1 Stack

La stack è una struttura dati che risiede nella memoria principale ed è molto usata nella programmazione a basso livello. Si tratta di un'area di memoria nella quale l'accesso ai dati avviene in modo LIFO (**L**ast **I**n **F**irst **O**ut). Una struttura LIFO è tale che l'ultimo elemento che vi è stato depositato è anche il primo che viene ripreso e cancellato. Questo rende lo stack ideale per la memorizzazione "provvisoria" di informazioni che è necessario "parcheggiare" in memoria in attesa che servano in seguito.

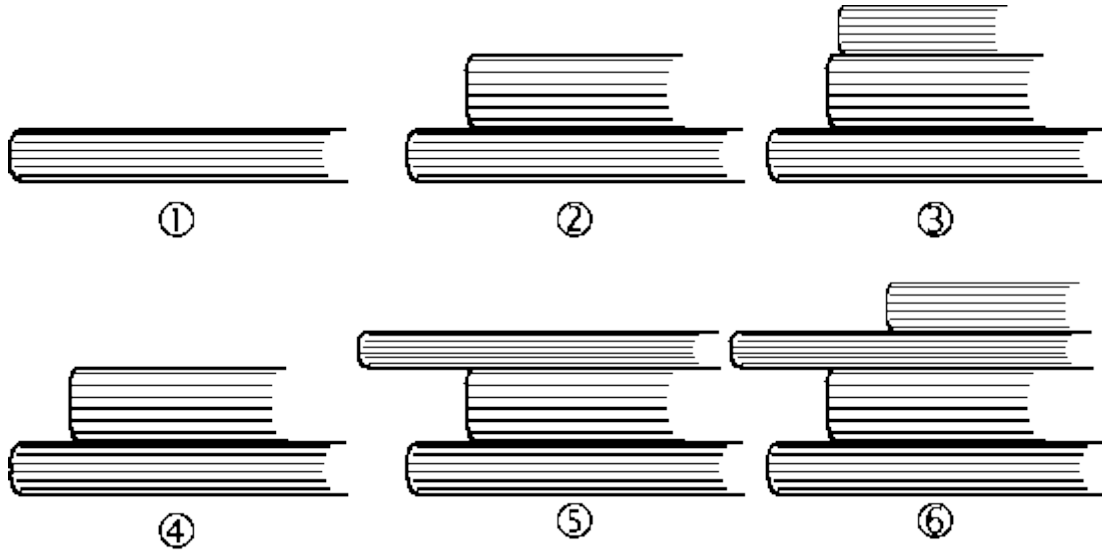

Figura 1: una struttura LIFO

Un uso tipico dello stack durante la programmazione corrente è quando finiscono i registri. Poniamo che, per qualsiasi ragione, ci si trovi con tutti i registri impegnati e che essi non possano essere cancellati, perché mantengono numeri che serviranno presto.
Se in questa condizione si presenta la necessità di utilizzare un registro per un calcolo la cosa più rapida da fare è memorizzare nello stack il contenuto di un registro che servirà più tardi ed usare quello per il calcolo, poi rileggere dallo stack il vecchio valore, quando serve ancora.
L'occasione più importante in cui si usa lo stack, è nelle chiamate alle procedure, anche se lo fa la CPU implicitamente e non il programmatore.
Si lavora sullo stack con due sole istruzioni: **PUSH** e **POP**.
PUSH ("spingi") immette un nuovo valore all'interno dello stack; POP lo toglie (to pop out significa "uscire precipitosamente").
Lo stack si basa su un semplice puntatore il cui valore viene aggiornato automaticamente dalla CPU ogni volta che essa esegue una PUSH od una POP.

Questo puntatore che chiameremo "**stack pointer**" indica sempre l'ultimo elemento dello stack, cioè l'indirizzo in memoria dell'ultimo valore che vi è stato immesso.

