# ASSEMBLER 8086 8088

*di Pierluigi Panunzi*

# La gestione della memoria
## (2ª parte)

Prima di introdurre nella presente puntata nuovi concetti, riteniamo molto utile fare una specie di riassunto dei principali concetti espressi la scorsa puntata, che bisogna avere ben chiari in mente prima di poter applicarli.

In poche parole, ricordiamo che i quattro segmenti, rispettivamente di codice, di dati, dello stack e di dati extra, sono tutti e quattro di ampiezza pari a 64 kbyte allocati nello spazio di memoria di 1 Mbyte gestibile dal microprocessore, a seconda del valore posto nei corrispondenti registri (CS, DS, SS, ES).

In particolare ricordiamo che il valore posto in ognuno dei quattro *segment register* rappresenta il *paragrafo* dal quale inizia il corrispondente segmento: sappiamo anche che tale valore viene detto genericamente «segment address» o «base address», contrapposto al cosiddetto «offset», questa volta dipendente dal tipo di segmento considerato.

Non fa certo male ricordare a questo punto che quando si parla di indirizzi all'interno di un programma, allora l'offset è dato dal registro IP (Instruction Pointer), mentre parlando di stack ad esempio durante operazioni di tipo «Push» e «Pop», allora l'offset è fornito dal registro SP (Stack Pointer).

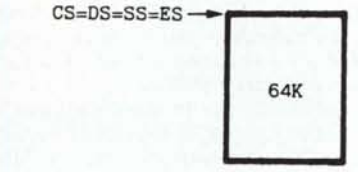
Abbiamo inoltre visto nelle scorse puntate che la prima operazione da compiere all'inizio del nostro programma deve essere quella di riempire opportunamente i registri di segmento (ad eccezione di CS) con delle istruzioni di MOV; abbiamo anche visto che bisogna «informare» l'assemblatore comunicandogli con la direttiva «ASSUME» i valori da assegnare ai registri: in alcuni casi particolari si può derogare a questa regola, ottenendo (a patto di non caricare o gestire i segment register nel corso del programma) un «programma dinamicamente rilocabile».

Caratteristica di tale programma è la possibilità di risiedere in un qualsiasi punto della memoria, senza avere differenze nel suo funzionamento.

Il fatto dunque di poter assegnare ai quattro registri di segmento (o meglio, eccettuato il CS, sul quale ritorneremo in dettaglio) dei valori anche nel corso del programma, conferisce all'assembler dell'8086/8088 una notevolissima potenza e flessibilità, soprattutto nel caso del «Multi-tasking».

Ecco che, facendo riferimento alle figure 1 e 2, possiamo vedere come comportarci in quattro situazioni differenti e come viceversa l'assembler si comporta di conseguenza alle nostre scelte.

Figura 1a




```
          NAME PROVA

PROGRAM   SEGMENT
          ASSUME CS:PROGRAM,DS:PROGRAM
          ASSUME ES:NOTHING,SS:PROGRAM

          MOV AX,PROGRAM
          MOV DS,AX
          MOV SS,AX
          MOV SP,OFFSET TOP

          MOV AX,ALFA
          ADD AX,BETA
          CALL DIVIDI
          MOV GAMMA,AL
          MOV DELTA,AH

          RET
ALFA      DW ?
BETA      DW ?
GAMMA     DB ?
DELTA     DB ?

DIVIDI:   SHR AX,1
          RET

STACK     DW 20 DUP(?)
TOP       LABEL NEAR

PROGRAM   ENDS

          END
```

*Listato 1 - Esempio di programma relativo alla situazione di figura 1a.*

### Analisi della figura 1a

È questo il caso di un sistema dotato di microprocessore 8086 (oppure 8088, il che non modifica il ragionamento), con «soltanto» 64 kbyte di memoria: praticamente è il caso a cui sono abituati i programmatori di micro ad 8 bit.

