03 - SUBROUTINE E FUNZIONI

Una **PROCEDURA o SUBROUTINE** è una sequenza di istruzioni che può essere invocata da un altro (o da un'altra parte di un) programma.

- Può essere invocata in più punti.
- Permette di riutilizzare codice.
- Permette di realizzare una struttura modulare.
- Il processore esegue il codice della procedura e poi torna al punto in cui era stata chiamata.

CHIAMATA A SUBROUTINE

Il meccanismo di chiamata a procedura:

- Un'istruzione per *chiamare la procedura*.
- Un'istruzione per *ritornare al punto iniziale*.

Il *punto di ritorno* è ogni volta diverso: è l'istruzione successiva a quella di chiamata.

DEFINIZIONE DI UNA FUNZIONE

Una **FUNZIONE** viene indicata in modo univoco dall'indirizzo in memoria della sua prima istruzione. In assembly si utilizza una *label* per definire tale indirizzo.
Per *chiamare* una funzione si eseguirà un *branch* alla label corrispondente.

RITORNO DA UNA FUNZIONE

1a SOLUZIONE

L'indirizzo di ritorno potrebbe venire salvato in una *locazione della memoria adibita* a tale scopo, per esempio quella all'indirizzo 0.

- **LIMITAZIONE**: se la subroutine ne chiamasse un'altra, quest'indirizzo verrebbe perso.

Per chiamare una procedura PROC:

```
    MOV R0, #0    @ Indirizzo dove salvare PC
    LDR R1, =RIT  @ Indirizzo di ritorno
    STR R1, [R0]  @ Salva RIT in M[0]
    B PROC        @ Salta alla procedura PROC
RIT: ....
```

Per ritornare al punto di partenza alla fine di PROC:

```
    MOV R0, #0
    LDR R0, [R0]
    MOV PC, R0    @ Ripristina PC all'indirizzo che era in M[0]
```

2a SOLUZIONE

L'indirizzo di ritorno potrebbe venire salvato nella *prima locazione di ciascuna subroutine*, con la convenzione che le istruzioni eseguibili della subroutine siano collocate a partire dalla locazione successiva.

- **LIMITAZIONE**: questa soluzione consente alla subroutine di chiamarne un'altra, ma se chiamasse se stessa verrebbe perso l'indirizzo di ritorno.

Per chiamare la procedura PROC:

```
    LDR R0, =PROC  @ Indirizzo dove salvare RIT
    LDR R1, =RIT   @ Indirizzo di ritorno
    STR R1, [R0]   @ Salva RIT in M[PROC]
    B PROC+4       @ Salta alla prima istruzione di PROC
RIT: ...
```

Struttura della procedura PROC:

```
PROC: .space 4     @ Word per l'indirizzo di ritorno
      .....        @ codice della funzione
    LDR R0, =PROC
    LDR R0, [R0]
    MOV PC, R0
```

3a SOLUZIONE

L'indirizzo di ritorno viene salvato in un *registro di CPU* anzichè in memoria.

- **LIMITAZIONE**: la subroutine PROC può chiamarne un'altra solo utilizzando un registro diverso.

Per chiamare la procedura PROC:

```
    LDR R14, =RIT  @ Indirizzo di ritorno
    B PROC         @ Salto a PROC
RIT:
```

Per ritornare al punto di partenza alla fine di PROC:

```
PROC: ...          @ Istruzioni della funzione
      ...
      MOV PC, R14  @ PC torna a RIT
```

4a SOLUZIONE

Le limitazioni delle soluzioni precedenti permettono di *non perdere l'indirizzo di ritorno*. Tuttavia possono essere evitate salvando l'indirizzo di ritorno in una posizione sicura, come lo *stack,* per poi essere ripristinato.

L'indirizzo di ritorno viene memorizzato in uno stack con un'operazione di *push*: ogni volta è in un posto diverso. Tale soluzione permette alla subroutine PROC di chiamarne *un'altra*, o

anche *se stessa*.

Le subroutine realizzate con tale soluzione sono *rientranti*: di esse possono essere in esecuzione più istanze contemporaneamente (condizione per poter chiamare ricorsivamente la procedura).

Per chiamare la procedura PROC:

```
    LDR R0, =RIT
    PUSH {R0}      @ RIT viene salvato nello stack
    B PROC
RIT:
```

Per tornare al punto di partenza alla fine di PROC:

```
PROC: ...
      ...
      POP{PC}      @ Carica in PC l'ultimo valore salvato nello stack
```

IMPLEMENTAZIONE ARM

ARM usa un mix delle *soluzioni 3 e 4*:

- Per invocare una procedura PROC si salva il punto di ritorno nel registro R14 (**LR** *link register*).
- Se è necessario invocare altre procedure all'interno di PROC, si salva prima il valore di LR nello stack.
- Permette di sfruttare l'efficienza della soluzione 3, ma permette anche di invocare altre procedure ad un costo leggermente superiore (che è comunque inevitabile).

Una procedura PROC viene chiamata con:

```
BL PROC
```

L'istruzione **BL** (*branch and link*) salva l'indirizzo dell'istruzione successiva a BL in R14 e carica in PC l'indirizzo della subroutine.

Tuttavia non esiste un'istruzione apposita per il ritorno da subroutine. Bisogna usare l'istruzione:

```
MOV PC, LR
```

PASSAGGIO DEI PARAMETRI

Si distinguono due tipi di parametri:

- Parametri di ingresso: dati passati alla subroutine.
- Parametri di uscita: risultati restituiti dalla subroutine.

Il veicolo più naturale e rapido per effettuare il passaggio dei parametri è costituito dai registri di CPU.

PASSAGGI PER VALORE E PER INDIRIZZO

PASSAGGIO PER VALORE

ESEMPIO: Calcolare il modulo della differenza tra due interi in complemento a due. Supponiamo di usare i registri R0, R1 e R2 per passare i valori dei parametri:

- R0: parametro di uscita
- R1 e R2: parametri di ingresso.

Funzione ABS:

```
ABS: SUBS R0, R1, R2     @ Calcola R1-R2
     RSBMI R0, R1, R2    @ Se negativo, calcola R2-R1
     MOV PC, LR          @ Ritorna
```

Chiamata di ABS:

```
.data
OP1: .word 13            @ Primo operando
OP2: .word 9             @ Secondo operando

.bss
RIS: .skip 4             @ Spazio per risultato

.text
 LDR R3, =OP1            @ Carica operandi in R1 e R2
 LDR R1, [R3]
 LDR R3, =OP2
 LDR R2, [R3]
 BL ABS                  @ Chiamata ABS
```

```
 LDR R3, =RIS            @ Salva risultato
 STR R0, [R3]
```

PASSAGGIO PER INDIRIZZO

Anzichè i valori dei parametri, è possibile passare gli *indirizzi delle locazioni* di memoria ove quei valori sono contenuti.
Utilizziamo i registri come prima.

