopo di questo libro è la presentazione dettagliata del linguaggio Assembler del processore *Intel 8086*: la scelta di tale linguaggio è stata dettata principalmente dal fatto che numerosi *Personal Computer* utilizzano processori della famiglia x86, che sono compatibili a livello software con l'8086 ed utilizzano quindi un sovrainsieme del linguaggio Assembler qui presentato. Per questa ragione, benchè il processore 8086 sia ormai da tempo obsoleto, il relativo linguaggio Assembler, che qui viene presentato, è tuttora utilizzato e rappresenta un importante riferimento nel settore.




Il libro si rivolge principalmente a studenti universitari dei corsi di Laurea e presuppone alcune conoscenze di base in campo informatico. In particolare, si assume che il lettore già conosca la struttura generale di un elaboratore, le nozioni elementari relative alla rappresentazione binaria dei numeri e dell'algebra booleana e che abbia già una qualche esperienza di programmazione in linguaggi di alto livello. In particolare, il testo fa spesso riferimento al linguaggio C quale mezzo per illustrare le specifiche ed il funzionamento dei frammenti di codice Assembler presentati.

L'approccio seguito prevede una suddivisione della presentazione in 2 parti: nella prima (Capitoli 2, 3 e 4) si introduce l'architettura del processore, l'ambiente di sviluppo ed il linguaggio Assembler nelle sue linee generali; in questa fase, le caratteristiche del linguaggio vengono presentate partendo da una serie di esempi, e deducendo da questi le nozioni fondamentali, senza pretesa né di esaustività, né di completezza. Nella seconda parte, il linguaggio viene invece affrontato in maniera sistematica e dettagliata; anche in questa parte, vengono presentati un'abbondante quantità di esempi ed esercizi, in modo da rendere più concreti i concetti introdotti. Al termine, i Capitoli 15 e 16

presentano una ricca serie di esercizi riassuntivi, in parte risolti, in parte lasciati al lettore.

Molto del materiale presentato è stato originariamente sviluppato per il corso di *Calcolatori Elettronici* del Corso di Laurea in Ingegneria Informatica del Politecnico di Torino. Gli autori hanno posto ogni cura nel tentare di limitare il numero di errori nel testo, e saranno grati a coloro che vorranno loro segnalare eventuali inesattezze comunque presenti.

All'interno del testo è stata utilizzata una convenzione grafica per la descrizione del processore x86 e del linguaggio Assembler. Tale convenzione è così schematizzabile:

<i>Convenzione</i>	<i>Descrizione</i>
AX	Introduce una nuova parola chiave del linguaggio.
 <i>PUSH sorgente</i>	Introduce una regola sintattica per il linguaggio.
MOV AX, CX	Indica un comando del linguaggio, oppure le parole chiave del linguaggio.
XCHG AX, VAR	Indica un frammento di programma in linguaggio Assembler o C.
 MOV DATO2, DATO1	Indica un frammento di programma in linguaggio Assembler che presenta errori a livello di compilazione o di esecuzione.
 MOV BH, DATO1 MOV DATO2, BH	Indica una soluzione corretta ad un errore precedentemente presentato.

2. Architettura dei microprocessori Intel 8086 e 8088

Si descrive in questo capitolo l'architettura e il modello di funzionamento dei microprocessori Intel 8086 e 8088, i primi processori della famiglia x86, attraverso cui è possibile delineare le caratteristiche salienti dell'architettura e inizialmente sviluppata da Intel.

2.1. Caratteristiche generali del processore 8086

L'8086 è un microprocessore a 16 bit progettato da Intel alla fine degli anni '70 e realizzato tramite un circuito integrato contenente approssimativamente 29.000 transistor; fu prodotto in diverse versioni che differiscono per tecnologia di fabbricazione e caratteristiche elettriche, quali ad esempio la frequenza massima di clock ammessa.

L'8086 dispone di un numero complessivo di 40 piedini (*pin*) (Fig. 2.1): 20 di indirizzo (di cui 16 fungono anche da piedini di dato), 16 di controllo, 1 di alimentazione, 2 di massa e 1 di *clock*.

Quando il processore è montato su una scheda, i piedini sono connessi ad una serie di *bus* che lo collegano con il resto del sistema (memorie, periferiche, ecc.). Dal punto di vista logico il bus è suddiviso in *Bus Dati (D-bus)*, *Bus Indirizzi (A-bus)* e *Bus di Controllo (C-bus)*, in base al tipo di piedini a cui ciascuna parte di bus è connessa (Fig. 2.2).

L'8086 dispone di 16 piedini di dato, dunque esso può lavorare con parole costituite da 2 byte (*word*). Il microprocessore gestisce l'accesso al *bus* in modo tale da condividere 16 pin tra i segnali di dato ed i segnali di indirizzo (*time multiplexing*). Questo significa che gli stessi piedini vengono utilizzati in momenti diversi per trasmettere segnali di dato o di indirizzo. Per chiarire meglio il meccanismo utilizzato analizziamo come l'8086 si comporta quando deve leggere o scrivere un dato dalla/sulla memoria.

L'operazione viene svolta in un *ciclo di bus*; ogni ciclo di bus consiste di almeno 4 cicli di clock (la cui frequenza può essere di 5Mhz, 8Mhz o 10Mhz a seconda delle versioni), denominati rispettivamente T_1 , T_2 , T_3 e T_4 :

- in T_1 il processore dispone sull'A-bus i 20 bit di indirizzo del dato da leggere o scrivere in memoria
- in T_2 il processore predispone il D-bus alla lettura od alla scrittura, attivando rispettivamente i pin RD o WR
- durante le fasi T_3 e T_4 avviene il trasferimento dei dati tra processore e memoria, utilizzando il D-bus.

La tecnica del multiplexing, se da un lato permette un uso efficiente dei piedini del processore e l'impiego di contenitori standard a 40 pin di tipo *Dual In line Package (DIP)*, dall'altro provoca un rallentamento nelle operazioni di accesso al bus in quanto un dato ed un indirizzo non possono essere trasmessi contemporaneamente.

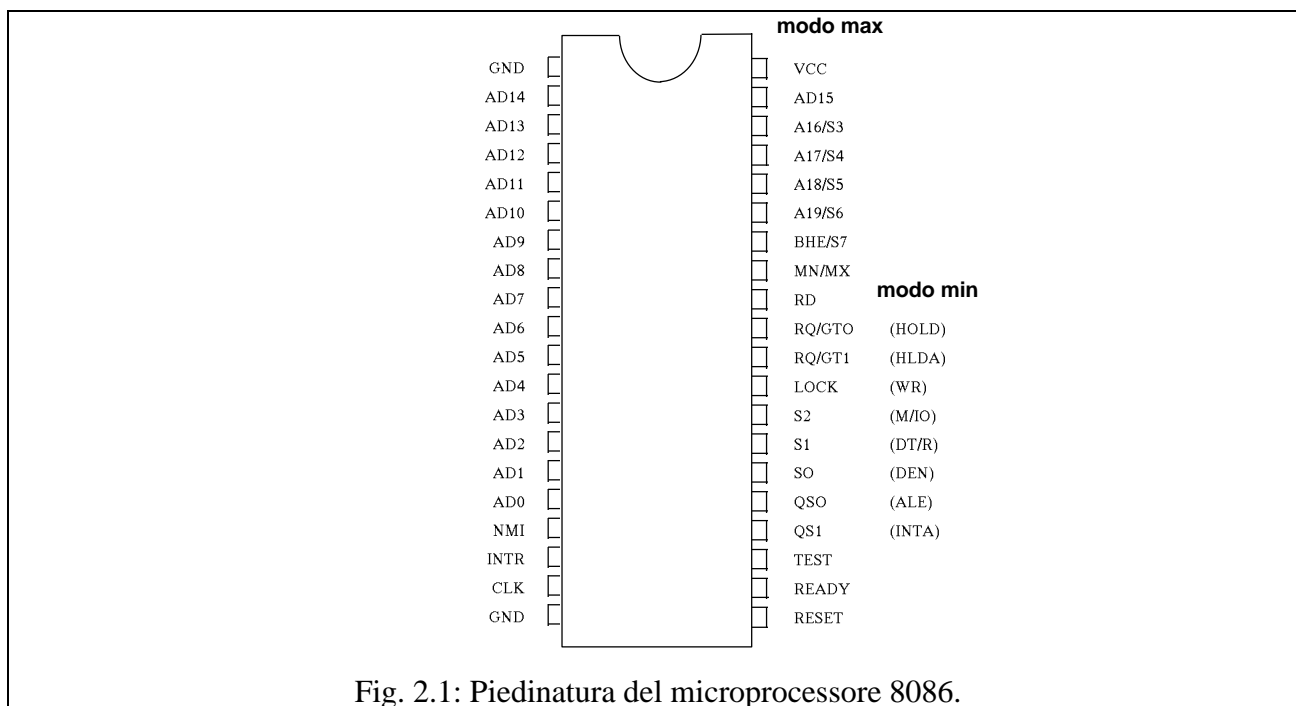


Fig. 2.1: Piedinatura del microprocessore 8086.

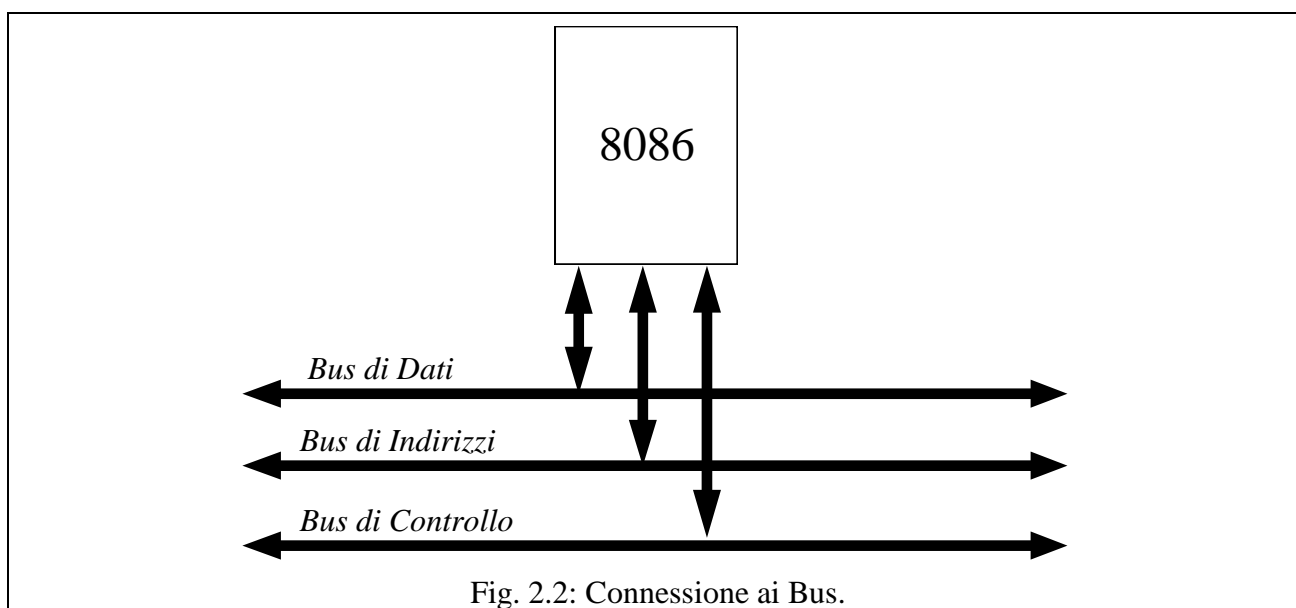


Fig. 2.2: Connessione ai Bus.

Le 16 linee di controllo gestiscono l'interfaccia tra la CPU ed i dispositivi esterni ad essi collegati: periferiche e memoria.

L'8086 può operare in due modi di funzionamento, a seconda del valore presente sul pin di controllo MIN/MAX:

- *minimum mode* (MIN/MAX = 0) adatto quando l'8086 è utilizzato all'interno di piccoli sistemi
- *maximum mode* (MIN/MAX = 1) per applicazioni di tipo multiprocessore.

La funzione di alcuni piedini cambia a seconda del modo operativo.

In Fig. 2.3 è rappresentato uno schema dell'architettura interna dell'8086. Nel seguito verranno descritte e analizzate le componenti principali.

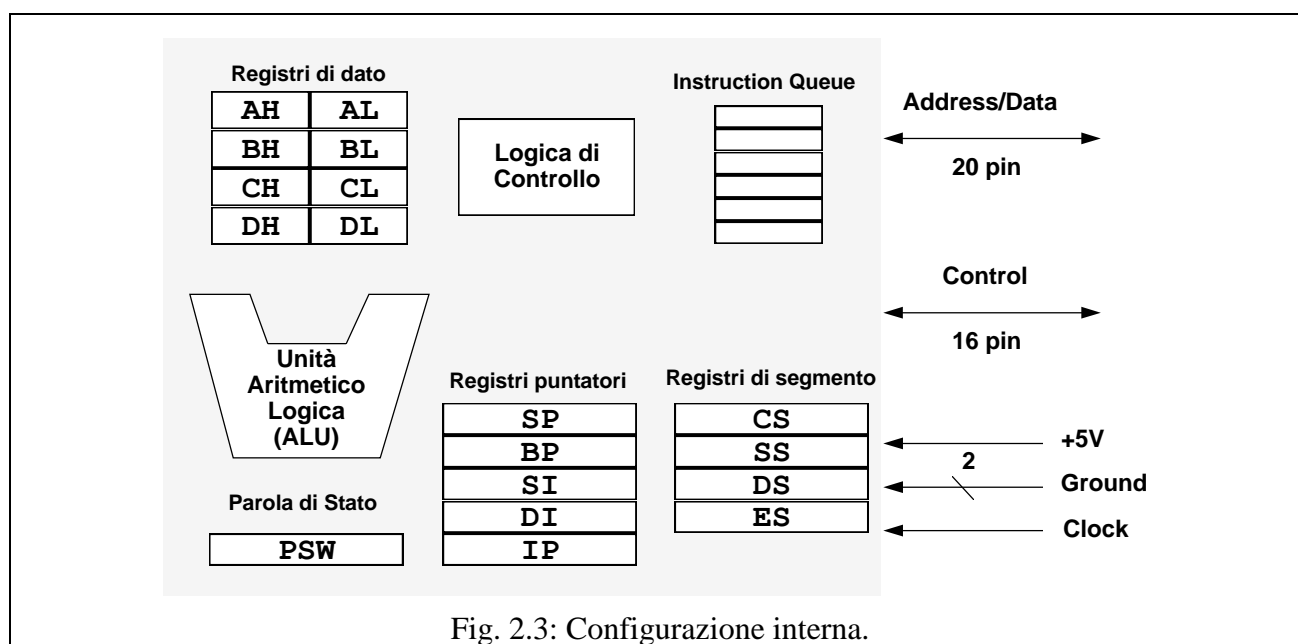
2.2. L'8086 visto dal programmatore

Per essere in grado di programmare un microprocessore non è necessario conoscere tutte le caratteristiche hardware dell'architettura: è sufficiente conoscerne il “modello software” che permette di capire struttura, connessioni e funzionalità di quelle parti del processore che sono direttamente accessibili attraverso un programma.

Per un programmatore è innanzitutto importante conoscere tutti i registri interni del microprocessore, la loro funzione nonché le capacità operative del sistema ed i limiti del medesimo. Inoltre è fondamentale conoscere come è organizzata la memoria esterna e come funziona il meccanismo di indirizzamento per la lettura delle istruzioni e l'*Input/Output* dei dati (I/O).

Il modello software dell'8086 (Fig. 2.3) include 12 registri interni di 16 bit ciascuno: 4 registri di dato (**AX**, **BX**, **CX** e **DX**), 2 registri puntatore (**BP** e **SP**), 2 registri indice (**SI** e **DI**) e 4 registri di segmento (**CS**, **DS**, **SS** e **ES**).

In aggiunta a questi, bisogna considerare un apposito registro denominato *Instruction Pointer* (**IP**), destinato a memorizzare l'indirizzo della successiva istruzione, ed un registro di stato *Processor Status Word* (**PSW**) contenente i flag di controllo e di stato del processore. Il modello di memoria infine prevede la possibilità di indirizzare sino ad 1 Mbyte di memoria esterna.



2.3. Descrizione funzionale dell'8086

Il processo di esecuzione delle istruzioni all'interno di un generico sistema di elaborazione è schematizzabile nei seguenti passi:

1. inizializzazione del *Program Counter*, cioè di quel registro in cui viene memorizzato l'indirizzo della prossima istruzione da eseguire;
2. lettura della prossima istruzione, indirizzata dal Program Counter (fase di ricerca o *fetch*);
3. decodifica dell'istruzione (fase di decodifica o *decode*);
4. esecuzione dell'operazione codificata nell'istruzione (fase di esecuzione o *execute*);
5. aggiornamento del Program Counter
6. ripetizione a partire dal passo 2.

Le fasi che caratterizzano l'esecuzione di un programma sono dunque principalmente quella di

fetch e quella di *execute*.

Questo meccanismo, tipicamente sequenziale, provoca ritardi nell'esecuzione dei programmi, in quanto il microprocessore, per eseguire un'istruzione, deve attendere che questa sia stata prelevata dalla memoria. L'8086 elimina gran parte di questo ritardo assegnando i due lavori a due unità separate interne al chip: l'unità di interfaccia al bus (*Bus Interface Unit*, **BIU**) e l'unità di Esecuzione (*Execution Unit*, **EU**), che operano concorrentemente.

Il meccanismo di funzionamento è schematizzato in Fig. 2.4. La BIU preleva le istruzioni dalla memoria e comunica con il mondo esterno provvedendo al trasferimento dei dati. La EU esegue le istruzioni prelevandole da una coda (*Instruction Queue*, **IQ**) caricata dalla BIU, ed eventualmente richiede alla BIU di effettuare cicli di bus per la lettura o la scrittura dei dati in memoria. Poiché le due unità sono indipendenti, la BIU può prelevare una nuova istruzione dalla memoria nello stesso momento in cui la EU esegue un'istruzione precedentemente prelevata dalla BIU e memorizzata nella IQ.

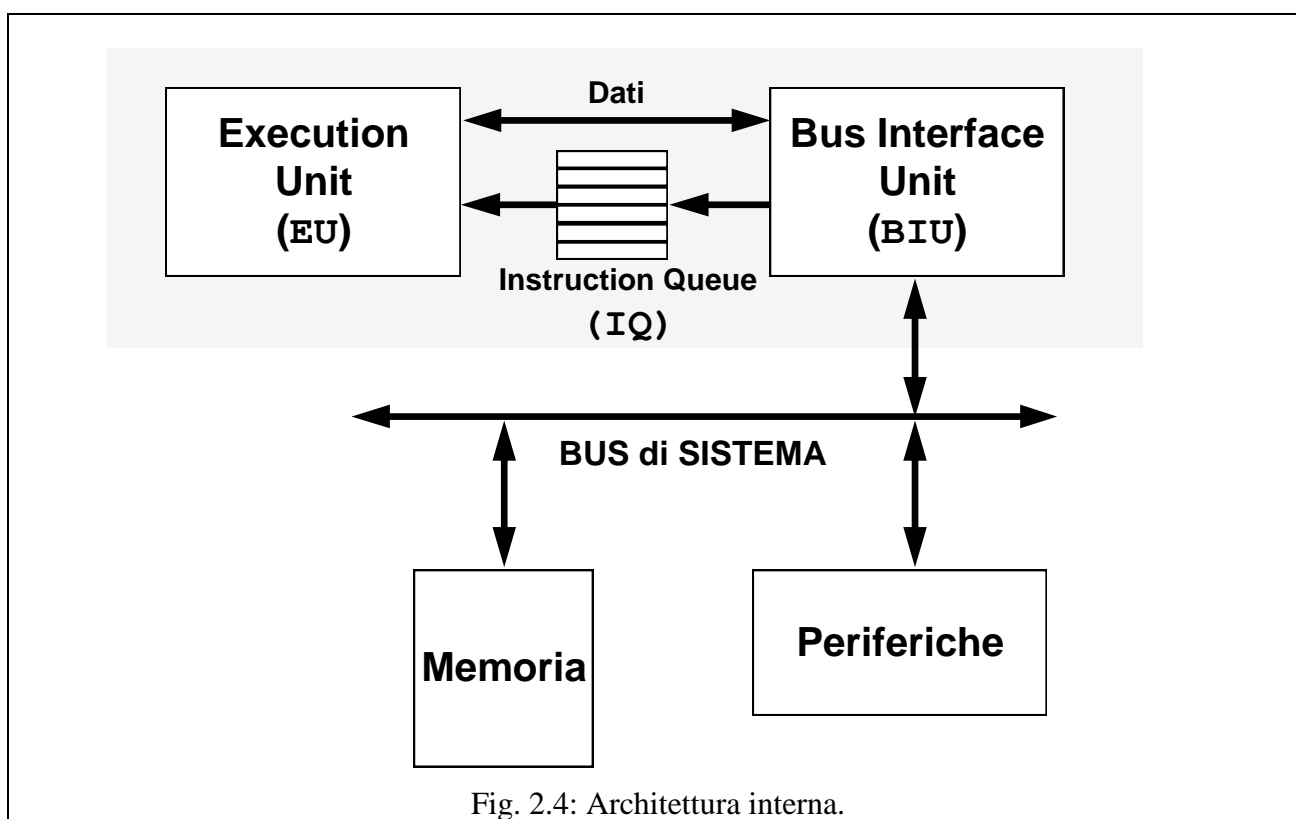


Fig. 2.4: Architettura interna.

2.3.1. L'unità di interfaccia con il bus (BIU)

La BIU è l'interfaccia dell'8086 verso il mondo esterno e gestisce tutte le operazioni che riguardano il bus, e precisamente il prelevamento delle istruzioni, la lettura e scrittura di dati dalla memoria e le operazioni di I/O verso le periferiche.

La BIU è inoltre responsabile dell'accodamento delle istruzioni e della generazione degli indirizzi.

Nel momento in cui la BIU preleva un'istruzione dalla memoria (fase di *prefetch*), provvede a memorizzarla nella *Instruction Queue*. Questa funziona come un buffer *FIFO* (*First-In-First-Out*), dalla quale la EU estrae i byte della successiva istruzione. Se la coda è vuota il primo byte immesso nella coda diventa immediatamente disponibile alla EU.

La dimensione della coda di istruzioni è di 6 byte nell'8086. Quando nella coda sono liberi alme-

no 2 byte, la BIU (se non è già impegnata nella lettura/scrittura di dati) esegue il *prefetch* della successiva istruzione attraverso un ciclo di bus che legge una coppia di due byte consecutivi di memoria. Se la coda è piena e la EU non richiede l'accesso di alcun dato dalla memoria, la BIU non esegue alcun ciclo di bus.

Poiché la BIU non può conoscere la sequenza in cui il programma verrà eseguito, preleva le istruzioni da locazioni di memoria consecutive. Quando la EU deve eseguire un'istruzione non consecutiva alla precedente, non la trova nella coda (tipicamente nel caso di chiamata a **procedura** o di ritorno da procedure o di salto); le istruzioni presenti nella coda risultano in questo caso inutilizzabili e devono essere scartate: la nuova istruzione è prelevata dal nuovo indirizzo di memoria e la coda è ricaricata. È questo l'unico caso in cui la EU deve aspettare il fetch dell'istruzione prima di poterla eseguire.

2.3.2. L'unità di esecuzione (EU)

La EU è responsabile della decodifica e dell'esecuzione delle istruzioni, prelevando le istruzioni dall'IQ e gli operandi o dalla memoria (attraverso la BIU) o dai registri interni.

La EU legge l'istruzione prelevando un byte alla volta dall'IQ, la decodifica, genera, se necessario, l'indirizzo degli operandi, trasferisce tale indirizzo alla BIU per la richiesta di un ciclo di lettura in memoria o su una porta di I/O, ed infine esegue l'istruzione. Al termine dell'esecuzione la EU può fare richiesta alla BIU di un eventuale ciclo per la scrittura in memoria o su una porta di I/O del risultato dell'istruzione.

La Fig. 2.5 visualizza un dettaglio delle due unità. Le sottocomponenti di ciascuna unità verranno descritte nel dettaglio in seguito.

2.3.3. Meccanismo di *pipeline*

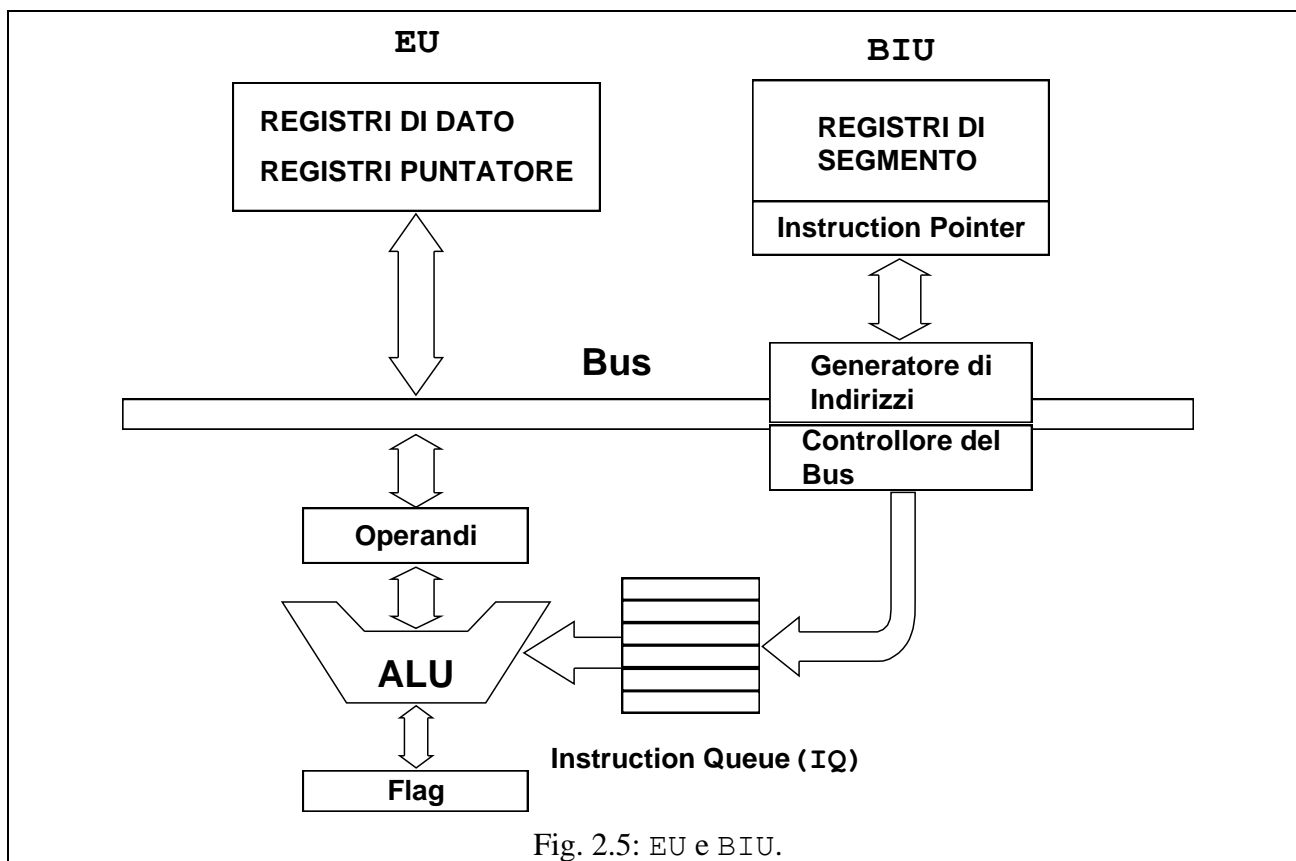
Il meccanismo di cooperazione tra le due unità operative interne all'8086 prende il nome di *pipeline*. Per capire come è strutturata una pipeline si può pensare all'analogia della linea di produzione in una fabbrica manifatturiera, nella quale i vari stadi sono posti in cascata e lavorano in modo che l'uscita di uno stadio coincida con l'ingresso del successivo. A regime, ogni stadio è caratterizzato da una coda di prodotti in attesa di essere processati. Se i tempi di esecuzione delle varie fasi sono uguali il sistema può funzionare in modo ideale: ogni stadio è sempre attivo e la sua coda è sempre vuota; al termine di un'operazione il prodotto è passato allo stadio a valle e l'unità produttiva è pronta a processare il prodotto che gli proviene dallo stadio a monte. In una *pipeline* ideale, a meno del transitorio iniziale in cui i vari stadi devono essere caricati, il tempo medio di fabbricazione è ridotto di un fattore pari al numero di stadi, rispetto al tempo complessivo ottenuto sommando i vari tempi di esecuzione dei singoli stadi.

Nel processore 8086 l'esecuzione di un'istruzione è organizzata come una semplice pipeline a due stadi: la fase di fetch (eseguita dalla BIU) e quella di esecuzione dell'istruzione (eseguita dalla EU).

La pipeline dell'8086 non ha un comportamento ideale per due motivi: da un lato i tempi di esecuzione dei due stadi non sono fissi, ma funzione del tipo di istruzione e dall'altro non tutte le istruzioni in uscita dalla BIU vengono processate dalla EU (nel caso di salti o chiamate a procedure il contenuto della coda di istruzioni deve essere svuotato).

Per sopperire al problema dei diversi tempi di elaborazione, la BIU esegue il prefetch nella IQ; in questo modo non appena la EU è pronta, le viene trasferita la prima istruzione disponibile nella coda.

In Fig. 2.6 è mostrato un confronto tra un'elaborazione sequenziale ed una elaborazione in pipeline: F_i ed E_i rappresentano rispettivamente le fasi di fetch e di esecuzione dell' i -esima istruzione.

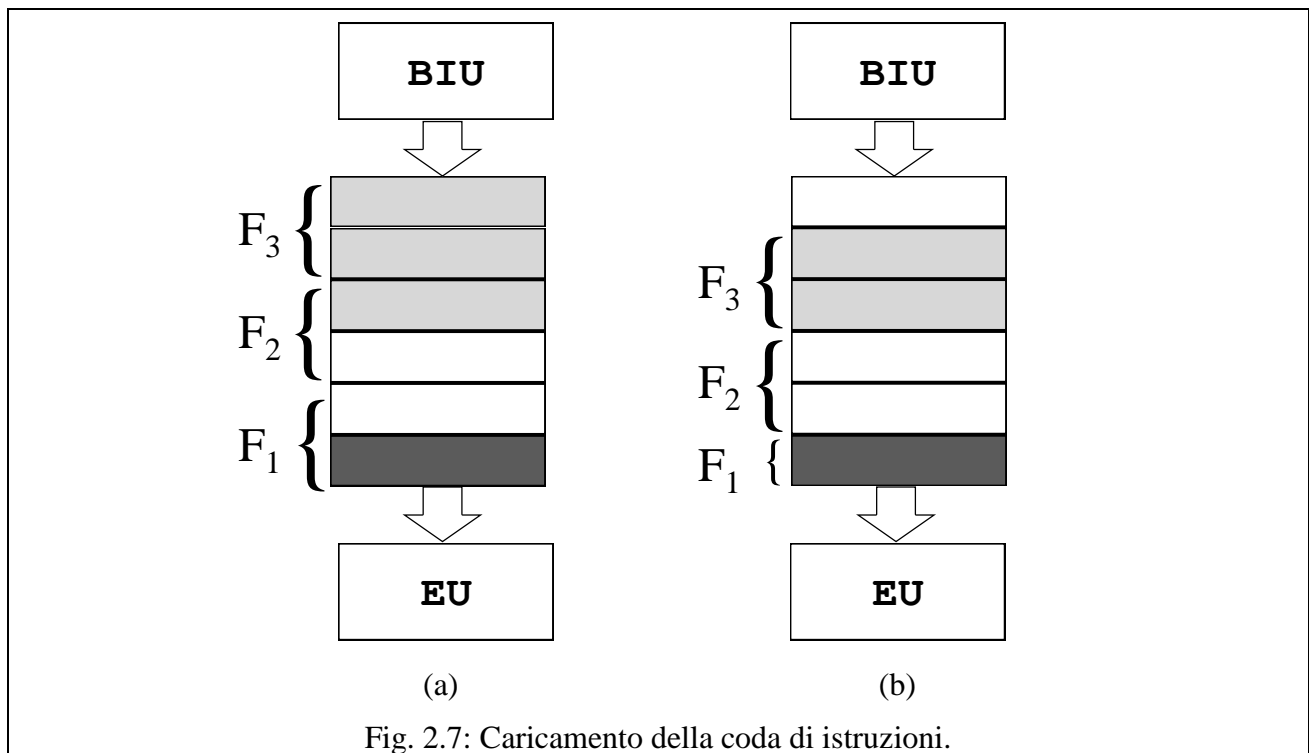
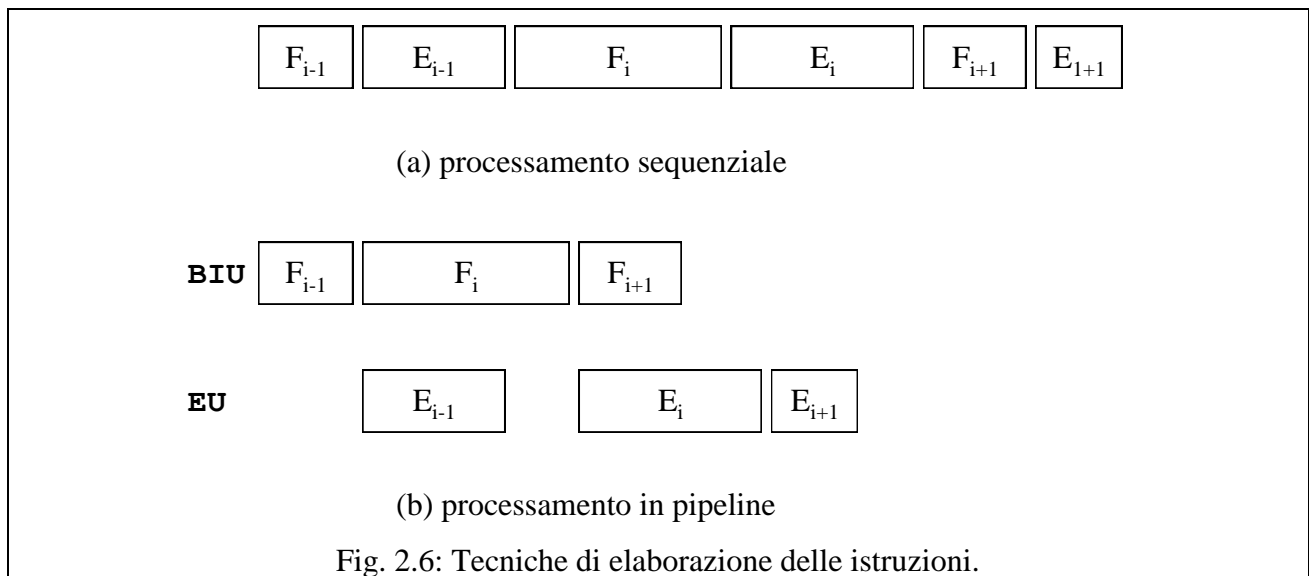


2.3.4. Gestione delle parole di memoria da parte della BIU

L'8086 gestisce i dati in modo da memorizzare il byte meno significativo di una parola nella locazione di memoria avente indirizzo minore (*little endian*).

Nel prelevare una word dalla memoria, la BIU richiede un solo accesso in memoria se il byte meno significativo della word è memorizzato in una locazione avente indirizzo pari; se il byte meno significativo è invece memorizzato ad una locazione avente indirizzo dispari, la BIU deve eseguire due accessi in memoria. Questa operazione è trasparente al programmatore, tranne che per un peggioramento nelle prestazioni. L'efficienza nell'accesso ai dati può essere dunque migliorata allineando i dati ad un indirizzo pari.

Durante il prefetch delle istruzioni, qualora vi siano almeno due byte liberi nella coda, la BIU preleva dalla memoria una word allineata all'indirizzo pari. L'unica eccezione si ha nel caso di salto ad indirizzo dispari. Quando ciò accade, la BIU carica nella coda un solo byte e si allinea all'indirizzo pari successivo; da qui riprende a prelevare word ad indirizzi pari. In Fig. 2.7 è mostrato un esempio di comportamento della BIU, in cui la coda di istruzioni è riempita da una sequenza di 3 fasi di prefetch (F_1 , F_2 ed F_3) successive ad un salto. Si assume che la lunghezza delle istruzioni da eseguire dopo il salto sia pari rispettivamente ad 1 byte, 2 byte e 3 byte. La prima istruzione si trova in Fig. 2.7(a) ad un indirizzo pari, ed in Fig. 2.7(b) ad un indirizzo dispari. In quest'ultimo caso è necessaria una quarta fase di prefetch per caricare completamente la terza istruzione.



2.4. Gestione della memoria

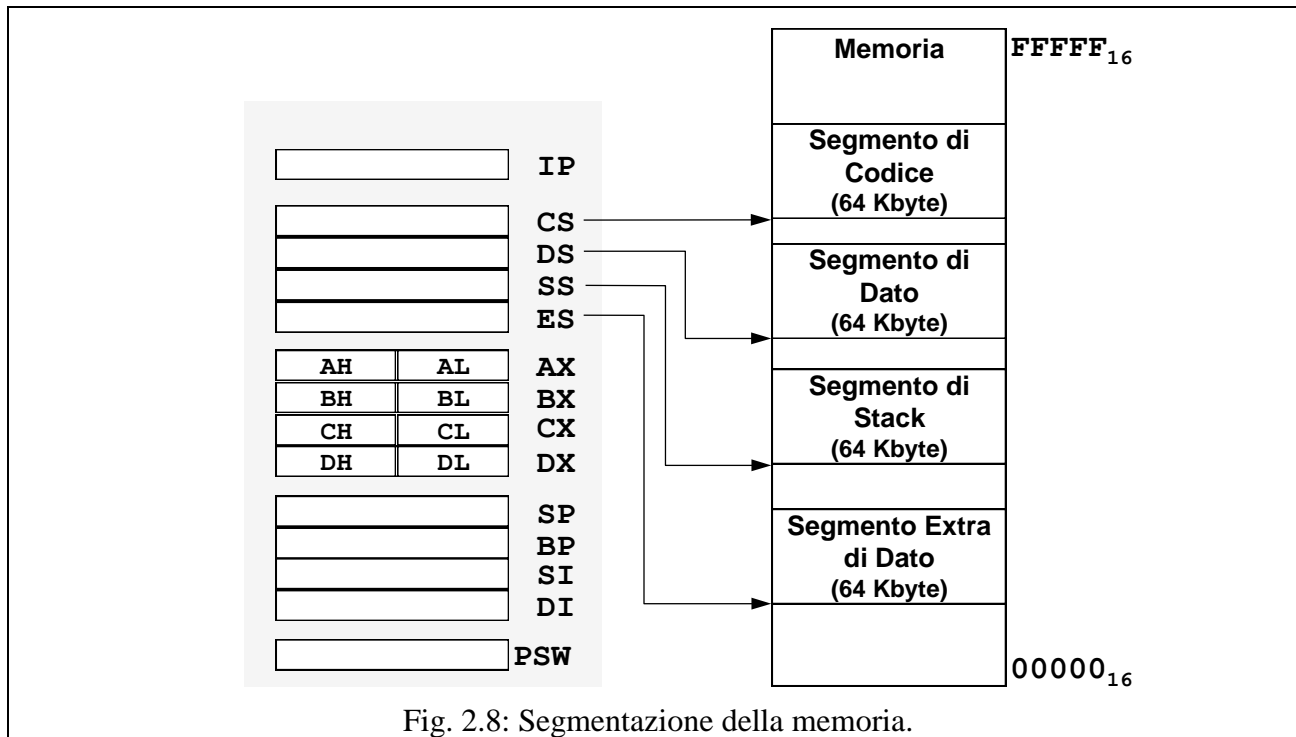
Il meccanismo di gestione della memoria è uno dei punti più importanti nella valutazione di un processore, poiché influenza da un lato le sue prestazioni, che possono essere fortemente limitate da un meccanismo non sufficientemente efficiente, dall'altro le sue possibilità di utilizzazione in sistemi complessi, per i quali può essere essenziale la disponibilità di spazi di memoria di grande dimensione.

2.4.1. Segmentazione della memoria

Sebbene l'8086 abbia uno spazio di indirizzamento complessivo pari ad 1 Mbyte, non tutta la memoria può essere direttamente accessibile contemporaneamente. Viene detta *memoria attiva*

quella porzione che, a un dato istante, è accessibile dal processore.

Lo spazio di indirizzamento è partizionato in blocchi da 64 Kbyte (65.536 locazioni consecutive di memoria) detti *segmenti* (Fig. 2.8). Ogni segmento rappresenta un'unità indipendente di memoria indirizzabile. Ad ogni segmento è assegnato un *indirizzo di base* (*base address*) che ne identifica la cella iniziale.



Solo quattro segmenti possono essere attivi allo stesso istante: un segmento di codice (*code segment*), un segmento di dato (*data segment*), un segmento di stack (*stack segment*) ed un secondo segmento di dato (*extra segment*).

Gli *indirizzi di base* di ciascuno dei segmenti attivi sono memorizzati nei quattro registri di segmento: CS (*code segment register*), DS (*data segment register*), SS (*stack segment register*) ed ES (*extra segment register*).

I quattro registri di segmento danno uno spazio attivo in memoria massimo pari a 256 Kbyte: 64 Kbyte allocati per il codice, 64 Kbyte per lo stack e 128 Kbyte per i dati.

I registri di segmento sono accessibili dall'utente, che può cambiarne il valore attraverso apposite istruzioni.

2.4.2. Generazione degli indirizzi

Il parallelismo dei registri per l'indirizzamento della memoria nell'8086 è pari a 16 bit. Ci si aspetterebbe dunque uno spazio di indirizzamento pari a 2^{16} , ossia 64 Kbyte, mentre l'8086 ha uno spazio di indirizzamento pari a 2^{20} byte, ossia 1 Mbyte.

Questo è possibile grazie al meccanismo di generazione degli indirizzi basato sull'utilizzo di due registri di indirizzo:

- il *registro di segmento*, che contiene l'*indirizzo di base* del segmento di memoria corrispondente alla parola indirizzata
- il *registro di offset*, contenente lo spiazzamento (*effective address*) della parola indirizzata all'interno del segmento.

L'indirizzo scritto sull'A-bus è detto *Indirizzo Fisico* e la sua generazione avviene tramite la se-

guente operazione, che coinvolge il contenuto dei due registri di indirizzo:



$$\text{Indirizzo Fisico} = \text{Spiazzamento} + (16 * \text{Indirizzo di base})$$

L'operazione di calcolo dell'*Indirizzo Fisico* viene eseguita in maniera efficiente sfruttando il fatto che la moltiplicazione per 16 è equivalente ad un semplice spostamento (*shift*) a sinistra di 4 posizioni del contenuto del registro di segmento, come mostrato in Fig. 2.9. Per indicare l'indirizzo fisico di una locazione di memoria si utilizzerà, nel seguito, la seguente notazione:



Registro di segmento : Spiazzamento

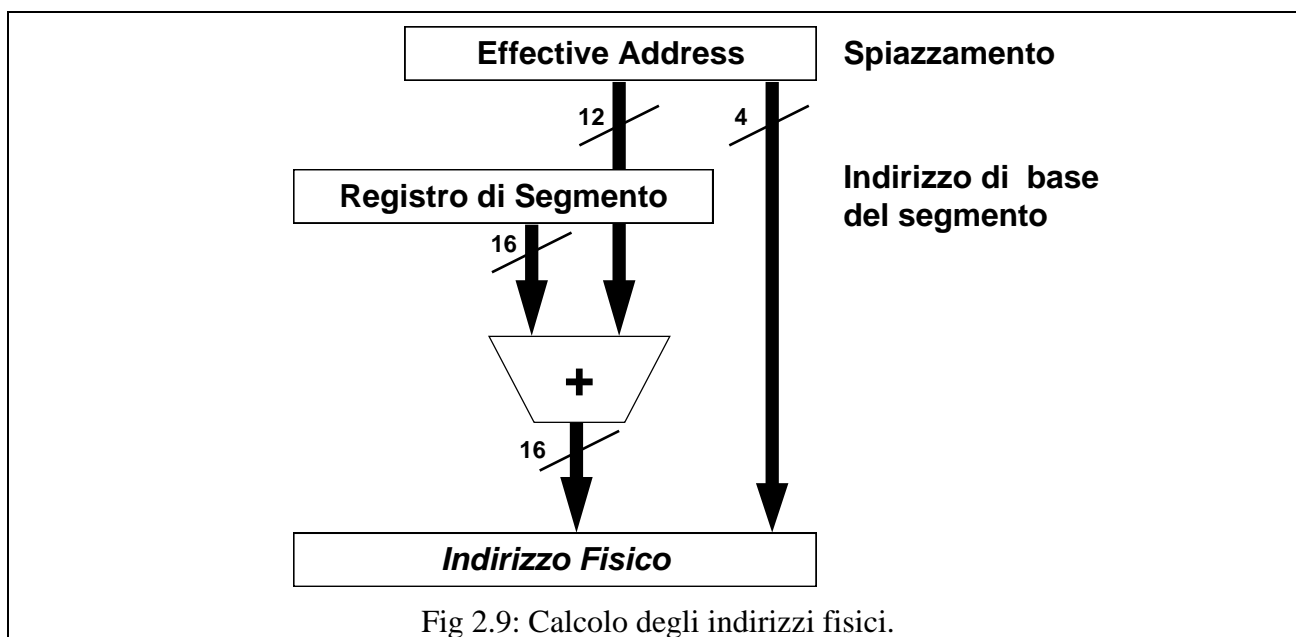


Fig 2.9: Calcolo degli indirizzi fisici.

Esempio

La notazione seguente indica la cella di memoria posta nel segmento indirizzato dal registro DS ad un offset pari al contenuto del registro SI.

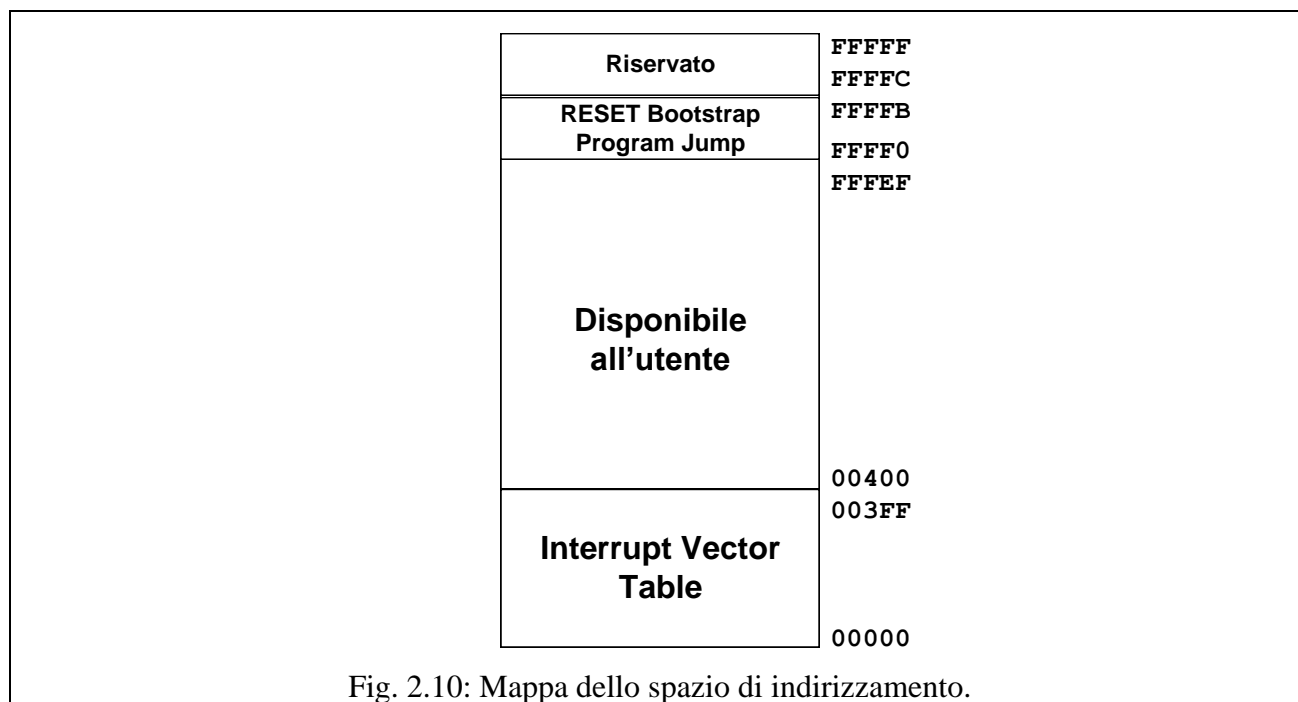
DS : SI

2.4.3. Utilizzo della memoria

Non tutto lo spazio di indirizzamento è utilizzabile dai programmi utente, in quanto, come è illustrato in Fig. 2.10, alcune locazioni di memoria sono riservate per usi specifici.

Gli indirizzi da FFFF0H fino a FFFFFH sono riservati e quindi non utilizzabili; in particolare gli indirizzi da FFFF0H a FFFFBH sono utilizzati per contenere un'istruzione di salto alla routine di caricamento del programma di *bootstrap*. A seguito di un RESET, la CPU esegue il fetch dell'istruzione memorizzata nella locazione FFFF0H, che tipicamente corrisponde ad un salto incondizionato alla opportuna procedura di bootstrap.

Le locazioni da 00000H fino a 003FFH sono riservate per le operazioni di gestione delle interruzioni. Ciascuno dei 256 possibili tipi di interrupt ha una propria routine di servizio, il cui indirizzo è contenuto in 4 byte (2 byte per il segmento e 2 per l'offset) posti in questa zona di memoria.



2.4.4. Lo stack

Lo *stack* è una struttura che permette di memorizzare i dati e di richiamarli secondo una strategia *Last-In-First-Out (LIFO)* (Fig. 2.11). I dati nello stack sono memorizzati come word.

All'inizio di un programma lo stack non contiene dati; le operazioni di caricamento e di prelievo dei dati dallo stack sono chiamate rispettivamente operazioni di *push* e di *pop* e vengono eseguite nell'8086 tramite apposite istruzioni.

La locazione di memoria nella quale è contenuto l'ultimo dato inserito è detta *cima dello stack* (*top of the stack*, TOS); la locazione di memoria che contiene il primo dato inserito è detta *fondo dello stack* (*bottom of the stack*, BOS).

Nell'8086 lo stack corrisponde al segmento di memoria puntato dal registro di segmento *Stack Segment* (SS) e dal registro di offset *Stack Pointer* (SP). L'indirizzo ottenuto attraverso la combinazione SS:SP corrisponde al TOS.

Lo stack cresce da locazioni di memoria con indirizzo maggiore verso quelle con indirizzo minore. Ogni operazione di *push* decrementa di 2 unità il contenuto di SP e trasferisce una word nella locazione puntata da SP; ogni operazione di *pop* estrae una word dalla locazione puntata da SP ed incrementa di 2 unità il contenuto di SP.

Quando si scrive un programma bisogna specificare all'assemblatore la dimensione dello spazio di memoria da riservare per allocare lo stack; la massima dimensione possibile è l'intero segmento di stack, pari dunque a 64 Kbyte. Il massimo numero di elementi che può contenere uno stack è dunque pari a 32K.

La Fig. 2.12 illustra l'effetto di una serie di operazioni sullo stack.

Usi dello stack

Normalmente, lo stack viene utilizzato nell'ambito del meccanismo di chiamata a procedura (per memorizzarne l'indirizzo di ritorno, per il salvataggio dei registri, per l'eventuale passaggio di parametri) e per il salvataggio di variabili temporanee.

Un'approfondita analisi sull'uso dello stack verrà fatta in seguito, in particolare nel Cap. 13.

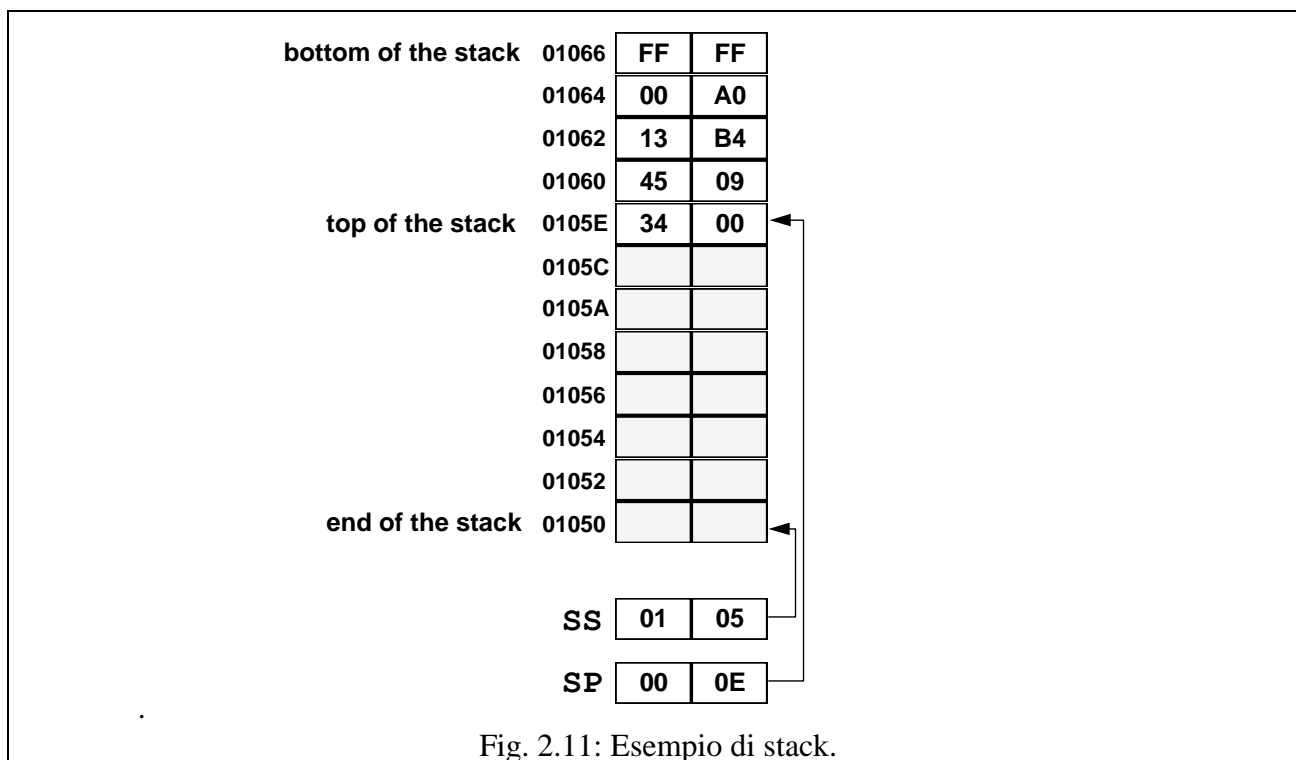


Fig. 2.11: Esempio di stack.

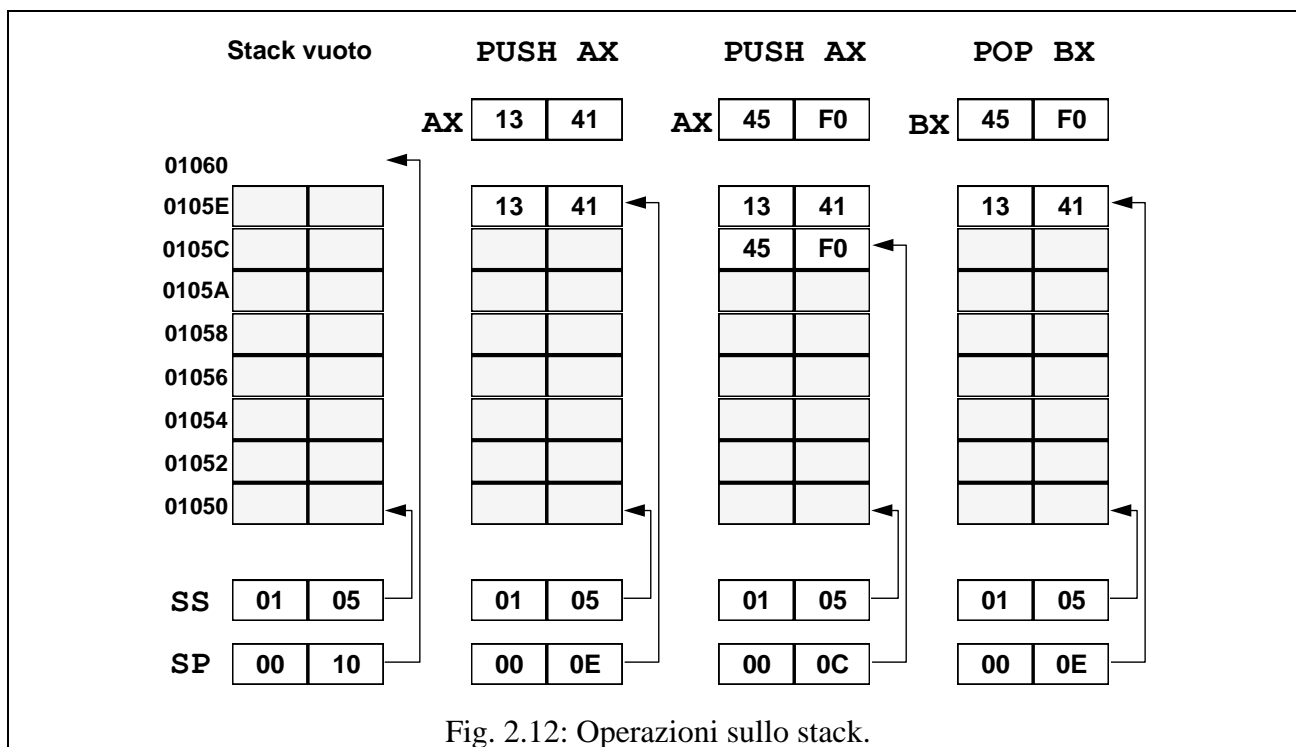


Fig. 2.12: Operazioni sullo stack.

2.5. I registri

Vengono ora descritti e analizzati i registri interni al processore.

2.5.1. L'Instruction Pointer

L'Instruction Pointer (IP) identifica la locazione ove è memorizzata la prossima istruzione da

eseguire all'interno del segmento di codice corrente. Concettualmente svolge le funzioni del *Program Counter* presente in molti altri processori, anche se in realtà l'IP non contiene realmente l'indirizzo fisico dell'istruzione da eseguire, bensì l'offset all'interno del segmento di codice. Come si è visto, l'8086 genera internamente l'indirizzo fisico della prossima istruzione da eseguire combinando il contenuto del registro IP e del registro di segmento CS.

La gestione dell'IP è fatta automaticamente dalla BIU, che ne aggiorna opportunamente il contenuto ogni volta che un'istruzione è prelevata dalla memoria, oppure quando il programma cambia la sequenza delle istruzioni. Ad esempio, quando avviene la chiamata di una procedura, il suo offset viene caricato all'interno di IP; se l'istruzione di chiamata si trova in un segmento differente deve essere aggiornato anche il registro CS.

2.5.2. I registri di dato

L'8086 dispone di quattro registri di dato *general-purpose* di 16 bit ciascuno (AX, BX, CX e DX). Durante l'esecuzione dei programmi, questi vengono tipicamente impiegati per la memorizzazione temporanea di risultati intermedi frequentemente usati. Il vantaggio nel memorizzare i dati in un registro interno piuttosto che in memoria è dovuto al minore tempo di accesso rispetto alla memoria.

Ciascuno dei 4 registri da 16 bit può essere trattato come 2 registri di 8 bit (il nome dei registri di 8 bit si ottiene sostituendo X con L o H a seconda che si tratti del byte basso o del byte alto, rispettivamente).

Tutti i 4 registri di dato sono utilizzabili indistintamente per memorizzare dati, sebbene alcune istruzioni utilizzino alcuni registri in maniera specifica.

A titolo d'esempio:

- AX è usato in moltiplicazioni e divisioni tra word, in operazioni di I/O ed in alcune operazioni tra stringhe; il registro AL è utilizzato nella moltiplicazione e nella divisione tra byte e nelle operazioni aritmetiche BCD; il registro AH è utilizzato nella moltiplicazione e nella divisione tra byte;
- BX è spesso usato per indirizzare dati in memoria;
- CX è usato come contatore nelle operazioni di *loop* e come elemento di conteggio per le operazioni tra stringhe; il registro CL è utilizzato come contatore nelle operazioni di *shift* e *rotate*;
- DX è usato nelle operazioni di moltiplicazione e divisione.

2.5.3. I registri puntatore ed i registri indice

L'8086 dispone di due registri puntatore (BP e SP) e di due registri indice (SI e DI), usati per memorizzare l'offset di locazioni di memoria. I valori di questi registri possono essere letti e modificati attraverso opportune istruzioni macchina.

I registri puntatore, *Base Pointer* (BP) e *Stack Pointer* (SP), sono usati come offset rispetto al valore corrente del *registro di segmento di stack* (SS) durante l'esecuzione di istruzioni che riguardano lo stack.

Il valore di SP rappresenta l'offset dell'ultimo dato memorizzato nello stack; combinato con il valore di SS esso fornisce l'indirizzo fisico della cima dello stack.

Anche il valore di BP rappresenta il valore di un offset all'interno del segmento di stack e viene impiegato in un particolare modo di indirizzamento detto *base addressing mode* (si veda Cap. 7). Un caso frequente di uso del registro BP si ha in connessione con il meccanismo di chiamata delle procedure, in particolare per realizzare il passaggio di parametri attraverso lo stack (si veda Cap. 13).

I registri indice sono usati per memorizzare gli indirizzi di offset nelle istruzioni che fanno acces-

so ai dati e vengono tipicamente combinati con i registri di segmento DS o ES. In istruzioni che usano il tipo di indirizzamento *indexed*, il registro *Source Index* (SI) è comunemente usato per memorizzare l'operando sorgente, mentre il registro *Destination Index* (DI) è impiegato per memorizzare l'offset dell'operando destinazione. Questi registri sono usati inoltre nelle istruzioni che eseguono manipolazione di stringhe. I registri indice possono anche essere utilizzati come registri sorgente e destinazione in operazioni aritmetiche e logiche; a differenza dei registri di dato possono però essere sempre utilizzati come registri di 16 bit e non come coppie di registri di 8 bit.

2.5.4. Il registro di stato

Il *registro di stato*, detto anche *Processor Status Word* (PSW), è un registro di 16 bit contenente 9 flag (Fig. 2.13), suddivisi in *flag di stato* e *flag di controllo*.

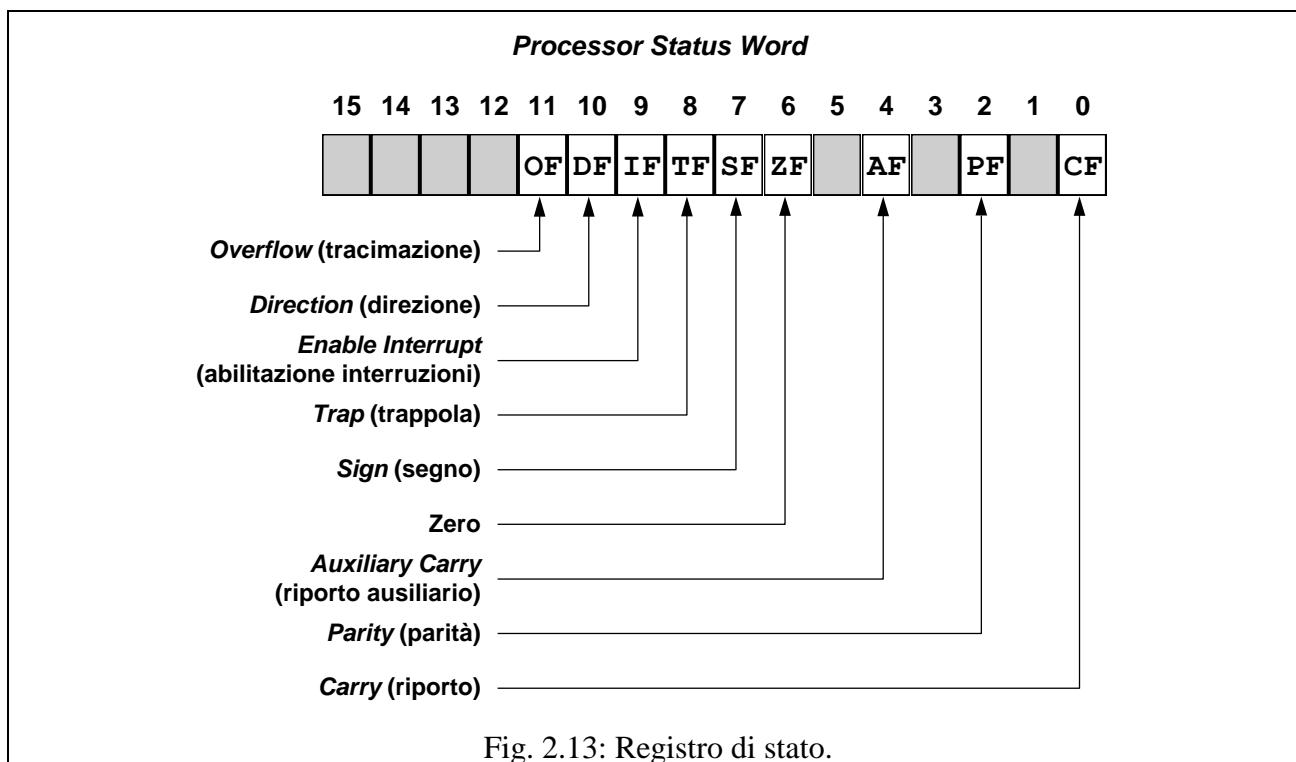
I flag di stato indicano le condizioni prodotte come risultato dell'esecuzione di istruzioni aritmetiche o logiche. Essi sono:

1. flag di carry (**CF**): viene forzato a 1 se un'operazione genera un riporto (*carry*) od un prestito (*borrow*) dall'operando destinazione;
2. flag di parità (**PF**): viene forzato a 1 se è pari il numero di bit a 1 nel byte meno significativo del risultato;
3. flag di carry ausiliario (**AF**): viene forzato a 1 se un'operazione genera un carry od un borrow dal bit 3; è usato nelle operazioni aritmetiche tra numeri rappresentati secondo la notazione BCD (si veda Cap. 10);
4. flag di zero (**ZF**): viene forzato a 1 se il risultato dell'operazione è 0;
5. flag di segno (**SF**): coincide con il bit più significativo del risultato di un'operazione;
6. flag di overflow (**OF**): viene forzato ad 1 se nell'esecuzione di un'operazione si è verificata una condizione di tracimazione (*overflow*).

In seguito all'esecuzione di un'istruzione, il valore di ciascun flag di stato può risultare o modificato o inalterato o indefinito.

Mentre i flag di stato danno informazioni circa le istruzioni che sono state appena eseguite, i flag di controllo modificano il comportamento delle istruzioni che verranno eseguite in seguito. I flag di controllo dell'8086 sono:

1. flag di direzione (**DF**): il valore del flag determina la direzione con cui sono eseguite le operazioni su stringhe; se è forzato ad 1 le istruzioni per la manipolazione delle stringhe decrementano l'indirizzo che punta agli operandi e in questo modo l'elaborazione delle stringhe procede da indirizzi alti verso indirizzi bassi; se è forzato a 0 le stesse istruzioni eseguono l'incremento dell'indirizzo, e l'elaborazione procede verso indirizzi crescenti;
2. flag di abilitazione all'interruzione (**IF**): gli interrupt mascherabili vengono riconosciuti e serviti se e solo se questo flag è forzato ad 1; viene azzerato per disabilitare gli interrupt mascherabili;
3. flag di trap (**TF**): se è forzato ad 1 il processore opera in modo *single-step*, generando una *trap* (ossia la chiamata ad un'apposita procedura di interruzione) al termine di ogni istruzione; questo tipo di operazione è utile per eseguire il *debug* del codice.



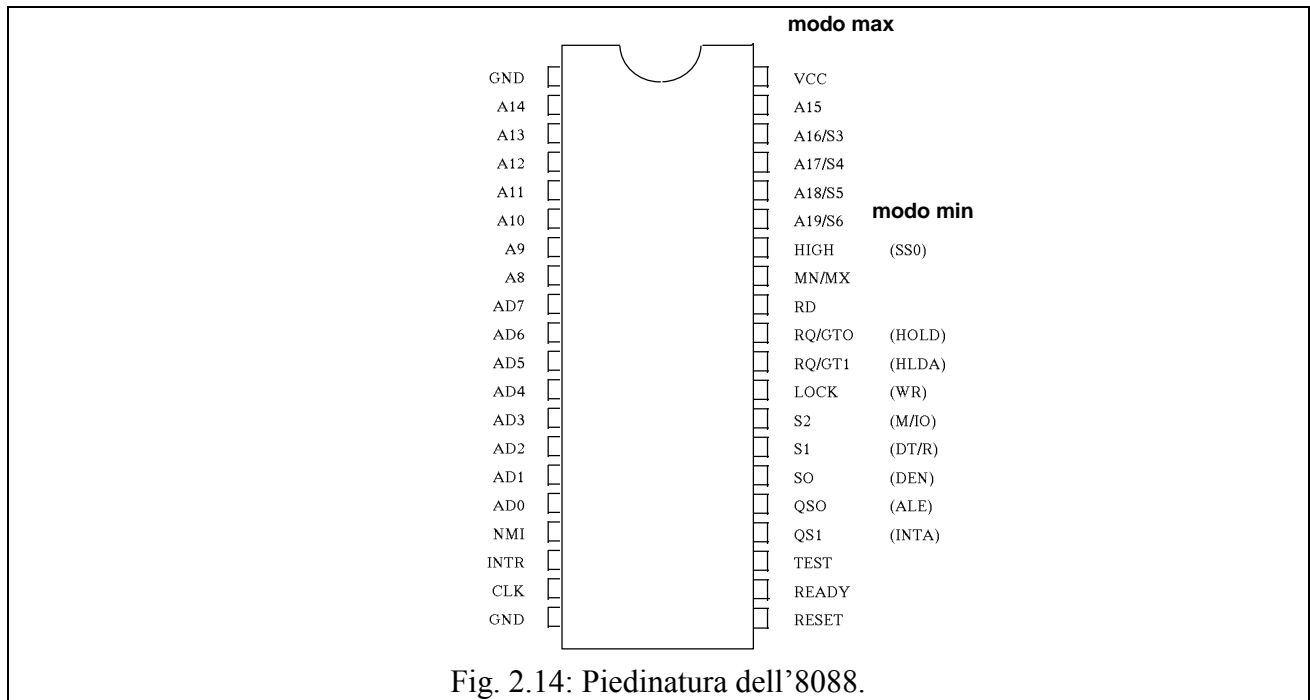
2.6. Il microprocessore 8088

L'8088 è un microprocessore con parallelismo di dati di 8-bit e con completa compatibilità software rispetto all'8086. In Fig. 2.14 è riportata la piedinatura.

L'8088 differisce dall'8086 unicamente in quanto il suo bus dati esterno ha ampiezza pari a 8 bit, invece dei 16 bit dell'8086. Nell'8088, operandi su 16 bit sono dunque letti o scritti in memoria tramite due cicli di bus consecutivi. Dal punto di vista architetturale, l'8088 ha una EU praticamente identica a quella dell'8086, mentre la BIU è diversa, in quanto diverso è il meccanismo di accesso ai bus.

Dal punto di vista del programmatore, i due processori, avendo un uguale insieme di istruzioni, sono assolutamente identici, a meno del tempo di esecuzione. La struttura dei registri interni è la stessa e tutte le istruzioni forniscono identici risultati. Le uniche differenze sono legate alla diversa interfaccia esterna:

- i pin multiplexati tra dati ed indirizzi sono 8 nell'8088 (invece dei 16 nell'8086);
- la IQ nella BIU è lunga 4 byte nell'8088, anziché 6 byte; la ridotta dimensione è dovuta al fatto che l'8088 può prelevare solo un byte alla volta e dunque i tempi di fetch più lunghi avrebbero impedito di utilizzare pienamente una coda di 6 byte;
- il prefetch è operato dalla BIU quando nella IQ c'è un byte libero, mentre per l'8086 occorrono 2 byte liberi per eseguire il prefetch;
- per l'8088 i tempi di esecuzione di tutte le istruzioni che fanno accesso alla memoria sono influenzati dall'interfaccia ad 8 bit e richiedono 1 ciclo di bus per ogni byte.



3. Dal codice sorgente al codice macchina

In questo capitolo verranno analizzati i vari passi attraverso cui si articola la traduzione di un codice sorgente scritto in linguaggio Assembler nel corrispondente codice macchina eseguibile.

3.1. Il codice sorgente

Chi scrive un programma in linguaggio Assembler deve pensare al programma come costituito da *istruzioni* e *dati*. Le istruzioni descrivono l'algoritmo ed i dati costituiscono il materiale su cui le istruzioni agiscono.

La parte che comprende le istruzioni è costituita da una o più procedure (o *subroutine*). Di queste una viene assunta principale (*main program*) ed è la prima ad essere eseguita.

3.2. L'assemblatore

Il compito dell'assemblatore è quello di leggere il programma sorgente scritto in linguaggio Assembler e generare un file contenente il *modulo oggetto*, vale a dire una traduzione del codice Assembler in un formato intermedio, non ancora eseguibile.

L'assemblatore deve: rilevare e segnalare errori di sintassi presenti nel codice sorgente; allocare spazio di memoria per i dati; trasformare *comandi* simbolici negli equivalenti codici binari di linguaggio macchina; assegnare indirizzi relativi alle istruzioni ed ai dati e memorizzare i riferimenti a procedure e dati esterni, ossia non definiti all'interno del modulo

L'assemblatore crea il modulo oggetto byte per byte. Per ogni linea nel codice sorgente contenente una istruzione esso genera l'equivalente istruzione macchina e la aggiunge al modulo oggetto.

Il modulo oggetto si avvicina molto al linguaggio macchina, ma non è ancora eseguibile. In particolare gli indirizzi delle istruzioni e dei dati hanno una numerazione relativa al modulo e non assoluta. Gli indirizzi devono essere successivamente *rilocati* dal *linker* e dal *loader* in funzione dell'indirizzo a partire dal quale il programma sarà caricato in memoria.

L'assemblatore inserisce le informazioni necessarie per il linker in una *relocation table* inclusa

nel modulo oggetto.

3.3. Il linker

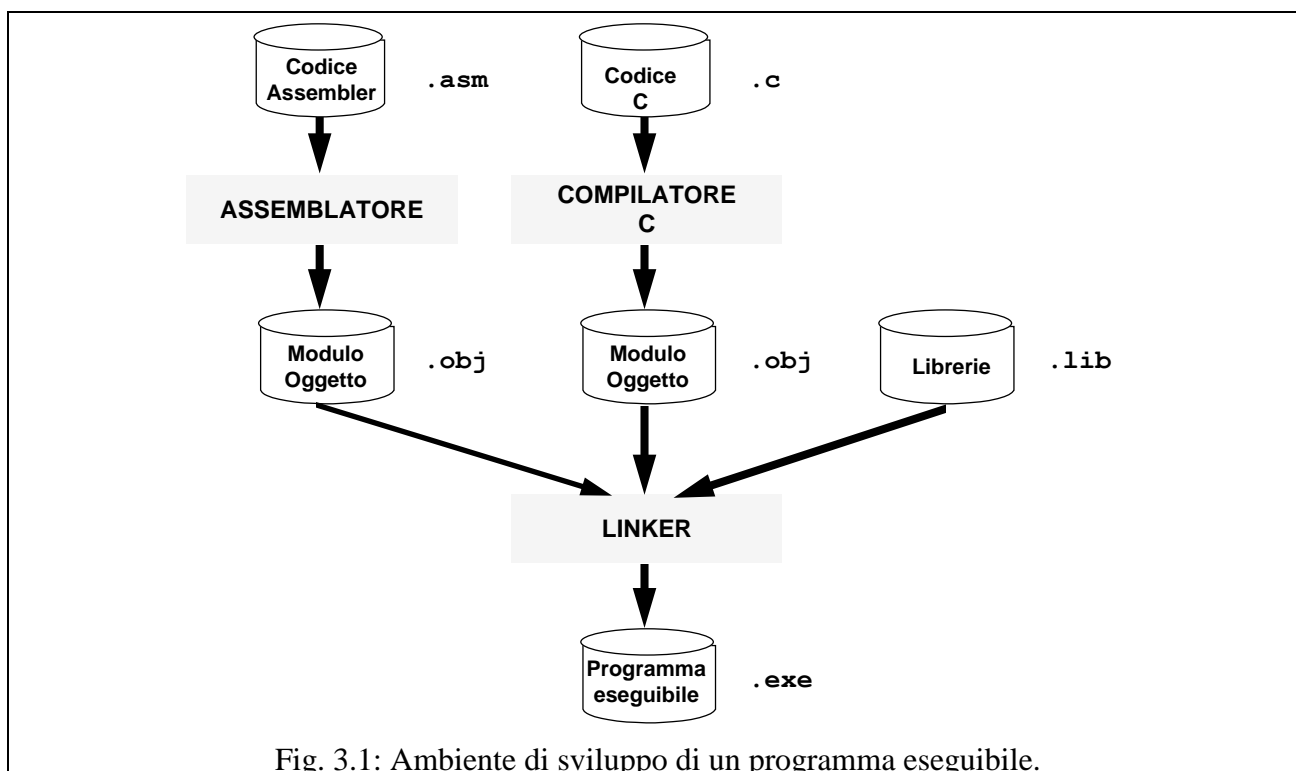
Il linker vede il programma come costituito da uno o più moduli oggetto. Il compito del linker è quello di generare un unico file eseguibile a partire dai diversi moduli oggetto; in particolare deve: ricalcolare (o rilocare) tutti gli indirizzi delle istruzioni delle procedure a seconda dell'indirizzo del segmento di codice assegnato al modulo e rilocare tutti gli indirizzi degli operandi in funzione dell'indirizzo del corrispondente segmento di dato, di stack od extra

Programmi di grande dimensione vengono tipicamente scritti su file sorgenti diversi, assemblati separatamente. È poi compito del linker unire insieme i diversi moduli per generare un unico programma eseguibile.

In generale è possibile che i moduli oggetto siano generati a partire da codici sorgenti scritti in linguaggi di programmazione diversi. Generalmente si scrive in Assembler solo una piccola parte delle procedure che costituiscono il programma completo, che normalmente è scritto in un linguaggio di alto livello.

Per il programmatore sovente è conveniente utilizzare procedure standard già compilate raggruppate in *librerie*. Il linker può dunque richiamare moduli oggetto presenti in libreria e unirli insieme ai moduli generati dal programmatore per generare il programma eseguibile.

La Fig. 3.1 presenta l'ambiente di sviluppo di un programma eseguibile contenente uno o più moduli Assembler, uno o più moduli generati scritti in un linguaggio di alto livello e librerie di funzioni.



3.4. Il loader

Il *loader* (che normalmente fa parte del Sistema Operativo) provvede ad eseguire le operazioni

necessarie al caricamento del programma nella memoria principale e all'eventuale *rilocazione* del codice: tale operazione modifica le istruzioni che dipendono dalla posizione in memoria di codice e dati e porta alla generazione del codice macchina finale, nella forma in cui questo verrà eseguito dal processore.

4. Introduzione al linguaggio Assembler

Questo capitolo mira a formare nel lettore i concetti introduttivi necessari per la scrittura di semplici programmi Assembler. Verranno pertanto rapidamente introdotte molte caratteristiche del linguaggio attraverso esempi costituiti da brevi programmi. La sintassi e le direttive utilizzate sono quelle del Microsoft Assembler (MASM), uno degli assembler più diffusi. Non ci si addenterà in un'analisi approfondita, che verrà invece svolta nei capitoli successivi. Al termine del capitolo il lettore avrà un'idea generale, anche se poco approfondita, del linguaggio e potrà affrontare in modo più proficuo la trattazione successiva.

4.1. Esempio 1: Scrittura di un valore in memoria

Il programma seguente ha il solo scopo di rappresentare lo schema di riferimento cui ispirare i programmi successivi. Il programma esegue una sola, semplicissima operazione: scrivere il valore 0 in una cella di memoria.

```

        .MODEL small
        .STACK
        .DATA
VAR      DW      ?
        .CODE
        .STARTUP
MOV      VAR, 0
        .EXIT
        END
```

Analizziamo nel dettaglio le varie righe che costituiscono questo programma. Distinguiamo innanzitutto tra le righe contenenti comandi per l'assemblatore (*direttive o pseudo-istruzioni*) da quelle contenenti *istruzioni* vere e proprie, che l'assemblatore trasformerà in istruzioni macchina per il processore. I comandi che iniziano con un carattere punto (.) appartengono alla prima categoria. La prima riga contiene l'indicazione per l'assemblatore sul modello di memoria adottato: il modello `small` prevede che il programma sia composto da due soli segmenti: uno per i dati e lo stack ed uno per il codice. È il modello di memoria utilizzato comunemente per i programmi più semplici. La seconda riga contiene una direttiva che riserva uno spazio di memoria per lo stack. La terza contiene una direttiva per l'allocazione delle variabili nel segmento di dato; i comandi di definizione di variabili devono comparire subito dopo questa direttiva. Nella riga successiva infatti appare la direttiva `DW`, tramite la quale il programmatore richiede la definizione di una variabile in memoria di nome `VAR` e di lunghezza pari ad una word (due byte). Il punto interrogativo (?) indica che la variabile non viene inizializzata all'atto della definizione.

La quinta riga contiene la direttiva per l'allocazione del segmento di codice; ad essa deve seguire il programma vero e proprio. Questo tuttavia deve interfacciarsi con il Sistema Operativo, che possiede il controllo del sistema nel momento in cui l'utente richiede l'esecuzione del programma: la direttiva `.STARTUP` gestisce la generazione automatica delle istruzioni necessarie alla creazione di questa interfaccia. Analogamente la direttiva `.EXIT` fa sì che l'assemblatore generi le istruzioni necessarie per il ritorno del controllo al Sistema Operativo, una volta che l'esecuzione del programma è terminata. L'ultima riga contiene una direttiva che segnala all'assemblatore la fine del modulo da assemblare.

Il programma vero e proprio è composto da un'unica istruzione: `MOV VAR, 0`. Essa comanda al

processore di caricare il valore 0 nella variabile VAR prima definita. L'istruzione MOV appartiene al gruppo delle *istruzioni per il trasferimento di dati*; tale trasferimento può corrispondere alla scrittura di un valore costante in un registro (come in questo caso), oppure comportare il trasferimento del contenuto di un registro in un altro registro, oppure di una parola di memoria in un registro, o viceversa. Non è mai possibile accedere con un'unica istruzione a due celle di memoria.

Si noti che il programma non produce effetti visibili all'esterno; per vedere l'effetto del programma occorre andare a leggere il contenuto della memoria.

4.2. Esempio 2: Somma di due valori

Il prossimo programma esegue la somma tra il contenuto di 2 celle di memoria. Poiché ciascuna istruzione non può fare accesso a più di un operando in memoria, è necessario far uso di un registro ausiliario, in questo caso AX.

```
                .MODEL    small
                .STACK
                .DATA
OPD1            DW        10
OPD2            DW        24
RESULT          DW        ?
                .CODE
                .STARTUP
MOV             AX, OPD1
ADD             AX, OPD2
MOV             RESULT, AX
                .EXIT
                END
```

Vediamo le novità rispetto al programma precedente: all'interno del segmento di dati, il cui inizio è segnalato dalla direttiva .DATA, vi sono le definizioni di tre variabili: OPD1, OPD2 e RESULT, corrispondenti, ciascuna, ad una coppia di byte; le prime due sono inizializzate rispettivamente ai valori 10 e 24; la terza non è inizializzata.

Nel segmento di codice vi sono tre istruzioni: la prima carica nel registro AX il valore contenuto nella variabile OPD1. La seconda (ADD) somma al contenuto del registro AX quello della variabile OPD2 e scrive il risultato nel registro AX stesso. La terza copia il contenuto del registro AX, ossia il risultato della somma, nella variabile RESULT. L'istruzione MOV ha sempre due operandi: nel caso, ad esempio, dell'istruzione MOV AX, OPD1 il primo (quello che specifica la *destinazione*) corrisponde ad un registro, il secondo (quello che specifica la *sorgente*) ad una parola di memoria. Il modo in cui vengono specificati gli operandi di un'istruzione prende il nome di *modo di indirizzamento*. L'istruzione vista usa quindi un indirizzamento tramite *registro* per specificare il primo operando ed un indirizzamento *diretto* per specificare il secondo.

Anche in questo caso il programma non produce alcun effetto visibile dall'esterno: analizzando il contenuto della memoria si potrebbe osservare come la cella di memoria corrispondente alla variabile RESULT contenga, dopo l'esecuzione, il valore 34.

4.3. Esempio 3: Somma degli elementi di un vettore (I versione)

Si assuma che nella memoria del sistema sia stato allocato un vettore di cinque elementi di tipo intero, opportunamente inizializzati. Il programma seguente calcola la somma dei valori contenuti nel vettore, e scrive il risultato nella variabile RESULT.

```

.MODEL SMALL
.STACK
.DATA
VETT    DW      5, 7, 3, 4, 3
RESULT  DW      ?
.CODE
.STARTUP
MOV     AX, 0
ADD     AX, VETT
ADD     AX, VETT+2
ADD     AX, VETT+4
ADD     AX, VETT+6
ADD     AX, VETT+8
MOV     RESULT, AX
.EXIT
END

```

Nel segmento dati sono state definite due variabili: la prima, denominata `VETT`, corrisponde ad un vettore di interi, del quale sono stati specificati i valori di inizializzazione; la seconda è una coppia di byte destinata a contenere il risultato.

Nel segmento di codice compare innanzitutto l'istruzione per l'azzeramento del registro `AX`. Le successive cinque istruzioni sommano il contenuto di un elemento del vettore al valore contenuto in `AX`, scrivendo poi in `AX` stesso il risultato. L'istruzione `ADD` appartiene al gruppo delle *istruzioni aritmetiche*, che permettono di eseguire le principali operazioni aritmetiche (somma, sottrazione, moltiplicazione, divisione) su numeri in rappresentazione binaria. Si noti il modo particolare con cui si specifica il secondo operando delle istruzioni di somma: `VETT+4`, ad esempio, indica il contenuto della cella di memoria il cui indirizzo è dato dall'indirizzo di partenza della variabile `VETT`, incrementato di 4 unità. Poiché ogni elemento del vettore **occupa** due byte, `VETT+4` corrisponde all'indirizzo del terzo elemento del vettore. L'ultima istruzione trasferisce il risultato, contenuto nel registro `AX`, nella variabile `RESULT`.