Ogni volta che viene eseguita una PUSH la CPU fa automaticamente, nell'ordine, le seguenti operazioni:

1 - Aggiornamento dello stack pointer, che assume il valore giusto per poter effettuare la scrittura nello stack
2 – Scrittura del numero che si vuole immettere nello stack all'indirizzo indicato dallo stack pointer.

L'operazione POP fa il contrario:
1 – Lettura della locazione di memoria indicata dallo stack pointer
2 – Aggiornamento del valore dello stack pointer

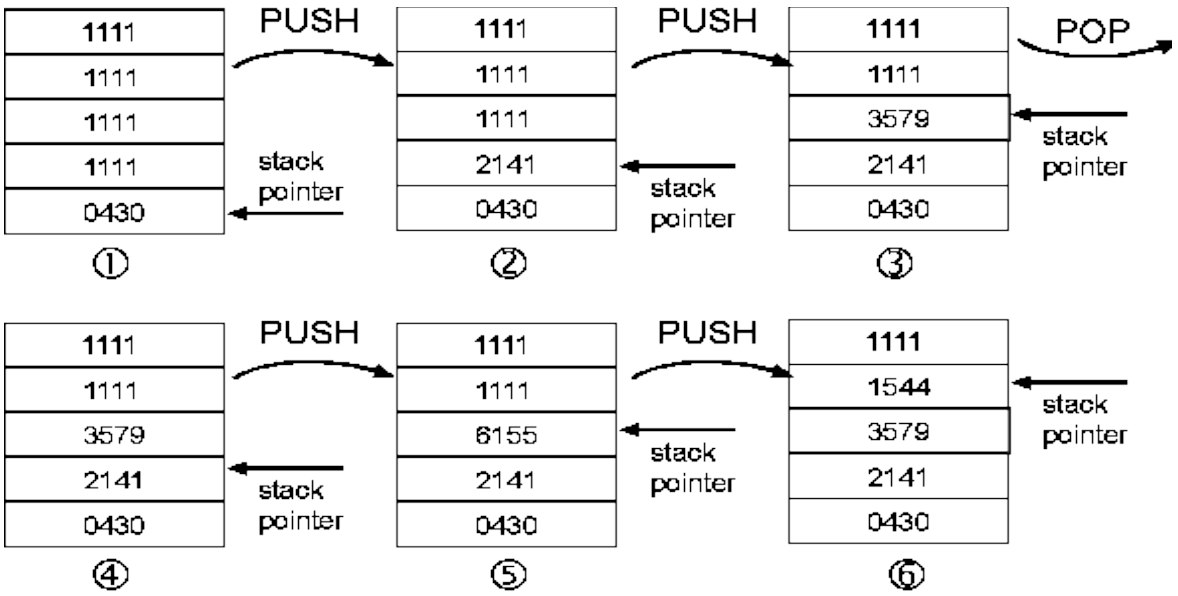

Figura 2: movimento nello stack

Si noti che il valore in memoria dell'elemento "vecchio" dello stack non viene cancellato (da punto 3 a 4 della Figura 2). Questa è un'operazione inutile, che farebbe perdere del tempo e che non serve.
Infatti basta spostare lo stack pointer per far sì che, usando le sole operazioni di PUSH e POP, sia impossibile recuperare i valori vecchi. Una successiva PUSH sovrascrive il valore vecchio (da 4 a 5 in Figura 2), mentre una POP toglie dallo stack un altro valore, che sta ad un indirizzo diverso.

