STACK, istruzioni PUSH e POP; PROCEDURE, istruzioni CALL e RET

Lo stack

Lo **stack** (tradotto talvolta in italiano con *pila* o *catasta*) è un'area di memoria privilegiata, dove il microprocessore può salvare momentaneamente e poi recuperare dati di varia natura. Precisiamo subito che quest'area è speciale solo per l'uso che ne facciamo: dal punto di vista hardware non ha niente di strano, anzi, si tratta semplicemente di un'area della normale memoria RAM.

La differenza fondamentale nell'uso di un'area di memoria normale e di un'area stack risiede nel modo con cui vengono scritti e letti i dati. In una memoria gestita normalmente, il programmatore fa eseguire al microprocessore delle istruzioni ordinarie di trasferimento dati, scrivendo e leggendo dati in locazioni il cui indirizzo viene deciso esplicitamente dal programmatore. Per esempio:

```
MOV [200h],EAX    ; scrive il contenuto di A nell'indirizzo 200h
MOV EAX, [200h]   ; legge da RAM all'indirizzo 200h e copia il contenuto nell'accumulatore A
```

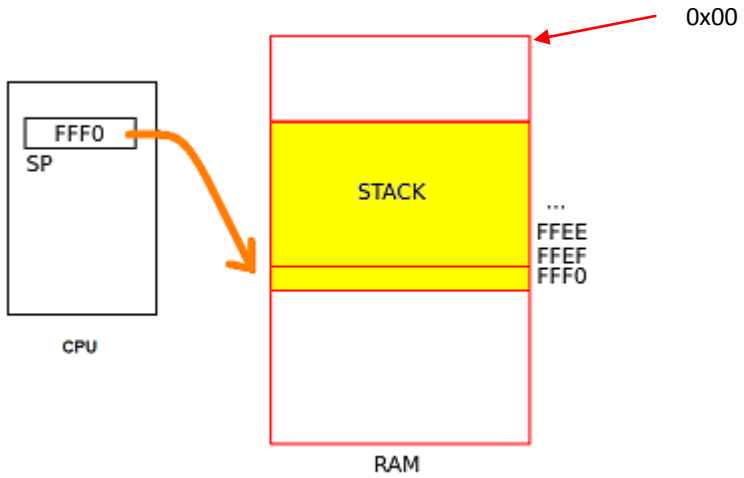
In alcuni casi però risulta comodo non essere obbligati a conoscere esplicitamente l'indirizzo del dato da trasferire. Questo è possibile se il µP dispone di un meccanismo automatico che faccia le nostre veci nel decidere gli opportuni indirizzi di memoria, mentre noi ci occupiamo soltanto dei dati da trasferire.

Il registro speciale, detto **stack pointer** (SP, *puntatore allo stack*), ha lo scopo appunto di gestire lo stack. Si presti attenzione a non confondere i termini: lo stack pointer (e la sua logica di gestione) si trovano all'interno del µP, mentre lo stack (area tipo **L**ast In **F**irst **O**ut) vero e proprio risiede nella RAM collegata al microprocessore.

La prima operazione che è necessario compiere per poter utilizzare lo stack è inizializzare lo stack pointer con l'indirizzo base dello stack. In pratica il programmatore decide in quale area della RAM desidera memorizzare lo stack e le sue dimensioni e inizializza lo SP con un'istruzione del tipo:

```
MOV SP,0FFF0h    ; inizializza SP con l'indirizzo 0FFF0h (esempio)
```

Questa istruzione viene generalmente eseguita all'inizio di ogni programma in assembly. La figura seguente mostra il significato di stack e di stack pointer:



Si osservi che lo SP all'inizio punta al fondo dell'area di stack, cioè all'indirizzo più alto nello stack.

