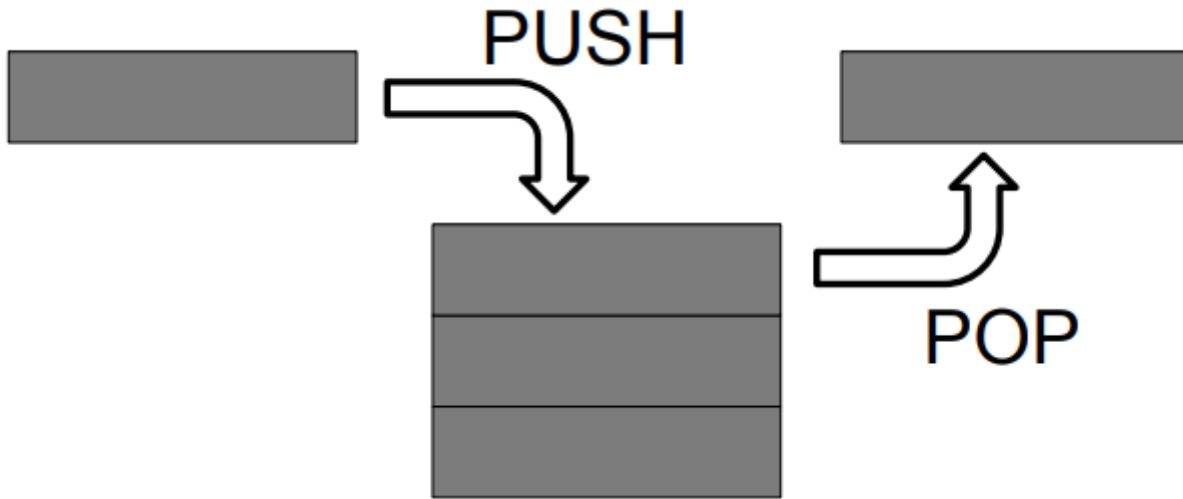


# Stack

Uno stack è una coda **LIFO** - Last In-First out



## Tipi di stack

### Aggiornamento dello stack dopo un push

#### Descending stack

Dopo un **push**, l'indirizzo alla cima dello stack decrementa. Dunque lo stack cresce dall'alto verso il basso (l'indirizzo più alto è l'inizio dello stack)

#### Ascending stack

Dopo un **push**, l'indirizzo alla cima dello stack aumenta. Dunque lo stack cresce dal basso verso l'alto (l'indirizzo più alto è l'inizio dello stack)

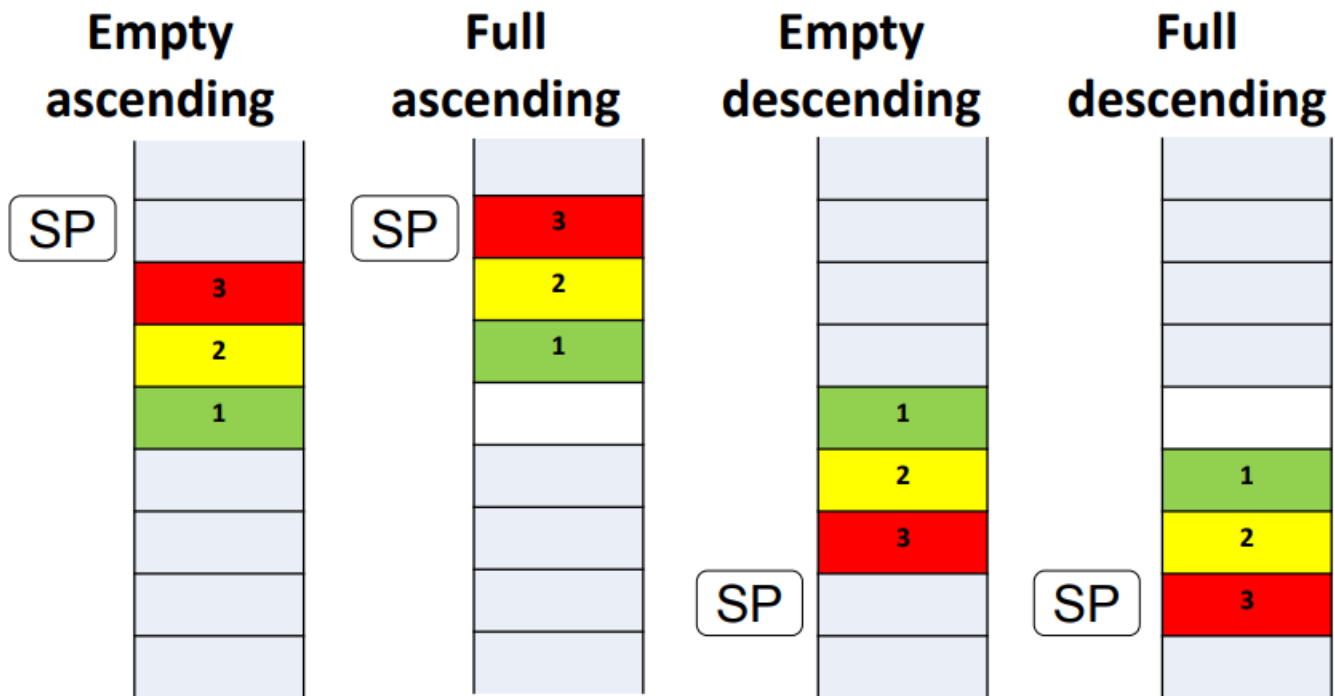
### Contenuto della cima dello stack

#### Empty stack

Lo stack pointer punta alla prima locazione libera dove il nuovo dato viene messo