4.4. Esempio 4: Somma degli elementi di un vettore (II versione)

Il programma precedente ha un evidente difetto: non è in grado di gestire vettori di dimensioni medio-grandi, in quanto richiede un'istruzione per ciascun elemento del vettore. Il programma seguente introduce i costrutti di salto, attraverso i quali è possibile realizzare cicli ed iterazioni, e il modo di indirizzamento *indiretto tramite registro*, attraverso il quale è possibile scandire il vettore.

```

DIM      EQU      15
          .MODEL   small
          .STACK
          .DATA
VETT     DW      2, 5, 16, 12, 34, 7, 20, 11, 31, 44, 70, 69, 2, 4, 23
RESULT   DW      ?
          .CODE
          .STARTUP
MOV      AX, 0          ; azzerà il registro AX
MOV      CX, DIM        ; carica in CX la dimensione del vettore
MOV      DI, 0          ; azzerà il registro DI
lab:     ADD      AX, VETT[DI] ; somma ad AX l'i-esimo elemento di VETT
          ADD      DI, 2    ; passa all'elemento successivo
          DEC      CX      ; decrementa il contatore
          CMP      CX, 0    ; confronta il contatore con 0
          JNZ      lab     ; se diverso da 0 salta
          MOV      RESULT, AX ; altrimenti scrivi il risultato
          .EXIT
          END

```

Nella prima riga del programma compare la direttiva `EQU`, che permette di definire un simbolo (nel nostro caso `DIM`) e di metterlo in corrispondenza con una espressione (15). L'assemblatore provvede a sostituire ovunque il simbolo con il corrispondente valore dell'espressione. Utilizzando la direttiva `EQU` si possono così avere a disposizione simboli corrispondenti a costanti numeriche o sequenze di caratteri, incrementando così la leggibilità e la manutenibilità dei programmi.

Ad ogni passo dell'iterazione, il programma somma al registro `AX` il contenuto della cella di memoria il cui indirizzo è dato dalla somma dell'indirizzo di partenza del vettore `VETT` e del contenuto del registro `DI`, che in tal modo funge da indice per la scansione del vettore stesso. Si noti tuttavia che la memoria è organizzata a byte, così che per passare da un elemento al successivo l'indice deve essere incrementato di due unità (istruzione `ADD DI, 2`). L'iterazione deve essere eseguita un numero di volte pari alla dimensione del vettore (`DIM`). Per fare ciò si usa il registro `CX`, inizializzato a `DIM`; ad ogni passo questi viene decrementato di una unità (istruzione `DEC CX`), e confrontato con il valore 0 (istruzione `CMP CX, 0`). Quando il risultato del test, rappresentato dal valore dei flag scritti dall'istruzione `CMP`, indica che `CX` è diverso da 0, il programma esegue un salto (istruzione `JNZ lab`) all'istruzione avente come etichetta `lab`. Altrimenti il programma passa ad eseguire l'istruzione successiva, che carica nella variabile `RESULT` la somma complessiva. L'istruzione `DEC` è un secondo esempio di istruzione aritmetica.

L'istruzione `ADD AX, VETT[DI]` utilizza per il secondo operando un indirizzamento di tipo *diretto con indice*: l'indirizzo dell'operando è dato dal risultato della somma tra l'indirizzo di una variabile ed il contenuto di un registro.

Nel programma compaiono inoltre i *commenti*. Nel linguaggio Assembler x86 tutto ciò che compare tra un carattere `;` e la fine della riga corrisponde ad un commento e viene ignorato dall'assemblatore.

4.5. Esempio 5: Input/Output

I programmi visti sinora non eseguono alcuna operazione di Input/Output. L'Assembler non fornisce direttamente alcun costrutto o istruzione a questo scopo, poiché le modalità di acquisizione dati e visualizzazione degli stessi dipendono fortemente dai dispositivi periferici connessi al sistema utilizzato. Ci si deve quindi appoggiare alle procedure fornite dal Sistema Operativo, qualora questo sia disponibile sul sistema utilizzato. Se si fa riferimento al Sistema Operativo MSDOS (e ai sistemi compatibili con esso), è possibile utilizzare le procedure di sistema per l'I/O che permettono facil-

mente di leggere un carattere da tastiera e di visualizzare un carattere su video. Il meccanismo utilizzato si basa su una *System Call* di DOS, ossia su un'istruzione `INT 21H`, che attiva una procedura in maniera analoga a quanto farebbe un segnale di interrupt esterno. Per specificare il tipo di servizio richiesto al DOS si deve caricare un apposito valore nel registro `AH` prima di eseguire l'istruzione `INT`.

In particolare risultano estremamente utili le seguenti funzioni:

- funzione 1: richiede la presenza del valore `01H` nel registro `AH`; esegue l'acquisizione di un carattere da tastiera e la scrittura del relativo codice ASCII nel registro `AL`
- funzione 2: richiede la presenza del valore `02H` nel registro `AH`; esegue la visualizzazione del carattere il cui codice ASCII è presente nel registro `DL`.

Il programma seguente esegue l'acquisizione da tastiera di 20 caratteri, la loro memorizzazione in un vettore e la loro visualizzazione in ordine inverso a quello di acquisizione.

```

DIM      EQU      20
          .MODEL   small
          .STACK
          .DATA
VETT      DB      DIM DUP(?)
          .CODE
          .STARTUP
          MOV      CX, DIM           ; carica in CX la dimensione del vettore
          MOV      DI, 0             ; azzerà il registro DI
          MOV      AH, 1             ; predisposizione del registro AH
lab1:     INT      21H               ; lettura di un carattere
          MOV      VETT[DI], AL      ; memorizzazione del carattere letto
          INC      DI                ; passa all'elemento successivo
          DEC      CX                ; decrementa il contatore
          CMP      CX, 0              ; confronta il contatore con 0
          JNZ      lab1              ; se diverso da 0 salta
          MOV      CX, DIM
          MOV      AH, 2             ; predisposizione del registro AH
lab2:     DEC      DI                ; aggiornamento del puntatore
          MOV      DL, VETT[DI]      ; predisposizione del registro DL
          INT      21H               ; visualizzazione di un carattere
          DEC      CX                ; decrementa il contatore
          CMP      CX, 0              ; confronta il contatore con 0
          JNZ      lab2              ; se diverso da 0 salta
          .EXIT
          END

```

All'interno del segmento di dato compare la direttiva `VETT DB DIM DUP(?)`; essa definisce una variabile `VETT` composta da `DIM` byte in cui non viene caricato alcun valore; il costrutto `DUP` permette di replicare per un numero di volte specificato l'operazione di allocazione di una variabile definita attraverso una direttiva di allocazione (in questo caso `DB`).

È fondamentale tener presente che le procedure DOS attivate attraverso l'istruzione `INT 21H` permettono di leggere o visualizzare esclusivamente caratteri in formato ASCII; eventuali operazioni di lettura o scrittura di numeri richiedono che sia il programmatore a farsi carico delle conversioni da stringa di caratteri ASCII a numero intero, e viceversa, tramite opportune procedure.

4.6. Esempio 6: Ricerca del carattere minore in una stringa

Il programma che segue esegue la lettura da tastiera di una stringa di 20 caratteri, e la ricerca in essa del carattere alfabeticamente minore. Tale carattere viene poi visualizzato.

```
.MODEL    small
.STACK
DIM      EQU    20
.DATA
TABLE    DB     DIM DUP(?)
.CODE
.STARTUP
MOV      CX, DIM
LEA      DI, TABLE
lab1:    MOV     AH, 1                ; lettura
        INT     21H
        MOV     [DI], AL
        INC     DI
        LOOP    lab1                ; ripeti per 20 volte
        MOV     CL, 0FFH            ; inizializzazione di CL
        MOV     DI, 0
ciclo:   CMP     CL, TABLE[DI]      ; confronta con il minimo attuale
        JB      dopo
        MOV     CL, TABLE[DI]      ; memorizza il nuovo minimo
dopo:    INC     DI
        CMP     DI, DIM
        JB      ciclo
output:  MOV     DL, CL
        MOV     AH, 2
        INT     21H                ; visualizzazione
.EXIT
END
```

Il programma fa uso dell'istruzione `LOOP`, che facilita l'implementazione dei costrutti iterativi: essa esegue il decremento del registro `CX` e il suo confronto con il valore 0. Se `CX` contiene un valore diverso da 0, viene eseguito un salto all'etichetta specificata, altrimenti si passa all'istruzione successiva.

Per accedere agli elementi di un vettore sono stati utilizzati in precedenza diversi modi di indirizzamento. Nell'istruzione `MOV [DI], AL` si è utilizzato il modo *indiretto tramite registro*. Per scrivere in `DI` l'indirizzo del primo elemento di `TABLE` si è utilizzata l'istruzione `LEA DI, TABLE`, che carica nel registro corrispondente al primo operando l'offset della variabile corrispondente al secondo operando.

4.7. Esempio 7: Procedura (I versione)

Il linguaggio Assembler x86 permette la definizione e l'uso di procedure in modo simile a quanto avviene nei linguaggi di alto livello; tuttavia, non esiste alcun meccanismo predefinito per il passaggio dei parametri, né per restituire un valore dalla procedura al programma chiamante.

Quale esempio di procedura, viene presentata ora la procedura `ACAPO` che esegue la visualizzazione di un carattere di *a capo* (*Carriage Return* o `CR`) e di un carattere di *allineamento* (*Line Feed* o `LF`). In questo caso si riporta il testo della sola procedura.

```

LF      EQU      10
CR      EQU      13
ACAP0   PROC
        PUSH     AX                ; salva i registri
        PUSH     DX
        MOV      AH, 2
        MOV      DL, CR
        INT      21H              ; stampa un carriage return
        MOV      DL, LF
        INT      21H              ; stampa un line feed
        POP      DX                ; ripristina i registri
        POP      AX
        RET                     ; ritorno al programma chiamante
ACAP0   ENDP

```

La procedura è delimitata dalle due direttive PROC e ENDP che ne segnalano l'inizio e la fine, rispettivamente. Come prima operazione, la procedura salva nello stack il valore dei due registri AX e DX che vengono modificati al suo interno. L'istruzione PUSH permette di eseguire in maniera semplice ed efficiente l'operazione di inserzione nello stack; al termine della procedura, due istruzioni POP eseguono il ripristino dei due registri (in ordine inverso, essendo lo stack una struttura *LIFO*). La procedura viene attivata dal programma chiamante mediante l'istruzione CALL. Tale istruzione esegue due operazioni: salva nello stack l'indirizzo di ritorno, corrispondente all'indirizzo dell'istruzione successiva a quella di CALL; inoltre carica nell'*Instruction Pointer* IP l'indirizzo della prima istruzione della procedura, eseguendo così il passaggio del flusso di esecuzione del programma. Al termine della procedura l'istruzione RET esegue l'operazione simmetrica di estrazione dallo stack dell'indirizzo di ritorno, che viene caricato nell'IP; in tal modo il flusso di esecuzione ritorna al programma chiamante.

4.8. Esempio 8: Procedura (II versione)

Come secondo esempio, viene presentata ora la procedura INPUT per la conversione da stringa di caratteri a numero binario su 16 bit. Il numero letto si suppone terminato da un carattere di *a capo*, e si assume che sia rappresentabile su 16 bit; al termine la procedura lascia nel registro DX il numero letto e convertito.

```

INPUT   PROC
        PUSH     AX                ; salva i registri
        PUSH     BX
lab:     MOV      DX, 0
        MOV      BX, 10
        MOV      AH, 1            ; legge un carattere
        INT      21H
        CMP      AL, CR           ; AL contiene il carattere <CR>?
        JE       fine            ; Sì: va a fine
        SUB      AL, '0'         ; No: sottrae la codifica ASCII di '0'
        XCHG     AX, BX
        MOV      BH, 0
        MUL      DX              ; moltiplica per 10
        MOV      DX, AX
        ADD      DX, BX          ; somma la cifra letta
        JMP      lab            ; ripeti
fine:    POP      BX              ; ripristina i registri
        POP      AX
        RET                     ; return
INPUT   ENDP

```

La procedura utilizza al suo interno alcune istruzioni non ancora introdotte: l'istruzione `MUL`, ad esempio, esegue la moltiplicazione tra il registro `AX` e l'operando specificato (in questo caso `DX`), lasciando il risultato nella coppia di registri `DX` (parte più significativa) e `AX` (parte meno significativa). L'istruzione `SUB AL, '0'` sottrae al valore contenuto nel registro `AL` il codice ASCII del carattere 0, scrivendo in `AL` il risultato.

L'istruzione `XCHG AX, BX` esegue lo scambio tra il contenuto dei due registri `AX` e `BX`; il suo uso risulta più semplice ed efficiente rispetto ad una serie di tre istruzioni `MOV`, che necessitano di un registro ausiliario per eseguire la stessa operazione.

4.9. Esempio 9: Calcolo di un polinomio

Si presenta ora la procedura `POLIN`, che esegue il calcolo del valore del polinomio x^3+2x^2+3x+7 . Il valore di x viene passato alla procedura tramite il registro `AX`, che al termine contiene il valore dell'espressione complessiva. Si assume che il risultato finale e tutti quelli intermedi siano rappresentabili su 16 bit.

```

    ...
.CODE
    ...
POLIN PROC
    PUSH    BX
    PUSH    CX
    MOV     BX, AX
    MUL     AX                ; calcola x*x
    PUSH    AX                ; salva x*x
    MUL     BX                ; calcola x*x*x
    POP     CX                ; estrae x*x
    SHL     CX, 1             ; calcola 2*x*x
    ADD     AX, CX            ; calcola x*x*x+2*x*x
    PUSH    AX                ; salva x*x*x+2*x*x
    MOV     AX, 3
    MUL     BX                ; calcola 3*x
    POP     BX                ; estrae x*x*x+2*x*x
    ADD     AX, BX            ; calcola x*x*x+2*x*x+3*x
    ADD     AX, 7             ; calcola x*x*x+2*x*x+3*x+7
    POP     CX
    POP     BX
    RET
POLIN ENDP
    ...

```

La procedura fa uso dello stack per il salvataggio di taluni risultati parziali. Inoltre utilizza l'istruzione `SHL CX, 1` per eseguire l'operazione di moltiplicazione per 2 del registro `CX`; si noti che l'istruzione `SHL` in realtà esegue sul primo operando l'operazione di shift verso sinistra di tante posizioni quante indicate dal secondo operando; sfruttando il fatto che la rappresentazione adottata è quella binaria, si può utilizzare l'operazione di shift verso sinistra per implementare la moltiplicazione per una potenza di 2.

5. Formato di un programma in linguaggio Assembler

In questo capitolo vengono descritte le regole da seguire per scrivere un programma in linguaggio Assembler; vengono inoltre analizzate le varie parti che costituiscono un programma.

In questo e nei successivi capitoli verranno utilizzati i metasimboli `{ }` per rappresentare entità che possono essere opzionalmente presenti all'interno di un comando.

5.1. Regole per scrivere un programma in linguaggio Assembler

Un codice in linguaggio Assembler è visto come un insieme di linee di testo contenenti *comandi* (*statement*) per l'assemblatore. Nella scrittura di un programma Assembler valgono le seguenti regole:

- ogni linea può contenere un solo comando;
- non esiste nessun formato stringente sull'indentazione dei comandi;
- il linguaggio è *case insensitive*, ossia non esiste differenza tra lettere maiuscole e minuscole.

Ciascun *comando* può essere costituito da quattro campi:



```
{etichetta:} {codice operativo} {operandi} {; commento}
```

Attraverso il campo *etichetta* è possibile fare riferimento dall'interno del programma ad un particolare comando. Il *codice operativo* caratterizza l'azione richiesta dal comando. I dati su cui il comando opera sono detti *operandi*. È possibile specificare un *commento* al comando per aumentare la leggibilità del codice.

Verranno ora spiegate le regole sintattiche necessarie per scrivere in maniera corretta le varie parti che contraddistinguono un programma.

5.2. Commenti

I commenti sono comandi (o parte di comandi) ignorati dall'assemblatore.

Per un programmatore Assembler i commenti, tipicamente usati per rendere più comprensibile il codice scritto e per descrivere le varie parti che lo costituiscono, assumono un significato importante a causa della intrinseca difficoltà nella comprensione e leggibilità del codice sorgente.

Vi sono diversi modi per includere commenti nel codice:

- ciascuna linea che inizi con un punto e virgola come primo carattere (a parte spazi o TAB) è considerata un commento ed è ignorata dall'assemblatore;

Esempio

```
; questo è un commento
```

- si può aggiungere un commento alla fine di un comando facendolo precedere da un carattere punto e virgola.

Esempio

La seguente istruzione carica il valore 99 nel registro AX:

```
MOV    AX, 99    ; carica il valore 99 in AX
```

5.3. Codice operativo

Il codice operativo è quella parte del comando (espressa attraverso una parola riservata) che specifica l'operazione da svolgere. È necessaria un'ulteriore distinzione all'interno della definizione di *comando*. È infatti possibile distinguere tra *istruzioni* e *direttive* (o *pseudo-istruzioni*).

Una *istruzione* è un comando che, a livello di linguaggio macchina, verrà tradotto dall'assemblatore in un'istruzione eseguibile dal processore.

Una *direttiva* è un comando che controlla l'operato dell'assemblatore. A differenza delle istruzioni, le direttive non causano la generazione di codice oggetto, ma sono interpretate dall'assemblatore come indicazioni e richieste del programmatore per convertire correttamente il codice sorgente in linguaggio macchina. Le direttive possono ad esempio definire segmenti e procedure, definire simboli, riservare spazio di memoria, ecc.

Esempi

I comandi seguenti rappresentano istruzioni convertite in linguaggio macchina dall'assemblatore:

```
ADD    AX, BX    ; AX = AX + BX
MOV    AX, SUM    ; copia in AX il valore di SUM
```

I comandi seguenti sono tre direttive per l'assemblatore. La prima permette di riservare 1 byte di memoria assegnandogli il nome SUM, la seconda specifica all'assemblatore l'inizio di un segmento di nome CSEG, la terza definisce una costante di nome LUNG e di valore 100.

```
SUM    DB        1    ; 1 byte per la variabile SUM
CSEG   SEGMENT    ; inizio del segmento di codice
LUNG   EQU       100  ; definizione di costante
```

5.4. Operandi

Gli operandi costituiscono quella parte del comando che specifica le entità su cui operano le istruzioni o le direttive all'assemblatore.

È possibile distinguere i seguenti tipi di operandi:

- contenuto di un registro;
- contenuto di una cella di memoria;
- indirizzo di una cella di memoria;
- costante numerica;
- stringa.

La differenza tra i diversi tipi di operando ed i diversi modi per accedere a tali operandi verranno analizzati nei Capp. 6 e 7, che descrivono, rispettivamente, le direttive che fanno uso di operandi ed i diversi metodi di indirizzamento.

5.5. Identificatori

Con il termine *identificatore* si intende una sequenza di caratteri alfabetici che il programmatore può assegnare a varie entità del programma quali variabili, costanti, procedure ed istruzioni, per farvi riferimento in modo semplice ed immediato.

Esempio

La seguente direttiva segnala l'inizio di una procedura: GETDATA è l'identificatore che definisce il nome di una procedura.

```
GETDATA PROC ; procedura acquisizione dati
```

Durante il processo di assemblaggio, l'assemblatore memorizza gli identificatori in una tabella chiamata *symbol table*, in cui, in corrispondenza del nome dell'identificatore, vengono associati gli indirizzi di offset e di segmento.

Esempi

L'istruzione seguente ha come operando GETDATA:

```
CALL GETDATA ; acquisizione dati
```

L'assemblatore sostituisce al nome GETDATA il suo indirizzo, memorizzato nella *symbol table*. Nel codice oggetto si ha dunque il codice di una istruzione che fa un salto alla procedura che inizia alla cella di memoria etichettata dal simbolo GETDATA. Nell'esempio seguente l'istruzione MOV copia il contenuto del registro AX nella variabile di nome SUM:

```
MOV SUM, AX ; copia il contenuto di AX in SUM
```

Nell'esempio seguente l'identificatore `ciclo` etichetta l'istruzione MOV; quando il programma esegue l'istruzione JMP effettua un salto all'istruzione MOV:

```
ciclo: MOV AX, BX ; copia il registro BX in AX
      ... ; altre istruzioni
      JMP ciclo
```

Esistono alcune regole per l'assegnazione dei nomi agli identificatori:

- si utilizzano esclusivamente le lettere (maiuscole e minuscole), le cifre (da 0 a 9) ed i seguenti quattro caratteri speciali: ? @ _ \$
- il primo carattere di un identificatore non può essere una cifra (questo permette all'assemblatore di distinguere tra numeri e identificatori);
- non sono ammessi nomi più lunghi di 247 caratteri;
- non possono essere utilizzate come identificatori alcune parole riservate all'assemblatore.

È buona norma evitare di iniziare un identificatore con un carattere @, poiché identificatori che iniziano con @ sono utilizzati dall'assemblatore come simboli interni.

Esempi

I seguenti identificatori sono validi:



```
HELLO
$MARKET
A12345
LONG_NAME
PART_2
```

I seguenti identificatori non sono validi:



```
LONG-NAME
2_PART
'CIAO'
ADD
```

Queste regole permettono grande libertà nella scelta degli identificatori; ciononostante il buon senso suggerisce di evitare identificatori molto lunghi, ma di scegliere preferibilmente nomi aventi un senso compiuto, attinente il più possibile all'entità a cui sono associati.

5.6. Costanti numeriche

È possibile specificare costanti numeriche in rappresentazione decimale, esadecimale o binaria. Per specificare una costante decimale basta scrivere il suo valore utilizzando le cifre 0-9.

Esempio

L'istruzione seguente carica il registro AX con il valore decimale 855.

```
MOV      AX, 855          ; carica in AX il valore 855
```

Per specificare una costante esadecimale si utilizzano le cifre 0-9 e le lettere A-F (maiuscole o minuscole). Occorre aggiungere il carattere H alla fine per indicare che il numero è in rappresentazione esadecimale.

Esempio

L'istruzione seguente carica nel registro AX il valore esadecimale 855 (equivalente al numero decimale 2133).

```
MOV      AX, 855H         ; carica in AX il valore 855 hex
```

Se il numero esadecimale comincia con una lettera, occorre far precedere il numero da uno 0. In tal modo l'assemblatore riesce a distinguere tra un numero ed un identificatore.

Esempio

La prima istruzione carica nel registro AX il valore esadecimale FF (equivalente al numero decimale 255); la seconda carica nel registro AX il valore della variabile di nome FFH.

```
MOV    AX, 0FFH      ; carica in AX il valore FF hex
MOV    AX, FFH       ; copia in AX il contenuto della
                     ; variabile di nome FFH
```

Per specificare un numero binario si usano le cifre 0 e 1 e si aggiunge il carattere B alla fine del numero.

Esempio

La seguente istruzione carica il valore binario 011010010110B (equivalente al numero decimale 1686 ed al numero esadecimale 696).

```
MOV    AX, 011010010110B ; carica in AX il numero
                          ; binario 011010010110
```


6. Direttive per l'assemblatore

In questo capitolo vengono presentate le principali direttive per l'assemblatore facendo specifico riferimento all'assemblatore *Microsoft MASM 6.0*.

Al termine viene fornito uno “scheletro” di riferimento per la scrittura di programmi Assembler. Nell'ultima sezione è introdotto l'emulatore **emu8086**, utile per un primo approccio all'architettura x86, e ne sono evidenziate le differenze di sintassi rispetto a MASM.

6.1. Direttive per la definizione dei dati

L'assemblatore fornisce un ristretto insieme di tipi di dato, di dimensione pari a multipli del *byte*, unità elementare di memoria. In Tab 6.1 sono elencati i diversi tipi di dato, la relativa denominazione Assembler e la corrispondente dimensione.

<i>Tipo di dato</i>	<i>Nome</i>	<i>Numero di byte</i>
BYTE	<i>byte</i>	1
WORD	<i>word</i>	2
DWORD	<i>doubleword</i>	4
QWORD	<i>quadword</i>	8
TBYTE	<i>tenbyte</i>	10

Tab. 6.1: Tipi di dato.

I dati più frequentemente usati sono di tipo BYTE o WORD. Il tipo BYTE è usato per memorizzare numeri o caratteri. Il tipo WORD permette di memorizzare numeri in un intervallo di rappresentazione più ampio. I tipi DWORD, QWORD e TBYTE sono in genere usati per memorizzare numeri reali per il *coprocessore matematico*.

Word e *doubleword* sono anche usate per memorizzare indirizzi: gli *offset* sono memorizzati in una *word*, mentre l'intero indirizzo (registro di segmento e offset) è memorizzato in una *doubleword*.

I formati utilizzati per la definizione di una variabile sono i seguenti:



```
{nome}  DIRETTIVA  ?  
{nome}  DIRETTIVA  valore
```

Il campo *nome* specifica l'identificatore della variabile, ma può essere omissa. Per definire un dato occorre utilizzare l'opportuna *DIRETTIVA* che ne specifica il tipo. Le *DIRETTIVE* utilizzabili verranno descritte in dettaglio nel seguito di questo Capitolo. Una variabile può essere inizializzata all'atto della definizione mediante il campo *valore*; il simbolo ? specifica che la variabile, all'atto della definizione, non assume alcun valore di inizializzazione.

Esempio

Il primo comando definisce una variabile non inizializzata di tipo `BYTE` e di nome `TOTALE`; il secondo definisce una variabile di tipo `WORD`, di nome `DOLLARO` ed inizializzata al valore 1588.

```
TOTALE    DB    ?
DOLLARO   DW    1588
```

In alcuni casi è necessario definire una *tabella* di dati: per fare ciò basta separare con virgole i diversi operandi. I possibili formati di definizione di una tabella di dati sono i seguenti:



```
{nome} DIRETTIVA    ?, ?, ?, ?, ..., ?
{nome} DIRETTIVA    val_1, val_2, val_3, ..., val_n
```

Il numero di caratteri `?` corrisponde al numero di variabili che devono essere allocate in memoria.

Esempi

Il primo comando definisce una tabella di 5 variabili di tipo `DWORD`; nel secondo si definisce una tabella contenente 10 variabili inizializzate di tipo `WORD`:

```
LISTA     DD    ?, ?, ?, ?, ?
TABELLA   DW    10, 100, 0, 25, 65, 23, 34, 2, 1, 125
```

6.1.1. Direttiva DUP

La direttiva **DUP** permette di definire tabelle costituite da valori replicati un certo numero di volte; il suo formato è il seguente:



```
{nome} DIRETTIVA    numero DUP (?)
{nome} DIRETTIVA    numero DUP (valore)
```

Il primo comando crea uno spazio di memoria necessario per memorizzare un *numero* di variabili non inizializzate del tipo specificato dalla *DIRETTIVA*; il secondo inizializza ogni variabile al valore specificato.

Esempi

La variabile di nome `WLISTA` è costituita da 200 word.

```
WLISTA     DW    200 DUP (?)
```

La tabella di nome `TRATTINI` è costituita da 35 byte, ciascuno contenente la codifica ASCII del carattere “-”.

```
TRATTINI   DB    35 DUP ("-")
```

È possibile creare costanti più complesse combinando espressioni `DUP` con altri valori.

Esempio

La seguente direttiva definisce una lista di word contenente i valori 1, 2, 3, 10 volte 0, 99 e

100; il numero totale di word allocate è dunque pari a 15.

```
WLISTA    DW      1, 2, 3, 10 DUP(0), 99, 100
```

Le espressioni DUP possono essere annidate.

Esempi

Il comando seguente definisce una tabella di 100 word in cui la sequenza 6, 7, 5 volte 0, 5, 8, 9 è ripetuta 10 volte:

```
WLISTA    DW      10 DUP (6, 7, 5 DUP(0), 5, 8, 9)
```

I due comandi seguenti sono equivalenti ed allocano la stessa area dati:

```
TABELLA    DW      3 DUP (3 DUP (1))
TABELLA    DW      1, 1, 1, 1, 1, 1, 1, 1, 1, 1
```

6.1.2. Puntatori a dati

L'identificatore associato ad una variabile, definito nella direttiva di definizione dei dati, svolge il ruolo di *puntatore* alla locazione di memoria in cui è memorizzata la variabile. Nel caso in cui al nome sia associata una tabella di dati, il nome punta al primo elemento della tabella.

Esempi

La seguente direttiva DB definisce una tabella di 4 variabili di tipo BYTE corrispondente all'identificatore LISTA:

```
LISTA     DB      4 DUP(?)
```

All'interno di una istruzione si può utilizzare il nome LISTA per fare riferimento al primo dato della tabella. La seguente istruzione esegue la copia del primo dato della tabella LISTA in AH:

```
MOV       AH, LISTA
```

Per fare riferimento ai dati successivi al primo contenuti in tabella si deve utilizzare l'operatore aritmetico + per incrementare l'indirizzo del puntatore di 1 unità per ogni dato di tipo BYTE, 2 unità per ogni dato di WORD, 4 unità per ogni dato di DWORD, ecc.

Esempi

L'istruzione seguente copia il secondo elemento della tabella nel registro BH:

```
MOV       BH, LISTA+1
```

Si consideri il seguente comando in cui è definita una tabella di 500 word:

```
WTAB      DW      500 DUP(?)
```

Volendo fare riferimento alla prima word si utilizza l'operando WTAB; per puntare alla seconda word si utilizza l'operando WTAB+2, per puntare alla terza word si utilizza l'operando WTAB+4, e così via. L'istruzione seguente copia il contenuto della quarta word della tabella

WTAB nel registro AX:

```
MOV    AX, WTAB+6
```

Non è necessario assegnare un identificatore a tutti i dati definiti. È possibile puntare ad un dato facendo riferimento all'indirizzo dell'ultimo identificatore definito.

Esempio

```
TABELLA  DB    1, 2, 3, 4, 5
          DB    2 DUP(?)
          DB    8, 9, 10
```

Questi comandi definiscono una tabella di 10 byte. I primi 5 e gli ultimi 3 sono inizializzati ad un valore, mentre i byte 6 e 7 non sono inizializzati. L'assemblatore alloca spazio di memoria per i dati nello stesso ordine in cui li incontra nel codice sorgente; è possibile dunque far riferimento ai diversi byte della tabella utilizzando l'identificatore TABELLA. È possibile puntare ai byte successivi anche se non sono stati definiti sulla stessa linea; si può quindi far riferimento ai byte 6 e 7 con TABELLA+5 e TABELLA+6.

6.1.3. Direttiva DB

La direttiva **DB** (*Define Bytes*) permette di definire strutture dati costituite da byte.

Utilizzo della direttiva DB per la memorizzazione di caratteri e stringhe

Ciascun carattere è memorizzato secondo la sua codifica ASCII. I seguenti due formati per la definizione di un carattere sono equivalenti:



```
{nome}  DB    'carattere'
{nome}  DB    "carattere"
```

Esempi

I due comandi seguenti sono equivalenti, entrambi definiscono una variabile di nome STELLA contenente il carattere '*'. L'assemblatore inizializza la variabile STELLA con la codifica ASCII del carattere '*':

```
STELLA  DB    '*'
STELLA  DB    "*" 
```

Una serie di caratteri consecutivi si definisce *stringa*. È possibile definire una stringa attraverso la direttiva DB.

Esempio

I due comandi seguenti sono equivalenti:

```
TORINO  DB    "T", "o", "r", "i", "n", "o"
TORINO  DB    "Torino"
```

L'assemblatore non alloca spazio in memoria per memorizzare gli apici delimitatori di stringa. Per includere all'interno della stringa memorizzata i doppi apici è necessario utilizzare come delimitatore di stringa i singoli apici; analogamente per includere all'interno della stringa i singoli apici è

necessario utilizzare come delimitatore di stringa i doppi apici.

Esempi

I due comandi seguenti definiscono due stringhe di caratteri:

```
ERR_MSG    DB    "Trovato l'errore!!"
OK_MSG     DB    'Programma "perfetto"!!!'
```

Può succedere di voler utilizzare un carattere ASCII cui non è associato nessun simbolo. In questo caso occorre specificare la sua codifica ASCII decimale od esadecimale, senza usare gli apici.

Ad esempio, dopo aver inviato un messaggio al video, abitualmente si vuole portare il cursore all'inizio della riga successiva. Per fare questo occorre aggiungere in coda al messaggio i due caratteri speciali di *Carriage Return* (CR, ritorno carrello) e di *Line Feed* (LF, avanzamento di una riga), aventi rispettivamente le codifiche ASCII esadecimali 0D e 0A.

Esempio

Il comando seguente memorizza un messaggio terminato dai caratteri ASCII CR e LF:

```
MSG1       DB    "Inizio del programma", 0DH, 0AH
```

È possibile definire stringhe contenenti caratteri e codici ASCII esadecimali separati da virgole.

Esempio

Il seguente comando definisce una stringa che (una volta inviata al video) visualizza un messaggio, va a capo e stampa il carattere ">".

```
MSG1       DB    "Inizio del programma", 0DH, 0AH, ">"
```

Utilizzo della direttiva DB per memorizzare numeri

In un byte è possibile memorizzare numeri interi con o senza segno all'interno dell'intervallo di numeri mostrato nella Tab. 6.2.

Esempio

Il primo comando definisce una tabella di costanti inizializzata con una sequenza di numeri interi senza segno; nel secondo la tabella è costituita da numeri interi con segno.

```
TAB1       DB    0, 34, 0FFH, 47, 253, 10011010B
TAB2       DB    0, -34, 127, -128, -77H, -00110000B
```

<i>Tipi di numeri</i>	<i>Intervallo</i>
Senza segno	0 ÷ 255
Con segno	-128 ÷ 127

Tab. 6.2: Intervallo di numeri che possono essere memorizzati in un byte

6.1.4. Direttiva DW

La direttiva **DW** (*Define Words*) definisce dati memorizzati su 1 word (2 byte). Su una word si

possono memorizzare un singolo carattere o una coppia di caratteri o un numero intero od un indirizzo di offset.

Utilizzo della direttiva DW per la memorizzazione di caratteri

In un dato di tipo WORD è possibile memorizzare un carattere o una coppia di caratteri. La direttiva DW non può essere utilizzata per memorizzare una sequenza di più di due caratteri.

Esempi

Il primo comando definisce una variabile di nome BICAR inizializzata con il valore "13"; il secondo definisce una variabile di nome CAR di tipo WORD in cui è memorizzato il solo carattere "1". In Fig. 6.1 è illustrato il contenuto della memoria dopo i due comandi seguenti. Nel caso di definizione di una coppia di caratteri, essi sono memorizzati in ordine inverso rispetto alla loro definizione (nel byte ad indirizzo minore il secondo carattere, nel byte successivo il primo carattere); mentre nel caso di memorizzazione di un solo carattere, l'assemblatore pone a 00H il byte ad indirizzo maggiore e scrive il codice ASCII del carattere nel byte ad indirizzo minore.

BICAR	DW	"13"
CAR	DW	"1"

Utilizzo della direttiva DW per la memorizzazione di numeri

Come per i byte, ciascuna word può memorizzare numeri interi con o senza segno; ovviamente l'intervallo di valori rappresentabili è maggiore, come illustrato in Tab. 6.3.

<i>Tipi di numeri</i>	<i>Intervallo</i>
Senza segno	0 ÷ 65535
Con segno	-32768 ÷ 32767

Tab. 6.3: Intervallo di numeri interi memorizzabili in una word.

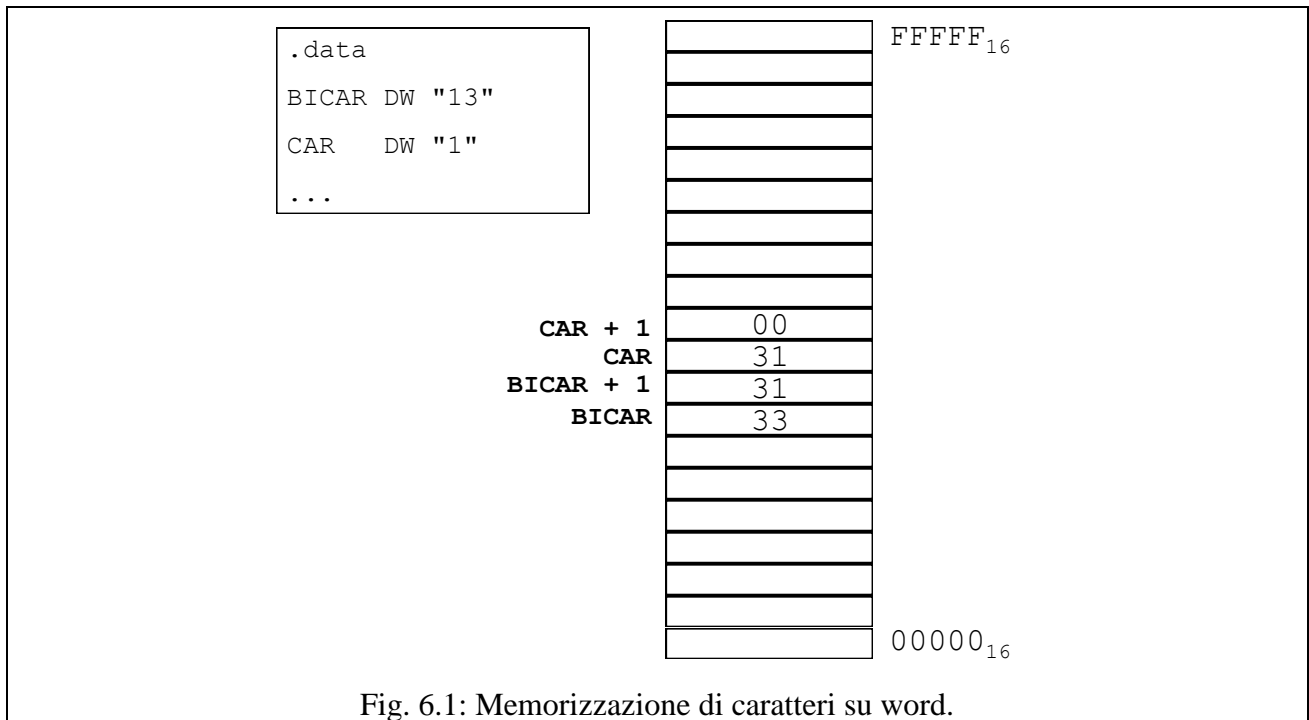


Fig. 6.1: Memorizzazione di caratteri su word.

Esempio

Il primo comando definisce una tabella di variabili inizializzata con una sequenza di numeri interi senza segno; nel secondo la tabella è costituita da numeri interi con segno.

```

TAB1      DW      0, 3499, 0FFAAH, 47, 1001101000000111B
TAB2      DW      0, -31999, -770H, -0011000010010011B

```

Utilizzo della direttiva DW per la memorizzazione degli offset

Una word è uno spazio di memoria sufficiente per memorizzare un offset. In una word si memorizzano gli indirizzi di un dato o di una procedura.

Per inizializzare una word con l'offset di un dato bisogna prima specificare l'identificatore del dato stesso.

Esempio

Il primo comando definisce una tabella di 100 variabili di tipo byte; l'identificatore LISTA punta all'indirizzo del primo byte della tabella. Nel secondo l'assemblatore scrive l'offset del primo byte della variabile LISTA nella variabile di tipo word di nome LISTOFFSET.

```

LISTA      DB      100 DUP(?)
LISTOFFSET DW      LISTA

```

6.1.5. Direttiva DD

La direttiva **DD** (*Define Doublewords*) definisce dati memorizzati su una *doubleword* (4 byte). Su una doubleword è possibile memorizzare un singolo carattere, una coppia di caratteri, un numero (intero o reale) od un indirizzo intero (registro di segmento e offset).

Utilizzo della direttiva DD per la memorizzazione di caratteri

L'assemblatore permette di memorizzare 1 o 2 caratteri su un dato di tipo DWORD; tali caratteri

sono memorizzati all'inizio della doubleword; gli altri byte assumono valore 0. Non si può utilizzare la direttiva DD per memorizzare una sequenza di più di 2 caratteri

Esempio

In Fig. 6.2 è mostrata lo stato della memoria dopo il comando seguente:

```
BICAR    DD    "AB"
```

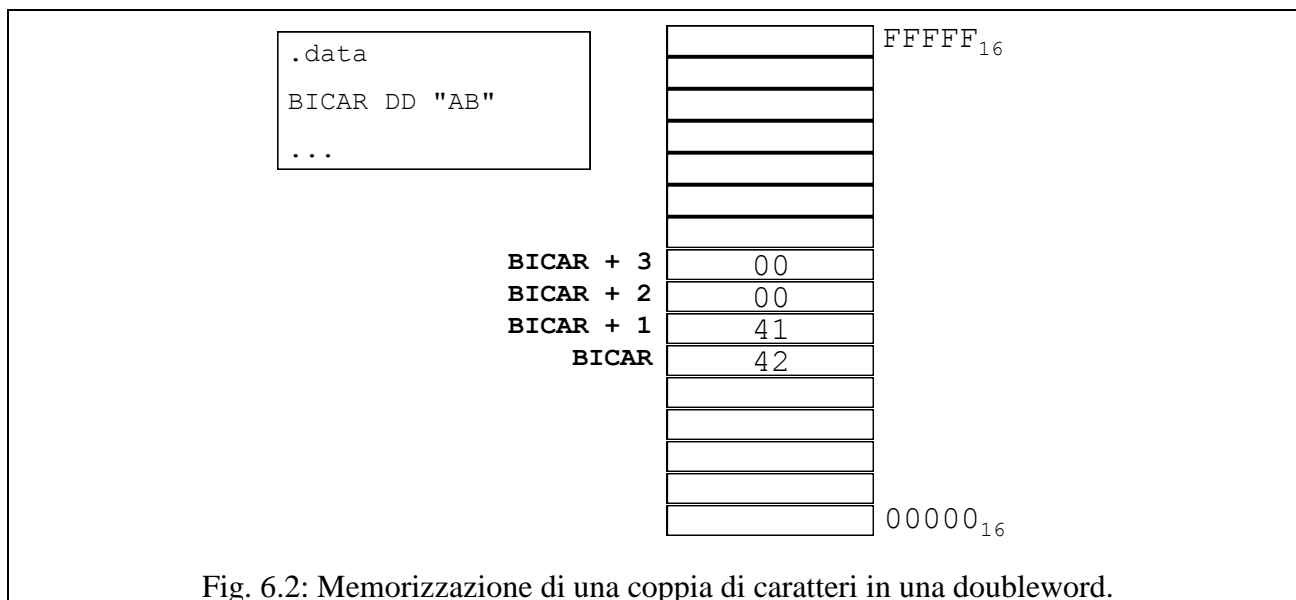


Fig. 6.2: Memorizzazione di una coppia di caratteri in una doubleword.

Utilizzo della direttiva DD per la memorizzazione di numeri

Le doubleword sono spesso utilizzate per memorizzare numeri. Senza opportune procedure aritmetiche *ad hoc*, le istruzioni aritmetiche del processore operano esclusivamente con byte o word, mentre il *coprocessore matematico* può lavorare con numeri aventi un intervallo di rappresentazione maggiore. Se si dispone di un coprocessore matematico si possono utilizzare le doubleword per memorizzare due tipi di numeri: interi con segno e numeri reali in virgola mobile (*floating point*). L'intervallo dei numeri rappresentabili su una doubleword è mostrato in Tab. 6.4.

Per definire un numero reale sono possibili due diverse notazioni: quella che prevede il punto come separatore tra parte intera e parte frazionaria, oppure la notazione scientifica in cui i numeri sono espressi nella forma $\pm M \cdot 10^{\pm E}$, dove M è chiamata *mantissa* ed E è chiamato *esponente*.

Per specificare i numeri in notazione scientifica si scrive la mantissa seguita dalla lettera E e dall'esponente.

Esempio

Il primo comando definisce una tabella di variabili doubleword inizializzata con una sequenza di numeri interi; il secondo definisce una tabella in cui le variabili sono numeri *floating point*.

```
TAB1      DD      123456789, 0F5BCDH
TAB2      DD      0, 3.141593, 3.123E10, -56.1E-12, 1E-10
```

Tipi di numeri	Intervallo
Interi	$-2^{31} \div 2^{31}-1$
Floating point	$\pm(10^{-38} \div 10^{38})$

Tab. 6.4: Intervallo di numeri validi memorizzabili su una doubleword.

Utilizzo della direttiva DD per la memorizzazione degli indirizzi

Un indirizzo è costituito dall'indirizzo di un segmento e da un offset. Per memorizzare un indirizzo intero occorre utilizzare una doubleword.

Per inizializzare una doubleword con l'indirizzo intero di un dato, bisogna prima specificare l'identificatore del dato stesso. L'assemblatore pone l'offset del dato nella prima word e l'indirizzo di segmento nella seconda word, come mostrato in Fig. 6.3.

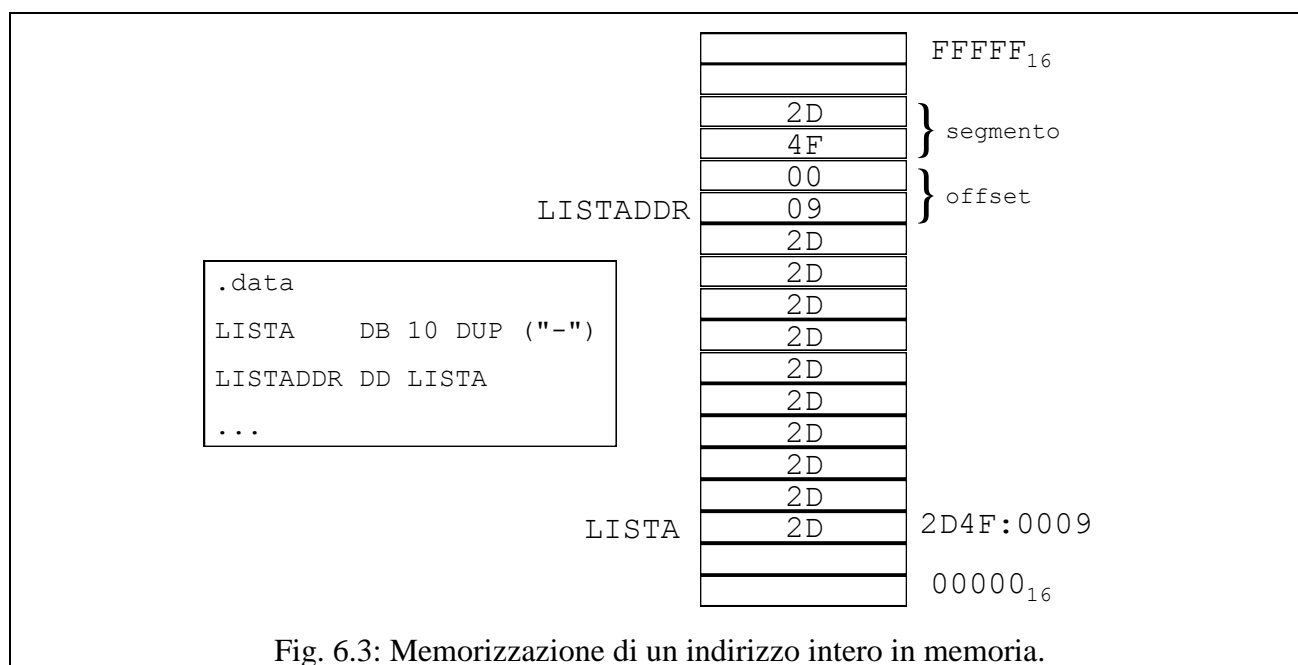


Fig. 6.3: Memorizzazione di un indirizzo intero in memoria.

Esempio

Il primo comando definisce una tabella di 10 variabili di tipo byte. L'identificatore `LISTA` punta all'indirizzo del primo byte della tabella. Nel secondo, l'assemblatore scrive l'indirizzo di segmento e l'offset del primo byte nella variabile di tipo doubleword di nome `LISTADDR`.

```
LISTA      DB      10 DUP (" ")
LISTADDR   DD      LISTA
```

6.1.6. Direttive DQ e DT

Le direttive **DQ** e **DT** permettono di definire dati costituiti rispettivamente da 8 e da 10 byte. Questi tipi di dato possono essere usati per memorizzare 1 o 2 caratteri oppure numeri.

Il comportamento dell'assemblatore quando si memorizzano caratteri su quadword e tenbyte è analogo al caso delle doubleword, ossia i caratteri da memorizzare sono posti all'inizio del dato ed i

byte rimanenti sono caricati con il valore 0.

Quadword e tenbyte sono utilizzati esclusivamente per memorizzare numeri di alta precisione per operazioni con il *coprocessore matematico*. Le quadword possono memorizzare numeri interi con segno e numeri floating point, mentre le tenbyte possono memorizzare numeri *floating point* o i numeri *decimali impaccati* (*packed decimal*).

I numeri decimali impaccati memorizzano 2 numeri decimali per ogni byte. Il primo byte serve per memorizzare il segno e gli altri 9 byte memorizzano ciascuno una coppia di cifre decimali. Il tipo di dato *tenbyte* permette dunque di memorizzare un numero decimale su 18 cifre. In Tab. 6.5 e 6.6 sono indicati i tipi di numero e l'intervallo di rappresentazione per quadword e tenbyte.

<i>Tipi di numeri</i>	<i>Intervallo</i>
Interi	$-2^{63} \div 2^{63}-1$
Floating point	$\pm(10^{-308} \div 10^{308})$

Tab. 6.5: Intervallo di numeri validi memorizzabili su una quadword.

<i>Tipi di numeri</i>	<i>Intervallo</i>
Interi impaccati	$-(10^{18}-1) \div 10^{18}-1$
Floating point	$\pm(10^{-4932} \div 10^{4932})$

Tab. 6.6: Intervallo di numeri validi memorizzabili su una parola di tipo TBYTE.

6.2. Operatori

Un *operatore* è una direttiva per l'assemblatore che permette di agire sul valore di un operando. Come tutte le pseudo-istruzioni, gli operatori sono riconosciuti dall'assemblatore e non generano alcuna istruzione macchina. L'assemblatore fornisce diversi tipi di operatori:

- operatori che calcolano gli attributi di una variabile
- operatori aritmetici
- operatori che modificano il tipo di una variabile.

6.2.1. Operatori ed attributi di una variabile

In questo paragrafo vengono descritti gli operatori che calcolano gli *attributi* delle variabili, essi sono: **TYPE**, **LENGTH**, **SIZE**, **SEG** e **OFFSET**. Il formato di tali direttive è il seguente:



OPERATORE identificatore

L'assemblatore calcola l'attributo specificato dall'operatore e sostituisce il suo valore nell'istruzione macchina generata.

- La direttiva **TYPE** applicata alla variabile di nome *identificatore* calcola il numero di byte necessari per rappresentare il singolo elemento della variabile.

- La direttiva `LENGTH` applicata alla variabile di nome *identificatore* calcola il numero di elementi che costituiscono la variabile.
- La direttiva `SIZE` applicata alla variabile di nome *identificatore* calcola il numero di byte complessivi necessari per memorizzare l'intera variabile.
- La direttiva `SEGMENT` applicata alla variabile di nome *identificatore* calcola l'indirizzo di partenza del segmento in cui risiede la variabile.
- La direttiva `OFFSET` applicata alla variabile di nome *identificatore* calcola l'offset del primo elemento della variabile.

Esempi

Sia `LISTA` una tabella di 100 word definita mediante il seguente comando:

```
LISTA      DW      100 DUP (?)
```

Si consideri la seguente istruzione:

```
MOV      AX, TYPE LISTA
```

L'assemblatore sostituisce al posto del secondo operando, il valore dell'espressione `TYPE LISTA` (in questo caso pari a 2).

Le seguenti istruzioni sono equivalenti:

```
MOV      AX, LENGTH LISTA  
MOV      AX, 100
```

Così come le seguenti istruzioni sono equivalenti:

```
MOV      AX, SIZE LISTA  
MOV      AX, 200
```

Per copiare l'indirizzo di segmento di `LISTA` nel registro `AX` si può usare la seguente istruzione:

```
MOV      AX, SEG LISTA
```

Per copiare l'offset di `LISTA` nel registro `AX` si può usare la seguente istruzione:

```
MOV      AX, OFFSET LISTA
```

L'operatore `LENGTH` funziona esclusivamente con variabili che sono state allocate utilizzando la direttiva `DUP`. Se la variabile è stata allocata in maniera diversa, l'operatore `LENGTH` restituisce il valore 1. Tale comportamento deve essere tenuto in conto attentamente per non commettere errori di programmazione.

Esempio

Dati i seguenti due comandi di allocazione:

```
TAB1      DB      90, 2, 0, 10 DUP (0), 20, 56  
TAB2      DB      100 DUP (0), 12, 0, 14
```


dopo l'esecuzione delle due istruzioni seguenti il registro AX conterrà il valore 1, mentre il registro BX conterrà il valore 100.

```
MOV    AX, LENGTH TAB1
MOV    BX, LENGTH TAB2
```

6.2.2. Operatori aritmetici

A livello di codice sorgente è possibile esprimere un operando sotto forma di espressione aritmetica contenente gli operatori aritmetici presentati in Tab. 6.7. L'utilizzo degli operatori aritmetici rende il programma più leggibile. Tali espressioni sono calcolate dall'assemblatore e vengono convertite in operando durante la fase di generazione delle istruzioni macchina. Queste operazioni non sono dunque eseguite durante l'esecuzione del programma, che non viene così rallentata.

Esempi

I seguenti comandi utilizzano operatori aritmetici:

```
TABLE    DB    2*1024 DUP(?)
COUNT1  DB    97/10
COUNT2  DB    97 MOD 10
MOV       AH, TABLE+2*(4+5)
```

<i>Opera- tore</i>	<i>Significato</i>
+	somma
-	sottrazione
*	moltiplica- zione
/	divisione
MOD	modulo

Tab. 6.7: Gli operatori aritmetici.

Le regole utilizzate dall'assemblatore per calcolare le espressioni sono quelle classiche dell'aritmetica.

Esistono due operatori di divisione; entrambi lavorano su numeri interi, ma differiscono in quanto l'operatore / dà come quoziente un numero intero scartando il resto, mentre l'operatore **MOD** fornisce come risultato il resto della divisione.

Esempio

Se si vuole definire una variabile `BUFFER` costituita da 12 Kbyte si possono scegliere le due alternative:

```
BUFFER    DB    12288 DUP(?)
BUFFER    DB    12*1024 DUP(?)
```

La seconda soluzione ha una comprensibilità maggiore rispetto alla prima.

È possibile scrivere un'espressione contenente sia operatori aritmetici sia operatori di attributo.

Esempio

La variabile di nome `BIGLIST` occupa uno spazio in memoria doppio rispetto alla variabile `LIST`.

```
LIST      DB      120 DUP(?)
BIGLIST   DB      2*(SIZE LIST) DUP(?)
```

6.2.3. Operatore PTR

Quando si fa riferimento ad un dato, l'assemblatore verifica il tipo e controlla se è compatibile con l'istruzione che deve essere eseguita.

Esempi

Si voglia eseguire un'istruzione che copia il valore della variabile `VAL` nel registro `AH`.

```
MOV      AH, VAL      ; copia il contenuto di VAL in AH
```

Se la variabile `VAL` è di tipo `BYTE` l'istruzione è assemblata regolarmente, altrimenti l'assemblatore segnala un errore di sintassi.

L'assemblatore tuttavia permette di far riferimento ad un dato in maniera non conforme alla sua definizione di tipo. Per fare questa operazione si utilizza l'operatore **PTR**, la cui sintassi è la seguente:



tipo *PTR* *nome*

L'operatore `PTR` forza l'assemblatore a trattare l'identificatore *nome* come se avesse il tipo specificato dal campo *tipo*.

Esempi

Data la seguente allocazione di variabile:

```
TOTALE    DW      ?
```

le istruzioni seguenti permettono di spezzare la variabile in 2 byte: la prima copia il primo byte della word `TOTALE` nel registro `BH`; la seconda copia il secondo byte della word `TOTALE` nel registro `CH`.

```
MOV      BH, BYTE PTR TOTALE
MOV      CH, (BYTE PTR TOTALE)+1
```

Data la seguente definizione di variabile:

```
COPPIA    DB      ?, ?
```

si vuole copiare i due byte di `COPPIA` in `AX`, il comando seguente viene considerato non valido dall'assemblatore.



```
MOV    AX, COPPIA
```

In alternativa si può utilizzare l'operatore PTR per modificare il tipo della variabile COPPIA:



```
MOV    AX, WORD PTR COPPIA
```

In questo caso in AX viene scritto il contenuto dei due byte corrispondenti agli indirizzi COPPIA e COPPIA+1.

L'utilizzo dell'operatore PTR deve essere effettuato con estrema cautela e possibilmente evitato, poiché può essere causa di errori.

Esempio

Esiste una sostanziale differenza nel modo di memorizzare la coppia di caratteri nelle due seguenti direttive.

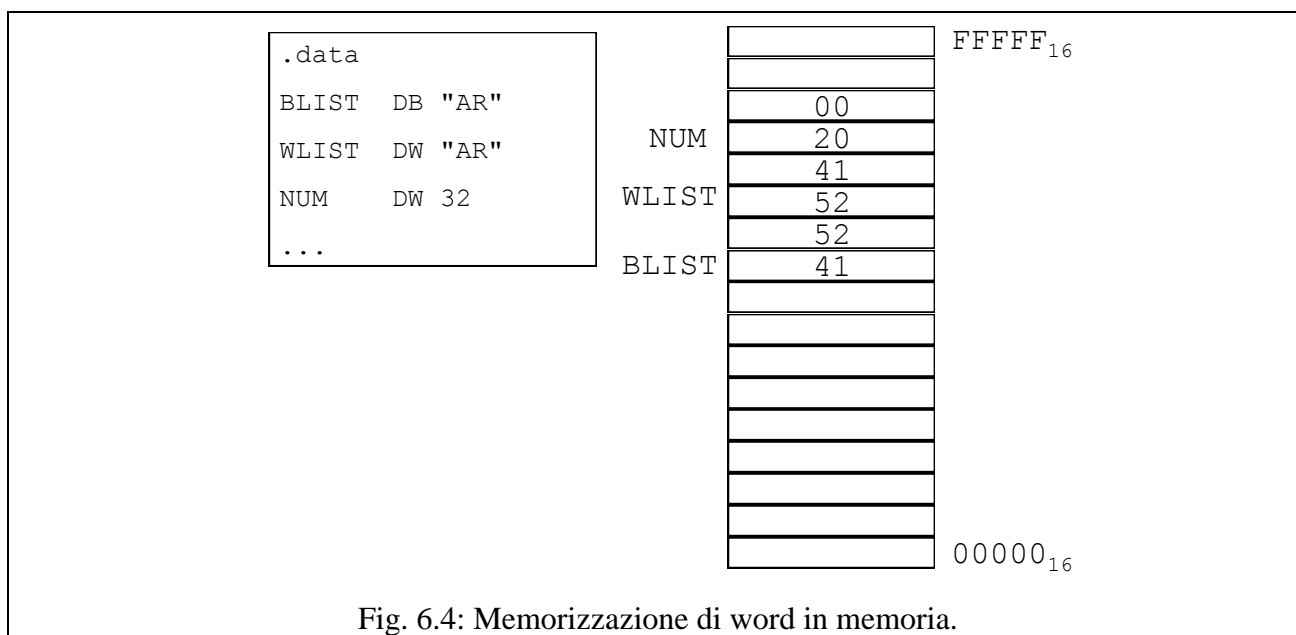
```
BLIST    DB    "AR"
WLIST    DW    "AR"
```

La stringa BLIST è memorizzata in byte contigui (prima 'A' e poi 'R'). Nel memorizzare una coppia di caratteri su una word l'assemblatore inverte l'ordine; memorizza nel primo byte il carattere 'R' e nel secondo il carattere 'A'; analogamente per i numeri interi, l'assemblatore memorizza prima il byte meno significativo e poi il byte più significativo (Fig. 6.4).

Questa ambiguità non dà problemi, perché il processore sa che quando deve fare il *fetch* di una word deve invertire l'ordine dei byte. Quando si utilizza l'operatore PTR per spezzare la word in più byte, l'assemblatore copia i byte esattamente come viene specificato dalla direttiva.

Similmente quando si accede a 2 byte convertendoli in word, il processore li tratta alla stessa stregua di una word e dunque li copia nel registro a 16 bit in ordine inverso. Questo può dunque essere fonte di errore.

È spesso possibile specificare la stessa operazione senza usare l'operatore PTR.



Esempio

Le due seguenti istruzioni copiano i 2 byte della variabile `COPPIA` nel registro `AX`:

```
MOV    AL, COPPIA
MOV    AH, COPPIA+1
```

In alcuni casi particolari l'utilizzo dell'operatore `PTR` è inevitabile; un tipico esempio si ha quando l'assemblatore non è in grado di distinguere se il dato è di tipo byte oppure word.

Esempio

L'istruzione seguente genera un errore a livello di assemblatore:



```
INC    [BX]
```

L'assemblatore non può sapere se la locazione di memoria indirizzata dal registro `BX` corrisponda ad una word o ad un byte. È necessario dunque utilizzare l'operatore `PTR` per specificare il tipo dell'operando.



```
INC    BYTE PTR [BX]
```

In questo modo l'istruzione incrementa il byte indirizzato dal registro `BX`.

6.2.4. Direttiva LABEL

La direttiva **LABEL** permette di definire un nome alternativo (*etichetta*) per far riferimento ad una particolare locazione all'interno di un programma; il suo formato è il seguente:



```
nome LABEL tipo
```

Tale direttiva permette di definire un *tipo* da associare all'identificatore *nome*. L'identificatore *nome* può fare riferimento o ad una variabile o ad una istruzione del segmento di codice (istruzione macchina o procedura). Se l'etichetta è associata ad un dato, il tipo può essere `BYTE` o `WORD` o `DWORD`; se l'etichetta è associata ad un'istruzione all'interno del segmento di codice il *tipo* dell'etichetta può essere o **NEAR** o **FAR**.

Esempio

Il primo comando definisce un'etichetta di nome `BNAME` abbinata ad un dato di tipo byte, il secondo definisce una variabile di tipo word:

```
BNAME LABEL BYTE
WNAME DW 100 DUP(?)
```

La direttiva `LABEL` non occupa spazio in memoria; dunque `BNAME` e `WNAME` hanno lo stesso offset. Per accedere alla variabile trattata come insieme di byte si può utilizzare l'identificatore `BNAME`, altrimenti si può fare riferimento all'identificatore `WNAME` per accedere alla variabile come insieme di word. In questo modo lo stesso dato ha due nomi e due tipi diversi.

La seguente istruzione copia il contenuto della seconda word di `WNAME` nel registro `AX`:

```
MOV    AX, WNAME+2
```

La seguente istruzione copia il terzo byte della variabile `BNAME` nel registro `AH`:

```
MOV    AH, BNAME+2
```

La precedente istruzione è equivalente alla seguente:

```
MOV    AH, BYTE PTR WNAME+2
```

Attraverso le *etichette* è possibile controllare il flusso di esecuzione del programma eseguendo salti a determinate locazioni.

Esempi

I seguenti comandi definiscono etichette:

```
ciclo LABEL NEAR
entry LABEL FAR
```

Una label di tipo `NEAR` corrisponde ad un indirizzo di codice cui si può far riferimento solo dall'interno dello stesso segmento, mentre una label di tipo `FAR` corrisponde ad un indirizzo cui si può far riferimento anche da un altro segmento.

La maggior parte delle etichette sono di tipo `NEAR`, l'assemblatore permette di definire etichette di tipo *near* in una forma semplificata:



```
nome:
```

L'assemblatore tratta l'identificatore *nome* come una etichetta di tipo `NEAR`.

Esempio

Il comando seguente definisce l'etichetta `ciclo`, corrispondente all'istruzione `MOV`:

```
ciclo:    MOV    AL, [SI]
```

6.3. Direttive per le definizioni di costanti

L'assemblatore fornisce due direttive per la definizione di costanti:

- la direttiva **EQU**
- la direttiva **=**.

6.3.1. La direttiva EQU

La direttiva `EQU` permette di definire simboli utili al programmatore: essi vengono sostituiti da specifici valori all'atto dell'assemblaggio. È bene sottolineare che la direttiva `EQU` non genera nessuna istruzione macchina, né alloca spazio in memoria, ma agisce esclusivamente sull'assemblatore.

Il formato della direttiva `EQU` è il seguente:



```
nome EQU espressione
```

L'assemblatore associa il valore dell'*espressione* all'identificatore *nome*; ogni volta che

l'assemblatore incontra l'identificatore *nome*, lo sostituisce con il valore dell'*espressione*.

Esempio

La seguente direttiva dice all'assemblatore di considerare il simbolo *K* equivalente al numero decimale 1024.

K	EQU	1024
----------	------------	-------------

La direttiva EQU è utile quando si usano ripetutamente gli stessi valori all'interno del codice sorgente.

Esempio

I comandi seguenti utilizzano il simbolo definito dalla direttiva EQU dell'esempio precedente:

ITEM1	DB	K DUP (?)
ITEM2	DB	2*K DUP (?)
ITEM3	DB	8*K DUP (?)

Un uso sistematico della direttiva EQU permette di incrementare notevolmente la leggibilità e la manutenibilità di un programma Assembler.

Esempio

Data la seguente definizione di costante simbolica:

LSIZE	EQU	100
--------------	------------	------------

All'interno del programma si possono fare diversi riferimenti al simbolo LSIZE:

WNAME	DB	LSIZE DUP (?)
WLIST	DW	LSIZE DUP (?)
	MOV	AX, LSIZE
	MOV	BH, LSIZE

L'assemblatore sostituisce il valore 100 tutte le volte che incontra il simbolo LSIZE. Nel caso in cui si volesse fare una modifica al codice e cambiare LSIZE da 100 in 200, tutto quello che si deve fare è modificare il comando EQU e riassemblare il programma.

Anche se la posizione delle direttive EQU è influente ai fini del programma eseguibile, è buona norma raggruppare tutte le direttive EQU all'inizio del file sorgente.

6.3.2. La direttiva =

La direttiva = permette di definire e modificare delle costanti simboliche, il formato di tale direttiva è il seguente:



nome = *espressione*

La direttiva = è analoga alla direttiva EQU: attraverso la direttiva = è però possibile ridefinire il valore del simbolo *nome* più di una volta all'interno del programma.

Esempio

I seguenti comandi sono un esempio di utilizzo della direttiva `=`.

```

EMP      =      6          ; equivalente a EMP EQU 6
EMP      ...
EMP      =      7          ; il simbolo EMP può essere ridefinito
EMP      ...
EMP      =      EMP + 1    ; seconda ridefinizione di EMP

```

6.4. Il *location counter*

L'assemblatore mantiene un valore chiamato *location counter* che contiene l'offset della prossima locazione di memoria disponibile. Tale valore è utilizzato dall'assemblatore durante la fase di compilazione del codice sorgente.

Quando l'assemblatore inizia l'analisi di un programma inizializza a 0 il valore del *location counter*, e provvede ad incrementarlo opportunamente ogni volta che deve riservare un'area di memoria per dati od istruzioni.

Esempio

Supponiamo che il seguente comando sia il primo di un programma:

```
DB      32 DUP ("STACK---")
```

L'assemblatore riserva $32 * 8 = 256$ byte di memoria per i dati. Il *location counter* dopo questo comando vale dunque 100H (equivalente al numero decimale 256).

All'interno di un programma si può far riferimento al valore corrente del *location counter* mediante il carattere speciale `$`.

Esempio

```

SALVE    DB      "Ciao"
L_SALVE  EQU     $-SALVE

```

L'assemblatore quando calcola l'espressione per la EQU sottrae il *location counter* corrente con l'offset della locazione di memoria di inizio della stringa SALVE; il risultato è dunque pari alla lunghezza della stringa. Cambiando il contenuto della stringa e ri assemblando il programma, l'assemblatore aggiorna automaticamente il valore del simbolo `L_SALVE`.

È possibile forzare il valore del *location counter* utilizzando la direttiva **ORG**. Il formato di tale direttiva è il seguente:



```
ORG      espressione
```

Quando l'assemblatore incontra la direttiva **ORG** calcola l'espressione ed aggiorna il valore del *location counter* al nuovo valore.

Esempio

La direttiva `ORG` è usata quando si vuole generare un *command file* (`.COM`). Per generare questo tipo di modulo eseguibile occorre riservare uno spazio di 256 byte all'inizio del programma. riservata al DOS, scrivendo all'inizio del programma il seguente comando:

```
ORG      100H
```

In questo modo quando l'assemblatore inizia l'analisi del codice sorgente modifica il *location counter* e riserva 256 byte liberi per il DOS.

6.5. Le macro

Una *macro* è costituita da un gruppo di istruzioni che esegue uno specifico compito. L'uso della macro si compone di due fasi:

1. la *definizione* della macro, ossia la specificazione delle istruzioni che costituiscono la macro;
2. la *chiamata* della macro all'interno del programma.

Il formato per la definizione di una macro è il seguente:



```
nome      MACRO {param1, param2, ... paramn}
           ...
           ENDM
```

Il campo *nome* definisce il nome della *MACRO*; ogni macro può lavorare su più parametri (*param1 ... paramn*) specificati nella linea di definizione della macro. Ogni macro è terminata dalla direttiva `ENDM`. L'assemblatore, in un passo di pre-compilazione, espande il blocco di istruzioni compreso tra la direttiva `MACRO` e la direttiva `ENDM` ogni volta che incontra, all'interno del codice sorgente, il simbolo *nome*.

Esempio

Si vuole definire una macro che esegua la copia tra due variabili contenute in memoria. Tale operazione non può essere effettuata con una sola istruzione, ma occorre utilizzare un registro temporaneo. Il seguente frammento di codice esegue la copia da un vettore sorgente ad un vettore destinazione.

```

LUNG    ...
        EQU    100
MUOVI   MACRO  V1, V2      ; V1 e V2 sono due parametri della macro
        PUSH   AX          ; salva nello stack il contenuto di AX
        MOV    AX, V2
        MOV    V1, AX
        POP    AX          ; ripristina il contenuto di AX
        ENDM
        .DATA
SORG     DW     LUNG DUP (?)
DEST     DW     LUNG DUP (?)
        ...
        .CODE
        ...
        MOV    CX, LUNG
        LEA    SI, SORG
        LEA    DI, DEST
ciclo:   MUOVI  [DI], [SI]
        ADD    SI, 2
        ADD    DI, 2
        LOOP   ciclo
        ...

```

L'assemblatore, dopo il passo di pre-compilazione, espande il frammento di codice precedente nel seguente:

```

LUNG     ...
        EQU    100
        .DATA
SORG     DW     LUNG DUP (?)
DEST     DW     LUNG DUP (?)
        ...
        .CODE
        ...
        MOV    CX, LUNG
        LEA    SI, SORG
        LEA    DI, DEST
ciclo:   PUSH   AX          ; salva nello stack il contenuto di AX
        MOV    AX, [SI]
        MOV    [DI], AX
        POP    AX          ; ripristina il contenuto di AX
        ADD    SI, 2
        ADD    DI, 2
        LOOP   ciclo
        ...

```

Esiste una stretta analogia tra le macro e le procedure. La differenza tra una macro ed una procedura sta nel fatto che la chiamata di una procedura ed il ritorno da essa sono eseguiti attraverso opportune istruzioni macchina, che dunque hanno un costo in termini di tempo di elaborazione. Tale costo viene invece evitato utilizzando le macro; si noti infatti che all'atto dell'esecuzione, di esse non resta traccia nel codice macchina, avendo l'assemblatore provveduto alla loro sostituzione con il codice corrispondente.

D'altro canto, usando le macro il codice eseguibile occupa un maggiore spazio in memoria poiché l'assemblatore espande lo stesso codice un numero di volte pari a quante volte compare il nome

delle macro nel codice sorgente.

6.5.1. Variabili locali all'interno di una macro

Se all'interno del codice di una macro compare un'etichetta, questa apparirà nel codice generato dall'assemblatore, dopo la fase di espansione, tante volte quante è stata richiamata la macro stessa, causando quindi un errore di assemblaggio. Per evitare questo problema le etichette che compaiono all'interno delle macro devono essere definite 'locali'. Una variabile locale è dunque un'etichetta definita ed utilizzabile all'interno di una macro, ma non disponibile al suo esterno. Per definire una variabile locale si usa la direttiva **LOCAL**, il cui formato è il seguente:



```
LOCAL    var1 {, var2 ...}
```

La direttiva **LOCAL** permette di definire identificatori locali alla macro. L'assemblatore sostituisce alla variabile locale un identificatore diverso per ogni chiamata della macro.

Esempio

Il seguente frammento di programma esegue la ripetizione della lettura di un carattere da tastiera fino a quando non venga letto il carattere 'Y'. Per la lettura di un carattere da tastiera è definita una macro. Si utilizza la *function call* di Sistema Operativo MS-DOS di codice 06H. Se un carattere è stato letto dallo *Standard Input* è caricato in AL ed il flag ZF è azzerato, altrimenti se nessun carattere è letto il flag ZF è forzato ad 1 (tale funzionalità si ha caricando il registro DL con il valore 0FFH).

```

LEGGI    ...
         MACRO    CAR
         LOCAL    read
         PUSH     AX
         PUSH     DX
read:    MOV      AH, 06H
         MOV      DL, 0FFH
         INT      21H
         JE       read
         MOV      CAR, AL
         POP      DX
         POP      AX
         ENDM
         .DATA
CAR      DB       ?
         ...
         .CODE
         ...
ciclo:   LEGGI    CAR
         CMP      CAR, 'Y'
         JNE      CICLO
         ...

```

L'assemblatore, dopo il passo di pre-compilazione, espande il precedente frammento di codice sostituendo l'etichetta locale `read` con l'identificatore `??0000`; alla successiva chiamata della stessa macro, `read` viene sostituita dall'identificatore `??0001` e così via.

```

...
.DATA
CAR    DB      ?
...
.CODE
...
ciclo: PUSH    AX
        PUSH    DX
??0000: MOV     AH, 06H
        MOV     DL, 0FFH
        INT     21H
        JE      ??0000
        MOV     CAR, AL
        POP     DX
        POP     AX
        CMP     CAR, 'Y'
        JNE     CICLO
...

```

6.6. Le direttive **SEGMENT** e **ENDS**

Le direttive **SEGMENT** ed **ENDS** permettono di definire un segmento, segnalandone rispettivamente l'inizio e la fine. La sintassi di definizione di un segmento è la seguente:



```

nome SEGMENT {align} {READONLY} {combine} {'class'}
...
nome ENDS

```

Il campo *nome* definisce il nome del segmento: all'interno di un *file* tutte le definizioni di segmento con lo stesso nome fanno riferimento allo stesso segmento fisico.

Il campo opzionale *align* definisce il tipo di allineamento con cui inizia il segmento selezionato. I possibili tipi di allineamento sono illustrati in Tab. 6.8.

Campo <i>align</i>	Indirizzo di partenza
BYTE	Il prossimo indirizzo di byte disponibile.
WORD	Il prossimo indirizzo di word disponibile.
DWORD	Il prossimo indirizzo di doubleword disponibile.
PARA	<i>DEFAULT</i> : il prossimo indirizzo di paragrafo disponibile (un paragrafo ha dimensione 16 byte).
PAGE	Il prossimo indirizzo di pagina disponibile (una pagina ha dimensione 256 byte).

Tab. 6.8: Tipi di allineamento per un segmento.

Il *linker* utilizza l'informazione sull'allineamento per determinare l'indirizzo relativo di partenza di ciascun segmento. Il *loader* del Sistema Operativo calcola l'indirizzo di partenza di ciascun segmento all'atto del caricamento del programma.

L'attributo opzionale *READONLY* protegge contro l'illegale modifica del segmento. In seguito alla presenza di tale attributo l'assemblatore controlla che non vengano generate istruzioni che modifichino il segmento: in tal caso viene generato un errore da parte dell'assemblatore. Si noti che tale controllo non elimina del tutto la possibilità che un'istruzione modifichi uno o più byte del segmen-

to, ad esempio attraverso un indirizzamento indiretto.

Il campo *combine* serve per definire come il *linker* deve combinare i segmenti che hanno lo stesso nome, ma che appaiono in *file* differenti.

I possibili tipi di combinazione tra segmenti sono mostrati in Tab. 6.9.

<i>Campo combine</i>	<i>Azione del linker</i>
PRIVATE	<i>DEFAULT</i> : non combina i segmenti di moduli differenti.
PUBLIC o MEMORY	Concatena tutti i segmenti aventi lo stesso nome per formare un unico segmento contiguo.
STACK	Concatena tutti i segmenti aventi lo stesso nome per formare un unico segmento di stack.
COMMON	Esegue la sovrapposizione dei segmenti: la lunghezza dell'area risultante è pari alla lunghezza del più grande dei segmenti combinati.
AT <i>exp</i>	Assume il paragrafo <i>exp</i> come locazione del segmento. Un segmento AT non può contenere codice o dati inizializzati, ma può essere utile per definire variabili che corrispondono a specifiche locazioni di memoria.

Tab. 6.9: Tipi di concatenamento tra segmenti su moduli diversi.

Definire variabili inizializzate in segmenti di tipo **COMMON** può provocare errori, perché con questi tipi di combinazione, il *linker* sovrappone i dati per ogni modulo, e dunque l'ultimo modulo che contiene dati inizializzati scrive sui dati degli altri moduli.

Per ciascun programma è necessario specificare almeno un segmento di stack (avente il tipo di combinazione **STACK**); diversamente il *linker* segnala un errore.

Il campo opzionale *class* permette un'ulteriore distinzione tra segmenti: due segmenti con lo stesso nome non sono combinati se il campo *class* è diverso.

Esempio:

I seguenti comandi definiscono 4 diversi segmenti:

```

CODE      SEGMENT   WORD PUBLIC 'CODE'
...
CODE      ENDS
...
DATA      SEGMENT   WORD PUBLIC 'DATA'
...
DATA      ENDS
CONST     SEGMENT   WORD PUBLIC 'CONST'
...
CONST     ENDS
...
STACK     SEGMENT   PARA STACK 'STACK'
...
STACK     ENDS

```

6.7. La direttiva GROUP

La direttiva **GROUP** permette di raggruppare più segmenti logici in un unico segmento fisico.

La sintassi della direttiva **GROUP** è la seguente:



```
nome GROUP nome_seg1 {, ...}
```

La direttiva **GROUP** raggruppa in un unico segmento di nome *nome* tutti i segmenti di nome *se-gname*. Il campo *nome* non deve coincidere con nessuno dei campi *nome_seg*. Il campo *nome_seg* può essere un nome di segmento assegnato dalla direttiva **SEGMENT**.

Esempio

Il frammento di codice Assembler seguente mostra la definizione di un gruppo di segmenti di nome **CGROUP** costituito dal raggruppamento dei segmenti di nome **XXX**, **YYY** e **ZZZ**:

```
CGROUP    GROUP    XXX,YYY    ; gruppo di segmenti di nome CGROUP
                                ; costituito dai segmenti XXX e YYY
XXX       SEGMENT
...
XXX       ENDS
YYY       SEGMENT
...
YYY       ENDS
...
CGROUP    GROUP    ZZZ        ; segmento ZZZ aggiunto al gruppo CGROUP
ZZZ       SEGMENT
...
ZZZ       ENDS
```

6.8. La direttiva **ASSUME**

La direttiva **ASSUME** serve per associare un registro di segmento ad un segmento di memoria. La sintassi della direttiva **ASSUME** è la seguente:



```
ASSUME segreg:segloc {, ...}
```

La direttiva **ASSUME** associa il registro di segmento *segreg* al segmento *segloc*. I registri *segreg* possono essere **CS**, **DS**, **ES**, **SS**. Il campo *segloc* può essere uno dei seguenti:

- il nome di un segmento definito attraverso la direttiva **SEGMENT**;
- il nome di un gruppo di segmenti definito attraverso la direttiva **GROUP**;
- le parole chiave **NOTHING** o **ERROR**;

La parola chiave **NOTHING** cancella l'assunzione corrente di segmento per il registro *segreg*.

La parola chiave **ERROR** disabilita l'uso del segmento *segreg*.

Esempio

Il comando seguente associa al gruppo di segmenti di nome **CGROUP** il registro **CS**, al segmento di nome **DATA** il registro **DS**, al segmento di nome **STACK** il registro **SS** e disabilita l'uso del registro **ES**.

```
ASSUME    CS:CGROUP, DS: DATA, SS: STACK, ES: ERROR
```

Quando l'assemblatore deve processare un'istruzione in cui si fa riferimento ad una variabile in memoria, determina in quale segmento essa è stata definita e, grazie alla direttiva **ASSUME**, ricava il registro di segmento da utilizzare. La direttiva **ASSUME** mette in relazione un registro di segmento con un segmento logico, ma non si occupa di inizializzare il registro stesso.

Ad esempio il Sistema Operativo **MS-DOS**, all'atto dell'esecuzione di un programma, carica il registro **CS** in modo da puntare al primo byte del segmento di codice ed il registro **SS** in modo da

puntare alla fine dello *stack*. I registri DS ed ES devono essere caricati da opportune istruzioni all'inizio del programma prima di ogni riferimento ai nomi interni ai segmenti ad essi associati.

Esempi

Il programma seguente esegue la copia di un blocco di dati da un'area di memoria sorgente (SOURCE) ad un'area destinazione (DEST). Il programma utilizza un segmento di stack di nome STACK, un segmento di dato di nome DSEG ed un segmento di codice di nome CSEG. Si può notare come le prime istruzioni eseguite dal programma consistono nel caricamento del registro DS.

```

STACK      SEGMENT  PARA STACK 'STACK'
            DB        64 DUP ('STACK ')
STACK      ENDS

DSEG       SEGMENT  PARA PUBLIC 'DATA'
SOURCE     DB        100, 40, 33, 39
DEST       DB        4 DUP (?)
DSEG       ENDS

CSEG       SEGMENT  PARA PUBLIC 'CODE'
            ASSUME    CS:CSEG, DS:DSEG, SS:STACK
BEGIN:     MOV        AX, DSEG      ; copia dell'indirizzo del segmento DSEG
            MOV        DS, AX       ; nel registro DS (passando attraverso AX)
            MOV        AL, SOURCE   ; copia da SOURCE
            MOV        DEST, AL     ; a DEST
            MOV        AL, SOURCE+1 ; copia da SOURCE+1
            MOV        DEST+1, AL   ; a DEST+1
            MOV        AL, SOURCE+2 ; copia da SOURCE+2
            MOV        DEST+2, AL   ; a DEST+2
            MOV        AL, SOURCE+3 ; copia da SOURCE+3
            MOV        DEST+3, AL   ; a DEST+3
            MOV        AH, 04CH     ; ritorno a DOS
            INT        21H
CSEG       ENDS          ; fine segmento di codice
            END        BEGIN       ; fine file sorgente

```

Il seguente programma esegue la copia della variabile VAR1, contenuta nel segmento DSEG, nella variabile VAR2, contenuta nel segmento ESEG. Le direttive servono all'assemblatore per risolvere correttamente il riferimento alle due variabili. Un'eventuale omissione di una delle direttive ASSUME comporta un errore a livello di assemblatore.

```

DSEG      SEGMENT  PARA PUBLIC 'DATA'
VAR1      DB      0
DSEG      ENDS
ESEG      SEGMENT  PARA PUBLIC 'DATA'
VAR2      DB      0
ESEG      ENDS
CSEG      SEGMENT  PARA PUBLIC 'CODE'
ASSUME    CS:CSEG,SS:STACK
BEGIN:    MOV      AX,DSEG
          MOV      DS,AX
          ASSUME    DS:DSEG      ; associa al registro DS il segmento DSEG
          MOV      AX, ESEG
          MOV      ES, AX
          ASSUME    ES:ESEG      ; associa al registro ES il segmento ESEG
          MOV      AL, VAR1      ; la variabile VAR1 è nel segmento DSEG
          MOV      VAR2, AL      ; la variabile VAR2 è nel segmento ESEG
          CSEG      ENDS
STACK     SEGMENT  PARA STACK 'STACK'
          DB      64 DUP("STACK ")
STACK     ENDS
          END      BEGIN

```

6.9. La direttiva END

La direttiva **END** permette di concludere un modulo di programma; il suo formato è il seguente:



END {etichetta}

Il campo *etichetta* definisce il punto di partenza del programma. La direttiva **END** ha dunque un duplice scopo: indica la fine del programma e contemporaneamente specifica all'assemblatore il punto di partenza del programma.

Il campo *etichetta* è opzionale, e non deve comparire al termine dei moduli che non contengono il programma principale.

Esempio

Il frammento di codice seguente presenta un esempio di utilizzo della direttiva **END**:

```

CSEG      ...
ASSUME    SEGMENT  PARA PUBLIC 'CODE'
          CS:CSEG
start:    ...
          END      start      ; inizio dell'esecuzione del programma
          ; fine del modulo

```

6.10. Restituzione del controllo al Sistema Operativo MS-DOS

Esistono diversi modi per restituire il controllo al Sistema Operativo al termine dell'esecuzione di un programma. Si consiglia di utilizzare una delle funzioni di sistema messe a disposizione dal Sistema Operativo stesso. In particolare la *function call* di codice 4CH permette di eseguire la ter-

minazione di un processo con la possibilità di restituire un valore di ritorno al Sistema Operativo attraverso il registro AL.

Esempio

Il seguente frammento di codice mostra le istruzioni necessarie per eseguire il ritorno al Sistema Operativo.

```
MOV    AL, 0          ; caricamento in AL del valore di ritorno
MOV    AX, 04CH
INT    21H
```

6.11. Scheletro di un programma (I versione)

Il seguente frammento di codice mostra un esempio di scheletro di programma in linguaggio Assembler.

```
STACK    SEGMENT      PARA STACK 'STACK'      ; segmento di stack
DB       32 DUP        ("STACK---")
STACK    ENDS
DATA     SEGMENT      WORD PUBLIC 'DATA'      ; segmento di dato
...                                             ; dati
DATA     ENDS
CSEG     SEGMENT      PARA PUBLIC 'CODE'      ; segmento di codice
ASSUME   CS:CSEG, DS:DATA, SS:STACK
start:
MOV      AX, DATA
MOV      DS, AX
...
MOV      AL, 0
MOV      AH, 04CH
INT      21H
CSEG     ENDS
END      start                                ; fine del modulo
```

6.12. Direttive semplificate per la gestione dei segmenti

Le ultime versioni degli assembler hanno introdotto speciali direttive che permettono al programmatore di costruire programmi e procedure in maniera semplice e veloce. Queste direttive si preoccupano di svolgere alcune mansioni che altrimenti sarebbero state a carico del programmatore. Esse non fanno parte del linguaggio Assembler, ma sono strettamente legate allo specifico assembler usato.

Alcune di queste direttive fanno sì che l'assemblatore generi al loro posto una serie opportuna di istruzioni; in questo caso tali direttive sono assimilabili a macro predefinite.

6.12.1. I modelli di memoria

La direttiva **.MODEL** definisce alcuni attributi che condizionano l'intero modulo ed è necessaria per abilitare la gestione semplificata dei segmenti. Deve essere posta nel file sorgente prima di ogni altra direttiva di segmento.

