

# Lab 6:

Supporto hardware alle procedure

# Obiettivi

- Tradurre procedure da C ad assembly
- Far pratica con le "convenzioni di chiamata"
- Far pratica con l'utilizzo dello stack

**Procedura:** sottoprogramma memorizzato che svolge un compito specifico basandosi sui parametri che gli vengono passati in ingresso.

# Esecuzione di una procedura

Per l'esecuzione di una procedura, un programma deve eseguire questi sei passi:

1. Mettere i **parametri** in un luogo accessibile alla procedura;
2. **Trasferire il controllo** alla procedura;
3. **Acquisire le risorse** necessarie per l'esecuzione della procedura;
4. **Eseguire** il compito richiesto;
5. Mettere il **risultato** in un luogo accessibile al programma chiamante;
6. **Restituire il controllo** al punto di origine, dato che la stessa procedura può essere chiamata in diversi punti di un programma.

# 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 il "register spilling"**
- Quando dobbiamo, usiamo lo stack per fare Register Spilling

# 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**

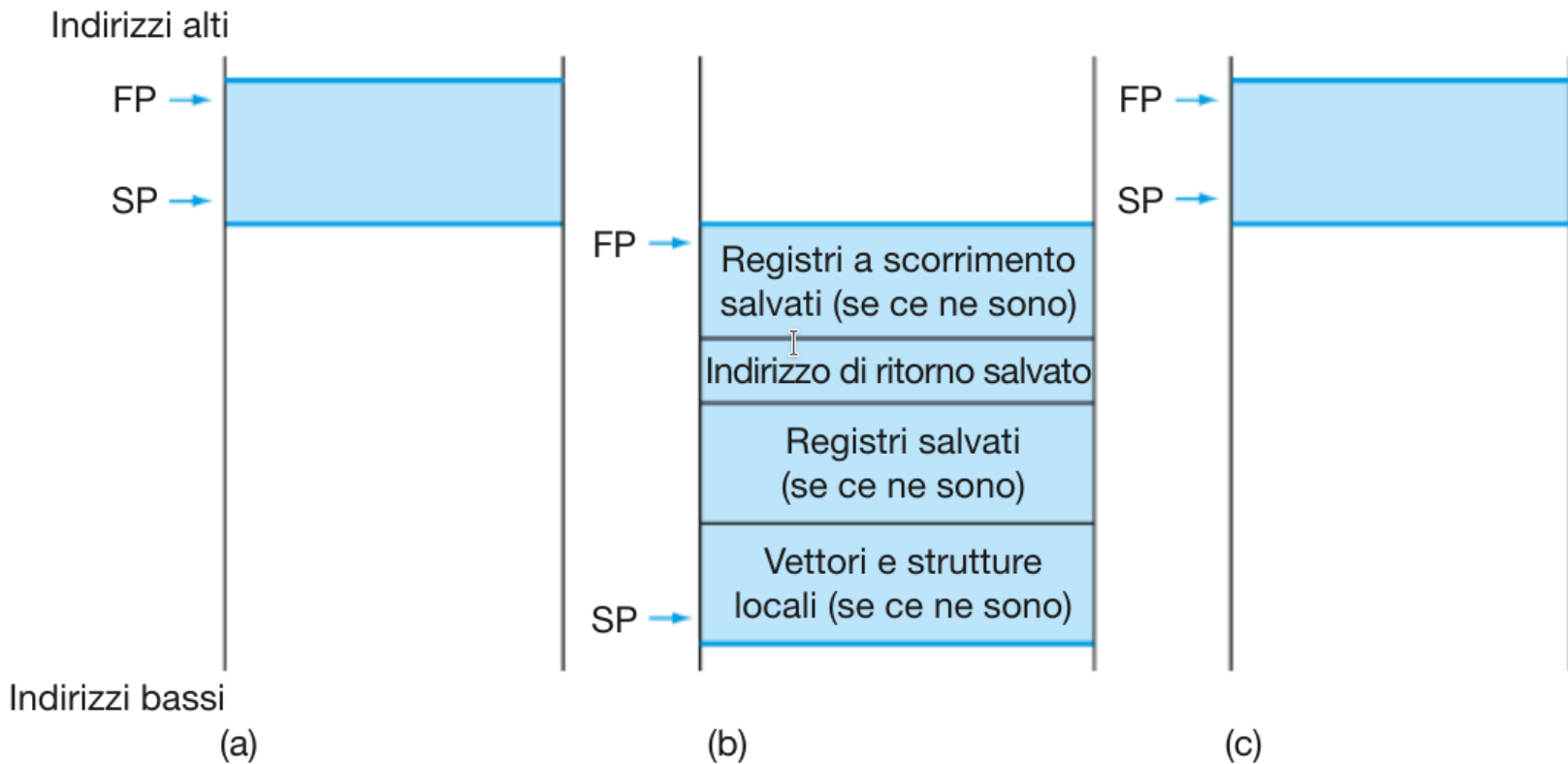


# Convenzioni di chiamata - **chiamato**

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**

# 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**

## 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 procedura **MCD(a,b)** da richiamare nel main. L'algoritmo di Euclide in pseudo-codice è il seguente:

```
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);  
}
```