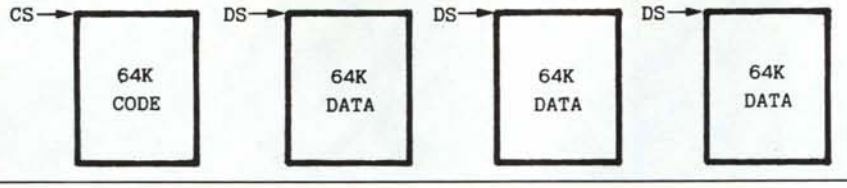
Ponendo allo stesso valore i segment register si ha che i quattro segmenti sono effettivamente sovrapposti e per questo motivo bisogna stare più che attenti che non si verifichino «sconfinamenti» tra le varie zone di programma, di dati e dello stack: in questo caso il programmatore deve prevedere l'estensione delle varie zone, soprattutto per quanto riguarda la zona riservata allo stack, in generale la più delicata.

Comunque 64 kbyte sono sempre tanti ed in generale con programmi non proprio particolarissimi non si hanno grossi problemi.

Come esempio della situazione di figura 1a, abbiamo riportato il mini-programma del listato numero 1: come programma non svolge che delle semplicissime funzioni (come pure gli altri tre che presenteremo), ma è importante in quanto racchiude in sé parecchie caratteristiche dell'assembler 8086/8088.

Anche se in questo momento alcuni degli elementi del programma rimarranno alquanto oscuri, è viceversa impor-

Figura 2a




tante capire bene i concetti base: nel corso delle varie puntate avremo modo di ritornare parecchie volte sui singoli argomenti.

Analizzando perciò il primo listato (e così sarà per gli altri), cerchiamo dunque di individuare e verificare i concetti sui quali ci siamo più volte soffermati: per quanto riguarda le istruzioni vere e proprie diciamo subito che basta un piccolissimo sforzo di immaginazione per interpretarne il funzionamento, dal momento che sono quasi sempre le stesse...

Abbiamo dunque detto che la nostra zona di memoria è ampia 64 kbyte ed in essa devono risiedere i quattro segmenti sovrapposti: dato che ancora non ci interessa, trascureremo sempre l'Extra Segment, per poi analizzarne le caratteristiche e funzioni a tempo debito.

Ecco che dunque il nostro programma è «chiuso» su di un unico segmento chiamato PROGRAM, contenente il programma (le istruzioni MOV, ADD, CALL, RET e SHR), la zona dati (le variabili ALFA, BETA, GAMMA e DELTA) e lo stack (formato da 20 word).

L'assemblatore è informato della situazione tramite la direttiva **ASSUME,** che assegna ai tre segment register il paragrafo dal quale inizia il segmento PROGRAM.

Come si può vedere scorrendo il piccolo listato, si nota addirittura una sovrapposizione (controllata!) della zona dati nella zona di programma, dal momento che le variabili vengono definite tra una «RET» ed un'istruzione di «shift a destra» posta all'etichetta «DIVIDI»; quindi vediamo lo stack posto subito dopo la RET della mini-subroutine ed ampio appunto 20 word.

Ritroviamo ancora una volta che lo Stack Pointer (SP) deve essere inizializzato con l'indirizzo «più alto» della zona di memoria assegnata allo stack stesso, dal momento che l'SP dell'8086, analogamente a quello di tutti i microprocessori, «cresce all'indietro» e cioè verso indirizzi più bassi: infatti le 20 word iniziano all'indirizzo dato dall'etichetta «STACK» e terminano all'indirizzo fornito dall'etichetta «TOP».

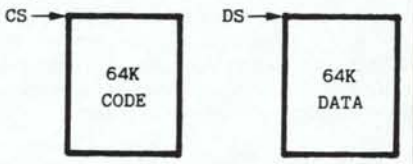
Dal momento che lo stack segment è gestito dalla coppia di registri SS e SP, ecco che SS viene caricato con il valore del paragrafo a cui inizia il segmento «PROGRAM», mentre SP viene caricato con l'offset dell'etichetta «TOP»: abbiamo dunque la prima occasione di evidenziare una differenza fondamentale tra il «contenuto» di una cella ed il suo indirizzo.

