L'Assembler 8086

Procedure e Macro

M. Rebaudengo - M. Sonza Reorda

Politecnico di Torino
Dip. di Automatica e Informatica

Le procedure

L'Assembler 8086 permette l'uso delle *procedure*, in modo analogo a quanto avviene nei linguaggi di alto livello.

Attraverso le *procedure* è possibile scrivere una volta sola quelle parti di codice che vengono ripetutamente eseguite in un programma, ottenendo due vantaggi:

- **maggior leggibilità del codice**
- **risparmio di tempo per il programmatore**
- **risparmio di memoria occupata dal codice.**

Definizione di Procedura

Per definire una procedura si utilizzano le direttive PROC e ENDP.

```
label     PROC tipo
          ...            ; corpo della
          ...            ; procedura
label     ENDP
```

Il campo *label* corrisponde al nome della procedura, il tipo può essere NEAR o FAR.

Una procedura di tipo NEAR è richiamabile solo all'interno dello stesso segmento di codice, mentre una procedura di tipo FAR può essere chiamata da procedure appartenenti a segmenti di codici diversi.

Se il tipo della procedura non è specificato, l'assemblatore assume che sia NEAR.

3 M. Rebaudengo, M. Sonza Reorda

Chiamata di una procedura

L'istruzione CALL trasferisce il controllo del flusso del programma ad una procedura.

<u>Formato</u>

```
CALL target
```

<u>Funzionamento</u>

L'istruzione CALL esegue le seguenti operazioni:

- **salva nello stack *l'indirizzo di ritorno***
- **trasferisce il controllo all'operando *target*.**

L'indirizzo di ritorno è l'indirizzo dell'istruzione successiva a quella di CALL.

Ad essa la procedura *ritorna* al termine della sua esecuzione.

4 M. Rebaudengo, M. Sonza Reorda

Tipi di procedure

L'istruzione `CALL` si comporta diversamente in base al tipo di procedura chiamata:

- **se la procedura chiamata è di tipo `NEAR`, carica nello stack solo il contenuto dell'Instruction Pointer (`IP`), cioè l'indirizzo di offset dell'istruzione successiva;**
- **se la procedura chiamata è di tipo `FAR`, carica nello stack prima il contenuto del registro di segmento `CS` e poi il contenuto del registro `IP`.**

5 **M. Rebaudengo, M. Sonza Reorda**

Operando *target*

L'operando target può essere:

- **un *indirizzo diretto***
- **un *indirizzo indiretto*.**

Nel caso di indirizzo diretto, l'operando target è costituito dal nome stesso della procedura.

Nel caso di indirizzo indiretto, *un registro base o di indice* specifica l'indirizzo della procedura da eseguire.

6 **M. Rebaudengo, M. Sonza Reorda**

Operando *target* indiretto

Nel caso di indirizzamento di tipo indiretto è compito del programmatore fare in modo che l'assemblatore conosca il tipo della procedura.

Per fare ciò occorre utilizzare l'operatore `PTR`:

- **se la procedura è di tipo `NEAR` occorre usare l'operatore `WORD PTR`;**
- **se la procedura è di tipo `FAR` occorre usare l'operatore `DWORD PTR`.**

7 M. Rebaudengo, M. Sonza Reorda

Esempio

```
          .DATA
ADDR1     DW        DISPLAY1
          ...
          .CODE
          ...
          LEA       BX, ADDR1
          CALL      WORD PTR [BX]
          ...
```

8 M. Rebaudengo, M. Sonza Reorda

Esempio (II)

```
          .DATA
ADDR2     DD        DISPLAY2
          ...
          .CODE
          ...
          LEA       BX, ADDR2
          CALL      DWORD PTR [BX]
          ...
```

9 M. Rebaudengo, M. Sonza Reorda

L'istruzione RET

L'istruzione `RET` permette di restituire il controllo alla procedura chiamante, una volta che la procedura chiamata ha terminato l'esecuzione.

<u>Formato</u>

`RET pop_value`

L'operando *`pop_value`* è opzionale e corrisponde ad un valore immediato; esso permette di eseguire l'operazione di liberazione dello stack al momento del ritorno alla procedura chiamante di un numero di byte pari a *`pop_value`*.

10 M. Rebaudengo, M. Sonza Reorda

L'istruzione RET

L'istruzione `RET` assume che l'indirizzo di ritorno si trovi in cima allo stack.