1.0.1 Stack 8086

PUSH e POP
La sintassi delle istruzioni PUSH e POP, in un 8086, è la seguente:

```
PUSH <OperandoDi16bit>
POP <OperandoDi16bit>
```

<OperandoDi16bit> può essere un registro di 16 bit o una word recuperata dalla memoria, nell'8086 non può essere un numero in immediato. La PUSH di un numero in immediato è possibile solo a partire dall'80286. Dal 386 in avanti è possibile fare PUSH e POP a 32 bit.
Dunque nell'8086 tutte le operazioni sullo stack sono solo a 16 bit, non è permesso fare PUSH o POP di valori di 8 bit.
Vediamo alcuni esempi:

```
PUSH AX  ; giusto
~~PUSH CL~~  ; NO! (CL ha 8 bit)
PUSH [UnaWord]   ; giusto a patto che fosse: UnaWord DW ?
PUSH ES:[UnaWord]; si può fare l'override
~~PUSH [UnByte]~~    ; sbagliato se era: UnByte DB?
PUSH WORD PTR [UnByte] ; giusto anche se era: UnByte DB? ma
; ATTENZIONE, può avere effetti collaterali: nello stack entra anche [UnByte + 1]
POP WORD PTR [BX]    ; giusto
POP WORD PTR [EBX]   ; giusto solo se 80386 e > (offset a 32 bit)
POP DWORD PTR [EBX]  ; giusto solo se 80386 e > (offset a 32 bit e PUSH di 32 bit
PUSH WORD PTR 6  ; solo se 80286 e >
POP ECX          ; solo se 80386 e >
~~POP 1~~            ; NON HA SENSO!
```

Il puntatore allo stack, come tutti i puntatori 8086, deve essere composto da due registri.
Come tutti si aspettano il registro di segmento che viene usato nelle PUSH e nelle POP da un 8086 è SS (Stack Segment!), mentre l'offset è SP (Stack Pointer). L'indirizzo segmentato dell'ultima word che è stata messa nello stack è perciò SS:SP.
Nel caso degli X86 a 32 bit, quando funzionano in modo protetto (es. Windows, Linux) usano un offset da 32 bit, contenuto nel registro ESP.
Una cosa interessante è il fatto che lo stack 8086 procede un po' come i gamberi, all'indietro.
Quando lo stack viene creato SP deve essere inizializzato con un valore alto, poi, ogni volta che viene aggiunto un nuovo elemento allo stack, quel valore viene decrementato.

Questo permette di rendersi conto facilmente di quando lo stack si esaurisce ("**stack overflow**").
Infatti se si continua ad eseguire delle PUSH, il valore dello stack pointer cala fino a divenire 0. Se ciò accade significa che lo spazio nello stack è esaurito.
Di solito in questo caso si deve interrompere il programma. Infatti, qualora lo stack sia esaurito e si effettui un'ulteriore PUSH, si esegue la scrittura in un'area di memoria probabilmente al di fuori di quella riservata allo stack, dove potrebbe esserci un qualsiasi programma, o i suoi dati.
Questo può provocare danni gravissimi: quando va bene il crash di sistema, quando va male la corruzione di dati senza che nessuno se ne accorga o addirittura la "presa di controllo" del nostro computer da parte di un programma "indesiderato".

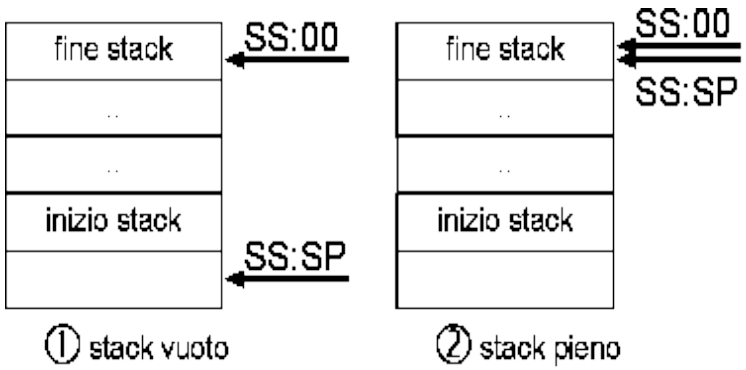

Figura 3: stack overflow

Dunque far procedere lo stack all'indietro ci fa capire facilmente quando è esso esaurito.
Se si vuole controllare lo stack overflow in un 8086 bisogna confrontare SP con zero dopo ogni PUSH.
Nell'8086 la CPU non è in grado di controllare automaticamente se lo stack si esaurisce, mentre le CPU dal 386 in poi lo possono fare. In questo caso esse sono in grado di comunicare al Sistema Operativo che c'è stato questo tipo di errore ("stack overflow") ed il S.O. può "terminare" (concludere forzatamente) il programma che ha sbagliato, per evitare che faccia danni ad altri programmi od all'intero sistema.

