1 Memoria e strutture dati

Nella prima parte di questo capitolo ci occupiamo di come definire ed utilizzare la memoria in Assembly; nella seconda parte vediamo come sono realizzate a basso livello le variabili e le strutture dati tipiche dei linguaggi di alto livello.
Come al solito l'occhio va all'8086, ma senza dimenticarsi che esistono anche altri processori nella "famiglia".

1.1 Inizio e fine di un programma

L'Assembly 8086 non ha una direttiva specifica per indicare dove il programma inizia, perché inizia semplicemente alla prima riga del file. In compenso esiste una direttiva che indicare il punto in cui tutto il codice finisce.
La direttiva che conclude un programma è END. La sua sintassi è la seguente:

```
    END [<EtichettaInizio>]
```

Il compilatore ignora tutto ciò che è scritto dopo una direttiva END.

Esempi:

```
    END
; oppure
    END Inizio
; o anche
    END Start
```

Scrivere END troppo presto, prima di una parte significativa di programma, "cancella" quella parte di programma, che per il compilatore non esiste!

<EtichettaInizio> è la label della prima istruzione del programma. Il compilatore fa in modo che quando il programma viene caricato e lanciato esegua la prima istruzione all'indirizzo che corrisponde a <EtichettaInizio>.

END non può mancare, pena l'impossibilità di compilare il programma.

Un esempio:

```
    .. dichiarazioni dei dati del programma ..
Inizio:  ; label della prima istruzione
    MOV AX, SEG DATA ; prima istruzione del programma (solo di esempio)
    MOV DS, AX
    .. resto del programma ..
    END Inizio   ; chiudendo il programma indico che l'esecuzione
                 ; deve iniziare da Inizio
```

Se <EtichettaInizio> non è indicata (come indicano le parentesi quadre essa è opzionale) il programma inizia da ciò che è scritto per primo. Naturalmente se ciò che è scritto per primo è una zona di memoria che contiene dati si va senz'altro verso il disastro!

Per evitare il disastro e cominciare comunque con un'istruzione si può mettere un salto che scavalchi tutta l'area dati, come viene illustrato nel seguente esempio:

```
    ; prima istruzione del programma:
    JMP PrimaIstruzioneInteressante
    ; questa istruzione fa un salto a PrimaIstruzioneInteressante
    ;(vedi oltre per i dettagli), in questo modo la prima istruzione
    ; che viene eseguita non è un dato!

    .. dichiarazioni dei dati del programma ..

PrimaIstruzioneInteressante:
    ; label della prima istruzione
    MOV AX, SEG DATA ; come prima
    .. resto del programma ..
    END ; nessuna etichetta di inizio
        ; => l'esecuzione parte dalla prima cosa che si scrive
```

1.2 Aree di memoria per dati e codice

L'Assembly 8086 prevede direttive che stabiliscono l'inizio e la fine di un'area di memoria, che può essere dedicata a contenere codice, dati od altro. Esiste anche una direttiva per concludere l'area di memoria iniziata.
La direttiva che inizia un'area di memoria è SEGMENT. La sua sintassi è la seguente:

```
    <NomeDelSegmento> SEGMENT
```

<NomeDelSegmento> è un identificatore che indica una locazione di memoria, cioè quella che abbiamo chiamato "etichetta". Può essere assegnato a piacimento, a patto naturalmente che non sia una parola chiave del linguaggio.
La direttiva SEGMENT è più complessa e flessibile di quanto descritto in queste righe, si descriverà in dettaglio nel seguito (capitolo "segmentazione"). Qui abbiamo detto quanto basta per cominciare a programmare.

La direttiva che conclude un'area di memoria è ENDS (**End S**egment). La sintassi:

```
    <NomeDelSegmento> ENDS
```

Un programma Assembly può essere composto di una o più aree di memoria.
Vediamo un esempio di definizione di un programma che ha un'unica area di memoria:

```
UnicaAreaDiMemoria SEGMENT
    .. zona riservata ai dati ..
ASSUME CS:UnicaAreaDiMemoria, DS:UnicaAreaDiMemoria
; ^ ASSUME associa i registri CS e DS a UnicaAreaDiMemoria
InizioCodice:
    .. zona riservata al codice ..
UnicaAreaDiMemoria ENDS
END InizioCodice ; NIENTE PUNTO (NON E' PASCAL)!
```

L'Assembler ha la necessità, per poter assegnare gli indirizzi delle etichette, di conoscere a quale registro di segmento sono associate le aree di memoria definite con SEGMENT .. ENDS. Questa associazione viene dichiarata con la direttiva ASSUME, il cui il significato verrà specificato meglio più avanti.
Vediamo un esempio con due aree di memoria, una riservata ai dati, l'altra alle istruzioni:

```
Dati SEGMENT
    .. zona riservata ai dati ..
Dati ENDS
Codice SEGMENT
ASSUME CS:Codice, DS:Dati
; ^ ASSUME associa il registro CS all'area "Codice" e DS all'area "Dati"
InizioCodice:
    .. zona riservata al codice ..
Codice ENDS
END In
```

Direttive di segmentazione semplificate

E' possibile anche usare direttive "semplificate" (la semplificazione non è molto significativa, in compenso si "nasconde" un po' troppo la realtà).

```
.DATA
<Pseudoistruzioni che definiscono i dati>

.CODE
<Istruzioni che costituiscono il programma>
```

Queste direttive non richiedono l'uso di una etichetta, né di una direttiva ENDS in fondo

Dunque invece di usare:

```
SEGMENT DATA
    <qui la definizione dei dati del programma>
ENDS

SEGMENT CODE
    <direttiva ASSUME>[1]
    <qui il codice delle istruzioni del programma>
ENDS
END <Ethichetta della prima istruzione da eseguire>
```

si usa:

```
.DATA
    <qui la definizione dei dati del programma>
.CODE
```

1 Per il dettaglio sulla direttiva ASSUME, vedi oltre

```
    <qui il codice delle istruzioni del programma>
END <Ethichetta della prima istruzione da eseguire>
```

Come si vede, con le direttive semplificate si risparmia la direttiva ASSUME, ma si perde un po' di "controllo" sulle cose che si possono fare. In definitiva l'adozione delle direttive di segmentazione complete o di quelle semplificate è più che altro una questione di gusti.
Nei programmi scritti per il Sistema Operativo Windows è possibile usare anche .DATA?, che indica l'inizio di un'area di dati non inizializzati, nella quale i dati non sono definiti all'inizio del programma ed assumono valori "casuali" (di solito i valori che sono stati scritti precedentemente da altri programmi)

1.2.1 Allocazione della memoria

L'atto di riservare aree di memoria viene detto, in gergo informatico, "**allocazione** della memoria" (memory allocation). Ogni Assembly deve avere delle direttive che permettano al programmatore di dichiarare come la memoria deve essere allocata. Vediamo ora quelle dell'Assembly 8086.

Direttiva define
Le direttive dell'Assembly X86 che allocano la memoria sono le "Define".
Esse hanno la forma:

```
    <DirettivaDefine> := <Nome Simbolico> D<Dimensione> [<CostruttoDUP>] (<Co-
stanteIniziale> | (?) )
```

La D sta per "Define" e specifica un'area di memoria della grandezza indicata da <Dimensione>

<Dimensione> è una lettera, da scrivere attaccata alla D:

```
    <Dimensione> := B | W | D | Q
```

B specifica un'area di memoria di un **B**yte, W di una **W**ord (due byte, 16 bit), D è una **D**ouble word (32 bit), Q è **Q**uad word, di 64 bit (quattro word). In definitiva avremo a disposizione le direttive: DB, DW, DD e DQ.

