



**d1.unito.it**

DIPARTIMENTO  
DI INFORMATICA

DI INFORMATICA  
DIPARTIMENTO

laboratorio di architettura  
degli elaboratori

**supporto hardware alle  
procedure**

Daniele Radicioni

chiamata di funzione

## procedure o funzioni

- strumento per strutturare i programmi per renderli più semplici da capire, e per permettere di riutilizzare il codice.
- operazioni effettivamente condotte dal programma per l'esecuzione di funzioni:
  - 1.mettere parametri in posti in cui la procedura li possa accedere
  - 2.trasferire il controllo alla procedura
  - 3.acquisire le risorse di storage necessarie per la procedura
  - 4.svolgere il compito assegnato
  - 5.mettere risultato in un posto in cui il chiamante lo possa accedere
  - 6.restituire il controllo al punto di partenza (una procedura può essere invocata da vari punti all'interno del programma)

## parametri e indirizzo di ritorno

- registri x10-x17 (a0–a7) sono 8 **registri per parametri**, utilizzati cioè per passare valori alle funzioni o restituire valori al chiamante;
- registro x1 (ra) contiene l'indirizzo di ritorno

## jal e jalr: passaggio di controllo

- l'istruzione **jal** (**jump and link**) serve per la chiamata di funzioni: produce un salto a un indirizzo e salva l'indirizzo dell'istruzione successiva a quella del salto nel registro `ra` (indirizzo di ritorno, detto appunto *link*):

```
jal x1, ProcAddress # salta a ProcAddress e salva indirizzo di ritorno in x1  
[jal ProcAddress]
```

- il ritorno da una procedura utilizza un salto indiretto, **jump and link register (jalr)**

```
jalr x0, 0(x1) # salta indietro all'indirizzo di ritorno presente in x1  
[jr ra]
```

- lo schema è quindi il seguente:
  - funzione **chiamante** mette i parametri in `x10–x17` e usa `jal x` per saltare alla funzione `X`
  - funzione **chiamata** svolge le proprie operazioni, inserisce i risultati negli stessi registri e restituisce il controllo al chiamante con l'istruzione `jr ra`.

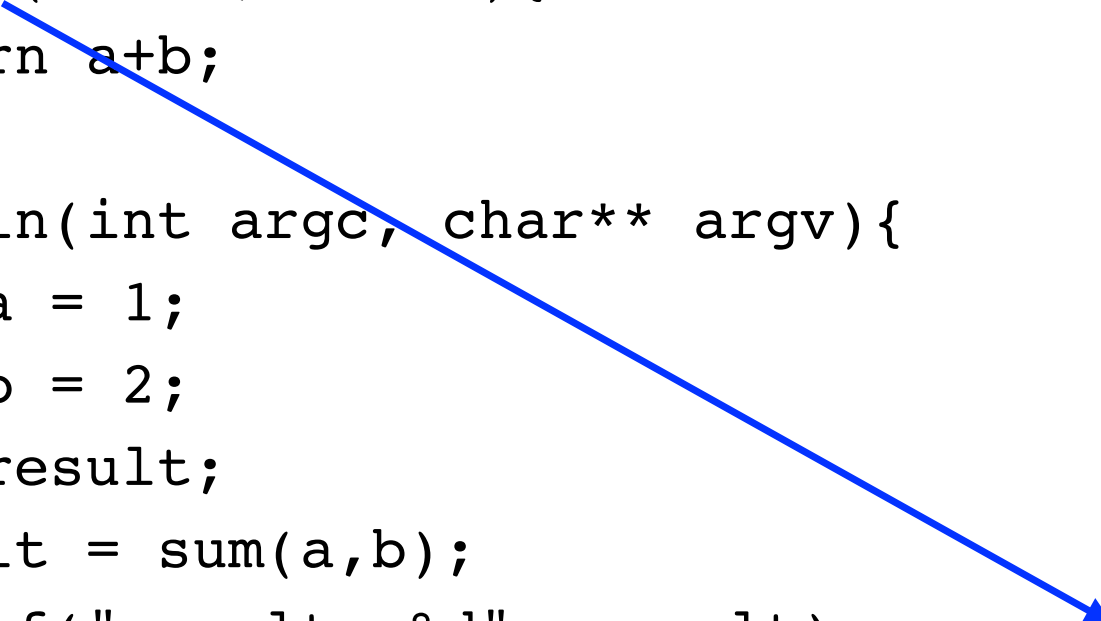
## primo esempio

```
int sum(int a, int b){  
    return a+b;  
}  
  
int main(int argc, char** argv){  
    int a = 1;  
    int b = 2;  
    int result;  
    result = sum(a,b);  
    printf("result: %d", result);  
    exit(0);  
}
```

## primo esempio

```
int sum(int a, int b){  
    return a+b;  
}  
  
int main(int argc, char** argv){  
    int a = 1;  
    int b = 2;  
    int result;  
    result = sum(a,b);  
    printf("result: %d", result);  
    exit(0);  
}
```

```
_start:  
    li a0, 5 # a  
    li a1, 2 # b  
  
    jal sum  
  
    li a7, 1  
    ecall  
    # manca exit...  
  
sum:  
    add a0, a0, a1  
    jr ra
```



## primo esempio

**registri per passaggio parametri  
(FUNZ. CHIAMANTE)**

```
int main(int argc, char** argv){  
    int a = 1;  
    int b = 2;  
    int result;  
    result = sum(a,b);  
}
```

