Allocazione dinamica

Argomenti:

- Subroutine rientranti
- Allocazione dinamica della memoria
- Stack-frame

La rientranza

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

- in seguito a una **chiamata ricorsiva** (diretta/indiretta)
- in seguito a una **interruzione** (il processore passa ad eseguire un altro programma che potrebbe chiamare la medesima subroutine interrotta),
- in un **sistema multiprocessore** (se la subroutine si trova in memoria condivisa, più di un processore può iniziarne la esecuzione).

La rientranza (2)

- Per essere **rientrante** una subroutine non deve alterare i dati su cui stava operando ogni sua attivazione precedente non terminata
- Quando questa proseguirà la sua esecuzione, deve trovare gli stessi valori che vi aveva lasciato.
- I dati **non fanno parte della subroutine**, ma **dell'istanza di esecuzione**: **la subroutine è fatta di solo codice (pure code).**

Subroutine rientranti

Se, ad ogni attivazione di una subroutine, i dati su cui essa opera sono allocati in un **luogo diverso** (**ad es. nello stack**), allora **la subroutine è rientrante**, cioè ne può iniziare una nuova esecuzione mentre è ancora in corso una sua esecuzione precedente.

I dati che vanno allocati ogni volta in un posto diverso sono:

- i **parametri di ingresso e di uscita** passati alla subroutine (anche l'indirizzo di ritorno è un parametro di ingresso),
- i **dati locali** su cui la subroutine opera.

Subroutine rientranti (2)

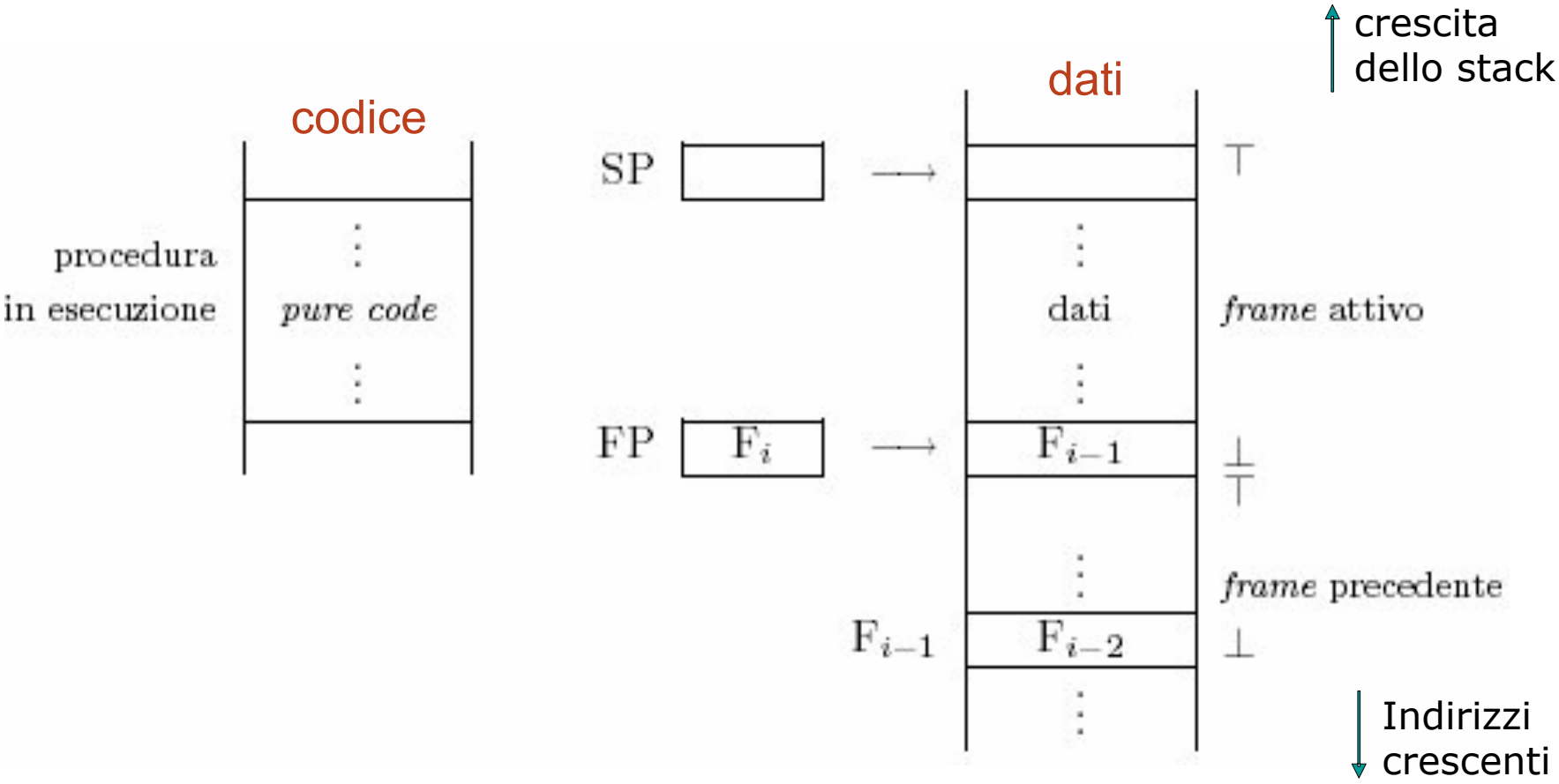
- La soluzione comunemente adottata prevede che, al momento di attivare una subroutine, venga allocata in cima allo stack un'area (**stack-frame**) in cui si trovano tutti i dati (**parametri e dati locali**) sui quali la subroutine stessa opererà; questa area verrà poi rimossa dallo stack, quando la subroutine termina.

- Questa soluzione si chiama **allocazione dinamica della memoria**.

