LINGUAGGIO ASSEMBLER PER 8086/8088

Linguaggio Macchina

- insieme di istruzioni che un elaboratore è in grado di eseguire direttamente
- strettamente correlato alla realizzazione fisica dell'elaboratore

Esempio di possibile istruzione macchina:

"Copia il valore della variabile alfa nel registro BX"

- Il campo Op-code specifica il tipo di operazione richiesta (es., "copia")
- Il campo Dest specifica l'operando di destinazione (es., con 3 bit si individua uno degli 8 registri generali)
- Il campo Source specifica l'operando sorgente (es., l'indirizzo di alfa relativo al segmento dati)

Per scrivere un codice in linguaggio macchina è necessario:

- conoscere l'architettura della macchina
- ragionare in termini del comportamento dell'elaboratore
- conoscere i dettagli relativi alla scrittura delle singole istruzioni:
  - codici numerici e formato interno delle istruzioni macchina
  - rappresentazione degli operandi
- gestire direttamente gli indirizzi in memoria per il riferimento ai dati e per i salti al codice

Ad esempio, per scrivere nel linguaggio macchina dell'8086 l'istruzione:

MOV destinazione, sorgente

è necessario scegliere tra:

- **14 diversi codici operativi, in funzione del tipo degli operandi**
- **svariate possibili rappresentazioni degli operandi stessi**

L'istruzione macchina che si ottiene ha una lunghezza variabile tra i **due e i sei byte**, a seconda delle scelte effettuate.

Linguaggio Assembler

- insieme di istruzioni di tipo elementare
- costringe ancora il programmatore a ragionare in termini della logica della macchina a cui si riferisce
- **risulta più ad alto livello rispetto al linguaggio macchina**
- nasconde i dettagli realizzativi delle singole istruzioni (codici operativi, formati, ecc.)
- associa ai dati e alle istruzioni **nomi simbolici che identificano in modo univoco le corrispondenti posizioni di memoria** (eliminando, così, la necessità di utilizzare indirizzi espliciti).

Istruzioni eseguibili - a cui corrispondono le istruzioni del linguaggio macchina

Direttive (o pseudoistruzioni) - controllano il comportamento dell'assemblatore in fase di traduzione - facilitano lo sviluppo dei programmi - permettono:
- la suddivisione di un'applicazione in più moduli
- il controllo del formato dei listati
- la definizione di macro, ecc.

L'architettura della macchina a cui il linguaggio si riferisce non viene nascosta in alcun modo

Ad esempio, l'assembler consente di riferire direttamente i registri, azione preclusa dai linguaggi di alto livello a causa della loro indipendenza dall'hardware.

Basta pensare al C……

Formato delle istruzioni

[label] istruzione/direttiva [operando/i] [; commento]

- **label** consente di dare un **nome simbolico** (da utilizzare come operando in altre istruzioni) a **variabili di memoria, valori, singole istruzioni, procedure**
- **istruzione/direttiva** è lo **mnemonico per un'istruzione o una direttiva**: individua il tipo di operazione da eseguire e il numero e il tipo degli operandi (la semantica dell'istruzione)
- **operando/i** è una combinazione di nessuna, una o più **costanti**, **riferimenti a registri** o **riferimenti alla memoria**;
  se un'istruzione ammette due operandi:
  - il primo è l'operando destinazione
  - il secondo è l'operando sorgente
  ad esempio, MOV AX, CX copia il contenuto del registro CX nel registro AX
- **commento** consente di rendere più leggibile il programma

E' indifferente l'uso di lettere maiuscole o minuscole.

Il tipo di operandi ammessi varia da istruzione a istruzione.

Esistono istruzioni che ammettono come operando solo una costante o solo un particolare registro generale.

L'assembler dell'8086 non ha la caratteristica dell'ortogonalità, caratteristica che renderebbe la fase di apprendimento dell'assembler più veloce.

Nomi simbolici

- caratteri ammessi per i nomi simbolici:
  A-Z a-z 0-9 _ $?
- il primo carattere di un nome non può essere un digit (0-9)
- ogni nome **deve essere univoco** (in genere, all'interno del modulo in cui viene definito)
- **non deve coincidere con una parola riservata** (ad esempio, il nome di un registro o di un operatore)

Costanti numeriche

- di default, tutti i valori numerici sono espressi **in base dieci**

- è possibile esprimere le costanti numeriche:
  - **in base 16** (esadecimale) mediante il suffisso **H** (il primo digit deve però essere numerico)
  - **in base 8** (octal) mediante il suffisso **O**
  - **in base 2** (binario) mediante il suffisso **B**

Ad esempio:

0FFh
0h
777O
11001B

3Fh
FFh errata !
778O errata !

DIRETTIVE

Istruzioni di tipo dichiarativo, che forniscono informazioni agli strumenti di sviluppo dei programmi (nel caso del Turbo Assembler, - l'assemblatore TASM e il linker TLINK).

Direttive di segmento (standard)

Permettono di controllare in modo completo la definizione dei vari tipi di segmenti.

La definizione di ogni segmento deve iniziare con una **direttiva SEGMENT** e deve terminare con una **direttiva ENDS** (end segment):

```
nomeSeg SEGMENT

contenuto del segmento

nomeSeg ENDS
```

La direttiva SEGMENT definisce l'inizio di un segmento.

nome SEGMENT [allineamento] [comb] ['classe']

- nome - **nome simbolico del segmento** - se è già stato definito un segmento con lo stesso nome, il segmento corrente è la continuazione del precedente
- allineamento - **tipo di allineamento del segmento nella memoria fisica** - il default è **PARA** (paragrafo, allineato ai 16 bit)
- comb - controlla le modalità di collegamento di segmenti con lo stesso nome, ma in moduli diversi (solo per programmi multi-modulo); la combinazione **STACK** permette di impostare SS:SP alla fine del segmento stack al momento dell'esecuzione del programma (n.b. lo stack viene utilizzato riempiendolo al contrario dagli indirizzi alti a quelli bassi).

