

## HARDWARE E SOFTWARE DEL $\mu$ P 8086

Dispensa interna Istituto Tecnico Industriale “M. Panetti” BARI  
Ettore Panella – Giuseppe Spalierno

### Premessa

Un sistema a  $\mu$ P si può pensare costituito da due strutture: una *software* e l'altra *hardware*. Esse interagiscono tra loro, ma per quanto concerne gli aspetti relativi alla programmazione non è necessario conoscere dettagliatamente la struttura hardware della macchina. Si pensi, ad esempio, all'uso dei wordprocessor, dei fogli elettronici, dei programmi di grafica, che non necessitano, sicuramente, di conoscenze hardware. Se, però, si vuole programmare in un linguaggio a basso livello, come l'assembler o si intende utilizzare il computer per l'interfacciamento a dispositivi esterni, allora è necessario conoscere la struttura generale del  $\mu$ P soprattutto per quanto riguarda l'uso dei registri interni, la gestione della memoria e i metodi di indirizzamento.

Nella seguente fig.1 si riporta lo schema a blocchi del  $\mu$ P 8086.

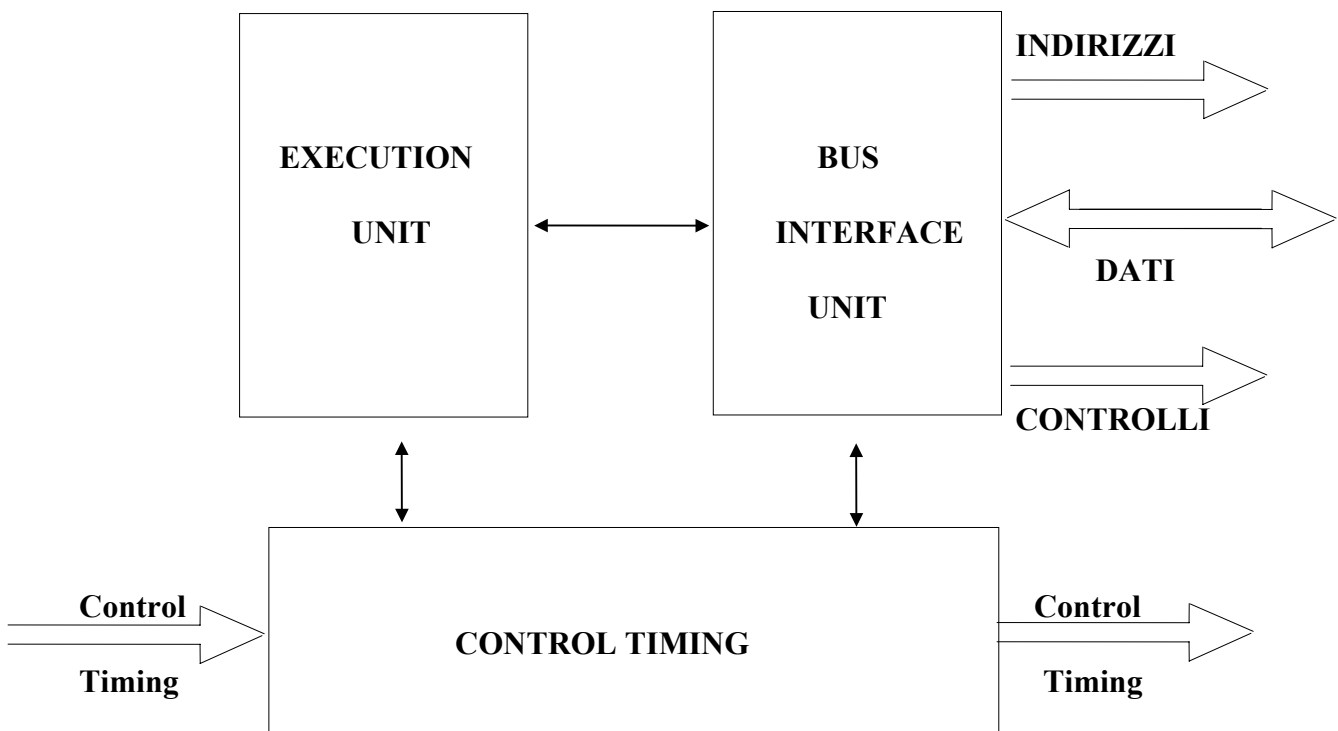


Fig.1 Schema a blocchi del  $\mu$ P 8086

Il  $\mu\text{P}$  8086 è un microprocessore con un BUS indirizzi a 20 bit e può, pertanto, indirizzare 1Mbyte di memoria. Il BUS dati è a 16 bit condiviso fisicamente dalle 16 linee meno significative del BUS indirizzi secondo la tecnica del multiplex temporale; inoltre è fornito di numerose linee di ingresso e uscita per la tempificazione di tutte le fasi operative e per la gestione dei dispositivi di I/O.

Il  $\mu\text{P}$  è costituito da due grandi blocchi denominati **BIU** (Bus Interface Unit) ed **EU** (Execution Unit).

L'unità **BIU** consente l'interfaccia del  $\mu\text{P}$  con il mondo esterno e gestisce, pertanto, tutte le operazioni di I/O. Essa contiene, tra l'altro:

- i 4 registri di segmento CS, DS, SS, ES;
- il registro puntatore IP;
- un sommatore a 20 bit, non accessibile al programmatore, per la generazione dell'indirizzo fisico a 20 bit;
- Una logica di controllo del BUS
- un registro a 6 byte per **la coda delle istruzioni**, non accessibile al programmatore;

Quest'ultimo registro consente il **prefetch** delle istruzioni con aumento della velocità operativa del sistema poiché l'accesso all'istruzione successiva è in un registro interno e non in memoria.

Ogni volta che nella coda delle istruzioni ci sono almeno 2 byte liberi, il BIU carica in tale registro le istruzioni successive del programma che sono quindi messe in coda in attesa di essere prelevate dalla EU. L'inconveniente del metodo si ha quando viene eseguita una istruzione di salto per cui la coda corrente dovrà essere completamente svuotata e sostituita da un'altra.

Tale tecnica trova applicazione poiché le istruzioni di salto sono in media poche in un programma.

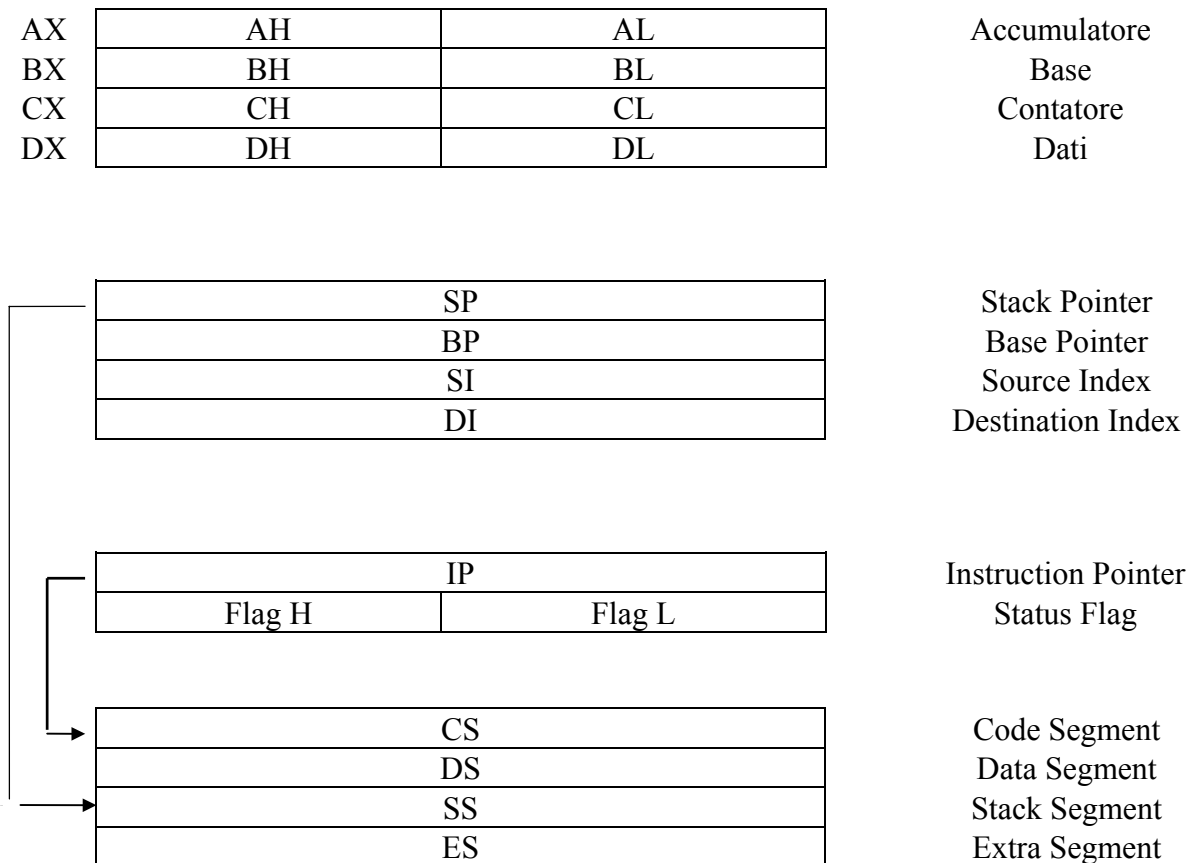
L'unità di esecuzione **EU** non si interfaccia con il mondo esterno ma solo con gli altri componenti tramite BUS interni. La EU contiene, tra l'altro:

- l'Unità Aritmetica Logica **ALU** a 16 bit per l'esecuzione delle operazioni;
- i 4 registri di uso generale AX, BX, CX e DX;
- il registro dei **Flag** ;
- i 2 registri puntatori SP e BP;
- i 2 registri indice DI e SI;
- 2 registri temporanei non accessibili al programmatore per operazioni interne;
- una logica di controllo della EU.

## 1. REGISTRI INTERNI

Il  $\mu\text{P}$  8086 contiene 14 registri a 16 bit di cui 4 registri di uso generale per i dati e 10 utilizzati per funzioni particolari.

In fig.2 si mostra una rappresentazione schematica dei registri interni.



**Fig. 2 Registri interni del  $\mu\text{P}$  8086.**

**I registri di uso generale** sono indicati con AX, BX, CX, DX. Tali registri possono essere usati anche come registri a 8 bit sostituendo alla X la lettera L se si intende usare la parte bassa del registro o la lettera H per la parte alta. Ad esempio, con AL si indica il byte meno significativo del registro a 16 bit AX.

Il **registro AX** è l'*accumulatore* e in esso sono memorizzati gli operandi e i risultati delle operazioni. Il registro AX è utilizzato anche in tutte le operazioni di I/O.

Il **registro BX** è il registro *base* utilizzato in operazioni di trasferimento dati con indirizzamento indiretto.

Il **registro CX** è il registro *contatore* impiegato nelle operazioni di iterazione come contatore dei cicli di loop.

Il **registro DX** è il registro *dati* utilizzato, insieme all'accumulatore, nelle operazioni di moltiplicazione, quando il risultato supera i 16 bit, o nelle operazioni di divisione per la gestione del quoziente e del resto. Esso è usato anche nelle operazioni di I/O.

I **registri dedicati** sono:

I **registri di segmento** CS (Code Segment), DS (Data Segment), SS (Stack Segment) e ES (Extra Segment) sono utilizzati per realizzare la *segmentazione della memoria* ovvero per la generazione dell'indirizzo fisico a 20 bit utilizzando registri a 16 bit. L'indirizzo fisico a 20 bit si ottiene combinando il contenuto di due registri: il primo è un *registro di segmento* il secondo fornisce l'*offset* cioè l'indirizzo all'interno del segmento di memoria utilizzato. Di tale metodo si discuterà nel successivo paragrafo 2.