Il funzionamento interno, semplice ma efficace, di una PUSH viene spiegato dalla seguente figura:

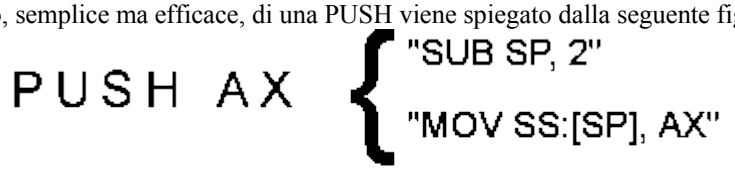

Figura 4: funzionamento di una PUSH 8086 (*)
La prima cosa che fa la PUSH è aggiornare "all'indietro" lo stack, togliendogli 2, poi salva all'indirizzo così ottenuto il valore di ciò che si vuole memorizzare. La memorizzazione avviene usando SS, quindi in un 8086 lo stack ha un segmento "dedicato", separato da quello del codice e dei dati.

(*) Nel disegno compare un errore di sintassi per l'8086. Infatti SP non può essere usato fra parentesi quadre. Peraltro è chiaro cosa si intende ed è quello che importa in questo contesto. Dato che stiamo trattando di un'istruzione "interna", la CPU può fare tutto quello che i suoi progettisti vogliono che faccia!

Il funzionamento della POP è il contrario di quello della PUSH:

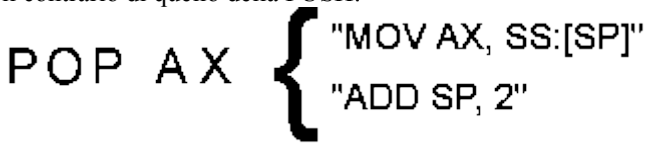

Figura 5: funzionamento di una POP 8086 (*)

In questo caso prima si recupera il valore, poi si sposta lo stack pointer.

Dato che il registro di segmento non viene coinvolto nell'operazione di PUSH e POP uno stack 8086 può essere lungo al massimo come il più grande dei segmenti, cioè 64 kByte.
Gli operandi possono essere sia registri (anche registri di segmento), sia locazioni di memoria che numeri in immediato (dal 286 in avanti).
Nelle CPU dal 386 in poi PUSH e POP possono avere operandi di 32 bit.

Utilizzazione dello stack

I programmi che usano lo stack devono lasciarlo così come lo hanno trovato. Tutto ciò che viene messo nello stack deve essere tolto prima della fine del programma.

Si possono mettere nello stack quanti numeri si vuole, facendo attenzione a non riempirlo, ma essi vanno sempre tolti prima della fine del programma, o della procedura, come avremo modo di vedere un po' oltre.

Data la caratteristica LIFO dello stack l'ordine di immissione dei numeri deve essere inverso rispetto a quello con cui lo stack viene ripristinato.

Perciò in tutti i casi "normali" l'ordine delle PUSH deve essere inverso rispetto all'ordine delle POP.