L'istruzione `RET` esegue le seguenti operazioni:

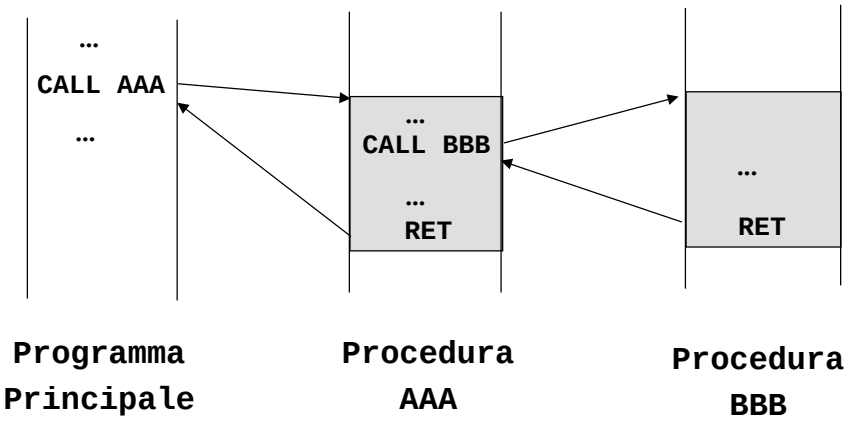
- ***pop* dallo stack dell'*indirizzo di ritorno***
- **estrazione dallo stack di *pop_value* byte**
- ***salto* all'indirizzo di ritorno.**

Se la procedura è di tipo `NEAR` il processore preleva dallo stack una word contenente l'offset dell'indirizzo di ritorno.

Se la procedura è di tipo `FAR` il processore preleva dallo stack due word equivalenti all'intero indirizzo di ritorno.

11 M. Rebaudengo, M. Sonza Reorda

CALL e RET



12 M. Rebaudengo, M. Sonza Reorda

Procedure Annidate

L'uso dello stack permette l'esecuzione di procedure annidate.

Il massimo livello di annidamento permesso è limitato dalle dimensioni dello stack.

13 M. Rebaudengo, M. Sonza Reorda

Salvataggio dei Registri

Occorre fare in modo che ogni procedura esegua come prima operazione il salvataggio nello stack di tutti i registri che vengono da essa modificati.

Al termine essa ripristina i valori originari.

Lo stack è una coda di tipo *LIFO*: l'ordine delle istruzioni `POP` deve essere l'inverso di quello delle istruzioni `PUSH`.

<u>Esempio</u>

```
XXX   PROC
      PUSH AX
      PUSH DX
      ...
      POP  DX
      POP  AX
      RET
XXX   ENDP
```

14 M. Rebaudengo, M. Sonza Reorda

Uso di PUSHA e POPA

A partire dall'80186 è possibile utilizzare le istruzioni PUSHA e POPA per salvare il contenuto di tutti i registri.

Esse non vanno usate nel caso in cui la procedura debba ritornare un valore attraverso un registro.

15 M. Rebaudengo, M. Sonza Reorda

Passaggio di Parametri

Le procedure comunicano attraverso i parametri.

Esistono diverse tecniche per effettuare il passaggio dei parametri, classificabili in base al metodo o in base al tramite.

I possibili <u>metodi</u> con cui i parametri vengono trasferiti sono:

- **una copia del valore del parametro (*by value*)**
- **l'indirizzo del parametro (*by reference*).**

I possibili <u>tramiti</u> attraverso i quali avviene il trasferimento dei dati sono:

- **le variabili globali**
- **i registri**
- **lo stack.**

16 M. Rebaudengo, M. Sonza Reorda

By value o *by reference*

In un passaggio *by value*, la procedura chiamante passa a quella chiamata una copia del parametro. Ogni possibile modifica del parametro all'interno della procedura modifica esclusivamente tale copia. La procedura chiamante non vede le modifiche effettuate sul parametro dalla procedura chiamata.

In un passaggio *by reference*, la procedura chiamante passa alla procedura chiamata l'indirizzo del parametro: ogni modifica del parametro effettuata da parte della procedura chiamata si ripercuote sul valore del parametro per la procedura chiamante.

