

Le procedure

- Porzioni di codice associate ad un **nome** che possono essere invocate più volte e che eseguono un compito specifico, avendo come input una lista di **parametri** e come output un **valore di ritorno**

- Vantaggi
 - Astrazione
 - Riusabilità del codice (library)
 - Maggiore organizzazione del codice
 - Testing più agevole

```
int somma (int x, int y)
{
    int rst;
    rst = x + y;
    return rst;
}
```

Procedure: un esempio

Programma (procedura) chiamante

```
...  
f=f+1;  
risultato = somma (f, g);  
...  
int somma (int x, int y)  
{  
    int rst;  
    rst = x + y + 2;  
    return rst;  
}
```

Procedura chiamata

Le procedure: passi da seguire

- Chiamante
 - Mettere i parametri in un luogo accessibile alla procedura
 - Trasferire il controllo alla procedura
- Chiamato
 - Acquisire le risorse necessarie per l'esecuzione della procedura
 - Eseguire il compito richiesto
 - Mettere il risultato in un luogo accessibile al programma chiamante
 - Restituire il controllo al punto di origine (la stessa procedura può essere chiamata in differenti punti di un programma)

Modifica del flusso di programma: invocazione

jal IndirizzoProcedura

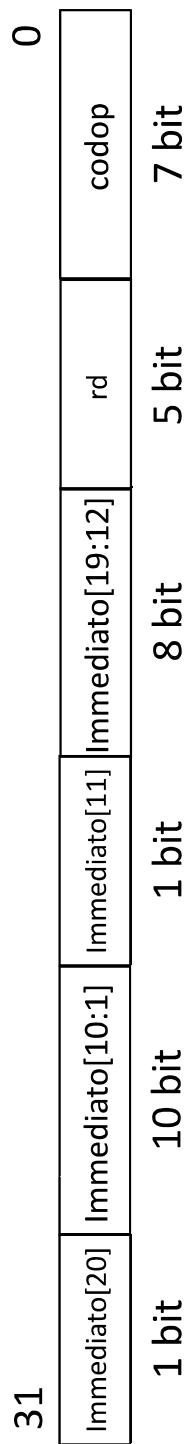
Jump-and-link

- Salta all'indirizzo (offset) con etichetta IndirizzoProcedura
- Memorizza il valore dell'istruzione successiva PC+4 nel registro x1 (return address, ra)
- Pseudo-istruzione, abbreviazione di
 - jal x1, IndirizzoProcedura

vedi
let imm

JAL e linguaggio macchina

- Viene introdotto un nuovo tipo: J



Modifica del flusso di programma: ritorno al chiamante



Jump-and-link register

```
jalr rd, offset(rs1)
```

- Salta ad un indirizzo qualsiasi

- $x[rd] = PC + 4 ; \quad PC = x[rs1] + sext(offset) \& \sim 1$

Segno esteso a 64bit
Bit 0 azzerato

- La procedura chiamata come ultima istruzione esegue

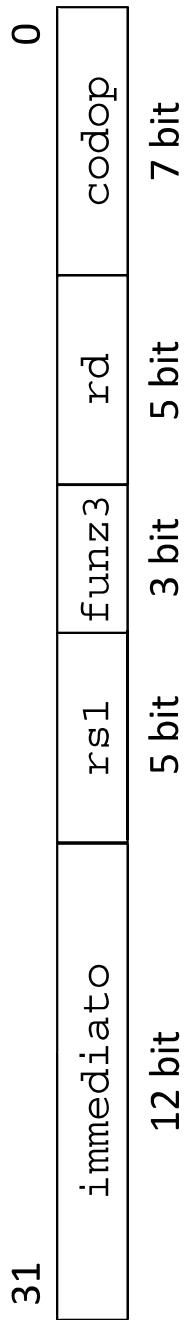
```
jalr x0, 0(x1)  Tipo I in linguaggio macchina
```

$x0 = PC + 4 ; \quad PC = x1 + 0$

Operazione nulla
Return Address

RISC-V Instruction Set

Formato di tipo I (immediato)

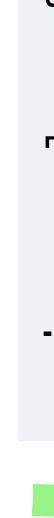


• Permette di codificare anche jal r

- Il campo immediato
 - Rappresentato in complemento a due
 - Valori possibili: da -2 048 a +2 047

Salti con offset più grandi

- Come per le costanti, anche i salti possono essere eseguiti anche ad istruzioni più lontane
- RISC-V introduce la possibilità di salto in un intervallo pari a 2^{32}
- Nuova istruzione

 **auipc rd, offset**
 **Tipo U in linguaggio macchina**

- Inserisce nel registro rd l'indirizzo di PC + (offset << 12)

Esempio: `auipc x5, 0x12345` 

`x5 = PC + 0x12345000`

Salti con offset più grandi

- Se usiamo auipc con i 20 bit più significativi dell'offset, allora possiamo aggiungere questa istruzione una istruzione che calcola
 - $PC = rd + offset[31..0]$
- Ottieniamo come risultato un salto incondizionato con offset più esteso

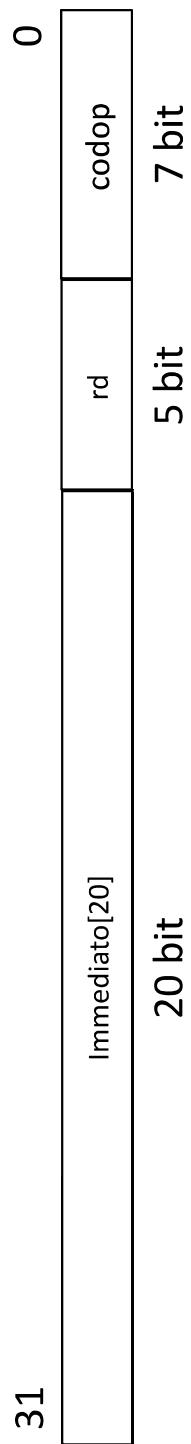
- L'istruzione da considerare è `jalr`! Ricapitolando:

```
auipc rd, offset [31..12]  
jalr x0, offset [0..11] (rd)
```

- Realizza un salto incondizionato a PC + offset [31..0]

AUIPC e linguaggio macchina

- viene usato il tipo U



~~Side effect – sovrascrittura dei registri (1)~~

```
int somma(int x, int y) {  
    int rst;  
    rst = x + y + 2;  
    return rst;  
}  
  
....  
f=f+1  
risultato=somma (f, g)  
...
```

x	<input type="checkbox"/>	x10
y	<input type="checkbox"/>	x11
rst	<input type="checkbox"/>	x20
f	<input type="checkbox"/>	x6



```
addi x6,x6,1  
jal SOMMA
```

- **Problema:** che cosa succede se il registro x5 contiene un valore usato dalla procedura chiamante?

• Soluzione:

Side effect – sovrascrittura dei registri (1)

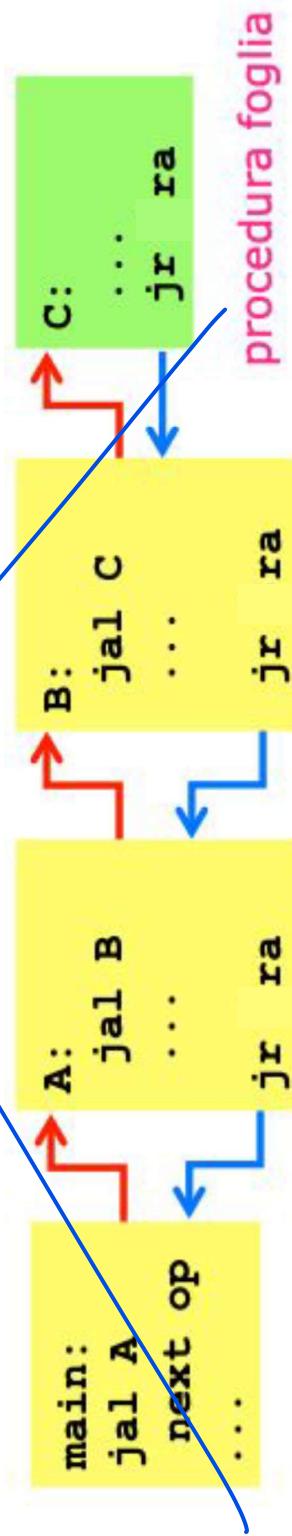
```
int somma ( int x, int y) {  
    int rst;  
    rst = x + y + 2;  
    return rst;  
}  
...  
f=f+1  
risultato=somma ( f , g)  
...
```

```
SOMMA : add x5,x10,x20  
        addi x20,x5,2  
        jalr x0,0(x1)  
        ...  
        ...  
        addi x6,x6,1  
        jal SOMMA
```

- **Problema:** che cosa succede se il registro x5 contiene un valore usato dalla procedura chiamante?
- **Soluzione:** nella procedura, salvare il valore di x5 in memoria prima di utilizzarlo (e ripristinarlo prima del ritorno al chiamante)

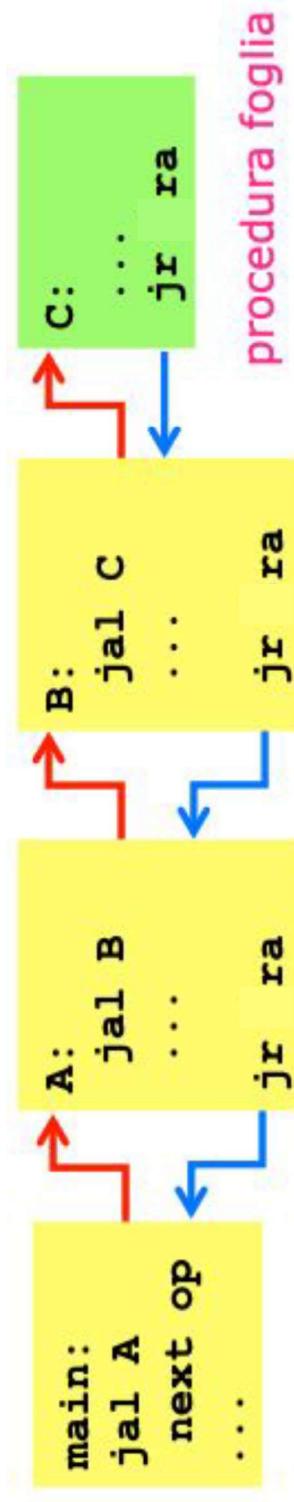
Side effect – procedure annidate (2)

- Problema
 - Nel caso di procedure annidate, il return address (`x1`) viene sovrascritto
- Soluzione:



Side effect – procedure annidate (2)

- Problema
 - Nel caso di procedure annidate, il return address (x_1) viene sovrascritto
- Soluzione:
 - La procedura chiamata, deve salvare in memoria il valore di x_1 prima di chiamare la procedura annidata con l'istruzione `jal`



Side effect – parametri numerosi (3)

•Problema:

- che cosa succede se i parametri e le variabili di una procedura superano il numero di registri disponibili?