Rispetto alla uguaglianza del nome si comporta:

- PRIVATE, non li combina (DEFAULT)
- PUBLIC, li combina e concatena
- STACK, concatena e crea lo stack
- COMMKON, li sovrappone
- MEMORY, sinonimo di PUBLIC
- AT *address*, allocazione precisa a un indirizzo

- classe viene usato per controllare l'ordine in cui i segmenti vengono collegati: **tutti i segmenti di una determinata classe vengono collocati in un blocco contiguo di memoria, indipendentemente dalla loro posizione nel codice sorgente -** utilizzare:
  CODE per i segmenti che contengono codice
  DATA per i segmenti che contengono dati
  STACK per i segmenti che contengono lo stack

Comportamento del linker durante la compilazione, ordinamento dei segmenti nelle classi:

.ALPHA (alfabetico)

.SEQ (sequenziale, default)

La direttiva ENDS definisce la fine di un segmento.

```
nome  ENDS
```

nome è il nome simbolico del segmento.

Esempio - Per creare un programma con tre segmenti (stack, dati e codice), si scriverà:

```
_Stack  SEGMENT  STACK  'STACK'
        dimensionamento dello stack
_Stack  ENDS


_Data   SEGMENT  'DATA'
        definizione dei dati del programma
_Data   ENDS


_Code   SEGMENT  'CODE'
        ASSUME CS:_Code, SS:_Stack
        ASSUME ES:NOTHING
Start:
        MOV  AX, _Data
        MOV  DS, AX
        ASSUME DS:_Data
        . . .
_Code   ENDS
```

Direttive per la definizione di procedure

La definizione di ogni procedura deve iniziare con una direttiva PROC e deve terminare con una direttiva ENDP (end procedure).

La direttiva PROC definisce l'inizio di una procedura.

```
nome  PROC [distanza]
```

- **nome** è il **nome simbolico della procedura** (da utilizzare nelle chiamate alla procedura con la istruzione CALL)
- **distanza** è:
  - **NEAR** - la procedura può essere chiamata solo all'interno del segmento in cui è stata definita (default)
  - **FAR** - la procedura può essere chiamata da qualsiasi segmento

Cambia molto l'indirizzamento nei due casi!!!!

La direttiva ENDP (end procedure) definisce la fine di una procedura.

```
nome  ENDP
```

nome è il nome simbolico della procedura.

Ad esempio, per definire la procedura FAR di nome FarProc, si scrive:

```
FarProc    PROC  FAR

           . . .

FarProc    ENDP
```

mentre per definire la procedura NEAR di nome NearProc, si scriverà:

```
NearProc   PROC  NEAR

           . . .

NearProc   ENDP
```

Direttive per la definizione dei dati

Permettono di definire:

- **il nome**
- **il tipo**
- **il contenuto**

delle variabili in memoria.

[nome] tipo espressione [, espressione] ...

- **nome** - **nome simbolico del dato**
- **tipo** - **lunghezza del dato** (se scalare) o di ogni elemento del dato (se array) - i tipi più utilizzati sono:

  DB riserva uno o più byte (8 bit)

  DW riserva una o più word (16 bit)

  DD riserva una o più doubleword (32 bit)

- **espressione** - **contenuto iniziale del dato**:
  - un' espressione costante
  - una stringa di caratteri (solo DB; char=byte)
  - un punto di domanda (nessuna inizializzazione)
  - un'espressione che fornisce un indirizzo (solo DW e DD, near o far address)
  - un'espressione **DUP**licata

```
ByteVar      DB  0           ;  1 byte inizializzato a 0
ByteArray    DB  1,2,3,4     ;  array di 4 byte
String       DB  '8','0','8','6'  ;  array di 4 caratteri
String       DB  '8086'      ;  equivale al
                             precedente
Titolo       DB  'Titolo',0dh,0ah ; ovvero 13 e 10
                 ;  stringa che contiene anche una
                 ;  coppia di caratteri CR/LF
Zeros        DB  256 dup (0)
                 ;  array di 256 byte inizializzati a
                 0
Tabella      DB  50 dup (?)
                 ;  array di 50 byte non inizializzati
WordVar      DW  100*50      ;  scalare di una word
Matrix       DW  1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0
                 ;  array di 16 word
Matrix       DW  4 dup (1, 3 dup (0))
                 ;  equivale al precedente
NearPointer  DW  Matrix
                 ;  contiene l'offset di Matrix
DoubleVar    DD  ? ;  scalare di una doubleword
FarPointer   DD  Matrix
                 ;  contiene l'offset e l'indirizzo del
                 ;  segmento di Matrix
```

MODALITA' DI INDIRIZZAMENTO DEGLI OPERANDI

L'**operando** di un'istruzione può essere:

- in un **registro**
- nell'**istruzione** stessa - operando immediato
- in **memoria**

L'**indirizzo** di un operando in memoria può essere ottenuto in molti modi diversi.

L'**offset** di un operando in memoria viene calcolato dall'EU in funzione della modalità di indirizzamento utilizzata nell'istruzione e viene detto **indirizzo effettivo dell'operando Effective Address** o **EA**

Operando Registro

E' l'indirizzamento più **compatto e veloce**

L'operando è già nella CPU e quindi non è necessario accedere alla memoria

Il registro può essere a 8 o a 16 bit

```
MOV  AX, CX ;  AX <- CX (16 bit)

MOV  AL, BL  ;  AL <- BL (8 bit)
```

Operando Immediato

L'operando è contenuto nell'istruzione

L'accesso all'operando è **veloce**
l'operando è nella instruction queue

Il dato può essere una costante di 8 o 16 bit

```
MOV  CX, 100          ;  CX <- 100 (16 bit)

MOV  AL, 12h;  AL <- 12h (8 bit)
```

