

Assembler

8086



Prof. Ivaldi Giuliano

Sommario

PREMESSA.....	3
STRUTTURA DI UN PROGRAMMA ASSEMBLER.....	3
LE VARIABILI.....	4
ISTRUZIONI BASILARI	4
INTERFACCIAMENTO CON L'UTENTE.....	5
COSTRUTTI FONDAMENTALI	7
I VETTORI.....	10
GESTIONE DELLA PORTA PARALLELA	13
LE PROCEDURE.....	16
LO STACK.....	16

PREMESSA

Per poter affrontare adeguatamente lo studio del linguaggio Assembler sarebbe necessario avere già un'idea del funzionamento di un microprocessore. Ogni microprocessore poi possiede un certo set di istruzioni, cioè può eseguire un numero più o meno grande (ma comunque finito) di determinate operazioni. Un programma è costituito da una sequenza di dette istruzioni che permette al microprocessore di assolvere un particolare compito di calcolo e/o controllo. Le istruzioni che il microprocessore deve leggere ed eseguire sono naturalmente immagazzinate nella memoria in forma di codice binario ovvero sono espresse in quello che si chiama *linguaggio macchina*. Ogni istruzione è costituita da un certo numero di byte ed il programma nel suo insieme è una successione di byte che va ad occupare una certa porzione di memoria. Redigere un programma direttamente in codice binario non è per nulla agevole ed è suscettibile di facili errori; l'uso del linguaggio Assembler consente di superare tali difficoltà con l'adozione di una forma simbolica (*codice mnemonico*) che richiama con una notazione sintetica il modo di operare di ogni istruzione. Tale linguaggio è pertanto già orientato verso l'uomo, che lo può usare più agevolmente del linguaggio macchina e tuttavia di quest'ultimo conserva tutti i vantaggi di sintesi e velocità di esecuzione, in quanto ad ogni istruzione in linguaggio Assembler corrisponde una sola istruzione in linguaggio macchina, ciò non vale per i linguaggio ad alto livello in cui ad una singola istruzione possono corrispondere più istruzioni in linguaggio macchina, anche se permettono una più semplice stesura dei programmi. La corrispondenza uno ad uno fra istruzione in linguaggio Assembler ed istruzione in linguaggio macchina vieta la possibilità di un unico linguaggio Assembler che, pertanto, è diverso da microprocessore a microprocessore.

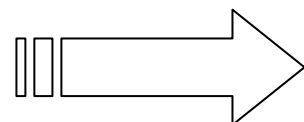
Per iniziare ad esempio, verranno indicati di seguito i passaggi per realizzare un programma in Assembler 8086 in ambiente DOS utilizzando come compilatore il MASM 8086 della Microsoft.

Realizzare un programma (pgm) in Assembler comporta i seguenti passaggi :

1. EDITOR : serve per scrivere il programma e si può utilizzare uno qualunque fra quelli disponibili (salvando il programma con l'estensione .asm)
Es. editor del DOS, Turbo Pascal , Turbo C
2. COMPILAZIONE : MASM nome_pgm.asm
Es. masm pippo.asm/zi
3. LINKER : LINK nome_pgm.obj
Es. link pippo.obj/co
4. ESEGUIRE IL PROGRAMMA : nome_pgm.exe
Es. pippo.exe
5. DEBUGGARE IL PROGRAMMA (se non funziona): CV nome_pgm.exe
Es. cv pippo.exe

STRUTTURA DI UN PROGRAMMA ASSEMBLER

```
.model small          ; dimensione programma
.stack               ; dichiarazione segmento di stack
.data               ; dichiarazione segmento dei dati
.                   ;
.                   ; dichiarazione variabili
.                   ;
.code               ; dichiarazione segmento del codice
    mov ax,@data    ; assegnamento indirizzo
    mov ds,ax       ; segmento data a ds
.                   ;
.                   ; corpo del programma
.                   ;
fine:
    mov ah,4Ch      ; ritorna al sistema
    mov al,00h      ; operativo
    int 21h
end
```



LE VARIABILI

- tipi essenziali : byte, contiene valori numerici da 0 a 255
word, contiene valori numerici da 0 a 65535 (come tipo 'unsigned int' in C)
- definizione : db define byte (1 byte = 8 bit)
dw define word (2 byte = 16 bit)
- vettori : direttiva → dup (i vettori verranno trattati in una sezione successiva)

Es.

```
.model small
.stack
.data
    var1 db ?           ; def. var1 di tipo byte e non la inizializza
    var2 db 15          ; def. var2 di tipo byte e le assegna 15
    var3 dw ?           ; def. var3 di tipo word e non la inizializza
    vettore db 5 dup(?) ; def. un vettore di 5 elementi di tipo byte
                        ; non inizializzato
.code
    ...                ; resto del programma
end
```