I **registri puntatori** SP (Stack Pointer) e BP (Base Pointer). Il primo contiene l'offset relativo alla successiva locazione a cui è possibile accedere nel corrente segmento di stack, il secondo contiene anch'esso un indirizzo (offset) all'interno del segmento di stack ed è usato per le operazioni di trasferimento dati con indirizzamenti indiretti.

I **registri indice** SI (Source Index) e DI (Destination Index) sono utilizzati come offset negli indirizzamenti all'interno del segmento dei dati.

Il **registro puntatore all'istruzione** IP (Instruction Pointer) consente, insieme al registro CS, di individuare l'indirizzo fisico della successiva istruzione del programma in esecuzione . Tale indirizzo si indica con **CS:IP**.

Il **registro di stato** F (Flag) è un registro a 16 bit di cui solo 9 utilizzati. Lo stato logico del singolo flag o è definito dal programmatore o è imposto automaticamente dal  $\mu P$  e indica la condizione prodotta dal risultato della precedente operazione logica o aritmetica.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

Flag impostati dal  $\mu P$  dopo un'operazione:

- **CF Carry Flag** è settato se c'è un riporto o un prestito nel bit più significativo nel risultato di una istruzione aritmetica.
- **PF Parity Flag** è settato se il risultato dell'istruzione ha parità pari.
- **AF Auxiliary carry Flag** è utilizzato in operazioni in codice BCD per segnalare un riporto tra il 4° e il 5° bit di un dato di un byte.
- **ZF Zero Flag** è settato se il risultato dell'operazione logica o aritmetica è 0.
- **SF Sign Flag** è settato se il risultato dell'operazione è negativo.
- **OF Overflow Flag** è settato se il risultato di una operazione aritmetica è di segno opposto a quello atteso.

Flag di controllo definiti dal programmatore:

- **TF Trap Flag** se settato forza il  $\mu P$  a operare in single step.
- **IF Interrupt Flag** se resettato il  $\mu P$  ignora qualsiasi richiesta di interruzione inviata, con un livello alto sulla linea INTR, dal periferico.
- **DF Direction Flag** se resettato il  $\mu P$  incrementa gli indirizzi per cui le stringhe vengono lette partendo dagli indirizzi più bassi verso quelli più alti; viceversa se settato gli indirizzi sono decrementati.

## 2. GESTIONE DELLA MEMORIA

Il BUS indirizzi del  $\mu P$  8086 è a 20 bit pertanto è possibile indirizzare 1Mbyte locazioni di memoria. I registri interni sono però a 16 bit e con un tal numero di bit si possono indirizzare solo 64Kbyte locazioni di memoria. L'indirizzamento a 20 bit si realizza mediante la seguente tecnica: l'indirizzo fisico effettivo ADDRESS a 20 bit si ottiene combinando, come segue, due parole a 16 bit denominate SEGMENT e OFFSET:

$$\text{ADDRESS} = \text{SEGMENT} * 16 + \text{OFFSET}$$

La memoria si può ritenere costituita da un insieme di finestre di 64K. SEGMENT individua la **base** della finestra mentre OFFSET individua la locazione di memoria all'interno della finestra prescelta. Si osservi che per ottenere l'indirizzo fisico effettivo ADDRESS la parola SEGMENT è stata moltiplicata per 16 (ciò equivale ad uno scorrimento a sinistra di 4 bit per cui un incremento unitario di SEGMENT equivale ad un incremento di 16 locazioni di memoria, ovvero di una cifra esadecimale. Tale incremento minimo è detto PARAGRAFO . Il valore di SEGMENT è contenuto in uno dei *registri di segmento* mentre OFFSET è fornito dal registro IP o da un operando contenuto in una istruzione del programma.

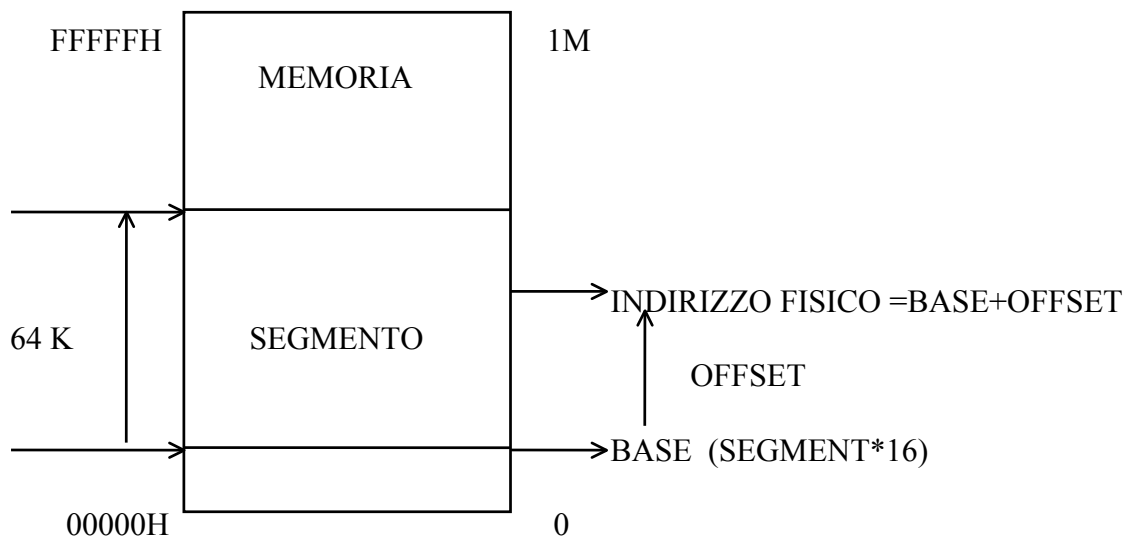
Ad esempio, se il registro CS contiene il numero esadecimale A02C e IP contiene 0C00, in forma simbolica si scrive:

CS:IP=A02C:0C00 e l'indirizzo fisico a 20 bit è:

$$\text{ADDRESS} = 16 * (\text{CS}) + (\text{IP}) = (16)_{10} * \text{A02C} + 0\text{C00} = \text{A02C0} + 0\text{C00} = \text{A0\text{E}C0}$$

In definitiva l'indirizzo fisico richiesto vale A0EC0.

In fig.3 si mostra una schematizzazione della gestione della memoria.



**Fig. 3 Segmentazione della memoria**

I 4 registri utilizzati per determinare la *base* dei segmenti sono:

- a) **CS ( Code Segment )** fornisce la base del segmento che contiene il codice per il prelevamento delle istruzioni del programma da eseguire. Il registro CS è normalmente associato al registro IP ( Instruction Pointer ) che contiene l'OFFSET rispetto alla base fornita da CS. La locazione di memoria relativa all'istruzione è data quindi dalla relazione:  $16*(CS) + (IP)$  e si indica brevemente con **CS:IP**;
- b) **SS ( Stack Segment )** fornisce la base del segmento che individua l'area di STACK. Il registro SS è normalmente associato ai registri SP ( Stack Pointer ) e BP ( Base Pointer ). La locazione di memoria puntata si indica con **SS:SP** oppure con **SS:BP** ;
- c) **DS ( Data Segment )** fornisce la base del segmento riservato ai dati e viene normalmente impiegato nelle operazioni di trasferimento dati;
- d) **ES ( Extra Segment )** fornisce la base di un segmento riservato ai dati e viene normalmente impiegato nella gestione delle stringhe.

In definitiva la memoria del  $\mu P$  si può considerare suddivisa in 4 segmenti di 64Kbyte indirizzati dai registri di segmento; comunque, non tutte le locazioni di memoria sono indirizzabili, poiché esistono zone di memoria riservate al sistema. In particolare, le locazioni di memoria comprese tra 00000H e 003FFH sono riservate alla gestione degli interrupt, mentre quelle comprese tra FFFF0H e FFFFFH sono riservate per la inizializzazione del sistema ( bootstrap ) o alle operazioni dopo un RESET.

I dati a 16 bit si possono memorizzare in corrispondenza degli indirizzi pari o di quelli dispari. In tutti i casi, però, il byte meno significativo è memorizzato all'indirizzo selezionato mentre quello più significativo nella locazione di memoria successiva ad indirizzo più alto.

### 3. GESTIONE DELLE INTERRUZIONI NEL $\mu P$ 8086

In generale, la tecnica delle interruzioni è utilizzata per la gestione e lo scambio di dati tra il  $\mu P$  e i dispositivi periferici. Quando un periferico chiede servizio al  $\mu P$  invia un segnale di interrupt all'unità centrale del  $\mu P$ . Se essa è abilitata ad accettare l'interrupt, termina l'istruzione corrente del programma attualmente in corso e passa ad eseguire un altro programma residente in memoria noto come *routine di interrupt*. Alla fine di tale routine il  $\mu P$  riprende il programma interrotto esattamente dall'istruzione successiva all'ultima eseguita.

Le interruzioni utilizzate nel  $\mu P$  8086 si possono classificare nel seguente modo:

- a) **PREDEFINITE** sono di *tipo software* e vengono determinate da particolari situazioni software dette TRAP, come ad esempio la divisione per zero, una operazione che ha causato un overflow oppure se si impiega il computer in funzionamento single-step.
- b) **GENERATE DA PROGRAMMA** sono di *tipo software* e si ottengono mediante particolari istruzioni denominate INT ( parag. 3.2).

**INTERRUZIONI HARDWARE** sono generate da eventi esterni mediante segnali logici che comandano particolari piedini del  $\mu P$ . Tali interruzioni sono impiegate per la gestione delle periferiche. La linea INTR ( Interrupt Request ) è mascherabile via software in funzione dello stato del **flag IF**. Se  $IF=1$  l'interruzione è abilitata e il  $\mu P$  esegue la routine di interrupt quando la linea INTR è al livello alto. Se  $IF=0$  l'interrupt è disabilitato e il  $\mu P$  ignora lo stato logico della linea INTR. Le istruzioni relative sono STI (Set Interrupt) e CLI (Clear Interrupt).

La linea NMI (Non Maskable Interrupt) è non mascherabile, pertanto, tale interrupt è sempre servito quando sulla linea NMI del  $\mu P$  giunge un fronte positivo.

### 3.1 Tabella delle interruzioni o dei puntatori

Le interruzioni provocano un salto ad un sottoprogramma il cui indirizzo è letto dal  $\mu P$  nella *tabella delle interruzioni o dei puntatori*. Si riporta la tabella relativa alle locazioni di memoria comprese tra 00000H e 003FFH a cui far riferimento per la gestione degli interrupt .

TIPO n	LOCAZIONE	INTERRUZIONE
0	00000H	TRAP-divis.per 0
1	00004H	TRAP-single step
2	00008H	NMI
3	0000CH	Istruzione INT
4	00010H	TRAP-overflow
5...31	00014H...0007CH	Riservati INTEL
32...255	00080H...003FCH	INTR

Gli elementi di questa tabella di puntatori consentono di determinare gli indirizzi fisici SEGMENT:OFFSET delle routine di gestione degli interrupt stessi. Il dispositivo che genera l'interrupt fornisce al  $\mu P$ , durante la fase di riconoscimento dell'interrupt, un dato a 8 bit che individua nella tabella di interrupt il puntatore da utilizzare per il salto. Il dato a 8 bit acquisito dal periferico interrompente è moltiplicato per 4 per ottenere l'indirizzo effettivo del puntatore della tabella. Ogni puntatore occupa 4 byte. In particolare i due byte di locazione più bassa individuano IP mentre quelli di più alti CS; in tal modo si ottiene l'indirizzo iniziale CS:IP della routine di interrupt.