**registri per passaggio parametri  
(FUNZ. CHIAMATA)**

**start:**

```
li a0, 5 # a  
li a1, 2 # b
```

**jal sum**

```
li a7, 1  
ecall  
# manca exit...
```

**sum:**

```
add a0, a0, a1  
jr ra
```



## primo esempio

```
int sum(int a, int b){  
    return a+b;  
}
```

**# ra= memAddress, goto sum**

```
int a = 1;  
int b = 2;  
int result;  
result = sum(a,b);  
printf("result: %d", result);  
exit(0);  
}
```

**\_start:**

li **a0**, 5 # a

li **a1**, 2 # b

**jal sum**

li a7, 1

ecall

*# manca exit...*

**sum:**

add a0, **a0**, **a1**

**jr ra**

nuova istruzione: **jr, jump register**

**domanda:** perché usiamo **jr**, e non semplicemente **j**?

**risposta:** perché la funzione può essere chiamata da molti punti del programma, e c'è quindi bisogno di un **meccanismo per tornare all'istruzione successiva alla chiamata, che può essere ovunque**: serve quindi un meccanismo più flessibile, che tenga conto dell'indirizzo (dell'istruzione successiva a quella di partenza): questo è **jr ra**.

```
_start:
    li a0, 5 # a
    li a1, 2 # b

    jal sum

    li a7, 1
    ecall
    # manca exit...

sum:
    add a0, a0, a1
    jr ra
```

# primo esempio

Text Segment			
Address	Code	Basic	Source
4194304	0x00500513	addi x10,x0,5	10: li a0, 5 # a
4194308	0x00200593	addi x11,x0,2	11: li a1, 2 # b
4194312	0x00c000ef	jal x1,12	13: jal sum
4194316	0x00100893	addi x17,x0,1	15: li a7, 1
4194320	0x00000073	ecall	16: ecall
4194324	0x00b50533	add x10,x10,x11	19: add a0, a0, a1
4194328	0x00008067	jalr x0,x1,0	20: jr ra

stato prima di eseguire jal sum:

- pc vale 4194312 e
- ra vale 0

\_start:

li a0, 5 # a

li a1, 2 # b

jal sum



li a7, 1

ecall

# manca exit...

sum:

add a0, a0, a1

jr ra

# primo esempio

Text Segment			
Address	Code	Basic	Source
4194304	0x00500513	addi x10,x0,5	10: li a0, 5 # a
4194308	0x00200593	addi x11,x0,2	11: li a1, 2 # b
4194312	0x00c000ef	jal x1,12	13: jal sum
4194316	0x00100893	addi x17,x0,1	15: li a7, 1
4194320	0x00000073	ecall	16: ecall
4194324	0x00b50533	add x10,x10,x11	19: add a0, a0, a1
4194328	0x000008067	jalr x0,x1,0	20: jr ra

stato dopo salto a sum:

- pc vale 4194324 e
- ra vale 4194316

\_start:

li a0, 5 # a

li a1, 2 # b

jal sum

li a7, 1

ecall

sum:

add a0, a0, a1 ←

jr ra

salvataggio sullo stack del contenuto  
dei registri

## funzionamento di sp

- altra questione, distinta dal meccanismo di chiamata e passaggio di parametri (seppure sempre collegata alla chiamata a funzione), è quella dell'**utilizzo dello stack nei passaggi di dati fra registri e memoria.**

# stack pointer

- nel caso servano più degli 8 registri a0–a7 (x10–x17), i registri possono essere copiati in memoria (**register spilling**)
- la struttura dati utilizzata a questo fine è lo stack, contenente un puntatore al registro allocato più di recente per mostrare dove la procedura successiva dovrebbe memorizzare i registri e da dove si possa recuperare il vecchio valore.
- lo **stack pointer (sp)** è x2; questo viene incrementato o decrementato di una parola ogni volta che viene inserito o tolto il contenuto di un registro.
  - stack 'cresce' da indirizzi di memoria alti verso indirizzi di memoria bassi (quindi **quando vengono inseriti dati nello stack il valore dello sp diminuisce**, e cresce quando i dati sono estratti dallo stack).

# tutto il codice dell'esempio che segue...

si trova nei files `esempio_sp.c` e in `esempio_sp.asm`, fra il materiale della lezione

- l'esempio seguente considera il caso di un registro che viene copiato su stack, riutilizzato (assume quindi altro valore) durante l'esecuzione della procedura chiamata, e ripristinato prima dell'uscita dalla procedura e del ritorno al chiamante.



```
int main(){  
    int a = 3;  
    int b = 4;  
    int result;  
    result = multiply(a,b);  
    printf("res: %d\n", result);  
    exit(0);  
}
```

chiamata di funzione



```
int multiply(int a, int b){  
    int i = 0;  
    int acc = 0;  
    while(i < b){  
        acc += a;  
        ++i;  
    }  
    return acc;  
}
```

```
.globl _start
```

```
.text
```

```
_start:
```

```
li a0, 3    # a = 3
```

```
li a1, 4    # b = 4
```

```
# teniamo d'occhio s0, disponibile nell'ambiente del chiamante e con valore  
# 13: dopo essere stato salvato su stack, sarà utilizzato dal chiamato come  
# accumulatore; poi il valore salvato su stack sarà recuperato (restored),  
# per essere nuovamente disponibile nell'ambiente del chiamante
```

```
li s0, 13
```

```
jal multiply
```

```
printresult:
```

```
li a7, 1
```

```
# a0 contiene già il risultato, quindi non è da assegnare
```

```
ecall
```

```
exit:
```

```
li a7, 10
```

```
ecall
```