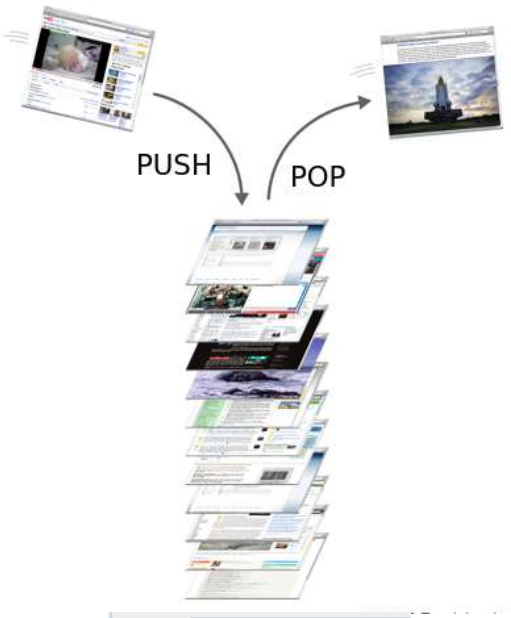
Le operazioni **push** e **pop** eseguite sullo stack fanno crescere l'area di stack "all'indietro" (backwards), cioè, all'aumentare dei dati salvati sullo stack, lo stack pointer viene progressivamente decrementato.

Per queste ragioni di solito l'area riservata allo stack occupa gli indirizzi più alti in RAM, a partire dall'ultimo indirizzo (in modo da riservare il maggior spazio possibile alla crescita dello stack).

Push e pop

Come suggerito dal nome stesso, lo stack viene gestito dal µP come una *pila* o *catasta* di oggetti (esempio una pila di fogli posati su una scrivania). Sull'area di memoria stack è possibile effettuare solo due operazioni e precisamente:

- **push**: aggiunge un dato in cima allo stack;
- **pop**: preleva l'ultimo dato presente in cima allo stack.



Le operazione di push e pop gestiscono lo stack secondo un criterio detto **First In First Out** (FIFO), ovvero: il primo dato a uscire è l'ultimo entrato (quello posizionato in cima allo stack).

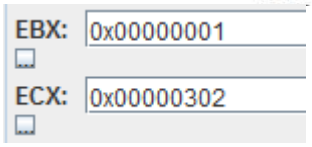
Le operazioni push e pop corrispondono a due istruzioni omonime:

```
PUSH <reg>   ; mette il contenuto del registro in cima allo stack
POP  <reg>   ; preleva il contenuto del registro dalla cima dello stack
```

Se le istruzioni precedenti operano su dati a **16 bit**, questi vengono trasferiti da/verso lo stack suddivisi in due byte, alto e basso, occupando pertanto due locazioni di memoria consecutive.

Consideriamo – in **ambiente Jasmin** – l'esempio che segue:

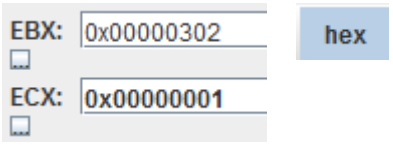
```
MOV SP, 0FFFh  ; inizializza SP con l'indirizzo 0FFFh cioè 4095d
               ;  per visibilità in ambiente Jasmin
MOV BX, 0001h  ; memorizza il valore 0001h nel registro B usando 16 bit
MOV CX, 0302h  ; memorizza il valore 0005h nel registro C usando 16 bit
PUSH BX        ; salva il contenuto di B sullo stack (in 0xFFE copia 0, in 0xFFD copia 1)
PUSH CX        ; salva il contenuto di C sullo stack (in 0xFFB copia 2, in 0xFFC copia 3)
POP BX         ; preleva la cima dello stack e la salva in B scambiando il valore
POP CX         ; preleva la cima dello stack e la salva in C scambiando il valore
```



Osserviamo che:

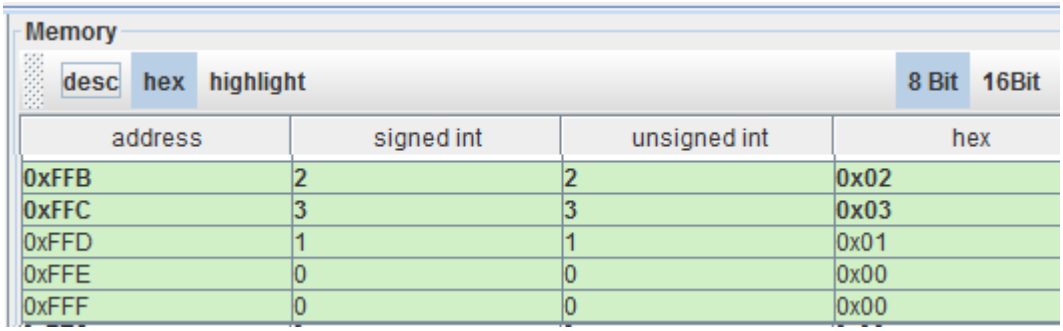
a) il programmatore non deve indirizzare esplicitamente le locazioni di memoria usate per salvare dati che vogliono usare;
b) un certo numero di PUSH deve essere sempre seguito da un numero identico di POP (altrimenti lo stack si riempie senza svuotarsi o viceversa);
c) la sequenza di recupero deve essere rovesciata rispetto a quella di salvataggio nel caso di voler ripristinare i valori originari: qui invece si vuole usare lo stack per **scambiare** i valori dei registri.