Ad esempio, l'interrupt NMI è di tipo  $n=2$  che, moltiplicato per 4, fornisce l'offset 0008H, pertanto, nella tabella delle interruzioni NMI individua i 4 byte compresi tra 0000H:0008H e 0000H:000BH e in tali locazioni si trovano i valori di CS:IP dell'indirizzo fisico della routine di interrupt di NMI.

Nel caso di NMI e TRAP le locazioni sono fisse. Le interruzioni  $INT_n$  con  $n=0...255$  interpretano  $n$  come byte di puntamento alla tabella. L'interruzione INT senza parametro  $n$  equivale a INT3 che punta alla locazione di memoria 0000CH.

Le interruzioni software definite dall'istruzione  $INT_n$ , non possono essere disabilitate e hanno la più alta priorità sia rispetto a NMI che a INTR. Fa eccezione l'interruzione di single-step che ha la più bassa priorità.

In dettaglio quando il  $\mu P$  deve servire un interrupt esegue i seguenti passi:

- salva nello STACK il contenuto del registro dei FLAG;
- resetta i due flag IF ( Interrupt Flag) e TF (Trap Flag) ;
- salva nello STACK il CS;
- pone in CS il contenuto della locazione di memoria pari a  $4*n+2$ , dove  $n$  è il tipo di interrupt ( $n=3$  è automatico per l'istruzione INT3) ; il termine +2 si riferisce al fatto che nella tabella degli interrupt il CS occupa la word più significativa;
- salva nello STACK il contenuto di IP;
- pone in IP il contenuto della locazione di memoria pari a  $4*n$ .

## L'istruzione IRET

L'istruzione **IRET** (Interrupt RETurn) è l'istruzione che solitamente chiude le routine di interrupt ed ha lo scopo di ripristinare lo stato di CS, di IP e dei Flag che erano stati salvati nello STACK all'atto del riconoscimento dell'interruzione.

### 3.2 Interruzioni del BIOS e del DOS

Alle diverse istruzioni software INTn sono associate delle routine di tipo generale utilizzate dal BIOS e dall' MSDOS per attivare delle routine di sistema necessarie per la gestione delle risorse hardware interne come il video, la tastiera, le porte seriali, la stampante i dischi, ecc.. I servizi forniti da tali interrupt dipendono dai valori che il programmatore imposta in ben determinati registri, come specificato nel manuale tecnico

Le **interruzioni del BIOS** sono numerose ma quelle che più interessano la stesura dei programmi in linguaggio assembly sono la INT 10H, per la gestione dei caratteri sullo schermo e la INT 16H per la gestione della tastiera..

Ad esempio, i seguenti comandi impostano il video in modalità grafica con una risoluzione di 640\*200 punti:

```
MOV  AH, 00      ; AH=00 determina la modalità di visualizzazione
MOV  AL, 06      ; AL=06 fissa la modalità in modo grafico 640*200
INT  10H
```

Se in AL si fosse caricato il valore 02 si sarebbe ottenuta la modalità testo con risoluzione 640\*200.(†)

Per quanto concerne le **interruzioni del DOS** valgono le stesse considerazioni fatte per quelle del BIOS.

Ad esempio, la INT 21H del DOS è denominata *chiamata di funzione*. In particolare i seguenti comandi consentono di terminare un programma e restituire il controllo al DOS:

```
MOV  AH, 4CH
INT  21H
```

Come per tutte le altre chiamate INTn modificando, opportunamente, i parametri da inserire in AH e in eventuali altri registri è possibile ottenere altri servizi.

---

† Si consulti il cap.13 del testo di Sistemi 2 di Salsano, Ferreri e Totano Edizioni Petrini.

#### 4. METODI DI INDIRIZZAMENTO

Per metodi di indirizzamento si intendono le procedure interne che il  $\mu P$  deve compiere per eseguire le operazioni specificate dall'istruzione corrente del programma in esecuzione. Si descrivono alcuni di tali metodi facendo riferimento ad esplicite istruzioni.

##### Indirizzamento immediato

L'operando sorgente è contenuto nell'istruzione, mentre la destinazione è un registro. Ad esempio:

```
MOV BX, 0A14CH
```

Muove (trasferisce) il numero esadecimale A14C nel registro BX, in particolare si avrà BH=A1 e BL=4C. Lo zero iniziale, nel numero esadecimale, è necessario quando tale numero inizia con un carattere alfabetico e il programma deve essere assemblato con MASM. In Debug la sintassi è:

```
MOV BX, A14C
```

##### Indirizzamento implicito

Si ha quando l'istruzione non necessita di alcun operando.

```
CLC ; resetta il flag di carry
```

```
STI ; setta il flag di interrupt
```

##### Indirizzamento con registro

Gli operandi sono contenuti nei registri.

```
MOV AX, CX ; trasferisci nel registro AX il dato contenuto nel registro CX.
```

```
MOV AH, AL ; trasferisci in AH il contenuto di AL.
```

##### Indirizzamento diretto

L'istruzione contiene l'indirizzo dell'operando.

```
MOV BL, VAR ; trasferisce il contenuto della variabile VAR nel registro BL.
```

L'assemblatore traduce automaticamente il riferimento alla variabile VAR in un indirizzo effettivo di memoria nel segmento dati DS. Se ad esempio, VAR è stata definita nel programma come byte esadecimale FA e l'assemblatore, automaticamente, alloca tale dato in memoria all'offset 0015H del segmento DS, allora l'istruzione precedente è tradotta in:

```
MOV BL, [0015] ; sintassi valida in ambiente Debug
```

ovvero, trasferisci in BL il byte contenuto nella locazione di memoria puntata di offset 0015 del segmento DS. Al termine dell'istruzione il registro BL conterrà il byte FA.

Analogamente per la seguente istruzione:

```
MOV RIS, AX ; trasferisce la word contenuta in AX nella variabile RIS.
```

Anche in questo caso a RIS sarà associato automaticamente dall'assemblatore un indirizzo effettivo di memoria nel segmento DS.

##### Indirizzamento indiretto

L'operando si trova ad un indirizzo il cui offset è contenuto nel registro puntatore BX, BP, SI o DI. Il registro di segmento è, per default, DS per BX, DI e SI mentre al registro di segmento SS è associato BP.

```
MOV AH, [BX] ; trasferisci in AH il byte memorizzato nella locazione DS:BX.
```

```
MOV BX, [BP] ; trasferisci in BX la word contenuta all'indirizzo SS:BP
```

MOV [SI], AL ; trasferisci il byte contenuto in AL nella locazione puntata da DS:SI.

### **Indirizzamento indiretto con base**

L'offset dell'operando è contenuto in un registro base BX o BP più un ulteriore spiazzamento. BX è l'offset per DS, mentre BP per SS.

MOV AX, [BX+4] ; trasferisci in AX il contenuto della word puntata da DS:(BX+4).

MOV AX, [BP+ VAR] ; trasferisci in AX il contenuto della word puntata da SS:(BP+VAR).

### **Indirizzamento indiretto con indice**

L'offset dell'operando è contenuto in un registro indice SI o DI più un ulteriore spiazzamento all'interno del segmento dati DS.

MOV AX, [DI+4] ; trasferisci in AX il contenuto della word puntata da DS:(DI +4).

MOV AX, [SI+ VAR] ; trasferisci in AX il contenuto della word puntata da DS:(SI +VAR).

### **Indirizzamento indiretto con indice e con base**

L'offset dell'operando è ottenuto da una combinazione dei registri indice e base. Come al solito BX, DI e SI si riferiscono al segmento dati DS, mentre BP a SS.

MOV [BX+2+DI],AL ; trasferisci il byte contenuto in AL nella locazione puntata da DS:(BX+2+DI).

MOV AX, [BP+DI+VAR] ; trasferisci in AX la word puntata da SS:(BP+DI+VAR).

### **Indirizzamento di stringhe**

Le istruzioni che riguardano le stringhe usano automaticamente il registro indice SI per l'operando sorgente contenuto nel segmento dati DS e il registro indice DI per l'operando destinazione in ES.

MOVSB ; sposta un byte dalla stringa sorgente di locazione DS:SI nel byte della stringa destinazione alla locazione ES:DI. Si noti che nell'istruzione non compare alcun registro.

Nei metodi di indirizzamento indiretto i registri BX, DI e SI sono associati, per default, con DS, mentre BP è associato con SS. E' possibile modificare tale corrispondenza tramite un opportuno codice che forza all'uso di un diverso segmento. Tale metodo è detto **VERRIDE** e si ottiene, semplicemente, specificando il nuovo segmento.

MOV AX, ES:[BX] ; trasferisce il contenuto della locazione puntata da ES:BX in AX, anche se BX normalmente è associato con DS.

## SET DI ISTRUZIONI DEL 8086

Le istruzioni del  $\mu P$  8086 si possono suddividere nei seguenti gruppi:

1. **Trasferimento**
2. **Aritmetiche**
3. **Logiche**
4. **Controllo**
5. **Rotazione**
6. **Salto**
7. **Iterazione**
8. **Manipolazione di stringhe**

Le istruzioni usano un numero di byte che dipende, dal tipo di istruzione, dal numero di operandi e dal loro formato (byte o word). Ogni operando può essere o un dato immediato o una locazione di memoria o un registro. Se una istruzione impiega due operandi, questi, non possono essere entrambi locazioni di memoria. Pertanto, se si vuole trasferire un dato da una locazione di memoria ad un'altra, è necessario utilizzare un registro come appoggio temporaneo.

### 1. ISTRUZIONI DI TRASFERIMENTO

Sono istruzioni che consentono di trasferire dei dati tra due operandi indicati, genericamente, con *destinazione* e *sorgente*. La sintassi dell'istruzione è del tipo:

ISTRUZIONE *destinazione, sorgente*

In tal modo si trasferisce il dato (byte o word) indicato in *sorgente*, nell'operando *destinazione*. L'operando sorgente può essere o un dato immediato, o il contenuto di una locazione di memoria oppure il contenuto di un registro; l'operando destinazione può essere o un registro o una locazione di memoria.

Le istruzioni di trasferimento non modificano i flag del registro di stato F.

#### **MOV destinazione, sorgente**

Trasferisci (muovi) *sorgente* in *destinazione*

##### Esempi

1) `MOV AX, CX` ; trasferisci il contenuto del registro CX nel registro AX.

2) `MOV BX, 0A427H` ; trasferisci il dato immediato esadecimale A427 nel registro BX.

Lo zero iniziale è necessario in ambiente MASM. In Debug non si deve inserire nè lo zero iniziale nè il carattere finale H.

Quando si indica un indirizzo indiretto di memoria è necessario specificare se si opera in byte o in word. Ad esempio:

`MOV [DI+BX], 100` ; potrebbe puntare sia ad un byte che ad una word e ciò produce un messaggio d'errore. Per ovviare a tale inconveniente è necessario specificare il tipo di espressione con il comando PTR, come segue:

`MOV WORD PTR [DI+BX], 100` ; indica che ci si riferisce ad una word.

### **PUSH sorgente**

Decrementa SP di 2 e memorizza il dato *sorgente* a 16 bit nella parte superiore dello stack.

### **POP destinazione**

Preleva una word dalla parte superiore dello stack, usando l'indirizzo SS:SP, lo trasferisce in *destinazione* e successivamente incrementa SP di 2 .

### **XCHG destinazione, sorgente**

Scambia fra loro il contenuto di due registri o di un registro e una locazione di memoria.

#### Esempio

XCHG DX, BX ; scambia il contenuto dei registri DX e BX.