ISTRUZIONI BASILARI

- di spostamento, MOV dest,sorg (dest = destinatario , sorg = sorgente)
Es. mov al,bl
dest e sorg NON possono essere entrambi contemporaneamente due variabili
- operazioni aritmetiche
 1. somma : ADD dest,sorg → dest=dest+sorg
Es. add al,bl → al = al +bl
 2. sottrazione : SUB dest,sorg → dest=dest-sorg
Es. sub al,bl → al = al-bl
 3. moltiplicazione : MUL sorg → ax = al*sorg
Es. mov al,5
mul 6 → ax=al*6=30
mov risultato,ax
risultato deve essere di tipo word (DW)
 4. divisione : DIV divisore → al=ax/divisore
ah=resto
Es. mov ax,20
div 4 → al = ax/4 = 5
ah = 0
mov quoziente,al
mov resto,ah
- di incremento e decremento
 - INC sorg → sorg = sorg + 1
Es. inc var → var = var + 1
 - DEC sorg → sorg = sorg - 1
Es. dec cx → cx = cx - 1
- di confronto, CMP op1,op2 ; op1>op2 ?
- di salto,
 - JA etichetta ; salta se op1>op2 (Jump Above)
 - JAE etich. ; op1>=op2 (Jump Above or Equal)
 - JB etich. ; op1<op2 (Jump Below)
 - JBE etich. ; op1<=op2 (Jump Below or Equal)
 - JE etich. ; op1=op2 (Jump Equal)
 - JNE etich. ; op1<>op2 (Jump Not Equal)
 - incondizionato JMP etich ; salta sempre

INTERFACCIAMENTO CON L'UTENTE

- Visualizzazione di un carattere¹
mov al,codice ASCII carattere
mov ah,0Eh
int 10h

Es. visualizza carattere (>)
mov al,'>'
mov ah,0Eh
int 10h

- Inserimento di un carattere da tastiera
mov ah,00h
int 16h

(il carattere premuto viene messo in al automaticamente e se è necessario lo si può salvare in una variabile così -> mov variabile,al)
Se si deve inserire un numero allora si aggiunge : sub variabile,30h
(se lo si deve stampare, add variabile,30h)

Es. - mov ah,00h
int 16h
cmp al,'A' ; tasto premuto = 'A' ?
je etich ; SI : salta a etich
... ; NO : continua

- mov ah,00h
int 16h
sub al,30h
cmp al,1 ; n° inserito = 1 ?
je etich_uno ; SI : salta a etich_uno
... ; NO : continua

N.B. : per cosa stanno int 10h,int 16h o int 21h?
L'istruzione 'int'² sta per

interrupt = interruzione

Senza entrare adesso nello specifico delle interruzioni,sappiate che si tratta di routine,cioè programmini già belli che pronti e che non vi resta che usare nel modo appena descritto e solo in quello!
Non si possono cambiare i registri utilizzati ! Dovete fare un 'copia e incolla' delle istruzioni scritte più sopra.

¹ Esistono diversi modi per stampare un carattere,uno più completo è il seguente che permette di stampare N volte un carattere con un dato colore nella pagina video attuale,la n°0

```
mov al,codice ascii carattere
mov bh,n°pagina      (=00h)
mov bl,colore        (da 0 a 15)
mov cx,n°ripetizioni
mov ah,09h
int 10h
```

² Esiste una differenza importante fra int 10h e 16h e l'int 21h : i primi due fanno parte del BIOS del PC che si trova nella ROM e che è indipendente dal sistema operativo usato Windows, Linux, etc..., mentre il terzo, cioè l'int 21h è un programmino del DOS e quindi di Windows, perciò può funzionare solo sotto tali sistemi operativi, ma non esiste ad esempio in Linux.

Quindi sotto DOS e Windows potreste provare anche questi servizi :

; input di un carattere	; input di un carattere senza visualizzazione	; visualizzazione di un carattere
mov ah,01h	mov ah,07h	mov dl,codice Ascii carattere
int 21h	int 21h	mov ah,02h
mov variabile,al	mov variabile,al	int 21h

- Visualizzazione di un messaggio

Es. visualizza stringa mess1

```
.model small
.stack
.data
mess1 db 'Inserisci un numero : ',00h
.code
    mov ax,@data
        mov ds,ax

    mov bx,offset mess1
    mov di,0000h
visualizza:
    mov al,byte ptr[bx+di]
    cmp al,00h           ; al = fine?
    je continua         ; SI: esci dal ciclo

    mov ah,0Eh          ; NO: stampa il
    int 10h             ; carattere
    inc di
    jmp visualizza

continua:
    ...
fine:
    mov ah,4Ch
    mov al,00h
    int 21h

end
```



COSTRUTTI FONDAMENTALI

- if ... then ... else

```
        cmp op1,op2           ; op1>op2 ?
        ja  etich_then        ; SI : allora fai then
etich_else:
        ...                   ; NO : allora fai else
        jmp continua         ; istruzioni else
etich_then :
        ...                   ; istruzioni then
continua:
        ...                   ; resto del programma
```

- ciclo do ... while

```
inizio_ciclo:
        ...                   ; istruzioni ciclo
        cmp op1,op2           ; op1 = op2 ?
        je  inizio_ciclo      ; SI : allora resta nel ciclo
        fine_ciclo:          ; NO : esci dal ciclo
        ...                   ; resto del programma
```

