

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

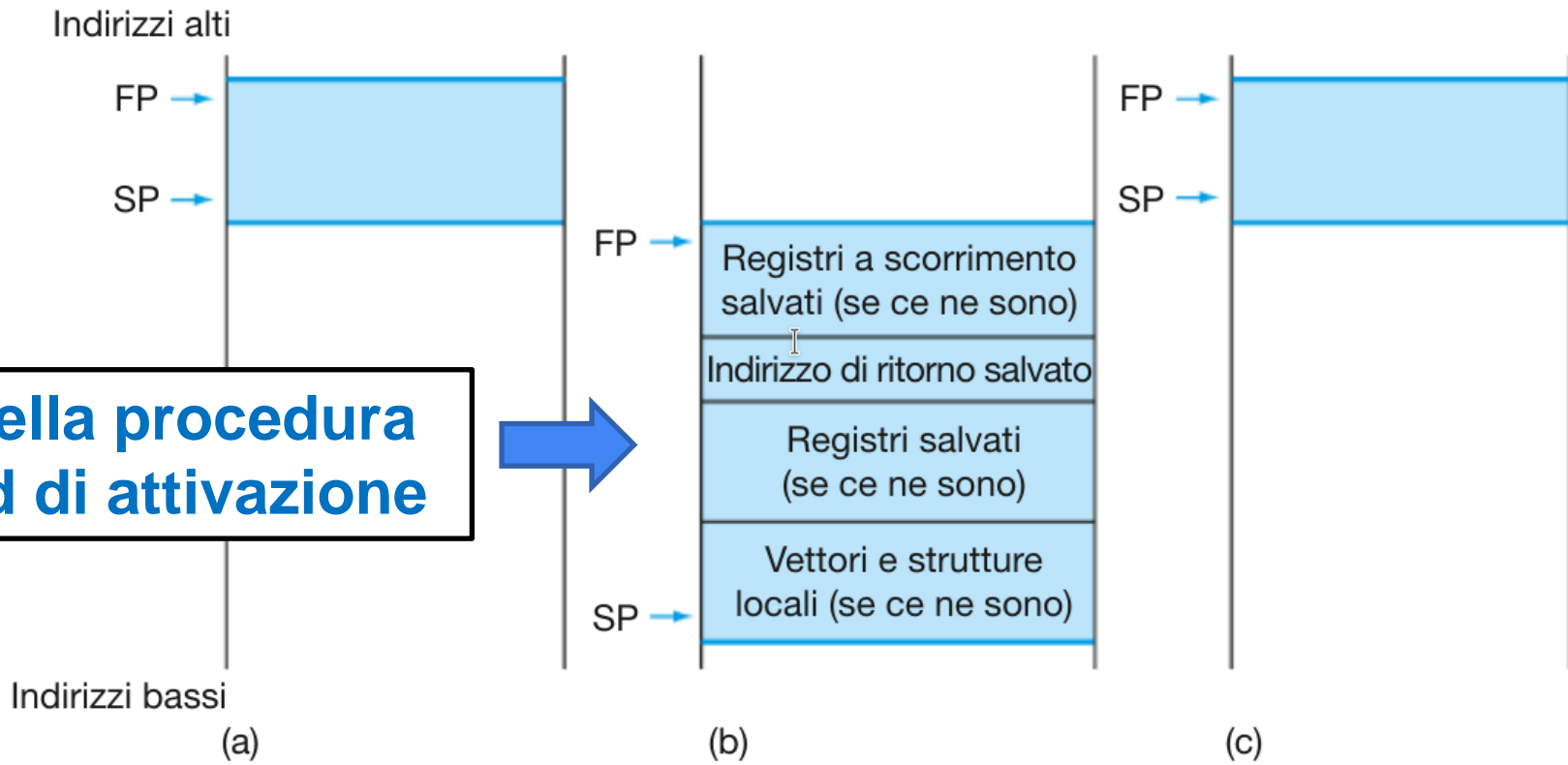
Procedura: sottoprogramma memorizzato che svolge un compito specifico basandosi sui parametri che gli vengono passati in ingresso.

