1 Segmentazione degli indirizzi di memoria nell'8086

Nel capitolo sull'architettura dell'8086 abbiamo detto dell'esistenza di registri di segmento, che servono ad estendere fino a 20 bit il parallelismo dell'address bus, utilizzando due registri da 16 bit sommati, dopo che uno dei due (il "segmento") è stato moltiplicato per 16.
Vediamo ora come si opera con i registri di segmento e cosa ci mette a disposizione l'Assembly per la loro gestione.

1.1 Calcolo dell'indirizzo a 20 bit

Ogni indirizzo che viene emesso sull'Address Bus di un 8086 è a 20 bit. Come già abbiamo visto il parallelismo interno della CPU è di 16 bit, per cui la gestione diretta di registri a 20 bit sarebbe stata scomoda e difficile.
Per questo il costruttore decise di far generare automaticamente alla CPU gli indirizzi da 20 bit, utilizzando due registri da 16 bit e realizzando l'accesso alla memoria con il (famigerato!) meccanismo della segmentazione.
Questo modo di calcolare l'indirizzo prevede la divisione dell'indirizzo in due "parti", parzialmente sovrapposte, cui si dà il nome di **segmento** e **offset**.
Per la parte di segmento la CPU utilizza sempre uno degli indirizzi di segmento, già brevemente introdotti nel capitolo che riguarda l'architettura dell'8086.
La parte di offset viene presa da un altro registro, oppure calcolata come somma di tre elementi: il contenuto di un registro base, di un registro indice e di un numero fisso detto "displacement".
Con segmento e offset la CPU esegue una strana "somma", che produce l'indirizzo a 20 bit usato sul bus.
La "somma" viene effettuata, ogni volta che si deve accedere ad una locazione di memoria, da uno speciale sommatore di indirizzi, situato nella BIU (Bue Interface Unit) della CPU (vedi Figura 1).
Questo sommatore ottiene un indirizzo di 20 bit a partire da due quantità di 16 bit con il seguente procedimento:
1) aggiunge 4 bit nulli a destra registro di segmento che utilizza, ottenendo un numero da 20 bit
2) somma con l'offset il numero a 20 bit ottenuto dal segmento
3) emette sull'Address bus il numero a 20 bit ottenuto dalla somma (i numeri come in Figura 1).

L'aggiunta dei 4 bit al registro di segmento, che equivale a moltiplicarlo per 16, non permette di raggiungere qualsiasi locazione di memoria, ma solo quelle multiple di 16. Per questo c'è bisogno anche dell'offset per utilizzare una locazione qualsiasi della memoria.

<FILE>
SommaSegm.FH5
</FILE>

Figura 1: somma per la segmentazione 8086

Vediamo, in binario ed in decimale, la stessa somma per segmentazione che appare in Figura 1, fatta sulla carta:

0	0	0	1	0	1	1	0	0	0	0	0	0	0	1	1	**0**	**0**	**0**	**0**	Segm + 0000
																				+
				0	0	0	0	1	0	1	0	0	0	1	0	0	1	0	0	offset
0	0	0	1	0	1	1	0	1	0	1	0	0	1	0	1	0	1	0	0	= ind. a 20 bit
			1				6				0				3				**0**	Segm + 0000
																				+
							0				A				2				4	Offset
			1				6				A				5				4	= ind. a 20 bit

Figura 2: segmento "più" offset

1.2 Segmento e offset

Il meccanismo della somma per la segmentazione ha conseguenze dirette su come il programmatore deve usare gli indirizzi di un 8086.
La memoria di un 8086 è accessibile in due modi: entro il "segmento" e fuori dal segmento.

Nella programmazione ordinaria i registri di segmento saranno sistemati una volta per tutte, poi ci si muoverà in memoria modificando il solo offset.

Il termine offset viene usato in molti contesti della tecnica (per esempio in Elettronica) con il significato di "differenza rispetto ad un valore di riferimento.
<EVIDENZIATO>
L'offset nella segmentazione esprime la distanza della locazione cui vogliamo accedere rispetto all'inizio del segmento.
</EVIDENZIATO>
In questo caso è un termine ben scelto, perché l'inizio del segmento è il "valore di riferimento " e ci si sposta utilizzando l'offset.
<EVIDENZIATO>
L'inizio del segmento è segnato dal registro di segmento.
</EVIDENZIATO>
Ogni registro di segmento punta ad una specifica posizione di memoria, che ha un indirizzo multiplo di 16, dato che ha quattro bit nulli a destra. Questo indirizzo è l'inizio del segmento.
<EVIDENZIATO>
Un segmento è l'area di memoria raggiungibile modificando solo l'offset. Dato che l'offset ha 16 bit un segmento dell'8086 può avere una dimensione massima di 64 kByte.
</EVIDENZIATO>
Una volta "fissato" l'indirizzo d'inizio del segmento, ci si può muovere all'interno di esso modificando il solo offset. Per andare fuori dal segmento corrente e raggiungere ogni possibile locazione di memoria è necessario modificare il valore del registro di segmento.
Un segmento può avere una dimensione qualsiasi compresa fra 0 e FFFFh (64kByte).
I segmenti possono anche coincidere: registri di segmento diversi possono puntare allo stesso indirizzo.
La Figura 3 mostra graficamente il registro di segmento che punta all'inizio del segmento, un indirizzo raggiunto con uno specifico offset e l'ultimo indirizzo raggiungibile del segmento.
<FILE>
SegOFFSET.FH5
</FILE>

Figura 3: segmento e offset

1.2.1 Notazione per la segmentazione

Nella notazione inventata dal produttore l'operazione di segmentazione viene indicata con il segno ":". Perciò se, trattando di indirizzi di memoria, si vedono due numeri separati da un duepunti, allora quello scritto a sinistra è il segmento e quello a destra l'offset. Le posizioni sono scelte in modo logico, dato che il segmento contribuisce più per la parte alta del numero a 20 bit, mentre l'offset fornisce tutti i 4 bit meno significativi.

