Calcolatori Elettronici
Linguaggio macchina i8086

Ing. Gestionale e delle Telecomunicazioni

A.A. 2009/10

Gabriele Cecchetti

Il linguaggio Assembler i8086

- Sommario:
  - Il processo di traduzione
  - Sintassi
  - L'istruzione MOV
  - Variabili, Array, Indirizzo effettivo di una variabile, Costanti
  - Istruzioni logico aritmetiche
  - Controllo di flusso
  - Lo stack e le procedure
- Riferimenti
  - G. Bucci "Architetture e organizzazione dei Calcolatori Elettronici – Fondamenti", Cap. 10

Generalità sul processo di traduzione

- Creazione del programma sorgente
  `file.asm`
- Assemblaggio dei moduli sorgente
  `file.obj`
- Collegamento dei moduli (eventualmente presenti in librerie – es. `mylibrary.lib`)
- Creazione eseguibile
  `file.exe`

Sintassi

- Codice fatto di linee di caratteri ASCII.
- Ogni linea rappresenta uno statement (istruzione) – esclusi i commenti che possono occupare più linee.
- Un generico statement ha quattro campi:

```
Nomi        Operazione      Operandi  ; Commento
```

Nomi

- Sono nomi scelti dal programmatore
- Possono contenere caratteri alfanumerici ed alcuni caratteri speciali
- Sono case-insensitive

Codice di operazione

- Può essere:
  - il codice mnemonico di un'istruzione macchina:
    - es. `MOV AX, BX ; carica BX in AX`
  - una direttiva per l'assemblatore:
    - es. `VAR DB 0` ; la variabile VAR, ampia 1 byte è inizializzata a 0.

Operandi

- Dipendono dal codice operazione

L'istruzione MOV

- Copia il **secondo operando** (sorgente) sul **primo operando** (destinatario).
- L'operando sorgente può essere un valore immediato, un registro generale o una locazione di memoria.
- L'operando destinatario può essere un registro generale o una locazione di memoria.
- Entrambi gli operandi debbono essere della stessa misura, che può essere un byte o una parola.

Sintassi MOV

```
MOV REG, memory
MOV memory, REG
MOV REG, REG
MOV memory, immediate
MOV REG, immediate
```

REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

memory: [BX], [BX+SI+7], variable, etc...

immediate: 5, -24, 3Fh, 10001101b, etc...

MOV per i registri selettore

```
MOV SREG, memory
MOV memory, SREG
MOV REG, SREG
MOV SREG, REG
```

SREG: DS, ES, SS, and only as second operand: CS.

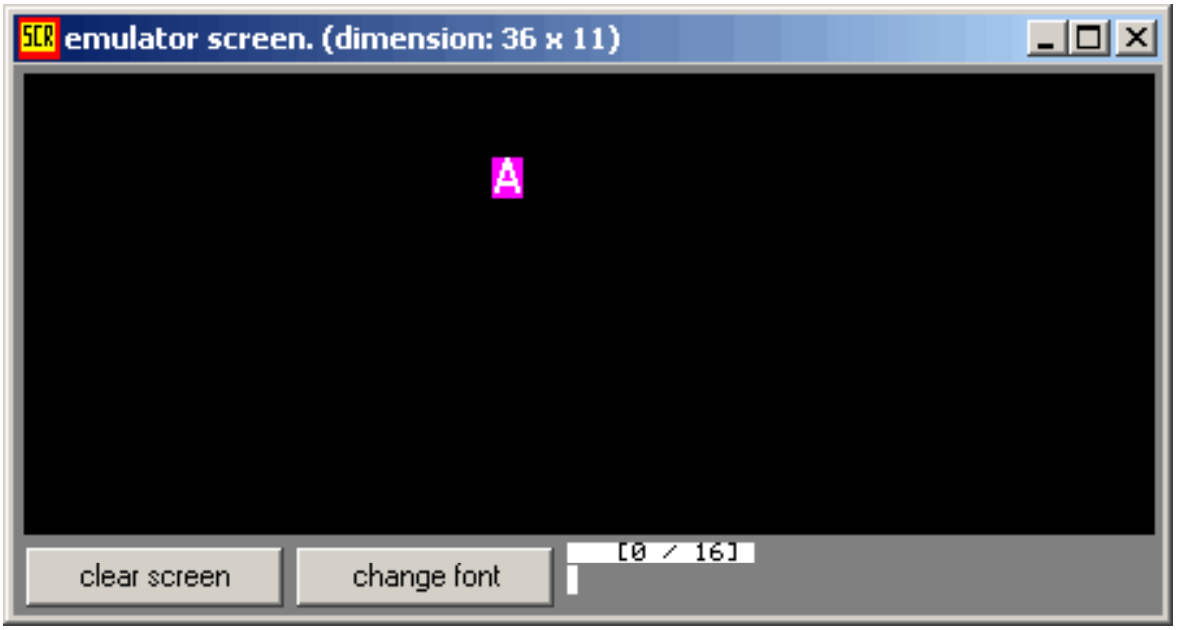
REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