Al termine dell'esecuzione di questa sequenza di istruzioni i due registri B e C si sono **scambiati il contenuto**: B contiene 0302h e C contiene 0001h.



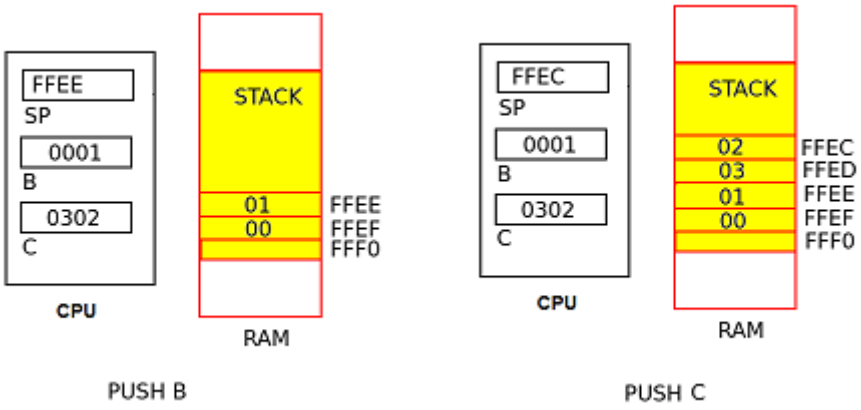
La figura precedente mostra il contenuto dei registri dopo l'esecuzione delle istruzioni di PUSH la cui sequenza di recupero, non rovesciata rispetto a quella di salvataggio ha l'effetto di scambiarne il contenuto.

La figura a lato mostra la situazione dello stack dopo l'esecuzione delle istruzioni PUSH analizzando il salvataggio dei singoli byte (8 bit)



address	signed int	unsigned int	hex
0xFFB	2	2	0x02
0xFFC	3	3	0x03
0xFFD	1	1	0x01
0xFFE	0	0	0x00
0xFFF	0	0	0x00

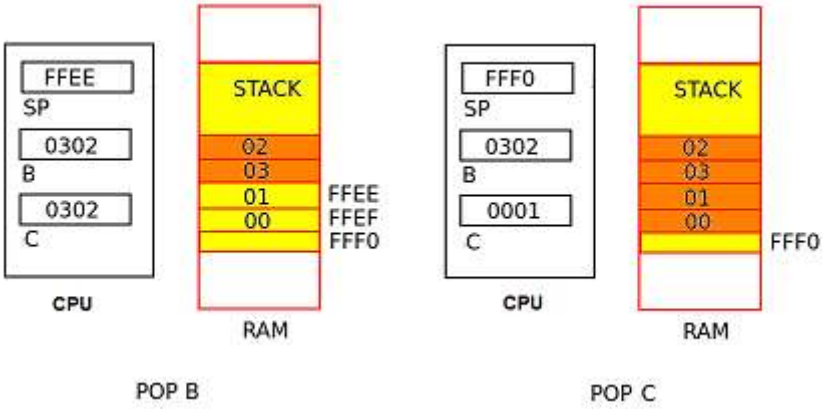
La figura seguente mostra la situazione dello stack e dei registri interni nel caso di inizializzare l'area di stack all'indirizzo FFF0h



Osserviamo in particolare che:

- lo stack pointer viene **decrementato** eseguendo un'istruzione PUSH di due unità (gli indirizzi nella memoria stack decrescono a ogni istruzione PUSH);
- l'istruzione PUSH memorizza il valore immediatamente al di sotto della locazione puntata dallo SP (a causa di tale comportamento la prima locazione, FFF0 nel nostro esempio, rimane inutilizzata);
- il valore contenuto nella coppia di registri B e C viene salvato sullo stack con i due byte, alto e basso, invertiti.