Esempio:
1234h:1234h = 13574h

Chiameremo "indirizzo segmentato" quello composto dalle sue due parti di 16 bit, segmento e offset, mentre chiameremo "indirizzo fisico" l'indirizzo a 20 bit che viene effettivamente scritto sull'address bus. Alcuni chiamano "indirizzo logico" l'indirizzo segmentato, per quanto esso sia un termine un po' inflazionato, dato che viene usato anche in altri contesti con significato diverso.

<FILE>
IndSegm.FH5
</FILE>

Figura 4: indirizzo segmentato ed indirizzo fisico

1.2.2 Pericoli e limitazioni della segmentazione 8086

Il principale svantaggio della segmentazione 8086 è il fatto che lo stesso indirizzo fisico può essere originato in molti modi diversi.

<FILE>
Molti.FH5
</FILE>

Figura 5: molti indirizzi segmentati per lo stesso indirizzo fisico

Analizzando attentamente la Figura 5 ci si rende conto che tutti gli indirizzi segmentati che vi sono indicati puntano allo stesso indirizzo fisico. C'è anche da dire che in verità quelle illustrate sono pochissime delle combinazioni segmento – offset che puntano allo stesso indirizzo.
Un altro pericolo latente nell'uso della segmentazione 8086 è che lo stesso offset può corrispondere ad indirizzi diversi, come illustriamo nel prossimo paragrafo.

Dato che i registri di segmento non sono nella EU, ma nella BIU, non hanno a disposizione la ALU. Pertanto sui registri di segmento non si possono svolgere operazione aritmetiche e neppure trasferire numeri in immediato. L'unica cosa che si può fare è copiare al loro interno il contenuto di un registro generale o leggerlo dalla memoria.
Esempi:

```
<CODICE>
    MOV DS, AX    ; GIUSTA: si può copiare in DS il contenuto di un registro generale
    MOV ES, [BP + 6]    ; GIUSTA: si può leggere dalla memoria
    ADD DS, 6       ; vietata l'operazione aritmetica
    MOV SS, 28      ; vietato il trasferimento in immediato
</CODICE>
```

1.2.3 Uso dei registri di segmento da parte della CPU

<EVIDENZIATO>
Tutte le operazioni relative alla segmentazione sono svolte automaticamente dalla CPU; il programmatore non è coinvolto in alcun modo nella "somma" fra segmento ed offset.
</EVIDENZIATO>
Inoltre egli è coinvolto nella scelta del registro di segmento da utilizzare nella "somma" solo in casi estremamente rari (vedere "override", nel prosieguo). Dunque nella stragrande maggioranza dei casi la CPU sceglierà automaticamente anche il registro di segmento da utilizzare nella "somma", con i criteri che andiamo ad esporre.

La CPU sceglie quale registro di segmento usare in base a ciò che sta facendo in quel momento. Il nome dei registri di segmento ci dice in che occasione essi vengono usati.

CS
CS (**C**ode **S**egment) deve puntare al codice, infatti contiene l'indirizzo del segmento di codice, cioè dell'area di memoria in cui risiede il programma in esecuzione. Perciò la CPU deve usare CS quando è in fase di fetch e deve recuperare il codice operativo della prossima istruzione da eseguire.
L'offset che la CPU utilizza sempre con CS è IP. Questo chiarisce l'affermazione fatta precedentemente che CS:IP è il Program Counter dell'8086. Infatti l'indirizzo segmentato CS:IP punta sempre alla prossima istruzione da eseguire, come ogni bravo Program Counter. Ad ogni fase di fetch la CPU aggiorna automaticamente IP in modo che CS:IP punti all'istruzione seguente.

DS
DS (**D**ata **S**egment) deve puntare ai dati: contiene il puntatore alla prima locazione del segmento in cui si memorizzano i dati. Tutte le istruzioni "normali" usano automaticamente il registro DS quando devono accedere alla memoria. Per esempio l'istruzione:

```
<CODICE>
    ADD [variabile + BX + SI + 3], 6
</CODICE>
```

utilizza DS come registro di segmento. Come offset utilizza ciò che c'è fra le parentesi quadre; in questo caso la CPU aggiunge, a tempo di esecuzione:
- il contenuto del registro base BX nel momento in cui l'istruzione viene eseguita
- il contenuto del registro base BX nel momento in cui l'istruzione viene eseguita
- il valore fisso del displacement, calcolato dal compilatore una volta per tutte a tempo di compilazione, costituito dall'indirizzo nella tabella dei simboli del simbolo "variabile" cui è aggiunto 3.