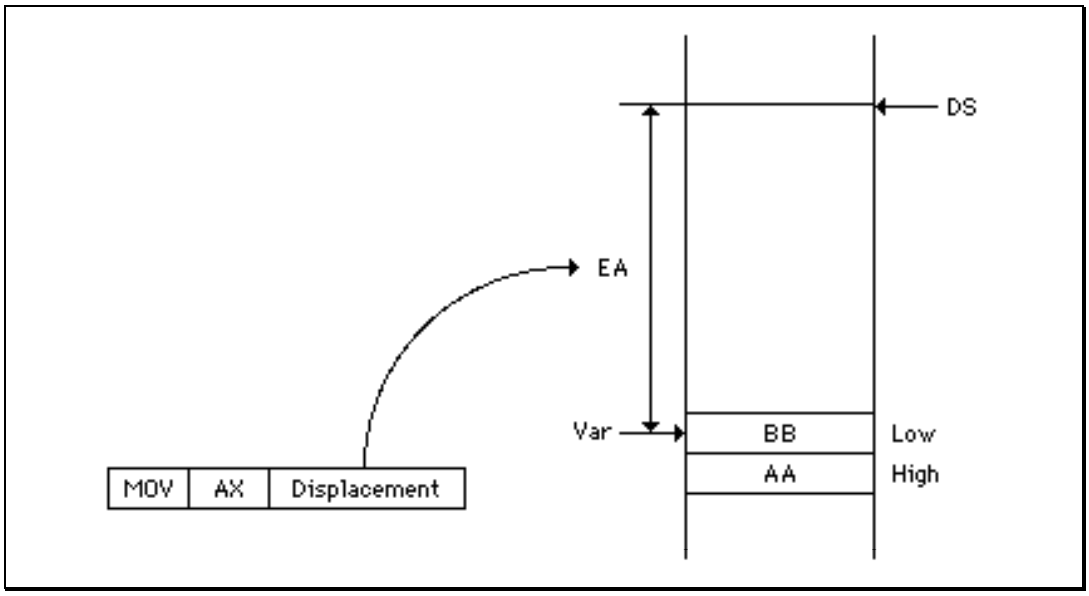
il valore 100 è memorizzato all'interno dell'istruzione in 2 byte e 12h in 1 byte (100 va messo in CX, azzero la parte alta di CX)

Indirizzamento Diretto

E' l'indirizzamento in memoria più semplice

L'EA è contenuto nel campo displacement dell'istruzione - un valore senza segno di 8 o 16 bit

```
Var     DW     ?
        . . .
        MOV  AX, Var;  AX <- Var (16 bit)
```



AH <- AA
AL <- DD

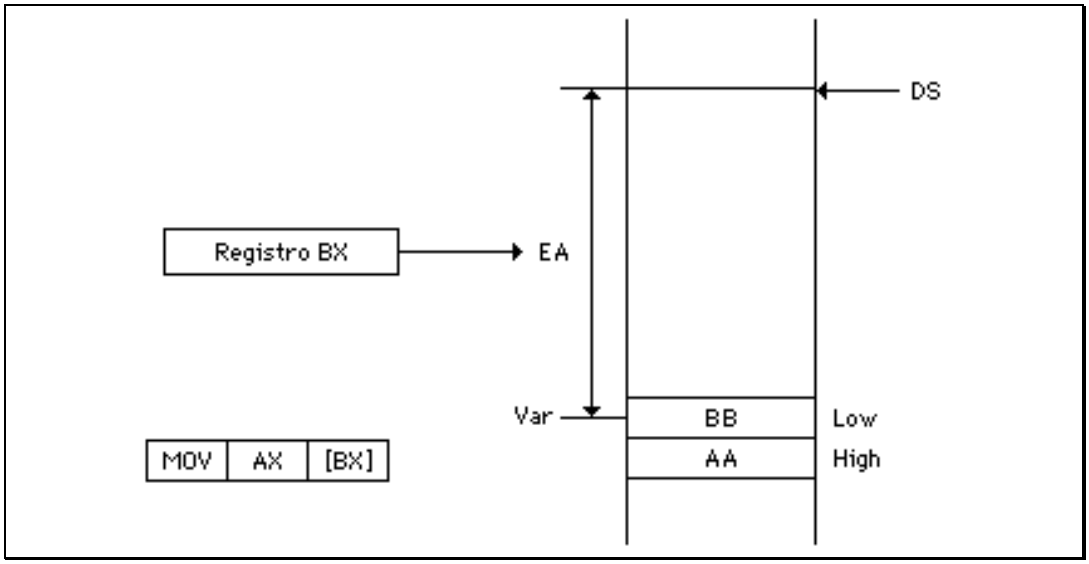
Indirizzamento Indiretto mediante Registro

L'EA dell'operando è contenuto in un registro base o indice (BX, BP, DI, SI)

Se si aggiorna in modo opportuno il contenuto del registro, la stessa istruzione può agire su più locazioni di memoria

Con le istruzioni JMP e CALL è possibile utilizzare un qualsiasi registro generale a 16 bit

```
MOV  BX, OFFSET Var  ;  BX <- offset di Var
MOV  AX, [BX]        ;  AX <- [BX] (16 bit)
```



Indirizzamento mediante Registro Base

L'EA si ottiene come somma

- di un valore di displacement
- del contenuto del registri BX o BP

Attenzione: utilizzando il registro BP, il segmento di default non è più DS, bensì SS.

Questa modalità di indirizzamento è utile per **indirizzare campi in una struttura dati** dove
- ogni campo ha un displacement fisso
- l'indirizzo di partenza della struttura viene memorizzato nel registro base

per operare su strutture identiche è sufficiente modificare il contenuto del registro base

```
typedef struct
        {
                int giorno,mese,anno;
        } dataType;

dataType d1,d2,d3;
```

d1, d2 e d3 hanno indirizzi differenti
d1.anno, d2.anno, d3.anno hanno lo stesso displacement (4 se sizeof(int)==2)

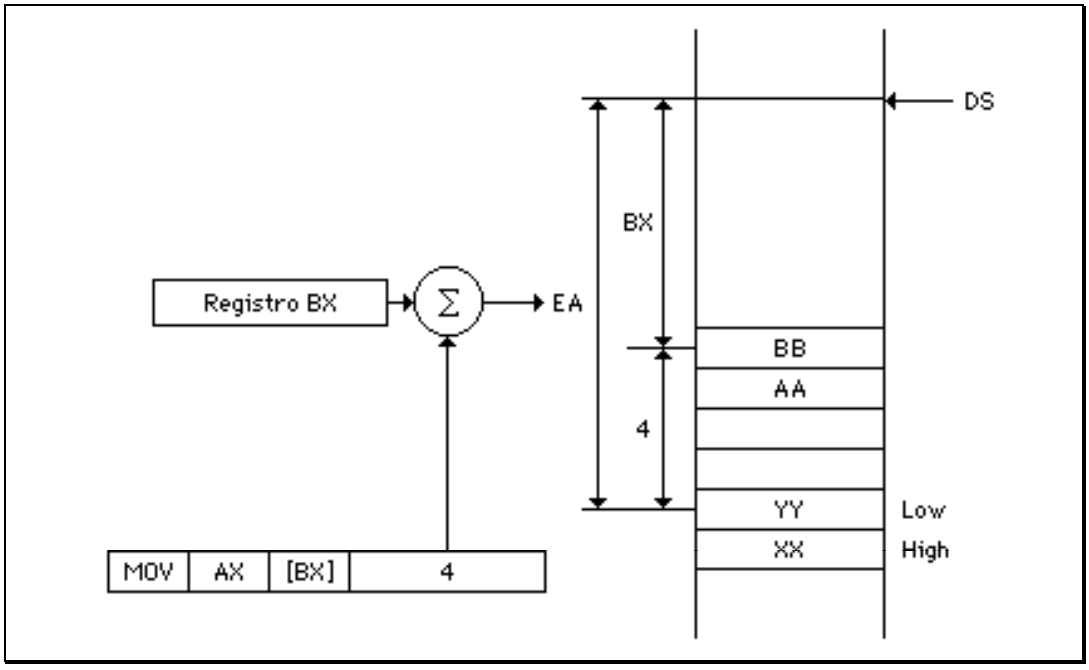
```
d1  DW    3 DUP (0)
d2  DW    3 DUP (0)
d3  DW    3 DUP (0)

    . . .

    MOV   AX, [BX+4]          ;  AX <- [BX+4] (16 bit)
```