- ciclo while

```
inizio_ciclo:
        cmp op1,op2           ; op1 = op2 ?
        je  fine_ciclo        ; SI : allora vai a fine_ciclo ed esci dal ciclo
        .                     ; ALTRIMENTI : fai le istruzioni
        .                     ; del ciclo
        .
        jmp inizio_ciclo
fine_ciclo:
        ...                   ; resto del programma
```

- ciclo for

```
        mov cx,n             ; in Assembly
inizio_ciclo:
        ...                   ; istruz. del ciclo
        loop inizio_ciclo
```

in alternativa con l'istruzione JUMP ...

```
        mov cx,0000h         ; in Assembly
inizio_ciclo:
        ...                   ; istruz. del ciclo
        inc cx
        cmp cx,n
        jne inizio_ciclo
```

```
for (cx=n;cx>0;cx--) /* in C */
{ /* inizio ciclo */
    ... ; istruz. del ciclo
} /* fine ciclo */
```

```
for (cx=0;cx!=n;cx++) /* in C */
{ /* inizio ciclo */
    ... ; istruz. del ciclo
} /* fine ciclo */
```

Es, pulizia schermo

```
mov cx,25
cancel:
    mov al,0Ah
    mov ah,0Eh
    int 10h
loop cancel
```

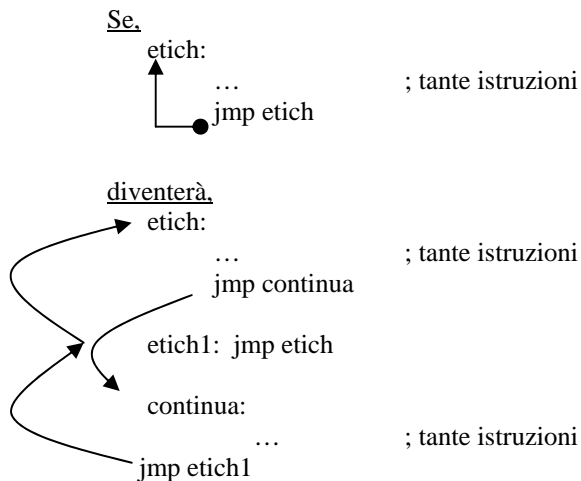
pulisce lo schermo generando 25 caratteri LINE FEED (= 0Ah , va a capo)

N.B.: I costrutti visti non sono da imparare a memoria!

Ciò che bisogna capire è come si costruiscono,combinando istruzioni CoMPare e JuMP a seconda delle proprie esigenze!

OSSERVAZIONI :

- in tutti i costrutti visti le istruzioni JUMP possono essere variate a seconda dei casi
- NON bisogna fare salti troppo lunghi !!! In tal caso,bisogna spezzare il salto in più parti in questo modo :



ESEMPI SUI CICLI

Tutti i costrutti si formano con istruzioni - CMP
- JMP

CICLO DO WHILE :

STAMPA DI 50 ASTERISCHI (do...while e for)

```
mov cl,00h           ; inizializza cl a 00h
inizio_ciclo:
    mov al,'*'       ; stampa asterisco
    mov ah,0Eh
    int 10h

    1 inc cl           ; incrementa cl di 1
    cmp cl,50        ; cl < 50 ?
    2 jb inizio_ciclo   ; SI : allora rimani nel ciclo
                                ; NO : allora esci dal ciclo

    ...altre istruzioni...
```

¹ In alternativa : add cl,01h

² In alternativa : je continua
jmp inizio_ciclo
continua:

CONTROLLO INSERIMENTO TASTI

(solo do...while)

```
inserim_numero:
mov ah,00h          ; inserim. carattere(n°)
int 16h

cmp al,30h         ; al < (30h='0')?
jb inserim_numero ; SI:salta a inserim_numero
cmp al,39h         ; al > (39h='9')?
ja inserim_numero ; SI:salta a inserim_numero

; NO : allora esci dal ciclo
sub al,30h         ; trasforma il carattere numerico in un numero
mov var,al        ; e lo salva in una variabile opportunamente definita
...altre istruzioni...
```

CICLO WHILE:

STAMPA DI 50 ASTERISCHI

(while e for)

```
mov cx,0000h      ; inizializza cx a 0000h
asterischi:
  cmp cx,50       ; cx = 50 ?
  je fine_asterischi ; SI : esci dal ciclo

  mov al,'*'      ; NO : stampa
  mov ah,0Eh      ; asterisco
  int 10h

  inc cx          ; incrementa cx
  jmp asterischi  ; salta a inizio ciclo
fine_asterischi:
  ...altre istruzioni...
```

CICLO FOR :

PULIZIA SCHERMO

```
mov cx,25         ; inizializza cx a 25
cancel:
  mov al,0Ah      ; stampa LINE FEED
  mov ah,0Eh
  int 10h
  loop cancel     ; decem. cx e salta a cancel
```



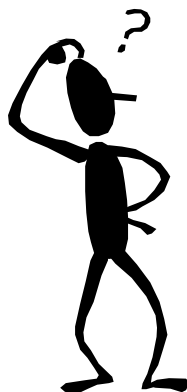
I VETTORI

Definizione:

nome_vettore db N dup(elem1,elem2,...,elemN)

