6 ARCHITETTURA

L'**architettura** di un calcolatore è come viene visto il calcolatore a basso livello dal programmatore. Essa è definita da un **set di istruzioni (linguaggio)** e dagli **operandi** di tali istruzioni che sono memorizzati nei registri.
Le parole che costituiscono il linguaggio sono chiamate istruzioni. Tutte le istruzioni definiscono un insieme ossia un set di istruzioni del calcolatore. Tutti i programmi su un calcolatore utilizzano il medesimo set di istruzioni.
Le istruzioni indicano sia l'operazione che il calcolatore deve effettuare sia gli operandi da usare, che si possono trovare in memoria, nei registri del processore o direttamente definiti nelle istruzioni.
Le singole istruzioni vengono poi codificate come stringhe binarie in quello che si definisce **linguaggio macchina**, per poi essere eseguite dal processore. L'architettura ARM rappresenta ogni istruzione con una parola di 32bit.
I processori sono dunque sistemi digitali che leggono ed eseguono istruzioni scritte in linguaggio macchina. Siccome per l'uomo è difficile leggere ed interpretare le istruzioni direttamente in linguaggio macchina si preferisce rappresentare le istruzioni in una forma simbolica: **il linguaggio assembly.**
Tutte le architetture definiscono nel proprio set di istruzioni delle istruzioni di base, che quindi sono uguali per tutte le architetture ad esempio come addizione, salto ecc.
E' bene ricordare che l'architettura non definisce la sottostante struttura circuitale di un calcolatore; infatti esistono spesso differenti realizzazioni circuitali della medesima architettura (come Intel e AMD con l'architettura x86) che possono eseguire gli stessi programmi ma con strutture circuitali diverse; sono magari ottimizzate per particolari applicazioni.
L'organizzazione specifica dei registri del processore, della memoria, dell'ALU e degli altri blocchi circuitali costruttivi del calcolatore viene chiamata microarchitettura.
Possono esistere diverse implementazioni di una microarchitettura per una stessa architettura come menzionato in precedenza.
Noi studiamo l'architettura **ARM acronimo di Advanced Risc Machines,** sviluppata nel 1980.
Si basa su un set di istruzioni RISC: reduced instruction set computer.
Il set RISC si basa su un'idea di progettazione di architetture per microprocessori che predilige lo sviluppo di un'architettura semplice e lineare. Questa semplicità di progettazione permette di realizzare microprocessori in grado di eseguire il set di istruzioni in tempi minori rispetto a una architettura CISC.

L'architettura ARM si basa su **4 principi di progettazione fondamentali:**
1. Regularity supports design simplicity (la regolarità favorisce la semplicità)
2. Make the common case fast (rendere veloci le cose frequenti)
3. Smaller is faster (più piccolo è più veloce)
4. Good design demands good compromises (un buon progetto richiede buoni compromessi).

Il linguaggio assembly è la forma human-readable del linguaggio macchina, nativo del calcolatore. Ogni istruzione in linguaggio assembly specifica sia l'operazione da svolgere che gli operandi da elaborare.

L’operazione più comune dei calcolatori è l’addizione.

Addizione: ADD a, b, c

La prima parte di un istruzione in linguaggio ARM Assembly (in questo caso ADD) è chiamata **mnemonico** e indica l’operazione da eseguire. Tale operazione di addizione deve essere eseguita sugli **operandi sorgente** (b,c) e il risultato ottenuto dall’operazione di addizione su **b** e **c** deve essere scritto in **a**, **l’operando destinazione**.

Sottrazione: SUB a, b, c

Applicazione del 1° principio ARM: Regularity supports design simplicity
Il formato costante delle istruzioni arm è un applicazione del primo principio Regularity supports design simplicity infatti istruzioni con un numero costante di operandi (due sorgenti e una destinazione) sono più facili da gestire in hardware. Istruzioni di alto livello più complesse vengono tradotte in più sequenze di semplici istruzioni arm (ad es. se ho una somma e una sottrazione in sequenza effettuo prima la somma e poi sul risultato ottenuto effettuo la sottrazione ottenendo il risultato finale).
Ogni istruzione è rappresentata da una parola di 32 bit (anche se alcune istruzioni richiederebbero meno di 32 bit di codifica, gestire istruzioni di lunghezza variabile aumenterebbe la complessità). **I formati sono consistenti, e questo facilita l’encoding in hardware.**