17 M. Rebaudengo, M. Sonza Reorda

Caso di studio

```
int som_vett(int *vett, int count)
{
 int i, somma = 0;
     for (i=0 ; i < count ; i++)
          somma += vett[i];
     return(somma);
}
```

18 M. Rebaudengo, M. Sonza Reorda

Variabili globali

Il modo più semplice per passare i parametri alle procedure consiste nell'utilizzare variabili globali, accessibili sia alla procedura chiamante sia a quella chiamata.

Questo metodo è estremamente semplice, ma sconsigliabile in quanto le procedure che utilizzano variabili globali come parametri sono poco riutilizzabili poiché la stessa procedura non può lavorare su dati diversi.

19 M. Rebaudengo, M. Sonza Reorda

Esempio

```
LUNG EQU  100
     .MODEL    small
     .STACK
     .DATA
VETT DW   LUNG DUP (?)
SOM  DW   ?
     ...
     .CODE
     ...
     CALL   SOM_VETT
     ...
```

```
SOM_VETT  PROC
          PUSH SI
          PUSH AX
          PUSH CX
          XOR  SI, SI
          XOR  AX, AX
          MOV  CX, LUNG
ciclo:    ADD  AX, VETT[SI]
          ADD  SI, 2
          LOOP ciclo
          MOV  SOM, AX
          POP  CX
          POP  AX
          POP  SI
          RET
SOM_VETT  ENDP
```

20 M. Rebaudengo, M. Sonza Reorda

Uso dei registri

I parametri di ingresso ed uscita possono essere letti e scritti utilizzando i registri general purpose.

È un metodo semplice ed estremamente rapido, ma utilizzabile solo quando i parametri sono in numero limitato.

21 M. Rebaudengo, M. Sonza Reorda

Esempio

```
LUNG  EQU 100
      .MODEL   small
      .STACK
      .DATA
VETT  DW  LUNG DUP (?)
SOMMA DW  ?
      .CODE
          …
      MOV  AX, LENGTH VETT
      LEA  BX, VETT
      CALL SOM_VETT
      MOV  SOMMA, AX
          …
```

```
SOM _VETT PROC
          PUSH BX
          PUSH CX
          MOV  CX, AX
          XOR  AX, AX
ciclo:    ADD  AX, [BX]
          ADD  BX, 2
          LOOP ciclo
          POP  CX
          POP  BX
          RET
SOM_VETT  ENDP
```

22 M. Rebaudengo, M. Sonza Reorda

Uso dello stack

Il metodo più utilizzato per il passaggio dei parametri si basa sullo stack.

Tale metodo non ha limiti sul numero di parametri (il limite sta nella dimensione dello stack).

Inoltre, la memoria richiesta per i parametri è allocata solo per il tempo corrispondente all'esecuzione della procedura.

23 M. Rebaudengo, M. Sonza Reorda

Caricamento dei dati nello stack

Prima dell'esecuzione dell'istruzione `CALL`, la procedura chiamante deve eseguire tante istruzioni di `PUSH` quanti sono i parametri da passare.

<u>Esempio</u>

```
…
PUSH LUNG
PUSH OFFSET VETT
CALL SOM_VETT
…
```

24 M. Rebaudengo, M. Sonza Reorda

Lettura dei parametri di ingresso

L'istruzione CALL è eseguita dopo che i parametri sono caricati nello stack; l'indirizzo di ritorno è caricato nello stack dopo tutti i parametri e si trova quindi in cima allo stack nel momento in cui la procedura inizia l'esecuzione.

Ciò significa che la procedura chiamata non può eseguire l'operazione di pop dallo stack per prelevare i parametri senza perdere l'indirizzo di ritorno.

25 M. Rebaudengo, M. Sonza Reorda

Lettura dei parametri di ingresso

(segue)