### **XLAT**

Trasferisce il byte indirizzato dalla somma di [AL]+[BX], relativo a DS, nel registro AL. Questa istruzione è utilizzata nella gestione delle tabelle. BX contiene l'indirizzo d'inizio della tabella e AL punta al byte da caricare entro la tabella.

### **IN accumulatore, porta**

Trasferisce il valore letto all'indirizzo *porta* nell'accumulatore.

#### Esempio

IN AL,200 ;trasferisce in AL il byte presente nella porta di indirizzo 200.

### **OUT porta, accumulatore**

Invia in uscita, all'indirizzo di *porta*, il contenuto dell'accumulatore (AL o AX).

### **LDS registro, memoria (LES registro, memoria)**

Carica i primi due byte indirizzati in memoria nel registro selezionato (il primo nella parte bassa e il secondo in quella alta) e i successivi due byte nel registro DS (nel registro ES per la LES). In tal modo si ottiene l'indirizzo fisico puntato da DS:*registro* (oppure ES:*registro* per la LES).

#### Esempio :

Supponiamo che nelle locazioni di memoria comprese tra 0200 e 0203 siano memorizzati i byte:

02, 2A, 04 e C7, l'istruzione:

LDS SI, [0200] ; pone SI=2A02 e DS=C704

### **LEA registro, DATO**

Load Effective Address. Trasferisce l'indirizzo di offset della locazione di memoria della variabile DATO in un registro a 16 bit. Nel registro destinazione viene posto l'indirizzo e non il valore contenuto nella locazione di memoria. E' equivalente all'istruzione:

MOV BX,OFFSET DATO

### Esempi

- 1) LEA BX, STRINGA[SI] ; carica in BX l'offset di STRINGA[SI]
  
- 2) LEA BX, [SI+10] ; carica in BX il valore di offset di [SI+10].

## **2. ISTRUZIONI ARITMETICHE**

Le istruzioni aritmetiche consentono di eseguire le operazioni fondamentali su operandi a 8 o 16 bit con e senza segno. Nei numeri con segno si utilizza la rappresentazione in complemento a 2. Il formato generale è del tipo:

### **ISTRUZIONE *destinazione, sorgente***

La *destinazione* può essere un registro o la memoria, mentre la *sorgente* può essere un registro, la memoria o un dato immediato.

### **ADD AX, DATO**

Somma il contenuto di DATO con quello di AX e pone il risultato in AX.

### **ADC AX, DATO**

Somma il contenuto di AX con quello di DATO più il valore del flag di carry. Il risultato è posto in AX.

### **SUB AX, BX**

Esegue la sottrazione tra il contenuto di AX con quello di BX e pone il risultato in AX.

### **SBB**

Esegue la sottrazione tra il contenuto di AX con quello di BX e con il valore del flag di carry e pone il risultato in AX.

### **AAA (AAS)**

Corregge il risultato di una addizione (o sottrazione per AAS) di operandi in codice BCD non compattato<sup>‡</sup> in modo che il risultato siano due cifre BCD, non compattate, una in AL e l'altra in AH.

---

<sup>‡</sup> Il codice BCD *compattato* usa 4 bit per codificare una cifra esadecimale per cui in un byte vi sono due cifre esadecimali. Il codice *non compattato* usa un intero byte per codificare una cifra decimale tenendo i 4 bit di ordine superiore tutti a 0.

### Esempio di programma somma con aggiustamento

MOV AL, 09 ; pone in AL il numero 9  
MOV BL, 08 ; pone in BL il numero 8  
ADD AL, BL ; in AL si trova la somma di [AL]+[BL] cioè il numero 17 che in binario si scrive: AL=00010001.  
AAA ; il risultato precedente viene aggiustato in due cifre BCD non compattate in modo che risulti AH=1 (AH=00000001) e AL=7 (AL=00000111).

### **DAA (DAS)**

Corregge il risultato di una addizione (o sottrazione per DAS) di due operandi BCD compattati in modo che il risultato siano 2 cifre BCD compattate poste in AL.

### **MUL sorgente**

Moltiplica, senza segno, numeri interi a 8 o 16 bit. L'operando *sorgente* contiene il moltiplicatore; il moltiplicando può essere posto in AL se a 8 bit o in AX se a 16 bit: Nel primo caso il risultato va in AX, nel secondo caso il risultato va nella coppia di registri DX:AX (word alta in DX, quella bassa in AX).

### **IMUL sorgente**

Come la precedente istruzione effettua la moltiplicazione con segno in logica a complemento a 2.

### **DIV sorgente**

Effettua la divisione senza segno. Il dividendo deve essere posto in AX (se a 16 bit) oppure nella coppia di registri DX:AX (se a 32 bit); il divisore a 8 o 16 bit è in *sorgente*. Il risultato può essere, rispettivamente, a 8 bit o a 16 bit. Nel primo caso il quoziente va in AL e il resto in AH, nel secondo caso il quoziente va in AX e il resto in DX.

### **IDIV sorgente**

Divisione con segno in complemento a 2.

### **AAM**

Va inserita dopo una moltiplicazione tra due numeri BCD non compattati. Il risultato è al massimo 81 (9\*9) e occupa solo AL con AH=0. L'istruzione divide AL per 10 e memorizza il quoziente in AH (cifra più significativa) e il resto in AL (cifra meno significativa).

### **AAD**

Va inserita prima di una istruzione di divisione. Essa converte 2 cifre BCD non compattate memorizzate in AL e in AH in un numero binario (al massimo 99) posto in AL con AH=0.

**INC sorgente**

Incrementa di una unità il contenuto di *sorgente*.

**DEC sorgente**

Decrementa di una unità il contenuto di *sorgente*.

**NEG sorgente**

Cambia il segno dell'operando *sorgente*

**CMP sorg1, sorg2**

Esegue l'operazione di sottrazione tra *sorg1-sorg2* senza modificare gli operandi ma solo i bit di flag.

### 3. ISTRUZIONI LOGICHE

Consentono di eseguire le operazioni logiche fondamentali.

**AND dest., sorg.**

Esegue l'AND bit a bit tra *destinazione* e *sorgente* con risultato in *destinazione*.

**OR dest., sorg.**

Esegue l'OR bit a bit tra *destinazione* e *sorgente* con risultato in *destinazione*.

**XOR dest., sorg.**

Esegue l'XOR bit a bit tra *destinazione* e *sorgente* con risultato in *destinazione*.

**TEST dest., sorg.**

Esegue l'AND bit a bit tra *destinazione* e *sorgente*, modifica i bit di flag, ma non sposta il risultato in *destinazione*.

### 4. ISTRUZIONI DI CONTROLLO

Sono delle istruzioni per la manipolazione dei bit di flag o per il controllo di alcune operazioni del sistema.

**CLC** ; resetta il flag di carry.

**CMC** ; complementa il flag di carry.

**STC** ; setta il flag di carry.

**CLD** ; resetta il flag di direzione.

**STD** ; setta il flag di direzione.

- CLI** ; inibisce le interruzioni mascherabili.
- STI** ; abilita le interruzioni mascherabili.
- LAHC**; trasferisce la parte bassa del registro dei flag in AH.
- SAHC** ; trasferisce il contenuto di AH nella parte bassa del registro dei flag.
- CBW** ; estende il segno, cioè, trasferisce il valore del bit 7 del registro AL in tutto il registro AH.
- CWD** ; estende il segno di AX; cioè, trasferisce il valore del bit 15 di AX in tutto il registro DX.
- HALT** ; manda in HALT il  $\mu$ P. Si può uscire da tale situazione con RESET o con un interrupt esterno.
- WAIT** ; pone in wait il  $\mu$ P: Si può uscire da tale situazione attivando la linea TEST. Un eventuale interrupt esterno viene servito e successivamente il  $\mu$ P torna in wait.
- NOP** ; No Operation. Non effettua alcuna operazione; viene utilizzata per lasciare spazio in memoria nel caso in cui si devono inserire nuove istruzioni nel programma.

### 5. ISTRUZIONI DI ROTAZIONE E SHIFT

Sono istruzioni che consentono di ruotare a destra o a sinistra il contenuto di registri o locazioni di memoria. Il formato generale delle istruzioni è:

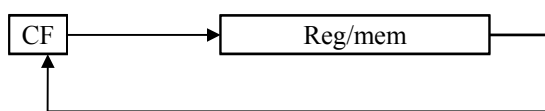
#### ISTRUZIONE *sorgente, count*

Le rotazioni o gli shift avvengono tante volte quanto è scritto nel registro contatore CL (count). Si deve, pertanto, inizializzare tale registro con l'istruzione:

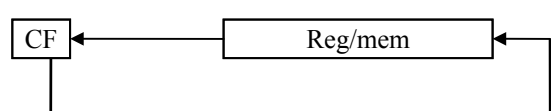
#### **MOV CL, n**

In tal modo  $count=n$  e le operazioni di rotazione o di shift avverranno n volte. Se  $n=1$  è sufficiente porre nell'istruzione  $count=1$  senza inizializzare CL.

Rotazioni a destra (RCR) o a sinistra (RCL) con carry.

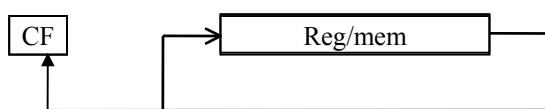


**RCR**

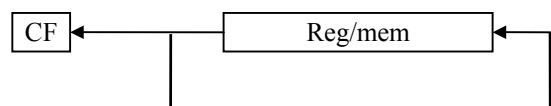


**RCL**

Rotazioni a destra (ROR) o a sinistra (ROL) senza carry.

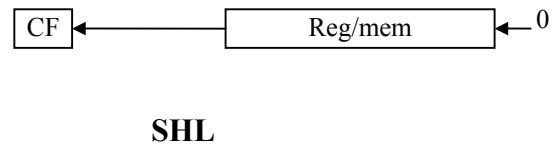
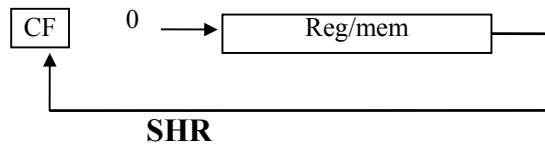


**ROR**

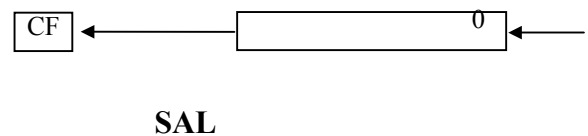
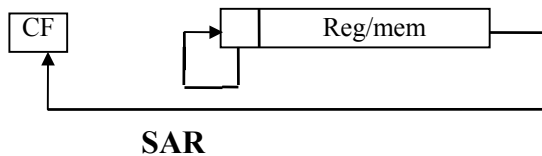


**ROL**

Scorrimento a destra (SHR) o a sinistra (SHL) con caricamento di 0.



Scorrimento aritmetico a destra (SAR) o a sinistra (SAL). L'istruzione SAL è equivalente a SHL.



## 6. ISTRUZIONI DI SALTO

Consentono di effettuare dei salti ad una procedura o ad una diversa parte di programma. Tali istruzioni non modificano i flag del registro di stato F.

### CALL label

Salta alla procedura (subroutine) di nome *label*. L'etichetta può riferirsi ad una subroutine nello stesso segmento (procedura vicina NEAR) o ad un altro segmento (procedura lontana FAR).

### RET

Istruzione di chiusura della subroutine di tipo NEAR per il ritorno al programma principale.

### RETF

Come la precedente ma per subroutine di tipo FAR.

### 6.1. Istruzioni di salto incondizionato

Permettono il salto ad altra parte del programma sia infrasegmentale (NEAR) che intersegmentale (FAR). Il primo tipo modifica solo il valore del puntatore di istruzione IP, il secondo tipo si ottiene modificando anche il valore del registro di segmento CS.