Allocazione dinamica della memoria

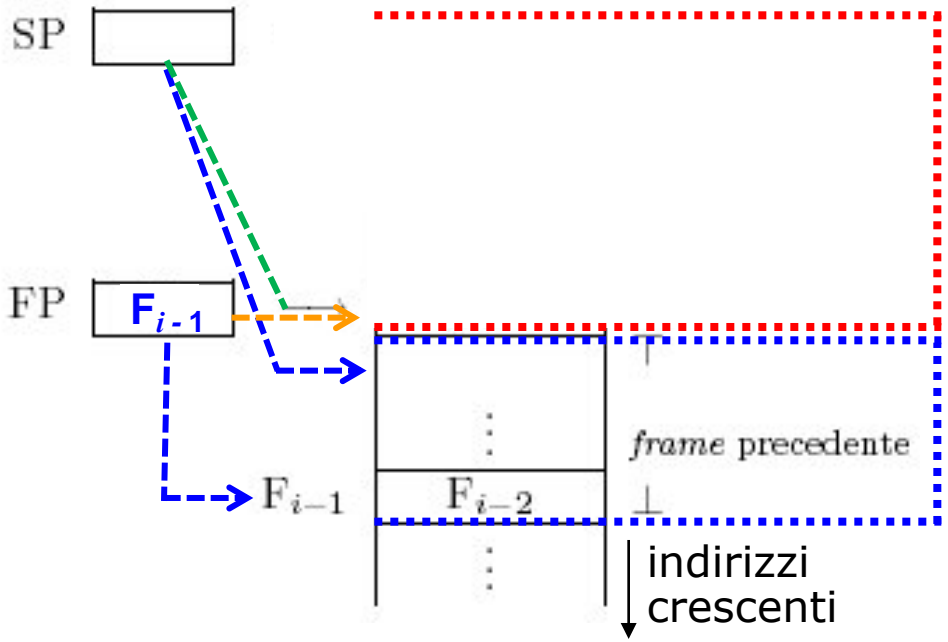
L'indirizzo dello stack-frame attivo è contenuto nello **Frame-Pointer** (FP)

- Il FP contiene la base dello stack-frame, ovvero l'indirizzo più grande dell'area di memoria in uno stack descending.



Allocazione e rilascio di un frame

- **Allocazione di un nuovo frame:**
  - **FP → *push***
  - **SP → FP**
  - **SP-e → SP**



- **Rilascio dell'area di memoria allocata:**
  FP → SP
  ***pop* → FP**

Esempio di subroutine ricorsiva (HLL)

```
void R(int I, int J, int *O) {
    int A, B, C;
        ⋮
    *O = I+J;
        ⋮
    if (A=0) R(A, B, &C);
        ⋮
}

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

par. di uscita (per indirizzo)

par. di ingresso (per valore)

dati locali

chiamata ricorsiva

Come si comporta un compilatore?

main(): espansione in assembly per l'ARM

- **main()** non ha parametri, ma solo dati locali.
- Dopo la dichiarazione di una subroutine in assembly, troviamo le istruzioni che allocano un nuovo frame in cima allo stack

```
@ int main(){ inizio del main

main:                       @ stack: Fig. A
PUSH {FP, LR}               @ FP, LR → push
MOV   FP, SP                @ SP → FP
                            @ stack: Fig. B
```

main(): espansione in assembly per l'ARM

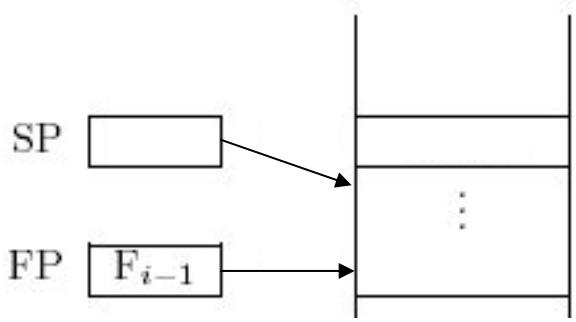


Fig. A

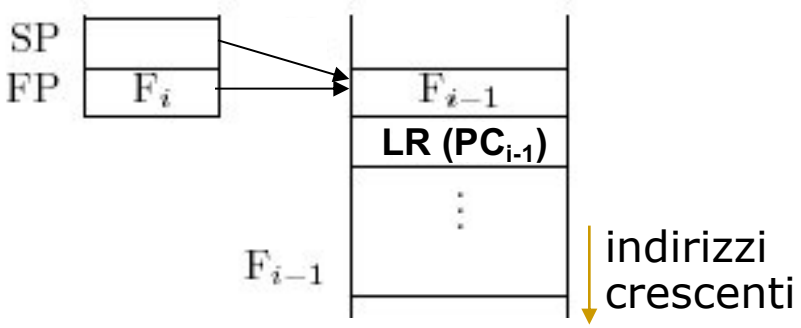


Fig. B

main(): dati locali nel frame

```
@ int  W, X, Y, Z; allocazione delle variabili locali

                        @ stack: Fig. B
  SUB SP, SP, #16       @ 4 interi di 4 byte = 16 byte
                        @ W = FP-4
                        @ X = FP-8
                        @ Y = FP-12
                        @ Z = FP-16
                        @ stack: Fig. C
```

main(): dati locali nel frame

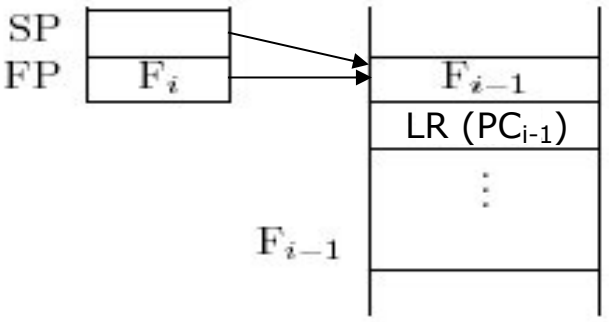


Fig. B

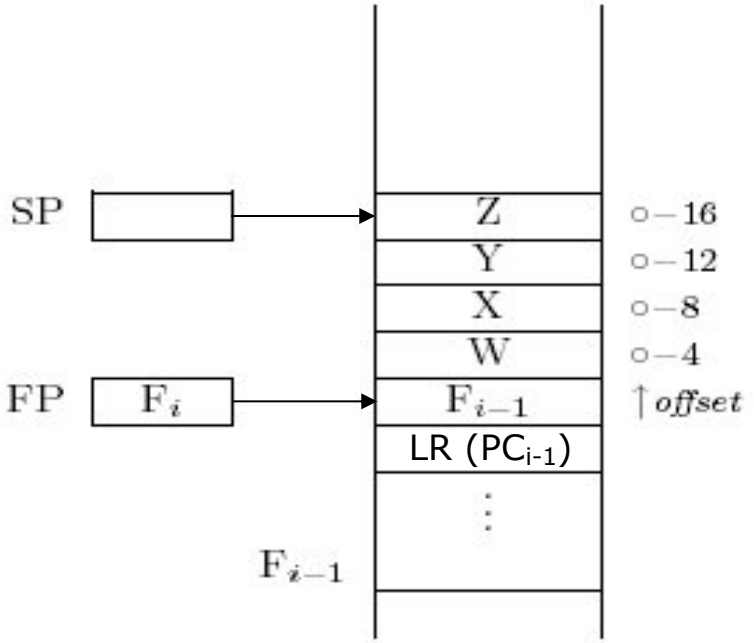


Fig. C

main(): chiamata alla subroutine R(…)