Funzione ABS:

```
ABS: LDR R1, [R1]        @ Carica gli operandi da memoria
     LDR R2, [R2]
     SUBS R3, R1, R2     @ Calcola
     RSBMI R3, R1, R2
     STR R3, [R0]        @ Salva risultato in memoria
     MOV PC, LR          @ Ritorna
```

Chiamata di ABS:

```
.data
OP1: .word 13
OP2: .word 9

.bss
RIS: .skip 4

.text
 LDR R1, =OP1            @ Carica indirizzi operandi
 LDR R2, =OP2
 LDR R0, =RIS            @ Carica indirizzo risultato
 BL ABS                  @ Chiama ABS
```

PARAMETRI PER VALORE O PER INDIRIZZO?

PER INDIRIZZO:

- Avendo a disposizione l'indirizzo, la subroutine può accedere al *valore originario* del parametro e *modificarlo*.

PER VALORE:

- Se alla subroutine viene passato il valore del parametro (la sua *copia*), eventuali modifiche apportate a questo valore non coinvolgono il valore originario.

DISCIPLINA DI PROGRAMMAZIONE

Conviene che, dopo l'esecuzione di una subroutine, risulti *modificato solo ciò che è espressamente previsto* che la subroutine modifichi, cioè i soli parametri di uscita.

Se la subroutine *modificasse i valori originari* dei parametri di ingresso, questo sarebbe un *effetto collaterale indesiderato* della sua esecuzione.

UTILIZZO DELLO STACK

Una subroutine può avere la necessità di utilizzare dei *registri* per collocarvi i suoi *risultati intermedi.* Questo utilizzo può violare la disciplina sopra spiegata.

Conviene allora utilizzare lo **STACK** per:

- *Salvare il contenuto* dei registri usati dalla subroutine prima di modificarli.
- *Ripristinare il contenuto* di questi registri prima di ritornare al programma chiamante.
- Ciò vale in particolare per il registro **LR/R14**.

SALVATAGGI NELLO STACK

```
      BL SUB1
      ...

SUB1 PUSH {LR, R0-R2}     @ Salva il contenuto di LR e dei registri usati
      ...
      LDR R2, [R1, #4]!
      ADD R0, R0, R2
      BL SUB2
      ...
      POP {LR, R0-R2}     @ Ripristina LR e i registri usati
      MOV PC, LR          @ Ritorna
```

NOTA: POP colloca i registri in memoria assegnando indirizzi crescenti con l'indice dei registri, indipendentemente dall'ordine scritto nell'istruzione.

Il salvataggio del contenuto dei registri nello stack permette l'*annidamento di subroutine*.

ALLOCAZIONE DINAMICA

SUBROUTINE RIENTRANTI

Una subroutine si dice *rientrante* se di essa può iniziare una nuova esecuzione mentre è ancora in corso una sua esecuzione precedente. Questo può accadere:

- In seguito a una *chiamata ricorsiva*.
- In seguito a una *interruzione*: il processore passa ad eseguire un altro programma che chiama la medesima subroutine interrotta.
- In un *sistema multiprocessore*: se la subroutine si trova in memoria condivisa, più di un processore può iniziarne l'esecuzione.

Per essere rientrante, una subroutine *non deve alterare* i dati su cui stava operando ogni sua attivazione precedente non terminata. Vedi > IMPLEMENTAZIONE ARM e > UTILIZZO DELLO STACK.

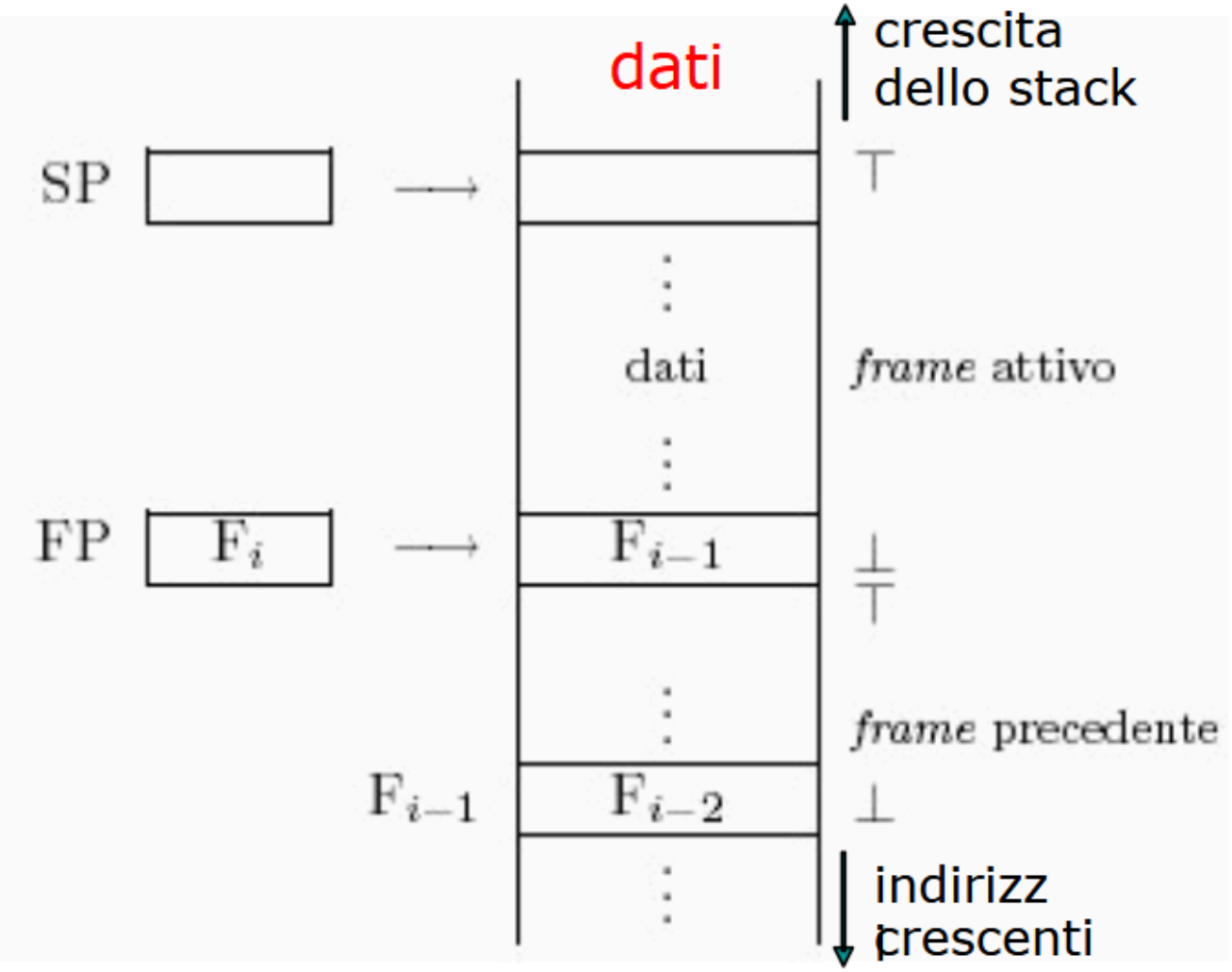
NOTA: i dati non fanno parte della subroutine, ma della sua *istanza d'esecuzione*. La subroutine invece è fatta di solo codice

I dati che devono essere allocati ogni volta in un posto diverso sono i *parametri di ingresso e di uscita* (compreso l'indirizzo di ritorno) e i *dati locali* su cui la subroutine opera.