A tal proposito possiamo già dire che per caricare il *contenuto* di una cella di memoria in un registro, basta indicarne il nome: un esempio è l'istruzione «MOV AX, ALFA» che pone nel registro AX (a 16 bit) il contenuto della word (ancora 16 bit) posta all'indirizzo dato dall'etichetta ALFA.

Viceversa per caricare in un registro l'offset di una certa locazione di memoria, al nome dell'etichetta bisogna anteporre il prefisso speciale **«OFFSET»**: ad esempio possiamo vedere l'istruzione già menzionata di caricamento dello Stack Pointer «MOV SP, OFFSET TOP».

Altra possibilità, che analizzeremo in dettaglio nel prosieguo, è quella di caricare in un registro il segmento al quale una certa variabile appartiene e ciò si ottiene premettendo al nome dell'etichetta il prefisso «SEG»: un esempio è l'istruzione «MOV AX, SEG ALFA», che carica in AX il valore del segmento a cui appartiene ALFA.

Figura 1b




```
        NAME PROVA

PROGRAM SEGMENT
        ASSUME CS:PROGRAM,DS:DATI
        ASSUME ES:NOTHING,SS:PROGRAM

        MOV AX,PROGRAM
        MOV SS,AX
        MOV AX,DATI
        MOV DS,AX
        MOV SP,OFFSET TOP

        MOV AX,ALFA
        ADD AX,BETA
        CALL DIVIDI
        MOV GAMMA,AL
        MOV DELTA,AH

        RET

DIVIDI: SHR AX,1
        RET

STACK   DW 20 DUP (?)
TOP     LABEL NEAR

PROGRAM ENDS

DATI    SEGMENT

ALFA    DW ?
BETA    DW ?
GAMMA   DB ?
DELTA   DB ?

DATI    ENDS

        END
```

*Listato 2 - Esempio di programma relativo alla situazione di figura 1b.*

```
        NAME PROVA

DATI1   SEGMENT

VAL1    DB ?
VAL2    DB ?
VAL3    DB ?

DATI1   ENDS

DATI2   SEGMENT

VAL4    DB ?
VAL5    DB ?
VAL6    DB ?

DATI2   ENDS

DATI3   SEGMENT

VAL7    DB ?
VAL8    DB ?
VAL9    DB ?

DATI3   ENDS


PROGRAM SEGMENT
        ASSUME CS:PROGRAM,DS:DATI1
        ASSUME ES:NOTHING,SS:NOTHING

        MOV AX,DATI1
        MOV DS,AX

        MOV AL,VAL1
        ADD AL,VAL2
        MOV VAL3,AL

        ASSUME DS:DATI2
        MOV AX,DATI2
        MOV DS,AX

        MOV AL,VAL4
        ADD AL,VAL5
        MOV VAL6,AL

        ASSUME DS:DATI3
        MOV AX,DATI3
        MOV DS,AX

        MOV AL,VAL7
        ADD AL,VAL8
        MOV VAL9,AL

        RET

PROGRAM ENDS

        END
```

*Listato 3 - Esempio di programma relativo alla situazione di figura 2a.*

## Analisi della figura 1b

È questo un caso più generale in cui almeno la zona dati è separata nettamente (dal punto di vista logico) dalla zona di programma vera e propria.

In questa situazione, che si può verificare osservando il listato numero 2, l'unica differenza rispetto alla precedente è che il segmento di dati («DATI») è separato dal segmento «PROGRAM» e perciò l'inizializzazione del registro DS ne deve tenere conto: inutile dire che, in entrambe le situazioni, il programma effettua le stesse operazioni sugli stessi operandi, ottenendo perciò i medesimi risultati.

