

Procedure

- L'utilità di un computer sarebbe molto limitata se non si potessero chiamare **procedure** (o **funzioni**)
- Possiamo pensare a una procedura come a una «scatola nera» che esegue un certo task, senza che ne conosciamo necessariamente i dettagli
- La cosa importante è definire chiaramente un protocollo (o convenzione) di chiamata alle procedure

Protocollo

1. Caricare i parametri di input della procedura
in posti noti a priori
2. Trasferire il controllo alla procedura
 1. Acquisire le risorse necessarie
 2. Eseguire il compito
 3. Caricare i valori di ritorno in posti noti
 4. Restituire il controllo al chiamante
3. Prendere il valore di ritorno della procedura e
“cancellare” le tracce (pulire registri, ecc.)

Protocollo

- Il modo in cui questo protocollo viene messo in pratica dipende dall'architettura e dalle convenzioni di chiamata del compilatore
- Ci limitiamo qui a pochi cenni per il RISC-V

Protocollo di chiamata

RISC-V

- L'idea di base è di cercare di usare *laddove possibile* i registri, perché sono il meccanismo più veloce per effettuare il passaggio dei parametri
- Convenzione per il RISC-V:
 - $x_{10}-x_{17}$ usati per i parametri in ingresso e per i valori di ritorno
 - x_1 indirizzo di ritorno per tornare al punto di partenza (chiamato anche *ra*, ovvero «return address»)

Protocollo di chiamata

RISC-V

- L'indirizzo $x1$ viene usato in particolare nelle istruzioni di «jump and link» (**jal**) e «jump and link register» (**jalr**), che effettuano il salto e contemporaneamente memorizzano in $x1$ l'indirizzo di ritorno
 - L'indirizzo che viene salvato in $x1$ è il PC (program counter) + 4 (istruzione successiva alla **jal/jalr**)

Salta e collega `jal x1, 100` $x1 = \text{PC}+4$; vai a $\text{PC}+100$

Salta e collega mediante registro `jalr x1, 100(x5)` $x1 = \text{PC}+4$; vai a $x5+100$

- Alla fine della procedura sarà sufficiente fare un salto:

`jalr x0, 0(x1)`

... e se i registri non bastano?

- In certi casi i registri non bastano perché la procedura ha più parametri di quanti sono i registri
- In tal caso si usa uno *stack* (pila), ovvero una struttura dati gestita in modalità LIFO in cui è possibile caricare, a partire da una posizione nota alla procedura (puntata dal registro $x2$, chiamato anche sp), i parametri in più
- Lo *stack* si può usare anche per caricare variabili locali (tramite un'operazione di *push*) e salvare dei valori di registri che dovranno essere ripristinati in seguito
- Alla fine della procedura si può ripulire lo *stack* (operazione *pop*) riportandolo alla situazione precedente

Esempio

- Consideriamo la procedura in C:

```
long long int esempio_foglia(long long int g,  
    long long int h, long long int i, long long int j) {  
    long long int f;  
    f = (g+h) - (i+j);  
    return f;  
}
```

- Cerchiamo di capire come verrebbe tradotta

Prologo

- Per prima cosa il compilatore sceglie un'etichetta associata all'indirizzo di entrata della procedura (nel nostro caso, *esempio_foglia*)
 - In fase di collegamento (linking) l'etichetta sarà collegata a un'indirizzo
- La prima operazione è quella di salvare in memoria tutti i registri che la procedura usa, in modo da poterli ripristinare in seguito
- Tale fase è chiamata **prologo** e potrebbe richiedere di allocare nello stack anche spazio per le variabili locali (se i registri non bastano)

Uso dei registri

- Parametri in input:
 - $g = x10; h = x11; i = x12; j = x13$
- Variabile locale: $f = x20$
- Usiamo, a titolo di esempio, anche i registri temporanei $x5$ e $x6$
- Supponiamo, sempre a titolo di esempio, di dover salvare $x5$, $x6$ e $x20$