```
@ R(X, Y, &Z) chiamata ad R

                          @ stack: Fig. C
                          @ inserzione parametri nello stack
LDR   R0, [FP, #-8]       @ X
PUSH {R0}                 @ →push
LDR   R0, [FP, #-12]      @ Y
PUSH   {R0}               @ →push
SUB   R0, FP ,#16         @ indirizzo di Z
PUSH   {R0}               @ →push
                          @ chiamata alla subroutine
                          @ stack: Fig. D
BL    R                   @ PC→LR, R→PC
                          @ stack: Fig. D
ADD   SP, SP, #12         @ rilascia l'area allocata per
                          @ i parametri
                          @ stack: Fig. C
```

main(): chiamata alla subroutine R(…)

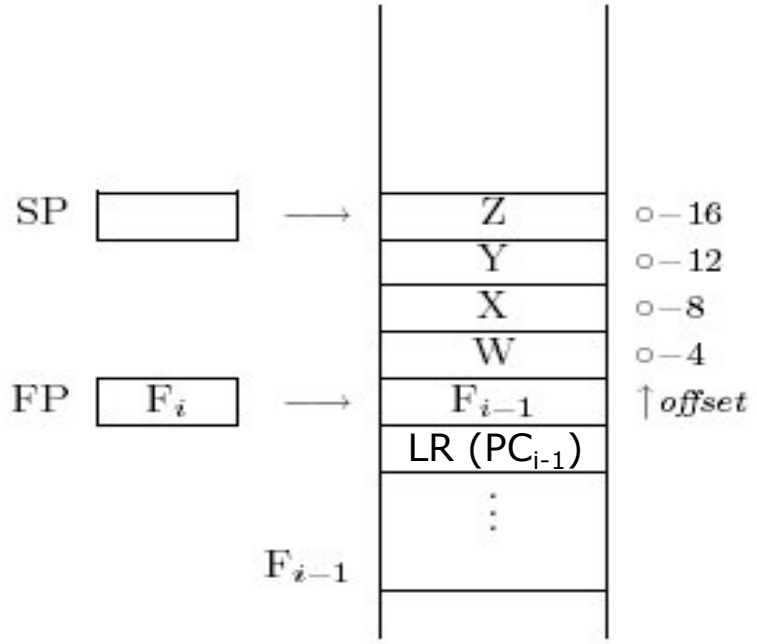


Fig. C

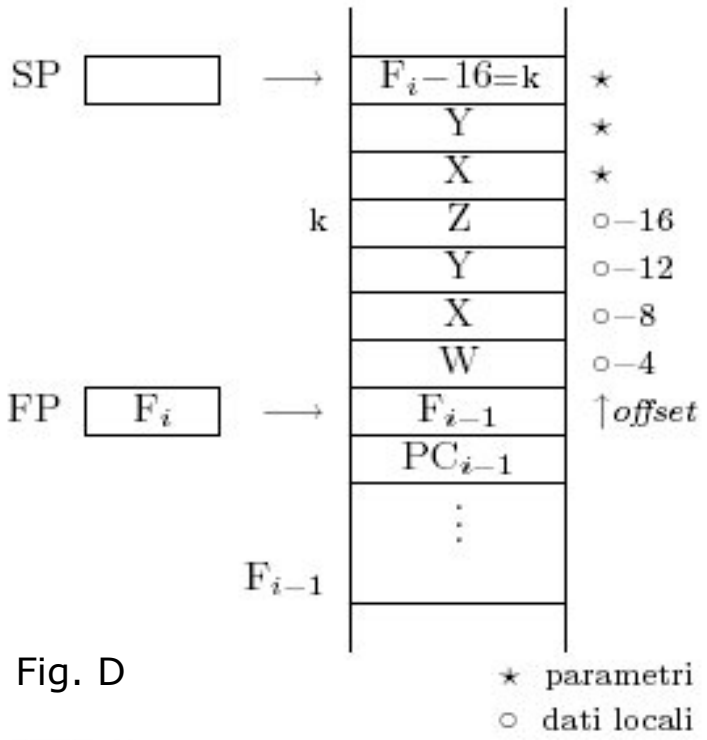


Fig. D

main(): rimozione del frame

```
@ } termine del main

                                @ stack: Fig. C
    MOV    SP, FP               @ FP→SP
                                @ stack: Fig. B
    POP     {FP, PC}            @ ripristina FP precedente
                                @ LR salvato→PC (ritorna
                                @ al programma chiamante)
                                @ stack: Fig. A
```

main(): rimozione del frame

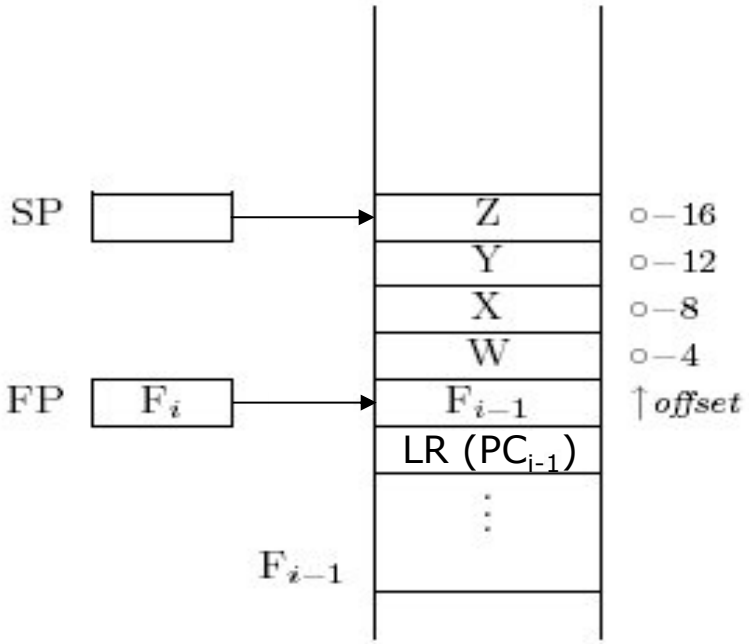


Fig. C

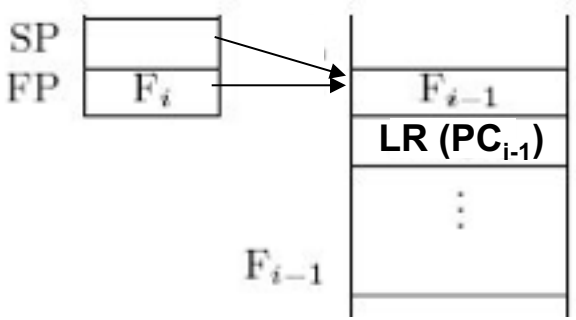


Fig. B

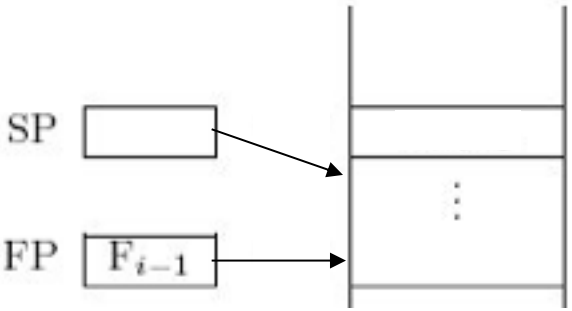


Fig. A

R(…): allocazione del nuovo frame

La subroutine R(…) ha parametri e dati locali.