Es. vetnum db 8 dup (6,2,8,3,9,1,7,2)
codice db 4 dup('c','i','a','o')

- Oss.
- def. un vettore di nome 'vetnum' di 8 elementi ed uno di nome 'codice' di 4 elementi di tipo byte inizializzati con i valori fra parentesi
 - al posto di 'db' si può usare 'dw' se necessario
 - se fra parentesi si mette un '?' gli elementi del vettore non vengono inizializzati
 - in assembler gli elementi di un vettore partono dalla posizione →0 e non dalla 1! (Come in C)



Uso dei vettori:

per lavorare con i vettori si usa il metodo di indirizzamento indiretto indicizzato, e precisamente...

- si carica l'offset del vettore in 'bx'
- si azzera uno dei registri indice 'di' o 'si'
- si tratta l'elemento di-esimo del vettore con il formalismo [bx+di], cioè [bx+di] è l'elemento di-esimo
- il formalismo appena visto richiede che [bx+di] sia preceduto da un riconoscimento del tipo di vettore su cui si sta lavorando, tipo byte o word, quindi a seconda dei casi si scriverà:
byte ptr[bx+di] o word ptr[bx+di]
- ricordate che i registri hanno compiti specifici e quindi devono essere usati come indicato
bx = registro base di e si = registri indice

Perciò con riferimento all'esempio precedente:

```
; stampa degli elementi del vettore codice
mov bx,offset codice
mov di,0000h
ciclo_stampa:
                cmp di,0004h
                je fine_ciclo_stampa
                mov al,byte ptr[bx+di]
                mov ah,0Eh
                int 10h
                inc di
                jmp ciclo_stampa
fine_ciclo_stampa:
```

Per il caricamento del vettore invece, basta far inserire il carattere come al solito e salvarlo nel vettore invece che in una variabile, naturalmente il tutto sarà gestito di nuovo con un ciclo

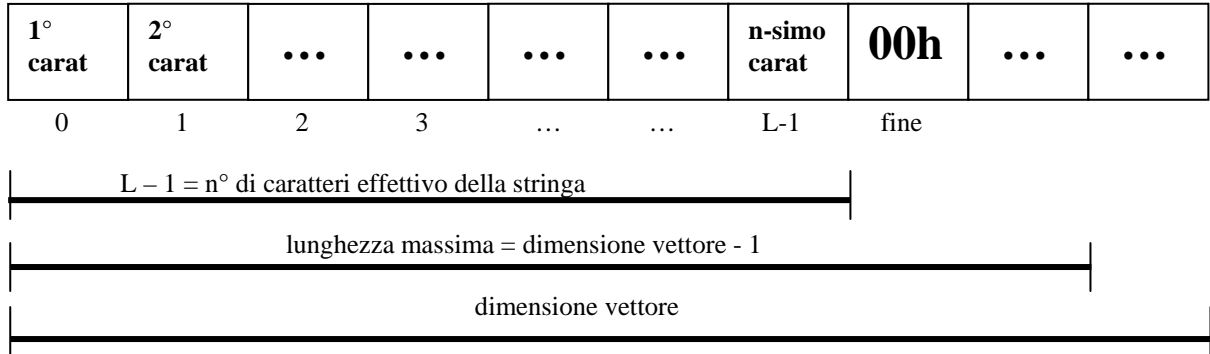
- Le stringhe in Assembler

Le stringhe in assembler come in ogni altro linguaggio non sono altro che vettori di caratteri e quindi possono essere trattate come tali...

STRINGA = VETTORE DI CARATTERI

- Definire una stringa significa quindi definire un vettore di caratteri terminante con 0 (=00h =fine stringa).
In questo caso il vettore-stringa è così strutturato:
1°byte – (ultimo byte-1) = caratteri stringa

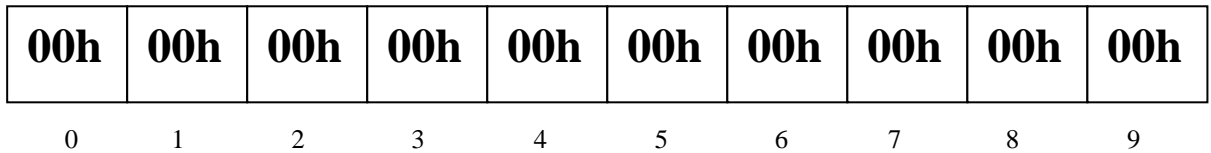
stringa



Oss, naturalmente la stringa può essere più corta della lunghezza dichiarata per il vettore,ma non può superarla

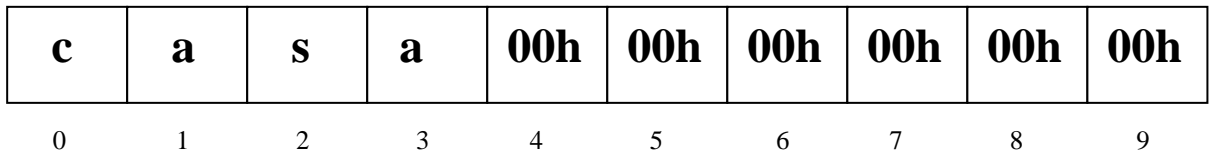
- ~ allora si definisce la stringa come segue:
nome_stringa db N dup(00h)
oss, proprio come un vettore !!