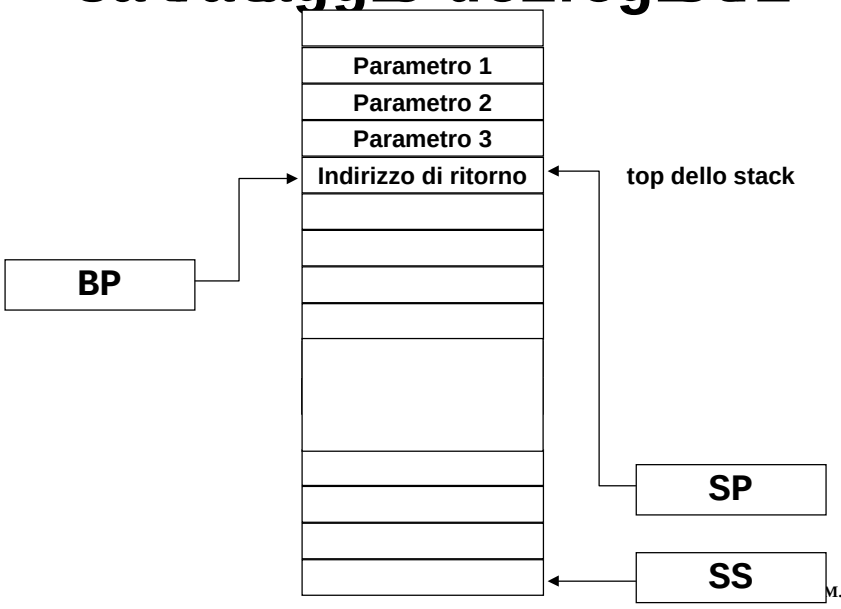
Per la lettura dei parametri si utilizza il registro *Base Pointer* (BP) per fare accesso allo stack.

Il registro BP permette di accedere ai dati presenti nello stack senza eseguire operazioni di *push* e di *pop*, ossia senza cambiare il contenuto del registro SP.

La prima operazione da effettuare all'interno di una procedura è copiare il valore del registro SP nel registro BP.

26 M. Rebaudengo, M. Sonza Reorda

Stato dello stack prima del salvataggio dei registri



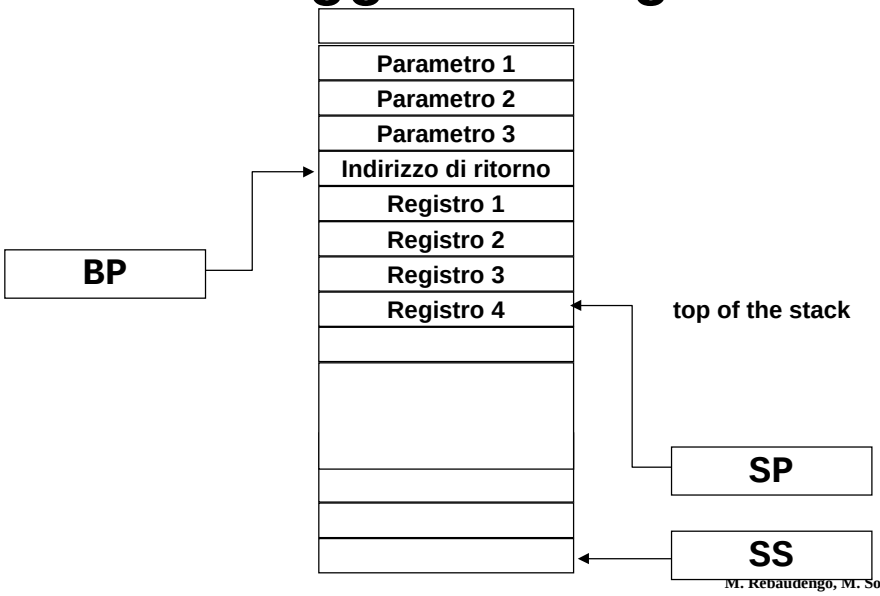
27 M. Sonza Reorda

Salvataggio dei registri

Una volta che il registro BP è caricato, la procedura chiamata può salvare i registri nello stack.