Text Segment			
Address	Code	Basic	Source
4194304	0x00300513	addi x10,x0,3	7: li a0, 3 # a = 3
4194308	0x00400593	addi x11,x0,4	8: li a1, 4 # b = 4
4194312	0x00d00413	addi x8,x0,13	14: li s0, 13
4194316	0x014000ef	jal x1,20	16: jal multiply
4194320	0x00100893	addi x17,x0,1	19: li a7, 1
4194324	0x00000073	ecall	21:
4194328	0x00a00893	addi x17,x0,10	24: .globl _start
4194332	0x00000073	ecall	25: .text
4194336	0xffc10113	addi x2,x2,-4	29: _start:
4194340	0x00812023	sw x8,0(x2)	30:
4194344	0x00000293	addi x5,x0,0	33: li a0, 3 # a = 3
4194348	0x00000413	addi x8,x0,0	34: li a1, 4 # b = 4

*# teniamo d'occhio s0, disponibile nell'ambiente del chiamante e con valore  
# 13: dopo essere stato salvato su stack, sarà utilizzato dal chiamato come  
# accumulatore; poi il valore salvato su stack sarà recuperato (restored)  
# per essere nuovamente disponibile nell'ambiente del chiamante*  
li s0, 13

stato prima di eseguire jal  
multiply:

- pc vale 4194316,
- ra vale 0,
- s0 vale 13.

jal multiply

printresult:

li a7, 1

*# a0 contiene già il risultato, quindi non è da assegnare*

ecall

exit:

li a7, 10

ecall

*multiply:*

```
addi sp, sp, -4 # creo spazio per 1 item. NB: STACK CRESCE VERSO IL BASSO!!!  
sw s0, 0(sp)    # salvo il contenuto di s0
```

```
# abbiamo a disposizione a0 e a1, settati dal chiamante
```

```
addi t0, zero, 0 # int i = 0;
```

```
addi s0, zero, 0 # acc = 0
```

*while:*

```
beq t0, a1, endwhile # while(i < b)
```

```
add s0, s0, a0        # acc += a;
```

```
addi t0, t0, 1
```

```
j while
```

*endwhile:*

```
add a0, zero, s0 # prima di fare il restore di s0 copio il suo valore in a0
```

```
lw s0, 0(sp) # restore dei registri per il chiamante
```

```
addi sp, sp, 4 # spostiamo sp e cancelliamo l'elemento
```

```
jr ra
```

Text Segment			
Address	Code	Basic	Source
4194336	0xffc10113	addi x2,x2,-4	29: addi sp, sp, -4 # creo spazio per 1 item. NB: STACK CRESCE
4194340	0x00812023	sw x8,0(x2)	30: sw s0, 0(sp) # salvo il contenuto di s0
4194344	0x00000293	addi x5,x0,0	33: addi t0, zero, 0 # int i = 0;
4194348	0x00000413	addi x8,x0,0	34: addi s0, zero, 0 # acc = 0
4194352	0x00b28863	beq x5,x11,16	36: beq t0, a1, endwhile # while(i < b)
4194356	0x00a40433	add x8,x8,x10	37: add s0, s0, a0 # acc += a;
4194360	0x00128293	addi x5,x5,1	
4194364	0xff5ff06f	jal x0,-12	
4194368	0x00800533	add x10,x0,x8	
4194372	0x00012403	lw x8,0(x2)	
4194376	0x00410113	addi x2,x2,4	
4194380	0x00008067	ialr x0.x1.0	

a questo punto s0 vale 0.

a questo punto s0 viene riassegnato con il valore iniziale

```

multiply:
    addi sp, sp, -4 # creo spazio per 1 item. NB: STACK CRESCE VERSO I
    sw s0, 0(sp)    # salvo il contenuto di s0

    # abbiamo a disposizione a0 e a1, settati dal chiamante
    addi t0, zero, 0 # int i = 0;
    addi s0, zero, 0 # acc = 0

while:
    beq t0, a1, endwhile # while(i < b)
    add s0, s0, a0        # acc += a;

    addi t0, t0, 1
    j while

endwhile:
    add a0, zero, s0 # prima di fare il restore di s0 copio il suo val
    lw s0, 0(sp)    # restore dei registri per il chiamante
    addi sp, sp, 4   # spostiamo sp e cancelliamo l'elemento
    jr ra
  
```

procedure annidate

procedure che chiamano altre procedure

```
int main(){
    int a = 3;
    int b = 4;
    int result;
    result = multiply(a,b);
    printf("res: %d\n", result);
    exit(0);
}
```

chiamata di funzione



```
int multiply(int a, int b){
    int i = 0;
    int acc = 0;
    while(i < b){
        acc = sum(acc,a);
        ++i;
    }
    return acc;
}
```

```
int main(){
    int a = 3;
    int b = 4;
    int result;
    result = multiply(a,b);
    printf("res: %d\n", result);
    exit(0);
}
```

chiamata di funzione

```
int multiply(int a, int b){
    int i = 0;
    int acc = 0;
    while(i < b){
        acc = sum(acc,a);
        ++i;
    }
    return acc;
}
```

chiamata di funzione

```
int sum (int primo, int secondo) {
    return primo + secondo;
}
```