```
@ void R(int I, int J, int *O) {

                                    @ stack: Fig. D
R:    PUSH   {FP, LR}               @ FP, LR → push
      MOV    FP, SP                 @ SP → FP
                                    @ I = FP+16
                                    @ J = FP+12
                                    @ → O = FP+8
                                    @ stack: Fig. E
```

R(…): allocazione del nuovo frame

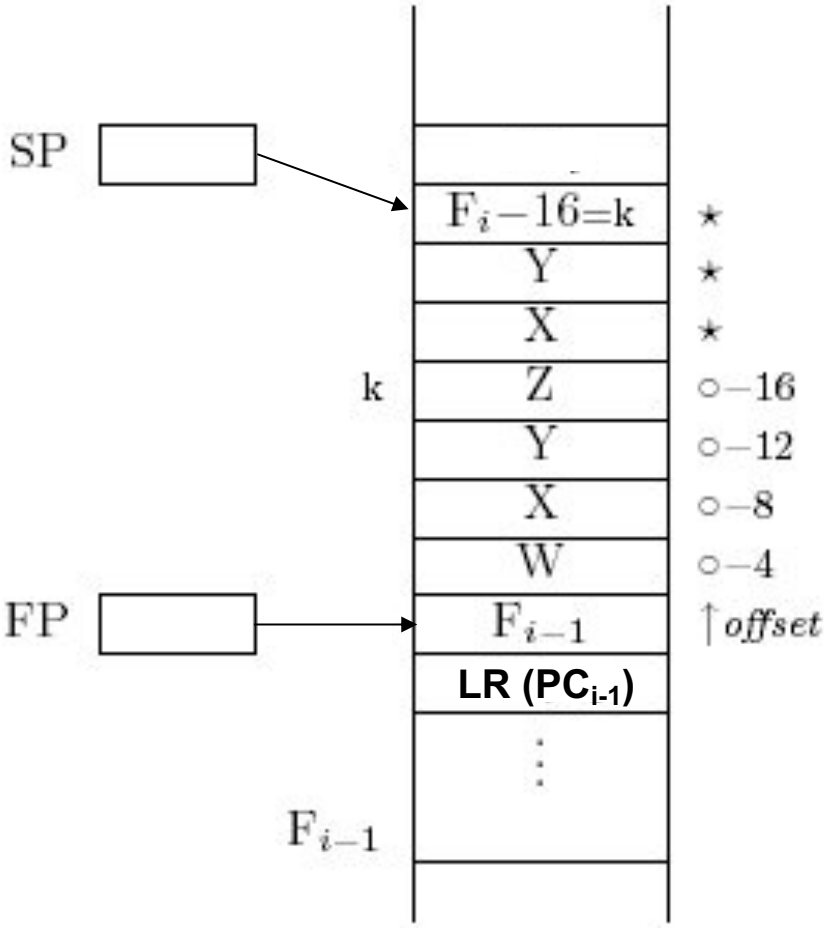


Fig. D

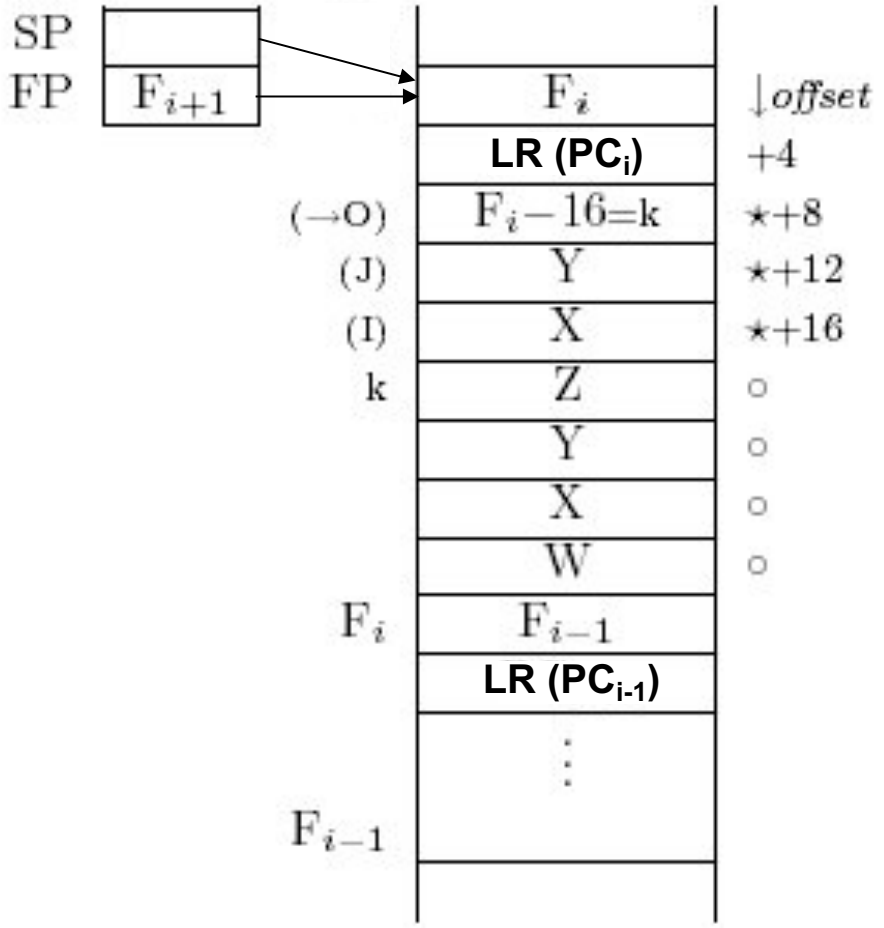


Fig. E

R(…): dati locali nel frame