28 M. Rebaudengo, M. Sonza Reorda

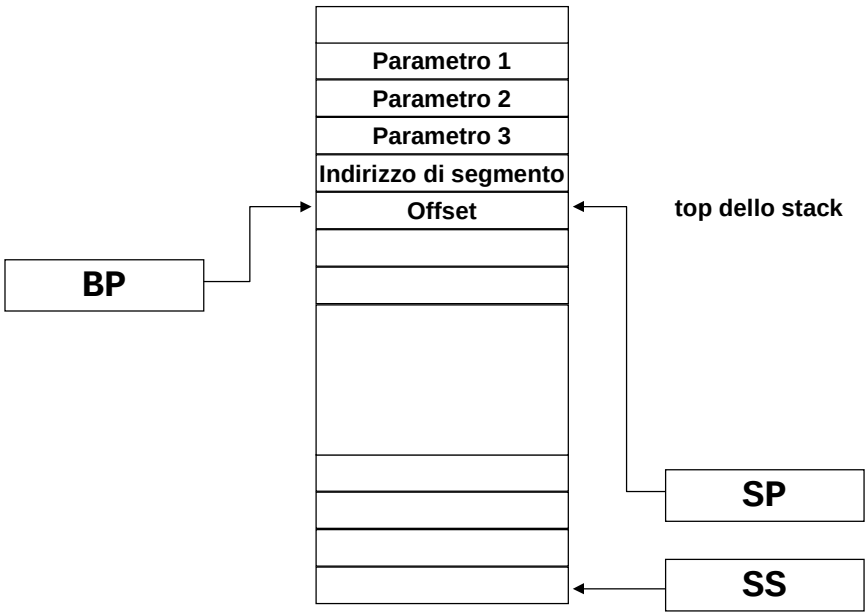
Stato dello stack dopo il salvataggio dei registri



29

M. Rebaudengo, M. Sonza Reorda

Procedura di tipo FAR



30

M. Rebaudengo, M. Sonza Reorda

Parametri in uscita

È possibile utilizzare lo stack anche per passare alla procedura chiamante i parametri di uscita.

Essi non possono essere caricati nello stack con un'operazione di push perché altrimenti sarebbero posizionati in cima allo stack e non permetterebbero un corretto ritorno alla procedura chiamante.

Anche per la scrittura dei parametri nello stack è necessario utilizzare il registro BP.

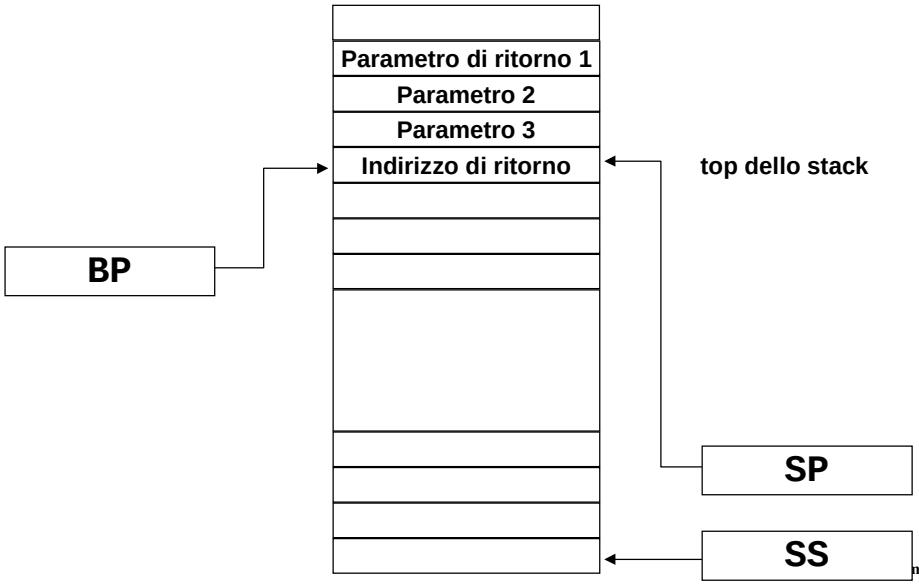
È compito della procedura chiamante eseguire le opportune operazioni di pop per la lettura dei valori di ritorno.

<u>Esempio</u>

```
MOV  [BP+6], SOMMA
```

31 M. Rebaudengo, M. Sonza Reorda

Stack prima della chiamata della RET



32 nza Reorda

Liberazione dello stack

Di norma è compito della procedura chiamante liberare lo stack, cancellando le parole che sono state utilizzate per il passaggio dei parametri.

La liberazione dello stack può essere fatta:

- **con successive operazioni di *pop***
- **incrementando opportunamente il valore di `SP`.**

<u>Esempio</u>

```
PUSH PARAM1
PUSH PARAM2
PUSH PARAM3
CALL MY_PROC
POP  DX
POP  DX
POP  DX
```

```
PUSH PARAM1
PUSH PARAM2
PUSH PARAM3
CALL MY_PROC
ADD  SP, 6
```

33

M. Rebaudengo, M. Sonza Reorda

Liberazione dello stack

(segue)

La pulizia dello stack può anche essere fatta all'interno della procedura chiamante mediante l'esecuzione dell'istruzione `RET`.

