

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.

Parametri e Indirizzo di Ritorno

- registri `a0–a7` (`x10–x17`) sono 8 registri per i parametri, utilizzati cioè per passare valori alle funzioni o restituire valori al chiamante
- registro `ra` (`x1`) 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 ra, ProcAddress # salta a ProcAddress e salva indirizzo di ritorno in ra  
[jal ProcAddress]
```

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

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

`jal` e `jalr`: Passaggio di Controllo

Lo schema è quindi il seguente:

- la funzione chiamante mette i parametri in `a0–a7` e usa `jal x` per saltare alla funzione `x`
- la funzione chiamata svolge le proprie operazioni, inserisce i risultati negli stessi registri e restituisce il controllo al chiamante con l'istruzione `jr ra`

Esempio (sum)

```
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);  
}
```

Esempio (sum)

```
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, 1 # a  
li a1, 2 # b  
li s1, 0 # result
```

```
jal sum  
add s1, a0, zero  
...
```

sum:

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


Esempio (sum)

```
int sum(int a, int b){
```

**registri per passaggio parametri
(FUNZIONE CHIAMANTE)**

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

**registri per passaggio parametri
(FUNZIONE CHIAMATA)**

```
}
```

_start:

```
li a0, 1 # a  
li a1, 2 # b  
li s1, 0 # result
```

```
jal sum  
add s1, a0, zero  
...
```

sum:

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

Esempio (sum)

domanda: perché usiamo `jr ra`, e non semplicemente `jump`?

risposta: perché la funzione può essere chiamata da molti punti del programma, anche dall'interno di altre procedure. C'è quindi bisogno di **un meccanismo per tornare all'istruzione successiva alla chiamata**. Serve un meccanismo che tenga conto dell'indirizzo salvato sul registro `ra`.

`_start:`