[<CostruttoDUP>] è una parte opzionale, che vedremo fra un po'

<CostanteIniziale>, oppure **(?)**, è l'indicazione del valore iniziale che l'Assembler deve far in modo che sia scritto in quella locazione di memoria all'atto del caricamento (load time). Può essere un numero decimale, binario o esadecimale

Come esempio assegniamo un'area di memoria di un byte e chiamiamola UnByte:

```
UnByte DB 0
```

La dichiarativa si legge: "UnByte (simbolo), Define Byte, valore iniziale zero". "Leggerla" in Inglese aiuta a ricordarsi l'istruzione e a non commettere errori quando la si usa.
Come altro esempio allochiamo due byte di memoria al nome UnaWord, senza inizializzazione:

```
UnaWord DW (?)
```

Il punto interrogativo significa che non ci interessa il valore che viene scritto inizialmente nelle due locazioni di memoria che corrispondono a UnaWord. L'Assembler scriverà il valore che gli fa più comodo, o lascerà ciò che era in memoria precedentemente.
Di conseguenza il programmatore non si potrà fidare del valore iniziale di UnaWord. Usare (?) al posto di un numero può far generare un file eseguibile più corto e/o un programma che viene caricato in modo più veloce.

Altri esempi
Per allocare aree di memoria più grandi di 4 word esiste il costrutto "**DUP**", che sta per "**Dup**licate". Esso è opzionale. La sua sintassi è questa:

```
    <CostruttoDUP> := <Numero> DUP
```

DUP fa ripetere la direttiva define che lo precede di tante volte quante indicate da <Numero>.

Per cui la direttiva:

```
Memoria1kWord DW 1024 DUP ("HL")
```

Significa che vengono allocate, in un'area di memoria contigua che si chiama Memoria1kWord, 1024 elementi di 2 byte word (1 word DW), per cui la memoria totale riservata è di 2 kByte. Nelle word allocate si scrive un valore iniziale di "HL", che, come considerato come numero a 16 bit, corrisponde a scrivere il codice ASCII di L nella parte bassa e quello di H nella parte alta di ogni word.

La sintassi dell'istruzione define è in verità un po' più complessa. E' possibile specificare una intera "lista" di valori iniziali da assegnare, usando la virgola:

```
<DirettivaDefine> := <Nome Simbolico> D<Dimensione> [<CostruttoDUP>] <ListaCostanti> | (?)

<ListaCostanti> := <CostanteIniziale> , ..
```

La scrittura precedente significa che, separando gli elementi con una virgola, è possibile assegnare, con la stessa direttiva, quanti elementi si vogliono dello stesso tipo.

Due esempi chiariranno la cosa:

```
TreByte DB 1, 2, 3
```

La direttiva precedente definisce un'area di memoria di tre byte, che viene inizializzata con i numeri 1, 2 e 3, in locazioni successive. Invece:

```
TreWord DW 1, 2, 3
```

definisce un'area di memoria di sei byte. Nello schema che segue vediamo come viene riempita la memoria se le due direttive appena viste sono scritte una dopo l'altra nel sorgente:

	Indirizzi	Memoria	Simboli	Direttive
		?		
		..		
(ipotesi)	C3216	1	TreByte	TreByte DB 1,
	C3217	2		,2
	C3218	3		,3
	C3219	1	TreWord	TreWord DB 1
	C321A	0		
	C321B	2		,2
	C321C	0		
	C321D	3		,3
	C321E	0		
		..		
		?		

Nelle locazioni di "TreWord" vediamo che all'indirizzo basso c'è la parte bassa del numero (CPU little endian).

La <CostanteIniziale> può essere indicata in qualunque base, a patto che il valore della base sia specificato alla fine del numero, il compilatore si prenderà carico della conversine di base, in modo che al momento dell'esecuzione del programma, nella locazione di memoria voluta ci sia il numero voluto, convertito in binario.
La frase precedente implica che la direttiva seguente riempie la memoria con quattro word uguali, tutte a 1:

```
    VettoreDiWord DW 65535, 0FFFFh, 1111111111111111b, 65535
```

Qualora la <CostanteIniziale> sia racchiusa fra apici la locazione di memoria corrispondente sarà inizializzata con il codice ASCII del carattere indicato, per esempio :

```
Car DB "A"
```

Metterà il valore 41 esadecimale, corrispondente al codice ASCII di "A", nella locazione che è assegnata a Car. Turbo Assembler non fa distinzioni fra apici e doppi apici, per cui:

```
Car DB 'A'
```

ha lo stesso effetto della precedente.

Se si mette più di un carattere fra apici il compilatore assegna più di un byte in locazioni successive, ciascuno con il codice ASCII del carattere indicato. Per esempio:

```
Stringa DB "AbcD"
```

Assegna la memoria in questo modo:

	Indirizzi	Memoria	Simboli	Direttive
		?		
		..		
(ipotesi)	06C16	41h ("A")	Stringa	Stringa DB "A
	06C17	62h ("b")		b
	06C18	63h ("c")		c
	06C19	44h ("D")		D"
		..		
		?		

Siccome i codici ASCII estesi sono di 8 bit non ha senso assegnarli con grandezze diverse dal byte:

```
    Stringa DW "AbcD"    ; !! NON HA SENSO (il compilatore non lo permette nemmeno) !!
```

L'Assembler non fa nessun controllo di tipo, per cui non c'è nessun problema nel mischiare costanti iniziali di tipo ASCII ad altre di tipo numerico. La seguente direttiva è perciò valida, ed anche molto usata:

```
    StringaTerminata DB "Una stringa per il C ha in fondo il null", 10,13, 0
```

La pseudoistruzione precedente mette in memoria tutti i codici ASCII indicati poi, dopo l'ultima "l" di "null", mette i numeri 10, 13 e 0, che sono i codici ASCII che permettono di andare a capo (10 e 13) e di indicare al C che la stringa è finita (il carattere "null" ha codice ASCII 0).

Vediamo ora un esempio di un primo programma "completo": la stampa di una stringa in MSDOS.
ATTENZIONE: il programma contiene alcune semplici istruzioni non ancora spiegare, ma i suoi commenti dovrebbero essere sufficienti per capire come funziona.

Per provare il programma fare così:
- salvare il seguente programma in un file ShowStri.ASM.
- compilarlo con: TASM ShowStri /zi
- collegarlo con: TLINK ShowStri /V
- verificarlo con: TD ShowStri
- eseguirlo con ShowStri

```
UnicoSeg SEGMENT
; allocazione della memoria che deve contenere il messaggio da visualizzare:
StringaMSDOS DB "AbcD$"

ASSUME CS:UnicoSeg, DS:UnicoSeg
InizioProg:
    ; un paio di magiche istruzioni che useremo sempre
    ; ma non spiegheremo subito:
    MOV AX, SEG UnicoSeg
    MOV DS, AX
    ; Scrittura della stringa con un comando MSDOS:
    ; metto in DX l'indirizzo dell'inizio della stringa da visualizzare:
    MOV DX, OFFSET StringaMSDOS
    MOV AH,09    ; richiesta del servizio DOS "stampa stringa"
    INT 21h      ; esecuzione del servizio DOS
    ; conclusione del programma:
    MOV AH,4Ch   ; richiesta del servizio DOS "terminate program"
    INT 21h      ; esecuzione del servizio DOS
    ; da questo punto il programma non passa mai perché è già finito
UnicoSeg ENDS
END InizioProg
```

Il funzionamento del programma precedente è più semplice di quanto non sembri a prima vista. Quando si saranno studiate pochi argomenti del seguito la sua comprensione sarà senz'altro immediata.