Come esempio vediamo il brano di codice illustrato nella Figura 6:

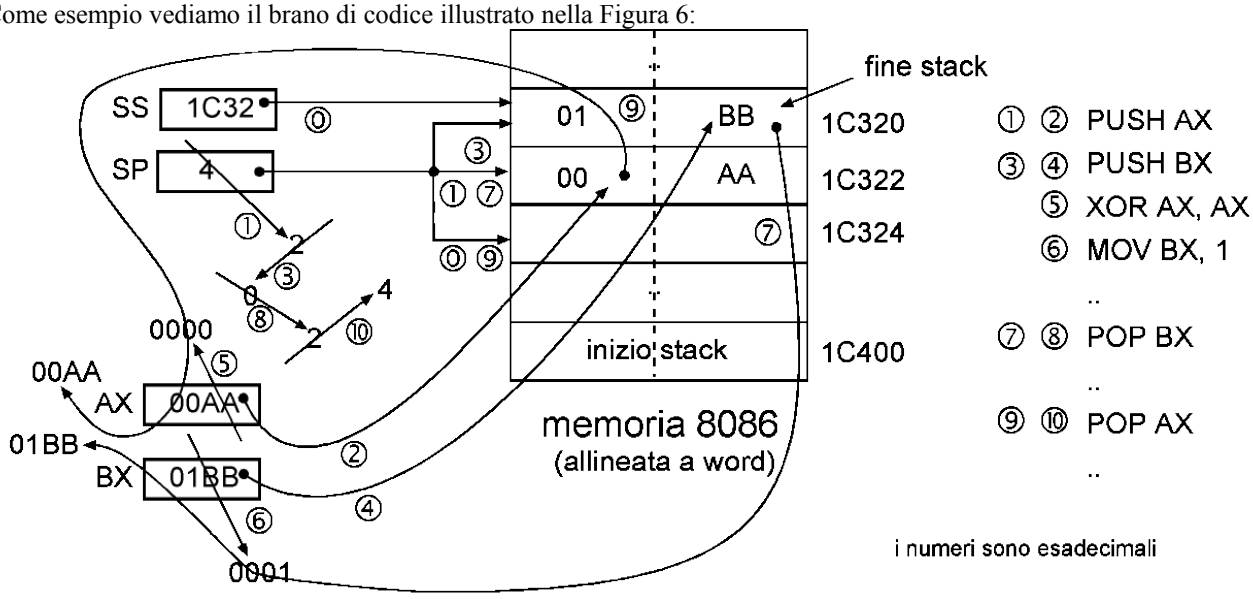

Figura 6: storia di due PUSH

Inizialmente il valore di SP è 4, ciò significa che siamo vicini all'esaurimento dello stack.
La prima istruzione (PUSH AX) mette il contenuto di AX nello stack; per questo:
1) toglie 2 ad SP (SP = 2)
2) scrive AX all'indirizzo SS:SP = 1C322 (00AA nella locazione 1C322)
La seconda istruzione è ancora una PUSH, perciò è analoga (PUSH BX):
3) toglie 2 ad SP (SP = 0)

si noti che ora lo stack è esaurito! Un'ulteriore PUSH causerebbe un errore perché SP diverrebbe –2 = FFFEh. La PUSH scriverebbe all'indirizzo 1C320h + FFFEh = 1C320h + 65534 = 2C31Eh. Se lo stack è più piccolo di 64 kByte (cosa probabile) a questo indirizzo non corrisponde un'area di stack, per cui il programma scrive in locazioni non controllabili, con probabile risultato disastroso.

Nel caso del nostro esempio non ci sono ulteriori POP dopo lo "stack overflow", per cui non succede nulla, ma questo è un caso "fortunato".