La direttiva `.MODEL` specifica il modello di memoria che si vuole usare, ed in particolare informa l'assemblatore sulla dimensione dei segmenti di codice e di dato. La sua sintassi è la seguente:



```
.MODEL modello {, opzione ...}
```

In Tab. 6.10 sono mostrate le parole chiave che possono essere utilizzate nei campi della direttiva `.MODEL`.

Il MASM supporta i modelli di memoria standard usati dalla *Microsoft* per i linguaggi di alto livello. Il modello di memoria scelto determina se i segmenti di codice o di dato occupano un unico segmento (tipo `NEAR`) oppure segmenti multipli (tipo `FAR`). In Tab. 6.11 sono schematizzate le caratteristiche dei diversi modelli di memoria.

<i>Campo</i>	<i>Parole chiave</i>	
<i>modello</i>	TINY, SMALL, COMPACT, MEDIUM, LARGE, HUGE	
<i>opzione</i>	linguaggio	C, BASIC, FORTRAN, PASCAL, STDCALL, SYSCALL
	stack	NEARSTACK (default), FARSTACK

Tab. 6.10: Parole chiave per la direttiva `.MODEL`.

<i>Modello di Memoria</i>	<i>Codice</i>	<i>Dati</i>	<i>Dati e Codice Combinati</i>
TINY	NEAR	NEAR	SI
SMALL	NEAR	NEAR	NO
MEDIUM	FAR	NEAR	NO
COMPACT	NEAR	FAR	NO
LARGE o HUGE	FAR	FAR	NO

Tab. 6. 11: Modelli di memoria.

Tutti i modelli di memoria sono validi per programmi per MS-DOS, tranne il modello `TINY` che è valido solo in ambiente MS-DOS. Il modello `TINY` usa un solo segmento per il codice e per i dati, e dunque le dimensioni dell'intero programma eseguibile non possono essere maggiori di 64 Kbyte.

Opzioni sul linguaggio

Attraverso la direttiva `.MODEL` è possibile garantire la compatibilità del codice Assembler con quello generato dai compilatori per i linguaggi di alto livello; in particolare essa forza l'assemblatore ad utilizzare la stessa convenzione per i nomi di simboli pubblici ed esterni e per l'ordine con cui gli argomenti sono passati ad una procedura.

Esempio

La direttiva seguente predispone un modello di memoria di tipo `LARGE` e utilizza le stesse convenzioni del linguaggio C per i nomi degli identificatori e per il passaggio dei parametri:

```
.MODEL    large, c
```

Opzioni sullo stack

Attraverso la direttiva **.MODEL** è possibile specificare se lo stack è posizionato in un gruppo di segmenti oppure è un segmento distinto.

Specificando l'attributo **NEARSTACK**, l'assemblatore posiziona il segmento di stack nel gruppo di dati **DGROUP** assieme al segmento di dato.

L'attributo **FARSTACK** posiziona lo stack in un segmento specifico diverso dal segmento di dato.

Esempio

La direttiva seguente organizza la memoria secondo il modello di tipo **LARGE**, con un segmento di stack separato dal segmento di dato:

```
.MODEL    large, farstack
```

6.12.2. Creazione dello stack

La direttiva **.STACK** permette di riservare uno spazio di memoria per lo stack; il formato di tale direttiva è il seguente:



```
.STACK    {mem}
```

La direttiva **.STACK** riserva uno spazio di memoria pari a *mem* byte. Per *default* lo spazio riservato per lo stack è di 1 Kbyte.

Esempio

La seguente direttiva riserva uno spazio di memoria di 2 Kbyte per lo stack:

```
.STACK    2048
```

6.12.3. Creazione del segmento di dato

Le direttive **.DATA**, **.DATA?**, **.CONST**, **.FARDATA** e **.FARDATA?** permettono di creare un segmento di dato. Il loro formato è il seguente:



```
.DATA  
.DATA?  
.CONST  
.FARDATA  
.FARDATA?
```

Segmenti di dato di tipo NEAR

La direttiva **.DATA** crea un segmento di dato di tipo **NEAR**. Questo segmento contiene i dati più frequentemente usati.

Usando la direttiva **.MODEL**, l'assemblatore definisce **DGROUP** come gruppo di segmenti di dati di tipo **NEAR**, e dunque attraverso la direttiva **.DATA** si definisce un segmento che appartiene al gruppo di segmenti **DGROUP**.

Le direttive **.DATA?** e **.CONST** definiscono segmenti di dato appartenenti anch'essi al gruppo

DGROUP. In questo caso l'assemblatore crea rispettivamente i segmenti di dato di nome `_BSS` e `CONST`.

- `CONST` è usata per memorizzare variabili inizializzate.
- `DATA?` è usata per memorizzare variabili non inizializzate.

Segmenti di dato di tipo FAR

Le direttive `.FARDATA` e `.FARDATA?` permettono di definire dati di tipo FAR. In questo caso l'assemblatore crea rispettivamente i segmenti di dato `FAR_DATA` e `FAR_BSS`.

- `FARDATA?` è usata per memorizzare variabili non inizializzate.

6.12.4. Creazione del segmento di codice

La direttiva `.CODE` permette di definire un segmento in cui specificare le istruzioni; il suo formato è il seguente:



```
.CODE    {nome}
```

La direttiva `.CODE` definisce un segmento di codice. Il campo opzionale *nome* definisce un eventuale nome da assegnare al segmento di codice in alternativa a quello di *default*.

In un modello di memoria di tipo `SMALL`, `COMPACT` e `TINY` la direttiva `.CODE` definisce un segmento di codice di tipo `NEAR` di nome `_TEXT`.

Quando si ha bisogno di uno spazio di codice superiore ai 64 Kbyte, bisogna usare i modelli di memoria `MEDIUM`, `LARGE` o `HUGE` in modo da creare segmenti di codice di tipo FAR. Con tali modelli di memoria l'assemblatore crea segmenti di codice differenti per ogni *file* di codice ed assegna ad ogni segmento il nome `MODNAME_TEXT`, in cui *MODNAME* coincide con il nome del file. In modelli di memoria in cui il codice è di tipo FAR è possibile avere più segmenti di codice all'interno di un unico modulo. Questo si ottiene separando i vari segmenti attraverso diverse direttive `.CODE`, ognuna avente come argomento un diverso nome di segmento.

Esempio

Il seguente frammento definisce due segmenti di codice con nomi diversi:

```
.CODE    FIRST
...
.CODE    SECOND
...
; primo segmento di codice
; secondo segmento di codice
```

Se, nei modelli `SMALL`, `COMPACT` e `TINY`, si usano più *file* sorgenti, l'assemblatore assegna ad ogni segmento il nome `_TEXT`; sarà compito del *linker* combinare i vari segmenti in uno unico.

6.13. Interfaccia con i sistemi operativi MS-DOS e compatibili

L'assemblatore *Microsoft MASM 6.0* mette a disposizione le direttive `.STARTUP` e `.EXIT` per iniziare e terminare un programma; il loro formato è il seguente:



```
.STARTUP
.EXIT    {val}
```

La direttiva `.STARTUP` permette di evitare la scrittura esplicita delle istruzioni iniziali di un

programma; analogamente la direttiva `.EXIT` evita la scrittura esplicita delle istruzioni che eseguono il ritorno al Sistema Operativo; essa accetta il parametro opzionale *val* corrispondente al valore di ritorno al Sistema Operativo.

Queste direttive vanno inserite nel *file* contenente il programma principale. A differenza delle altre direttive, le direttive `.STARTUP` e `.EXIT` producono istruzioni macchina.

La direttiva `.STARTUP` va posta all'inizio dell'esecuzione del programma, solitamente nella locazione immediatamente seguente la direttiva `.CODE` all'interno del modulo contenente il programma principale.

Esempio

Il seguente frammento di codice è generato dall'assemblatore in corrispondenza della direttiva `.STARTUP` nel caso di modello di stack di tipo `NEARSTACK`:

```

MOV     DX, DGROUP
MOV     DS, DX           ; inizializzazione di DS
MOV     BX, SS
SUB     BX, DX           ; calcolo dello stack pointer
SHL     BX, 1
SHL     BX, 1
SHL     BX, 1
SHL     BX, 1
CLI                               ; disabilitazione degli interrupt
MOV     SS, DX           ; inizializzazione di SS
ADD     SP, BX           ; inizializzazione di SP
STI
...
```

La direttiva `.EXIT` genera il codice opportuno per il ritorno al Sistema Operativo.

Esempio

Il frammento di codice seguente è generato da un comando `.EXIT 1`.

```

MOV     AL, 1           ; copia in AL del valore di ritorno
MOV     AH, 04CH
INT     21H
```

6.14. Scheletro di un programma (II versione)

Si mostra uno scheletro di programma che fa uso delle direttive `MASM` per la gestione dei segmenti, utilizzabile per sviluppare programmi contenuti in un'unico file.

```

.MODEL      small
.STACK      ; stack di 1 kbyte
.DATA       ; inizia il segmento di dato
...         ; dichiarazioni di costanti e variabili
.CODE       ; inizia il segmento di codice
.STARTUP    ; inizia la procedura principale
...         ; istruzioni
.EXIT       ; ritorno a MS-DOS
END         ; fine del modulo
```

6.15. Direttive per la definizione di identificatori globali esterni

La realizzazione di programmi di una certa complessità comporta la loro suddivisione in più moduli distinti. I vantaggi di avere diversi moduli sono legati alla leggibilità del programma ed alla maggiore rapidità nella fase di assemblaggio: durante le fasi di aggiornamento e modifica di un programma, l'assemblaggio è limitato ai soli moduli modificati.

Ogni modulo può definire un'area dati ed un'area di codice, ma solo un modulo (di solito quello contenente la procedura principale) può definire un'area di stack.

La ripartizione del programma su più moduli comporta l'introduzione del concetto di *variabile globale esterna*: le variabili globali esterne devono essere *definite* in un unico modulo e possono essere *dichiarate* e quindi utilizzate in più moduli.

È possibile definire come variabili globali esterne sia le variabili di dato sia le procedure. Le direttive **PUBLIC** e **EXTRN** permettono di definire ed utilizzare le variabili globali esterne.

La direttiva **PUBLIC** permette di specificare all'assemblatore quali sono le variabili globali, cioè quali nomi possono essere referenziati dall'interno degli altri moduli. La direttiva **EXTRN** specifica all'assemblatore quali nomi, definiti in un altro modulo del programma, sono utilizzati all'interno del modulo.

Ciascuna variabile globale richiede quindi due direttive:

- una direttiva **PUBLIC** nel modulo in cui è definita;
- una direttiva **EXTRN** in ciascun modulo in cui è usata.

Il formato della direttiva **PUBLIC** è il seguente:



PUBLIC *nome*

L'identificatore *nome* può essere usato come riferimento esterno in altre parti del programma.

Esempi

I due comandi seguenti specificano all'assemblatore che i nomi **MSG2** e **EXT_PROC** possono essere utilizzati in altri moduli appartenenti al programma:

```
PUBLIC    MSG2
PUBLIC    EXT_PROC
```

Se in un programma è presente la dichiarazione di variabili globali, l'assemblatore inserisce all'inizio del modulo oggetto una serie di campi appositi. Tali campi sono letti dal *linker*, che in tal modo è informato di quali nomi definiti in quel modulo oggetto possono essere usati come riferimenti esterni da altri moduli. Il formato della direttiva **EXTRN** è il seguente:



EXTRN *nome:tipo*

Se l'identificatore *nome* rappresenta un *dato* il campo *tipo* può essere uno tra **BYTE**, **WORD**, **DWORD**, **QWORD** o **TBYTE**; se il *nome* rappresenta un *nome di procedura* il *tipo* può essere **NEAR** o **FAR**; se il *nome* rappresenta una costante numerica il *tipo* è *abs*.

Esempi

Il primo comando dichiara come variabile globale esterna la variabile di tipo **BYTE** e di nome **MSG2** definita in un altro modulo; il secondo dichiara come esterna la procedura **EXT_PROC** di tipo **NEAR** definita in un altro modulo.

```

EXTRN      MSG2:BYTE
EXTRN      EXT_PROC:NEAR

```

6.16. Uso dell'emulatore emu8086

emu8086 è un *software d'emulazione*, ossia un programma che permette l'esecuzione di software originariamente scritto per un ambiente (hardware o software) diverso da quello su cui esso viene eseguito. Nel caso di **emu8086** il codice, scritto e compilato per il processore x86 e MS-DOS, viene eseguito su una *macchina virtuale* in ambiente MS-Windows, che riproduce il comportamento di memoria, monitor e dispositivi di I/O attraverso moduli software e opportune finestre grafiche.

Una macchina virtuale emula l'ambiente di un calcolatore mediante un cosiddetto "strato di virtualizzazione", che intercetta le richieste avanzate dal programma dell'utente al sistema, le traduce e le inoltra verso le risorse del sistema ospite (CPU, memoria, disco, rete e altre risorse hardware). Pertanto, il sistema non risulta utilizzato in modo diretto e le prestazioni sono in genere penalizzate, ma i maggiori controlli effettuati dalla macchina virtuale nell'interazione con risorse critiche permettono di evitare il blocco del programma o del sistema operativo.

emu8086 è un ambiente completo per la programmazione in Assembler x86 con un'interfaccia grafica piuttosto intuitiva. Esso comprende un editor, un assemblatore, l'emulatore di una CPU x86 e un debugger. Permette di creare file COM (eseguibili più piccoli e semplici, che alla chiamata sono copiati in RAM ed eseguiti) ed EXE (codici strutturati che per l'esecuzione necessitano di passi aggiuntivi effettuati dal *loader*) per il sistema operativo MS-DOS, file binari puri e boot loader (con estensione default *boot*). È supportata l'esecuzione in modalità *step-by-step* ed è possibile utilizzare e personalizzare periferiche esterne al processore.

L'emulatore impone alcune limitazioni alla programmazione, forzando l'utente a lavorare su un unico file sorgente e consentendo soltanto una gestione semplificata dei segmenti; tuttavia, può rivelarsi molto utile in una fase di approccio iniziale all'architettura x86 e alla relativa programmazione in linguaggio Assembler, perché evita la generazione di situazioni critiche per il sistema e permette controllo e osservabilità superiori sull'esecuzione del programma rispetto a un debugger che interagisca con il processore (per esempio, consente l'esecuzione step-by-step in due direzioni, verso l'istruzione successiva o la precedente).

emu8086 supporta la grande maggioranza delle convenzioni di linguaggio di MASM, e i codici binari prodotti sono spesso identici a quelli ottenibili con l'assemblatore Microsoft: le principali differenze di sintassi tra MASM e **emu8086** sono riassunte di seguito.

6.16.1. Direttive proprie di emu8086

Esistono alcune direttive proprie di **emu8086**. Se un file sorgente scritto in questo ambiente deve essere compilato con MASM, le linee di codice che contengono queste direttive devono essere commentate.

Di seguito l'elenco delle direttive proprie di **emu8086** che definiscono il tipo di file che sarà creato al momento della compilazione. Se omesse, **emu8086** chiede il tipo di file da creare all'utente.



```

#MAKE_COM#
#MAKE_BIN#
#MAKE_BOOT#
#MAKE_EXE#

```

#MAKE_COM# permette di generare un file di tipo **.COM**, il formato di file eseguibile più semplice, il cui codice eseguibile è caricato all'indirizzo 100h del segmento di codice (i primi 256 byte sono utilizzati dal sistema operativo per alcuni dati del programma, quali ad esempio i parametri dalla linea di comando). In questo caso la direttiva **ORG 100h** deve essere aggiunta prima del codice. L'esecuzione comincia sempre dal primo byte del file, e la dimensione del codice è limitata a un singolo segmento.

#MAKE_EXE# determina la generazione di un file di tipo **.EXE**, un formato di file eseguibile più avanzato, non limitato nella dimensione e nel numero dei segmenti. Il punto di partenza dell'esecuzione è determinato dal programmatore.

#MAKE_BIN# permette di generare un semplice file eseguibile binario, la cui esecuzione comincia da **CS:IP**. È un formato di file esclusivo dell'emulatore **emu8086**.

#MAKE_BOOT# genera un file che può essere utilizzato come prima traccia di un floppy disk (boot sector), per permettere il boot del sistema. In questo caso, la direttiva **ORG 7C00h** deve essere aggiunta prima del codice, perché all'avvio il computer carica in memoria la prima traccia di un disco all'indirizzo **0000:7C00**. La dimensione di un file **.BOOT** deve essere inferiore a 512 byte (dimensione limitata dal settore di boot). L'esecuzione del programma parte dal primo byte del file. Anche questo è un formato di file esclusivo di **emu8086**.

Le seguenti direttive permettono di definire il valore iniziale di un registro:



```
#CS = valore#
#DS = valore#
...
```

All'inizio dell'esecuzione del programma, i registri saranno caratterizzati dal valore specificato.

6.16.2. Operatori non supportati

emu8086 non supporta alcuni degli operatori che calcolano gli attributi delle variabili, ovvero **TYPE**, **LENGTH** e **SIZE**. Pertanto, i relativi valori devono essere esplicitamente introdotti nel codice quando necessario.

Inoltre, non è possibile dichiarare dati di tipo quadword o tenbyte, e le relative direttive **DQ** e **TB** non sono implementate.

6.16.3. Operatore aritmetico di modulo

La sintassi utilizzata per il calcolo del modulo a livello di codice sorgente è **%** (come nel linguaggio ANSI C) invece di **MOD**.

6.16.4. Abbreviazione degli operatori **WORD PTR** e **BYTE PTR**

emu8086 supporta versioni abbreviate degli operatori **WORD PTR** e **BYTE PTR**, non definite in MASM: **WORD PTR** può essere sostituito da **W.**, mentre **BYTE PTR** da **B.**

Esempio:

```
LEA BX, VAR1
MOV WORD PTR [BX], 1234h
MOV W.[BX], 1234H
```

La prima istruzione **MOV** utilizza il formato standard, utilizzabile sia in MASM, sia in

emu8086.

La seconda istruzione **MOV** utilizza la sintassi abbreviata propria di **emu8086**.

6.16.5. Direttiva **=** per la definizione di costanti

In **emu8086** la direttiva **=** ha lo stesso comportamento di **EQU**.

6.16.6. Direttive non supportate: **GROUP** e **ASSUME**

La direttiva **GROUP**, per raggruppare più segmenti logici in un unico segmento fisico, non è supportata. Inoltre, **emu8086** ignora la direttiva **ASSUME**. L'utilizzo esplicito di codici indicanti il segmento (**CS:**, **DS:**, **ES:** o **SS:**) è preferito, e richiesto da **em8086** quando il dato cui si vuole accedere è in un segmento diverso da **DS**.

Esempio:

```
MOV AH, [BX]           ; lettura di un byte da DS:BX
MOV AH, ES:[BX]        ; lettura di un byte da ES:BX
```

La prima istruzione preleva il byte di offset **BX** dal data segment; la seconda dall'extra segment.

6.16.7. Istruzioni per il controllo del processore non supportate

emu8086 non supporta correttamente alcune istruzioni di controllo del processore che presuppongono l'interazione con moduli esterni. L'istruzione **HLT** ha come risultato il blocco del processore, dal quale non è possibile uscire mediante l'emulazione di un interrupt esterno. Le istruzioni **WAIT**, **ESC** e **LOCK** generano errori in compilazione.

6.16.8. Esempio di conversione di programma da **emu8086** a **MASM**

Per rendere compatibile in **MASM** un codice scritto per **emu8086** è necessario effettuare alcune modifiche, tenendo in considerazione quanto introdotto nei paragrafi precedenti. Di seguito si riporta un codice per **emu8086** e una versione modificata dello stesso codice compatibile con **MASM**.

Esempio (versione **emu8086**):

```
zero EQU 0
size EQU 15%10           ; operatore di modulo: %

#MAKE_EXE#               ; creazione di file di tipo EXE
#AX = zero#              ; azzeramento registri (non è richiesta l'esecuzione
#DX = zero#              ; di istruzioni)
#BX = zero#
#DI = zero#

.DATA
var1 DD 12901
var2 DD -1314
vet  size DUP (?)

.CODE
.STARTUP
MOV AX, W.var1           ; forma contratta dell'operatore PTR
MOV DX, W.var1+2
ADD AX, W.var2
ADC DX, W.var2+2
MOV CX, size             ; non è possibile usare l'operatore LENGTH
ciclo: mov vet[DI], 0
add DI, 2
LOOP ciclo
[...]
```

Esempio (versione **MASM**):

```
zero EQU 0
size EQU 15 MOD 10       ; operatore di modulo: MOD
                          ; sono stati rimossi gli operandi tipici di emu8086
                          ; (introdotti da "%")

.DATA
var1 DD 12901
var2 DD -1314
vet  size DUP (?)

.code
.startup
XOR AX, AX               ; per garantire l'azzeramento dei registri è
XOR DX, DX               ; necessario utilizzare specifiche istruzioni
XOR BX, BX
XOR DI, DI
MOV AX, WORD PTR var1    ; forma tradizionale dell'operatore PTR
MOV DX, WORD PTR var1+2
ADD AX, WORD PTR var2
ADC DX, WORD PTR var2+2
MOV CX, LENGTH vet       ; uso dell'operatore LENGTH (facoltativo)
ciclo: MOV vet[DI], 0
ADD DI, 2
LOOP ciclo
[...]
```

7. I modi di indirizzamento

In questo capitolo verranno presentati i diversi modi con cui il linguaggio Assembler x86 permette di specificare gli *operandi*, ossia i dati su cui il processore esegue l'operazione indicata da ciascuna istruzione. A seconda dell'istruzione essi possono essere in numero variabile (da nessuno a due). Gli operandi possono far parte dell'istruzione stessa, risiedere in uno dei registri, essere memorizzati in una locazione di memoria oppure trovarsi su una porta di I/O.

Per accedere ai diversi operandi l'architettura x86 dispone di diversi *modi di indirizzamento* e precisamente i modi *Register*, *Immediate*, *Direct*, *Register Indirect*, *Base Relative*, *Direct Indexed* e *Base Indexed*.

Per illustrare i vari modi di indirizzamento utilizzeremo l'istruzione *MOV*, che esegue il trasferimento di un dato da un operando *sorgente* ad un operando *destinazione*, secondo il seguente formato:



MOV *destinazione, sorgente*

7.1. Register Addressing

Il modo di indirizzamento *Register* (Fig. 7.1) utilizza, come operando, il contenuto di uno dei registri interni al microprocessore; qualunque registro di 16 bit può essere usato come operando sorgente o destinazione, ma solo i quattro registri *AX*, *BX*, *CX* e *DX* possono essere usati per memorizzare sia un byte sia una word.

Esempi

L'istruzione seguente copia nel registro *CL* il byte contenuto nel registro *BH*:

MOV CL, BH

L'istruzione seguente copia nel registro SI il contenuto del registro BX:

```
MOV     SI, BX
```

7.2. Immediate Addressing

Il modo di indirizzamento *Immediate* (Fig. 7.2) permette di caricare nell'operando destinazione (registro o locazione di memoria) una costante di tipo byte o di tipo word. Tale costante prende il nome di *valore immediato* e viene codificata dall'assemblatore all'interno dell'istruzione macchina stessa. Il valore immediato può corrispondere solo ad un operando sorgente e non ad un operando destinazione.

Esempio

La seguente istruzione è priva di significato e determina un errore a livello di assemblaggio:

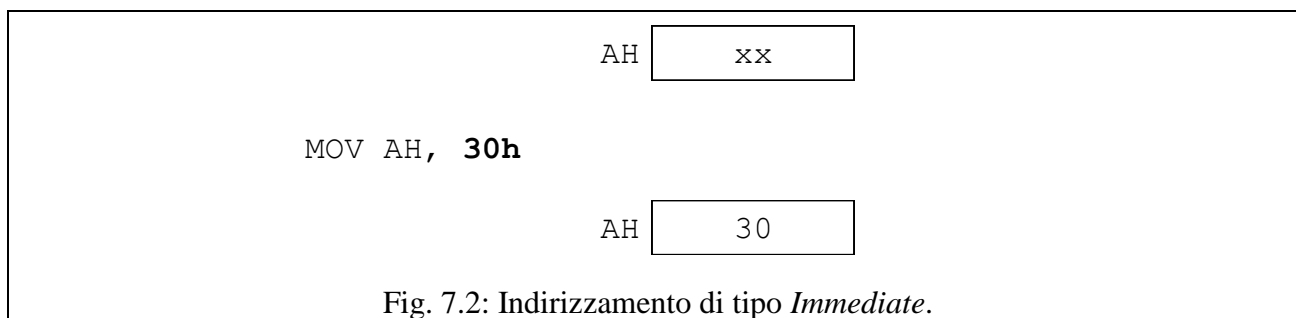
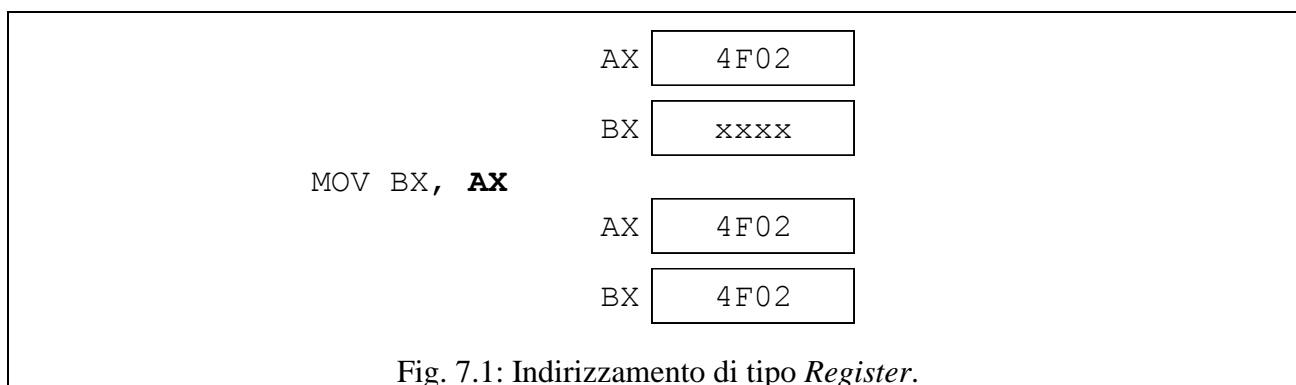


```
MOV     30H, AH
```

La seguente istruzione copia il valore immediato 30H nel registro AH:



```
MOV     AH, 30H
```



7.3. Direct Addressing

L'indirizzamento di tipo *Direct* (Fig. 7.3) permette di utilizzare, come operando, il contenuto di una cella di memoria il cui indirizzo è specificato tramite un simbolo definito attraverso una pseu-

do-istruzione. Tale operando è detto *operando diretto* e può essere espresso in uno dei seguenti formati:



nome
nome+spiazzamento
nome-spiazzamento
[nome]
nome[spiazzamento]
nome[-spiazzamento]

Gli operatori + e - sono utilizzati dall'assemblatore per calcolare l'indirizzo dell'operando diretto. Negli ultimi tre casi l'operatore [] può essere utilizzato per racchiudere il *nome* della variabile od il valore dello *spiazzamento*. Il calcolo dell'indirizzo è fatta dall'assemblatore e costituisce un valore numerico costante.

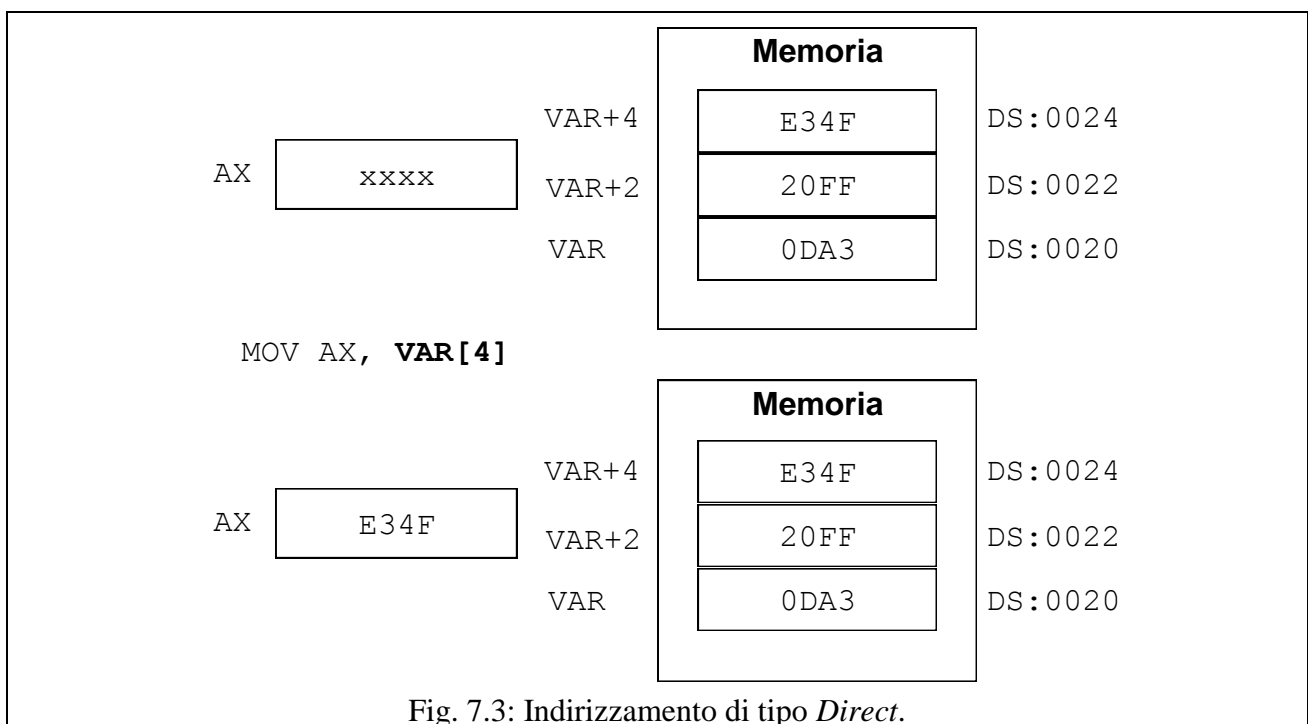


Fig. 7.3: Indirizzamento di tipo *Direct*.

Esempio

L'istruzione seguente copia il valore contenuto in AX nella locazione di memoria di nome VAR:

```
MOV    VAR, AX
```

Le seguenti istruzioni, per il linguaggio macchina generato, sono equivalenti a coppie:

```

MOV    VAR, AL
MOV    [VAR], AL

MOV    VAR+4, AL
MOV    VAR[4], AL

MOV    VAR-4, AL
MOV    VAR[-4], AL

```

7.3.1. Segment Override

Nel modo di indirizzamento diretto il registro di segmento di *default* è il registro DS. Un apposito operatore, detto di *segment override*, (“:”) permette di utilizzare, nel calcolo dell’indirizzo fisico, un registro di segmento diverso da quello di *default*.

Esempio

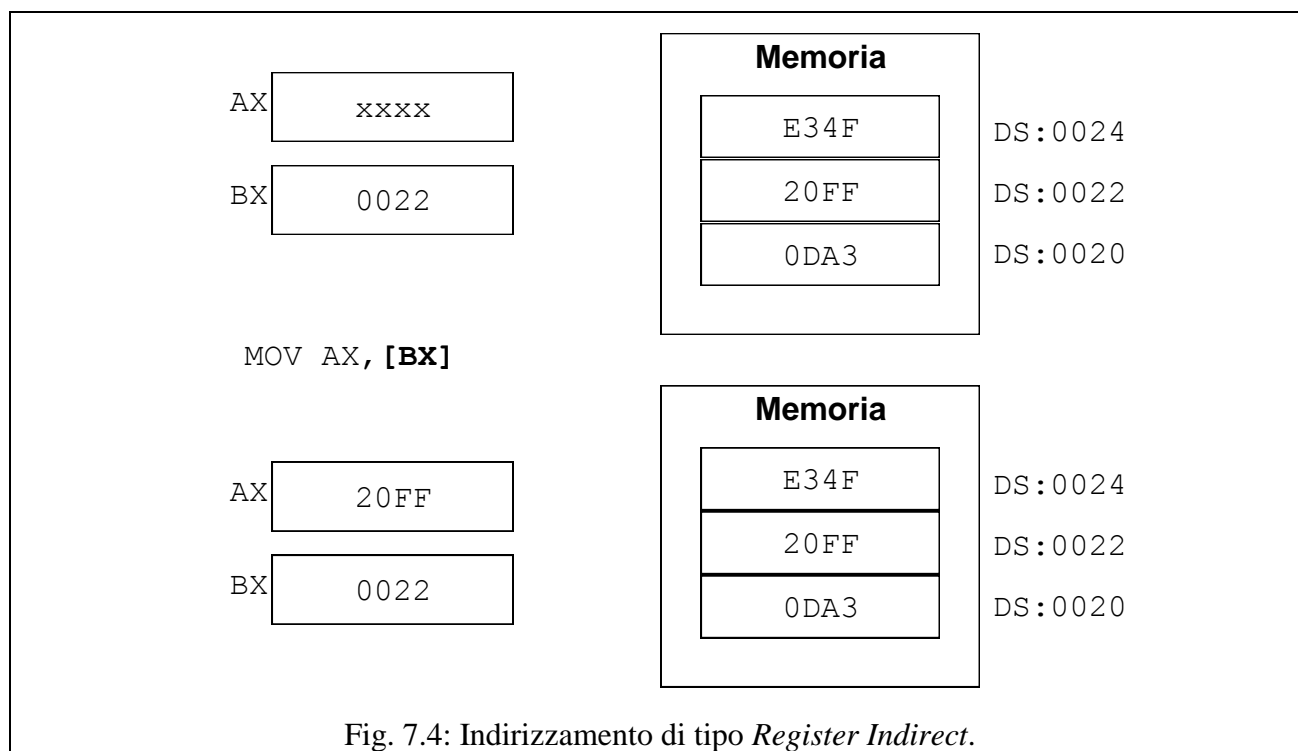
La locazione di memoria VAR2 si trova nel segmento puntato da ES. L’istruzione seguente copia il valore della variabile VAR2 nel registro AX:

```
MOV     AX, ES:VAR2
```

7.4. Register Indirect Addressing

L’indirizzamento di tipo *Register Indirect* (Fig. 7.4) permette di utilizzare il contenuto di un registro quale *effective address* dell’operando. I registri utilizzati per memorizzare l’indirizzo di un operando sono il registro base (BX), i registri indice (SI e DI) ed il registro *base pointer* (BP).

In Tab. 7.1 sono espressi gli abbinamenti di *default* tra registro di offset e registro di segmento per il calcolo dell’indirizzo fisico; tali abbinamenti possono essere modificati usando l’operatore di *segment override*.



Registro	Registro di segmento
BX	DS
SI	DS
DI	DS (ES per le istruzioni su stringhe)
BP	SS

Tab. 7.1: Abbinamento tra registro di offset e registro di segmento.

Esempio

L'istruzione seguente copia nel registro AX il contenuto nella locazione di memoria avente come *effective address* il contenuto del registro BX. L'indirizzo fisico dell'operando è dato dalla combinazione DS:BX.

```
MOV     AX, [BX]
```

Esercizio: Copia di un vettore di interi (I versione).

Si scriva un frammento di programma che esegua la copia di un vettore di interi da un'area di memoria sorgente ad un'area di memoria destinazione. Una possibile soluzione in linguaggio C è la seguente:

```
#define LUNG 500
main()
{
  int i, dest[LUNG], sorg[LUNG];
  ...
  for (i=0 ; i<LUNG ; i++) dest[i] = sorg[i];
}
```

Il seguente frammento di codice è una possibile soluzione in linguaggio Assembler:

```
LUNG      EQU      500
           .MODEL  small
           .DATA
SORG       DW       LUNG DUP (?)
DEST       DW       LUNG DUP (?)
           .CODE
           ...
           MOV      SI, OFFSET SORG ; copia in SI l'offset del vettore SORG
           MOV      DI, OFFSET DEST ; copia in DI l'offset del vettore DEST
           MOV      CX, LUNG        ; copia in CX del numero di elementi
ciclo:     MOV      AX, [SI]
           MOV      [DI], AX
           ADD      SI, 2            ; aggiornamento dei registri indice
           ADD      DI, 2
           LOOP     ciclo
           ...
```

7.5. Base Relative Addressing

Nel modo di indirizzamento di tipo *Base Relative*, l'indirizzo effettivo dell'operando è calcolato sommando uno *spiazzamento* al contenuto di un registro o di base (BX o BP) o di indice (SI o DI). Un indirizzamento di tipo *Base Relative* può essere espresso in uno dei seguenti formati, tra loro equivalenti:



```
[Registro di base] + spiazzamento
[Registro di indice] + spiazzamento
[Registro di base + spiazzamento]
[Registro di indice + spiazzamento]
spiazzamento[Registro di base]
spiazzamento[Registro di indice]
```

Esempio

Le tre istruzioni seguenti vengono assemblate nella stessa istruzione in linguaggio macchina:

```
MOV    AX, [BX]+4
MOV    AX, [BX+4]
MOV    AX, 4[BX]
```

Usando come registro di base il registro BP, il calcolo dell'indirizzo fisico è ottenuto utilizzando il contenuto del registro di segmento di stack SS, anziché il registro DS. Questo permette l'accesso a dati contenuti nello *stack* ed è particolarmente utile per il passaggio di parametri alle procedure. L'utilizzo di tale modo di indirizzamento verrà spiegato nel dettaglio nel Cap. 13.

In Fig. 7.5 è raffigurato il meccanismo di calcolo dell'*effective address* per l'indirizzamento di tipo *Base Relative*; si noti che *i* può essere o un valore numerico, od una costante definita attraverso la direttiva EQU o la direttiva =.

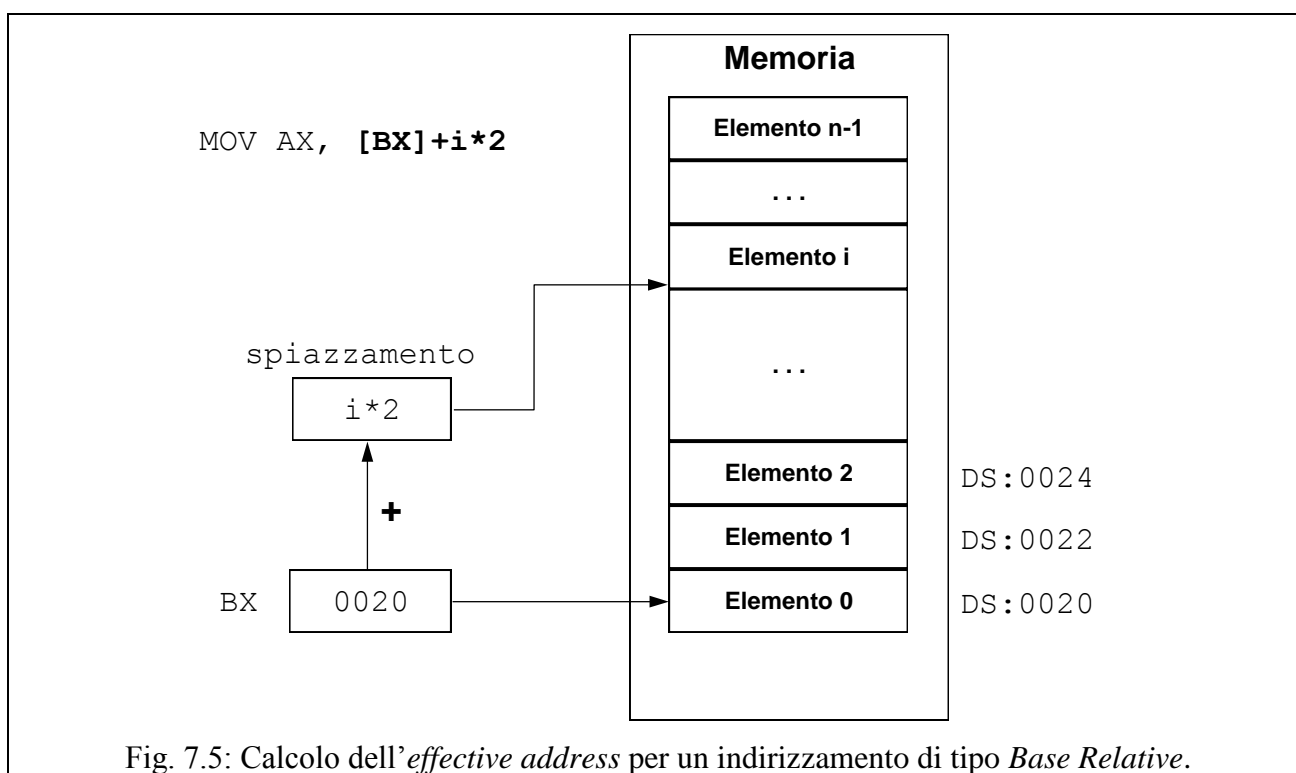


Fig. 7.5: Calcolo dell'*effective address* per un indirizzamento di tipo *Base Relative*.

7.6. Direct Indexed Addressing

Nel caso dell'indirizzamento di tipo *Direct Indexed* (Fig. 7.6), l'*effective address* dell'operando si ottiene sommando il valore di un indirizzo di *offset* di una variabile al contenuto di uno dei *registri indice* (SI o DI) o di *base* (BX o BP). I possibili formati sono i seguenti:



Offset[Registro Indice]
Offset[Registro di Base]

Esempio

Nell'istruzione seguente l'operando destinazione è indirizzato in modo *Direct Indexed*.

```
MOV     TABLE[DI], AX
```

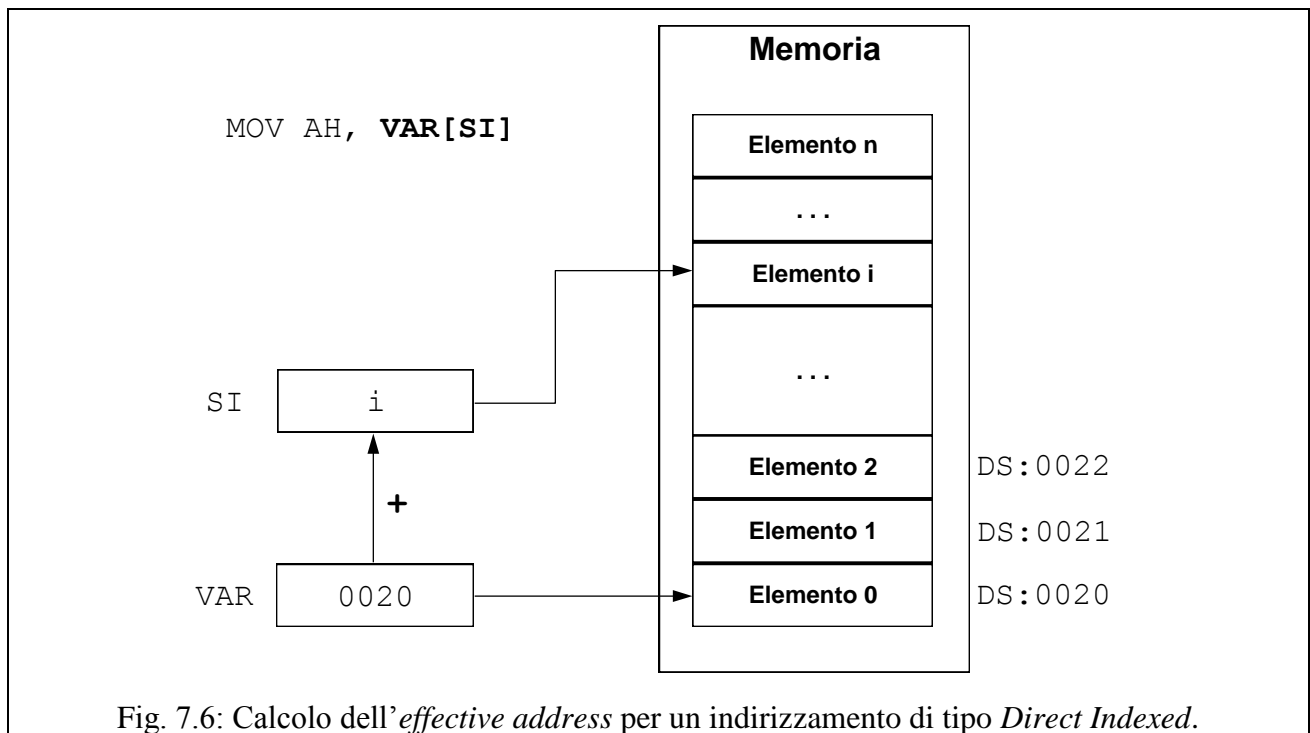
Esercizio: Copia di un vettore di interi (II versione).

Si presenta ora una seconda soluzione all'esercizio proposto precedentemente.

```

LUNG     EQU     500
          .MODEL  small
          .DATA
SORG      DW     LUNG DUP(?)
DEST      DW     LUNG DUP(?)
          .CODE
...
MOV       SI, 0      ; inizializzazione del registro indice
MOV       CX, LUNG    ; caricamento in CX del numero di elementi
ciclo:    MOV      AX, SORG[SI]
          MOV      DEST[SI], AX
          ADD      SI, 2      ; incremento del displacement
          LOOP     ciclo      ; scansione conclusa ? No => va a ciclo
          ...              ; Sì

```



7.7. Base Indexed Addressing

Nel caso di indirizzamento di tipo *Base Indexed* il valore dell'*effective address* dell'operando è ottenuto dalla somma di tre contributi:

1. il contenuto di un registro di base (BP o BX)
2. il contenuto di un registro di indice (SI o DI)
3. un eventuale *spiazzamento*.

I possibili formati sono i seguenti:



```
spiazzamento[BX][DI]
spiazzamento[BX][SI]
spiazzamento[BP][DI]
spiazzamento[BP][SI]
[BX][DI]
[BX][SI]
[BP][DI]
[BP][SI]
```

Il campo opzionale di *spiazzamento* può essere un indirizzo di variabile oppure una costante numerica. Se sono presenti entrambi, l'assemblatore esegue la somma dei due valori durante la fase di assemblaggio.

Esempio

Le seguenti istruzioni sono esempi di utilizzo dell'indirizzamento di tipo *Base Indexed*.

```
MOV    AX, TABLE[BX][DI]+6
MOV    AX, [BX][DI]
MOV    AX, TABLE[BX][DI]
MOV    AX, 4[BX][DI]
```

Il modo di indirizzamento *Base Indexed* permette una facile memorizzazione di matrici bidimensionali, utilizzando opportunamente i registri di base, di indice e lo spiazzamento. Le matrici possono essere memorizzate eseguendo la conversione nell'equivalente vettore in due modi alternativi: *per righe* o *per colonne*. La Fig. 7.7 illustra un esempio di vettorizzazione di una matrice di 3 righe e 5 colonne. In (a) la matrice è vettorizzata per righe, in (b) per colonne.

	0	1	2	3	4
0	A	B	C	D	E
1	F	G	H	I	L
2	M	N	O	P	Q

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	L
10	M
11	N
12	O
13	P
14	Q

(a)

0	A
1	F
2	M
3	B
4	G
5	N
6	C
7	H
8	O
9	D
10	I
11	P
12	E
13	L
14	Q

(b)

Fig. 7.7: Vettorizzazione di una matrice bidimensionale.

Nel caso di memorizzazione di una matrice per righe lo *spiazzamento* punta all'indirizzo di partenza del vettore, il *registro base* scandisce le righe e punta all'indirizzo di partenza della riga *i* ed il *registro indice* scandisce le colonne. In Fig. 7.8 è illustrato un esempio di calcolo dell'*effective address* di un operando in un modo di indirizzamento di tipo *Base Indexed* nel caso di una matrice di

m righe ed n colonne memorizzata per righe.

Esercizio: Copia di una riga di una matrice di dati.

Si realizzi un frammento di programma che esegua la copia della quarta riga di una matrice di 4 righe e 5 colonne da una matrice sorgente ad una matrice destinazione.

La soluzione proposta in linguaggio C è la seguente:

```
main()
{
  int i, sorg[4][5], dest[4][5];
  ...
  for (i=0 ; i < 5 ; i++) dest[3][i] = sorg[3][i];
}
```

La soluzione proposta in linguaggio Assembler è la seguente:

```
NUMRIGHE EQU 4
NUMCOL EQU 5
.MODEL small
.DATA
SORG DW NUMRIGHE*NUMCOL DUP (?) ; matrice sorgente
DEST DW NUMRIGHE*NUMCOL DUP (?) ; matrice destinazione
.CODE
...
MOV BX, NUMCOL*3*2 ; caricamento in BX dello spiazamento
; del primo elemento della quarta riga
MOV SI, 0 ; inizializzazione del registro SI
MOV CX, NUMCOL ; caricamento in CX del numero di colonne
ciclo: MOV AX, SORG[BX][SI]
MOV DEST[BX][SI], AX
ADD SI, 2 ; incremento dell'indice di colonna
LOOP ciclo ; fine della scansione ? No => va a ciclo
... ; Sì
```

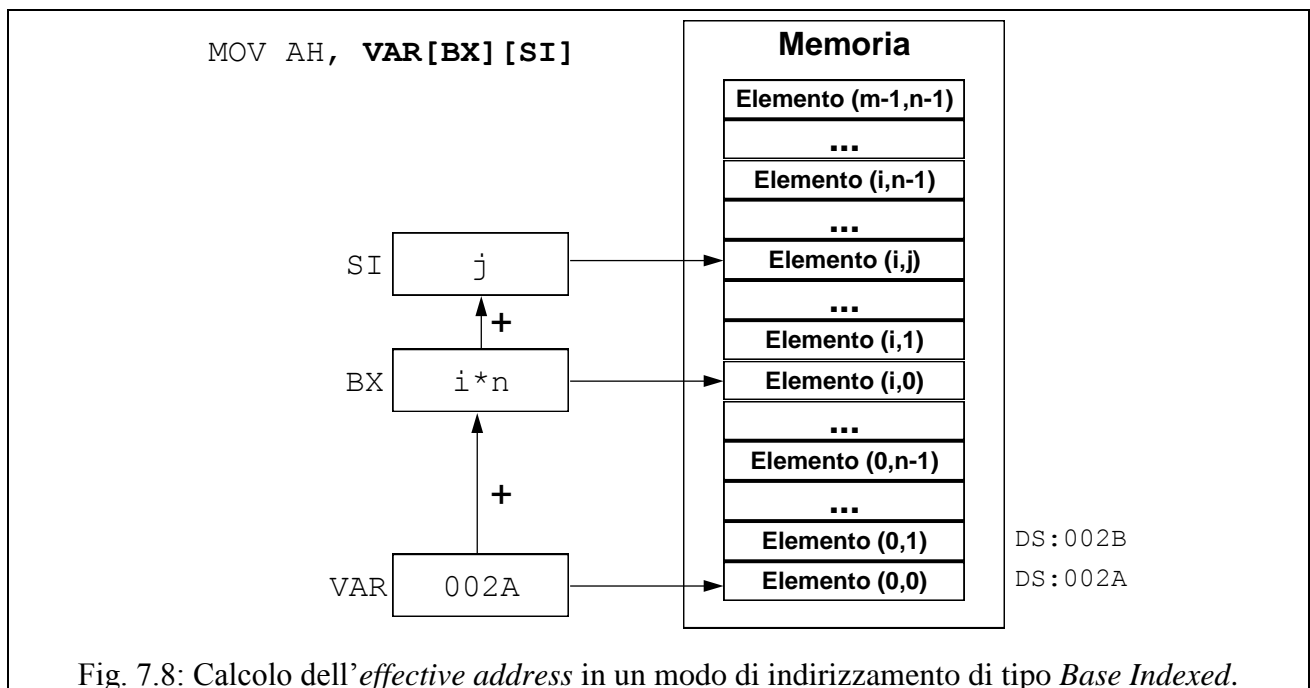


Fig. 7.8: Calcolo dell'*effective address* in un modo di indirizzamento di tipo *Base Indexed*.

8. Le istruzioni di trasferimento dati

In questo capitolo verranno presentate e descritte le istruzioni per il trasferimento di dati, vale a dire quelle istruzioni che permettono di trasferire dati tra i registri interni o tra un registro interno ed una locazione di memoria.

8.1. L'istruzione MOV

L'istruzione **MOV** (*MOV*e) copia un dato da una posizione ad un'altra. Il suo formato è il seguente:



MOV *destinazione, sorgente*

I dati vengono letti dall'operando *sorgente* e memorizzati nell'operando *destinazione*. L'operando *destinazione* può essere un registro od una locazione di memoria, l'operando *sorgente* può essere un registro, una locazione di memoria oppure un valore immediato. L'istruzione **MOV** non modifica né il valore dei flag né il contenuto dell'operando *sorgente*.

Vi sono alcune restrizioni nell'uso dell'istruzione **MOV**:

- i due operandi devono essere dello stesso tipo (o entrambi byte o entrambi word).

Esempio

L'istruzione seguente non è lecita:



MOV **BL, DX** ; copia di un registro a 16 bit
 ; in un registro a 8 bit

- Il registro **IP** non può essere né *sorgente* né *destinazione* ed il registro **CS** non può essere *destinazione*.

Questo vincolo è una protezione interna del processore contro manipolazioni da parte del

programmatore: i registri CS e IP sono gestiti unicamente dal processore.

Esempio

Le istruzioni seguenti causano un errore al momento dell'assemblaggio:

☹ `MOV CS, 4530H`
`MOV AX, IP`

- Non si può copiare un valore immediato direttamente in un registro di segmento e non si può eseguire il trasferimento del contenuto di un registro di segmento in un altro registro di segmento.

Esempio

Le due istruzioni seguenti non sono lecite:

☹ `MOV SS, 2000H`
`MOV ES, DS`

Per eseguire le stesse operazioni occorre utilizzare un registro come memoria temporanea: a titolo d'esempio, le seguenti istruzioni copiano il valore 2000H nel registro SS:

☺ `MOV AX, 2000H`
`MOV SS, AX`

- Non si può copiare direttamente il contenuto di una cella di memoria in un'altra cella di memoria.

Esempio

Dette DATO1 e DATO2 due variabili di tipo byte, la seguente istruzione non è lecita:

☹ `MOV DATO2, DATO1`

Per eseguire l'operazione precedente occorre copiare temporaneamente il contenuto della cella di memoria sorgente in un registro. Le istruzioni seguenti copiano in DATO2 il contenuto del byte memorizzato in DATO1, utilizzando il registro BH.

☺ `MOV BH, DATO1`
`MOV DATO2, BH`

8.2. L'istruzione XCHG

L'istruzione **XCHG** (*eXCHG*ange) permette di eseguire lo scambio tra due registri o tra un registro ed una locazione di memoria. Il suo formato è il seguente:



`XCHG operand1, operand2`

Dopo l'esecuzione di questa istruzione il contenuto di *operand1* è pari al valore che *operand2* aveva prima dell'esecuzione dell'istruzione stessa e viceversa. L'istruzione XCHG non

modifica il valore dei flag.

Esempi

La seguente istruzione esegue lo scambio tra il contenuto del registro AX e la variabile di nome VAR:

```
XCHG    AX, VAR
```

Essa è equivalente alla seguente sequenza di istruzioni:

```
MOV     DX, AX      ; DX è un registro temporaneo
MOV     AX, VAR
MOV     VAR, DX
```

Le istruzioni seguenti costituiscono un esempio di uso dell'istruzione XCHG: la prima scambia i contenuti dei registri AX e BX, la seconda scambia il contenuto del registro AX con il valore di SOMMA, la terza scambia il valore di DL con il dato contenuto in LIST[SI], la quarta scambia il contenuto dei due byte interni al registro CX.

```
XCHG    AX, BX
XCHG    AX, SOMMA
XCHG    LIST[SI], DL
XCHG    CL, CH
```

Vi sono alcune restrizioni sull'uso dell'istruzione XCHG:

- gli operandi devono avere la stessa lunghezza (o byte o word);
- nessuno dei due operandi può essere un registro di segmento;
- non è possibile scambiare il contenuto di due locazioni di memoria.

Esempi

Le seguenti istruzioni non sono lecite:



```
XCHG    AX, BL
XCHG    AX, ES
XCHG    DATO1, DATO2
```

Lo scambio del contenuto di due locazioni di memoria deve essere effettuato utilizzando un registro temporaneo. Le seguenti tre istruzioni eseguono lo scambio del contenuto delle locazioni DATO1 e DATO2.



```
MOV     AH, DATO2
XCHG    AH, DATO1
MOV     DATO2, AH
```

Esercizio: *Inversione di un vettore.*

Si realizzi un frammento di programma che esegua l'inversione del contenuto di un vettore di caratteri: al termine dell'esecuzione, gli elementi del vettore devono essere memorizzati nell'ordine inverso rispetto a quello iniziale. La soluzione proposta in linguaggio C è la seguente:

```
#define LUNG 150
main()
{
    int i;
    char vett[LUNG], temp;
    ...
    for (i=0 ; i < (LUNG/2) ; i++)
    {
        temp = vett[LUNG-1-i];
        vett[LUNG-1-i] = vett[i];
        vett[i] = temp;
    }
    ...
}
```

La soluzione proposta in linguaggio Assembler è la seguente:

```
LUNG      EQU      150
           .MODEL   small
           .STACK
           .DATA
VETT      DB       LUNG DUP (?)
           ...
           .CODE
           ...
           MOV      SI, 0                ; SI punta al primo elemento
           MOV      DI, LUNG-1          ; DI punta all'ultimo elemento
           MOV      CX, LUNG/2
ciclo:    MOV      AH, VETT[SI]          ; scambio del contenuto
           XCHG     AH, VETT[DI]
           MOV      VETT[SI], AH
           INC      SI                  ; aggiornamento degli indici
           DEC      DI
           LOOP     ciclo
           ...
```

8.3. L'istruzione LEA

L'istruzione **LEA** (*Load Effective Address*) trasferisce l'offset dell'operando sorgente nell'operando destinazione. Il suo formato è il seguente:



LEA destinazione, sorgente

L'offset dell'operando *sorgente* viene copiato nell'operando *destinazione*. L'operando *sorgente* deve essere una locazione di memoria, mentre quello *destinazione* deve essere un registro *general purpose* di 16 bit. L'istruzione LEA non modifica il valore dei flag.

Esempio

La seguente istruzione copia nel registro `SI` l'offset della variabile di nome `VAR`, definita di tipo byte:

```
LEA    SI, VAR
```

L'operatore `OFFSET` permette di eseguire un'operazione analoga a quella effettuata dall'istruzione `LEA`, ma ha un campo di utilizzo più limitato in quanto accetta solo nomi di locazioni di memoria e non indirizzi specificati attraverso un registro indice; in quest'ultimo caso è necessario dunque utilizzare l'istruzione `LEA`.

Esempi

Le due istruzioni seguenti sono semanticamente equivalenti:

```
MOV    SI, OFFSET VAR
LEA    SI, VAR
```

La seguente istruzione non è lecita:



```
MOV    AX, OFFSET VAR[SI]
```

Per eseguire la precedente operazione si possono utilizzare le seguenti istruzioni:



```
MOV    AX, OFFSET VAR
ADD    AX, SI
```

Utilizzando l'istruzione `LEA` è possibile effettuare l'operazione con un'unica istruzione:



```
LEA    AX, VAR[SI]
```

Esercizio: Copia di un vettore di interi (III versione).

Viene proposta ora una terza versione del problema della copia di un vettore di interi introdotta nel capitolo precedente.

```
LUNG    EQU    500
        .MODEL small
        .STACK
        .DATA
SORG     DW     LUNG DUP(?)
DEST     DW     LUNG DUP(?)
        ...
        .CODE
        ...
        LEA     SI, SORG      ; equivalente a MOV SI, OFFSET SORG
        LEA     DI, DEST      ; equivalente a MOV DI, OFFSET DEST
        MOV     CX, LUNG
ciclo:   MOV     AX, [SI]
        MOV     [DI], AX
        ADD     SI, 2
        ADD     DI, 2
        LOOP    ciclo
        ...
```


8.4. L'istruzione XLAT

L'istruzione **XLAT** (*translate*) permette una facile gestione di tabelle di conversione. Il suo formato è il seguente:



XLAT

Durante l'esecuzione dell'istruzione XLAT, il processore esegue la somma del contenuto dei registri AL e BX, trasferendo poi in AL il dato avente come offset il risultato di tale somma. L'istruzione XLAT non modifica il valore dei flag.

L'uso più frequente dell'istruzione XLAT si ha nell'accesso a tabelle di conversione (*look-up table*), nel qual caso occorre far sì che il registro AL contenga l'indice nella tabella e che il registro BX contenga l'offset di inizio della tabella all'interno del segmento.

Esistono alcuni vincoli da rispettare affinché l'istruzione XLAT sia eseguita correttamente:

- i dati memorizzati nella tabella di conversione devono essere di tipo byte (per poter essere correttamente copiati in AL);
- il massimo numero di elementi in tabella deve essere pari a 256 (poiché essi sono indicizzati da un registro ad 8 bit).

Esercizio: Conversione da numero decimale a codifica ASCII esadecimale.

Si vuole eseguire la conversione di un numero binario di valore compreso tra 0 e 15 nel codice ASCII della corrispondente cifra esadecimale.

La soluzione proposta in linguaggio C è la seguente:

```
main()
{
    int tab[16]={0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
                0x38, 0x39, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46};
    int ascii_hex, num;
    ...
    ascii_hex = tab[num];
    ...
}
```

La soluzione proposta in linguaggio Assembler è la seguente:

```
                .MODEL    small
                .STACK
                .DATA
                                ; tabella di conversione da numero
                                ; binario a codice ASCII esadecimale
TAB            DB        30H,31H,32H,33H,34H,35H,36H,37H,38H,39H
                DB        41H,42H,43H,44H,45H,46H
NUM            DB        ?
ASCII_HEX      DB        ?
                ...
                .CODE
                ...
                LEA       BX, TAB            ; copia dell'offset di TAB in BX
                MOV       AL, NUM            ; copia di NUM in AL
                XLAT                     ; conversione
                MOV       ASCII_HEX, AL      ; copia di AL in ASCII_HEX
                ...
```

Esercizio: Conversione da numero binario a codice Gray.

Si vuole eseguire la conversione in *codifica Gray* di 10 numeri binari compresi tra 0 e 15. Si ricorda che la codifica Gray garantisce che le codifiche di numeri decimali interi consecutivi differiscano per un solo bit. In Tab. 8.1 è riportata la codifica Gray su 3 bit per i numeri interi da 0 a 7. La soluzione proposta in linguaggio C è la seguente:

```
main()
{
  int tab[16]={0x00, 0x01, 0x03, 0x02, 0x06, 0x07, 0x05, 0x04,
              0x0C, 0x0D, 0x0F, 0x0E, 0x0A, 0x0B, 0x09, 0x08};
  int Gray[10], num[10], i;
  ...
  for (i=0 ; i < 10 ; i++)
    Gray[i] = tab[num[i]];
  ...
}
```

La soluzione proposta in linguaggio Assembler è la seguente:

```
LUNG      EQU      10
           .MODEL   small
           .STACK
           .DATA

           ; tabella di conversione da
           ; numero decimale a codice Gray

TAB        DB      00000000B, 00000001B, 00000011B, 00000010B
              00000110B, 00000111B, 00000101B, 00000100B
              00001100B, 00001101B, 00001111B, 00001110B
              00001010B, 00001011B, 00001001B, 00001000B

NUM        DB      LUNG DUP (?)
GRAY       DB      LUNG DUP (?)
...
.CODE
...
LEA        SI, NUM      ; copia dell'offset di NUM in SI
LEA        DI, GRAY     ; copia dell'offset di GRAY in DI
MOV        CX, LUNG
LEA        BX, TAB      ; copia dell'offset di TAB in BX
ciclo:    MOV        AL, [SI] ; copia di NUM in AL
          XLAT        ; conversione
          MOV        [DI], AL ; copia di AL in GRAY
          INC        SI      ; scansione del vettore NUM
          INC        DI      ; scansione del vettore GRAY
          LOOP       ciclo
          ...
```

<i>Numero Decimale</i>	<i>Codifica Gray</i>
0	000
1	001
2	011
3	010
4	110
5	111
6	101
7	100

Tab. 8.1: Codifica Gray su 3 bit per i numeri decimali da 0 a 7.

8.5. Le istruzioni LDS e LES

Le istruzioni **LDS** (*Load Data Segment register*) e **LES** (*Load Extra Segment register*) permettono di copiare un indirizzo intero (indirizzo di segmento ed offset) in una coppia di registri. Il loro formato è il seguente:



<i>LDS</i>	<i>destinazione, sorgente</i>
<i>LES</i>	<i>destinazione, sorgente</i>

Le istruzioni LDS e LES hanno due operandi: un registro *destinazione* di 16 bit ed un indirizzo *sorgente*, contenente un indirizzo intero memorizzato in una doubleword (16 bit per l'offset e 16 bit per l'indirizzo di segmento). Le istruzioni LDS ed LES non modificano il valore dei flag.

L'istruzione LDS copia l'offset nel registro *destinazione* e l'indirizzo di segmento nel registro DS.

L'istruzione LES copia l'offset nel registro *destinazione* e l'indirizzo di segmento nel registro ES.

Esempi

Siano STR1_IND e STR2_IND le doubleword che contengono l'indirizzo intero rispettivamente delle variabili STR1 e STR2, come definito nei comandi seguenti:

```
STR1      DB      100 DUP(?)
STR1_IND  DD      STR1
STR2      DB      100 DUP(?)
STR2_IND  DD      STR2
```

L'istruzione seguente copia l'offset della variabile STR1 in SI e l'indirizzo di segmento in DS:

```
LDS      SI, STR1_IND
```

ed è equivalente alla seguente coppia di istruzioni:

```
LEA      SI, STR1
MOV      DS, SEG STR1
```

L'istruzione seguente copia l'offset della variabile STR2 in DI e l'indirizzo di segmento in ES:

```
LES      DI, STR2_IND
```

ed è equivalente alla seguente coppia di istruzioni:

```
LEA      DI, STR2
MOV      ES, SEG STR2
```

8.6. Le istruzioni PUSH e POP

Le istruzioni **PUSH** (*PUSH word onto stack*) e **POP** (*POP word off stack to destination*) permet-

tono di manipolare il contenuto dello stack. Il loro formato è il seguente:



PUSH *sorgente*
POP *destinazione*

Le istruzioni **PUSH** e **POP** operano su operandi di 16 bit e possono essere utilizzate per copiare nello stack il contenuto di registri *general purpose*, di registri di segmento e di locazioni di memoria. Le istruzioni **PUSH** e **POP** non modificano il valore dei flag.

L'istruzione **PUSH** decrementa il valore del registro **SP** di due unità e trasferisce una word dall'operando *sorgente* all'elemento dello stack indirizzato da **SP**.

L'istruzione **POP** trasferisce una word dall'elemento dello stack indirizzato da **SP** all'operando *destinazione* ed incrementa il registro **SP** di due unità.

Esempi

Le seguenti istruzioni sono lecite:

```

PUSH    SI
PUSH    CX
POP     SS
POP     ALPHA      ; variabile di tipo word
POP     AX

```

L'8086 non permette di eseguire l'operazione di **PUSH** con un operando immediato: tale operazione è tuttavia lecita per tutti i processori della famiglia Intel a partire dall'80186.

Esempi

La seguente istruzione genera un errore a livello di assemblatore:



```

PUSH    7

```

La stessa istruzione è lecita per un processore 80386:



```

.386
PUSH    7

```

8.7. Le istruzioni **PUSHA** e **POPA**

Le istruzioni **PUSHA** (*PUSH All registers onto stack*) e **POPA** (*POP All registers off stack*) sono disponibili solo per i processori della famiglia Intel a partire dall'80186. Si tratta di istruzioni senza operandi che eseguono le operazioni di *push* e di *pop* di tutti i registri *general purpose* (**AX**, **BX**, **CX**, **DX**, **SP**, **BP**, **SI**, **DI**). Il loro formato è il seguente:



PUSHA
POPA

All'atto dell'esecuzione dell'istruzione **PUSHA**, il processore esegue il caricamento dei registri nello stack nel seguente ordine: **AX**, **CX**, **DX**, **BX**, **SP**, **BP**, **SI** e **DI**. Il valore del registro **SP** caricato nello stack è pari al valore che tale registro aveva prima del caricamento del primo registro nello

stack. Le istruzioni **PUSHA** e **POPA** non modificano il valore dei flag.

All'atto dell'esecuzione dell'istruzione **POPA** il processore esegue il ripristino dei registri procedendo in ordine inverso rispetto a quanto effettuato dall'istruzione **PUSHA**.

Le istruzioni **PUSHA** e **POPA** sono particolarmente utili all'interno di una procedura per il salvataggio ed il ripristino del contenuto dei registri nello stack. L'uso di **PUSHA** e **POPA** è significativamente più veloce rispetto all'utilizzo di una equivalente sequenza di istruzioni di **PUSH** e **POP**.

8.8. Le istruzioni **PUSHF** e **POPF**

Le istruzioni **PUSHF** (*PUSH Flags onto stack*) e **POPF** (*POP Flags off stack*) permettono di salvare e ripristinare dallo stack i 16 bit della parola di stato del processore (PSW). Il loro formato è il seguente:



PUSHF
POPF

L'istruzione **PUSHF** decrementa di due unità il contenuto di **SP** e trasferisce nello stack il contenuto del registro **PSW**.

L'istruzione **POPF** trasferisce la parola indirizzata da **SP** nel registro **PSW** ed incrementa di due unità il valore di **SP**.

8.9. Le istruzioni **SAHF** e **LAHF**

Le due istruzioni **SAHF** (*Store AH in Flags*) e **LAHF** (*Load AH from Flags*) permettono di accedere al valore dei flag. Il loro formato è il seguente:



SAHF
LAHF

L'istruzione **SAHF** trasferisce il valore di alcuni bit del registro **AH** nei flag **SF**, **ZF**, **AF** e **CF**. L'istruzione **LAHF** trasferisce i valori dei flag **SF**, **ZF**, **AF** e **CF** nel registro **AH**. La Tab. 8.2 mostra i bit del registro **AH** coinvolti. Tali istruzioni sono state introdotte per compatibilità con i processori Intel precedenti (8080 e 8085). In pratica esse sono obsolete e sono state soppiantate, nell'uso, dalle istruzioni **PUSHF** e **POPF**.

<i>Flag</i>	<i>bit in AH</i>
SF	7
ZF	6
AF	4
PF	2
CF	0

Tab. 8.2: Bit utilizzati dalle istruzioni **SAHF** e **LAHF**.

8.10. Le istruzioni IN e OUT

Attraverso le istruzioni **IN** (*INput byte or word*) e **OUT** (*OUTput byte or word*) il processore scambia dati con le periferiche di I/O. Ad ogni dispositivo periferico connesso al microprocessore è associato un indirizzo su 16 bit. I formati delle due istruzioni di I/O sono:



<i>IN</i>	<i>registro, porta</i>
<i>OUT</i>	<i>porta, registro</i>

Il campo *registro* può essere o *AX* o *AL*; il campo *porta* è un indirizzo su 16 bit. Le istruzioni *IN* e *OUT* non modificano il valore dei flag.

L'istruzione *OUT* esegue il trasferimento del contenuto del registro accumulatore verso il dispositivo periferico specificato. L'istruzione *IN* esegue il caricamento nel registro accumulatore del dato memorizzato nel dispositivo specificato.

8.11. Le istruzioni che modificano i flag

Il processore mette a disposizione del programmatore alcune istruzioni che permettono di modificare i flag.

In generale non è opportuno modificare artificialmente i flag di stato, in quanto essi sono usati per valutare gli effetti di una istruzione. In alcuni casi particolari può essere necessario fissare il valore del flag di *carry* (*CF*). Le seguenti istruzioni permettono di lavorare sul flag di *carry*:

1. **STC** (*SeT Carry flag*): fissa *CF* al valore 1;
2. **CLC** (*CLear Carry flag*): fissa *CF* al valore 0;
3. **CMC** (*CoMplement Carry flag*): complementa il valore di *CF*.

Queste istruzioni sono senza operandi ed il loro formato è il seguente:



STC
CLC
CMC

Più importante è invece manipolare i flag di controllo *DF* (*direction flag*) e *IF* (*interrupt flag*). Per ognuno di questi flag esiste una coppia di istruzioni che permette di modificare il valore del corrispondente flag:

1. **STD** (*SeT Direction flag*): fissa *DF* al valore 1;
2. **CLD** (*CLear Direction flag*): fissa *DF* al valore 0;
3. **STI** (*SeT Interrupt flag*): fissa *IF* al valore 1;
4. **CLI** (*CLear Interrupt flag*): fissa *IF* al valore 0.

Anche queste istruzioni sono senza operandi ed il loro formato è il seguente:



STD
CLD
STI
CLI

9. Le istruzioni di controllo del flusso

In questo capitolo sono descritte le istruzioni che permettono di controllare il *flusso* di un programma, ossia l'ordine con cui vengono eseguite le istruzioni.

Le istruzioni di controllo del flusso sono raggruppabili in istruzioni di salto condizionato, di salto incondizionato, istruzioni che operano sui flag ed istruzioni che generano un ciclo.

Verranno inoltre presentate alcune regole semplici che permettono di realizzare in Assembler gli elementari costrutti di controllo del flusso di un programma disponibili nei linguaggi di alto livello.

9.1. L'istruzione di confronto: **CMP**

L'istruzione **CMP** (*CoMPare two operands*) permette di confrontare due dati eseguendo l'aggiornamento dei valori dei flag di stato. Il suo formato è il seguente:



<i>CMP</i> <i>destinazione, sorgente</i>
--

L'istruzione **CMP** tratta gli operandi come numeri ed esegue la sottrazione tra l'operando *destinazione* e l'operando *sorgente* senza restituirne il risultato, ma aggiornando opportunamente i flag di stato.

L'istruzione **CMP** permette i seguenti confronti:

- tra due registri
- tra una locazione di memoria ed un registro
- tra un valore immediato ed un registro
- tra un valore immediato ed una locazione di memoria.

Esistono alcuni vincoli da rispettare per la corretta esecuzione dell'istruzione **CMP**:

- gli operandi da confrontare devono avere la stessa lunghezza (o entrambi byte o entrambi word);
- non è ammesso il confronto tra due locazioni di memoria;

- l'operando destinazione non può essere un operando di tipo immediato, mentre può esserlo quello sorgente.

Esempi

Le seguenti istruzioni di confronto sono lecite:



```

CMP    AX, DX
CMP    SI, BP
CMP    BH, CL
CMP    WMEM, SI      ; WMEM variabile di tipo word
CMP    BMEM, CH      ; BMEM variabile di tipo byte
CMP    ALPHA[DI], DX ; ALPHA variabile di tipo word
CMP    BETA[BX][DI], 0 ; BETA variabile di tipo byte
CMP    AL, 0FH

```

Esempi

Le seguenti istruzioni di confronto non sono lecite:



```

CMP    AX, BH
CMP    WMEM1, WMEM2      ; WMEM1 e WMEM2: tipo word
CMP    7, CH

```

Per eseguire il confronto tra due locazioni di memoria, occorre dunque prima copiare il valore di una delle due locazioni in un registro e quindi eseguire il confronto tra il contenuto di tale registro ed il valore della seconda locazione di memoria.

Esempio

Le seguenti istruzioni eseguono il confronto tra il contenuto delle locazioni di memoria WMEM1 e WMEM2:



```

MOV    AX, WMEM1
CMP    AX, WMEM2

```

9.2. Le istruzioni di salto

Le istruzioni vengono normalmente eseguite sequenzialmente nell'ordine in cui compaiono nel programma; le istruzioni di salto (*jump*) forzano invece il processore ad eseguire l'istruzione che si trova in una locazione di memoria diversa da quella dell'istruzione successiva.

I salti possono essere di tipo NEAR oppure FAR a seconda che l'istruzione cui si salti appartenga o meno allo stesso segmento di codice.

Le istruzioni di salto si suddividono in istruzioni di *salto condizionato* e istruzioni di *salto incondizionato*.

Un salto è *incondizionato* se viene effettuato dal processore senza il controllo di alcuna *condizione*. Viceversa, un salto *condizionato* viene effettuato solo se una determinata *condizione* relativa ai flag di stato è *vera*.

A livello di processore, il meccanismo di salto funziona nei seguenti modi:

- per salti di tipo NEAR, il processore copia nell'*Instruction Pointer* l'offset dell'istruzione a cui si deve saltare;
- per salti di tipo FAR, il processore modifica sia il contenuto dell'*Instruction Pointer* sia il contenuto del registro di segmento CS, scrivendo in essi i valori corrispondenti all'indirizzo

(offset e segmento) dell'istruzione a cui saltare.

9.2.1. Le istruzioni di salto condizionato

Il formato delle istruzioni di salto condizionato è il seguente:



Jxxx etichetta

dove *xxx* è un suffisso che specifica la condizione sui flag.

Il flusso di esecuzione delle istruzioni dipende dal risultato della condizione: se la condizione è *vera*, il processore continua l'esecuzione saltando all'istruzione etichettata dalla *etichetta*, altrimenti l'esecuzione del programma continua con la successiva istruzione in sequenza.

Le istruzioni di salto condizionato si dividono in tre gruppi:

- quelle in cui la condizione è relativa ad un singolo flag
- quelle in cui la condizione è relativa al risultato di un confronto
- quelle in cui la condizione riguarda il contenuto del registro CX.

Le istruzioni di salto dipendenti da un singolo flag

Per ogni flag esistono due istruzioni di salto condizionato: una che esegue il salto se il flag vale 1 ed una che esegue il salto se il flag vale 0.

La Tab. 9.1 riassume le istruzioni di salto condizionato che dipendono dai flag. Nel caso del flag PF esistono due coppie di istruzioni equivalenti. Non esiste alcuna istruzione di salto associata al valore del flag AF (*auxiliary flag*). Le istruzioni per l'aritmetica BCD usano il flag AF senza richiedere alcun intervento esplicito da parte del programmatore (vedi Cap. 10).

<i>Opcode</i>	<i>Significato</i>
JZ	Salta se ZF = 1
JNZ	Salta se ZF = 0
JS	Salta se SF = 1
JNS	Salta se SF = 0
JO	Salta se OF = 1
JNO	Salta se OF = 0
JC	Salta se CF = 1
JNC	Salta se CF = 0
JP o JPE	Salta se PF = 1
JNP o JPO	Salta se PF = 0

Tab. 9.1: Istruzioni di salto condizionato che testano i flag.

Le istruzioni di salto dipendenti dal risultato di un confronto

Una situazione molto frequente all'interno di un programma è quella in cui bisogna eseguire istruzioni diverse a seconda del risultato di un'operazione di confronto:



CMP destinazione, sorgente
Jxxx etichetta

Anche queste istruzioni di salto condizionato verificano una condizione sui flag di stato; in generale tale condizione coinvolge più di un flag.

Esistono due insiemi di istruzioni di salto condizionato dipendenti dal risultato di un confronto destinati, rispettivamente, al confronto tra numeri con segno e tra numeri senza segno.

Il confronto tra caratteri è riconducibile a quello tra numeri senza segno.

Il processore controlla flag diversi a seconda del tipo di istruzione.

In Tab. 9.2 sono indicati i nomi delle istruzioni di salto condizionato che controllano il risultato di un confronto tra numeri con segno.

<i>Opcode</i>	<i>Significato</i>
JL o JNGE	Salta se <i>destinazione</i> < <i>sorgente</i>
JG o JNLE	Salta se <i>destinazione</i> > <i>sorgente</i>
JLE o JNG	Salta se <i>destinazione</i> ≤ <i>sorgente</i>
JGE o JNL	Salta se <i>destinazione</i> ≥ <i>sorgente</i>
JE	Salta se <i>destinazione</i> = <i>sorgente</i>
JNE	Salta se <i>destinazione</i> ≠ <i>sorgente</i>

Tab. 9.2: Istruzioni di salto condizionato che controllano il risultato di un confronto tra numeri con segno.

In Tab. 9.3 sono indicati i nomi delle istruzioni di salto condizionato dipendenti dal risultato di un confronto tra numeri senza segno. Si noti che alcune di queste istruzioni sono equivalenti ad istruzioni della Tab. 9.1, ad esempio **JE** è equivalente a **JZ**.

<i>Opcode</i>	<i>Significato</i>
JB o JNAE	Salta se <i>destinazione</i> < <i>sorgente</i>
JA o JNBE	Salta se <i>destinazione</i> > <i>sorgente</i>
JBE o JNA	Salta se <i>destinazione</i> ≤ <i>sorgente</i>
JAE o JNB	Salta se <i>destinazione</i> ≥ <i>sorgente</i>
JE	Salta se <i>destinazione</i> = <i>sorgente</i>
JNE	Salta se <i>destinazione</i> ≠ <i>sorgente</i>

Tab. 9.3: Istruzioni di salto condizionato dipendenti dal risultato di un confronto tra numeri senza segno.

L'istruzione di salto condizionato dal contenuto del registro CX

L'istruzione **JCXZ** (*Jump if CX is Zero*) controlla il contenuto del registro CX. Il suo formato è il seguente:

	<i>JCXZ</i>	<i>etichetta</i>
---	-------------	------------------

L'istruzione esegue il salto all'istruzione individuata tramite *etichetta* se il contenuto del registro CX è pari a 0, altrimenti esegue l'istruzione successiva nel codice.


In Tab. 9.4 sono riepilogate tutte le istruzioni di salto condizionato, ed in particolare sono evidenziate le condizioni che il processore valuta per decidere se eseguire o meno il salto.

<i>Opcode</i>	<i>Flag o Registri testati</i>
JA o JNBE	CF = 0 e ZF = 0
JAE o JNB	CF = 0
JB o JNAE	CF = 1
JBE o JNA	CF = 1 o ZF = 1
JC	CF = 1
JCXZ	CX = 0
JE	ZF = 1
JG o JNLE	ZF = 0 e SF = OF
JGE o JNL	SF = OF
JL o JNGE	(SF = 1 e OF = 0) o (SF = 0 e OF = 1)
JLE o JNG	ZF = 1 o (SF = 1 e OF = 0) o (SF = 0 e OF = 1)
JNC	CF = 0
JNE	ZF = 0
JNO	OF = 0
JNP o JPO	PF = 0
JNS	SF = 0
JNZ	ZF = 0
JO	OF = 1
JP o JPE	PF = 1
JS	SF = 1
JZ	ZF = 1

Tab. 9.4: Riepilogo delle istruzioni di salto condizionato.

9.2.2. L'istruzione di salto incondizionato: **JMP**

Un salto incondizionato è un salto che viene sempre eseguito, senza il controllo di alcuna condizione. Per eseguire un salto incondizionato si utilizza l'istruzione **JMP** (*jump*), il cui formato è il seguente:

	<i>JMP</i>	<i>destinazione</i>
---	------------	---------------------

L'operando *destinazione* contiene l'indirizzo dell'istruzione a cui il processore deve saltare, espresso sotto forma di etichetta di una istruzione o di indirizzamento indiretto.

Esempi

Le istruzioni **JMP** incluse nel frammento di codice seguente separano parti di codice che non devono essere eseguite in sequenza:

```

label1:    ...           ; caso 1
           JMP    continua ; salto del blocco seguente
label2:    ...           ; caso 2
           JMP    continua
           ...
continua:  ...

```

Il frammento di codice seguente realizza un ciclo di istruzioni ripetuto 10 volte:

```

RIP      EQU      10
...
lab1:    MOV      COUNT, RIP
        CMP      COUNT, 0
        JNG      lab2      ; while (COUNT > 0) do
...      ; istruzioni
        DEC      COUNT      ; COUNT = COUNT - 1
        JMP      lab1
lab2:    ...

```

L'istruzione `JMP` permette di gestire due tipi di salti: *diretti* o *indiretti*.

Salti diretti

Nei salti diretti l'indirizzo *destinazione* specifica l'indirizzo a cui saltare; ne esistono tre tipi: *short*, *near* e *far*.

Nei salti di tipo *short* e di tipo *near* l'istruzione macchina è codificata in modo che la differenza tra il contenuto attuale dell'`IP` e l'offset dell'istruzione a cui saltare sia contenuta rispettivamente in un byte od in una word.

Nei salti di tipo *far* l'istruzione macchina è codificata in modo da contenere su due word l'indirizzo intero dell'istruzione a cui saltare (offset e registro di segmento). I salti di tipo *far* causano la modifica sia del registro `IP` sia del registro di segmento `CS`.

Salti indiretti

Nei salti indiretti l'indirizzo *destinazione* non specifica l'indirizzo, ma fornisce un puntatore all'indirizzo a cui saltare.

L'indirizzo può essere contenuto in un registro o in una variabile o in una tabella cui si accede tramite un indice.

Esempi

Nella prima istruzione il registro `AX` contiene l'indirizzo dell'istruzione a cui saltare. Nella seconda la variabile `WVAR` contiene l'indirizzo dell'istruzione a cui saltare. Nella terza istruzione per calcolare l'indirizzo della locazione di memoria che contiene l'indirizzo a cui saltare il processore calcola la somma tra l'offset della variabile `TABLE` ed il contenuto del registro `BX`.

```

JMP      AX
JMP      WVAR      ; variabile di tipo word
JMP      TABLE[BX]

```

I salti indiretti possono essere utilizzati per implementare i costrutti di tipo *CASE* (vedi oltre).

9.2.3. Codifica degli indirizzi in istruzioni di salto

L'assemblatore impone delle regole per la codifica degli indirizzi diverse per l'istruzione `JMP` e per le istruzioni di salto condizionato.

Codifica dell'indirizzo di salto in `JMP`

Come visto nel paragrafo precedente, l'assemblatore permette la massima libertà nel tipo di salto effettuato attraverso l'istruzione `JMP`.

Nel caso di salti diretti l'assemblatore ottimizza il tipo di salto generando l'istruzione macchina opportuna. Esso calcola innanzitutto la distanza dell'istruzione *destinazione* da quella corrente e se questa è minore di 128 byte genera un'istruzione macchina di salto di tipo *short*; se l'istruzione

destinazione dista più di 128 byte, ma appartiene allo stesso segmento genera un'istruzione macchina di salto di tipo *near*; altrimenti genera un'istruzione macchina di salto di tipo *far*.

Codifica dell'indirizzo di salto nelle istruzioni di salto condizionato

Un'istruzione di salto condizionato può eseguire un salto solo ad indirizzi di tipo *near* che distino al più 128 byte dall'istruzione di corrente. Per ragioni di efficienza, l'indirizzo dell'operando *destinazione* è codificato in linguaggio macchina in un unico byte, nel quale è memorizzato un numero in complemento a 2 che indica la distanza in byte tra l'istruzione successiva a quella di salto e l'istruzione *destinazione*.

Una buona regola di programmazione prevede che si effettuino salti ad indirizzi ravvicinati all'interno del codice e dunque la limitazione sulla distanza non pone problemi. Ciononostante è possibile che particolari condizioni richiedano di effettuare un salto condizionato ad un'istruzione localizzata ad una distanza superiore ai 128 byte. In questo caso si deve o riscrivere il codice in modo da avvicinare l'istruzione a cui saltare o fare in modo di effettuare il salto all'indirizzo lontano mediante un salto incondizionato.