### JMP n

Sostituisce al contenuto di IP il valore IP+n. Il massimo salto dipende dal parametro n che può variare da -128 a +127. In ambiente Debug si inserisce l'offset dell'indirizzo a cui si vuole saltare.

### JMP label

Sostituisce in IP il valore dichiarato di *label*. Istruzione valida in MASM.

### **JMP BX**

Pone il contenuto del registro BX nel registro IP e quindi effettua il salto.

### **JMP [00B7]**

Pone il contenuto della locazione di memoria di indirizzo esadecimale 00B7 nel registro IP e quindi effettua il salto.

### **JMP FAR [BX]**

Si tratta di un salto FAR. Il valore di IP sono i primi due byte indirizzati dal contenuto del registro BX, i secondi due byte costituiscono il valore del registro di segmento CS. In tal modo si è costruito l'indirizzo fisico CS:IP per il salto intersegmentale.

### **JMP FAR [00B7]**

Come la precedente istruzione con riferimento all'indirizzo puntato dal contenuto della locazione di memoria 00B7.

## **6.2. Istruzioni di salto condizionato**

Sono istruzioni che consentono di effettuare un salto indietro o in avanti di tipo infrasegmentale, compreso tra -128 e +127, rispetto all'indirizzo della successiva istruzione che si deve eseguire. L'offset di IP è, pertanto, un solo byte con segno.

Il formato dell'istruzione è del tipo:

### **JXXX label**

dove XXX è il codice mnemonico della condizione di salto e può essere di 1 2 o 3 lettere. La *label* è l'etichetta che individua l'indirizzo a cui saltare.

Si riportano alcune istruzioni di salto non scrivendo, per semplicità, l'etichetta *label*.

<b>JNC</b>	; salta se non c'è riporto (flag CF=0)
<b>JC</b>	; salta se c'è riporto (flag CF=1)
<b>JNZ</b>	; salta se non è zero (flag ZF=0)
<b>JZ</b>	; salta se zero (flag ZF=1)
<b>JNS</b>	; salta se non è negativa (flag SF=0)
<b>JS</b>	; salta se è negativo (flag SF=1)
<b>JNP</b>	; salta se non c'è parità (flag PF=0)
<b>JP</b>	; salta se c'è parità (flag PF=1)
<b>JNA</b>	; salta se minore o uguale ( flag CF=1 o ZF=1)
<b>JN</b>	; salta se maggiore (flag CF=0 o ZF=0)
<b>JL</b>	; salta se minore (flag SF e OF diversi)
<b>JNO</b>	; salta se non c'è overflow (flag OF=0)
<b>JO</b>	; salta se c'è overflow (flag OF=1)
<b>JCXZ</b>	; salta se il registro contatore CX=0

Le istruzioni di salto, come quelle di trasferimento dati, non modificano i flag del registro di stato F per cui si deve prestare molta attenzione all'uso delle istruzioni di salto condizionato. Si consideri, ad esempio, il seguente programma:

```
MOV AX, 0000H ; trasferisci il numero 0000H nell'accumulatore
JZ Pippo ; salta alla label di nome Pippo se il flag Z=1
```

Il programma non salterà alla label Pippo anche se l'istruzione MOV ha memorizzato in AX=0. Per realizzare il salto alla label Pippo si deve utilizzare il seguente programma:

```
MOV AX, 0000H
OR AX, AX
JZ Pippo
```

In questo caso, l'operazione logica OR fornisce come risultato sempre AX=0 ma modifica i flag ed in particolare Z=1.

```
JZ Pippo ; salta alla label Pippo poiché Z=1.
```

## 7. ISTRUZIONI DI ITERAZIONE

Sono istruzioni utilizzate per ripetere più volte una stessa procedura. La più importante di tali istruzioni è:

### LOOP *label*

Decrementa il registro contatore, precedentemente inizializzato, e salta alla *label* finché CX non diviene uguale a zero.

#### Esempio

```
MOV CX, 04 ; poni nel registro CX il numero 4
L1: ----- ; istruzioni che si desidera iterare
-----
LOOP L1 ; decrementa CX e salta a L1 finché CX non diventa zero. Ovvero,
ripeti le istruzioni comprese tra L1 e LOOP del numero contenuto
inizialmente in CX (CX=4).
```

## 8. ISTRUZIONI SULLE STRINGHE

Sono istruzioni formate da operazioni elementari denominate *primitive* che permettono di effettuare dei trasferimenti di byte o di word tra due diverse zone di memoria. Tali istruzioni sono vantaggiosamente utilizzate per trasferire blocchi di dati grazie anche alla possibilità di ripetere più volte l'operazione con l'istruzione REP.

### MOVSB

Trasferisce un byte dalla locazione di memoria (sorgente) puntata da DS:SI alla locazione puntata da ES:DI. Dopo il trasferimento i registri SI e DI sono incrementati se il flag DF=0, mentre se DF=1 sono decrementati.

### **MOVSW**

Come la precedente istruzione ma per una word.

### **CMPSB**

Confronta il contenuto di due byte e aggiorna i registri SI e DI con le modalità viste per MOVSB.

### **CMPSW**

Come la precedente istruzione ma per una word.

### **LODSB**

Carica in AL il byte puntato dai registri ES:SI

### **LODSW**

Carica in AX la word puntata dai registri ES:SI

### **STOB**

Esegue l'operazione inversa alla LODSB.

### **STOW**

Esegue l'operazione inversa all LODSW.

### **REP**

Ripete l'istruzione sulla stringa finché il contenuto del contatore CX diventa zero. Questa istruzione va inserita prima dell'istruzione di stringa.

### Esempio

Il seguente programma permette di trasferire 100 byte che iniziano dalla locazione di memoria del segmento dati puntata da DS:0200 alle locazioni di memoria il cui inizio è nel segmento extra ed è puntata da ES:0400.

CLD	; pone il flag di direzione DF=0 in modo da realizzare l'autoincremento.
MOV CX, 100	; carica in CX il numero di elementi da trasferire.
LEA SI, 0200	; carica in SI l'offset della locazione iniziale della sorgente.
LEA DI, 0400	; carica in DI l'offset della locazione iniziale della destinazione.
REP	; ripeti la successiva istruzione di trasferimento finché CX è diverso da 0.
MOVSB	; esegui il trasferimento.

Nell'esempio si è supposto che i valori di DS e ES siano stati precedentemente definiti.

## PROGRAMMAZIONE ASSEMBLY IN AMBIENTE DEBUG

### **Premessa**

Si descrivono i principali comandi utilizzati per la stesura e la manipolazione di programmi scritti in linguaggio assembler mediante il DEBUG del sistema operativo MSDOS. E' bene ricordare che l'assemblatore richiede che i valori numerici siano espressi in forma esadecimale, che quando un operando è un indirizzo di memoria si deve specificare se si tratta di byte scrivendo il prefisso BY oppure di word con il prefisso WO in caso di ambiguità. In tutti i casi non è permesso l'uso di nomi simbolici o di etichette. In ambiente DEBUG sono consentite anche delle pseudoistruzioni come DB ( Define Byte ) e DW ( Define Word ) che permettono di allocare in memoria i dati a partire dall'indirizzo specificato dall'istruzione. I dati alfabetici devono essere racchiusi tra virgolette e sono memorizzati in codice ASCII.

Nelle istruzioni di salto o di chiamate a subroutine all'interno dello stesso segmento l'indirizzo di salto è indicato semplicemente tramite il suo offset. Se invece l'indirizzo di salto è prelevato dalla memoria è necessario specificare se è intrasegmentale con il prefisso NE ( NEAR-vicino ) o intersegmentale con il prefisso FAR ( lontano ). Se si omette il prefisso il salto è inteso intrasegmentale.

### **1. Comandi del DEBUG**

Per avviare il programma DEBUG digitare:

```
C:\>DEBUG
```

Se il programma già esiste digitare:

```
C:\>DEBUG nomefile.est
```

Avviato il DEBUG appare il prompt costituito da un trattino. Si possono immettere i vari comandi costituiti da lettere maiuscole o minuscole. Se nel comando si verifica un errore di sintassi, DEBUG riproduce la riga di comando ed indica l'errore con un accento circonflesso (^) e la parola *Errore*. Ad esempio:

```
D CS:100 CS:110          riga di comando
   ^Errore              risposta di DEBUG
```

Tutti i comandi di DEBUG accettano parametri, eccetto il comando Q. I parametri possono essere separati da caratteri di delimitazione come spazi o virgole. L'inserimento di tali parametri è obbligatorio soltanto tra due valori esadecimali consecutivi. I seguenti comandi sono equivalenti:

```
dcs:100 110
D CS:100 110
d, cs : 100 110
```

Si descrivono i comandi più importanti.

**-A [indirizzo]**

**ASSEMBLE** consente di assemblare il programma in linguaggio assembly a partire dall'indirizzo specificato. Se l'indirizzo è omissso il programma viene allocato a partire dall'ultima locazione di memoria precedentemente assemblata. Se il comando è usato per la prima volta e l'indirizzo è omissso il programma viene allocato all'indirizzo di offset 0100 del segmento di codice corrente ovvero CS:0100.

**-C intervallo indirizzo**

**COMPARE** confronta la porzione di memoria specificata dall'opzione *intervallo* con una porzione della stessa dimensione, a partire dall'indirizzo specificato. Se i due blocchi sono identici, non avviene alcuna visualizzazione e Debug ritorna al prompt altrimenti esse vengono visualizzate. Esempio:

-C 100, 1FF 300 ; confronta i blocchi di memoria da 100 a 1FF con i blocchi di memoria da 300 a 3FF.

**-D**

**DUMP** visualizza il contenuto di 128 byte nel segmento partendo dall'ultima locazione di memoria visualizzata da un precedente comando DUMP. Se usato per la prima volta parte dalla locazione DS:0100.

**-D [indirizzo]**

Visualizza 128 byte a partire dall'indirizzo specificato nel segmento dati.

-D indirizzo1, indirizzo2

Visualizza il contenuto della memoria compreso tra gli indirizzi specificati. I caratteri non stampabili sono indicati con dei puntini.

Esempio:

-D DS: 100 115 visualizza tutti i byte contenuti tra le righe 100 e 115 nel segmento DS.

**-D segmento:indirizzo**

Visualizza 128 byte relativi al segmento e all'indirizzo specificati.

Esempio:

-D CS : 0200 ; visualizza 128 byte a partire dall'indirizzo 200 nel segmento di codice CS.

Ciascun comando DUMP digitato senza parametri visualizza i byte immediatamente successivi agli ultimi visualizzati.

**-E indirizzo [elenco]**

**ENTER** consente di immettere nella locazione di memoria specificata un nuovo valore. Se, con il comando E, si digita un elenco di valori, i valori in byte a partire dall'indirizzo specificato vengono sostituiti dai valori dell'elenco. Nel caso si digita l'indirizzo senza elenco, Debug visualizza l'indirizzo ed il relativo contenuto, ripete l'indirizzo sulla riga seguente, ed attende l'input.

Esempio 1:

-E CS : 100 risponde con 04BA : 0100 EB.\_

Per cambiare questo valore, ad esempio, in 2F digitare il numero 2F. Per ignorare i byte seguenti si deve premere la barra spaziatrice. Premendo il tasto INVIO si ritorna ai comandi Debug.

### Esempio 2

-E ES: 200

-E 200 27 33 18 'MARE' ; inserisce a partire dalla locazione di memoria 0200 i numeri 27 33 18 e i caratteri relativi alla parola MARE.