```
li a0, 1 # a
li a1, 2 # b
li s1, 0 # result
```

```
jal sum
```

```
add s1, a0, zero
```

```
...
```

`sum:`

```
add a0, a0, a1
```

```
jr ra
```

Esempio (sum)

| Address | Code | Basic | |
|------------|-------------|-------------------|----------------------|
| 0x00400000 | 0x00100513 | addi x10,x0,1 | 6: li a0, 1 # a |
| 0x00400004 | 0x00200593 | addi x11,x0,2 | 7: li a1, 2 # b |
| 0x00400008 | 0x00000493 | addi x9,x0,0 | 8: li s1, 0 # result |
| 0x0040000c | 0x01c000ef | jal x1,0x0000001c | 10: jal sum |
| 0x00400010 | 0x000504b3 | add x9,x10,x0 | 11: add s1, a0, zero |
| 0x00400014 | 0x00900533 | add x10,x0,x9 | 14: mv a0, s1 |
| 0x00400018 | 0x00100893 | addi x17,x0,1 | 15: li a7, 1 |
| 0x0040001c | 0x00000073 | ecall | 16: ecall |
| 0x00400020 | 0x00a00893 | addi x17,x0,10 | 19: li a7, 10 |
| 0x00400024 | 0x00000073 | ecall | 20: ecall |
| 0x00400028 | 0x00b50533 | add x10,x10,x11 | 24: add a0, a0, a1 |
| 0x0040002c | 0x000008067 | jalr x0,x1,0 | 25: jr ra |

stato **prima** di eseguire `jal sum`

- pc vale 0x000000000040000c

- ra vale 0x0000000000000000

`_start:`

`li a0, 1 # a`

`li a1, 2 # b`

`li s1, 0 # result`

`jal sum`

`add s1, a0, zero`

`...`

`sum:`

`add a0, a0, a1`

`jr ra`

Esempio (sum)

| Address | Code | Basic | |
|------------|-------------|-------------------|----------------------|
| 0x00400000 | 0x00100513 | addi x10,x0,1 | 6: li a0, 1 # a |
| 0x00400004 | 0x00200593 | addi x11,x0,2 | 7: li a1, 2 # b |
| 0x00400008 | 0x00000493 | addi x9,x0,0 | 8: li s1, 0 # result |
| 0x0040000c | 0x01c000ef | jal x1,0x0000001c | 10: jal sum |
| 0x00400010 | 0x000504b3 | add x9,x10,x0 | 11: add s1, a0, zero |
| 0x00400014 | 0x00900533 | add x10,x0,x9 | 14: mv a0, s1 |
| 0x00400018 | 0x00100893 | addi x17,x0,1 | 15: li a7, 1 |
| 0x0040001c | 0x00000073 | ecall | 16: ecall |
| 0x00400020 | 0x00a00893 | addi x17,x0,10 | 19: li a7, 10 |
| 0x00400024 | 0x00000073 | ecall | 20: ecall |
| 0x00400028 | 0x00b50533 | add x10,x10,x11 | 24: add a0, a0, a1 |
| 0x0040002c | 0x000008067 | jalr x0,x1,0 | 25: jr ra |

ra

pc

stato **dopo** aver eseguito jal sum

- pc vale 0x0000000000400028

- ra vale 0x0000000000400010

_start:

li a0, 1 # a

li a1, 2 # b

li s1, 0 # result

jal sum

add s1, a0, zero

...

sum:

add a0, a0, a1

jr ra

Esempio (sum)

| Address | Code | Basic | |
|------------|-------------|-------------------|----------------------|
| 0x00400000 | 0x00100513 | addi x10,x0,1 | 6: li a0, 1 # a |
| 0x00400004 | 0x00200593 | addi x11,x0,2 | 7: li a1, 2 # b |
| 0x00400008 | 0x00000493 | addi x9,x0,0 | 8: li s1, 0 # result |
| 0x0040000c | 0x01c000ef | jal x1,0x0000001c | 10: jal sum |
| 0x00400010 | 0x000504b3 | add x9,x10,x0 | 11: add s1, a0, zero |
| 0x00400014 | 0x00900533 | add x10,x0,x9 | 14: mv a0, s1 |
| 0x00400018 | 0x00100893 | addi x17,x0,1 | 15: li a7, 1 |
| 0x0040001c | 0x00000073 | ecall | 16: ecall |
| 0x00400020 | 0x00a00893 | addi x17,x0,10 | 19: li a7, 10 |
| 0x00400024 | 0x00000073 | ecall | 20: ecall |
| 0x00400028 | 0x00b50533 | add x10,x10,x11 | 24: add a0, a0, a1 |
| 0x0040002c | 0x000008067 | jalr x0,x1,0 | 25: jr ra |

ra

pc

stato **dopo** aver eseguito jr ra
- pc vale 0x0000000000400010
- ra vale 0x0000000000400010

_start:

```
li a0, 1 # a
li a1, 2 # b
li s1, 0 # result
```

jal sum

add s1, a0, zero

...

sum:

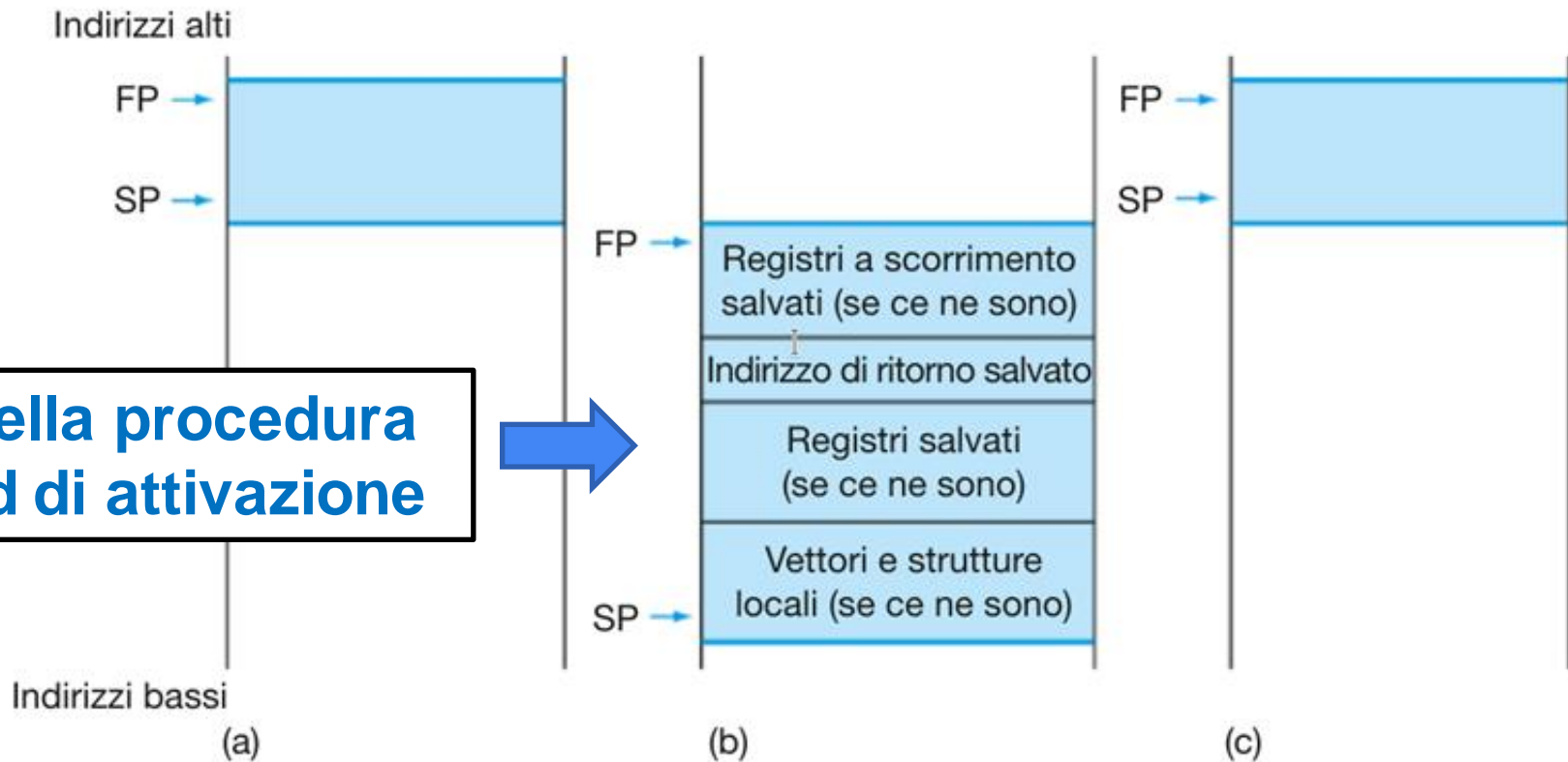
```
add a0, a0, a1
jr ra
```

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




**frame della procedura
o record di attivazione**

- 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 res;  
    result = multiply(a,b);  
  
    printf("res: %d\n", res);  
    exit(0);  
}
```