Per inciso segnaliamo il fatto che è auspicabile porre in testa al programma la direttiva «NAME» seguita dal nome del programma stesso: in tal modo si ottie-

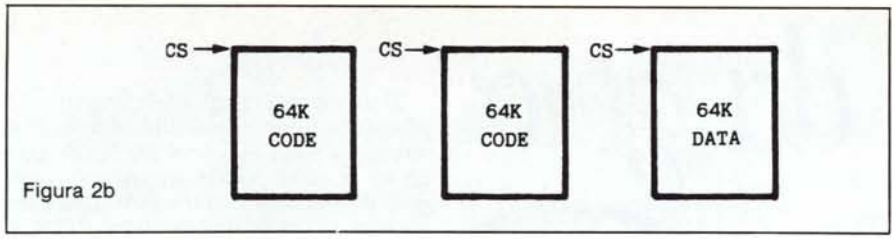


Figura 2b

ne, all'atto della compilazione del programma, come intestazione di ogni pagina di listato il nome indicato nella NAME. In mancanza invece di tale comando, l'assemblatore considererà il «modulo» di programma come «ANONYMOUS» etichettando in tal maniera (bruttina...) tutte le pagine di cui si compone il listato oggetto della compilata.

Invece una dimenticanza che comporta l'arresto dell'assemblatore è quella del comando «END»: terminando invece il proprio modulo con tale direttiva, allora si comunica all'assemblatore il termine logico e fisico del proprio programma.

## Analisi della figura 2a

Coloro i quali erano dell'opinione che 64 kbyte sono pochi per un segmento di dati o di programma (tanto per fare un esempio), troveranno un metodo generale per allargare oltre ogni limite la zona di memoria su cui il nostro programma agisce.

Ecco che perciò, facendo riferimento al listato numero 3, abbiamo creato ben tre segmenti di dati (contenenti nell'esempio appena tre byte!) che possono contenere invece ognuno 64 kbyte di memoria, per un totale di 192 kbyte di dati direttamente raggiungibili dal nostro programma.

Il trucco, se così si può chiamare, dal momento che invece è un'operazione più che lecita, consiste nel cambiare dinamicamente nel corso del programma il contenuto del segment register DS, comunicando volta per volta con la direttiva ASSUME ognuno di questi cambiamenti. Tra l'altro questa è la maniera più pratica di indirizzare locazioni di memoria appartenenti ad un certo segmento, solo se si è più sicuri che da un certo punto in poi del programma si farà riferimento solo a «quel» dato segmento: invece se ora si fa riferimento ad una variabile del primo segmento e subito dopo ad una di un altro segmento seguita ancora da un'altra variabile posta in un terzo segmento, ecco che diventa oneroso tutte le volte dover cambiare il valore del segment register DS, con il rischio di dimenticare qualche passaggio ed ottenere perciò che una data etichetta non è «raggiungibile» con l'attuale valore del DS.

A questo scopo, ma ritorneremo in dettaglio nel seguito sull'argomento, esiste una particolare tecnica detta del «Segment Override», consistente nel forzare un certo valore al posto del contenuto del DS, per ottenere così il «segment address» di una certa variabile di cui si conosce l'offset, sotto forma simbolica data dalla sua etichetta. Tornando ai tre segmenti di dati, noi abbiamo solo per semplicità indicato tre variabili da un byte l'una per ogni segmento, mentre nella realtà possono essere in numero e tipo completamente differenti: inoltre il numero «tre» di segmenti può essere aumentato a piacere dal programmatore, tenendo conto dell'effettiva quantità di memoria disponibile, del numero di byte o word effettivamente necessari per le operazioni e non da ultimo del fatto di poter eventualmente cercare un'ottimizzazione riducendo così il «consumo» di memoria.