### **-F intervallo elenco**

**FILL** riempie le posizioni di memoria comprese nell'*intervallo* specificato con i valori contenuti nell'*elenco*.

Esempio

-F 04BA:0100 L 100 40 12 34 67 23

Riempie le posizioni da 04BA:0100 fino a 04BA:1FF con i byte specificati. I cinque valori sono poi ripetuti fino al riempimento di tutti i 100H byte.

### **-G [=indirizzo]**

**GO** esegue il programma a partire dall'indirizzo specificato. Se si omette l'indirizzo il programma viene eseguito a partire dalla locazione di memoria individuata dai registri CS:IP.

### **-G =indirizzo1 indirizzo2**

Esegue le istruzioni comprese tra l'indirizzo1 e l'istruzione precedente l'indirizzo2.

### **-H valore1 valore2**

HEX esegue operazioni aritmetiche esadecimali sui due parametri specificati.

Esempio:

-H 19H 10H

02A9 0095            Debug risponde fornendo la somma e la differenza dei due valori indicati.

### **-L [indirizzo]**

**LOAD** carica il programma dall'indirizzo specificato. Se l'indirizzo è omissso il programma è caricato a partire dall'offset 0100 di CS. Per caricare un programma, precedentemente memorizzato, si può usare una delle seguenti sintassi:

a) C:\>DEBUG *nomeprogramma.COM*

b) C:\>DEBUG

-N *nomeprogramma*

-L [indirizzo] ; se l'indirizzo è omissso il programma è caricato da CS:0100.

### **-M indir.1 indir.2 indir.3**

**MOVE** permette di spostare i byte compresi tra indirizzo1 e indirizzo2 a partire da indirizzo3. L'area origine e destinazione possono essere sovrapposti.

### **-N nomefile.est**

**NAME** assegna il nome con cui salvare il programma.

Ad esempio:

-N PIPPO.COM ; assegna il nome PIPPO.COM al programma.

### **-Q**

**QUIT** consente di uscire dall'ambiente DEBUG e ritornare in DOS.

**-R**

**REGISTER** visualizza il contenuto di tutti i registri del  $\mu P$ , e l'istruzione presente all'indirizzo puntato dai registri CS:IP.

**-R nome registro**

Visualizza il contenuto del registro selezionato.

-R BX; comando per visualizzare il contenuto di BX

BX 0002 ; il registro BX contiene 0002

: ; si può immettere un nuovo valore in BX. Battendo il tasto INVIO si lascia inalterato il valore di BX.

**-R F**

**REGISTER FLAG** visualizza lo stato del registro dei FLAG con la seguente sintassi:

Nomi dei flag	Set (flag=1)	Clear (flag=0)
Overflow	OV=Overflow	NV=No overflow
Direzione	DN=Decremento	UP=Incremento
Interrupt	EI=Abilitato	DI=Disabilitato
Segno	NG=Negativo	PL=Positivo
Zero	ZR=Zero	NZ=Non zero
Riporto Ausil.	AC=si	NA=no
Parità	PE=Pari	PO=Dispari
Riporto	CY=Riporto	NC=Non riporto

**-T [=indirizzo]**

**TRACE** esegue il programma in single step dall'indirizzo specificato.

Esempio:

-T=0100 esegue l'istruzione presente all'indirizzo specificato e mostra il contenuto dei registri.

Digitando successivi comandi T (senza =indirizzo) vengono eseguite le istruzioni successive.

**-U [indirizzo]**

**UNASSEMBLE** consente di leggere i valori esadecimali allocati in memoria a partire dall'indirizzo specificato per 32 byte. Se l'indirizzo è omesso viene mostrato il contenuto della memoria partire dall'ultima disassemblata da un precedente comando. Se il comando è dato per la prima volta ed è omesso l'indirizzo il disassemblaggio inizia dalla locazione 0100 del segmento di codice corrente.

**-U indirizzo1 indirizzo2**

Consente di disassemblare le locazioni di memoria comprese tra gli indirizzi specificati.

**-W [ indirizzo ]**

**WRITE** salva su disco il programma che inizia all'indirizzo specificato. Il numero di byte di cui è costituito il programma deve essere caricato nei registri BX e CX. In CX si caricano le quattro cifre esadecimali di ordine inferiore e in BX quelle di ordine superiore. Spesso il numero di istruzioni del programma è limitato per cui si carica solo il registro CX e si pone BX=0: E' necessario, comunque, aver assegnato un nome al file con il comando N. Se l'indirizzo è omesso inizia da

0100 nel segmento corrente. Se si è fatto uso del comando G o del comando T bisogna reimpostare CX prima di usare il comando W senza parametri.

### Messaggi di errore

Durante la sessione di messa a punto, si potrebbe ricevere uno dei seguenti messaggi di errore. Qualsiasi errore chiude il comando Debug sotto il quale è avvenuto, senza però chiudere Debug.

- FE** Indicatore errato. I caratteri digitati non formano valori accettabili.
- BP** Troppi punti di interruzione ( più di 10) al comando G.
- RE** Registro errato. Si è digitato un nome di un registro non valido.
- DF** Indicatore doppio. Si sono digitati più valori per un indicatore.

## 2. STRUTTURA DI UN PROGRAMMA DEBUG

Si descrive la tipica struttura di un programma scritto in ambiente DEBUG.

Supponiamo di voler scrivere un programma di nome PIPPO.COM che si desidera salvare nell'unità floppy A. Si impartiscono i seguenti comandi:

```
C:\>DEBUG ; si attiva il programma debug
-N A:PIPP0.COM ; si assegna il nome, con estensione .COM, al programma
-A ; si immette il comando A per assemblare il programma
0CA2:0100 1° Istruzione ; si scrive la 1° istruzione a partire dall'offset 0100 del segmento CS
; Il valore CS=0CA2 potrebbe essere diverso
----- ; si scrivono le istruzioni successive
-----
-----
0CA2:013B RETF ; l'ultima istruzione è, normalmente, RETF ed indica la fine del
; programma.
; si batte 2 volte sul tasto INVIO per tornare al prompt di debug.
```

Si osservi che il programma è compreso tra gli offset 0100 e 013B e, pertanto, occupa 3C byte locazioni di memoria. Per salvare il programma sul floppy dell'unità A si impartiscono i seguenti comandi:

```
-R BX ; per analizzare il contenuto del registro BX
BX 0CA2 ; contenuto di BX
: 0 ; si pone BX=0
-R CX
CX 0A27 ; per analizzare il contenuto di CX
: 003C ; si pone in CX il numero di byte delle istruzioni del programma
-W ; salva il programma
```

Per leggere, successivamente, il programma si può utilizzare una delle seguenti procedure:

- 1) C:\>DEBUG A:PIPP0.COM ; con il comando -U è possibile analizzare il programma
- 2) C:\>DEBUG  
-N A:PIPP0.COM

-L ; il programma è caricato all'offset 0100 del segmento di codice corrente.

## Esempi di programmi in DEBUG

### Esempio 1

Si riporta il listato di un programma scritto in debug in grado di leggere dei caratteri da tastiera, memorizzarli in un banco dati e visualizzarli su monitor.

```
C:\>debug a:LEGGI.COM ; richiama dall'unità floppy A il file LEGGI.COM
-u 100 116 ; mostra il listato delle istruzioni del programma

104A:0100 BE0002 MOV SI,0200 ; pone SI=200. Inizio banco dati.
104A:0103 B407 MOV AH,07 ; insieme alla successiva istruzione consente di
104A:0105 CD21 INT 21 ; memorizzare in AL il codice del tasto premuto.
104A:0107 3C0D CMP AL,0D ; confronta AL con il codice del tasto INVIO.

104A:0109 740B JZ 0116 ; se premuto il tasto INVIO salta a RETF.
104A:010B 8804 MOV [SI], AL ; salva il valore di AL nel banco dati.
104A:010D 46 INC SI ; incrementa il contenuto di SI.
104A:010E 88C2 MOV DL, AL ; insieme alle successive due istruzioni consente di
104A:0110 B406 MOV AH, 06 ; visualizzare il valore contenuto in DL sul monitor
104A:0112 CD21 INT 21
104A:0114 EBED JMP 0103 ; ritorna all'indirizzo 0103 per ripetere l'acquisizione
104A:0116 CB RETF ; fine del programma

-g = 100 116 ; vengono eseguite le istruzioni comprese tra gli
indirizzi ;
; 0100 e 0116
1 2 3 4 5 6 ; caratteri immessi da tastiera, con spazio ( codice 20)
; tra i caratteri numerici.
```

; dopo aver premuto il tasto INVIO si ottiene la seguente schermata:

```
AX=070D BX=0000 CX=0017 DX=0036 SP=FFFE BP=0000 SI=020B DI=0000
DS=104A ES=104A SS=104A CS=104A IP=0116 NV UP EI PL ZR NA PE NC
104A:0116 CB RETF
```

```
-d 200 210 ; comando per analizzare le locazioni di memoria tra
; 200 e 210
```

```
104A:0200 31 20 32 20 33 20 34 20-35 20 36 57 56 9C C6 06 1 2 3 4 5 6WV...
104A:0210 6C l
```

Se si analizzano i primi 11 byte ( B in esadecimale) a partire dalla locazione 200 si possono ritrovare i valori relativi ai tasti premuti.

**N.B.** I commenti inseriti dopo il punto e virgola nel listato non si devono scrivere quando si assembla il programma in ambiente Debug.

### Esempio 2

Programma per il trasferimento di 10 byte dall'indirizzo 200H all'indirizzo 300H.

```
-U 100 10C
18BA:0100 FC          CLD
18BA:0101 BE0002     MOV  SI,0200
18BA:0104 BF0003     MOV  DI,0300
18BA:0107 B90A00     MOV  CX,000A
18BA:010A F3         REPZ
18BA:010B A4         MOVSB
18BA:010C CB         RETF
-
-D 200 20F
18BA:0200 9A 4F A5 4F B0 4F BB 4F-C6 00 00 4F D1 4F DC 4F .O.O.O.O...O.O.O
-
-D 300 30F
18BA:0300 74 34 B8 06 33 CD 21 8A-C2 8A FE 3C 19 76 02 B0 t4..3.!....<.v..
-
-G=100 10C
```

```
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=020A DI=030A
DS=18BA ES=18BA SS=18BA CS=18BA IP=010C NV UP EI PL NZ NA PO NC
18BA:010C CB          RETF
-D 200 20F
18BA:0200 9A 4F A5 4F B0 4F BB 4F-C6 00 00 4F D1 4F DC 4F .O.O.O.O...O.O.O
-D300 30F
18BA:0300 9A 4F A5 4F B0 4F BB 4F-C6 00 FE 3C 19 76 02 B0 .O.O.O.O...<.v..
-
-R CX
CX 0000
:0010
-N STRINGA1.COM
-W
Scrittura di 00010 byte in corso
```

### Esempio 3

Programma per il confronto di un blocco di 10 byte posto a partire dall'indirizzo 0200 con un blocco di 10 byte posto a partire dall'indirizzo 300. Quando si incontra l'istruzione REPZ si incrementano SI e DI, si decrementa CX e se i byte confrontati sono uguali si ripete il confronto finché CX=0 oppure se si incontrano due byte omologhi non coincidenti. Nel seguente esempio si suppongono coincidenti 3 byte consecutivi per cui il programma arresta l'esecuzione con CX=6.

```
-D 200 20F
18BA:0200 9A 4F A5 4F B0 4F BB 4F-C6 00 00 4F D1 4F DC 4F .O.O.O.O...O.O.O
-D 300 30F
18BA:0300 9A 4F A5 4F B0 4F BB 4F-C6 00 FE 3C 19 76 02 B0 .O.O.O.O...<.v..
```

-E 300  
18BA:0300 9A. 4F. A5. 4F.FF

-  
-U 100 10C  
18BA:0100 FC CLD  
18BA:0101 BE0002 MOV SI,0200  
18BA:0104 BF0003 MOV DI,0300  
18BA:0107 B90A00 MOV CX,000A  
18BA:010A F3 REPZ  
18BA:010B A6 CMPSB  
18BA:010C CB RETF

-  
-G= 100 10C

AX=0000 BX=0000 CX=0006 DX=0000 SP=FFEE BP=0000 SI=0204 DI=0304  
DS=18BA ES=18BA SS=18BA CS=18BA IP=010C NV UP EI PL NZ NA PE CY  
18BA:010C CB RETF

-R CX  
CX 0006  
:0010  
-N STRINGA2.COM

-W  
Scrittura di 00010 byte in corso

#### Esempio 4

Programma che genera un ritardo di alcuni secondi. Il ritardo dipende dalla frequenza di clock del computer e dai valori caricati nei registri CX e DX.