Applicazione del 2° principio ARM: Make the common case fast
L’uso di più istruzioni assembly per eseguire attività complesse è un esempio di applicazione del secondo principio: rendere veloci le cose frequenti. Infatti l’architettura ARM rende veloci le cose frequenti perché **comprende solo istruzioni semplici e di uso frequente**. Il numero di istruzioni diverse è tenuto basso in modo tale che il circuito richiesto per decodificare ed eseguire le istruzioni sia semplice e veloce. **Elaborazioni più complesse eseguite più raramente sono realizzate con sequenze di molteplici istruzioni ARM semplici. ARM quindi si basa su un'architettura RISC che ha un insieme limitato di istruzioni**. Architetture con molte istruzioni complesse si basano su un architettura CISC (complex instruction set computer). CISC comporta hardware aggiuntivo e sovraccarichi che rallentano anche le istruzioni semplici. **L’architettura RISC minimizza la complessità hardware e la codifica necessaria per le istruzioni mantenendo piccolo l’insieme di diverse istruzioni.**

Esempio

a = b + c - d;	**ADD t, b, c** *; t = b + c*
	SUB a, t, d *; a = t - d*

Operandi
Un’istruzione lavora su **operandi i quali hanno un nome simbolico e afferiscono a locazioni di memoria fisiche**. Infatti le istruzioni richiedono locazioni fisiche dalle quali prelevare i dati binari. **Gli operandi e quindi i dati possono possono essere memorizzati nei registri del processore, o in memoria oppure essere costanti (immediates) memorizzate nelle istruzioni stesse.**
I calcolatori usano locazioni diverse per gli operandi al fine di ottimizzare velocità e capacità: gli operandi memorizzati come costanti o nei registri sono veloci da raggiungere ma possono contenere solo piccole quantità di dati. Dati aggiuntivi devono essere prelevati dalla memoria,

capiente ma lenta. **ARM opera su dati a 32 bit.** Non esistono istruzioni che operano direttamente sulla memoria centrale. Ogni volta che si deve eseguire un'istruzione su un dato che si trova in memoria si deve fare prima un load del dato dalla memoria in un registro e poi si può operare su quel registro contenente il dato che ci interessa. Il risultato dell'operazione sarà salvato in un altro registro e quindi se si vuole salvare il dato in memoria centrale bisogna effettuare un'operazione di STORE.

Immediate (costanti)
Un immediate è un operando costante memorizzato direttamente all'interno di un'istruzione.
I valori degli immediates sono immediatamente disponibili nell'istruzione stessa e non richiedono accessi a registri o memoria. Un immediate è preceduto dal carattere # **e può essere scritto in decimale o esadecimale**. In assembly ARM le costanti esadecimali cominciano con **0x**. Gli immediates sono numeri senza segno **(unsigned) a 8 o 12 bit.**

Registri: applicazione del 3° principio, smaller is faster
Per quanto riguarda i registri del processore, ARM ha solo **16 registri,** più veloci della memoria e ogni registro dell'architettura è costituito da **32 bit**.
Le istruzioni ARM operano esclusivamente sui registri, quindi i dati residenti nella memoria devono essere spostati in un registro prima di poter essere elaborati.
Infatti le istruzioni hanno bisogno di raggiungere rapidamente gli operandi per poter essere eseguite velocemente, ma gli operandi salvati nella memoria richiedono tempi lunghi d'accesso in memoria per poter essere recuperati. **Per questo vengono definiti un numero limitato di registri per memorizzare gli operandi più usati.**
ARM ha 16 registri globalmente indicati come **register file**. **Meno sono i registri e più rapidamente sono accessibili, questa è un'applicazione del 3° principio più piccolo è più veloce.** Infatti leggere un dato da pochi registri è molto più rapido che leggerlo da una memoria grande. Il register file di solito è costituito da un array di memoria SRAM.

ARM ha 16 registri a 32 bit (R0… R15) che sono fisicamente equivalenti fra loro, ma dal punto di vista logico sono usati con scopi specifici, soprattutto per risolvere il problema delle chiamate a funzione. Per questo sono state introdotte delle convenzioni.