Direttiva di definizione di costanti
In molte occasioni è utile sostituire a valori costanti un'indicazione simbolica. Questo permette di scrivere programmi "parametrizzati" cioè programmi nei quali sono presenti valori costanti che però possono essere cambiati facilmente. Supponiamo di scrivere un programma che contenga dieci stringhe di 10 elementi. Per allocare la memoria possiamo fare così:

```
ST1 DB 10 DUP (?)
ST2 DB 10 DUP (?)
..
ST10 DB 10 DUP (?)
```

Se in un secondo tempo il programma deve essere modificato per funzionare con stringhe da 2048 Byte dobbiamo sostituire tutte le occorrenze di 10 che si trovano nelle definizioni. Se per non perdere tempo usiamo un text editor, possiamo cercare e sostituire automaticamente tutte le occorrenze della stringa "10" nel sorgente. Il risultato è il seguente:

```
ST1 DB 2048 DUP (?)
ST2 DB 2048 DUP (?)
..
ST2048 DB 2048 DUP (?)
```

Come si può vedere nella "nuova" label ST2048, la cosa ha effetti collaterali!
Per ovviare a questi problemi esiste una direttiva Assembly che esegue una sostituzione "letterale" di una stringa ad un'altra nel sorgente Assembly. Essa è la direttiva EQU.
La direttiva EQU ha la seguente sintassi:

```
<Simbolo> EQU <Stringa>
```

<Simbolo> è un normale identificatore definito dall'utente, come quello che si usa per le etichette, mentre <Stringa> è tutto ciò che va dalla fine di EQU alla fine della riga, oppure al punto e virgola di inizio commento.
<Simbolo> non è un'etichetta, nel senso definito precedentemente, perché non ha corrispondenza con nessun indirizzo.
L'Assembler, prima di compilare il programma, effettua una sostituzione, in tutto il sorgente, di <Stringa> al posto di <Simbolo>.
Per risolvere con la EQU il problema delineato precedentemente si fa così:

```
Lunghezza EQU 10
ST1 DB Lunghezza DUP (?)
ST2 DB Lunghezza DUP (?)
..
ST10 DB Lunghezza DUP (?)
```

Da notare che questa scrittura equivale in tutto e per tutto alla prima delle due scritture precedenti, perché l'Assembler sostituisce Lunghezza con 10 prima di compilare. Peraltro con questa scrittura per cambiare la lunghezza di tutte le stringhe sarà sufficiente modificare una linea di codice:

```
Lunghezza EQU 1024
```

La direttiva precedente renderà tutte le stringhe lunghe 1 kByte, senza effetti collaterali.
Si può considerare che la EQU sia una direttiva analoga alla #define del C. Come #define cambia automaticamente il sorgente con la stringa indicata, prima di compilare.

1.3 Trasferimenti di numeri e modi d'indirizzamento

Analizzeremo ora le modalità che permettono di trasferire informazioni all'interno della CPU e fra la CPU e la memoria. Il discorso sarà centrato sull'8086, ma si proverà a generalizzare e a dare qualche indicazione anche su modalità di trasferimento che non sono presenti in questa CPU.
I modi di trasferimento delle informazioni sono chiamati "modi d'indirizzamento". Qualche volta questo termine è usato in modo improprio, dato che le relative istruzioni non fanno nessun riferimento alla memoria in fase di esecuzione, per cui non usano nessun "indirizzo".
Per esemplificare le istruzioni useremo il codice mnemonico MOV, comune a diversi Assembly:

```
MOV <Destinazione>, <Sorgente>
; move
; funzionamento:
; <Destinazione> <- <Sorgente>
```

L'ordine con cui sono scritti la destinazione ed il sorgente è lo stesso che viene usato nell'istruzione MOV dell'Assembly 8086. Le istruzioni che seguono non sono specifiche di una CPU, per quanto si usi una notazione simile a quella utilizzata nell'Assembly 8086.

1.3.1 Trasferimento fra registri

```
MOV <RegistroDestinazione>, <RegistroSorgente>
```

Il contenuto di un registro viene trasferito in un altro. La memoria non è interessata da questa istruzione, che si svolge esclusivamente all'interno della CPU.
Spesso questa modalità di trasferimento dei dati viene detta "indirizzamento" fra registri. Si tratta di un nome non molto azzeccato, dato che questo trasferimento avviene senza coinvolgere la memoria e quindi gli indirizzi.

Esempi:

```
MOV AX, BX   ; Copia in AX il valore corrente di BX, cambia solo AX, il valore
             ; precedente di AX viene perso. Trasferimento a 16 bit.
MOV AL, BL   ; trasferimento a 8 bit
MOV EDX, ECX ; trasferimento a 32 bit (solo CPU da 386 in poi (386>))
```

Nell'8086 non è possibile trasferire dati fra registri di dimensioni diverse.
Esempi di istruzioni vietate e permesse:

```
MOV AX, BL   ; VIETATA: AX è da 16 bit, BL da 8
MOV AL, AH   ; permessa: AL e AH sono registri da 8 bit indipendenti(anche se
             ; assieme formano AX)
```

1.3.2 Trasferimento in modo immediato

```
MOV <Destinazione>, <Numero>
```

Un numero fisso, contenuto nel codice in linguaggio macchina del programma, e perciò immutabile, viene trasferito nella destinazione. In fase di execute la memoria non è interessata da questa istruzione.
Esempi:

```
MOV AX, 10   ; Scrive in AX il numero fisso 10, da 16 bit. Valore finale di AX: 10,
             ; Cambia solo il valore di AX
MOV AL, 10   ; trasferimento in immediato a 8 bit: il 10 è un 10 "da 8 bit".
MOV [100], 10    ; trasferimento di 10 in immediato alla locazione di indirizzo 100
```

1.3.3 Indirizzamento diretto o assoluto

```
MOV <Destinazione>, [<Numero>]
```

Si trasferisce in <Destinazione> il contenuto della locazione di memoria indicata dall'istruzione. L'indirizzo della locazione nella quale si lavora è <Numero> ed è scritto nel codice del programma. Perciò non può essere cambiato a tempo d'esecuzione.
Esempi:

```
MOV AL, [10] ; Legge in AL il contenuto corrente della locazione di indirizzo 10
MOV AX, [10] ; In AX il numero da 16 bit che comincia alla locazione di indirizzo 10;
             ; trasferisce due byte che si trovano in due locazioni consecutive
             ; in AL il contenuto della locazione 10, in AH il contenuto
             ; della locazione 11
MOV EAX, [10]    ; (386 >) In EAX il numero da 32 bit che comincia alla locazione
                 ; di indirizzo 10, vengono trasferiti i contenuti delle locazioni
                 ; 10, 11, 12, e 13.
MOV AX, [UnaWord]
```

L'ultimo esempio va commentato più ampiamente. A prima vista non sembra che UnaWord sia un numero.
Se ci ricordiamo però che l'Assembler, quando compila il programma, sostituisce il simbolo UnaWord con l'indirizzo che ad esso corrisponde nella tabella dei simboli ci rendiamo conto che in verità anche l'ultimo esempio è un indirizzamento assoluto.

L'indirizzamento assoluto si usa quando si deve avere accesso a specifiche locazioni in memoria, il cui indirizzo non deve cambiare durante tutta l'esecuzione del programma. Oltre che con le "variabili", ciò accade spesso nel caso di operazioni di I/O, nelle quali i dispositivi con cui la CPU comunica rispondono a particolari indirizzi di memoria.

1.3.4 Indirizzamento indiretto con registro

Tramite l'indirizzamento indiretto è possibile cambiare durante l'esecuzione del programma la locazione cui si accede.

```
MOV <Destinazione>, [<RegistroPerIndirizzo>]
```

Esistono molti tipi di indirizzamento indiretto; in quello "con registro" si accede alla memoria per il tramite di un registro, nel quale viene preventivamente messo l'indirizzo al quale si vuole lavorare.
Esempi:

```
MOV AX, [BX] ; In AX si legge il numero da 16 bit che comincia alla locazione
             ; il cui indirizzo è il valore corrente di BX
MOV AX, [SI] ; come la precedente, a parte che l'indirizzo è contenuto in SI
MOV AH, [SI] ; come la precedente, solo che prende dalla memoria una sola locazione
```

Istruzione analoga in C:
A = (*B)
Si usa quando si vuole cambiare, sotto controllo del programma, la posizione in memoria in cui si lavora. Ciò avviene tipicamente quando si vuole esaminare il contenuto di una intera area di memoria.

Nell'8086 i registri che si possono usare per puntare in modo indiretto alla memoria sono solo quattro: i registri base ed i registri indice. I registri base sono quelli che iniziano per B (BX e BP), i registri indice sono quelli che finiscono per I (SI e DI).

```
<RegistroPerIndirizzo> := <RegistroBase> | <RegistroIndice>
<RegistroBase> := BX | BP
<RegistroIndice> := SI | DI
```

Nelle CPU dal 386 in poi questa limitazione è stata rimossa e si possono usare tutti i registri della CPU, ad eccezione dei registri di segmento.