ALLOCAZIONE DINAMICA DELLA MEMORIA

La soluzione comunemente adottata prevede che al momento di attivare una subroutine venga allocata in cima allo stack un'area (**STACK FRAME**) in cui salvare tali dati. Quest'area verrà poi rimossa dallo stack quando l'esecuzione termina.
Tale soluzione si chiama **allocazione dinamica della memoria**.

Si utilizza un registro che svolge la funzione di **Frame-Pointer (FP)** e punta all'inizio del *frame*, ovvero all'indirizzo più alto dell'area di memoria occupata dall'istanza della subroutine.



OSS: Lo stack cresce verso indirizzi più bassi, quindi l'inizio del frame è l'indirizzo più alto dell'area allocata.

STACK-FRAME

In generale, quando una subroutine viene chiamata si alloca un nuovo frame e quando la sua esecuzione termina l'area di memoria viene rilasciata.

ALLOCAZIONE di un NUOVO FRAME:

- FP --> push (il vecchio frame pointer viene salvato)
- SP --> FP (FP ora punta alla testa del nuovo frame)
- SP-e --> SP (SP viene alzato)

RILASCIO DELL'AREA DI MEMORIA:

- FP --> SP (SP ora punta alla base del frame, cioè alla testa di quello precedente)
- pop --> FP (il vecchio FP viene ripristinato)

ESEMPIO DI SUBROUTINE RICORSIVA

Esempio (codice in C):

```c
void R(int I, int J, int *O){
    int A,B,C;

    //...

    *O = I+J;

    if(A==0)
        R(A, B, &C);

    return;
}

int main(){
    int W, X, Y, Z;
    //...
    R(X, Y, &Z);
}
```

Main:

- W, X, Y, Z sono *dati locali*.

Subroutine R:

- I, J sono *parametri di ingresso* (per valore).
- *O è *parametro d'uscita* (per indirizzo).
- A, B, C sono *dati locali*.

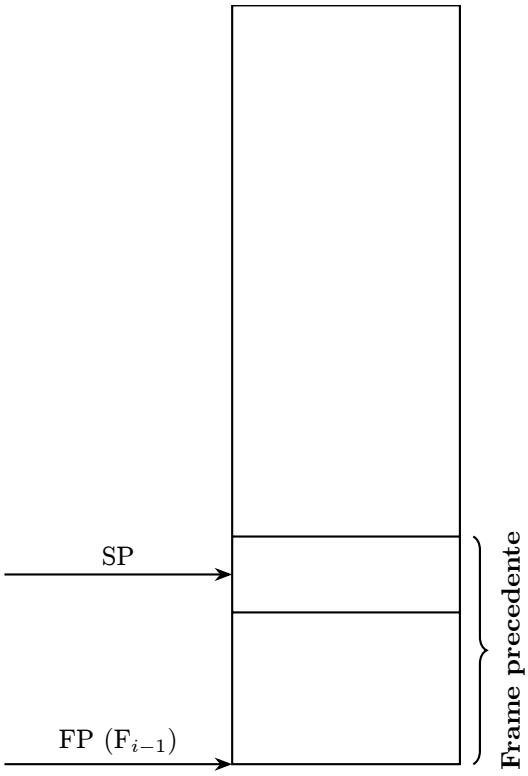
Vediamo l'equivalente in Assembly ARM:

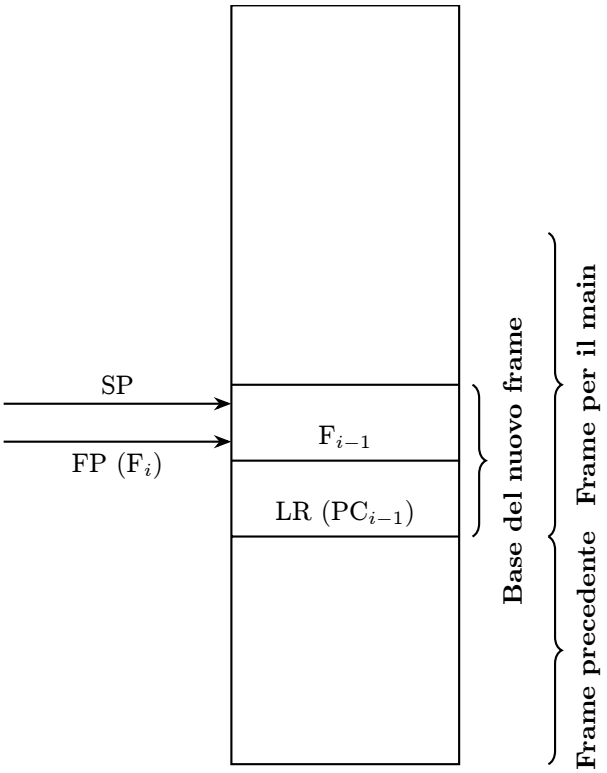
1 - Allocazione frame per il main

Viene allocato un *frame* per il *main*, che ha solo dati locali:

```
main:
 PUSH {FP,LR}
 MOV FP, SP
```

PRIMA e **DOPO** l'allocazione del nuovo frame:

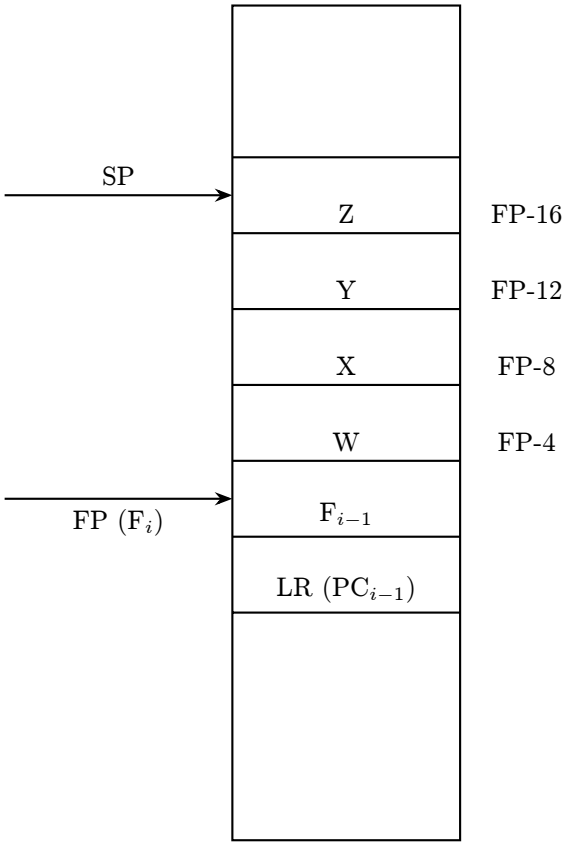




- Vengono pushati nello stack *LR* (punto di ritorno) e *FP* (Frame pointer precedente). Questi costituiscono la base del nuovo frame.
- Il **Frame Pointer** viene spostato a **Stack Pointer** (testa dello stack), dove si trova ora l'indirizzo del vecchio *FP*.