A:\>DEBUG

-  
-N RITARDO.COM

-  
-L  
-  
-U 100 10F  
19C9:0100 BAFF00 MOV DX,00FF  
19C9:0103 B9FFFF MOV CX,FFFF  
19C9:0106 90 NOP  
19C9:0107 E2FD LOOP 0106  
19C9:0109 4A DEC DX  
19C9:010A 75F7 JNZ 0103  
19C9:010C B8004C MOV AX,4C00  
19C9:010F CD21 INT 21

-  
-G=100 10F

AX=4C00 BX=0000 CX=0000 DX=0000 SP=FFFE BP=0000 SI=0000 DI=0000  
DS=19C9 ES=19C9 SS=19C9 CS=19C9 IP=010F NV UP EI PL ZR NA PE NC  
19C9:010F CD21 INT 21

-



## MACROASSEMBLER

### Generalità

Si descrivono le fasi operative necessarie per la messa a punto di un programma in assembler mediante l'uso dell'assemblatore MASM della Microsoft:

- 1) l'editing
- 2) l'assemblaggio
- 3) il link

La **fase di editing** consiste nello scrivere le istruzioni del programma che si intende eseguire in codice mnemonico utilizzando un qualsiasi EDITOR di testi, come ad esempio il programma EDIT di MSDOS.

Il programma deve in tutti i casi terminare con l'istruzione END.

Il listato ottenuto rappresenta il *codice sorgente* del programma da eseguire e deve essere salvato attribuendogli un nome e l'estensione ASM. Ad esempio, il programma può essere salvato col nome PLUTO.ASM.

La **fase di assemblaggio** consente di tradurre il codice sorgente ottenuto nella fase di editing nel *codice oggetto rilocabile*. Ciò si ottiene eseguendo il programma MASM seguito dal nome del programma di lavoro. Ad esempio:

```
C:\> MASM PLUTO.ASM;
```

L'estensione può essere omessa poiché MASM accetta solo file con estensione ASM. L'assemblatore controlla la presenza di eventuali errori e se non ve ne sono genera il file PLUTO.OBJ.

Se nella riga di comando è omesso il punto e virgola dopo l'estensione ASM allora l'assemblatore ci chiederà se vogliamo generare altri due file uno con estensione LST che contiene i numeri di linea, la traduzione in codice macchina e la tabella dei simboli del codice sorgente, l'altro, con estensione CRF che fornisce una lista dei simboli e il numero di riga in cui essi compaiono. Tale file è utile in fase di debug. Se non si desidera generare tali file è sufficiente premere il tasto INVIO.

La **fase di link** permette di tradurre il file con estensione .OBJ ottenuto in fase di assemblaggio in un file eseguibile con estensione .EXE. Ciò si ottiene col seguente comando:

```
C:\>LINK PLUTO.OBJ;
```

L'estensione può essere omessa poiché LINK accetta solo file con estensione .OBJ. Viene generato, in tal caso, un file eseguibile di nome PLUTO.EXE. Se nella riga di comando si omette il punto e virgola il programma LINK ci chiederà se vogliamo generare un file con estensione .MAP che contiene gli indirizzi fisici dei segmenti e le loro dimensioni. Infine LINK ci chiederà il nome di eventuali librerie di collegamento con la scritta:

Libraries (.LIB )

E' sufficiente premere il tasto INVIO per ignorare tali operazioni.

## 1. LA STRUTTURA DEI DATI

I dati sono definiti in diversi formati e ad essi è possibile associare anche un nome o *etichetta*. Si riportano alcuni esempi esplicativi delle possibili strutture dei dati con relativo commento.

LUCA DB 54 ; alla variabile LUCA è associato il numero decimale 54 di tipo byte.

MAR DW 0A5F3H ; alla variabile MAR è associato il numero esadecimale A53F di tipo word. Lo zero alla sinistra è necessario quando il numero esadecimale inizia con un carattere alfabetico.

BIX DB 10011101B ; alla variabile BIX è associato il numero binario 10011101 di tipo byte.

ALFA DB 'E' ; alla variabile ALFA è associato il carattere ASCII della lettera E di tipo byte.

PIPPO DB 'Sei bravo',13,10 ; la variabile PIPPO contiene una sequenza di caratteri che termina con un ritorno a capo ( codice ASCII 13 ) e un avanzamento di linea ( codice ASCII 10 ) .

PIPPO [2] ; consente di indirizzarsi ai singoli caratteri contenuti in una variabile stringa. Il numero tra parentesi rappresenta la posizione del carattere nella stringa. Nell'esempio la lettera 'i' della stringa 'sei bravo'.

PLUTO DB\$-stringa ; la variabile PLUTO, di tipo byte, ha un valore pari al numero di caratteri contenuti nella variabile *stringa*.

ROBY DD ? ; la variabile ROBY è di tipo Double Word a 4 byte e ad essa non è stato associato alcun valore.

Le variabili si possono, inoltre, definire di tipo DQ ( Quad Word a 8 byte ) e di tipo DT ( Tenbyte a 10 byte ).

Si ricordi che in ambiente DEBUG i dati numerici sono implicitamente considerati di tipo esadecimale, se non diversamente specificato.

Mediante l'operatore DUP(xx) è possibile assegnare ad una variabile lo stesso valore ripetuto più volte e indicato tra parentesi. Ad esempio:

NAVE DB 50 DUP(0) ; assegna alla variabile NAVE 50 byte tutti di valore 0

MARE DW 100 DUP(27) ; assegna alla variabile MARE 100 word tutte di valore 27.

Anche la direttiva EQU permette di assegnare ad una variabile un valore numerico oppure una etichetta. Ad esempio.

LUNA EQU 2BC7H ; assegna a LUNA il valore esadecimale 2BC7.

SOLE EQU 'Sei bravo'; assegna a SOLE la stringa *Sei bravo*.

## 2. STRUTTURA DEL PROGRAMMA SORGENTE

Il programma sorgente è costituito da due parti fondamentali, una relativa all'algoritmo software vero e proprio, l'altra relativa all'assemblatore che fornisce le *direttive* necessarie per individuare l'inizio e la fine dei diversi segmenti in cui è divisa la memoria in funzione del programma da realizzare. Tali direttive si riferiscono al solo programma assemblatore e, pertanto, non fanno parte del file relativo al codice eseguibile.

La parte relativa all'algoritmo è una *procedura* che inizia con la direttiva **PROC** e termina con **ENDP** (fine procedura). La direttiva di procedura può essere definita **NEAR** o **FAR** a seconda se ci si riferisce ad indirizzi all'interno dello stesso segmento ( *infrasegmentale* ) o che interessa altri segmenti ( *intersegmentale* ). Ad esempio:

```
MAIN  PROC  FAR
      -----
      CALL PIPPO
      -----
      RET
MAIN  ENDP

PIPPO  PROC  NEAR
      -----
      -----
      RET
PIPPO  ENDP
```

La procedura di nome MAIN è dichiarata di tipo lontana FAR (tra diversi segmenti) ed è costituita da un insieme di istruzioni (non specificate ed indicate con dei trattini) che termina con ENDP. All'interno di tale procedura è richiamata un'altra procedura denominata PIPPO di tipo vicina (all'interno di uno stesso segmento) NEAR. Le istruzioni RET sono necessarie per indicare al  $\mu P$  la fine della procedura.

## 3. DIRETTIVE ALL'ASSEMBLATORE

### 3.1. LA DIRETTIVA SEGMENT

La parte relativa ai segmenti di memoria inizia con la direttiva **SEGMENT** e termina con **ENDS** (fine direttiva di segmento) attribuendo, comunque, all'inizio un nome alla direttiva. La struttura generale di una direttiva di segmento da fornire all'assemblatore è del tipo:

*nome* SEGMENT [*align*] [*combine*] [*'class'*]

Le parentesi quadre ci informano, come al solito, che i parametri indicati sono opzionali.

Si descrive la struttura della direttiva SEGMENT.

*nome* è l'etichetta che fornisce il programmatore per individuare il segmento.

SEGMENT indica che si sta operando su di una direttiva relativa ad un segmento di memoria.

Il parametro **Align** indica al LINKER come allocare il segmento nella memoria. Align può assumere i seguenti parametri: PAGE, PARA, WORD, e BYTE.

Con PAGE si ha un allineamento all'inizio della pagina di memoria. Ogni pagina, inizia ad un multiplo di 256 byte.

Con il parametro PARA, tra i più usati, si ha l'allineamento all'inizio del paragrafo successivo ed è quindi possibile l'indirizzamento a tutte le 64K locazioni di memoria di un segmento.

Con WORD si fa iniziare il segmento all'indirizzo pari.

Con BYTE il segmento inizia un byte dopo l'ultimo segmento definito. In tal caso si ottimizza l'utilizzo della memoria ma risulta più complesso l'indirizzamento poiché i registri del  $\mu P$  consentono l'indirizzamento con passi di 16 byte corrispondenti ad un paragrafo.

Il parametro **Combine** indica al LINKER come deve trattare più segmenti aventi stesso nome relativi, però, a moduli oggetto diversi. Se combine non viene specificato il LINKER elabora i segmenti con stesso nome come se avessero nomi diversi. Ovviamente combine non ha alcun effetto su segmenti con nomi diversi. Combine può assumere i seguenti parametri: PUBLIC, STACK, COMMON, MEMORY e AT.

Il parametro PUBLIC permette di concatenare in un unico segmento più segmenti con stesso nome.

Il parametro STACK è equivalente a PUBLIC ma nel caso in cui il segmento si riferisce allo stack il parametro combine deve assumere il valore STACK e non può essere omissso pena un messaggio d'errore da parte del LINKER. Tale errore può essere ignorato solo se il programma prevede, in qualche suo punto, la inizializzazione dei registri SS e SP.

Il parametro COMMON consente di sovrapporre più segmenti con stesso nome. Il segmento risultante ha lo stesso nome e la stessa dimensione del più grande.

Il parametro MEMORY si comporta come PUBLIC ed è utilizzato solo in caso di incompatibilità con altri LINKER della Intel.

Il parametro AT *indirizzo* indica al LINKER che il segmento dovrà risiedere all'indirizzo assoluto specificato da *indirizzo*.

Il parametro **class** consente di classificare in modo differenziato i vari segmenti. Ciò risulta utile quando si linkano più moduli nei quali alcuni segmenti devono essere raggruppati. Il parametro class è definito dal programmatore ma alcuni parametri sono di particolare importanza come:

CODE, DATA, CONST, STACK e BSS. In particolare BSS è utilizzato per concatenare dati non inizializzati. Se il LINKER è eseguito con l'opzione */dosseg* il segmento class CODE è inserito per primo anche se nel programma sorgente è ultimo.