1.3.5 Indirizzamento base (displacement o based)

```
MOV <Destinazione>, [<RegistroPerIndirizzo> + <Displacement>]
```

Si utilizza un registro per contenere la "base" dell'indirizzo, dalla quale si parte, ad essa si aggiunge un numero fisso per spostarsi in memoria del numero di locazioni volute. Il numero fisso aggiunto alla "base" viene detto , detto "displacement" (spostamento).
La CPU calcola, durante la fase di execute di questa istruzione, l'indirizzo al quale deve andare ad operare, sommando il contenuto di <RegistroPerIndirizzo> con il numero 30, ottenuto in fase di fetch. Il risultato dell'accesso in memoria viene poi trasferito in <Destinazione>.

```
MOV AL, [BX + 30]; Valore finale di AL: il contenuto corrente della locazione
                 ; il cui indirizzo è ciò che contenuto in BX cui viene
                 ; aggiunto 30
MOV DX, [BX + 30]; istruzione che trasferisce due byte in DX, il primo è
                 ; all'indirizzo (BX + 30) il secondo a (BX + 31)
```

1.3.6 Indirizzamento indicizzato (indexed)

La CPU utilizza un registro per contenere la "base" dell'indirizzo ed un altro registro per contenere lo "spostamento" rispetto alla base (che spesso è l'indice di un vettore).

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

Valore finale di R0: il contenuto corrente della locazione il cui indirizzo è dato dalla somma di BX e SI
Cambia solo il valore di AX

Per ottenere l'indirizzo la CPU somma i due registri a tempo d'esecuzione.

1.3.7 Indirizzamento scalato (scaled o index)

Con l'indirizzamento scalato i programmi in Assembly possono essere più simili a quelli in linguaggi di più alto livello, perché tra le parentesi quadre si può usare l'indice in una struttura dati complessa, come un array.
L'8086 non ha un indirizzamento di questo tipo, ma ce l'hanno le CPU sue discendenti, dall'80386 in poi.

```
MOV <Destinazione>, [<RegistroPerIndirizzo> + <Indice> * <Scala>]
```

In <Destinazione> viene trasferito il contenuto della locazione all'indirizzo che viene calcolato moltiplicando <Indice> per <Scala> ed aggiungendo <RegistroPerIndirizzo>. Per dettagli, limitazioni ed esempi vedere nel volume successivo, nella trattazione della programmazione in modo protetto.

Esempio:

```
MOV AX, [BX + SI * 4]         ; !! istruzione solo per 386> !!
```

Questo tipo di indirizzamento è presente nelle CPU X86 solo a partire dall'80386.

1.3.8 Indirizzamento indiretto in memoria (o indiretto)

Quello che viene chiamato "indirizzamento indiretto", senza altri attributi, è un modo di accedere alla memoria che non esiste nella famiglia X86. Durante un trasferimento con indirizzamento indiretto si fa un primo accesso alla memoria, all'indirizzo indicato da un puntatore. Lì si trova l'indirizzo da usare effettivamente per trasferire il dato che interessa.

```
MOV <Destinazione>, [[Op2]]   ; La notazione usata è dell'autore
```

n.b. la notazione non può far riferimento all'8086 perché esso non usa questo modo di indirizzamento.

Valore finale di <Destinazione>: il contenuto corrente della locazione il cui indirizzo si trova il memoria all'indirizzo il cui valore è contenuto in Op2.

1.3.9 Indirizzamento con autoincremento

Questo modo di indirizzamento non esiste nella famiglia X86[2]. Anche in questo caso la notazione usata è dell'autore.

```
MOV <Destinazione>, [SI++]    ; La notazione usata è dell'autore
                              ; accede all'indirizzo contenuto in SI e salva
                              ; il valore letto in <Destinazione>, poi incrementa SI
```

In questa istruzione non cambia solo il valore di <Destinazione>, ma anche il puntatore, che viene incrementato automaticamente, dopo la lettura del dato dalla memoria, per essere pronto per la prossima iterazione.

1.3.10 Indirizzamento con autodecremento

Non esiste nella famiglia X86. E' analogo al precedente, con la differenza che il valore del registro indice viene decrementato automaticamente dopo la lettura del dato in memoria.

```
MOV <Destinazione>, [SI--]    ; La notazione usata è dell'autore
```

Con l'8086 usando le istruzioni di stringa (vedi oltre) si possono ottenere risultati analoghi a quelli che si avrebbero se esistesse un indirizzamento con autoincremento o autodecremento.
I modi di indirizzamento più usati sono immediato, con registro, indiretto con registro, indiretto e base.
CPU diverse usano modi d'indirizzamento diversi, alcune hanno modi d'indirizzamento davvero complicati e strani, anche se la tendenza attuale è verso la semplificazione, dato che i modi d'indirizzamento più complicati sono più difficili da usare, i programmi che ne fanno uso sono più proni agli errori ed i compilatori del linguaggi ad alto livello tendono a non usarli.

L'architettura di CPU sofisticate (VAX) e dei microprocessori RISC (*) tende a fare in modo che tutte le istruzioni possano utilizzare sempre tutti i modi d'indirizzamento dei quali la CPU dispone. In questo caso si parla di "set d'istruzioni **ortogonale**"[3].
Se un set di istruzioni è ortogonale si semplifica ciò che il programmatore deve tenere a mente e ci sono meno casi particolari o eccezioni alle regole del linguaggio. Ciò può portare ad una maggiore produttività nella programmazione a basso livello e ad una semplificazione dei compilatori.

Conoscere i "nomi" dei modi d'indirizzamento è un valido aiuto nella fase di debugging del programma. Quando un Assembler non è in grado di compilare un'istruzione del sorgente scrive delle indicazioni d'errore che possono far riferimento al modo d'indirizzamento, come p.es. "invalid immediate mode operation". Se non si sa cosa vuol dire "immediate mode" non si riesce a togliere l'errore.

2 peraltro le cosiddette "istruzioni di stringa" X86 fanno qualcosa del genere
3 per la definizione di RISC, vedi iin seguito

1.3.11 Dettagli sull'8086

Scritture
Gli esempi forniti nel paragrafo precedente sono tutti relativi a letture, dalla memoria alla CPU.
Invertendo l'ordine con cui sono scritti gli operandi si ottengono delle scritture: il dato che si vuole trasferire va dalla CPU alla memoria. Vediamo alcuni esempi che corrispondono a quelli precedenti:

```
MOV [10], AL ; Scrive il valore corrente di AL alla locazione 10,
             ; con indirizzamento assoluto
MOV [UnaWord], AX    ; scrive AX con indirizzamento assoluto nelle due locazioni che
                     ; corrispondono a UnaWord
MOV [SI], AH ; scrive con indirizzamento indiretto tramite il registro SI
MOV [BX + 30], AX    ; scrive all'indirizzo BX con displacement 30
MOV [BX + SI], AX    ; scrive all'indirizzo che ha base BX e indice SI
MOV [BX + SI * 4], EAX   ; indirizzamento scalato in scrittura (solo per 386>)
```

Le altre modalità di indirizzamento descritte nel paragrafo precedente non sono disponibili nell'8086.

Naturalmente la seguente non ha alcun senso:

```
MOV 10, AX   ; VIETATO!! L'indirizzamento immediato non ha senso in scrittura:
             ; come si potrebbe scrivere nel numero 10?
```

In alcuni casi l'istruzione fa riferimento a due operandi diversi da registri. Se ciò accade ci sono due possibilità di "indirizzamento" nella stessa istruzione. Fatto salvo il fatto che due indirizzi diversi in fase di execute non possono essere usati, è possibile fare i seguenti trasferimenti "misti":

```
MOV [Destinazione], 10   ; l'indirizzamento di [Destinazione] è assoluto,
                         ; il numero 10 è trasferito in immediato
MOV [BX], 10             ; l'indirizzamento in memoria è indiretto tramite BX,
                         ; il numero 10 è trasferito in immediato
MOV [BX], [10]           ; NON SI PUO' !
```

Nell'8086 non solo l'istruzione di trasferimento (MOV) ammette come operando una locazione di memoria. Infatti ciò avviene per quasi tutte le istruzioni.
Per esempio:

```
ADD [Somma], 10
```

È un'istruzione legittima che va in memoria all'indirizzo che l'Assembler associa a Somma, legge il contenuto della locazione, lo somma a 10 e scrive il risultato nella stessa locazione "Somma".