Salvataggio sullo stack del contenuto dei registri

Stack Pointer (sp)

- Nel caso servano più degli 8 registri $a0-a7$ ($x10-x17$), **dobbiamo copiare i valori in memoria**
- La struttura dati utilizzata a questo fine è lo **stack**.
- **Lo stack pointer (sp)** contiene l'indirizzo della cima dello stack
- Lo stack memorizza i registri che devono essere salvati prima della chiamata alle procedure, i parametri addizionali da passare alla procedura, le variabili locali ecc.
- Il processo di trasferimento in memoria delle variabili utilizzate meno di frequente (oppure di quelle che verranno utilizzate successivamente) si chiama **register spilling** (versamento dei registri)

Stack



- Lo stack 'cresce' **da indirizzi di memoria alti verso indirizzi di memoria bassi**
- Quindi quando vengono inseriti dati nello stack il valore dello **sp diminuisce**
- **sp aumenta** quando i dati sono estratti dallo stack

Esempio (multiply)

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

chiamata di funzione



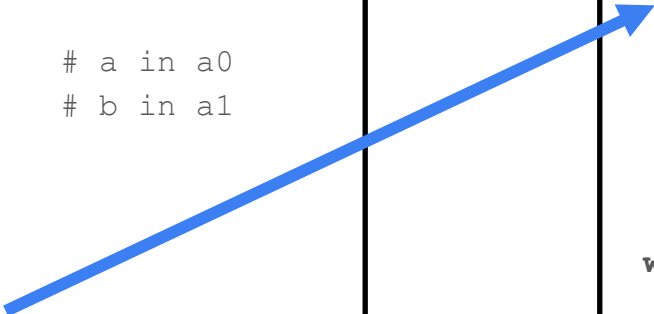
```
int multiply(int a, int b){  
    int i = 0;  
    int acc = 0;  
    while(i < b){  
        acc += a;  
        ++i;  
    }  
    return acc;  
}
```

Esempio (multiply)

```
_start:
    li a0, 3          # a in a0
    li a1, 4          # b in a1

    li s1, 10
    li t0, 13

    jal multiply
    add t1, a0, zero   # result t1
    ...
```



Semplifichiamo: multiply non usa il frame pointer (non ne ha bisogno...)


```
# a0 -> a
# a1 -> b
# return in a0
multiply:
    addi    sp, sp, -8
    sd      s1, 0(sp)
    li      s1, 0      # acc
    li      t0, 0      # i

whileloop:
    beq      t0, a1, endwhile
    add      s1, s1, a0
    addi     t0, t0, 1
    j        whileloop

endwhile:
    add      a0, s1, zero
    ld      s1, 0(sp)
    addi     sp, sp, 8
    jr      ra
```

Esempio (multiply)

```
_start:  
    li a0, 3          # a in a0  
    li a1, 4          # b in a1  
  
    li s1, 10  
    li t0, 13  
  
    jal multiply  
    add t1, a0, zero  # result t1  
    ...
```



**Il chiamante aveva impostato altri 2
registri (s1 e t0)**


```
# a0 -> a  
# a1 -> b  
# return in a0  
multiply:  
    addi    sp, sp, -8  
    sd      s1, 0(sp)  
    li      s1, 0      # acc  
    li      t0, 0      # i  
  
whileloop:  
    beq     t0, a1, endwhile  
    add     s1, s1, a0  
    addi    t0, t0, 1  
    j       whileloop  
  
endwhile:  
    add     a0, s1, zero  
    ld      s1, 0(sp)  
    addi    sp, sp, 8  
    jr      ra
```


Esempio (multiply)

```
_start:
    li a0, 3          # a in a0
    li a1, 4          # b in a1

    li s1, 10
    li t0, 13

    jal multiply
    add t1, a0, zero   # result t1
    ...
```



Cosa possiamo aspettarci per s1 e t0 dopo multiply?

```
# a0 -> a
# a1 -> b
# return in a0
multiply:
    addi    sp, sp, -8
    sd      s1, 0(sp)
    li      s1, 0      # acc
    li      t0, 0      # i

whileloop:
    beq      t0, a1, endwhile
    add      s1, s1, a0
    addi     t0, t0, 1
    j        whileloop

endwhile:
    add      a0, s1, zero
    ld       s1, 0(sp)
    addi     sp, sp, 8
    jr       ra
```

Esempio (multiply)