Questa può avere un operando immediato, che rappresenta il numero di byte da togliere dallo stack.

34

M. Rebaudengo, M. Sonza Reorda

Esempio

```
LUNG    EQU  100
        .MODEL     small
        .STACK
        .DATA
VETT    DW   LUNG DUP (?)
SOMMA   DW         ?
TEMP    DW         ?
        .CODE
        MOV  AX, LUNG
        LEA  BX, VETT
        SUB  SP, 2
        PUSH AX
        PUSH BX
        CALL SOM_VETT
        ADD  SP, 4
        POP  SOMMA
```

```
SOM_VETT    PROC
            MOV  BP, SP
            PUSH BX
            PUSH CX
            PUSH AX
            MOV   CX, [BP+4]
            MOV   BX, [BP+2]
            XOR   AX, AX
ciclo:      ADD   AX, [BX]
            ADD   BX, 2
            LOOP  ciclo
            MOV   [BP+6], AX
            POP   AX
            POP   CX
            POP   BX
            RET
SOM_VETT    ENDP
```

35 M. Rebaudengo, M. Sonza Reorda

Condizione di errore

Un'informazione importante che la procedura chiamata deve ritornare è relativa al successo o all'insuccesso dell'esecuzione della procedura stessa.

Un possibile modo per segnalare se una procedura è terminata correttamente consiste nell'utilizzare il flag di *carry*:

- **se la procedura termina correttamente il flag `CF` viene azzerato prima di restituire il controllo;**
- **se all'interno della procedura è avvenuto un errore, il flag `CF` viene posto al valore `1`.**

È compito della procedura chiamante controllare il valore del flag `CF` mediante le istruzioni di salto condizionato. per verificare la corretta esecuzione della procedura.

36 M. Rebaudengo, M. Sonza Reorda

Esercizio

Si realizzi una procedura che esegua la somma di un vettore di numeri interi positivi; se è stata riscontrata una condizione di errore, la procedura chiamante deve visualizzare un messaggio di errore.

```
void main( void)
{     int dati[100], somma;
      somma = som_vett(dati, 100);
      if (somma == -1)
            printf("Errore nei dati\n");
}
int som_vett(int *vett, int count)
{     int i, somma = 0;
      for (i=0 ; i<count; i++)
            if (dati[i] >= 0)
                  somma += dati[i];
            else return(-1);
      return(somma);
}
```


M. Rebaudengo, M. Sonza Reorda

Soluzione Assembler

```
LUNG        EQU   100
            .MODEL      small
            .STACK
            .DATA
DATI        DW    LUNG DUP (?)
SOMMA       DW          ?
TEMP        DW          ?
MSG         DB    "Errore nei dati"
            DB    0DH,0AH,"$"
            .CODE
            MOV   AX, LUNG
            LEA   BX, LIST
            PUSH  AX
            PUSH  BX
            CALL  SOM_VETT
                                    POP   TEMP
                                    POP   TEMP
                                    MOV   SOMMA, AX
                                    JNC   fine
                                    LEA   DX, MSG
                                    MOV   AH, 09H
                                    INT   21H
                              fine: ...
```


M. Rebaudengo, M. Sonza Reorda

```
SOM_VETT    PROC
            MOV   BP, SP
            PUSH  BX
            PUSH  CX
            PUSH  DX
            MOV   CX, [BP+4]
            MOV   BX, [BP+2]
            XOR   AX, AX
ciclo:      MOV   DX, [BX]
            CMP   DX, 0
            JNL   ok
            STC
            JMP   fin

ok:         ADD   AX, [BX]
            ADD   BX, 2
            LOOP  ciclo
            CLC
fin:        POP   DX
            POP   CX
            POP   BX
            RET
SOM_VETT    ENDP
```

39 M. Rebaudengo, M. Sonza Reorda

Fram e diuna Procedura

La chiamata di una procedura causa nel caso generale l'inserzione nello stack di:

- **parametri**
- **indirizzo di ritorno**
- **registri**
- **variabili locali.**

L'insieme delle celle di memoria nello stack occupate da tali valori prende il nome di *frame*, e della sua dimensione si deve tenere conto per il dimensionamento dello stack.