NOTA: F_{i-1} è il frame precedente al main, F_i è il frame del main.

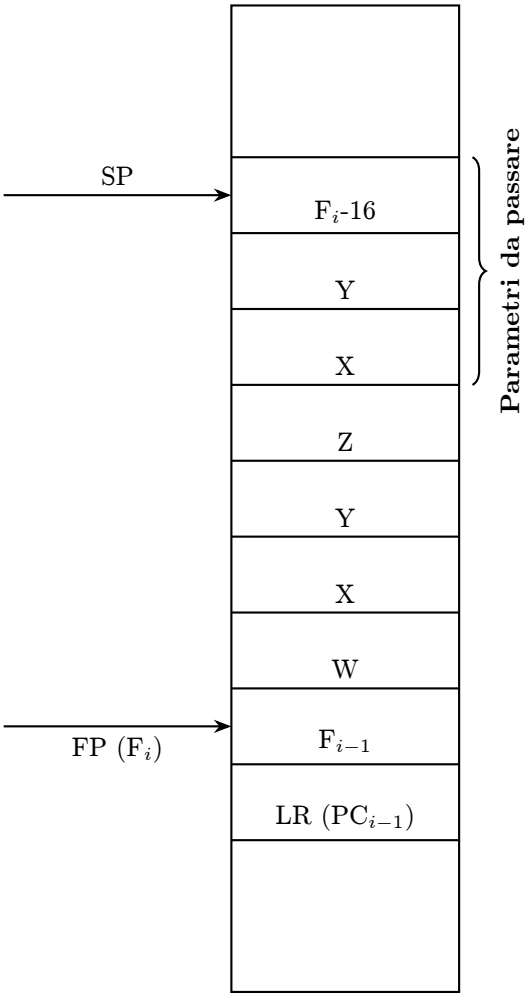
Nel main c'erano *4 parametri locali*: si trovano nelle posizioni da FP-4 a FP-16 (man mano che si sale verso la testa dello stack, gli indirizzi di memoria sono decrescenti).



2 - Chiamata alla subroutine R

```
LDR R0, [FP, #-8]
PUSH {R0}
LDR R0, [FP, #-12]
PUSH {R0}
SUB R0, FP, #16
PUSH {R0}

BL R
```

I parametri *X* e *Y* (FP-8 e FP-12) sono passati per *valore*: vengono copiati in R0 e pushati nello stack.
Z è passato per *indirizzo*: in R0 viene mezzo l'indirizzo di Z, che viene poi pushato nello stack.

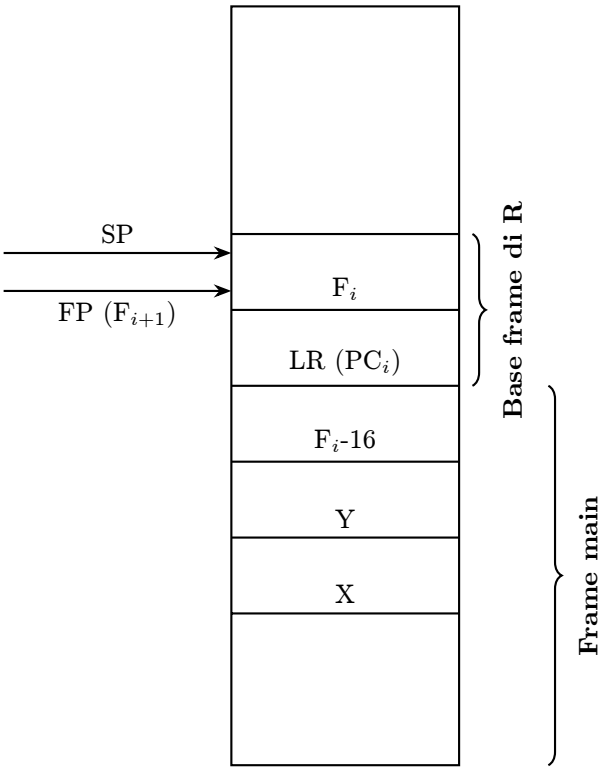


OSS: Vengono aggiunti nello stack, in ordine, i parametri da passare alla funzione: valore di *X*, valore di *Y* e indirizzo di *Z* ($F_i - 16$).

3 - Esecuzione subroutine

```
R: PUSH {FP, LR}
   MOV FP, SP
```

All'*inizio* della subroutine R bisogna allocare il nuovo frame:

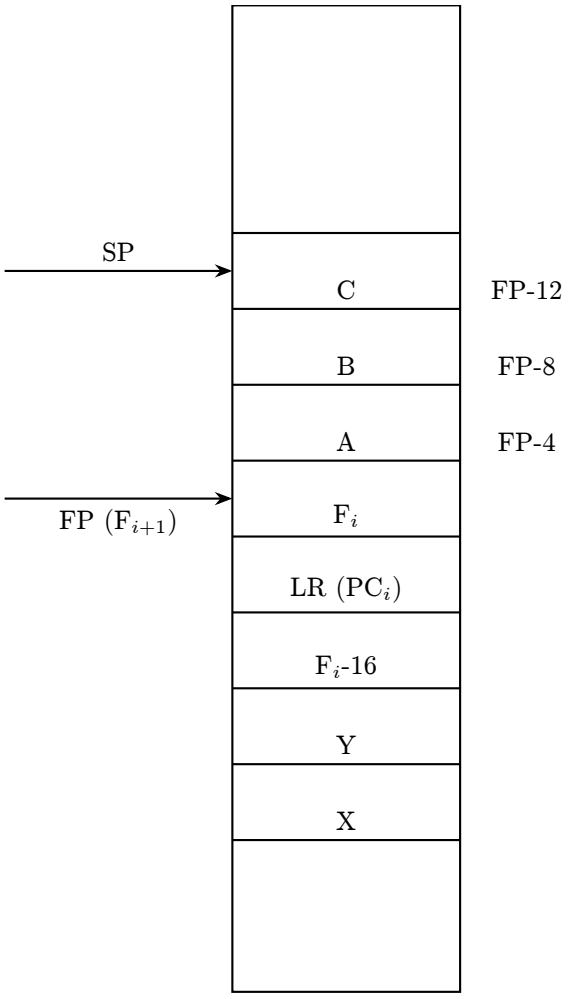


Come prima, si salvano il punto di ritorno e l'indirizzo della base del frame precedente.
NOTA: F_{i+1} è il frame della prima istanza di R.