Per cui l'indirizzo segmentato utilizzato è DS:(BX + SI + (variabile + 3)).

SS
SS (**S**tack **S**egment) punta all'inizio del segmento di stack. Sullo stack torneremo più avanti, qui si può dire che, siccome le istruzioni che usano lo stack sono solo due (PUSH e POP), la CPU usa automaticamente il registro SS quando esegue quelle due istruzioni.
La parte di offset per le operazioni sullo stack è fornita da SP, per cui quando la CPU riconosce il codice operativo di un'istruzione quale una PUSH od una POP opera all'indirizzo segmentato SS:SP.

Uso di BP
L'altro caso in cui l'uso di SS è automatico è quando nel calcolo della parte di offset è coinvolto il registro BP. Infatti quando si usa BP viene utilizzato il registro di segmento SS.
Vediamo un esempio di codice che lascia un po' perplessi:

```
<CODICE>
    MOV BP, BX
    MOV AX, [BX]
    MOV CX, [BP]
    ; conclusione sbagliata: AX è sempre uguale a CX
    ; (BX = BP => l'indirizzo è uguale)
```

```
    ; NO! Lavorano su segmenti diversi => l'indirizzo NON è uguale
</CODICE>
```

La vista di questo codice ci fa pensare di essere di fronte ad un modo strano per copiare il contenuto della stessa locazione di memoria nei due registri AX e CX. A prima vista infatti l'indirizzamento è uguale.
Ma in verità l'unica cosa che è uguale è l'offset dei due indirizzi. Perché l'indirizzo fisico fosse uguale dovrebbero esserlo anche le due parti di segmento.
Ma ciò non è assolutamente detto, dato che MOV AX, [BX] usa il segmento DS (indirizzo segmentato DS:BX), mentre MOV CX, [BP] usa SS (indirizzo SS:BP).
La seguente figura illustra un esempio che chiarisce ulteriormente quanto detto:

<FILE>
DueLoc.FH5
</FILE>

Figura 6: due locazioni diverse con lo stesso offset

Con i valori indicati in figura e le linee di codice precedenti, AX e CX non sarebbero uguali, ma AX varrebbe 20, mentre CX 1000.
Questo esempio ci fa vedere un'altra stranezza dello schema di segmentazione scelto per l'8086. Due accessi alla memoria che sembrano puntare allo stesso indirizzo in verità indicano indirizzi del tutto diversi.
Questo può portare facilmente ad errori. La segmentazione è forse la più importante causa di problemi di affidabilità nei programmi per MS DOS. Al giorno d'oggi i programmi scritti per i processori a 32 bit della famiglia X86, che vanno dal 386 in poi, possono usare uno schema di accesso alla memoria che fa di fatto a meno della segmentazione (vedere nel prossimo volume il "modello piatto" della programmazione in modo protetto).

ES

ES (**E**xtra **S**egment) è un registro di segmento a disposizione del programmatore. Esso viene usato automaticamente dalla CPU solo durante l'esecuzione di poche istruzioni di stringa, molto particolari. Le istruzioni "normali" usano ES solo se lo si indica esplicitamente tramite un'operazione che viene detta "override" (vedi prossimo paragrafo).

Nelle CPU dal 386 in poi sono inclusi anche due nuovi registri di segmento: FS e GS, che sono "extra", usati solo in override.

1.2.4 Override

La parola "ovveride" o "forzatura di segmento", indica lo "scavalcamento" (over – ride, appunto) del segmento che la CPU usa automaticamente (per "default").
La sintassi dell'Assembly 8086 relativa all'override è semplice: basta anteporre alla parentesi quadra il nome del registro di segmento che si vuole usare al posto di quello di default e separare il nome dalla parentesi quadra con un ":".

```
<SINTASSI>
    <IstruzioneConOverride> := <CodiceMnemonico> <RegistroDiSegmento> : _
                               <OperandoInMemoria> [, <operando>]
</SINTASSI>
```

Alcuni esempi:

```
<CODICE>
    MOV AX, ES:[BX]  ; l'indirizzo segmentato che viene utilizzato è
                     ; ES:BX, invece che DS:BX
    ADD BYTE PTR DS:[BP], 10 ; somma all'indirizzo DS:BP, invece che in SS:BP,
                             ; che è il default quando si usa BP fra parentesi quadre
    MUL CS:[Primo]   ; usa l'indirizzo CS: OFFSET di Primo
                     ; si noti che facendo l'override si possono memorizzare
                     ; dati anche nel segmento di codice
    MOV WORD PTR DS:[SI + 1], 0   ; questo override è inutile, perché l'indirizzo sarebbe
                                  ; stato comunque DS:(SI + 1)
</CODICE>
```

Esiste anche un'altra sintassi per l'override, che spieghiamo con un esempio:

```
<CODICE>
    MOV AX, [ES:BX]
</CODICE>
```

che viene compilato esattamente come il primo degli esempi precedenti.
La preferenza dell'autore va alla prima sintassi illustrata, perché usandola si lasciano dentro le parentesi quadre entità relative al solo offset.

