L'Assembler 8086

Informazioni generali

M. Rebaudengo - M. Sonza Reorda

Politecnico di Torino
Dip. di Automatica e Informatica

Sommario

- **Introduzione**
- **Pseudo-istruzioni**
- **Operatori**
- **Modi di indirizzamento**
- **Istruzioni**

Introduzione

I programmi Assembler sono composti di:

- ***istruzioni*: generano un'istruzione macchina**
- ***direttive* o *pseudo-istruzioni*: sono comandi per l'assemblatore.**

<u>Esempio</u>

Istruzione:	`ADD`	`AX, 5`
Direttiva:	`VAR1`	`DB ?`

Formato delle istruzioni nel codice sorgente

```
label: mnemonico operando, operando   ;commento
```

dove

`label:` **identifica l'indirizzo di partenza di una istruzione**

`operando:` **in numero variabile da 0 a 2**

`mnemonico:` **identifica una istruzione**

`commento:` **qualsiasi sequenza di caratteri terminata da un fine-riga**

Esempio

```
lab1: mov ax, 5  ; carico ax
```

Formato delle istruzioni nel codice oggetto

Ogni istruzione a livello di codice oggetto corrisponde ad una sequenza di byte in numero variabile tra 1 e 6.

I bit che compongono una istruzione possono essere suddivisi in due gruppi:

- **il *codice operativo*, che specifica l'operazione che deve essere svolta**
- **gli *operandi*, in numero variabile tra 0 e 2; il modo in cui si specifica ove risiede ciascun operando è noto come *modo di indirizzamento*.**

Variabili

Sono utilizzate per identificare in modo simbolico una zona di memoria contenente dati.

Nel codice macchina sono sostituite dall'offset della corrispondente cella di memoria.

All'atto della definizione della variabile, si definiscono

- **il nome**
- **la dimensione**
- **il tipo di dato contenuto (eventualmente)**
- **il valore di inizializzazione (eventualmente).**

Costanti

Nel codice sorgente possono essere specificate usando varie rappresentazioni

- **binarie: 001101B**
- **ottali: 15O, 15Q**
- **esadecimali: 72H, 0DH, 0BEACH (devono iniziare con un numero)**
- **decimali: 13, 13D**
- **stringhe: 'S', 'Ciao'**
- **reali in base 10: 2.345925, 715E-3.**

Nel codice macchina sono sempre e solo rappresentate in binario.

Identificatori

Sono i nomi che possono essere assegnati a istruzioni, variabili, procedure, costanti, segmenti.

Sono così composti:

- **il primo carattere può essere una lettera (a-z, A-Z), oppure uno dei 4 caratteri @ _ $?**
- **gli altri caratteri possono essere una lettera, un numero, o uno dei 4 caratteri sopra.**

Le *parole chiave* non possono fungere da identificatori.

Formato del codice sorgente

- **Il significato del codice è indipendente**
  - **dal fatto che i caratteri siano minuscoli o maiuscoli (*case insensitiveness*)**
  - **dalla presenza di spazi**
- **Ogni istruzione deve occupare una riga: altrimenti la riga precedente deve terminare con \.**

Pseudo-istruzioni

Sono direttive per l'Assemblatore, che non corrispondono a istruzioni macchina nel codice generato.

Le categorie principali di pseudo-istruzioni sono:

- **pseudo-istruzioni per la definizione di variabili**
- **pseudo-istruzioni per la definizione di costanti**
- **pseudo-istruzioni per la gestione dei segmenti.**

Pseudo-istruzioni per la definizione di variabili

Formato:

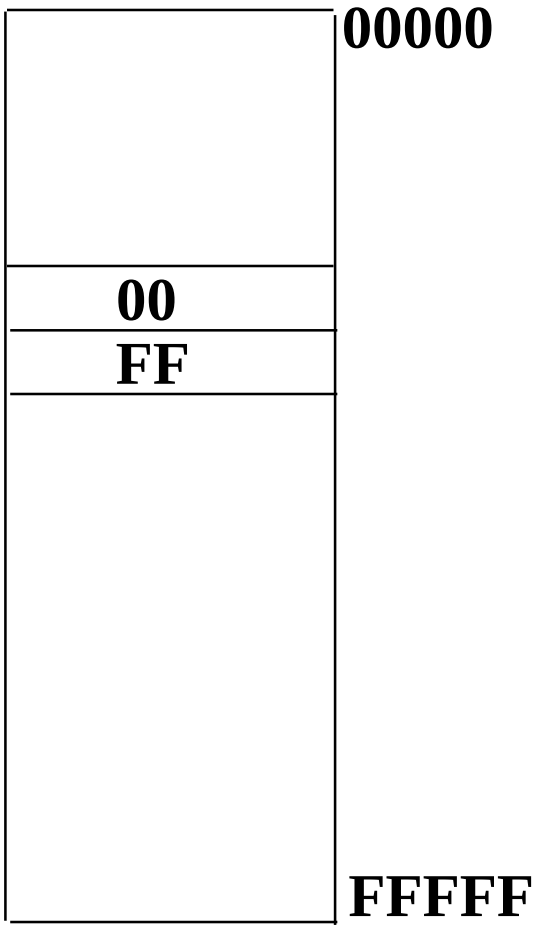
`[nome] direttiva valore`

- ***`direttive`*: `BYTE` (`DB`), `WORD` (`DW`), `DWORD` (`DD`), `QWORD` (`DQ`)**
- ***`valore`* può essere:**
  - **un valore numerico**
  - **una stringa tra apici**
  - **il carattere `?`**
  - **il costrutto `num DUP (val)` che replica `num` volte il valore `val`.**

Rappresentazione delle parole

Si presuppone che i dati memorizzati in una parola abbiano il byte meno significativo memorizzato nel byte con indirizzo minore (*little endian*).

Nell'esempio si vede come sia memorizzato il valore FF00.



Direttiva DB

La direttiva DB permette di definire strutture dati costituite da byte.