Una limitazione importante relativa a tutte le istruzioni 80X86 è la seguente: uno solo degli operandi di un'istruzione può essere in memoria.
Non è mai possibile accedere a due locazioni di memoria diverse durante la stessa istruzione.
Un'istruzione Assembly 80X86 può fare riferimento ad un solo indirizzo di memoria. L'altro operando può essere solo un registro od un numero fisso.
La vista di due parentesi quadre nella stessa istruzione 8086 significa che c'è un errore di sintassi nel programma.

Vediamo alcuni esempi di ciò che è permesso e di ciò che non lo è:

```
; Supponiamo che sia:
Somma DB ?
Addendo DB ?

MOV AX, 1    ; nessun accesso in memoria in fase di execute: permesso

MOV AX, SI   ; nessun accesso in memoria in fase di execute: permesso

MOV BX, [8]  ; accesso in memoria in fase di execute alla sola locazione 8: permesso

ADD [Somma], 16  ; accesso in memoria alla sola locazione che il compilatore assegna a
                 ; Somma: permesso

ADD BYTE PTR [8], 8  ; accesso in memoria alla sola locazione 8: permesso
                     ; (vedi in seguito per i dettagli su BYTE PTR)

MOV [BX], [11]       ; accesso alla locazione il cui indirizzo è in BX ed alla
```

```
                    ; locazione 11: VIETATO

ADD [Somma], [Addendo]   ; accesso a due locazioni diverse: VIETATO

ADD [Somma], Addendo ; IDENTICA alla precedente, accesso a due locazioni diverse: VIETATO

ADD Somma, Addendo   ; IDENTICA alla precedente, accesso a due locazioni diverse: VIETATO

ADD A, [10]          ; Accesso alla locazione che il compilatore associa ad A ed
                     ; alla locazione 10: VIETATO
```

Una parziale eccezione a questa regola è rappresentata dalle istruzioni di stringa, che lavorano su due diverse locazioni di memoria. Peraltro c'è da dire che esse non hanno nessun operando esplicito, per cui non compaiono due parentesi quadrate (vedere nel seguito per i dettagli).

Esempio:

```
    SCASB    ; è un'istruzione di stringa, non ha operandi espliciti
```

Accesso alla memoria nell'8086

Come già visto la presenza di parentesi quadre significa: "accedi alla memoria, all'indirizzo che trovi all'interno delle parentesi". Ciò che è "fuori" dalle parentesi è il contenuto delle locazioni di memoria indicate all'interno.
L'8086 ha limitazioni significative, rimosse solo dal 386 in poi, sull'uso dei registri nell'accesso indiretto alla memoria.
All'interno delle parentesi quadre si possono usare solo i registri base (quelli che iniziano con B: BX e BP) e/o i registri indice (quelli che finiscono per I: SI e DI).
La CPU è in grado di sommare a tempo di esecuzione un registro base ad un registro indice e ad un numero fisso detto "**displacement**".
Questo significa che è possibile scrivere così:

```
[BX + SI]
```

Questa espressione significa che la CPU esegue, a tempo d'esecuzione, la somma fra il contenuto corrente di BX e quello di SI ed utilizza come indirizzo per l'accesso alla memoria la somma che ha appena calcolato.
Nell'8086 non si possono sommare due registri base, né due registri indice, per cui i seguenti indirizzamenti sono vietati:

```
[SI + DI]    ; VIETATO
[BP + BX]    ; VIETATO
```

La seguente figura aiuta nella memorizzazione dei registri che possono essere usati nell'accesso alla memoria, e di quelli che non possono essere sommati insieme:
<FILE>
accesso.FH5
</FILE>

Figura 1: limitazione 8086 nell'accesso alla memoria

Gli unici registri puntatori dell'8086, quelli che possono stare fra le parentesi quadre, sono quelli indicati nella figura.
I registri messi in colonna nella figura non si possono sommare insieme. Ai registri indice e base si può aggiungere un numero **fisso**, detto **displacement**.
Il registro BP ha una sottolineatura ondulata per significare che non è il caso di usarlo prima di aver capito bene la segmentazione e lo stack.
Ogni elemento della figura è opzionale, fra le parentesi quadre se ne può mettere uno solo.

Vediamo ora alcuni accessi alla memoria "legali":

```
[1 + 2]  ; la somma viene calcolata una volta per tutte dall'Assembler
         ; a tempo di compilazione e diviene [3] nel programma compilato
[A]      ; è un numero fisso, viene determinato dal compilatore a tempo di compilazione
         ; e sostituito con il numero che compete ad A nella tabella dei simboli del
         ; programma
[A + 1]  ; è un numero fisso calcolabile a tempo di compilazione, come somma fra 1
         ; e l'indirizzo di A
[A + SI]     ; A è un numero fisso, SI è il valore contenuto in SI al momento dell'esecu-
zione
         ; dell'istruzione, per cui la somma deve essere fatta a runtime, da parte
         ; della CPU
[A + BX + 3] ; la somma A + 3 è un displacement e viene calcolata a tempo di compilazione
             ; la somma fra (A + 3) e BX viene invece fatta dalla CPU a tempo
             ; d'esecuzione
```

Poniamo ora l'attenzione su ciò che è "fuori" dalle parentesi quadre. Mentre dentro le parentesi quadre c'è sempre l'offset di un indirizzo, che è sempre un numero di 16 bit, ciò che è "fuori" può essere lungo 8, 16 o anche 32 bit.
Con una CPU X86 è possibile trasferire 1 Byte, 2 Byte od anche 4 Byte con una singola istruzione.

Quando il compilatore può capire di quanti byte deve essere il trasferimento accetta l'istruzione e riesce a compilarla. Questo è il caso delle seguenti istruzioni:

```
MOV AX, [10] ; trasferimento di due byte: la dimensione della destinazione stabilisce
             ; il parallelismo del trasferimento
ADD [28], EBX    ; operazione a 32 bit, perché l'operando sorgente è EBX, da 32 bit
                 ; (CPU 386>)
MUL BX           ; operazione a 16 bit
MOV BL, [BX]     ; trasferimento da 8 bit, governato da BL
```

Nel caso delle seguenti istruzioni può sussistere un certa ambiguità, che il compilatore in genere risolve facendo delle assunzioni:

```
Valore DW (?)
SUB [Valore], 1
```

In questa istruzione non si sa di quanti bit è il numero 1. Il compilatore può però assumere che l'operazione sia fatta con lo stesso numero di bit con il quale precedentemente si era definito Valore; dato che "Valore" era stato definito come: "Valore DW (?)" l'Assembler assume che la sottrazione sia a 16 bit.
Dunque in questi casi il compilatore assume che il parallelismo dell'operazione sia lo stesso dichiarato quando si è fatta la "define" della "variabile".
Ne consegue che, con "Valore DW (?)", la seguente istruzione dà errore:

```
SUB AL, [Valore] ; errore, una cosa che è dichiarata word non ci sta in AL!.
```

Illustriamo ora un'anomalia del compilatore TASM (e anche di MASM). Esso compila senza dare errori un'istruzione come la seguente:

```
MOV [Inizio - 10], AL
```

Il valore del displacement è di 16 bit e viene sempre considerato dalla CPU un numero senza segno, per cui si possono usare numeri positivi fino a 65535. L'istruzione precedente va perciò ad un indirizzo maggiore di Inizio, non minore, come la scrittura darebbe ad intendere.
Come conferma si potrebbe può provare a compilare le due seguenti istruzioni:

```
MOV AL, [Inizio + 0FF81h]     ; viene identica alla successiva!
MOV AL, [Inizio - 7Fh]        ; viene identica alla precedente!
```

si ottiene la stessa istruzione in linguaggio macchina. C'è da dire che 7Fh è il complemento a due di 0FF81h. Il compilatore accetta la sintassi con la sottrazione e compila "regolarmente" con il complemento a due. La locazione cui si accede però è sempre in avanti, quindi la seconda istruzione scritta è sbagliata perché dà l'impressione si acceda a locazioni che stanno PRIMA di Inizio.