L'override viene realizzato in linguaggio macchina anteponendo al codice operativo dell'istruzione normale un prefisso di un byte che indica, oltre il fatto che la CPU deve fare l'override, anche quale registro di segmento deve essere usato al posto di quello di default. Il compilatore si preoccupa di inserire a tempo di compilazione il prefisso in questione nel codice oggetto.

Vediamo in linguaggio macchina come funziona l'override, analizzando alcuni esempi:
<CODICE>
```
Sorgente                          Linguaggio macchina
MOV word ptr [2], 1               C706 0200 0100
MOV word ptr DS:[2], 1            C706 0200 0100
MOV word ptr ES:[2], 1            26 C706 0200 0100
MOV word ptr SS:[2], 1            36 C706 0200 0100
MOV word ptr CS:[2], 1            2E C706 0200 0100
MOV word ptr DS:[2 + BP],         3E C786 0200 0100
```
</CODICE>
Anche l'override del secondo di questi esempi è "inutile", però il compilatore che ha generato il relativo linguaggio macchina se ne è reso conto e non ha anteposto il prefisso di override. Negli altri casi invece c'è sempre un prefisso di override.

<EVIDENZIATO>
Schema di riepilogo sull'uso dei registri di segmento:

Codice	CS	usato nelle fasi di fetch
Dati	DS	usato nelle fasi di execute delle istruzioni "normali"
Stack	SS	usato nelle operazioni nello stack e quando si usa [BP]
Extra	ES	usato come segmento di riserva in override e in alcune istruzioni di stringa
Extra	FS	(386>) usato solo con override
Extra	GS	(386>) usato solo con override

</EVIDENZIATO>

1.2.5 Jump NEAR e FAR

Siccome il Program Counter di un 8086 è costituito da due registri, CS e IP, si possono individuare due tipi di jump incondizionate: quelle che modificano solo IP e quelle che modificano sia CS che IP.
Siccome le prime rimangono all'interno del segmento, esse vengono dette **jump NEAR** (vicine).
Quelle invece che, modificando anche CS, possono raggiungere ogni punto dello spazio di indirizzi della CPU vengono dette **jump FAR** (lontane).

Per indicare all'Assembler quale tipo di jump si vuole fare si usa il costrutto PTR, nella sua forma NEAR o FAR:
<SINTASSI>
```
    JMP [NEAR PTR] <label>
    ; jump near pointer
    ; IP <- OFFSET <label>

    JMP [FAR PTR] <label>
    ; jump far pointer
    ; IP <- OFFSET <label>
    ; CS <- SEG <label>
```
</SINTASSI>
Nella JMP NEAR se <label> è effettivamente al di fuori del segmento, il compilatore non può far altro che fermarsi e non concludere la compilazione.
Nella JMP FAR se <label> è in effetti all'interno del segmento cui deve saltare viene riscritto anche CS, che però rimane uguale.
Se non si indica né NEAR né FAR la jump viene fatta NEAR.

Esempio:
<CODICE>
```
    JMP
    JMP NEAR PTR AX  ; Come si può vedere dalla tabella delle istruzioni,
                     ; in Appendice, è possibile caricare l'indirizzo (near)
                     ; da un registro da 16 bit
    JMP FAR PTR [IndirizzoFAR]
```
</CODICE>
Nella Figura 1 sono illustrate due jump. La JMP NEAR salta indietro di 19051 locazioni, ma rimane dentro al segmento attuale (in CS c'è A326h), per cui può essere NEAR.
La JMP FAR salta in avanti di 2 (due!) locazioni ma deve essere FAR, perché è il punto di arrivo è fuori dal segmento e per raggiungerlo si deve cambiare il valore di CS (che diviene B326, come mostrato dall'operazione di somma per la segmentazione)
<FILE>
JMPnearFAR.FH5
</FILE>

Figura 7: salti NEAR e FAR

1.3 Direttive di segmentazione

Ogni Assembler per 8086 ha direttive diverse per permettere al programmatore di dichiarare come deve essere utilizzata la memoria e come deve essere divisa in segmenti.
L'Assembler MASM 8086 ha direttive che permettono il controllo completo dell'uso dei segmenti (direttive standard).
L'Assembler TASM ha introdotto direttive di segmentazione "semplificate", che sono più corte da scrivere ma danno minore controllo su ciò che si fa sui segmenti. TASM può funzionare anche con direttive standard.

1.3.1 Direttive standard

SEGMENT
La direttiva standard SEGMENT serve per iniziare una zona di programma sorgente che verrà successivamente associata ad un registro di segmento.
Il segmento deve essere sempre concluso da una direttiva ENDS, prima che ne inizi uno nuovo con un'altra direttiva SEGMENT.

```
<SINTASSI>
    <NomeSegmento> SEGMENT [<allineamento>] [<combine>] ['<classe>']
</SINTASSI>
```

<nome segmento> è un'etichetta il cui simbolo può essere assegnato a piacere. L'etichetta sarà associata a quel segmento e deve essere utilizzata per la successiva ENDS.

<allineamento>, <combine>, e <classe> sono facoltativi. Il compilatore assegna ad essi valori di default che è necessario modificare sono con applicazioni sofisticate .