4) scrive BX nell'ultima locazione dello stack SS:SP = 1C324 (01BB nella locazione 1C320)
La terza e la quarta istruzione modificano i valori di AX e BX, in modo che AX valga 0 e BX 1.
Successivamente verranno eseguite altre istruzioni, qui non dettagliate, fino a che non servirà ripristinare i valori vecchi di AX e BX. Allora si eseguiranno le POP.
La prima POP ripristina il valore di BX:
7) Lettura in BX della locazione SS:SP (SS:SP = 1C320 => BX <- 01BB)
8) Aggiunta di 2 a SP (SP = 2)
La successiva POP AX ripristina il valore iniziale di AX:
9) Lettura a SS:SP del valore di AX (SS:SP = 1C322 => AX <- 00AA)
10) Aggiunta di 2 a SP (SP = 4)
A questo punto la situazione, per quel che riguarda il contenuto di AX, BX, SS e SP, è identica a quella che si aveva all'inizio.
Il modo di lavorare con lo stack appena illustrato è quello che si usa nel 99% dei casi. Bisogna però far rilevare che l'uso dello stack si presta a "trucchi" e non è per nulla obbligatorio rimettere con la POP il valore dallo stack allo stesso registro da cui si era preso inizialmente. Questo significa che è possibile fare programmi come questo:

```
PUSH AX
```

```
PUSH BX
.. utilizzazione di AX e BX
; ora BX è impegnato, riprendo in CX il suo vecchio valore:
POP CX
; il numero che era in BX ora è in CX
.. elaborazione di quel numero ..
; ora AX è impegnato, CX non mi serve più
POP CX
; il numero che all'inizio era in AX ora è in CX
.. elaborazione di quel numero ..
```

In fin dei conti lo stack è solo un puntatore alla memoria, non può "ricordarsi" del registro dal quale era stato preso il numero che contiene! Ciò che lo stack ricorda è solo il valore di quel numero.

Vediamo un esempio di codice che usa un trucco con lo stack:

```
; scambio di AX con BX:
PUSH AX
PUSH BX
; .. altre elaborazioni ..
POP AX  ; !! attenzione: non è un errore !!
        ; voglio scambiare il contenuto di AX e BX
POP BX
```

Questo programma esegue, in modo inutilmente complicato, lo scambio del contenuto di AX con quello di BX. Come programma per lo scambio di due numeri non è un granchè, perché lo stesso effetto si poteva ottenere con una istruzione sola all'interno della CPU invece che con quattro accessi alla memoria (XCHG AX, BX).
Ma non è questo il punto. Se si vede un programma come il precedente, il 99% delle volte lo scambio è non voluto, cioè non si tratta di un trucco, ma di un semplice bug!
Se invece è davvero un trucco esso va ampiamente documentato, come illustrato nell'esempio.
Poniamo infatti che il programma, dopo un certo tempo che è stato finito, riveli dei difetti e debba essere corretto. Se il programma non è commentato chi lo vede penserà senz'altro che l'ordine di lettura dallo stack sia un errore e potrebbe anche correggerlo, pensando di aver tolto un errore quando invece ne ha aggiunto un altro che non c'era!

La CPU non limita le operazioni su SP, quindi è possibile farle. Quando si modifica SP bisogna sapere molto bene quello che si sta facendo ed usare molta cautela, perché lo stack è una struttura che non si ci si può permettere di rovinare.
Se possibile è meglio lasciar modificare lo stack pointer solo alle PUSH ed alle POP.
Esempio:

```
ADD SP, 10   ; "toglie" 5 elementi dallo stack, senza leggerli
             ; E' possibile fare questa istruzione, ma è sconsigliato
             ; (a tutti tranne ai "guru" dell'Assembly, come i miei lettori)
```

Vantaggi dello stack:
- E' di uso molto semplice
- Non è necessario allocare memoria per ogni informazione che si salva al suo interno, dato che la memoria dello stack è allocata in blocco all'inizio del programma
- quando non c'è più bisogno di un dato che è nello stack, basta toglierlo e la memoria dello stack è a disposizione per altre memorizzazioni temporanee

Svantaggi dello stack:
- Può esaurirsi
- Errori nella programmazione o l'esaurimento dello stack possono dare risultati catastrofici.