C'è un caso in cui per il compilatore è impossibile stabilire, nemmeno implicitamente, con quanti bit l'istruzione deve lavorare; esso è illustrato di seguito:

```
MOV [SI], 1          ; SBAGLIATO!
```

Non è dato di sapere quanti bit ha il numero 1, inoltre anche la destinazione non aiuta, perché c'è un indirizzamento indiretto tramite registro; si usa solo il registro SI, che è di 16 bit, ma è usato come indirizzo, per cui non abbiamo nessuna indicazione su quanti byte devono essere trasferiti.
In un'istruzione come la precedente l'unica cosa che il compilatore può fare è fermarsi ed emettere un messaggio del tipo: "Argument needs type override".
L'unico modo per risolvere questa ambiguità è rendere disponibile un costrutto sintattico con cui il programmatore possa dichiarare: "questa deve essere un'operazione da 8 bit".

PTR

Il costrutto sintattico del quale si evidenziava l'esigenza nel paragrafo precedente esiste ed è la direttiva **PTR** (la sigla viene dalla parola **P**oin**t**e**r**). PTR ha la seguente sintassi:

```
<CostruttoPTR> := <ParallelismoOperazione> PTR
<ParallelismoOperazione> := BYTE | WORD | DWORD
```

<CostruttoPTR> va scritto prima della parentesi quadra che individua l'accesso alla memoria del quale si vuole specificare il parallelismo (vedi esempio successivo).
<ParallelismoOperazione> indica il parallelismo nell'accesso alla memoria con il quale il programmatore vuole effettuare l'istruzione. BYTE indica che si vogliono trasferire 8 bit, WORD indica 16 bit, DWORD 32.

Dal 386 in poi si può usare anche QWORD, dato che alcune istruzioni del 386 usano operandi da 64 bit.

Per risolvere l'ambiguità dell'ultima istruzione vista si può perciò fare così:

```
MOV WORD PTR [SI], 1      ; giusto!
```

In questo modo il trasferimento viene fatto a 16 bit.
PTR può essere usato anche con un 386, che ha data bus di 32 bit; in questo caso si possono usare istruzioni del tipo:

```
MOV DWORD PTR [SI], 1        ; giusto (solo 386>)
```

Se si vuole trasferire un numero diverso di dati rispetto a quelli che verrebbero trasferiti automaticamente, si può usare PTR. Supponiamo per esempio di voler cancellare la sola parte alta del numero "Valore", che avevamo prima definito come numero di 16 bit (DW). Il modo per scrivere nella sola parte alta è il seguente:

```
MOV BYTE PTR [Valore + 1], 0
```

Alcuni compilatori per 8086, come per esempio NASM, non fanno nessuna assunzione sul parallelismo dell'operazione, per cui, con quei compilatori, se uno dei due operandi non è un registro bisogna esplicitamente indicare se l'operazione deve essere "a byte" o con altro parallelismo.

BP e lo stack

Un'altra caratteristica molto importante cui bisogna far attenzione è che il registro di segmento di default che viene usato quando si mette BP fra le parentesi quadre è SS e non DS, come negli altri casi. Ciò significa che BP servirà per avere accesso ai dati dello stack, come vedremo in particolare quando tratteremo delle procedure.

Così, per il momento, non useremo BP, fino a che non sarà ben chiaro come bisogna fare per non sbagliare!

Altre istruzioni di trasferimento

XCHG E**xch**an**g**e

```
XCHG <operando1>, <operando2>
; Exchange
; scambia il contenuto degli operandi
; funzionamento:
; tmp <- <operando2>
; <operando2> <- <operando1>
; <operando1> <- tmp
```

Scambia il contenuto degli operandi. Essi possono essere registri e (uno solo dei due) locazioni di memoria.
Esempio:

```
; scambio fra AX e BX, senza XCHG:
MOV CX, AX   ; devo coinvolgere un altro registro!
MOV AX, BX
MOV BX, CX

; scambio fra AX e BX, con XCHG:
XCHG AX, BX  ; nessun altro registro coinvolto
             ; e un'istruzione sola
```

Le istruzioni XCHG che scambiano il contenuto di un registro e di una locazione di memoria sono da evitarsi nei programmi per le più moderne CPU della famiglia X86 (486, Pentium). Infatti esse vengono eseguite bloccando tutte le altre istruzioni che potrebbero eseguire contemporaneamente e penalizzando fortemente le prestazioni del sistema.

Una brutta sintassi

L'Assembly 8086 ammette una sintassi comoda ma pericolosa, che si sconsiglia decisamente.
Per selezionare un determinato elemento in una struttura dati è ammessa la seguente sintassi:

```
<NomeSimbolico> [ <OffsetIndirizzo> ]
```

dove <NomeSimbolico> è il nome di un'area di dati dichiarata con una "define" e <OffsetIndirizzo> segue le stesse regole appena spiegate per la formazione dell'indirizzo.
Questa sintassi permette quindi di scrivere così:
A[6] o anche B[BX]
costrutti che hanno lo stesso significato di:
[A + 6] e [B + BX]
La prima scrittura evoca la sintassi dei linguaggi ad alto livello relativa agli array. Questo rende più "naturale" la lettura dei programmi, ma può dare adito ad ambiguità che possono portare ad errori importanti. Infatti nei linguaggi come il C ed il Pascal fra le parentesi quadre si scrive l'indice dell'array, mentre in Assembly c'è "solo" un indirizzo, che è cosa

ben diversa. Per chiarire la differenza facciamo un confronto fra due versioni, una in C ed una in Assembly dello "stesso" programma:

C	Assembly
int a[3], temp	a DW 3 DUP (?)
	Temp DW (?)
..	..
temp = a[2]	MOV AX, a[2]
	MOV Temp, AX

Sembra, a prima vista, che le due versioni siano identiche ma se le esaminiamo meglio scopriamo che sono molto diverse.
L'istruzione C "temp = a[2]" scrive nella variabile temp il valore dell'elemento di indice 2 del vettore, a[ ], che è costituito da elementi di tipo int, di due byte (consideriamo un C a 16 bit, come Turbo C++). Perciò essa leggera dalla memoria agli indirizzi (a + 4) e (a + 5) (o meglio, con sintassi C: (&a + 5))
La corrispondente istruzione Assembly ("MOV AX, a[2]") legge il contenuto delle locazioni di indirizzo (a + 2) e (a + 3).
Perciò i due frammenti di programma, anche se simili all'aspetto, accedono a locazioni ed hanno effetti del tutto diversi.
C'è un'altra ragione per cui non è bello usare la notazione "tipo array". Esaminiamo questa istruzione:

```
MOV AL, AmbiguoEsubdolo
```

E pensiamo a cosa fa l'istruzione se AmbiguoEsubdolo è definito in questi due modi:

```
AmbiguoEsubdolo DB 48
```

oppure:

```
AmbiguoEsubdolo EQU 48
```

Nel primo caso (DB) l'istruzione è compilata così:

```
MOV AL, [<indirizzo di AmbiguoEsubdolo nella tabella dei simboli>]
```

c'è un accesso alla memoria in fase di execute ed esiste una locazione di memoria cui viene associato AmbiguoEsubdolo, il trasferimento è con indirizzamento diretto. In AL finisce un numero che non sappiamo, letto dalla memoria.

Nel secondo caso (EQU) l'istruzione è la seguente:

```
MOV AL, 48
```

non c'è alcun accesso alla memoria ed il trasferimento è in immediato, in AL finisce un 48 da 8 bit.

Questo modo di operare porta un'ambiguità di fondo e fa nascere equivoci che possono portare ad errori clamorosi.
Dunque, per questo ed anche per chiarezza e rigore formale, d'ora in poi scriveremo sempre le parentesi quadre quando intenderemo accedere a locazioni di memoria, anche quando l'Assembly sarebbe disposto a dimenticarle.
Esistono Assembler per 8086, più rigorosi di quelli derivati direttamente dal primo Assembler prodotto da Intel, che obbligano a mettere sempre le parentesi quadre quando si vuole accedere alla memoria.
Peraltro molti preferiscono la sintassi "tipo array", per cui si possono trovare libri e programmi che seguono quella convenzione.

Un'ultima annotazione sul calcolo dell'indirizzo deriva da alcuni errori che vengono compiuti comunemente.
Il fatto che la CPU esegua calcoli sugli indirizzi induce alcuni a pensare che quei calcoli si possano eseguire liberamente anche quando non si deve accedere alla memoria.