memory: [BX], [BX+SI+7], variable, etc...

Esempio di programma con MOV

```
.model small
.code
 .startup
MOV AX, 0B800h      ; set AX to hexadecimal value
                    ; of B800h.
MOV DS, AX          ; copy value of AX to DS.
MOV CL, 'A'         ; set CL to ASCII code of 'A',
                    ; it is 41h.
MOV CH, 1101_1111b  ; set CH to binary value.
MOV BX, 15Eh        ; set BX to 15Eh.
MOV [BX], CX        ; copy contents of CX to memory
                    ; at B800:015E
.exit               ; returns to operating system.
END
```


Output programma MOV



Variabili

- Una variabile è una locazione di memoria.
- Le variabili possono essere indicate in modo mnemonico, per esempio con il nome `var1`.
- Il compilatore supporta due tipi di variabili: **`BYTE`** e **`WORD`**.

Variabili: sintassi

Dichiarazione per una variabile:

nome **DB** *valore*

nome **DW** *valore*

DB – significa Define Byte.
DW – significa Define Word.

nome – può essere una qualsiasi combinazione di lettere o numeri purchè inizi con una lettera. E' possibile dichiarare variabili senza nome (che comunque hanno un indirizzo).

valore – può essere qualsiasi valore numerico in qualsiasi sistema di numerazione supportato (esadecimale, binario, o decimale), oppure il simbolo "**?**" per le variabili che non sono inizializzate.

Esempio di uso delle variabili

```
VAR1 DB 7
var2 DW 1234h
…
MOV AL, var1
MOV BX, var2
RET             ; stops the program.
```

Il compilatore non è case-sentitive, così "**VAR1**" and "**var1**" riferiscono la stessa variabile.

Array (1/2)

Esempio:

```
a DB 48h, 65h, 6Ch, 6Ch, 6Fh, 00h
b DB 'Hello', 0
```

b is è una copia esatta dell'array a, quando il compilatore vede una stringa tra apici, automaticamente la converte ad un insieme di bytes.

E' possibile accedere agli elementi dell'array nei modi seguenti:

```
MOV AL, a[3]
```

oppure usando uno qualsiasi dei registri indici della memoria **BX, SI, DI, BP**, per esempio:

```
MOV SI, 3
MOV AL, a[SI]
```

Array (2/2)

- Per dichiarare array di grandi dimensioni è possibile usare l'operatore **DUP**.
- Sintassi:
  numero DUP (valore(i))
  numero – numero di duplicati da creare (qualsiasi valore costante).
  valore – espressione che DUP duplicherà.
- Per esempio:
  `c DB 5 DUP(9)` ; è un modo alternativo di dichiarare:
  `c DB 9, 9, 9, 9, 9`
- Altro esempio
  `d DB 5 DUP(1, 2)` ; è un modo alternativo di dichiarare:
  `d DB 1, 2, 1, 2, 1, 2, 1, 2, 1, 2`

Indirizzo di una variabile

- Ottenere l'indirizzo di una variabile può essere molto utile in alcune situazioni come, ad esempio, quando è necessario passare parametri ad una procedura.
- Si hanno a disposizione due istruzioni
  - LEA
  - OFFSET

LEA: esempio

```
…
MOV AL, VAR1            ; check value of VAR1 by moving
                        ; it to AL.
LEA BX, VAR1            ; get address of VAR1 in BX.
MOV BYTE PTR [BX], 44h  ; modify the contents of VAR1.
MOV AL, VAR1            ; check value of VAR1 by moving
                        ; it to AL.
RET
VAR1 DB 22h
END
```


OFFSET: esempio

```
…
MOV AL, VAR1            ; check value of VAR1 by moving
                        ; it to AL.
MOV BX, OFFSET VAR1     ; get address of VAR1 in BX.
MOV BYTE PTR [BX], 44h  ; modify the contents of VAR1.
MOV AL, VAR1            ; check value of VAR1 by moving
                        ; it to AL.
RET
VAR1 DB 22h
END
```