È utilizzata per la memorizzazione di:

- **caratteri**
- **stringhe**
- **numeri interi.**

Esempi:

```
CITTA     DB   "T","o","r","i","n","o"
CITTA2    DB   "Torino"
STR2      DB   'Programma perfetto !'
TAB       DB   1, 123, 84, 5 DUP (?), -10
```

Direttiva DW

La direttiva `DW` permette di definire strutture dati costituite da word (2 byte).

È utilizzata per la memorizzazione di:

- **1 o 2 caratteri**
- **numeri interi**
- **offset.**

La direttiva `DW` NON può essere utilizzata per memorizzare una sequenza di caratteri.

Esempi:

```
TAB       DW   1, 350, -4000, 1024
LISTA     DB   100 DUP (?)
LIST_OFF  DW   LISTA
```

Direttiva DW per la memorizzazione di offset

La direttiva DW può essere utilizzata per allocare la memoria necessaria per memorizzare un offset.

Esempio

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

Direttiva DD

La direttiva `DD` permette di definire strutture dati costituite da *doubleword* (4 byte).

È utilizzata per la memorizzazione di:

- **1 o 2 caratteri**
- **numeri interi**
- **indirizzi interi (registro di segmento e offset).**

La direttiva `DD` NON può essere utilizzata per memorizzare una sequenza di caratteri.

Esempi:

```
TAB       DD   125000
LISTA     DB   100 DUP (?)
LIST_ADD  DD   LISTA
```

Direttiva DD per la memorizzazione di indirizzi

La direttiva DD può essere utilizzata per allocare la memoria necessaria per memorizzare un indirizzo (offset+segmento).

Esempio

```
LISTA     DB   100 DUP (?)
LISTADDR  DD   LISTA
```

La memorizzazione avviene con l'offset nella word avente indirizzo minore.

Definizione di costanti

Formato:

simbolo EQU espressione

simbolo = espressione

Definiscono costanti simboliche durante l'assemblaggio.

Le costanti definite con = possono essere cambiate di valore nel corso del programma, a differenza di quelle definite con EQU.

espressione **può essere un'espressione intera, una stringa di 1 o 2 caratteri, o un indirizzo.**

Esempi

```
column  EQU 80
row     EQU 25
screen  EQU colum*row
```

Direttive per la gestione dei segmenti

Permettono la definizione e la gestione dei segmenti.

A partire dalla versione 5.0 sono state introdotte alcune pseudo-istruzioni (`.MODEL`, `.DATA`, `.CODE`, `.STACK`) che semplificano il problema.

Direttiva `.MODEL`

La direttiva `.MODEL` definisce gli attributi di base relativi all'intero modulo sorgente:

- **modello di memoria**
- **convenzione dei nomi**
- **sistema operativo (`DOS` o `WINDOWS`)**
- **tipo di stack.**

Direttiva `.MODEL`

Formato

```
.MODEL    modello [opzioni, ...]
```

Uso

Il `modello` è obbligatorio e definisce le dimensioni dei segmenti di codice e di dato.

Le opzioni possibili sono relative a:

- **convenzioni dei nomi e delle chiamate per procedure e simboli pubblici**
- **tipo di *stack*.**

Modelli di memoria

Il MASM (Assemblatore Microsoft) e molti ambienti di sviluppo supportano i modelli di memoria standard usati dai linguaggi di alto livello Microsoft.

I possibili modelli di memoria sono i seguenti:

	Default Code	Default Data	Data e Code Combinati
TINY	Near	Near	Sì
SMALL	Near	Near	No
MEDIUM	Far	Near	No
COMPACT	Near	Far	No
LARGE	Far	Far	No
HUGE	Far	Far	No

Tipi di stack

Nel comando `.MODEL` è possibile specificare il tipo di stack:

- **`NEARSTACK` (default): il segmento di stack ed il segmento di dato sono all'interno dello stesso segmento fisico (`DS` ed `SS` coincidono).**
- **`FARSTACK`: i segmenti di dato e di stack sono distinti.**

Linguaggio

All'interno della direttiva `.MODEL` è possibile specificare un'opzione che garantisce la compatibilità con linguaggi di alto livello.

Le possibili opzioni sono:

- **`PASCAL`**
- **`BASIC`**
- **`C`**
- **`FORTRAN`.**

Creazione dei segmenti di dato

I programmi possono contenere sia dati di tipo `NEAR` sia dati di tipo `FAR`.

In generale i dati più importanti e più frequentemente usati sono memorizzati in un'area dati di tipo `NEAR`, dove l'accesso è più veloce, mentre i dati meno frequentemente usati sono memorizzati in segmenti di dato di tipo `FAR`.

La direttiva `.DATA` crea un segmento di dato di tipo `NEAR`.

La direttiva `.FARDATA` crea un segmento di dato di tipo `FAR`.
La direttiva `.STARTUP` in un modello di memoria che prevede più segmenti di dato NON inizializza i registri `DS` ed `ES`. Tale operazione deve essere fatta dal programmatore.

Esempio

```
                .MODEL        compact
                .STACK        2048
                .FARDATA     segm1
vett1       DB 100 DUP (?)
                .FARDATA     segm2
vett2       DB 100 DUP (?)
                .CODE
ASSUME DS:segm1, ES:segm2
                .STARTUP
                MOV   AX, segm1
                MOV DS, AX
                MOV AX, segm2
                MOV ES, AX
                ...
                .EXIT
                END
```

Creazione dei segmenti di codice

Nei modelli di memoria `tiny`, `small` e `compact` il segmento di codice è di tipo `NEAR`: più segmenti di codice su più moduli sono mappati su uno stesso segmento fisico di codice.

Nei modelli di memoria `medium` e `large` il segmento di codice è di tipo `FAR`: ogni direttiva `.CODE` corrisponde ad un segmento di codice fisico diverso.

In uno stesso modulo è possibile creare più segmenti di codice utilizzando diverse direttive `.CODE` e assegnando un nome ad ogni singolo segmento.

Direttiva END

Formato

```
END {etichetta}
```

Significato

Conclude un modulo di programma.

Se il modulo contiene la prima istruzione del programma, la relativa etichetta deve essere specificata come operando.