Vediamo ora l'effetto dell'esecuzione delle due istruzioni POP B e POP C:



Si osservi che alla fine lo SP contiene lo stesso valore che conteneva inizialmente (FFF0). Tuttavia è bene sapere che le CPU più vecchie non effettuavano alcun controllo di correttezza sulle operazioni compiute sullo stack e che pertanto era compito del programmatore accertarsi del fatto che il numero di POP fosse uguale al numero di PUSH precedentemente eseguite.

Osserviamo inoltre che i valori non vengono fisicamente cancellati dalla memoria RAM dopo essere stati prelevati con un'istruzione POP. Tuttavia essi, trovandosi al di sotto della locazione puntata da SP, risultano in pratica inaccessibili con ulteriori operazioni di POP sullo stack.

Salvataggio e ripristino di tutti i registri interni con lo stack

L'area di stack è utilizzata prevalentemente per salvare temporaneamente il contenuto di registri interni e per ripristinarli in seguito.

La coppia PUSH/POP è ideale quando è necessario usare un registro senza rischiare di perderne il contenuto; può succedere infatti di avere tutti i registri caricati con valori importanti e, non avendone altri (dato il loro limitato numero...) di non sapere come fare. In questi casi è possibile salvare i registri sullo stack (con l'istruzione PUSH), riusarli liberamente e quindi ripristinare i valori iniziali (con la POP).

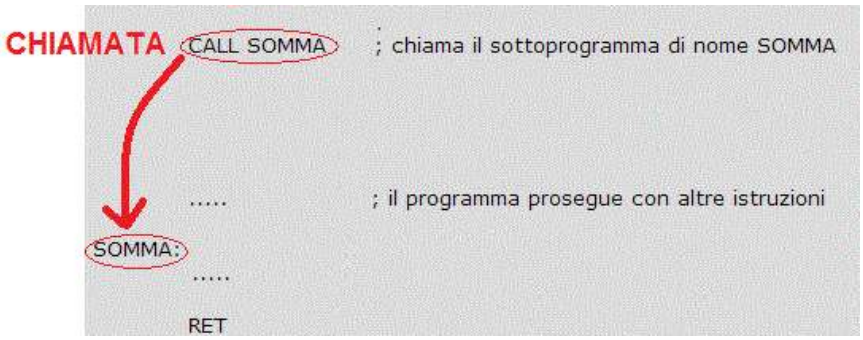
Esistono istruzioni per salvare/ ripristinare tutti i registri (a partire da DI) :

```
PUSHA      ; salvataggiodi tutti i registri nello stack
POPA       ; ripristino di tutti registri
```

Si usano quando il programmatore ha bisogno di usare tutti i registri interni alla CPU, in un qualche compito "locale", ma senza perderne il contenuto volendo completamente ripristinare la situazione precedente.

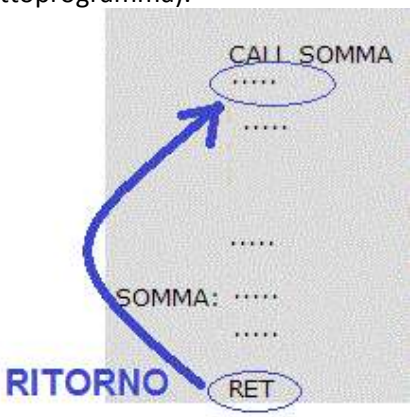
Sottoprogrammi: CALL e RET

In informatica si dice **sottoprogramma** (o anche funzione, subroutine, routine, ***procedura***) una sequenza di istruzioni utilizzabile come se si trattasse di un programma autonomo, ma la cui esecuzione è subordinata a quella di un programma principale. I sottoprogrammi sono molto utili per svolgere compiti ripetitivi senza essere obbligati ogni volta a riscrivere lo stesso codice.