# chiamate annidate

- PROBLEMA: sovrascrittura dei valori nei registri a0–a7 e in ra.
  - nel momento in cui iniziamo ad eseguire `multiply` ra viene assegnato con un valore riferito al chiamante (il `main`, nel nostro caso), e quando `multiply` chiama `sum`, ra viene sovrascritto con il ritorno relativo alla procedura `multiply`...
- dobbiamo quindi **salvare il primo indirizzo di ritorno (al `main`) prima di chiamare `sum`.**

```
int multiply(int a, int b){  
    int i = 0;  
    int acc = 0;  
    while(i < b){  
        acc = sum(acc, a);  
        ++i;  
    }  
    return acc;  
}
```

## chiamate annidate

- in generale può essere necessario salvare anche altre informazioni oltre a `ra`.
- le **aree di memoria** rilevanti per l'esecuzione di un programma in C sono infatti:
  - **static**: sono le variabili globali, dichiarate una sola volta all'interno del programma, che continuano ad essere raggiungibili durante il corso di tutta l'esecuzione;
  - **heap**: variabili dichiarate dinamicamente (tramite funzione `malloc`);
  - **stack**: è lo spazio che le procedure utilizzano durante la propria esecuzione. è qui che possiamo salvare i valori dei registri.

# convenzione

- per evitare costose operazioni di spilling (salvataggio su stack) e di restore (ri-salvataggio da stack a registri) utilizziamo una convenzione. dividiamo i registri in 2 categorie:
  - quelli **preservati nel passaggio fra chiamate di funzione**:
    - in questi casi il chiamante si attende che i valori non siano alterati
    - appartengono a questa classe `sp`, `gp`, `tp`, e i `saved registers s0–s11` (dove `s0` è il frame pointer, `fp`).
  - quelli **non preservati** fra le chiamate:
    - `a0–a7`, `ra`, e i registri temporanei `t0–t6`.

# Convenzioni di chiamata

Nome	Utilizzo	Chi lo salva
zero	La costante 0	N.A.
ra	Indirizzo di ritorno	Chiamante
sp	Puntatore a stack	Chiamato
gp	Puntatore globale	---
tp	Puntatore a thread	---
t0-t2	Temporanei	Chiamante
s0/_fp	Salvato/puntatore a frame	Chiamato
s1	Salvato	Chiamato
a0-a1	Argomenti di funzione/valori restituiti	Chiamante
a2-a7	Argomenti di funzione	Chiamante
s2-s11	Registri salvati	Chiamato
t3-t6	Temporanei	Chiamante

- **Register Spilling:**  
Trasferire variabili da registri a memoria.
- I registri sono più veloce che la memoria, quindi vogliamo **evitare "register spilling"**
- Usiamo lo **stack** per questa operazione

## Convenzioni di chiamata - **chiamante**

Nome	Utilizzo	Chi lo salva
zero	La costante 0	N.A.
ra	Indirizzo di ritorno	Chiamante
sp	Puntatore a stack	Chiamato
gp	Puntatore globale	---
tp	Puntatore a thread	---
t0-t2	Temporanei	Chiamante
s0/_fp	Salvato/puntatore a frame	Chiamato
s1	Salvato	Chiamato
a0-a1	Argomenti di funzione/valori restituiti	Chiamante
a2-a7	Argomenti di funzione	Chiamante
s2-s11	Registri salvati	Chiamato
t3-t6	Temporanei	Chiamante

**Sempre**

## Convenzioni di chiamata - **chiamante**

Nome	Utilizzo	Chi lo salva
zero	La costante 0	N.A.
ra	Indirizzo di ritorno	Chiamante
sp	Puntatore a stack	Chiamato
gp	Puntatore globale	---
tp	Puntatore a thread	---
t0-t2	Temporanei	Chiamante
s0/_fp	Salvato/puntatore a frame	Chiamato
s1	Salvato	Chiamato
a0-a1	Argomenti di funzione/valori restituiti	Chiamante
a2-a7	Argomenti di funzione	Chiamante
s2-s11	Registri salvati	Chiamato
t3-t6	Temporanei	Chiamante

**Se servono  
al chiamante**

**Se servono  
al chiamante**

## Convenzioni di chiamata - **chiamante**

Nome	Utilizzo	Chi lo salva
zero	La costante 0	N.A.
ra	Indirizzo di ritorno	Chiamante
sp	Puntatore a stack	Chiamato
gp	Puntatore globale	---
tp	Puntatore a thread	---
t0-t2	Temporanei	Chiamante
s0/_fp	Salvato/puntatore a frame	Chiamato
s1	Salvato	Chiamato
a0-a1	Argomenti di funzione/valori restituiti	Chiamante
a2-a7	Argomenti di funzione	Chiamante
s2-s11	Registri salvati	Chiamato
t3-t6	Temporanei	Chiamante