Ricordiamo che R possedeva 3 parametri locali: *A*, *B* e *C*.

```
SUB SP, SP, #12    @ SP viene spostato di 3 locazioni
```

Situazione dello stack:



I *parametri d'ingresso* della subroutine erano *I*, *J* e **O*, corrispondenti a **X**, **Y**, e ***Z** ($F_i - 16$). Per accedervi, nella subroutine saranno presenti le seguenti istruzioni:

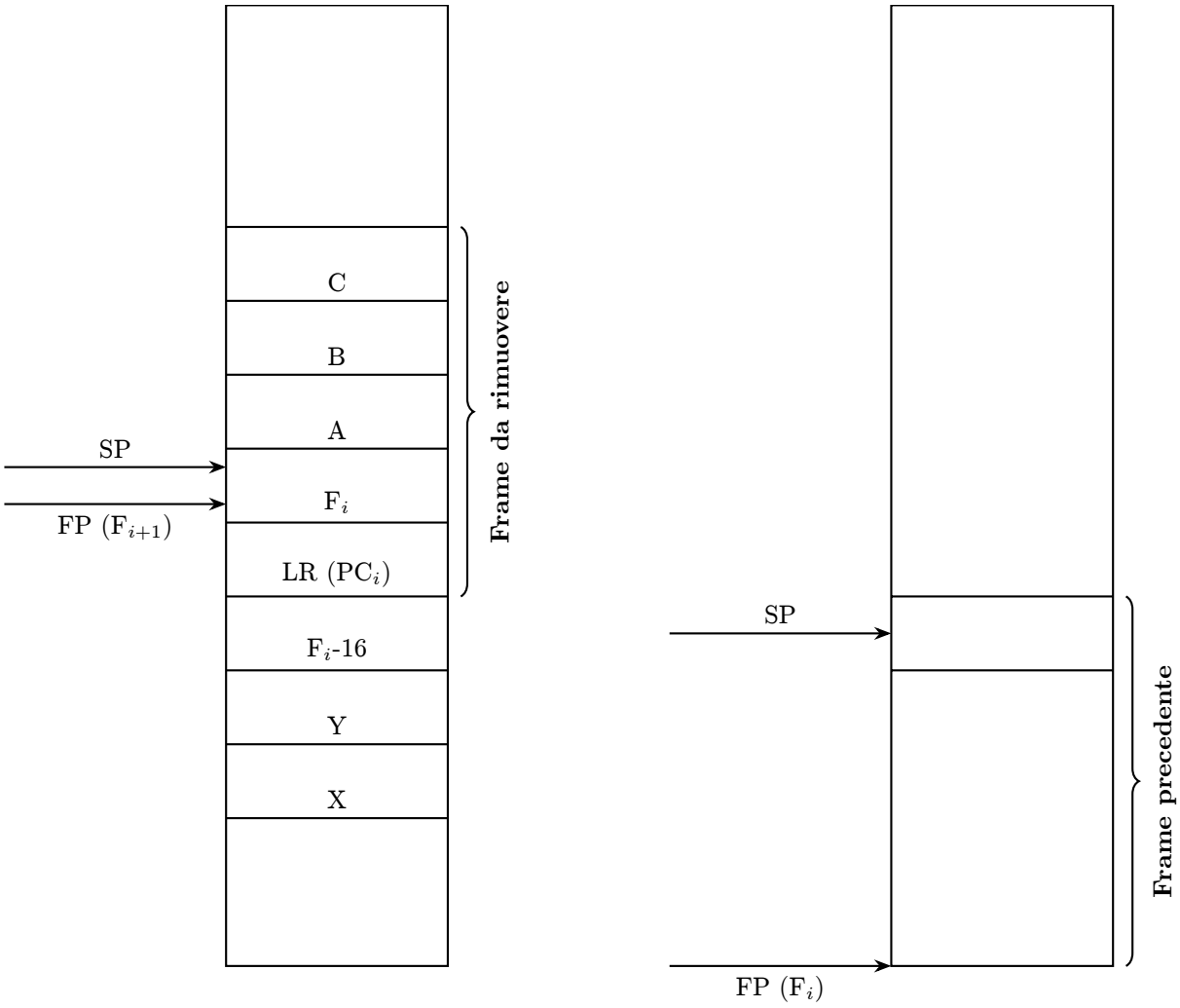
```
LDR R0, [FP, #16]    @ Valore di I, cioè X (a 4 locazioni da FP)
LDR R1, [FP, #12]    @ Valore di J, cioè Y (a 3 locazioni da FP)

ADD R0, R0, R1       @ I+J

LDR R1, [FP, #8]     @ Indirizzo di O, cioè Z (a 2 locazioni da FP)
STR R0, [R1]         @ Salva il risultato all'indirizzo O, cioè Z.
```

4 - Ritorno dalla subroutine

Ora assumiamo che il caso ricorsivo *non* sia verificato e vediamo cosa succede:

```
MOV SP, FP      @ Frame svuotato
POP {FP, PC}    @ Frame rimosso
```

Il frame deve essere *svuotato* e *rimosso*.



Prima lo *Stack Pointer* viene portato al livello del *FP* (prima istruzione), poi vengono ripristinati il *Frame pointer precedente* e il *PC* all'indirizzo di ritorno: ora *SP* punta alla testa del frame precedente.

NOTA: erano anche state allocate 3 locazioni nel frame del main per i parametri d'ingresso di R. Tale spazio deve essere liberato, perciò:

```
ADD SP, SP, #12    @ Rilascia l'area allocata per X, Y, Z
```

5 - Caso chiamata ricorsiva

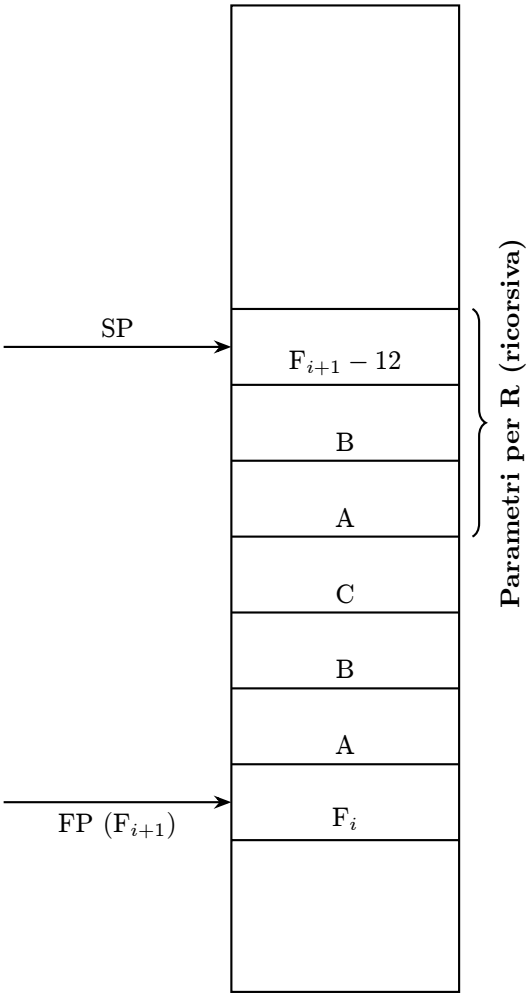
Ora assumiamo invece che il caso ricorsivo sia verificato.
La subroutine verifica la condizione:

```
LDR R0, [FP, #-4]    @ Carica A
CMP R0, #0           @ A==0?
```