•Soluzione:

Side effect – parametri numerosi (3)

• Problema:

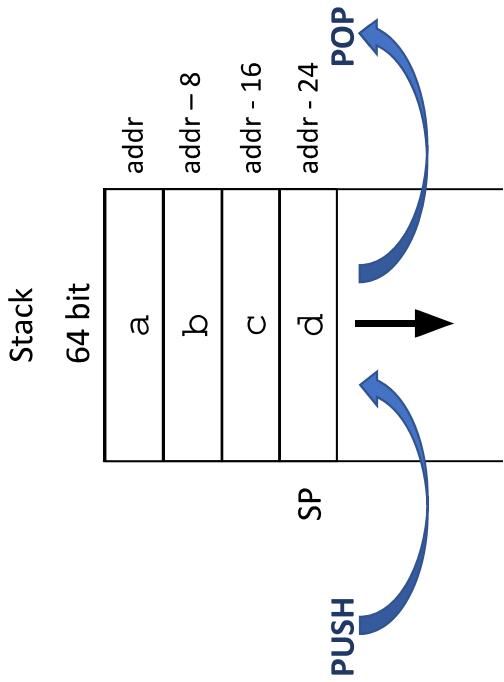
- che cosa succede se i parametri e le variabili di una procedura superano il numero di registri disponibili?

• Soluzione:

- Salvare temporaneamente i dati in memoria per caricarli nei registri prima del loro utilizzo all'interno della procedura

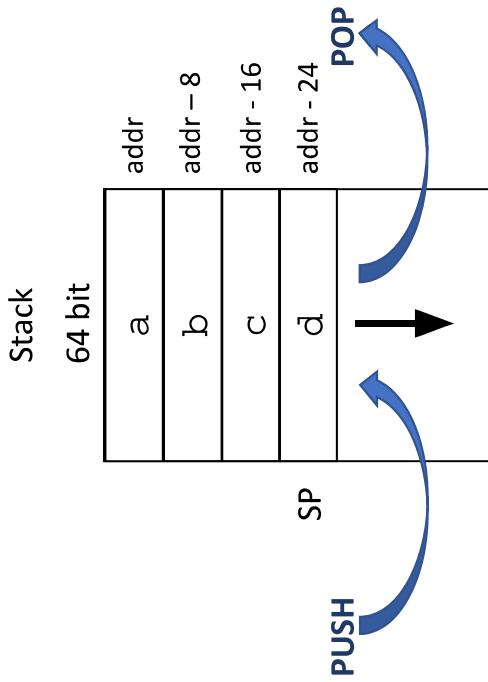
Lo stack

- In quale area di memoria è possibile salvare i registri per evitarne la perdita del valore?
- In memoria viene definita una struttura dati **dinamica**, lo stack, con queste caratteristiche
 - Accesso: Last In First Out (LIFO)
 - Operazioni:
 - PUSH: aggiunge un elemento «in cima allo stack»
 - POP: rimuove un elemento «dalla cima dello stack»
 - La cima dello stack è identificata dallo **Stack Pointer** (SP)



Lo stack

- In RISC V si usa la convenzione
 - **grow-down**: lo stack cresce da indirizzi di memoria alti verso indirizzi di memoria bassi
 - **last-full**: lo stack pointer (SP) contiene l'indirizzo dell'ultima cella di memoria occupata nello stack
 - Il valore di SP è salvato nel registro x2 (a.k.a. sp)



- PUSH
 - Decrementa SP
 - Scrive in M[SP]
- POP
 - Legge da M[SP]
 - Incrementa SP

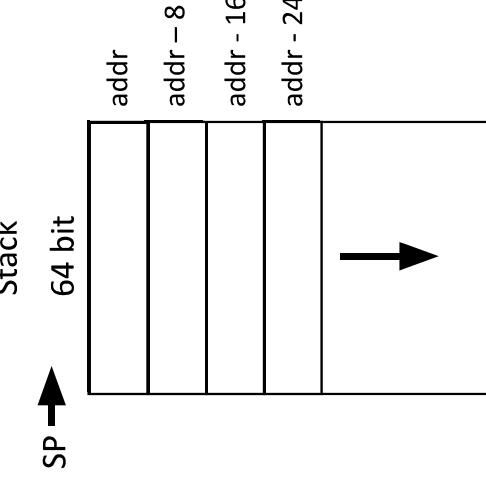
```
addi sp, sp, -8  
sd x20, 0(sp)
```

```
ld x20, 0(sp)  
addi sp, sp, 8
```

Nell'esempio di prima:

```
int somma(int x, int y) {  
    int rst;  
    rst = x + y + 2;  
    return rst;  
}  
...  
f=f+1  
risultato=somma(f,g)  
...  
}
```

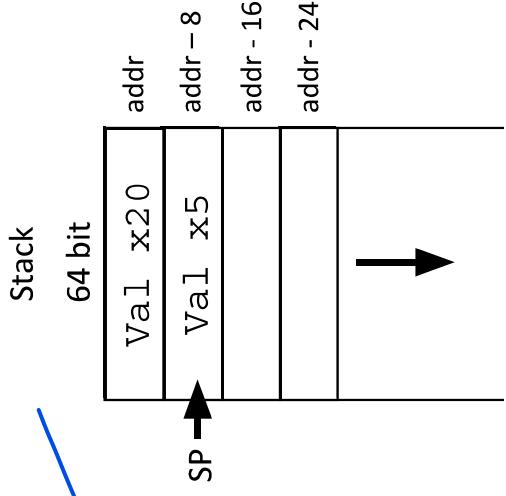
```
SOMMA: addi sp,sp,-16  
       sd x5,0(sp)  
       sd x20,8(sp) rst  
       add x5,x10,x20  
       addi x20,x5,2  
       addi x10,x20,0  
       ld x5,0(sp)  
       ld x20,8(sp)  
       addi sp,sp,16  
return jalr x0,0(x1)  
...  
...  
addi x6,x6,1  
jal SOMMA
```



Nell'esempio di prima:

```
int somma(int x, int y) {  
    int rst;  
    rst = x + y + 2;  
    return rst;  
}  
...  
f=f+1  
risultato=somma (f,g)  
...  
...
```

```
SOMMA: addi sp,sp,-16  
sd x5,0(sp)  
sd x20,8(sp)  
add x5,x10,x20  
addi x20,x5,2  
addi x10,x20,0  
ld x5,0(sp)  
ld x20,8(sp)  
addi sp,sp,16  
jalr x0,0(x1)  
...  
...  
addi x6,x6,1  
jal SOMMA
```



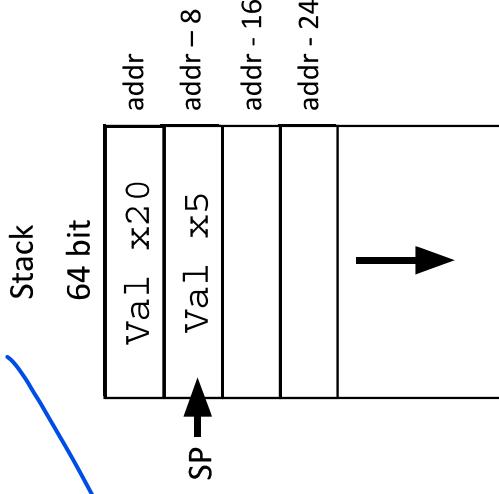
PUSH: salvataggio del valore di x5 e x20

Nell'esempio di prima:

```
int somma(int x, int y) {  
    int rst;  
    rst = x + y + 2;  
    return rst;  
}  
...  
f=f+1  
risultato=somma (f,g)  
...  
...
```

```
SOMMA: addi sp,sp,-16  
sd x5,0(sp)  
sd x20,8(sp)  
add x5,x10,x20  
addi x20,x5,2  
addi x10,x20,0  
ld x5,0(sp)  
ld x20,8(sp)  
addi sp,sp,16  
jalr x0,0(x1)  
...:  
...:  
addi x6,x6,1  
jal SOMMA
```

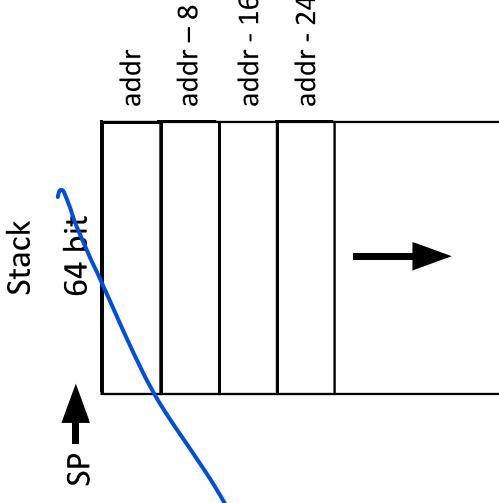
Corpo della procedura



Nell'esempio di prima:

```
int somma(int x, int y) {  
    int rst;  
    rst = x + y + 2;  
    return rst;  
}  
...  
f=f+1  
risultato=somma (f,g)  
...  
int somma (int x, int y) {  
    int rst;  
    rst = x + y + 2;  
    return rst;  
}
```

```
SOMMA: addi sp,sp,-16  
sd x5,0(sp)  
sd x20,8(sp)  
add x5,x10,x20  
addi x20,x5,2  
addi x10,x20,0  
1d x5,0(sp)  
1d x20,8(sp)  
addi sp,sp,16  
jalr x0,0(x1)  
...:  
...:  
addi x6,x6,1  
jal SOMMA
```