il registro BX contiene un indirizzo nel segmento dati corrente (d1, d2 o d3)

per accedere all'operando (l'anno) è necessario sommare 4 a tale indirizzo



Indirizzamento mediante Registro Indice

L'EA si ottiene come somma

- di un valore di displacement
- del contenuto di un registro indice (SI o DI)

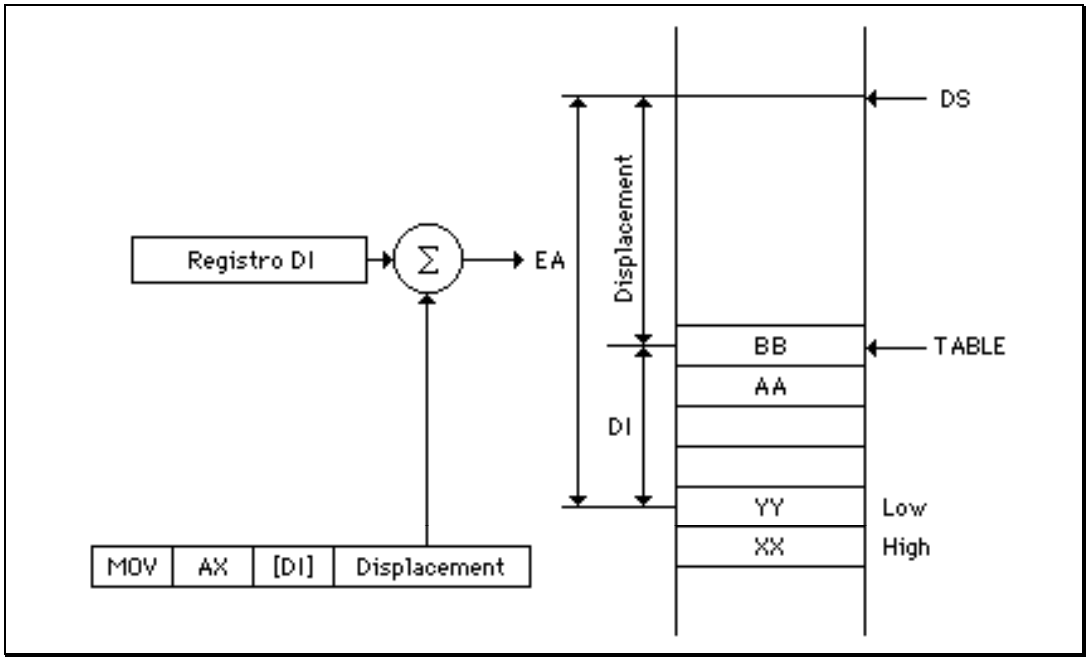
Questa modalità di indirizzamento è utile per **accedere ai diversi elementi di un array**

- il displacement fornisce l'indirizzo di partenza dell'array
- il registro indice seleziona uno degli elementi (il primo elemento viene selezionato quando il registro indice vale 0)

TABLE è l'indirizzo simbolico di un array di word contenuto nel data segment corrente

```
MOV  DI, 4                ;  inizializza DI a 4
MOV  AX, [TABLE+DI] ;  AX <- [TABLE+DI] (16 bit)
```

copia in AX il terzo elemento dell'array (nell'ordine, gli elementi dell'array hanno come displacement relativo a TABLE i valori 0, 2, 4, ecc.)



Indirizzamento mediante Registro Base e Registro Indice

L'EA si ottiene come somma

- di un valore di displacement
- del contenuto di un registro base (BX o BP)
- del contenuto di un registro indice (SI o DI)

Due componenti dell'indirizzo possono variare al momento dell'esecuzione

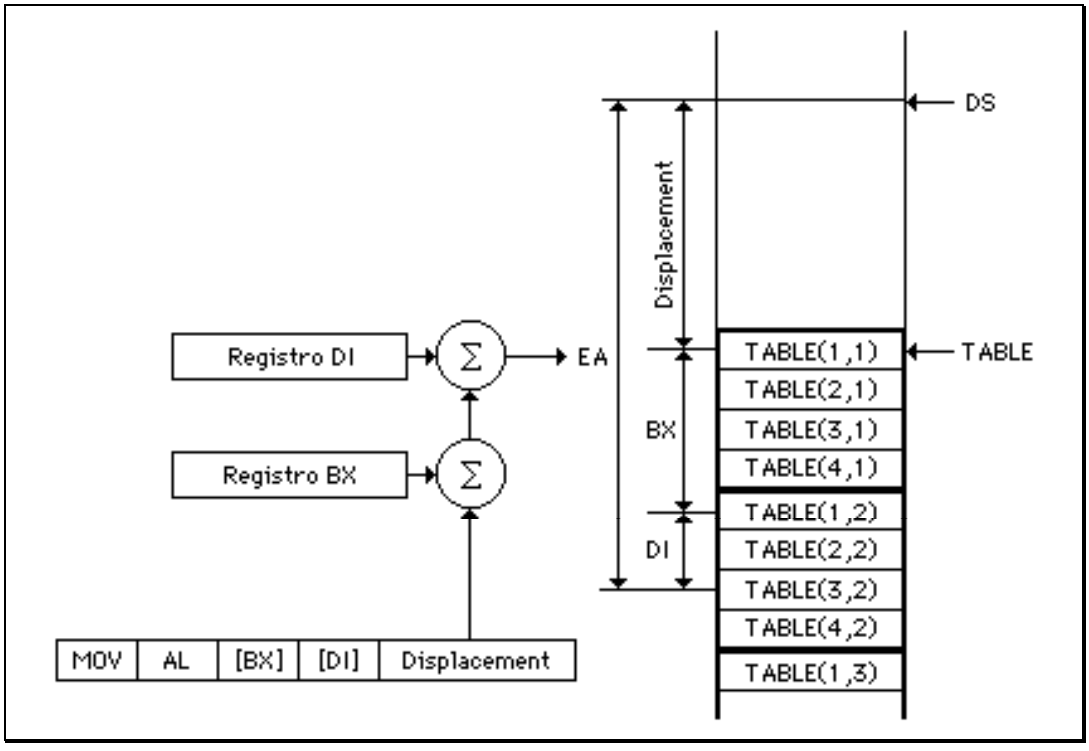
Questa modalità di indirizzamento è utile per accedere, ad esempio, ai diversi elementi di una matrice: il displacement fornisce l'indirizzo di partenza dell'array, mentre il registro base e il registro indice selezionano uno degli elementi (il primo elemento viene selezionato quando entrambi i registro valgono 0).