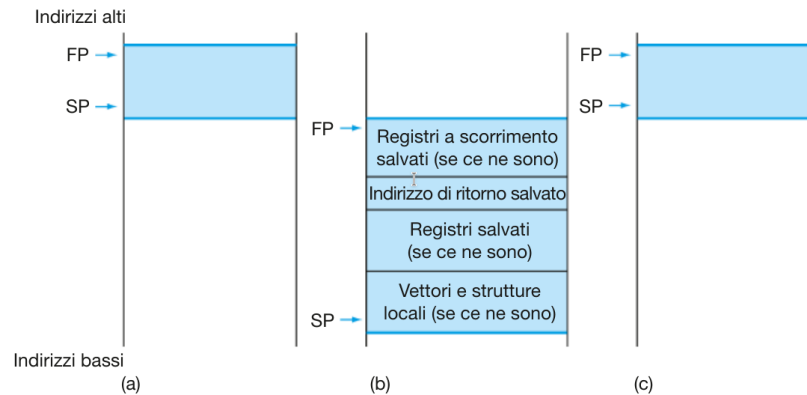
- Quante istruzioni RISC-V sono necessarie per implementare la funzione?
- Quante istruzioni RISC-V verranno eseguite per completare la funzione quando a=24, b=30?

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

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

**mcd:**

**ret**



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

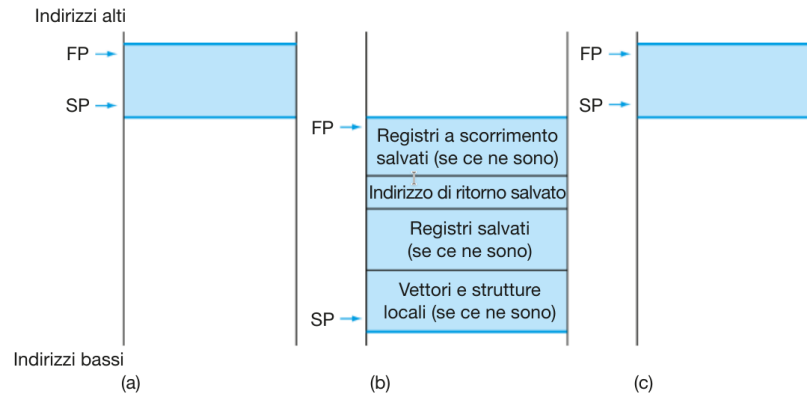
# Lab 6 - Esercizio 1 - MCD(a,b)

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

**mcd:**

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

```
ld      fp, 0(sp)
addi    sp, sp, 8
ret
```



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

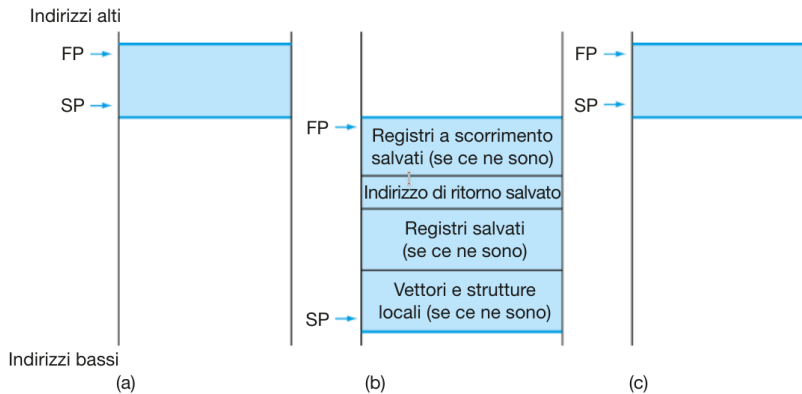
# Lab 6 - Esercizio 1 - MCD(a,b)

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

**mcd:**

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

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

```
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 riversare i registri in memoria?
- Quante istruzioni RISC-V sono necessarie per implementare la procedura?
- Quante istruzioni RISC-V verranno eseguite per completare la procedura quando  $a=12$ ,  $b=9$ ?

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