Costanti

- Per definire le costanti si usa la direttiva **EQU** :
  `nome EQU < qualsiasi espressione >`

- Esempio:

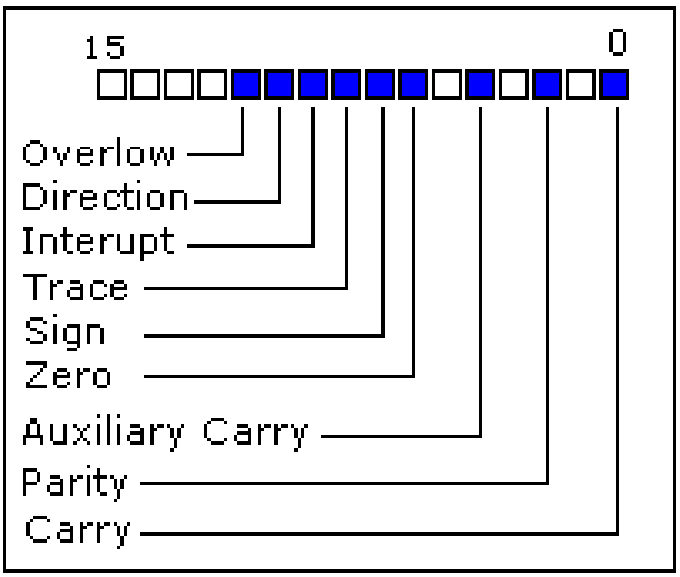
```
k EQU 5
MOV AX, k
```

è funzionalmente identico al codice:

```
MOV AX, 5
```


Istruzioni logico aritmetiche

- La maggior parte delle istruzioni logico aritmetiche modificano il registro dei *flags*.



ADD, SUB,CMP, AND, TEST, OR, XOR (1/2)

- Tipi di operandi supportati:

```
REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate
```

  - **REG**: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.
  - **memory**: [BX], [BX+SI+7], variable, etc...
  - **immediate**: 5, -24, 3Fh, 10001101b, etc...

ADD, SUB,CMP

- **ADD** – Somma il secondo operando al primo.

```
MOV AL, 5     ; AL = 5
ADD AL, -3    ; AL = 2
```

- **SUB** – Sottrae il secondo operando al primo.

```
MOV AL, 5
SUB AL, 1     ; AL = 4
```

- **CMP** – Sottrae il secondo operando dal primo *solo per i flags*, ossia non memorizza il risultato.

```
MOV AL, 5
MOV BL, 5
CMP AL, BL    ; AL = 5, ZF = 1
```

AND, TEST

- **AND** – Prodotto logico fra tutti i bit dei due operandi.

```
MOV AL, 'a'          ; AL = 01100001b
AND AL, 11011111b    ; AL = 01000001b ('A')
```

- **TEST** – Come l' **AND** ma *solo per i flags*.

```
MOV AL, 00000101b
TEST AL, 1           ; ZF = 0.
TEST AL, 10b         ; ZF = 1.
```


OR, XOR

- **OR** – Somma logica fra tutti i bit dei due operandi.

```
MOV AL, 'A'          ; AL = 01000001b
OR AL, 00100000b     ; AL = 01100001b ('a')
```

- **XOR** – OR esclusivo fra tutti i bit dei due operandi.

```
MOV AL, 00000111b
XOR AL, 00000010b ; AL = 00000101b
```

MUL, IMUL, DIV, IDIV

- Operandi suportati:

  REG

  memory

  - **REG**: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.
  - **memory**: [BX], [BX+SI+7], variable, etc...

MUL

- **MUL** – Moltiplicazione senza segno:
  - Quando l'operando è un **byte**:

    `AX = AL * operando`
  - Quando l'operando è una **word**:

    `(DX AX) = AX * operando`
  - Esempio:

```
MOV AL, 200 ; AL = 0C8h
MOV BL, 4
MUL BL ; AX = 0320h (800)
```

IMUL

- **IMUL** – Moltiplicazione con segno:
  - Quando l'operando è un **byte**:
    `AX = AL * operando`
  - Quando l'operando è una **word**:
    `(DX AX) = AX * operando`
  - Esempio:

```
MOV AL, -2
MOV BL, -4
IMUL BL       ; AX = 8
```