Esempi:

```
<CODICE>
Dati SEGMENT ; di solito non si mettono opzioni
    .. qui ci sono dati o istruzioni
Dati ENDS ; conclusione di Dati prima che inizi Code

Code SEGMENT BYTE COMMON 'CODE'

Code ENDS

Catasta SEGMENT PARA STACK 'STACK'
    .. qui ci sono dati o istruzioni
Catasta ENDS
</CODICE>
```

<AVANZATO>
Un programma può avere più di un'area di memoria SEGMENT con lo stesso nome. L'Assembler tratta tutte le zone del programma che hanno lo stesso <NomeSegmento> come uno stesso segmento, accodandole in memoria.

L'allineamento dice al compilatore quanti byte dopo la fine del segmento precedente far cominciare il segmento di cui si parla.
Gli allineamenti possibili sono:

BYTE	il compilatore allinea i segmenti in modo da utilizzare il primo indirizzo disponibile, dopo la fine del segmento precedente.
WORD	utilizza il primo indirizzo multiplo di 2 disponibile
DWORD	utilizza il primo indirizzo multiplo di 4 disponibile
PARA	utilizza il primo indirizzo multiplo di 16 disponibile (paragrafo)
PAGE	utilizza il primo indirizzo multiplo di 256 disponibile (pagina)

Per default viene utilizzato PARA.

La combine dice al compilatore come far combinare insieme segmenti dello stesso nome e della stessa classe.
Le possibili combine sono:

AT	fa in modo che l'inizio del segmento venga posto ad uno specifico indirizzo di memoria
COMMON	fa in modo che i segmenti in questione inizino alla stessa locazione, per cui si essi si sovrappongono
PUBLIC	i segmenti in questione vengono concatenati, uno di seguito all'altro, e diventano uno solo
PRIVATE	i segmenti in questione non vengono combinati con altri segmenti
STACK	come PUBLIC, ma per segmenti usati come stack (linker e loader fanno in modo che SS:SP sia puntato al posto giusto)

L'uso della combine è di particolare interesse nel caso di programmi a più moduli separatamente compilati.

La classe serve a far porre tutti i segmenti della stessa classe in zone di memoria contigue, indipendentemente dal loro ordine nel programma sorgente. Esempi di classi: 'STACK', 'CODE', 'DATA'.
</AVANZATO>

ENDS
La direttiva standard ENDS serve per terminare la zona di programma che rappresenta un segmento.
Come già detto SEGMENT e ENDS non si possono nidificare.
<SINTASSI>
```
    <nome segmento> ENDS
```
</SINTASSI>
Sono già stati dati degli esempi nella trattazione della direttiva SEGMENT.

ASSUME
La direttiva ASSUME comunica al compilatore a quale segmento punta ogni registro di segmento.
<SINTASSI>
```
    ASSUME <Nome di un registro di segmento>: <etichetta di un segmento>, ..
```
</SINTASSI>

Esempio:
```
<CODICE>
ASSUME CS:Code, DS:Dati, SS:Catasta
</CODICE>
```

Essa è una direttiva "pura", non fa eseguire nessuna istruzione di macchina e serve solo al compilatore per poter calcolare gli offset degli indirizzi in memoria cui si fa riferimento
Dichiarando a quale SEGMENT si vuole associare DS il compilatore calcolerà gli offset delle locazioni da usare nelle MOV con riferimento a quel SEGMENT.
ASSUME non "fa" niente, nel senso che non cambia automaticamente nessuno dei registri di segmento. Se un'ASSUME indica il cambiamento di uno dei registri di segmento, il programmatore dovrà comunque preoccuparsi di caricare, con delle MOV, il nuovo valore nel relativo registro di segmento.
Una ASSUME fatta dopo un'altra ridefinisce i registri di segmento di cui tratta, mentre quelli di cui non tratta rimangono uguali a prima.
Senza almeno una ASSUME del CS un programma non può essere compilato.

OFFSET
Questa direttiva, cui abbiamo già accennato in termini più "ristretti" nel capitolo sull'accesso alla memoria, dice al compilatore di prendere il valore della parte di OFFSET dell'indirizzo associato a <label>.
<SINTASSI>
```
    OFFSET <label>
```
</SINTASSI>

Esempio:
```
<CODICE>
    MOV BX, OFFSET Vettore
    ; tipicamente l'offset si mette in un registro "puntatore" come BX,
    ; perché dovrà essere usato per puntare alla memoria
    ; (all'interno delle parentesi quadre)
</CODICE>
```

Il valore dell'offset cercato può essere determinato a tempo di compilazione. Il compilatore può calcolarlo solo se prima è stata fatta una ASSUME a DS del SEGMENT dove c'è <label>.

Come già visto nel capitolo dedicato all'accesso alla memoria esistono due modi per ottenere l'effetto di caricare in un registro l'offset di un indirizzo: oltre a OFFSET esiste anche l'istruzione LEA (vedi indice analitico).

SEG
<SINTASSI>
```
    SEG <label>
```
</SINTASSI>
Questa direttiva dice al compilatore di prendere il valore della parte di segmento dell'indirizzo associato a <label>.(*)
<NOTA>
<AVANZATO>
(*) A tempo d'esecuzione questa informazione viene messa dal loader nella parte iniziale della memoria occupata dal programma (che si chiama PSP (program segment prefix)). Il compilatore, a tempo di compilazione, può dare l'indicazione di leggere alla giusta posizione in quell'area di memoria.