```
@ int A, B, C;
                        @ stack: Fig. E
  SUB SP, SP, #12       @ 3 interi di
                        @ 4 byte = 12 byte

                        @ A = FP-4
                        @ B = FP-8
                        @ C = FP-12
                        @ stack: Fig. F
```

R(…): dati locali nel frame

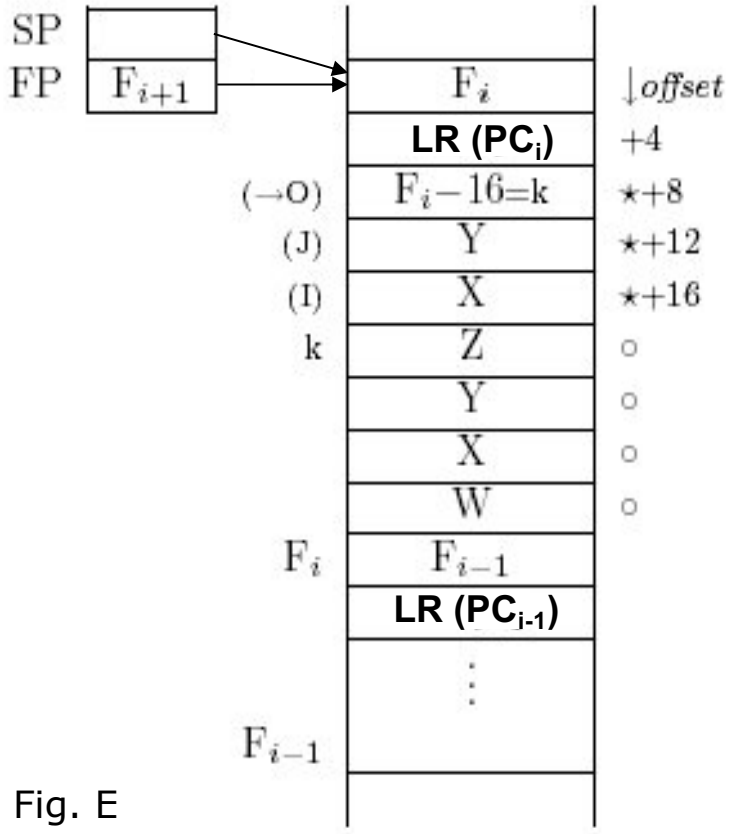


Fig. E

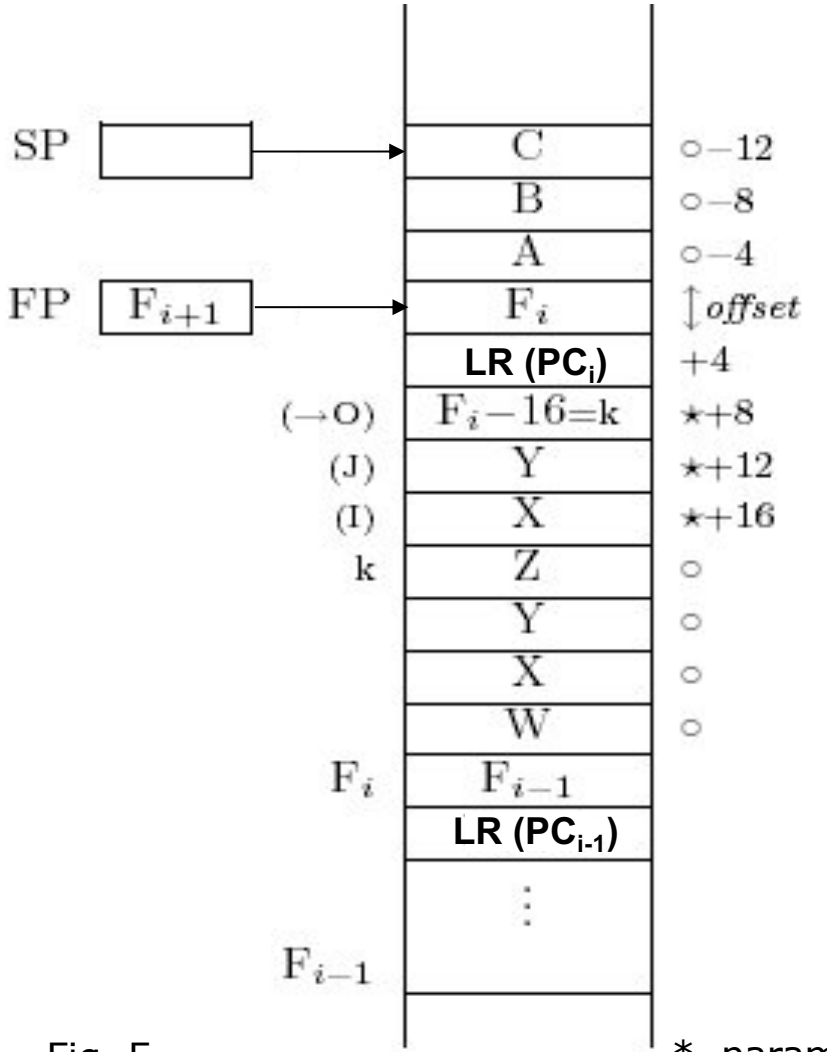


Fig. F

* parametri

o dati locali

R(…): accesso ai parametri

```
@ *O = I+J;
                              @ stack: Fig. F
  LDR R0, [FP, #16]           @ I
  LDR R1, [FP, #12]           @ J
  ADD R0, R0, R1              @ I+J
  LDR R1, [FP, #8]            @ →O (indirizzo di Z)
  STR R0, [R1]                @ risultato
                              @ in →O, cioè in Z
                              @ stack: Fig. F
```

R(…): chiamata ricorsiva

```
@ if (A==0)
    LDR R0, [FP, #-4]     @ il valore di A
    CMP R0, #0            @ = 0 ?
    BNE END_IF            @ Salta chiamata ricorsiva