Esempio

L'istruzione seguente esegue un salto all'istruzione con etichetta `INV_DATA` se il valore del flag `CF` è pari ad 1. Se tale istruzione è posta ad una distanza superiore a 127 byte, l'assemblatore rileva un errore.



```
CONTROLLO:  JC      INV_DATA
            ...
INV_DATA:   ...           ; tra CONTROLLO ed INV_DATA la
                        ; distanza supera i 128 byte
```

Il codice seguente permette di risolvere il problema:



```
CONTROLLO:  JNC     GOOD_DATA
            JMP     INV_DATA
GOOD_DATA:  ...
INV_DATA:   ...
```

Esercizio: Ricerca del massimo in un vettore di numeri positivi.

Si vuole ricercare il massimo all'interno di un vettore composto da 10 numeri positivi. L'equivalente programma in linguaggio C è il seguente:

```
#define LUNG 10
main()
{
    int i, massimo = -1;
    unsigned int vett[LUNG];
    ...
    for (i=0 ; i < LUNG ; i++)
        if (vett[i] > massimo)
            massimo = vett[i];
    ...
}
```

La soluzione proposta in linguaggio Assembler è la seguente:

```

LUNG      EQU      10
          .MODEL   small
          .DATA
VETT      DW      LUNG DUP (?)
MASSIMO   DW      ?
          .CODE
          ...
          LEA      SI, VETT      ; copia dell'offset di VETT in SI
          MOV      AX, 0          ; in AX viene memorizzato
                                   ; il massimo valore temporaneo
ciclo:    MOV      CX, LUNG
          CMP      AX, [SI]      ; VETT[SI] >= AX ?
          JA       scans        ; No: va a scans
          MOV      AX, [SI]      ; Sì: aggiornamento del massimo
scans:    ADD      SI, 2
          LOOP     ciclo
          MOV      MASSIMO, AX   ; copia di AX nella variabile MASSIMO
          ...

```

Esercizio: Calcolo del numero di lettere minuscole in una stringa.

Si vuole calcolare il numero di lettere minuscole presenti all'interno di una stringa di caratteri. Il frammento di programma controlla il codice ASCII di ciascun carattere della stringa e verifica se è compreso all'interno dell'insieme dei codici delle lettere minuscole.

La soluzione proposta in linguaggio C è la seguente:

```

#define LUNG 20
main()
{ int i;
  char vett[LUNG], minuscole = 0;
  ...
  for (i=0 ; i< LUNG ; i++)
    if ((vett[i] >= 'a') && (vett[i] <= 'z')) minuscole++;
  ...
}

```

La soluzione proposta in linguaggio Assembler è la seguente:

```

LUNG      EQU      20
CAR_A     EQU      "a"
CAR_Z     EQU      "z"
          .MODEL   small
          .DATA
VETT      DB      LUNG DUP (?)
MINUSCOLE DW      ?
          .CODE
          ...
          MOV      SI, 0
          MOV      AX, 0          ; AX: numero di caratteri minuscoli
          MOV      CX, LUNG
ciclo:    CMP      VETT[SI], CAR_A ; VETT[SI] >= 'a' ?
          JB       salta        ; No: va a salta
          CMP      VETT[SI], CAR_Z ; Sì: VETT[SI] <= 'z' ?
          JA       salta        ; No: va a salta
          INC      AX            ; Sì: carattere minuscolo
salta:    INC      SI
          LOOP     ciclo
          MOV      MINUSCOLE, AX
          ...

```

Esercizio: Conteggio dei numeri positivi e dei numeri negativi in un vettore.

Si vuole calcolare il numero di elementi positivi ed il numero di elementi negativi compresi in una tabella di numeri interi. La soluzione proposta in linguaggio C è la seguente:

```
#define LUNG 10
main()
{
    int i, count_neg = 0, count_pos = 0, vett[LUNG];
    ...
    for (i=0 ; i< LUNG ; i++)
        if (vett[i] >= 0)
            count_pos++;
        else count_neg++;
    ...
}
```

La soluzione proposta in linguaggio Assembler è la seguente:

```
LUNG      EQU      10
           .MODEL   small
           .DATA
VETT       DB      LUNG DUP (?)
COUNT_POS DW      ?
COUNT_NEG DW      ?
           .CODE
           ...
           MOV      SI, 0
           MOV      AX, 0           ; contatore dei numeri positivi
           MOV      BX, 0           ; contatore dei numeri negativi
           MOV      CX, LUNG
ciclo:     CMP      VETT[SI], 0     ; VETT[SI] > 0 ?
           JGE      pos
           INC      BX              ; No: numero negativo
           JMP      continua
pos:       INC      AX              ; Si: numero positivo
continua:  INC      SI
           LOOP     ciclo
           MOV      COUNT_POS, AX  ; copia di AX nella variabile COUNT_POS
           MOV      COUNT_NEG, BX  ; copia di BX nella variabile COUNT_NEG
           ...
```


Esercizio: Somma degli elementi di un vettore con verifica della correttezza del risultato.

Si vuol realizzare un programma che valuta se un'operazione di somma tra numeri ha generato un errore. Si supponga di avere una tabella di 50 byte in cui sono memorizzati numeri interi positivi; i numeri devono essere sommati tra loro e, se la somma genera un *overflow*, tutti i dati devono essere azzerati. La soluzione proposta in linguaggio C è la seguente:

```
#define LUNG 50
main()
{
    int i, j;
    char vett[LUNG], somma = 0;
    ...
    for (i=0 ; i< LUNG ; i++)
        { old_somma = somma;
          somma += vett[i];
          if (somma < old_somma)
              { for (j=0; j<LUNG ; j++) vett[j] = 0;
                return;
              }
        }
}
```

La soluzione proposta in linguaggio Assembler è la seguente:

```
LUNG      EQU      50
           .MODEL   small
           .STACK
           .DATA
VETT       DB      LUNG DUP(?)
           .CODE
           ...
           LEA      SI, VETT          ; copia in SI dell'offset di VETT
           MOV      AX, 0             ; Azzeramento dei registri AH e AL
           MOV      CX, LUNG
ciclo:     ADD      AH, [SI]          ; AH = AH + [SI]
           JO       azzerata         ; flag CF = 1 ? Sì: overflow
           INC      SI               ; No: scansione del vettore
           LOOP     ciclo
           JMP      esci
azzerata:  LEA      SI, VETT          ; inizializzazione del registro SI
           MOV      CX, LUNG         ; inizializzazione del registro CX
ciclo2:    MOV      [SI], AL         ; in AL c'è 0: azzeramento del vettore
           INC      SI               ; scansione del vettore
           LOOP     ciclo2
esci:      ...
```

9.3. Le istruzioni che gestiscono una sequenza

L'architettura x86 mette a disposizione alcune istruzioni per l'implementazione di determinate strutture di controllo.

9.3.1. L'istruzione LOOP

L'istruzione **LOOP** (*LOOP until count complete*) permette di ripetere per un numero definito di volte una certa sequenza di istruzioni. Il suo formato è il seguente:



LOOP

etichetta

All'atto dell'esecuzione dell'istruzione `LOOP`, il processore esegue le seguenti operazioni:

- decrementa di una unità il contenuto del registro `CX`;
- esegue il controllo sul contenuto del registro `CX`:
 - se il valore di `CX` è diverso da 0, salta all'istruzione avente etichetta *etichetta*;
 - altrimenti esegue l'istruzione successiva.

L'istruzione `LOOP` è normalmente usata per eseguire un numero predefinito di volte una sequenza di istruzioni (*cicli o loop*). Le operazioni da effettuare per realizzare un ciclo sono le seguenti:

- caricare nel registro `CX` il numero di volte per cui il ciclo deve essere ripetuto;
- associare un'etichetta alla prima istruzione del ciclo;
- chiudere il ciclo con un'istruzione di `LOOP` che esegua un salto all'inizio del ciclo.

Esempio

```

label:      MOV      CX, NUMERO      ; NUMERO: variabile di tipo word
            ...
            ...
            LOOP     label          ; --CX, Se CX ≠ 0 va a label
            ...

```

9.3.2. Le istruzioni `LOOPE`, `LOOPZ`, `LOOPNE` e `LOOPNZ`

Le istruzioni **`LOOPE`** (*LOOP if Equal*) e **`LOOPNE`** (*LOOP if Not Equal*) permettono di gestire cicli più sofisticati rispetto a quelli che si ottengono con l'istruzione `LOOP`. Le istruzioni **`LOOPZ`** (*LOOP if Zero*) e **`LOOPNZ`** (*LOOP if Not Zero*) sono equivalenti rispettivamente a `LOOPE` e `LOOPNE`. Il formato delle istruzioni `LOOPE`, `LOOPZ`, `LOOPNE` e `LOOPNZ` è il seguente:



```

LOOPE      etichetta
LOOPZ      etichetta
LOOPNE     etichetta
LOOPNZ     etichetta

```

All'atto dell'esecuzione dell'istruzione `LOOPE` (o `LOOPZ`), il processore esegue le seguenti operazioni:

- decrementa di un'unità il contenuto del registro `CX`;
- esegue il controllo sul contenuto del registro `CX` e sul valore del flag `ZF`:
 - se il valore di `CX` è diverso da 0 e il flag `ZF` è uguale a 1, salta all'istruzione avente etichetta *etichetta*;
 - se il valore di `CX` è uguale a 0 oppure il flag `ZF` è uguale a 0 esegue l'istruzione successiva.

All'atto dell'esecuzione dell'istruzione `LOOPNE` (o `LOOPNZ`), il processore esegue le seguenti operazioni:

- decrementa di una unità il contenuto del registro `CX`;
- esegue il controllo sul contenuto del registro `CX` e sul valore del flag `ZF`:
 - se il valore di `CX` è diverso da 0 e il flag `ZF` è uguale a 0, salta all'istruzione avente etichetta *etichetta*;
 - se il valore di `CX` è uguale a 0 oppure il flag `ZF` è uguale a 1 esegue l'istruzione successiva.

Le due istruzioni vengono usate subito dopo un'istruzione di `CMP` che setta il flag `ZF`.

Esempi

Il frammento di codice seguente utilizza l'istruzione `LOOPE`: il ciclo viene ripetuto fino a che il valore del registro `CX` è diverso da zero ed il registro `AX` e la variabile `VAL` sono uguali.

```
lab1:      MOV      CX, NUM      ; NUM: variabile di tipo word
          ...                ; istruzioni che costituiscono il ciclo
          CMP      AX, VAL      ; AX = VAL ?
          LOOPE    lab1        ; se CX ≠ 0 e AX = VAL: va a lab1
          ...
```

Il frammento di codice seguente utilizza l'istruzione `LOOPNE`: il ciclo viene ripetuto fino a che il valore del registro `CX` è diverso da zero ed il registro `AX` e la variabile `VAL` sono diversi.

```
lab1:      MOV      CX, NUM      ; NUM: variabile di tipo word
          ...                ; istruzioni che costituiscono il ciclo
          CMP      AX, VAL      ; AX = VAL ?
          LOOPNE   lab1        ; se CX ≠ 0 e AX ≠ VAL: va a lab1
          ...
```

La Tab. 9.5 riassume il funzionamento delle istruzioni per la gestione dei cicli.

<i>Opcod</i>	<i>Operazione</i>	<i>Salto</i>
LOOP	Esegue il ciclo	se $CX \neq 0$
LOOPE o LOOPZ	Esegue il ciclo se il flag <code>ZF</code> è uguale a 1	se $CX \neq 0$ e $ZF = 1$
LOOPNE o LOOPNZ	Esegue il ciclo se il flag <code>ZF</code> è uguale a 0	se $CX \neq 0$ e $ZF = 0$

Tab. 9.5: Istruzioni che gestiscono un ciclo.

Esercizio: Ricerca di un numero all'interno di una tabella (I versione).

Si voglia cercare all'interno di una tabella di interi la prima occorrenza del valore `-1`.

Il programma scandisce il vettore con un ciclo che termina non appena trova il numero ricercato. La soluzione proposta in linguaggio C è la seguente:

```
#define LUNG 100
main()
{
    int i;
    int vett[LUNG];
    ...
    for (i = 0 ; i < LUNG ; i++)
        if (vett[i] == -1) break;
    ...
}
```

La soluzione proposta in linguaggio Assembler è la seguente:

```

LUNG      EQU      100
          .MODEL   small
          .DATA
VETT      DW       LUNG DUP (?)
          .CODE
          ...
          MOV      SI, -2          ; indice del vettore inizializzato a -2
          MOV      CX, LUNG
ciclo:    ADD      SI, 2          ; scansione del vettore
          CMP      VETT[SI], -1 ; VETT[SI] = -1 ?
          LOOPNE   ciclo          ; No e CX ≠ 0: va all'istruzione ciclo
          ...

```

L'istruzione che aggiorna il valore dell'indice è stata anticipata per fare in modo che l'istruzione di confronto sia l'ultima istruzione a modificare i flag prima di LOOPNE. Il primo valore valido dell'indice è 0 e dunque il valore di inizializzazione deve essere posto a -2.

Esercizio: Ricerca di un numero all'interno di una tabella (II versione).

Si vuole realizzare un programma che scandisca una tabella alla ricerca del primo valore diverso da 0. La soluzione proposta in linguaggio C è la seguente:

```

#define LUNG 100
main()
{
    int i;
    int vett[LUNG];
    ...
    for (i = 0 ; i < LUNG ; i++)
        if (vett[i]) break;
    ...
}

```

La soluzione proposta in linguaggio Assembler è la seguente:

```

          .MODEL   small
          .DATA
VETT      DW       LUNG DUP (?)
          .CODE
          ...
          MOV      SI, -2          ; indice del vettore inizializzato a -2
          MOV      CX, LUNG
ciclo:    ADD      SI, 2          ; scansione del vettore
          CMP      VETT[SI], 0    ; VETT[SI] = 0 ?
          LOOPE    ciclo          ; Sì e CX ≠ 0: va all'istruzione ciclo
          ...

```

9.4. I costrutti per il controllo del flusso

In questo paragrafo verranno presentate le sequenze di istruzioni Assembler che implementano i costrutti per il controllo del flusso di un programma.

9.4.1. Costrutto IF-THEN

Il costrutto *IF-THEN* permette di creare un'alternativa nel flusso di esecuzione di un programma. Il costrutto si basa su una *condizione* ed una *sequenza* di istruzioni; la *sequenza* è eseguita

se la *condizione* è verificata. In linguaggio C il costrutto *IF-THEN* è il seguente:



```
if (condizione)
    sequenza;
```

In Assembler la *condizione* è relativa al valore dei flag di stato o del registro CX.

L'implementazione in linguaggio Assembler avviene quindi nei seguenti passi:

1. esecuzione di un'istruzione che modifica il valore dei flag o del registro CX;
2. valutazione della *condizione*;
 - a. se la *condizione* è verificata: esecuzione della *sequenza* di istruzioni
 - b. altrimenti: salto per evitare la sequenza.

In linguaggio Assembler una possibile implementazione del costrutto *IF-THEN* è la seguente:



```
CMP      op1, op2      ; confronto tra op1 e op2
JNcond   cont          ; cond verificata ? No: va a cont
...      ; Sì: ramo then
cont:
```

Tale frammento di codice esegue il confronto tra gli operandi *op1* e *op2*; l'istruzione di salto verifica una espressione relazionale tra gli operandi: se la condizione è verificata viene eseguita la sequenza di istruzione successiva (*ramo then*), altrimenti viene eseguito un salto all'istruzione avente etichetta *cont*.

Esercizio: Azzeramento dei numeri negativi di un vettore.

Si vuole realizzare un programma che annulli tutti i numeri negativi in un vettore di numeri interi. La soluzione in linguaggio C è la seguente:

```
#define LUNG 20
main()
{ int i, vett[LUNG];
  ...
  for (i=0 ; i<LUNG ; i++)
    if (vett[i] < 0)
      vett[i] = 0;
  ...
}
```

La soluzione proposta in linguaggio Assembler è la seguente:

```
LUNG      EQU      20
           .MODEL   small
           .DATA
VETT      DW        LUNG DUP (?)
           .CODE
           ...
           MOV      SI, 0
           MOV      CX, LUNG
ciclo:    CMP      VETT[SI], 0 ; VETT[SI] < 0 ?
           JNL      label      ; No: va a label
           MOV      VETT[SI], 0 ; Sì: Azzeramento dell'elemento
label:    ADD      SI, 2        ; Scansione del vettore
           LOOP     ciclo
           ...
```

9.4.2. Costrutto *IF-THEN-ELSE*

Il costrutto *IF-THEN-ELSE* permette la gestione di due possibili blocchi di istruzioni che vengono eseguiti in alternativa in base al risultato di una condizione.

In linguaggio C il formato del costrutto *IF-THEN-ELSE* è il seguente:



```
if (condizione)
    sequenza1;
else
    sequenza2;
```

Se la *condizione* è vera il processore esegue il blocco di istruzioni *sequenza1* (*ramo then*), altrimenti esegue il blocco *sequenza2* (*ramo else*).

Una possibile implementazione del costrutto *IF-THEN-ELSE* in Assembler è la seguente:



```

CMP      op1, op2      ; confronto tra op1 e op2
Jcond    lab1          ; cond è vera ? Si: va a lab1
...      ; No: ramo else
JMP      continua
lab1:    ...            ; ramo then
continua:
```

Tale frammento di codice esegue il confronto tra gli operandi *op1* e *op2*; l'istruzione di salto verifica una espressione relazionale tra gli operandi: se la condizione è verificata viene eseguita la sequenza di istruzione avente etichetta *lab1* (*ramo then*), altrimenti viene eseguito l'istruzione successiva (*ramo else*).

Esercizio: *Numero positivo o numero negativo?*

Si vuole realizzare un frammento di programma che, dato un vettore di 10 numeri interi, visualizzi un messaggio diverso a seconda che il numero sia positivo o negativo.

Una possibile soluzione in linguaggio C è la seguente:

```
#include <stdio.h>
#define LUNG 10
main()
{ int i;
  int vett[LUNG];
  ...
  for (i=0 ; i<LUNG ; i++)
    if (vett[i] > 0) printf("Numero positivo\n");
    else printf("Numero negativo\n");
  ...
}
```

La soluzione proposta in linguaggio Assembler è la seguente:

```

LUNG      EQU      10
          .MODEL   small
          .DATA
MSG1      DB       "Numero Negativo",0Dh,0Ah,"$"
MSG2      DB       "Numero Positivo",0Dh,0Ah,"$"
VETT      DW       LUNG DUP (?)
          .CODE
          ...
          MOV      SI, 0
          MOV      CX, LUNG
ciclo:    CMP      VETT[SI], 0      ; VETT[SI] > 0 ?
          JNG      lab1            ; No: va a lab1
          LEA      DX, MSG2        ; Sì: carica in DX l'offset di MSG2
          JMP      lab2
lab1:     LEA      DX, MSG1        ; carica in DX l'offset di MSG1
lab2:     MOV      AH, 09h         ; visualizzazione su video
          INT      21h
          ADD      SI, 2           ; scansione del vettore
          LOOP     ciclo
          ...

```

9.4.3. Costrutto CASE

Il costrutto *CASE* è costituito da diversi blocchi di istruzioni; ogni blocco è eseguito se una determinata espressione assume un particolare valore.

In linguaggio C il formato del costrutto *CASE* è il seguente:



```

switch (espressione)
{
case val1: sequenza1;
           break;
case val2: sequenza2;
           break;
...
default:   sequenza_def;
}

```

Il costrutto *CASE* è organizzato nelle seguenti fasi:

- valutazione dell'*espressione*
- se l'espressione è uguale a *val1*
 - esecuzione del blocco di istruzioni *sequenza1* ed uscita dal costrutto;
- se l'espressione è uguale a *val2*
 - esecuzione del blocco di istruzioni *sequenza2* ed uscita dal costrutto;
- ...
- se nessuna uguaglianza è verificata
 - esecuzione del blocco di istruzioni *sequenza_def*.

Una possibile implementazione in linguaggio Assembler è la seguente:



```

    CMP      cond1      ; cond1 è vera?
    Jcond1   lab_1      ; Sì: va a lab_1
    CMP      cond2      ; No: cond2 è vera?
    Jcond2   lab_2      ; Sì: va a lab_2
    CMP      cond3      ; No: cond3 è vera?
    Jcond3   lab_3      ; Sì: va a lab_3
    JMP      lab_4      ; No: va a lab_4
lab_1: sequenza1      ; ramo 1
    JMP      cont       ; break
lab_2: sequenza2      ; ramo 2
    JMP      cont       ; break
lab_3: sequenza3      ; ramo 3
    JMP      continua   ; break
lab_4: sequenza4      ; ramo di default
cont: ...

```

Esercizio: Quale carattere è stato premuto su tastiera?

Si vuole realizzare un frammento di codice che legga da tastiera un carattere e visualizzi un messaggio diverso a seconda del carattere letto. Una possibile implementazione in C è la seguente:

```

#include <stdio.h>
main()
{ char c;
  scanf("%c",&c);
  switch(c)
  {
    case 'a': printf("carattere a\n");
              break;
    case 'b': printf("carattere b\n");
              break;
    default:  printf("né a né b\n");
  }
}

```

Per poter risolvere questo esercizio in Assembler occorre introdurre la *function call* MS-DOS che permette di leggere un carattere da tastiera. Tale *function call* si attiva caricando il valore 08H nel registro AH; dopo l'esecuzione dell'istruzione INT 21H, il registro AL contiene il codice ASCII del carattere letto da tastiera.

Una possibile soluzione in linguaggio Assembler è la seguente:


```

.MODEL    small
.DATA
MSG1      DB      "carattere a",0Dh,0Ah,"$"
MSG2      DB      "carattere b",0Dh,0Ah,"$"
MSG3      DB      "né a né b",0Dh,0Ah,"$"
.CODE
...
MOV       AH, 08h
INT       21h          ; legge un carattere da tastiera
ciclo:    CMP       AL, "a"      ; carattere = 'a' ?
          JNE       lab1        ; No: va a lab1
          LEA       DX, MSG1     ; Sì: carica in DX l'offset di MSG1
          JMP       lab3        ; break
lab1:     CMP       AL, "b"      ; carattere = 'b' ?
          JNE       lab2        ; No: va a lab2
          LEA       DX, MSG2     ; Sì: carica in DX l'offset di MSG2
          JMP       lab3        ; break
lab2:     LEA       DX, MSG3     ; default: carica in DX l'offset di MSG3
lab3:     MOV       AH, 09h      ; visualizza su video
          INT       21h
          ...

```

Nel caso in cui i valori possibili del *CASE* siano consecutivi è possibile utilizzare le cosiddette *tabelle di jump*, contenenti gli indirizzi delle locazioni di memoria a cui saltare. Il processore fa accesso alla tabella e salta all'indirizzo in essa contenuto.

Esempio

Il seguente codice C mostra un esempio di costruito di tipo *CASE*:

```

switch(var)
{
    case '1': codice_1;
              break;
    case '2': codice_2;
              break;
    case '3': codice_3;
              break;
}

```

La soluzione proposta in linguaggio Assembler è la seguente:

```

TAB      .DATA
        DW    lab_1
        DW    lab_2
        DW    lab_3
        .CODE
        ...
        DEC    VAR          ; decremento il valore di VAR
        MOV    BX, VAR      ; BX assume un valore compreso
                             ; tra 0 (VAR = 1) e 2 (VAR = 3)
        SHL    BX, 1        ; shift a sinistra di una posizione di BX
                             ; equivalente a moltiplicare per 2
        JMP    TAB[BX]      ; il processore esegue il salto ad uno
                             ; tra gli indirizzi contenuti in TABELLA
                             ; a seconda del valore di BX

lab_1:   ...
        JMP    continue
lab_2:   ...
        JMP    continue
lab_3:   ...
        JMP    continue
        ...
continue:

```

Per impedire errori occorre verificare che il valore assunto dall'operando del *CASE* sia incluso all'interno dell'intervallo previsto; in caso negativo bisogna impedire che venga effettuato un salto ad un indirizzo fuori dalla *tabella di jump*.

9.4.4. Costrutto *REPEAT-UNTIL*

Il costrutto *REPEAT-UNTIL* permette la gestione di un ciclo. In linguaggio C il costrutto *REPEAT-UNTIL* è implementato nel modo seguente:



```

do
    sequenza;
while (condizione);

```

La sequenza di operazioni è la seguente:

1. esecuzione della *sequenza* di istruzioni
2. verifica della *condizione*:
 - a. se la *condizione* è vera salta al passo 1
 - b. altrimenti si esce dal ciclo.

In Assembler è possibile realizzare due diverse implementazioni del costrutto *REPEAT-UNTIL*, che differiscono per la condizione di terminazione:



```

lab:      sequenza
        CMP    cond          ; cond è vera ?
        Jcond  lab          ; Sì: ripete il ciclo

```



```

lab:      sequenza
        CMP    cond          ; cond è falsa ?
        JNcond lab          ; Sì: ripete il ciclo

```

Esercizio: Lettura di caratteri da tastiera fino a leggere il carattere 'r'.

Si vuole realizzare un frammento di programma che legga da tastiera un carattere fino a che non viene letto il carattere 'r'. Una possibile soluzione in linguaggio C è la seguente:

```
main()
{ char c;
  ...
  do
    scanf("%c",&c);
  while (c != 'r');
  ...
}
```

Una possibile soluzione in linguaggio Assembler è la seguente:

```
        .MODEL    small
        .CODE
        ...
ciclo:  MOV     AH, 01h
        INT     21H      ; lettura del carattere da tastiera
        CMP     AL, "r"   ; carattere = 'r' ?
        JNE     ciclo     ; No: ripete il ciclo
        ...             ; Sì: esce dal ciclo
```

Per la lettura di un carattere da tastiera è stata utilizzata la *function call* DOS avente *call number* 01H: essa è analoga a quella avente *call number* 08H, con la differenza che il carattere letto da tastiera viene visualizzato su video (*eco* del carattere).

9.4.5. Costrutto *FOR*

Il costrutto *FOR* permette di ripetere una sequenza di istruzioni per un numero prefissato di volte.

In linguaggio C un possibile formato del costrutto *FOR* è il seguente:



```
int i;
...
for (i=0 ; i < numero ; i++)
    sequenza;
```

In linguaggio Assembler l'istruzione *LOOP* permette di implementare agevolmente il costrutto *FOR*; un esempio di implementazione è il seguente:



```
ciclo:  MOV     CX, numero
        sequenza
        LOOP    ciclo      ; decremento di CX, CX = 0 ?
                                ; No: va a ciclo
        ...             ; Sì: esce dal ciclo
```

La sequenza di operazioni è la seguente:

1. caricamento in CX del numero di ripetizioni del ciclo (*numero*)
2. esecuzione del blocco di istruzioni (*sequenza*)
3. decremento del contenuto del registro CX
4. controllo sul contenuto del registro CX:

- a. se CX è diverso da 0 salta al passo 2
- b. se CX è uguale a 0 esce dal ciclo.

Esercizio: Lettura e visualizzazione di una stringa di caratteri.

Si vuole realizzare un frammento di programma che legga da tastiera 10 caratteri e li visualizzi su video. Una possibile implementazione in linguaggio C è la seguente:

```
main()
{ char vett[11];
  int i,
  ...
  for (i=0 ; i<10 ; i++)
      scanf("%c",&vett[i]);
  vett[10] = '\0';
  printf("%s\n",vett);
  ...
}
```

La soluzione proposta in linguaggio Assembler è la seguente:

```
LUNG      EQU      10
           .MODEL   small
           .DATA
VETT       DB      LUNG DUP (?),0Dh, 0Ah,"$"
           .CODE
           ...
           MOV      SI, 0
           MOV      CX, LUNG      ; copia in CX del numero di iterazioni
ciclo:     MOV      AH, 01h        ; lettura di un carattere da tastiera ed
           INT      21h           ; eco del carattere su video
           MOV      VETT[SI], AL  ; copia del carattere letto nel vettore
           INC      SI            ; scansione del vettore
           LOOP     ciclo         ; decrementa CX, CX = 0 ?
                                   ; No: ripeti il ciclo
           LEA      DX, VETT      ; Sì: copia in DX dell'offset di VETT
           MOV      AH, 09h
           INT      21h           ; visualizzazione su video
           ...
```

Variante al ciclo FOR

È possibile realizzare costrutti di tipo *FOR* più complicati aventi una condizione di uscita costituita da un AND di due condizioni distinte. L'equivalente costrutto in linguaggio C è il seguente:



```
int i;
...
for (i=0 ; (i < numero) && (condizione) ; i++)
    sequenza;
```

Il blocco di istruzioni *sequenza* è eseguito finché il contatore *i* è minore di *numero* e l'espressione in *condizione* è verificata.

In linguaggio Assembler le istruzioni di controllo del flusso `LOOPE` e `LOOPNE` permettono di implementare in maniera semplice tale costrutto. Esse variano unicamente per la diversa condizione di terminazione:



```

MOV          CX, numero
ciclo: sequenza
      CMP          cond          ; cond è vera ?
      LOOPE        ciclo         ; decremento di CX
                                   ; (CX ≠ 0 AND cond vera) ?
                                   ; Si: ripete il ciclo
      ...          ; No: esce dal ciclo

```



```

MOV          CX, numero
ciclo: sequenza
      CMP          cond          ; cond è vera ?
      LOOPNE       ciclo         ; decremento di CX
                                   ; (CX ≠ 0 AND cond falsa) ?
                                   ; Si: ripete il ciclo
      ...          ; No: esce dal ciclo

```

Esercizio: *Lettura e visualizzazione di una stringa di caratteri terminata da un carattere CR.*

Si veda un esempio di applicazione del costrutto `FOR` con doppia condizione di terminazione. Si vuole realizzare un programma che legga da tastiera 10 caratteri che devono essere memorizzati in un vettore e stampati su video. La lettura deve terminare una volta letto il carattere `CR` (*carriage return*).

Una possibile implementazione in linguaggio C è la seguente:

```

#define LUNG 10
main()
{ char vett[LUNG+1];
  char c = '\0';      /* inizializzazione della variabile c */
  int i;
  ...
  for (i=0 ; (i<LUNG) && (c != '\n') ; i++)
  {
    scanf("%c",&c);
    vett[i] = c;
  }
  vett[i] = '\0';
  printf("%s\n",vett);
  ...
}

```

Una soluzione proposta in linguaggio Assembler è la seguente:

```

LUNG      EQU      10
CR         EQU      13
EOS        EQU      "$"
          .MODEL    small
          .DATA
VETT       DB      (LUNG+1) DUP (?)
          .CODE
...
          MOV       SI, 0
          MOV       CX, LUNG          ; copia in CX del numero di iterazioni
ciclo:     MOV       AH, 01h          ; lettura di un carattere da tastiera
          INT       21H              ; eco del carattere su video
          MOV       VETT[SI], AL      ; copia del carattere nel vettore
          INC       SI               ; scansione del vettore
          CMP       AL, CR            ; il carattere letto è un carattere CR ?
          LOOPNE    ciclo             ; decrementa CX, CX = 0 ?
          ; CX ≠ 0 AND AL ≠ 0DH: ripete il ciclo
          MOV       VETT[SI], EOS     ; copia in VETT del carattere "$"
          LEA       DX, VETT          ; copia in DX dell'offset di VETT
          MOV       AH, 09H           ; visualizzazione su video
          INT       21H
          ...

```

9.4.6. Costrutto *WHILE*

Il costrutto *WHILE* permette di eseguire un ciclo di istruzioni fintanto che una condizione rimane vera. In linguaggio C il formato del costrutto *WHILE* è il seguente:



```

while (condizione)
    sequenza;

```

La sequenza di operazioni eseguite è la seguente:

1. calcolo della *condizione*:
 - a. se la condizione è vera: viene eseguita la *sequenza* di istruzioni e si ritorna al passo 1
 - b. altrimenti: si esce dal ciclo.

La differenza rispetto al costrutto *REPEAT-UNTIL* sta nel fatto che la valutazione della condizione è qui effettuata *prima* di eseguire la sequenza di istruzioni.

In linguaggio Assembler una possibile implementazione del ciclo *WHILE* è la seguente:



```

ciclo:     CMP       cond              ; cond è vera ?
          JNcond     cont              ; No: esce dal ciclo
          sequenza                      ; Sì: ciclo
          JMP       ciclo              ; ritorna a inizio ciclo

cont:     ...

```

Esercizio: Somma gli elementi di un vettore di numeri interi.

Si vuole realizzare un programma che sommi gli elementi di un vettore di numeri interi fino a quando il valore della somma diventa un numero positivo. Una possibile realizzazione in linguaggio C è la seguente:

```
#define LUNG 10
main()
{ int vett[10], i, somma;
  ...
  i = 1;
  somma = vett[0];
  while ( (somma <=0) && (i<LUNG))
    somma += vett[i++];
  ...
}
```

La soluzione proposta in linguaggio Assembler è la seguente:

```
LUNG      EQU      10
           .MODEL   small
           .DATA
VETT       DW       LUNG DUP (?)
SOMMA      DW       ?
           .CODE
           ...
           MOV      AX, VETT      ; copia in AX primo elemento del vettore
           MOV      SI, 2         ; scansione del vettore
           MOV      CX, LUNG      ; copia in CX il numero di elementi
ciclo:     CMP      AX, 0         ; AX > 0 ?
           JNLE     esci         ; Sì: esce dal ciclo
           ADD      AX, VETT[SI]  ; No: somma del contenuto del vettore
           ADD      SI, 2         ; scansione del vettore
           LOOP     ciclo         ; ritorno ad inizio ciclo
esci:      MOV      SOMMA, AX     ; copia in SOMMA del contenuto di AX
           ...
```

10. Le istruzioni aritmetiche

In questo capitolo verranno descritte le istruzioni aritmetiche che l'architettura x86 mette a disposizione del programmatore.

10.1. Le istruzioni ADD e SUB

Le istruzioni **ADD** (*ADDition*) e **SUB** (*SUBtract*) eseguono rispettivamente l'addizione e la sottrazione tra numeri binari interi; il loro formato è il seguente:



<i>ADD</i>	<i>destinazione, sorgente</i>
<i>SUB</i>	<i>destinazione, sorgente</i>

L'istruzione **ADD** esegue un'addizione tra l'operando *destinazione* e l'operando *sorgente* e scrive il risultato nell'operando *destinazione*; l'operando *sorgente* rimane immutato.

L'istruzione **SUB** esegue una sottrazione tra l'operando *destinazione* (*minuendo*) e l'operando *sorgente* (*sottraendo*): il risultato è memorizzato nell'operando *destinazione*, mentre l'operando *sorgente* rimane immutato. Le istruzioni **ADD** e **SUB** aggiornano il valore di tutti i flag di stato.

Le regole sintattiche per le istruzioni di **ADD** e **SUB** impongono che:

- gli operandi devono essere dello stesso tipo (o entrambi byte o entrambi word);
- l'operando *destinazione* può essere un registro oppure un dato contenuto in una locazione di memoria;
- l'operando *sorgente* può essere un registro, un dato contenuto in una locazione di memoria, oppure un valore immediato;
- entrambi gli operandi non possono essere locazioni di memoria.

Esempi

Le seguenti operazioni di somma e sottrazione sono lecite:



```

ADD    AL, VAL    ; AL = AL + VAL
SUB    BX, SI     ; BX = BX - SI
ADD    CH, 5      ; CH = CH + 5
SUB    WVAL, AX   ; WVAL = WVAL - AX
ADD    WVAL, 5    ; WVAL = WVAL + 5

```

Le seguenti operazioni di somma e sottrazione non sono lecite:



```

ADD    AX, VAL    ; VAL: variabile di tipo byte
SUB    BX, AH     ; BX e AH sono di tipo diverso
ADD    5, CH      ; valore immediato come primo operando
SUB    WVAL, AH   ; WVAL: variabile di tipo word
ADD    VAL1, VAL2 ; somma tra due locazioni di memoria

```

Per poter eseguire la somma tra due locazioni di memoria è necessario utilizzare un registro *general purpose*.



```

MOV    AH, VAL2   ; copia il contenuto di VAL1 in AH
ADD    VAL1, AH   ; VAL1 = VAL1 + AH

```

10.2. L'istruzione CBW

L'istruzione **CBW** (*Convert Byte to Word*) permette di convertire un byte nella word equivalente. Questa istruzione non ha operandi; il suo formato è il seguente:



CBW

L'istruzione CBW esegue l'estensione del segno del contenuto di AL nel registro AH:

- se il registro AL contiene un numero positivo in AH è caricato il valore 00H;
- se il registro AL contiene un numero negativo in AH è caricato il valore FFH.

L'istruzione CBW non modifica lo stato dei flag.

Essa risulta spesso utile quando si devono aggiungere (o sottrarre) un numero di tipo byte ed uno di tipo word. Le operazioni da eseguire sono in tal caso le seguenti:

1. caricare il dato di tipo byte in AL;
2. eseguire l'estensione del segno con l'istruzione CBW;
3. eseguire l'operazione aritmetica desiderata tra AX ed il dato di tipo word.

Esempio

Il seguente frammento di codice esegue la somma tra la variabile di tipo byte di nome VALORE ed il contenuto del registro SI; il risultato è memorizzato in SI.

```

MOV    AL, VALORE ; copia della variabile VALORE in AL
CBW    ; conversione da byte a word
ADD    SI, AX     ; SI = SI + AX

```

10.3. L'istruzione ADC

L'istruzione **ADC** (*ADD with Carry*) risulta particolarmente utile nell'addizione tra numeri interi rappresentati su 32 bit o più. Il suo formato è il seguente:



ADC destinazione, sorgente

L'istruzione **ADC** somma al contenuto dell'operando *destinazione* il contenuto dell'operando *sorgente* ed il valore del flag **CF**: il risultato complessivo viene memorizzato nell'operando *destinazione*, lasciando invariato l'operando *sorgente*. L'istruzione **ADC** aggiorna il valore di tutti i flag di stato.

Se il flag **CF** vale 0 l'istruzione **ADC** si comporta come una istruzione **ADD**; altrimenti aggiunge 1 alla somma dei due operandi prima di memorizzare il risultato nell'operando *destinazione*.

L'istruzione **ADC** si rivela utile nel caso di addizione tra numeri interi memorizzati su doubleword (32 bit) o quadword (64 bit).

Nel caso di somma tra due doubleword occorre:

1. sommare le due word meno significative utilizzando l'istruzione **ADD**
2. sommare le due word più significative utilizzando l'istruzione **ADC**.

La prima somma può generare un riporto (*carry*), che deve essere sommato al risultato della seconda addizione utilizzando l'istruzione **ADC**.

Per l'esecuzione della somma tra numeri su 32 bit e su 64 bit si vedano gli esercizi seguenti.

Esercizio: *Somma tra due numeri rappresentati su 32 bit.*

Si scriva un frammento di codice Assembler in grado di eseguire la somma di due numeri rappresentati su 32 bit contenuti nelle variabili **NUMA** e **NUMB**, ponendo il risultato in **NUMC**.

```

.MODEL    small
.DATA
NUMA      DD      ?
NUMB      DD      ?
NUMC      DD      ?
.CODE
...
MOV       AX, WORD PTR NUMA
ADD       AX, WORD PTR NUMB      ; somma tra loro le 2 word meno
                                ; significative
MOV       WORD PTR NUMC, AX
MOV       AX, WORD PTR NUMA+2
ADC       AX, WORD PTR NUMB+2    ; somma tra loro le 2 word più
                                ; significative + l'eventuale
                                ; carry della somma precedente
MOV       WORD PTR NUMC+2, AX
...

```

Esercizio: *Somma tra due numeri rappresentati su 64 bit.*

Si scriva un frammento di codice Assembler in grado di eseguire la somma tra due numeri interi rappresentati su 64 bit contenuti nelle variabili **NUMA** e **NUMB**, ponendo il risultato in **NUMC**.

```

.MODEL small
.DATA
NUMA DQ ?
NUMB DQ ?
NUMC DQ ?
.CODE
...
MOV AX, WORD PTR NUMA
ADD AX, WORD PTR NUMB ; somma tra loro le 2 prime word
MOV WORD PTR NUMC, AX
MOV AX, WORD PTR NUMA+2
ADC AX, WORD PTR NUMB+2 ; somma tra loro le 2 seconde word
; + CF
MOV WORD PTR NUMC+2, AX
MOV AX, WORD PTR NUMA+4
ADC AX, WORD PTR NUMB+4 ; somma tra loro le 2 terze word
; + CF
MOV WORD PTR NUMC+4, AX
MOV AX, WORD PTR NUMA+6
ADC AX, WORD PTR NUMB+6 ; somma tra loro le 2 quarte word
; + CF
MOV WORD PTR NUMC+6, AX
...

```

Una seconda soluzione che utilizza un ciclo di istruzioni è la seguente:

```

.MODEL small
.DATA
NUMA DQ ?
NUMB DQ ?
NUMC DQ ?
.CODE
...
CLC ; azzeramento del flag CF
LEA SI, WORD PTR NUMA ; copia l'offset di NUMA in SI
LEA DI, WORD PTR NUMB ; copia l'offset di NUMB in DI
LEA BX, WORD PTR NUMC ; copia l'offset di NUMC in BX
MOV CX, 4 ; 4 iterazioni
ciclo: MOV AX, [SI]
ADC AX, [DI] ; somma tra 2 word + CF
PUSHF
MOV [BX], AX
ADD SI, 2
ADD DI, 2
ADD BX, 2
POPF
LOOP ciclo
...

```

È stato necessario salvare nello stack i flag (attraverso l'istruzione `PUSHF`) dal momento che le successive istruzioni di `ADD` modificano il valore dei flag. L'istruzione `POPF` permette di ripristinare il valore dei flag.

10.4. L'istruzione **SBB**

L'istruzione **SBB** (*SuBtract with Borrow*) risulta particolarmente utile nella sottrazione tra numeri interi su 32 bit e su 64 bit. Il suo formato è il seguente:



SBB destinazione, sorgente

L'istruzione SBB esegue la sottrazione tra l'operando *destinazione* e l'operando *sorgente*: il valore del flag CF viene sottratto al risultato ed il valore ottenuto viene copiato nell'operando *destinazione*, mentre l'operando *sorgente* rimane immutato.

Se il flag CF vale 0 l'istruzione SBB si comporta come l'istruzione SUB; altrimenti sottrae 1 alla differenza tra l'operando *destinazione* e l'operando *sorgente* prima di memorizzare il risultato nell'operando *destinazione*.

L'istruzione SBB è utilizzata nel caso di sottrazione tra numeri interi memorizzati su doubleword (32 bit) o quadword (64 bit), dove occorre sottrarre una word alla volta, cominciando da quella meno significativa.

Nel caso di sottrazione tra due doubleword occorre:

1. sottrarre le due word meno significative utilizzando l'istruzione SUB
2. sottrarre le due word più significative utilizzando l'istruzione SBB.

La prima sottrazione può richiedere un prestito (*borrow*), che deve essere sottratto al risultato della seconda sottrazione utilizzando l'istruzione SBB.

Per l'esecuzione della sottrazione tra numeri rappresentati su 32 bit e su 64 bit si vedano gli esercizi seguenti.

Esercizio: Sottrazione tra due numeri rappresentati su 32 bit.

Si scriva un frammento di codice Assembler in grado di eseguire la sottrazione tra numeri rappresentati su 32 bit contenuti nelle variabili NUMA e NUMB. La soluzione proposta in linguaggio Assembler è la seguente:

```

        .MODEL    small
        .DATA
NUMA    DD        ?
NUMB    DD        ?
NUMC    DD        ?
        .CODE
        ...
        MOV      AX, WORD PTR NUMA
        SUB      AX, WORD PTR NUMB      ; sottrazione tra le 2 word meno
                                         ; significative
        MOV      WORD PTR NUMC, AX
        MOV      AX, WORD PTR NUMA+2
        SBB      AX, WORD PTR NUMB+2    ; sottrazione tra le 2 word più
                                         ; significative meno il borrow
                                         ; della sottrazione precedente
        MOV      WORD PTR NUMC+2, AX
        ...

```

Esercizio: *Sottrazione tra due numeri rappresentati su 64 bit.*

Si scriva un frammento di codice Assembler in grado di eseguire la sottrazione tra numeri su 64 bit contenuti nelle variabili NUMA e NUMB. La soluzione proposta in linguaggio Assembler è la seguente:

```

                .MODEL small
                .DATA
NUMA            DQ      ?
NUMB            DQ      ?
NUMC            DQ      ?
                .CODE
                ...
                CLC                      ; azzeramento del flag CF
                LEA     SI, WORD PTR NUMA ; copia l'offset di NUMA in SI
                LEA     DI, WORD PTR NUMB ; copia l'offset di NUMB in DI
                LEA     BX, WORD PTR NUMC ; copia l'offset di NUMC in BX
                MOV     CX, 4              ; 4 iterazioni
ciclo:          MOV     AX, [SI]
                SBB     AX, [DI]          ; differenza tra due word - borrow
                PUSHF
                MOV     [BX], AX
                ADD     SI, 2
                ADD     DI, 2
                ADD     BX, 2
                POPF
                LOOP    ciclo
                ...

```

10.5. Le istruzioni INC e DEC

Le istruzioni **INC** (*INC*rement *destination by one*) e **DEC** (*DEC*rement *destination by one*) permettono rispettivamente di incrementare e decrementare di un'unità il contenuto dell'operando. Il loro formato è il seguente:



<i>INC</i>	<i>operando</i>
<i>DEC</i>	<i>operando</i>

L'istruzione **INC** incrementa l'*operando* di un'unità, mentre l'istruzione **DEC** decrementa l'*operando* di un'unità. Per entrambe le istruzioni l'*operando* può essere un registro oppure una locazione di memoria. Le due istruzioni aggiornano tutti i flag di stato tranne il flag CF.

Esercizio: Calcolo della radice quadrata.

Un modo per calcolare la radice quadrata approssimata di un numero intero consiste nel contare la quantità di numeri dispari che possono essere sottratti dal numero di partenza. La soluzione proposta in linguaggio C è la seguente:

```
main()
{
    int num, sqr=0, disp=1;
    ...
    num--;
    while (num >= 0)
    {
        sqr++;
        disp += 2;
        num -= disp;
    }
    ...
}
```

La soluzione proposta in linguaggio Assembler è la seguente:

```
.MODEL    small
.DATA
NUM       DW      ?
VAR       DW      ?
SQR       DW      ?
.CODE
...
MOV       AX, NUM
MOV       VAR, AX      ; VAR = NUM
DEC       VAR          ; VAR = VAR - 1
MOV       BX, 0        ; BX conta i numeri dispari sottratti
MOV       CX, 1        ; in CX vi sono i numeri dispari
ciclo:    CMP       VAR, 0      ; VAR >= 0 ?
JNGE      continua     ; No: va a continua
INC       BX           ; Sì: incrementa il contatore
ADD       CX, 2        ; prossimo numero dispari
SUB       VAR, CX       ; sottrae il numero dispari
JMP       ciclo
continua: MOV       SQR, BX    ; copia il risultato in SQR
...
```

10.6. L'istruzione NEG

L'istruzione **NEG** (*NEG*ate) permette di invertire il segno di un numero intero. Il suo formato è il seguente:



NEG *operando*

L'istruzione NEG cambia il segno dell'*operando*, che si assume rappresentato in complemento a 2 e può essere un registro oppure una locazione di memoria di tipo byte oppure word. L'istruzione NEG aggiorna il valore di tutti i flag di stato.

Esercizio: Calcolo del modulo di un vettore di interi.

Si realizzi un programma che calcoli il modulo del contenuto di tutte le celle di un vettore di numeri interi. La soluzione proposta in linguaggio C è la seguente:

```
main()
{
    int i, vett[10];
    ...
    for (i=0 ; i < 10 ; i++)
        if (vett[i] < 0)
            vett[i] *= -1;
    ...
}
```

La soluzione proposta in linguaggio Assembler è la seguente:

```
LUNG      EQU      10
           .MODEL   small
           .DATA
VETT      DW      LUNG DUP (?)
           .CODE
           ...
           MOV      SI, 0
           MOV      CX, LUNG
ciclo:    CMP      VETT[SI], 0      ; elemento < 0 ?
           JNL      continua      ; No: va a continua
           NEG      VETT[SI]      ; Sì: calcola il modulo
continua:  ADD      SI, 2          ; scansione del vettore
           LOOP     ciclo
           ...
```

10.7. Le istruzioni MUL e IMUL

Le istruzioni **MUL** (*MULTiply, unsigned*) e **IMUL** (*Integer MULTiply*) permettono di eseguire il prodotto tra numeri interi, rispettivamente senza segno e con segno. Il loro formato è il seguente:



<i>MUL</i>	<i>operando</i>
<i>IMUL</i>	<i>operando</i>

L'unica differenza tra le due istruzioni è il tipo di dato su cui esse lavorano; l'istruzione **MUL** opera su numeri interi senza segno, l'istruzione **IMUL** su numeri interi con segno.

L'*operando* può essere un registro oppure una locazione di memoria; il suo tipo può essere **BYTE** oppure **WORD**. Non è ammessa la moltiplicazione per un valore immediato.

Entrambe le istruzioni aggiornano il valore dei flag **CF** ed **OF**, mentre il valore degli altri flag di stato è indefinito.

I due operandi devono avere la stessa lunghezza ed il risultato viene memorizzato in un operando avente lunghezza doppia rispetto ad essi. I due casi possibili sono:

- se si specifica un operando di tipo **BYTE**, il processore esegue la moltiplicazione tra l'operando ed il contenuto del registro **AL** ed il risultato è copiato nel registro **AX**;
- se si specifica un operando di tipo **WORD**, il processore esegue la moltiplicazione tra l'operando ed il contenuto del registro **AX** ed il risultato è copiato nei registri **DX** (word più significativa) ed **AX** (word meno significativa).

L'istruzione **MUL** aggiorna i flag **CF** ed **OF** in modo da segnalare se la metà meno significativa

dei registri è sufficiente a contenere il risultato:

- in una moltiplicazione tra byte i flag CF ed OF valgono 0 se il registro AH è nullo;
- in una moltiplicazione tra word i flag CF ed OF valgono 0 se il registro DX è nullo.

L'istruzione `IMUL` aggiorna i flag CF ed OF in modo da segnalare se la metà meno significativa dei registri è sufficiente a contenere il risultato:

- in una moltiplicazione tra byte i flag CF ed OF valgono 0 se il registro AH vale 0 o FFH a seconda del segno;
- in una moltiplicazione tra word i flag CF ed OF valgono 0 se il registro DX vale 0 o FFFFH a seconda del segno.

Esercizio: *Calcolo del quadrato di un numero intero.*

Si vuole realizzare un frammento di programma che esegua il calcolo del quadrato di un numero intero senza segno. La soluzione proposta in linguaggio Assembler è la seguente:

```

.MODEL    small
.DATA
NUM       DW      ?
RES       DD      0
.CODE
...
MOV       WORD PTR RES+2, 0
MOV       AX, NUM           ; AX = NUM
MUL       AX                ; DX,AX = NUM * NUM
MOV       WORD PTR RES, AX  ; copia la word bassa in RES
JNC       esce              ; word alta = 0 ?
MOV       WORD PTR RES+2, DX ; No: copia la word alta in RES+2
esce:     ...

```

10.7.1. Moltiplicazione per una costante

A partire dal processore 80186, l'istruzione `IMUL` permette la moltiplicazione tra un numero intero ed un valore immediato, secondo uno dei seguenti formati:



```

IMUL    operando, costante
IMUL    destinazione, operando, costante

```

L'*operando* e la *destinazione* possono essere uno dei registri *general purpose* AX, BX, CX, DX, SP, BP, SI e DI. Il processore moltiplica l'*operando* per la *costante*; se è specificato un campo *destinazione*, l'istruzione copia il risultato nel registro *destinazione*, altrimenti lo copia nel registro *operando*.

Il risultato del prodotto è memorizzato comunque su 16 bit. Occorre fare dunque attenzione alla correttezza del risultato: se il risultato della operazione è rappresentabile tramite una word, l'istruzione azzerà i flag CF ed OF; in caso contrario, l'istruzione pone ad 1 i flag CF ed OF (in questo caso vi è un errore di *overflow* nella moltiplicazione).

Esercizio: *Conversione di valuta.*

Si realizzi un frammento di programma che converta il costo di un prodotto da franchi francesi in lire italiane. La soluzione proposta in linguaggio Assembler è illustrata nel seguito; si noti che è stata utilizzata la direttiva `.386` per abilitare l'assemblatore ad utilizzare le istruzioni disponibili sul processore 80386.


```

FFRANCO    EQU    297
            .386                      ; direttiva per l'assemblatore
            .MODEL small
            .DATA
F_COST     DW      ?
IT_COST    DW      ?
ERR_MSG    DB      "Overflow nella moltiplicazione", 0DH, 0AH, "$"
            .CODE
            ...
            MOV     AX, F_COST
            IMUL    AX, FFRANCO        ; AX = AX * 297
            JNC     ok                 ; CF = 1 ?
            LEA     DX, ERR_MSG        ; Sì: visualizzazione messaggio errore
            MOV     AH, 09H
            INT     21H
            JMP     esci
ok:         MOV     IT_COST, AX        ; No: copia del risultato in IT_COST
esci:       ...

```

10.7.2. Moltiplicazione tra dati di tipo diverso

Le istruzioni **MUL** ed **IMUL** permettono di eseguire la moltiplicazione solo tra dati dello stesso tipo (o entrambi byte o entrambi word). È possibile moltiplicare un byte per una word utilizzando opportunamente l'istruzione **CBW**.

Esempio

Il frammento di codice seguente esegue la moltiplicazione tra la variabile di tipo byte di nome **BVAL** e la variabile di tipo word di nome **WVAL**:

```

            .DATA
BVAL        DB      ?
WVAL        DW      ?
            .CODE
            ...
            MOV     AL, BVAL           ; copia in AL
            CBW     ; conversione da AL ad AX
            IMUL    WVAL               ; DX,AX = AX * WVAL
            ...

```

10.8. Le istruzioni **DIV** e **IDIV**

Le istruzioni **DIV** (*DIVision, unsigned*) e **IDIV** (*Integer DIVision, signed*) permettono di eseguire l'operazione di divisione tra numeri interi rispettivamente senza segno o con segno. Il loro formato è il seguente:



```

DIV    operando
IDIV   operando

```

L'*operando* può essere il contenuto di un registro oppure il contenuto di una locazione di memoria. Le istruzioni **DIV** e **IDIV** non aggiornano i flag: il valore dei flag di stato dopo un'istruzione di divisione è indefinito.

L'unica differenza tra le due istruzioni è il tipo di dato su cui esse lavorano; l'istruzione **DIV** opera su numeri interi senza segno e l'istruzione **IDIV** su numeri interi con segno.

Entrambe le istruzioni possono eseguire due tipi di operazioni:

- divisione tra un operando di tipo word ed un operando di tipo byte
 - divisione tra un operando di tipo doubleword ed un operando di tipo word,
- restituendo comunque due risultati: il *quoziente* ed il *resto*.

Il comportamento dell'istruzione è diverso a seconda del tipo di operazione:

- se l'operando è di tipo BYTE, il processore esegue la divisione tra il contenuto del registro AX (*dividendo*) ed il contenuto dell'*operando* (*divisore*); come risultato della divisione, l'istruzione restituisce il *quoziente* nel registro AL ed il *resto* nel registro AH.
- se l'operando è di tipo WORD, il processore esegue la divisione tra il contenuto della coppia di registri DX,AX (*dividendo*) ed il contenuto dell'*operando* (*divisore*); come risultato della divisione, l'istruzione restituisce il *quoziente* nel registro AX ed il *resto* nel registro DX.

Una condizione di *overflow* si può verificare nel caso in cui il divisore sia troppo piccolo. In tal caso il processore rileva l'errore e salta alla *procedura di interruzione per la gestione della divisione per zero* (interrupt di tipo 0).

Esempio

La sequenza di istruzioni seguenti causa un *overflow di divisione*:

```
MOV    AX, 1024
MOV    BL, 2
DIV    BL           ; il quoziente (512) non può essere
                   ; memorizzato nel registro AL
```

L'istruzione IDIV è stata realizzata in modo che il resto ed il quoziente abbiano lo stesso segno.

Esempio

Se si divide il numero -53 per 7, vi sono due possibili soluzioni:

- quoziente pari a -8, resto pari a +3
- quoziente pari a -7, resto pari a -4.

L'istruzione IDIV restituisce il secondo risultato.

```
      .DATA
NUMA  DW      -53
NUMB  DB       7
      .CODE
...
MOV    AX, NUMA
IDIV   NUMB       ; NUMA / NUMB: AL = -7, AH = -4
```

Esercizio: Scomposizione in fattori primi di un numero intero.

Si realizzi un frammento di programma che effettui la scomposizione in fattori primi di un numero intero.

Si utilizza il seguente algoritmo:

1. il dividendo è inizializzato al numero di partenza ed il divisore è inizializzato a 2;
2. si esegue la divisione tra dividendo e divisore;
3. se il resto della divisione è 0, il divisore è un fattore primo ed il quoziente viene copiato nel dividendo;
4. altrimenti, il divisore è incrementato di una unità;
5. se il resto è maggiore di 1 si ritorna al passo 2.

Una possibile implementazione in linguaggio C è la seguente:

```
main()
{
  int dvs=1, i=0, num, quo, factor[100];
  ...
  factor[i++] = dvs++;
  do
  { quo = num / dvs;
    if ( (num%dvs) == 0)
    { factor[i++] = dvs;
      num = quo;
    }
    else dvs++;
  }
  while (num > 1);
  ...
}
```

La soluzione proposta in linguaggio Assembler è la seguente:

```
LUNG      EQU      1000
           .MODEL   small
           .DATA
NUM        DW       ?
FACTOR     DB       LUNG DUP(?)      ; vettore sufficientemente grande !
           .CODE
           ...
           MOV      SI, 0
           MOV      FACTOR[SI], 1    ; 1 è un fattore primo
           INC      SI                ; aggiorna indice
           MOV      AX, NUM           ; in AX: dividendo = NUM
           MOV      CL, 2             ; in CL: divisore = 2
ciclo:     MOV      BX, AX            ; salvataggio del dividendo
           DIV      CL                ; AX / CL
           CMP      AH, 0             ; resto = 0 ?
           JNZ      else              ; No: va a else
then:      MOV      FACTOR[SI], CL    ; Sì: CL è un fattore primo
           INC      SI                ; aggiorna indice
           JMP      continua
else:      MOV      AX, BX            ; ripristino del dividendo da BX
           INC      CL                ; incremento del divisore
continua:  CMP      AL, 1             ; quoziente > 1 ?
           JG       ciclo             ; Sì: ritorna a ciclo
           ...                       ; No: esce
```

Esercizio: *Calcolo della media dei numeri positivi presenti in un vettore.*

Si realizzi un frammento di programma che calcoli il valore medio dei numeri positivi compresi in un vettore di numeri interi con segno. La soluzione proposta in linguaggio C è la seguente:

```
main()
{
    int i, count=0, somma=0, avg, vett[10];
    ...
    for (i=0 ; i<10 ; i++)
        if (vett[i] > 0)
            { count++;
              somma += vett[i]; }
    avg = somma/count;
    ...
}
```

La soluzione proposta in linguaggio Assembler è la seguente:

```
LUNG      EQU      10
           .MODEL   small
           .DATA
VETT      DW      LUNG DUP (?)
COUNT    DB      ?
AVG        DB      ?
           .CODE
           ...
           MOV      CX, LUNG
           MOV      SI, 0
           MOV      BX, 0          ; somma totale
           MOV      COUNT, 0      ; in COUNT: numero di positivi
ciclo:     CMP      VETT[SI], 0    ; VETT[] > 0 ?
           JNG      continua      ; No: va a continua
           INC      COUNT          ; Sì: incrementa il contatore
           ADD      BX, VETT[SI]   ; BX = BX + VETT[SI]
continua:  ADD      SI, 2          ; scansione del vettore
           LOOP     ciclo
           MOV      AX, BX        ; copia in AX del dividendo
           IDIV     COUNT         ; BX / COUNT
           MOV      AVG, AL       ; copia in AVG del quoziente
           ...
```

Esercizio: *Il gioco della morra.*

Si vuole realizzare un programma che simuli il gioco detto *della morra*. I giocatori devono contemporaneamente *tirare* un numero compreso tra 0 e 5 e *gridare* un secondo numero compreso tra 0 e 10. Vince il giocatore che indovina la somma dei numeri *tirati*. L'utente gioca contro il computer. Il computer sceglie due numeri casuali tra 0 e 5: il primo è il numero *tirato*, la somma dei due numeri è il numero *gridato*; l'utente fornisce da tastiera il numero da lui *tirato* ed il numero che lui suppone che il computer abbia *tirato*.

La soluzione proposta in linguaggio C è la seguente:

```
#include <stdio.h>
#include <time.h>
main()
{
    time_t t;
    int comp1, comp2, user1, user2;
    int somma_c, somma_g, somma;
    srand((unsigned) time(&t));
    comp1 = rand()%6;
    comp2 = rand()%6;
    somma_c = comp1 + comp2;
    printf("Numero tirato\n");
    scanf("%d",&user1);
    printf("Numero tirato dal computer\n");
    scanf("%d",&user2);
    somma_g = user1 + user2;
    somma = comp1 + user1;
    if (somma == somma_g)
        printf("Ha vinto il computer\n");
    if (somma == somma_c)
        printf("Ha vinto il giocatore\n");
}
```

La soluzione proposta in linguaggio Assembler utilizza le *function call* MS-DOS: il problema di generare un numero pseudo-casuale tra 0 e 5 è stato risolto leggendo il *timer* di sistema attraverso la *function call* 2CH. Essa restituisce il valore corrente del *timer* di sistema nel registro DX; il contenuto del registro DL viene diviso per 6: si assume come numero casuale il resto, necessariamente compreso tra 0 e 5. La soluzione proposta in linguaggio Assembler è la seguente:

```
.MODEL    small
.DATA
SOMMA    DB    ?
MSG1     DB    "Numero tirato",0DH,0AH,"$"
MSG2     DB    "Numero tirato dal computer",0DH,0AH,"$"
MSG3     DB    0DH,0AH,"Ha vinto il computer$"
MSG4     DB    0DH,0AH,"Ha vinto il giocatore$"
.CODE
...
MOV      AH, 2CH      ; lettura del timer di sistema
INT      21H
MOV      AL, DL       ; copia del timer in AL
MOV      AH, 0        ; azzeramento del registro AH
MOV      CL, 06H      ; copia in CL del divisore: 6
DIV      CL           ; timer / 6
MOV      BH, AH       ; in BH: valore giocato dal computer
MOV      BL, BH       ; copia in BL
MOV      AL, DH       ; copia del timer in AL
MOV      AH, 0        ; azzeramento del registro AH
MOV      CL, 06H      ; copia in CL del divisore: 6
DIV      CL           ; timer / 6
ADD      BL, AH       ; in BL: somma ipotizzata dal computer
LEA      DX, MSG1     ; messaggio di richiesta di dato: giocatore
MOV      AH, 09H      ; visualizzazione su video
INT      21H
MOV      AH, 01H      ; lettura di un carattere da tastiera
INT      21H
SUB      AL, 30H      ; conversione da ASCII a cifra binaria
MOV      CH, AL       ; in CH: valore giocato dal giocatore
LEA      DX, MSG2     ; messaggio di richiesta di dato: computer
MOV      AH, 09H      ; visualizzazione su video
INT      21H
```

```

MOV     AH, 01H      ; lettura di un carattere da tastiera
INT     21H
SUB     AL, 30H      ; conversione da ASCII a cifra binaria
MOV     CL, AL
ADD     CL, CH        ; in CL: somma ipotizzata dal giocatore
MOV     SOMMA, BH    ; copia del valore giocato dal computer
ADD     SOMMA, CH     ; somma dei valori giocati
CMP     BL, SOMMA    ; BL = SOMMA ?
JNZ     no_comp      ; No: va a no_comp
LEA     DX, MSG3     ; Sì: vittoria del computer
MOV     AH, 09H
INT     21H          ; visualizzazione del messaggio su video
no_comp: CMP     CL, SOMMA ; CL = SOMMA ?
JNZ     esci         ; No: va a esci
LEA     DX, MSG4     ; Sì: vittoria del giocatore
INT     21H
esci:    ...

```

10.8.1. L'istruzione CWD

L'istruzione **CWD** (*Convert Word to Doubleword*) permette di convertire una word in una doubleword, secondo il seguente formato:



CWD

L'istruzione CWD esegue l'estensione del segno del registro AX verso il registro DX, che viene riempito di bit a 1 se il valore in AX è negativo, di bit a 0 altrimenti. L'istruzione CWD non modifica il valore di alcun flag.

Le istruzioni di divisione obbligano a dividere una word per un byte oppure una doubleword per una word. Per dividere dati dello stesso tipo occorre convertire preventivamente il tipo del dividendo prima di eseguire l'operazione di divisione. Per convertire un byte in una word si usa l'istruzione CBW, mentre per convertire una word in una doubleword si utilizza l'istruzione CWD. Uno degli usi più frequenti dell'istruzione CWD è dunque quello di predisporre i dati per la divisione tra due word.

Esempio

Il frammento di codice seguente esegue la divisione tra il contenuto del registro CX ed il contenuto del registro BX.

```

MOV     AX, CX      ; copia di CX in AX
CWD     ; conversione da word a doubleword
IDIV    BX          ; divisione tra CX e BX

```

10.9. I numeri BCD

La codifica BCD (*Binary Coded Decimal*) costituisce un diverso modo di rappresentare i numeri: ciascuna cifra del numero decimale viene codificata in binario puro su un numero prefissato di bit.

Esistono due codifiche BCD:

1. una cifra decimale per byte (*numeri decimali non impaccati*);
2. due cifre decimali per byte (*numeri decimali impaccati*).

Rispetto alla rappresentazione binaria i numeri BCD occupano più spazio in memoria degli equivalenti numeri binari e le operazioni tra numeri BCD sono più difficili da gestire. Ciononostante, i numeri BCD sono spesso utilizzati in quanto, da un lato, permettono di eseguire calcoli su numeri molto grandi senza alcun errore di rappresentazione (per questo motivo sono usati nei calcoli di na-

tura finanziaria) e, dall'altro, risulta immediata la conversione da codifica ASCII a numero decimale e viceversa.

Il processore non fornisce istruzioni che eseguano direttamente le operazioni tra numeri BCD, ma fornisce istruzioni che, in combinazione con quelle per i numeri binari, permettono di ottenere lo stesso risultato.

10.9.1. Le istruzioni AAA e DAA

Le istruzioni **AAA** (*Ascii Adjust for Addition*) e **DAA** (*Decimal Adjust for Addition*) vengono impiegate nelle operazioni di addizione tra numeri BCD. Esse non hanno operandi ed il loro formato è il seguente:



AAA
DAA

Le istruzioni AAA e DAA eseguono la conversione da rappresentazione binaria a BCD del risultato di una istruzione ADD. L'istruzione AAA si usa per convertire un numero binario in numero BCD non impaccato, mentre l'istruzione DAA si usa per convertire un numero binario in numero BCD impaccato. Esse lavorano sul contenuto del registro AL. L'istruzione AAA modifica lo stato dei flag CF e AF, gli altri flag di stato assumono valore indefinito; l'istruzione DAA modifica il valore di tutti i flag di stato, tranne OF che assume valore indefinito.

I numeri BCD devono essere sommati un byte alla volta. La sequenza di operazioni da effettuare è la seguente:

1. si copia il primo addendo in AL;
2. si somma ad AL il secondo addendo usando l'istruzione ADD;
3. si converte il risultato da binario a BCD.

Esempi

Il seguente frammento di codice esegue la somma tra i numeri BCD non impaccati U1 ed U2; il risultato viene scritto nel registro AL.

```

        .DATA
U1      DB      ?
U2      DB      ?
        .CODE
        MOV     AL, U1
        ADD     AL, U2
        AAA

```

Il seguente frammento di codice esegue la somma tra i numeri BCD impaccati P1 e P2; il risultato viene scritto nel registro AL.

```

        .DATA
P1      DB      ?
P2      DB      ?
        .CODE
        MOV     AL, P1
        ADD     AL, P2
        DAA

```

Se l'operazione di somma ha generato un *overflow*, le istruzioni AAA e DAA forzano il flag CF ad 1 ed incrementano di un'unità il contenuto del registro AH. Si ha una condizione di *overflow* quando il risultato è maggiore di 9 per numeri BCD non impaccati, o maggiore di 99 per numeri BCD im-

paccati.

Nel caso in cui il numero BCD sia composto da più byte, occorre eseguire la somma in più passi, sommando un byte alla volta, secondo la seguente procedura:

1. si somma una cifra alla volta a partire dalla cifra meno significativa;
2. si esegue, per ogni cifra, la conversione a numero BCD attraverso l'istruzione AAA.

Quando si sommano byte successivi al primo bisogna tenere conto di eventuali riporti generati dalla somma dei byte precedenti; l'istruzione ADC permette tale operazione.

Per eseguire correttamente la somma tra due numeri BCD su n byte bisogna allocare uno spazio di memoria di $n+1$ byte per memorizzare il risultato, in modo da poter gestire il caso in cui l'ultima operazione di somma generi un riporto.

Esempi

Il seguente frammento di codice esegue la somma tra due numeri (U1 ed U2) di tipo BCD non impaccati di 3 cifre. Il risultato viene scritto nella variabile U3 (memorizzata su 4 byte)

```

      .DATA
U1    DB      3 DUP(?)
U2    DB      3 DUP(?)
U3    DB      4 DUP(?)
      .CODE
      ...
      MOV     AL, U1+2
      ADD     AL, U2+2      ; somma tra i byte meno significativi
      AAA                    ; conversione binario => BCD
      MOV     U3+3, AL      ; copia nel byte meno significativo
      MOV     AL, U1+1
      ADC     AL, U2+1      ; somma tra i 2 secondi byte + CF
      AAA                    ; conversione binario => BCD
      MOV     U3+2, AL      ; copia nel terzo byte
      MOV     AL, U1
      ADC     AL, U2        ; somma tra i 3 primi byte + CF
      MOV     AH, 0         ; azzeramento registro AH
      AAA                    ; conversione binario => BCD
                        ; riporto della somma va in AH
      MOV     U3+1, AL      ; copia nel secondo byte
      MOV     U3, AH        ; copia del riporto nel primo byte
      ...

```


Una soluzione alternativa, che utilizza un ciclo di istruzioni, è la seguente:

```

DIM      EQU      3
          .DATA
NUMA     DB        DIM DUP(?)
NUMB     DB        DIM DUP(?)
NUMC     DB        (DIM+1) DUP(?)
          .CODE
          ...
          CLC                      ; azzeramento del flag CF
          LEA      SI, BYTE PTR NUMA+DIM-1 ; offset di NUMA+2 in SI
          LEA      DI, BYTE PTR NUMB+DIM-1 ; offset di NUMB+2 in DI
          LEA      BX, BYTE PTR NUMC+DIM   ; offset di NUMC+3 in BX
          MOV      CX, DIM                 ; 3 iterazioni
ciclo:    MOV      AL, [SI]
          MOV      AH, 0                   ; azzeramento del registro AH
          ADC      AL, [DI]                 ; somma tra 2 byte + CF
          PUSHF
          AAA
          MOV      [BX], AL
          DEC      SI
          DEC      DI
          DEC      BX
          POPF
          LOOP     ciclo
          MOV      [BX], AH                ; riporto nel primo byte
          ...

```

La procedura di somma tra due numeri BCD impaccati su più byte differisce da quanto detto sinora per due punti:

1. l'istruzione di conversione è DAA;
2. l'istruzione DAA non modifica il contenuto del registro AH in presenza di carry.

Esempi

Il seguente frammento di codice esegue la somma tra due numeri (P1 e P2) di tipo BCD impaccati di 3 cifre. Il risultato viene scritto nella variabile P3.

```

          .DATA
P1        DB        3 DUP(?)
P2        DB        3 DUP(?)
P3        DB        4 DUP(?)
          .CODE
          ...
          MOV      AL, P1+2
          ADD      AL, P2+2                ; somma tra i byte meno significativi
          DAA                      ; conversione binario => BCD
          MOV      P3+3, AL                ; copia nel byte meno significativo
          MOV      AL, P1+1
          ADC      AL, P2+1                ; somma tra i 2 secondi byte + CF
          DAA                      ; conversione binario => BCD
          MOV      P3+2, AL                ; copia nel terzo byte
          MOV      AL, P1
          ADC      AL, P2                  ; somma tra i 2 primi byte + CF
          DAA                      ; conversione binario => BCD
          MOV      P3+1, AL                ; copia nel secondo byte
          JNC      continua                ; ultima somma ha generato carry ?
          MOV      P3, 01H                ; Sì: copia 1 nel primo byte
continua:  ...

```

Una soluzione alternativa, che utilizza un ciclo di istruzioni, è la seguente:

```

DIM      EQU      3
          .DATA
NUMA     DB        DIM DUP(?)
NUMB     DB        DIM DUP(?)
NUMC     DB        (DIM+1) DUP(?)
          .CODE
          ...
          CLC                      ; azzeramento del flag CF
          LEA      SI, BYTE PTR NUMA+DIM-1 ; offset di NUMA+2 in SI
          LEA      DI, BYTE PTR NUMB+DIM-1 ; offset di NUMB+2 in DI
          LEA      BX, BYTE PTR NUMC+DIM   ; offset di NUMC+3 in BX
          MOV      CX, DIM                 ; 3 iterazioni
ciclo:    MOV      AL, [SI]
          ADC      AL, [DI]                ; somma tra 2 byte + CF
          PUSHF
          DAA
          MOV      [BX], AL
          DEC      SI
          DEC      DI
          DEC      BX
          POPF
          LOOP     ciclo
          JNC      continua                ; è stato generato carry?
          MOV      [BX], 01H                ; Sì: copia 1 nel primo byte
continua: ...

```

10.9.2. Le istruzioni AAS e DAS

Le istruzioni **AAS** (*Ascii Adjust for Subtraction*) e **DAS** (*Decimal Adjust for Subtraction*) vengono impiegate nelle operazioni di sottrazione tra numeri BCD. Esse non hanno operandi ed il loro formato è il seguente:



AAS
DAS

Le istruzioni AAS e DAS eseguono la conversione da rappresentazione binaria a BCD del risultato di una istruzione SUB. L'istruzione AAS si usa per convertire un numero binario in numero BCD non impaccato, mentre l'istruzione DAS si usa per convertire un numero binario in numero BCD impaccato. Ambedue le istruzioni lavorano sul contenuto del registro AL. L'istruzione AAS modifica lo stato dei flag CF e AF, gli altri flag di stato assumono valore indefinito; l'istruzione DAS modifica il valore di tutti i flag di stato tranne OF, che non viene modificato.

L'operazione di sottrazione tra due numeri BCD è molto simile a quella di addizione. I passi da eseguire sono:

1. si copia il minuendo in AL;
2. si sottrae il sottraendo ad AL;
3. si esegue la conversione.

Esempi

Il seguente frammento di codice esegue la sottrazione tra i numeri BCD non impaccati U1 ed U2, memorizzando il risultato in AL:

```

U1      .DATA
        DB      ?
U2      DB      ?
        .CODE
        MOV     AL, U1
        SUB     AL, U2      ; U1 - U2
        AAS     ; conversione da binario a BCD

```

Il seguente frammento di codice esegue la sottrazione tra i numeri BCD impaccati P1 e P2, memorizzando il risultato in AL:

```

P1      .DATA
        DB      ?
P2      DB      ?
        .CODE
        MOV     AL, P1
        SUB     AL, P2      ; P1 - P2
        DAS     ; conversione da binario a BCD

```

Se il sottraendo è maggiore del minuendo, le istruzioni AAS e DAS forzano il flag CF ad 1 per indicare la presenza di un prestito (*borrow*).

Nel caso in cui il numero BCD sia composto di più byte, occorre eseguire la sottrazione in più passi, sottraendo un byte alla volta, nel modo seguente:

1. si sottrae una cifra alla volta a partire dalla cifra meno significativa;
2. si esegue la conversione di ogni cifra a numero BCD attraverso l'istruzione AAS.

Quando si sottraggono i byte successivi al primo bisogna tenere conto di eventuali prestiti generati dalla sottrazione precedente; l'istruzione di sottrazione SBB permette tale operazione. La differenza tra due numeri BCD impaccati utilizza l'istruzione di conversione DAS.

Esempi

Il seguente frammento di codice esegue la sottrazione tra due numeri BCD non impaccati di 3 cifre (U1 e U2), memorizzando il risultato nella variabile U3.

```

      .DATA
U1    DB      3 DUP(?)
U2    DB      3 DUP(?)
U3    DB      3 DUP(?)
      .CODE
      ...
      MOV     AL, U1+2
      SUB     AL, U2+2      ; diff. tra i byte meno significativi
      AAS     ; conversione da binario a BCD
      MOV     U3+2, AL      ; copia nel byte meno significativo
      MOV     AL, U1+1
      SBB     AL, U2+1      ; diff. tra i 2 secondi byte - CF
      AAS     ; conversione da binario a BCD
      MOV     U3+1, AL      ; copia nel secondo byte
      MOV     AL, U1
      SBB     AL, U2        ; diff. tra i 2 primi byte - CF
      AAA     ; conversione da binario a BCD
      MOV     U3, AL        ; copia nel byte più significativo
      JC      sub_error     ; sottrazione ha generato un borrow ?
      ...                  ; No: ok
sub_error: ...              ; Si: errore

```

In caso di *borrow* dopo la sottrazione tra i byte più significativi, l'operazione genera un risultato errato che deve essere opportunamente gestito.

Il seguente frammento di codice esegue la sottrazione tra due numeri BCD impaccati di 3 cifre (P1 e P2), memorizzando il risultato nella variabile P3.

```

DIM    EQU     3
      .DATA
P1     DB      DIM DUP(?)
P2     DB      DIM DUP(?)
P3     DB      DIM DUP(?)
      .CODE
      ...
      MOV     AL, P1+2
      SUB     AL, P2+2      ; diff. tra i byte meno significativi
      DAS     ; conversione da binario a BCD
      MOV     P3+2, AL      ; copia nel byte meno significativo
      MOV     AL, P1+1
      SBB     AL, P2+1      ; diff. tra i 2 secondi byte - CF
      DAS     ; conversione da binario a BCD
      MOV     P3+1, AL      ; copia nel secondo byte
      MOV     AL, P1
      SBB     AL, P2        ; diff. tra i 2 primi byte - CF
      DAS     ; conversione da binario a BCD
      MOV     P3, AL        ; copia nel byte più significativo
      JC      sub_error     ; sottrazione ha generato un borrow ?
      ...                  ; No: ok
sub_error: ...              ; Si: errore

```

10.9.3. L'istruzione AAM

L'istruzione **AAM** (*Ascii Adjust for Multiplication*) viene impiegata nella moltiplicazione tra numeri BCD. Essa è priva di operandi ed usa il seguente formato:



AAM

Il processore permette di eseguire operazioni di moltiplicazione esclusivamente tra numeri BCD non impaccati. Nel caso di moltiplicazioni tra numeri BCD impaccati, occorre prima convertire i fattori in numeri BCD non impaccati. L'istruzione AAM modifica i flag PF, SF e ZF, mentre i flag AF, CF e OF assumono valore indefinito.

Per moltiplicare due numeri BCD occorre moltiplicare un byte alla volta, eseguendo le seguenti operazioni:

1. si copia il primo fattore nel registro AL;
2. si moltiplica il contenuto del registro AL per il secondo fattore usando l'istruzione MUL;
3. si converte il risultato attraverso l'istruzione AAM.

Poiché ogni operando può variare nell'intervallo tra 0 a 9, il risultato dell'istruzione MUL può variare nell'intervallo compreso tra 0 e 81; l'istruzione AAM converte il risultato della moltiplicazione ponendo la cifra più significativa in AH e quella meno significativa in AL.

Esempio

Il seguente frammento di codice esegue la moltiplicazione tra i due numeri BCD non impaccati U1 ed U2; la cifra più significativa del risultato è posta in AH, quella meno significativa in AL:

```

.DATA
U1      DB      ?
U2      DB      ?
.CODE
...
MOV     AL, U1
MUL     U2      ; AX = U1 * U2
AAM     ; conversione da binario a BCD

```

La moltiplicazione tra due numeri BCD composti da più cifre è un'operazione complicata ed esula dagli scopi del presente testo.

L'istruzione AAM può essere utilizzata anche per eseguire divisioni per 10: in particolare il contenuto di AL viene diviso per 10: il quoziente è posto in AH ed il resto in AL.

Esempio

Il seguente frammento di codice esegue la divisione per 10 della costante 126.

```

.CODE
...
MOV     AL, 126
AAM     ; in AH = 12, in AL = 6
...

```

10.9.4. L'istruzione AAD

L'istruzione **AAD** (*Ascii Adjust for Division*) viene impiegata nella divisione tra numeri BCD. Essa è priva di operandi ed il suo formato è il seguente:



AAD

L'istruzione AAD esegue la conversione del numero BCD non impaccato memorizzato in AX nell'equivalente numero binario senza segno su 1 byte. Essa modifica i flag PF, SF e ZF, mentre i flag AF, CF e OF assumono valore indefinito.

Il processore può eseguire esclusivamente operazioni di divisione tra numeri BCD non impaccati. Per eseguire l'operazione di divisione tra numeri BCD impaccati occorre prima convertire dividendo e divisore in numeri BCD non impaccati.

Il processore esegue un'operazione di divisione tra un numero BCD non impaccato su 2 byte ed un numero BCD non impaccato su un byte.

A differenza delle altre operazioni tra numeri BCD, la divisione richiede che la conversione sia effettuata prima dell'esecuzione dell'operazione di divisione. La sequenza di operazioni da effettuare è la seguente:

1. si copia il dividendo in AX;
2. si esegue la conversione attraverso l'istruzione AAD;
3. si divide il contenuto del registro AX per il divisore.

L'istruzione AAD assume che il registro AX contenga una coppia di numeri BCD non impaccati (la cifra più significativa in AH e quella meno significativa in AL). Essa converte il numero in un valore binario copiato in AL (il contenuto del registro AH è sempre nullo).

Dopo l'istruzione DIV il quoziente sarà contenuto in AL ed il resto in AH.

Se il quoziente è un numero minore di 10, il valore binario di AL è equivalente al numero BCD; altrimenti occorrerà convertire questo numero nelle due cifre separate. Il modo più facile per fare questa operazione prevede l'impiego dell'istruzione AAM; questa istruzione distrugge il valore di AH (il resto), che va quindi preventivamente salvato.

Esempio

Il seguente frammento di codice esegue una divisione tra i due numeri BCD non impaccati U2 ed U1:

```

    .DATA
U1      DB      ?
U2      DB      2 DUP(?)

    .CODE
...
MOV     AH, U2
MOV     AL, U2+1    ; in AX: dividendo
AAD     ; conversione da BCD a numero binario
DIV     U1          ; U2 / U1
; in AL il quoziente, in AH il resto
MOV     BL, AH      ; salvataggio del resto in BL
AAM     ; AL / 10: cifra più significativa in AH;
; cifra meno significativa in AL
...

```

Come per la moltiplicazione, la divisione tra numeri BCD su più cifre è complicata e non verrà qui affrontata.

11. Le istruzioni per la manipolazione dei bit

In questo capitolo vengono descritte le istruzioni per la manipolazione dei bit all'interno di una parola. Queste possono essere classificate in *istruzioni logiche*, che permettono di modificare o controllare uno o più bit ed *istruzioni di scorrimento*, che permettono di cambiare la posizione dei bit.

11.1. Le istruzioni logiche

Le istruzioni logiche implementano le operazioni logiche elementari. Esse sono:

- **AND** (*logical AND*);
- **OR** (*logical inclusive OR*);
- **XOR** (*exclusive OR*);
- **NOT** (*logical NOT*);
- **TEST** (*logical compare*).

11.1.1. L'istruzione AND

L'istruzione AND esegue l'operazione logica *and* bit a bit. Il suo formato è il seguente:



AND destinazione, sorgente

L'istruzione AND esegue l'operazione logica *and* bit a bit tra l'operando *destinazione* e l'operando *sorgente*: l'*i-esimo* bit del risultato viene calcolato con un'operazione di *and* tra l'*i-esimo* bit dell'operando *sorgente* e l'*i-esimo* bit dell'operando *destinazione*; l'operando *sorgente* rimane invariato.

L'operando *destinazione* può essere un registro o una locazione di memoria; l'operando *sorgente* può essere un registro, una locazione di memoria oppure un valore immediato. Gli operandi *sorgente* e *destinazione* devono avere la stessa lunghezza e non possono essere entrambi una locazione di memoria. L'istruzione AND aggiorna il valore di tutti i flag di stato, tranne il

flag AF.

L'operando *sorgente* è anche detto *maschera di bit* (*bit mask*) poiché permette di selezionare i bit dell'operando *destinazione* da modificare.

La Fig. 11.1 riporta un esempio di funzionamento dell'istruzione AND; la Tab. 11.1 riassume la tavola della verità della funzione logica *and*.

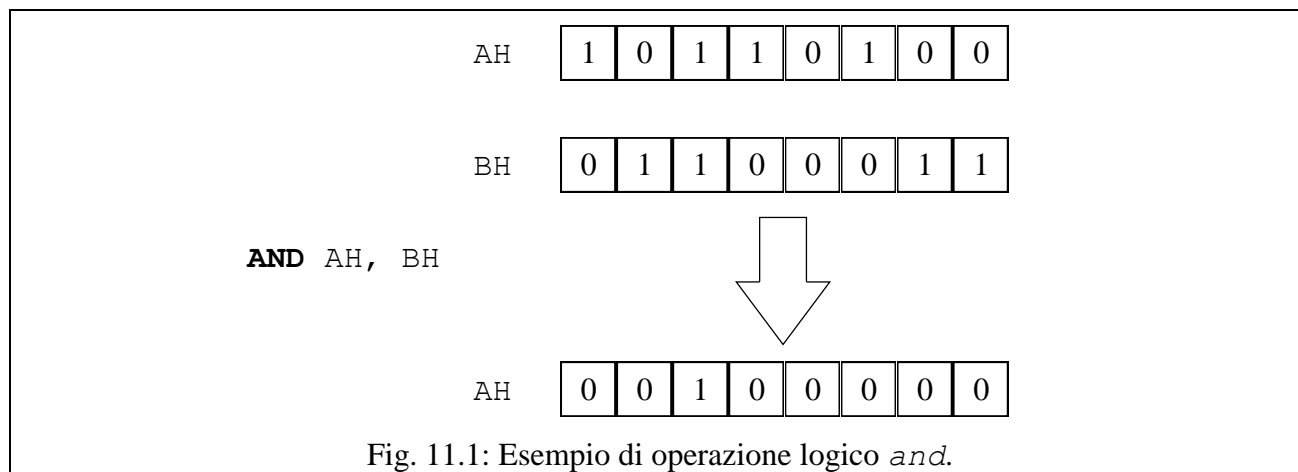


Fig. 11.1: Esempio di operazione logica *and*.

b_1	b_2	$b_1 \text{ and } b_2$
0	0	0
0	1	0
1	0	0
1	1	1

Tab. 11.1: Operazione logica *and*.