**Se ci sono  
parametri e  
valori di ritorno**

## Convenzioni di chiamata - **chiamato**

Nome	Utilizzo	Chi lo salva
zero	La costante 0	N.A.
ra	Indirizzo di ritorno	Chiamante
sp	Puntatore a stack	Chiamato
gp	Puntatore globale	---
tp	Puntatore a thread	---
t0-t2	Temporanei	Chiamante
s0/_fp	Salvato/puntatore a frame	Chiamato
s1	Salvato	Chiamato
a0-a1	Argomenti di funzione/valori restituiti	Chiamante
a2-a7	Argomenti di funzione	Chiamante
s2-s11	Registri salvati	Chiamato
t3-t6	Temporanei	Chiamante

**add/sub  
sempre lo stesso  
numero di byte**



## Convenzioni di chiamata - **chiamato**

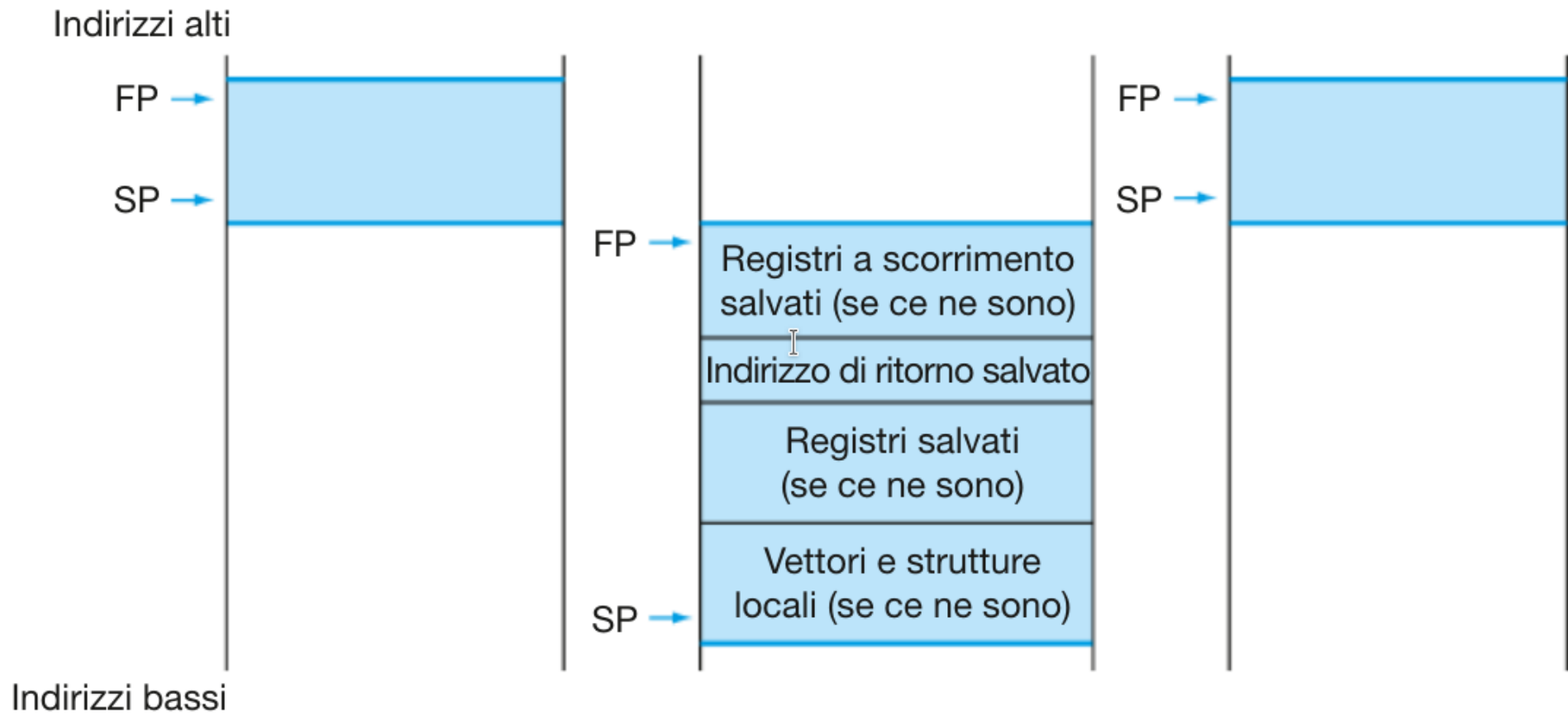
Nome	Utilizzo	Chi lo salva
zero	La costante 0	N.A.
ra	Indirizzo di ritorno	Chiamante
sp	Puntatore a stack	Chiamato
gp	Puntatore globale	---
tp	Puntatore a thread	---
t0-t2	Temporanei	Chiamante
s0/_fp	Salvato/puntatore a frame	Chiamato
s1	Salvato	Chiamato
a0-a1	Argomenti di funzione/valori restituiti	Chiamante
a2-a7	Argomenti di funzione	Chiamante
s2-s11	Registri salvati	Chiamato
t3-t6	Temporanei	Chiamante

**quando vengono  
usati**

# allocazione di spazio sullo stack

- il C (come altri linguaggi) ha **due classi di memorizzazione**:
  - **automatic**: variabili **locali alle funzioni**, che vengono distrutte all'uscita dalla funzione;
  - **static**: variabili che continuano ad esistere durante le varie chiamate fra procedure.
- usiamo lo stack per memorizzare le variabili che eccedono la capacità dei registri;
- nozione di **frame della procedura** o **record di attivazione**: segmento dello stack contenente i registri salvati e le variabili locali alla procedura.