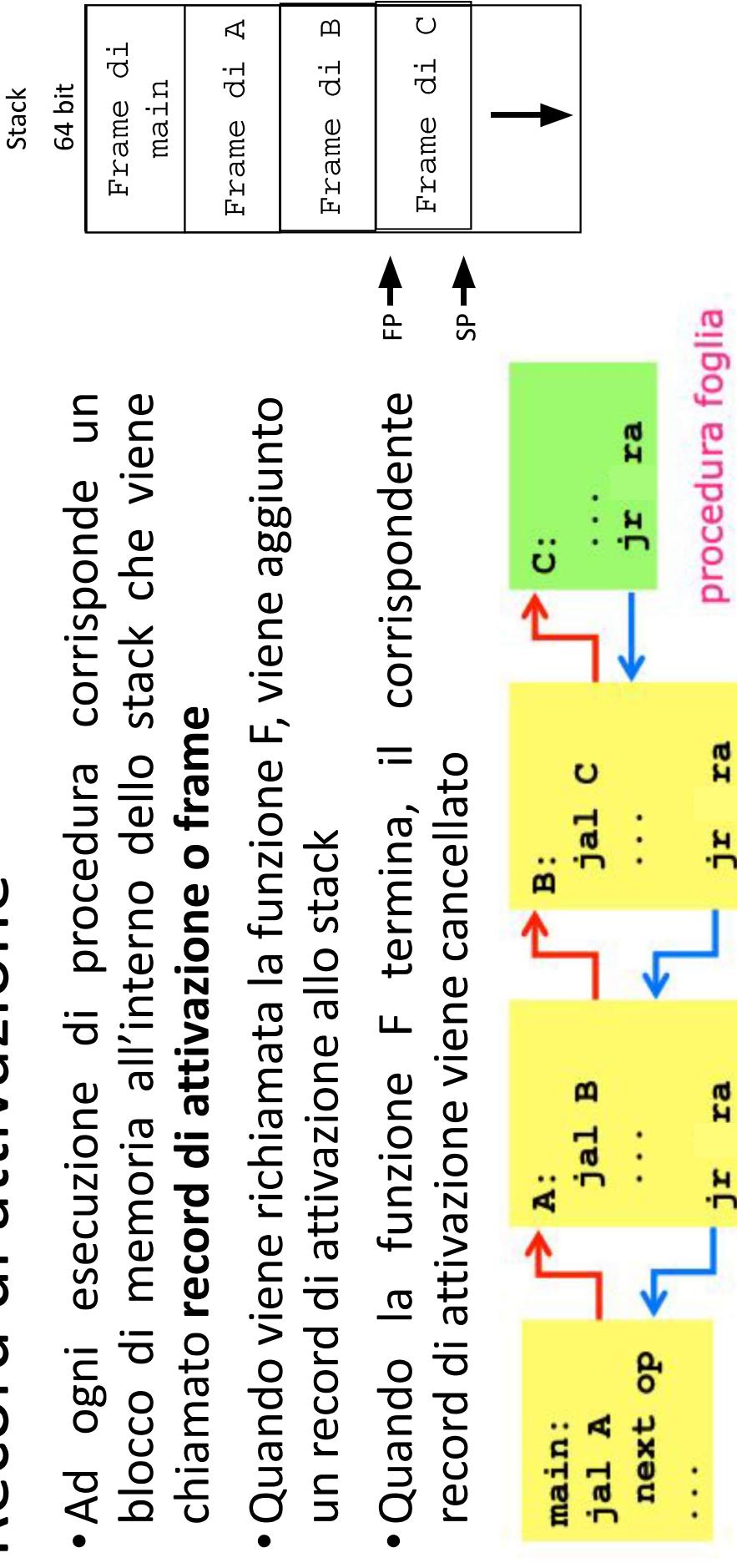


POP: ripristino del valore di x5 e x20

Record di attivazione

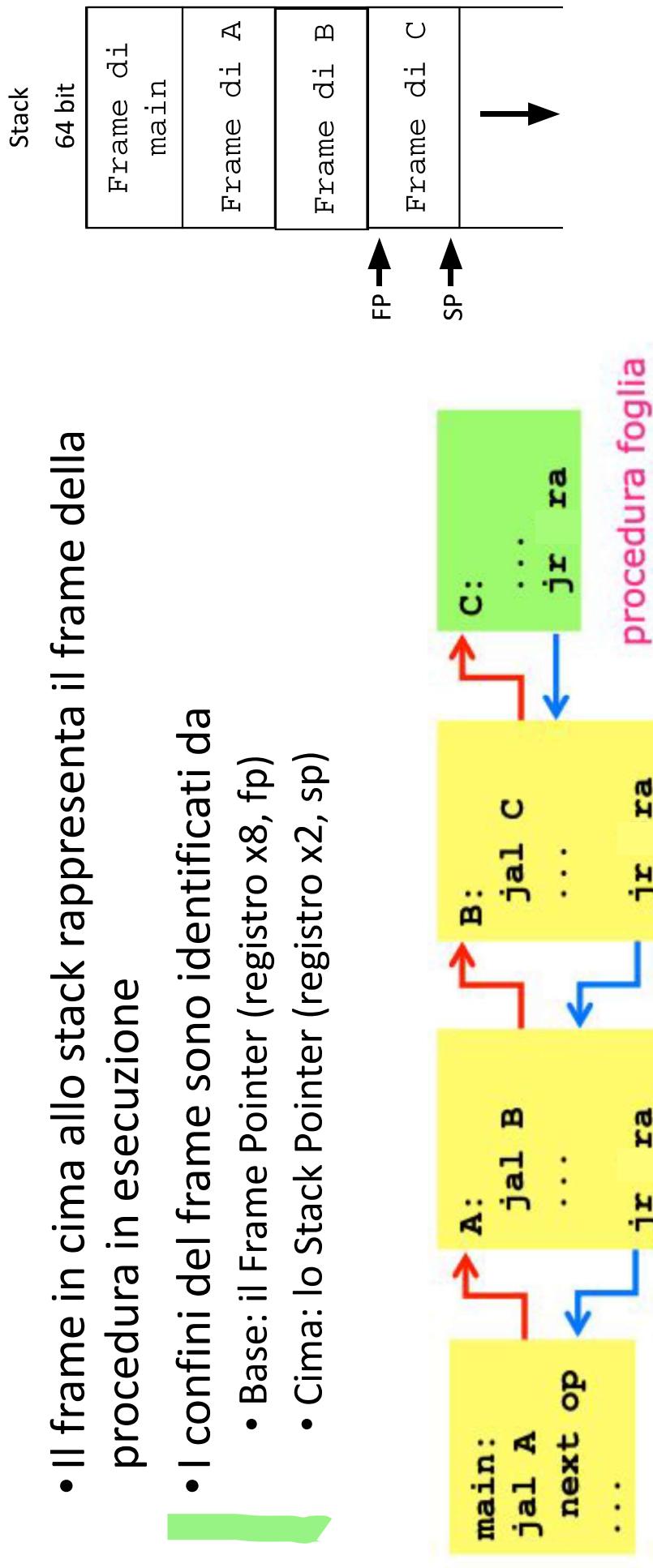
- Ad ogni esecuzione di procedura corrisponde un blocco di memoria all'interno dello stack che viene chiamato **record di attivazione o frame**
- Quando viene richiamata la funzione F, viene aggiunto un record di attivazione allo stack

- Quando la funzione F termina, il corrispondente record di attivazione viene cancellato



Record di attivazione

- Il frame in cima allo stack rappresenta il frame della procedura in esecuzione
 - I confini del frame sono identificati da
 - Base: il Frame Pointer (registro x8, fp)
 - Cima: lo Stack Pointer (registro x2, sp)



I registri e convenzioni sul loro uso

Registro	Nome	Utilizzo
x0	zero	Costante zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Puntatore a thread
x8	s0 / fp	Frame pointer (il contenuto va preservato se utilizzato dalla procedura chiamata)
x10-x11	a0-a1	Passaggio di parametri nelle procedure e valori di ritorno
x12-x17	a2-a7	Passaggio di parametri nelle procedure
x5-x7	t0-t2	Registri temporanei, non salvati in caso di chiamata
x28-x31	t3-t6	
x9	s1	Registri da salvare: il contenuto va preservato se utilizzati dalla procedura chiamata
x18-x27	s2-s11	

Convenzione nell'uso e salvataggio dei registri

- Chi è responsabile di salvare i registri quando si effettuano chiamate di funzioni?
 - La funzione chiamante conosce quali registri sono importanti per sé e che dovrebbero essere salvati
 - La funzione chiamata conosce quali registri userà e che dovrebbero essere salvati prima di modificarli
- Bisogna evitare le inefficienze Minimo salvataggio dei registri:
 - La funzione chiamante potrebbe salvare tutti i registri che sono importanti per sé, anche se la procedura chiamata non li modificherà
 - La funzione chiamata potrebbe salvare tutti i registri che si appresta a modificare, anche quelli che non verranno poi utilizzati dalla procedura chiamante una volta che la procedura chiamata le avrà restituito il controllo

Convenzione nell'uso e salvataggio dei registri

Necessità di stabilire delle convenzioni

- I registri $x_{10} - x_{17}$ ($a_0 - a_7$) , $x_5 - x_7$ e $x_{28} - x_{31}$ ($t_0 - t_6$)
 - possono essere modificati dal chiamato **senza nessun meccanismo di ripristino**
 - Il chiamante se necessario dovrà salvare i valori dei registri prima dell'invocazione della procedura
- I registri x_1 (ra) , x_2 (sp) , x_3 (gp) , x_4 (tp) , x_8 (fp) , x_9 e $x_{18} - x_{27}$ ($s_1 - s_{11}$)
 - Se modificati dal chiamato devono essere salvati e poi ripristinati prima del ritorno al chiamante
 - Il chiamante non è tenuto al loro salvataggio e ripristino

vad, tress. t'df

Le fasi di una invocazione di procedura

Possiamo dividere l'invocazione di una procedura in 7 fasi:

1. Pre-chiamata
 2. Invocazione della procedura
 3. Prologo del chiamato
 4. Corpo della procedura
 5. Epilogo lato chiamato
 6. Ritorno al chiamante
 7. Post-chiamata
- chiamante
- chiamato
- chiamante

Le fasi di una invocazione di procedura

Fase 1 – Pre-chiamata del chiamante

- Eventuale **salvataggio** registri da preservare nel chiamante

- Si assume che $x_{10}-x_{17}$ (a_0-a_7) , $x_{5}-x_7$ e $x_{28}-x_{31}$ (t_0-t_6) , possano essere sovrascritti dal chiamato

- se li si vuole preservare vanno salvati nello stack (dal chiamante) – vedi caso 1

- il caso 2 mostra un caso in cui non è necessario salvare il contenuto del registro associato alla variabile f

Preparazione degli argomenti della funzione

- I primi 8 argomenti vengono posti in $x_{10}-x_{17}$, ovvero a_0-a_7 (nuovi valori)
- Gli eventuali altri argomenti oltre l'ottavo vanno salvati nello stack (EXTRA_ARGS), così che si trovino subito sopra il frame della funzione chiamata

1. Pre-chiamata	Chiamante
2. Invocazione della procedura	
3. Prologo del chiamato	Chiamato
4. Corpo della procedura	
5. Epilogo lato chiamato	
6. Ritorno al chiamante	
7. Post-chiamata	

```
int somma (int x, int y) {  
    x=x+y;  
    return x;  
}  
...:  
f=f+1;  
risultato=somma (f,g);  
printf ("%d", risultato);  
return;
```