L'istruzione **CALL** avvia l'esecuzione del sottoprogramma (**chiamata**) ed è simile a un'istruzione di salto (jump) in quanto provoca un salto nella sequenza di esecuzione del programma all'etichetta SOMMA (naturalmente, volendo complicarsi la vita, invece dell'etichetta sarebbe possibile scrivere direttamente l'indirizzo della locazione di memoria di inizio del sottoprogramma):

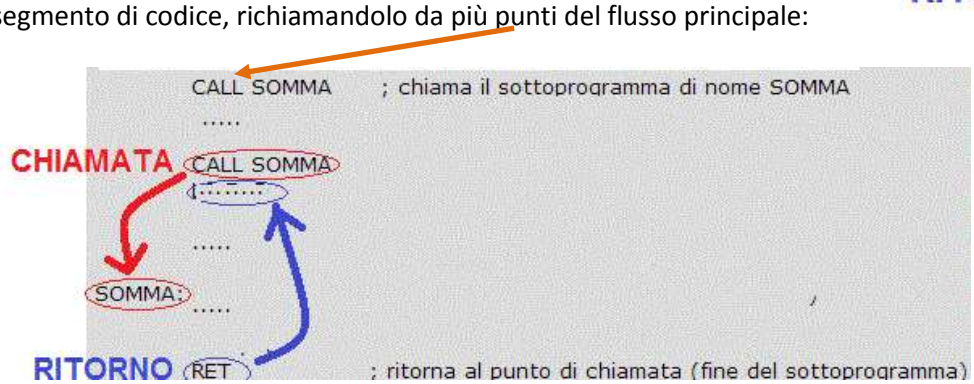
Il sottoprogramma viene quindi eseguito fino all'istruzione **RET** (*ritorno*). Questa istruzione obbliga il µP ad effettuare un altro "salto speciale" fino alla locazione di memoria immediatamente successiva a quella della CALL che ha "chiamato" il sottoprogramma.



Osserviamo la particolarità del meccanismo di chiamata e ritorno.

Volendo, la chiamata al sottoprogramma potrebbe essere effettuata anche con un semplice salto, dato che ne conosciamo l'indirizzo (identificato dall'etichetta SOMMA);

ma la stessa cosa non si può fare per il ritorno se si sfrutta lo stesso segmento di codice, richiamandolo da più punti del flusso principale:



Infatti, l'indirizzo a cui ritornare non sarebbe lo stesso nei vari casi: il sottoprogramma dovrebbe tornare al chiamante, nei vari casi, con istruzioni di salto diverse, perché diversi sarebbero gli indirizzi di ritorno.

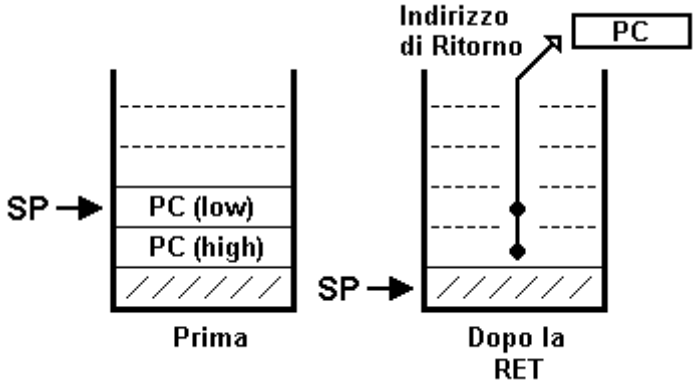
Evidentemente il µP deve memorizzare da qualche parte l'indirizzo di ritorno al momento dell'esecuzione dell'istruzione CALL, in modo da poter usare tale indirizzo al momento dell'esecuzione dell'istruzione RET.

L'area in cui vengono salvati gli indirizzi di ritorno dei sottoprogrammi è lo stack.

Lo stack nella chiamata a sottoprogrammi

L'istruzione **CALL** deve salvare da qualche parte l'indirizzo di ritorno dal sottoprogramma in modo che tale indirizzo possa essere usato dall'istruzione **RET**. L'area in cui avviene il salvataggio e il recupero dell'indirizzo è lo **stack**.