#### Full stack

Lo stack pointer punta all'ultimo dato *pushato*



## Lettura/Scrittura dello stack

Per leggere/scrivere dallo stack, delle istruzioni utili sono **LDM** e **STM**

### LDM e STM

Permettono il loading/storing multiplo di words.

**Attenzione!!:** *regList* viene ordinata automaticamente e l'ordine in cui vengono salvati è il seguente:

- il registro con il numero più basso viene salvato nell'indirizzo di memoria più basso. Dunque, in uno stack **full descending**, un **STMDB SP!, R1,R2,R3**:
- R1 sarà nell'indirizzo più basso mentre R3 in quello più alto
- R1 sarà in cima allo stack, R3 alla coda dello stack

**LDM{xx}/STM{xx} <Rn>{!}, <regList>**

dove:

- XX** rappresenta il tipo di stack.
  - IA:** increment-after.
  - DB:** decrement-before
- Rn** è il registro che contiene l'indirizzo dello SP(di solito è **R13**).
- regList** è una lista di registri
- !:** post-indexing ovvero aggiorna Rn alla fine dell'istruzione.

In particolare:

- **STM**: salva in memoria, nell'indirizzo contenuto da **Rn**, la lista di registri **regList**, in ordine sequenziale.
- **LDM**: salva in **regList**, i valori contenuti nella memoria all'indirizzo contenuto in **Rn**.

Stack type	PUSH	POP
Full descending	STMDB STMFD	LDM LDMIA LDMFD
Empty ascending	STM STMIA STMEA	LDMDB LDMEA

STMDB e LDMIA implementano uno stack full descending.

Si possono usare anche **PUSH** e **POP** come sorta di alias per **STDMB** e **LDMIA**.

`PUSH <regList>` is the same as  
`STMDB SP!, <regList>`

`POP <regList>` is the same as  
`LDMIA SP!, <regList>`

## Subroutines

Analoghe alle funzioni presenti nei linguaggi ad alto livello, come il **C**.

## Funzionamento

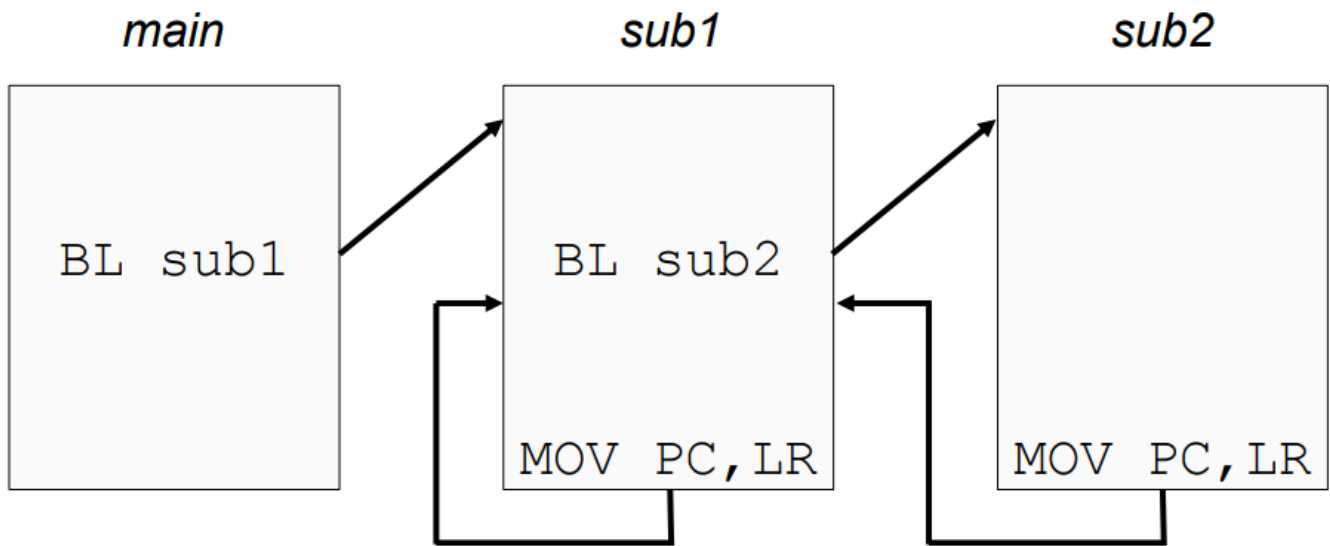
Si può chiamare una subroutine attraverso le funzioni di salto **BL <label>** e **BLX <Rn>**.