DIV

- **DIV** – Divisione senza segno:
  - Quando l'operando è un **byte**:
    `AL = AX / operando`
    `AH = resto (modulo)`
  - Quando l'operando è una **word**:
    `AX = (DX AX) / operando`
    `DX = resto (modulo)`
  - Esempio:

```
MOV AX, 203    ; AX = 00CBh
MOV BL, 4
DIV BL         ; AL = 50 (32h), AH = 3
```

IDIV

- **IDIV** – Divisione con segno:
  - Quando l'operando è un **byte**:

    `AL = AX / operando`

    `AH = resto (modulo)`
  - Quando l'operando è una **word**:

    `AX = (DX AX) / operando`

    `DX = resto (modulo)`
  - Esempio:

```
MOV AX, -203 ; AX = 0FF35h
MOV BL, 4
IDIV BL      ; AL = -50 (0CEh), AH = -3 (0FDh)
```


INC, DEC, NOT, NEG

- Operandi:

  `REG`

  `memory`
  - **REG**: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.
  - **memory**: [BX], [BX+SI+7], variable, etc…

INC, DEC, NOT, NEG: esempi

- **INC** – Incrementa l'operando (Operando++)

```
MOV AL, 4
INC AL                  ; AL = 5
```

- **DEC** – Decrementa l'operando (Operando--)

```
MOV AL, 255             ; AL = 0FFh (255 or -1)
DEC AL                  ; AL = 0FEh (254 or -2)
```

- **NOT** – Inverte ogni bit dell'operando (complemento A 1)

```
MOV AL, 00011011b
NOT AL                  ; AL = 11100100b
```

- **NEG** – Rende l'operando negativo (complemento a 2)

```
MOV AL, 5               ; AL = 05h
NEG AL                  ; AL = 0FBh (-5)
NEG AL                  ; AL = 05h (5)
```


Controllo di flusso: salti incondizionati

- Sintassi dell'istruzione **JMP**:

```
JMP label
```

- Per dichiarare un'etichetta nel programma occorre digitare il suo nome e aggiungere "**:**" alla fine.
- Le etichette possono essere qualsiasi stringa alfanumerica che non inizia con un numero.
- Le etichette possono essere dichiarare su linee separate o prima di qualsiasi istruzione.
- Esempio:

```
x1:
    MOV AX, 1
x2: MOV AX, 2
```

JMP: esempio

```
        …
        mov ax, 5    ; set ax to 5.
        mov bx, 2    ; set bx to 2.
        jmp calc     ; go to 'calc'.
back:   jmp stop     ; go to 'stop'.
calc:   add ax, bx   ; add bx to ax.
        jmp back     ; go 'back'.
stop:   ret          ; return to operating system.
```


Salti condizionali: test sul singolo flag

Instruction	Description	Condition	Opposite Instruction
JZ , JE	Jump if Zero (Equal).	ZF = 1	JNZ, JNE
JC , JB, JNAE	Jump if Carry (Below, Not Above Equal).	CF = 1	JNC, JNB, JAE
JS	Jump if Sign.	SF = 1	JNS
JO	Jump if Overflow.	OF = 1	JNO
JPE, JP	Jump if Parity Even.	PF = 1	JPO
JNZ , JNE	Jump if Not Zero (Not Equal).	ZF = 0	JZ, JE
JNC , JNB, JAE	Jump if Not Carry (Not Below, Above Equal).	CF = 0	JC, JB, JNAE
JNS	Jump if Not Sign.	SF = 0	JS
JNO	Jump if Not Overflow.	OF = 0	JO
JPO, JNP	Jump if Parity Odd (No Parity).	PF = 0	JPE, JP

Salti condizionali: numeri con segno

Instruction	Description	Condition	Opposite Instruction
JE , JZ	Jump if Equal (=). Jump if Zero.	ZF = 1	JNE, JNZ
JNE , JNZ	Jump if Not Equal (<>). Jump if Not Zero.	ZF = 0	JE, JZ
JG , JNLE	Jump if Greater (>). Jump if Not Less or Equal (not <=).	ZF = 0 and SF = OF	JNG, JLE
JL , JNGE	Jump if Less (<). Jump if Not Greater or Equal (not >=).	SF <> OF	JNL, JGE
JGE , JNL	Jump if Greater or Equal (>=). Jump if Not Less (not <).	SF = OF	JNGE, JL
JLE , JNG	Jump if Less or Equal (<=). Jump if Not Greater (not >).	ZF = 1 or SF <> OF	JNLE, JG