In pratica l'istruzione CALL esegue un *push* automatico del contenuto del program counter (il registro/contatore che contiene sempre l'indirizzo della successiva istruzione da eseguire) sullo stack. Viceversa la RET esegue una pop dalla cima dello stack al program counter:



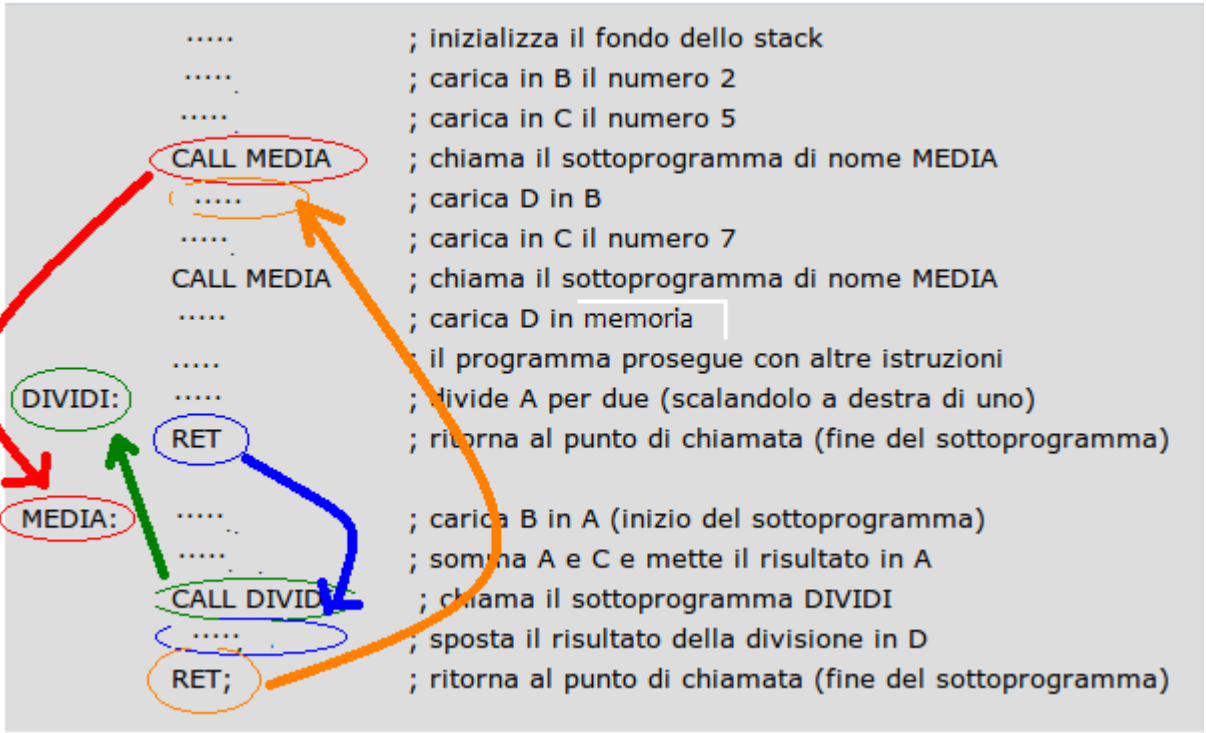
Tale meccanismo di salvataggio e di recupero è assolutamente automatico e trasparente per il programmatore, il quale deve solo occuparsi di allocare un'area di memoria per lo stack (inizializzando opportunamente il valore dello *stack pointer*).

L'utilità di questo meccanismo di salvataggio e ripristino risulta evidente se si considera il caso, tutt'altro che infrequente, di un sottoprogramma che al proprio interno ne chiama un altro.