## Analisi della figura 2b

Per analizzare meglio la figura in esame, facciamo riferimento al listato numero 4, nel quale troviamo appunto rappresentati sotto forma di semplici «frammenti» di programma i blocchi che compongono la figura 2b.

In questa situazione, invece di avere una «moltiplicazione» di segmenti di dati, si ha una certa proliferazione di Code Segment: la prima differenza che si nota rispetto ai multipli Data Segment è che in questo caso il registro CS non può essere caricato direttamente con una MOV (pensandoci bene si vede il perché ciò è oltremodo ovvio).

Ciò si ottiene invece con quella che in termini tecnici propri dell'assembler 8086/8088 viene detta una «long jump» o «intersegment jump» contrapposta alla più comune «short jump» o «intrasegment jump». Anche se come al solito ritorneremo più in dettaglio nel seguito, accenniamo ora al significato di questi termini, utilizzando le conoscenze sin qui acquisite.

Ricordiamo dunque (fino alla noia...) che un segmento è lungo 64 kbyte e su di esso «regna» il CS, nel caso, come quello in esame, in cui parliamo di segmento di programma: ora, tutto quanto avviene all'interno del segmento in esame e perciò nell'ambito dei suoi 64 kbyte in termini di salti o chiamate a subroutine assume l'attributo di «corto» («short») in quanto, sottintendendo che il CS rimane lo stesso, consente di ottenere una codifica più breve come computo di byte.

Ad esempio un salto ad un'etichetta posta all'interno del segmento in cui ci troviamo un attimo prima del salto stesso, viene codificata (con 3 byte) senza dover alterare il valore del CS attuale.

Invece se l'etichetta alla quale dobbiamo saltare (a seguito di una JMP o di una CALL) appartiene ad un altro segmento, allora dobbiamo essere ben consci del fatto che il nuovo segmento avrà come CS un valore diverso da quello di partenza: allora all'atto del salto il microprocessore deve essere a conoscenza di questo cambiamento da effettuare al valore corrente del CS.

```
        NAME PROVA
DATI    SEGMENT
DATI    ENDS

PROG1   SEGMENT
        ASSUME CS:PROG1,DS:DATI
        ASSUME ES:NOTHING,SS:NOTHING

        MOV AX,DATI
        MOV DS,AX

        JMP FAR PTR LAB2

PROG1   ENDS

PROG2   SEGMENT
        ASSUME CS:PROG2

LAB2    LABEL FAR
        MOV AL,BH

        JMP FAR PTR LAB3

PROG2   ENDS

PROG3   SEGMENT
        ASSUME CS:PROG3

LAB3    LABEL FAR
        ADD AX,AX

        RET

PROG3   ENDS

        END
```

*Listato 4 - Esempio di programma relativo alla situazione di figura 2b.*

Perciò, invece di ricorrere ad una fantomatica istruzione di MOV all'interno del registro CS, ecco che l'assemblatore stesso costruisce una codifica «lunga» («long») dell'istruzione di salto stessa inglobando in essa l'informazione del nuovo valore del CS, con 5 byte invece di 3 del caso «corto».

A sua volta, per aiutare l'assemblatore nel suo complicato lavoro, il programmatore deve indicare se il salto è all'interno del segmento oppure no: nel nostro caso di salti lunghi, «fuori» dal segmento di partenza, bisogna porre tra il termine JMP e l'etichetta di arrivo i due operatori «FAR» e «PTR».

Mentre l'interpretazione del termine «far» (corrispondente a «lontano») è abbastanza ovvia, come pure è abbastanza comprensibile l'attributo di «LABEL FAR» all'etichetta dove dobbiamo saltare, risulta invece ancora troppo presto analizzare il significato del termine «PTR», che verrà usato in moltissime operazioni, tra le quali gli indirizzamenti indiretti.

Con questo terminiamo la presente puntata e diamo l'appuntamento alla prossima nella quale cominceremo ad entrare nel vivo dell'assembler, analizzando via via tutte le sue sfaccettature.

MC