Name	Use
R0	Argument / return value / temporary variable
R1-R3	Argument / temporary variables
R4-R11	Saved variables
R12	Temporary variable
R13 (SP)	Stack Pointer
R14 (LR)	Link Register
R15 (PC)	Program Counter

argomento (argument): [registri R0 a R3] il registro può essere utilizzato per memorizzare un argomento (o parametro) che la funzione chiamante passa alla funzione chiamata durante le chiamate a funzione.

valore di ritorno (return value): [registro R0] è un valore che viene memorizzato solo dal registro R0. Nelle chiamate a funzione, una volta che la funzione chiamante ha chiamato una subroutine, questa funzione chiamata viene eseguita e deve ritornare un valore di ritorno al suo

termine, che viene memorizzato sempre nel registro R0. Questo è il motivo per cui possiamo ritornare solo un valore per ogni funzione in C ad esempio, visto che abbiamo un unico registro a disposizione per memorizzare il valore di ritorno. La funzione chiamante, al termine della subroutine, va a vedere nel registro R0 il valore di ritorno della funzione chiamata.

variabile temporanea (temporary variable): [registri R0 a R3] le variabili si dividono in temporanee e saved variables. Le temporary variables sono quelle variabili che una funzione utilizza in maniera temporanea e che possono essere modificate da altre funzioni. Al termine dell'esecuzione i valori nei registri che contengono la variabile possono cambiare.

saved variables: [registri R4 a R11] sono delle variabili contenute nei registri che non possono essere modificate; quando la funzione chiamante chiama una subroutine, quando essa ritorna il valore in R0 deve lasciare i valori contenuti nei registri su cui ha operato così come li ha trovati al momento della chiamata a funzione. La funzione chiamata se vuole operare su questi registri deve prima memorizzare i valori iniziali di questi registri, salvandoli in memoria con un operazione di STORE, dopodiché può operare su questi registri ma prima di ritornare alla funzione chiamante deve effettuare un operazione di LOAD dalla memoria per ripristinare i valori iniziali dei registri saved variables che ha utilizzato, prima di ritornare il valore di ritorno alla funzione chiamante. In questo modo la funzione chiamante non si "accorge" delle operazioni effettuate su quei registri.

I registri R13, R14 e R15 gestiscono il processo di chiamata e di ritorno di una funzione

R15 (PC - program counter): fra questo registro e il program counter effettivo c'è sempre una differenza di 4, vuol dire che R15 punta sempre all'istruzione successiva a quella che sta per essere eseguita. Il PC contiene l'indirizzo dell'istruzione corrente che si sta eseguendo, non si può utilizzare come operando.

R14 (LR - link register): serve per effettuare il ritorno dalla funzione chiamata alla funzione chiamante

R13 (SP - stack pointer): contiene l'indirizzo che punta alla testa dello stack utilizzato per le chiamate a funzione. Infatti si deve organizzare la memoria in modo tale che due funzioni non vadano ad occupare entrambe le stesse locazioni di memoria; non si deve fare in modo che per esempio la funzione chiamante ha salvato un valore in memoria ad un certo indirizzo e la funzione chiamata utilizza proprio l'indirizzo di quel valore salvato per salvare altri dati. Quindi deve esserci un'organizzazione dello spazio degli indirizzi in memoria in modo che nessuna funzione occupi lo spazio di un'altra funzione: ciò è realizzato mediante uno stack o pila.

Istruzione MOV
L'istruzione mov è utile per inizializzare i valori dei registri e quindi delle variabili. In particolare inizializza variabili con immediates, spostando il contenuto della costante in un registro (es. MOV R4, #0 oppure MOV R5, #0xFF0) . Oppure l'istruzione mov può usare anche registri come operandi sorgente copiando il contenuto di un registro in un altro registro (es. MOV R7, R1). Quindi MOV viene usato per spostare il contenuto di un registro in un altro registro o per generare delle costanti.

Memoria
I dati possono essere salvati anche in memoria che al contrario dei registri che sono pochi e veloci, essa è grande e più lenta. Per questo motivo le variabili utilizzate spesso vengono tenute nei registri. Siccome in ARM le istruzioni operano solo sui registri , i dati presenti in memoria devono essere copiati nei registri prima di essere elaborati.