Salti condizionali: numeri senza segno

Instruction	Description	Condition	Opposite Instruction
JE , JZ	Jump if Equal (=). Jump if Zero.	ZF = 1	JNE, JNZ
JNE , JNZ	Jump if Not Equal (<>). Jump if Not Zero.	ZF = 0	JE, JZ
JA , JNBE	Jump if Above (>). Jump if Not Below or Equal (not <=).	CF = 0 and ZF = 0	JNA, JBE
JB , JNAE, JC	Jump if Below (<). Jump if Not Above or Equal (not >=). Jump if Carry.	CF = 1	JNB, JAE, JNC
JAE , JNB, JNC	Jump if Above or Equal (>=). Jump if Not Below (not <). Jump if Not Carry.	CF = 0	JNAE, JB
JBE , JNA	Jump if Below or Equal (<=). Jump if Not Above (not >).	CF = 1 or ZF = 1	JNBE, JA

Esempio di uso dei salti condizionali

- Generalmente, quando è richiesto di confrontare valori numerici si usa l'istruzione **CMP** (compie la stessa operazione dell'istruzione **SUB** (sottrazione), ma non mantiene il risultato, cambia solo i flags).
- Esempio: se occorre confrontare due numeri, 5 e 2 la CMP esegue:
  5 – 2 = 3 ossia il risultato non è zero e pertanto ZF = 0.
- Esempio: occorre confrontare due numeri, 7 e 7 la CMP esegue:
  7 – 7 = 0 ossia il risultato è zero e pertanto ZF = 1 e pertanto l'eventuale istruzione successiva **JZ** o **JE** effettuerà il salto.
- Esempio:

```
        org 100h
        mov al, 25  ; set al to 25.
        mov bl, 10  ; set bl to 10.
        cmp al, bl  ; compare al - bl.
        je equal    ; jump if al = bl (zf = 1).
        mov al, 'n' ; if it gets here, then al <> bl,
        jmp stop    ; so al ← 'n', and jump to stop.
equal:  mov al, 'y' ; if gets here, then al = bl, so
                    ; al ← 'y'.
stop:   ret         ; gets here no matter what.
```


Loops

instruction	operation and jump condition	opposite instruction
LOOP	decrease cx, jump to label if cx not zero.	DEC CX and JCXZ
LOOPE	decrease cx, jump to label if cx not zero and equal (zf = 1).	LOOPNE
LOOPNE	decrease cx, jump to label if cx not zero and not equal (zf = 0).	LOOPE
LOOPNZ	decrease cx, jump to label if cx not zero and zf = 0.	LOOPZ
LOOPZ	decrease cx, jump to label if cx not zero and zf = 1.	LOOPNZ
JCXZ	jump to label if cx is zero.	OR CX, CX and JNZ

- I loops sono principalmente equivalenti alle istruzioni di salto condizionale precedute da un'istruzione di confronto.
- Tutte le istruzioni loop usano il registro **CX** per contare i passi.

Considerazioni sui salti condizionali

- Tutti i salti condizionali hanno la limitazione di saltare solo **127** bytes in avanti e **128** bytes indietro.
- L'istruzione **JMP** non ha questa limitazione.

Lo stack

- Lo stack è un'area di memoria per memorizzare dati temporanei.
- Lo stack è usato dalle procedure (istruzioni **CALL**, **RET**, **INT**, e **IRET).**
- Lo stack può essere utilizzato per contenere anche altri dati temporanei.
- Esistono due istruzioni che operano con lo stack:
  - **PUSH** – memorizza valori di 16 bit nello stack.
  - **POP** – preleva valori di 16 bit dallo stack.
- **PUSH** e **POP** operano solo con valori di 16 bit!

PUSH: sintassi

```
PUSH REG
PUSH SREG
PUSH memory
PUSH immediate
```

- **REG**: AX, BX, CX, DX, DI, SI, BP, SP.
- **SREG**: DS, ES, SS, CS.
- **memory**: [BX], [BX+SI+7], 16 bit variable, etc...
- **immediate**: 5, -24, 3Fh, 10001101b, etc...

POP: sintassi

```
POP REG
POP SREG
POP memory
```

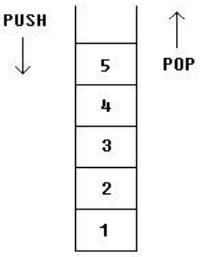
- **REG**: AX, BX, CX, DX, DI, SI, BP, SP.
- **SREG**: DS, ES, SS.
- **memory**: [BX], [BX+SI+7], 16 bit variable, etc...