# Stack



- Se lo stack **non contiene variabili locali** alla procedura, il compilatore risparmia tempo di esecuzione **evitando di impostare e ripristinare il frame**.
- Quando viene utilizzato, **FP** viene inizializzato con **l'indirizzo** che ha **SP** all'atto della chiamata della procedura e **SP** viene ripristinato al termine della procedura utilizzando il valore di **FP**

esercizi

## Lab 6 - Esercizio 1 - MCD(a,b)

- Scrivere una procedura RISC-V per il calcolo del massimo comune divisore di due numeri interi positivi a e b. A tale scopo implementare l'algoritmo di Euclide come metodo MCD(a,b) da richiamare nel main.
- Quante istruzioni RISC-V sono necessarie per implementare la funzione?
- Quante istruzioni RISC-V verranno eseguite per completare la funzione quando a=24, b=30?

```
// algoritmo di Euclide per MCD
int MCD(int a, int b) {
    while (a != b)
        if (a > b)
            a = a - b;
        else
            b = b - a;
    return a;
}

void main() {
    int a = 24;
    int b = 30;
    int result;

    result = MCD(a,b);
    printf("%d\n", result);
}
```

## Lab 6 - Esercizio 1 - MCD(a,b)

```
# a0 -> a
# a1 -> b
# return MCD su a0
mcd:
```

```
mcd_while:
    beq    a0, a1, mcd_end
    bge    a1, a0, mcd_else
    sub    a0, a0, a1
    j      mcd_while

mcd_else:
    sub    a1, a1, a0
    j      mcd_while

mcd_end:

    ret
```

```
int MCD(int a, int b) {
    while (a != b)
        if (a > b)
            a = a - b;
        else
            b = b - a;
    return a;
}
```

## Lab 6 - Esercizio 1 - MCD(a,b)

```
# a0 -> a
# a1 -> b
# return MCD su a0
mcd:
```

```
mcd_while:
    beq    a0, a1, mcd_end
    bge    a1, a0, mcd_else
    sub    a0, a0, a1
    j      mcd_while
mcd_else:
    sub    a1, a1, a0
    j      mcd_while
mcd_end:

    ret
```

```
void main() {
    int a = 24;
    int b = 30;
    int result;

    result = MCD(a,b);
    printf("%d\n", result);
}
```

```
_start:
    li     a0, 24
    li     a1, 30
    jal     ra, mcd
    mv      t0, a0

print:
    addi    a0, t0, 0
    li      a7, 1
    ecall
```

## Lab 6 - Esercizio 2 - mcm(a,b)

- Scrivere una procedura RISC-V per il calcolo del minimo comune multiplo di due numeri interi positivi a e b, MCM(a,b), da richiamare nel main, utilizzando la seguente relazione:

$$\text{mcm}(a, b) = (a * b) / \text{MCD}(a, b)$$

- È possibile realizzare la funzione senza versare registri in memoria?
- Quante istruzioni RISC-V sono necessarie per implementare la procedura?
- Quante istruzioni RISC-V verranno eseguite per completare la procedura quando a=12, b=9?



```

.globl _start
.data
.text
_start:
    ...
    ...

    jal mcm

print_exit_0:
    li a7, 1
    ecall
    li a7, 10
    ecall

```

```

# mcm(a,b) = (a*b) / MCD(a,b)
mcm:
    ...    # estendo stack
    ...    # salvo sp

    ...

    jal mcd
    # a0 contiene mcd(a,b)
    ...

    ... # restore ra
    ... # riposiziono sp

    ...

```

```
# mcm(a,b) = (a*b) / MCD(a,b)
```

```
mcm:
```

```
... # estendo stack
```

```
... # salvo sp
```

```
...
```

```
jal mcd
```

```
# a0 contiene mcd(a,b)
```

```
...
```

```
... # restore ra
```

```
... # riposiziono sp
```

```
...
```

```
# a0 -> a
```

```
# a1 -> b
```

```
# return MCD su a0
```

```
mcd:
```

```
    mcd_while:
```

```
        beq a0, a1, mcd_end
```

```
        blt a1, a0, else
```

```
        sub a1, a1, a0
```

```
        j mcd_while
```

```
    else:
```

```
        sub a0, a0, a1
```

```
        j mcd_while
```

```
    mcd_end:
```

```
        jr ra
```

## Lab 6 - Esercizio 3 – strlen (String Length)

- Scrivere una procedura RISC-V per calcolare la lunghezza di una stringa in C, escluso il carattere terminatore. Le string in C sono memorizzate come un array di byte in memoria, dove il byte `'\0'` (`0x00`) rappresenta il fine della string.

```
unsigned long strlen(char *str) {  
    unsigned long i;  
    for (i = 0; str[i] != '\0'; i++);  
    return i;  
}
```

```
.globl _start  
.data  
    src: .string "This is the source string."
```

## Lab 6 - Esercizio 4 – strcpy (String Copy)

Scrivere una procedura RISC-V per copiare una stringa ad un'altra (strcpy).  
Assumere che `dst` abbia spazio sufficiente in memoria per ricevere i byte di `src`,  
`strlen(dst) >= strlen(src)`