Es. edificio db 10 dup(00h)
Oss, lo 00h sta per 'stringa vuota,non inizializzata'



- ~ per definire la stringa inizializzandola...
nome_stringa db 'contenuto stringa',N dup(00h)

Es. edificio db 'casa',6 dup(00h)



- Trattamento delle stringhe:
il trattamento di stringhe definite in questo modo sarà identico a quello di un normale vettore
- inserimento: sarà costituito da un ciclo che sposterà ogni carattere inserito nel vettore-stringa e che terminerà con la pressione di INVIO, bisognerà quindi ricordarsi di sostituire quest'ultimo con 00h
- stampa: sarà costituita da un ciclo che inizierà dalla 1° cella (byte 0) e terminerà quando il contenuto della L-esima cella sarà 00h (vedi pag. 7)

Nota: per andare a capo prima o dopo la stringa, si dovrà aggiungere dove necessario questo tratto di codice:

```

; a capo
mov al,0Dh      ; stampa Carriage Return
mov ah,0Eh
int 10h

```

```

mov al,0Ah      ; stampa Line Feed
mov ah,0Eh
int 10h

```



GESTIONE DELLA PORTA PARALLELA

Gestire la porta parallela è interessante e divertente perché permette di capire lo scopo per cui si studiano questi argomenti a metà fra elettronica ed informatica, l'Assembler ed in seguito il C.

- Nozioni tecniche

La porta parallela oltre al connettore ben visibile sul retro del PC, è formata da 3 registri (si ricorda che i registri sono piccole aree di memoria che memorizzano i dati temporaneamente) :

- registro di controllo
- registro di stato
- registro dati

I registri sono tutti ad 8 bit.

Il registro di controllo permette di configurare la porta parallela e di impostare la tipologia di trasmissione dati.

Il registro di stato della porta parallela, come quello del microprocessore permette di avere delle informazioni sul dispositivo collegato.

Sul registro dati viaggeranno i dati da trasmettere o ricevere.

Per poter lavorare sui 3 registri si userà un indirizzo di memoria (un po' come quando si vuole spedire una lettera ad un amico, solo che qui l'indirizzo sarà rappresentato da un numero esadecimale) :

<u>registro</u>	<u>indirizzo</u>
dati	0378h
stato	0379h
controllo	0379h

Vedremo ora come mettere in pratica ciò che abbiamo appena appreso, intanto per dare un significato a questa parte di teoria consiglierò, con l'aiuto degli insegnanti di elettronica, di costruire un semplice circuito con 8 diodi led in parallelo ognuno collegato a massa tramite una resistenza da un lato e ad un cavo flat con connettore Cannon 25 poli dall'altro (i led vanno collegati ai pin dal 2 al 9 del connettore Cannon).

Per gestire la porta parallela si devono seguire i seguenti passi :

1. configurazione (impostazione, programmazione della porta, necessaria per ogni dispositivo del PC che si vuole utilizzare)
2. utilizzo della porta

I due passaggi sopra descritti si realizzano nei seguenti modi :

1. configurazione : per semplici circuiti si può omettere, comunque un esempio un po' fittizio può essere il seguente

```
mov al,01h           ; al = attivazione trasmissione
mov dx,037Ah        ; dx = indir. reg. di controllo
out dx,al           ; spedizione dati
```

2. utilizzo porta : si utilizzano due nuove istruzioni la IN e la OUT che funzionano questo modo

<u>OUT</u>	<u>IN</u>
mov al, dato da trasmettere	mov dx, indirizzo registro
mov dx, indirizzo registro	in al, dx
out dx, al	mov variabile, al

Es	<u>OUT</u>	<u>IN</u>
	mov al, 00000001b	mov dx, 0379h
	mov dx, 0378h	in al, dx
	out dx, al	mov variabile, al

A questo punto, se avete realizzato il circuito con i led più sopra descritto, abbiamo in mano tutti gli strumenti che ci servono per realizzare il primo programma per la gestione della porta parallela: cercheremo con esso di creare un gioco di luci simili all'effetto realizzato in un vecchio telefilm americano "Supercar" , faremo quindi accendere e spegnere in sequenza i led dal primo all'ultimo e dall'ultimo al primo, e grazie ad un ciclo lo faremo ripetere più volte.