</AVANZATO>
</NOTA>
Esempio:
```
<CODICE>
    MOV AX, SEG Data
</CODICE>
```

Il registro di destinazione più logico per una direttiva SEG dovrebbe essere un registro di segmento (CS, DS, SS, ES), ma, come abbiamo già accennato, non è possibile trasferire numeri in immediato nei registri di segmento. Per cui si deve prima trasferire SEG in un registro generale, dal quale verrà successivamente copiato nel registro di segmento voluto. Questa lunga frase di riassume nell'esempio seguente:
```
<CODICE>
    MOV SS, SEG StackLocale  ; non è possibile per le caratteristiche
                             ; hardware degli 80x86
    ; dato che non si può fare la precedente ci adattiamo e facciamo così:
    MOV AX, SEG StackLocale  ; la parte di segmento di quella etichetta
                             ; finisce in AX
    MOV SS, AX               ; per essere poi copiata in SS
</CODICE>
```

Esempio:
```
<CODICE>
    ; per visualizzare una stringa in MSDOS bisogna passare l'offset
    ; dell'indirizzo al servizio DOS numero 9:
    MOV DX, OFFEST Stringa
    ; in genere le prossime istruzioni non si devono mettere perché DS
    ; punta già al posto giusto. Se questo non è il caso si deve fare così:
    MOV AX, SEG Stringa
    MOV DS, AX
    MOV AH,09    ; servizio DOS "stampa stringa"
    INT 21h      ; esecuzione del servizio DOS
</CODICE>
```

Vediamo un elenco di risultati in un esempio con due segmenti. Non commenteremo questi risultati, lasciando al lettore l'onere di comprenderli:
```
<CODICE>
Suno SEGMENT
    A DB ?
    B DB ?
Suno ENDS
Sdue SEGMENT
    C DB ?
Sdue ENDS
</CODICE>
```
Si ha:
SEG Suno = SEG A = SEG B diverso SEG S2 = SEG C

OFFSET Suno = OFFSET A = OFFSET Sdue = OFFSET C = 0

OFFSET B = 2

Siamo ora in grado di spiegare pienamente le "magiche" istruzioni che abbiamo sempre scritto all'inizio di ogni programma in Assembly.

1.3.2 Direttive semplificate

La direttiva .MODEL è stata introdotta nel Turbo Assembler e sostituisce le direttive standard con notazioni più concise. .MODEL va usata insieme alle direttive: .DATA, che va messa all'inizio del segmento dei dati, .STACK, all'inizio del segmento di stack, .CODE, all'inizio del segmento di codice e END (senza punto), alla fine di tutto. Le varie aree del programma non vanno concluse con ENDS, basta che inizi una nuova area con una direttiva "punto". Non è necessario usare l'ASSUME.
Il compilatore fa automaticamente in modo che il programma esegua la sua prima istruzione all'indirizzo che corrisponde a .CODE.

```
<SINTASSI>
    MODEL <modello di segmentazione>
</SINTASSI>
```

Esempio:
```
<CODICE>
.MODEL HUGE
</CODICE>
```

I modelli di segmentazione disponibili in Turbo assembler sono:
TINY Codice NEAR, Dati NEAR, coincidenti (è il modello di memoria con il quale si possono fare i file .COM)
SMALL Codice NEAR, Dati NEAR, segmento di codice diverso dal segmento di dati
MEDIUM Codice FAR, Dati NEAR
COMPACT Codice NEAR, Dati FAR
LARGE Codice NEAR, Dati FAR, singoli array minori di 64k
HUGE Codice FAR, Dati FAR, singoli array comunque grandi

Come ci si aspetta il codice far è più lento di quello near, ed i dati far sono molto più difficili da gestire rispetto ai corrispondenti near. Il compilatore peraltro non è d'aiuto. Esso, conoscendo il modello di memoria, è in grado di controllare che il programma risponda alle caratteristiche indicate; l'onere di realizzare tutte le complicazioni richieste dai modelli di memoria far è ancora tutto del programmatore.

1.3.3 Accesso a locazioni assolute

Dato che l'accesso alla memoria dell'8086 è sempre "segmentato" si deve tener presente la segmentazione ogni volta che si vuole accedere a specifiche locazioni di memoria.
L'accesso a specifiche locazioni di memoria non è molto comune ma esistono parecchie sono locazioni fisse d'interesse cui si potrebbe accedere, come la tabella dei vettori d'interruzione (fra 00000h e 003FFh, vedi il prossimo volume), l'area di memoria del DOS (fra 00400h e 005FFh) o le aree della memoria video della scheda grafica (fra F4000h e FFFFFh)

Vediamo per esempio come si può fare per accedere al sedicesimo byte dell'area di memoria del DOS:
```
<CODICE>
    ..
    MOV AX, 40h
    MOV DS, AX
    MOV AL, [16d]    ; accesso a 0040h:0010h
    ..
</CODICE>
```

Questo brano di programma funziona, ma è codice oscuro e poco documentato. Un modo più chiaro di operare è questo:

```
<CODICE>
; all'inizio del programma diamo due definizioni:
SegmMemoriaDOS EQU 40h           ; ciò che voglio mettere nel registro di segmento
OffsLocazioneCheVoglio EQU 16d   ; ciò che voglio usare come offset
    ..
    MOV AX, SEG SegmMemoriaDOS
    MOV DS, AX
    MOV AX, [OffsLocazioneCheVoglio]
    ..
</CODICE>
```
Il risultato è lo stesso delle tre linee di codice viste in precedenza, ma è più autodocumentato.