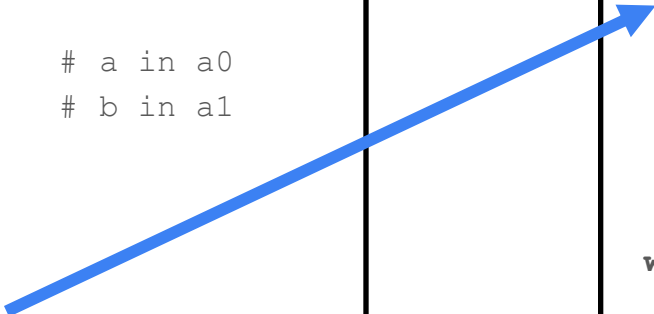
chiamata di funzione



```
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  
    ...
```



```
# 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 0x000000000000000000**a**
- t0 vale 0x000000000000000000**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 x1,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 0x0000000000**400014**
- ra vale 0x0000000000**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 res;  
    result = multiply(a,b);  
  
    printf("res: %d\n", res);  
    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

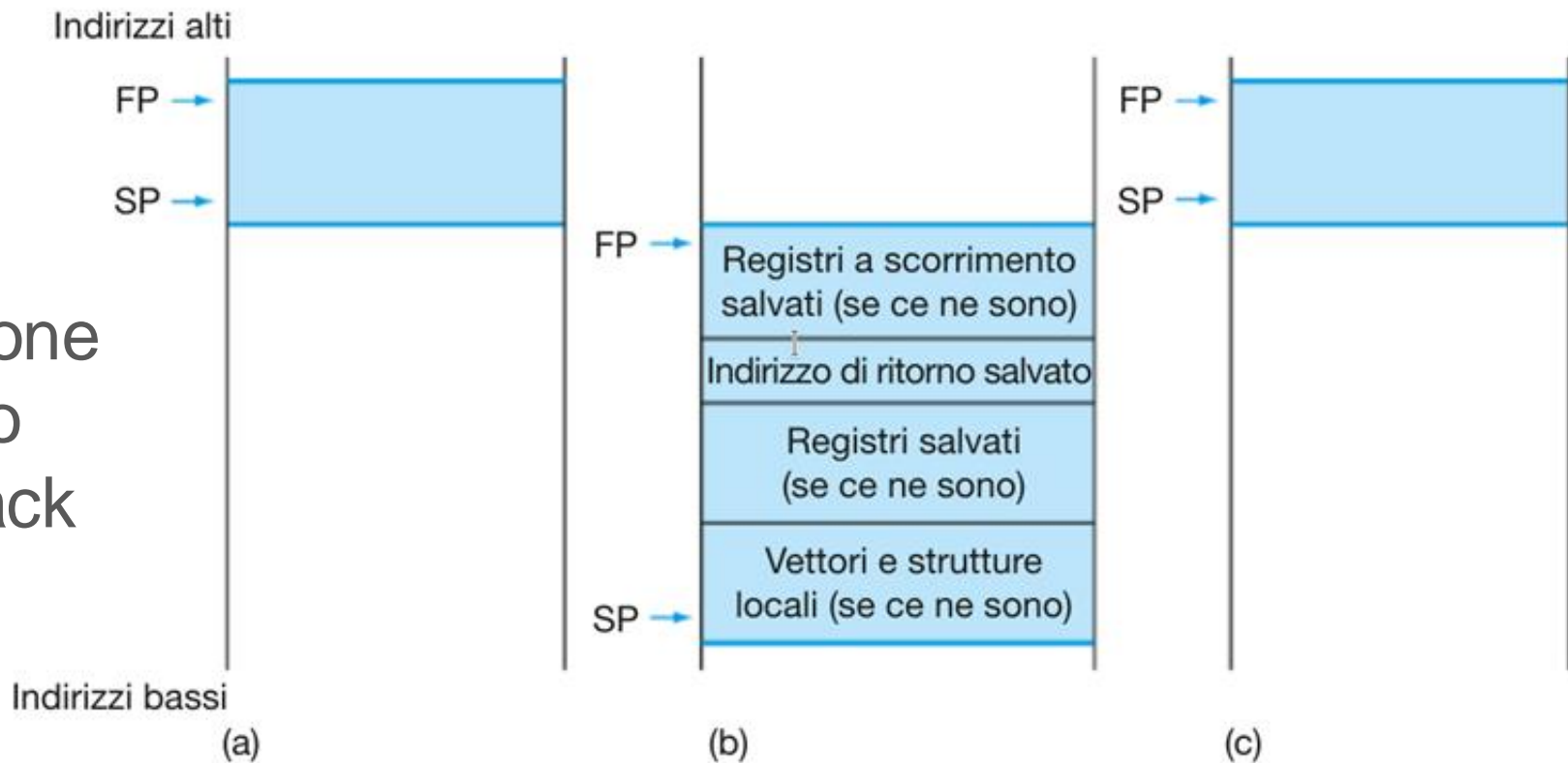
Per evitare costose operazioni di spilling (salvataggio su stack) e di restore (ri-salvataggio da stack a registri) utilizziamo una convenzione. Dividiamo i registri in 2 categorie: **quelli preservati** nel passaggio fra chiamate di funzione, e quelli **non preservati** fra le chiamate

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

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);  
}
```