@ R(A, B, &C);            @ stack: Fig. F
    LDR   R0, [FP, #-4]   @ A
    PUSH {R0}             @ →push
    LDR   R0, [FP, #-8]   @ B
    PUSH {R0}             @ →push
    SUB   R0, FP, #12     @ indirizzo di C
    PUSH {R0}             @ →push
                          @ stack: Fig. G
    BL R                  @ PC→LR, R→PC
                          @ stack: Fig. G
    ADD SP, SP, #12       @ rilascia l'area allocata
                          @ per parametri
  END_IF:                 @ stack: Fig. F
```

Situazione dello stack

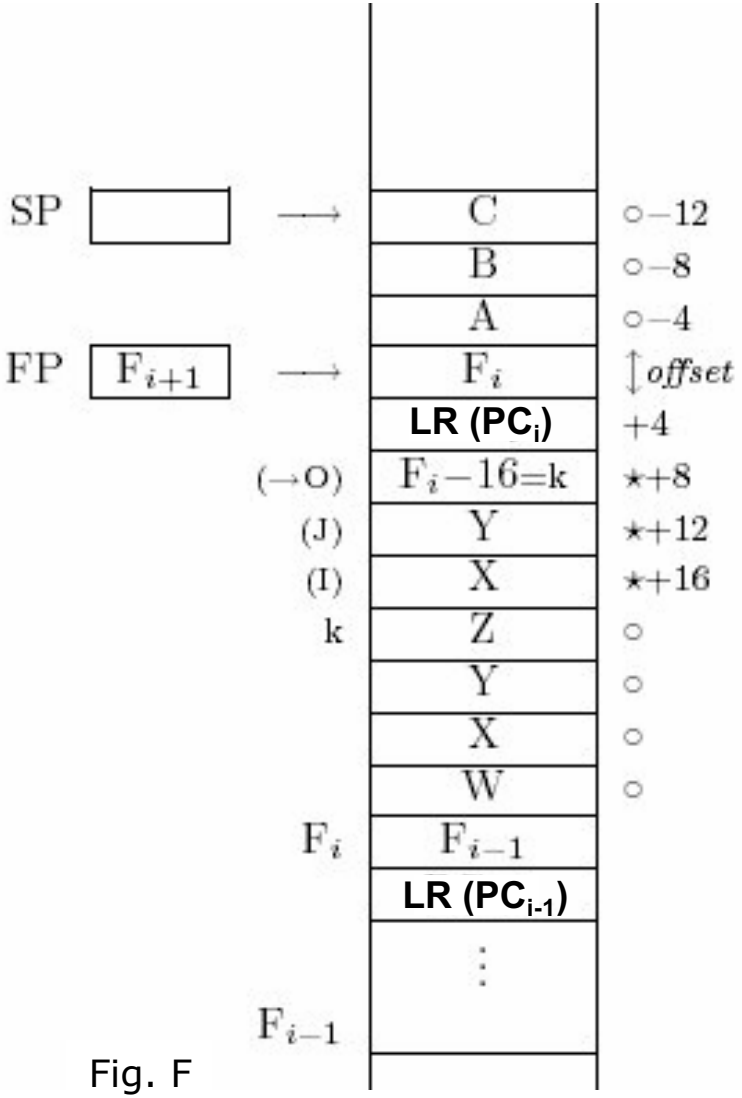


Fig. F

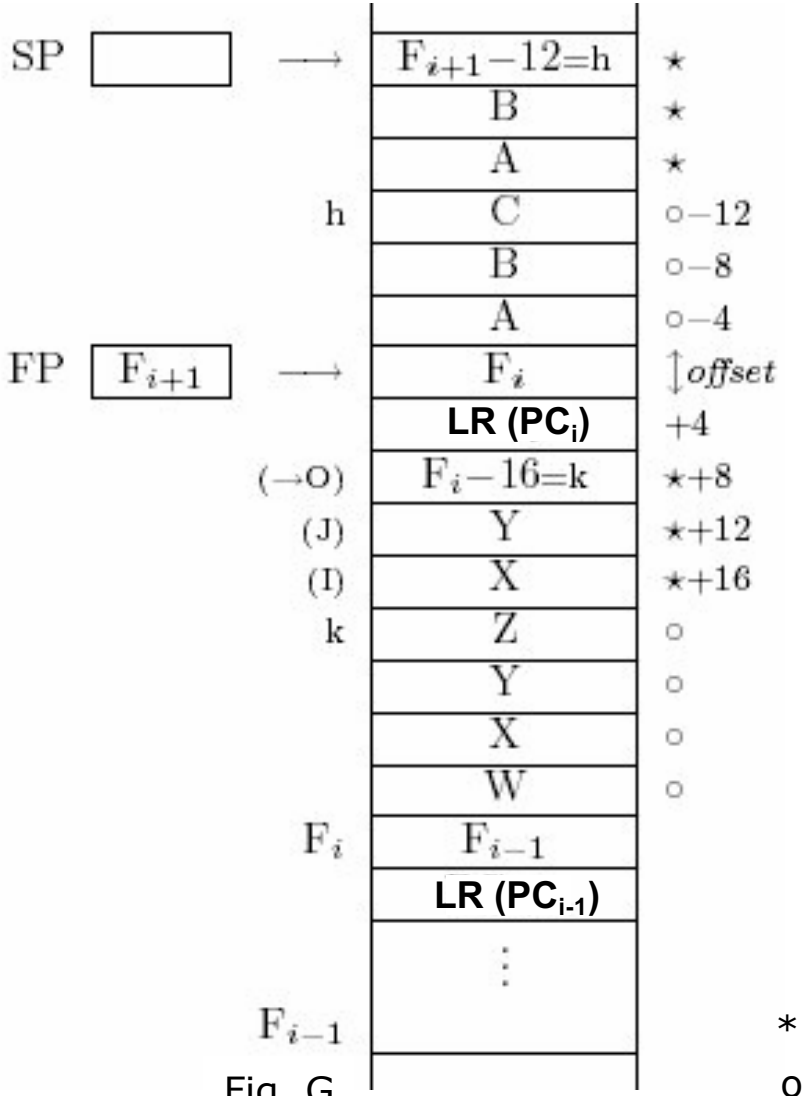


Fig. G

* parametri

o dati locali

R(…): rimozione del frame