TABLE è l'indirizzo simbolico di una matrice di byte 4x4, memorizzata per colonne nel data segment corrente

```
MOV BX, 4                    ;  inizializza BX a 4
MOV DI, 2                    ;  inizializza DI a 2
MOV AL, [TABLE+BX+DI] ;  AL <- TABLE(3,2)
```

BX contiene il displacement tra l'indirizzo di partenza della matrice e la colonna selezionata (la seconda)

DI contiene il displacement tra l'indirizzo di partenza della colonna selezionata e la riga selezionata (la terza)



SET DI ISTRUZIONI

- **istruzioni per il trasferimento dati**
- **istruzioni aritmetiche**
- **istruzioni per la manipolazione di bit**
- **istruzioni per il trasferimento del controllo**
- **istruzioni che operano sulle stringhe**
- **istruzioni per il controllo del processore**
- **istruzioni che lavorano sul registro flag**
- **istruzioni per la gestione delle periferiche di input/output**

Principali istruzioni per il trasferimento dati

MOV (MOVe)

Istruzione di trasferimento general-purpose.

Trasferisce un byte o una word

- **tra due registri**
- **tra un registro e una locazione di memoria**

L'operando sorgente può essere una costante.

Per trasferire un dato da una locazione di memoria a un'altra:

```
MOV  AX, fromVar    ;  copia fromVar in AX
MOV  toVar, AX      ;  e quindi in toVar
```

Non è possibile caricare un valore immediato (come l'indirizzo di un segmento) **in un registro segmento**:

```
MOV  AX, segData    ;  indirizzo di segData in AX
MOV  DS, AX         ;  e quindi in DS
```

Il registro CS non è utilizzabile come destinazione di un'istruzione MOV.

MOVS (MOV per la copia di stringhe) e **LODS** (load di una stringa dalla memoria)

MOVS copia il contenuto della stringa di dati sorgente nella stringa di dati destinazione:

MOVS dst-string, src-string

Permette la copia sia di stringhe di byte che di stringhe di word, a seconda della dichiarazione delle stringhe stesse. (MOVSB e MOVSW)

La stringa sorgente è all'indirizzo DS:SI, quella destinazione all'indirizzo ES:DI. SI e DI vengono automaticamente incrementati o decrementati a seconda del valore del DF (direction flag).

LODS trasferisce il valore contenuto nella memoria all'indirizzo DS:SI e lo carica in **AL** (se ad 8 bit) o in **AX** (se a 16 bit). Incrementa o decrementa (a seconda del valore di DF) automaticamente SI ad ogni passo.

Utilizzata prevalentemente per lavorare su stringhe di dati provenienti da dispositivi di I/O mappati in memoria.

PUSH e **POP**

Operazione di **push** o di **pop al top dello stack selezionato da SS:SP**.

L'unico operando (sempre una word) può essere **un registro o una locazione di memoria**.

PUSHF (PUSH Flags onto stack)
POPF (POP Flags off stack)

Trasferiscono il contenuto del registro flag (16 bit) **nello stack e viceversa**.

PUSHF	_	PUSH del registro flag
POPF	_	POP del registro flag

Le due istruzioni sono necessarie in quanto il registro flag non ha un nome. Non può essere referenziato direttamente

LEA (Load Effective Address)

Trasferisce l'EA dell'operando sorgente (e non il suo valore!) nell'operando destinazione.

L'operando sorgente **deve** essere un riferimento alla memoria e l'operando destinazione **deve** essere un registro a 16 bit.

Un'operazione simile può essere ottenuta usando il MOV con la direttiva **OFFSET**

MOV AX, OFFSET TABLE

Carica in AX l'EA della variabile TABLE.

Però, l'istruzione LEA permette di utilizzare l'indicizzazione dell'operando sorgente.
OFFSET calcola l'EA a compile-time calcolando lo scostamento dall'inizio del segmento dati.
Non è quindi possibile utilizzare la seguente istruzione per l'accesso ad un vettore VETT:

MOV BX, OFFSET TABLE[SI]

che può invece essere implementata con LEA:

LEA BX, TABLE[SI]

Impiega 2 cicli di clock + calcolo dell'EA.

Principali istruzioni aritmetiche

Le istruzioni aritmetiche prevedono operandi di al massimo 16 bit, in quanto la ALU dell'8086 ha un parallelismo di 16 bit. E' possibile anche utilizzare operandi di 8 bit.

ADD e **ADC** (ADD with Carry)

Sommano 2 operandi di **8** o **16** bit. Uno dei due operandi può risiedere in memoria, l'altro è necessariamente un registro (o una costante, nel caso dell'operando sorgente).

ADD somma l'operando sorgente all'operando destinazione e memorizza il risultato nell'operando destinazione:

Con la sintassi del C:

destinazione = destinazione + sorgente;

destinazione += sorgente;

ADC effettua la stessa operazione, comprendendo nella somma il flag del riporto (CF) (il cui valore è stabilito dall'istruzione precedente):

destinazione = destinazione + sorgente + CF;
destinazione += sorgente + CF;

L'istruzione ADC permette di effettuare la somma di numeri maggiori di 16 bit:

Esempio: Somma di due numeri di 32 bit collocati in coppie di registri:

(AX:BX) <- (AX:BX) + (CX:DX)

```
ADD  BX, DX    ; somma i 16 bit meno significativi
ADC  AX, CX    ; somma i 16 bit più significativi
```

La ADD modifica il contenuto del registro dei flag, e in particolare CF che assume il significato di **bit di riporto**.