```
# Procedure MCM(a,b)
# a0 -> a
# a1 -> b
# return MCM su a0
mcm:
```

```
mul    t1, a0, a1
jal    ra, mcd
div    a0, t1, a0
```

```
ret
```

Serve salvare qualcosa?

**Simulare questo codice su  
RARS**


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

```
# Procedure MCM(a,b)
# a0 -> a
# a1 -> b
# return MCM su a0
```

```
mcm:
    addi    sp, sp, -8
    sd      ra, 0(sp)

    mul     t1, a0, a1
    jal     ra, mcd
    div     a0, t1, a0

    ld      ra, 0(sp)
    addi    sp, sp, 8
    ret
```



ra → sovrascritto!

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

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

```
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 3 – strlen (String Length)

```
.globl _start
```

```
.data
```

```
    src: .string "This is the source string."
```

```
.text
```

```
_start:
```

```
    # call strlen
```

```
    la    a0, src
```

```
    jal   ra, strlen
```

```
    # print the size, ret in a0
```

```
    li    a7, 1
```

```
    ecall
```

Main

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

```
strlen:
    add    t0, zero, zero        # i = 0

                                     # Start of for loop
strlen_loop:
    add    t1, t0, a0            # Add the byte offset for str[i]
    lb     t1, 0(t1)             # Dereference str[i]
    beq    t1, zero, strlen_end  # if str[i] == 0, break for loop
    addi   t0, t0, 1             # i++
    j      strlen_loop           # loop

strlen_end:
    addi   a0, t0, 0             # Move t0 into a0 to return
    ret
```

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

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

`strlen(dst) >= strlen(src)`

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

```
void strcpy(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:                                     # else put a \0
    sb     zero, 0(a1)

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 una procedura **swap(v, x, y)** che scambi i valori di **v[x]** e **v[y]**, dove **v** è l'indirizzo di un array in memoria. Scrivere poi un'altra procedura **invert(v, s)**, che utilizzi **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 procedura?
- Quante istruzioni RISC-V verranno eseguite per completare la procedura 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 5 - Inverte Array

```
# Procedure swap(v, x, y)
# a0 -> address of v
# a1 -> index x
# a2 -> index y
swap:
    slli    a1, a1, 2      # calculates offset of x
    slli    a2, a2, 2      # calculates offset of y
    add     t0, a0, a1     # address of v[x]
    add     t1, a0, a2     # address of v[y]

    lw      t2, 0(t0)      # swap the values
    lw      t3, 0(t1)
    sw      t3, 0(t0)
    sw      t2, 0(t1)
    ret                                # return
```

# Lab 6 - Esercizio 5 - Inverte Array

**invert:**

```
    addi    sp, sp, -32  
    sd      ra, 0(sp)  
    sd      a0, 8(sp)  
    sd      s0, 16(sp)  
    sd      s1, 24(sp)
```

```
    addi    s0, zero, 0  
    addi    s1, a1, -1
```

**LOOP\_invert:**

```
    blt     s1, s0, END_invert  
    ld      a0, 8(sp)  
    mv      a1, s0  
    mv      a2, s1  
    jal     ra, swap  
    addi    s0, s0, 1  
    addi    s1, s1, -1  
    j      LOOP_invert
```

**END\_invert:**

```
    ld      ra, 0(sp)  
    ld      s0, 16(sp)  
    ld      s1, 24(sp)  
    addi    sp, sp, 32  
    ret
```

# Lab 6 - Esercizio 5 - Inverte Array

**invert:**

```
addi    sp, sp, -32
sd       ra, 0(sp)
sd       a0, 8(sp)
sd       s0, 16(sp)
sd       s1, 24(sp)
```

```
addi    s0, zero, 0
addi    s1, a1, -1
```

**LOOP\_invert:**

```
blt     s1, s0, END_invert
ld       a0, 8(sp)
mv       a1, s0
mv       a2, s1
jal      ra, swap
addi    s0, s0, 1
addi    s1, s1, -1
j       LOOP_invert
```

**END\_invert:**

```
ld       ra, 0(sp)
ld       s0, 16(sp)
ld       s1, 24(sp)
addi    sp, sp, 32
ret
```

- Non possiamo assumere che a0, a1 e a2 saranno ancora validi dopo "swap"
- Non possiamo assumere che i registri t\* saranno ancora validi
- I registri s\*, invece, rimangono validi, anche se usati da swap



# Lab 6 - Esercizio 5 - Inverte Array

invert:

```
addi    sp, sp, -32
sd       ra, 0(sp)
sd       a0, 8(sp)
sd       s0, 16(sp)
sd       s1, 24(sp)
```

```
addi    s0, zero, 0
addi    s1, a1, -1
```

LOOP\_invert:

```
blt     s1, s0, END_invert
ld       a0, 8(sp)
mv       a1, s0
mv       a2, s1
jal      ra, swap
addi    s0, s0, 1
addi    s1, s1, -1
j       LOOP_invert
```

END\_invert:

```
ld       ra, 0(sp)
ld       s0, 16(sp)
ld       s1, 24(sp)
addi    sp, sp, 32
ret
```

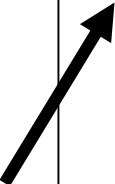
- Non possiamo assumere che a0, a1 e a2 saranno ancora validi dopo "swap"
- Non possiamo assumere che i registri t\* saranno ancora validi
- I registri s\*, invece, rimangono validi, anche se usati da swap
- Salvare a0, in questo caso, è **inutile**, perché swap non lo usa. Sarà un compito del **compilatore** trovare i casi dove il codice assembly può essere ottimizzato

# Lab 6 - Esercizio 5 - Print Array

```
# Procedure print(v)
# a0 -> v address
# a1 -> v size
print:
    addi    sp, sp, -32
    sd      s1, 0(sp)
    sd      a0, 8(sp)
    sd      a1, 16(sp)
    sd      ra, 24(sp)

    li      s1, 0

...
```



```
LOOP_print:
    beq     s1, a1, EXIT_print
    slli    t1, s1, 2
    add     t1, t1, a0
    lw      t0, 0(t1)

    addi    a0, t0, 0           # print a number
    li      a7, 1
    ecall

    addi    a0, zero, 0x20     # print space
    li      a7, 11
    ecall

    ld      a0, 8(sp)          # recover the value of a0
    ld      a1, 16(sp)         # recover the value of a1
    addi    s1, s1, 1          # move to the next word
    j       LOOP_print

...
```

```
EXIT_print:
    addi    a0, zero, 0x0A     # new line
    li      a7, 11
    ecall

    ld      s1, 0(sp)
    ld      ra, 24(sp)
    addi    sp, sp, 32
    ret
```

## 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?



# Lab 6 - Esercizio 6 - Somma Array (Iterativa)

```
sumi:
    addi    sp, sp, -8
    sd      ra, 0(sp)

    li      t0, 0                # final sum
LOOP_sumi:
    ble     a1, zero, END_sumi   # if s1 <= 0 goto end
    lw      t1, 0(a0)            # first element of the vector
    add     t0, t0, t1           # sum the element
    addi    a1, a1, -1           # decrement the counter
    addi    a0, a0, 4            # move to the next word in the array
    j       LOOP_sumi

END_sumi:
    mv      a0, t0               # load the result
    ld      ra, 0(sp)           # restore the return address
    addi    sp, sp, 8           # restore the stack pointer
    ret
```

# Lab 6 - Esercizio 6 - Somma Array (Ricorsiva)

**sumr:**

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

```
mv    s1, zero
```

```
# if size <= 0 end
```

```
ble   a1, zero, END_SUMR
```

```
# otherwise recursively call sumr
```

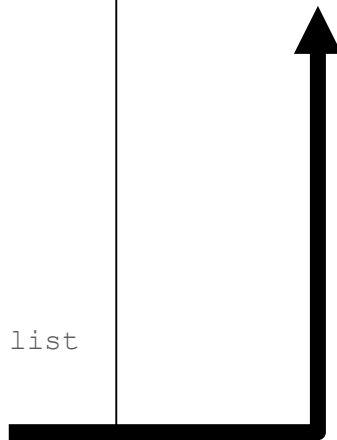
```
addi  a0, a0, 4
addi  a1, a1, -1
jal   ra, sumr
```

```
# a0 has the sum of the tail of the list
```

```
add   s1, a0, zero
ld    a0, 8(sp)      # recover the saved a0
ld    t0, 0(a0)      # value in the head of the list
add   s1, s1, t0     # sum head with sumr(tail)
```

**END\_SUMR:**

```
mv    a0, s1          # load the result
ld    s1, 16(sp)      # restore the saved register
ld    ra, 0(sp)       # restore the return address
addi  sp, sp, 24      # restore the stack pointer
ret
```



# Lab 6 - Bonus - Fibonacci Ricorsivo

Tradurre il seguente frammento di codice C in codice assembly RISC-V.

```
int fib(int n) {  
    if (n==0)  
        return 0;  
    else if (n==1)  
        return 1;  
    else  
        return(fib(n-1) + fib(n-2));  
}
```

- Quante istruzioni RISC-V sono necessarie per implementare la funzione?
- Quante istruzioni RISC-V verranno eseguite per completare la funzione quando N=8?
- Per N=8, quanti registri sono stati versati in memoria (*register spilling*) durante l'esecuzione?

## Lab 6 - Bonus - Fibonacci Ricorsivo