Un modo un po' più sofisticato di raggiungere lo stesso scopo è questo:
```
<CODICE>
MemoriaDOS SEGMENT AT 40h
                ; ^ il compilatore sa che MemoriaDOS deve iniziare
                ; all'indirizzo 40h
    ORG 10h  ; il compilatore sposta il "location counter" di 10h
             ; per arrivare 16 byte oltre l'inizio della
             ; memoria DOS.
; definizione di un'etichetta per la locazione che devo usare (40h:10h):
LocazioneCheVoglio DW ?
MemoriaDOS   ENDS
</CODICE>
```
Il compilatore, quando trova un segmento di tipo AT, non alloca memoria, ma usa le definizioni del segmento per fare riferimento a posizioni fisse nella memoria.
Nel corso del programma si può utilizzare il segmento AT precedentemente definito in questo modo:
```
<CODICE>
ASSUME  DS: MemoriaDOS
    MOV AX, SEG MemoriaDOS
```

```
    MOV DS, AX
    MOV AX, [LocazioneCheVoglio]
</CODICE>
```
Il risultato è lo stesso dei due casi precedenti, ma è più sicuro, perché il compilatore, sapendo cosa si intende fare, può fare molti più controlli sulla correttezza del modo di procedere.

Un errore nell'Assembler
Il compilatore TASM accetta senza dare errore la seguente istruzione:
```
<CODICE>
    ADD byte ptr 10h:[0A0h], 3Ah ; compila ma è SBAGLIATA
</CODICE>
```
La compilazione di questa istruzione dà l'illusione che l'accesso alla memoria venga fatto all'indirizzo 10h:A0h. Questo non è vero perché se si va a vedere cosa il compilatore produce esso è uguale a ciò che viene prodotto da:
```
<CODICE>
    ADD byte ptr [0A0h], 3Ah
</CODICE>
```
L'indirizzo che viene effettivamente usato, anche nella prima istruzione, è DS:A0!

1.3.4 Caricamento di puntatori FAR

Per caricare un registro di segmento si può usare la direttiva SEG. Per caricare un intero indirizzo segmentato con una sola istruzione si usa quella fra LDS, LSS o LES (LFS, LGS per 386>), che carica la parte di segmento nel registro di segmento voluto.
Queste istruzioni caricano sia la parte di offset sia la parte di segmento dell'indirizzo caricandole dalla memoria.
```
<SINTASSI>
    LDS <Registro da 16 bit>, [<Puntatore alla memoria>]
    ; Load full pointer using DS
    ; funzionamento:
    ; MOV <Registro da 16 bit>, [<Puntatore alla memoria>]
    ; MOV AX, [<Puntatore alla memoria> + 2]
    ; MOV DS, AX
</SINTASSI>
```
<Registro da 16 bit> è tipicamente uno dei registri "puntatori" (BX, SI, SI ,DI).
<Puntatore alla memoria> è l'indirizzo al quale si deve leggere l'indirizzo da caricare
Dopo l'istruzione DS:<Registro da 16 bit> punta all'indirizzo segmentato che si trova nei quattro byte che partono da DS: [<puntatore alla memoria>]

L**SS** carica nello stack segment register (SS), per il resto è uguale a LDS;
L**ES** carica nello extra segment register (ES);
L**FS** carica nello F segment register (FS, 386>);
L**GS** carica nello G segment register (GS, 386>);

Esempi:
```
<CODICE>
    LDS SI,      [IndirizzoStringa1]  ; SI <- WORD PTR [IndirizzoStringa1]
                                      ; DS <- WORD PTR [IndirizzoStringa1 + 2]

    LES DI,      [IndirizzoStr2]      ; SI <- WORD PTR [IndirizzoStr2]
                                      ; DS <- WORD PTR [IndirizzoStr2 + 2]
    ; le due istruzioni precedenti sono ideali per preparare ad
    ; un'istruzione di stringa (vedi il capitolo "Altre istruzioni")

    LDS SI, DWORD PTR SS:[SI+12] ; si può fare l'override in lettura
    LSS AX, ES:[21h * 4]         ; si può caricare in AX, anche
                                 ; se non è molto utile ..
    LES AX, AX       ; NO! Non si può caricare da un registro
    LDS AX, 210      ; NO! Non si può in immediato
 </CODICE>
```

<AVANZATO>
Queste istruzioni sono state estese per funzionare anche nelle CPU a 32 bit della famiglia X86, anche se nella programmazione in "modo piatto" (vedi prossimo volume) sono usate molto raramente.
Le istruzioni di caricamento puntatori a 32 bit caricano l'offset a 32 bit tipico delle CPU X86 a partire dal 386.
</AVANZATO>