### 3.2. LA DIRETTIVA ASSUME

La direttiva ASSUME serve ad indicare all'assemblatore quale nome di segmento è da considerarsi associato ad un registro di segmento in una determinata porzione del programma. La struttura generale è la seguente:

ASSUME CS:CODICE , DS:DATI, SS:STACK

La direttiva ASSUME è fondamentale poiché in memoria possono risiedere più segmenti dello stesso tipo, all'interno della procedura principale, e pertanto si deve comunicare all'assemblatore quale di questi si intende usare. In questo modo il programmatore può fare a meno di specificare l'indirizzo di segmento per ogni dato, indirizzo o elemento dello stack e potrà quindi usare solo l'offset.

### 4. STRUTTURA DI UN PROGRAMMA IN MACROASSEMBLER

Si descrive la struttura tipica con cui deve essere scritto un programma in assembler in ambiente MASM. Ovviamente si possono dare altre strutture, ma quella che si descrive è idonea per la stesura e la comprensione di numerosi programmi di uso generale.

Il programma inizia con la definizione dei segmenti di STACK, DATI e CODICE . Ciò che è scritto dopo il punto e virgola è un commento ed è ignorato in fase di assemblaggio.

#### Struttura del programma

##### **; definizione del segmento di stack**

```
STACK      SEGMENT      PARA      STACK      'STACK'
```

; in questa parte del programma deve essere definita la struttura dello STACK. Il successivo  
; comando indica la fine della direttiva di STACK.

```
STACK      ENDS                ; fine direttiva di STACK
```

##### **; definizione del segmento dei dati**

```
DATA       SEGMENT      PARA      PUBLIC     'DATA'
```

; in questa parte del programma devono essere definite le variabili e i dati che si intendono  
; utilizzare. Il successivo comando indica la fine della direttiva DATA.

```
DATA       ENDS                ; fine direttiva DATA
```

##### **; definizione del segmento di codice**

```
CODE       SEGMENT      PARA      PUBLIC     'CODE'
```

; I precedenti segment STACK, DATA e CODE sono stati allineati al paragrafo con PARA;  
 ; come parametro di *combine* si è utilizzato *combine= PUBLIC* sia per il segmento DATI  
 ; che per quello di CODICE, mentre per lo STACK si è utilizzato *combine= STACK*. Ciò  
 ; permette al LINKER di inizializzare correttamente i registri SS e SP.

**; definizione della procedura (PROC) per il programma principale (MAIN)**

```
MAIN      PROC      FAR
```

; il programma principale è una procedura dichiarata lontana FAR. La procedura può anche  
 ; essere dichiarata vicina con NEAR.

**; indirizzamento dei segmenti**

```
ASSUME   CS:CODE,  DS:DATA,  SS:STACK
```

; La direttiva ASSUME, non viene assemblata, ma serve all'assemblatore per indicare quale  
 segmento è da considerarsi associato in un registro di segmento in una particolare area di  
 memoria.

; Un programma scritto con il Macroassembler MASM deve, normalmente, contenere le  
 ; seguenti istruzioni necessarie per salvare il contenuto di alcuni registri affinché al  
 termine ; del programma il sistema possa tornare in ambiente DOS in modo corretto. Ciò  
 perché ; quando si lancia un programma eseguibile, il DOS, costruisce un blocco di  
 controllo ; chiamato PSP ( Programm Prefix Segment) in cui immagazzina tutte le  
 informazioni per ; l'esecuzione del programma; successivamente legge il codice, i dati , lo  
 stack e il PSP.

```
PUSH     DS           ; salva DS nello STACK
SUB      AX, AX       ; azzera AX
PUSH     AX           ; salva AX nello STACK
MOV      AX, DATA    ; trasferisce DATA in AX
MOV      DS, AX       ; trasferisce AX in DS
```

**; inserimento delle istruzioni del programma principale.**

```
MAIN     ENDP         ; fine procedura
CODE     ENDS         ; fine direttiva CODE
        END  MAIN     ; fine del programma
```

**Esempio di un semplice programma**

Si riporta il listato di un semplice programma in MASM che consente di scrivere sullo schermo il  
 messaggio ' Questo è il mio primo programma'.

```

DATA      SEGMENT      PARA      'DATA'
          messaggio    DB 'Questo è il mio primo programma', '$'
DATA      ENDS

STACK    SEGMENT      PARA      STACK      'STACK'
STACK    ENDS          ; lo stack non è utilizzato in questo programma e
          ; pertanto, non è richiesta alcuna definizione.

CODE     SEGMENT      PARA      'CODE'
          ASSUME      CS:CODE, DS:DATA, SS:STACK

MAIN     PROC          NEAR

          PUSH        DS          ; salva DS nello STACK
          SUB         AX, AX      ; azzera AX
          PUSH        AX          ; salva AX nello STACK

          MOV         AX, DATA   ; trasferisce DATA in AX
          MOV         DS, AX      ; trasferisce AX in DS
          ; seguono le istruzioni del programma principale

          MOV         AX, DATA   ; L'indirizzo di segmento in cui si trova il
          MOV         DS, AX      ; messaggio da visualizzare viene caricato in
DS.
          LEA         DX, messaggio ; l'offset viene caricato in DX
          ; in DS:DX c'è il primo carattere della stringa
da
          MOV         AH, 09H     ; stampare che deve terminare con $.
          INT         21H        ; consente di stampare la stringa sullo schermo.

          ; seguono le istruzioni di chiusura del programma

          MOV         AX, 4C00H   ; ritorno al DOS
          INT         21H
          MAIN       ENDP
          CODE       ENDS
          END        MAIN        ; fine del programma

```

Scritto il programma con un editor di testi (ad esempio, Edit del DOS) lo si deve salvare assegnandogli un *nome* e l'*estensione* .ASM, successivamente si deve tornare in DOS ed impartire i seguenti comandi:

- C:\> cd MASM ; per entrare in ambiente MASM
  - C:\MASM>MASM ; per lanciare il programma MASM
  - Source filename [.ASM] *nome programma*; ; fase di assemblaggio. Genera il file *nomeprogramma.OBJ*
  - C:\MASM>LINK
  - Object Modules [.OBJ] *nome programma*; ; fase di link genera Genera il file *nomeprogramma.EXE*
- C:\MASM> *nome programma* ; esecuzione del file eseguibile

## **5. ANALISI IN AMBIENTE DEBUG**

Se il programma non prevede che i risultati siano mostrati su video o su stampante, essi saranno stati memorizzati in prefissate locazioni di memoria e per poter analizzare tali risultati è necessario provare il programma in ambiente DEBUG.

Si deve lanciare il DEBUG con il comando:

```
C:\>DEBUG nome programma.EXE
```

In tal modo si entra in DEBUG e il programma è caricato ma non eseguito. Si analizza lo stato dei registri con il comando:

```
-R
```

In particolare CS:IP forniscono l'indirizzo della prima istruzione del programma. Per disassemblare le istruzioni si deve immettere il comando:

```
-U CS:IP
```

Individuati gli indirizzi relativi alla prima e ultima istruzione ( RETF ) si potrà eseguire il programma con il comando:

```
-G= Indirizzo prima istruzione      Indirizzo ultima istruzione
```

Per analizzare i dati elaborati dal programma si deve, in generale, visualizzare lo stato dei registri e i valori memorizzati nel segmento dati DS con il comando:

```
-D DS:0
```

Di seguito si riportano alcuni listati di programmi scritti con MASM e provati in ambiente DEBUG.

## SOMMARIO

<b><i>HARDWARE E SOFTWARE DEL <math>\mu</math>P 8086</i></b>	<b><i>1</i></b>
<b>Premessa</b>	<b>1</b>
<b>1. REGISTRI INTERNI</b>	<b>3</b>
<b>2. GESTIONE DELLA MEMORIA</b>	<b>5</b>
<b>3. GESTIONE DELLE INTERRUZIONI NEL <math>\mu</math>P 8086</b>	<b>6</b>
3.1 Tabella delle interruzioni o dei puntatori	7
3.2 Interruzioni del BIOS e del DOS	8
<b>4. METODI DI INDIRIZZAMENTO</b>	<b>9</b>
Indirizzamento immediato	9
Indirizzamento implicito	9
Indirizzamento con registro	9
Indirizzamento diretto	9
Indirizzamento indiretto	9
Indirizzamento indiretto <i>con base</i>	10
Indirizzamento indiretto <i>con indice</i>	10
Indirizzamento indiretto <i>con indice e con base</i>	10
Indirizzamento di stringhe	10
<b><i>SET DI ISTRUZIONI DEL 8086</i></b>	<b><i>11</i></b>
<b>1. ISTRUZIONI DI TRASFERIMENTO</b>	<b>11</b>
MOV destinazione, sorgente	11
PUSH sorgente	12
POP destinazione	12
XCHG destinazione, sorgente	12
XLAT	12
IN accumulatore, porta	12
OUT porta, accumulatore	12
LDS registro, memoria (LES registro, memoria)	12
LEA registro, DATO	12
<b>2. ISTRUZIONI ARITMETICHE</b>	<b>13</b>
ADD AX, DATO	13
ADC AX, DATO	13
SUB AX, BX	13
SBB	13
AAA (AAS)	13
DAA (DAS)	14
MUL sorgente	14
IMUL sorgente	14
DIV sorgente	14
IDIV sorgente	14
AAM	14
AAD	14
INC sorgente	15
DEC sorgente	15
NEG sorgente	15
CMP sorg1, sorg2	15
<b>3. ISTRUZIONI LOGICHE</b>	<b>15</b>
AND dest., sorg.	15
OR dest., sorg.	15
XOR dest., sorg.	15

TEST dest., sorg. _____	15
<b>4. ISTRUZIONI DI CONTROLLO _____</b>	<b>15</b>
<b>5. ISTRUZIONI DI ROTAZIONE E SHIFT _____</b>	<b>16</b>
<b>6. ISTRUZIONI DI SALTO _____</b>	<b>17</b>
CALL label _____	17
RET _____	17
RETF _____	17
6.1. Istruzioni di salto incondizionato _____	17
6.2. Istruzioni di salto condizionato _____	18
<b>7. ISTRUZIONI DI ITERAZIONE _____</b>	<b>19</b>
LOOP <i>label</i> _____	19
<b>8. ISTRUZIONI SULLE STRINGHE _____</b>	<b>19</b>
MOVSB _____	19
MOVSW _____	20
CMPSB _____	20
CMPSW _____	20
LODSB _____	20
LODSW _____	20
STOB _____	20
STOW _____	20
REP _____	20
 <b><i>PROGRAMMAZIONE ASSEMBLY IN AMBIENTE DEBUG _____</i></b>	 <b><i>21</i></b>
<b>Premessa _____</b>	<b>21</b>
<b>1. Comandi del DEBUG _____</b>	<b>21</b>
Messaggi di errore _____	25
<b>2. STRUTTURA DI UN PROGRAMMA DEBUG _____</b>	<b>25</b>
 <b><i>MACROASSEMBLER _____</i></b>	 <b><i>30</i></b>
<b>Generalità _____</b>	<b>30</b>
<b>1. LA STRUTTURA DEI DATI _____</b>	<b>31</b>
<b>2. STRUTTURA DEL PROGRAMMA SORGENTE _____</b>	<b>32</b>
<b>3. DIRETTIVE ALL'ASSEMBLATORE _____</b>	<b>32</b>
3.1. LA DIRETTIVA SEGMENT _____	32
3.2. LA DIRETTIVA ASSUME _____	34
<b>4. STRUTTURA DI UN PROGRAMMA IN MACROASSEMBLER _____</b>	<b>34</b>
<b>5. ANALISI IN AMBIENTE DEBUG _____</b>	<b>37</b>