SUB (SUBtract) e **SBB** (SuBtract with Borrow)

SUB sottrae l'operando sorgente all'operando destinazione e memorizza il risultato nell'operando destinazione:

destinazione = destinazione – sorgente;
destinazione –= sorgente;

SBB effettua la stessa operazione, comprendendo nella sottrazione il flag del riporto (CF) borrow = prestito**:**

destinazione = destinazione – sorgente - CF;
destinazione –= sorgente - CF;

L'istruzione SBB permette di effettuare la sottrazione di numeri maggiori di 16 bit.

Sottrazione di due numeri di 32 bit
(AX:BX) <- (AX:BX) – (CX:DX)

```
SUB  BX, DX    ; sottrae i 16 bit meno significativi
SBB  AX, CX    ; sottrae i 16 bit più significativi
```

La SUB modifica il contenuto del registro dei flag, e in particolare CF che assume il significato di **bit di prestito**.

INC (INCrement destination by 1)

Incrementa di 1 il valore del suo operando. Più veloce e compatta di ADD operando, 1.

Var++; Var +=1; Var = Var + 1;

DEC (DECrement destination by 1)

Decrementa di 1 il valore del suo operando.

Var--; Var -=1; Var = Var - 1;

NEG (NEGate)

Cambia segno al suo operando (secondo la complementazione in **complemento a 2)**

NEG(x) = [NOT (x) + 1]

CMP

(CoMPare destination with source)

CMP agisce esattamente come l'istruzione SUB, **senza però modificare l'operando destinazione**.

Lo scopo è quello di modificare il valore di alcuni flag (ved. SUB).

MUL (MULtiply, unsigned) e **IMUL** (Integer MULtiply, signed)

Se l'operando è **un byte**, viene eseguito:

AX = AL * sorgente; (16 bit = 8 bit x 8 bit)

Se l'operando è **una word**, viene eseguito:

DX:AX = AX * sorgente; (32 bit = 16 bit x 16 bit)

Esistono due distinte istruzioni di moltiplicazione: MUL per i numeri senza segno e IMUL per i numeri con segno (in complemento a 2); l'operazione di moltiplicazione è diversa nei due casi.

Esempio in C:

int a, b; unsigned int c, d;

... a * b ... → ... IMUL ...
... c * d ... → ... MUL ...

Il compilatore utilizza l'istruzione corretta sulla base del **tipo** degli operandi.

NB: l'operando sorgente non può essere una costante.

DIV (DIVide, unsigned) e **IDIV** (Integer DIVide, signed)

L'operazione di divisione intera dà luogo a **due** risultati: **quoziente** e **resto** (esistono operatori specifici in C e Pascal). In ASM, una sola istruzione fornisce entrambi i risultati.

se l'operando è **un byte**, viene eseguito:

AL = Quoziente(AX / sorgente);
AH = Resto(AX / sorgente);

Dimensione degli operandi: 8 bit = 16 bit/8 bit

se l'operando è **una word**, viene eseguito:

AX = Quoziente(DX:AX / sorgente);
DX = Resto(DX:AX / sorgente);

Dimensione degli operandi: 16 bit = 32 bit/16 bit

ATTENZIONE: in operazioni con numeri negativi, deve risultare negativo DX:AX

DIV: operandi senza segno
IDIV: operandi con segno (compl. a 2)

NB: l'operando sorgente non può essere una costante

Istruzioni
per la manipolazione di bit

AND (logical AND)
OR (inclusive-OR)
XOR (eXclusive-OR)

Eseguono le corrispondenti operazioni logiche tra le coppie di bit di pari posizione degli operandi (corrispondono agli operatori **a bit** del C - &, |, .. - non a quelli **logici** - &&, ||, ..)

```
AND   dst, src      ; dst &= src
OR    dst, src      ; dst |= src
XOR   dst, src      ; dst ^= src
```

NOT (logical NOT)

Complementa tutti i bit del suo operando (complemento a 1).

C'è differenza tra NOT e NEG (complemento a 2)!!!!!!

SAL (Shift Arithmetic Left)

- shift verso sinistra di un numero con segno
- segno del numero in CF
- 0 nel bit meno significativo

SAR (Shift Arithmetic Right)

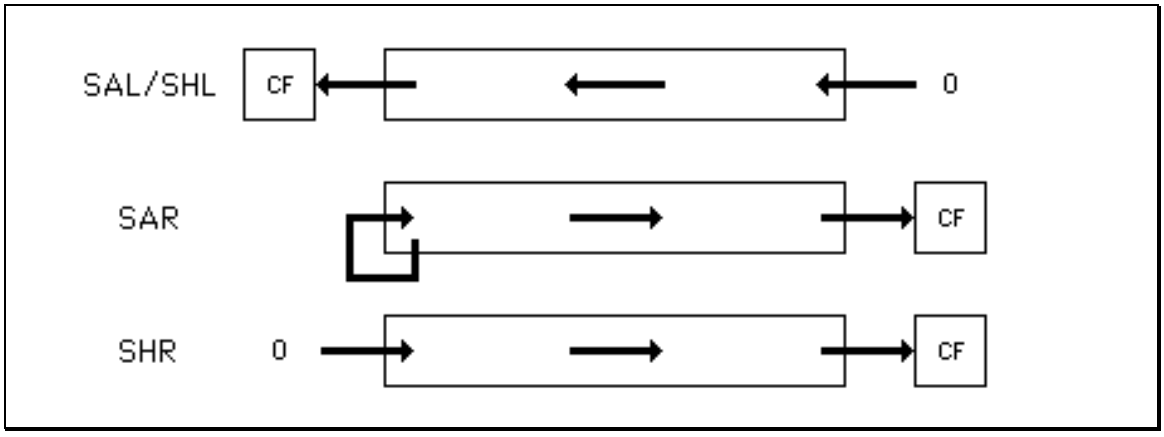
- shift verso destra di un numero con segno
- conserva il segno del numero (bit più significativo)
- pone il bit meno significativo in CF

SHL (SHift logical Left)

- shift verso sinistra di un numero senza segno
- bit più significativo in CF
- 0 nel bit meno significativo

SHR (SHift logical Right)