Se si, bisogna eseguire **R(A, B, &C)**:
Passiamo *A* e *B* per valore, *C* per *indirizzo*.

```
LDR R0, [FP, #-4]
PUSH {R0}
LDR R0, [FP, #-8]
PUSH {R0}
SUB R0, FP, #12
PUSH {R0}

BL R

ADD SP, SP, #12  @ Come prima, al ritorno rilascia l'area allocata per i parametr
```

Situazione dello stack:



A tal punto verrà allocato un altro Stack Frame F_{i+2} dedicato all'*istanza ricorsiva di R*, che verrà poi liberato in modo analogo a quanto visto sopra.

Quando anche l'esecuzione della prima istanza di R sarà terminata, tutti gli Stack Frame *temporanei* creati per le istanze delle subroutine saranno rimossi e si tornerà a F_i , frame del main.

COMPILAZIONE

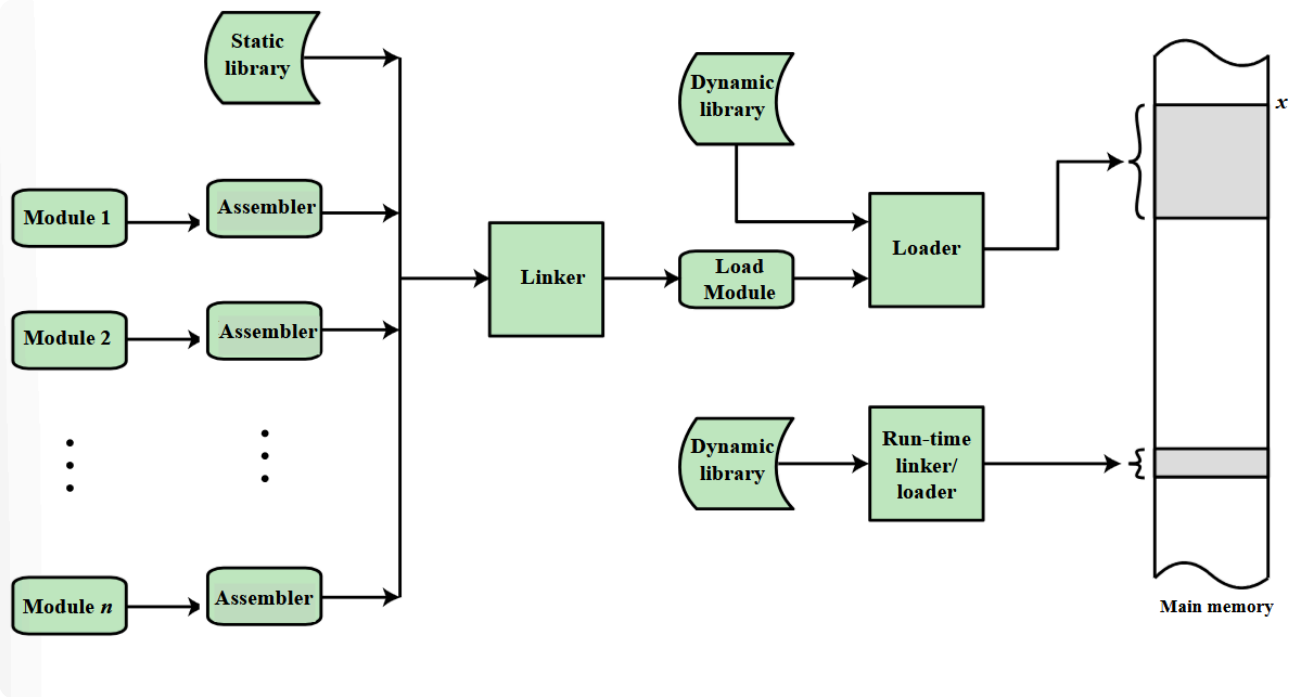
CREAZIONE PROGRAMMA ESEGUIBILE

Un *programma* è generalmente costituito da più file (*moduli*) che vengono poi uniti per formare il software intero.

Il sistema di sviluppo in linguaggio assembly comprende:

1. **TEXT EDITOR**: per la scrittura e modifica del testo sorgente.
2. **ASSEMBLATORE**: per tradurre il testo sorgente in *modulo oggetto*.
3. **LINKER**: per costruire il programma *eseguibile completo*.
4. **LOADER**: per caricare in memoria l'eseguibile.
5. **DEBUGGER**: per eseguire il programma sotto il controllo del programmatore.

La catena di sviluppo può essere così schematizzata:



Assemblatore e *Linker* sono invocati dal comando

```
-gcc
```

Il *Loader* è avviato dal SO quando si esegue il programma.
Il *Debugger*, nel nostro caso, è **gdbgui**.

ASSEMBLER

L'assemblatore traduce ogni modulo in un *modulo oggetto* (estensione **.o**), *segnala* eventuali *errori* e genera i *file di listing* (mostrano come le istruzioni assembly sono state tradotte in

linguaggio macchina).
I moduli sorgente sono costituiti da:

- Istruzioni *macchina*.
- Istruzioni *per l'assemblatore* (direttive, pseudo-istruzioni...).

I moduli contengono anche **simboli**, ovvero stringhe alfanumeriche con un significato definito dal linguaggio (opcode, indirizzamenti...) o dal programmatore (costanti, label..).
In particolare, i simboli definiti dal programmatore possono essere:

- **LOCALI**: visibili solo nel modulo corrente.
- **GLOBALI**: visibili in tutti i moduli.
- **ESTERNI**: definiti in altri moduli.

Inoltre, un simbolo può avere valore:

- *Assoluto*: il suo valore non cambia (come la direttiva .equ).
- *Da rilocare*: il loro valore dipende dalla posizione del codice in memoria (come le label).

Se un simbolo viene *usato prima della sue definizione* si parla di **forward reference**.

FUNZIONAMENTO DELL'ASSEMBLATORE

L'assembler effettua *due scansioni* del file di input:

1. La prima cerca tutte le definizioni di simboli.
2. La seconda converte le istruzioni e risolve l'indirizzo dei simboli.