40 M. Rebaudengo, M. Sonza Reorda

Esercizio: tabelle di jump

Si voglia realizzare un programma che deve elaborare un vettore di numeri interi. Il tipo di operazione da svolgere è scelto in base al valore fornito dall'utente tramite tastiera.

41 M. Rebaudengo, M. Sonza Reorda

Soluzione 1

```
LUNG        EQU   100
            .DATA
SCELTA      DB    ?
VETT        DB    LUNG DUP (?)
            .CODE
            MOV   AH, 1
            INT   21H
            MOV   SCELTA, AL
            CMP   SCELTA, 1
            JE    lab1
            CMP   SCELTA, 2
            JE    lab2
            CMP   SCELTA, 3
            JE    lab3
            CMP   SCELTA, 4
            JE    lab4
            JMP   lab4
lab1:       CALL  SOMMA_VETT
            JMP   lab5
lab2:       CALL  MEDIA_VETT
            JMP   lab5
lab3:       CALL  MAX_VETT
            JMP   lab5
lab4:       CALL  MIN_VETT
lab5:       ...
```

42 M. Rebaudengo, M. Sonza Reorda

Soluzione 2

```
LUNG            EQU   100
                .DATA
JUMP_TABLE      DW    SOMMA_VETT
                DW    MEDIA_VETT
                DW    MAX_VETT
                DW    MIN_VETT
SCELTA          DB    ?
VETT            DB    LUNG DUP (?)
                .CODE
                MOV   AH, 1
                INT   21H
                MOV   SCELTA, AL
                LEA   BX, JUMP_TABLE
                MOV   AX, SCELTA
                DEC   AX
                SHL   AX, 1
                ADD   BX, AX
                CALL  WORD PTR [BX]
```

43

M. Rebaudengo, M. Sonza Reorda

Macro

Una *macro* è un nome simbolico che il programmatore dà ad una serie di caratteri (il *testo* della macro) o ad una serie di istruzioni (*macro procedure* o *macro funzioni*). Quando l'assemblatore pre-processa il programma, ricerca i nomi delle macro precedentemente definite e quando ne trova una, sostituisce al nome della macro il relativo testo.

In questo modo il programmatore può evitare di ripetere lo stesso pezzo di codice più volte nel suo programma.

44

M. Rebaudengo, M. Sonza Reorda

Definizione e Chiamata di una Macro

Formato

```
nome MACRO
     testo
ENDM
```

Esempio

```
beep MACRO
     MOV  AH, 2
     MOV  DL, 7
     INT  21H
ENDM
```

Chiamata della Macro

```
     ...
     beep
     ...
```

Espansione della macro

```
     ...
     MOV  AH, 2
     MOV  DL, 7
     INT  21H
     ...
```

45 M. Rebaudengo, M. Sonza Reorda

Parametri di una Macro

Definizione

```
writechar MACRO char
     MOV  AH, 2
     MOV  DL, char
     INT  21h
ENDM
```

Chiamata

```
writechar 7
writechar 'A'
```

46 M. Rebaudengo, M. Sonza Reorda

Simboli locali ad una Macro

Quando si definisce un'etichetta all'interno di una macro, è necessario comunicare all'assemblatore che tale simbolo è locale.

L'assemblatore provvede allora a ridenominare in modi diversi l'etichetta in corrispondenza di ogni espansione di macro, evitando così la ripetizione del nome.

47 M. Rebaudengo, M. Sonza Reorda

Esempio

```
Power      MACRO factor, exponent
           LOCAL again, gotzero
           XOR   DX, DX
           MOV   AX, 1
           MOV   CX, exponent
           JCXZ  gotzero
           MOV   BX, factor
again:     MUL   BX
           LOOP again
gotzero:
           ENDM
```

48 M. Rebaudengo, M. Sonza Reorda

Le direttive PUBLIC e EXTRN

Le direttive PUBLIC e EXTRN permettono di definire ed utilizzare variabili globali esterne. Per variabili globali esterne si intendono quelle entità (variabili o procedure) definite ed utilizzate in diversi moduli sorgenti.

La direttiva PUBLIC permette di specificare all'assemblatore quali sono le variabili globali, cioè quali nomi possono essere richiamati dall'interno di altri moduli.