- shift verso destra di un numero senza segno
- 0 nel bit più significativo
- pone il bit meno significativo in CF



ROL (ROtate Left)

• shift verso sinistra di un numero
• bit più significativo in CF
• bit più significativo copiato nel meno significativo

ROR (Rotate Right)

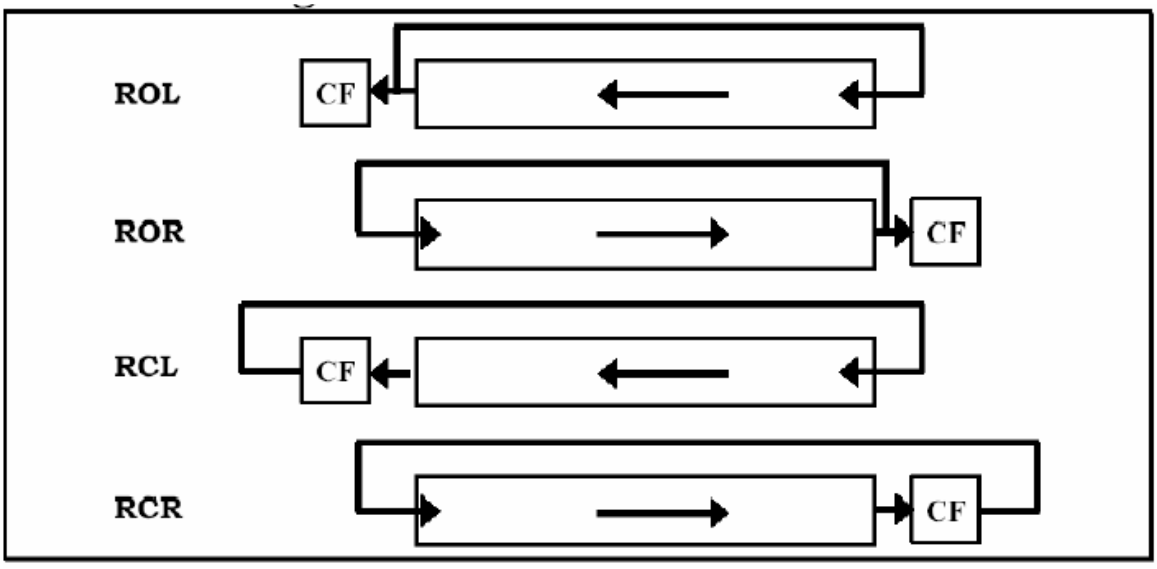
• shift verso destra di un numero
• bit meno significativo in CF
• bit meno significativo copiato nel più significativo

RCL (Rotate through Carry Left)

• shift verso sinistra di un numero
• CF copiato nel bit meno significativo
• bit più significativo in CF

RCR (Rotate through Carry Right)

• shift verso destra di un numero
• CF copiato nel bit più significativo
• bit meno significativo in CF



Istruzioni per il trasferimento del controllo

CALL (CALL a procedure)
RET (RETurn from procedure)

CALL trasferisce il controllo dal programma chiamante alla procedura chiamata

- salva l'indirizzo di ritorno sullo stack
- passa il controllo alla procedura chiamata

RET trasferisce il controllo dalla procedura chiamata al programma chiamante

- legge dallo stack l'indirizzo di ritorno salvato dalla corrispondente CALL
- ripassa il controllo al chiamante

esistono chiamate **near** (all'interno dello stesso segmento di codice, salva solo l'offset di ritorno) e **far** (tra segmenti diversi, deve salvare CS e offset di ritorno - più lenta della precedente).

INT (INTerrupt)
IRET (Interrupt RETurn)

INT trasferisce il controllo dal programma corrente alla procedura di risposta all'interruzione (RRI) indicato come numero (costante)

- salva il registro flag nello stack
- mette a 0 sia l'IF che il TF
- esegue una CALL FAR all'indirizzo INT*4

(quindi prima salva nello stack i precedenti valori di CS e IP)

IRET ridà il controllo al programma corrente cha ha chiamato la routine di risposta ad un'interruzione

- legge dallo stack l'indirizzo di ritorno salvato dalla CALL FAR (quindi IP e CS – ordine inverso)
- esegue una POPF per ristabilire i valori del registro flag prima della chiamata all'interrupt
- per il punto precedente è necessario ritornare da un interrupt con un IRET e non con una RET

JMP (JuMP unconditionally)

Trasferisce il controllo all'istruzione specificata dall'operando, in modo incondizionato. Simile all'istruzione di CALL, ma non prevede il rientro nel programma chiamante (non salva nulla sullo stack).

L'operando di JMP è simile all'operando dell'istruzione CALL (può essere **NEAR o FAR):**

```
        JMP  NearLabel
        . . .
NearLabel:
        . . .
```

NearLabel appartiene al segmento codice corrente

```
        JMP  far ptr FarLabel
        . . .
FarLabel  LABEL  far:
        . . .
```

FarLabel appartiene a un segmento codice qualsiasi.

FAR PTR è l'operatore dell'ASM che impone la chiamata o il salto di tipo FAR.

Jxxx (Jump conditionally)

Le istruzioni per il trasferimento condizionato controllano se una condizione è verificata (test sul registro dei flag):

- **se la condizione è verificata**
  il controllo passa all'istruzione di label
- **se la condizione non è verificata**
  l'esecuzione prosegue con la successiva istruzione

Uso del salto condizionato:

Il salto condizionato è utilizzato per implementare gli operatori relazionali (==, !=, >, <, >=, <=).
Ad esempio, per controllare se AX e BX contengono lo stesso valore, si utilizza l'istruzione CMP seguita da un opportuno salto condizionato:

```
CMP  AX, BX
JE  LabelZero
```

Per alcuni tipi di test è necessario scegliere tra **due differenti istruzioni di salto**, a seconda che si stia testanto il risultato di un'operazione tra **valori con o senza segno** (mnemonico: “G”, “L”: greater, less (con s.); “A”, “B”: above, below (senza s.))