Esercizio: *Conversione da codice ASCII a numero BCD.*

Si vuole realizzare un frammento di programma che esegua la conversione di una cifra dal codice ASCII alla corrispondente codifica BCD non impaccata.

I codici ASCII dei numeri variano da 30H per il carattere '0' a 39H per il carattere '9'; per ricavare la rappresentazione BCD è sufficiente *mascherare* i quattro bit più significativi del codice ASCII. La soluzione proposta in linguaggio C è la seguente:

```
main()
{
    int i, num[4];
    char ascii[4];
    ...
    for (i=0 ; i < 4 ; i++)
        num[i] = (ascii[i] & 0x0F);
    ...
}
```

La soluzione proposta in linguaggio Assembler è la seguente:

```

LUNG      EQU      4
          .MODEL    small
          .DATA
ASCII     DB        LUNG DUP (?)
NUM       DB        LUNG DUP (?)
          .CODE
          ...
          MOV       SI, 0
          MOV       CX, LUNG
ciclo:    MOV       AL, ASCII[SI]    ; copia il carattere ascii in AL
          AND       AL, 0FH          ; mascheramento dei 4 bit alti
          MOV       NUM[SI], AL      ; copia in NUM del valore numerico
          INC       SI              ; scansione del vettore
          LOOP      ciclo
          ...

```

11.1.2. L'istruzione OR

L'istruzione OR esegue l'operazione logica *or* bit a bit. Il suo formato è il seguente:



OR *destinazione, sorgente*

L'istruzione OR esegue l'operazione logica *or* bit a bit tra l'operando *destinazione* e l'operando *sorgente*; il risultato di tale operazione viene copiato nell'operando *destinazione*, mentre l'operando *sorgente* rimane invariato.

L'operando *destinazione* può essere un registro od una locazione di memoria; l'operando *sorgente* può essere un registro, una locazione di memoria oppure un valore immediato. Gli operandi *sorgente* e *destinazione* devono avere la stessa lunghezza e non possono essere entrambi una locazione di memoria. L'istruzione OR aggiorna il valore di tutti i flag di stato tranne il flag AF.

La Fig. 11.2 riporta un esempio di funzionamento dell'istruzione OR; la Tab. 11.2 riassume l'operazione svolta dalla funzione logica *or*.

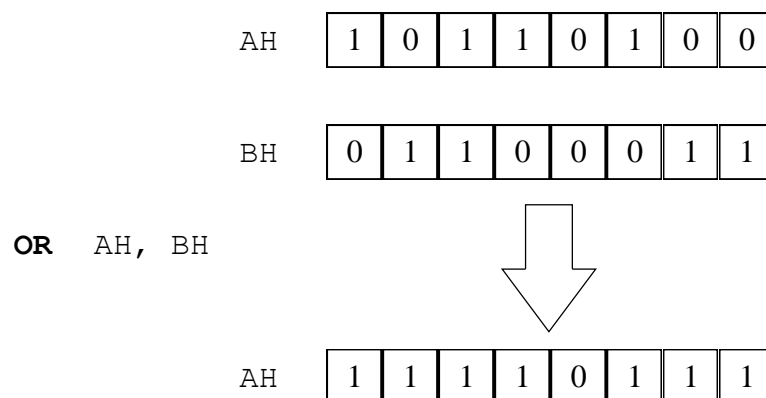


Fig. 11.2: Esempio di operazione logica *or*.

b_1	b_2	$b_1 \text{ or } b_2$
0	0	0
0	1	1
1	0	1
1	1	1

Tab. 11.2: Operazione logica *or*.

Esercizio: Conversione da numero BCD a codice ASCII.

Si vuole realizzare un frammento di programma che effettui la conversione degli elementi di un vettore dalla rappresentazione BCD non impaccata al corrispondente valore ASCII.

La soluzione proposta in linguaggio C è la seguente:

```
main()
{
    int i,num[4];
    char ascii[4];
    ...
    for (i=0 ; i<4 ; i++)
        ascii[i] = (num[i] | 0x30);
    ...
}
```

La soluzione proposta in linguaggio Assembler è la seguente:

```
LUNG      EQU      4
           .MODEL   small
           .DATA
NUM        DB       LUNG DUP (?)
ASCII      DB       LUNG DUP (?)
           .CODE
           ...
           MOV      SI, 0
           MOV      CX, LUNG
ciclo:     MOV      AL, NUM[SI]      ; copia in AL del valore numerico
           OR       AL, 30H         ; maschera di OR: 00110000
           MOV      ASCII[SI], AL   ; copia in ASCII del carattere ascii
           INC      SI              ; scansione del vettore
           LOOP     ciclo
           ...
```

11.1.3. L'istruzione XOR

L'istruzione XOR esegue l'operazione logica *exor* bit a bit. Il suo formato è il seguente:



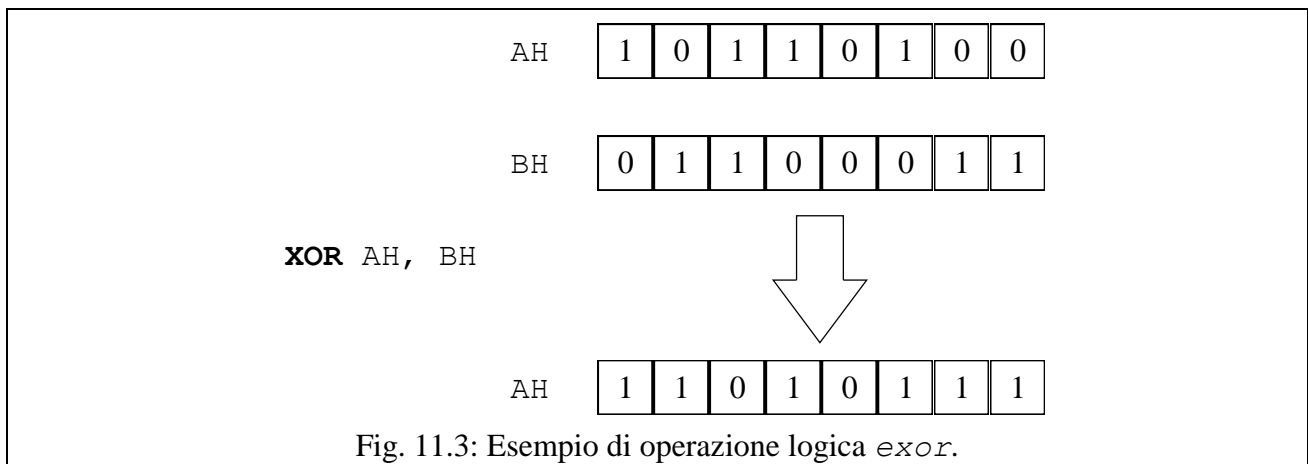
XOR *destinazione, sorgente*

L'istruzione XOR esegue l'operazione logica *exor* bit a bit tra l'operando *destinazione* e l'operando *sorgente*; il risultato di tale operazione viene copiato nell'operando *destinazione*, mentre l'operando *sorgente* rimane invariato.

L'operando *destinazione* può essere un registro od una locazione di memoria; l'operando *sorgente* può essere un registro, una locazione di memoria oppure un valore immediato. Gli operandi *sorgente* e *destinazione* devono avere la stessa lunghezza e non possono essere entrambi una locazione di memoria. L'istruzione XOR aggiorna il valore di tutti i flag di stato (tranne

il flag AF).

La Fig. 11.3 riporta un esempio di funzionamento dell'istruzione XOR; la Tab. 11.3 riassume l'operazione svolta dalla funzione logica *exor*.



b_1	b_2	$b_1 \text{ exor } b_2$
0	0	0
0	1	1
1	0	1
1	1	0

Tab. 11.3: Operazione logica *exor*.

L'istruzione XOR può essere utilizzata per eseguire l'azzeramento del contenuto di un registro.

Esempio

Le due istruzioni seguenti sono equivalenti ed eseguono entrambe l'operazione di azzeramento del registro AX. La seconda istruzione richiede un numero di cicli macchina inferiore, ed è dunque preferibile rispetto alla prima.

```
MOV    AX, 0
XOR    AX, AX
```

11.1.4. L'istruzione NOT

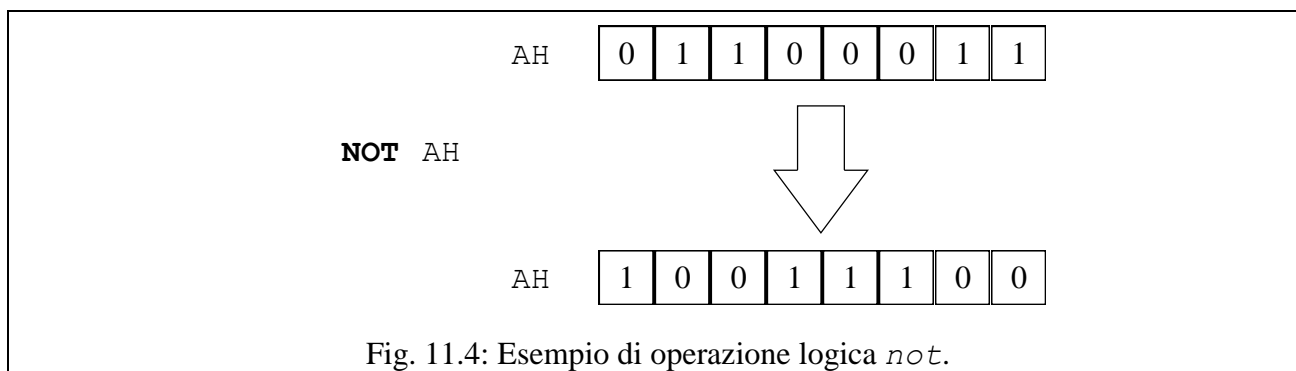
L'istruzione NOT esegue l'operazione logica di complementazione bit a bit del contenuto di un operando. Il suo formato è il seguente:



NOT *operando*

L'istruzione NOT complementa il contenuto dell'*operando* che può essere un registro oppure il contenuto di una locazione di memoria. L'istruzione NOT non modifica il valore di alcun flag.

La Fig. 11.4 riporta un esempio di funzionamento dell'istruzione NOT; la Tab. 11.4 riassume l'operazione svolta dalla funzione logica *not*.



<i>b</i>	<i>not b</i>
0	1
1	0

Tab. 11.4: Operazione logica *not*.

11.1.5. L'istruzione TEST

L'istruzione TEST permette di controllare il valore di una parola. Il suo formato è il seguente:



TEST destinazione, sorgente

L'istruzione TEST esegue l'operazione logica *and* bit a bit tra l'operando *destinazione* e l'operando *sorgente*, senza modificarne il contenuto; l'istruzione aggiorna opportunamente il valore di tutti i flag di stato tranne il flag AF. L'operando *destinazione* può essere un registro oppure una locazione di memoria; l'operando *sorgente* può essere un registro, una locazione di memoria o un valore immediato. Gli operandi *sorgente* e *destinazione* devono avere la stessa lunghezza e non possono essere entrambi una locazione di memoria.

Esempio

Il seguente frammento di codice azzerà il bit 2 della variabile DATO se il bit 7 vale 1.

```

TEST    DATO, 10000000B    ; DATO AND 80H
JZ      continua          ; flag ZF = 1 ?
AND     DATO, 11111011B    ; No: azzerà bit 2
continua: ...

```

Esercizio: Testa o croce?

Si vuole realizzare un programma che simuli il lancio di una moneta. Una possibile realizzazione in C è la seguente:

```

#include <time.h>
main()
{ time_t t;
  srand ((unsigned) time(&t));
  if (rand()%2) printf("TESTA\n");
  else printf("CROCE\n");
}

```

La soluzione proposta in linguaggio Assembler è la seguente:

```

.MODEL    small
.STACK
.DATA
TESTA     DB      "TESTA",0Dh,0Ah,"$"
CROCE     DB      "CROCE",0DH,0AH,"$"
.CODE
.STARTUP
MOV       AH, 2CH
INT       21H          ; lettura in DX del timer di sistema
TEST     DH, 1          ; bit 0 = 0 ?
JNZ      lab_t          ; No: va a lab_t
LEA       DX, CROCE     ; Sì: copia in DX l'offset di CROCE
JMP       video
lab_t:    LEA       DX, TESTA ; copia in DX l'offset di TESTA
video:    MOV       AH, 09H
INT       21H          ; visualizza su video
.EXIT
END

```

Esercizio: *Somma dei numeri pari e dei numeri dispari.*

Si vuole realizzare un frammento di programma che esegua la somma dei valori assoluti di tutti i numeri pari e di tutti i numeri dispari contenuti in un vettore di numeri interi. La soluzione proposta in linguaggio C è la seguente:

```

#include <math.h>
main()
{ int vett[10], i, pari, dispari;
  ...
  for (i=pari=dispari=0 ; i<10 ; i++)
    if (vett[i] & 1) dispari += abs(vett[i]);
    else pari += abs(vett[i]);
}

```

La soluzione proposta in linguaggio Assembler è la seguente:

```

LUNG      EQU      10
.MODEL    small
.DATA
VETT      DW      LUNG DUP (?)
PARI      DW      ?
DISPARI    DW      ?
.CODE
...
MOV       CX, LUNG
XOR       SI, SI      ; azzeramento del registro SI
MOV       PARI, 0      ; azzeramento della variabile PARI
MOV       DISPARI, 0   ; azzeramento della variabile DISPARI
ciclo:    MOV       AX, VETT[SI] ; copia in AX dell'elemento del vettore
CMP       AX, 0        ; AX > 0 ?
JNL      tst          ; Sì: va a tst
NEG       AX           ; complemento a 2 di AX
tst:      TEST     AX, 1 ; bit 0 = 0 ?
JNZ      dispa        ; No: va a dispa
ADD       PARI, AX     ; PARI = PARI + AX
JMP       continua
dispa:    ADD       DISPARI, AX ; DISPARI = DISPARI + AX
continua: ADD       SI, 2      ; scansione del vettore
LOOP     ciclo
...


```

11.2. Le istruzioni di scorrimento

Le istruzioni di scorrimento permettono di modificare la posizione dei bit all'interno di una parola, spostandoli verso sinistra o verso destra di un numero definito di posizioni, e possono essere classificate in:

- istruzioni di *shift*, in cui l'ultimo bit nella direzione dello scorrimento è copiato nel flag CF ed il primo bit è posto a 0 o ad un valore uguale a quello del bit di segno;
- istruzioni di *rotazione*, in cui l'ultimo bit nella direzione della rotazione viene copiato al posto del primo bit.

Il formato delle istruzioni di scorrimento è il seguente:

	OPCODE <i>operando, conteggio</i>
---	--

OPCODE è il nome di una istruzione di scorrimento; il campo *operando* può essere o un registro oppure una locazione di memoria; il campo *conteggio* può essere od il valore immediato 1 oppure il registro CL.

A partire dall'80186 è possibile utilizzare come *conteggio* qualunque valore immediato.


11.2.1. Le istruzioni di shift

Le istruzioni di shift permettono di eseguire lo scorrimento verso sinistra o verso destra di un determinato numero di posizioni. Esse sono:

- **SHL** (*SHift logical Left*);
- **SHR** (*SHift logical Right*);
- **SAL** (*Shift Arithmetic Left*);
- **SAR** (*Shift Arithmetic Right*).

Le istruzioni SHL e SHR

Le istruzioni SHL e SHR (Fig. 11.5) eseguono rispettivamente lo scorrimento a sinistra e a destra di un operando. Il loro formato è il seguente:

	<i>SHL</i> <i>operando, conteggio</i>
	<i>SHR</i> <i>operando, conteggio</i>

L'istruzione SHL esegue lo scorrimento a sinistra del contenuto dell'*operando* di un numero di posizioni pari al valore di *conteggio*.

L'istruzione SHR esegue lo scorrimento a destra del contenuto dell'*operando* di un numero di posizioni pari al valore di *conteggio*.

L'ultimo bit in uscita viene copiato nel flag CF; tutte le posizioni vuote vengono caricate con bit di valore 0. Le istruzioni SHL e SHR aggiornano il valore di tutti i flag di stato (tranne AF).

Uno degli usi principali delle istruzioni di *shift* è quello di eseguire operazioni di moltiplicazione o divisione per potenze di due su numeri senza segno. Lo *shift* a destra di n posizioni è equivalente alla divisione per 2^n ; lo *shift* a sinistra di n posizioni è equivalente alla moltiplicazione per 2^n .

Esempi

Il seguente frammento di codice divide per 8 il contenuto della variabile DATO:

```
MOV    CL, 3
SHR    DATO, CL
```

Il seguente frammento di codice esegue la stessa operazione su un processore 80386:

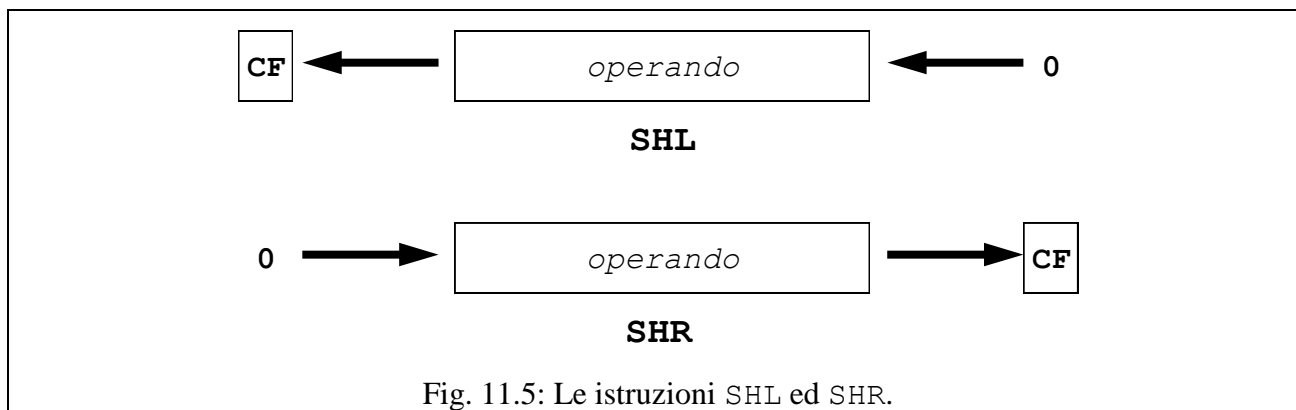
```
.386
...
SHR    DATO, 3
```

Il seguente frammento di codice moltiplica per 16 il contenuto della variabile DATO:

```
MOV    CL, 4
SHL    DATO, CL
```

Il seguente frammento di codice esegue la stessa operazione su un processore 80386:

```
.386
...
SHL    DATO, 4
```



Esercizio: Isolamento e spostamento di un campo di bit.

Si vuole realizzare un frammento di programma che esegua la selezione dei bit 3, 4 e 5 in una parola da 8 bit e che posizioni tali bit nella parte bassa della parola, azzerando i 5 bit più significativi. La soluzione proposta in linguaggio C è la seguente:

```
main()
{
    char dato;
    ...
    dato &= 0x38;
    dato >>= 3;
    ...
}
```


La soluzione proposta in linguaggio Assembler è la seguente:

```

        .386
        .MODEL    small
        .DATA
DATO    DB        ?
        .CODE
        ...
        MOV     AL, DATO      ; copia in AL della variabile DATO
        AND     AL, 00111000B ; selezione dei bit 3, 4 e 5
        SHR     AL, 3         ; scorrimento a destra di 3 posizioni
        MOV     DATO, AL      ; copia di AL in DATO
        ...

```

Esercizio: Calcolo dell'area di un triangolo (I versione).

Si vuole realizzare un frammento di programma che esegua il calcolo dell'area di un triangolo. La soluzione proposta in linguaggio C è la seguente:

```

main()
{
    int base, altezza, area;
    ...
    area = (base * altezza) >> 1;
    ...
}

```

La soluzione proposta in linguaggio Assembler è la seguente:

```

        .MODEL    small
        .DATA
BASE     DB        ?
ALTEZZA  DB        ?
AREA     DW        ?
        .CODE
        ...
        MOV     AL, BASE      ; copia in AL della variabile BASE
        MUL     ALTEZZA      ; AX = BASE * ALTEZZA
        SHR     AX, 1         ; divisione per 2
        MOV     AREA, AX     ; copia di AX nella variabile AREA
        ...

```

Le istruzioni SAL e SAR

Le istruzioni SAL e SAR (Fig. 11.6) permettono di eseguire operazioni di moltiplicazione e divisione tra un numero intero con segno ed una potenza di 2. Il loro formato è il seguente:



SAL	operando, conteggio
SAR	operando, conteggio

L'istruzione SAL esegue lo spostamento verso sinistra dei bit dell'*operando* di un numero di posizioni pari al valore del campo *conteggio*; i bit vuoti a destra sono riempiti di bit a 0.

L'istruzione SAR esegue lo spostamento verso destra dei bit dell'*operando* di un numero di posizioni pari al valore del campo *conteggio*; i bit vuoti a sinistra sono riempiti di bit pari al valore del bit più significativo dell'*operando*.

Le istruzioni SAL e SAR aggiornano il valore di tutti i flag di stato (tranne AF).

L'istruzione SAL permette di eseguire l'operazione di moltiplicazione di un numero intero con

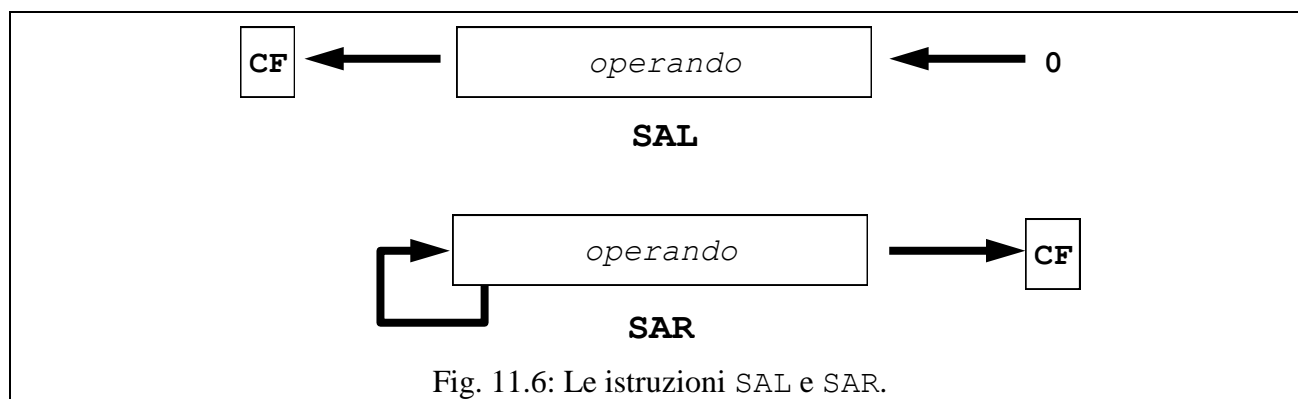
segno per una potenza di 2. In una moltiplicazione i bit bassi sono azzerati sia per numeri negativi che per numeri positivi. Eseguendo l'istruzione SAL l'ultimo bit in uscita viene copiato nel flag CF. L'istruzione SAL è del tutto equivalente all'istruzione SHL.

L'istruzione SAR permette di eseguire l'operazione di divisione di un numero intero con segno per una potenza di 2. Per effettuare correttamente l'operazione di divisione occorre eseguire l'estensione del segno verso il bit più significativo, ossia per ogni shift a destra deve essere caricato un 1 nel caso di numero negativo ed uno 0 nel caso di numero positivo. Per ogni shift a destra il bit meno significativo viene copiato nel flag CF.

Esempio

La seguente istruzione esegue la divisione per 8 della variabile DATO di tipo intero con segno:

```
.386
...
SAR    DATO, 3    ; shift aritmetico a destra di 3 bit
```



11.2.2. Le istruzioni di rotazione

Le istruzioni di rotazione permettono di far ruotare il contenuto di un operando. Esse sono:

- **ROR** (*ROtate Right*);
- **ROL** (*ROtate Left*);
- **RCR** (*Rotate Right through Carry*);
- **RCL** (*Rotate Left through Carry*).

Le istruzioni ROR e ROL

Le istruzioni ROR e ROL (Fig. 11.7) permettono di eseguire la rotazione del contenuto di un operando. Il loro formato è il seguente:

	<i>ROR</i>	<i>operando, conteggio</i>
	<i>ROL</i>	<i>operando, conteggio</i>

Nelle istruzioni ROR e ROL l'ultimo bit in uscita viene copiato nel flag CF; in particolare in una rotazione a destra (ROR) in CF è copiato il bit meno significativo, mentre in una rotazione a sinistra (ROL) in CF è copiato il bit più significativo. Le istruzioni ROR e ROL aggiornano esclusivamente il valore dei flag CF e OF.

Esercizio: Scambio tra il contenuto dei nibble in un byte.

Si vuole realizzare un frammento di programma che esegua lo scambio tra il *nibble* meno significativo (4 bit bassi) ed il nibble più significativo (4 bit alti) all'interno di un byte.

La soluzione proposta in linguaggio C è la seguente:

```
main()
{
    char numero1, numero2, temp;
    ...
    temp = (numero1 & 0xF0);
    numero2 = (numero1 << 4);
    temp >>= 4;
    numero2 |= temp;
    ...
}
```

La soluzione proposta in linguaggio Assembler è la seguente:

```

                .386
                .MODEL    small
                .DATA
NUMERO1        DB        ?
NUMERO2        DB        ?
                ...
                .CODE
                ...
                MOV       AL, NUMERO1      ; copia della variabile NUMERO1 in AL
                ROL       AL, 4            ; rotazione a sinistra di 4 bit
                MOV       NUMERO2, AL      ; copia di AL nella variabile NUMERO2
                ...
```

Le istruzioni RCR e RCL

Le istruzioni RCR e RCL (Fig. 11.7) permettono di eseguire la rotazione del contenuto di un operando utilizzando il flag CF come bit aggiuntivo; il loro formato è il seguente:



<i>RCR</i>	<i>operando, conteggio</i>
<i>RCL</i>	<i>operando, conteggio</i>

Nelle istruzioni RCR e RCL il bit di carry è considerato un bit aggiuntivo che partecipa alla rotazione.

Con RCR la rotazione avviene come se il flag CF fosse un bit in più posto alla destra della parola da ruotare; con RCL la rotazione avviene come se il flag CF fosse un bit in più posto alla sinistra della parola da ruotare. Le istruzioni RCR e RCL aggiornano esclusivamente il valore dei flag CF e OF.

Esercizio: Divisione per 16 di una parola da 32 bit.

Si vuole realizzare un frammento di programma che esegua la divisione per 16 di un numero intero memorizzato su 32 bit. La soluzione proposta in linguaggio Assembler è la seguente:

```

        .MODEL    small
        .DATA
NUMERO  DD        ?
        .CODE
        ...
        MOV      CX, 4
ciclo:  SHR      WORD PTR NUMERO+2,1  ; divisione per 2 della word alta
                                   ; il bit 16 viene copiato in CF
        RCR      WORD PTR NUMERO,1   ; rotazione a destra di 1 bit
                                   ; CF viene copiato nel bit 15
        LOOP     ciclo
        ...

```

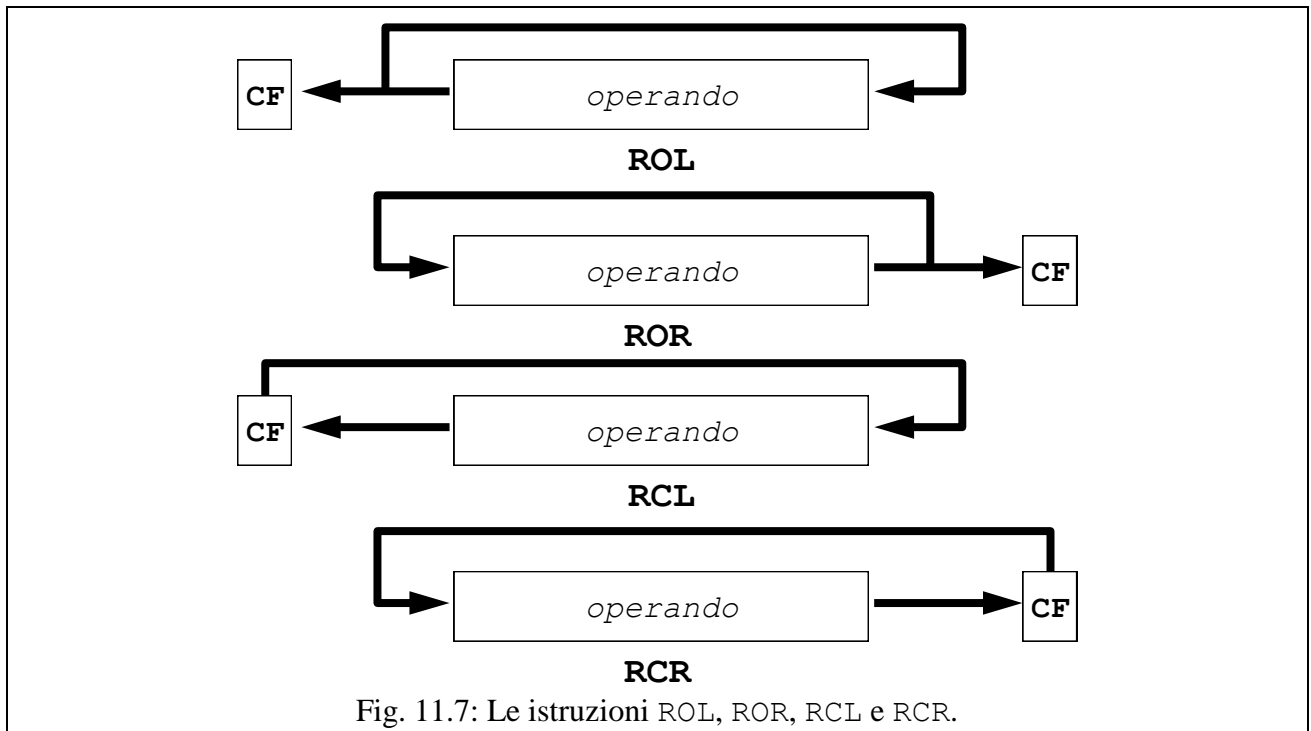
Esercizio: Calcolo dell'area di un triangolo (II versione).

Si mostra ora una seconda versione del frammento di programma che esegue il calcolo dell'area di un triangolo. La base, l'altezza e l'area sono variabili di tipo word ed il programma verifica che il calcolo dell'area non produca *overflow*. La soluzione proposta in linguaggio Assembler è la seguente:

```

        .MODEL    small
        .DATA
BASE    DW        ?
ALTEZZA DW        ?
AREA    DW        ?
        .CODE
        ...
        MOV      AX, BASE           ; copia in AX della variabile BASE
        MUL      ALTEZZA           ; DX,AX = BASE * ALTEZZA
        SHR      DX, 1              ; carico CF con il bit 0 di DX
        RCR      AX, 1              ; divisione per 2 e copia di CF nel bit 15
        CMP      DX, 0              ; DX = 0 ?
        JNE      error             ; Sì: overflow nella moltiplicazione
        MOV      AREA, AX           ; copia di AX nella variabile AREA
error:   ...                        ; istruzioni di gestione dell'overflow

```



12. Le istruzioni per la manipolazione di stringhe

12.1. Introduzione

L'architettura x86 possiede un gruppo di istruzioni per la manipolazione di *stringhe*, ossia di sequenze contigue di byte o word. Tali istruzioni permettono di eseguire in maniera efficiente operazioni quali lo spostamento di un blocco di dati da un'area di memoria ad un'altra, la ricerca di un valore all'interno di un blocco, la scrittura di un blocco con un valore costante, ecc.

In particolare, le operazioni permesse sono la *copiatura* da una stringa sorgente ad una destinazione, il *confronto* tra due stringhe, la *ricerca* di un valore all'interno di una stringa e la *modifica* del contenuto di una stringa.

Tutte le istruzioni per la manipolazione delle stringhe hanno alcune caratteristiche simili:

- possono lavorare su una o due stringhe: le istruzioni per lo spostamento di un blocco e quelle per il confronto di due blocchi lavorano su due stringhe; quelle per la ricerca di un valore in un blocco, o per la scrittura di un valore costante in tutti gli elementi di un blocco, lavorano su una stringa;
- i due registri (uno per le istruzioni che lavorano su una sola stringa) utilizzati come indici all'interno delle due stringhe vengono automaticamente aggiornati al termine dell'operazione elementare, in modo da puntare ciascuno all'elemento successivo all'interno della rispettiva stringa;
- ciascuna istruzione esegue una singola operazione, lavorando su un singolo elemento alla volta; il processore fornisce tuttavia un meccanismo per la ripetizione di ogni istruzione o per un numero prefissato di volte o fino al verificarsi di una determinata condizione.

12.1.1. Preparazione dei registri

Tutte le istruzioni per la manipolazione delle stringhe richiedono che:

- la prima stringa (denominata *sorgente*) si trovi nel *segmento di dato* puntato da DS ed il registro SI memorizzi l'*indirizzo di offset* dell'elemento da elaborare all'interno della sequenza;

- l'eventuale seconda stringa (denominata *destinazione*) si trovi nel *segmento extra di dato* puntato da ES ed il registro DI memorizzi l'*indirizzo di offset* dell'elemento da elaborare.

Prima di utilizzare un'istruzione per la manipolazione di stringhe occorre quindi inizializzare opportunamente i registri coinvolti.

Data ed Extra Segment coincidenti

Se i dati sono contenuti in un unico segmento di dato, le operazioni da fare per predisporre i registri per la manipolazione delle stringhe sono le seguenti:

- copiare il contenuto del registro DS nel registro ES in modo da far coincidere i due segmenti;
- copiare gli offset delle due stringhe nei registri SI e DI.

Esempio

Il seguente frammento di codice prepara i registri per la manipolazione delle stringhe STR1 ed STR2 memorizzate nel segmento di dato. Per copiare il contenuto di DS in ES è stato utilizzato lo *stack*:

```

LUNG      .MODEL    small
          EQU       100
          .DATA
STR1      DB        LUNG DUP (?)
STR2      DB        LUNG DUP (?)
          .CODE
          ...
          PUSH     DS
          POP      ES           ; caricamento del registro ES
          LEA      SI, STR1     ; caricamento in SI dell'indirizzo
                                ; di partenza della stringa STR1
          LEA      DI, STR2     ; caricamento in DI dell'indirizzo
                                ; di partenza della stringa STR2
          ...

```

Data ed Extra Segment separati

In programmi complessi i dati sono suddivisi su più segmenti ed in generale i registri DS ed ES contengono indirizzi di segmenti diversi.

La condizione di *default* prevista dal processore è quella di avere la stringa sorgente nel segmento di *dato* e la stringa destinazione nel segmento *extra*: in questo caso per caricare i registri SI e DI basta eseguire l'istruzione LEA.

Esempio

Il seguente frammento di codice predispone i registri di indice SI e DI per la manipolazione delle stringhe STR1 (memorizzata nel segmento di dato) ed STR2 (memorizzata nel segmento *extra* di dato):

```

          .MODEL    compact           ; un segmento di codice
          EQU       100               ; più segmenti di dato
          .FARDATA  segm1
LUNG      DB        LUNG DUP (?)
STR1      .FARDATA  segm2
          DB        LUNG DUP (?)
STR2      .CODE
          ASSUME    DS:segm1, ES:segm2
          LEA      SI, STR1
          LEA      DI, STR2
          ...

```

Nel caso in cui la stringa destinazione si trovi nel segmento di dato occorre far coincidere temporaneamente il contenuto dei registri ES e DS, salvando il precedente valore di ES. Le operazioni da eseguire sono le seguenti:

1. salvare il contenuto di ES;
2. copiare il contenuto di DS in ES;
3. eseguire l'istruzione di manipolazione sulla stringa;
4. ripristinare il contenuto di ES.

Il registro ES può essere salvato nello *stack* ed il suo caricamento effettuato tramite l'istruzione LES.

Esempio

Nel frammento di codice seguente si fa l'ipotesi che i registri DS ed ES contengano indirizzi di segmento diversi e che le stringhe STR1 ed STR2 siano entrambe memorizzate nel segmento di dato indirizzato da DS. La doubleword STRADD contiene l'indirizzo completo della stringa destinazione; l'istruzione LES copia l'indirizzo di segmento in ES e l'indirizzo di offset in DI.

```

.model    compact                ; un segmento di codice
                                ; più segmenti di dato

LUNG      EQU      100
.FARDATA  segm1
STR1      DB       LUNG DUP (?)
STR2      DB       LUNG DUP (?)
STRADD    DD       STR2
.FARDATA  segm2
...
.CODE
ASSUME    DS:segm1, ES:segm2
...
PUSH     ES                    ; salvataggio di ES
LEA      SI, STR1              ; caricamento di SI
LES      DI, STRADD            ; caricamento di DI e ES
...                               ; istruzioni di manipolazione
POP      ES                    ; ripristino di ES
...

```

12.1.2. La ripetizione delle istruzioni per la manipolazione di stringhe

Ogni istruzione per la manipolazione di stringhe opera su un singolo elemento della stringa (byte o word) per volta. Per elaborare l'intera stringa occorre eseguire un ciclo che permetta di ripetere l'istruzione di manipolazione per tutti gli elementi della stringa stessa. Il processore mette a disposizione una classe di istruzioni utili per implementare i cicli per la manipolazione delle stringhe:

- **REP** (*REPeat string operation*);
- **REPE** (*REPeat string operation while Equal*)
- **REPNE** (*REPeat string operation while Not Equal*).

Queste istruzioni vengono utilizzate in abbinamento con una delle istruzioni di manipolazione ed appaiono sulla stessa riga del codice. Il loro formato è il seguente:



```

REP      string_istruz
REPE     string_istruz
REPNE    string_istruz

```

Ognuno di questi comandi ripete l'esecuzione dell'istruzione *string_istruz* per un numero di volte pari al contenuto del registro CX.

Le istruzioni `REPE` e `REPNE` possono essere utilizzate abbinate alle istruzioni di confronto e di ricerca presentate oltre: l'istruzione `REPE` ripete il ciclo finché il registro `CX` ha un valore diverso da 0 e le parole confrontate sono uguali (è utile per cercare una parola che non corrisponda ad un determinato valore); l'istruzione `REPNE` ripete il ciclo finché il registro `CX` ha un valore diverso da 0 e le parole confrontate sono diverse (è utile per cercare una parola che abbia un particolare valore).

Per usare le istruzioni `REP`, `REPE` e `REPNE` occorre dunque caricare il registro `CX` con la dimensione della stringa; analogamente a quanto avviene nel caso dell'istruzione `LOOP`, ad ogni esecuzione dell'istruzione di ripetizione il contenuto del registro `CX` viene decrementato di una unità. La differenza sostanziale tra l'istruzione `LOOP` e l'istruzione `REP` è che quest'ultima permette di ripetere un'unica istruzione di manipolazione di stringhe, mentre l'istruzione `LOOP` permette di ripetere una generica sequenza di istruzioni.

Esempio

Il seguente frammento di codice esegue un trasferimento di 100 byte dalla stringa `STR1` alla stringa `STR2`, utilizzando l'istruzione `MOVSB`, che sposta un elemento dalla stringa sorgente a quella destinazione, aggiornando poi i due registri indice `SI` e `DI`:

```

LUNG      .MODEL    small
          EQU      100
          .DATA
STR1      DB        LUNG DUP (?)
STR2      DB        LUNG DUP (?)
          .CODE
          ...
          PUSH     DS
          POP      ES
          LEA      SI, STR1          ; caricamento del registro SI
          LEA      DI, STR2          ; caricamento del registro DI
          MOV      CX, LUNG          ; caricamento del registro CX
          REP      MOVSB             ; ripetizione dell'istruzione MOVSB
          ...

```

Tale frammento di codice è operativamente equivalente a quello seguente che fa uso dell'istruzione `LOOP` per controllare il ciclo di istruzioni:

```

LUNG      .MODEL    small
          EQU      100
          .DATA
STR1      DB        LUNG DUP (?)
STR2      DB        LUNG DUP (?)
          .CODE
          ...
          PUSH     DS
          POP      ES
          LEA      SI, STR1          ; caricamento del registro SI
          LEA      DI, STR2          ; caricamento del registro DI
          MOV      CX, LUNG          ; caricamento del registro CX
ciclo:    MOVSB      ; copia da STR1 a STR2
          LOOP     ciclo
          ...

```

12.1.3. Aggiornamento del contenuto dei registri indice

Le istruzioni per la manipolazione delle stringhe prevedono che i registri `SI` e `DI` contengano l'offset delle parole da elaborare. Quando si esegue ripetutamente una stessa istruzione di manipolazione, il contenuto dei registri è automaticamente aggiornato per indirizzare la parola successiva

nella stringa, in base al valore del *flag di direzione* (DF), contenuto nella PSW:

- se il flag DF vale 0, dopo ogni esecuzione dell'istruzione di manipolazione il contenuto dei registri di indice è incrementato di un'unità, per le stringhe di byte, o di due unità, per le stringhe di word;
- se il flag DF vale 1, dopo ogni esecuzione dell'istruzione di manipolazione il contenuto dei registri di indice è decrementato di un'unità, per le stringhe di byte, o di due unità, per le stringhe di word.

A seconda del valore del flag DF le operazioni sulle stringhe vengono quindi eseguite *in avanti* (*forward*) oppure *all'indietro* (*backward*). Il valore del flag DF può essere modificato mediante le due istruzioni STD e CLD, che lo forzano rispettivamente a 1 e a 0.

Esempi

Il seguente frammento di codice copia la stringa STR1 nella stringa STR2, con una scansione in avanti.

```

LUNG      .MODEL    small
          EQU      100
          .DATA
STR1      DB        LUNG DUP (?)
STR2      DB        LUNG DUP (?)
          .CODE
          ...
          PUSH     DS
          POP      ES
          LEA      SI, STR1
          LEA      DI, STR2
          MOV      CX, LUNG
          CLD                      ; DF = 0: scansione in avanti
          REP      MOVSB
          ...

```

Il seguente frammento di codice copia la stringa STR1 nella stringa STR2 con una scansione all'indietro.

```

LUNG      .MODEL    small
          EQU      100
          .DATA
STR1      DB        LUNG DUP (?)
STR2      DB        LUNG DUP (?)
          .CODE
          ...
          PUSH     DS
          POP      ES
          LEA      SI, STR1 + SIZE STR1 - TYPE STR1
          LEA      DI, STR2 + SIZE STR2 - TYPE STR2
          MOV      CX, LUNG
          STD                      ; DF = 1: scansione all'indietro
          REP      MOVSB
          ...

```

I registri SI e DI devono essere inizializzati con l'indirizzo dell'ultimo elemento della stringhe STR1 ed STR2. L'istruzione MOVSB aggiorna il contenuto di SI e DI dopo la copiatura di ciascun byte; nel primo esempio DF vale 0 e MOVSB incrementa di una unità i registri di indice; nel secondo DF vale 1 e MOVSB decrementa di una unità i registri SI e DI.

È bene ricordarsi di aggiornare sempre il valore del flag DF prima di eseguire un ciclo di manipolazione di stringhe poiché non è garantito che il valore di DF sia mantenuto costante tra due cicli di

manipolazione. Infatti effettuando la chiamata ad una funzione di libreria C non viene ripristinato il valore del flag DF precedente alla chiamata.

12.1.4. Riassunto delle operazioni necessarie per manipolare le stringhe

Le operazioni da compiere per l'esecuzione di un ciclo di istruzioni per la manipolazione di stringhe sono:

1. preparazione dei registri;
2. caricamento del registro CX con il numero di elementi della stringa;
3. aggiornamento del flag DF;
4. esecuzione dell'istruzione REP (o REPE o REPNE) abbinata all'opportuna istruzione di manipolazione.

12.2. Copiatura di una stringa

Le istruzioni **MOVSB** (*MOVE Byte String*) e **MOVSW** (*MOVE Word String*) permettono di copiare una stringa da un indirizzo sorgente ad un indirizzo destinazione. Il loro formato è il seguente:



MOVSB
MOVSW

L'istruzione **MOVSB** copia il byte avente indirizzo DS:SI nella locazione di memoria ES:DI.

L'istruzione **MOVSW** copia la word avente indirizzo DS:SI nella locazione di memoria ES:DI.

I registri SI e DI devono contenere, rispettivamente, l'offset della stringa sorgente e di quella destinazione. Ad ogni esecuzione dell'istruzione vengono aggiornati entrambi i registri di indice.

Esempio

Il seguente frammento di codice copia la stringa di word STR1 in STR2.

```


LUNG      .MODEL    small
          EQU       100
          .DATA
STR1      DW        LUNG DUP (?)
STR2      DW        LUNG DUP (?)
          .CODE
          ...
          PUSH      DS
          POP       ES
          LEA       SI, STR1          ; caricamento di SI
          LEA       DI, STR2          ; caricamento di DI
          MOV       CX, LUNG          ; caricamento di CX
          CLD                           ; scansione in avanti
          REP       MOVSW              ; ripetizione del trasferimento
          ...

```

L'operazione di copiatura con scansione in avanti può causare errore nel caso in cui le due stringhe siano parzialmente "sovrapposte" in memoria. Si veda l'esempio seguente che illustra il meccanismo di errore.

Esempio

Supponiamo di avere due stringhe di lunghezza 100 byte i cui indirizzi di inizio distino 5 byte: siano, in altri termini, parzialmente sovrapposte:




```

.MODEL    small
LUNG      EQU      100
DISP      EQU      5
.DATA
STR        DB      LUNG DUP (?)
           DB      DISP DUP (?)
.CODE
...
PUSH      DS
POP        ES
LEA        SI, STR
LEA        DI, STR + DISP
MOV        CX, LUNG
CLD
REP        MOVSB                ; scansione in avanti
...

```

Con l'esecuzione della prima istruzione MOVSB, il primo byte della stringa STR viene copiato nella locazione di indirizzo STR+5. Il precedente valore della stringa in STR+5 è modificato e dunque la copiatura è eseguita in maniera errata.

Una possibile soluzione consiste nell'eseguire la copiatura con una scansione all'indietro:



```

.MODEL    small
LUNG      EQU      100
.DATA
STR        DB      LUNG DUP (?)
           DB      5 DUP (?)
.CODE
...
LEA        SI, STR + SIZE STR - TYPE STR
LEA        DI, STR + SIZE STR - TYPE STR + 5
MOV        CX, LUNG
STD
REP        MOVSB                ; scansione all'indietro
...

```

In generale, il problema si pone solo quando la stringa sorgente e la stringa destinazione sono nello stesso segmento, si sovrappongono e la stringa sorgente comincia ad un indirizzo minore della stringa destinazione.

Riassumendo:

- se la stringa sorgente e la stringa destinazione non si sovrappongono, le istruzioni MOVSB e MOVSW non modificano la stringa sorgente;
- la copia con scansione in avanti non deve essere eseguita quando la stringa sorgente e la stringa destinazione sono nello stesso segmento, si sovrappongono e l'offset della stringa sorgente è inferiore all'offset di quella destinazione.

Esercizio: Inizializzazione di una stringa.

Si vuole realizzare un frammento di programma che esegua l'inizializzazione di una stringa replicando più volte una stessa sequenza di caratteri. Ad esempio il seguente programma in linguaggio C replica 20 volte la stringa "Ciao " nel vettore `vett`:

```
main()
{
    char vett[101], i;
    ...
    vett[0]='\0';
    for (i=0 ; i<20 ; i++)
        strcat(vett,"Ciao ");
    ...
}
```

La soluzione proposta in linguaggio Assembler è la seguente:

```
L1      EQU      95
        .MODEL   small
        .DATA
STRING  DB        "Ciao "
L2      EQU      $-STRING
        DB        L1 DUP(?)
        .CODE
        ...
        PUSH     DS
        POP      ES
        LEA      SI, STRING          ; stringa sorgente
        LEA      DI, STRING + L2    ; stringa destinazione
        MOV      CX, L1              ; dimensione della stringa
                                           ; destinazione
        CLD                          ; scansione in avanti
        REP      MOVSB               ; ciclo di trasferimento di byte
        ...
```

Per risolvere questo problema è stata utilizzata la seguente tecnica: la stringa sorgente è inizialmente caricata con il contenuto della sequenza da replicare; il registro `DI` è inizializzato con l'indirizzo successivo all'ultimo elemento della sequenza. Eseguendo la copia *in avanti* dalla stringa sorgente alla stringa destinazione si ha una replica del contenuto della sequenza.

12.3. Confronto tra stringhe

Le istruzioni **CMPSB** (*CoMPare Byte String*) e **CMPSW** (*CoMPare Word String*) permettono di confrontare due stringhe tra loro. L'istruzione **CMPSB** si usa per stringhe di byte, mentre la **CMPSW** per stringhe di word.



CMPSB
CMPSW

I registri `SI` e `DI` contengono, rispettivamente, l'indirizzo di offset della prima e della seconda stringa. Ad ogni esecuzione di una delle due istruzioni viene effettuato il confronto tra le locazioni di memoria aventi indirizzo `DS:SI` ed `ES:DI`.

L'istruzione di confronto viene tipicamente usata congiuntamente alle istruzioni **REPE** o **REPNE**, in particolare si usa **REPE** per verificare se le due stringhe sono uguali e si usa **REPNE** se sono diverse. Il ciclo di confronto termina nel caso in cui la stringa sia terminata (il registro `CX` ha valore

nullo), oppure qualora sia stata trovata una disuguaglianza (con REPE) od una uguaglianza (con REPNE).

Esempi

Il seguente frammento di codice esegue un confronto tra le due stringhe STR1 ed STR2; il confronto termina non appena viene riscontrata una differenza:

```

LUNG      .MODEL    small
          EQU       100
          .DATA
STR1      DB        LUNG DUP (?)
STR2      DB        LUNG DUP (?)
          .CODE
          ...
          PUSH      DS
          POP       ES
          LEA       SI, STR1
          LEA       DI, STR2
          MOV       CX, LUNG
          CLD
          REPE      CMPSB
          ...

```

Il seguente frammento di codice mostra un ciclo di confronto tra le due stringhe STR1 ed STR2; il confronto termina non appena viene riscontrata una uguaglianza tra due elementi:

```

LUNG      .MODEL    small
          EQU       100
          .DATA
STR1      DB        LUNG DUP (?)
STR2      DB        LUNG DUP (?)
          .CODE
          ...
          PUSH      DS
          POP       ES
          LEA       SI, STR1
          LEA       DI, STR2
          MOV       CX, LUNG
          CLD
          REPNE     CMPSB
          ...

```

A confronto terminato, bisogna essere in grado di distinguere tra le due condizioni di terminazione: poiché ogni esecuzione dell'istruzione di confronto aggiorna i flag, analizzando lo stato del flag ZF è possibile determinare se la condizione di uguaglianza è stata verificata oppure no. In particolare si è soliti ricorrere ad un'istruzione di salto condizionato: JE per saltare se il confronto ha dato esito positivo, JNE nel caso contrario.

Esempio

Il frammento di codice seguente esegue un confronto tra due stringhe; se esse sono uguali, il contenuto del registro AX è azzerato, altrimenti in AX viene caricato il valore 1.

```

LUNG      EQU      100
          .DATA
STR1      DB        LUNG DUP (?)
STR2      DB        LUNG DUP (?)
          .CODE
          ...
          PUSH      DS
          POP        ES
          LEA        SI, STR1
          LEA        DI, STR2
          MOV        CX, LUNG
          CLD
          REPE       CMPSB
          JE         lab1          ; gli ultimi elementi confrontati
                                   ; sono uguali ? Sì: va a lab1
                                   ; No: AX = 1
          MOV        AX, 1
          JMP        lab2
lab1:     XOR        AX, AX          ; AX = 0
lab2:     ...

```

All'uscita del ciclo è normalmente utile conoscere quale parola ha causato la terminazione. Tale informazione è contenuta nel registro di indice SI. Poiché esso viene aggiornato alla fine di ciascun confronto, all'uscita del ciclo esso contiene l'offset della parola successiva a quella che ha causato la terminazione. È necessario dunque modificare opportunamente il registro SI per accedere alla parola che ha causato la terminazione. In particolare, nel caso di scansione in avanti, occorre decrementarne il contenuto di un'unità per stringhe di byte e di due unità per stringhe di word; nel caso di scansione all'indietro il contenuto di SI deve invece essere incrementato di un'unità per stringhe di byte e di due unità per stringhe di word.

Esempio

Il seguente frammento di codice esegue il confronto tra due stringhe; se le due stringhe sono uguali il registro AL viene azzerato, altrimenti in AL viene copiato il valore dell'elemento della seconda stringa che ha causato la terminazione del ciclo.

```

          .MODEL     small
LUNG      EQU      100
          .DATA
STR1      DB        LUNG DUP (?)
STR2      DB        LUNG DUP (?)
          .CODE
          ...
          PUSH      DS
          POP        ES
          LEA        SI, STR1
          LEA        DI, STR2
          MOV        CX, LUNG
          CLD
          REPE       CMPSB
          JE         lab1          ; stringhe uguali ?
          DEC        DI            ; No: decrementa DI
          MOV        AL, [DI]      ; copia in AL l'elemento di STR2
                                   ; che ha causato la fine del ciclo
          JMP        lab2
lab1:     XOR        AL, AL        ; Sì: azzerava il registro AL
lab2:     ...

```

12.4. Scansione di una stringa

Le istruzioni **SCASB** (*SCAn Byte String*) e **SCASW** (*SCAn Word String*) permettono di scandire una stringa per ricercare un valore specifico. L'istruzione SCASB si usa per stringhe di byte, la SCASW per stringhe di word. Il loro formato è il seguente:



SCASB
SCASW

Queste istruzioni sono simili alle istruzioni di confronto, con la differenza che esse elaborano solo una stringa e confrontano ciascun elemento con un determinato valore. L'istruzione SCASB usa i registri AL e DI, mentre l'istruzione SCASW usa i registri AX e DI. I registri AL ed AX contengono il valore da confrontare, mentre DI contiene l'offset della stringa da scandire. Ad ogni esecuzione dell'istruzione SCASB (o SCASW) viene effettuato il confronto tra il contenuto della locazione di memoria avente indirizzo ES:DI ed il contenuto del registro *accumulatore*.

Anche le istruzioni SCASB e SCASW utilizzano le istruzioni REPE e REPNE per generare il ciclo: REPE per cercare un valore diverso dal contenuto dei registri AL ed AX, REPNE per cercare un valore coincidente. Il ciclo di scansione termina al verificarsi di una delle seguenti condizioni:

1. la stringa è terminata (il registro CX ha valore nullo)
2. è stata trovata una disuguaglianza (nel caso dell'istruzione REPE) o un'uguaglianza (nel caso dell'istruzione REPNE) tra il contenuto del registro *accumulatore* e la stringa indirizzata da DI.

Esercizio: Ricerca del primo carattere alfabetico in una stringa di caratteri.

Si realizzi un frammento di programma in grado di scandire una stringa di caratteri, visualizzandone il primo carattere diverso dal carattere *spazio*. Una possibile soluzione in C è la seguente:

```
#include <stdio.h>
main()
{
    int i;
    char string[25];
    ...
    strcpy(string, "      Fatti non foste a viver come bruti ...");
    for (i=0 ; i<25 ; i++)
        if (string[i] != ' ')
            { printf("%c",string[i]);
              break;
            }
    ...
}
```


Una possibile soluzione in linguaggio Assembler è la seguente:

```

                .MODEL    small
                .DATA
STRING          DB      "      Fatti non foste a viver come bruti ..."
LUNG            EQU     $$-STRING
ST_ADD         DD      STRING
                .CODE
...
                MOV      AL, " "      ; copia in AL del carattere da confrontare
                LES      DI, ST_ADD    ; copia in DI dell'offset di STRING
                                ; copia in ES dell'indirizzo di segmento
                MOV      CX, LUNG      ; copia in CX della lunghezza di STRING
                CLD                    ; scansione in avanti
                REPE     SCASB         ; finché elemento di STRING = " "
                JE       esci          ; elemento di STRING = " " ?, Si: va a esci
                DEC      DI           ; No: decrementa registro indice
                MOV      DL, [DI]      ; copia in DL il valore diverso
                MOV      AH, 02H       ; visualizza su video
                INT      21H
esci:           ...

```

Esercizio: Ricerca del codice terminatore di un vettore di interi.

Si realizzi un frammento di programma che scandisca un vettore di interi alla ricerca del codice terminatore -1; se tale codice manca deve essere visualizzato un messaggio di errore.

La soluzione proposta in linguaggio C è la seguente:

```

#include <stdio.h>
main()
{
    char found = 0;
    int i, string[100];
    ...
    for (i=0 ; i<100 ; i++)
        if (string[i] == -1)
        {
            found = 1;
            break;
        }
    if (!found)
        printf("Manca il codice terminatore\n");
    ...
}

```

La soluzione proposta in linguaggio Assembler è la seguente:

```

LUNG      EQU      100
EOS        EQU      -1
.MODEL    small
.DATA
STRING    DW        LUNG DUP (?)
ST_ADD    DD        STRING
ER_MSG    DB        "Manca il codice terminatore",0DH,0AH,"$"
.CODE
...
MOV        AX, EOS      ; copia in AX il valore -1
LES        DI, ST_ADD    ; copia in DI dell'offset di STRING
                        ; copia in ES dell'indirizzo di segmento
MOV        CX, LUNG      ; copia in CX della lunghezza di STRING
CLD                          ; scansione in avanti
REPNE     SCASW          ; finché elemento di STRING ≠ -1
JE         esci          ; elemento di STRING = -1 ?
LEA        DX, ER_MSG    ; No: stampa messaggio di errore
MOV        AH, 09H
INT        21H
esci:      ...           ; Si: esci

```

12.5. Inizializzazione di una stringa

Le istruzioni **STOSB** (*STORe Byte String*) e **STOSW** (*STORe Word String*) permettono di inizializzare tutti gli elementi di una stringa ad un determinato valore. Il loro formato è il seguente:



STOSB
STOSW

I registri AL ed AX contengono il valore con cui deve essere inizializzata la stringa (AL per l'istruzione SCASB e AX per l'istruzione SCASW); il registro DI contiene l'offset della stringa destinazione. Le istruzioni in esame copiano il valore contenuto nei registri AL (od AX) nella locazione di memoria avente indirizzo ES:DI. Ogni esecuzione aggiorna il valore del registro DI coerentemente con il valore del flag DF:

- se il flag DF vale 0, il registro DI è incrementato di un'unità (istruzione SCASB) o di due unità (SCASW);
- se il flag DF vale 1, il registro DI è decrementato di una unità (istruzione SCASB) o di due unità (SCASW).

Esercizio: Inizializzazione di una stringa di caratteri.

Si realizzi un frammento di programma che scriva il carattere *spazio* in tutti gli elementi di una stringa. La soluzione proposta in linguaggio C è la seguente:

```
main()
{
    int i;
    char str[26];
    ...
    for (i=0 ; i<25 ; i++)
        str[i] = ' ';
    str[25] = '\\0';
    ...
}
```

La soluzione proposta in linguaggio Assembler è la seguente:

```
LUNG      EQU      25
           .MODEL   small
           .DATA
STR        DB       LUNG DUP(?)
ST_ADD     DD       STR
           .CODE
           ...
           MOV      AL, " "          ; copia in AL del carattere spazio
           LES      DI, ST_ADD       ; copia in DI dell'offset di STR
                                           ; copia in ES dell'indirizzo di segmento
           MOV      CX, LUNG         ; copia in CX della dimensione di STR
           CLD                      ; scansione in avanti
           REP      STOSB           ; ciclo di scansione di STR
           ...
```

12.6. Elaborazione di una stringa

Le istruzioni **LODSB** (*LOaD Byte String*) e **LODSW** (*LOaD Word String*) permettono di copiare un elemento di una stringa rispettivamente nei registri AL e AX. Il loro formato è il seguente:



LODSB
LODSW

Il registro SI contiene l'offset della stringa sorgente. L'effetto di queste istruzioni è quello di copiare il contenuto della locazione di memoria avente indirizzo DS:SI o in AL (istruzione LODSB) o in AX (istruzione LODSW). Ogni esecuzione aggiorna il valore del registro SI, coerentemente con il valore del flag DF:

- se il flag DF vale 0, il registro SI è incrementato di una unità (istruzione LODSB) e di due unità (LODSW);
- se il flag DF vale 1, il registro SI è decrementato di una unità (istruzione LODSB) e di due unità (LODSW).

Queste istruzioni sono utili, in coppia con le istruzioni STOSB e STOSW per eseguire una stessa operazione su tutti gli elementi di una stringa.

Lo schema di funzionamento è il seguente:

1. si copia l'offset della stringa sorgente in SI;
2. si copia l'offset della stringa destinazione in DI;

3. si copia la lunghezza della stringa in CX;
4. si aggiorna il flag DF;
5. per ogni parola della stringa sorgente:
 - si copia la parola della stringa sorgente nel registro AL (o AX);
 - si elabora il contenuto del registro;
 - si copia il contenuto del registro AL (o AX) nella stringa destinazione.

Esercizio: *Conteggio del numero di spazi in una stringa di caratteri.*

Si realizzi un frammento di programma che conti il numero di spazi inclusi in una stringa di caratteri. La soluzione proposta in linguaggio C è la seguente:

```
main()
{
    char sorg[100];
    int i, count = 0;
    ...
    for (i=0 ; i<100 ; i++)
        if (sorg[i] == ' ') count++;
    ...
}
```

La soluzione proposta in linguaggio Assembler è la seguente:

```
LUNG      EQU      100
           .MODEL   small
           .DATA
SORG       DB       LUNG DUP (?)
           .CODE
           ...
           LEA      SI, SORG      ; inizializzazione di SI
           MOV      CX, LUNG      ; inizializzazione di CX
           XOR      BX, BX        ; BX: contatore di spazi inizializzato a 0
           CLD                ; scansione in avanti
ciclo:     LODSB                ; copia in AL l'elemento di SORG
           CMP      AL, ' '       ; AL = ' ' ?
           JNE      next          ; No: va a next
           INC      BX            ; Sì: incrementa BX
next:      LOOP     ciclo         ; CX = 0 ?, No: va a ciclo
           ...                  ; Sì: fine
```

Esercizio: *Copia di un vettore di interi ed azzeramento dei termini negativi.*

Si vuole realizzare un frammento di programma che trasferisca un vettore di interi da una zona di memoria ad un'altra, con la condizione che i termini negativi siano trasformati in termini di valore nullo. La soluzione proposta in linguaggio C è la seguente:

```
main()
{
    int sorg[100], dest[100], i;
    ...
    for (i=0 ; i<100 ; i++)
        if (sorg[i] < 0) dest[i] = 0;
        else dest[i] = sorg[i];
    ...
}
```

La soluzione proposta in linguaggio Assembler è la seguente:

```

LUNG      EQU      100
          .MODEL    small
          .DATA
SORG      DW        LUNG DUP (?)
DEST      DW        LUNG DUP (?)
ST_ADD    DD        DEST
          .CODE
          ...
          LEA       SI, SORG      ; inizializzazione di SI
          LES       DI, ST_ADD    ; inizializzazione di DI ed ES
          MOV       CX, LUNG      ; inizializzazione di CX
          CLD        ; scansione in avanti
ciclo:    LODSW      ; copia in AX l'elemento di SORG
          CMP       AX, 0         ; AX < 0 ?
          JNL       lab          ; No: va a lab
          XOR       AX, AX        ; Sì: azzerà il registro AX
lab:      STOSW      ; copia in DEST il contenuto di AX
          LOOP      ciclo        ; CX = 0 ?, No: va a ciclo
          ...                  ; Sì: fine

```

Esercizio: *Conversione da caratteri minuscoli a caratteri maiuscoli.*

Si realizzi un frammento di programma che trasferisca una stringa di caratteri da un'area di memoria sorgente ad una destinazione, eseguendo la conversione dei caratteri alfabetici da minuscoli a maiuscoli. La soluzione proposta in linguaggio C è la seguente:

```

main()
{
    char sorg[100], dest[100];
    int i;
    ...
    for (i=0 ; i<100 ; i++)
        if ((sorg[i] <= 'z') && (sorg[i] >= 'a'))
            dest[i] = sorg[i] + 'A' - 'a';
        else
            dest[i] = sorg[i];
    ...
}

```

La soluzione proposta in linguaggio Assembler è la seguente:

```

LUNG      EQU      100
          .MODEL    small
          .DATA
SORG      DB        LUNG DUP (?)
DEST      DB        LUNG DUP (?)
ST_ADD    DD        DEST
          .CODE
          ...
          LEA       SI, SORG      ; inizializzazione di SI
          LES       DI, ST_ADD    ; inizializzazione di DI e ES
          MOV       CX, LUNG      ; inizializzazione di CX
          CLD                     ; scansione in avanti
ciclo:    LODSB                     ; copia in AL l'elemento di SORG
          CMP       AL, 'z'       ; AL <= 'z' ?
          JNLE      copia         ; No: va a copia
          CMP       AL, 'a'       ; Sì, AL >= 'a' ?
          JNGE      copia         ; No: va a copia
          ADD       AL, 'A'-'a'   ; conversione da minuscolo a maiuscolo
copia:    STOSB                     ; copia in DEST il contenuto di AL
          LOOP      ciclo         ; CX = 0 ?, No: va a ciclo
          ...                     ; Sì: fine

```

Esercizio: *Eliminazione degli spazi in una stringa.*

Si realizzi un frammento di programma che copi una stringa di caratteri da una zona di memoria sorgente ad una destinazione, senza copiare gli spazi. La soluzione proposta in linguaggio C è la seguente:

```

main()
{
    char sorg[100], dest[100];
    int i;
    ...
    for (i=0,j=0 ; i<100 ; i++)
        if (sorg[i] != ' ')
            dest[j++] = sorg[i];
    ...
}

```

La soluzione proposta in linguaggio Assembler è la seguente:

```

LUNG      EQU      100
          .MODEL    small
          .DATA
SORG      DB        LUNG DUP (?)
DEST      DB        LUNG DUP (?)
          .CODE
          ...
          PUSH      DS
          POP       ES            ; caricamento del registro ES
          LEA       SI, SORG      ; inizializzazione di SI
          LEA       DI, DEST      ; inizializzazione di DI
          MOV       CX, LUNG      ; inizializzazione di CX
          CLD                     ; scansione in avanti
ciclo:    LODSB                     ; copia in AL l'elemento di SORG
          CMP       AL, ' '       ; AL ≠ ' ' ?
          JE        scans         ; No: va a scans
          STOSB                     ; Sì: copia in DEST il contenuto di AL
scans:    LOOP      ciclo         ; CX = 0 ?, No: va a ciclo
          ...                     ; Sì: fine

```

12.7. Istruzioni con operandi

Ogni istruzione per la manipolazione di stringhe ha un'equivalente istruzione, con suffisso **S** (al posto di SB o SW), caratterizzata dall'avere uno o due operandi espliciti:

- **MOVS** (*MOVe String*);
- **CMPS** (*CoMPare String*);
- **SCAS** (*SCAn String*);
- **LODS** (*LOaD String*);
- **STOS** (*STOre String*).

Il loro formato è il seguente:



```
MOVS    dest, sorg
CMPS    dest, sorg
SCAS    dest
LODS    sorg
STOS    dest
```

Le istruzioni di manipolazione con operandi si comportano in modo analogo rispetto alle equivalenti istruzioni senza operando, e vengono tipicamente impiegate per:

- selezionare in modo automatico il tipo degli operandi (byte o word): l'assemblatore è in grado di capire quale istruzione macchina codificare (ad es. MOVSB oppure MOVSW) in base alla definizione del dato;
- effettuare il *segment override* esplicito sull'indirizzo della stringa sorgente.

Quest'ultimo caso si verifica quando la stringa sorgente si trova nel segmento di dato *extra*. In questo caso è necessario effettuare il *segment override* nel momento del caricamento di SI e nel momento di esecuzione dell'istruzione per la manipolazione della stringa.

Esempio

Il frammento di codice seguente esegue la copiatura della stringa STR1 (memorizzata nel segmento di dato *extra*) nella stringa STR2 (memorizzata nello stesso segmento).

```
.MODEL    compact                ; un segmento di codice
                                ; più segmenti di dato

LUNG      EQU      100
          .FARDATA  seg1
          ...
          .FARDATA  seg2
STR1      DB        LUNG DUP (?)
STR2      DB        LUNG DUP (?)
          .CODE
          ASSUME    DS:seg1, ES:seg2
          ...
          LEA       SI, ES:STR1    ; inizializzazione di SI
          LEA       DI, STR2      ; inizializzazione di DI
          MOV       CX, LUNG      ; caricamento di CX
          CLD        ; scansione in avanti
REP       MOVSB     STR2, ES:STR1  ; trasferimento da STR1 a STR2
          ...
```

13. Le procedure

In questo capitolo viene descritto il modo in cui l'Assembler x86 permette di gestire le procedure; in particolare verranno presentate sia le operazioni che devono essere effettuate per definire ed utilizzare le procedure in un programma, sia le principali tecniche utilizzate per il passaggio di parametri tra il programma chiamante e la procedura chiamata.

13.1. Le procedure

Una *procedura*, detta anche *subroutine* o *sottoprogramma*, è una parte di programma costituita da un gruppo di istruzioni che eseguono un compito specifico; ogni procedura, memorizzata in memoria una volta sola, può essere eseguita un numero qualsiasi di volte.

L'utilizzo delle procedure permette di risparmiare spazio in memoria e di rendere più modulare lo sviluppo di programmi. Il principale svantaggio risiede nel tempo di elaborazione necessario per le operazioni di chiamata e di ritorno tra la procedura chiamante e quella chiamata.

Le procedure possono accettare valori in ingresso e fornire valori di ritorno in uscita: tali valori, chiamati *parametri*, costituiscono di fatto l'interfaccia tra la procedura ed il programma; si può infatti pensare ad una procedura come ad una *scatola nera* che, una volta attivata da parte del programma chiamante da cui riceve eventuali parametri, svolge il proprio compito e restituisce al chiamante eventuali parametri di ritorno.

È buona regola di programmazione organizzare il proprio programma in una serie di procedure semplici e chiare: ciò permette di dividere programmi di grande dimensione in sottoprogrammi più facilmente manutenibili e, conseguentemente, di semplificare le operazioni di analisi, verifica e correzione degli errori.

13.2. Definizione di una procedura

In Assembler x86 una procedura è definita tramite le direttive **PROC** ed **ENDP**, che permettono di dichiarare all'assemblatore l'inizio e la fine di una procedura e che dunque non generano alcuna istruzione macchina. Il formato della direttiva **PROC** è il seguente:

*etichetta PROC tipo*

Il campo *etichetta* corrisponde al *nome* della procedura, mentre il campo *tipo* definisce il *tipo* della procedura, che può essere NEAR o FAR. Una procedura di tipo NEAR è richiamabile solo all'interno dello stesso segmento di codice, mentre una procedura di tipo FAR può essere richiamata da procedure appartenenti a segmenti di codice diversi. Se il tipo della procedura non è specificato, l'assemblatore assume che esso sia coerente con il modello di memoria specificato.

La direttiva ENDP indica la fine di una procedura; il suo formato è il seguente:

*etichetta ENDP*

Il campo *etichetta* deve essere lo stesso usato nella corrispondente direttiva PROC.

Esempio

Il seguente frammento di codice definisce una procedura di tipo NEAR e di nome L_CAR.

```
L_CAR PROC NEAR
...      ; istruzioni che costituiscono
...      ; il corpo della procedura
L_CAR ENDP
```

13.3. Chiamata di una procedura

L'istruzione **CALL** (*CALL a procedure*) trasferisce il controllo del flusso del programma ad una specifica procedura; il suo formato è il seguente:

*CALL destinazione*

L'operando *destinazione* specifica l'indirizzo di inizio della procedura chiamata.

L'istruzione CALL provvede a salvare nello *stack* l'*indirizzo di ritorno* ed a trasferire il controllo all'operando *destinazione*, senza modificare il valore di alcun flag.

L'indirizzo di ritorno corrisponde all'indirizzo dell'istruzione successiva a quella di CALL; ad essa la procedura ritorna al termine della sua esecuzione.

Se la procedura chiamata è di tipo NEAR, l'istruzione CALL carica nello *stack* solo il contenuto dell'*Instruction Pointer* (IP), cioè l'indirizzo di *offset* dell'istruzione successiva.

Se la procedura chiamata è di tipo FAR, l'istruzione CALL carica nello *stack* prima il contenuto del registro di segmento di codice CS e poi il contenuto del registro IP.

L'operando *destinazione* può essere un indirizzo *diretto* o *indiretto*. Nel primo caso l'operando *operando* è costituito dal nome stesso della procedura.

Esempio

La seguente istruzione esegue la chiamata della procedura di nome L_CAR:

```
CALL L_CAR
```

Nel caso di indirizzamento indiretto l'indirizzo della procedura chiamata viene specificato tramite un registro di base o di indice. È compito del programmatore fare in modo che l'assemblatore conosca il tipo della procedura, utilizzando l'operatore PTR, ed in particolare:

- WORD PTR se la procedura è di tipo NEAR;
- DWORD PTR se la procedura è di tipo FAR.

Esempi

Nel frammento di codice seguente, la variabile ADDR1 memorizza l'indirizzo di offset della procedura DISPLAY1 di tipo NEAR. Tale procedura è chiamata tramite un indirizzamento indiretto.

```

ADDR1    .DATA
         DW      DISPLAY1
         .CODE
         ...
         LEA     BX, ADDR1
         CALL    WORD PTR [BX]
         ...

```

Nel frammento di codice seguente, la variabile ADDR2 memorizza l'indirizzo intero della procedura DISPLAY2 di tipo FAR. Tale procedura è chiamata tramite un indirizzamento indiretto.

```

ADDR2    .DATA
         DD      DISPLAY2
         .CODE
         ...
         LEA     BX, ADDR2
         CALL    DWORD PTR [BX]
         ...

```

Il processore distingue il tipo di procedura (NEAR o FAR) in base all'istruzione macchina che codifica l'istruzione CALL: esistono due diverse istruzioni macchina, una per procedure di tipo NEAR ed una per procedure di tipo FAR. È dunque compito dell'assemblatore generare l'opportuna istruzione macchina in base alla definizione della procedura.

13.4. Ritorno da una procedura

L'istruzione **RET** (*RETurn from procedure*) permette di restituire il controllo alla procedura chiamante, una volta che la procedura chiamata ha terminato l'esecuzione. Il suo formato è il seguente:



RET {*pop-value*}

L'operando *pop-value* è un valore immediato opzionale; esso permette di eseguire l'operazione di liberazione dello *stack* al momento del ritorno alla procedura chiamante; il processore esegue l'operazione di *pop* dallo *stack* di un numero di byte pari a *pop-value*. Il valore di *default* dell'operando *pop-value* è 0. L'istruzione RET non modifica il valore di alcun flag.

L'istruzione RET assume che l'indirizzo di ritorno si trovi in cima allo *stack*. Questo implica che, nel caso in cui la procedura chiamata abbia modificato lo *stack* memorizzandovi dati, questi devono essere rimossi prima dell'esecuzione dell'istruzione RET.

L'istruzione RET esegue le seguenti operazioni:

1. *pop* dallo *stack* dell'indirizzo di ritorno;

2. estrazione dallo *stack* di un numero di byte pari a *pop-value*;
3. salto all'indirizzo di ritorno.

Se la procedura è di tipo NEAR il processore preleva dallo *stack* una word contenente l'offset dell'indirizzo di ritorno, mentre nel caso di procedura di tipo FAR dallo *stack* vengono prelevate due word equivalenti all'intero indirizzo di ritorno CS:IP.

Esistono due diverse istruzioni macchina che permettono di ritornare alla procedura principale: una per un ritorno di tipo FAR ed una per un ritorno di tipo NEAR. È compito dell'assemblatore generare l'opportuna istruzione in linguaggio macchina, in base al tipo di procedura.

13.5. Salvataggio dei registri

Per il funzionamento corretto del programma è estremamente importante che l'effetto della procedura non sia distruttivo sul resto del programma. Questa implica, tra l'altro, che il contenuto dei registri prima della chiamata della procedura sia lo stesso al momento del ritorno. Per ottenere questo, la prima operazione da eseguire all'interno di una procedura è il salvataggio nello *stack* di tutti i registri modificati all'interno. Tale operazione è totalmente a carico del programmatore.

Per ripristinare il contenuto dei registri occorre eseguire l'operazione di *pop* dei registri dallo *stack* prima dell'esecuzione dell'istruzione RET.

È bene ricordare che lo *stack* è una coda di tipo *LIFO* e dunque l'ordine delle istruzioni POP deve essere l'inverso dell'ordine delle istruzioni PUSH.

Esempio

Il frammento di procedura seguente mostra un esempio di salvataggio e di ripristino dei registri AX e BX:

```

name    PROC    NEAR
        PUSH    AX          ; salvataggio dei registri nello stack
        PUSH    BX
        ...                ; istruzioni
        POP     BX          ; ripristino dei registri dallo stack
        POP     AX
        RET      ; ritorno alla procedura chiamante
name    ENDP

```

A partire dal processore 80186 è possibile utilizzare le istruzioni PUSHAD e POPAD per salvare nello *stack* il contenuto di tutti i registri.

Esempio

Il frammento di procedura seguente mostra un esempio di salvataggio e di ripristino di tutti i registri per un processore 80386:

```

name    .386
        ...
        PROC    NEAR
        PUSHAD  ; salvataggio di tutti i registri
        ...    ; istruzioni
        POPAD   ; ripristino di tutti i registri
        RET     ; ritorno alla procedura chiamante
name    ENDP

```

13.6. Punto di ingresso e di uscita di una procedura

L'indirizzo da cui ha inizio l'esecuzione di una procedura è detto *punto di ingresso*; l'indirizzo in cui una procedura termina è detto *punto di ritorno* ed è caratterizzato dall'istruzione RET. Ogni pro-

cedura deve avere almeno un punto di ritorno.

Benché una procedura possa avere più punti di ingresso e di ritorno, al fine di facilitare il *debugging* è buona norma scrivere procedure che abbiano un solo punto di ingresso ed un solo punto di ritorno.

È possibile che si abbiano diversi punti in cui logicamente termina l'esecuzione di una procedura; per mantenere un unico punto di ritorno è conveniente eseguire un salto da ogni punto logico di terminazione all'indirizzo corrispondente all'unica istruzione `RET` della procedura stessa.

Esempio

Il frammento di codice seguente mostra un esempio di procedura avente due punti logici di ritorno, ma un'unica istruzione `RET`:

```

LUNG      EQU      100
          .DATA
VETT      DB        LUNG DUP (?)
          .CODE
...
SOMM_VETT PROC
...
          MOV      CX, LUNG
          XOR      AX, AX      ; inizializzazione del contatore
          MOV      SI, 0
ciclo:    CMP      VETT[SI], 0 ; VETT[SI] = 0 ?
          JZ       ritorno     ; Sì => ritorna alla procedura chiamante
          ADD      AX, VETT[SI] ; No => contatore = contatore + VETT[SI]
          INC      SI
          LOOP     ciclo
          ...
ritorno:  RET
SOMM_VETT ENDP
          ...

```

Si sconsiglia inoltre vivamente l'attivazione della procedura con istruzioni diverse da `CALL`, poiché diventa estremamente critico gestire correttamente il contesto relativo alla procedura (indirizzo di ritorno e salvataggio dei registri).

Esempio

Il frammento di codice seguente mostra un esempio di procedura avente due punti di ingresso: il primo ottenuto tramite una istruzione `CALL`, il secondo ottenuto con l'esecuzione di un'istruzione di salto ad un'istruzione interna alla procedura.

```

☹          .CODE
          ...
          CALL     SOMM_VETT
          ...
          JMP      lab1
          ...
SOMM_VETT PROC
          ...
lab1:     ...
ritorno:  RET
SOMM_VETT ENDP
          ...

```

13.7. Passaggio di parametri

Esistono diverse tecniche per effettuare il passaggio dei parametri tra procedura chiamante e

chiamata, classificabili in base al *metodo* o in base al *tramite*.

I possibili *metodi* con cui i parametri sono trasferiti sono:

1. una copia del valore del parametro (passaggio *by value*);
2. l'indirizzo del parametro (passaggio *by reference*).

In un passaggio *by value*, la procedura chiamante passa a quella chiamata una *copia* del parametro. Ogni possibile modifica del parametro all'interno della procedura modifica esclusivamente tale copia. La procedura chiamante non “vede” le modifiche effettuate sul parametro dalla procedura chiamata.

In un passaggio *by reference* la procedura chiamante passa alla procedura chiamata l'indirizzo del parametro: la procedura chiamata e quella chiamante operano direttamente sulla stessa variabile.

I possibili *tramiti* con cui avviene il trasferimento dei dati sono:

1. le variabili globali;
2. i registri;
3. lo *stack*.

Nel seguito viene presentata una rassegna delle diverse tecniche, analizzando la validità di ciascuna soluzione.

Esempio

La seguente procedura, scritta in linguaggio C, esegue la somma degli elementi di un vettore di interi di nome `vett` e di dimensione `count`.

```
int som_vett(int *vett, int count)
{
    int i, somma = 0;
    for (i=0 ; i < count ; i++)
        somma += vett[i];
    return(somma);
}
```

La procedura dispone di due parametri di ingresso (l'indirizzo iniziale del vettore di interi e la sua lunghezza) e restituisce un parametro di ritorno (il valore della somma degli elementi del vettore). Il parametro `vett` è un esempio di parametro passato *by reference*; `count` è un esempio di parametro passato *by value*.

13.7.1. Uso di variabili globali

Il modo più semplice per passare parametri alle procedure consiste nell'utilizzare variabili globali, accessibili sia dalla procedura chiamante sia da quella chiamata.

Questo metodo, seppure estremamente semplice, è sconsigliabile in quanto in contrasto con la logica stessa dell'uso delle procedure: le procedure che utilizzano variabili globali come parametri sono poco riutilizzabili in quanto non in grado di operare su dati posti altrove in memoria.

Esempio

Si mostra ora un esempio di realizzazione in linguaggio Assembler di un frammento di programma che esegue una chiamata ad una procedura equivalente a quella mostrata precedentemente in linguaggio C, facendo uso delle variabili globali per il passaggio dei parametri:

```

LUNG      EQU      100
           .MODEL   small
           .DATA
VETT       DW      LUNG DUP (?)
COUNT    DW      LENGTH VETT
SOMMA     DW      ?
           .CODE
           ...
           CALL    SOM_VETT
           ...
SOM_VETT   PROC                ; procedura di somma vettore
           PUSH    SI          ; salvataggio dei registri nello stack
           PUSH    AX
           PUSH    CX
           XOR     SI, SI      ; azzeramento del registro SI
           XOR     AX, AX      ; azzeramento del registro AX
           MOV     CX, COUNT   ; CX = dimensione della stringa
ciclo:     ADD     AX, VETT[SI] ; AX = AX + VETT[SI]
           ADD     SI, 2       ; aggiornamento dell'indice
           LOOP    ciclo      ; scansione del vettore
           MOV     SOMMA, AX   ; copia in SOMMA il risultato
           POP     CX          ; ripristino dei registri dallo stack
           POP     AX
           POP     SI
           RET                ; ritorno alla procedura chiamante
SOM_VETT   ENDP              ; fine della procedura
           ...

```

13.7.2. Uso di registri

I parametri di ingresso ed uscita possono essere trasferiti utilizzando i registri *general purpose*. È un metodo semplice ed efficiente, ma utilizzabile solo quando i parametri sono in numero limitato.

Il parametro di ritorno di una procedura è molto frequentemente passato attraverso il registro accumulatore (AX o AL).

Esempio

Si mostra ora un esempio di realizzazione in linguaggio Assembler di un frammento di programma che esegue chiamate ad una procedura equivalente a quella mostrata precedentemente in linguaggio C, facendo uso dei registri per il passaggio dei parametri:

```

LUNG      EQU      100
           .MODEL   small
           .DATA
VETT      DW      LUNG DUP (?)
SOMMA     DW      ?
VET2      DW      2*LUNG DUP (?)
           .CODE
           ...
           MOV      AX, LENGTH VETT ; copia in AX la lunghezza di VETT
           LEA      BX, VETT        ; copia in BX dell'offset di VETT
           CALL     SOM_VETT        ; chiamata alla procedura
           MOV      SOMMA, AX       ; copia del risultato da AX a SOMMA
           ...
           MOV      AX, LENGTH VET2 ; copia in AX della lunghezza di VET2
           LEA      BX, VET2        ; copia in BX dell'offset di VET2
           CALL     SOM_VETT        ; chiamata alla procedura
           MOV      SOMMA, AX       ; copia del risultato da AX a SOMMA
           ...
SOM_VETT  PROC                                ; procedura di somma vettore
           PUSH     BX               ; salvataggio dei registri nello stack
           PUSH     CX
           MOV      CX, AX           ; CX = dimensione del vettore
           XOR      AX, AX           ; azzeramento del registro AX
ciclo:    ADD      AX, [BX]          ; AX = AX + [BX]
           ADD      BX, 2            ; aggiornamento dell'indice del vettore
           LOOP     ciclo           ; scansione del vettore
           POP      CX              ; ripristino dei registri dallo stack
           POP      BX
           RET                     ; ritorno alla procedura chiamante
SOM_VETT  ENDP                      ; fine procedura somma vettore
           ...

```

Si noti come sia stato possibile utilizzare la stessa procedura con dati diversi (il vettore VETT di dimensione LUNG ed il vettore VET2 di dimensione 2*LUNG). Il registro AX è stato utilizzato sia per passare il parametro di ingresso (lunghezza del vettore), sia il parametro di uscita (somma degli elementi del vettore).

13.7.3. Uso dello stack

Il metodo più utilizzato per il passaggio dei parametri si basa sullo *stack*. Tale metodo non ha limiti sul numero di parametri passati (a meno del limite fisico di allocazione dello stack) e permette un comodo riutilizzo delle procedure su dati diversi; inoltre non richiede un'allocazione statica di memoria per contenere i parametri, come nel caso delle variabili globali. Analizziamo separatamente le fasi necessarie per il passaggio corretto dei parametri.

Caricamento dei parametri nello stack

Prima dell'esecuzione dell'istruzione CALL, la procedura chiamante deve eseguire tante istruzioni di PUSH nello *stack* quanti sono i parametri da passare.

Esempio

Il frammento di codice seguente mostra l'operazione che deve eseguire la procedura chiamante:

```

...
PUSH    LUNG        ; caricamento della lunghezza di VETT
PUSH    OFFSET VETT ; caricamento dell'offset di VETT
CALL    SOM_VETT    ; chiamata alla procedura
...

```

Letture dei parametri di ingresso

La lettura dei parametri da parte della procedura chiamata è un'operazione delicata: l'istruzione `CALL` è eseguita dopo che i parametri sono caricati nello *stack*; l'indirizzo di ritorno è caricato nello *stack* dopo tutti i parametri e si trova in cima allo *stack* nel momento in cui la procedura inizia l'esecuzione. Ciò significa che la procedura chiamata non può eseguire l'operazione di *pop* dallo *stack* per prelevare i parametri senza perdere l'indirizzo di ritorno.