Consideriamo l'esempio seguente:

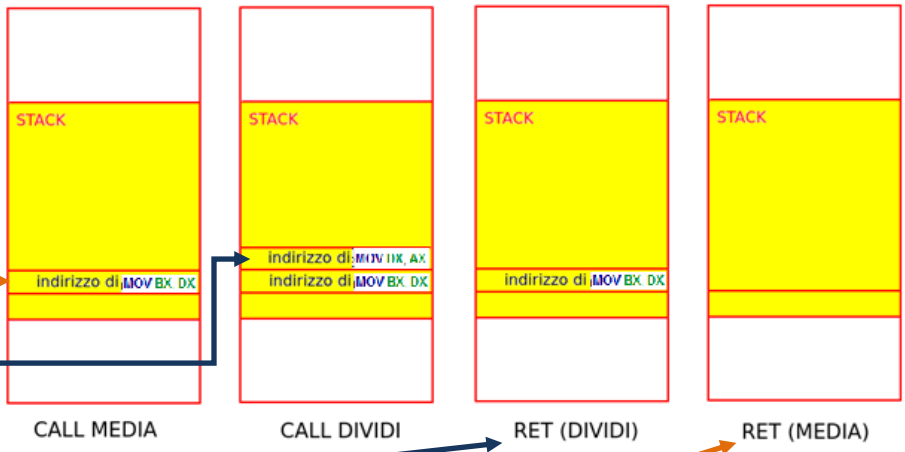
```
        MOV SP,FFh          ; inizializza il fondo dello stack a FFh
        MOV BX, 2           ; carica in B il numero 2
        MOV CX, 5           ; carica in C il numero 5
        CALL MEDIA          ; chiama il sottoprogramma di nome MEDIA
        MOV BX, DX          ; carica D in B
        MOV CX, 7           ; carica in C il numero 7
        CALL MEDIA          ; chiama il sottoprogramma di nome MEDIA
        MOV [0x00], DX      ; carica D in memoria
                            ; il programma prosegue con altre istruzioni
DIVIDI: SHR AX,1            ; divide A per due (scalandolo a destra di uno)
        RET                 ; ritorna al punto di chiamata (fine del sottoprogramma DIVIDI)

MEDIA:  MOV AX,BX           ; carica B in A (inizio del sottoprogramma)
        ADD AX,CX           ; somma A e C e mette il risultato in A
        CALL DIVIDI          ; chiama il sottoprogramma DIVIDI
        MOV DX, AX          ; sposta il risultato della divisione in D
        RET                 ; ritorna al punto di chiamata (fine del sottoprogramma MEDIA)
```

In questo esempio il programma principale chiama il sottoprogramma MEDIA, il quale a sua volta chiama DIVIDI. Affinché il meccanismo delle chiamate e dei ritorni funzioni correttamente occorre che la RET di DIVIDI torni a MEDIA, il quale a sua volta deve tornare al programma principale:



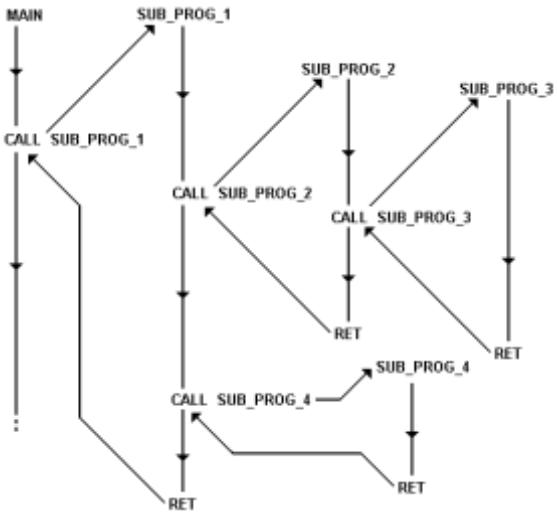
E qui entra in gioco lo stack. Infatti l'indirizzo dell'istruzione **MOV BX,DX** nel programma principale viene salvato sullo stack al momento della chiamata (**CALL**) alla procedura MEDIA.

Quando, all'interno di MEDIA, viene chiamata DIVIDI, viene salvato sullo stack l'indirizzo dell'istruzione successiva **MOV DX,AX** (istruzione del sottoprogramma MEDIA).



Quindi la **RET** di DIVIDI preleva questo indirizzo dallo stack, mentre la seconda **RET**, quella di MEDIA, preleva dallo stack il primo indirizzo salvato.

Si osservi che, ogni volta che si esegue una CALL, viene salvato un nuovo indirizzo di ritorno sulla cima dello stack, con conseguente crescita dello stack stesso.