La seguente istruzione è legittima:

```
MOV AX, [BX + 11]    ; scrive in AX il contenuto della locazione
                     ; il cui indirizzo è uguale al valore corrente
                     ; di BX cui è aggiunto 11
; il numero BX + 11 è un indirizzo come dice anche il fatto
; che sta fra parentesi quadre
```

Questa istruzione può essere eseguita perché nella BIU c'è un sommatore di indirizzi (vedi) che permette di calcolare a tempo d'esecuzione l'indirizzo da usare.
Quel sommatore funziona però solo per gli indirizzi; non si può pretendere di farlo funzionare anche per i dati.
Perciò la seguente istruzione è sbagliata:

```
    MOV AX, BX + 11  ; sbagliata, non compila
```

quest'istruzione non si può compilare. Se con questa istruzione si vuole scrivere in AX il valore contenuto in BX aggiunto di 11, non si deve far lavorare il sommatore di indirizzi, che somma solo indirizzi e perciò vuole le parentesi quadre, ma la ALU.
Per far lavorare la ALU bisogna usare la ADD; per scrivere in AX il numero (BX + 11) bisogna fare così:

```
    MOV AX, BX
    ADD AX, 11   ; In AX ora c'è BX  + 11
```

```
Il sommatore d'indirizzi è un sommatore, no un moltiplicatore; un altro errore che si
vede spesso è il seguente:
```

```
    MOV AX, [BX * 4] ; sbagliata perché provo a calcolare l'indirizzo
                     ; con una moltiplicazione
```

Il sommatore d'indirizzi fa tutto quel che può ma, almeno fino al 286, non può fare moltiplicazioni (dal 386 le può fare per 2, 4 e 8).

Ottenere l'indirizzo di una locazione: OFFSET e LEA
Molto spesso è necessario ottenere l'indirizzo di una label. In Assembly 8086 questo è possibile in due modi diversi, che in molte occasioni sono equivalenti. Un modo è l'uso della direttiva OFFSET, l'altro è l'istruzione LEA.

Direttiva OFFSET
OFFSET è una direttiva del compilatore che produce l'indirizzo dell'etichetta utilizzata[4]

OFFSET si usa sempre in istruzioni MOV, con una la seguente sintassi:

```
    MOV <Registro>, OFFSET <label>
```

In <Registro> viene scritto il valore dell'indirizzo della label.
Dato che la direttiva sostituisce OFFSET <label> con un numero, l'accesso della MOV è di tipo immediato.
Nel caso della direttiva OFFSET il registro usato è tipicamente uno dei registri puntatori (BX, BP, SI, DI), dato che essi sono gli unici che possono essere usati fra le parentesi quadre.
Esempio:

```
    MOV SI, OFFEST Stringa   ; mette la parte di offset dell'indirizzo in SI
    ; se poi di fa così:
    MOV AL, [SI]
    ; si legge il primo carattere di Stringa
```

Istruzione LEA
LEA (**L**oad **E**ffective **A**ddress) è un'istruzione, quindi funziona a tempo d'esecuzione. Esegue il calcolo dell'offset dell'indirizzo della locazione indicata e lo mette nel registro indicato.

```
    LEA <Registro>, <Memoria>
```

<Memoria> è l'indicazione di un normale accesso alla memoria, come si potrebbe scrivere in una MOV.
<Registro> non può essere un registro di segmento.
Nei casi più semplici l'effetto di LEA è identico alla OFFSET:

```
    LEA DI, [Stringa]; ha lo stesso effetto della seguente:
    MOV DI, OFFSET Stringa
```

In questo caso OFFSET è migliore, perché è più autoesplicativa.
L'accesso non è in immediato, come con OFFSET, LEA svolge effettivamente un calcolo a tempo d'esecuzione e l'indirizzo che viene calcolato può essere più complicato. In questi casi LEA diventa più interessante di OFFSET. Supponiamo di volere l'indirizzo del terzo elemento del vettore di word "Vettore" e confrontiamo come possiamo ottenerlo con LEA e OFFSET:

Con LEA	Linguaggio Macchina	Con OFFSET	L. M.
LEA SI, [Vettore + 6]	8D360600	MOV SI, OFFEST Vettore	BE0000
		ADD SI, 6	83C606

La versione con LEA non è solo più compatta (4 byte contro 6), ma è anche molto più veloce nell'esecuzione, dato che la ADD da sola è più "costosa" di tutta la LEA, in termini di tempo.
Nel caso seguente i vantaggi sono ancora superiori:

4 o meglio, come vedremo nel capitolo relativo alla segmentazione, la parte di offset dell'indirizzo dell'etichetta

```
LEA DI, [BX + SI + 6]; carica in DI il numero BX + SI + 6
MOV DI, OFFSET Stringa
```

Usando LEA in questa istruzione il calcolo viene fatto a tempo d'esecuzione, con OFFSET questo non sarebbe stato possibile, dato che in BX ed in SI ci sono già indirizzi, per usare OFFSET si dovrebbe fare un codice più complesso, di alcune righe (si lascia come esercizio la realizzazione di questo codice).

1.3.12 Architetture Load/Store o "memory to memory"

La maggior parte dei primi microprocessori, nei quali era indispensabile mantenere una grande semplicità interna, aveva un'architettura "ad accumulatore", detta "**load/store**". Con quelle CPU prima di compiere operazioni aritmetiche bisognava caricare i dati in registri specifici, solo in seguito si potevano eseguire i calcoli, solo fra pochissimi registri ben specificati.
Il risultato dell'operazione finiva sempre nello stesso registro, detto **accumulatore**, e da lì doveva essere copiato "a mano", con un'istruzione, in memoria o negli altri registri.
Le CPU di architettura load/store (diverse dall'8086!) tendono a favorire al massimo l'uso dei registri, per cui hanno due sole istruzioni che comunicano con la memoria: la **LOAD** per leggere (load = "carica") e la **STORE** per scrivere (store = "immagazzina"). Le istruzioni LOAD e STORE non possono far altro che leggere o scrivere, nelle CPU load/store sono vietate le istruzioni come per esempio le seguenti:

```
; istruzioni "memory to memory" dell'8086:
ADD AX, [Somma]
ADD [Somma], 3
```

la prima delle due istruzioni legge un operando dalla memoria e lo somma direttamente ad AX, mentre la seconda fa addirittura due accessi alla locazione [Somma], uno in lettura, per procurarsi l'operando ed uno in scrittura, per memorizzare il risultato.

Le CPU come l'8086, che, al contrario, sono in grado di effettuare le operazioni direttamente in memoria, sono dette "**memory to memory**". A voler parlare in linea di principio, esse potrebbero addirittura fare a meno dei registri, come di fatto succedeva nel 6502, la CPU del Commodore 64.
Anche nelle architetture memory to memory sono peraltro presenti dei registri; questo per ragioni di efficienza, dato che utilizzando i registri un programma può funzionare in modo molto più veloce.

Sembra un po' assurdo ma, per tutt'altre ragioni, il modo di funzionare delle CPU ad accumulatore è condiviso dalle CPU più moderne, quelle di tipo RISC. Queste CPU hanno moltissimi registri e le operazioni aritmetiche possono essere svolte solo fra di essi. L'unico modo per caricare gli operandi e scaricare i risultati è usare una LOAD od una STORE.
Questo incoraggia il programmatore ad usare il più possibile i registri, dato che ve ne sono molti e le operazioni dirette in memoria sono impossibili.
Inoltre la presenza di molti registri all'interno della CPU rende più facile effettuare lunghi calcoli senza fare mai accesso alla memoria, utilizzando i registri per le memorizzazioni dei risultati intermedi. Questo permette ai programmatori Assembly, ma soprattutto ai compilatori, di ottimizzare molto più facilmente il codice, con l'utilizzazione massiccia dei registri in luogo della memoria.