1

```
int somma (int x, int y) {  
    x=x+y;  
    return x;  
}  
...:  
f=f+1;  
risultato=somma (f,g);  
return;
```

2

Le fasi di una invocazione di procedura



- **Fase 2 – Invocazione della procedura**

- Istruzione `jal NOME PROCEDURA`

- **Fase 3 – Prologo lato chiamato**

- Eventuale allocazione del call-frame sullo stack (aggiornare `sp`)
 - Eventuale salvataggio registri che si intende sovrascrivere
 - Salvataggio degli argomenti $x10-x17$ ($a0-a7$) solo se la funzione ha necessità di riusarli nel corpo di questa funzione, successivamente a ulteriori chiamate a funzione che usino tali registri, (nota: negli altri casi $x10-x17$ possono essere sovrascritti)
 - Salvataggio di $x1$ (ra) nel caso in cui la procedura non sia foglia
 - Salvataggio di $x8$ (fp), solo se utilizzato all'interno della procedura
 - Salvataggio di $x9$ e $x18-x27$ ($s1-s11$) se utilizzati all'interno della procedura (il chiamante si aspetta di trovarli intatti)
 - Eventuale inizializzazione di fp : punta al nuovo call-frame

Le fasi di una invocazione di procedura

- **Fase 4 – Corpo della procedura**

- Istruzioni che implementano il corpo della procedura

- **Fase 5 – Epilogo lato chiamato**

- Se deve essere restituito un valore dalla funzione

- Tale valore viene posto in x_{10} (e \times_{11}) ovvero a_{0-a_1}

- I registri (se salvati) devono essere ripristinati

- $x_{10-x_{17}}$, cioè a_{0-a_7} (nel caso siano stati salvati all'interno della funzione)
 - x_9 e $x_{18-x_{27}}$ ($s_{1-s_{11}}$)
 - x_1 (r_a)
 - x_8 (f_p)
- Notare che sp deve solo essere aumentato di opportuno offset (lo stesso sottratto nella Fase 3)





Le fasi di una invocazione di procedura

- **Fase 6 – Ritorno al chiamante**

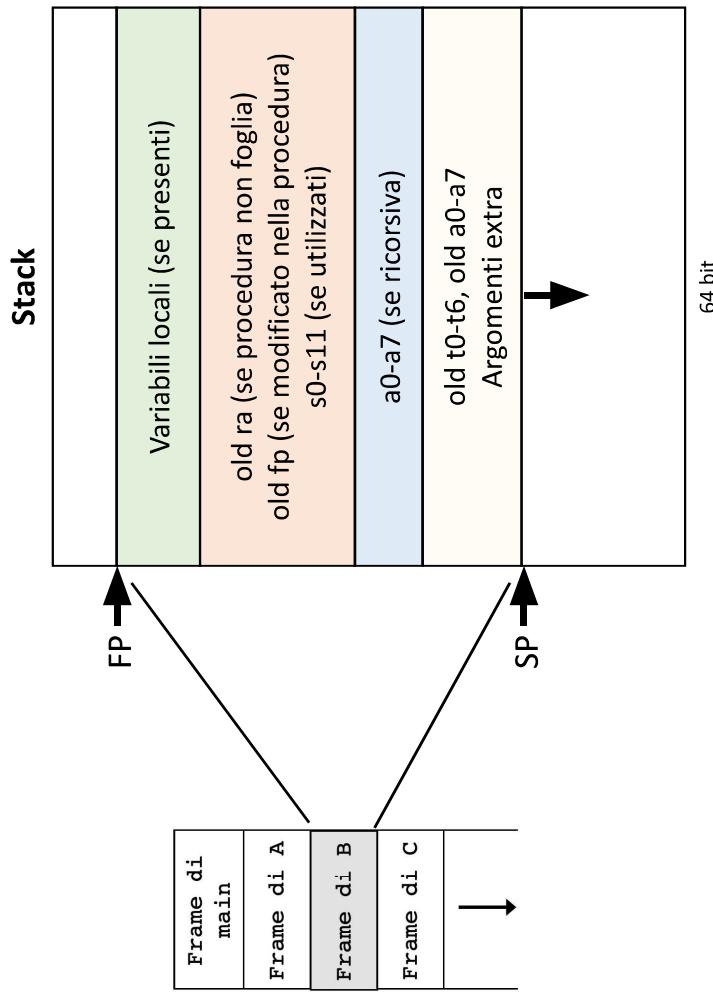
- Istruzione `j1ra x0, 0 (x1)`

- **Fase 7 – Post-chiamata lato chiamante**

- Eventuale uso del risultato della funzione (in `x10 e x11`, cioè `a0-a1`)
- Ripristino dei valori `x5-x7` e `x28-x31` (`t0-t6`), `x10-x17` (`a0-a7`) vecchi, eventualmente salvati

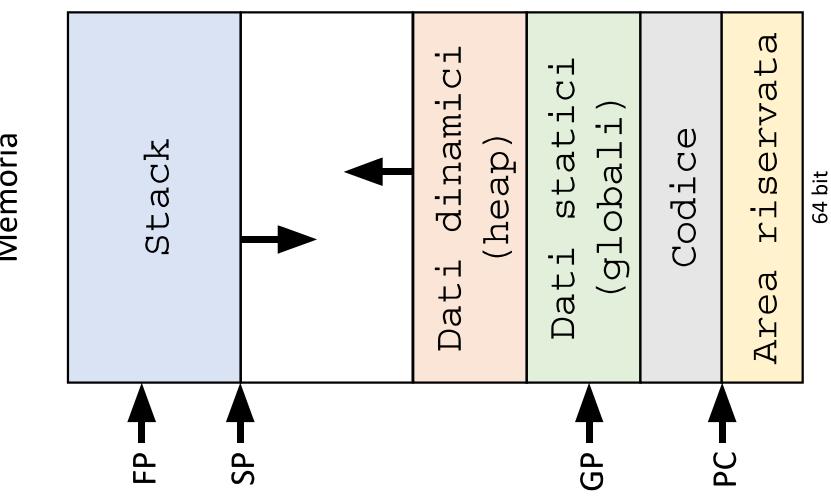
Record di attivazione: struttura

- Se utilizzato, il registro fp (frame pointer) viene inizializzato al valore di sp all'inizio della chiamata
- fp consente di avere un riferimento alle variabili locali che non muta con l'esecuzione della procedura
- Se lo stack non contiene variabili locali alla procedura, il compilatore risparmia tempo di esecuzione evitando di impostare e ripristinare il frame.



Organizzazione della memoria

- In memoria, oltre allo stack, trovano posto
 - Dati allocati dinamicamente (ad esempio attraverso una malloc)
 - Dati statici e variabili globali
 - Codice del programma
- Lo stack e i dati dinamici crescono in direzioni differenti, in modo da ottimizzarne la gestione



Esempio: area di un triangolo

```
long moltiplicazione(long a, long b) {  
    long rst=a;  
    for(long i=1;i<b;i++)  
        rst = rst+a;  
    return rst;  
}  
  
long area(long base, long altezza) {  
    long rst = moltiplicazione(base, altezza) / 2;  
    return rst;  
}
```

```
Funzione moltiplicazione  
moltiplicazione:  
    addi sp, sp, -16      # allocazione del frame nello stack  
    sd ra, 0(sp)         # salvataggio dell'indirizzo di ritorno  
    sd s0, 0(sp)         # salvataggio del precedente frame pointer  
    addi s0, sp, 8        # aggiornamento del frame pointer  
    sd a0, 0(s0)          # salvataggio variabile locale  
  
    ld t2, 0($0)  
    li t0, 1  
for:  
    bge t0,a1,endfor  
    add t2,a0,t2  
    addi t0,t0,1  
    j for  
endfor:  
    ld s0, 0(sp)          # recupera il frame pointer  
    ld ra, 0(sp)           # recupera l'indirizzo di ritorno  
    addi sp, sp, 16         # elimina il call frame dallo stack  
    add a0,t2,zero          # valore di ritorno in a0  
    jr ra                  # ritorna al chiamante
```

Si supponga di non avere una
operazione di moltiplicazione

Il codice può essere ottimizzato

Esempio: area di un triangolo

```
long moltiplicazione(long a, long b) {
    long rst=a;
    for(long i=1;i<b;i++)
        rst = rst+a;
    return rst;
}

long area(long base, long altezza) {
    long rst = moltiplicazione(base, altezza)/2;
    return rst;
}
```

Si supponga di non avere una operazione di moltiplicazione

```
Funzione area con invocazione della funzione moltiplicazione

area:
    # crea il call frame sullo stack (24 byte=ra+fp+rst)
    # lo stack cresce verso il basso
    addi sp, sp, -24      # allocazione del call frame nello stack
    sd ra, 8(sp)         # salvataggio dell'indirizzo di ritorno
    sd s0, 0(sp)          # salvataggio del precedente frame pointer
    addi s0, sp, 24        # aggiornamento del frame pointer

    # calcolo dell'area
    jal ra, moltiplicazione
    sd a0, 0($0)          # salva il risultato della moltiplicazione
                           # nella variabile locale rst

    srai a0,a0,1

    # uscita dalla funzione
    ld s0, 0(sp)           # recupera il frame pointer
    ld ra, 8(sp)           # recupera l'indirizzo di ritorno
    addi sp, sp, 24          # elimina il call frame dallo stack
    jr ra                  # ritorna al chiamante
```

Il codice può essere ottimizzato

Esempio: area di un triangolo

```
long moltiplicazione(long a, long b) {
    long rst=a;
    for(long i=1;i<b;i++)
        rst = rst+a;
    return rst;
}

long area(long base, long altezza) {
    long rst = moltiplicazione(base, altezza)/2;
    return rst;
}
```

Si supponga di non avere una
operazione di moltiplicazione

```
Invocazione della funzione area con
parametri base=20 , altezza=23

_start:
    li a0, 20 # salvo la base in a0
    li a1, 23 # salvo l'altezza in a1

    # chiama triangolo(base,altezza)
    jal ra, area # altezza in a0, base in a1
    add t0, a0, zero # salva il risultato in t0

    # stampa messaggio per il risultato
    la a0, visris
    addi a7, zero, 4
    ecall

    # stampa il risultato
    add a0, t0, zero
    addi a7, zero, 1
    ecall

    # stampa \n
    la a0, RTN
    addi a7, zero, 4
    ecall

    # exit
    addi a7, zero, 10
    ecall
```