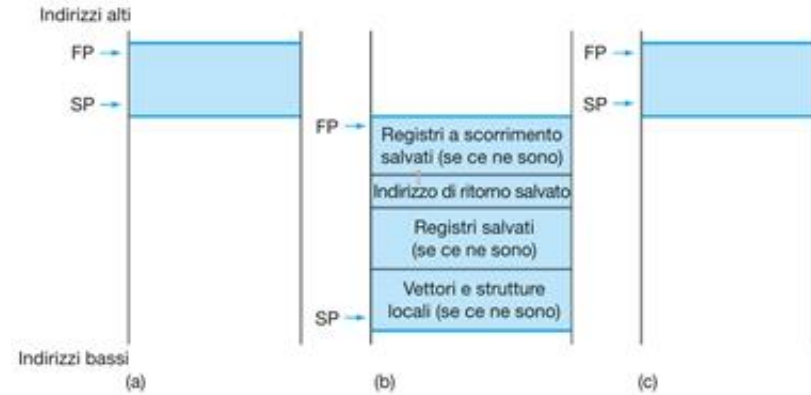
- Quante istruzioni RISC-V sono necessarie per implementare la funzione?
- Quante istruzioni RISC-V verranno eseguite per completare la funzione quando a=24, b=30?

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 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**

Codifica ASCII

- *American Standard Code for Information Interchange*
- Utilizza 8 bit (1 byte) per rappresentare i caratteri
- `load byte unsigned (lbu)` prende un byte dalla memoria mettendolo negli 8 bit di un registro, collocati più a destra
- `store byte (sb)` prende il byte corrispondente agli 8 bit di un registro, collocati più a destra, e lo salva in memoria

```
lbu x12, 0(x10) // Leggi un byte dall'indirizzo sorgente  
sb x12, 0(x11) // Scrivi il byte all'indirizzo di destinazione
```

Codifica ASCII

| Valore ASCII | Carattere | Valore ASCII | Carattere | Valore ASCII | Carattere | Valore ASCII | Carattere | Valore ASCII | Carattere | Valore ASCII | Carattere |
|--------------|-----------|--------------|-----------|--------------|-----------|--------------|-----------|--------------|-----------|--------------|-----------|
| 32 | Spazio | 48 | 0 | 64 | @ | 80 | P | 96 | ` | 112 | p |
| 33 | ! | 49 | 1 | 65 | A | 81 | Q | 97 | a | 113 | q |
| 34 | " | 50 | 2 | 66 | B | 82 | R | 98 | b | 114 | r |
| 35 | # | 51 | 3 | 67 | C | 83 | S | 99 | c | 115 | s |
| 36 | \$ | 52 | 4 | 68 | D | 84 | T | 100 | d | 116 | t |
| 37 | % | 53 | 5 | 69 | E | 85 | U | 101 | e | 117 | u |
| 38 | & | 54 | 6 | 70 | F | 86 | V | 102 | f | 118 | v |
| 39 | ' | 55 | 7 | 71 | G | 87 | W | 103 | g | 119 | w |
| 40 | (| 56 | 8 | 72 | H | 88 | X | 104 | h | 120 | x |
| 41 |) | 57 | 9 | 73 | I | 89 | Y | 105 | i | 121 | y |
| 42 | * | 58 | : | 74 | J | 90 | Z | 106 | j | 122 | z |
| 43 | + | 59 | ; | 75 | K | 91 | [| 107 | k | 123 | { |
| 44 | , | 60 | < | 76 | L | 92 | \ | 108 | l | 124 | |
| 45 | - | 61 | = | 77 | M | 93 |] | 109 | m | 125 | } |
| 46 | . | 62 | > | 78 | N | 94 | ^ | 110 | n | 126 | ~ |
| 47 | / | 63 | ? | 79 | O | 95 | _ | 111 | o | 127 | DEL |