**Nota:** strcpy deve utilizzare strlen, come in questo codice in C:

```
void strncpy(char *dst, char *src) {  
    unsigned long i;  
    unsigned long n;  
    n = strlen(src);  
    m = strlen(dst);  
    for (i = 0; i < n; i++)  
        dst[i] = src[i];  
    for ( ; i < m; i++)  
        dst[i] = '\0';  
    return;  
}
```

**.data**

**src:** .string "source"

**dst:** .string "-----"

## Lab 6 - Esercizio 4 – strcpy (String Copy)

```
# strlen
.globl _start

.data
    src: .string "source"
    dst: .string "-----"

.text
_start:
    # call strcpy
    la    a0, src
    la    a1, dst
    jal   ra, strcpy

    # print the size of dst
    la    a0, dst
    jal   ra, strlen
    li    a7, 1
    ecall
```

Main

## Lab 6 - Esercizio 4 – strcpy (String Copy)

```
# a0 = const char *str
# a1 = const char *dst
```

```
strcpy:
```

```
    addi sp, sp, -32
    sd   ra, 0(sp)
    sd   a0, 8(sp)
    sd   a1, 16(sp)
    sd   s1, 24(sp)
```

```
    jal  ra, strlen    # strlen src
    add  s1, a0, zero   # s1 = n
```

```
    ld   a0, 16(sp)    # strlen dst
    jal  ra, strlen
    add  t0, a0, zero   # t0 = m -> assuming m > n
    sub  t1, t0, s1     # t1 = m-n
```

```
    ld   a0, 8(sp)      # recover a0
    ld   a1, 16(sp)     # recover a1
STRCPY_L1:
    beq  t0, zero, STRCPY_L4 # done if i == m
    ble  t0, t1, STRCPY_L2   # if > m-n, copy char
    lb   t2, 0(a0)          # dereference str[i]
    sb   t2, 0(a1)          # str[i] -> dst[i]
    addi a0, a0, 1          # increment a0
    j    STRCPY_L3
STRCPY_L2:
    sb   zero, 0(a1)        # else put a \0
STRCPY_L3:
    addi a1, a1, 1          # increment other regs
    addi t0, t0, -1
    j    STRCPY_L1         # loop
STRCPY_L4:
    ld   s1, 24(sp)
    ld   ra, 0(sp)
    addi sp, sp, 32
    ret
```

## Lab 6 - Esercizio 5 - Inverte Array

Scrivere un metodo **swap(v, x, y)** che scambia i valori di **v[x]** e **v[y]**, dove **v** è l'indirizzo di un array in memoria. Scrivere poi un altro metodo **invert(v, s)**, che utilizza **swap** per invertire un array in memoria.

Nota: L'indirizzo di **v** deve essere passato come parametro ad **invert** dal main, insieme a **s** (size), che rappresenta il numero di word in **v**.

- Quante istruzioni RISC-V sono necessarie per implementare la funzione?
- Quante istruzioni RISC-V verranno eseguite per completare la funzione quando l'array contiene 16 elementi?
- Quanti registri sono stati versati in memoria (*register spilling*) durante l'esecuzione?

*Bonus: Realizzare un metodo **print(v, s)** che stampa **v** ad schermo*

## Lab 6 - Esercizio 6 - Somma Array

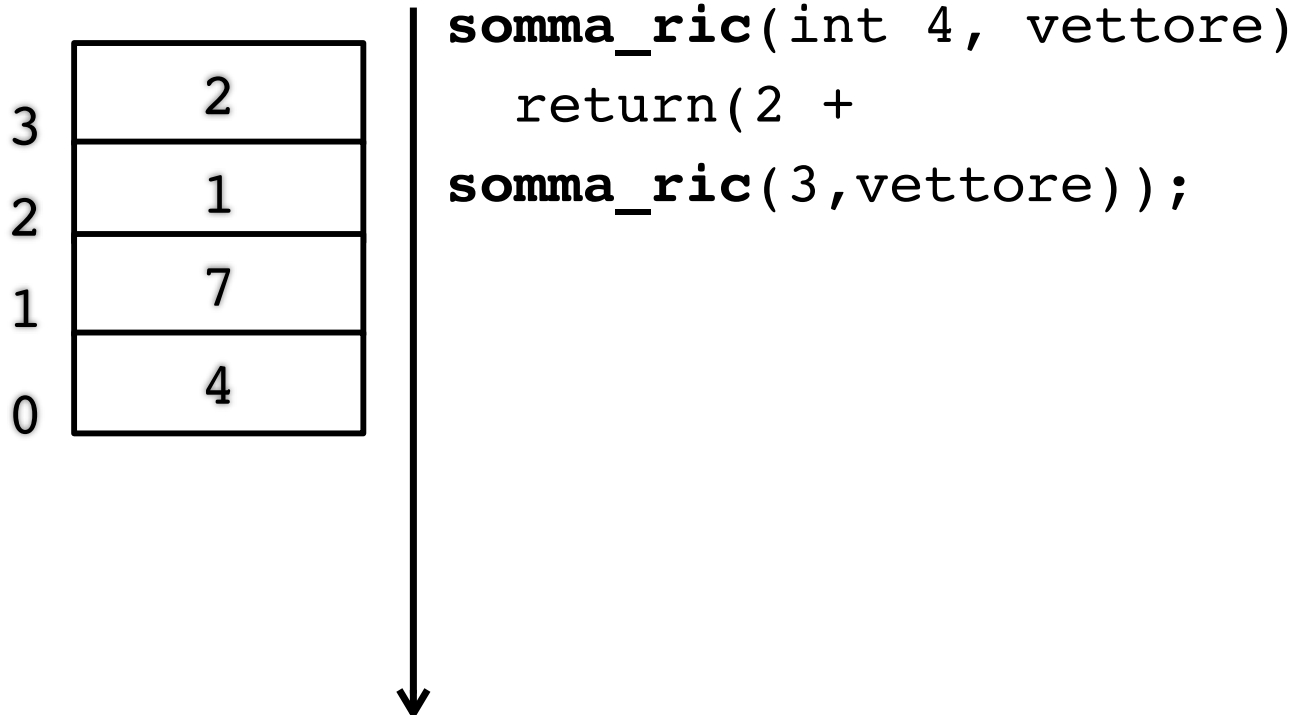
- Scrivere due versioni per una procedura che calcoli la somma di un array di word in memoria:
- una iterativa (cfr. Lab 5, Esercizio 4)
- una ricorsiva  $\rightarrow$   $\text{somma} := v[1] + \text{somma}(v[2:s])$
- Quante istruzioni RISC-V sono necessarie per realizzare le procedure?
- Quante istruzioni RISC-V verranno eseguite per completare le procedure quando l'array contiene 16 elementi?
- Quanti registri sono stati versati in memoria (*register spilling*) durante l'esecuzione delle due versioni?



```
int somma_it(int size, int vettore[]) {  
    int res = 0;  
    int i = 0;  
    while(i < size)  
        res += vettore[i++];  
  
    return res;  
}
```

```
int somma_ric(int size, int* vettore) {  
    if(size <= 0)  
        return 0;  
  
    return (vettore[size-1] + somma_ric(size-1, vettore));  
}
```

```
int somma_ric(int size, int* vettore) {  
    if(size <= 0)  
        return 0;  
  
    return (vettore[size-1] + somma_ric(size-1,  
vettore));  
}
```



```
int somma_ric(int size, int* vettore) {  
    if(size <= 0)  
        return 0;  
  
    return (vettore[size-1] + somma_ric(size-1,  
vettore));  
}
```