Il codice può essere ottimizzato

RISC-V Instruction Set

149

Un esempio

```

_start:          li a0, 20 # salvo la base in a0
                li a1, 23 # salvo l'altezza in a1

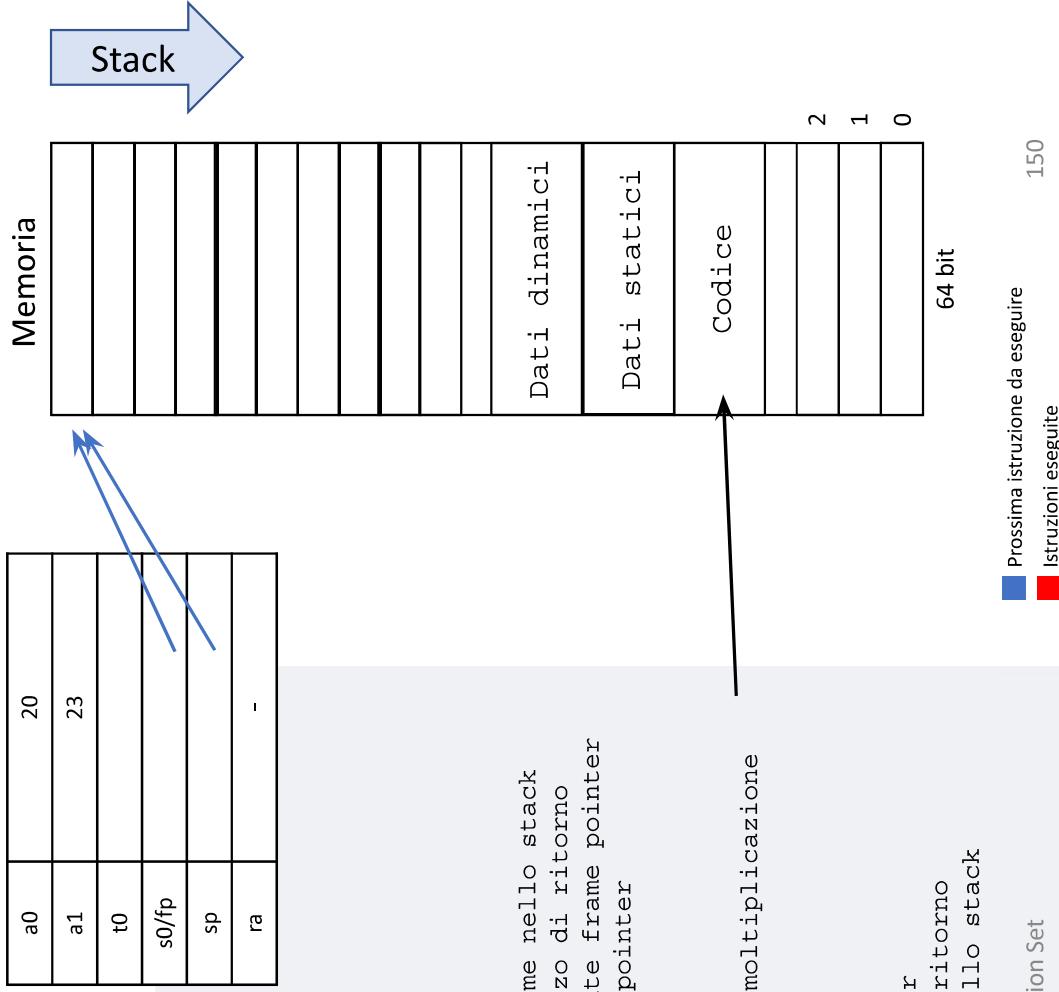
# chiama triangolo(base,altezza)
0x0000000000400008 jal ra, area      # altezza in a0, base in a1
add t0, a0, zero      # salva il risultato in t0
0x000000000040000C

area:
# crea il call frame sullo stack (24 byte=ra+fp+rst)
# lo stack cresce verso il basso
0x0000000000400010 addi sp, sp, -24    # allocazione del call frame nello stack
0x0000000000400014 sd ra, 8(sp)        # salvataggio dell'indirizzo di ritorno
0x0000000000400018 sd s0, 0(sp)         # salvataggio del precedente frame pointer
0x000000000040001C addi s0, sp, 16       # aggiornamento del frame pointer

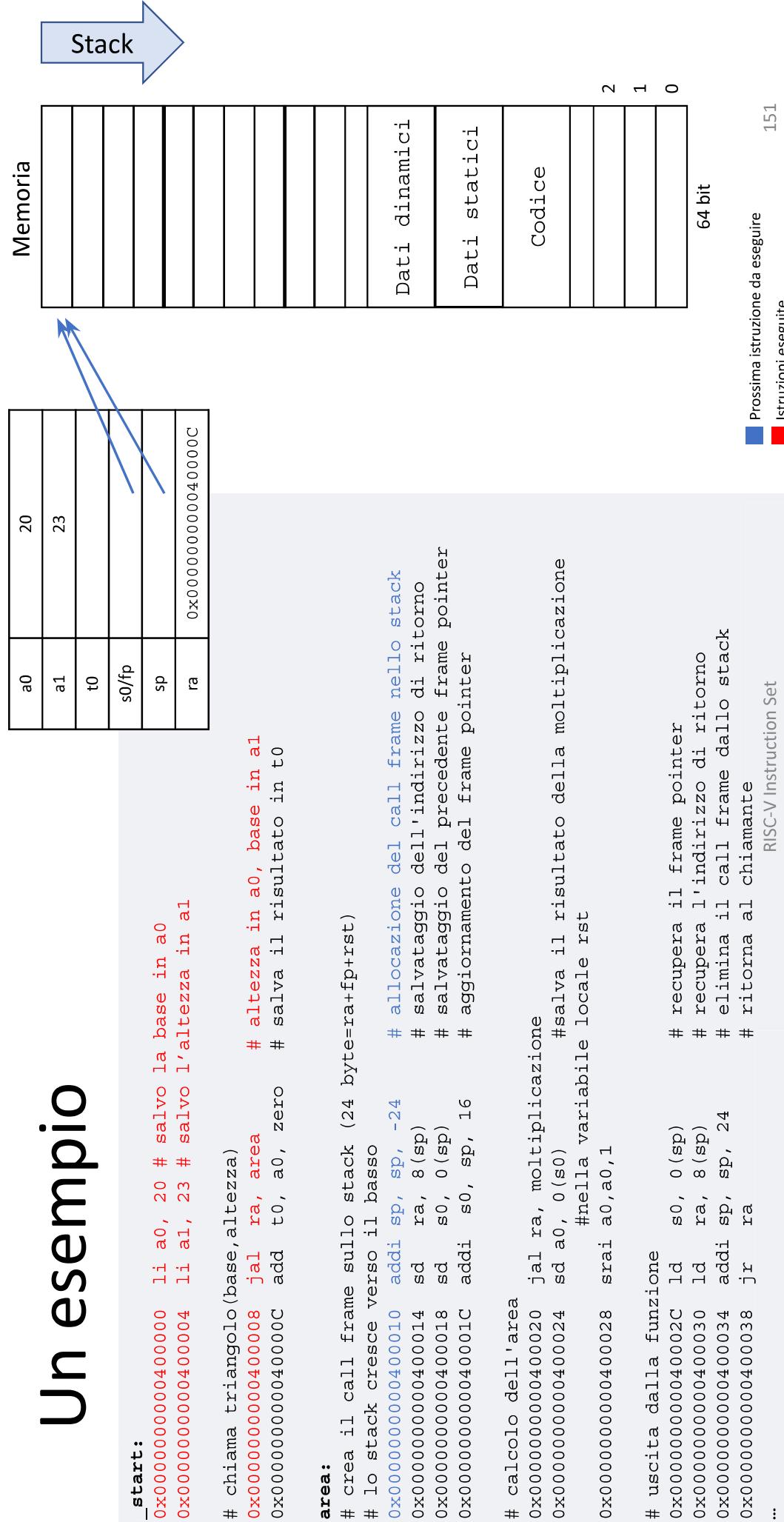
# calcolo dell'area
0x0000000000400020 jal ra, moltiplicazione
0x0000000000400024 sd a0, 0(s0)          #salva il risultato della moltiplicazione
                                            #nella variabile locale rst
0x0000000000400028 srai a0,a0,1

# uscita dalla funzione
0x000000000040002C ld s0, 0(sp)          # recupera il frame pointer
0x0000000000400030 ld ra, 8(sp)           # recupera l'indirizzo di ritorno
0x0000000000400034 addi sp, sp, 24        # elimina il call frame dallo stack
0x0000000000400038 jr ra                  # ritorna al chiamante
...

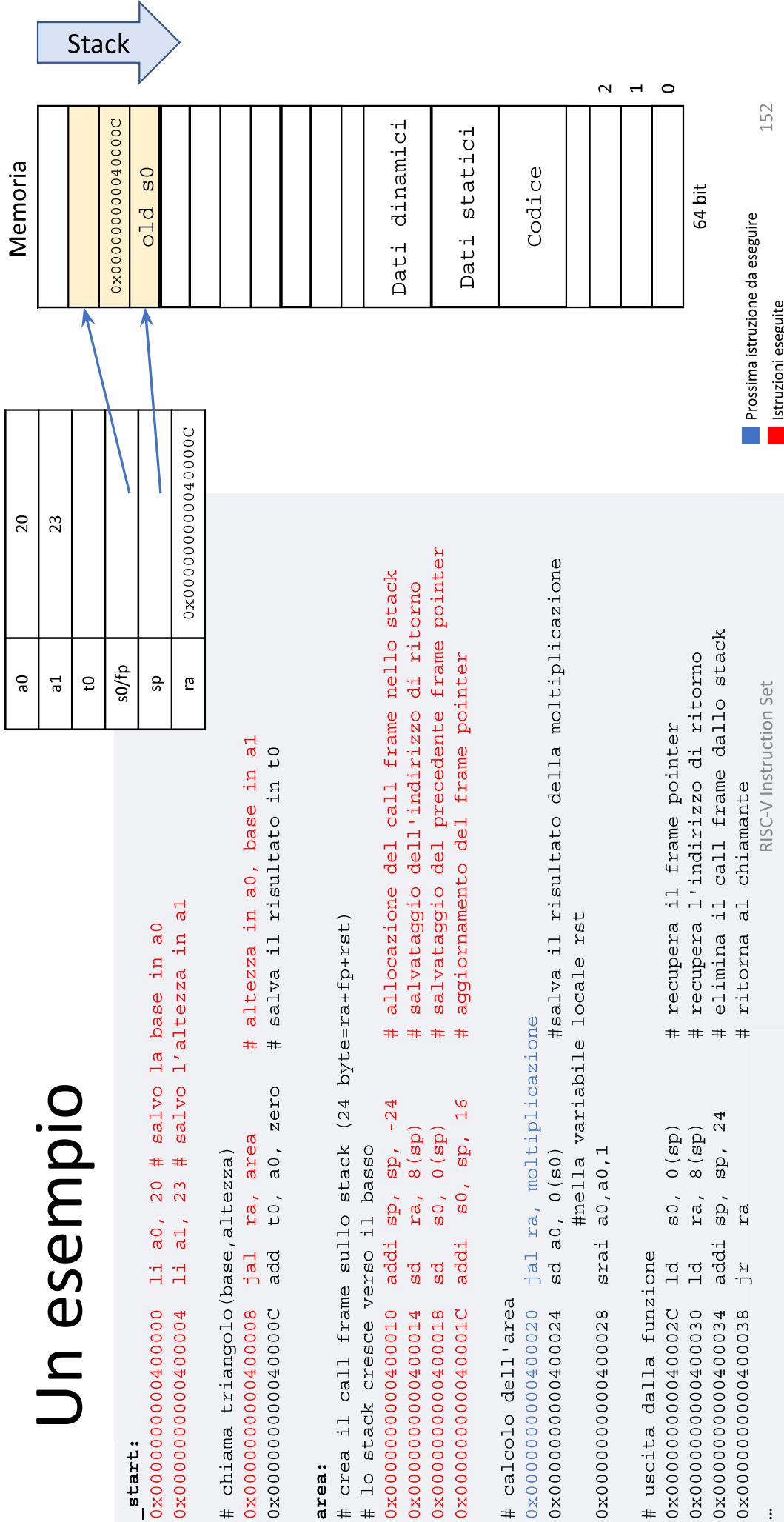
```