La direttiva EXTRN specifica all'assemblatore quali nomi, definiti in un altro modulo, sono utilizzati all'interno del modulo.

49 M. Rebaudengo, M. Sonza Reorda

Le direttive PUBLIC e EXTRN

(segue)

Ciascuna variabile globale richiede quindi due direttive:

- **una direttiva PUBLIC nel modulo in cui è definita**
- **una direttiva EXTRN nel modulo in cui è usata.**

Formato

```
PUBLIC nome
EXTRN nome:tipo
```

Se il nome rappresenta una variabile il tipo può essere BYTE, WORD, DWORD, QWORD o TBYTE; se il nome rappresenta un nome di procedura, il tipo può essere NEAR o FAR.

50 M. Rebaudengo, M. Sonza Reorda

Esempio

Si realizzi un programma che esegue il caricamento di una stringa da tastiera. Tale stringa, una volta convertita in caratteri maiuscoli, deve essere visualizzata su video. Un errore nel caricamento della stringa deve essere segnalato mediante un messaggio su video.

Il programma è ripartito in 3 moduli (o file):

- **Modulo I: programma principale**
- **Modulo II: procedura `CONV_MAIU`**
- **Modulo III: procedure `VISU_STR` e `LOAD_STR`.**

51 M. Rebaudengo, M. Sonza Reorda

Modulo principale

```
EXTRN   LOAD_STR:NEAR, CONV_MAIU:NEAR, VISU_STR:NEAR
PUBLIC  MSG
       .MODEL small
       .STACK
       .DATA
MSG DB  "Stringa terminata da un <CR>",0DH,0AH,"$"
STRING  DB  80 DUP (?)
        .CODE
        .STARTUP
        LEA BX, STRING
        MOV AX, LENGTH STRING
        PUSH AX
        PUSH BX
        CALL LOAD_STR
        JC esci
        PUSH BX
        CALL CONV_MAIU
        PUSH BX
        CALL VISU_STR
esci:   .EXIT
        END
```

52 M. Rebaudengo, M. Sonza Reorda

Modulo II

```
PUBLIC      CONV_MAIU
            .MODEL small
            .CODE
CONV_MAIU   PROC NEAR
            MOV   BP, SP
            PUSH  BX
            MOV   BX, [BP+2]
ciclo:      CMP   BYTE PTR [BX], "$"
            JE    fine
            CMP   BYTE PTR [BX], 'z'
            JG    last
            CMP   BYTE PTR [BX], 'a'
            JL    last
            ADD   BYTE PTR [BX], 'A'-'a'
last:       INC   BX
            JMP   ciclo
fine:       POP   BX
            RET   2
CONV_MAIU   ENDP
            END
```

53 M. Rebaudengo, M. Sonza Reorda

Modulo III

```
PUBLIC      LOAD_STR, VISU_STR
EXTRN       MSG:BYTE
            .MODEL      small
            .DATA
MSG2 DB     0DH,0AH,"STRINGA TROPPO LUNGA",0DH,0AH,"$"
            .CODE
VISU_STR    PROC  NEAR
            MOV   BP, SP
            PUSH  AX
            PUSH  DX
            MOV   DX, [BP+2]
            MOV   AH, 09H
            INT   21H
            POP   DX
            POP   AX
            RET   2
VISU_STR    ENDP
```

54 M. Rebaudengo, M. Sonza Reorda

Modulo III

(segue)

```
LOAD_STR    PROC  NEAR
            MOV   BP, SP
            PUSH  AX
            PUSH  BX
            PUSH  CX
            PUSH  DX
            MOV   CX, [BP+4]
            DEC   CX
            MOV   BX, [BP+2]
            LEA   DX, MSG
            MOV   AH, 09H
            INT   21H
ciclo:      MOV   AH, 1
            INT   21H
            MOV   [BX], AL
            INC   BX
            CMP   AL, 13
            LOOPNE      ciclo
```

```
            JE    ok
            STC
            LEA   DX, MSG2
            MOV   AH, 09H
            INT   21H
            JMP   fine
ok:         MOV   [BX-1], "$"
            CLC
fine:       POP   DX
            POP   CX
            POP   BX
            POP   AX
            RET   4
LOAD_STR    ENDP
            END
```

55

M. Rebaudengo, M. Sonza Reorda