Le due scansioni sono necessarie a gestire le forward references.
L'assemblatore mantiene il contatore **LC** (*location counter*) che indica a che indirizzo verrà salvata la prossima istruzione:

- Il valore iniziale di LC è 0.
- LC viene opportunamente incrementato a ogni istruzione letta.

PRIMA SCANSIONE

Viene costruita la *tabella dei simboli* che contiene per ogni simbolo definito nel sorgente le seguenti informazioni:

- Nome simbolo.
- Il suo valore.

- Visibilità: locale/globale.

Operazioni svolte:

1. Leggi la prossima riga del file.
2. Se contiene la definizione di un simbolo, inserisci il suo nome, valore (se possibile) e visibilità nella tabella.
3. Incrementa LC del numero di byte necessari per codificare la riga letta.
4. Ripeti dal passo 1.

SECONDA SCANSIONE

Ogni *simbolo* presente nella tabella viene sostituito con il *suo valore* (in caso di forward references, la prima scansione non era sufficiente ad aver inserito tutti i valori), ogni istruzione macchina viene *codificata in linguaggio macchina*, viene *allocato lo spazio per i dati*.

Operazioni svolte:

1. Leggi la prossima riga.
2. Sostituisci ogni simbolo presente nella tabella dei simboli con il suo valore, registra l'utilizzo dell'indirizzo.
3. Codifica l'istruzione.
4. Scrivi l'istruzione nel file oggetto.
5. Scrivi riga sorgente e linea oggetto nel file di listing.
6. Torna al punto 1.

NOTA: siccome le costanti (.equ) erano state registrate nella tabella nella prima scansione, ogni istruzione che le utilizza può essere codificata, senza dover salvare le costanti in memoria.

ESEMPIO SCANSIONI

PRIMA SCANSIONE:

```
LC=0    .TEXT
LC=0    .global _start
LC=0    _start: mov R1, #var_a
LC=0x4  mov R0, #var_b
LC=0x8  add R3, R1, R2
LC=0xC  ldr R0, =ind_c
LC=0x10 str R3, [R0]
LC=0x14 trap:  b trap
LC=0x18 .equ var_a, 5
LC=0x18 .equ var_b, 9
LC=0x18 ind_c: .space 4
LC=0x1C .skip 0x100
LC=0x11C stack: .word 4
```

Simbolo	Valore	Locale/ globale
_start		G
trap	0x14	L
var_a	5	L
var_b	9	L
ind_c	0x18	L
stack	0x11C	L

SECONDA SCANSIONE:

```
LC=0    .TEXT
LC=0    .global _start
LC=0    _start: mov R1, #var_a   e3a01005
LC=0x4  mov R0, #var_b           e3a00009
LC=0x8  add R3, R1, R2           e0813002
LC=0xC  ldr R0, =ind_c           e59f0008
LC=0x10 str R3, [R0]             e5803000
LC=0x14 trap:  b trap            ea000005
LC=0x18 .equ var_a, 5
LC=0x18 .equ var_b, 9
LC=0x18 ind_c: .space 4          00000000
LC=0x1C .skip 0x100              0.....0
LC=0x11C stack: .word 4          00000000
                                 00000018
```

Aggiornamento tabella simboli nella seconda scansione:

Simbolo	Valore	Locale/ globale	Indirizzi uso
_start	0x0	G	-
trap	0x14	L	0x14
var_a	5	L	0x0
var_b	9	L	0x4
ind_c	0x18	L	0x120
stack	0x11C	L	-

Alcuni simboli potrebbero essere stati definiti in altri sorgenti, per cui nella seconda scansione l'assemblatore crea una *tabella dei simboli esterni* in cui elenca tutti i *simboli con valore mancante* e gli indirizzi nel modulo oggetto in cui il valore mancante dovrà essere inserito (dal linker nella fase successiva).

```
        .text
        .global  _start
_start:
        ldr   sp, =stack
        ldr   r8, =in1
        ldr   r0, [r8]
        ldr   r8, =in2
        ldr   r1, [r8]
        ldr   r2, =out
        bl addf
...
```

Simbolo esterno	Lista di indirizzi in cui il simbolo è usato
addf	0x18

ASSEMBLAGGIO DI FILE CON PIU' SEGMENTI

L'assemblatore lavora *indipendentemente* su ciascun segmento (text, data, bss):

- Ogni segmento è costruito in un proprio spazio di indirizzamento (LC parte da 0).
- I *segmenti* vengono *assemblati indipendentemente*, ma verranno ricomposti dal linker in un unico spazio di indirizzamento.
- Un simbolo usato in .text ma definito in .bss/.data viene gestito come *esterno*.
- Viene comunque generato *solo un file oggetto*.

Alla file il modulo oggetto contiene:

- La *traduzione* di ogni segmento.
- La *tabella dei simboli*, con indicazione se sono locali o globali.
- La *tabella dei simboli esterni*, con la lista degli indirizzi da aggiornare nel modulo oggetto.

LINKER

Il *linker* unisce tutti i moduli oggetto in un *unico file eseguibile* contenente:

- Il programma in linguaggio macchina.
- Informazioni di supporto (ad esempio l'indirizzo di partenza per l'esecuzione del file).

Genera i segmenti **TEXT, DATA, BSS** *complessivi* del modulo eseguibile, collocando i rispettivi segmenti di ciascun modulo.

Il linker esegue quindi le seguenti operazioni:

1. Calcola l'*estensione di memoria* occupata da ciascun segmento di ciascun modulo.
2. Posiziona i segmenti in memoria e calcola il nuovo indirizzo iniziale.
3. Ogni indirizzo di memoria definito da una label viene riallocato.
4. Tutti i riferimenti esterni vengono finalmente risolti.

ESEMPIO

Assemblaggio di due file: *main.s* e *addf.s*.

Vediamo prima **main.s**.
Immaginiamo che il contenuto di *main.s* sia:

```
.text
.global _start
-start:
    LDR SP, =stack
    LDR R8, =in1
    LDR R0, [R8]
```

```
    LDR R8, =in2
    LDR R1, [R8]
    LDR R2, =out
    BL addf
main_end: b main_end

.data
in1: .word 0x00000012
in2: .word 0x00000034

.bss
out:   .space 4
       .space 256
stack: .space 4
```

Generiamo il modulo oggetto:

```
arm-linux-gnueabi-as -o main.o main.s
```

Per il segmento *.text* di *main.s* l'assemblatore ha prodotto:

```
00000000 <_start>:
 0: e59fd018 ldr sp, [pc, #28] ; <.text+0x20>
 4: e59f8018 ldr r8, [pc, #28] ; <.text+0x24>
 8: e5980000 ldr r0, [r8]
 c: e59f8014 ldr r8, [pc, #24] ; <.text+0x28>
10: e5981000 ldr r1, [r8]
14: e59f2010 ldr r2, [pc, #20] ; <.text+0x2c>
18: eb?????? bl ???????? <addf>
0000001c <main_end>:
1c: ea000005 b 1c <main_end>
20: ????????
24: ????????
28: ????????
2c: ????????
```