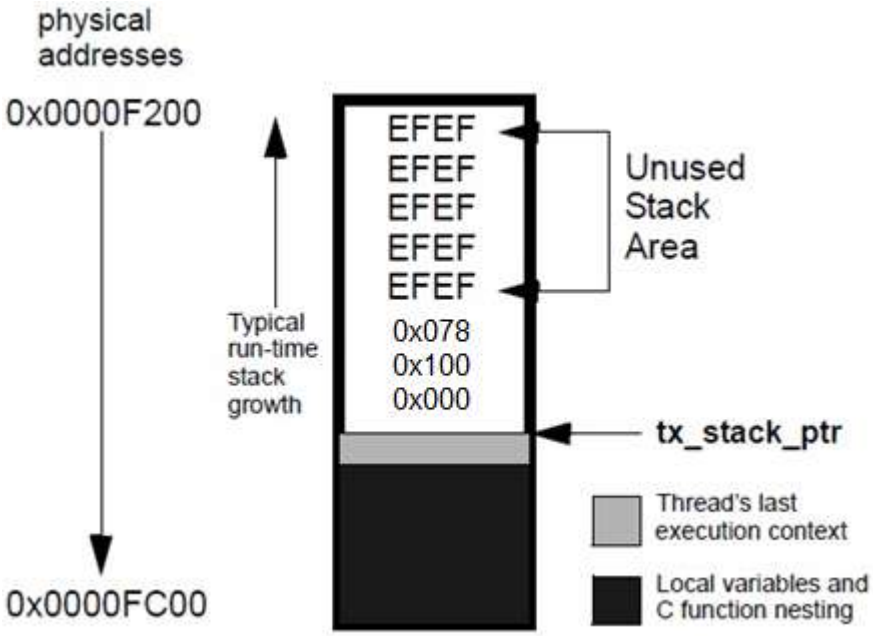
Se non è previsto nessun automatismo di controllo del valore assunto dallo stack pointer, un numero di chiamate annidate (cioè di sottoprogrammi che chiamano altri sottoprogrammi) troppo elevato provoca il rischio che lo stack straripi (**stack overflow**) andando a sovrascrivere altre aree di memoria.

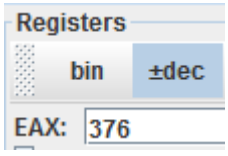
Per evitare problemi, si sovradimensionava la zona di memoria destinata allo stack, cercando di stimare le necessità in termini di annidamenti ed uso normale delle istruzioni di PUSH e POP.

Microprocessori più recenti e più evoluti possiedono dei meccanismi di protezione contro questo tipo di overflow della memoria.

Ambiente JASMIN

```
; uso di procedura
; si vogliono sommare due numeri
;  num1<-- 100H    cioè 256
;  num2 <-- 78H     cioè 120
;  ed il risultato si inizializza a zero  ris <-- 0H
num1: dd 100h
num2: dd 78h
ris: dd 0h

; passo nello stack il primo parametro che
conterrà il risultato (registro qualunque)
MOV AX,[ris]
PUSH AX ; assegna zero al primo
parametro nello stack  (ris <-- 0)

MOV AX,[num1] ; passo il secondo
parametro (Acc<--num1)
PUSH AX

MOV AX,[num2] ; passo il terzo parametro
(Acc<--num2)
PUSH AX

CALL addizione ;  procedura ...  segmento
individuato da label
POP AX ; recupero parametri
POP AX
POP AX

MOV AX,[ris]
MOV [0x10], AX ; salvo il risultato ris in cella  di memoria per test
JP fine

addizione:                          ; inizio PROCEDURA
PUSHA   ; salvo tutti i registri
MOV AX, [num1]  ; num1 in Accumulatore
ADD AX, [num2]  ; sommo con num2
MOV [ris], AX
POPA  ; ripristino
RET                                 ; fine PROCEDURA
fine:
MOV [0x14], AX; salvo il risultato ris in cella  di memoria per test
```





Effetto in MEMORIA

Memory

desc | hex | highlight | 8 Bit | 16Bit | 32Bit

address	signed int	unsigned int	hex
0x0	256	256	0x00000100
0x4	120	120	0x00000078
0x8	376	376	0x00000178
0xC	0	0	0x00000000
0x10	376	376	0x00000178
0x14	376	376	0x00000178