Per fare accendere e spegnere un led il concetto necessario è questo :

- accensione : si invia al registro dati un 1 binario
- spegnimento : si invia al registro dati uno 0 binario³

Es.

```
mov al,00000001b      ; accende l'ultimo led a destra e spegne tutti gli altri
mov dx,0378h
out dx,al
```

Questo è il codice che realizza il programma appena descritto :

(il programma non funziona correttamente sulle ultime versioni di Windows dal 2000 in poi, che non permettono più l'accesso diretto alla porta parallela, in tal caso sarà necessario utilizzare un emulatore della porta parallela)

```
dati    equ 0378h      ; def. 2 'costanti' o meglio 2 etichette
control equ 037Ah      ; con il valore degli indirizzi dei registri

.model small
.stack
.data
.code
    mov ax,@data
    mov ds,ax

;imposta porta parallela
    mov dx,control
    mov al,01h
    out dx,al

    mov ah,00h          ; azzera contatore ciclo
    mov al,00000001b    ; imposta l'accensione dell'ultimo led
ciclo_led:
    inc ah              ; incrementa contatore ciclo
    cmp ah,10           ; contatore ciclo = 10
    je fine             ; SI : va a fine
andata:
    cmp al,80h          ; al = 80h = 1000 0000 b ?
    je ritorno          ; SI : allora fa il ciclo di ritorno
    shl al,01h          ; NO : allora shifta il contenuto di al
                        ; a sinistra di 1

    ; invia i dati
    mov dx,dati
    out dx,al

    mov cx,0FFFh        ; creazione di un delay (ritardo)
delay11:
    push cx             ; utilizzando due cicli 'for downto' a vuoto
    mov cx,08FFh        ; annidati, questo è necessario per permettere
delay12:
    loop delay12        ; la visualizzazione dell'accensione e spegnim.
                        ; dei led, che altrimenti data l'alta velocità
    pop cx              ; di esecuzione del microprocessore
                        ; resterebbero apparentemente tutti accesi
    loop delay11
    jmp andata

ritorno:
    cmp al,01h          ; al = 01h = 0000 0001 b ?
    je ciclo_led        ; SI : allora salta a ciclo_led
    shr al,01h          ; NO : shifta il contenuto di al a destra di 1

    ; invia i dati
    mov dx,dati
    out dx,al
```

³ Questo rispecchia quello che vi hanno sempre insegnato sui computer :

0 circuito aperto non passa corrente (0 volt, led spento)
1 circuito chiuso passa corrente (5 volt, led acceso)

```
mov cx,0FFFh          ; creazione di un delay (ritardo)
delay21:
  push cx
  mov cx,05FFh
  delay22:
    loop delay22
  pop cx
  loop delay21
  jmp ritorno

fine:
  mov ah,4Ch
  mov al,00h
  int 21h

end
```



LE PROCEDURE

In assembler non esiste una distinzione fra procedure e funzioni, si chiamano tutte PROCEDURE, sta a voi costruirle in modo che realizzino una procedura o una funzione

Le nuove istruzioni da usare sono :

- CALL nome_procedura ; per richiamarla nel main
- mentre per dichiararla a fine programma, prima dell' end

```
nome_ procedura PROC  
...  
RET  
nome_procedura ENDP
```

Oss : non ci sono problemi per le etichette basta che fra le procedure ed il main non si ripetano

```
Es. .model small  
    .code  
    ...  
    CALL somma  
    ...  
    mov ah,4Ch  
    mov al,00h  
    int 21h  
  
    somma PROC  
    ...  
    RET  
    somma ENDP  
end
```

- Passaggio dei parametri

Ci sono vari metodi per passare i parametri alle procedure, il più usato è quello che sfrutta lo STACK, che verrà studiato nella prossima sezione, per ora potete accontentarvi di utilizzare i REGISTRI che sono come delle variabili globali e possono essere portati dentro e fuori dalle procedure come se nulla fosse

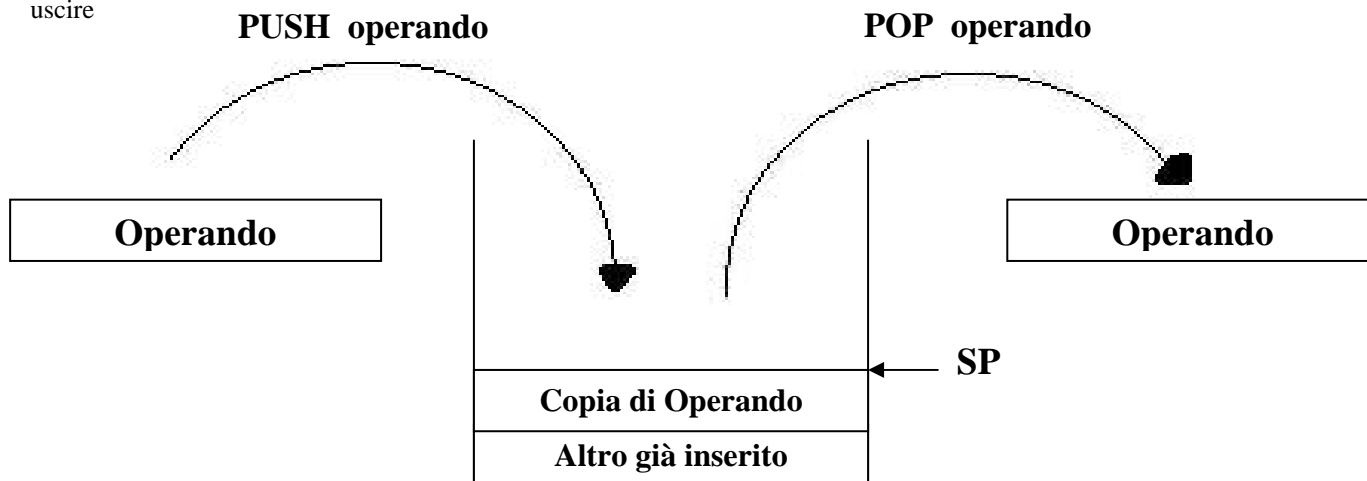
LO STACK

Definizione di stack:

la pila o stack è un insieme lineare di byte nel quale è possibile effettuare operazioni di inserimento(PUSH) o estrazione (POP) solo da un unico estremo, detto cima della pila(TOP), individuato da un apposito puntatore denominato puntatore di testa o stack pointer(SP)

Lo SP punta all'ultimo elemento inserito nello stack !

Lo stack quindi utilizza una struttura di tipo LIFO (Last In First Out) in cui l'ultimo elemento ad essere inserito è il primo ad uscire



Es.

Paragoni con “strutture lifo quotidiane”

- pila di piatti
- scatola cilindrica di pastiglie
- riempimento zaino di montagna

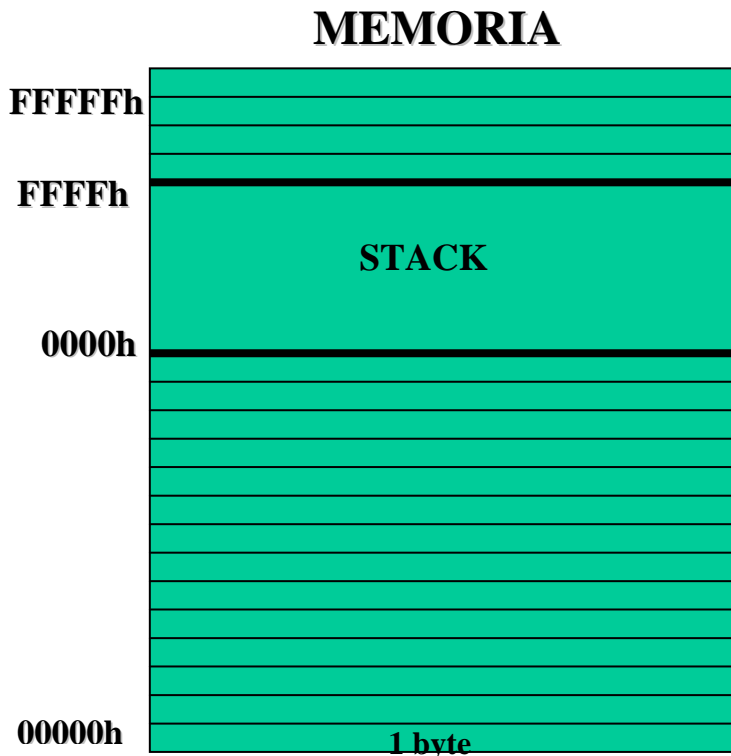
Lo stack quindi è un’area di memoria (segmento) privilegiata,che si usa per memorizzare o ripristinare dati (o gli indirizzi di memoria stessi) quando necessario.

Anche se il puntatore allo stack è SP, esso non si può usare per lavorare sulle celle, questo per evitare di perdere l’indirizzo della testa dello stack,al suo posto si potrà però usare un altro registro BP(Base Pointer)

Operazioni possibili sullo Stack

- PUSH operando a 16 bit : inserimento di un elemento nello stack
- POP operando a 16 bit : estrazione di un elemento dallo stack

Per visualizzare meglio il funzionamento dello stack è meglio immaginare che gli indirizzi di memoria partano dal basso,come nell’immagine seguente



Lo SP si decrementa di 2 byte ogni volta che si effettua una PUSH e si incrementa di 2 byte ogni volta che si effettua una POP !

Es. push ax ; salva una copia di ax sullo stack e decrementa SP di 2
 push var ; dove var è una variabile di tipo word(dw)
 pop cx⁴ ; mette in cx il contenuto della cella puntata da SP
 ; e incrementa SP di 2

push al ; NO!ERRORE al è un byte non una word!
 per risolvere il problema si può usare questo trucco:

```
mov ah,00h ; ripulisce ah,perché il n° è in al
push ax
```

lo stesso si può fare per una variabile v di tipo byte:

```
mov bl,v
mov bh,00h
push bx
```

⁴ Le pop devono essere eseguite in ordine inverso rispetto alle push,ma il µP non fa nessun controllo,quindi attenzione agli errori!

Es. corretto	push var	scorretto,ma possibile	push var
	push dx		push dx

	pop dx		pop cx
	pop var		pop bx

- [Lo stack e le procedure](#)

In precedenza abbiamo visto come dichiarare e richiamare le procedure in assembler, ma cosa succede 'dietro' una CALL ed una RET ?