Istruzioni condizionali ed iterative

Costrutto if in assembly
Il codice assembly del costrutto if valuta la condizione opposta rispetto a quella presente nel codice di alto livello. Dal momento che ogni istruzione ARM può essere condizionata, il codice assembly arm può essere sostituito in forme più compatte usando gli mnemonici tipo EQ, NE.. detti Mnemonici di condizione (tabella 6.3 pag. 227)
La versione con istruzioni condizionate è più corta e anche più veloce perchè richiede di eseguire meno istruzioni. Poi i salti a volte introducono ritardi mentre l'esecuzione condizionata di istruzioni è sempre veloce. In generale quando un blocco di codice è costituito da una sola istruzione conviene usare l'esecuzione condizionata invece dei salti, che diventano utili quando il blocco di codice è più lungo perché evitano la fase di fetch di istruzioni che poi non saranno eseguite.

CMP: effettua la sottrazione tra i due registri e aggiorna i flag di stato NZCV

if/else: valuta la condizione opposta rispetto a quella presente nel codice di alto livello. se la condizione opposta è vera, l'istruzione di salto salta il blocco if e va eseguire il blocco else. altrimenti si esegue il blocco if che termina con un salto incondizionato per saltare il blocco else, anche in questo caso ogni istruzione può essere condizionata usando una forma compatta con mnemonici condizionali.

costrutto while: come per il csotrutto if/else il codice assembly del ciclo while valuta la condizione opposta rispetto a quella presente nel codice di alto livello: se la condizione opposta è vera allora si esce dal ciclo, altrimenti il salto condizionato non viene effettuato e si esegue il corpo del ciclo.

costrutto for: ha una semantica del seguente tipo
for (initialization; condition; loop operation)
statement

- initialization: eseguita prima che il loop inizi
- condition: condizione di continuazione che è verificata all'inizio di ogni iterazione
- loop operation: eseguita alla fine di ogni iterazione
- statement: eseguito ad ogni iterazione, ovvero fintantoché la condizione di continuazione è verificata

In ARM, i loop decrescenti fino a 0 sono più efficienti.
Si risparmiano 2 istruzioni per ogni iterazione:
- Si accorpano decremento e comparazione: **SUBS** R0, R0, #1
- Solo un branch invece di due

Arrays

Per facilitare memorizzazione e accessi dati simili possono essere raggruppati in una struttura chiamata array. L'array consente l'accesso ad una quantità di dati simili. Esso memorizza i suoi dati (elementi) in indirizzi sequenziali di memoria. Ogni elemento dell'array è identificato da un numero detto Index che da accesso al singolo elemento. Il Size dell'array definisce il numero di elementi dell'array.

In un array (parole di 32 bit) l'indirizzo del primo elemento è detto Base address e rappresenta l'array. Tutti gli altri elementi dell'array sono raggiungibili a partire dal base address.

Siccome l'array è costituito da parole di 32 bit, gli elementi dell'array avranno indirizzi che vanno di 4 in 4.

#0 = array[0]
#4 = array[1]
#8 = array[2]

C Code

```
int array[5];
array[0] = array[0] * 8;
array[1] = array[1] * 8;
```

ARM Assembly Code

```
; R0 = array base address
  MOV R0, #0x60000000          ; R0 = 0x60000000

  LDR R1, [R0]                 ; R1 = array[0]
  LSL R1, R1, 3                ; R1 = R1 << 3 = R1*8
  STR R1, [R0]                 ; array[0] = R1

  LDR R1, [R0, #4]             ; R1 = array[1]
  LSL R1, R1, 3                ; R1 = R1 << 3 = R1*8
  STR R1, [R0, #4]             ; array[1] = R1
```

C Code

```
int array[200];
int i;

for (i=199; i >= 0; i = i - 1)
     array[i] = array[i] * 8;
```

ARM Assembly Code

```
; R0 = array base address, R1 = i
  MOV R0, 0x60000000
  MOV R1, #199

FOR
  LDR  R2, [R0, R1, LSL #2]     ; R2 = array(i)
  LSL  R2, R2, #3               ; R2 = R2<<3 = R2*8
  STR  R2, [R0, R1, LSL #2]     ; array(i) = R2
  SUBS R1, R1, #1               ; i = i - 1
                                ; and set flags
  BPL  FOR                      ; if (i>=0) repeat loop
```

Chiamate di funzioni
Le funzioni ricevono valori di ingresso denominati parametri e forniscono un valore di uscita denominato valore di ritorno. Quando una funzione ne chiama un'altra, la prima funzione detta caller (chiamante) e la seconda funzione detta callee (chiamato) devono accordarsi stipulando una specie di "contratto" su dove mettere i parametri e il valore di ritorno.
In ARM convenzionalmente il caller mette fino a 4 parametri nei registri R0-R3 (argument) prima di eseguire la chiamata a sottoprogramma e il callee mette il valore di ritorno nel registro R0 prima di terminare. Con questa convenzione entrambe le funzioni sanno dove trovare parametri e valore di ritorno. Il callee non deve interferire con le attività del caller e non deve invadere il suo spazio di memoria. Questo significa che deve sapere dove mettere il valore di ritorno ma anche che non deve modificare nessuno dei registri o delle celle di memoria necessari al chiamante.