Una prima idea potrebbe essere quella di salvare in un registro l'indirizzo di ritorno e di caricarlo nello *stack* prima di restituire il controllo alla procedura chiamante. Questa tecnica funziona, ma è piuttosto laboriosa ed inefficiente.

Una soluzione più efficiente consiste nell'utilizzare il registro *Base Pointer* (BP) per fare accesso allo *stack*. Il registro BP permette di indirizzare dati presenti nello *stack* senza eseguire operazioni di *push* o *pop*, ossia senza cambiare il contenuto del registro *Stack Pointer* (SP). Questo è possibile eseguendo la copia del contenuto di SP in BP. Tale operazione deve essere fatta prima del salvataggio dei registri, poiché il salvataggio dei registri nello *stack* necessariamente modifica il contenuto di SP.

In Fig. 13.1 è mostrato lo stato dello *stack* ed il contenuto dei registri SS, SP e BP prima del salvataggio dei registri.

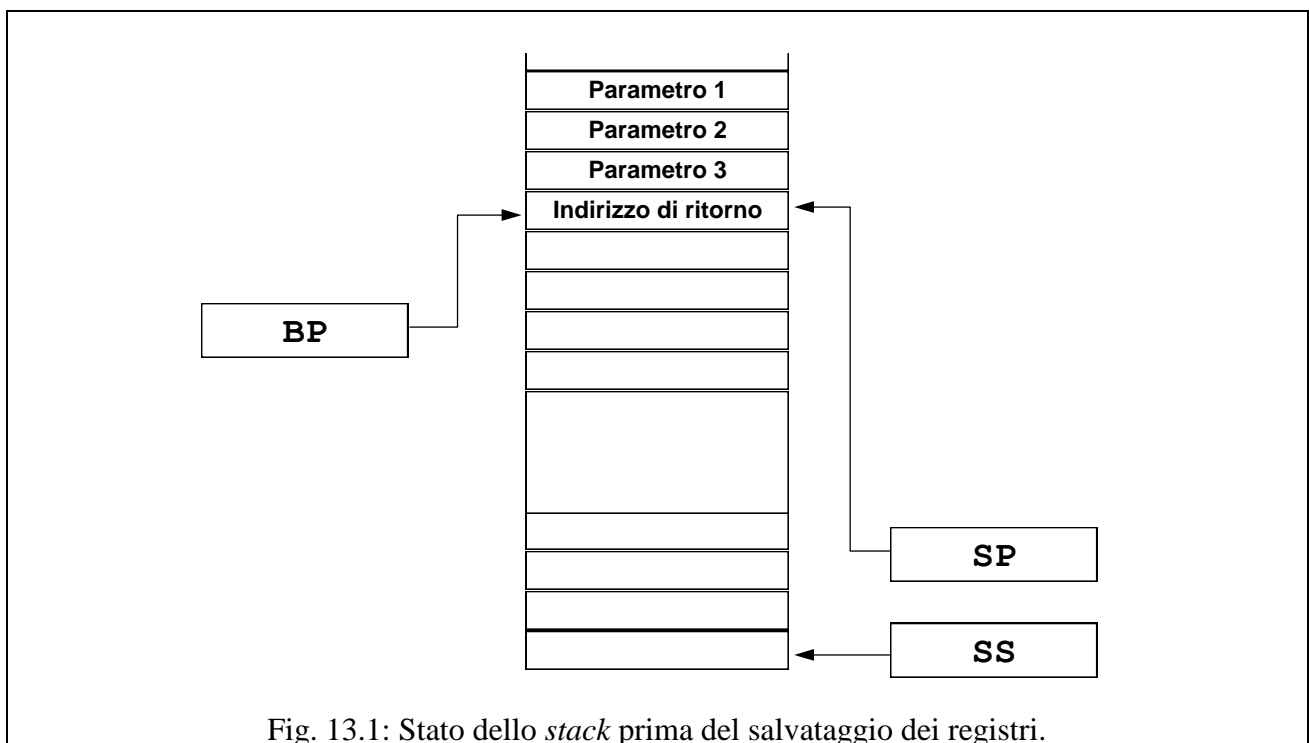
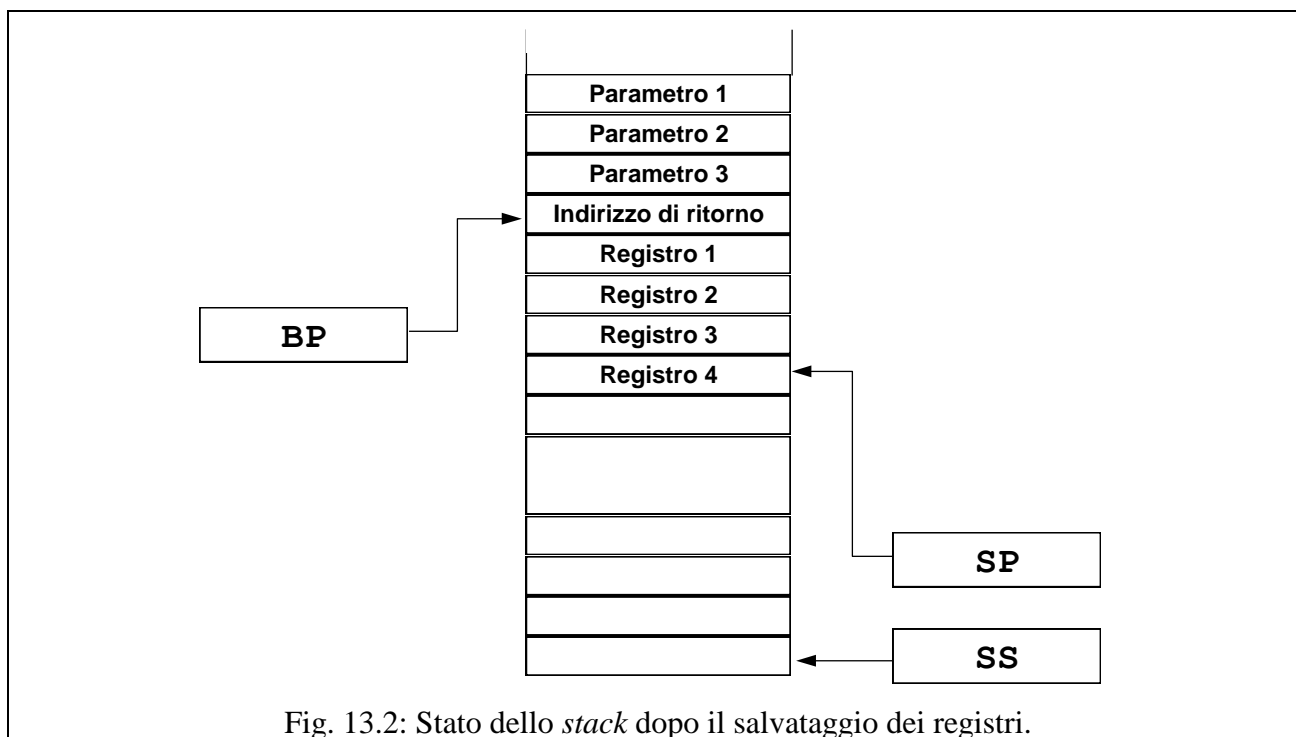
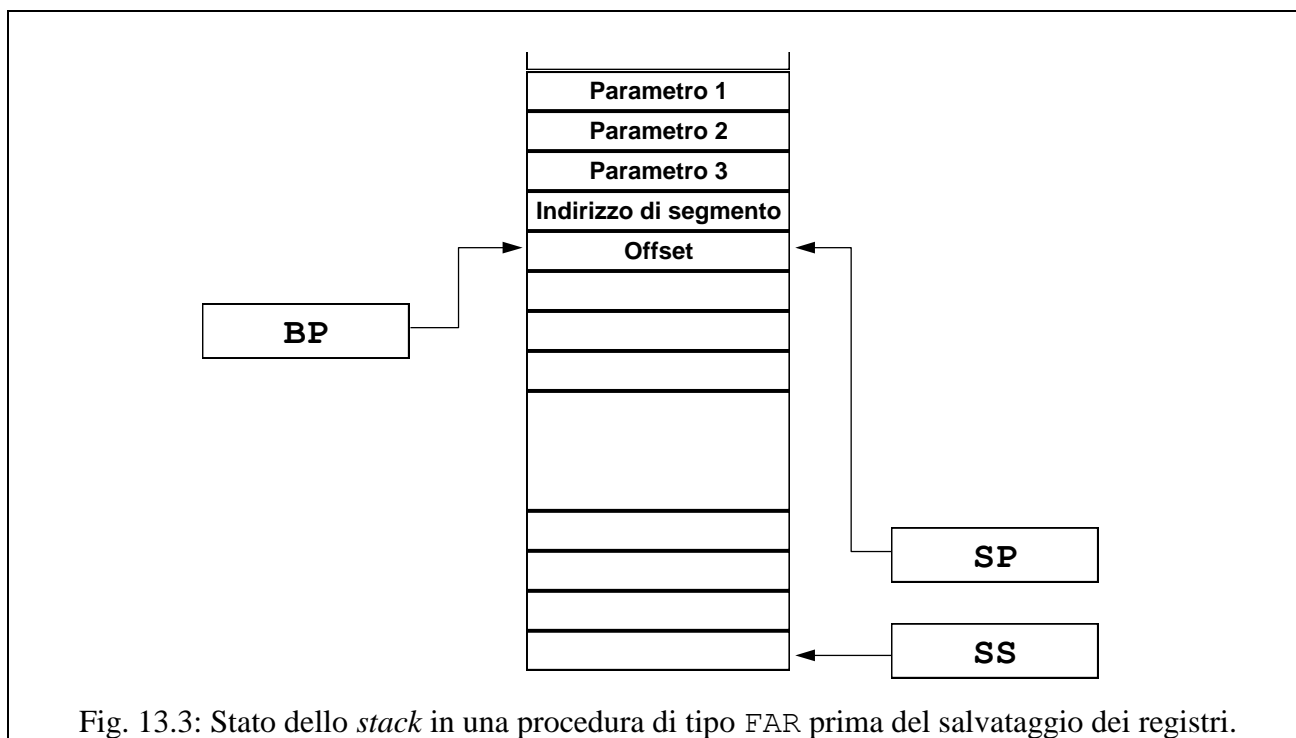


Fig. 13.1: Stato dello *stack* prima del salvataggio dei registri.

Una volta che il registro BP è caricato, la procedura chiamata può salvare i registri nello *stack*. La Fig. 13.2 mostra lo stato dello *stack* dopo il salvataggio dei registri.



Le considerazioni fatte finora sono valide per procedure di tipo **NEAR**, in cui l'indirizzo di ritorno è costituito da una word. Analogo discorso può essere fatto nel caso in cui la procedura chiamata sia di tipo **FAR**, con l'accorgimento che l'indirizzo di ritorno è qui costituito da due word. Lo stato dello *stack* prima e dopo il salvataggio dei registri nel caso di una procedura di tipo **FAR** è mostrato in Fig. 13.3.



Utilizzo di variabili locali

Una variabile locale è una variabile definita unicamente durante la chiamata della procedura stes-

sa. Le variabili locali sono memorizzate nello *stack*. L'allocazione dell'area dati di variabili locali avviene attraverso o una serie di istruzioni *PUSH* o sottraendo a *SP* un valore corrispondente alla dimensione dell'area dati. Per accedere all'area di variabili locali si utilizza il registro *BP*.

Esempio

La seguente istruzione riserva un'area dati di variabili locali di 3 word.

```
SUB    SP, 6
```

Passaggio dei parametri in uscita

È possibile utilizzare lo *stack* anche per passare alla procedura chiamante i parametri di uscita. Essi non possono essere caricati nello *stack* con un'operazione di *push* perché in tal caso sarebbero posizionati in cima allo *stack* e non permetterebbero un corretto ritorno alla procedura chiamante. Anche per la scrittura dei parametri nello *stack* è necessario utilizzare il registro *BP*.

È compito della procedura chiamante eseguire le opportune operazioni di *pop* per la lettura dei valori di ritorno.

Esempio

L'istruzione seguente restituisce alla procedura chiamante il contenuto della variabile *SOMMA* caricandolo nello *stack*:

```
MOV    [BP+6], SOMMA
```

In Fig. 13.4 è mostrato lo stato dello *stack* in una procedura di tipo *NEAR* al momento della restituzione del controllo alla procedura chiamante. Come si può notare, il valore del parametro di ritorno è scritto in una posizione dello *stack* che precedentemente conteneva il valore di un parametro di ingresso.

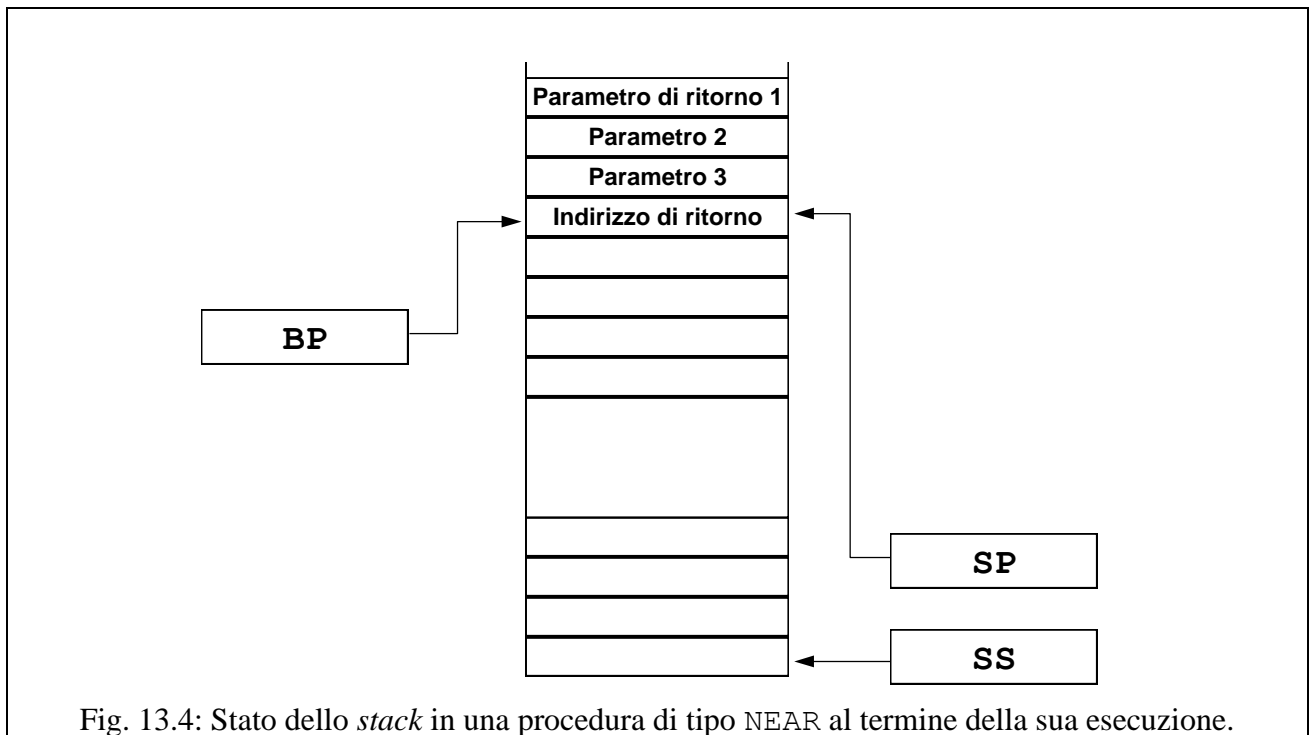


Fig. 13.4: Stato dello *stack* in una procedura di tipo *NEAR* al termine della sua esecuzione.

Liberazione dello stack

L'ultima operazione da eseguire è la liberazione dello *stack*. È solitamente compito della procedura chiamante liberare lo *stack*, cancellando le parole che sono state utilizzate per il passaggio dei parametri. Questo può essere fatto o con successive operazioni di *pop*, o incrementando opportunamente il valore del registro *Stack Pointer*.

Esempi

Il seguente frammento di codice mostra un esempio di passaggio di parametri attraverso lo *stack*. La liberazione dello *stack* viene fatta mediante l'esecuzione di 3 istruzioni *POP*.

```

...
PUSH    PARAM1
PUSH    PARAM2
PUSH    PARAM3
CALL    MY_PROC
POP      DX          ; pop dello stack
POP      DX
POP      DX
...
```

Il seguente frammento di codice è equivalente al precedente; in questo caso la liberazione dello *stack* viene effettuata incrementando opportunamente il valore del registro *SP*.

```

...
PUSH    PARAM1
PUSH    PARAM2
PUSH    PARAM3
CALL    MY_PROC
ADD     SP, 6
...
```

Se la procedura non restituisce alcun parametro memorizzato nello *stack*, la liberazione dello *stack* può essere fatta all'interno della procedura chiamata mediante l'esecuzione dell'istruzione *RET*.

Esempio

Il seguente frammento di programma mostra un'altra soluzione al problema illustrato precedentemente; essa utilizza lo *stack* per il passaggio dei parametri in ingresso ed in uscita.

```

LUNG      EQU      100
           .MODEL   small
           .DATA
VETT       DW       LUNG DUP (?)
SOMMA      DW       ?
TEMP       DW       ?
           .CODE
...
MOV        AX, LUNG
LEA        BX, VETT
SUB        SP, 2      ; allocazione per il parametro di ritorno
PUSH       AX         ; primo parametro caricato nello stack
PUSH       BX         ; secondo parametro caricato nello stack
CALL       SOM_VETT   ; chiamata della procedura
ADD        SP, 4      ; liberazione dello stack
               ; equivalente a 2 istruzioni di POP
POP        SOMMA      ; lettura del parametro di ritorno
...
SOM_VETT   PROC       ; procedura somma vettore
```

```

        MOV     BP, SP      ; copia del valore dello stack pointer
        PUSH   BX          ; salvataggio dei registri nello stack
        PUSH   CX
        PUSH   AX
        MOV     CX, [BP+4]  ; copia del primo parametro in CX
        MOV     BX, [BP+2]  ; copia del secondo parametro in BX
        XOR     AX, AX      ; azzeramento del registro AX
ciclo:  ADD     AX, [BX]     ; AX = AX + [BX]
        ADD     BX, 2       ; aggiornamento dell'indice del vettore
        LOOP    ciclo      ; scansione del vettore
        MOV     [BP+6], AX  ; caricamento di AX sullo stack
        POP     AX         ; ripristino dei registri dallo stack
        POP     CX
        POP     BX
        RET                     ; ritorno alla procedura chiamante
SOM_VETT ENDP              ; fine procedura
        ...

```

Riepilogo delle varie operazioni per l'esecuzione di una procedura

Le operazioni che possono essere effettuate durante la chiamata e l'esecuzione di una procedura sono le seguenti:

- la procedura chiamante
 - copia nello stack i parametri
 - esegue l'istruzione CALL
- la procedura chiamata
 - salva nello stack il registro BP
 - trasferisce in BP il valore di SP
 - riserva nello stack eventuali locazioni per le variabili locali alla procedura, sottraendo a SP un valore corrispondente alla dimensione dell'area dati di variabili locali
 - esegue il salvataggio dei registri nello *stack*
 - accede ai parametri tramite BP: supponendo che la procedura chiamata sia di tipo NEAR l'ultimo parametro messo nello stack dal chiamante sarà all'indirizzo [BP+4], il penultimo all'indirizzo [BP+6], e così via; supponendo che la procedura chiamata sia di tipo FAR l'ultimo parametro messo nello stack dal chiamante sarà all'indirizzo [BP+6], il penultimo all'indirizzo [BP+8], e così via;
 - accede all'area dati di variabili locali attraverso BP: detto N il numero di byte di cui essa si compone, il suo indirizzo di testa sarà dato da [BP-N], e quello della sua ultima locazione da [BP-2] (supponendo che le variabili locali siano di tipo WORD)
 - scrittura dell'eventuale parametro di ritorno
 - ripristina il contenuto dei registri
 - libera lo *stack deallocando* l'area di memoria contenenti le variabili locali
 - ripristina il contenuto di BP
 - restituisce il controllo alla procedura chiamante mediante l'istruzione RET
- la procedura chiamante
 - legge l'eventuale parametro di ritorno
 - ripulisce lo *stack deallocando* l'area contenente i parametri.

Esempio

Il frammento di codice seguente presenta la sequenza di istruzioni per la chiamata di una procedura con passaggio di parametri attraverso lo *stack*:

```

...
PUSH    param_1
...
PUSH    param_n
CALL    my_proc
...
; lettura dell'eventuale
; parametro di ritorno
ADD     SP, dim_area_parametri
...
```

Il frammento di codice seguente presenta lo schema per la procedura chiamata:

```

my_proc  PROC
          PUSH    BP
          MOV     BP, SP
          SUB     SP, dim_area_dati_locali
          PUSH    registri
          ...
          ...
          ; istruzioni
          ...
          ; scrittura dell'eventuale
          ; parametro di ritorno
          POP     registri
          ADD     SP, dim_area_dati_locali
          POP     BP
          RET
my_proc  ENDP
```

13.8. Condizione di errore

Un'informazione importante che la procedura chiamata deve restituire alla procedura chiamante è relativa al successo o all'insuccesso dell'esecuzione della procedura stessa. Un possibile modo per segnalare se una procedura è terminata correttamente consiste nell'utilizzare il flag di *carry*, ad esempio azzerandolo nel caso in cui la procedura termini correttamente e forzandolo ad 1 in caso contrario.

È compito della procedura chiamante controllare il valore del flag CF mediante le istruzioni di salto condizionato per verificare la corretta esecuzione della procedura.

Esercizio: *Calcolo della somma di un vettore di interi.*

Si realizzi una procedura che esegua la somma di un vettore di numeri interi positivi; se è stata riscontrata una condizione di errore, la procedura chiamante deve visualizzare un messaggio di errore. La soluzione proposta in linguaggio C è la seguente:

```

#include <stdio.h>
int som_vett(int *vett, int count);
main()
{
    int dati[100], somma;
    ...
    somma = som_vett(dati, 100);
    if (somma == -1)
        printf("Errore nei dati\n");
    ...
}
```

```

int som_vett(int *vett, int count)
{
    int i, somma = 0;
    for (i=0 ; i<count; i++)
        if (dati[i] >= 0)
            somma += dati[i];
        else return(-1);
    return(somma);
}

```

La soluzione proposta in linguaggio Assembler è la seguente:

```

LUNG      EQU      100
           .MODEL   small
           .DATA
DATI       DW      LUNG DUP (?)
SOMMA     DW      ?
TEMP      DW      ?
MSG       DB      "Errore nei dati",0DH,0AH,"$"
           .CODE
           ...
           MOV      AX, LUNG
           LEA      BX, DATI
           PUSH     AX           ; primo parametro nello stack
           PUSH     BX           ; secondo parametro nello stack
           CALL     SOM_VETT      ; chiamata della procedura
           MOV      SOMMA, AX     ; lettura del parametro di ritorno
           POP      AX           ; liberazione dello stack
           POP      AX
JNC        fine                 ; CF = 1 ?, No: va a fine
           LEA      DX, MSG       ; Sì (condizione di errore)
           MOV      AH, 09H       ; visualizza un messaggio di errore
           INT      21H
fine:      ...
SOM_VETT   PROC                ; procedura di somma vettore
           MOV      BP, SP        ; copia dello stack pointer in BP
           PUSH     BX           ; salvataggio dei registri nello stack
           PUSH     CX
           PUSH     DX
           MOV      CX, [BP+4]    ; copia del primo parametro in CX
           MOV      BX, [BP+2]    ; copia del secondo parametro in BX
           XOR      AX, AX        ; azzeramento di AX
ciclo:     MOV      DX, [BX]
           CMP      DX, 0         ; elemento del vettore < 0 ?
           JNL      ok           ; No: va a ok
           STC                ; Sì: forza ad 1 il flag CF (errore)
           JMP      fin          ; salta alla fine della procedura
ok:        ADD      AX, [BX]      ; AX = AX + [BX]
           ADD      BX, 2         ; aggiornamento dell'indice del vettore
           LOOP     ciclo        ; scansione del vettore
           CLC                ; azzeramento di CF
fin:       POP      DX           ; ripristino dei registri dallo stack
           POP      CX
           POP      BX
           RET                ; ritorno alla procedura chiamante
SOM_VETT   ENDP
           ...

```

Nella soluzione proposta è stato utilizzato lo *stack* per il passaggio dei parametri di ingresso (numero di elementi ed offset del vettore), il registro AX per il parametro di ritorno (somma degli elementi del vettore) ed il flag CF per la segnalazione di una condizione di errore verificatosi all'interno della procedura.

Esercizio: *Elaborazione di una stringa caricata da tastiera.*

Si vuole realizzare un programma che esegua il caricamento di una stringa da tastiera. Tale stringa, una volta convertita in caratteri maiuscoli, deve essere visualizzata su video. Un errore nel caricamento della stringa deve essere segnalato mediante messaggio su video.

Il codice in linguaggio C che esegue tali operazioni è il seguente:

```
#include <stdio.h>
char load_str (int lung, char *vett);
void conv_maiu (char *vett);
void visu_str (char *vett);
main()
{
    char err, string[80];
    err = load_str(80, string);
    if (!err)
        { conv_maiu(string);
          visu_str(string);
        }
}
char load_str (int lung, char *vett)
{
    int i=0;
    char ko = 1;
    lung--;
    printf("Scrivi una stringa terminata da un <CR>\n");
    do
        { vett[i] = getchar();
          if (vett[i] == '\n')
              { vett[i] = '\0';
                ko = 0;
                break;
              }
          i++;
        }
    while (i<lung);
    if (ko) printf("\nSTRINGA TROPPO LUNGA\n");
    return(ko);
}
void conv_maiu (char *vett)
{
    int i = 0;
    while (vett[i] != '\0')
        { if ((vett[i] <= 'z') && (vett[i] >= 'a'))
            vett[i] += ('A'-'a');
          i++;
        }
}
void visu_str (char *vett)
{ printf("%s",vett);
}
}
```

Si propone una soluzione in linguaggio Assembler in cui il programma è suddiviso in tre moduli: uno contenente il programma principale, uno contenente le procedure per la gestione dell'I/O (LOAD_STR e VISU_STR) ed uno contenente la procedura di elaborazione del vettore (CONV_MAIU). Il modulo seguente contiene il programma principale:

```
EXTRN    LOAD_STR:NEAR, CONV_MAIU:NEAR, VISU_STR:NEAR
PUBLIC   MSG
DIM      EQU      80
          .MODEL   small
          .STACK
```

```

.DATA
MSG DB "Scrivi una stringa terminata da un <CR>",0DH,0AH,"$"
STRING DB DIM DUP (?)
.CODE
.STARTUP
LEA BX, STRING
MOV AX, LENGTH STRING
PUSH AX ; copia del primo parametro nello stack
PUSH BX ; copia del secondo parametro nello stack
CALL LOAD_STR ; caricamento della stringa
JC esci ; errore ?, Si: va a esci (errore)
PUSH BX ; No: copia il parametro nello stack
CALL CONV_MAIU ; conversione in caratteri maiuscoli
PUSH BX ; copia il parametro nello stack
CALL VISU_STR ; visualizzazione su video
esci: .EXIT ; fine del programma principale
END ; fine del modulo

```

Il modulo seguente contiene la procedura di gestione della stringa:

```

PUBLIC CONV_MAIU
.MODEL small
.CODE
CONV_MAIU PROC NEAR ; procedura di conversione di una stringa
MOV BP, SP ; copia dello stack pointer in BP
PUSH BX ; salvataggio del registro BX
MOV BX, [BP+2] ; copia del parametro in BX
cicl: CMP BYTE PTR [BX], "$" ; [BX] = "$" ?
JE fin ; Si: va a fin
CMP BYTE PTR [BX], 'z' ; No: [BX] > 'z'?
JG last ; Si: va a last
CMP BYTE PTR [BX], 'a' ; No: [BX] < 'a' ?
JL last ; Si: va a last
ADD BYTE PTR [BX], 'A'-'a' ; conversione da minuscolo
; a maiuscolo
last: INC BX ; scansione del ciclo
JMP cicl ; ritorna a cicl
fin: POP BX ; ripristino del registro BX
RET 2 ; ritorno alla procedura chiamante e
; pulizia dello stack
CONV_MAIU ENDP ; fine della procedura
END ; fine del modulo

```

Il modulo seguente contiene le procedure di gestione dell'I/O:

```

PUBLIC LOAD_STR, VISU_STR
EXTRN MSG:BYTE
CR EQU 13
.MODEL small
.DATA
MSG2 DB 0DH,0AH,"STRINGA TROPPO LUNGA",0DH,0AH,"$"
.CODE
LOAD_STR PROC NEAR
MOV BP, SP ; copia dello stack pointer in BP
PUSH AX ; salvataggio dei registri
PUSH BX
PUSH CX
PUSH DX
MOV CX, [BP+4] ; copia del primo parametro in CX
DEC CX ; decremento di CX (per memorizzare "$")
MOV BX, [BP+2] ; copia del secondo parametro in BX
LEA DX, MSG ; copia dell'offset di MSG in DX
MOV AH, 09H ; visualizzazione della stringa su video
INT 21H
ciclo: MOV AH, 1 ; lettura di un carattere da tastiera

```



```

        INT     21H
        MOV     [BX], AL      ; copia del carattere letto in [BX]
        INC     BX           ; scansione del vettore
        CMP     AL, CR       ; carattere letto è CR ?
        LOOPNE  ciclo        ; No e CX ≠ 0: ritorna a ciclo
        JE      ok           ; Sì: va a ok
        STC     ; No: errore, forzo il flag CF a 1
        LEA     DX, MSG2     ; carica in DX l'offset di MSG2
        MOV     AH, 09H      ; visualizza il messaggio su video
        INT     21H
        JMP     fine         ; salta alla fine della procedura
ok:      MOV     [BX-1], "$"  ; caricamento del carattere terminatore
        CLC     ; successo: azzeramento del flag CF
fine:    POP     DX          ; ripristino dei registri dallo stack
        POP     CX
        POP     BX
        POP     AX
        RET     4           ; ritorno alla procedura chiamante
                                ; pulizia dello stack
LOAD_STR ENDP              ; fine della procedura
VISU_STR PROC NEAR        ; procedura di visualizzazione su video
        MOV     BP, SP      ; copia dello stack pointer in BP
        PUSH    AX          ; salvataggio dei registri nello stack
        PUSH    DX
        MOV     DX, [BP+2]  ; copia del parametro in DX
        MOV     AH, 09H     ; visualizzazione su video
        INT     21H
        POP     DX          ; ripristino dei registri dallo stack
        POP     AX
        RET     2           ; ritorno alla procedura chiamante e
                                ; pulizia dello stack
VISU_STR ENDP              ; fine della procedura VISU_STR
                                ; fine del modulo
END

```

13.9. Tabelle di jump

Una tabella di *jump* è una lista di indirizzi cui un programma può saltare; ad ogni elemento della tabella corrisponde l'indirizzo di una procedura. Le tabelle di *jump* sono utilizzate per avere un modo efficiente per chiamare procedure diverse a seconda del valore di una determinata variabile.

Esercizio: *Elaborazione di un vettore di interi.*

Si vuol realizzare un programma che elabori un vettore di numeri positivi. Il tipo di operazione da svolgere (calcolo della somma, calcolo della media, ricerca del valore massimo o minimo) è scelto in base al valore fornito dall'utente tramite tastiera. Una prima soluzione utilizza una serie di salti condizionati in cascata, come mostrato nell'esempio seguente:

```

SOMMA    EQU     '1'
MEDIA    EQU     '2'
MAX       EQU     '3'
MIN       EQU     '4'
LUNG      EQU     100

        .DATA
SCELTA    DB      ?
VETT      DB      LUNG DUP (?)

        .CODE
        ...
        MOV     AH, 1          ; lettura di un carattere
        INT     21H
        MOV     SCELTA, AL      ; copia il carattere in SCELTA
        CMP     SCELTA, SOMMA
        JE      lab1
        CMP     SCELTA, MEDIA
        JE      lab2

```

```

                CMP     SCELTA, MAX
                JE      lab3
                CMP     SCELTA, MIN
                JE      lab4
                JMP     lab4
lab1:           CALL    SOMMA_VETT
                JMP     lab5
lab2:           CALL    MEDIA_VETT
                JMP     lab5
lab3:           CALL    MAX_VETT
                JMP     lab5
lab4:           CALL    MIN_VETT
lab5:           ...

```

La soluzione proposta non è particolarmente funzionale, in quanto risulta altamente costosa la modifica e l'aggiunta di un'eventuale nuova opzione.

Una soluzione alternativa consiste nell'impiego delle tabelle di *jump*. Per prima cosa bisogna definire una tabella nel segmento di dato contenente gli offset delle varie procedure. Data una tabella di *jump* è possibile fare riferimento alle varie procedure attraverso gli indirizzi contenuti nella tabella stessa. A seconda del valore di *SCELTA* verrà effettuata la chiamata alla opportuna procedura andando a prelevare il corrispondente indirizzo memorizzato in tabella.

L'esempio seguente mostra la soluzione che utilizza le tabelle di *jump*:

```

LUNG           EQU      100
                .DATA
JUMP_TABLE     DW        SOMMA_VETT
                DW        MEDIA_VETT
                DW        MAX_VETT
                DW        MIN_VETT
SCELTA         DB        ?
VETT           DB        LUNG DUP (?)
                .CODE
                ...
                MOV      AH, 1                ; lettura di un carattere
                INT      21H
                MOV      SCELTA, AL           ; copia il carattere letto in SCELTA
                LEA      BX, JUMP_TABLE
                MOV      AL, SCELTA
                SUB      AL, '0'              ; converte carattere ASCII in numero
                XOR      AH, AH
                DEC      AX
                SHL      AX, 1
                ADD      BX, AX
                CALL     WORD PTR [BX]
                ...

```

I vantaggi dell'uso delle tabelle di *jump* derivano dal fatto di avere un codice:

1. più compatto, avendo un'unica istruzione di chiamata a procedura;
2. facilmente modificabile, poiché aggiunte o sostituzioni di procedure implicano unicamente un aggiornamento della tabella di *jump*.

14. Le istruzioni per il controllo del processore

In questo capitolo verranno presentate alcune istruzioni dell'Assembler x86 cui si è ritenuto di dover dare una minore importanza, in quanto essenzialmente connesse con il funzionamento hardware di un sistema a microprocessore (negli esempi riportati di seguito, l'8086) e con l'interazione di quest'ultimo con le relative periferiche. Una speciale attenzione verrà dedicata tuttavia al meccanismo di interruzione. In particolare attraverso tale meccanismo è possibile fare accesso alle funzioni di libreria offerte dal Sistema Operativo; al fine di facilitare l'uso delle più comuni funzioni viene fornita una tabella riassuntiva delle loro caratteristiche.

14.1. Le istruzioni per la gestione delle interruzioni

14.1.1. Le interruzioni hardware

La funzione principale del meccanismo dell'interruzione consiste nel permettere ad un dispositivo esterno di interrompere il processore e richiedere ad esso l'esecuzione di una appropriata sequenza di operazioni. Il processore 8086 possiede per questa funzione tre piedini: **INTR**, **INTA** e **NMI**. Attraverso i piedini INTR e NMI un dispositivo esterno può richiedere l'interruzione del programma in corso di esecuzione, e l'attivazione di una opportuna procedura di servizio. Il flag IF permette di abilitare o disabilitare le richieste che giungono sul piedino INTR. Al fine di selezionare l'opportuna procedura di servizio, il dispositivo esterno può (attraverso un protocollo di comunicazione che coinvolge il Data Bus ed il piedino INTA) comunicare all'8086 un codice su 8 bit che identifica il *tipo di interruzione*. Ne consegue che il numero di tipi di interruzione ammessi è 256.

La corrispondenza tra tipo di interruzione e relativa *procedura di servizio* è contenuta nella cosiddetta *Interrupt Vector Table*: questa corrisponde ad una zona di memoria di ampiezza pari a 1Kbyte posta agli indirizzi di memoria che vanno da 00000H a 003FFH. Per ogni tipo di interruzione la *Interrupt Vector Table* contiene 4 byte corrispondenti all'indirizzo di offset e di segmento della relativa procedura di servizio.

Per accedere a questa informazione il processore deve quindi moltiplicare per 4 il valore contenuto negli 8 bit che identificano il tipo di interruzione, ed utilizzare il valore risultante come indiriz-

zo in memoria per accedere alle parole contenenti l'indirizzo della procedura di servizio.

L'attivazione della procedura di servizio di una interruzione è simile a quella di una normale procedura: tuttavia, oltre a salvare nello stack il contenuto corrente dei registri IP e CS, in questo caso viene anche salvato il valore della PSW. In tal modo, una volta terminata la procedura di servizio, è possibile riprendere l'esecuzione del programma interrotto senza che siano cambiati i valori dei flag. Il termine della procedura di servizio è caratterizzato dall'istruzione IRET: questa differisce dalla più comune istruzione RET unicamente in quanto esegue il ripristino (estraendo i relativi valori dallo stack) dei registri IP, CS e PSW.

14.1.2. Le interruzioni software

L'8086 permette di attivare una procedura di servizio dell'interruzione anche via software, attraverso l'istruzione INT.

Questa causa l'attivazione di un meccanismo analogo a quello descritto per le interruzioni causate da dispositivi esterni. L'operando (su 8 bit) associato all'istruzione INT viene utilizzato per identificare il tipo dell'interruzione, ed accedere quindi all'interno della Vector Table per reperire l'indirizzo della relativa procedura di servizio. Attraverso questo meccanismo il DOS mette a disposizione del programmatore una serie di procedure per la gestione a vari livelli dei dispositivi di Input/Output, quali dischi, video, tastiera, nonché di alcune funzionalità di sistema, quali il clock o l'allocazione/deallocazione della memoria dinamica. Nel corso di questo testo si è ad esempio spesso utilizzata l'istruzione INT 21H che permette ad esempio di accedere alle funzioni di servizio del video e della tastiera.

L'istruzione INTO permette infine di attivare una particolare procedura di servizio (avente tipo pari a 4), atta a gestire il caso in cui si sia verificato un overflow.

14.1.3. L'istruzione INT

L'istruzione **INT** (*INTerrupt*) esegue l'attivazione di una procedura di servizio dell'interruzione. Il suo formato è il seguente:



INT {operando}

Il campo *operando* è opzionale: esso specifica il tipo di interruzione da eseguire; qualora il campo *operando* sia assente, viene eseguita la procedura di servizio avente tipo pari a 3. L'istruzione INT esegue le seguenti operazioni:

- salva nello stack il contenuto del registro PSW;
- azzeri i flag TF ed IF;
- salva nello stack il contenuto del registro CS;
- carica in CS la parola posta all'indirizzo $\text{operando} * 4 + 2$ all'interno della *Interrupt Vector Table*;
- salva nello stack il contenuto del registro IP;
- carica in IP la parola posta all'indirizzo $\text{operando} * 4$.

L'istruzione INT modifica esclusivamente i flag TF ed IF.

14.1.4. L'istruzione **INTO**

L'istruzione **INTO** (*INTerrupt if Overflow*) esegue l'attivazione condizionale di una specifica procedura di servizio dell'interruzione. Il suo formato è il seguente:



INTO

L'istruzione **INTO** controlla il valore del flag **OF**: se esso vale 1 esegue l'attivazione della procedura di interruzione avente tipo 4, secondo le stesse modalità descritte per l'istruzione **IRET**. Diversamente **INTO** non ha alcun effetto.

L'istruzione **INTO** modifica esclusivamente i flag **TF** ed **IF**.

14.1.5. L'istruzione **IRET**

L'istruzione **IRET** (*Interrupt RETurn*) esegue, al termine di una procedura di servizio dell'interruzione, le operazioni necessarie per il ritorno del controllo al programma interrotto. Il formato dell'istruzione è il seguente:



IRET

L'istruzione **IRET** esegue le seguenti operazioni:

- carica nel registro **IP** il valore della prima parola estratta dallo stack;
- carica nel registro **CS** il valore della seconda parola estratta dallo stack;
- carica nel registro **PSW** il valore della terza parola estratta dallo stack.

L'istruzione **IRET** modifica lo stato di tutti i flag ripristinando dallo stack il registro **PSW**.

14.2. Le istruzioni per la sincronizzazione con l'esterno

Le istruzioni seguenti sono utilizzate fundamentalmente per sincronizzare il processore 8086 con altri dispositivi, quali ad esempio il coprocessore matematico.

14.2.1. L'istruzione **HLT**

L'istruzione **HLT** (*HaLT*) fa entrare il processore in uno stato di inattività, da cui può uscire attraverso l'attivazione della procedura di reset, oppure di una interruzione esterna attraverso i piedini **INTR** e **NMI**. Il suo formato è il seguente:



HLT

L'istruzione **HLT** può venire utilizzata per far sì che il processore si ponga in uno stato di inattività in attesa di una particolare condizione esterna in grado di scatenare una interruzione, ad esempio la battitura di un tasto sulla tastiera. L'istruzione **HLT** non modifica il valore dei flag.

14.2.2. L'istruzione WAIT

L'istruzione **WAIT** (*WAIT*) fa entrare il processore in uno stato di inattività, da cui esce nel momento in cui viene attivato il piedino **TEST**. Il formato dell'istruzione è il seguente:



WAIT

L'istruzione **WAIT** fa entrare il processore in uno stato di inattività analogo a quello generato dall'istruzione **HLT**; tuttavia in questo caso il processore, ad intervalli di 5 colpi di clock, verifica se il piedino **TEST** è attivo ed in tal caso riprende l'esecuzione dall'istruzione successiva. Si noti che se il piedino **TEST** è già attivo all'atto dell'esecuzione dell'istruzione **WAIT**, questa non ha nessun effetto. Qualora giunga una richiesta di interruzione durante il periodo di inattività, questa viene servita, ed al termine il processore rientra nello stato di inattività.

L'istruzione **WAIT** viene utilizzata principalmente per sincronizzare il processore con un coprocessore esterno (ad esempio il coprocessore matematico 8087).

L'istruzione **WAIT** non modifica il valore dei flag.

14.2.3. L'istruzione ESC

L'istruzione **ESC** (*ESCAPE*) esegue un accesso in memoria, ad un indirizzo specificato. Il suo formato è il seguente:



ESC *codice, sorgente*

L'istruzione **ESC** viene utilizzata in presenza di un coprocessore: il coprocessore si attiva quando l'8086 esegue il fetch di un'istruzione **ESC**. L'operando *codice* individua l'istruzione che il coprocessore deve eseguire. L'operando *sorgente* corrisponde all'indirizzo dell'eventuale operando, che viene caricato dalla memoria e automaticamente letto dal coprocessore.

Nessun flag viene modificato dall'istruzione **ESC**.

14.2.4. L'istruzione LOCK

Il comando **LOCK** (*LOCK the bus*) è un prefisso che, premesso ad un'istruzione, fa sì che il controllo del bus resti all'8086 per tutta l'esecuzione dell'istruzione. Esso si usa in ambiente multiprocessore. Il suo formato è il seguente:



LOCK

Il prefisso **LOCK** fa sì che il piedino di **LOCK** resti attivo per tutto il periodo in cui viene eseguita l'istruzione cui è associato, disabilitando così l'accesso al bus da parte di qualunque altro processore ad esso connesso.

14.3. L'istruzione NOP

L'istruzione **NOP** (*No Operation*) non esegue alcuna operazione. Il suo formato è il seguente:



NOP

L'istruzione **NOP** non ha alcun effetto: non modifica né la memoria, né i registri (ad esclusione di

IP), né i flag. Tuttavia, essa può essere utile in vari casi: quando si vuole modificare il codice eseguibile di un programma senza riassemblarlo (ad esempio cancellando un'istruzione e scrivendo al suo posto un numero opportuno di istruzioni NOP) e per creare dei cicli di ritardo di lunghezza prefissata.

14.4. Le funzioni di sistema del DOS

Il Sistema Operativo DOS fornisce al programmatore una serie di funzioni di libreria accessibili principalmente attraverso l'istruzione `INT 21H`. La funzione da attivare viene individuata sulla base del valore presente nel registro `AH`. In Tab. 14.1 è riportato sinteticamente il funzionamento delle funzioni più usate.

<i>AH</i>	<i>Funzione svolta</i>	<i>Parametri di Ingresso</i>	<i>Parametri di Uscita</i>
01H	Legge un carattere da tastiera (eseguendone l'eco su video)	-	AL = carattere letto
02H	Visualizza un carattere	DL = carattere da visualizzare	-
05H	Stampa un carattere	DL = carattere da stampare	-
08H	Legge un carattere da tastiera (senza eseguire l'eco su video)	-	AL = carattere letto
09H	Visualizza una stringa	(DS:DX) = Indirizzo della stringa. La stringa deve terminare con un \$.	-
0AH	Legge una riga di caratteri da tastiera (eseguendone l'eco su video) e la scrive in un buffer	(DS:DX) = Indirizzo del buffer. Il primo byte contiene la lunghezza del buffer.	Il secondo byte contiene il numero di caratteri letti.
2AH	Legge la data	-	CX = anno (1980-2099) DH = mese (1-12) DL = giorno (1-31)
2CH	Legge l'ora	-	CH = ore (0-23) CL = minuti (0-59) DH = secondi (0-59) DL = centesimi di secondo (0-99)
4CH	Termina un processo	AL = Valore di ritorno	-

Tab. 14.1: Alcune funzioni di sistema di `INT 21H`.

15. Programmazione avanzata

In questo capitolo vengono descritte alcune modalità di programmazione Assembler di tipo avanzato. In particolare verranno illustrati gli accorgimenti necessari per eseguire la corretta integrazione tra programmi scritti in Assembler e programmi scritti in C; verranno inoltre presentate sia alcune procedure che permettono di eseguire operazioni dinamiche di allocazione e deallocazione di strutture dati sia alcuni esempi di procedure recursive.

15.1. Procedure Assembler richiamabili da un programma C

In questo paragrafo viene spiegato come scrivere procedure in linguaggio Assembler che possono essere chiamate da programmi scritti in linguaggio C. I vari moduli sorgente, *compilati* separatamente, vengono *linkati* insieme al fine di costituire un unico programma eseguibile. Tale modo di procedere, abbastanza frequente nella pratica, permette di limitare ad alcune parti particolarmente critiche l'uso del linguaggio Assembler, mentre altre parti possono essere più comodamente sviluppate in linguaggi di alto livello.

15.1.1. Regole per la procedura Assembler

Dichiarazione della procedura chiamata

Al fine di poter *linkare* una procedura Assembler con un programma chiamante C è innanzitutto necessario che ci sia compatibilità tra i segmenti usati.

La dichiarazione della procedura chiamata dipende dal modello usato dal programma chiamante C. È necessario infatti utilizzare lo stesso modello di memoria sia per il modulo C sia per il modulo Assembler: la procedura Assembler va dichiarata come NEAR per modelli *tiny*, *small* e *compact* oppure come FAR per modelli *medium*, *large* o *huge*.

Il nome della procedura Assembler deve essere reso pubblico tramite una dichiarazione PUBLIC, così come il nome di ogni altra variabile che si desidera rendere accessibile dall'esterno.

Al contrario, i nomi di tutte le variabili e procedure definite esternamente al modulo Assembler e da esso utilizzate vanno dichiarate come tali tramite la direttiva `EXTRN`.

Convenzioni per i nomi

La convenzione sui nomi specifica come il compilatore altera il nome degli identificatori prima di memorizzarli nel file oggetto.

Tutti i nomi delle entità comuni ai moduli C ed a quello Assembler devono tener conto del fatto che il compilatore C premette sempre, nella costruzione della *symbol table* interna, un carattere `'_'`. Questo significa che il nome della procedura Assembler deve iniziare con tale carattere, così come quello di tutte le variabili che sono state rese pubbliche e possono essere usate dal modulo C.

È possibile risolvere tale problema utilizzando l'opzione di linguaggio `c` nella direttiva `.MODEL`. Specificando tale opzione l'assemblatore aggiunge il carattere `'_'` davanti a tutti gli identificatori del modulo Assembler.

Si tenga conto che il linguaggio C è *case sensitive*; per fare in modo di preservare il *case* negli identificatori (ossia per impedire che venga effettuata la conversione delle lettere minuscole in lettere maiuscole) è necessario compilare i moduli Assembler con l'opzione `/Cp` e *linkare* i vari moduli con l'opzione `/NOI`.

Compatibilità del tipo di dato

Il linguaggio C presenta una molteplicità di tipi di dati, mentre come si è visto il linguaggio Assembler presenta un numero ristretto di possibili tipi di dato. La Tab. 15.1 presenta la corrispondenza dei tipi di dato tra il linguaggio C ed il MASM.

C	MASM
char	BYTE
short, int	WORD
long, float	DWORD
double	QWORD
long double	TBYTE

Tab. 15.1: Compatibilità dei tipi di dato.

I puntatori in C specificano indirizzi di variabili o di funzioni. In base al modello di memoria utilizzato un puntatore occupa una word (puntatore di tipo `NEAR`) oppure una doubleword (puntatore di tipo `FAR`). La Tab. 15.2 riassume i tipi di dato da utilizzare per i puntatori per i diversi modelli di memoria.

modello di memoria	puntatore a funzione	puntatore a dato
tiny	WORD	WORD
small	WORD	WORD
medium	DWORD	WORD
compact	WORD	DWORD
large	DWORD	DWORD
huge	DWORD	DWORD

Tab. 15.2: Tipi di dato per i puntatori.

Convenzione sui parametri di ingresso

Il codice generato dal compilatore C passa i parametri alle procedure mettendoli nello stack in ordine inverso a quello in cui compaiono nella chiamata.

Ad essi si può quindi fare accesso attraverso il registro BP. Le prime istruzioni da eseguire all'interno della procedura Assembler sono dunque le seguenti:

```
PUSH    BP           ; salvataggio del registro BP
MOV     BP, SP       ; copia in BP del valore di SP
```

Dopo l'esecuzione di questa coppia di istruzioni:

- se la procedura è di tipo NEAR, l'indirizzo di ritorno è memorizzato all'indirizzo [BP+2], il primo parametro è indirizzabile attraverso [BP+4], e così via;
- se la procedura è di tipo FAR, il primo parametro è memorizzato all'indirizzo [BP+6], in quanto l'indirizzo di ritorno è memorizzato sotto forma di segmento e di offset.

Variabili locali

All'interno della procedura può essere allocato spazio per eventuali variabili locali, così come accade nei linguaggi di alto livello.

Per fare questo è necessario riservare un'area dello stack utilizzabile per la memorizzazione di variabili locali. Tale operazione può essere fatta o con un numero opportuno di istruzioni PUSH, oppure decrementando il contenuto di SP attraverso un'istruzione SUB, come mostrato di seguito:

```
SUB     SP, nbyte
```

dove *nbyte* indica il numero di byte che si intendono destinare alla memoria locale alla procedura.

Alle variabili locali si può fare accesso attraverso il registro BP.

Esempio

L'istruzione seguente copia il contenuto del registro AX nella prima word dell'area dello stack in cui sono memorizzate le variabili locali.

```
MOV     [BP-2], AX    ; copia di AX nello stack
```

Salvataggio dei registri

Il compilatore C della Microsoft richiede che eventuali procedure chiamate da un programma C non modifichino i valori contenuti nei registri SI, DI, SS, DS e BP. Nel caso in cui tali registri debbano essere utilizzati, essi devono essere opportunamente salvati nello stack e poi ripristinati al termine. È comunque buona norma di programmazione salvare tutti i registri utilizzati dalla procedura.

In Fig. 15.1 è mostrato uno schema di ambiente di procedura Assembler richiamabile da programma C.

Convenzione sui parametri di uscita

Il parametro eventualmente ritornato dalla procedura Assembler è atteso dal chiamante nel *registro accumulatore*. Se il tipo di dato di ritorno è un `char` il parametro di ritorno è passato attraverso il registro AL; se il tipo è un `int` o un indirizzo di tipo NEAR il registro utilizzato è AX; se il tipo è un `long` o un indirizzo di tipo FAR il parametro di ritorno è copiato nella coppia di registri DX

(word più significativa) e AX (word meno significativa).

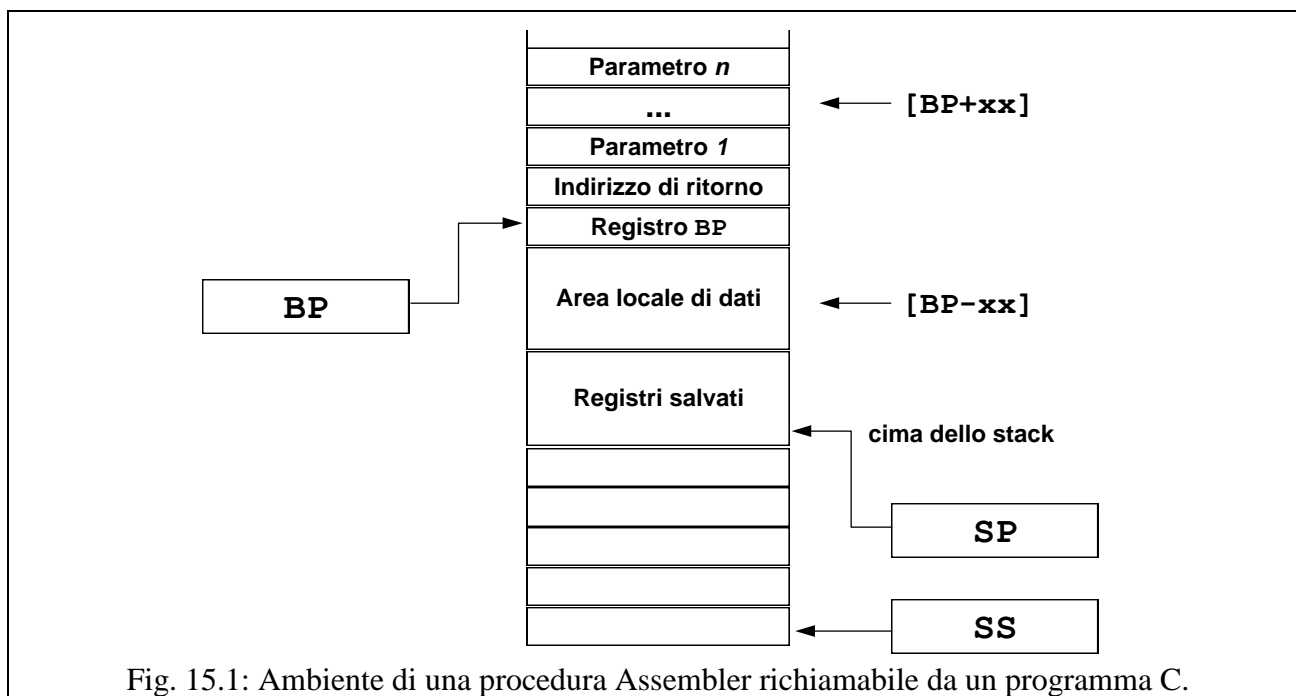


Fig. 15.1: Ambiente di una procedura Assembler richiamabile da un programma C.

Uscita dalla procedura

Le operazioni da eseguire a conclusione della procedura sono:

1. ripristinare i valori dei registri eventualmente salvati all'inizio;
2. liberare l'area locale di dati (se esiste) incrementando opportunamente il contenuto del registro SP;
3. eseguire l'istruzione RET.

15.1.2. Regole per la procedura C chiamante

Il nome della procedura chiamata e tutte le variabili globali definite nel modulo Assembler devono essere dichiarate come *extern* all'interno della procedura C.

È compito del programma chiamante C svuotare lo stack dello spazio destinato ai parametri di ingresso. Tale operazione è effettuata dal compilatore C in maniera automatica.

Esercizio: Calcolo di una semplice espressione aritmetica.

Si vuole scrivere una procedura Assembler di nome *power2* richiamabile da un programma scritto in linguaggio C per il calcolo dell'espressione $x \cdot 2^y$. Alla procedura *power2* vengono passati i due parametri interi x e y ; la funzione restituisce nel registro AX il risultato dell'espressione. La procedura non fa uso di memoria locale. Si suppone che il programma chiamante sia compilato usando il modello di memoria *small*. Il programma chiamante C è il seguente:

```
#include <stdio.h>
extern int power2 (int factor, int power);
void main()
{
    printf(" 3 volte 2 elevato a 5 = %d\n", power2(3,5) );
}
```

Il file contenente la procedura Assembler è il seguente:

```

PUBLIC    _power2
.MODEL    small
.CODE
_power2 PROC
    PUSH    BP
    MOV     BP, SP
    MOV     AX, [BP+4]          ; primo parametro (factor)
    MOV     CX, [BP+6]          ; secondo parametro (power)
    SHL     AX, CL
    POP     BP
    RET                                ; in AX c'è il risultato
_power2 ENDP
END

```

Esercizio: *Inversione del contenuto di una stringa.*

Si vuole eseguire una procedura Assembler di nome `str_inv` richiamabile da un programma scritto in linguaggio C per l'inversione del contenuto di una stringa; al termine dell'esecuzione, gli elementi del vettore devono essere memorizzati nell'ordine inverso rispetto a quello iniziale. Il programma chiamante C è il seguente:

```

#include <stdio.h>
#include <stdlib.h>
extern char * str_inv (char *str);
void main()
{
    char *s;
    s = strdup("Salve, mondo !");
    printf("%s\n", str_inv(s));
}

```

Il file contenente la procedura Assembler è il seguente:

```

PUBLIC    _str_inv
.MODEL    small
.CODE
_str_inv PROC
    PUSH    BP
    MOV     BP, SP
    PUSH    SI
    PUSH    DI
    MOV     AX, DS
    MOV     ES, AX
    MOV     DI, WORD PTR [BP+4]    ; in DI l'inizio della stringa
    MOV     SI, DI
    XOR     AX, AX                  ; azzera AX
    MOV     CX, 0FFFFH
    REPNE   SCASB                  ; cerca la fine della stringa
    SUB     DI, 2
    NOT     CX
    DEC     CX                      ; in CX c'è la dimensione
                                    ; della stringa
    ciclo:  SHR     CX, 1            ; CX = CX / 2
    MOV     AH, [SI]
    XCHG    AH, [DI]               ; scambio del contenuto
    MOV     [SI], AH
    INC     SI                      ; aggiornamento degli indici
    DEC     DI
    LOOP    ciclo
    POP     DI
    POP     SI

```

```

                MOV     AX, WORD PTR [BP+4]    ; parametro di ritorno
                POP     BP
                RET
_str_inv        ENDP
                END

```

15.2. Procedure C richiamabili da un programma Assembler

In questo paragrafo viene spiegato come è possibile richiamare una procedura scritta in linguaggio C all'interno di una procedura Assembler. In particolare se si vogliono richiamare funzioni di libreria C occorre inizializzare le variabili globali di libreria. Per fare ciò è necessario eseguire il codice di *startup* di C. Per risolvere tale problema vengono fornite due possibili soluzioni.

15.2.1. Codice C di *startup*

Per poter utilizzare funzioni di libreria C è necessario *linkare* il modulo C di *startup*, che esegue le opportune inizializzazioni richieste dalle funzioni di libreria. È necessario seguire i seguenti passi:

- specificare la convenzione *c* nel comando `.MODEL;`
- definire come esterna la costante `_acrtused` per *linkare* il modulo C di *startup*;
- dichiarare come esterne tutte le funzioni C utilizzate;
- includere una procedura pubblica di nome *main* coincidente con l'*entry point* del programma (questo perché il codice C di *startup* chiama la procedura `_main`);
- omettere il campo di *entry point* nella direttiva `END`;
- assemblare con l'opzione `/Cp` per preservare il *case* dei nomi della procedura;
- *linkare* i vari moduli con l'opzione `/NOI` includendo le opportune librerie C.

Esempio

Si vuole scrivere un frammento di codice Assembler che esegue la visualizzazione di una stringa facendo uso della funzione di libreria C `printf()`.

```

EOS      EQU      0
LF       EQU      10
CR       EQU      13
                EXTRN _acrtused:abs, printf:NEAR
                PUBLIC main
                .MODEL small, c
                .STACK
                .DATA
my_str   DB      "Salve Mondo!", CR, LF, EOS
                .CODE
main     PROC
...
        LEA      AX, my_str
        PUSH     AX
        CALL     printf
        POP      AX
        ...
        RET
main     ENDP
                END

```

15.2.2. Programma principale in C

Il problema dell'inizializzazione dell'ambiente di libreria C può essere risolto utilizzando un *main* scritto in C che richiama la procedura principale del programma Assembler. In questo modo l'inizializzazione delle librerie C è trasparente al modulo Assembler.

Il *main* in C esegue come unica operazione la chiamata alla procedura principale Assembler.

Nel modulo Assembler occorre dichiarare come esterne tutte le funzioni C utilizzate e dichiarare come pubblica la procedura principale.

Nel modulo C occorre dichiarare come esterna la procedura Assembler richiamata.

Esempio

Si propone ora la soluzione al problema mostrato precedentemente. Il modulo C è il seguente:

```
extern void visual (void);
main ()
{
    visual();
}
```

Il modulo Assembler è il seguente:

```
EOS      EQU      0
LF       EQU      10
CR       EQU      13
          .MODEL   small, c
          EXTRN    printf:NEAR
          PUBLIC   visual
          .DATA
my_str   DB        "Salve Mondo!", CR, LF, EOS
          .CODE
visual   PROC
...
          LEA      AX, my_str
          PUSH     AX
          CALL     printf
          POP      AX
          RET
...
visual   ENDP
          END
```

15.3. Strutture dinamiche

In questo paragrafo vengono presentate alcune tecniche atte a permettere l'uso di strutture dinamiche di memorizzazione all'interno di programmi Assembler. Verranno cioè proposte delle procedure in grado di sostituire almeno in parte le primitive di allocazione e deallocazione dinamica della memoria presenti in numerosi linguaggi di programmazione di alto livello. La disponibilità di tali primitive permette l'uso di strutture dinamiche quali liste od alberi, che si rivelano essenziali nella risoluzione di taluni problemi. Verrà infine presentato un esempio relativo alla gestione delle liste.

15.3.1. Allocazione e deallocazione

Le comuni primitive di allocazione e deallocazione della memoria messe a disposizione dal Sistema Operativo della macchina su cui si lavora permettono di gestire la memoria a disposizione

dell'utente in modo tale da mascherarne la natura *statica* (allocazione a priori, senza possibilità di deallocazione) e da permetterne un uso *dinamico* (allocazione e deallocazione secondo le necessità). Questo significa che quando un programma C esegue una chiamata alla funzione di libreria `malloc()`, questa ritorna un puntatore ad una parte di memoria che il Sistema Operativo aveva destinato al processo corrispondente al programma. La libreria C mantiene poi via via aggiornata la mappa corrispondente allo stato di occupazione di tale memoria, ricordando in ogni istante le locazioni utilizzate e quelle libere, sulla base dell'evolversi delle chiamate alle primitive `malloc()` e `free()`, eventualmente richiedendo al Sistema Operativo ulteriore memoria. Come esercizio si propone una soluzione al problema della allocazione e della deallocazione della memoria che utilizza strategie simili a quelle adottate dalle analoghe funzioni di libreria C, personalizzandole però allo specifico problema considerato e permettendo dunque una maggiore efficienza.

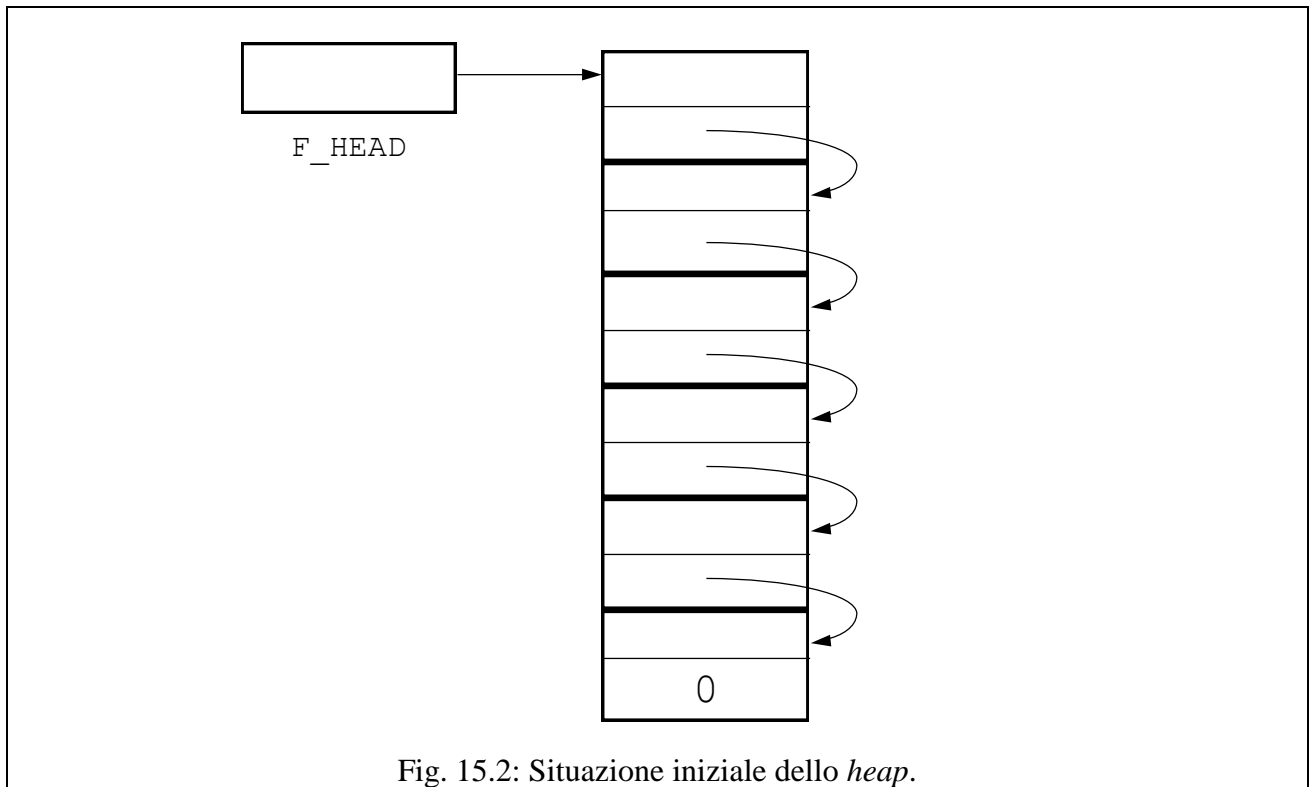
Si supponga dunque di voler far uso di una lista composta da elementi corrispondenti a record di dimensione pari a `DIM_REC` byte. Sia poi `N_MAX` il numero massimo di elementi che si pensa di dover utilizzare, compatibilmente con la disponibilità di memoria del sistema a disposizione.

Il programma dovrà allocare inizialmente un vettore `HEAP` composto di `N_MAX * DIM_REC` byte, che costituiranno la memoria da cui prelevare i record di cui sarà composta la lista. Sono dunque necessarie due procedure:

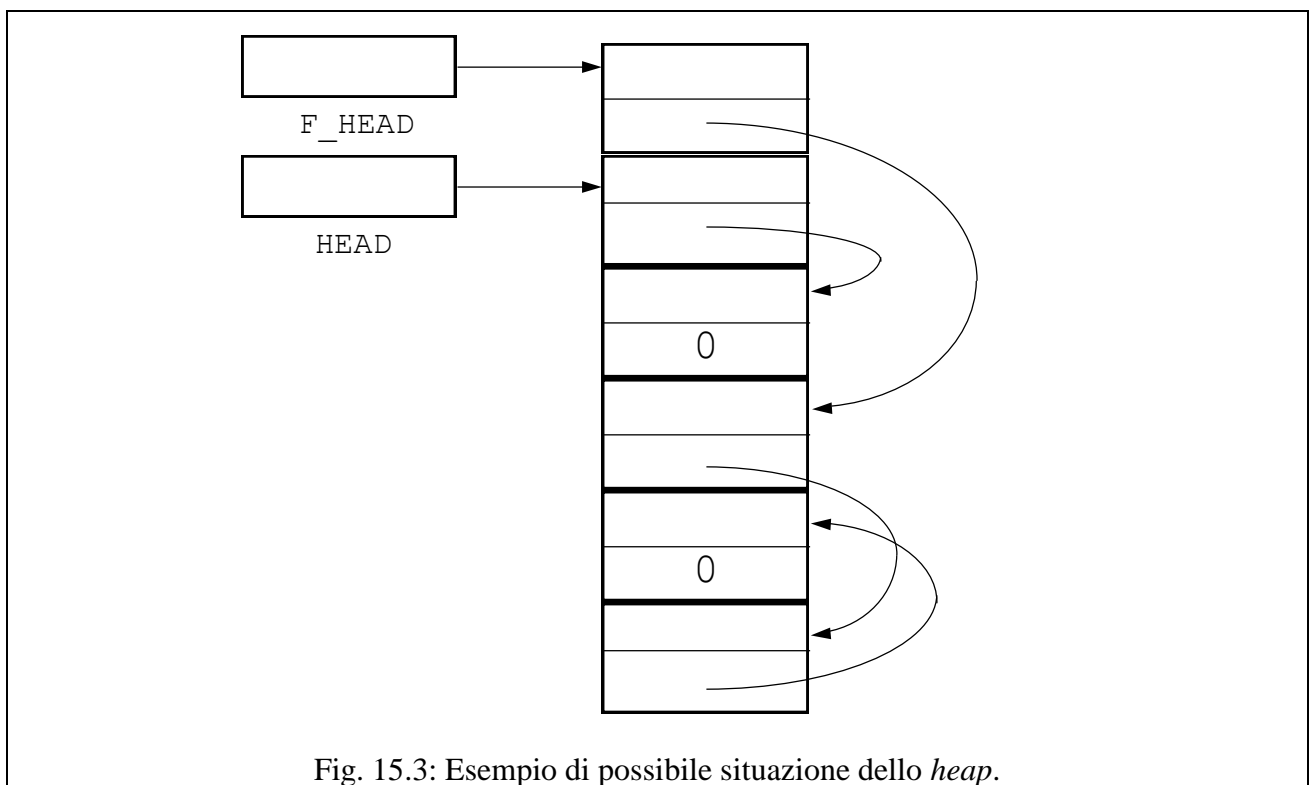
- **ALLOC**: ogni volta che viene chiamata, essa cerca in `HEAP` un record correntemente non utilizzato, e ritorna in `DX` il puntatore ad esso. Nel caso in cui `HEAP` abbia dimensioni inferiori a 64 Kbyte (e possa quindi essere interamente allocato in un solo segmento), tale puntatore può essere un intero su 16 byte corrispondente all'offset del record selezionato rispetto all'inizio del segmento.
- **FREE**: riceve in `DX` il puntatore ad un record che si vuole rendere nuovamente disponibile.

Come tener traccia di quali record all'interno di `HEAP` sono utilizzati e di quali sono liberi? Un metodo consiste nell'organizzare tutti i record liberi in una lista (denominata *free list*), di cui si conserva nella variabile `F_HEAD` il puntatore al primo elemento.

Ogni qual volta viene attivata, la procedura **ALLOC** restituisce il puntatore al primo elemento in tale lista, aggiornando opportunamente `F_HEAD`. Ogni volta che viene chiamata, **FREE** inserisce l'elemento reso disponibile in testa alla lista. Inizialmente tutti i record in `HEAP` sono contenuti nella *free list*, e per semplicità si fa in modo che ognuno abbia come successore nella lista stessa l'elemento fisicamente successivo in `HEAP` (Fig. 15.2) poi, mano a mano che avvengono le chiamate a **ALLOC** e **FREE**, la situazione si modifica; un esempio è presentato in Fig. 15.3 dove si suppone che gli elementi non più nella *free list* appartengano ad una lista la cui testa è memorizzata in `HEAD`.

Fig. 15.2: Situazione iniziale dello *heap*.

Si noti che è necessario disporre di un valore corrispondente al puntatore nullo (il *NULL* del linguaggio C). È frequente definire come *NULL* il valore `0000`, avendo però l'accortezza di fare in modo che il primo elemento di *HEAP* non coincida con il primo byte del segmento in cui è allocato, il cui offset è appunto `0000`. Inoltre il valore `0000`, quando ritornato da *ALLOC*, va ad indicare che non vi è più memoria disponibile in *HEAP*.

Fig. 15.3: Esempio di possibile situazione dello *heap*.

Esercizio: Manipolazione di una lista.

Si propone ora un modulo completo di interfaccia utente, che permette di eseguire le seguenti operazioni:

- *inserzione* di un dato valore nella lista (comando *I*);
- *cancellazione* di un dato valore dalla lista (comando *C*);
- *visualizzazione* degli elementi nella lista (comando *V*);
- *terminazione* del programma (comando *E*).

Gli elementi della lista sono record composti di due campi, ciascuno su 2 byte:

- il campo *value*, contenente il valore da memorizzare;
- il campo *next*, corrispondente al puntatore al successivo elemento nella lista.

In questo caso quindi si ha $DIM_REC = 4$.

La lista viene gestita in modo non ordinato, inserendo gli elementi in testa alla stessa.

La soluzione proposta in linguaggio Assembler è presentata di seguito. In essa sono utilizzate le procedure INPUT ed OUTPUT presentate nel Cap. 15: la procedura INPUT legge un numero da tastiera e scrive in DX l'equivalente valore binario; la procedura OUTPUT visualizza il numero binario letto in DX. La procedura ACAPO è la stessa presentata nel Cap. 4.

```

.MODEL    large
EXTRN     INPUT:FAR, OUTPUT:FAR, ACAPO:FAR
N_ELEM    EQU     10
DIM        EQU     2*N_ELEM

.STACK
.DATA
HEAD       DW      0
F_HEAD     DW      0
HEAP       DW      DIM DUP(0)
MEM_MESS   DB      'Memoria dinamica esaurita', 0DH, 0AH, '$'
DEL_MESS   DB      'Numero elementi cancellati', 0DH, 0AH, '$'
PROMPT     DB      '>>$'

.CODE
.STARTUP
CALL       INIT                ; inizializzazione heap
MOV        HEAD, 0             ; testa lista
rd_cmd:    LEA        DX, PROMPT ; stampa il prompt
MOV        AH, 9
INT        21H
MOV        AH, 1               ; legge il comando
INT        21H
CMP        AL, 'I'             ; inserimento
JE         cmd_i
CMP        AL, 'C'             ; cancellazione
JE         cmd_c
CMP        AL, 'V'             ; visualizza
JE         cmd_v
CMP        AL, 'E'             ; fine
JE         fine
CALL       ACAPO
JMP        rd_cmd

cmd_i:     CALL       INPUT
CALL       ACAPO
CALL       INSERT
JMP        rd_cmd

cmd_v:     CALL       ACAPO
CALL       DISPLAY
JMP        rd_cmd

cmd_c:     CALL       INPUT
CALL       ACAPO
CALL       DELETE
LEA        DX, DEL_MESS
MOV        AH, 9

```

```

        INT      21H
        MOV      DX, CX                ; stampa numero elementi cancellati
        CALL     OUTPUT
        JMP      rd_cmd
fine:    .EXIT
;*****
;                      INSERT
; Inserisce l'elemento in DX nel primo slot della lista.
;*****
INSERT   PROC     NEAR
        PUSH     AX
        PUSH     BX
        CALL     ALLOC
        CMP      BX, 0                ; errore in allocazione
        JE       in_fine
        MOV      [BX], DX             ; campo dato
        MOV      AX, HEAD
        MOV      [BX]+2, AX           ; campo NEXT
        MOV      HEAD, BX            ; aggiorna HEAD
in_fine: POP      BX
        POP      AX
        RET
INSERT   ENDP
;*****
;                      DISPLAY
; Stampa il contenuto della lista.
;*****
DISPLAY  PROC     NEAR
        PUSH     BX
        PUSH     DX
        MOV      BX, HEAD
d_loop:  CMP      BX, 0
        JE       d_fine
        MOV      DX, [BX]
        CALL     OUTPUT
        MOV      BX, [BX]+2
        JMP      d_loop
d_fine:  POP      DX
        POP      BX
        RET
DISPLAY  ENDP
;*****
;                      DELETE
; Cancella dalla lista tutti gli elementi con il valore contenuto in DX.
; Ritorna in CX il numero di elementi cancellati.
;*****
DELETE   PROC     NEAR
        PUSH     SI
        PUSH     DI
        PUSH     AX
        PUSH     BX
        PUSH     DX
        XOR      CX, CX
        MOV      DI, HEAD            ; DI punta all'elemento da
                                      ; confrontare
        MOV      SI, DI              ; SI punta a quello precedente
k_loop:  CMP      DI, 0
        JE       k_fine              ; fine lista
        CMP      [DI], DX
        JE       found
        MOV      SI, DI              ; avanza al successivo
        MOV      DI, [DI]+2
        JMP      k_loop
found:   CMP      DI, HEAD
        JE       first                ; salta se l'elemento è il primo
        MOV      AX, [DI]+2          ; modifica il campo NEXT
                                      ; del precedente
        MOV      [SI]+2, AX

```

```

tutti:    MOV     BX, DI
          CALL    FREE
          MOV     DI, AX
          INC     CX
          JMP     k_loop
first:    MOV     AX, [DI]+2
          MOV     HEAD, AX
          JMP     tutti
k_fine:   POP     DX
          POP     BX
          POP     AX
          POP     DI
          POP     SI
          RET
DELETE    ENDP
;*****
;                               INIT
; Inizializza lo heap.
;*****
INIT      PROC     NEAR
          PUSH     DX
          PUSH     SI
          LEA      SI, HEAP           ; inizializza la testa della lista
          MOV      F_HEAD, SI
          MOV      CX, N_ELEM-1      ; inizializza i campi next
                                     ; della free list
          XOR      SI, SI
i_loop:   INC     SI
          INC     SI
          LEA      DX, HEAP[SI]
          INC     DX
          INC     DX
          MOV      HEAP[SI], DX
          INC     SI
          INC     SI
          LOOP    i_loop
          INC     SI                  ; ultimo elemento
          INC     SI
          MOV      HEAP[SI], 0
          POP     SI
          POP     DX
          RET
INIT      ENDP
;*****
;                               ALLOC
; Restituisce in BX il puntatore al primo elemento libero nello heap.
; Ritorna 0 in caso di errore.
;*****
ALLOC     PROC     NEAR
          PUSH     AX
          PUSH     DX
          MOV      BX, F_HEAD
          CMP      BX, 0
          JE       no_mem             ; salta se non vi è memoria
                                     ; disponibile
          MOV      DX, [BX]+2         ; aggiorna F_HEAD
          MOV      F_HEAD, DX
          XOR      AX, AX
          MOV      [BX], AX           ; azzera l'elemento restituito
          MOV      [BX]+2, AX
          JMP      a_fine
no_mem:   LEA      DX, MEM_MESS
          MOV      AH, 9
          INT      21H
a_fine:   POP     DX
          POP     AX
          RET
ALLOC     ENDP

```

```

;*****
;
;                               FREE
; Inserisce nella free list l'elemento puntato da BX.
;*****
FREE      PROC      NEAR
          PUSH      AX
          MOV       AX, F_HEAD
          MOV       [BX]+2, AX
          MOV       F_HEAD, BX
          POP       AX
          RET
FREE      ENDP
          END

```

15.4. La recursione

L'Assembler permette la recursione, che deve essere gestita dal programmatore stesso.

Nulla infatti vieta che una procedura richiami se stessa: in tal caso l'indirizzo di ritorno messo nello stack è quello della procedura stessa, e nello stack si accumuleranno tanti di questi indirizzi, quante sono state le chiamate recursive. Nel seguito vengono riportati alcuni esempi di programmi recursivi, concentrando l'attenzione sia sul passaggio di parametri sia sulla definizione di variabili locali o temporanee.

Esercizio: *Il calcolo del fattoriale.*

Si tratta del più classico dei problemi recursivi, benché la soluzione iterativa sia in questo caso sicuramente più efficiente. Il programma è composto da una procedura `FACT` e da un *main* che legge da tastiera il numero di cui si vuole calcolare il fattoriale, chiama la procedura `FACT` e visualizza il risultato.

Si noti come il programma proposto non permetta, a causa delle dimensioni del risultato, di calcolare il fattoriale di numeri maggiori di 8.

```

EXTRN     .MODEL    large
          INPUT:FAR, OUTPUT:FAR, ACAPO:FAR
          .DATA
OUT_VAL   DW        0
PROMPT    DB        'VALORE DI INPUT: $'
          .STACK
          .CODE
          .STARTUP
          MOV       DX, OFFSET PROMPT
          MOV       AH, 9
          INT       21H
          CALL      INPUT
          MOV       BX, DX
          CALL      FACT          ; mette in AX il risultato
          CALL      ACAPO
          MOV       DX, AX
          CALL      OUTPUT
          .EXIT
;*****
;                               FACT
; Calcola il fattoriale di un numero: prende il parametro di input in BX e
; lascia il risultato in AX.
;*****
FACT      PROC      NEAR
          PUSH      BX
          CMP       BX, 1
          JE        return
          DEC       BX
          CALL      FACT
          return

```

```

                INC     BX
                MUL     BX
                JMP     fine
return:         MOV     AX, 1
                XOR     DX, DX
fine:          POP     BX
                RET
FACT          ENDP
                END

```

Esercizio: Splitting.

Si vuole scrivere un programma in grado di *espandere* (*splitting*) stringhe di bit contenenti 0, 1 e X, producendo tutte le possibili stringhe ottenibili da quella data, tramite la sostituzione di ciascuna X (qui equivalente ad un *don't care*) con un 1 o uno 0.

Per fare ciò si utilizza un algoritmo recursivo basato su una procedura `SPLIT` che esegue l'espansione di una stringa, utilizzando le seguenti variabili globali:

- *ibuff*: contiene la stringa letta da tastiera;
- *obuf*: contiene la stringa in via di espansione, che alla fine verrà visualizzata;
- *curr_index*: indice corrente all'interno di *ibuff*; deve essere inizializzato a 0 dal programma chiamante.

La procedura `SPLIT` considera il primo carattere della stringa *ibuff*. Se vale 0 oppure 1, copia il carattere nella corrispondente posizione all'interno di *obuf*, e poi richiama se stessa sul successivo carattere di *ibuff*; se il carattere vale X richiama se stessa due volte, la prima volta scrivendo in *obuf* il valore 0, la seconda scrivendo il valore 1. Quando `SPLIT` viene chiamata sull'ultimo carattere di *ibuff*, esegue la visualizzazione di *obuf*. Sia *len* la lunghezza di *ibuff*. Il codice C della procedura proposta è il seguente:

```

void split(void)
{
    if (curr_index==len)
        { printf("%s\n", obuff);
          return;
        }
    else
        switch (ibuff[curr_index])
        { case '0':
          obuff[curr_index++] = '0';
          split();
          break;
          case '1':
          obuff[curr_index++] = '1';
          split();
          break;
          case 'X':
          obuff[curr_index++] = '0';
          split();
          obuff[curr_index-1] = '1';
          split();
          break;
        }
    return;
}

```

La soluzione in linguaggio Assembler che esegue la stessa sequenza di operazioni è la seguente:

```

.MODEL    small
LF        EQU    10
CR        EQU    13
DIM       EQU    30                ; dimensione massima della
                                   ; stringa da espandere

.DATA
OBUFF     DB      DIM     DUP  ('0')
IBUFF     DB      DIM     DUP  ('0')
LEN       DW      0
ERR_MESS  DB      'Carattere non ammesso$'

.CODE
.STARTUP
MOV       CX, DIM                ; lettura stringa di input
MOV       SI, 0
MOV       AH, 1
lab1:     INT      21H
MOV       IBUFF[SI], AL
INC       SI
CMP       AL, CR
LOOPNE    lab1
DEC       SI
MOV       LEN, SI
XOR       BX, BX
CALL      SPLIT
.EXIT

;*****
;                               SPLIT
; Riceve in BX il numero d'ordine del carattere da considerare all'interno
; della stringa che deve essere espansa.
;*****
SPLIT     PROC
PUSH      AX
PUSH      DX
PUSH      SI
CMP       BX, LEN                ; stringa vuota ?
JNE       ancora
MOV       CX, LEN                ; Sì: visualizza
MOV       AH, 2
XOR       SI, SI
lab2:     MOV      DL, OBUFF[SI]
INT       21H
INC       SI
LOOP      lab2
MOV       DL, CR
INT       21H
MOV       DL, LF
INT       21H
JMP       fine
ancora:   MOV      DL, IBUFF[BX]   ; No, considera il primo carattere
CMP       DL, '0'
JNE       not_z
MOV       OBUFF[BX], '0'        ; '0'
INC       BX
CALL      SPLIT
DEC       BX
JMP       fine
not_z:    CMP      DL, '1'        ; '1'
JNE       not_one
MOV       OBUFF[BX], '1'
INC       BX
CALL      SPLIT
DEC       BX
JMP       fine
not_one:  CMP      DL, 'X'        ; 'X'
JNE       error

```



```

        MOV     OBUFF[BX], '0'      ; trasforma la X in 0
        INC     BX
        CALL    SPLIT
        DEC     BX
        MOV     OBUFF[BX], '1'      ; trasforma la X in 1
        INC     BX
        CALL    SPLIT
        DEC     BX
        JMP     fine
error:   MOV     AH, 9                ; carattere diverso da 0, 1 e X
        LEA     DX, ERR_MESS
        INT     21H
fine:    POP     SI
        POP     DX
        POP     AX
        RET
SPLIT    ENDP
        END

```

15.4.1. Il passaggio di parametri

Si consideri ora il problema del passaggio dei parametri ad una procedura recursiva. La tecnica più adatta risulta quella che fa uso dello stack.

In questo caso ad ogni chiamata della procedura vengono memorizzati nello stack:

- l'indirizzo di ritorno (su 16 o 32 bit a seconda che la procedura sia NEAR o FAR);
- i parametri passati alla procedura;
- eventuali registri salvati dalla procedura stessa.

Esercizio: *Il Tour del Cavaliere*.

Si tratta di un problema classico, basato sul gioco degli scacchi. Si desidera trovare una sequenza di mosse mediante le quali un cavallo, posto inizialmente nella casella (0,0), possa toccare tutte le caselle della scacchiera senza ripassare mai due volte dalla stessa. La soluzione è basata sulla procedura recursiva MUOVI che determina se, data una certa casella occupata con la mossa *n-esima*, esiste una soluzione compatibile con tale mossa. La procedura ha tre parametri:

- il numero della mossa corrente;
- la prima coordinata della casella corrispondente all'ultima mossa fatta;
- la seconda coordinata della stessa casella.

I tre parametri vengono passati utilizzando lo stack. In esso è anche ricavato lo spazio per una variabile locale necessaria per il calcolo degli offset all'interno della matrice SCACC, realizzata tramite un vettore. In Fig. 15.4 vengono riportate le 8 possibili caselle raggiungibili da un cavallo posto nella casella centrale, numerate secondo l'ordine con il quale vengono considerate dalla procedura MUOVI. Per calcolare la nuova casella raggiunta in conseguenza dell'*i-esimo* tentativo, tale procedura fa uso di due vettori A e B di 8 elementi ciascuno; essi sono costruiti in modo tale che sommando alle due coordinate della casella corrente il valore contenuto nel loro *i-esimo* elemento si ottengono le coordinate della nuova casella.