Address	Code	Basic	
0x00400000	0x00300513	addi x10,x0,3	15: li a0, 3
0x00400004	0x00400593	addi x11,x0,4	16: li a1, 4
0x00400008	0x00a00493	addi x9,x0,10	18: li s1, 10
0x0040000c	0x00d00293	addi x5,x0,13	19: li t0, 13
0x00400010	0x058000ef	jal x1,0x00000058	21: jal multiply



stato **prima** di eseguire jal

multiply

- pc vale 0x000000000000**400010**
- ra vale 0x000000000000000000
- s1 vale 0x0000000000000000**a**
- t0 vale 0x0000000000000000**d**

```
# a0 -> a
# a1 -> b
# return in a0
```

multiply:

```
    addi    sp, sp, -8
    sd      s1, 0(sp)
    li      s1, 0      # acc
    li      t0, 0      # i
```

whileloop:

```
    beq     t0, a1, endwhile
    add     s1, s1, a0
    addi    t0, t0, 1
    j       whileloop
```

endwhile:

```
    add     a0, s1, zero
    ld      s1, 0(sp)
    addi    sp, sp, 8
    jr      ra
```

Esempio (multiply)

0x0040006c	0x00913023	sd x9,0(x2)	74: sd s1, 0(sp) # s1
0x00400070	0x00000493	addi x9,x0,0	76: li s1, 0 # acc
0x00400074	0x00000293	addi x5,x0,0	77: li t0, 0 # i
0x00400078	0x00b28863	beq x11,x11,0x00000010	80: beq t0, a1, endwhile
0x0040007c	0x00a484b3	add x1,x9,x10	81: add s1, s1, a0
0x00400080	0x00128293	addi x5,x5,1	82: addi t0, t0, 1
0x00400084	0xff5ff06f	jal x1,0xffffffff4	84: j whileloop

Multiply usa sia s1 che t0 come registri di appoggio

- pc vale 0x000000000000**400078**
- ra vale 0x000000000000**400014**
- s1 vale 0x000000000000000000
- t0 vale 0x000000000000000000

```
# a0 -> a
# a1 -> b
# return in a0
```

multiply:

```
addi sp, sp, -8
sd s1, 0(sp)
li s1, 0 # acc
li t0, 0 # i
```

whileloop:

```
beq t0, a1, endwhile
add s1, s1, a0
addi t0, t0, 1
j whileloop
```

endwhile:

```
add a0, s1, zero
ld s1, 0(sp)
addi sp, sp, 8
jr ra
```

Esempio (multiply)

0x0040006c	0x00913023	sd x9,0(x2)	74:	sd	s1, 0(sp)	# s1
0x00400070	0x00000493	addi x9,x0,0	76:	li	s1, 0	# acc
0x00400074	0x00000293	addi x5,x0,0	77:	li	t0, 0	# i
0x00400078	0x00b28863	beq x5,x11,0x00000010	80:	beq	t0, a1, endwhile	
0x0040007c	0x00a484b3	add x9,x9,x10	81:	add	s1, s1, a0	
0x00400080	0x00128293	addi x5,x5,1	82:	addi	t0, t0, 1	
0x00400084	0xff5ff06f	jal x0,0xffffffff4	84:	j	whileloop	

Il chiamato deve salvare i registri *s** (se usati)

```
# a0 -> a
# a1 -> b
# return in a0
```

multiply:

```
    addi    sp, sp, -8
    sd      s1, 0(sp)
    li      s1, 0      # acc
    li      t0, 0      # i
```

whileloop:

```
    beq     t0, a1, endwhile
    add     s1, s1, a0
    addi    t0, t0, 1
    j       whileloop
```

endwhile:

```
    add     a0, s1, zero
    ld      s1, 0(sp)
    addi    sp, sp, 8
    jr      ra
```

Esempio (multiply)

Address	Code	Basic	
0x00400000	0x00300513	addi x10,x0,3	15: li a0, 3
0x00400004	0x00400593	addi x11,x0,4	16: li a1, 4
0x00400008	0x00a00493	addi x9,x0,10	18: li s1, 10
0x0040000c	0x00d00293	addi x5,x0,13	19: li t0, 13
0x00400010	0x058000ef	jal x1,0x00000058	21: jal multiply
0x00400014	0x00050333	add x6,x10,x0	22: add t1, a0, zero

t0 sovrascritto

stato **dopo** di eseguire jal multiply

- pc vale 0x000000000000**400014**
- ra vale 0x000000000000**400014**
- s1 vale 0x0000000000000000**a**
- t0 vale 0x0000000000000000**4**

```
# a0 -> a
# a1 -> b
# return in a0
```