Per contratto (convenzione):
- il caller deve passare gli argomenti alla funzione chiamata (callee) e poi deve eseguire un jump alla prima istruzione sempre della funzione chiamata (callee).
-il callee ossia la funzione chiamata esegue le istruzioni contenute al suo interno e poi deve ritornare il risultato finale al caller, ritornando nel punto in cui è stata chiamata dal caller. Poi non deve sovrascrivere i registri saved variables (R4-R11) e la memoria che è occupata dal caller.

Convenzioni ARM per chiamata a funzione
Il passaggio dal caller al callee viene effettuato attraverso un'istruzione di BL (branch and link), compito del caller. **L'istruzione di BL memorizza l'indirizzo di ritorno (cioè l'indirizzo dell'istruzione successiva a BL del caller) nel registro LR (link register) cioè R14 e poi salta al callee che esegue le sue istruzioni.**
Dopo che il callee ha eseguito le istruzioni al suo interno deve passare il valore di ritorno e tornare al caller. Ciò viene eseguito attraverso un'istruzione di MOV che prima salva il valore di ritorno nel registro R0 e poi con un'altra istruzione di MOV si copia il valore del LR (cioè l'indirizzo di ritorno al caller) nel PC (ossia il registro R15, program counter) così si ritorna al punto in cui il programma chiamante ha chiamato il callee.
Chiamata a funzione: branch and link BL
Return da funzione: ripristina in PC il valore del link register. MOV PC, LR
Argomenti: R0-R3
Valore di ritorno: R0

Parametri di ingresso e valori di ritorno. Le saved variables e lo stack delle funzioni.
Per convenzione ARM le funzioni usano i registri R0-R3 per i parametri di ingresso e il registro R0 per il valore di ritorno. Se è necessario chiamare una funzione con più di 4 parametri, i parametri aggiuntivi vanno messi nello stack e non è possibile sovrascrivere indebitamente i registri R4-R11 dedicati alle saved variables.
Quindi una funzione usa lo stack per memorizzare temporaneamente i valori di questi registri prima di operare su di essi.

Infatti se una funzione chiamata ha bisogno di utilizzare questi registri "saved variables" può salvare i valori dati dal programma chiamante subito in memoria con operazioni di store STR, operare con questi registri , e poi prima di ritornare al chiamante ripristinare i valori di questi registri mediante operazioni di load LDR a partire dagli indirizzi in cui si erano salvati il contenuto di questi registri inizialmente.
Quindi per salvare i valori di questi registri prima di operare su di essi viene utilizzato lo stack, nel quale si salvano questi valori. Se salvassimo questi valori in indirizzi della memoria a caso si potrebbero sporcare delle aree di memoria riservate ad altri programmi o aree di memoria che la

funzione chiamante ha utilizzato. Lo stack fa si che non ci sia confusione negli indirizzi di memoria che si vanno ad utilizzare, cioè che ogni funzione utilizzi porzioni di memoria diverse.

Lo stack delle funzioni
Lo stack costituisce un modo semplice di organizzare la memoria per salvare temporaneamente il valore dei registri "saved variables" che servono temporaneamente ad una funzione chiamata (callee). Ogni chiamata a funzione ha una sua porzione ben definita dello stack. Esso è organizzato come una pila, che funziona secondo una tecnica detta LIFO (last in first out).
Le operazioni sullo stack sono di Expands e Contracts.
Expands: lo stack si espande, cioè usa più memoria, quando ha bisogno (operazione di push)
Contracts: lo stack si contrae, cioè libera memoria, quando non si hanno più bisogno delle variabili temporanee memorizzate in precedenza (operazione di pop)
Lo stack è di tipo LIFO perché l'ultimo elemento messo sullo stack cioè l'elemento pushed, è il primo che potrà essere estratto (cioè popped).