In tal modo l'assemblatore, il linker ed il loader possono comunicare al processore l'istruzione da cui iniziare l'esecuzione del programma.

Operatori

Si distinguono in

- **operatori per il calcolo degli attributi di una variabile**

  (`TYPE, LENGTH, SIZE, SEG, OFFSET`)

- **operatori aritmetici, logici e relazionali**

  (`+, -, *, /, MOD, AND, OR, NOT, XOR, EQ, GT,` etc.)

- **operatori che modificano il tipo di una variabile (`PTR`).**

Gli operatori vengono gestiti dall'assemblatore e permettono di esprimere in modo più leggibile le costanti.

Operatore OFFSET

Formato:

OFFSET variabile

Funzionamento:

L'operatore OFFSET restituisce il valore dell'offset di una variabile.

Applicazione:

L'operatore OFFSET può essere usato in alternativa all'istruzione LEA.

Esempio:

Le due seguenti istruzioni sono equivalenti:

```
MOV     AX, OFFSET VAR
LEA     AX, VAR
```

Limiti dell'operatore `OFFSET`

L'operatore `OFFSET` può essere applicato solo ad operandi indirizzati direttamente attraverso un nome di variabile e non ad operandi indirizzati indirettamente.

Esempio

```
MOV  AX, OFFSET VAR[SI] ;Errore !!
```

Si possono utilizzare le seguenti istruzioni:

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

oppure:

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

Operatori TYPE, LENGTH e SIZE

Formato:

```
TYPE        variabile
LENGTH      variabile
SIZE        variabile
```

Funzionamento:

L'operatore `TYPE` restituisce il numero di byte dell'operando.

L'operatore `LENGHT` restituisce il numero di unità allocate per l'operando.

L'operatore `SIZE` restituisce lo spazio di memoria utilizzato dall'operando.

```
SIZE = LENGHT * TYPE
```

Operatore LENGTH

L'operatore LENGTH ha senso solo per variabili allocate attraverso l'operatore DUP. In tutti gli altri casi restituisce infatti il valore 1.

Esempio

```
EXP         DW          5 DUP (?)
EXP2        DW          1, 2, 3, 4, 5
            MOV         AX, LENGTH EXP
            MOV         BX, TYPE EXP
            MOV         CX, SIZE EXP
            MOV         DX, LENGTH EXP2
```

Dopo queste istruzioni in AX c'è 5, in BX c'è 2, in CX c'è 10 ed in DX c'è 1.

Operatore SEG

Formato:

SEG ***variabile***

Funzionamento:

L'operatore `SEG` restituisce l'indirizzo di inizio del segmento a cui appartiene l'operando *`variabile`*.

Esempio:

```
LEA     SI, STR
MOV     AX, SEG STR
MOV     DS, AX
```

è equivalente a:

```
LDS     SI, STR
```

Operatori aritmetici, logici e relazionali

- **aritmetici: +, -, *, /, MOD**
- **logici: AND, OR, NOT, XOR**
- **relazionali: EQ, NE, LT, LE, GT, GE**

Gli operatori possono comparire esclusivamente in espressioni valutabili al tempo di assemblaggio.

Operatore PTR

Formato:

tipo PTR nome

Funzionamento:

L'operatore PTR forza l'assemblatore a modificare per l'istruzione corrente il *tipo* del dato avente come identificatore *nome*.

Esempio:

```
            .DATA
TOT         DW          ?
            .CODE
            MOV         BH, BYTE PTR TOT
            MOV         CH, BYTE PTR TOT+1
```

Operatore PTR (II)

Esempio

```
            .DATA
COPPIA      DB   2 DUP (?)
            .CODE
            MOV  AX, COPPIA                    ;ERRORE!
            MOV  AX, WORD PTR COPPIA
```

Esempio

```
            INC  [BX]
```

La cella da incrementare corrisponde ad una word o ad un byte?

L'assemblatore non può saperlo e genera errore.

Soluzione:

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

Modi di indirizzamento

Il *modo di indirizzamento* specifica l'operando di una istruzione.

Gli operandi possono essere contenuti:

- **in registri**
- **nell'istruzione stessa**
- **in memoria**
- **su una porta di I/O.**

Modi di indirizzamento (II)

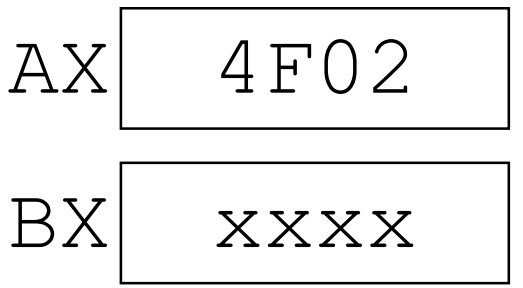
I modi di indirizzamento sono i seguenti:

- **Register**
- **Immediate**
- **Direct**
- **Register Indirect**
- **Base Relative**
- **Direct Indexed**
- **Base Indexed.**

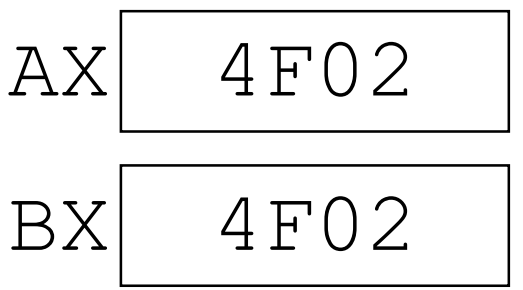
Register Addressing

Nell'istruzione è specificato il registro da utilizzare come operando.

Esempio




```
MOV BX, AX
```



Immediate Addressing

Nell'istruzione stessa è indicato il dato da usare come operando.

Esempio

BH | xxxx

```
MOV BH, 07h
```

BH | 07

Note

- **L'operando può anche essere un simbolo definito con una pseudo-istruzione EQU.**

  Esempio

```
K EQU   1024
    :
MOV   CX, K
```

- **il dato indicato nell'istruzione viene trasformato dall'assemblatore in formato binario su 8 o 16 bit, e scritto nel campo opportuno del codice macchina.**

Direct Addressing

Nell'istruzione sorgente è contenuto l'*identificatore* di una variabile. Nel codice macchina comparirà l'Effective Address della parola di memoria da utilizzare come operando.