Prologo

- La procedura usa i registri $x5, x6, x20$
- Il registro sp punta alla testa dello stack (che cresce verso il basso)
- Il codice del prologo è il seguente:

```
esempio_foglia: addi sp, sp, -24  
sd x5, 16(sp)  
sd x6, 8(sp)  
sd x20, 0(sp)
```

Decrementiamo sp di 24, per far posto a 3 double word

Salvataggio di $x5$ in $sp+16$

Salvataggio di $x6$ in $sp+8$

Salvataggio di $x20$ in $sp+0$

Esecuzione della procedura

- Dopo il prologo, avviene l'esecuzione della procedura:

```
add x5, x10, x11  
add x6, x12, x13  
sub x20, x5, x6
```

In x5 salviamo g+h

In x6 salviamo i+j

In x20: (g+h)-(i+j)

Epilogo

- Dopo aver eseguito la procedura, si ripristinano i registri usati e si setta il valore di ritorno

```
addi x10, x20, 0  
ld x20, 0(sp)  
ld x6, 8(sp)  
ld x5, 16(sp)  
addi sp, sp, 24  
jalr x0, 0(x1)
```

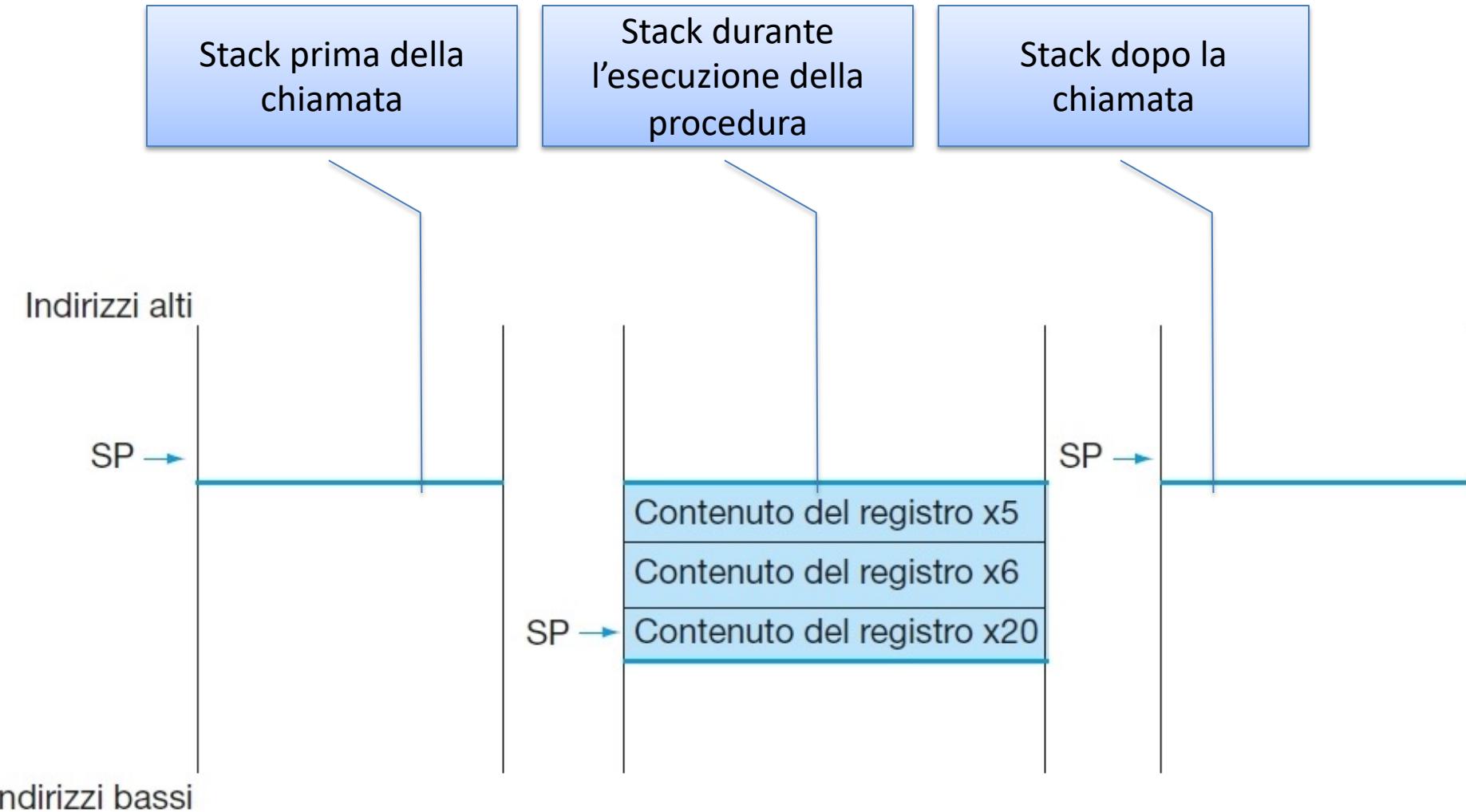
Trasferisco x20 in x10
(valore di ritorno)