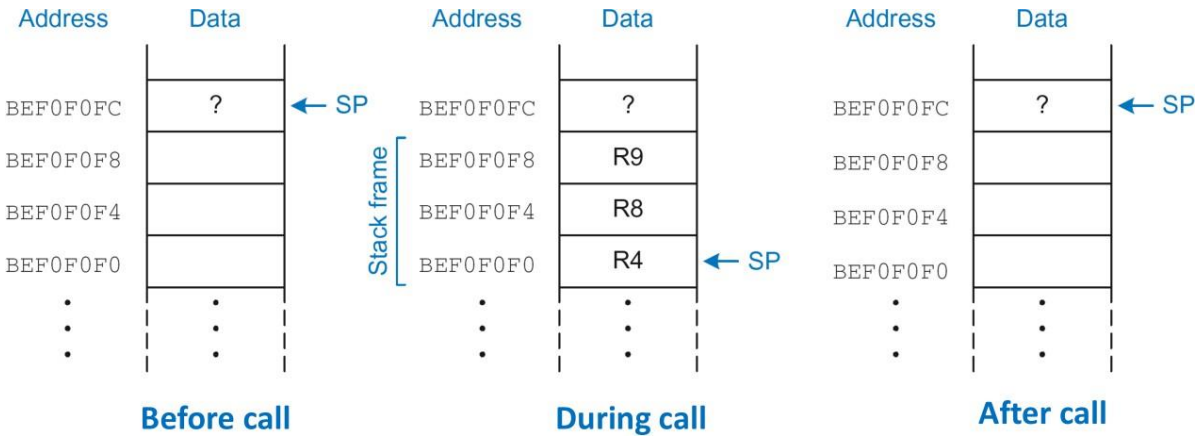
Una funzione può allocare spazio sullo stack per memorizzare le variabili locali, ma deve deallocare tale spazio prima di ritornare alla funzione chiamante.
Lo stack di ARM cresce in memoria verso il basso, cioè l'espansione dello stack avviene in senso decrescente, dagli indirizzi maggiori verso indirizzi minori quando il programma ha bisogno di spazio ulteriore.

Il registro R13 (SP - stack pointer) è un registro che contiene un indirizzo che "punta" alla cima dello stack, cioè all'ultimo elemento inserito. Lo stack pointer SP inizia a un indirizzo di memoria alto e si decrementa se serve spazio aggiuntivo. Ad esempio se deve contenere ulteriori parole di memoria temporanea l'indirizzo si decrementa di 4,8,16 byte ecc espandendo lo stack.

Uno degli usi più importanti dello stack è il salvataggio e ripristino dei valori dei registri "saved variables" usati da una funzione, senza modificare alcun registro tranne R0 che contiene il valore di ritorno.
La funzione chiamata deve salvare i registri sullo stack prima di modificarli e ripristinare i valori originari prima di terminare. In particolare deve svolgere i seguenti passi:
- allocare inizialmente spazio sullo stack per memorizzare i valori contenuti in uno o più registri
- salvare i valori dei registri nello stack
- eseguire le proprie attività utilizzando i registri salvati
- ripristinare i valori originali dei registri prelevandoli dallo stack
- deallocare spazio nello stack

ARM Assembly Code

```
; R2 = result
DIFFOFSUMS
  SUB SP, SP, #12     ; make space on stack for 3 registers
  STR R4, [SP, #-8]   ; save R4 on stack
  STR R8, [SP, #-4]   ; save R8 on stack
  STR R9, [SP]        ; save R9 on stack
  ADD R8, R0, R1      ; R8 = f + g
  ADD R9, R2, R3      ; R9 = h + i
  SUB R4, R8, R9      ; result = (f + g) - (h + i)
  MOV R0, R4          ; put return value in R0
  LDR R9, [SP]        ; restore R9 from stack
  LDR R8, [SP, #-4]   ; restore R8 from stack
  LDR R4, [SP, #-8]   ; restore R4 from stack
  ADD SP, SP, #12     ; deallocate stack space
  MOV PC, LR          ; return to caller
```



Registri preservati e non preservati

Preserved *Callee-Saved*	Nonpreserved *Caller-Saved*
R4-R11	R12
R14 (LR)	R0-R3
R13 (SP)	CPSR
stack above SP	stack below SP