Per il segmento *.data* l'assemblatore ha prodotto:

```
00000000 <in1>:
0: 00000012 …
00000004 <in2>:
4: 00000034
```

E per il segmento *.bss* ha prodotto:

```
00000000 <out>:
...
00000104 <stack>:
104: 00000000
```

Vediamo ora **addf.s**.
Immaginiamo che il contenuto di *addf.s* sia:

```
.text
.global addf
addf:
 PUSH {R0}
 ADD R0, R0, R1
 STR R0, [R2]
 POP {R0}
 MOV PC, LR
```

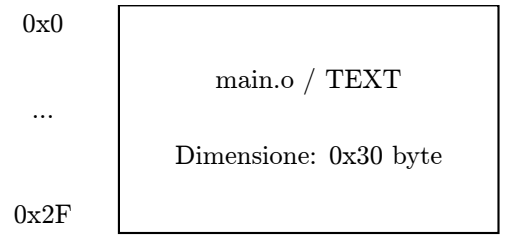
Generiamo il modulo oggetto:

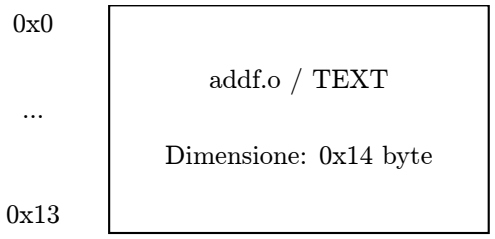
```
arm-linux-gnueabi-as -o addf.o addf.s
```

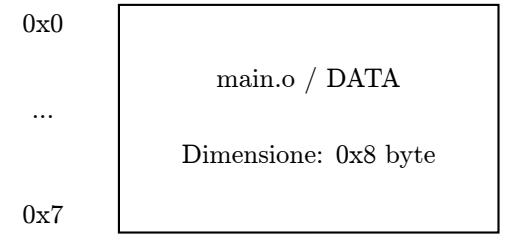
L'assemblatore avrà generato:

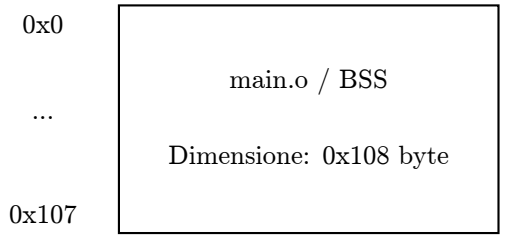
```
00000000 <addf>:
0: e92d0001 stmdb sp!, {r0}
4: e0800001 add r0, r0, r1
8: e5820000 str r0, [r2]
c: e8bd0001 ldmia sp!, {r0}
10: e1a0f00e mov pc, lr
```

La situazione dei segmenti è la seguente:

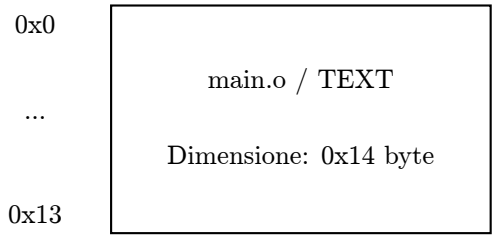


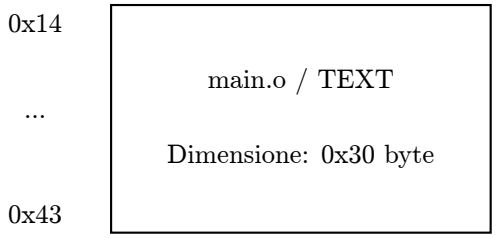


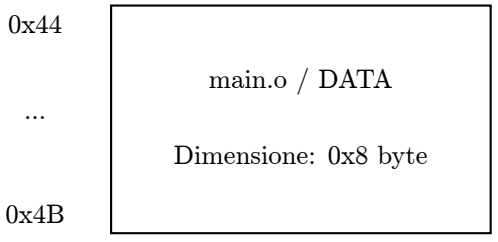


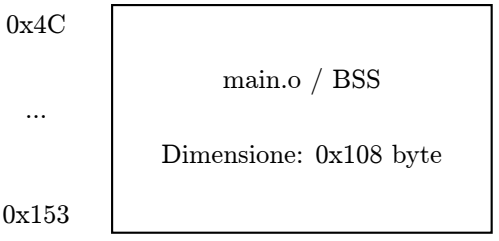


Il linker li unirà nel seguente modo:









A questo punto, gli indirizzi definiti tramite *label* in memoria devono essere aggiornati: per ogni label **X** con valore **V** all'interno di un segmento con *indirizzo iniziale* **ADR**, il linker sostituisce tutte le occorrenze di **X** con il valore **V+ADR**. In pratica le label sono shiftate di un offset pari all'indirizzo iniziale del loro segmento.

NOTA: *non* vengono riallocati:

- Indirizzi/valori assoluti definiti da costanti (.EQU non produce indirizzi).
- Indirizzi definiti da offset (autorelativi, frame pointer).

Inoltre, avendo ora a disposizione *tutte* le *tabelle dei simboli esterni* (e anche aggiornate con i nuovi indirizzi), è possibile *risolvere* ogni *riferimento a simboli esterni*.

LOADER

Il programma eseguibile contiene ora tutte le informazioni necessarie per l'esecuzione:

- *Istruzioni macchina* in text.
- *Dati* in data/bss.
- Punto di *inizio* (_start).
- *Dimensioni* segmenti.
- *Tabelle* dei simboli.
- Informazioni di *debug*.

Per eseguire il programma, viene prima invocato il **LOADER**:

- Carica in memoria le istruzioni macchina e i dati agli *indirizzi indicati nell'eseguibile*.
- Carica nel registro **PC** l'indirizzo di *_start*.
- In certi casi, il loader può riallocare il programma in nuovi indirizzi in modo analogo alla procedura seguita dal linker.

LIBRERIE

Le *librerie software* sono collezioni di moduli oggetto che contengono *subroutine* che possono essere invocate da altri programmi.
Ne esistono di due tipi:

- Librerie **statiche**.
  Il loro codice è incluso al momento dell'assemblaggio e linking (es. addf.s nel nostro caso).
  Ad ogni modifica della libreria, è necessario ricompilare il programma che la usa.

- Librerie **dinamiche**.
  Il loro codice è incluso in un momento *successivo al linking*.
  Il linker *non risolve* i simboli della libreria dinamica. Questi sono risolti:
  - Dal **loader** quando il programma viene caricato in memoria (**load-time** dynamic library).
  - Dal sistema operativo quando il programma esegue l'istruzione con il simbolo mancante (**run-time** dynamic library).