Alla variabile può essere sommato o sottratto un *displacement* attraverso gli operatori + e -.

Una notazione equivalente prevede l'utilizzo delle parentesi quadre per racchiudere il displacement o l'identificatore della variabile.

L'indirizzo fisico è ottenuto sommando l'EA con il contenuto di DS.

Direct Addressing

(segue)

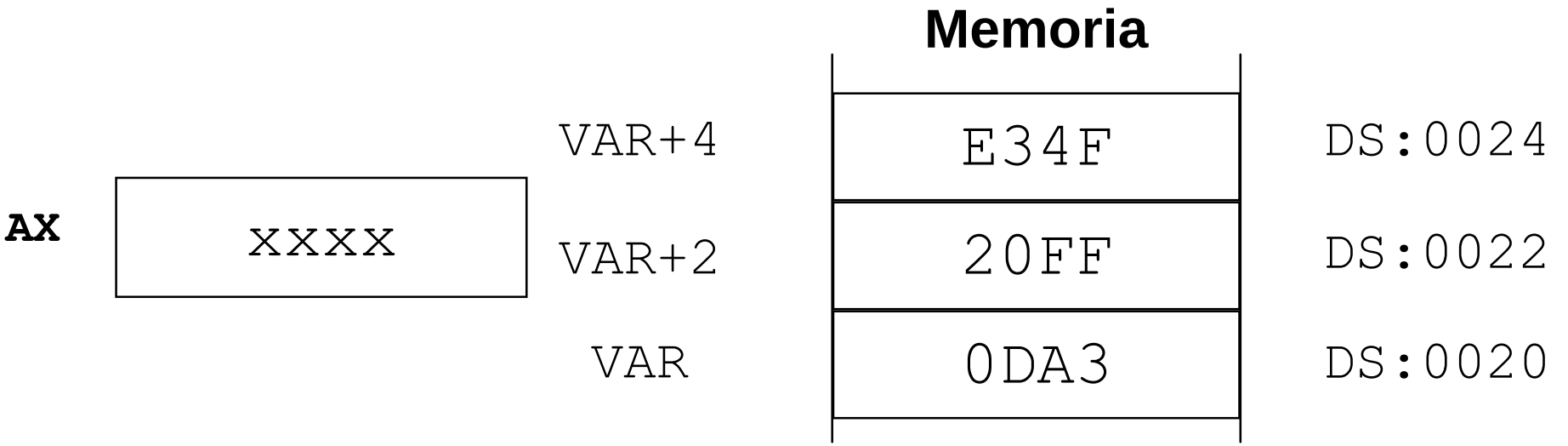
L'indirizzo fisico è ottenuto sommando l'EA con il contenuto di `DS`.

Formalismo:

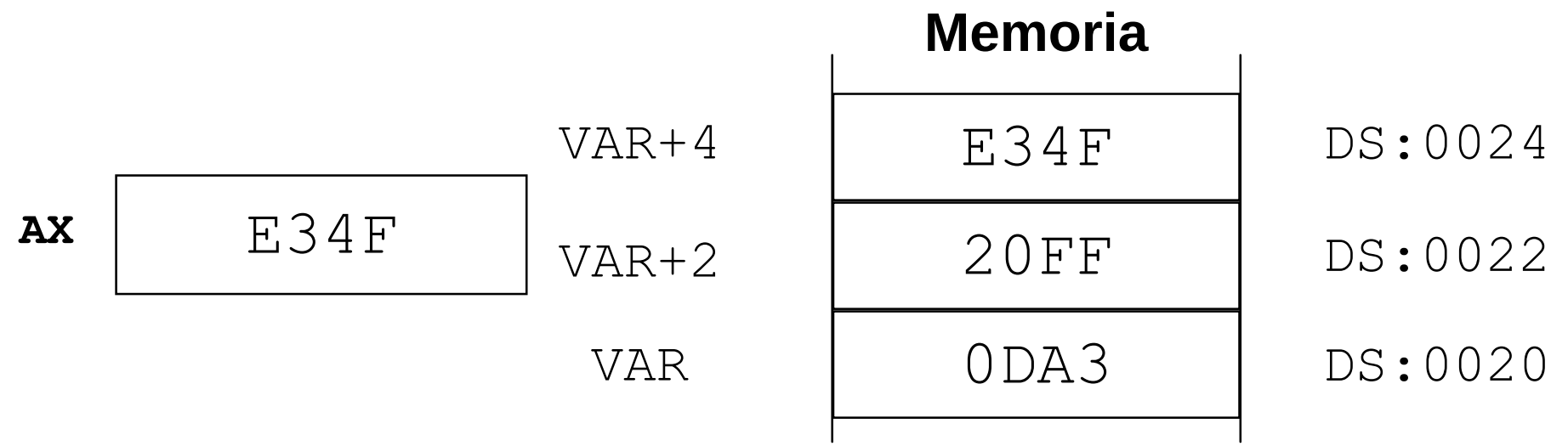
```
nome
[nome]
nome+displacement
nome[displacement]
nome-displacement
nome[-displacement]
```

Esempi:

```
MOV AX, TABLE
MOV AX, TABLE+2
MOV AX, TABLE[2]
```


Memoria
AX
xxxx
VAR+4
E34F
DS:0024
VAR+2
20FF
DS:0022
VAR
0DA3
DS:0020

```
MOV AX, VAR[4]
```


Memoria
AX
E34F
VAR+4
E34F
DS:0024
VAR+2
20FF
DS:0022
VAR
0DA3
DS:0020

Segment Override

Nel modo di indirizzamento diretto il registro di segmento di default è il registro `DS`.

L'operatore di segment override (`:`) permette di utilizzare per il calcolo dell'indirizzo fisico un registro di segmento diverso da quello di *default*.

<u>Esempi</u>

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

Register Indirect Addressing

L'Effective Address dell'operando è contenuto in uno dei seguenti registri:

- **Base (BX)**
- **Index Register (DI oppure SI)**
- **Base Pointer (BP).**

Viene detto *Indirect* perchè nell'istruzione è indicato dove trovare l'indirizzo dell'operando.