Ripristino i registri
prelevando i valori
esattamente da dove
li avevo messi

Ripristino lo stack (tutto
ciò che è sotto sp non è
significativo)

Salto al punto da
dove sono arrivato

Evoluzione dello stack



Un bagno di realtà

- Nessun compilatore farebbe mai questo...
- Infatti
 - I registri temporanei x5 e x6 in realtà **non** devono essere salvati durante la chiamata (avremmo potuto evitare due ld e due sd)
- RISC-V adotta la seguente convenzione:
 - x5-x7 e x28-x31: registri temporanei, che non sono salvati in caso di chiamata a procedura
 - x8-x9 e x18-x27: registri da salvare il cui contenuto deve essere preservato in caso di chiamata a procedura

Ulteriori complicazioni

- In realtà le cose possono complicarsi per via di:
 - Variabili locali
 - Procedure annidate
- Questo si risolve usando sempre lo stack e allocando al suo interno variabili locali e valori del registro di ritorno $x1$
- Vediamo un esempio

Procedura ricorsiva

- Consideriamo il seguente esempio

```
long long int fact(long long int n) {  
    if (n < 1)  
        return 1;  
    else  
        return n*fact(n-1);  
}
```

- Qui (ma sarebbe lo stesso con una qualsiasi funzione che chiama un'altra funzione) abbiamo due problemi:
 - Sovrascrittura di *x1*, che rende impossibile il ritorno una volta finita la chiamata innestata
 - Sovrascrittura di *x10*, usato per il parametro *n*

Procedura ricorsiva

- Soluzione: salvaguardare i registri $x1, x10$
- Vediamo come...
- Primo step: creiamo spazio nello stack e salviamo $x1$ e $x10$

fact:

```
addi sp, sp, -16      # Allociamo spazio per due elementi  
sd x1, 8(sp)          # Salviamo x1  
sd x10, 0(sp)         # Salviamo x10
```

- Secondo step, testiamo se $n < 1$ (ritornando 1 in caso affermativo):

```
addi x5, x10, -1      # Calcoliamo  $x5 = n-1$   
bge x5, x0, L1        # Se  $n-1 \geq 0$ , saltiamo a L1
```

Procedura ricorsiva

- Se $n < 1$, chiudiamo la procedura:

```
addi x10, x0, 1      # Metodo del RISC-V per caricare 1 in x10
addi sp, sp, 16        # Ripuliamo lo stack
jalr x0, 0(x1)        # Ritorniamo al programma chiamante
```

notiamo che non ripristiniamo $x1$ e $x10$ (come si farebbe normalmente), perché, essendo l'ultima chiamata della ricorsione, i loro valori non devono essere più usati (in quanto n e *return address* non cambieranno ulteriormente)

- Se $n \geq 1$, decrementiamo n e richiamiamo fact:

L1:

```
addi x10, x10, -1 # Decrementiamo n (arg. diventa: n-1)
jal x1, fact        # Chiamiamo fact(n-1)
```