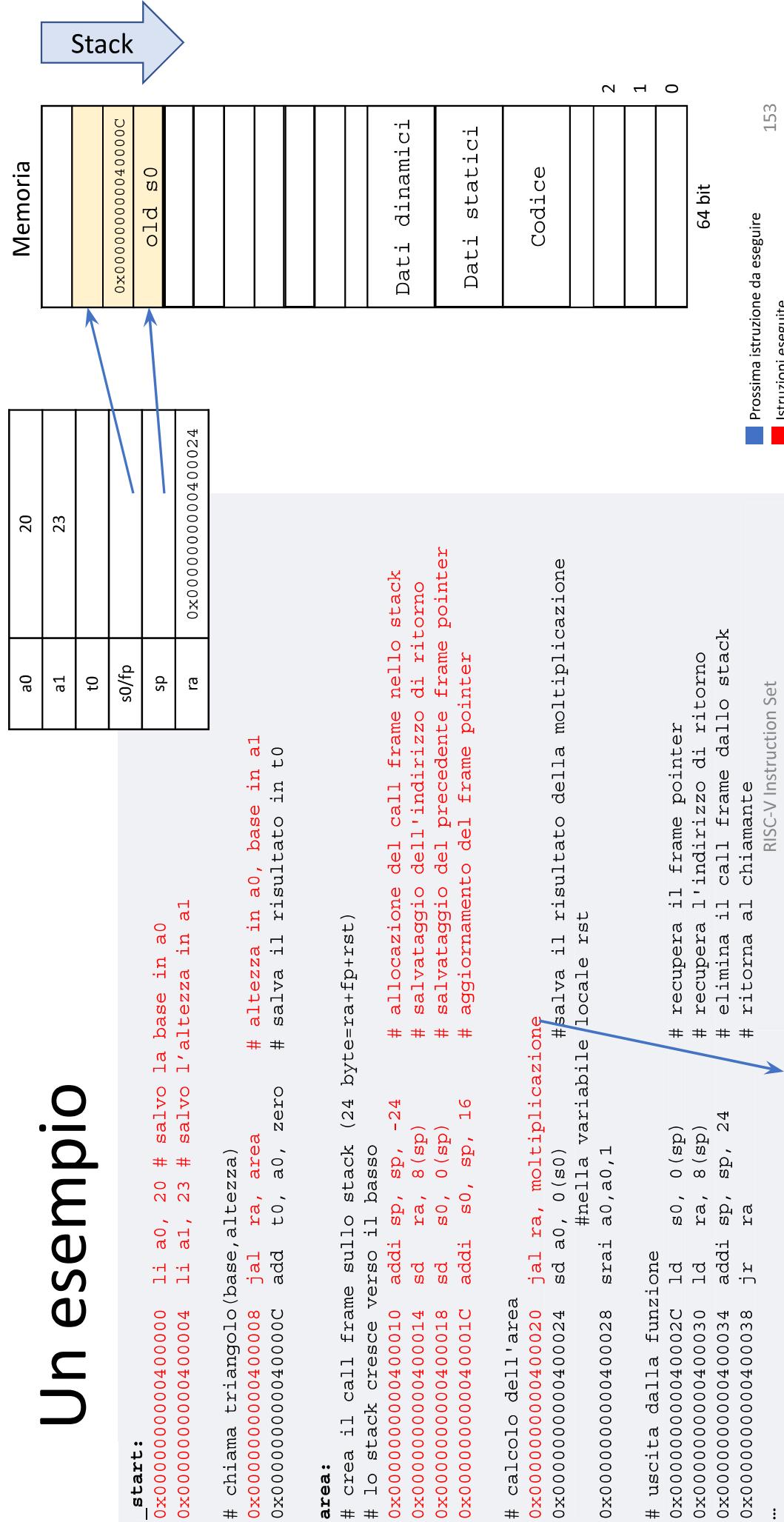
Un esempio



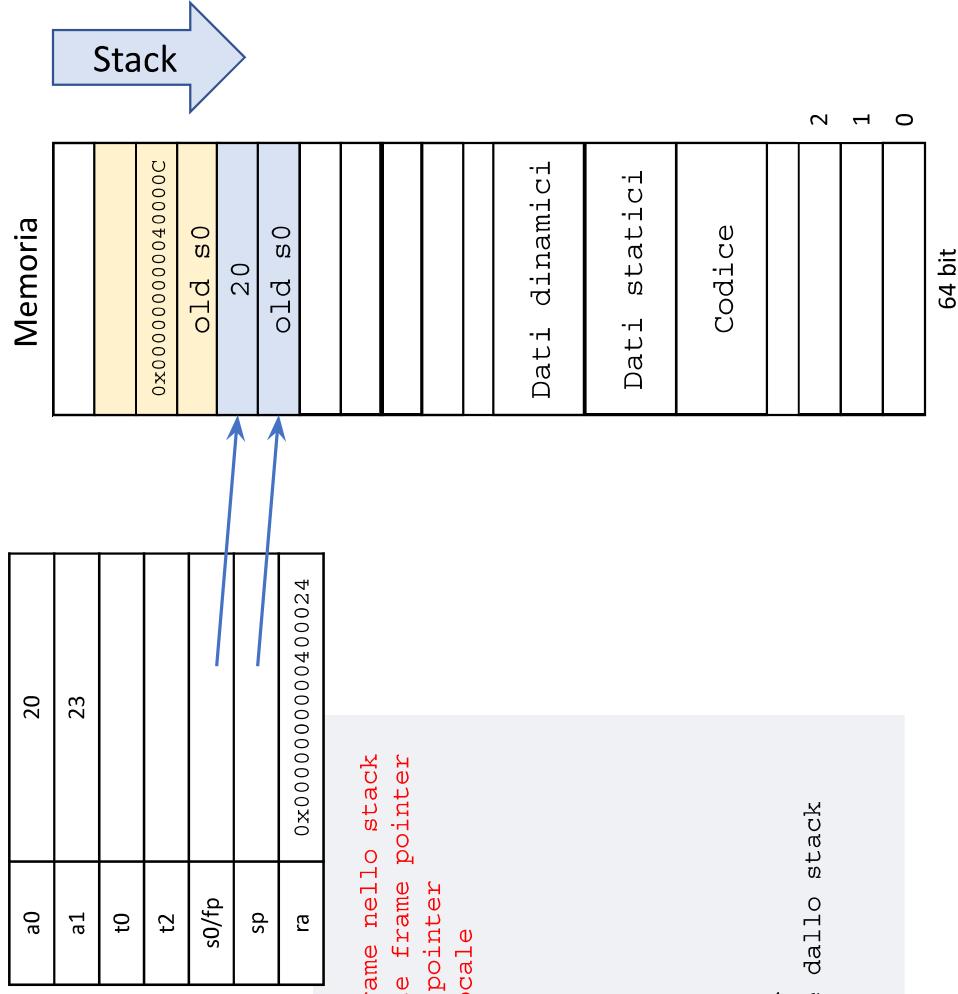
Un esempio



Un esempio



Un esempio



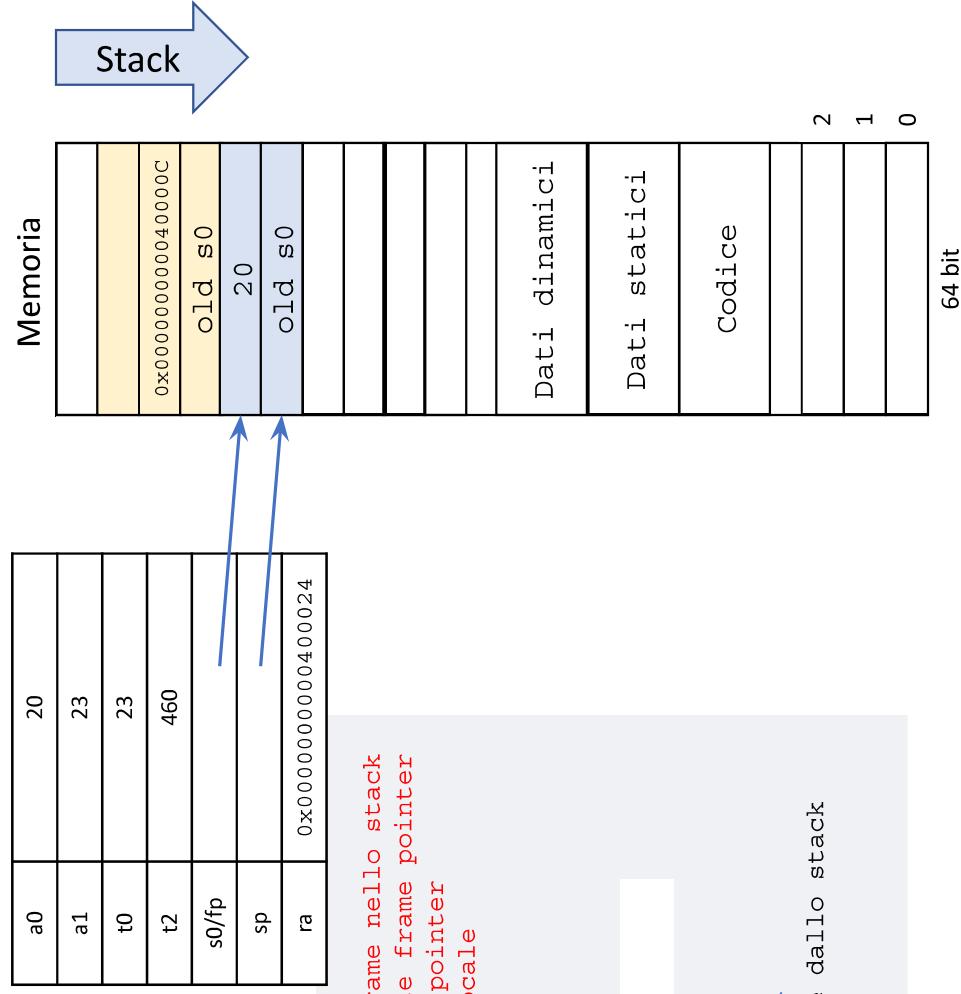
■ Prossima istruzione da eseguire
■ Istruzioni eseguite