```
@ }
                          @ stack: Fig. F
   MOV    SP, FP          @ FP→SP
                          @ stack: Fig. E
   POP    {FP, PC}        @ ripristina FP precedente +
                          @ LR salvato→PC (ritorna
                          @ al programma chiamante)
                          @ stack: Fig. D
```

R(…): rimozione del frame

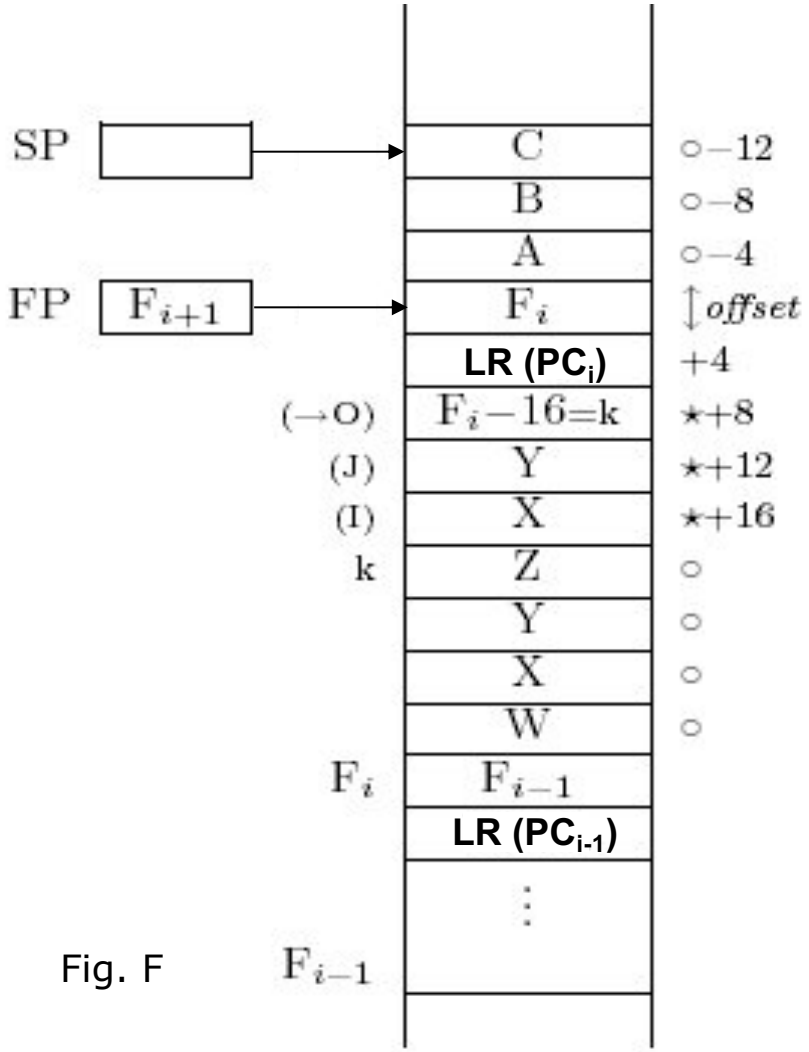


Fig. F

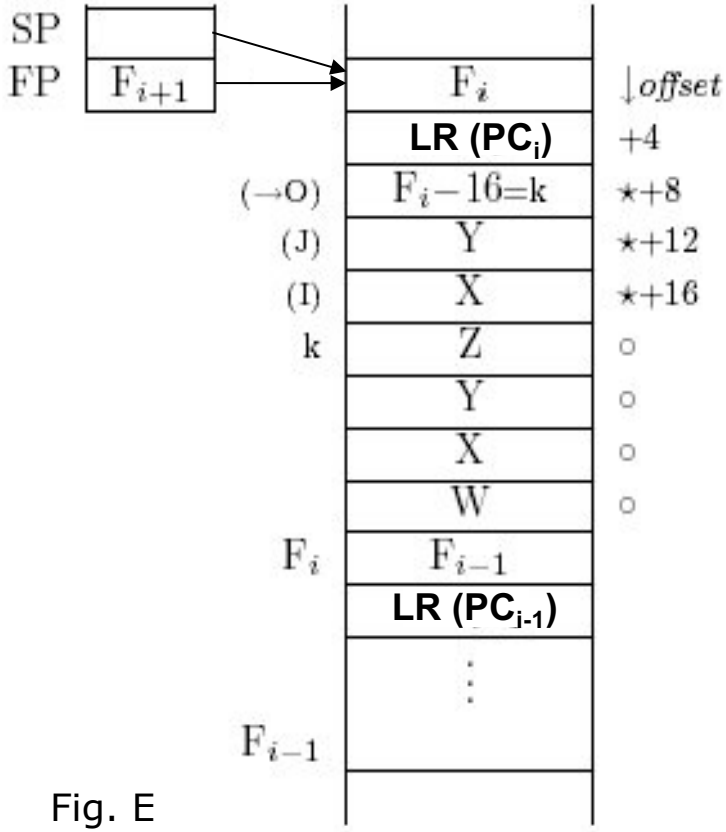


Fig. E

R(…): rimozione del frame

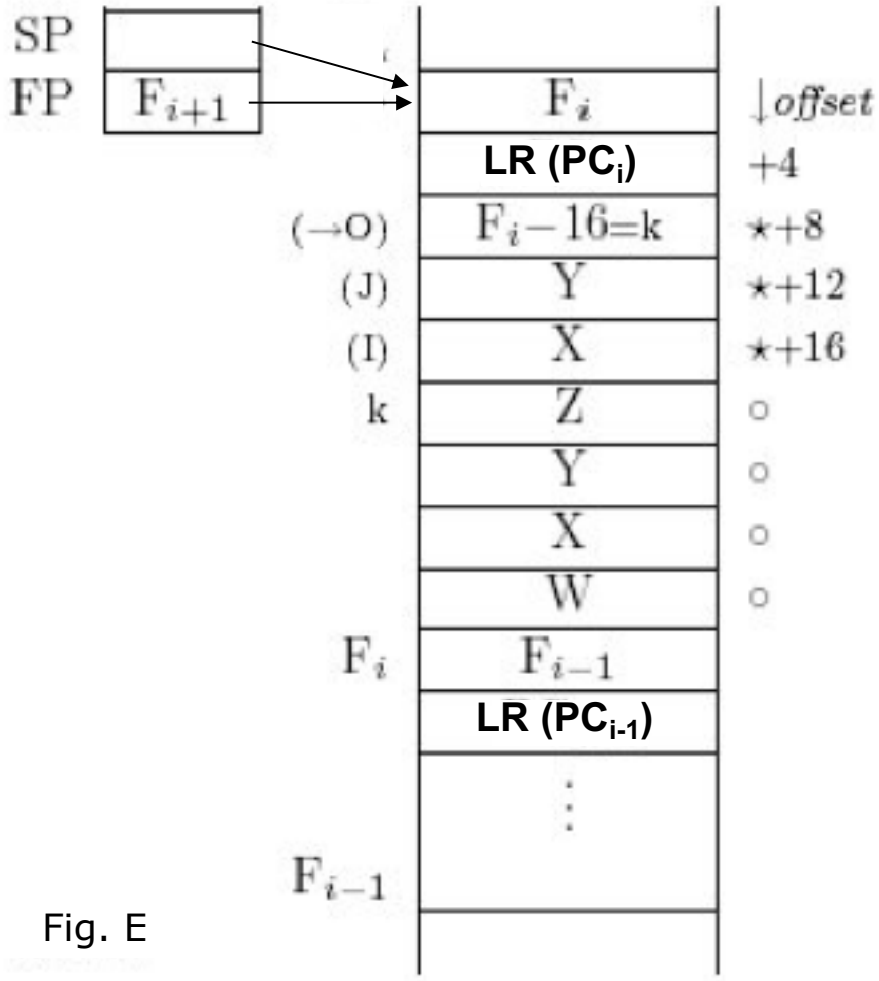


Fig. E

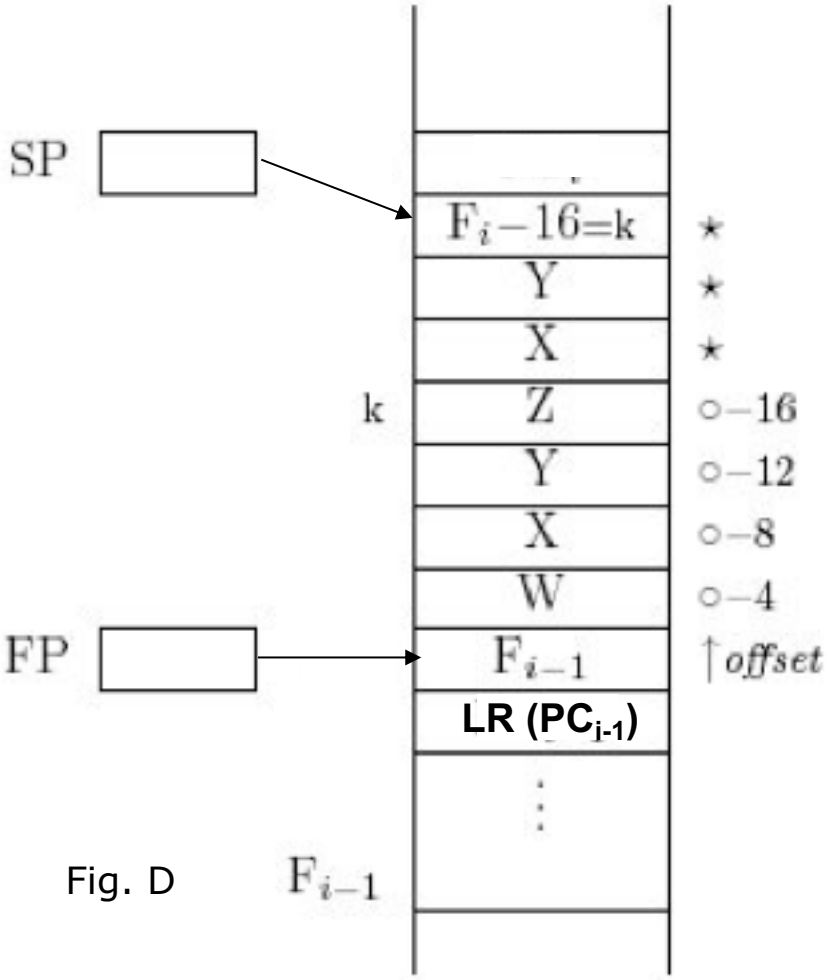


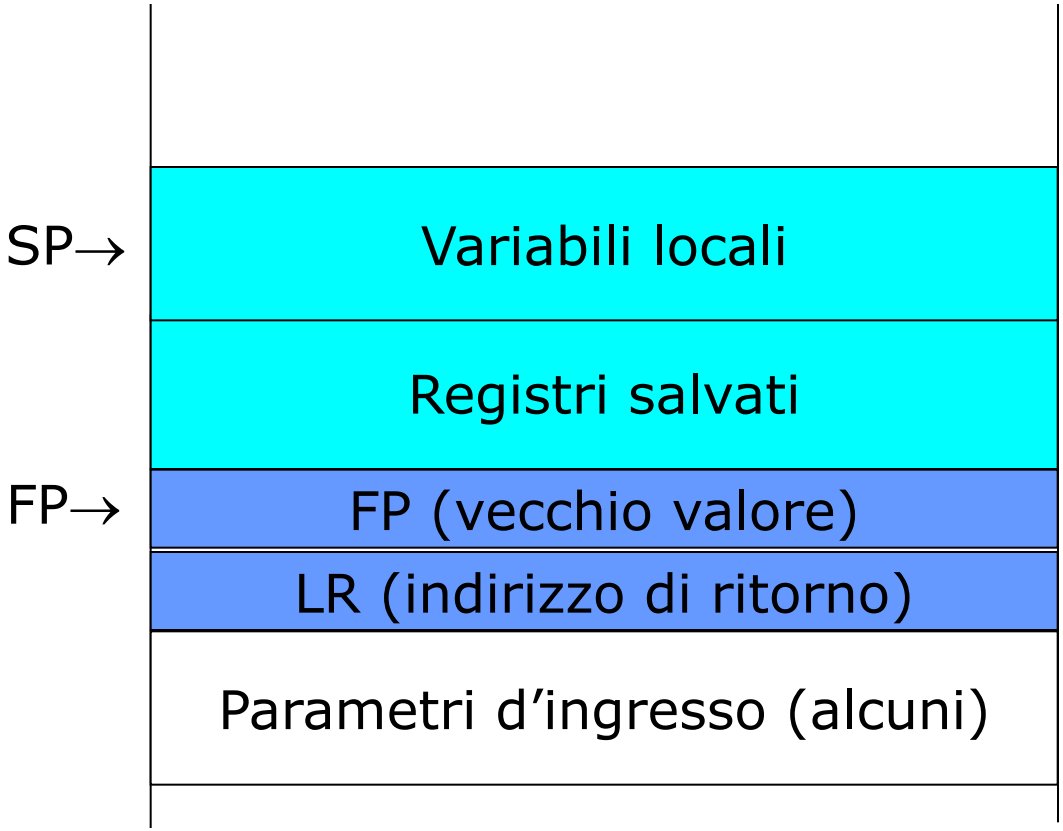
Fig. D

Struttura stack frame

Adottiamo la seguente **convenzione** per lo stack frame (basata su GCC)**:**

1. Eventuali parametri di ingresso prima della creazione del nuovo frame
2. Vengono salvati i vecchi valori di LR e FP con l'istruzione PUSH {LR, FP}
3. Il nuovo frame inizia nella word che contiene il vecchio frame con l'istruzione MOV FP, SP
4. Al di sopra: salvataggio di **altri registri**;
5. Al di sopra: **variabili locali.**

Struttura stack frame



Passaggio dei parametri

- Se i parametri sono pochi, conviene inserirli nei registri, dopodiché è responsabilità della subroutine salvarli nello stack, se necessario (per poterli ripristinare o per essere rientrante).

- Se i parametri sono molti, se ne possono passare alcuni nei registri, altri nello stack.

- Approccio basato su compilatore GCC