Procedura ricorsiva

- A questo punto ripristiniamo x_{10} e x_1 e ripuliamo lo stack

```
addi x6, x10, 0    # Carichiamo risultato di fact(n-1) in x6
ld x10, 0(sp)      # Ripristino dell'argomento n, x10
ld x1, 8(sp)        # Ripristino del reg. di ritorno, x1
addi sp, sp, 16     # Ripuliamo lo stack
```

- Non ci resta che moltiplicare $\text{fact}(n-1)$, che è in x_6 , per n , che è in x_{10} , e ritornare

```
mul x10, x10, x6      # Restituiamo  $n * \text{fact}(n-1)$ 
jalr x0, 0(x1)        # Ritorniamo al programma chiamante
```

Storage class

- Le variabili in C sono in genere associate a locazioni di memoria, caratterizzate per:
 - Tipo (int, char, float, ecc.)
 - Storage class
- Il C ha due possibili storage class
 - Automatic: variabili locali che hanno un ciclo di vita collegato alla funzione
 - Static: sopravvivono alle chiamate di procedura. Sono essenzialmente variabili globali, oppure variabili definite esplicitamente come static all'interno di una procedura.
- Le variabili statiche sono memorizzate in una zona di memoria specifica
 - Nel RISC-V, questa zona è accessibile attraverso il registro x3, chiamato anche *gp* («global pointer»)