1.4 Realizzazione di dati tipizzati e di strutture dati

In Assembly non esistono tipi di dati. Il compilatore non è interessato a sapere che tipo di numeri verrà memorizzato all'interno delle locazioni di memoria che esso deve allocare.
Il programmatore ha totale libertà nella manipolazione della memoria, il che è vantaggioso se si vuole avere grande flessibilità operativa, ma è anche pericoloso, perché il compilatore non aiuta nel controllo di tipo.
Nei linguaggi di alto livello fortemente tipizzati, come per esempio il Pascal o le versioni moderne di BASIC, se si prova a fare istruzioni che non rispettano il tipo dichiarato delle variabili c'è una indicazione di errore, che previene i malfunzionamenti gravi.
Tutto ciò non esiste in Assembly, ove posso trattare il contenuto di una word di memoria una volta come integer, una volta come unsigned integer ed anche farla a pezzi, operando con la sola parte bassa o con quella alta. Il compilatore lascerà fare tutte queste angherie sulla povera "variabile" e genererà codice "legale", anche se non necessariamente "funzionante".
Dunque il programmatore Assembly dovrà essere molto accorto e smaliziato, se vorrà evitare errori!

Vediamo un esempio, che si potrà apprezzare meglio dopo aver studiato le istruzioni di salto:

```
A DW (?)         ; A è una word
..
CMP [A], 21  ; confronto tutto A con 21
JG intMaggiore   ; lo considero un numero con segno
JA unsigMinore   ; lo considero un numero senza segno
; se arriva qui è uguale a 21
MOV BYTRE PTR [A + 1], 1 ; aggiungo 256 al numero 12, scrivendo
                         ; 1 nella sola sua parte alta
                         ; (prima c'era 0)
..
```

Naturalmente l'esempio appena visto non ha molto senso come programma, ma esso verrà tranquillamente accettato da ogni Assembler per 8086, anche se tratta A prima come int, poi come unsigned int, poi come due char.
Questo dimostra come in Assembly sia possibile avere grande controllo sulla CPU e fare trucchi impensabili con gli altri linguaggi. Naturalmente in questi casi è facile perdere il controllo del programma, ed esso può facilmente divenire non affidabile. Dunque in questi casi si impone disciplina e autocontrollo. In parole povere non bisogna esagerare con i trucchi.

Dati con tipo

!!!! to be completed !!!!
Tabella riepilogativa sulla rappresentazione dei tipi numerici

"Assembly"	Visual BASIC PC (ver.5.0>)	Turbo C e C++ Per X86 16 bit	Visual C e C++ per X86 32 bit	gcc per Linux – Windows 32 bit	Turbo Pascal per X86 16 bit	Java multipiattaforma
Intero 8 bit con segno	-	char	char	char		byte
Intero 8 bit senza segno	Byte	unsigned char	unsigned char	unsigned char		-
Intero 16 bit con segno	Integer	int	Short int	short int		short
Intero 16 bit senza segno	-	unsigned int	unsigned short int	unsigned short int		-
Intero 32 bit con segno	Long	long int	int	int		int
Intero 64 bit con segno	-	-	long int	-		long
Virgola mobile 32 bit	Single	float	float	float	real	float (IEEE 754)
Virgola mobile 64 bit	Double	double	double	double		double (IEEE 754)
Virgola mobile 96 bit	-	-	-	long double	-	-

In C il tipo integer (int) è della grandezza che più fa comodo nel linguaggio macchina della CPU per la quale il programma viene compilato.

Casting
Viene anche detto "promozione". La presenza di operandi di tipo diverso implica la conversione di tutte le variabili dei tipi più "piccoli" in variabili temporanee del tipo "più grande". I linguaggi di alto livello fanno il casting automaticamente, mentre in Assembly lo si può fare solo "a mano". Un esempio semplice è fornito in XXXXXX.

Uso di base e indice per scandire una matrice.

1.4.1 Strutture dati in Assembly

In Assembly non esistono strutture dati.
L'unica COSA che abbiamo a disposizione è il "contenitore" delle strutture di dati, vale a dire la memoria. Con le direttive che abbiamo visto il programmatore può allocare aree di memoria destinate ad essere usate come strutture dati. Oltre a quello il linguaggio non dà altro; il compito di "gestire" la memoria allocata è tutto sulle spalle del programmatore. Vediamo come si gestiscono alcune delle strutture dati tipiche dei programmi ad alto livello.

Array
Molti valori omogenei

Vettori

Matrici

Stringhe

Accesso ad elementi di una struttura di dati

!!!! to be completed !!!!

Supponiamo che sia stato definito un vettore, così:

```
Vettore DW 10 DUP (?)
```

Vediamo i diversi modi con cui possiamo leggere l'elemento 5 di questo vettore.

Approccio "con displacement"
Per accedere ad uno qualsiasi degli indirizzi del vettore uso il suo nome come punto di inizio gli aggiungo un registro puntatore con il quale sposto

```
MOV BX, 10
MOV AX, [Vettore + BX]    ; indirizzo dell'inizio della struttura +
                          ; displacement dell'elemento che interessa
```

In questo modo si accede al quinto elemento della struttura dati "vettore".
Questo modo di raggiungere gli elementi di un struttura dati è simile a quello adottato nei linguaggi ad alto livello, con la differenza che non si tratta con indici ma con indirizzi.

Approccio "con puntatore"
In un registro puntatore metto l'indirizzo d'inizio del vettore. Per spostarmi all'elemento voluto modifico il valore del puntatore.

```
MOV BX, OFFSET Vettore    ; BX "punta" all'inizio del vettore
ADD BX, 10       ; sposto al quinto elemento la locazione cui BX punta
MOV AX, [BX]     ; lettura del quinto elemento
```

In alternativa, con LEA

```
LEA BX, [Vettore + 10]    ; BX "punta" direttamente al quinto elemento
MOV AX, [BX]     ; lettura del quinto elemento
```

Nella seconda versione si mette il puntatore direttamente sulla locazione che interessa, facendo la somma fra l'indirizzo di Vettore e 10 con la LEA, senza ADD.

Approccio "base + indice"

```
MOV BX, OFFSET Vettore    ; BX punta alla "base" del vettore
MOV SI, 10                ; SI fa spostare al quinto elemento
MOV AX, [BX + SI]         ; lettura del quinto elemento
```

Questo approccio può essere utile quando si deve rapidamente "tornare all'inizio" della struttura. In questo caso basta azzerare SI e l'ultima istruzione carica il primo elemento.

Approccio "scalato" (scaled indexed)

```
MOV BX, OFFSET Vettore    ; BX punta alla "base" del vettore
MOV SI, 5                 ; SI è il vero e proprio indice del vettore, non
                          ; è un indirizzo!
MOV AX, [BX + SI * 2]     ; lettura del quinto elemento (386>, N.B. questa
                          ; sintassi non può funzionare, la CPU deve essere
                          ; in "modo protetto".
```

Questo è il modo più efficace e semplice da programmare. Si usa sempre l'indice dell'elemento della struttura in cui si vuole lavorare, l'indirizzo viene calcolato al momento del trasferimento.
Nella famiglia X86 questo modo di funzionamento è possibile solo a partire dal 386. Il valore di "scala" che viene usato può essere solo 2, 4 o 8 o 16, così che il calcolo dell'indirizzo può avvenire con 1, 2, 3 o 4 operazioni di shift, cosa che non dà luogo a particolari penalizzazioni per la velocità.

Funzionamento del compilatore Assembly

A questo punto del testo si conosce abbastanza dell'Assembly per capire come funziona un Assembler.
Un compilatore è un programma per computer, quindi bisogna pensare che non è intelligente. Questo significa che deve funzionare in modo "automatico". Ciò che può fare un programma non intelligente è leggere il testo del file sorgente una lettera alla volta, verificando che corrisponda alle regole del linguaggio sorgente ed effettuando nel contempo la traduzione nel linguaggio oggetto. Dunque il compilatore inizia a scandire tutto il testo del file sorgente, a partire dal primo carattere. Quando trova una parola (token) la confronta con l'elenco delle parole chiave del linguaggio. Se essa è una parola chiave il compilatore deve eseguire il comando che riceve con quella parola, altrimenti deve considerare che il token sia un identificatore definito dall'utente.
Tutti i compilatori, anche quelli dei linguaggi ad alto livello, devono far corrispondere ai simboli definiti dall'utente indirizzi di memoria, ciò viene fatto al loro interno creando una struttura di dati detta "tabella dei simboli". La tabella dei simboli è un oggetto completamente astratto, che esiste solo nel sorgente del programma compilatore.

!!!! to be completed !!!!