1.0.2 Stack 80386

Nelle CPU a 32 bit della famiglia X86, dal 386 in poi, esistono registri ed operazioni a 32 bit, che naturalmente includono anche la PUSH e la POP a 32 bit.

```
PUSH <OperandoDi32bit>

POP <OperandoDi32bit>
```

<OperandoDi32bit> si usa come nel caso a 16 bit. Nel 386 PUO' essere un numero in immediato.
Per compatibilità le vecchie istruzioni PUSH e POP con operando a 16 bit esistono ancora, immutate.

```
PUSH WORD PTR 1  ; giusto
PUSH DWORD PTR 1 ; giusto
~~POP 1~~               ; naturalmente è ancora sbagliato!
```

Se un X86 a 32 bit viene programmato nel modello "flat", come accade in tutti i S.O. significativi, si può ignorare il registro di segmento SS, che punta allo stesso indirizzo degli altri registri di segmento.
Logica l'estensione a 32 bit del funzionamento "a basso livello":
- il puntatore allo stack è ESP (Enhanced Stack Pointer), registro di 32 bit.
- La PUSH a 32 bit toglie 4 ad ESP, poi scrive l'operando da 4 Byte nello stack
- La POP a 32 bit legge nell'operando il valore presente nello stack, poi aggiunge 4 a ESP.

PUSHF, POPF
Queste due istruzioni salvano e ripristinano dallo stack il valore di tutti i flag. Sono disponibili nella famiglia X86 dal 186 in poi. Dal 386 la PUSHF può salvare anche i flag estesi di quella CPU.
Dato che lavorano implicitamente PUSHF e POPF non hanno operandi:

```
PUSHF
; PUSH FLAGS

POPF
; POP FLAGS
```

PUSHA, POPA
Queste due istruzioni salvano e ripristinano dallo stack il valore di tutti i registri generali della CPU. Sono disponibili nella famiglia X86 dal 186 in poi. Dal 386 la PUSHA salva anche i registri a 32 bit.
PUSHA e POPA non hanno operandi:

```
PUSHA
; PUSH All general registers

POPA
; POP All general registers
```

Definizione di uno stack
Un programma non impegnativo per lo stack, che ne faccia un uso moderato, può fare a meno di definire uno stack "locale", nella sua area di memoria. Infatti può usare lo stack che viene messo a disposizione dal Sistema Operativo.
Come esempio si può prendere il S.O. MS DOS, che dispone di alcuni stack interni. Ad uno di essi viene fatto puntare SS:SP prima di lanciare un programma d'utente. Quindi quando un nostro programma inizia ad eseguire SS:SP punta già ad uno stack "valido" fornito dal DOS. Il programma può utilizzare quello stack semplicemente operando le PUSH che deve fare e liberandolo con le relative POP.
Qualora il nostro programma debba fare un uso intenso dello stack e non di si possa fidare della capienza di quello del DOS è necessario allocare la memoria ed inizializzare SS:SP per farlo puntare al nostro stack, invece che a quello del DOS.
Per definire uno stack locale per prima cosa bisogna allocare la memoria da destinargli. Questo si può fare con una normale Define:

```
LabelStack DB 1024 DUP ("Stack")
```

Questa direttiva assegna uno stack di 5 kByte nel quale vengono ripetuti per 1 k volte i codici ASCII della stringa "stack". Questo è utile in fase di debugging, perché permette di individuare più velocemente l'area di memoria riservata allo stack ed anche di capire fino a dove lo stack è stato utilizzato. Infatti le locazioni ove c'è ancora scritto "stack" non sono mai state usate, mentre dove lo stack è stato scritto saranno rimasti gli ultimi valori scritti.
Nella parte di codice bisognerà poi assegnare il valore giusto ad SS e SP, che dovranno puntare all'indirizzo più alto assegnato allo stack (ricordiamo che SP va all'indietro).
Vediamo un brano di programma che definisce uno stack di 1 kByte con un'area di memoria ad esso dedicato e lo inizializza in modo da poterlo utilizzare in seguito per le nostre PUSH e POP.