Codifica ASCII

| Valore ASCII | Carattere | Valore ASCII | Carattere | Valore ASCII | Carattere | Valore ASCII | Carattere | Valore ASCII | Carattere | Valore ASCII | Carattere |
|--------------|-----------|--------------|-----------|--------------|-----------|--------------|-----------|--------------|-----------|--------------|-----------|
| 32 | Spazio | 48 | 0 | 64 | @ | 80 | P | 96 | ` | 112 | p |
| 33 | ! | 49 | 1 | 65 | A | 81 | Q | 97 | a | 113 | q |
| 34 | " | 50 | 2 | 66 | B | 82 | R | 98 | b | 114 | r |
| 35 | # | 51 | 3 | 67 | C | 83 | S | 99 | c | 115 | s |
| 36 | \$ | 52 | 4 | 68 | D | 84 | T | 100 | d | 116 | t |
| 37 | % | 53 | 5 | 69 | E | 85 | U | 101 | e | 117 | u |
| 38 | & | 54 | 6 | 70 | F | 86 | V | 102 | f | 118 | v |
| 39 | ' | 55 | 7 | 71 | G | 87 | W | 103 | g | 119 | w |
| 40 | (| 56 | 8 | 72 | H | 88 | X | 104 | h | 120 | x |
| 41 |) | 57 | 9 | 73 | I | 89 | Y | 105 | i | 121 | y |
| 42 | * | 58 | : | 74 | J | 90 | Z | 106 | j | 122 | z |
| 43 | + | 59 | ; | 75 | K | 91 | [| 107 | k | 123 | { |
| 44 | , | 60 | < | 76 | L | 92 | \ | 108 | l | 124 | |

Il linguaggio C termina le stringhe con un byte che contiene il valore 0 (carattere “null” in ASCII, non mostrato nella tabella)

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."
```

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 size, ret in a0
```

```
    li    a7, 1
```

```
    ecall
```

Main

Esercizio 4 – strcmp

Scrivere una procedura RISC-V `strcmp` per confrontare due stringhe di caratteri. `strcmp(str1, str2)` ritorna 0 se `str1` è uguale a `str2`, 1 nel caso contrario.

risultato atteso = 1

```
.globl _start
.data
    str1: .string "first"
    str2: .string "second"
```


Esercizio 4 – strcmp

```
.globl _start
```

```
.data
```

```
    str1: .string "first."
```

```
    str2: .string "second."
```

```
.text
```

```
_start:
```

```
    # call strcmp
```

```
    la    a0, str1
```

```
    la    a1, str2
```

```
    jal   ra, strcmp
```

```
...
```

Main

Esercizio 5 – strchr

Scrivere una procedura RISC-V `strchr(str, char)` per restituire l'indirizzo in memoria della prima occorrenza di `char` in `str`.

`strchr(str, char)` ritorna 0 se `char` non è presente in `str`.

.data

str: .string "my long string"

char: .string "g"

.text

_start:

la a0, str

la t1, char

lbu a1, 0(t1)

jal ra, **strchr**

str è all'indirizzo = 0x0000000010010000

risultato atteso = 0x0000000010010006

Esercizio 6 – strchr

Scrivere una procedura RISC-V `strchr(str, char)` per restituire l'indirizzo in memoria dell'ultima occorrenza di `char` in `str`.

`strchr(str, char)` ritorna 0 se `char` non è presente in `str`.

.data

str: .string "my long string"

char: .string "g"

.text

_start:

la a0, str

la t1, char

lbu a1, 0(t1)

jal ra, **strchr**

str è all'indirizzo = 0x0000000010010000

risultato atteso = 0x000000001001000d