multiply:

```
    addi    sp, sp, -8
    sd      s1, 0(sp)
    li      s1, 0      # acc
    li      t0, 0      # i
```

whileloop:

```
    beq     t0, a1, endwhile
    add     s1, s1, a0
    addi    t0, t0, 1
    j       whileloop
```

endwhile:

```
    add     a0, s1, zero
    ld      s1, 0(sp)
    addi    sp, sp, 8
    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(a, acc);
        ++i;
    }
    return acc;
}
```

chiamata di funzione

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

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). 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(a, acc);  
        ++i;  
    }  
    return acc;  
}
```


Convenzioni di chiamata

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, e quelli **non preservati** fra le chiamate

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

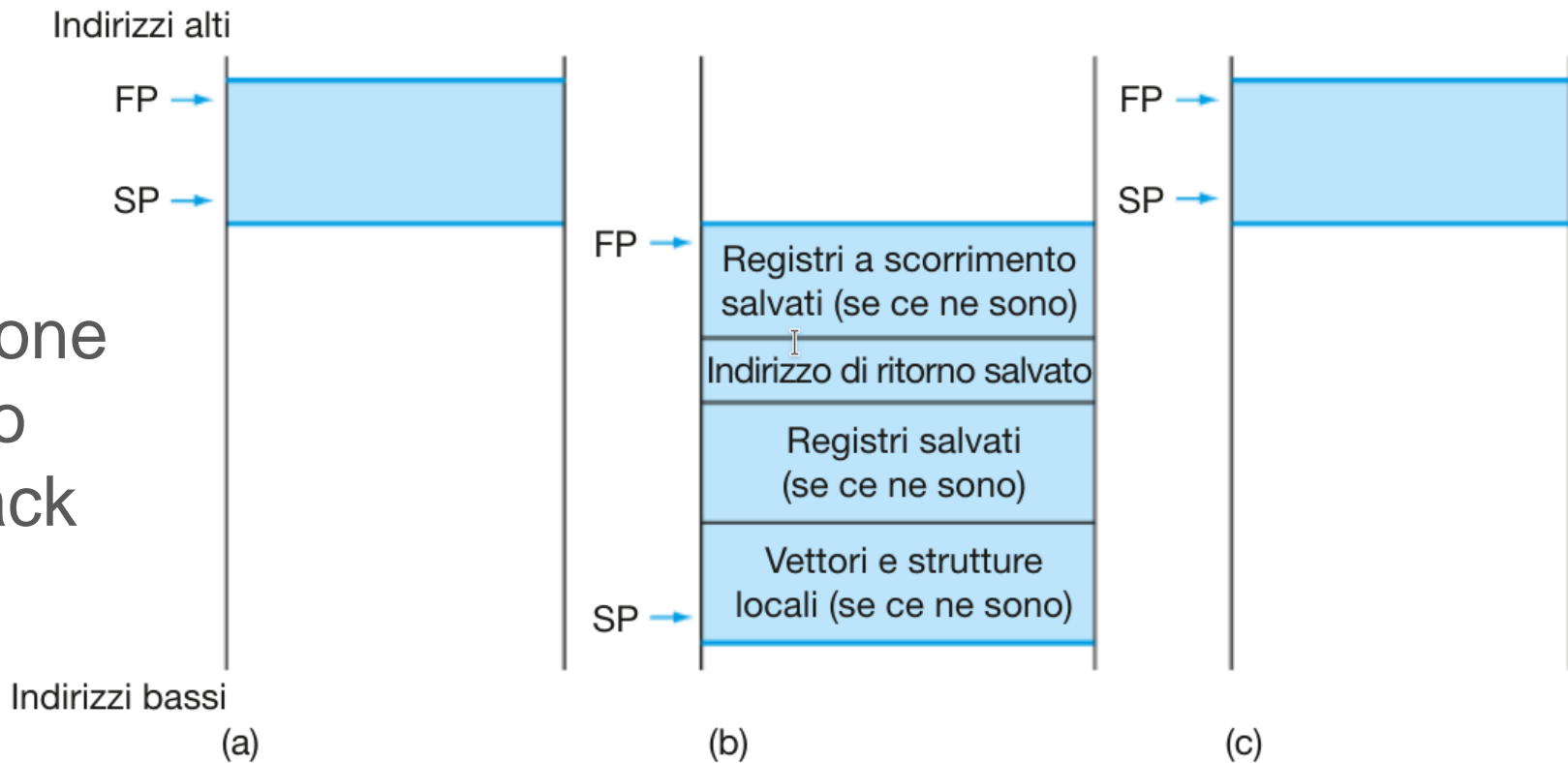
**add/sub
sempre lo stesso
numero di byte**

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**

Allocazione di spazio sullo 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**

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);  
}
```