Esempio a 16 bit:

```
STACKlocale SEGMENT
  FineStack DB 1024 DUP ("S")
    ; per inizializzare lo stack in seguito, bisogna avere un'etichetta
    ; che punti alla locazione successiva alla fine del segmento di stack,
    ; il modo migliore per farlo è questo (ce ne sono anche altri!):
InizioStack: LABEL WORD
    ; con la direttiva LABEL si definisce solo un'etichetta che non occupa memoria,
    ; ma che punta dove vogliamo
STACKlocale ENDS
```

```
.. prima di utilizzare SS devo fare l'ASSUME, che dichiara che SS è "qui":
ASSUME SS:STACKlocale

.. poi, nell'area di codice:.

    ; sistemazione SS:
    MOV AX, SEG STACKlocale
    MOV SS, AX
    ; sistemazione SP:
    MOV SP, OFFSET InizioStack
```

C'è da notare innanzi tutto che, come recitano le etichette assegnate, la fine dello stack sta all'indirizzo più alto del segmento, mentre l'inizio, cui viene fatto puntare SP inizialmente, deve essere un byte dopo la fine dell'area allocata allo stack. Così la prima PUSH per prima cosa toglierà due ad SP, poi scriverà il primo elemento dello stack al posto giusto (vedi anche Figura 3, punto 1).
Per poter fare l'assegnazione di SP serve un'etichetta, da piazzare dopo la definizione dello stack. Si potrebbe mettere una define, ma si sprecherebbe almeno un byte. Per evitarlo si usa la direttiva LABEL WORD, che produce un'etichetta che ha l'indirizzo che le compete per la sua posizione nel sorgente, ma non alloca neppure un byte memoria.
Dato che InizioStack è scritto subito dopo "FineStack DB 1024 DUP ("S")", che alloca tutta la memoria per lo stack, il suo indirizzo sarà proprio quello che si deve scrivere in SP inizialmente.

Esempio a 32 bit (MASM32). Proviamo a creare uno stack nell'area dei dati inizializzati, in modo analogo a quanto fatto nell'esempio predente. Si tenga presente che se la dimensione dello stack è significativa, è meglio crearlo nell'area non inizializzata:

```
.DATA
  FineStack DB 1024 DUP ("S")
  InizioStack LABEL DWORD

..

.DATA?
; quando si finalizza un programma è meglio mettere qui lo stack

.. poi, nell'area di codice:

.CODE
Inizio:

    ; sistemazione ESP:
    MOV ESP, OFFSET InizioStack

.. qui c'è il resto del programma
```

Dimensionamento dello stack
Siccome non si sa a priori quale può esser il massimo di elementi che verranno scritti consecutivamente nello stack, esso va dimensionato con una certa abbondanza, ricordando quali sono gli effetti del suo esaurimento!
Questo può essere un problema quando si programma per i microcontrollori. Nei sistemi embedded di solito la memoria è poca, o pochissima, per cui fare stack grandi può essere impossibile, o troppo costoso.
La dimensione dello stack va scelta con grande attenzione, effettuando prove e simulazioni sui prototipi, con sistemi di sviluppo e profiler. Durante la fase di test usare uno stack molto grande e verificare quanto il programma lo usa in condizioni "estreme" (massimo sovraccarico). Esistono strumenti di sviluppo che tengono traccia dell'uso dello stack, per cui si può vedere quanti valori vi sono stati immessi e la dinamica del suo uso, per prendere una decisione ben ponderata.

Stack e Sistemi Operativi

Ogni Sistema Operativo ha un programma "loader", che legge dall'hard disk l'eseguibile del programma, lo carica in memoria e lo fa eseguire. In tutti i Sistemi Operativi il loader, prima di lanciare il programma, sistema lo stack pointer in modo che punti ad uno stack valido, che il programma potrà usare senza doverne definire uno al suo interno.
Quindi se in un programma si fa una PUSH senza aver definito uno stack locale, esso funzionerà lo stesso, dato che userà lo stack che gli è stato "passato" dal Sistema Operativo. E' chiaro però che, se che l'uso dello stack è troppo intensivo, esso potrebbe essere esaurito, dato che non si sa esattamente quale sia la dimensione dello stack che viene passato dal S.O.