Istruzioni di salto da utilizzare subito dopo l'istruzione CMP, in funzione:

- del tipo di test che si desidera effettuare
- del tipo di dato confrontato

Per saltare se	Valori senza segno	Valori con segno
Destinazione = Sorgente	JE	JE
Destinazione ≠ Sorgente	JNE	JNE
Destinazione > Sorgente	JA	JG
Destinazione ≥ Sorgente	JAE	JGE
Destinazione < Sorgente	JB	JL
Destinazione ≤ Sorgente	JBE	JLE

Esempio: realizzazione di un test a tre vie

```
        CMP  AX, BX
        JAE  LabelGreaterOrEqual
        . . .               ; istruzioni eseguite se AX < BX
LabelGreaterOrEqual:
        JA LabelGreater
        . . .               ; istruzioni eseguite se AX = BX
LabelGreater:
        . . .               ; istruzioni eseguite se AX > BX
```

esegue **tre differenti blocchi di istruzioni**, in funzione dei valori **senza segno** contenuti nei registri AX e BX
nel caso di valori **con segno**, **sostituire JAE con JGE e JA con JG**.

Elenco dei possibili salti condizionati:

Istruzione	Descrizione	Salta se ...
JA	Jump if Above	CF = 0 e ZF = 0
JAE	Jump if Above or Equal	CF = 0
JB	Jump if Below	CF = 1
JBE	Jump if Below or Equal	CF = 1 o ZF = 1
JC	Jump if Carry	CF = 1
JE	Jump if Equal	ZF = 1
JG	Jump if Greater	ZF = 0 e SF = OF
JGE	Jump if Greater or Equal	SF = OF
JL	Jump if Less	SF ≠ OF
JLE	Jump if Less or Equal	ZF = 1 o SF ≠ OF
JNA	Jump if Not Above	CF = 1 o ZF = 1
JNAE	Jump if Not Above nor Equal	CF = 1
JNB	Jump if Not Below	CF = 0
JNBE	Jump if Not Below nor Equal	CF = 0 e ZF = 0
JNC	Jump if No Carry	CF = 0
JNE	Jump if Not Equal	ZF = 0
JNG	Jump if Not Greater	ZF = 1 o SF ≠ OF
JNGE	Jump if Not Greater nor Equal	SF _ OF
JNL	Jump if Not Less	SF = OF
JNLE	Jump if Not Less nor Equal	ZF = 0 e SF = OF
JNO	Jump if No Overflow	OF= 0
JNP	Jump if No Parity (odd)	PF = 0
JNS	Jump if No Sign	SF = 0
JNZ	Jump if Not Zero	ZF = 0
JO	Jump on Overflow	OF =1
JP	Jump on Parity (even)	PF = 1
JPE	Jump if Parity Even	PF = 1
JPO	Jump if Parity Odd	PF = 0
JS	Jump on Sign	SF = 1
JZ	Jump if Zero	ZF = 1

LOOP (LOOP on count) e **REP** (REPeat)

LOOP implementa un ciclo **for**. Il numero di iterazioni del ciclo (compreso tra la **near** label indicata e l'istruzione LOOP) è contenuto in CX

- decrementa CX di 1
- se CX<>0 salta all'istruzione indicata dalla label
- altrimenti finisce il ciclo eseguendo l'istruzione successiva alla LOOP

REP non è un'istruzione, ma un prefisso (vedi campo PREFIX dell'istruzione) che può essere anteposto ad una qualsiasi istruzione che lavora sulle stringhe (noi abbiamo visto MOVSe LODS, ma esistono anche CMPS, STOS e SCAS). L'istruzione viene ripetuta finchè CX è diverso da 0. Eseguite le istruzioni, CX viene decrementato di 1.
Implementa un ciclo **while**.

Esempio:

```
CLD             ; Clear Direction Flag Muove nella
                ; direzione in avanti
LEA SI, BUFFER1         ; SI punta alla sorgente
LEA DI, BUFFER2      ; … e DI alla destinazione
MOV CX, 100      ; REP usa CX come contatore
REP MOVSB
```

Istruzioni che lavorano sul registro flag

CL*x* (CLear flag *x*) e **ST*x*** (SeT flag *x*)

***x* può assumere i valori:**
- **C** per il CF (Carry Flag)
- **D** per il DF (Direction Flag)
- **I** per l'IF (Interrupt Flag)

Istruzioni per la gestione delle periferiche di Input/Output

IN (INput byte or word) e **OUT** (OUTput byte or word)

IN trasferisce un byte o una word da una porta di I/O ad AL (se un byte) o ad AX (se una word). La porta può essere specificata o con un valore immediato (massimo 8 bit, quindi possibile solo per porte con un numero compreso tra 0 e 255) o tramite il registro DX a 16 bit (che permette l'accesso a tutte le 64K porte di I/O possibili). Stessa cosa per **OUT**.

```
IN AL, 45h
OUT DX, AL
OUT 250, AX
```

Interrupt supportati

INT 10h / **AH = 02h** - set cursor position.
input:
DH = row.
DL = column.
BH = page number (0..7).

INT 10h / **AH = 08h** - read character and **attribute** at cursor position.

input:
BH = page number.
return:
AH = **attribute**.
AL = character.

INT 10h / **AH = 09h** - write character and **attribute** at cursor position.

input:
AL = character to display.
BH = page number.
BL = **attribute**.
CX = number of times to write character.

INT 10h / **AH = 0Ah** - write character only at cursor position.

input:
AL = character to display.

BH = page number.
CX = number of times to write character.

INT 10h / **AH = 13h** - write string.

input:
AL = write mode:
    bit 0: update cursor after writing;
    bit 1: string contains **attributes**.
BH = page number.
BL = **attribute** if string contains only characters (bit 1 of AL is zero).
CX = number of characters in string (attributes are not counted).
DL,DH = column, row at which to start writing.
ES:BP points to string to be printed.

Bit color table:
Character attribute is 8 bit value, low 4 bits set foreground color, high 4 bits set background color. Background blinking not supported.

HEX	BIN	COLOR
0	0000	black
1	0001	blue
2	0010	green
3	0011	cyan
4	0100	red

5	0101	magenta
6	0110	brown
7	0111	light gray
8	1000	dark gray
9	1001	light blue
A	1010	light green
B	1011	light cyan
C	1100	light red
D	1101	light magenta
E	1110	yellow
F	1111	white

INT 20h - exit to operating system.

INT 21h / **AH=09h** - output of a string at DS:DX.

INT 21h / **AH=0Ah** - input of a string to DS:DX, fist byte is buffer size, second byte is number of chars actually read.

INT 21h / **AH=4Ch** - exit to operating system.

INT 21h / **AH=01h** - read character from standard input, with echo, result is stored in AL.

INT 21h / **AH=02h** - write character to standard output, DL = character to write, after execution AL = DL.