Queste istruzioni salvano l'indirizzo della prossima istruzione in **LR(R14)** e poi scrivono il valore di **label** o **Rn** in **PC**.

Le subroutines possono venir definite anche attraverso le *direttive* **PROC** e **FUNC**.

## Nested subroutines

E' possibile fare chiamate annidate a *subroutines*. Tuttavia bisogna prestare particolare attenzione a salvare l'indirizzo in **LR**.



In questo caso, *sub1* non riesce a ritornare al *main* in quanto *sub1*, prima di chiamare *sub2*, salva nel **LR**, precedentemente occupato dall'indirizzo per tornare al *main*, l'indirizzo per tornare a *sub1*.

Per risolvere ciò, ogni subroutine come prima istruzione deve fare il **PUSH** dei registri usati e del **LR**.

Alla fine della subroutine invece, bisogna fare il **POP** dei registri usati e del **PC**.

## Passare parametri e risultato

Ci sono tre approcci:

- **registri**
- by **reference**(un registro con l'indirizzo della memoria)
- attraverso lo **stack**.

### Per registro

```

sub1 PROC
    ;sullo stack pusho il contenuto del LR(quindi indirizzo di ritorno)
    PUSH {LR}
    CMP r0, r1
    ITE HS
    SUBHS r2, r0, r1
    SUBLO r2, r1, r0
    ;salvo in PC il contenuto della cima dello stack(quindi LR)
    POP {PC}
ENDP
  
```

### Per reference

```

MOV r0, #0x34
MOV r1, #0xA3
LDR r3, =mySpace
STMIA r3, {r0, r1} ;parametro passato per reference
BL sub2
LDR r2, [r3] ; r2 contains the result

sub2 PROC
;Mi salvo i valori di r2,r4,r5 (e LR) sullo stack, in quanto
verranno
;modificati
PUSH {r2, r4, r5, LR}
;mi salvo r0 e r1 in r4 e r5
LDMIA r3, {r4, r5}
CMP r4, r5
ITE HS
SUBHS r2, r4, r5
SUBLO r2, r5, r4
;salvo il risultato(r2) in [r3]
STR r2, [r3]
;ripristino r2,r4,r5(e PC)
POP {r2, r4, r5, PC}
ENDP

```

## Per stack

```

MOV r0, #0x34
MOV r1, #0xA3
PUSH {r0, r1, r2};r0 e r1 sono gli argomenti, r2 conterrà il risultato
BL sub3
POP {r0, r1, r2} ; r2 contains the result

sub3 PROC
PUSH {r6, r4, r5, LR}
;salvo in R4 il contenuto di Sp+16 e Sp+20, ovvero r0 e r1
LDR r4, [sp, #16]
LDR r5, [sp, #20]
CMP r4, r5
ITE HS
SUBHS r6, r4, r5
SUBLO r6, r5, r4
;salvo in Sp+24, ovvero dove c'è R2, il risultato(r6)
STR r6, [sp, #24]
;ripristino i registri

```

```
POP {r6, r4, r5, PC}
```

```
ENDP
```

## ABI - Application Binary Interface

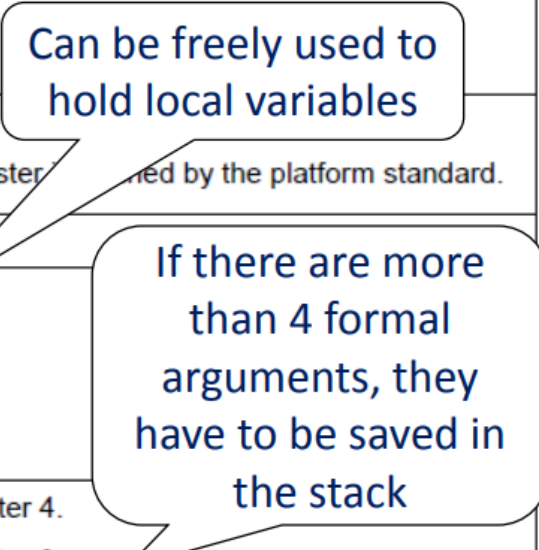
Sono delle specifiche che un eseguibile deve sottostare per girare in un determinato *environment*.

Sono tipo dei protocolli di comunicazione tra Assembly e altri linguaggi (tipo il **C**)

Servono ai realizzatori di compilatori/assemblatori etc per capire come muoversi.

Un esempio di ABI per le chiamate delle *procedures* (ABI AAPCS):

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.



Per gli argomenti, **bisogna** usare i primi 4 registri. Se la nostra procedura ha più di 4 argomenti, bisogna salvarli sullo stack.

Ci sono ovviamente altre regole da tenere in considerazione.