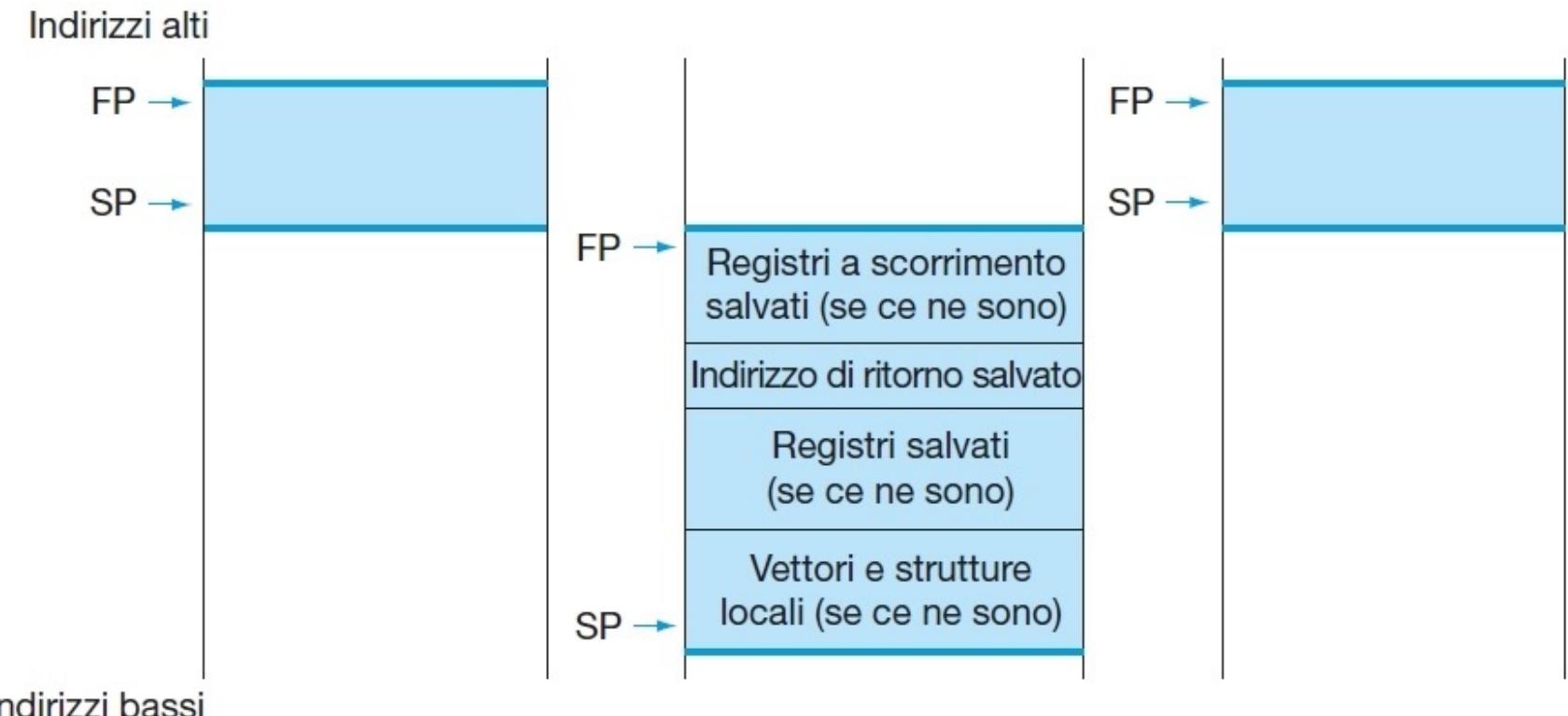
Variabili locali

- L'ultimo problema è costituito dalle variabili locali
- Quando sono poche, si riescono a usare registri
- Quando sono tante (ad esempio struct complesse o array) non ci sono abbastanza registri
- Quello che si fa è allocare le variabili locali nello stack (come abbiamo già visto)

Record di attivazione

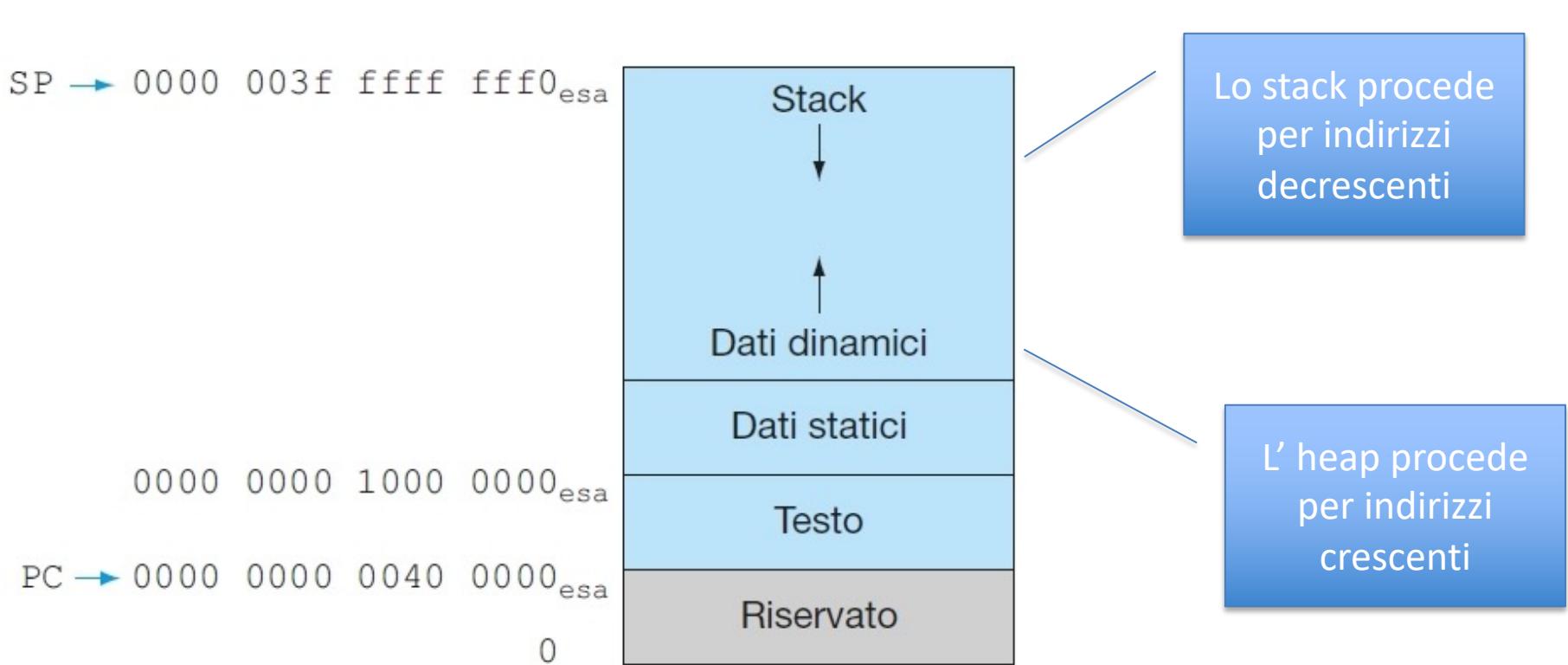
- Il segmento di stack che contiene registri salvati e variabili locali relative a una procedura viene chiamato **record di attivazione** (o stack frame)
- Le variabili locali vengono individuate tramite un offset a partire da un puntatore
- Il registro *sp* è alle volte scomodo come base, in quanto come abbiamo visto cambia (cresce verso il basso) durante l'esecuzione della procedura
- Alcuni programmi RISC-V utilizzano il registro *x8*, chiamato anche *fp*, come «frame pointer», ovvero puntatore alla prima parola doppia dello stack frame di una procedura (mantenuto costante durante la funzione, per cui può essere usato come base)