ARM divide i registri in preservati e non preservati.
Ogni funzione deve salvare e ripristinare tutti i registri preservati che intende usare, ma può modificare liberamente i registri non preservati.
Visto che il callee può modificare qualsiasi registro non preservato, è responsabilità del

chiamante caller salvare il contenuto di questo tipo di registri prima di effettuare la chiamata, se gli servono, e ripristinarlo dopo il ritorno del callee.
La parte di stack più in alto dello stack pointer è automaticamente preservato dato che il callee evita qualsiasi modifica di parole di memoria a indirizzi maggiori di SP: in questo modo si evita di accedere agli stack frame di altre funzioni. Lo stack pointer è esso stesso preservato, perché il callee dealloca lo spazio allocato all'inizio prima di terminare, sommando a SP lo stesso valore che aveva sottratto.

Il callee quindi ha il compito di preservare i valori dei registri "saved" quindi deve effettuare dei STR e LDR dei registri R4-R11 prima di usarli (str) e dopo averli usati (ldr). Non deve sporcare il valore del LR e deve fare in modo di ripristinare il valore iniziale dello SP. Non deve sporcare nessun indirizzo che sta al di sopra dello SP.

Il caller deve salvarsi i valori dei registri temporanei R0-R3 , R12 se gli servono poiché temporanei. Deve salvarsi i valori del CPSR se gli servono. Non deve salvare qualcosa negli indirizzi al di sotto dello SP.

Funzioni pop e push in assembly
Salvare e ripristinare registri dallo stack è una operazione così frequente che ARM mette a disposizione delle istruzioni specifiche, Pop e Push, che consentono di avere un codice più succinto e di evitare la gestione a basso livello dello stack con i LDR e STR dei registri "saved".

- **Push** consente di salvare in memoria più registri aggiornare consistentemente lo stack:
PUSH {R4, R8, R9} *//decrementa di 12 il valore iniziale dello SP*

- **Pop** ripristina uno o più registri e incrementa consistentemente il valore dello stack:
PUSH {R4, R8, R9} *//incrementa di 12 il valore iniziale dello SP*

Chiamate a funzioni innestate
Tutte le funzioni che richiamano altre funzioni innestate in esse devono fare un push del LR corrente all'inizio nello stack per non sporcare e perdere il LR quando loro termineranno e dovranno ritornare il valore di ritorno alla funzione che le ha chiamate (caller), dato che il LR viene sovrascritto dalla funzione innestata

C Code

```
int f1(int a, int b) {
  int i, x;
  x = (a + b)*(a - b);
  for (i=0; i<a; i++)
    x = x + f2(b+i);
  return x;
}
int f2(int p) {
  int r;
  r = p + 5;
  return r + p;
}
```

ARM Assembly Code

```
; R0=a, R1=b, R4=i, R5=x
F1
  PUSH {R4,  R5, LR}
  ADD   R5,  R0, R1
  SUB   R12, R0, R1
  MUL   R5,  R5, R12
  MOV   R4,  #0
FOR
  CMP   R4, R0
  BGE   RETURN
  PUSH {R0, R1}
  ADD   R0, R1, R4
  BL    F2
  ADD   R5, R5, R0
  POP  {R0, R1}
  ADD   R4, R4, #1
  B     FOR
RETURN
  MOV   R0, R5
  POP  {R4, R5, LR}
  MOV   PC, LR
```

```
; R0=p, R4=r
F2
  PUSH {R4}
  ADD   R4, R0, 5
  ADD   R0, R4, R0
  POP  {R4}
  MOV   PC, LR
```

Chiamate ricorsive
Si comporta sia da chiamante che da chiamato quindi deve salvare registri preservati e nonpreservati

Fattoriale

C Code

```
int factorial(int n) {

  if (n <= 1)
    return 1;



  else
    return (n * factorial(n-1));
}
```

ARM Assembly Code

```
0x94 FACTORIAL  STR R0, [SP, #-4]!
0x98            STR LR, [SP, #-4]!
0x9C            CMP R0, #2
0xA0            BHS ELSE
0xA4            MOV R0, #1
0xA8            ADD SP, SP, #8
0xAC            MOV PC, LR
0xB0 ELSE       SUB R0, R0, #1
0xB4            BL  FACTORIAL
0xB8            LDR LR, [SP], #4
0xBC            LDR R1, [SP], #4
0xC0            MUL R0, R1, R0
0xC4            MOV PC, LR
```