Se si desidera provare il programma, è conveniente usare una scacchiera di dimensioni inferiori alle classiche 8×8 , in quanto in tal caso la ricerca della soluzione comporta un elevato tempo di calcolo. Nell'esempio si è usata una scacchiera 5×5 .

		3		2		
	4				1	
	5				8	
		6		7		

Fig. 15.4: Ordine in cui vengono considerate le possibili mosse di un cavallo.

La soluzione proposta in linguaggio C è la seguente:

```

#define DIM 5
void muovi (int, int, int);
void display_scac(void);
int a[8], b[8], scacc[DIM][DIM];
main()
{
    int i, j, result;
    a[0] = 2; a[1] = 1; a[2] = -1; a[3] = -2;
    a[4] = -2; a[5] = -1; a[6] = 1; a[7] = 2;
    b[0] = 1; b[1] = 2; b[2] = 2; b[3] = 1;
    b[4] = -1; b[5] = -2; b[6] = -2; b[7] = -1;
    for (i=0; i<DIM; i++)
        for (j=0; j<DIM; j++)
            scacc[i][j] = 0;
    scacc[0][0] = 1;
    for (i=0; i<DIM; i++)
        for (j=0; j<DIM; j++)
            muovi(2, i, j);
}
void muovi (int mossa, int posx, int posy)
{
    int i, newposx, newposy;
    if (mossa == (DIM*DIM+1))
    {
        display_scac();
        return;
    }
    for (i=0; i<8; i++)
    {
        newposx = posx + a[i];
        newposy = posy + b[i];
        if ((newposx<DIM) && (newposx>=0) && (newposy<DIM) && (newposy>=0))
        {
            if (scacc[newposx][newposy] == 0)
            {
                scacc[newposx][newposy] = mossa;
                muovi (mossa+1, newposx, newposy);
                scacc[newposx][newposy] = 0;
            }
        }
    }
}

```

```

void display_scacc(void)
{
    int i, j;
    printf("\n");
    for (i=0; i<DIM; i++)
    { for (j=0; j<DIM; j++) printf("%2d ", scacc[i][j]);
      printf("\n");
    }
}

```

Il programma Assembler che esegue la stessa sequenza di operazioni logiche è il seguente:

```

        .MODEL    large
EXTRN   INPUT:FAR, OUTPUT:FAR
DIM     EQU      5
        .STACK
        .DATA
SCACC   DB        DIM*DIM DUP (0)
A       DW        2,1,-1,-2,-2,-1,1,2
B       DW        1,2,2,1,-1,-2,-2,-1
        .CODE
        .STARTUP
        MOV       SCACC[0], 1
        MOV       AX, 0
        PUSH      AX
        PUSH      AX
        MOV       AX, 2
        PUSH      AX
        CALL      MUOVI
        ADD       SP, 6
        CMP       AX, 1
        JNE       fine
        MOV       SI, 0
        MOV       CX, DIM*DIM
        MOV       DX, 0
        MOV       AX, 10
loop1:  MOV       DL, SCACC[SI]
        CALL      OUTPUT
        INC       SI
        DEC       AX
        JNE       avanti
        CALL      ACAPO
        MOV       AX, 10
avanti:  LOOP      loop1
fine:    .EXIT
;*****
;
;                      MUOVI
; Esegue una mossa del cavallo a partire dalla locazione passata come
; parametro. Se la nuova mossa porta in una casella libera ed appartenente
; alla scacchiera la occupa e richiama se stessa, altrimenti prova una nuova
; mossa. Se nessuna delle mosse possibili ha successo ritorna 0, altrimenti
; ritorna 1.
;*****
MUOVI   PROC      NEAR
        PUSH      BP
        MOV       BP, SP
        SUB       SP, 2
        PUSH      BX
        PUSH      CX
        PUSH      DX
        PUSH      SI
        PUSH      DI
        MOV       CX, [BP+4]
        CMP       CX, DIM*DIM+1
        JB        cont
        MOV       AX, 1

```

```

cont:      JMP      mfine
mloop:     MOV      DI, 0                ; DI contiene i
           MOV      BX, [BP+6]          ; BX contiene x
           MOV      SI, [BP+8]          ; SI contiene y
           ADD      BX, A[DI]           ; calcola newx
           JS       nogood
           ADD      SI, B[DI]           ; calcola newy
           JS       nogood
           CMP      BX, DIM
           JAE      nogood
           CMP      SI, DIM
           JAE      nogood
           MOV      [BP-2], BX          ; calcola gli offset in SCACC
           MOV      AX, DIM
           MUL      BX
           MOV      BX, AX
           ADD      BX, SI
           CMP      SCACC[BX], 0        ; casella libera ?
           JNE      nogood
good:      MOV      SCACC[BX], CL        ; Si: occupa la casella
           INC      CX
           PUSH     SI
           PUSH     [BP-2]
           PUSH     CX
           CALL     MUOVI
           ADD      SP, 6
           CMP      AX, 0
           JNE      mfine
           DEC      CX                  ; nessuna soluzione possibile
           MOV      SCACC[BX], 0        ; rilascia la casella
nogood:    INC      DI                  ; tenta un'altra mossa
           INC      DI
           CMP      DI, 16
           JNE      mloop
           MOV      AX, 0
mfine:     POP      DI
           POP      SI
           POP      DX
           POP      CX
           POP      BX
           ADD      SP, 2
           MOV      SP, BP
           POP      BP
           RET
MUOVI     ENDP
END

```


16. Esercizi svolti

In questo capitolo vengono riportati una serie di esercizi di programmazione, accompagnati ciascuno da una possibile soluzione.

16.1. Calcolo del numero di combinazioni semplici di elementi di un insieme

Nel calcolo combinatorio si definisce *combinazione semplice (senza ripetizioni)* una presentazione di elementi di un insieme nella quale non ha importanza l'ordine dei componenti e non si può ripetere lo stesso elemento più volte. Dati n elementi distinti e un numero intero positivo $k \leq n$, il numero di combinazioni semplici possibili $C(n, k)$ è dato dalla seguente formula:

$$C(n, k) = \binom{n}{k} = \frac{n \cdot (n-1) \cdot (n-2) \dots (n-k+1)}{k!}$$

Si scriva una procedura COMBINA in grado di calcolare il numero di combinazioni semplici dati i parametri n e k ricevuti come variabili globali di tipo *byte*. Il risultato dovrà essere restituito attraverso la variabile globale di tipo *word* risultato.

Sia lecito supporre che durante le operazioni intermedie non si presenti *overflow*.

16.1.1. Codice

```
.MODEL small
.STACK
.DATA
n      DB 6
k      DB 3
risultato DW ?
.CODE
.STARTUP
CALL COMBINA
.EXIT
;*****
;                                COMBINA
; Procedura per il calcolo del numero di combinazioni semplici di elementi
; di un insieme
; Dati in ingresso: n, k - variabili globali (byte)
; Risultati: risultato - variabile globale (word)
;*****
COMBINA PROC
    MOV CL, k
    XOR CH, CH
    DEC CX
    MOV AL, n
    XOR AH, AH
    MOV BL, n
    XOR BH, BH
    DEC BX
ciclo1:  MUL BX
    DEC BX
    LOOP ciclo1
    MOV BL, k
    MOV CL, k
    XOR CH, CH
    DEC CX
ciclo2:  DIV BL
    SUB BL, 1
    LOOP ciclo2
    MOV risultato, AX
```

```
RET
ENDP COMBINA
END
```

16.2. Riconoscimento degli anni bisestili

Si abbia un vettore contenente alcuni interi rappresentanti anni passati ($0 \div 2011$). Si scriva una procedura che sia in grado di determinare se tali anni sono bisestili. Si ricorda che un anno è bisestile se il suo numero è divisibile per 4, con l'eccezione che gli anni secolari (quelli divisibili per 100) sono bisestili solo se divisibili anche per 400.

In altre parole, in forma di pseudocodice il programma richiesto si può esprimere come segue:

```
IF (anno divisibile per 100)
{ IF (anno divisibile per 400)
  Anno_bisestile = TRUE
  ELSE Anno_bisestile = FALSE
}
ELSE
{ IF (anno divisibile per 4)
  Anno_bisestile = TRUE
  ELSE Anno_bisestile = FALSE
}
```

La procedura deve ricevere come input:

- tramite il registro SI l'offset di un vettore di *word* contenente gli anni da valutare
- tramite il registro DI l'offset di un vettore di *byte* della stessa lunghezza, che dovrà contenere, al termine dell'esecuzione della procedura, nelle posizioni corrispondenti agli anni espressi nell'altro vettore, il valore 1 se l'anno è bisestile oppure 0 nel caso opposto
- tramite il registro BX la lunghezza di tali vettori.

Esempio:

```
anni:      1945, 2008, 1800, 2006, 1748, 1600
risultato: 0,    1,    0,    0,    1,    1
lunghezza: 6
```

16.2.1. Codice

```
LUNG      EQU 6
          .MODEL small
          .STACK
          .DATA
anni       DW 1945, 2008, 1800, 2006, 1748, 1600
ris        DB LUNG DUP (?)
          .CODE
          .STARTUP
          LEA SI, anni
          LEA DI, ris
          MOV BX, LUNG
          CALL BIESTILE
          .EXIT

;*****
;          BIESTILE
; Procedura per il riconoscimento di anni bisestili
; (l'algoritmo è stato ottimizzato per ridurre il numero di divisioni
; potenzialmente lente da eseguire)
; Dati in ingresso: SI - offset del vettore di word rappresentante anni
;                   DI - offset del vettore di byte per risultato
;                   BX - lunghezza vettori
;*****
BIESTILE PROC
          PUSH AX
          PUSH BX
```

```

        PUSH CX
        PUSH SI
        PUSH DI
ciclo:  MOV [DI], 0
        MOV AX, [SI]
        MOV CL, 100      ; determinazione anno secolare
        DIV CL
        CMP AH, 0
        JNZ non_sec
        TEST AX, 3        ; verifica divisibilità per 400
        JNZ next
        MOV [DI], 1
        JMP next
non_sec: TEST AH, 3        ; anno non secolare: verifica divisibilità per 4
        JNZ next
        MOV [DI], 1
next:   ADD SI, 2
        INC DI
        DEC BX
        JNZ ciclo
        POP DI
        POP SI
        POP CX
        POP BX
        POP AX
        RET
BISESTILE ENDP
END

```

16.3. Generazione di segnali di *clock*

Sia dato un sistema elettronico regolato da 3 segnali di clock con frequenze indipendenti tra di loro, e inizialmente in fase, come in Fig. 16.1.

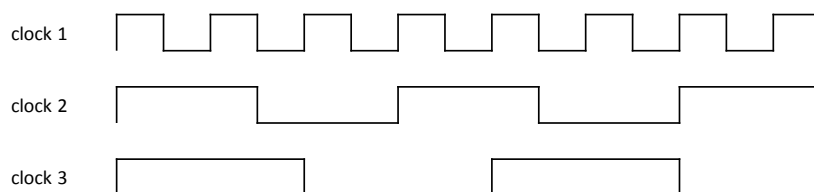


Fig. 16.1: Esempio di segnali da generare.

Si scriva una procedura `CLOCKS` in grado di generare tre vettori di `LUNG` elementi che descrivano gli andamenti dei tre segnali di clock nel tempo. La procedura riceve nei registri `AH`, `BH` e `DH`, rispettivamente, i 3 periodi di clock (che supponiamo corrispondano a numeri pari), e salva nei vettori di byte (opportunamente dichiarati) `clock1`, `clock2` e `clock3`, di lunghezza `LUNG`, l'evoluzione dei tre segnali.

NB: La procedura deve essere chiamata una volta sola, e all'interno della procedura si utilizza un solo ciclo.

- Esempio:

```

LUNG = 20, AH = 2, BH = 6, CH = 8
clock1 = 10101010101010101010
clock2 = 11100011100011100011
clock3 = 11110000111100001111

```


16.3.1. Codice

```

LUNG      EQU 20
          .MODEL small
          .STACK
          .DATA

clock1    DB LUNG DUP (?)
clock2    DB LUNG DUP (?)
clock3    DB LUNG DUP (?)
frequencies DB 2, 6, 8

          .CODE
          .STARTUP
          MOV AH, frequencies[0]
          MOV BH, frequencies[1]
          MOV DH, frequencies[2]
          CALL CLOCKS
          .EXIT

;*****
;                                CLOCKS
; Procedura per la generazione di segnali di clock
; Dati in ingresso: AH - periodo primo clock
;                   BH - periodo secondo clock
;                   DH - periodo terzo clock
; Risultati: clock1, clock2, clock3 - vettori globali di LUNG byte
;*****
CLOCKS    PROC
          XOR AL, AL
          XOR BL, BL
          XOR DL, DL
          SHR AH, 1
          SHR BH, 1
          SHR DH, 1
          MOV CX, LUNG-1
          XOR SI, SI
          MOV BYTE PTR clock1[SI], 1
          MOV BYTE PTR clock2[SI], 1
          MOV BYTE PTR clock3[SI], 1
mio_ciclo: PUSH CX
          INC AL
          INC BL
          INC DL
          MOV CL, clock1[SI]
          CMP AL, AH
          JNE next1
          NOT CL
          AND CL, 1
          XOR AL, AL
next1:    INC SI
          MOV clock1[SI], CL
          DEC SI
          MOV CL, clock2[SI]
          CMP BL, BH
          JNE next2
          NOT CL
          AND CL, 1
          XOR BL, BL
next2:    INC SI
          MOV clock2[SI], CL
          DEC SI
          MOV CL, clock3[SI]
          CMP DL, DH
          JNE next3
          NOT CL
          AND CL, 1
          XOR DL, DL
next3:    INC SI
          MOV clock3[SI], CL
          POP CX

```

```

        LOOP mio_ciclo
        RET
CLOCKS  ENDP
        END

```

16.4. Calcolo dei prezzi scontati

Dati in memoria i seguenti due vettori di 50 word ciascuno:

- `prezzi` rappresentante i prezzi di 50 articoli venduti in un negozio
- `scontati` inizialmente di contenuto indeterminato,

si scriva una procedura in grado di calcolare il prezzo scontato di ciascun articolo e salvarlo nel corrispondente elemento del vettore `scontati`. La procedura deve leggere da una variabile intera di tipo word denominata `sconto` l'ammontare dello sconto percentuale da applicare. Si esegua un arrotondamento alla cifra superiore se la parte decimale del prezzo risultante è maggiore o uguale a 0,5.

Inoltre, la procedura deve salvare in una variabile di tipo word `totsconto` l'ammontare totale delle riduzioni effettuate.

Esempio:

```

prezzi: 39, 1880, 2394, 1000, 1590
sconto: 30
scontati: 27, 1316, 1676, 700, 1113
totsconto: 2071

```

16.4.1. Codice

```

DIM      EQU 5
        .MODEL small
        .STACK
        .DATA
prezzi    DW 39, 1880, 2394, 1000, 1590
scontati  DW DIM DUP (?)
sconto    DW 30
totsconto DW ?
        .CODE
        .STARTUP
        CALL SCNTI
        .EXIT
;*****
;                               SCNTI
; Procedura per il calcolo di prezzi scontati
; Dati in ingresso: prezzi - vettore globale di DIM word
;                  sconto - variabile globale word
; Risultati: scontati - vettore globale di DIM word
;            totsconto - variabile globale word
;*****
SCNTI    PROC
        MOV totsconto, 0
        MOV CX, DIM    ; contatore elementi
        MOV SI, 0      ; indice prezzi
        MOV BX, 100
ciclo:   MOV AX, prezzi[SI]
        SUB BX, sconto ; calcolo frazione prezzo
        MUL BX          ; calcolo percentuale
        MOV BX, 100
        DIV BX
        CMP DX, 50     ; arrotondamento
        JB next
        ADD AX, 1
next:    MOV scontati[SI], AX
        MOV DX, prezzi[SI]

```

```

                SUB DX, AX
                ADD totsconto, DX
                ADD SI, 2
                LOOP ciclo
                RET
SCONTI         ENDP
                END

```

16.5. Calcolo del valore di un insieme di monete

Si scriva una procedura **CALCOLA** in grado di calcolare il valore di un insieme di monete di diverso valore (espresso in centesimi di Euro). La procedura deve ricevere come input nei registri **SI** e **DI**, rispettivamente, gli **indirizzi** dei seguenti vettori:

- **valore**, vettore di word indicante il valore di ciascun tipo di moneta
- **monete**, vettore di byte indicante il numero di monete di ciascun tipo.

Ad esempio, con

```
valore dw 1, 2, 5, 10, 20, 50, 100, 200
```

```
monete db 100, 23, 17, 0, 79, 48, 170, 211
```

si hanno 100 monete da 1 centesimo, 23 monete da 2 centesimi, e così via.

La procedura deve fornire il risultato aggiornando due variabili precedentemente dichiarate, di tipo **word**, denominate **euro** e **cent**, e rappresentanti rispettivamente l'importo in euro e in centesimi. Nell'esempio, il valore risultante è pari a 63411 centesimi, quindi le due variabili varranno rispettivamente 634 e 11.

16.5.1. Codice

```

LUNG          EQU 8
              .MODEL small
              .STACK
              .DATA
valore        DW 1, 2, 5, 10, 20, 50, 100, 200
monete        DB 100, 23, 17, 0, 79, 48, 170, 211
euro          DW ?
cent          DW ?
              .CODE
              .STARTUP
              LEA SI, valore
              LEA DI, monete
              CALL CALCOLA
              .EXIT
;*****
;                                CALCOLA
; Procedura per il calcolo del valore di un insieme di monete
; Dati in ingresso: SI - offset del vettore di word valore
;                   DI - offset del vettore di byte monete
; Risultati: euro - variabile globale word
;            cent - variabile globale word
;*****
CALCOLA PROC
    PUSH AX
    PUSH BX
    PUSH CX
    PUSH DX
    MOV CX, LUNG
    PUSH CX
    PUSH 0
    PUSH 0
CICLO:      POP BX

```

```

        POP CX
        MOV AL, [DI]
        XOR AH, AH
        MUL WORD PTR [SI]
        ADD AX, BX
        ADC DX, CX
        ADD SI, 2
        INC DI
        POP CX
        DEC CX
        PUSH CX
        PUSH DX
        PUSH AX
        JNZ CICLO
        POP AX
        POP DX
        POP CX
        MOV CX, 100
        DIV CX
        MOV euro, AX
        MOV cent, DX
        POP DX
        POP CX
        POP BX
        POP AX
        RET
CALCOLA ENDP
        END

```

16.6. Media mobile

Si scriva una procedura `MEDIAM` in grado di effettuare il calcolo della media mobile semplice su un insieme di dati. La media mobile semplice (SMA) è così definita:

$$SMA = (C_i + C_{i+1} + C_{i+2} + \dots + C_{i+n-1})/n$$

dove C_i è il primo valore preso in considerazione e C_n quello più recente.

Siano dati:

- un vettore di *word* `vett1` contenente `DIM` elementi interi positivi (`DIM` è dichiarato come costante), che rappresenta l'insieme di dati da analizzare
- un vettore di *word* `vett2` della stessa dimensione, nelle prime locazioni del quale dovranno essere scritti i risultati (troncati all'unità).
- una variabile di tipo *byte* `num` che rappresenta il numero di valori su cui calcolare di volta in volta la media (n nell'equazione).

Si supponga di avere tutte le variabili già inizializzate in memoria, in modo da poterle utilizzare come variabili globali.

Esempio:

```

DIM = 6
num = 3
vett1    4,      6,      7,      12,     99,     192
vett2    5,      8,     39,    101,      0,       0

```

16.6.1. Codice

```

DIM      EQU 6
.MODEL small

```

```

        .STACK
        .DATA
vet1    DW 4, 6, 7, 12, 99, 192
vet2    DW DIM DUP(?)
num     DB 4
        .CODE
        .STARTUP
        CALL MEDIAM
        .EXIT
;*****
;
;               MEDIAM
; Procedura per il calcolo della media mobile
; Dati in ingresso: vet1 - vettore globale di DIM word
;                   num - variabile globale word
; Risultati: vet2 - variabile globale di DIM word
;*****
MEDIAM PROC
        XOR SI, SI
        MOV CX, DIM
        MOV BL, num
        XOR BH, BH
        SUB CX, BX
        INC CX
ciclo1: PUSH CX
        MOV CX, BX
        MOV DI, SI
        XOR AX, AX
        XOR DX, DX
ciclo2: ADD AX, vet1[DI]
        ADC DX, 0
        ADD DI, 2
        LOOP ciclo2
        DIV BX
        MOV vet2[SI], AX
        ADD SI, 2
        POP CX
        LOOP ciclo1
        RET
MEDIAM ENDP
END

```

16.7. Identificazione di numeri primi

Si scriva una procedura `PRIMI` in grado di determinare se ciascuno dei numeri naturali (≥ 2) contenuti in un vettore è primo oppure no. Si ricorda che un numero è primo quando è divisibile solamente per 1 e per se stesso. Siano dati:

- un vettore di *byte* `numeri` contenente `DIM` elementi (`DIM` è dichiarato come costante)
- un vettore di *byte* `risultato` della stessa dimensione che dovrà contenere, per ogni numero analizzato, un valore logico 1 se il numero nella stessa posizione è primo e 0 se non lo è. Tale vettore sarà modificato dalla procedura.

Si supponga che il programma chiamante lanci la procedura una volta sola, passando gli indirizzi iniziali dei vettori `numeri` e `risultato`, rispettivamente, attraverso i registri `SI` e `DI`.

Esempio:

DIM EQU 6						
numeri	2,	15,	36,	37,	20,	97
risultato	1,	0,	0,	1,	0,	1

16.7.1. Codice

```

DIM      EQU 6
         .MODEL small
         .STACK
         .DATA
numeri    DB 2, 15, 36, 37, 20, 97
ris       DB DIM DUP(?)
         .CODE
         .STARTUP
         LEA SI, numeri
         LEA DI, ris
         CALL PRIMI
         .EXIT
;*****
;                               PRIMI
; Procedura per la determinazione di numeri primi
; Dati in ingresso: SI - offset di un vettore di DIM byte
;                               DI - offset di un vettore di DIM byte (per risultati)
;*****
PRIMI     PROC
          MOV CX, DIM
ciclo1:   PUSH CX
          MOV BL, [SI]
          CMP BL, 2
          JBE primo           ; 2 primo per ipotesi
          MOV CL, BL
          XOR CH, CH
          DEC CX
          MOV BYTE PTR [DI], 0
ciclo2:   MOV AL, BL
          XOR AH, AH
          DIV CL
          CMP AH, 0
          JE  next
          DEC CX
          CMP CX, 1
          JNE CICLO2
primo:    MOV BYTE PTR [DI], 1
next:     INC SI
          INC DI
          POP CX
          LOOP ciclo1
          RET
PRIMI     ENDP
         END

```

16.8. Gestione di un magazzino di tessuti

Un magazzino di tessuti contiene pezze rettangolari di varie misure. Quando un cliente ordina una pezza di una specifica misura (larghezza, altezza), occorre determinare se è possibile soddisfare tale richiesta, verificando se tra le pezze disponibili ve ne sia almeno una di dimensione sufficiente. Le pezze non devono essere cucite insieme, ma possono essere ruotate di 90°. Al cliente deve essere fornita la pezza più piccola in grado di soddisfare la richiesta.

Sia data una matrice di *byte* (istanziata in memoria come variabile globale) contenente le misure (larghezza, altezza) di ciascuna pezza (considerare DIM pezze). Si scriva una procedura CERCA in grado di ricercare la pezza di area minima tale da soddisfare la richiesta del cliente.

Il programma chiamante deve passare le dimensioni richieste dal cliente attraverso i registri AL e AH, e riceve dalla procedura il numero della pezza scelta attraverso il registro DX. Nel caso in cui non sia possibile soddisfare la richiesta, DX dovrà contenere il valore esadecimale FFFF.

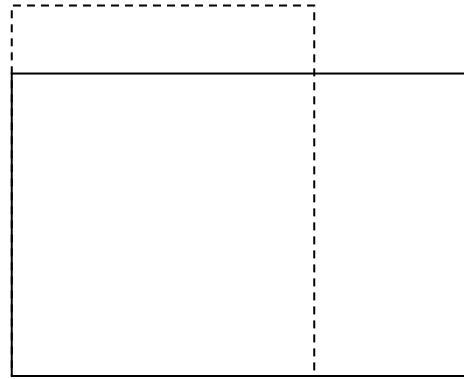
Esempio:

```

DIM EQU 5
pezze db 4, 4   richiesta: 4,5
       db 6, 7
       db 9, 5
       db 6, 4
       db 3, 6

```

Pezza scelta: 3



16.8.1. Codice

```

DIM EQU 5
.MODEL small
.STACK
.DATA
pezze DB 5, 4
      DB 3, 6
      DB 9, 2
      DB 4, 4
      DB 6, 7
ris DW ?
.CODE
.STARTUP
MOV AL, 4
MOV AH, 4
CALL CERCA
MOV ris, DX
.EXIT
;*****
;                                CERCA
; ; Procedura per la ricerca della pezza di dimensioni minime soddisfacente
; ; i requisiti.
; ; Dati in ingresso: AL - larghezza richiesta
; ;                                AH - lunghezza richiesta
; ;                                pezze - matrice di byte (DIM righe, 2 colonne)
; ; Risultati: DX - indice risultato (FFFF se richiesta non soddisfabile)
;*****
CERCA PROC
    PUSH AX
    PUSH BX
    PUSH CX
    PUSH DI
    MOV CX, DIM
    MOV BX, 0
    MOV DX, 0FFFFH ; inizializzazione minimo a massimo valore possibile
    MOV DI, -1 ; indice minimo
ciclo: CMP AL, pezze[BX][0]
    JA test2
    CMP AH, pezze[BX][1]
    JA test2
    JMP bene
test2: CMP AL, pezze[BX][1]
    JA next
    CMP AH, pezze[BX][0]
    JA next
bene:  PUSH AX
    MOV AL, pezze[BX][0]
    MUL pezze[BX][1] ; calcolo area pezza
    CMP AX, DX
    JA next1
    MOV DX, AX
    MOV DI, BX
next1: POP AX

```

```

next: ADD BX, 2
      LOOP ciclo
      CMP DI, 0
      JL fine
      SHR DI, 1
      MOV DX, DI
fine: POP DI
      POP CX
      POP BX
      POP AX
      RET
CERCA ENDP
END

```

16.9. Filtro per indirizzi IP

Un *indirizzo IP* è un numero che identifica univocamente un dispositivo collegato a una rete che utilizza *Internet Protocol* come protocollo di comunicazione. L'*Internet Protocol version 4* (IPv4) prevede che l'indirizzo sia costituito da 32 bit (4 byte) suddivisi in 4 gruppi da 8 bit (1 byte), separati ciascuno da un punto. Ciascuno di questi 4 byte è poi convertito in formato decimale di più facile interpretazione. Un esempio di indirizzo IPv4 è 130.192.182.133, che corrisponde a 82C0B685h.

Si scriva una procedura `FILTRO` in grado di elaborare una sequenza di indirizzi IPv4 e contare quanti di essi soddisfino la seguente condizione: l'indirizzo deve essere confrontato bit a bit con un *riferimento* dato, ma nel confronto devono essere considerati soltanto i bit nelle posizioni che, in una variabile *doubleword maschera*, hanno valore corrispondente a '1'. Se i bit confrontati corrispondono, la condizione è soddisfatta. Viceversa, i bit nelle posizioni corrispondenti a valori '0' nella maschera non devono essere considerati per il confronto.

Esempio:

Riferimento	82C0B685h	10000010.11000000.10110110.10000101
Maschera	FFFC0000h	11111111.11111100.00000000.00000000
Indirizzo 1	82C028D1h	10000010.11000000.00101000.11010001
→	soddisfa requisiti	
Indirizzo 2	81C0276Ah	10000001.11000000.00100111.01101010
→	non soddisfa requisiti.	

Sia dato in memoria un vettore `address` di *doubleword* contenente la sequenza di indirizzi IPv4 (la dimensione del vettore è pari a `DIM`, assegnata come costante). Sia data inoltre una variabile *doubleword* `mask` contenente la maschera. La procedura riceve come parametro l'indirizzo di riferimento attraverso lo *stack*, e restituisce il numero di elementi che soddisfano la condizione sempre attraverso lo *stack*.

Esempio di programma chiamante:

```

push 82C0h      ; parte alta di indirizzo di riferimento
push B685h      ; parte bassa di indirizzo di riferimento
sub SP, 2       ; spazio riservato per risultato
call filtro
pop AX          ; prelevamento risultato da stack
add SP, 4

```

16.9.1. Codice

```

DIM      EQU 8
.MODEL small
.STACK
.DATA
address DD 82C0051AH, 0C0A80A01H, 4A7D276AH, 0D5FE1150H

```



```

mask      DD 0C7EF88C8H, 82C0B621H, 82C0A4F5H, 0ADC01874H
          DD 0FFFF0000H
          .CODE
          .STARTUP
          PUSH 82C0H
          PUSH 276AH
          SUB SP, 2
          CALL FILTRO
          POP AX
          ADD SP, 4
          .EXIT
;*****
;                               FILTRO
; Procedura per il filtraggio di indirizzi IPv4
; Dati in ingresso: address - vettore globale di DIM doubleword
;                   mask - variabile globale doubleword
;                   indirizzo di riferimento tramite stack (doubleword)
; Risultati: numero di indirizzi che soddisfano condizione (word)
;*****
FILTRO    PROC
          MOV BP, SP
          XOR SI, SI
          XOR BX, BX
          MOV CX, DIM
ciclo:    MOV AX, WORD PTR address[SI]
          XOR AX, [BP+4]
          AND AX, WORD PTR mask
          JNZ next
          MOV AX, WORD PTR address[SI+2]
          XOR AX, [BP+6]
          AND AX, WORD PTR mask[2]
          JNZ next
          INC BX
next:     ADD SI, 4
          LOOP ciclo
          MOV [BP+2], BX
          RET
          ENDP FILTRO
          END

```

16.10. Classificazione di caratteri

Dato un vettore di caratteri in memoria di dimensione pari alla costante DIM, si scriva una procedura CLASSIFICA che l'analizzi ed effettui le seguenti operazioni:

1. Copi tutti i caratteri alfabetici minuscoli trovati, nello stesso ordine, in un altro vettore di caratteri di dimensione DIM;
2. Converta ciascuna sequenza di cifre consecutive in numero intero positivo e copi di volta in volta gli interi trovati in un vettore di *word* avente un numero di elementi pari a DIM;
3. Trascuri tutti gli altri caratteri (non alfabetici, alfabetici maiuscoli)
4. Restituisca in AX il numero di caratteri alfabetici minuscoli e in BX il numero di interi trovati.

I vettori su cui la procedura lavora corrispondono a variabili globali. Valga l'ipotesi che le sequenze di cifre non eccedano mai la rappresentazione su *word*.

Esempio:

```

DIM EQU 18
vettore  db "ciao__100come3stai?"
lettere  db DIM DUP (?)
numeri    dw DIM DUP (?)

```

La procedura dovrà fornire:

```

lettere:  "ciaocomestai"
numeri:   100, 3
AX = 12
BX = 2.

```

16.10.1. Codice

```

DIM      EQU 18
         .MODEL small
         .STACK
         .DATA
vettore  DB "ciao__100come3stai?"
lettere  DB DIM DUP (?)
numeri   DW DIM DUP (?)
         .CODE
         .STARTUP
         CALL CLASSIFICA
         .EXIT
;*****
;                                CLASSIFICA
; Procedura per la classificazione di caratteri
; Dati in ingresso: vettore - vettore globale di DIM byte
; Risultati: lettere - vettore globale di DIM byte
;             numeri - vettore globale di DIM word
;             AX - numero di caratteri minuscoli trovati
;             BX - numero di interi trovati
;*****
CLASSIFICA PROC
    XOR SI, SI      ; indice lettere
    XOR DI, DI      ; indice numeri
    XOR DX, DX      ; DX è flag per numero trovato
    LEA BX, vettore ; offset vettore
    MOV CX, DIM
ciclo:   CMP [BX], 'a'
        JB next
        CMP [BX], 'z'
        JA next
        MOV DL, [BX] ; è una lettera
        MOV lettere[SI], DL
        INC SI
        JMP verifica
next:    CMP [BX], '0'
        JB altro
        CMP [BX], '9'
        JA altro
        MOV DH, 1    ; è una cifra
        PUSH CX
        PUSH DX
        MOV CX, 10
        MUL CX
        MOV DL, [BX]
        SUB DL, '0'
        XOR DH, DH
        ADD AX, DX
        POP DX
        POP CX
        JMP incrementa
verifica: CMP DH, 1    ; se l'ultimo elemento letto era una cifra...
        JNE incrementa
        MOV numeri[DI], AX ; ...costruisco l'intero
        ADD DI, 2
        XOR DX, DX
        XOR AX, AX
        JMP incrementa
altro:   CMP DH, 1
        JE  verifica
incrementa: ADD BX, 1

```

```

                LOOP ciclo
                CMP DH, 1
                JNE fine
                MOV numeri[DI], AX
                ADD DI, 2
fine:           MOV AX, SI
                MOV BX, DI
                SHR BX, 1
                RET
CLASSIFICA     ENDP
                END

```

16.11. Allineamento di byte

Siano dati:

- un vettore di *byte* `vet1` contenente `DIM` elementi (`DIM` è dichiarato come costante)
- un vettore di *word* `vet2`, della stessa dimensione, non inizializzato.

Si scriva una procedura `ALLINEA` in grado di ottenere, a partire dai dati espressi come *byte* in `vet1`, una sequenza di valori su 12 bit componendo *nibble* (insiemi di 4 bit) di dati consecutivi, come esemplificato di seguito:

dati	risultati
0010 1101	
0100 0010	0000 0010 1101 0100
0100 1011	0000 0010 0100 1011
1000 0001	
0110 0011	0000 1000 0001 0110
1100 0000	0000 0011 1100 0000
1111 1111	
0000 1011	0000 1111 1111 0000

La procedura deve memorizzare i risultati ottenuti in `vet2` (azzerando il *nibble* più significativo), procedendo fino a quando sono disponibili dati su `vet1` sufficienti a comporre un risultato; deve inoltre restituire al programma chiamante il numero di risultati memorizzati attraverso il registro `DI`. Si supponga che il programma chiamante lanci la procedura una volta sola. Si utilizzino variabili globali per l'indirizzamento dei vettori.

16.11.1. Codice

```

DIM EQU 11
.MODEL small
.STACK
.DATA
vet1 DB 45, 66, 74, 129, 99, 192, 255, 11, 98, 230, 187
vet2 DW DIM DUP(?)
.CODE
.STARTUP
CALL ALLINEA
.EXIT
;*****
;                               ALLINEA
; Procedura per l'allineamento di byte
; Dati in ingresso: vet1 - vettore globale di DIM byte
; Risultati: vet2 - vettore globale di DIM word
;                DI - numero di risultati trovati
;*****
ALLINEA PROC
    XOR SI, SI

```

```

        XOR DI, DI
        MOV CL, 4
        MOV CH, 0
ciclo:  MOV AH, vet1[SI]
        INC SI
        CMP SI, DIM
        JE fine
        MOV AL, vet1[SI]
        MOV BH, AL
        SHR AX, CL
        MOV vet2[DI], AX
        ADD DI, 2
        INC SI
        CMP SI, DIM
        JE fine
        MOV BL, vet1[SI]
        AND BX, 0FFFH
        MOV vet2[DI], BX
        INC SI
        ADD DI, 2
        CMP SI, DIM
        JNZ ciclo
fine:   SHR DI, 1
        RET
ALLINEA ENDP
        END

```

16.12. Verifica della monotonia di una sequenza di interi

Data una sequenza di interi con segno, rappresentati come *word* in memoria, si scriva una procedura *monotono* in grado di determinare la posizione della più lunga sottosequenza non-decrescente nel vettore e il numero di elementi che la compongono.

Il vettore su cui la procedura lavora corrisponde a una variabile globale; la procedura deve restituire nel registro AX il numero di elementi della sottosequenza e nel registro BX l'indice del primo elemento di tale sottosequenza.

Si assuma che sia stata definita una costante DIM pari alla dimensione del vettore.

Si lavori inoltre nell'ipotesi per cui esista una singola sottosequenza della dimensione massima.

Esempio:

```
vett    dw 15, 64, 9, 2, 4, 5, 9, 1, 294, 52, -4, 5
```

La procedura dovrà fornire (si assuma che gli elementi del vettore abbiano indice variabile tra 0 e DIM-1):

AX = 4

BX = 3.

16.12.1. Codice

```

DIM     EQU 12
        .MODEL small
        .STACK
        .DATA
vet      DW 15, 64, 9, 2, 4, 5, 9, 1, 294, 52, -4, 5
        .code
        .startup
        CALL MONOTONO
        .exit

```

```

;*****
;                                MONOTONO
; Procedura per la determinazione della più lunga sottosequenza
; non-decrescente di interi in un vettore
; Dati in ingresso: vet - vettore globale di DIM word
; Risultati: AX - numero di elementi nella sottosequenza trovata
;            BX - indice del primo elemento della sottosequenza trovata
;*****
MONOTONO PROC
    PUSH CX
    PUSH DX
    PUSH SI
    PUSH DI
    MOV CX, DIM -1      ; contatore ciclo
    XOR SI, SI          ; indice scansione vettore di word
    MOV DI, 1           ; lunghezza sottosequenza corrente
    MOV AX, 1           ; lunghezza sottosequenza massima
    XOR BX, BX          ; indice sequenza massima
    PUSH BX             ; indice seq massima nello stack
ciclo:  MOV DX, vet[SI]
        ADD SI, 2
        CMP DX, vet[SI]
        JLE NEXT        ; controllo monotonia della sequenza
        CMP DI, AX
        JBE nomax        ; controllo per ricerca massimo
        MOV AX, DI
        ADD SP, 2        ; equivalente a POP
        PUSH BX
        MOV BX, SI
nomax:  XOR DI, DI
next:   INC DI
        LOOP ciclo
        CMP DI, AX      ; controllo per ricerca massimo
        JBE nomax2      ; (utile se la sottosequenza cercata
        MOV AX, DI      ; è alla fine del vettore)
        ADD SP, 2
        JMP fine
nomax2: POP BX           ; prelevamento risultato da stack
fine:   SHR BX, 1
        POP DI
        POP SI
        POP DX
        POP CX
        RET
MONOTONO ENDP
END

```

16.13. Rimozione di occorrenze multiple consecutive di caratteri

Si scriva una procedura `CONVERTI` in grado di rimuovere tutte le occorrenze di caratteri ripetuti consecutivamente in una stringa. Ad esempio, la stringa “notte rossa” (dimensione 11) deve essere trasformata nella stringa “note rosa” (dimensione 9).

La procedura deve ricevere come input tramite *stack*:

- l'indirizzo della stringa di origine (tale stringa dovrà essere sovrascritta dalla nuova stringa elaborata)
- la dimensione in *byte* della stringa origine.

Sempre tramite *stack*, la procedura deve fornire come output la dimensione della stringa trasformata. Non è ammesso l'uso di altre variabili in memoria.

Si supponga che il programma chiamante contenga il seguente codice:

```

[...]  

lea ax, stringa  

push ax  

mov ax, DIMENSIONE  

push ax  

sub sp, 2  

call converti  

pop ax  

mov DIMENSIONE_AGGIORNATA, ax  

[...]
```

16.13.1. Codice

```

DIM      EQU 11  

         .MODEL small  

         .STACK  

         .DATA  

stringa  DB "notte rossa"  

newdim   DW ?  

         .CODE  

         .STARTUP  

LEA AX, stringa  

PUSH AX  

MOV AX, DIM  

PUSH AX  

SUB SP, 2  

CALL CONVERTI  

POP newdim  

ADD SP, 4  

.EXIT  

;*****  

;          CONVERTI  

; Procedura per la rimozione di occorrenze multiple consecutive di caratt.  

; non-decrescente di interi in un vettore  

; Dati in ingresso: offset stringa di origine (tramite stack)  

;                   dimensione stringa di origine (tramite stack)  

; Risultati: stringa originaria trasformata  

;            dimensione stringa trasformata (tramite stack)  

;*****  

CONVERTI PROC  

    MOV BP, SP  

    MOV CX, [BP+4]  

    MOV SI, [BP+6]  

    MOV DI, SI  

    INC DI  

    DEC CX  

    MOV BX, 1  

ciclo:   MOV AL, [DI]  

        CMP AL, [SI]  

        JE  next  

        INC SI  

        MOV [SI], AL  

        INC BX  

next:    INC DI  

        LOOP ciclo  

        MOV [BP+2], BX  

        RET  

CONVERTI ENDP  

END
```

16.14. Conversione ASCII-binario

Si desidera scrivere una procedura che legga da tastiera un numero intero positivo come sequenza di caratteri ASCII, e restituisca nel registro DX la rappresentazione binaria del numero stesso. Se il numero letto è troppo grande per essere rappresentato su 16 bit deve essere visualizzato un opportuno messaggio d'errore.

16.14.1. Codice

```
CR      EQU      13
        .MODEL   large
        PUBLIC   INPUT
        .DATA
ERR_MESS DB      "Numero troppo grande", 0DH, 0AH, "$"
        .CODE
;*****
;                               INPUT
; Procedura di lettura e conversione di un numero. Il numero letto e
; decodificato viene scritto in DX.
;*****
INPUT    PROC     FAR
        PUSH     AX
        PUSH     BX
lab0:    XOR      DX, DX
lab1:    MOV      BX, 10
        MOV      AH, 1                ; legge un carattere
        INT      21H
        CMP      AL, CR              ; è un CR ?
        JE       fine                ; Sì: fine
        CMP      AL, '0'             ; No: controlla se è un numero
        JB       lab1
        CMP      AL, '9'
        JA       lab1
        SUB      AL, '0'              ; sottrae la codifica di '0'
        XCHG     AX, BX
        XOR      BH, BH
        MUL      DX                  ; moltiplica per 10
        CMP      DX, 0
        JNE      i_err
        MOV      DX, AX
        ADD      DX, BX              ; somma la cifra letta
        JC       i_err
        JMP      lab1
i_err:   LEA      DX, ERR_MESS
        MOV      AH, 9
        INT      21H
        JMP      lab0
fine:    POP      BX
        POP      AX
        RET
INPUT    ENDP
        END
```

16.15. Conversione binario-ASCII

Si desidera scrivere una procedura che riceve in ingresso nel registro DX un numero intero positivo in rappresentazione binaria, e visualizza su video la corrispondente sequenza di caratteri ASCII.

16.15.1. Implementazione

Viene utilizzato un algoritmo in due passi: nel primo si divide il numero binario trasformando il

resto in codice ASCII e ripetendo l'operazione sul quoziente, sino a che questo è diverso da zero; nel secondo passo si visualizzano le cifre così ottenute in ordine inverso a quello di generazione. Per memorizzare le cifre calcolate così ottenute si utilizza un vettore temporaneo CBUF.

16.15.2. Codice

```

CR      EQU      13
LF      EQU      10
        .MODEL   large
        PUBLIC   OUTPUT
        .DATA
CBUF     DB       5 DUP(0)
        .CODE
;*****
;
;               OUTPUT
; Procedura di conversione e visualizzazione del numero letto in DX.
;*****
OUTPUT   PROC     FAR
        PUSH     DI
        PUSH     AX
        PUSH     BX
        PUSH     DX
        XOR      DI, DI                ; primo passo
        MOV      AX, DX
conv:    XOR      DX, DX
        MOV      BX, 10
ciclo:   DIV      BX
        ADD      DL, '0'              ; trasformazione del resto
                                           ; in codice ASCII
                                           ; memorizzazione nel buffer
        MOV      CBUF[DI], DL
        INC      DI
        XOR      DX, DX
        CMP      AX, 0
        JNE      ciclo
lab:     DEC      DI                ; secondo passo
        MOV      DL, CBUF[DI]
        MOV      AH, 2
        INT      21H                ; visualizzazione di una cifra
        CMP      DI, 0
        JNE      lab
        MOV      DL, CR              ; stampa un CR
        MOV      AH, 2
        INT      21H                ; stampa un LF
        MOV      DL, LF
        INT      21H
        POP      DX
        POP      BX
        POP      AX
        POP      DI
        RET
OUTPUT   ENDP
        END

```

16.16. Buffer circolare

Si vogliono realizzare due procedure che implementano le operazioni di *inserimento* e *cancellazione* di un dato in una struttura dati che implementa la strategia *FIFO* (*First-In First-Out*); tale struttura è organizzata sotto forma di *buffer circolare*.

16.16.1. Implementazione

Il buffer circolare può essere realizzato utilizzando un vettore dimensionato in base al numero massimo di elementi che si pensa di dover memorizzare. Chiameremo MAX tale valore.

Le operazioni di lettura/scrittura sul buffer vengono eseguite tramite due indici, che mantengono memoria rispettivamente di quale sia la casella contenente il dato da più tempo inserito, e di quale la casella da più tempo vuota. Siano BOUT e BIN i due indici, rispettivamente. Ad ogni inserzione, il nuovo elemento viene inserito nella casella indicata da BIN, che viene poi incrementato. Ogni volta che si estrae un elemento dal buffer, si legge quello contenuto nella casella indicata da BOUT, che viene poi a sua volta incrementato. In ambedue i casi, l'incremento viene fatto tenendo conto che il buffer è *circolare*, per cui una volta che un indice giunge al fondo del vettore, viene riportato in testa. Questo significa che, detta DIMREC la dimensione in byte del singolo elemento del buffer, le operazioni di incremento dei due indici vengono fatte modulo $\text{DIMREC} * \text{MAX}$.

Nella gestione di un buffer circolare esiste il problema di determinare quali siano le condizioni che caratterizzano le due situazioni di *buffer pieno* e *buffer vuoto*, in corrispondenza delle quali non è possibile eseguire operazioni di input ed output, rispettivamente. Si può dimostrare che dal solo esame del valore dei due indici BIN e BOUT non è possibile determinare se un buffer sia completamente pieno, oppure vuoto. Vi sono quindi due soluzioni:

- si utilizza un contatore delle caselle occupate, controllando il valore di questo prima di fare una operazione di inserimento o estrazione;
- si utilizzano al più $\text{MAX} - 1$ caselle del buffer.

Le due soluzioni presentano approssimativamente gli stessi vantaggi/svantaggi, sia in termini di uso di memoria sia di efficienza. Si seguirà qui la seconda soluzione. In tal caso le due condizioni da verificare per conoscere la situazione del buffer sono:

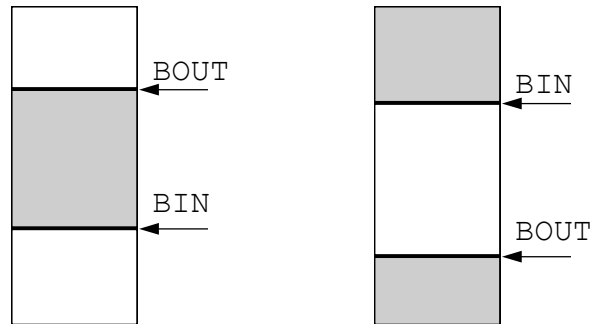
- buffer vuoto: $\text{BIN} = \text{BOUT}$
- buffer pieno: in questo caso la condizione è diversa a seconda che il buffer sia nella situazione $\text{BOUT} < \text{BIN}$ (Fig. 16.2.a) oppure nella situazione inversa $\text{BIN} < \text{BOUT}$ (Fig. 16.2.b):
 - se $\text{BOUT} < \text{BIN}$:
buffer pieno $\Leftrightarrow \text{BIN} - \text{BOUT} = (\text{MAX} - 1) * \text{DIMREC}$
 - se $\text{BIN} < \text{BOUT}$:
buffer pieno $\Leftrightarrow \text{BOUT} - \text{BIN} = (\text{MAX} - 1) * \text{DIMREC}$.

Nel seguito viene proposto un programma composto da un semplice *main* che gestisce un'interfaccia in grado di accettare dall'utente i comandi di

- inserzione di un elemento nel buffer (+numero);
- estrazione di un elemento dal buffer (-);
- uscita dal programma (E).

Il *main* richiama le procedure ENQUEUE e DEQUEUE che eseguono le operazioni di inserzione ed estrazione di elementi dal buffer circolare, secondo le regole esposte sopra. Gli elementi sono costituiti da interi di 16 bit, per cui DIMREC vale 2. Ne conseguono una serie di semplificazioni nei calcoli, sia per incrementare i due indici, sia per verificare le condizioni di buffer pieno/vuoto.

Il programma fa uso delle procedure INPUT e OUTPUT viste in precedenza, oltre che della procedura ACAPO presentata nel Cap. 4.



a) $BOUT < BIN$ b) $BIN < BOUT$

Fig. 16.2: Buffer circolare.

16.16.2. Codice

```

.MODEL large
EXTRN INPUT:FAR, OUTPUT:FAR, ACAPO:FAR
DIM EQU 5 ; numero di elementi del buffer
.DATA
BUFF DW DIM DUP(0)
BIN DW 0
BOUT DW 0
FULL_MESS DB 'BUFFER PIENO', 0DH, 0AH, '$'
EMPT_MESS DB 'BUFFER VUOTO', 0DH, 0AH, '$'
PROMPT DB '>>$'
.STACK
.CODE
.STARTUP
rd_cmd: LEA DX, PROMPT ; stampa il prompt
MOV AH, 9
INT 21H
MOV AH, 1 ; legge il comando
INT 21H
CMP AL, '+' ; inserisce
JE cmd_i
CMP AL, '-' ; estrae
JE cmd_e
CMP AL, 'E' ; end
JE fine
JMP rd_cmd
cmd_i: CALL INPUT
CALL ACAPO
CALL ENQUEUE
JMP rd_cmd
cmd_e: CALL ACAPO
CALL DEQUEUE
CMP DX, 0FFFFH ; errore
JE rd_cmd
CALL OUTPUT
JMP rd_cmd
fine: .EXIT
;*****
; ENQUEUE
; Inserisce il contenuto di DX nel buffer.
;*****
ENQUEUE PROC FAR
PUSH AX
PUSH BX
PUSH DX
PUSH SI
MOV SI, BIN

```

```

        CMP     SI, BOUT
        JB      compt      ; se BIN<BOUT salta
        MOV     BX, BIN     ; testa se il buffer è pieno
        SUB     BX, BOUT
        CMP     BX, 2*DIM-2
        JE      full
        JMP     insert
compt:   MOV     BX, BOUT     ; testa se il buffer è pieno
        SUB     BX, BIN
        DEC     BX
        DEC     BX
        JE      full
insert:  MOV     BUFF[SI], DX ; inserisce nel buffer
                                   ; l'elemento contenuto in DX
        INC     SI
        INC     SI
        CMP     SI, 2*DIM
        JNE     e_lab
        XOR     SI, SI
e_lab:   MOV     BIN, SI
        JMP     e_fine
full:    LEA     DX, FULL_MESS
        MOV     AH, 9
        INT     21H
e_fine:  POP     SI
        POP     DX
        POP     BX
        POP     AX
        RET
ENQUEUE ENDP
;*****
;
;               DEQUEUE
; Estrae un numero dal buffer e lo mette in DX. Ritorna il valore FFFFh
; se il buffer è vuoto.
;*****
DEQUEUE PROC FAR
        PUSH    AX
        PUSH    SI
        MOV     SI, BOUT
        CMP     SI, BIN     ; testa se il buffer è vuoto
        JE      empty
        MOV     DX, BUFF[SI] ; estrae un elemento dal buffer
        INC     SI
        INC     SI
        CMP     SI, 2*DIM
        JNE     d_lab
        XOR     SI, SI
d_lab:   MOV     BOUT, SI
        JMP     d_fine
empty:   LEA     DX, EMPT_MESS
        MOV     AH, 9
        INT     21H
        MOV     DX, 0FFFFH
        JMP     d_fine
d_fine:  POP     SI
        POP     AX
        RET
DEQUEUE ENDP
END

```

16.17. Ricerca di una sottomatrice

Si scriva una procedura Assembler per la ricerca di una sottomatrice MAT2 all'interno di una matrice MAT1.

Le due matrici sono memorizzate per righe, sono composte di interi con segno, ed hanno dimensioni contenute in memoria alle locazioni X1, Y1 e X2, Y2, rispettivamente. La matrice MAT1 è composta da un numero massimo di 100 righe e 100 colonne, mentre la matrice MAT2 è composta da un numero massimo di 10 righe e 10 colonne.

In caso di successo, la procedura deve visualizzare le coordinate dell'elemento nell'angolo in alto a sinistra della sottomatrice trovata; diversamente deve visualizzare un opportuno messaggio. Nel caso la sottomatrice da cercare sia presente più volte all'interno della matrice MAT1, è sufficiente fornire le coordinate di una delle sottomatrici corrispondenti.

16.17.1. Implementazione

La procedura MATCMP utilizza alcune procedure ausiliarie:

- CONVERT, che esegue il calcolo dell'offset dell'elemento le cui coordinate sono contenute in AX,DX;
- COMPARE, che confronta la matrice MAT2 con la sottomatrice di dimensione X2·Y2, il cui primo elemento ha le coordinate contenute in AX,DX;
- GETMAT acquisisce una matrice da tastiera, leggendo innanzitutto il valore delle due dimensioni.

16.17.2. Codice

```

.MODEL large
EXTRN INPUT:FAR, OUTPUT:FAR, ACAPO:FAR
MAX1 EQU 100
MAX2 EQU 10
.STACK
.DATA
MAT1 DW MAX1*MAX1 DUP (0)
MAT2 DW MAX2*MAX2 DUP (0)
X1 DW 0
Y1 DW 0
X2 DW 0
Y2 DW 0
Y12 DW 0
X12 DW 0
DIM1 DW 0
DIM2 DW 0
.CODE
.STARTUP
LEA DI, MAT1
CALL GETMAT
MOV X1, AX
MOV Y1, DX
LEA DI, MAT2
CALL GETMAT
MOV X2, AX
MOV Y2, DX
CALL MATCMP
.EXIT
;*****
;
; MATCMP
; Verifica se la matrice MAT2 è contenuta in MAT1: se sì visualizza le
; coordinate dell'elemento in alto a sinistra della sottomatrice.
;*****
MATCMP PROC NEAR
PUSH AX
PUSH BX
PUSH CX
PUSH DX
PUSH DI

```

```

        PUSH    SI
        MOV     AX, X2
        MUL     Y2
        MOV     DIM2, AX                ; dimensione MAT2
        MOV     AX, Y1
        SUB     AX, Y2
        MOV     Y12, AX
        MOV     AX, X1
        SUB     AX, X2
        MOV     X12, AX
        XOR     AX, AX                ; riga
lab1:    XOR     DX, DX                ; colonna
lab2:    CALL    COMPARE                ; esegue il confronto
        CMP     BX, 0
        JE      found
        INC     DX
        CMP     DX, Y12
        JBE     lab2
        INC     AX
        CMP     AX, X12
        JBE     lab1
ko:      JMP     fine
found:   XCHG    AX, DX
        CALL    OUTPUT
        MOV     DX, AX
        CALL    OUTPUT
fine:    POP     SI
        POP     DI
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET

MATCMP   ENDP
;*****
;                                COMPARE
; Esegue il confronto tra MAT2 e la sottomatrice a partire dall'elemento
; di coordinate AX,DX. In BX ritorna 0 se sono uguali, 1 altrimenti.
;*****
COMPARE  PROC    NEAR
        PUSH    AX
        PUSH    CX
        PUSH    SI
        PUSH    DI
        CALL    CONVERT                ; calcola DI
        LEA     SI, MAT2
        MOV     CX, DIM2
        MOV     BX, Y2
        SHL     BX, 1
        ADD     BX, DI                ; termine riga
ciclo:   MOV     AX, [DI]
        CMP     AX, [SI]
        JNE     diff
        INC     SI
        INC     SI
        INC     DI
        INC     DI
        CMP     DI, BX
        JNE     lab3
        ADD     DI, Y12
        ADD     DI, Y12                ; indirizzo nuova riga
        MOV     BX, Y2                ; nuovo termine riga
        SHL     BX, 1
        ADD     BX, DI
lab3:    DEC     CX
        JNE     ciclo
        XOR     BX, BX
        JMP     basta

```

```

diff:    MOV     BX, 1
basta:   POP     DI
         POP     SI
         POP     CX
         POP     AX
         RET
COMPARE  ENDP
;*****
;
;               CONVERT
; Calcola in DI l'offset dell'elemento di MAT1 le cui coordinate sono
; contenute in AX, DX.
;*****
CONVERT  PROC    NEAR
         PUSH    AX
         PUSH    CX
         PUSH    DX
         MOV     CX, DX
         LEA     DI, MAT1
         MUL     Y1
         ADD     AX, CX
         SHL     AX, 1
         ADD     DI, AX
         POP     DX
         POP     CX
         POP     AX
         RET
CONVERT  ENDP
;*****
;               GETMAT
; Acquisisce una matrice: legge il numero di righe e di colonne, li mette
; in AX e DX, e poi legge gli elementi, memorizzandoli a partire
; dall'indirizzo contenuto in DI.
;*****
GETMAT   PROC    NEAR
         PUSH    CX
         CALL    INPUT                ; numero righe
         PUSH    DX
         MOV     AX, DX
         CALL    ACAPO
         CALL    INPUT                ; numero colonne
         PUSH    DX
         MOV     CX, DX
         CALL    ACAPO
         MUL     CX                    ; calcolo numero elementi
         MOV     CX, AX
labget:  CALL    INPUT
         CALL    ACAPO
         MOV     [DI], DX
         INC     DI
         INC     DI
         LOOP    labget
         POP     DX                    ; ripristino numero colonne
         POP     AX                    ; ripristino numero righe
         POP     CX
         RET
GETMAT   ENDP
END

```

16.18. Mappa geografica digitalizzata

Si supponga di avere in memoria la rappresentazione digitalizzata di una carta geografica, sotto forma di una matrice di byte composta da 200 righe e 200 colonne, memorizzata per righe.

La carta è relativa ad una zona, di cui viene semplicemente riportata la suddivisione in regioni:

tutti i byte della matrice contengono il valore 0, tranne quelli corrispondenti alle linee di demarcazione tra regioni, che contengono il valore 0FFH. I byte delle colonne 1 e 200, così come delle righe 1 e 200, contengono il valore 0FFH.

Si scriva una procedura Assembler che determini il numero di regioni in cui è suddivisa la zona rappresentata, e lo visualizzi. Si assuma che la matrice rappresenti correttamente una zona suddivisa in regioni: le linee di demarcazione sono sempre continue e corrispondono a curve chiuse.

In Fig. 16.3 è illustrata una matrice di 20 righe e 20 colonne in cui è indicato con 1 il valore FF.

```

11111111111111111111
10010000000000000001
10001000000000000001
10001000000000000001
10001000000000000001
10001000000000000001
10001000000000000001
10000111100000000001
10000000100000000001
10000000100000000001
10000000100000000001
10000001111111111111
100000100000000100001
100000100000000100001
111111000000000100001
100000000000000100001
100000000000000100001
100000000000000100001
100000000000000100001
100000000000000100001
100000000000000100001
11111111111111111111

```

Fig. 16.3: Esempio di mappa.

16.18.1. Implementazione

Per determinare il numero di regioni in cui è suddivisa l'immagine si procede marcando per ognuna tutti i punti che la compongono. Terminata una regione si cerca un punto non ancora marcato e quindi appartenente ad una nuova regione, e si marcano tutti i punti che in essa ricadono.

Si ripete l'operazione, sino a che non esiste più alcun punto non marcato, e quindi nessuna regione non ancora visitata.

Per la soluzione del problema si è utilizzato un algoritmo che fa uso di un buffer circolare per la memorizzazione dei punti da analizzare, espresso in pseudo-C nella forma seguente:

```

nreg = 1;
do
{  cerca una casella non marcata;
  if (trovata)
  {
    do
    {  marca la casella;
      inserisci nel buffer i suoi vicini non marcati e non di confine;
      estrai una casella dal buffer;
    }
    while (ci sono caselle nel buffer);
    incrementa nreg;
  }
}
while (trovata);
visualizza (nreg - 1);

```

16.18.2. Codice

```

.MODEL large
MAX EQU 10000 ; dimensione buffer circolare
DIM EQU 11 ; dimensione matrice
EXTRN OUTPUT:FAR, ENQUEUE:FAR, DEQUEUE:FAR
.STACK
.DATA
MATRIX DB 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255
DB 255, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 255
DB 255, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 255
DB 255, 255, 255, 0, 255, 255, 255, 255, 255, 255, 0, 255
DB 255, 0, 255, 0, 255, 0, 0, 0, 255, 0, 255, 255
DB 255, 0, 255, 0, 255, 255, 255, 0, 255, 0, 255, 255
DB 255, 0, 255, 0, 255, 0, 255, 0, 255, 0, 255, 255
DB 255, 0, 255, 0, 255, 0, 255, 0, 255, 0, 255, 255
DB 255, 0, 255, 0, 255, 0, 0, 0, 0, 0, 0, 255
DB 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255
ERR_MESS DB 'Numero troppo grande', 0DH, 0AH, '$'
FULL_MESS DB 'BUFFER PIENO', 0DH, 0AH, '$'
EMPT_MESS DB 'BUFFER VUOTO', 0DH, 0AH, '$'
.CODE
.STARTUP
MOV AX, 1 ; contatore numero regioni
loop0: XOR SI, SI ; indice corrente in MATRIX
MOV CX, DIM*DIM
loop1: INC SI ; cerca se esistono altre
; regioni da esplorare
CMP MATRIX[SI], 0
LOOPNE loop1
JCXZ eefine
MOV MATRIX[SI], AL ; marca il primo elemento
loop2: CALL EXPAND
CALL DEQUEUE
MOV SI, DX
CMP SI, 0FFFFh
JNE loop2
INC AX ; passa a un'altra regione
JMP loop0
eefine: MOV DX, AX
DEC DX ; visualizza numero di regioni
CALL OUTPUT
.EXIT ; ritorno a DOS
;*****
; EXPAND
; Trova in SI il puntatore ad una casella di MATRIX; per ognuna delle 4
; caselle a questa adiacenti deve controllare se sono nella matrice e se
; sono libere. In caso affermativo le occupa, scrivendovi il valore
; contenuto nel registro AL, e mette il relativo puntatore nel buffer.
; Essendo i bordi della matrice tutti settati a 1, il primo controllo
; coincide con il secondo, e viene saltato.
;*****
EXPAND PROC NEAR
PUSH DI
PUSH DX
MOV DI, SI
ADD DI, DIM
CMP MATRIX[DI], 0
JNE lab1
MOV MATRIX[DI], AL
MOV DX, DI
CALL ENQUEUE
lab1: MOV DI, SI
DEC DI
CMP MATRIX[DI], 0
JNE lab2

```



```

                MOV     MATRIX[DI], AL
                MOV     DX, DI
                CALL    ENQUEUE
lab2:           MOV     DI, SI
                SUB     DI, DIM
                CMP     MATRIX[DI], 0
                JNE     lab3
                MOV     MATRIX[DI], AL
                MOV     DX, DI
                CALL    ENQUEUE
lab3:           MOV     DI, SI
                INC     DI
                CMP     MATRIX[DI], 0
                JNE     lab4
                MOV     MATRIX[DI], AL
                MOV     DX, DI
                CALL    ENQUEUE
lab4:           POP     DX
                POP     DI
                RET
EXPAND         ENDP
                END

```

16.19. Ricerca degli anagrammi

Si supponga di aver accesso in memoria ad un elenco di parole composte dalle sole lettere maiuscole dalla A alla Z. Ogni parola nell'elenco è seguita dal carattere \$; l'ultima parola è seguita dai due caratteri \$\$.

Si scriva un programma Assembler che esegue le seguenti operazioni:

- legga da tastiera una parola (controllando che sia composta da sole lettere maiuscole);
- visualizzi tutte le parole dell'elenco che sono un anagramma della parola data.

Si assuma che l'elenco abbia una lunghezza non superiore ai 40Kbyte.

Esempio

Supponendo che l'elenco sia così costituito:

SALUTE\$ANCORA\$GRANO\$MAMMA\$ARNESE\$RAMPA\$ARGANO\$RAGNO\$PIOGGIA\$\$

se da tastiera viene inserita la parola RANGO il programma deve visualizzare le parole GRANO e RAGNO.

16.19.1. Soluzione proposta

Una prima possibile soluzione consiste nel generare tutte le possibili parole che sono un anagramma della parola data e, per ognuna, verificare se esiste nell'elenco. A parte la complessità di calcolo, esiste il problema della generazione di tutti i possibili anagrammi di una parola.

Una soluzione più semplice ed efficiente è quella di trasformare la parola data e tutte quelle contenute nell'elenco in una forma *canonica*, ad esempio quella in cui tutte le lettere sono ordinate alfabeticamente. Se, ad esempio, da tastiera viene introdotta la parola RANGO, essa viene prima trasformata in AGNOR, quindi, per ogni parola nell'elenco, si controlla che abbia la stessa lunghezza di RANGO; se così è, la si converte nella forma canonica e la si confronta con AGNOR.

Per rappresentare le stringhe si utilizzano diversi vettori nei quali il primo byte contiene la lunghezza del vettore, supposta inferiore a 256.

16.19.2. La procedura di ordinamento

Per ordinare il vettore di n caratteri si utilizza l'algoritmo di ordinamento noto come *Bubble-sort*. Esso consiste nello scandire per $n-1$ volte il vettore, considerando ogni coppia di caratteri adiacenti,

verificando che rispetti l'ordinamento (crescente/decrecente) desiderato, e scambiando gli elementi della coppia se sono in ordine opposto.

L'algoritmo in pseudo-C risulta essere:

```
for(j=0; j<n-1; j++)
  for(i=0; i<n-1; i++)
    if (vett[i] > vett[i+1])
      scambio vett[i] e vett[i+1];
```

L'efficienza della procedura può essere incrementata introducendo un flag, che permetta di sapere quando una scansione completa del vettore non ha causato alcuno scambio sospendendo in tal caso l'elaborazione.

La soluzione proposta in linguaggio Assembler è la seguente:

```
.MODEL large
PUBLIC SORT
.DATA
SAVE_CNT DW ?
START_ADD DW ?
.CODE
;*****
;
; BUBBLE
; Procedura di ordinamento di un vettore con l'algoritmo di bubble-sort. Il
; vettore è localizzato nel segmento di dato, a partire dall'offset
; contenuto in DI, ed è composto di byte. Il primo elemento contiene la sua
; lunghezza (numero di elementi + 1), che si suppone minore di 256.
;*****
SORT PROC FAR
    PUSH AX
    PUSH BX
    PUSH CX
    MOV CL, [DI]
    XOR CH, CH
    DEC CX
    MOV SAVE_CNT, CX
    INC DI
    MOV START_ADD, DI
init:  MOV BX, 1
    MOV CX, SAVE_CNT
    MOV DI, START_ADD
next:  MOV AL, [DI]
    CMP [DI+1], AL
    JAE cont
    XCHG [DI+1], AL
    MOV [DI], AL
    SUB BX, BX
cont:  INC DI
    LOOP next
    CMP BX, 0
    JE init
    POP CX
    POP BX
    POP AX
    RET
SORT ENDP
END
```

16.19.3. Codice

```
DIM EQU 20
CR EQU 13
```

```

.MODEL large
EXTRN  SORT:FAR
.STACK
.DATA
BUFF  DB  'SALUTE$ANCORA$GRANO$MAMMA$ARNESE$POZZO$RAMPONE$ARGANO$'
      DB  'RAGNO$PIOGGIA$$'
PAR1   DB  DIM DUP (0)
PAR2   DB  DIM DUP (0)
PAR3   DB  DIM DUP (0)
.CODE
.STARTUP
      MOV     BX, 1                ; acquisisce da tastiera
      MOV     AH, 1                ; la parola da cercare
lab1:   INT     21H
      CMP     AL, CR                ; è un CR ?
      JE      end1
      CMP     AL, 'A'                ; è una lettera maiuscola ?
      JB      lab1
      CMP     AL, 'Z'
      JA      lab1
      MOV     PAR1[BX], AL
      INC     BX
      JMP     lab1
end1:   DEC     BX
      MOV     PAR1, BL
      LEA     DI, PAR1                ; ordina la parola da cercare
      CALL    SORT
      CALL    ACAPO
      XOR     SI, SI
lab2:   MOV     BX, 1                ; carica parola in PAR2 e PAR3
ciclo:  MOV     AL, BUFF[SI]
      MOV     PAR2[BX], AL
      MOV     PAR3[BX], AL
      INC     BX
      INC     SI
      CMP     AL, '$'
      JNE     ciclo
      DEC     BX
      DEC     BX
      MOV     PAR2, BL                ; ordina la parola in PAR2
      LEA     DI, PAR2
      CALL    SORT
      CALL    CMPSTR                ; confronta PAR1 e PAR2
      CMP     DX, 0
      JE      lab3                ; se sono diverse salta
      MOV     AH, 9                ; visualizza la parola
      LEA     DX, PAR3
      INC     DX
      INT     21H
      CALL    ACAPO
lab3:   MOV     AL, BUFF[SI]          ; BUFF è finito ?
      CMP     AL, '$'
      JNE     lab2
      .EXIT
;*****
;                               CMPSTR
;
; Procedura per il confronto di due stringhe, contenute nelle due variabili
; esterne PAR1 e PAR2. Si suppone che il primo elemento sia costituito dalla
; lunghezza (su 8 bit) della stringa stessa.
; Ritorna in DX il risultato: 0 se sono diverse, 1 se sono uguali.
;*****
CMPSTR  PROC    NEAR
      PUSH    AX
      PUSH    CX
      PUSH    DI
      PUSH    SI
init:   LEA     DI, PAR1
      LEA     SI, PAR2

```

```

                MOV     AL, [DI]
                CMP     AL, [SI]
                JNE     diff
                MOV     CL, AL
                XOR     CH, CH
                DEC     CX
lab:            INC     DI
                INC     SI
                MOV     AL, [DI]
                CMP     AL, [SI]
                JNE     diff
                LOOP    lab
equal:         MOV     DX, 1
                JMP     fine
diff:          XOR     DX, DX
fine:          POP     SI
                POP     DI
                POP     CX
                POP     AX
                RET
CMPSTR        ENDP
                END

```

16.20. Elaborazione di una matrice

Si supponga di avere in memoria una matrice di interi costituita da 150 righe e 150 colonne. Tale matrice è memorizzata per righe, e si assume che gli interi abbiano tutti valore compreso tra 0 e 1.000.

Si scriva una routine Assembler richiamabile da un programma C e di nome `minimum`, che elabori la matrice, incrementando di uno il valore contenuto in ognuno degli elementi che costituiscono un minimo locale. Un elemento è un *minimo locale* se il suo valore è minore o uguale di quello contenuto negli 8 elementi ad esso contigui.

La routine non deve fare uso di più di 50 Kbyte di memoria, 45 dei quali già sono costituiti dalla matrice da elaborare; essa riceve come unico parametro l'indirizzo della matrice.

16.20.1. Soluzione proposta

Il problema principale consiste nel non poter costruire mano a mano una matrice risultato conservando contemporaneamente la matrice originale. Tuttavia si noti come per determinare se un elemento della riga i -esima sia un minimo sia sufficiente conoscere i valori di tre sole righe della matrice originale: la i -esima, la $(i-1)$ -esima, e la $(i+1)$ -esima. Una possibile soluzione consiste quindi nel tenere sempre in memoria una copia di queste tre righe della matrice originale, così da poter scrivere direttamente sulla matrice originale i valori della matrice risultato, non appena questi sono stati calcolati.

Nel generico istante, per il calcolo della riga i -esima, il vettore `VETT1` contiene i valori della riga $(i-1)$ -esima della matrice originaria, il vettore `VETT2` quelli della riga i -esima, ed il vettore `VETT3` quelli della riga $(i+1)$ -esima, l'algoritmo utilizzato risulta essere il seguente:

1. calcola gli elementi della riga i -esima e li scrive nella matrice;
2. incrementa i ;
3. copia `VETT2` in `VETT1`;
4. copia `VETT3` in `VETT2`;
5. copia la $(i+1)$ -esima riga della matrice in `VETT3`;
6. ritorna al punto 1.

Per semplificare il trattamento degli elementi posti sul bordo della matrice si introducono due ri-

ghe e due colonne fittizie, riempite con il valore MAXINT; si noti infatti che quando si considerano gli elementi sul bordo, questi non cambiano il loro stato di minimo/non minimo introducendo attorno alla matrice originale queste due righe e queste due colonne.

Si definiscono quindi le seguenti procedure:

1. COPY: copia in VETT1 una riga della matrice originale;
2. NULLCOPY: scrive in VETT1 una riga di MAXINT;
3. SHIFT: copia VETT2 in VETT3 e VETT1 in VETT2;
4. LOCMIN: considera tutti gli elementi di una riga, determina se sono dei minimi (utilizzando VETT1, VETT2 e VETT3) ed eventualmente incrementa il valore corrispondente nella matrice.

Per la memorizzazione dei tre vettori VETT1, VETT2 e VETT3 si utilizza un'area dati locale alla procedura, memorizzata nello stack. Poiché ognuno dei tre vettori è composto da DIM+2 word, occorre dunque riservare uno spazio di $6 \cdot \text{DIM} + 12$ byte. In più, nello stack viene memorizzato l'indirizzo della riga che si sta considerando.

16.20.2. Codice

```

MAXINT    EQU    0FFFFH
DIM        EQU    150
.MODEL    small
.CODE
PUBLIC    _minimum
_minimum   PROC
    PUSH    BP
    MOV     BP, SP
    SUB     SP, 6*DIM+14          ; allocazione dell'area di variabili
                                   ; locali alla procedura

    PUSH    AX
    PUSH    BX
    PUSH    CX
    PUSH    DX
    PUSH    DI
    PUSH    SI
    MOV     BX, [BP+4]           ; indirizzo di partenza della
                                   ; matrice
    MOV     [BP-6*DIM-14], BX   ; indirizzo di partenza della
                                   ; prossima riga da copiare nello
                                   ; stack
    CALL    NULLCOPY           ; inserisce 2 righe di MAXINT
                                   ; nello stack

    CALL    SHIFT
    CALL    COPY
    CALL    SHIFT
    CALL    COPY
    MOV     CX, DIM-2
lab1:     CALL    LOCMIN
    CALL    SHIFT
    CALL    COPY
    LOOP    lab1
    CALL    LOCMIN
    CALL    SHIFT
    CALL    NULLCOPY
    ADD     WORD PTR [BP-6*DIM-14], 2*DIM
    CALL    LOCMIN
    POP     SI
    POP     DI
    POP     DX
    POP     CX
    POP     BX
    POP     AX
    ADD     SP, 6*DIM+14

```

```

        POP      BP
        RET
minimum ENDP
;*****
;                                COPY
; Copia nello stack, a partire dalla locazione BP-2*DIM-4 le DIM word
; esistenti in memoria a partire dalla locazione il cui indirizzo è
; contenuto in BP-6*DIM-14. Prima della prima word e dopo l'ultima
; inserisce una word di valore MAXINT.
;*****
COPY    PROC
        PUSH     AX
        PUSH     BX
        PUSH     CX
        PUSH     DI
        MOV      [BP-2], MAXINT
        MOV      [BP-4-2*DIM], MAXINT
        MOV      CX, DIM
        XOR      DI, DI
lab2:   MOV      BX, [BP-6*DIM-14]
        MOV      AX, [BX][DI]
        MOV      [BP-2*DIM-2][DI], AX
        INC      DI
        INC      DI
        LOOP     lab2
        ADD      BX, 2*DIM
        MOV      [BP-6*DIM-14], BX
        POP      DI
        POP      CX
        POP      BX
        POP      AX
        RET
COPY    ENDP
;*****
;                                NULLCOPY
; Copia nello stack, a partire dalla locazione BP-2*DIM-2, (DIM+2) word
; contenenti il valore MAXINT.
;*****
NULLCOPY PROC
        PUSH     AX
        PUSH     BX
        PUSH     CX
        PUSH     DI
        MOV      CX, DIM+2
        XOR      DI, DI
lab20:  MOV      [BP-2*DIM-4][DI], MAXINT
        INC      DI
        INC      DI
        LOOP     lab20
        POP      DI
        POP      CX
        POP      BX
        POP      AX
        RET
NULLCOPY ENDP
;*****
;                                SHIFT
; Sposta di 2*DIM posizioni verso l'alto le 2*DIM+2 word esistenti nello
; stack a partire dalla locazione BP-4*DIM-8.
;*****
SHIFT   PROC
        PUSH     AX
        PUSH     CX
        PUSH     SI
        MOV      CX, DIM*2+4
        XOR      SI, SI
lab3:   MOV      AX, [BP-4*DIM-8][SI]
        MOV      [BP-6*DIM-12][SI], AX

```

```

        INC     SI
        INC     SI
        LOOP    lab3
        POP     SI
        POP     CX
        POP     AX
        RET

SHIFT    ENDP
;*****
;
;               LOCMIN
; Considera gli elementi della riga contenuta nello stack a partire dalla
; locazione BP-4*DIM-2; di ognuno determina se è un minimo locale e in
; caso affermativo lo incrementa di 1.
;*****
LOCMIN    PROC
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    SI
        PUSH    DI
        MOV     CX, DIM
        XOR     SI, SI
lexb5:    MOV     AX, [BP-4*DIM-6][SI]
        MOV     DI, SI           ; casella contigua n.1
        ADD     DI, 2
        MOV     BX, [BP-4*DIM-6][DI]
        CMP     BX, MAXINT
        JE      next1
        CMP     AX, BX
        JBE     next1
        JMP     nomin
next1:    MOV     DI, SI           ; casella contigua n.2
        ADD     DI, 2*DIM+6
        MOV     BX, [BP-4*DIM-6][DI]
        CMP     BX, MAXINT
        JE      next2
        CMP     AX, BX
        JBE     next2
        JMP     nomin
next2:    MOV     DI, SI           ; casella contigua n.3
        ADD     DI, 2*DIM+4
        MOV     BX, [BP-4*DIM-6][DI]
        CMP     BX, MAXINT
        JE      next3
        CMP     AX, BX
        JBE     next3
        JMP     nomin
next3:    MOV     DI, SI           ; casella contigua n.4
        ADD     DI, 2*DIM+2
        MOV     BX, [BP-4*DIM-6][DI]
        CMP     BX, MAXINT
        JE      next4
        CMP     AX, BX
        JBE     next4
        JMP     nomin
next4:    MOV     DI, SI           ; casella contigua n.5
        SUB     DI, 2
        MOV     BX, [BP-4*DIM-6][DI]
        CMP     BX, MAXINT
        JE      next5
        CMP     AX, BX
        JBE     next5
        JMP     nomin
next5:    MOV     DI, SI           ; casella contigua n.6
        SUB     DI, 2*DIM+6
        MOV     BX, [BP-4*DIM-6][DI]
        CMP     BX, MAXINT
        JE      next6

```

```

        CMP     AX, BX
        JBE     next6
        JMP     nomin
next6:   MOV     DI, SI                ; casella contigua n.7
        SUB     DI, 2*DIM+4
        MOV     BX, [BP-4*DIM-6][DI]
        CMP     BX, MAXINT
        JE      next7
        CMP     AX, BX
        JBE     next7
        JMP     nomin
next7:   MOV     DI, SI                ; casella contigua n.8
        SUB     DI, 2*DIM+2
        MOV     BX, [BP-4*DIM-6][DI]
        CMP     BX, MAXINT
        JE      next8
        CMP     AX, BX
        JBE     next8
        JMP     nomin
next8:   MOV     BX, [BP-6*DIM-14]    ; incremento la cella della matrice
        SUB     BX, 4*DIM
        ADD     BX, SI
        INC     WORD PTR [BX]
nomin:   INC     SI
        INC     SI
        DEC     CX
        JZ      lab30
        JMP     lab5
lab30:   POP     DI
        POP     SI
        POP     CX
        POP     BX
        POP     AX
        RET
LOCMIN  ENDP
        END

```

16.21. Interpolazione di una funzione

Si scriva una procedura Assembler richiamabile da un programma C per il calcolo del valore di una funzione in un punto, utilizzando il metodo dell'interpolazione lineare.

La funzione è nota in forma tabulare ed assume valori interi rappresentati su 16 bit.

La procedura riceve come parametri:

- il puntatore alla testa di una tabella contenente sia 256 valori noti f_i della funzione, sia i valori x_i della variabile indipendente cui essi corrispondono; la tabella è composta da numeri interi su 16 bit, nell'ordine $x_1, f_1, x_2, f_2, \dots$; i valori x_i si suppongono in ordine crescente;
- il valore intero della variabile x , in corrispondenza del quale si vuole calcolare il valore di f ; se tale valore è uno di quelli della tabella, allora la procedura ritorna il valore in essa contenuto, altrimenti deve calcolarlo sulla base dei valori $f(x_i)$ e $f(x_{i+1})$ assunti dalla funzione nei due punti x_i, x_{i+1} tra cui cade x . Si utilizza per questo la formula:

$$f(x) = f(x_i) + (f_{i+1} - f_i) \cdot (x - x_i) / (x_{i+1} - x_i)$$

- il puntatore ad una variabile in cui la procedura scrive il valore di output, rappresentato da un numero intero su 16 bit.

Si suppone che il valore di x_i non sia esterno a quelli riportati in tabella.

16.21.1. Soluzione proposta

La procedura ricerca se il valore della x passato come parametro è nella tabella; in caso afferma-

tivo viene restituito il corrispondente valore della funzione, altrimenti si usa una macro CALC, a cui si passano i due valori di x_i e x_{i+1} nella tabella intorno a quello passato, ed i corrispondenti valori f_i ed f_{i+1} della funzione.

Si noti come la macro calcoli la formula eseguendo prima la moltiplicazione, poi la divisione, in modo da minimizzare l'errore derivante dal calcolo.

16.21.2. Codice

```

CALC      .MODEL    small
          MACRO     ICS1, EFFE1, ICS2, EFFE2
          LOCAL     salta1, salta2, salta3, salta4
          MOV       WORD PTR [BP-2], 0    ; flag di segno
          MOV       CX, EFFE2
          SUB       CX, EFFE1              ; f2-f1
          JNC       salta1                ; salta se il risultato è positivo
          NOT       WORD PTR [BP-2]
          salta1:   SUB       AX, ICS1      ; x-x1
          JNC       salta2
          NOT       WORD PTR [BP-2]
          salta2:   MUL       CX           ; |x-x1|*|f2-f1|
          MOV       CX, ICS2
          SUB       CX, ICS1              ; x2-x1
          JNC       salta3
          NOT       WORD PTR [BP-2]
          salta3:   DIV       CX           ; |x-x1|*|f2-f1| / |x2-x1|
          MOV       DX, EFFE1
          MOV       CX, [BP-2]
          OR        CX, CX
          JNS       salta4
          NOT       AX
          INC       AX
          salta4:   ADD       AX, DX
          ENDM
          .CODE
          PUBLIC    _interp
          _interp   PROC
          PUSH      BP
          MOV       BP, SP
          SUB       SP, 2
          PUSH      BX
          PUSH      CX
          PUSH      DX
          PUSH      DI
          PUSH      SI
          MOV       BX, [BP+4]             ; puntatore alla tabella
          MOV       AX, [BP+6]             ; valore x
          XOR       SI, SI
          CMP       AX, [BX+SI]
          JE        found                  ; il valore cercato è il primo
          lab:      CMP       AX, [BX+SI+4] ; il valore cercato è nella tabella
          JE        found
          JL        lab1
          ADD       SI, 4
          JMP       lab
          found:    MOV       AX, [BX+SI+6]
          JMP       fine
          lab1:     CALC     [BX+SI], [BX+SI+2], [BX+SI+4], [BX+SI+6]
          fine:     MOV       BX, [BP+8]    ; scrivi il risultato
          MOV       [BX], AX
          XOR       AX, AX                 ; valore di ritorno
          POP       SI
          POP       DI
          POP       DX
          POP       CX
          POP       BX

```

```
                ADD     SP, 2
                POP     BP
_interp        RET
                ENDP
                END
```


17. Esercizi proposti

In questo capitolo verranno presentate le specifiche di una serie di problemi la cui soluzione è lasciata come esercizio. Si tenga conto che i testi degli esercizi sono da intendersi come traccia, lasciando quindi spesso spazio per interpretazioni soggettive che tuttavia non stravolgono gli obiettivi didattici che sono alla base degli esercizi stessi.

17.1. Buffer FIFO

Si scrivano due procedure Assembler per la gestione di una struttura FIFO. Gli elementi da memorizzare sono stringhe di dimensione massima pari a 30 caratteri.

Le due procedure sono:

- `insert`: inserisce un elemento nel buffer; riceve in `SI` l'offset nel segmento di dato al quale si trova la parola da inserire, che ha un `$` come ultimo carattere; in `DX` scrive 1 se l'inserimento è stato eseguito correttamente, 0 se ci sono stati errori (*buffer pieno*);
- `extract`: estrae la stringa da più tempo nel buffer, scrivendola all'indirizzo il cui offset è contenuto in `DI`, e terminandola con un `$`; se ci sono errori (*buffer vuoto*) scrive uno 0 in `DX`, un 1 diversamente.

Per la memorizzazione del buffer si supponga di avere a disposizione un'area di 1 Kbyte a partire dalla locazione `BUFF`.

17.2. Sostituzione dei caratteri di tabulazione

Si supponga di avere in memoria un testo, memorizzato sotto forma di vettore di caratteri, contenente i caratteri visualizzabili (lettere, numeri, segni di interpunzione), il carattere *spazio*, il carattere di tabulazione ed i due caratteri `CR` e `LF`, che compaiono sempre in coppia.

Si scriva una procedura Assembler *retab* richiamabile da un programma C che legga il vettore e ne scriva uno nuovo, nel quale ad ogni carattere di tabulazione (codice ASCII 9) è stato sostituito il corrispondente numero di spazi (codice ASCII 32), in modo che il risultato di un'eventuale co-

mando di stampa o visualizzazione del testo rimanga inalterato. Si supponga che l'effetto del carattere di tabulazione sia quello di allineare il successivo carattere alla successiva colonna di indice multiplo di 8 a partire dall'inizio linea.

Si ipotizzi per il vettore iniziale (terminato da un carattere di codice ASCII 0) una lunghezza massima di 20.000 caratteri; il vettore risultato è già stato allocato di una dimensione sufficiente.

La procedura riceve come parametri il puntatore al vettore di caratteri di partenza e quello al vettore risultato; non viene ritornato alcun valore.

17.3. Sottrazione tra insiemi

Si scriva una procedura Assembler richiamabile da un programma C che esegua l'operazione di sottrazione tra insiemi. Gli elementi di ciascun insieme sono costituiti da stringhe di lunghezza variabile, ognuna terminata da un carattere \$, e sono memorizzati sotto forma di vettori di caratteri; all'interno di ogni vettore sono scritte, una di seguito all'altra, le stringhe componenti l'insieme. L'ultima stringa è terminata da un doppio carattere \$.