Evoluzione dello stack



Dati dinamici

- In aggiunta a variabili globali e variabili locali, abbiamo anche le variabili dinamiche (quelle che in C/C++ vengono allocate con chiamate malloc/free, new/delete), salvate nell'«heap» (letteralmente, «cumulo»)
- Il RISC-V adotta il seguente schema di memoria:



Convenzioni sui registri

- In base alle convenzioni definite dal RISC-V, l'uso dei registri può essere riassunto come segue:

Nome	Numero del registro	Utilizzo	Da conservare nella chiamata?
x0	0	Costante 0	n.a.
x1 (ra)	1	Registro di ritorno (registro di collegamento)	sì
x2 (sp)	2	Stack pointer	sì
x3 (gp)	3	Global pointer	sì
X4 (tp)	4	Thread pointer	sì
x5-x7	5-7	Variabili temporanee	no
x8-x9	8-9	Variabili da preservare	sì
x10-x17	10-17	Argomenti/risultati	no
x18-x27	18-27	Variabili da preservare	sì
x28-x31	28-31	Variabili temporanee	no

Elaborazione

- Per quanto le ricorsioni siano eleganti, spesso sono causa di varie inefficienze

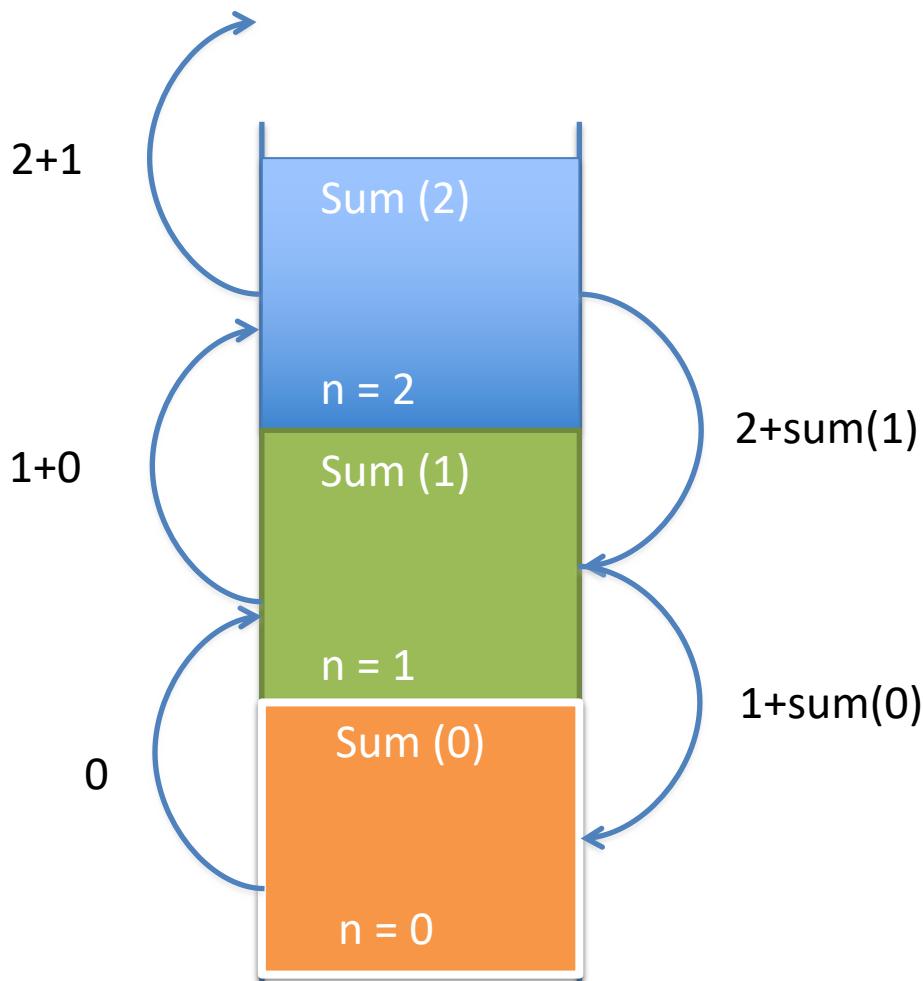
Never hire a programmer who solves the factorial with a recursion!