RISC-V Instruction Set

154

64 bit

Un esempio



```

moltiplicazione:
0x0000000000400038 addi sp, sp, -16      # allocazione del call frame nello stack
0x000000000040003C sd s0, 0(sp)           # salvataggio del precedente frame pointer
0x0000000000400040 addi s0, sp, 0          # aggiornamento del frame pointer
0x0000000000400044 sd a0, 0($0)            # salvataggio variabile locale

0x0000000000400048 ld t2, 0($0)
0x000000000040004C li t0, 1

for:
0x0000000000400050 bge t0, a1,endfor
0x0000000000400054 add t2,a0,t2
0x0000000000400058 addi t0,t0,1
0x000000000040005C j for

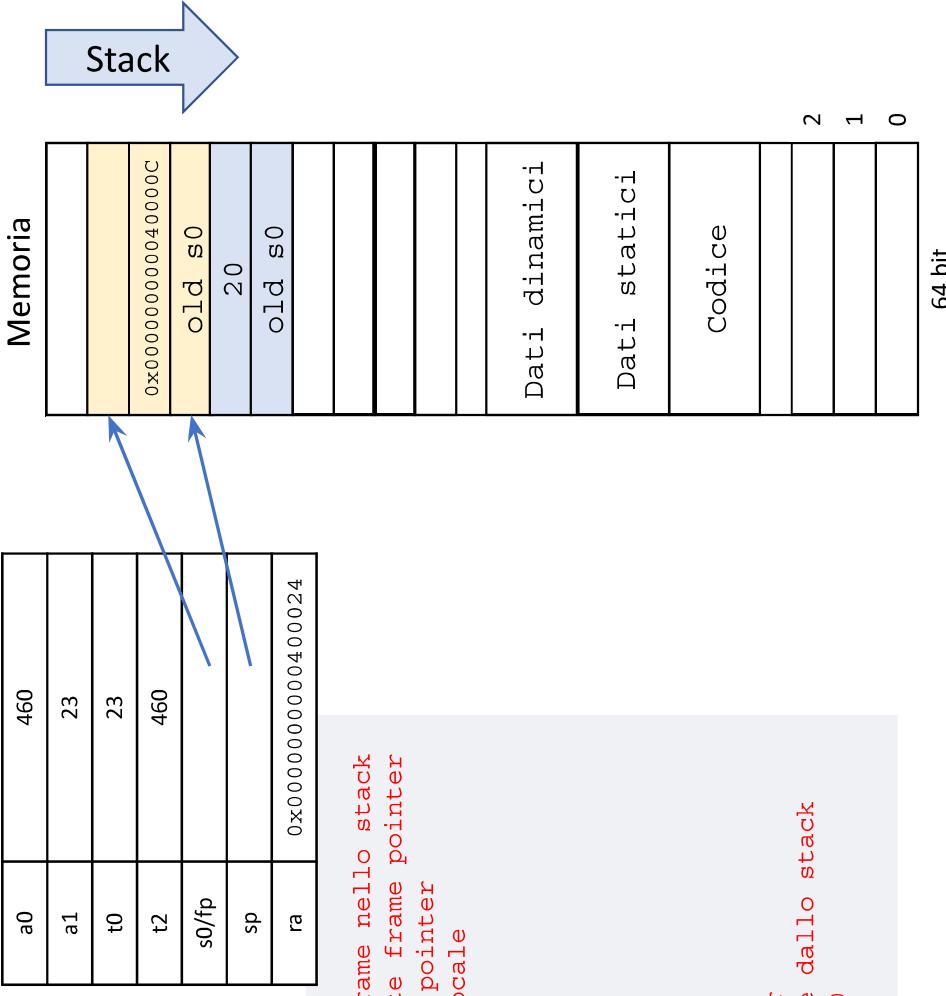
endfor:                                         # recupera il frame pointer
0x0000000000400060 ld s0, 0(sp)           # elimina il call frame dallo stack
0x0000000000400064 addi sp, sp, 16         # valore di ritorno in a0
0x0000000000400068 add a0,t2,zero        # ritorna al chiamante
0x000000000040006C jr ra

```

155
Prossima istruzione da eseguire
Istruzioni eseguite

RISC-V Instruction Set

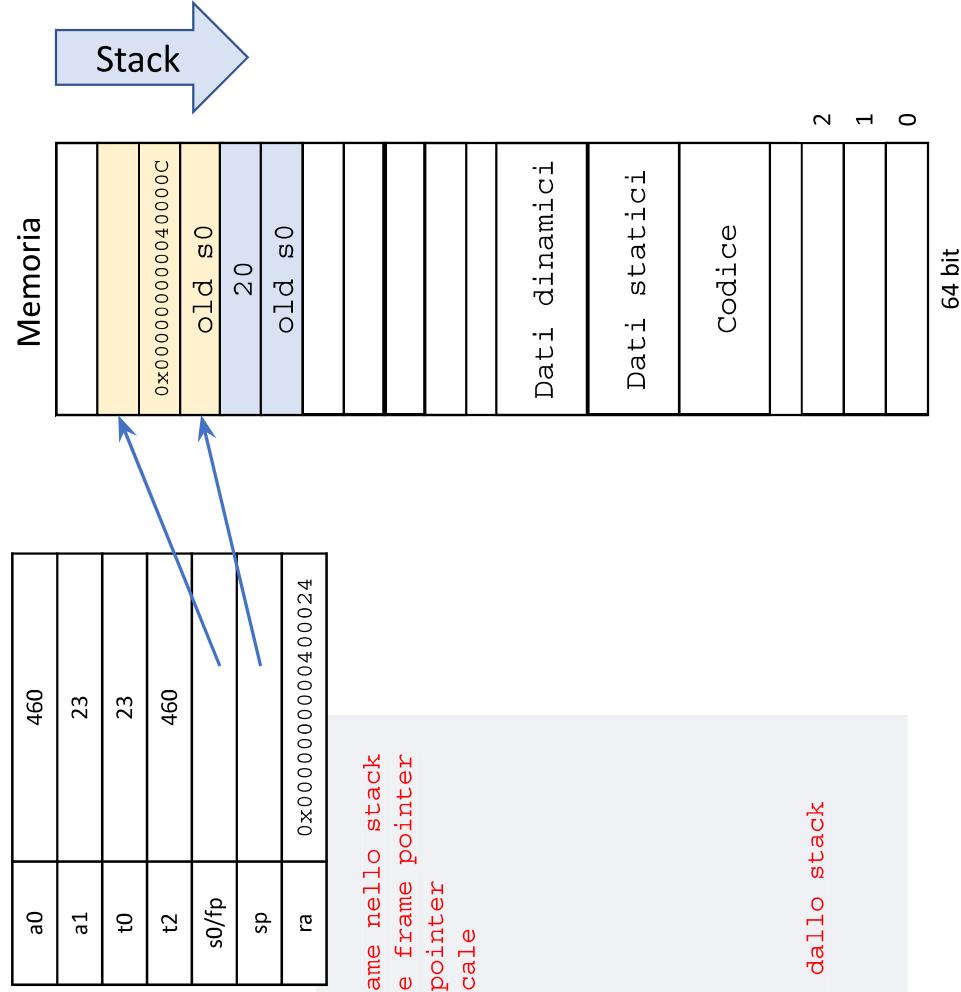
Un esempio



64 bit

- Prossima istruzione da eseguire
- Istruzioni eseguite

Un esempio



moltiplicazione:

```

0x000000000000400038 addi sp, sp, -16
0x00000000000040003C sd s0, 0(sp)
0x000000000000400040 addi s0, sp, 0
0x000000000000400044 sd a0, 0(s0)

0x000000000000400048 ld t2, 0($0)
0x00000000000040004C li t0, 1

for:
0x000000000000400050 bge t0, a1, endfor
0x000000000000400054 add t2, a0, t2
0x000000000000400058 addi t0, t0, 1
0x00000000000040005C j for

endfor:
0x000000000000400060 ld s0, 0(sp)
0x000000000000400064 addi sp, sp, 16
0x000000000000400068 add a0, t2, zero
0x00000000000040006C jr ra

```

```
# allocazione del call frame nello stack  
# salvataggio del precedente frame pointer  
# aggiornamento del frame pointer  
# salvataggio variabile locale
```

```
# recupera il frame pointer
# elimina il call frame dallo stack
# valore di ritorno in a0
# ritorna al chiamante
```