Uso dello stack

- Le istruzioni PUSH e POP operano con una strategia di tipo **LIFO** (Last In First Out), ciò significa che se noi immettiamo questi valori uno ad uno nello stack:

  1, 2, 3, 4, 5

  il primo valore che estrarremo tramite l'istruzione POP sarà **5**, quindi **4**, **3**, **2**, e solo alla fine **1**.
- E' importante effettuare lo stesso numero di PUSH e di POP, altrimenti lo stack potrebbe corrompersi non permettendo al programma di ritornare al sistema operativo.
- L'uso delle istruzioni **PUSH** e **POP** è quindi essenzialmente il seguente:
  - memorizzare il valore originale del registro nello stack (usando **PUSH**);
  - usare il registro per qualsiasi scopo;
  - ripristinare il valore originale del registro prelevandolo dallo stack (usando **POP**).



Esempi di uso dello stack

```
MOV AX, 12h
PUSH AX       ; Stack ← AX
MOV AX, 58h   ; modify the AX
              ; value.
POP AX        ; AX ← Stack:
              ; restore the
              ; original value
              ; of AX.
```

```
MOV AX, 12h ; store 1212h
       ; in AX.
MOV BX, 34h ; store 3434h
       ; in BX
PUSH AX       ; Stack ← AX
PUSH BX       ; Stack ← BX
POP AX ; set AX to          ;
  original value
       ; of BX.
POP BX ; set BX to
       ; original value
       ; of AX.
```

Considerazioni sullo stack

- L'area di memoria riservata allo stack è impostata dal registro **SS** (Stack Segment), e dal registro **SP** (Stack Pointer).
- Di solito il sitema operativo imposta il valore di questi registri quando il programma inizia l'esecuzione.
- L'effetto dell'istruzione "**PUSH *sorgente***" è il seguente:
  - sottrae 2 dal registro **SP** ;
  - scrive il valore di ***source*** all'indirizzo **SS:SP**.
- L'effetto dell'istruzione "**POP *destinatario***" è il seguente:
  - scrive il valore contenuto all'indirizzo **SS:SP** sul destinatario;
  - aggiunge **2** al registro **SP**.
- L'indirizzo corrente puntato da **SS:SP** è chiamato ***top* dello stack**.

Procedure

- Una procedura è una parte di codice che può essere chiamata dal programma per eseguire un compito specifico.
- Le procedure rendono i programmi più strutturati e facili da comprendere.
- Generalmente una procedura ritorna allo stesso punto da dove è stata chiamata.

Sintassi delle procudure

- La sintassi per la dichiarazione di una procedura è la seguente:

```
name PROC
      ; codice della procedura…
RET
name ENDP
```

dove:

- name – è il nome della procedura.
- RET è usata per ritornare al chiamante.
- PROC e ENDP sono direttive per il compilatore per indicargli la presenza di una procedura e calcolarne gli indirizzi.
- CALL è usata per chiamare una procedura.

Esempio di procedura

```
    …
    CALL m1
    MOV AX, 2
    RET   ; return to operating system.
m1  PROC
    MOV BX, 5
    RET   ; return to caller.
m1  ENDP
    …
```

Passaggio dei parametri

```
    …
    MOV AL, 1
    MOV BL, 2
    CALL m2
    CALL m2
    CALL m2
    CALL m2
    RET ; return to operating system.
m2  PROC
    MUL BL ; AX = AL * BL.
    RET ; return to caller.
m2  ENDP
    …
```


Passaggio dei parametri tramite stack

```
    …
    MOV AX, VAR1
    PUSH AX
    MOV AX, VAR2
    PUSH
    CALL F1
    MOV BX, AX
    ; BX ← valore
    ; restituito da F1
    RET
```

```
F1  PROC
    PUSH BP
    MOV BP,SP
    MOV AX,6[BP]
    ;AX ← 1° par.
    ADD AX,4[BP]
    ;AX += 2° par.
    MOV SP,BP
    POP BP
F1  ENDP
```

Programma completo

```
.data
   pkey db "press any key...$“
   … ; altre variabili
```

```
.code
   .startup
; … codice …

    lea dx, pkey
    mov ah, 9
    int 21h
    mov ah, 1
    int 21h
   .exit
end
```