- Consideriamo la seguente funzione:

```
long long int sum(long long int n) {  
    if (n > 0)  
        return sum(n-1) + n;  
    else  
        return n;  
}
```

Elaborazione

- Consideriamo la chiamata $\text{sum}(2)$



Il valore di ritorno viene costruito tornando su per la catena di chiamate. Ogni frame di attivazione deve essere preservato per produrre il risultato corretto.

Elaborazione

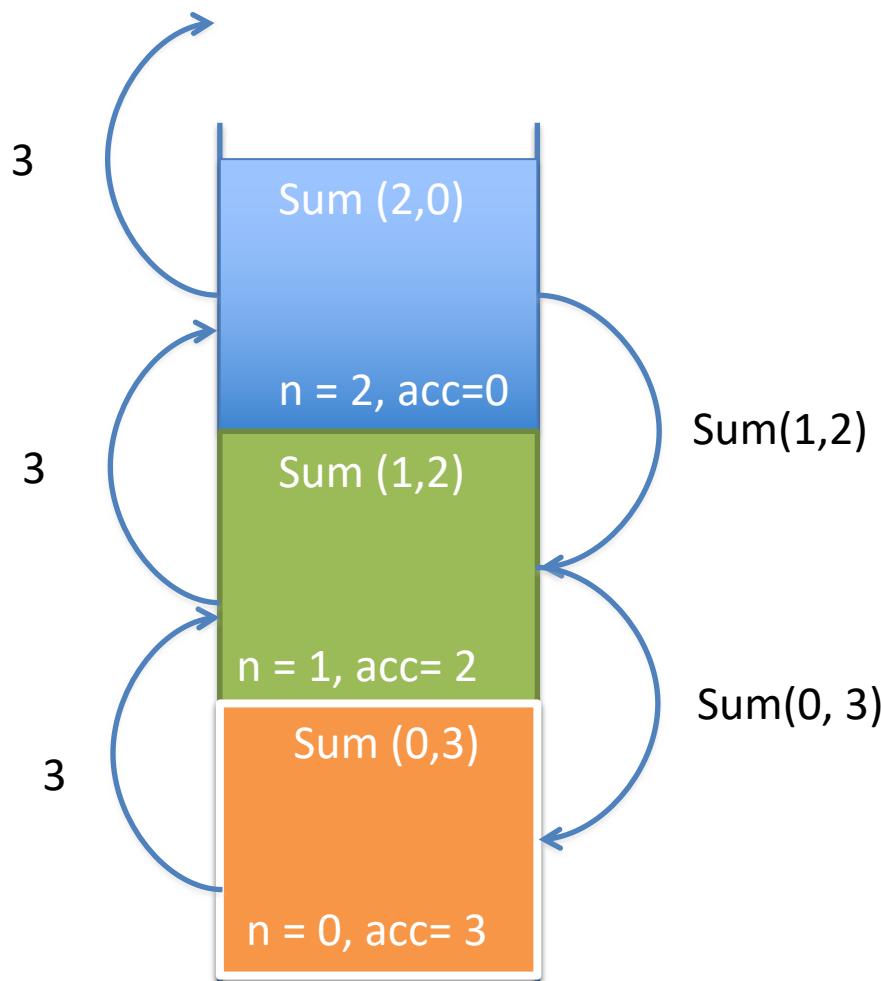
- Consideriamo ora il codice così modificato

```
long long int sum(long long int n, long long int acc) {  
    if (n > 0)  
        return sum(n-1, acc+n);  
    else  
        return acc;  
}
```