RISC-V Instruction Set

157

157

Un esempio

```

_start:          li a0, 20 # salvo la base in a0
                li a1, 23 # salvo l'altezza in a1

# chiama triangolo(base,altezza)
0x0000000000400008 jal ra, area      # altezza in a0, base in a1
add t0, a0, zero      # salva il risultato in t0
0x000000000040000C

area:
# crea il call frame sullo stack (24 byte=ra+fp+rst)
# lo stack cresce verso il basso
0x0000000000400010 addi sp, -24      # allocazione del call frame nello stack
0x0000000000400014 sd ra, 8(sp)      # salvataggio dell'indirizzo di ritorno
0x0000000000400018 sd s0, 0(sp)      # salvataggio del precedente frame pointer
0x000000000040001C addi s0, sp, 16      # aggiornamento del frame pointer

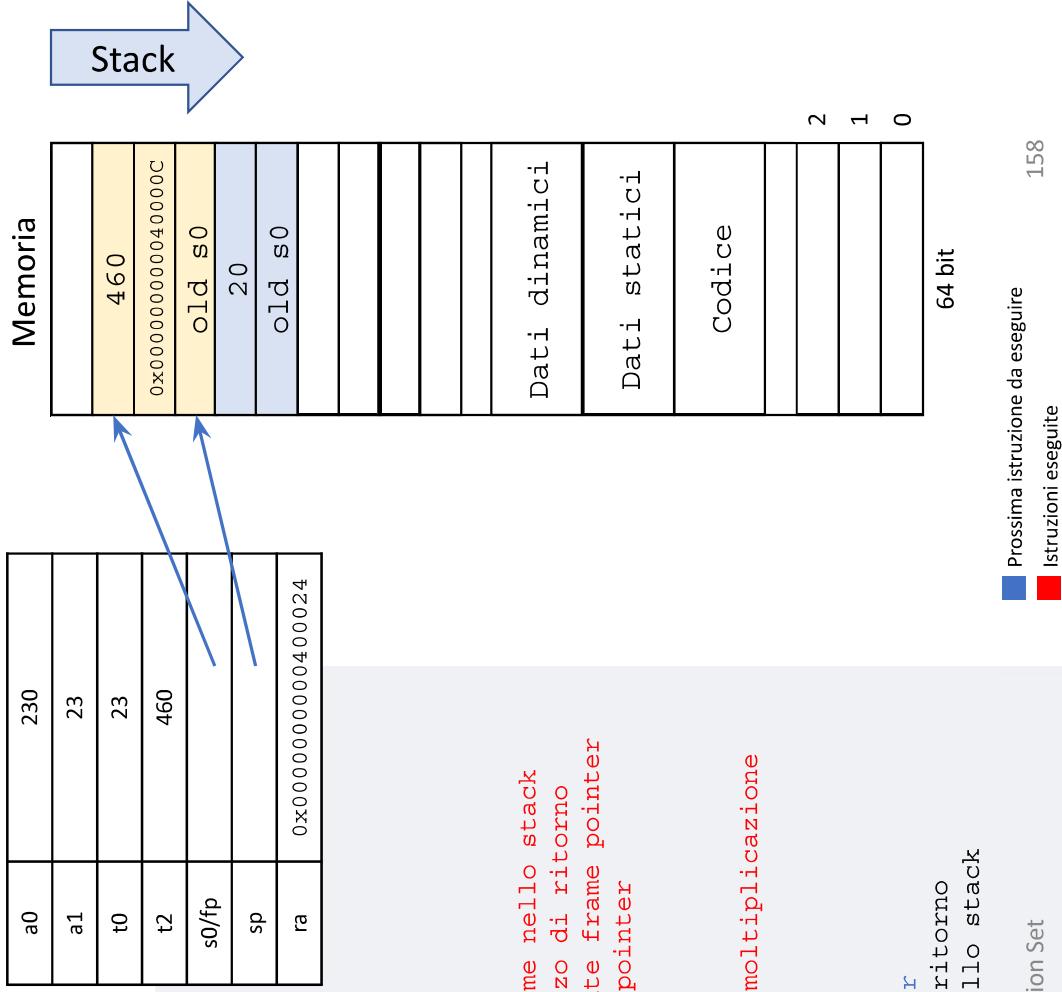
# calcolo dell'area
0x0000000000400020 jal ra, moltiplicazione
0x0000000000400024 sd a0, 0(s0)      #salva il risultato della moltiplicazione
                                         #nella variabile locale rst
                                         #divide per due

0x0000000000400028 srai a0,1.        # divide per due

# uscita dalla funzione
0x000000000040002C ld s0, 0(sp)      # recupera il frame pointer
0x0000000000400030 ld ra, 8(sp)      # recupera l'indirizzo di ritorno
0x0000000000400034 addi sp, sp, 24      # elimina il call frame dallo stack
0x0000000000400038 jr ra             # ritorna al chiamante
                                         ...
                                         ...

# RISC-V Instruction Set
# Prossima istruzione da eseguire
# Istruzioni eseguite

```



■ Prossima istruzione da eseguire
■ Istruzioni eseguite

158

Un esempio

```

_start:          li a0, 20 # salvo la base in a0
                li a1, 23 # salvo l'altezza in a1

# chiama triangolo(base,altezza)
0x0000000000400008 jal ra, area      # altezza in a0, base in a1
add t0, a0, zero      # salva il risultato in t0
0x000000000040000C

area:
# crea il call frame sullo stack (24 byte=ra+fp+rst)
# lo stack cresce verso il basso
0x0000000000400010 addi sp, -24      # allocazione del frame nello stack
0x0000000000400014 sd ra, 8(sp)     # salvataggio dell'indirizzo di ritorno
0x0000000000400018 sd s0, 0(sp)      # salvataggio del precedente frame pointer
0x000000000040001C addi s0, sp, 16    # aggiornamento del frame pointer

# calcolo dell'area
0x0000000000400020 jal ra, moltiplicazione
0x0000000000400024 sd a0, 0(s0)      #salva il risultato della moltiplicazione
                                         #nella variabile locale rst
                                         #divide per due

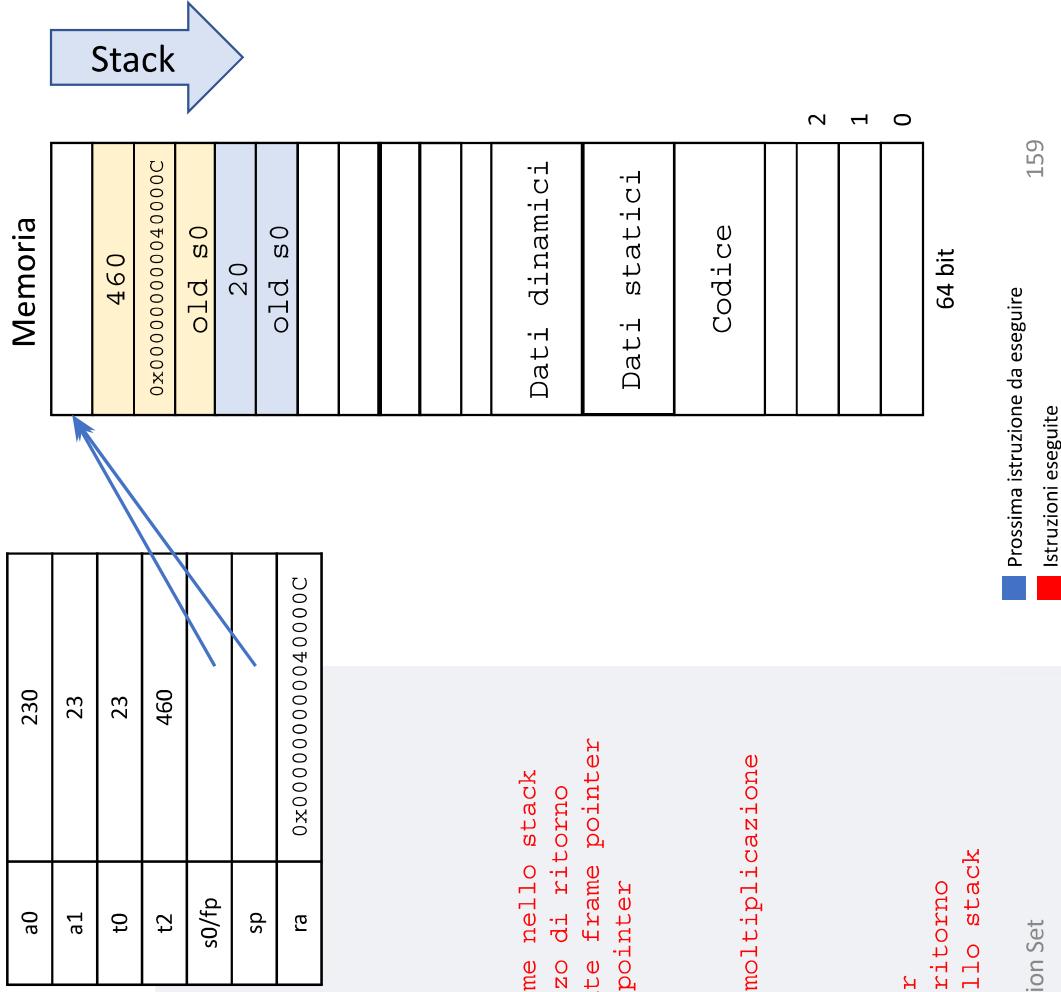
0x0000000000400028 srai a0,1.        #divide per due

# uscita dalla funzione
0x000000000040002C ld s0, 0(sp)      # recupera il frame pointer
0x0000000000400030 ld ra, 8(sp)      # recupera l'indirizzo di ritorno
0x0000000000400034 addi sp, sp, 24    # elimina il call frame dallo stack
0x0000000000400038 jr ra             # ritorna al chiamante
                                         ...

```

Prossima istruzione da eseguire
 Istruzioni eseguite

159



Un esempio

```

_start:          li a0, 20 # salvo la base in a0
                li a1, 23 # salvo l'altezza in a1

# chiama triangolo(base,altezza)
0x0000000000400008 jal ra, area      # altezza in a0, base in a1
add t0, a0, zero      # salva il risultato in t0
0x000000000040000C

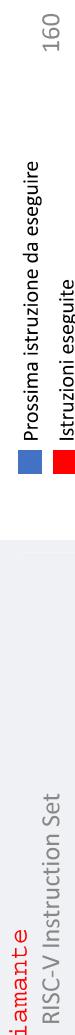
area:
# crea il call frame sullo stack (24 byte=ra+fp+rst)
# lo stack cresce verso il basso
0x0000000000400010 addi sp, -24      # allocazione del frame nello stack
0x0000000000400014 sd ra, 8(sp)      # salvataggio dell'indirizzo di ritorno
0x0000000000400018 sd s0, 0(sp)      # salvataggio del precedente frame pointer
0x000000000040001C addi s0, sp, 16      # aggiornamento del frame pointer

# calcolo dell'area
0x0000000000400020 jal ra, moltiplicazione
0x0000000000400024 sd a0, 0(s0)      #salva il risultato della moltiplicazione
                                         #nella variabile locale rst
                                         #divide per due

0x0000000000400028 srai a0,1.        #divide per due

# uscita dalla funzione
0x000000000040002C ld s0, 0(sp)      # recupera il frame pointer
0x0000000000400030 ld ra, 8(sp)      # recupera l'indirizzo di ritorno
0x0000000000400034 addi sp, sp, 24      # elimina il call frame dallo stack
0x0000000000400038 jr ra             # ritorna al chiamante
                                         ...

```



█ Prossima istruzione da eseguire
█ Istruzioni eseguite

Un esempio