Esempio finale

```
<CODICE>
```

```
; NEAR_FAR.ASM
; Programma di esempio per TASM - MASM con chiamate, anche indirette,
; near e far. Il programma ha due segmenti di dati e due di codice.
; Il programma cambia semplicemente i valori di alcune variabili comuni, con
; procedure e valori messi in file diversi.

; ****************** un primo segmento dati **********************************
dati1   SEGMENT
 ; qui ci sono le normali direttive per l'allocazione della memoria

 indirizzofar DD ?   ; 32 bit per scriversi un indirizzo FAR, sul quale
                     ; fare una CALL FAR indiretta
 VarInDati1  DB ?    ; variabile nel segmento dati1
dati1   ENDS

; ****************** un secondo segmento dati ********************************
dati2   SEGMENT
 VarInDati2  DB ?    ; variabile nel segmento dati2
dati2   ENDS


; ******************* Inizia un primo segmento di codice **********************
; ******************* da cui il programma parte inizialmente

code1  SEGMENT
ASSUME CS:code1, DS:dati1, ES:dati2
; nel seguito useremo DS per puntare al segmento dati1 ed ES per dati2

P1ProcNEARinCode1 PROC NEAR
    ; procedura near scritta prima del codice del main, nello stesso segmento
    ; aumenta di 1 le variabili (Più 1)

    INC [VarInDati1] ; VarInDati1 è nel segmento dati1

    INC ES:[VarInDati2] ; VarInDati2 è nel segmento dati2
    RET                 ; il compilatore mette una RETN (near)
P1ProcNEARinCode1 ENDP

 ; ****** Main program ******
inizio:
    ; mettiamo in DS l'inizio del segmento dati1 (lo lasceremo lì!):
    MOV  AX, SEG dati1   ; il loader non carica DS, lo dobbiamo fare noi
    MOV  DS, AX

    ; mettiamo in ES l'inizio del segmento dati2 (lo lasceremo lì!):
    MOV  AX, SEG dati2
    MOV  ES, AX

    CALL P1ProcNEARinCode1       ; chiamata near, non necessita specificarlo
                                 ; perché si è usato PROC NEAR

    CALL P2ProcNEARinCode1dopo   ; chiamata near

    CALL FAR PTR P4ProcFARinCode1     ; chiamata far nello stesso segmento,
    ; è possibile e normale, se P4ProcFARinCode1
    ; viene chiamata anche da altri moduli

    CALL FAR PTR P8ProcFARinCode2     ; chiamata far in un altro segmento

         CALL NEAR PTR P16ProcNEARinCode2 ; una chiamata così non ha senso,
         ; perché è assurdo chiamare con chiamata near una procedura al di
         ; fuori del segmento codice corrente.
         ; In realtà, se si prova con TASM, esso fa delle assunzioni non
         ; richieste e si calcola un offset per arrivare lo stesso a
         ; P16ProcNEARinCode2 pur con una chiamata NEAR.
         ; Nell'opinone dell'Autore in questo caso TASM non dovrebbe
         ; compilare e dare un'indicazione di errore!

    ; SALTO INDIRETTO a P8ProcFARinCode2, uso del primo segmento di dati
    ; prepara in memoria l'indirizzo per una CALL
    MOV WORD PTR [indirizzofar], OFFSET P8ProcFARinCode2
    MOV WORD PTR [indirizzofar + 02h], SEG P8ProcFARinCode2
    ; chiamata indiretta, ad un indirizzo far che è in memoria:
    CALL [indirizzofar]
```

```
    MOV  AH, 4ch ; fine programma
    INT  21h

P2ProcNEARinCode1dopo PROC NEAR
    ; procedura near scritta dopo il codice del main, nello stesso
    ; segmento, aumenta di due le variabili comuni
    ADD [VarInDati1], 2 ; VarInDati1 è nel segmento dati1

    ADD ES:[VarInDati2], 2   ; VarInDati2 è nel segmento dati2
    RET                      ; il compilatore mette una RETN (near)
P2ProcNEARinCode1dopo ENDP

P4ProcFARinCode1 PROC FAR
    ; procedura far nello stesso segmento, aggiunge 4 alle variabili
    ; comuni:
    ADD [VarInDati1], 4
    ADD ES:[VarInDati2], 4
    RET                      ; il compilatore mette una RETF (far)
P4ProcFARinCode1 ENDP

code1  ENDS

; ************* Inizia un altro segmento di codice *************
code2  SEGMENT PARA
; !! il code segment per questa parte di programma deve essere CS =>
; !! qui ci vuole l'ASSUME del nuovo segmento:
ASSUME  CS:code2
P8ProcFARinCode2 PROC FAR
     ; procedura far in un altro segmento, aggiunge 8 alle variabili
    ADD [VarInDati1], 8
    ADD ES:[VarInDati2], 8
    RET
P8ProcFARinCode2 ENDP

P16ProcNEARinCode2 PROC NEAR
    ; procedura near nel segmento code2
    ; aggiunge 16 alle variabili comuni
    ADD [VarInDati1], 16
    ADD ES:[VarInDati2], 16
    RET
P16ProcNEARinCode2 ENDP
code2  ENDS

END inizio ; fine di tutto, si indica anche da dove partire con il codice

; si consiglia di eseguire questo programma nel debugger, passo a passo, tenendo d'occhio
; i valori nello stack, CS e IP.

</CODICE>
```

Problemi ed esercitazioni

Domande