- Stavolta la chiamata ricorsiva è l'ultima cosa che la procedura fa (si limita a ritornare)
- Questa ricorsione viene detta «di coda»
- Ritornando al nostro esempio, questo diventa sum(2,0)

Elaborazione

- Consideriamo la chiamata $\text{sum}(2,0)$



In questo caso lungo i rami di ritorno non faccio che restituire 3 (valore di ritorno della chiamata). Conservare i dati nel frame di attivazione è inutile. Posso usare un unico frame di attivazione e svolgere la ricorsione in iterazione.

Codice

- La procedura ricorsiva viene compilata come segue segue:

sum:

```
add x5, x0, x11      # Sposta x11 in x5
blt x10, x0, L5      # Se x10 < 0, salta a L5

addi sp, sp, -8       # Prologo: crea spazio nello stack
sd x1, 0(sp)          # Prologo: salva x1
add x11, x11, x10     # Somma x10 ad x11 (acc=acc+n)
addi x10, x10, -1     # Decrementa x10 (n=n-1)
jal x1, sum           # Ricorsione (modifica x1)

ld x1, 0(sp)          # Ripristina x1
addi sp, sp, 8         # Rimetti a posto lo stack
```

L5:

```
add x10, x0, x5        # sposta x5 in x10 (valore ritorno)
jalr x0, 0(x1)
```

Ottimizzazione

- Alcuni compilatori sanno riconoscere la ricorsione di coda:
 - Il chiamante ritorna subito dopo la *jal*
 - *x5* e gli altri registri non cambiano
 - Il chiamato potrebbe direttamente ritornare al *ra* del chiamante
 - In altre parole il chiamante salva *ra* e lo recupera per nulla
- Torniamo al nostro esempio della somma...

Ottimizzazione

- Quello che potremmo fare guardando al codice è molto semplicemente osservare che possiamo sostituire la **jalr** con un salto incondizionato a sum, con grandi semplificazioni

sum:

```
add x5, x0, x11      # Sposta x11 in x5
blt x10, x0, L5       # Se x10 < 0, salta a L5

addi sp, sp, -8      # Prologo: crea spazio nello stack
sd x1, 0(sp)         # Prologo: salva x1
add x11, x11, x10     # Somma x10 ad x11 (acc=acc+n)
addi x10, x10, -1      # Decrementa x10 (n=n-1)
jal x0, sum            # Ricorsione (NON modifica x1)

ld x1, 0(sp)          # Ripristina x1
addi sp, sp, 8         # Rimetti a posto lo stack
```

L5:

```
add x10, x0, x5        # sposta x5 in x10 (valore ritorno)
jalr x0, 0(x1)
```

Ottimizzazione

- Il codice risultante è più corto e anche più efficiente
- Il gcc fa questo con -O2

sum:

```
add x5, x0, x11      # Sposta x11 in x5
blt x10, x0, L5       # Se x10 < 0, salta a L5

add x11, x11, x10     # Somma x10 ad x11 (acc=acc+n)
addi x10, x10, -1     # Decrementa x10 (n=n-1)
beq x0, x0, sum       # Ricorsione (NON modifica x1)
```

L5:

```
add x10, x0, x5       # sposta x5 in x10 (valore ritorno)
jalr x0, 0(x1)
```