<u>Esempio</u>

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

Registri di segmento

Ciascun registro è abbinato ad un particolare registro di segmento:

`DS` ⇒ `BX`

`DS` ⇒ `SI`

`DS` ⇒ `DI`

(tranne per le istruzioni di manipolazione di stringhe in cui il registro di segmento è `ES`).

`SS` ⇒ `BP`.

Esempio

Codice per il trasferimento di un vettore in un altro usando il Register Indirect Addressing.

```
                :
        MOV SI, OFFSET SOURCE_VECT
        MOV DI, OFFSET DEST_VECT
        MOV CX, LENGTH SOURCE_VECT
QUI:    MOV AX, [SI]
        MOV [DI], AX
        ADD SI, 2
        ADD DI, 2
        DEC   CX
        CMP   CX, 0
        JNZ   QUI
                :
```

Base Relative Addressing

L'Effective Address dell'operando è calcolato sommando il contenuto di uno dei Base Register (BX o BP) ad un *displacement* rappresentato da una costante presente nell'istruzione stessa.

Formato assembler:

```
[ <register> ] + <constant>
```

Esempio

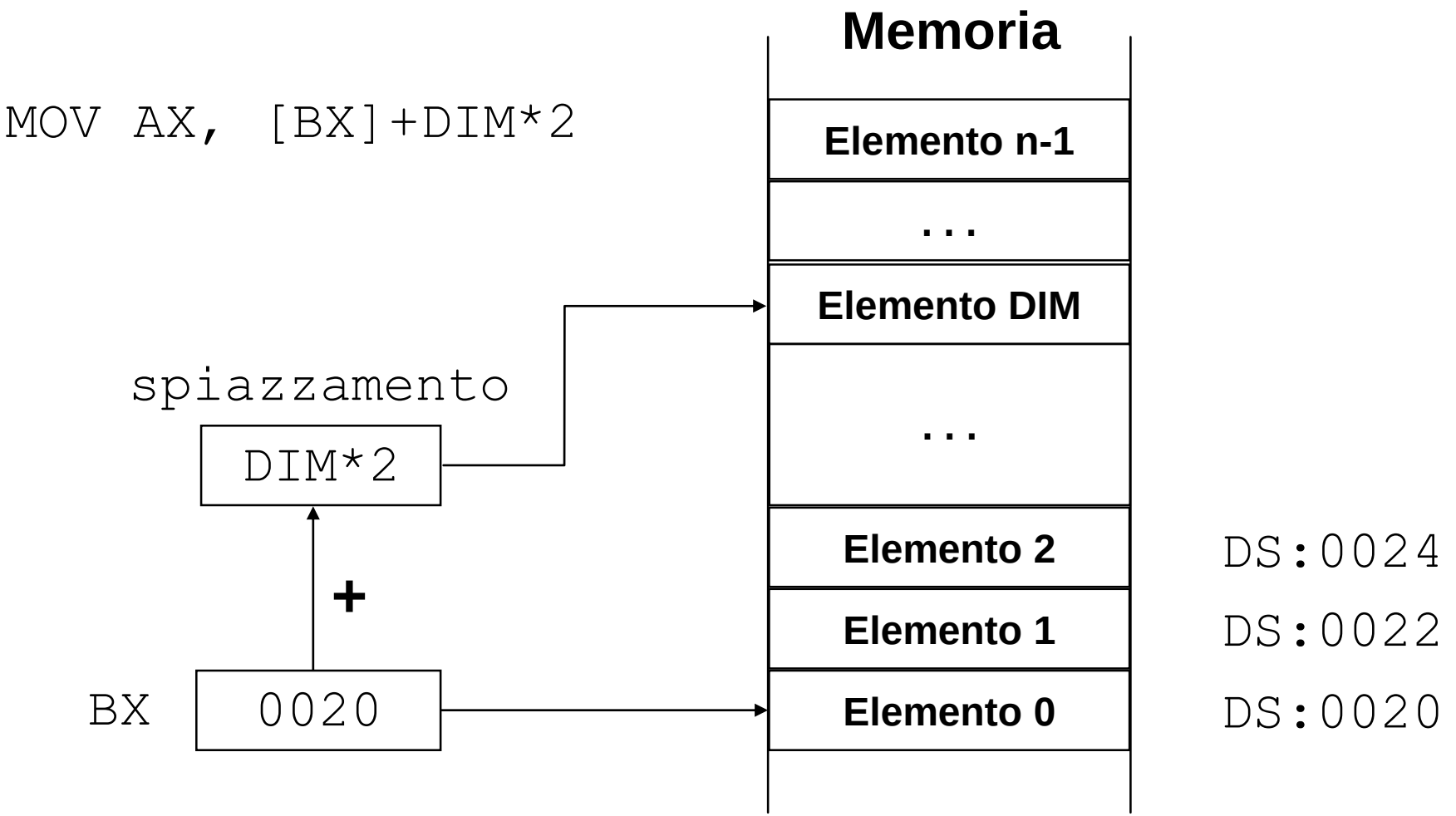
```
MOV AX, [BX]+4
```

Nota

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

sono equivalenti.