3	2
2	1
1	7
0	4


↓

```
somma_ric(int 4, vettore)  
    return(2 +  
somma_ric(3,vettore));  
    return(1 +  
somma_ric(2,vettore));
```

```
int somma_ric(int size, int* vettore) {  
    if(size <= 0)  
        return 0;  
  
    return (vettore[size-1] + somma_ric(size-1,  
vettore));  
}
```

3	2
2	1
1	7
0	4

**somma\_ric**(int 4, vettore)  
 return(2 +  
**somma\_ric**(3,vettore));  
 return(1 +  
**somma\_ric**(2,vettore));  
 return(7 +  
**somma\_ric**(1,vettore));  
 return(4 +  
**somma\_ric**(0,vettore));



```
int somma_ric(int size, int* vettore) {
    if(size <= 0)
        return 0;

    return (vettore[size-1] + somma_ric(size-1,
vettore));
}
```

3	2
2	1
1	7
0	4

↓

```
somma_ric(int 4, vettore)
    return(2 +
somma_ric(3,vettore));
    return(1 +
somma_ric(2,vettore));
    return(7 +
somma_ric(1,vettore));
    return(4 +
somma_ric(0,vettore));
```

↑

```
somma_ric(int 4, vettore)
    return(2 + 12);

    return(1 + 11);

    return(7 + 4);

    return(4 + 0);
```

```
int somma_ric(int size, int* vettore) {
    if(size <= 0)
        return 0;

    return (vettore[size-1] + somma_ric(size-1,
vettore));
}
```

3	2
2	1
1	7
0	4

↓

```
somma_ric(int 4, vettore)
    return(2 +
somma_ric(3,vettore));
    return(1 +
somma_ric(2,vettore));
    return(7 +
somma_ric(1,vettore));
    return(4 +
somma_ric(0,vettore));
```

↑

```
somma_ric(int 4, vettore)
    return(2 + 12);

    return(1 + 11);

    return(7 + 4);

    return(4 + 0);
```

```
int somma_ric(int size, int* vettore) {
    if(size <= 0)
        return 0;

    return (vettore[size-1] + somma_ric(size-1,
vettore));
}
```

3	2
2	1
1	7
0	4

↓

```
somma_ric(int 4, vettore)
    return(2 +
somma_ric(3,vettore));
    return(1 +
somma_ric(2,vettore));
    return(7 +
somma_ric(1,vettore));
    return(4 +
somma_ric(0,vettore));
```

↑

```
somma_ric(int 4, vettore)
    return(2 + 12);

    return(1 + 11);

    return(7 + 4);

    return(4 + 0);
```

```
int somma_ric(int size, int* vettore) {
    if(size <= 0)
        return 0;

    return (vettore[size-1] + somma_ric(size-1,
vettore));
}
```

3	2
2	1
1	7
0	4

↓

```
somma_ric(int 4, vettore)
    return(2 +
somma_ric(3,vettore));
    return(1 +
somma_ric(2,vettore));
    return(7 +
somma_ric(1,vettore));
    return(4 +
somma_ric(0,vettore));
```

↑

```
somma_ric(int 4, vettore)
    return(2 + 12);

    return(1 + 11);

    return(7 + 4);

    return(4 + 0);
```