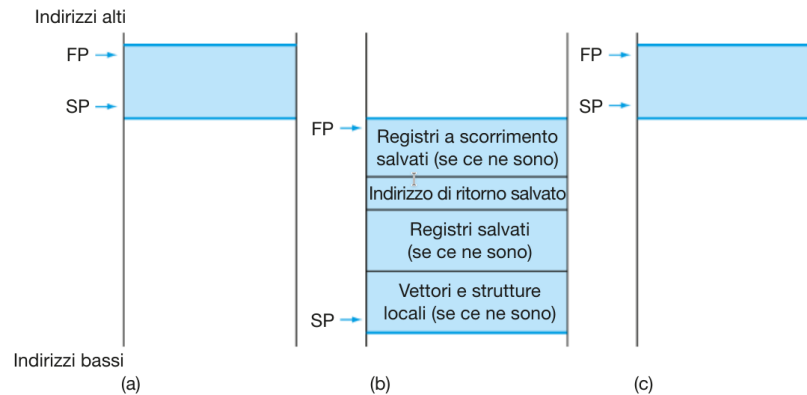
- Quante istruzioni RISC-V sono necessarie per implementare la funzione?
- Quante istruzioni RISC-V verranno eseguite per completare la funzione quando a=24, b=30?

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;  
}
```

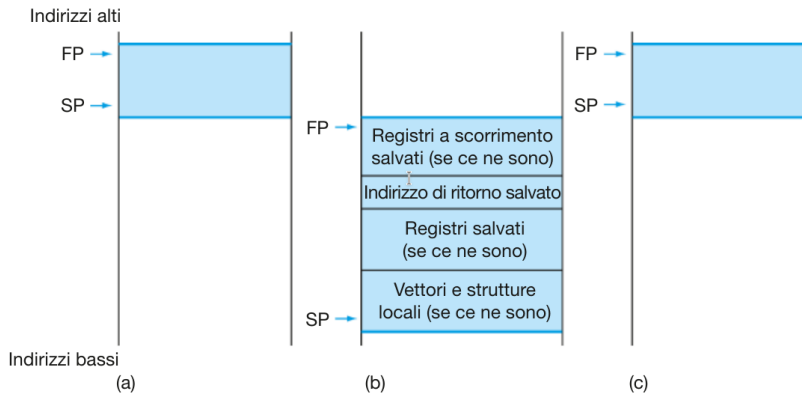
Esercizio 1 - MCD(a,b)

```
# 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;  
}
```

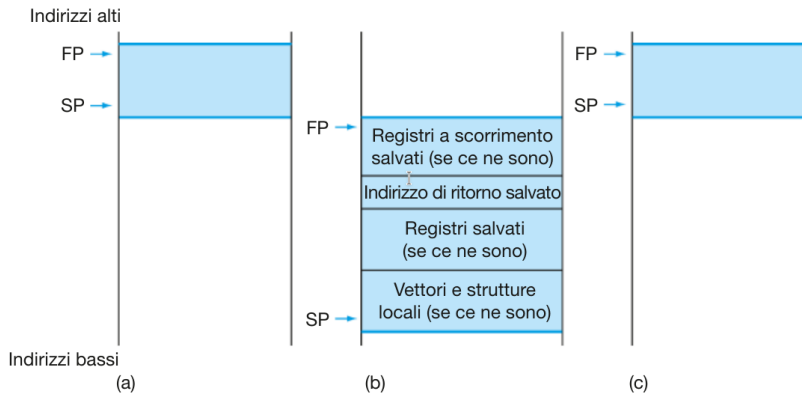
Esercizio 1 - MCD(a,b)

```
# 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
```



ottimizziamo: mcd non usa il frame pointer (non ne ha bisogno...)

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

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 $a=12$, $b=9$?

Esercizio 2 - MCM(a,b)

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

```
mul    s1, a0, a1
jal    ra, mcd
div    a0, s1, a0
```

```
ret
```

Serve salvare qualcosa?

**Simulare questo codice su
RARS**

ra → sovrascritto!

Completare questo esercizio e
consegnarlo su Moodle