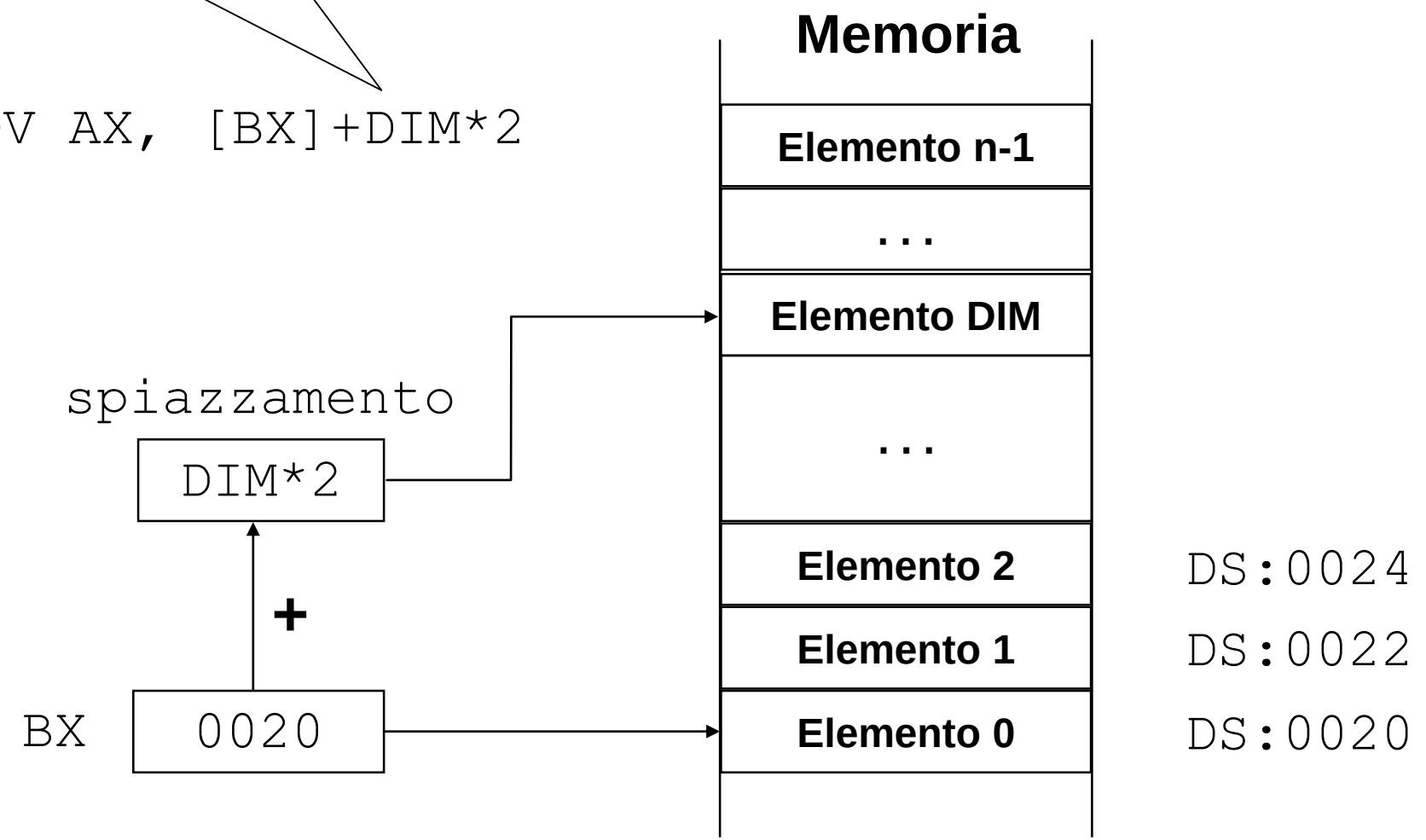
Calcolo dell'EA per un indirizzamento *Base Relative*



Calcolo dell'EA per un spiazzamento *Base Relative*

DIM è una costante il cui valore è noto al tempo di compilazione

```
MOV AX, [BX]+DIM*2
```

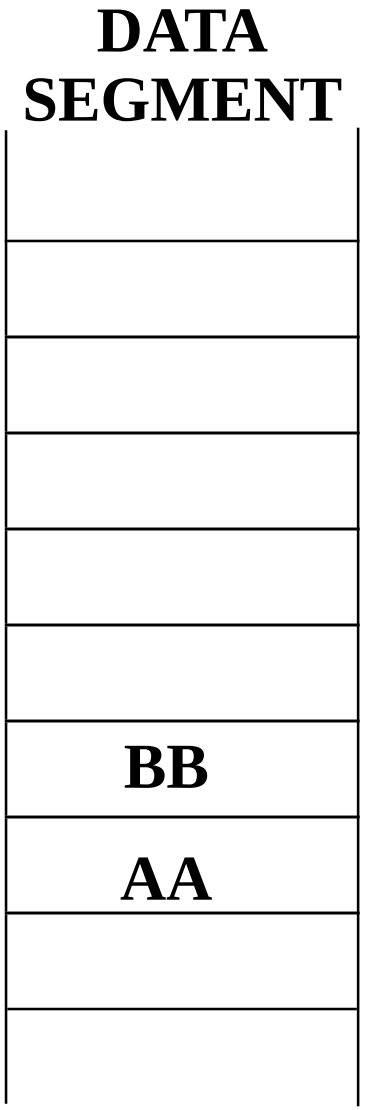


Esempio

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

BX | 001A

AX | AABB



Direct Indexed Addressing

L'Effective Addressing dell'operando è calcolato sommando il valore di un *offset* (corrispondente ad una variabile) al contenuto di un *displacement* (contenuto in uno degli Index Register SI o DI).