La procedura riceve come parametri:

- l'indirizzo di partenza del vettore contenente il primo insieme;
- l'indirizzo di partenza del vettore contenente il secondo insieme;
- l'indirizzo di partenza del vettore in cui scrivere l'insieme differenza; questo conterrà tutte le stringhe presenti nel primo insieme, ma non nel secondo.

Si assuma che i tre vettori stiano tutti nello stesso segmento di dato.

La procedura ritorna il numero di elementi contenuti nell'insieme risultato.

17.4. Triangolo di Floyd

Si scriva una procedura Assembler richiamabile da un programma C avente il seguente prototipo:

```
int floyd (int n, char *v);
```

Il parametro *v* rappresenta un vettore di 10.000 caratteri; *n* è un intero tra 0 e 99.

La procedura deve costruire una matrice triangolare di caratteri, memorizzata per righe nel vettore *v*, le cui righe contengono il cosiddetto *triangolo di Floyd*, così fatto:

```
01
02   03
04   05   06
07   08   09   10
...

```

Il triangolo deve contenere tutti i numeri sino a quello indicato dal parametro *n*. L'ultima riga, se non è completa, deve essere riempita usando caratteri * al posto dei numeri. La procedura ritorna il numero di righe.

Esempio

Se la procedura viene richiamata con parametro 8, la matrice deve contenere:

```
01
02   03
04   05   06
07   08   **   **

```

ed il valore ritornato è 4.

17.5. Formattazione di una stringa

Si scriva una procedura Assembler che esegua l'operazione di *formattazione* di un testo facendo in modo che ogni riga venga *giustificata* rispetto al margine destro. La procedura produce un nuovo testo che differisce dal precedente per la presenza di spazi aggiuntivi inseriti tra le parole. Il numero di spazi tra le varie parole di una stessa riga deve essere il più possibile costante all'interno della riga.

La procedura riceve come parametri:

- in SI il puntatore ad un vettore contenente il testo da formattare; questo è formato esclusivamente da caratteri (maiuscoli e minuscoli), spazi e CR; un carattere avente codice ASCII 0 conclude il testo, la cui lunghezza non è superiore a 64Kbyte;
- in DI il puntatore ad uno spazio di memoria che deve contenere il testo prodotto; tale spazio si suppone dimensionato in maniera sufficiente a contenere il testo dopo l'operazione di formattazione; il testo deve essere concluso da un carattere avente codice ASCII 0;
- in DX la lunghezza di ogni riga, espressa come numero intero compreso tra 20 e 80.

Si suppone che ciascuna riga nel vettore iniziale sia composta da un numero di caratteri inferiore od uguale al valore presente in DX.

17.6. Filtraggio di una sequenza

In memoria è contenuto un vettore di DIM interi, ciascuno su 16 bit.

Si scriva una procedura Assembler che trasformi il vettore sostituendo ad ogni elemento la media dei valori assunti dall'elemento stesso, dai k elementi che lo precedono, e dai k che lo seguono, secondo la formula

$$a_i = \frac{b_i + b_{i-k} + \dots + b_{i-1} + b_{i+1} + \dots + b_{i+k}}{2k+1}$$

dove a_i è il nuovo valore dell' i -esimo elemento del vettore, e b_i è il i -esimo elemento del vettore originario. Per i primi k elementi del vettore la media va fatta considerando i $k-1$ elementi che precedono quello esaminato, ed i k successivi; analogamente si procede per gli elementi successivi al $(DIM-k)$ -esimo.

La procedura riceve come parametri:

- il valore del parametro k (<50) in AX;
- l'indirizzo di partenza del vettore in BX;
- il numero DIM (<20.000) di elementi esistenti nel vettore in CX.

In nessun caso la procedura deve utilizzare più di 1 Kbyte di memoria.

Si tenga conto che nel considerare i k elementi precedenti a quello su cui si è posizionati si devono utilizzare i valori del vettore originario, e non quelli calcolati nei k passi precedenti.

17.7. Compressione di una stringa

Si scriva una procedura Assembler richiamabile da C che esegua la compressione di una stringa utilizzando un codice su 5 bit. La stringa è composta esclusivamente da lettere maiuscole e spazi. Ogni carattere è codificato su 5 bit; lo spazio corrisponde al codice 00000; le lettere hanno un codice corrispondente al proprio numero d'ordine nell'alfabeto: alla A corrisponde il codice 00001, alla B il codice 00010, ecc.

La procedura riceve come parametri:

- il puntatore al vettore contenente la stringa da comprimere, che si assume conclusa da un carattere 00H;
- il puntatore al vettore in cui scrivere la stringa compattata, che si suppone allocato di lunghezza sufficiente; nel primo byte del vettore andrà scritto il codice su 5 bit corrispondente al primo carattere, più i primi 3 bit del codice del secondo carattere; nel secondo byte andranno scritti gli ultimi 2 bit del secondo carattere, i 5 bit del terzo carattere, il primo bit del quarto carattere, e così via.

Se nella stringa è presente un carattere diverso da una lettera maiuscola o uno spazio la procedura ritorna il valore 0; altrimenti ritorna il numero di caratteri compattati.

I bit necessari a completare l'ultimo byte della stringa risultato vanno riempiti con 0.

17.8. Compattamento di valori

Si scriva una procedura Assembler richiamabile da C in grado di gestire una serie di numeri il cui valore è compreso tra 0 e 2000. La serie comprende al più 10.000 numeri.

Per risparmiare spazio in memoria ogni numero è rappresentato su 11 bit, in modo tale che all'interno del vettore ove la serie di numeri è memorizzata il secondo numero comincia al quarto bit del secondo byte, il terzo al settimo bit del terzo byte, il quarto al secondo bit del quinto byte, e così via.

Si scrivano le seguenti due procedure Assembler:

- **compact**: esegue il compattamento di una serie di numeri, ricevendo nello stack i seguenti parametri:
 - l'offset nel segmento di dato del vettore da compattare, contenente in ogni word un numero;
 - la lunghezza in byte del vettore da compattare;
 - l'offset nel segmento di dato del vettore risultato, in cui sono scritti i numeri compattati.Nella prima word del vettore deve venire scritta la lunghezza in byte del vettore stesso.
- **substitute**: esegue la sostituzione, all'interno del vettore compattato, di una stringa di bit con un'altra stringa di bit, ricevendo nello stack i seguenti parametri:
 - l'offset nel segmento di dato del vettore compattato;
 - l'offset nel segmento di dato di un vettore contenente la stringa di bit da cercare (il primo byte contiene il numero di bit di cui è costituita);
 - l'offset nel segmento di dato di un vettore contenente la stringa di bit da scrivere al posto della precedente (il primo byte contiene il numero di bit da cui è costituita).

Si noti che le due stringhe non hanno necessariamente la stessa lunghezza, e che questa non supera in ogni caso il valore 200.

La routine ritorna in AX il valore 0 se la ricerca (e la sostituzione) ha avuto esito positivo, 1 altrimenti. Nel caso il vettore risultato assuma una dimensione superiore ai 20.000 byte, l'operazione non deve essere eseguita, e la routine deve ritornare il valore 1.

17.9. Trasmissione seriale

Un sistema 8086 riceve dati da una linea seriale. I dati sono organizzati in *messaggi*: ogni messaggio viene acquisito dall'interfaccia seriale e da essa trasferito nella memoria del sistema, a partire dalla etichetta MESSAGE, secondo il formato descritto in seguito.

Ogni messaggio è composto da due parti: una *intestazione* ed un *corpo*. Il *corpo* comprende i dati

veri e propri, corrispondenti a interi positivi organizzati in pacchetti, a loro volta formati da parole, ognuna composta da n bit. L' intestazione comprende 4 byte:

- 2 byte forniscono il numero di pacchetti del messaggio
- 1 byte fornisce il numero di parole costituenti ogni pacchetto
- 1 byte fornisce il numero n (compreso tra 4 e 64) di bit costituenti una parola.

Le parole sono scritte una di seguito all'altra, senza tener conto dell'allineamento in byte della memoria. Alla fine di ogni pacchetto è inserita una parola di parità per il controllo della correttezza dei dati.

Si scriva una procedura Assembler che legga un messaggio dalla memoria, esegua i controlli di parità sui singoli pacchetti, e scriva all'indirizzo MAX il valore della parola di valore massimo, all'indirizzo MIN quello della parola di valore minimo, entrambi espressi su 64 bit. In caso di errore all'indirizzo ERROR si deve scrivere il numero d'ordine (su 16 bit) del pacchetto contenente il dato errato.

17.10. Compattamento di un segnale

Si supponga di avere memorizzata, nel segmento ESEG, la descrizione di un segnale, rappresentato come sequenza di 2^{15} campioni. Di ogni campione viene fornito:

- l'istante di campionamento (su 8 bit), espresso come distanza (in unità di tempo) dall'istante di campionamento precedente;
- il valore (su 8 bit) assunto dal segnale in corrispondenza dell'istante di campionamento; tale valore è espresso come differenza rispetto al valore del campione precedente. Il valore fornito per il primo campione si suppone essere il valore assoluto del segnale nell'istante di campionamento.

Si scriva una routine Assembler 8086 che esegua il compattamento della rappresentazione del segnale e crei nel segmento DSEG (a partire dall'indirizzo SIGNAL) una nuova rappresentazione secondo le regole seguenti:

- del segnale compattato viene fornito un valore per ogni intervallo di monotonia del segnale originario;
- ognuno di tali valori viene rappresentato da una coppia di elementi, ciascuno su 16 bit, corrispondenti rispettivamente all'istante finale dell'intervallo, ed al valore medio assoluto assunto dal segnale durante lo stesso.

Per il calcolo del valor medio V_{ave} assunto dal segnale durante un intervallo composto da n campioni, ognuno di valore v_i e relativo all'istante t_i , si utilizzi la formula seguente:

$$V_{ave} = \frac{1}{n} \sum_{i=1}^n v_i$$

Si noti che per il calcolo corretto di tale formula è necessario eseguire i calcoli su 32 bit.

17.11. Ricerca in un dizionario

Si scriva una procedura Assembler, denominata *search*, richiamabile da un programma C per la ricerca di parole all'interno di un vocabolario.

Il vocabolario è supposto memorizzato a partire dalla locazione VOCAB; le parole, in ordine alfabetico, sono scritte utilizzando la tecnica della *delta compression*: per ogni parola è indicato, su un byte, il numero di lettere iniziali uguali alla parola precedente, seguito dalle lettere diverse. L'ultima

parola è seguita dal carattere *. Si assume che il vocabolario non occupi più di 50Kbyte.

Esempio

Se il vocabolario fosse composto dalle 13 parole di sinistra, la sua rappresentazione in memoria sarebbe quella riportata a destra:

arma	VOCAB:	arma
armiere		3iere
articolo		2ticolo
artrosi		3rosi
bollare		0bollare
bollito		4ito
buccia		1uccia
buco		3o
orma		0orma
orso		2so
stima		0stima
stimare		5re
staccare		2accare
		*

La procedura *search* riceve come parametro la parola da cercare, composta esclusivamente da lettere minuscole e da un numero arbitrario di caratteri jolly ?, e ritorna il numero di parole del vocabolario che soddisfano la maschera di ricerca.

Tali parole vengono inoltre visualizzate, insieme con il contatore corrispondente al numero d'ordine di ciascuna.

Esempio

Se si chiede di ricercare nel vocabolario usato come esempio la parola ?rma, viene visualizzato:

```
1   arma
9   orma
```

17.12. Analisi delle ore di entrata/uscita

Si scriva una procedura Assembler richiamabile da un programma C che esegua le seguenti operazioni:

- legga da un vettore le ore di entrata e uscita dei dipendenti di una ditta, relativi ad una giornata di lavoro; i tempi sono espressi in ore e minuti e sono compresi tra 00:00 e 23:59;
- determini il momento in cui all'interno della ditta era presente il massimo numero di dipendenti;
- stampi i nomi dei dipendenti presenti all'interno della ditta in quel momento.

La procedura ha il seguente prototipo:

```
int max_pres (int *vett1, char **vett2, int num);
```

dove:

- *vett1* è un vettore di interi contenente *num* gruppi di 5 byte ciascuno: ogni gruppo contiene le informazioni relative all'ingresso o uscita di un dipendente, nel seguente formato:
 - i primi due byte memorizzano il codice identificatore del dipendente
 - il terzo byte contiene l'ora dell'entrata/uscita
 - il quarto byte contiene i minuti
 - il quinto byte può valere 1 oppure -1, a seconda che si tratti di una entrata o uscita, rispettivamente.

- *vett2* è un vettore di puntatori a caratteri: l'elemento *i-esimo* contiene il puntatore alla stringa contenente il nome dell'*i-esimo* dipendente, terminato da un `'\0'`.
- *num* è il numero di passaggi registrati nel corso della giornata.

La procedura restituisce come valore di ritorno il numero di dipendenti presenti all'interno della ditta nel momento di massima presenza.

Si assuma che tutti i dati siano memorizzati nello stesso segmento, in modo che i puntatori corrispondano ad offset su 16 bit.

Nessun dipendente è presente all'interno della ditta alle ore 00:00.

17.13. Analisi delle presenze

Si supponga di avere in memoria, a partire dall'indirizzo `TABLE`, i dati relativi alle presenze sul lavoro dei 1.000 dipendenti di una ditta. Per ogni dipendente sono utilizzati 46 byte, su cui sono registrate le presenze in ditta: ogni bit rappresenta un giorno. Il primo bit del primo byte rappresenta il primo giorno, e così via sino al quarantaseiesimo byte, ove sono registrate le presenze/assenze relative agli ultimi 5 giorni dell'anno. Se l'*i-esimo* bit vale 1, il dipendente era presente sul lavoro nel giorno *i* dall'inizio dell'anno: se vale 0 era assente.

Gli ultimi 3 bit dell'ultimo byte sono privi di significato. Le sequenze di 46 byte relative ai 1000 dipendenti sono memorizzate consecutivamente, in modo da occupare circa 46 Kbyte.

Si scriva una routine Assembler 8086 che sulla base dei dati contenuti in memoria determini in quale giorno lavorativo (vanno esclusi i giorni di sabato e domenica) si è avuta la massima percentuale di assenteismo nella ditta, e visualizzi il numero d'ordine di tale giorno. La routine acquisisce inizialmente da tastiera (si scelga il modo più opportuno) l'indicazione relativa al giorno della settimana corrispondente al primo giorno dell'anno. Si assume che l'anno considerato non sia bisestile.

17.14. Visita in ampiezza di un grafo

Si scriva una procedura Assembler che esegua la visita in ampiezza di un grafo non orientato.

La procedura deve essere richiamabile da un programma C; i suoi parametri sono:

- l'offset della testa di un vettore contenente la descrizione del grafo; il vettore è formato da *n* elementi, ognuno dei quali corrisponde ad un vertice del grafo; l'elemento in posizione *i-esima* contiene i dati relativi al vertice *i*; ogni elemento è costituito da:
 - un intero su 16 bit che contiene il numero di vertici adiacenti;
 - un puntatore (su 16 bit) ad un vettore contenente gli indici dei vertici adiacenti;
- il numero *n* di vertici costituenti il grafo;
- l'indice del vertice da cui deve iniziare la visita.

La procedura restituisce come valore di ritorno il numero di vertici toccati durante la visita; per ogni vertice visitato deve essere visualizzato il relativo indice.

17.15. Analisi di connettività di una grafo

Si scriva una procedura Assembler richiamabile da un programma C che determini se un grafo non orientato è connesso o meno. La procedura ha il seguente prototipo:

```
char connesso (int *vett, int num);
```

dove:

- *vett1* è un vettore di interi che descrive la topologia del grafo; per ogni vertice compare il numero dei vertici adiacenti, seguito dagli indici degli stessi; i vertici compaiono in ordine di indice

crescente (vedi esempio);

- *num* è il numero di vertici componenti il grafo.

La procedura restituisce il valore 1 se il grafo è connesso, 0 altrimenti.

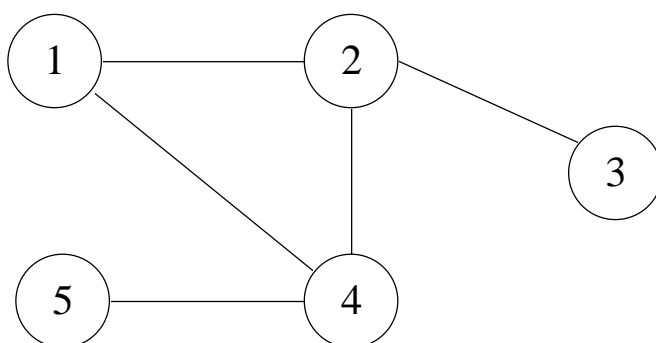
Si utilizzi il seguente algoritmo, basato su un vettore di appoggio *aux* di lunghezza *num* e inizializzato a 0:

- partendo da un vertice qualsiasi, si marca il corrispondente elemento di *aux*, insieme con quelli dei vertici adiacenti;
- per ognuno dei vertici marcati in *aux* si marcano eventuali vertici adiacenti non ancora marcati;
- si ripete sino a che non esistono più vertici non marcati adiacenti a vertici marcati.

Il grafo è connesso se al termine tutti i vertici sono stati marcati.

Esempio

Si consideri il seguente grafo:



Il vettore che descrive il grafo è:

2	vertice 1
2	
4	
3	vertice 2
1	
3	
4	
1	vertice 3
2	
3	vertice 4
1	
2	
5	
1	vertice 5
4	

17.16. Calcolo del ciclo hamiltoniano di lunghezza minima

Si scriva una procedura Assembler richiamabile da un programma C che ritorni la lunghezza del ciclo hamiltoniano di lunghezza minima in un grafo pesato completo.

La procedura ha il seguente prototipo:

```
int min_hamilt_cycle (int size, int *graph);
```

dove *size* è il numero di vertici del grafo, e *graph* è un vettore di interi positivi su 16 bit che rappresenta la matrice di adiacenza (memorizzata per righe) del grafo stesso. Nell'elemento (*i*, *j*) della matrice è memorizzato il peso dell'arco che connette i vertici di indice *i* e *j*, rispettivamente.

17.17. Livellamento di un grafo

Si scriva una procedura Assembler richiamabile da un programma C che esegua il *livellamento* di un grafo orientato. La procedura ha il seguente prototipo:

```
int level (char *graph, int num, int root, int *visited);
```

dove:

- *graph* è una matrice quadrata di caratteri che descrive la topologia del grafo orientato; la matrice è memorizzata per righe ed il valore dell'elemento (i, j) è 1 se i vertici i e j sono connessi, 0 altrimenti;
- *num* è il numero di vertici componenti il grafo;
- *root* è l'indice del vertice da cui far partire l'operazione di livellamento;
- *visited* è un vettore di *num* elementi interi inizializzati a 0, in cui la procedura deve scrivere per ogni vertice il numero del livello corrispondente.

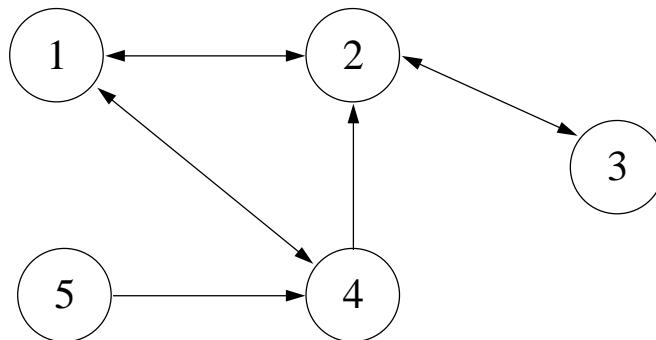
La procedura ritorna il numero di vertici visitati.

Si consiglia di far uso del seguente algoritmo:

- si marca il vertice di partenza con il livello 1;
- per ogni livello
 - per ogni vertice marcato in *visited* con l'indice del passo precedente, si scandiscono i vertici adiacenti;
 - se uno di essi non è ancora stato visitato, lo si marca con il numero del livello attuale
- si ripete sino a che esistono nel grafo vertici adiacenti a quelli del livello precedente e non ancora marcati.

Esempio

Si consideri il seguente grafo:



La matrice che descrive il grafo è:

```

0 1 0 1 0
1 0 1 0 0
0 1 0 0 0
1 1 0 0 0
0 0 0 1 0
  
```

Supponendo che il vertice di partenza sia quello di indice 1, la procedura riempie il vettore *visited* con i seguenti valori:

```
1 2 3 2 3
```

17.18. Calcolo della massima *clique* in un grafo

Si scriva una procedura Assembler richiamabile da un programma C che esegua il calcolo della

dimensione della *clique* di dimensione massima in un grafo.

La procedura ha il seguente prototipo

```
int max_clique (int *graph, int dim);
```

dove *graph* è il puntatore (su 16 bit) ad un vettore di interi che descrive il grafo, e *dim* è il numero di vertici di cui questo è composto.

Il vettore puntato da *graph* contiene *dim* puntatori, ciascuno su 16 bit: il puntatore corrispondente all'*i-esimo* elemento contiene l'indirizzo della testa di un vettore contenente gli indici dei vertici adiacenti a quello *i-esimo*. Il vettore è terminato da un intero di valore 0. Gli indici (ciascuno su 16 bit) variano tra 1 e *dim*.

La procedura ritorna la dimensione della clique massima.

Si consiglia di seguire l'algoritmo implementato dalla seguente procedura recursiva in linguaggio C:

```
void clique (int size)
{
    int i;
    if (size > max)
        max = size;
    for (i=0; i<dim; i++)
        if (maxclique[i] == OUT)
            if (connected_to_all( i))
            {
                maxclique[i] = IN;
                clique (size+1);
                maxclique[i] = OUT;
            }
}
```

dove *maxclique* è un vettore di dimensione *dim* contenente la clique corrente, e la procedura *connected_to_all* ritorna il valore 1 se il vertice il cui indice è passato come parametro è connesso con tutti quelli che fanno parte della clique corrente, 0 altrimenti.

17.19. Calcolo della sottosequenza di costo massimo

Si consideri un insieme di 100 città, ognuna identificata tramite un numero tra 0 e 99. Si supponga che un ipotetico viaggiatore passi il suo tempo visitando le 100 città, sempre nello stesso ordine.

Si scriva una procedura Assembler che, nota la sequenza nella quale il viaggiatore visita le 100 città, determini, tra tutte le sottosequenze di *n* città visitate consecutivamente, quella durante la quale il viaggiatore percorre la massima distanza.

La procedura deve essere richiamabile da un programma C; i suoi parametri sono:

- l'offset della testa di un vettore, di dimensione pari a 200 word, contenente, per ogni città, la coppia di coordinate che ne individuano la posizione su un piano cartesiano (ogni coordinata è rappresentata in complemento a 2 su una word);
- l'offset della testa di un secondo vettore, di dimensione pari a 100 word, contenente la sequenza secondo la quale il viaggiatore visita le città;
- il numero *n*, che può assumere valori tra 2 e 50.

La procedura visualizza la sottosequenza cercata.

Si scriva la procedura in modo da verificare la presenza di un *overflow* nel calcolo della distanza.

17.20. Torre di Hanoi

Si scriva un programma Assembler che risolva il rompicapo noto come *Torre di Hanoi*. Esso si

svolge utilizzando 3 pioli ed un numero n di dischi, di dimensioni diverse; all'inizio i dischi sono posizionati sul primo piolo, in ordine di diametro crescente; lo scopo del gioco è trovare una sequenza con cui spostare (uno alla volta) i dischi, in modo tale che, al termine, essi si trovino tutti su un altro piolo, di nuovo ordinati per diametro crescente. Durante gli spostamenti si deve fare in modo che su ciascun piolo non vi sia mai un disco più grande al di sopra di uno più piccolo.

Il numero n di dischi con cui lavorare deve essere letto all'inizio da tastiera.

Si consiglia di usare come traccia il seguente programma C:

```
#include <stdio.h>
void hanoi( int, int, int);
int mosca = 1;
main()
{
    int n;
    printf( "Numero dischi: ");
    scanf( "%d", &n);
    hanoi( n, 0, 2);
}
void hanoi( int n, int source, int dest)
{
    int aux;
    if( n > 0)
    {
        aux = 3 - (source + dest);
        hanoi( n - 1, source, aux);
        printf( "%d %d %d \n", mosca++, source, dest);
        hanoi( n - 1, aux, dest);
    }
}
```

17.21. Memorizzazione di una data in forma compatta

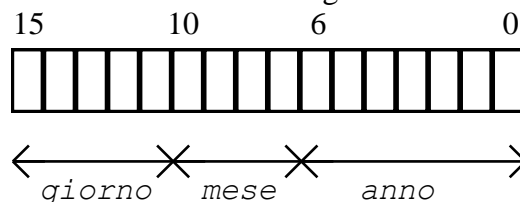
Si scriva una procedura Assembler richiamabile da un programma C che esegua la trasformazione necessaria per memorizzare una data in forma compatta all'interno di una word.

La procedura ha il seguente prototipo:

```
int pack (int anno, char *mese, int giorno);
```

La data viene passata alla procedura *pack* attraverso i tre parametri *anno*, *mese*, *giorno*, dove *anno* e *giorno* sono interi su 16 bit, e *mese* è una stringa contenente il nome del mese, terminato da un carattere '\0'.

La procedura compatta la data su 16 bit secondo il seguente formato:



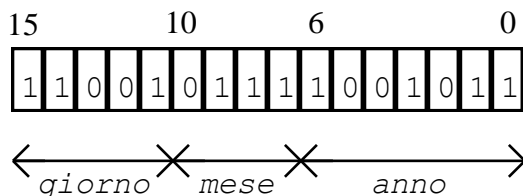
L'anno viene memorizzato assumendo che esso appartenga a questo secolo e memorizzando quindi solo la differenza rispetto a 1900. Il mese viene memorizzato in forma di numero tra 1 e 12.

Esempio

Si supponga di avere:

- *giorno* = 25
- *mese* = 'luglio'
- *anno* = 1975

Il risultato dell'impaccamento è:



17.22. Conversione da data a numero d'ordine del giorno

Si scriva una procedura Assembler richiamabile da C che esegua la conversione da data a numero d'ordine del giorno all'interno dell'anno. La procedura ha il seguente prototipo:

```
int dton (int day, char *month, int year);
```

Si assuma che gli anni divisibili per 4 e non divisibili per 100 siano bisestili.

La procedura deve ritornare il valore -1 in caso di errore nei parametri passati.

Esempio

La chiamata

```
c = dton( 23, "febbraio", 1990);
```

pone in *c* il valore 54.

17.23. Conversione da data a numero d'ordine della settimana

Si scriva una procedura Assembler richiamabile da C che esegua la conversione da data a numero d'ordine della settimana. La procedura ha il seguente prototipo:

```
int dtoweeek (int day, char *month, int year);
```

È noto che la prima settimana del 1994 comincia dal 3 Gennaio e che gli anni divisibili per 4 e non divisibili per 100 siano bisestili.

La procedura deve ritornare il valore -1 in caso di errore dei parametri passati.

Esempio

La chiamata

```
c = dtoweeek( 19, "aprile", 1994);
```

pone in *c* il valore 16.

17.24. Inserimento di un elemento in un albero binario di ricerca

Si scriva una procedura Assembler richiamabile da un programma C che esegua l'inserzione di un elemento all'interno di un albero binario di ricerca. L'elemento corrisponde ad un record composto da 2 campi di dato:

- una stringa terminata da un carattere '\0', di lunghezza massima 20 byte
- un intero su 16 bit.

L'albero memorizza una struttura dati nella quale la chiave è costituita dal campo intero.

La procedura ha il seguente prototipo:

`void insert (void **head, char *string, int value, void *block);`
 dove *head* è il puntatore alla variabile contenente il puntatore alla testa dell'albero, *string* è il puntatore alla stringa, e *value* l'intero da memorizzare nell'elemento da inserire. Il parametro *block* rappresenta il puntatore ad un blocco di 26 byte contigui da utilizzare per la memorizzazione del nuovo record (20 byte per la stringa, 2 per l'intero, 2+2 per i due puntatori); in tal modo la procedura *insert* non ha la necessità di allocare memoria.

Si assuma che tutti i dati siano memorizzati nello stesso segmento, in modo che i puntatori corrispondano ad offset su 16 bit.

17.25. Shell sort

Si scriva una procedura Assembler richiamabile da un programma C che esegua l'ordinamento di un vettore di interi utilizzando l'algoritmo noto come *shell sort*. La procedura ha il seguente prototipo:

```
void sort(int *vett, int lun);
```

dove *vett* è il puntatore al vettore, che contiene *lun* interi su 16 bit.

Per facilitare la scrittura della procedura Assembler si riporta, nel seguito, la corrispondente procedura C:

```
void sort (int *vett, int lun)
{ int gap, i, j, temp;
  for (gap=lun/2; gap>0; gap/=2)
    for (i=gap; i<lun; i++)
      for (j=i-gap; j>=0 && vett[j]>vett[j+gap]; j-=gap)
        swap (vett, j, j+gap);
}
```

La procedura *swap* esegue lo scambio di posizione tra l'elemento di indice *j* e quello di indice *j+gap* all'interno del vettore *vett*.

17.26. Selection sort

Si implementi una procedura Assembler richiamabile da un programma C che esegua l'ordinamento di un vettore di interi su 32 bit, utilizzando l'algoritmo noto come *selection sort*.

Il prototipo della procedura è

```
void ssort (long int *vett, int n);
```

dove *vett* è il vettore da ordinare, ed *n* il numero di interi (su 32 bit) che esso contiene.

L'algoritmo di *selection sort* consiste nell'eseguire sul vettore da ordinare $n-1$ passi, in ognuno dei quali viene ricercato l'elemento da sistemare nella posizione *i*-esima, scegliendo quello più piccolo tra gli ultimi $n-i$ elementi del vettore.

Si consiglia di seguire per la procedura Assembler lo stesso schema seguito nella seguente procedura C, che implementa l'algoritmo richiesto:

```
void ssort (long int *vett, int n)
{
    int small, i, j;
    for (i=0; i<n; i++)
    {
        small = i;
        for (j=i+1; j<n; j++)
            if( vett[j] < vett[small])
                small = j;
        swap (vett, small, i);
    }
}
```

La procedura *swap* esegue lo scambio di posizione tra l'elemento di indice *small* e quello di indice *i* all'interno dell vettore *vett*.

17.27. Analisi di una matrice sparsa

Si abbia una matrice quadrata i cui elementi sono valori rappresentati su 32 bit, dei quali i 16 più significativi costituiscono la parte intera (rappresentata in complemento a due) e i 16 meno significativi la parte frazionaria, assumendo che il bit più significativo abbia peso 2^{-1} e quello meno significativo abbia peso 2^{-16} .

La matrice è memorizzata sotto forma di matrice sparsa ed è strutturata in modo tale che, per ogni elemento non nullo, sono forniti tre valori:

- primo indice (16 bit);
- secondo indice (16 bit);
- valore (32 bit).

Gli elementi nulli vengono ignorati; quelli non nulli sono invece memorizzati per righe, secondo il formato descritto, in un vettore composto da $8 \cdot \text{MAX}$ byte, supponendo che la matrice non abbia mai un numero di elementi diversi da 0 superiore a MAX.

Si scriva una procedura Assembler richiamabile da un programma C avente come parametri il nome del vettore contenente la matrice, la dimensione di questa e il numero di elementi non nulli memorizzati nel vettore. La procedura deve determinare per ogni elemento, anche nullo, se questo è un elemento di *sella*.

Un elemento si definisce *di sella* se sono contemporaneamente vere le due condizioni:

- sulla riga cui appartiene è quello con valore minimo;
- sulla colonna cui appartiene è quello con valore massimo.

Per ogni elemento di *sella* la routine visualizza i due indici ed il valore, approssimato all'intero più vicino.

17.28. Ordinamento delle colonne di una matrice

Si scriva una procedura Assembler richiamabile da un programma C che esegua l'ordinamento delle colonne di una matrice di interi.

La procedura ha il seguente prototipo:

```
void matsort (int *mat, int m, int n);
```

La procedura manipola la matrice di interi, memorizzata per righe nel vettore *mat*, ordinando ogni colonna in ordine crescente. I parametri *m* ed *n* sono rispettivamente il numero di righe e di colonne della matrice.

Esempio

La matrice passata come parametro sia la seguente:

3	7	2	11
5	4	1	-6
15	23	-19	0
22	-9	12	11
3	1	20	-1

La procedura la trasforma nella seguente maniera:

3	-9	-19	-6
3	1	1	-1
5	4	2	0
15	7	12	11
22	23	20	11

17.29. Somma delle diagonali

Si scriva una procedura Assembler per il calcolo dei valori corrispondenti alla somma degli elementi sulle diagonali di una matrice.

La matrice si suppone composta di *n* righe ed *n* colonne; ciascun elemento della matrice è di tipo intero su 16 bit. Le somme corrispondono a interi su 32 bit che vanno scritti in due vettori di *n*+1 elementi, uno per gli elementi sulle diagonali *diritte*, l'altro per quelli sulle diagonali *rovesciate*.

Si noti che per tutti gli elementi su una stessa diagonale *diritta* la somma delle coordinate è data da uno stesso valore, mentre per tutti quelli su una stessa diagonale *rovesciata* è costante il valore della differenza delle coordinate.

La procedura riceve in AX il valore *n*, in BX l'offset della matrice (memorizzata per righe) e in SI e DI gli offset dei due vettori risultato, che si suppongono già allocati di dimensioni sufficienti.

Si suppone che sia la matrice, sia i due vettori risiedano nello stesso segmento di dato.

17.30. Prodotto di polinomi

Si scriva una procedura Assembler richiamabile da un programma C che esegua il prodotto di due polinomi a coefficienti interi.

La procedura ha il seguente prototipo:

```
int poliprod (int *p1, int n1, int *p2, int n2, long int *p3);
```

I due polinomi sono memorizzati ciascuno in un vettore di interi; per ogni coefficiente non nullo il vettore contiene una coppia di valori rappresentanti rispettivamente il coefficiente e la potenza del termine corrispondente. Le coppie sono memorizzate in ordine di potenza decrescente. I numeri di coefficienti non nulli dei polinomi *p1* e *p2* sono contenuti rispettivamente in *n1* ed *n2*. La procedura calcola il polinomio risultato e lo scrive, con lo stesso formato, nel vettore *p3*, ritornando il numero di coefficienti non nulli. I coefficienti del vettore *p3* sono rappresentati ciascuno su 32 bit. Il vettore *p3* si suppone già allocato di lunghezza sufficiente.

Esempio

I due polinomi

$$f_1(x) = 5x^6 - 8x^3 - 5 \qquad f_2(x) = 3x^5 + 2x^3 - 6$$

hanno come prodotto il polinomio

$$f_3(x) = 15x^{11} + 10x^9 - 24x^8 - 46x^6 - 15x^5 + 38x^3 + 30$$

I tre polinomi sono memorizzati con i seguenti tre vettori:

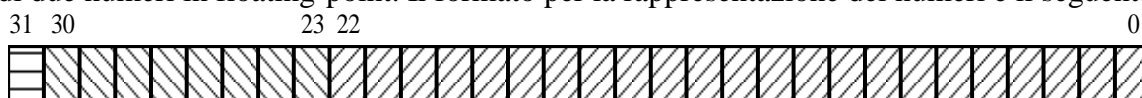
5 6 -8 3 -5 0

3 5 2 3 -6 0

15 11 10 9 -24 8 -46 6 -15 5 38 3 30 0

17.31. Moltiplicazione tra numeri in floating-point (I versione)

Si scriva una procedura Assembler richiamabile da un programma C che esegua la moltiplicazione di due numeri in floating-point. Il formato per la rappresentazione dei numeri è il seguente:



Segno Esponente Mantissa

L'esponente è rappresentato complemento a 2 e la mantissa è rappresentata in modulo ed è normalizzata.

L'algoritmo di moltiplicazione tra numeri in floating-point è così riassumibile:

$$X \times Y = (X_M \times Y_M) \times 2^{(X_E + Y_E)}$$

Il prototipo della procedura è il seguente:

```
float mult ( float x1, float x2 );
```

Si supponga che il risultato sia rappresentabile su 32 bit.

17.32. Moltiplicazione tra numeri in floating-point (II versione)

Si ripeta l'esercizio precedente, tenendo invece conto che:

- l'esponente è rappresentato in eccesso 127
- la mantissa è rappresentata in modulo ed è normalizzata; il bit più significativo, essendo necessariamente 1, è sottinteso.

17.33. Determinazione della temperatura a regime di una piastra

Si consideri una piastra metallica suddivisa in 10.000 elementi omogenei, secondo una maglia a forma di quadrato di lato pari a 100 elementi. La temperatura di ognuno degli elementi si suppone costante all'interno dell'elemento e, nel generico istante, è pari alla media di quelle degli 8 elementi adiacenti.

Si scriva un programma Assembler che calcoli la temperatura a regime di ogni elemento della piastra, supponendo che:

- le temperature degli elementi di bordo (colonna e riga 0, colonna e riga 99) sono costanti al valore 0 (ossia non dipendono dagli elementi intorno).
- la temperatura dell'elemento in posizione (0,0) è anch'essa costante, e viene introdotta da tastiera, variando tra 0 e 1.000.

Il programma esegue una serie di iterazioni, in ognuna delle quali calcola la temperatura di ogni elemento (tranne quelli di bordo), sulla base delle temperature degli elementi vicini, all'iterazione precedente. Il processo si arresta allorché l'ultima iterazione non ha portato alcuna modifica alle

temperature degli elementi.

Si eseguano i calcoli assumendo che la temperatura del singolo elemento sia rappresentata da un intero su 16 bit, e che tutti gli elementi partano da una temperatura pari a 0.

Al termine, il programma scrive su video, per ogni elemento, il valore delle coordinate e quello della temperatura finale.

17.34. Prenotazione posti

Si vuole realizzare un sistema per la prenotazione dei posti sul treno Torino-Pisa. Il treno è composto di 6 carrozze, ognuna contenente 100 posti. Le fermate intermedie effettuate sono 4: Asti, Alessandria, Genova, La Spezia.

Viene preso in considerazione il solo viaggio Torino-Pisa (non il ritorno) e si suppone che esso abbia frequenza giornaliera. Il sistema permette la prenotazione dei posti nei 3 mesi successivi a quello attuale.

Si richiede nell'ordine:

1. la definizione della struttura dati utilizzata; tenendo presente che essa non deve comunque superare i 64 Kbyte di occupazione totale, si consiglia di usare un vettore di 600 byte per ciascuna corsa, riservando un byte ad ogni posto e al suo interno un bit per ognuna delle tratte;
2. la procedura per la prenotazione di un posto; essa riceve come parametri di ingresso:
 - a. l'indicazione del giorno e mese in cui si vuole effettuare il viaggio; il giorno viene indicato con un numero intero tra 1 e 31, il mese con un intero tra 1 e 3;
 - b. l'indicazione della tratta su cui si vuole prenotare il posto, sotto forma di una coppia di numeri interi compresi tra 1 e 6, indicanti rispettivamente la stazione di partenza e quella di arrivo.

I parametri di output sono il numero della carrozza e del posto prenotato; nel caso non esistano posti liberi sulla tratta richiesta vengono ritornati due valori nulli. Si tenga presente che una generica richiesta di prenotazione non può essere soddisfatta se non esiste alcun posto che sia libero su tutte le sottotratte componenti la tratta richiesta;

3. il programma principale, che:
 - legge da tastiera i 4 numeri corrispondenti a giorno, mese, stazione di partenza e stazione di arrivo (separati tra loro da uno spazio; il quarto numero è seguito da una coppia di caratteri CR e LF);
 - richiama la procedura di cui sopra;
 - visualizza i due numeri corrispondenti al vagone e al posto prenotato (0 se non vi sono posti liberi).

Il programma si suppone senza fine. Si assuma che i dati inseriti da tastiera siano corretti.

17.35. Gestione di un ospedale

Si realizzi un programma Assembler per la gestione di un ospedale.

L'ospedale dispone di 2.000 letti; ciascuno di essi può essere o libero o occupato da un paziente. Per ogni paziente esiste un calendario di controlli da effettuare in giorni successivi.

Il programma accetta da tastiera i seguenti comandi:

- A: segnala l'arrivo di un nuovo paziente. Viene visualizzato il numero del primo letto libero a cui egli viene assegnato.
- V n m : richiede l'inserzione di un controllo sul paziente nel letto n per il giorno m . I giorni sono numerati da 1 a 365. Il programma deve segnalare se si richiede erroneamente un con-

trollo per un letto vuoto, o se la stessa richiesta era già stata fatta in precedenza. In questi due casi nessuna nuova richiesta viene memorizzata internamente.

- $R\ n$: causa la visualizzazione dell'elenco dei giorni per cui è stata richiesta una visita per il paziente nel letto n .
- $U\ n$: segnala l'uscita dall'ospedale del paziente nel letto n , che deve quindi essere marcato come libero. Tutte le richieste di controlli per tale letto devono essere cancellate.

Si supponga che ogni comando sia sintatticamente corretto e sia concluso da un CR. Si assuma sempre la presenza di uno spazio tra i vari elementi del comando. Il programma si suppone senza fine. All'inizio tutti i letti sono liberi.

Per la memorizzazione dei controlli richiesti su ogni paziente si consiglia l'uso di una struttura dati di tipo lista.

17.36. Sistema *client-server*

Si consideri un generico sistema composto da 100 utenti (*client*) e 10 operatori (*server*), ognuno contraddistinto da un numero identificativo che va rispettivamente da 1 a 100 e da 1 a 10, rispettivamente. I primi richiedono dei servizi, i secondi li forniscono, soddisfacendo però le richieste di un solo utente alla volta.

Gli istanti di arrivo delle richieste da parte degli utenti non sono prevedibili, così come non lo sono i tempi necessari agli operatori per soddisfare ognuna delle richieste degli utenti.

Ogni volta che un utente ha una richiesta di servizio, fa una richiesta di interruzione che esegue una routine Assembler di nome SERV1 avente come parametro di ingresso, nella locazione di indirizzo PAR1, il suo numero d'ordine. La routine restituisce all'indirizzo PAR2 il numero dell'operatore che da più tempo ha concluso il suo lavoro, oppure 0 se nessuno degli operatori è libero all'istante corrente.

Ogni volta che un operatore conclude il servizio per un utente e ritorna libero, chiama a sua volta una routine SERV2, passando come parametro di input il suo numero d'ordine, nella locazione di indirizzo PAR3. La routine ha un parametro di output, che essa scrive all'indirizzo PAR4, corrispondente al numero dell'utente che da più tempo ha richiesto un servizio, ed ancora non l'ha ottenuto. Se non vi sono richieste pendenti, la routine ritorna il valore 0.

Si scrivano le due routine SERV1 e SERV2, supponendo che:

- nessuna delle due possa venire interrotta da una nuova richiesta di interrupt;
- inizialmente tutti gli operatori siano liberi e non vi siano richieste pendenti;
- un utente non possa richiedere un nuovo servizio sino a che non è stata completamente soddisfatta la sua precedente richiesta.

Si scriva infine un programma per la verifica della correttezza delle due routine; tale programma deve:

- acquisire da tastiera l'indicazione di quale delle due routine richiamare, e con quale parametro di ingresso;
- chiamare la corrispondente routine;
- visualizzarne il parametro di output.

17.37. Classifica di una gara automobilistica

Si scriva una procedura Assembler di servizio dell'interrupt per la gestione della classifica di una gara automobilistica.

La procedura viene attivata al momento del passaggio di una macchina in corrispondenza del traguardo. La procedura legge i seguenti dati:

1. il numero d'ordine della macchina che transita davanti alla cellula, memorizzato in un byte nella locazione di memoria 00200:00002;
2. il tempo d'orologio di passaggio, memorizzato nella locazione di memoria 00200:00003 in un record contenente tre campi di dimensione un byte in cui vengono memorizzati ora, minuto primo e minuto secondo ed un campo di dimensione una word contenente i millesimi di secondo.

La procedura di servizio dell'interrupt deve leggere i dati relativi al passaggio di una macchina e deve tenere aggiornata in memoria una struttura dati che memorizza la classifica generale della competizione.

Inoltre la procedura deve tenere aggiornata una struttura dati che memorizza il miglior tempo sul giro per ogni macchina.

17.38. Sistema di avvistamento radar

Si consideri un sistema di avvistamento aereo basato su un radar in grado di coprire un'area di dimensione $30 \times 30 \text{ Km}^2$. Le informazioni che giungono dal radar devono venire elaborate in modo da determinare quanti aerei si trovano all'interno della zona coperta e quale sia la loro posizione.

Ad ogni avvistamento il radar invia alla centrale di elaborazione un segnale di interrupt, dopo aver scritto nella memoria del sistema (alle locazioni indicate) 4 interi corrispondenti alle seguenti informazioni:

- TIME: istante di avvistamento, espresso in secondi trascorsi da un istante di riferimento;
- POSX, POSY: coordinate orizzontali dell'aereo avvistato, espresse in metri e quindi comprese tra i valori -15.000 e +15.000;
- HIGH: altezza del punto di avvistamento, espressa in metri e inferiore a 20.000.

Si scriva la routine Assembler di gestione dell'interrupt, in modo che le informazioni prodotte dal radar vengano gestite secondo il seguente algoritmo: per ogni avvistamento in un punto P_2 individuato dalle tre coordinate POSX, POSY e HIGH si controlli se si è verificato un precedente avvistamento in un punto P_1 da cui P_2 possa, noti i tempi t_1 e t_2 dei due avvistamenti, essere raggiunto con velocità inferiore a V_{max} . In tal caso si suppone che i due avvistamenti siano relativi allo stesso aereo. Se dopo T_{max} secondi dall'avvistamento di un aereo non ne segue un altro relativo allo stesso aereo, si suppone che esso sia uscito dalla zona coperta dal sistema. V_{max} e T_{max} sono costanti intere su 16 bit, con dimensioni $[m/s]$ e $[s]$, rispettivamente.

Ogni volta che viene chiamata, la routine visualizza le informazioni aggiornate relative alla posizione degli aerei nella zona coperta. Per fare questo si consiglia l'uso di una struttura (buffer o lista) contenente i tempi e le coordinate degli avvistamenti relativi agli aerei che si suppone si trovino all'interno dell'area coperta. L'aggiornamento della struttura avviene utilizzando le seguenti regole:

- se si ha un avvistamento non riconducibile a nessuno di quelli precedenti, lo si inserisce nella struttura;
- avvistamenti risalenti ad istanti passati da più di t_{max} secondi vanno cancellati dalla struttura;
- se ad un avvistamento ne segue un altro relativo allo stesso aereo, il vecchio avvistamento viene sostituito da quello nuovo.

Nel caso che un nuovo avvistamento possa essere ricondotto a più di uno tra quelli precedenti, si scelga tra questi quello a distanza minima.

Si supponga che tra un interrupt ed il successivo trascorra sempre un tempo sufficiente alla completa esecuzione della routine.

Si ricorda che, nello spazio tridimensionale, la distanza d tra due punti $A(x_1, y_1, z_1)$ e $B(x_2, y_2, z_2)$ è data da

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}.$$

17.39. Gara ciclistica

Si supponga di dover gestire il sistema di cronometraggio per una gara ciclistica di velocità che si svolge su un circuito che i concorrenti devono percorrere più volte. Lungo il tracciato sono stati predisposti 50 punti di cronometraggio che registrano l'istante in cui ogni concorrente transita da quel punto e lo inviano ad un sistema centralizzato di elaborazione. Questo è costituito da un sistema 8086 su cui viene attivato un interrupt ogni volta che ad esso perviene l'informazione relativa al transito di un concorrente per un punto di cronometraggio. Tale informazione è registrata in memoria in 3 parole poste agli indirizzi `POSIZ`, `CONCORR` e `TEMPO` che indicano, rispettivamente, il punto di cronometraggio (numerati da 1 a 50), il numero del concorrente (da 1 a 100) e l'istante di transito; quest'ultimo è rappresentato come tempo trascorso dalla partenza, espresso in minuti (1 byte) e secondi (1 byte).

Si scriva la sola routine di servizio dell'*interrupt*, la quale deve eseguire le seguenti operazioni:

- calcolare la velocità media sostenuta dal concorrente nel tratto tra il punto di cronometraggio da cui proviene la segnalazione ed il precedente, utilizzando le informazioni contenute nella tabella posta alla locazione `DIST`, in cui sono scritte le distanze (in metri) tra un punto di cronometraggio ed il precedente;
- visualizzare tale velocità (espressa in *m/s*) ed il numero del relativo concorrente, se essa è la più alta fatta registrare fino a quel momento su un qualsiasi tratto del circuito da un qualsiasi concorrente;
- visualizzare la situazione di gara ogni volta che un concorrente completa per primo un nuovo giro, ossia ogni volta che l'informazione proviene dal punto di cronometraggio posto sul traguardo ed è relativa al concorrente in quel momento in testa; in tal caso si deve visualizzare la situazione di ogni concorrente, espressa come ritardo (in minuti e secondi) rispetto al primo; per fare questo si considera l'ultima segnalazione relativa ad ogni concorrente pervenuta sino a quel momento.

Si facciano le seguenti ipotesi:

- il punto di cronometraggio numero 1 sia il primo che si incontra dopo la partenza, ed il numero 50 quello posto sul traguardo;
- la routine di gestione dell'*interrupt* è così veloce, da rendere nulla la probabilità che essa venga interrotta da un'altra richiesta di interrupt.

17.40. Istogramma orizzontale

Si scriva un programma Assembler che realizzi l'istogramma corrispondente ad un insieme di numeri.

L'insieme è composto da 10 numeri letti da tastiera e di valore compreso tra 0 e 20. Il programma deve costruire una matrice di caratteri di dimensione 20x10 contenente l'istogramma verticale corrispondente e visualizzare su video il risultato stampando un numero opportuno di caratteri '*'.

Esempio

Si supponga di leggere da tastiera i seguenti 10 numeri:

3 5 7 2 4 8 20 3 4 9

L'istogramma che deve essere visualizzato è il seguente:

[illegible]

17.41. Analisi di un circuito

Si consideri un circuito combinatorio e si supponga che tutti i suoi elementi (compresi gli ingressi) siano numerati in modo progressivo.

La descrizione del circuito è contenuta in memoria a partire dall'indirizzo CIRC. Tale descrizione è strutturata in blocchi, tanti quanti sono gli elementi del circuito. Ogni blocco comprende un numero variabile di interi su 16 bit: il primo corrisponde al numero di porte alimentate dall'elemento in questione, mentre i successivi rappresentano ognuno l'indice di una di tali porte. Si tenga presente che anche gli ingressi della rete sono considerati come elementi, e quindi a ciascuno di essi corrisponde un blocco nella descrizione. Le uscite del circuito sono caratterizzate dal fatto che i corrispondenti blocchi contengono un solo intero, avente valore 0.

Si richiede una procedura *Assembler* che, leggendo in memoria la descrizione del circuito e ricevendo come parametro l'indice di un suo elemento, visualizzi gli indici di tutte le uscite raggiungibili a partire da tale elemento.

Per la soluzione del problema si consiglia di usare il seguente algoritmo: a partire dall'elemento fornito come parametro, si marcano tutti gli elementi da questo direttamente alimentati; poi si marcano tutti gli elementi alimentati da quelli marcati, e così via, sino ad arrivare alle uscite. A questo punto le uscite marcate sono quelle raggiungibili dall'elemento di partenza.

Esempio

Si consideri il circuito in Fig. 17.1; la relativa rappresentazione in memoria è:

```

1 8
1 5
2 5 6
2 6 7
1 11
2 8 9
3 9 10 11
1 10
1 12
2 12 13
2 13 14
1 14
0
0
0

```

dove ogni riga corrisponde ad un blocco, a sua volta corrispondente ad un diverso elemento del circuito.

La chiamata della procedura con parametro 2 restituirebbe come risultato gli indici 9 e 10, con parametro 8 il solo indice 10.

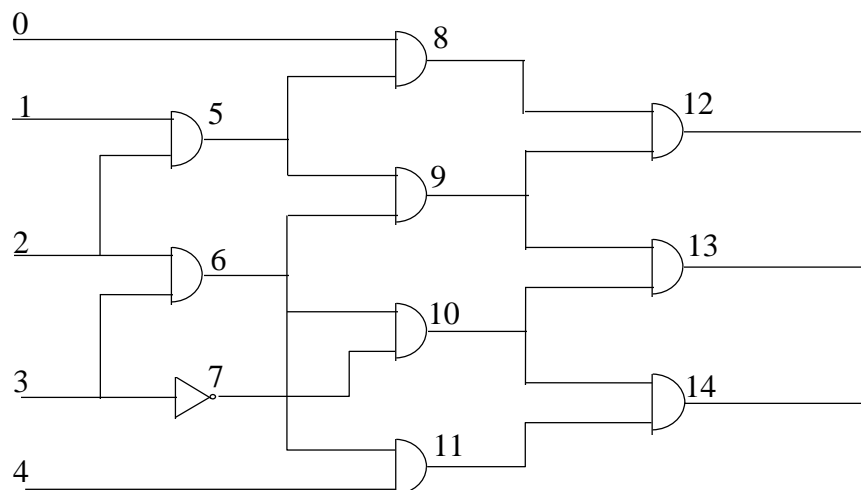


Fig. 17.1: Esempio di circuito.

17.42. Crittografia di un testo

Si scriva una procedura Assembler richiamabile da linguaggio C che esegua la crittografia di un testo.

L'algoritmo di crittografia utilizza una tabella di traslazione che ad ogni carattere dell'alfabeto fa corrispondere un nuovo carattere; il carattere sostitutivo viene letto nella tabella in corrispondenza del carattere vecchio incrementato di una quantità pari alla lunghezza della parola. Gli spazi non vengono crittografati.

Il prototipo della procedura è il seguente:

```
int critto (char *testo, char *critto, char *tab, int car);
```

I parametri `testo` e `critto` corrispondono rispettivamente all'indirizzo di inizio del testo originale e di quello crittografato (che si suppone essere correttamente allocato preventivamente); il

parametro `tab` contiene l'indirizzo del vettore di traslazione di dimensione pari al valore `car`. La procedura restituisce il numero di parole crittografate.

Esempio

Sia dato il seguente testo:

Riprende la seduta con le dichiarazioni

Sia inoltre dato il seguente vettore di traslazione:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
M	S	G	H	A	Z	J	O	V	D	I	R	B	T	X	N	Y	L	Q	F	U	E	P	K	C	W

Il corrispondente testo crittografato risulta essere:

wykwberb tg cidmwj zly tj yenuetatbesme

17.43. Elaborazione di un'immagine

Si scriva una procedura Assembler richiamabile dal C che esegua l'elaborazione di un'immagine. L'immagine ha le seguenti caratteristiche:

- dimensione *64x64 pixel*;
- per ogni pixel è memorizzato un numero intero senza segno su 32 bit che memorizza la sua luminosità;
- la matrice di pixel è memorizzata per righe in un vettore.

Si definiscono come *primi vicini* gli 8 pixel che per entrambe le coordinate differiscono al più di una sola unità.

La procedura deve verificare la correttezza di un'immagine contenuta in memoria, verificando che la luminosità del pixel differisca dalla media delle luminosità degli 8 primi vicini di un valore inferiore ad una soglia stabilita.

Il prototipo della procedura è il seguente:

```
char check (unsigned long *vet, unsigned long thresh, int *x, int *y);
```

Il primo parametro indica l'indirizzo di inizio del vettore contenente l'immagine in memoria, il secondo parametro indica la soglia massima, il terzo e quarto parametro indicano le coordinate del primo punto che non soddisfa le specifiche.

La procedura restituisce un codice di errore: 0 se tutti i pixel soddisfano la specifica di luminosità (in tal caso i parametri `x` e `y` non hanno significato) e 1 se esiste almeno un punto che non soddisfa le specifiche (in tal caso `x` e `y` contengono le coordinate del punto).

I punti sui bordi vanno confrontati con i soli vicini validi.

17.44. Correzione di un questionario

Si scriva una procedura Assembler 8086, richiamabile da programma scritto in linguaggio C, che esegue la correzione automatica di un questionario per studenti. Il questionario è costituito da un numero di domande passato come parametro alla procedura (`n_domande`) aventi ciascuna 5 possibili risposte.

Il prototipo della procedura è il seguente:

```
int correct(struct quest *table, int n_stud, int n_domande);
```

Per ogni studente è memorizzato un vettore di lunghezza pari a `n_domande` byte: i 5 bit più significativi di ogni byte memorizzano le risposte date per ogni domanda. Un bit a 1 indica che la corrispondente risposta è stata scelta; una risposta contenente più bit ad 1 deve essere considerata *non valida*.

Un vettore con analogo formato contiene le risposte esatte al questionario.

La procedura deve, per ogni studente, calcolare la correttezza delle risposte seguendo il seguente criterio di valutazione: +4 punti per ogni risposta esatta, -4 punti per ogni risposta errata, 0 per mancata risposta oppure risposta *non valida*.

Il parametro `table` contiene il puntatore ad un vettore di `n_stud+1` elementi del tipo `struct quest` così definito:

```
struct quest
{
    int matr;
    char *risposte;
    int punteggio;
};
```

Il primo elemento del vettore `table` contiene il puntatore al vettore delle risposte esatte al questionario.

La procedura restituisce come risultato il numero di studenti che hanno ottenuto un punteggio maggiore di zero.

17.45. Calcolo della media dei voti

Si scriva una procedura Assembler richiamabile da C che calcoli la media dei voti conseguiti da un insieme di studenti universitari.

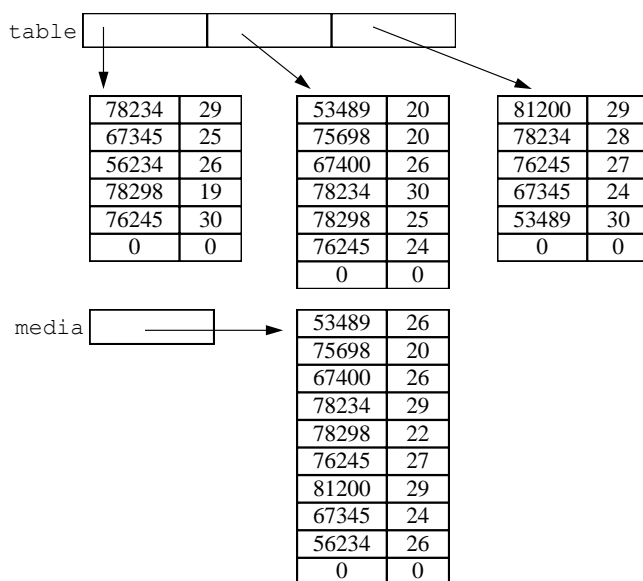
Per ogni esame i dati sono memorizzati in una matrice composta da `n` righe e 2 colonne: nella prima colonna sono memorizzati i numeri di matricola e nella seconda colonna sono memorizzati i voti compresi tra 0 e 30. Ogni matrice di dati è terminata da un record avente i 2 campi nulli. Le matrici sono memorizzate per righe in vettori di lunghezze opportune.

Il prototipo della procedura è il seguente

```
void voti (int **table, int n, int *media);
```

Il parametro `table` rappresenta un vettore di puntatori alle tabelle di esami; il numero di tabelle è pari ad `n` ed il parametro `media` specifica il puntatore al vettore contenente i risultati.

La procedura deve leggere le tabelle dei voti degli esami e deve scrivere, per ogni studente, la media dei voti degli esami sostenuti sotto forma di numero intero, come è mostrato nella figura seguente, in cui si suppone che `n` valga 3.



18. Bibliografia

1. J. L. Antonakos, *An Introduction to the Intel Family of Microprocessors*, Macmillan Publishing Company, New York (USA), 1993
2. B. Brey, *Assembly language programming*, Macmillan Publishing Company, New York (USA), 1994
3. R. C. Detmer, *Introduction to 80x86 Assembly Language and Computer Architecture*, Jones & Bartlett Publishers, Burlington, Massachusetts (USA), 2009 (2nd edition)
4. *Emu8086: 8086 microprocessor emulator integrated disassembler*,
<http://ziplib.com/emu8086>
5. G. W. Gorsline, *16-Bit Modern Microcomputers*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (USA), 1986
6. R. Hyde, *The Art of Assembly Language*, No Starch Press, San Francisco, California (USA) 2010 (2nd edition)
7. H. Harley, *Assembler Inside & Out*, Osborne McGraw-Hill, Berkeley (USA), 1992
8. K. R. Irvine, *Assembly Language for x86 Processors (6th Edition)*, Prentice Hall, Upper Saddle River, New Jersey (USA), 2010
9. Y. Liu, G. Gibson, *Microcomputer System: the 8086/8088 Family*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (USA), 1986
10. M. A. Mazidi, J. Gillispie-Mazidi, *80X86 IBM PC and Compatible Computers: Assembly Language, Design and Interfacing*, Prentice Hall, Englewood Cliffs, New Jersey (USA) (4th edition)
11. *Microsoft MASM Programmer's Guide*, Assembly-Language Development System Version 6.1, Microsoft Corporation, Redmond (USA), 1992
12. *Microsoft MS-DOS Programmer's Reference*, Microsoft Press, Redmond (USA), 1991
13. P. Prinetto, M. Sonza Reorda, *Esercizi di programmazione in Assembler 8086/8088*, Libreria Editrice Universitaria Levrotto&Bella, Torino, 1990
14. M. Saleh, *Practical considerations towards virtual engineering education at undergraduate level*, IEEE Conference on Emerging Technologies and Factory Automation, 2005, pp. 35-39

15. L. Scanlon, *IBM PC & XT Assembly Language*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (USA), 1983
16. W. A. Triebel, A. Singh, *The 8088 and 8086 Microprocessors*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (USA), 1991
17. Wikibooks contributors, *x86 Disassembly*, Createspace, Charleston, South Carolina (USA) 2001 (anche disponibile all'indirizzo http://en.wikibooks.org/wiki/X86_Disassembly)

Appendice: *Instruction Set* 8086

Si forniscono ora alcune tabelle di consultazione rapida con la descrizione dell'insieme di istruzioni del microprocessore Intel 8086. Per ogni istruzione vengono fornite le seguenti informazioni:

- tipi di operandi ammessi
- dimensione, espressa in numero di byte (colonna *Byte*)
- tempo di esecuzione, espresso in numero di cicli di clock (colonna *Clock*)
- numero di trasferimenti da memoria per la lettura degli operandi (colonna *Trasf.*)
- effetto sui flag.

Per alcune istruzioni nella colonna della durata è riportata la sigla EA. In tali casi per calcolare l'*Effective Address* occorre aggiungere un numero di cicli di clock opportuni in base al tipo di indirizzamento utilizzato. Questi tempi aggiuntivi sono riportati nella Tab. A.1.

Per il calcolo del tempo totale richiesto per l'esecuzione di un'istruzione occorre aggiungere inoltre 4 colpi di clock per ogni trasferimento di un operando posto in memoria ad indirizzo dispari.

Per le istruzioni di salto condizionato e le istruzioni di gestione dei cicli sono riportati due tempi di esecuzione possibili. Il tempo più grande si riferisce al caso in cui viene effettuato il salto, il tempo inferiore si riferisce al caso in cui l'esecuzione prosegue con l'istruzione successiva.

In Tabella A.2 sono riportate le sigle relative alla modifica dei flag.

In Tabella A.3 sono inoltre riportate per ogni istruzione le pagine a cui si fa riferimento all'interno del testo.

Esempio

Sia data la seguente istruzione:

MOV	[BP][SI]+2, CX
------------	-----------------------

Consultando la tabella A.3 si può notare che in corrispondenza dell'istruzione MOV vengono riportati diversi casi corrispondenti ai diversi modi di indirizzamento. L'istruzione in esempio esegue un trasferimento da registro a memoria e per tale caso la tabella riporta un numero di colpi di clock pari a $9 + EA$. L'*Effective Address* dell'operando destinazione è calcolato come

Spiazzamento + Base + Indice ed il numero di colpi di clock necessari è quindi pari a 12 (si veda la Tab. A.1). Il tempo totale richiesto per l'esecuzione dell'istruzione è quindi pari a 21 colpi di clock.

Si noti che nel caso in cui l'operando in memoria sia posto ad un indirizzo dispari, occorre aggiungere altri 4 colpi di clock per effettuare un trasferimento aggiuntivo da memoria.

Il numero complessivo di byte occupati da questa istruzione è pari a 3.

L'istruzione non modifica il valore di alcun flag.

<i>Modo di indirizzamento</i>		<i>Clock</i>
Solo spiazzamento		6
Solo Base o Indice	[BX] o [BP] o [SI] o [DI]	5
Spiazzamento + Base o Indice		9
Base + Indice	[BX] [SI] o [BX] [DI]	7
Base + Indice	[BP] [SI] o [BP] [DI]	8
Spiazzamento + Base + Indice	[BX] [SI] disp o [BX] [DI] disp	11
Spiazzamento + Base + Indice	[BP] [SI] disp o [BP] [DI] disp	12

Tab. A.1: Colpi di *clock* per il calcolo dell'*Effective Address*.

<i>Sigla</i>	<i>Significato</i>
A	Modificato in base al risultato dell'istruzione
U	Indefinito
0	Forzato a 0
1	Forzato a 1
R	Ripristinato dallo stack
nulla	Non modificato

Tab. A.2: Legenda relativa alla modifica dei flag.

AAA	<i>ASCII adjust for addition</i>			pag. 138-139	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	4	-	AAA	AF = A PF = U CF = A SF = U OF = U ZF = U

AAD	<i>ASCII adjust for division</i>			pag. 145	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	2	60	-	AAD	AF = U PF = A CF = U SF = A OF = U ZF = A

AAM	<i>ASCII adjust for multiply</i>			pag. 144	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	83	-	AAM	AF = U PF = A CF = U SF = A OF = U ZF = A

AAS	<i>ASCII adjust for subtraction</i>			pag. 141-142	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	4	-	AAS	AF = A PF = U CF = A SF = U OF = U ZF = U

ADC	<i>Add with carry</i>			pag. 125-126	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro, registro	2	3	-	ADC AX, SI	AF = A PF = A CF = A SF = A OF = A ZF = A
registro, memoria	2-4	9+EA	1	ADC DX, BETA[SI]	
memoria, registro	2-4	16+EA	2	ADC ALFA[SI], DI	
registro, immediato	3-4	4	-	ADC BX, 256	
memoria, immediato	3-6	17+EA	2	ADC GAMMA, 030H	
accumulatore, immediato	2-3	4	-	ADC AL, 5	

ADD	<i>Addition</i>			pag. 123-124	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro, registro	2	3	-	ADD CX, DX	AF = A PF = A CF = A SF = A OF = A ZF = A
registro, memoria	2-4	9+EA	1	ADD DI, ALFA	
memoria, registro	2-4	16+EA	2	ADD TEMP, CL	
registro, immediato	3-4	4	-	ADD CL, 2	
memoria, immediato	3-6	17+EA	2	ADD ALFA, 2	
accumulatore, immediato	2-3	4	-	ADD AX, 200	

AND	<i>Logical AND</i>			pag. 147-148	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro, registro	2	3	-	AND AL, BL	AF = U PF = A CF = 0 SF = A OF = 0 ZF = A
registro, memoria	2-4	9+EA	1	AND CX, FLAG	
memoria, registro	2-4	16+EA	2	AND TEMP[DI], AL	
registro, immediato	3-4	4	-	AND CX, 0F0H	
memoria, immediato	3-6	17+EA	2	AND BETA, 01H	
accumulatore, immediato	2-3	4	-	AND AX, 01001000B	

CALL	<i>Call a procedure</i>			pag. 182-183	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
procedura <i>near</i>	3	19	1	CALL NEAR_PROC	nessuna modifica
procedura <i>far</i>	5	28	2	CALL FAR_PROC	
puntatore in memoria	2-4	21+EA	2	CALL TABLE[SI]	
registro	2	16	1	CALL AX	

CBW	<i>Convert byte to word</i>			pag. 124	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	2	-	CBW	nessuna modifica

CLC	<i>Clear carry flag</i>			pag. 97	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	2	-	CLC	CF = 0

CLD	<i>Clear direction flag</i>			pag. 97	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	2	-	CLD	DF = 0

CLI	<i>Clear interrupt flag</i>			pag. 97	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	2	-	CLI	IF = 0

CMC	<i>Complement carry flag</i>			pag. 97	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	2	-	CMC	CF = A

CMP	<i>Compare destination to source</i>			pag. 99-100	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro, registro	2	3	-	CMP BX, CX	AF = A PF = A CF = A
registro, memoria	2-4	9+EA	1	CMP DH, ALFA	SF = A OF = A ZF = A
memoria, registro	2-4	9+EA	1	CMP [BP+2], SI	
registro, immediato	3-4	4	-	CMP BL, 2	
memoria, immediato	3-6	10+EA	1	CMP TABLE[DI], 2	
accumulatore, immediato	2-3	4	-	CMP AL, 01010110B	

CMPS	<i>Compare string</i>			pag. 169-170	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
destinazione - sorgente	1	22	2	CMPS	AF = A PF = A CF = A
azione ripetuta	1	9+22/rep	2/rep	REPE CMPS	SF = A OF = A ZF = A

CWD	<i>Convert word to doubleword</i>			pag. 137	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	5	-	CWD	nessuna modifica

DAA	<i>Decimal adjust for addition</i>			pag. 138-139	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	4		DAA	AF = A PF = A CF = A SF = A OF = A ZF = A

DAS	<i>Decimal adjust for subtraction</i>			pag. 141-142	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	4	-	DAS	AF = A PF = A CF = A SF = A OF = A ZF = A

DEC	<i>Decrement by 1</i>			pag. 128-129	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro a 16 bit	1	2	-	DEC AX	AF = A PF = A SF = A
registro a 8 bit	2	3	-	DEC AL	OF = A ZF = A
memoria	2-4	15+EA	2	DEC ARRAY[SI]	

DIV	<i>Division, unsigned</i>			pag. 132-133	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro a 8 bit	2	80-90	-	DIV CL	AF = U PF = U CF = U
registro a 16 bit	2	144-162	-	DIV BX	SF = U OF = U ZF = U
memoria	2-4	(86-168) + EA	1	DIV ALFA	

ESC	<i>Escape</i>			pag. 204	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
immediato, memoria	2-4	8+EA	1	ESC 6, VET[SI]	nessuna modifica
immediato, registro	2	2	-	ESC 20, AL	

HLT	<i>Halt</i>			pag. 203	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	2	-	HLT	nessuna modifica

IDIV	<i>Integer Division</i>			pag. 132-133	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro a 8 bit	2	101-112	-	IDIV BL	AF = U PF = U CF = U
registro a 16 bit	2	165-184	-	IDIV CX	SF = U OF = U ZF = U
memoria	2-4	(107-190) +EA	1	IDIV VET[SI]	

IMUL	<i>Integer Multiplication</i>			pag. 130-131	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro a 8 bit	2	80-98	-	IMUL CL	AF = U PF = U CF = A
registro a 16 bit	2	128-154	-	IMUL BX	SF = U OF = A ZF = U
memoria	2-4	(86-160) +EA	1	IMUL VETT[DI]	

IN	<i>Input byte or word</i>			pag. 96-97	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
accumulatore, immediato	2	10	1	IN AL, 0FFEAH	nessuna modifica
accumulatore, DX	1	8	1	IN AX, DX	

INC	<i>Increment by 1</i>			pag. 128-129	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro a 16 bit	1	2	-	INC CX	AF = A PF = A SF = A
registro a 8 bit	2	3	-	INC BL	OF = A ZF = A
memoria	2-4	15+EA	2	INC VET[SI][BX]	

INT	<i>Interrupt</i>			pag. 202	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
immediato (tipo = 3)	1	52	5	INT 3	IF = 0
immediato (tipo ≠ 3)	2	51	5	INT 67	TF = 0

INTO	<i>Interrupt if overflow</i>			pag. 203	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	4 o 53	5	INTO	IF = 0 TF = 0

IRET	<i>Interrupt return</i>			pag. 203	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	24	3	IRET	AF = R PF = R CF = R SF = R OF = R ZF = R DF = R IF = R TF = R

JA / JNBE	<i>Jump if above</i> <i>Jump if not below nor equal</i>			pag. 101-103	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
label	2	4 o 16	-	JA LAB1	nessuna modifica

JAE / JNB	<i>Jump if above or equal</i> <i>Jump if not below</i>			pag. 101-103	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
label	2	4 o 16	-	JAE LAB1	nessuna modifica

JB / JNAE	<i>Jump if below</i> <i>Jump if not above nor equal</i>			pag. 101-103	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
label	2	4 o 16	-	JB LAB1	nessuna modifica

JBE / JNA	<i>Jump if below or equal</i> <i>Jump if not above</i>			pag. 101-103	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
label	2	4 o 16	-	JBE LAB1	nessuna modifica

JC	<i>Jump if carry</i>			pag. 101-103	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
label	2	4 o 16	-	JC LAB1	nessuna modifica

JCXZ	<i>Jump if CX is zero</i>			pag. 101-103	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
label	2	6 o 18	-	JCXZ LAB1	nessuna modifica

JE / JZ	<i>Jump if equal</i> <i>Jump if zero</i>			pag. 101-103	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
label	2	4 o 16	-	JE LAB1	nessuna modifica

JG / JNLE	<i>Jump if greater</i> <i>Jump if not less nor equal</i>			pag. 101-103	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
label	2	4 o 16	-	JG LAB1	nessuna modifica

JGE / JNL	<i>Jump if greater or equal</i> <i>Jump if not less</i>			pag. 101-103	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
label	2	4 o 16	-	JGE LAB1	nessuna modifica

JL / JNGE	<i>Jump if less</i> <i>Jump if not greater nor equal</i>			pag. 101-103	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
label	2	4 o 16	-	JL LAB1	nessuna modifica

JLE / JNG	<i>Jump if less or equal</i> <i>Jump if not greater</i>			pag. 101-103	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
label	2	4 o 16	-	JLE LAB1	nessuna modifica

JMP	<i>Jump</i>			pag. 103-105	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
short-label	2	15	-	JMP SHORT	nessuna modifica
near-label	3	15	-	JMP NEAR LAB	
far-label	5	15	-	JMP FAR LAB	
memoria	2-4	18+EA	1	JMP VET[SI]	
registro	2	11	-	JMP CX	

JNC	<i>Jump if not carry</i>			pag. 101-103	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
label	2	4 o 16	-	JNC LAB1	nessuna modifica

JNE / JNZ	<i>Jump if not equal</i> <i>Jump if not zero</i>			pag. 101-103	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
label	2	4 o 16	-	JNE LAB1	nessuna modifica

JNO	<i>Jump if not overflow</i>			pag. 101-103	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
label	2	4 o 16	-	JNO LAB1	nessuna modifica

JNP / JPO	<i>Jump if not parity</i> <i>Jump if parity odd</i>			pag. 101-103	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
label	2	4 o 16	-	JNP LAB1	nessuna modifica

JNS	<i>Jump if not sign</i>			pag. 101-103	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
label	2	4 o 16	-	JNS LAB1	nessuna modifica

JO	<i>Jump if overflow</i>			pag. 101-103	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
label	2	4 o 16	-	JO LAB1	nessuna modifica

JS	<i>Jump if sign</i>			pag. 101-103	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
label	2	4 o 16	-	JS LAB1	nessuna modifica

LAHF	<i>Load AH from flags</i>			pag. 96	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	4	-	LAHF	nessuna modifica

LDS	<i>Load pointer using DS</i>			pag. 94	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro, memoria	2-4	16+EA	2	LDS SI, STR_IND	nessuna modifica

LOCK	<i>Lock bus</i>			pag. 204	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	2	-	LOCK	nessuna modifica

LODS	<i>Load string</i>			pag. 175-176	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
stringa sorgente	1	12	1	LODS STRING	nessuna modifica
azione ripetuta	1	9+13/rep	1/rep	REP LODS STRING	

LOOP	<i>Loop</i>			pag. 108-109	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
label	2	5/17	-	LOOP AGAIN	nessuna modifica

LOOPE/LOOPZ	<i>Loop if equal</i> <i>Loop if zero</i>			pag. 109-110	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
label	2	6 o 18	-	LOOPE AGAIN	nessuna modifica

LOOPNE/LOOPNZ	<i>Loop if not equal</i> <i>Loop if not zero</i>			pag. 109-110	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
label	2	5 o 19	-	LOOPNE AGAIN	nessuna modifica

LEA	<i>Load effective address</i>			pag. 90-91	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro, memoria	2-4	2+EA	-	LEA BX, [BP][DI]	nessuna modifica

LES	<i>Load pointer using ES</i>			pag. 94	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro, memoria	2-4	16+EA	2	LES DI, STR_IND	nessuna modifica

MOV	<i>Move</i>			pag. 77-85, 87-88	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
memoria, accumulatore	3	10	1	MOV ARRAY[SI],AL	nessuna modifica
accumulatore, memoria	3	10	1	MOV AX, RESULT	
registro, registro	2	2	-	MOV AX, CX	
registro, memoria	2-4	8+EA	1	MOV BP, TOP	
memoria, registro	2-4	9+EA	1	MOV VET[DI], CX	
registro, immediato	2-3	4	-	MOV CL, 2	
memoria, immediato	3-6	10+EA	1	MOV MASK, 02CH	
reg. segmento, registro	2	2	-	MOV ES, CX	
reg. segmento, memoria	2-4	8+EA	1	MOV DS, SEG_BASE	
registro, reg. segmento	2	2	-	MOV BP, SS	
memoria, reg. segmento	2-4	9+EA	1	MOV DATA, CS	

MOVS/MOVSb/MOVSsw	<i>Move string(byte/word)</i>			pag. 166-167	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
destinazione, sorgente	1	18	2	MOVSb	nessuna modifica
azione ripetuta	1	9+17/rep	2/rep	REP MOVSsw	

MUL	<i>Multiplication, unsigned</i>			pag. 130-131	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro a 8 bit	2	70-77	-	MUL BL	AF = U PF = U CF = A SF = U OF = A ZF = U
registro a 16 bit	2	118-133	-	MUL CX	
memoria	2-4	(76-139) + EA	1	MUL ALFA	

NEG	<i>Negate</i>			pag. 129-130	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro	2	3	-	NEG AL	AF = A PF = A CF = 1 SF = A OF = A ZF = A
memoria	2-4	16+EA	2	NEG MULTI	

NOP	<i>No operation</i>			pag. 204	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	3	-	NOP	nessuna modifica

NOT	<i>Logical Not</i>			pag. 151-152	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro	2	3	-	NOT AX	nessuna modifica
memoria	2-4	16+EA	2	NOT CHARACTER	

OR	<i>Logical inclusive OR</i>			pag. 149-150	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro, registro	2	3	-	OR AL, BL	AF = U PF = A CF = 0 SF = A OF = 0 ZF = A
registro, memoria	2-4	9+EA	1	OR DX, FLAG	
memoria, registro	2-4	16+EA	2	OR FLAG, CL	
accumulatore, immediato	2-3	4	-	OR AL, 01001000B	
registro, immediato	3-4	4	-	OR CX, 0F0H	
memoria, immediato	3-6	17+EA	2	OR BETA, 01H	

OUT	<i>Output byte or word</i>			pag. 96-97	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
immediato, accumulatore	2	10	1	OUT 44, AX	nessuna modifica
DX, accumulatore	1	8	1	OUT DX, AL	

POP	<i>Pop word off stack</i>			pag. 94-95	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro	1	8	1	POP DX	nessuna modifica
registro di segmento	1	8	1	POP DS	
memoria	2-4	17+EA	2	POP PARAM	

POPF	<i>Pop flags off stack</i>			pag. 96	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	8	1	POPF	AF = R PF = R CF = R SF = R OF = R ZF = R DF = R IF = R TF = R

PUSH	<i>Push word onto stack</i>			pag. 94-95	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro	1	11	1	PUSH SI	nessuna modifica
registro di segmento	1	10	1	PUSH ES	
memoria	2-4	16+EA	2	PUSH VETT	

PUSHF	<i>Pop flags onto stack</i>			pag. 96	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	10	1	PUSHF	nessuna modifica

RCL	<i>Rotate left through carry</i>			pag. 158-159	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro, 1	2	2	-	RCL CX, 1	CF = A OF = A
registro, CL	2	8+4/bit	-	RCL AL, CL	
memoria, 1	2-4	15+EA	2	RCL ALPHA, 1	
memoria, CL	2-4	20+EA+ 4/bit	2	RCL [BX], CL	

RCR	<i>Rotate right through carry</i>			pag. 158-159	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro, 1	2	2	-	RCR BX, 1	CF = A OF = A
registro, CL	2	8+4/bit	-	RCR BL, CL	
memoria, 1	2-4	15+EA	2	RCR [BX], 1	
memoria, CL	2-4	20+EA+ 4/bit	2	RCR ARRAY[DI], CL	

REP	<i>Repeat string operation</i>			pag. 163-164	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	2	-	REP MOVS DST, SRC	nessuna modifica

REPE/REPZ	<i>Repeat string operation while equal / while zero</i>			pag. 163-164	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	2	-	REPE CMPS DT, SRC	nessuna modifica

REPNE/REPNZ	<i>Repeat string operation while not equal / while not zero</i>			pag. 163-164	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	2	-	REPNE CMPS DT, SR	nessuna modifica

RET	<i>Return from procedure</i>			pag. 183-184	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
intra-segmento, no pop	1	8	1	RET	nessuna modifica
intra-segmento, pop	3	12	1	RET 4	
inter-segmento, no pop	1	18	2	RET	
inter-segmento, pop	3	17	2	RET 2	

ROL	<i>Rotate left</i>			pag. 157-158	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro, 1	2	2	-	ROL BX, 1	CF = A OF = A
registro, CL	2	8+4/bit	-	ROL DI, CL	
memoria, 1	2-4	15+EA	2	ROL BYTE[DI], 1	
memoria, CL	2-4	20+EA+ 4/bit	2	ROL ALPHA, CL	

ROR	<i>Rotate right</i>			pag. 157-158	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro, 1	2	2	-	ROR AL, 1	CF = A OF = A
registro, CL	2	8+4/bit	-	ROR BX, CL	
memoria, 1	2-4	15+EA	2	ROR PORT, 1	
memoria, CL	2-4	20+EA+ 4/bit	2	ROR CMD_WRD, CL	

SAHF	<i>Store AH into flags</i>			pag. 96	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	4	-	SAHF	AF = R PF = R CF = R SF = R ZF = R

SAL/SHL	<i>Shift arithmetic left</i> <i>Shift logical left</i>			pag. 154-157	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro, 1	2	2	-	SAL AL, 1	CF = A OF = A
registro, CL	2	8+4/bit	-	SAL DI, CL	
memoria, 1	2-4	15+EA	2	SAL [BX], 1	
memoria, CL	2-4	20+EA+ 4/bit	2	SAL ALPHA, CL	

SAR	<i>Shift arithmetic right</i>			pag. 156-157	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro, 1	2	2	-	SAR AL, 1	AF = U PF = A CF = A SF = A OF = A ZF = A
registro, CL	2	8+4/bit	-	SAR DI, CL	
memoria, 1	2-4	15+EA	2	SAR [BX], 1	
memoria, CL	2-4	20+EA+ 4/bit	2	SAR ALPHA, CL	

SBB	<i>Subtract with borrow</i>			pag. 126-127	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro, registro	2	3	-	SBB BX, CX	AF = A PF = A CF = A
registro, memoria	2-4	9+EA	1	SBB DI, PAY	SF = A OF = A ZF = A
memoria, registro	2-4	16+EA	2	SBB BALANCE, AX	
accumulatore, immediato	2-3	4	-	SBB AX, 2	
registro, immediato	3-4	4	-	SBB CL, 1	
memoria, immediato	3-6	17+EA	2	SBB COUNT[SI], 10	

SCAS	<i>Scan string</i>			pag. 172-173	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
stringa destinazione	1	15	1	SCAS LINE	AF = A PF = A CF = A
ripeti operazioni	1	9+15/rep	1/rep	REPNE SCAS BUF	SF = A OF = A ZF = A

SHR	<i>Shift logical right</i>			pag. 154-155	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro, 1	2	2	-	SHR SI, 1	CF = A OF = A
registro, CL	2	8+4/bit	-	SHR SI, CL	
memoria, 1	2-4	15+EA	2	SHR BYTE[SI], 1	
memoria, CL	2-4	20+EA+ 4/bit	2	SHR WORD CL	

STC	<i>Set carry flag</i>			pag. 97	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	2	-	STC	CF = 1

STD	<i>Set direction flag</i>			pag. 97	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	2	-	STD	DF = 1

STI	<i>Set interrupt enable flag</i>			pag. 97	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	2	-	STI	IF = 1

STOS	<i>Store byte or word string</i>			pag. 174-175	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
stringa destinazione	1	11	1	STOS PRINT_LINE	nessuna modifica
ripete operazioni	1	9+10/rep	1/rep	REP STOS DISPLAY	

SUB	<i>Subtraction</i>			pag. 123-124	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro, registro	2	3	-	SUB CX, BX	AF = A PF = A CF = A
registro, memoria	2-4	9+EA	1	SUB DX, VET[SI]	SF = A OF = A ZF = A
memoria, registro	2-4	16+EA	2	SUB [BP+2], CL	
accumulatore, immediato	2-3	4	-	SUB AL, 10	
registro, immediato	3-4	4	-	SUB SI, 5280	
memoria, immediato	3-6	17+EA	2	SUB [BP+4], 1000	

TEST	<i>Test</i>			pag. 152-153	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro, registro	2	3	-	TEST SI, DI	AF = U PF = A CF = 0 SF = A OF = 0 ZF = A
registro, memoria	2-4	9+EA	1	TEST SI, VAR	
accumulatore, immediato	2-3	4	-	TEST AL, 00100000B	
registro, immediato	3-4	5	-	TEST BX, 0CC4H	
memoria, immediato	3-6	11+EA	-	TEST, CODE, 01H	

WAIT	<i>Wait while TEST pin not asserted</i>			pag. 204	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
nessuno	1	3+5n	-	WAIT	nessuna modifica

XCHG	<i>Exchange</i>			pag. 88-90	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
accumulatore, registro	1	3	-	XCHG AX, BX	nessuna modifica
memoria, registro	2-4	17+EA	2	XCHG SEM, AX	
registro, registro	2	4	-	XCHG AL, BL	

XLAT	<i>Translate</i>			pag. 92-93	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
tabella sorgente	1	11	1	XLAT ASCII_TAB	nessuna modifica

XOR	<i>Logical exclusive or</i>			pag. 150-151	
<i>Operandi</i>	<i>Byte</i>	<i>Clock</i>	<i>Trasf.</i>	<i>Esempi</i>	<i>Flag</i>
registro, registro	2	3	-	XOR CX, BX	AF = U PF = A CF = 0 SF = A OF = 0 ZF = A
registro, memoria	2-4	9+EA	1	XOR CL, MASK	
memoria, registro	2-4	16+EA	2	XOR ALPHA[SI], DX	
accumulatore, immediato	2-3	4	-	XOR AL, 01000100B	
registro, immediato	3-4	4	-	XOR SI, 00C2H	
memoria, immediato	3-6	17+EA	2	XOR CODE, 0D2H	

Tab. A.3: *Instruction Set* 8086: operandi, tempi di esecuzione, dimensione e flag modificati.