- mentre il μP sta eseguendo la CALL il suo Instruction Pointer (program counter) sta già puntando all'indirizzo dell'istruzione successiva, la CALL fa una push di tale indirizzo, cioè di IP sullo stack, dopodiché inserisce in IP l'indirizzo di inizio della procedura, così la prossima istruzione da eseguire diventa la 'procedura' stessa e nello stesso tempo non si è perso l'INDIRIZZO DI RITORNO(IR) a cui tornare una volta terminata la procedura (scusate le ripetizioni, utili però nella comprensione del procedim.)

Quindi la CALL = push IP + jmp procedura (automatico)

- arrivato alla RET il μP fa una pop dell'indirizzo di ritorno e lo mette in IP ed in questo modo si ritorna al programma principale.

Quindi la RET = pop IP (automatico)

All'ingresso di una procedura è buona norma salvare sullo stack tutti i registri del μP modificati nel sottoprogramma e che verranno ripristinati alla fine della procedura stessa. (all'inizio però, per non complicarsi troppo i calcoli per il passaggio dei parametri, i registri possono essere salvati sullo stack all'esterno della procedura, prima di fare le push dei parametri, l'unico che si dovrà salvare all'ingresso del sottoprogramma rimarrà comunque BP)

<p>Es. ...push parametri... call nome_proc ...pop parametri...</p> <p> nome_proc proc push ax push bx push cx push dx push di push si push bp ...resto del sottoprogramma... pop bp pop si pop di pop dx pop cx pop bx pop ax ret nome_proc endp</p>	<p>oppure</p>	<p>push ax</p> <p>push bx push cx push dx push di push si ...push dei parametri... call nome_proc ...pop dei parametri... pop si pop di pop dx pop cx pop bx pop ax</p> <p>nome_proc proc push bp ...resto del sottoprogramma... pop bp ret nome_proc endp</p>
---	---------------	--

- [Il passaggio dei parametri attraverso lo stack](#)

Immaginiamo di voler costruire le due seguenti procedure assembler: (il formalismo usato in questo caso è quello del linguaggio C)

```
s = somma(a,b)
somma(a,b,&s)
```

entrambe devono fare la stessa cosa e cioè, $s = a + b$, ma la fanno in modo diverso:

- nel primo caso si è di fronte ad una funzione $s = \text{somma}(a,b)$ dove a e b sono passati per valore (quindi verrà fatta una copia del loro valore sullo stack) ed s viene restituito all'uscita della procedura
- nel secondo caso invece, $\text{somma}(a,b,\&s)$, a e b sono sempre passati per valore, mentre s è passato per referenza (o per riferimento, o per indirizzo) e quindi ciò che viene passato sullo stack è il suo indirizzo e non una sua copia (la & sta infatti per 'indirizzo di...' e quindi $\&s = \text{indirizzo di } s$), quindi s non verrà restituito e modificato alla fine della procedura, ma nella procedura stessa perché si lavorerà con il suo specifico indirizzo del data segment e quindi con la variabile vera e propria.

N.B. : quanto spiegato è quello che avviene per tutti i linguaggi, solo che è normalmente invisibile ai vostri occhi perché fatto automaticamente, mentre in assembler ciò non è più vero e tocca a voi gestire il tutto.

Vediamo ora quindi come realizzare le due procedure sopra descritte ed in particolare il passaggio dei parametri per valore, quello per referenza e la restituzione di un valore (si ipotizzino le variabili di tipo word, dw)

1. s = somma(a,b)

```

...push dei registri del µP...
push a
push b
push s
call somma          ; s = somma(a,b)
pop s               ; restituzione risultato in s
add sp,4            ; a e b passati per valore, allora no restituzione
                    ; è un altro modo per ripulire lo stack, visto che a e b
                    ; non sono stati modificati dalla procedura è inutile
                    ; ripristinarli, spostiamo direttamente SP dello spazio
                    ; occupato dalle copie delle due variabili cioè 4 byte=2*2

...pop dei registri del µP...

```

```

somma proc
    push bp
    mov bp,sp          ; così bp punta dove punta sp
    pars equ word ptr[bp+4] ; a pars associo l'indir. copia di s
    parb equ word ptr[bp+6] ; a parb associo l'indir. copia di b
    para equ word ptr[bp+8] ; a para associo l'indir. copia di a

    mov ax,para        ; somma parametri in pars
    add ax,parb
    mov pars,ax

    pop bp
    ret
somma endp

```

La simulazione di questo programma sullo stack è visibile nelle successive fotocopie formate da diapositive

2. somma(a,b,&s)

```

...push registri del µP...
push a                ; passa a e b per valore
push b
mov cx,offset s       ; passa l'indirizzo di s
push cx               ; e quindi s è passato per referenza
call somma            ; somma(a,b,&s);
add sp,6
...pop registri del µP...

```

```

somma proc
    push bp
    mov bp,sp
    par_a equ word ptr[bp+8] ; a para associo l'indir. copia di a
    par_b equ word ptr[bp+6] ; a parb associo l'indir. copia di b
    mov si,word ptr[bp+4]    ; si = indirizzo di s -> [si]=contenuto di s
    mov ax,par_a             ; somma I parametri in ax
    add ax,par_b
    mov word ptr[si],ax      ; mette il risultato in [si]=contenuto di s , cioè in s
    pop bp
    ret
somma endp

```