Formato

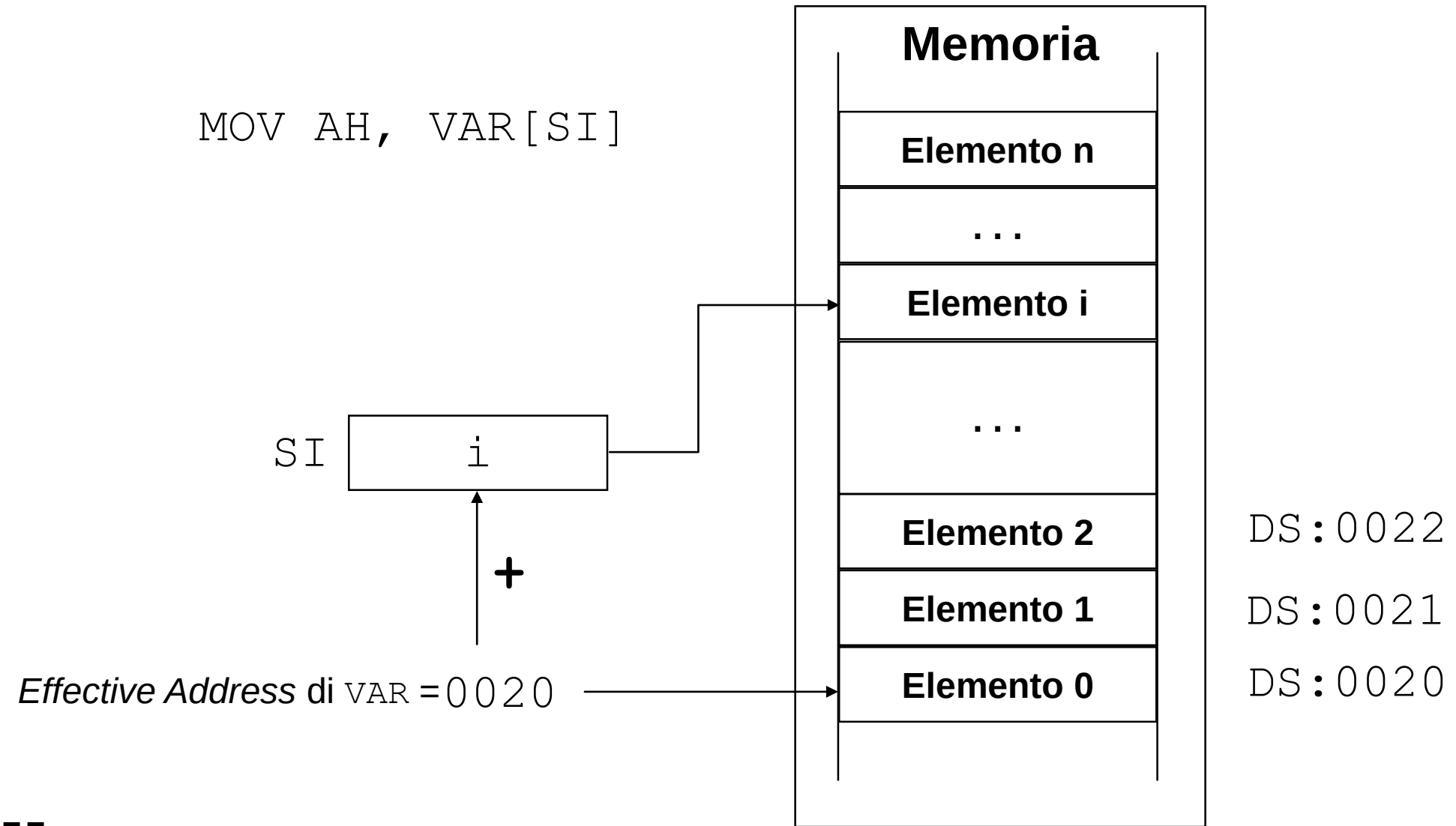
```
<variable> [SI]

<variable> [DI]
```

Esempio

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

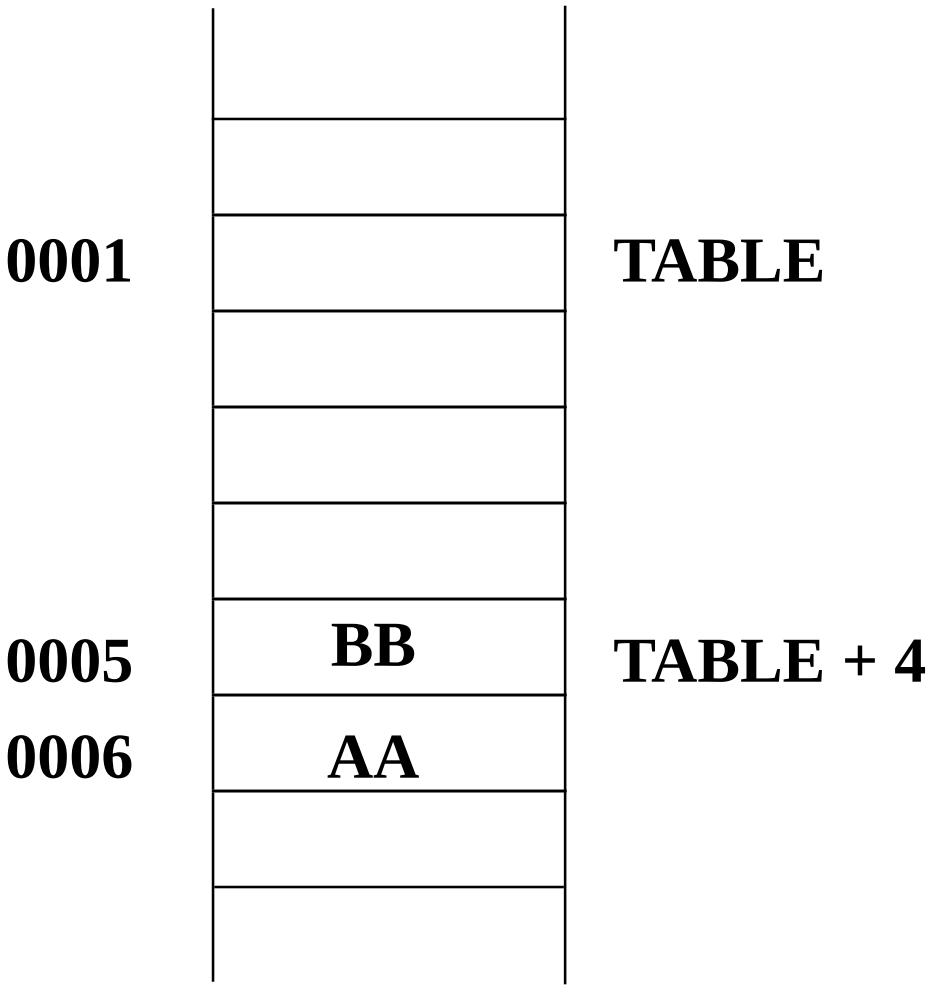
Calcolo dell'EA per un indirizzamento *Direct Indexed*



Esempio

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

DI	0004
AX	AABB



Esempio

Codice per il trasferimento di un vettore in un altro usando il Direct Indexed Addressing.

```
                :
            MOV   SI, 0
            MOV   CX, 100
QUI:        MOV   AX, SOURCE_VECT [SI]
            MOV   DEST_VECT [SI], AX
            ADD   SI, 2
            DEC   CX
            CMP   CX, 0
            JNZ   QUI
                :
```

Base Indexed Addressing

L'Effective Address dell'operando è calcolato come somma dei seguenti termini:

- **contenuto di uno dei *Base Register* (BX o BP)**
- **contenuto di uno degli *Index Register* (SI o DI)**
- **un campo opzionale *displacement* che può essere un identificatore di variabile oppure una costante numerica.**

Base Indexed Addressing

(segue)

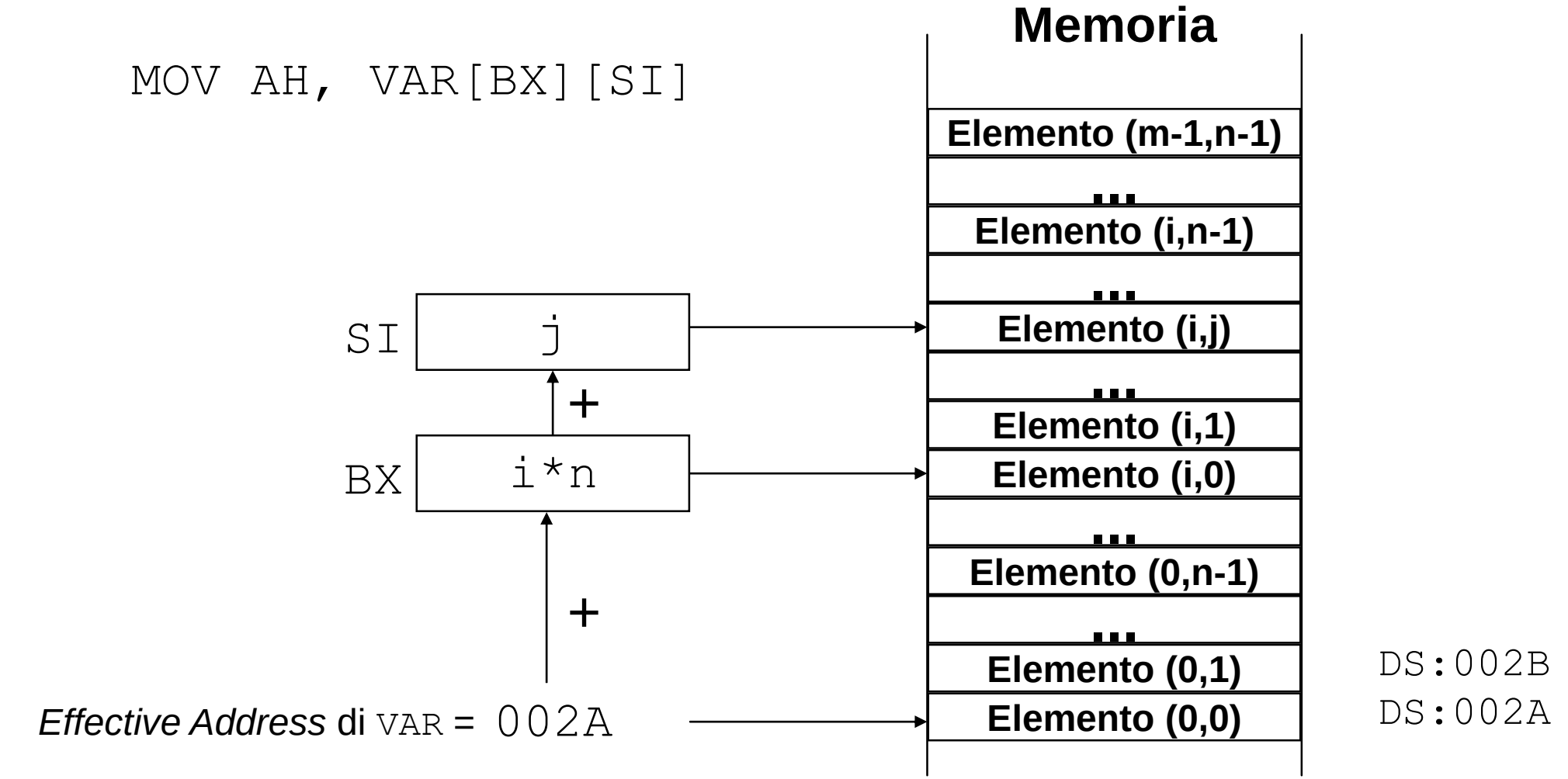
Formato

```
<variable> [BX] + [SI]
<variable> [BX] + [DI]
<variable> [BP] + [SI]
<variable> [BP] + [DI]
```

Esempio

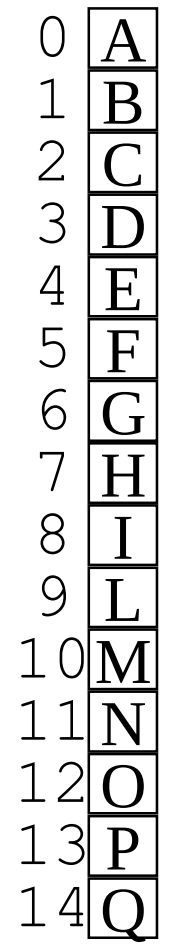
```
MOV AX, TAB[BX][DI]+6
```

Calcolo dell'EA per un indirizzamento *Base Indexed*



Memorizzazione di una matrice

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



Memorizzazione per righe

Copia di una riga in una matrice di dati

Specifiche:

Date due matrici `SORG` e `DEST` di dimensione 4x5, si deve copiare la quarta riga da `SORG` a `DEST`.

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

Soluzione Assembler

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

Istruzioni

L'Assembler 8086 rende disponibili 92 istruzioni, raggruppabili nelle seguenti classi:

- **Trasferimento dati**
- **Aritmetiche**
- **Manipolazione di bit**
- **Control Transfer**
- **Manipolazione di stringhe**
- **Interrupt Handling**
- **Process Control.**