

Grupo ARCOS

**uc3m** | Universidad **Carlos III** de Madrid

# Tema 3 (IV)

## Fundamentos de la programación en ensamblador

Estructura de Computadores  
Grado en Ingeniería Informática



# Contenido

- ▶ Fundamentos básicos de la programación en ensamblador
- ▶ Ensamblador del RISC-V 32, modelo de memoria y representación de datos
- ▶ Formato de las instrucciones y modos de direccionamiento
- ▶ **Llamadas a procedimientos y uso de la pila**
  - ▶ ¿Cómo se llama a una función/subrutina?
  - ▶ ¿Dónde guardar la dirección de retorno en rutinas no terminales?
  - ▶ ¿Cuál es el convenio de paso de parámetros?
  - ▶ ¿Cuál es el convenio de uso de registros?
  - ▶ ¿Cómo son las variables locales?

# Contenido

- ▶ Fundamentos básicos de la programación en ensamblador
- ▶ Ensamblador del RISC-V 32, modelo de memoria y representación de datos
- ▶ Formato de las instrucciones y modos de direccionamiento
- ▶ **Llamadas a procedimientos y uso de la pila**
  - ▶ ¿Cómo se llama a una función/subrutina?
  - ▶ ¿Dónde guardar la dirección de retorno en rutinas no terminales?
  - ▶ ¿Cuál es el convenio de paso de parámetros?
  - ▶ ¿Cuál es el convenio de uso de registros?
  - ▶ ¿Cómo son las variables locales?

# Procedimientos y funciones

```
int factorial(int x) {  
    int i;  
    int r=1;  
    for (i=1;i<=x;i++) {  
        r*=i;  
    }  
    return r;  
}  
...  
r1 = factorial(3) ;  
...
```

- ▶ Un función (procedimiento, método) en alto nivel es un subprograma que realiza una tarea específica cuando se le invoca
  - ▶ Recibe argumentos o parámetros de entrada
  - ▶ Devuelve algún resultado

# Funciones en un lenguaje de alto nivel

## Pasos en la ejecución de una función

```
int main() {  
    int z;  
    1 x=3;  
    z=factorial(x);  
    print_int(z);  
}
```

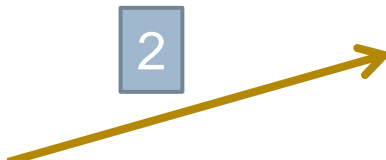
```
int factorial(int x) {  
    int i;  
    int r=1;  
    for (i=1;i<=x;i++) {  
        r*=i;  
    }  
    return r;  
}
```

1. Situar los parámetros en un lugar donde la función pueda accederlos
2. Transferir el control a la función
3. Adquirir los recursos de almacenamiento necesarios para la función
4. Realizar la tarea deseada
5. Guardar el resultado donde la función llamante pueda acceder a él
6. Devolver el control al punto de origen

# Funciones en un lenguaje de alto nivel

## Pasos en la ejecución de una función

```
int main() {  
    int z;  
    x=3;  
    z=factorial(x);  
    print_int(z);  
}  
  
int factorial(int x) {  
    int i;  
    int r=1;  
    for (i=1;i<=x;i++) {  
        r*=i;  
    }  
    return r;  
}
```

A blue square containing the number '2' is positioned above the function call 'z=factorial(x);' in the main function. A yellow arrow points from this square to the start of the 'factorial' function definition.

1. Situar los parámetros en un lugar donde la función pueda accederlos
2. **Transferir el control a la función**
3. Adquirir los recursos de almacenamiento necesarios para la función
4. Realizar la tarea deseada
5. Guardar el resultado donde la función llamante pueda acceder a él
6. Devolver el control al punto de origen

# Funciones en un lenguaje de alto nivel

## Pasos en la ejecución de una función

```
int main() {  
    int z;  
    x=3;  
    z=factorial(x);  
    print_int(z);  
}
```

3



```
int factorial(int x) {  
    int i;  
    int r=1;  
    for (i=1;i<=x;i++) {  
        r*=i;  
    }  
    return r;  
}
```

1. Situar los parámetros en un lugar donde la función pueda accederlos
2. Transferir el control a la función
3. **Adquirir los recursos de almacenamiento necesarios para la función**
4. Realizar la tarea deseada
5. Guardar el resultado donde la función llamante pueda acceder a él
6. Devolver el control al punto de origen

# Funciones en un lenguaje de alto nivel

## Pasos en la ejecución de una función

```
int main() {  
    int z;  
    x=3;  
    z=factorial(x);  
    print_int(z);  
}
```

```
int factorial(int x) {  
    int i;  
    int r=1;  
    for (i=1;i<=x;i++) {  
        r*=i;  
    }  
    return r;  
}
```

Variables locales



1. Situar los parámetros en un lugar donde la función pueda accederlos
2. Transferir el control a la función
3. **Adquirir los recursos de almacenamiento necesarios para la función**
4. Realizar la tarea deseada
5. Guardar el resultado donde la función llamante pueda acceder a él
6. Devolver el control al punto de origen



# Funciones en un lenguaje de alto nivel

## Pasos en la ejecución de una función

```
int main() {  
    int z;  
    x=3;  
    z=factorial(x);  
    print_int(z);  
}
```

```
int factorial(int x) {  
    int i;  
    int r=1;  
    for (i=1;i<=x;i++) {  
        r*=i;  
    }  
    return r;  
}
```

4



1. Situar los parámetros en un lugar donde la función pueda accederlos
2. Transferir el control a la función
3. Adquirir los recursos de almacenamiento necesarios para la función
4. Realizar la tarea deseada
5. Guardar el resultado donde la función llamante pueda acceder a él
6. Devolver el control al punto de origen

# Funciones en un lenguaje de alto nivel

## Pasos en la ejecución de una función

```
int main() {  
    int z;  
    x=3;  
    z=factorial(x);  
    print_int(z);  
}
```

```
int factorial(int x) {  
    int i;  
    int r=1;  
    for (i=1;i<=x;i++) {  
        r*=i;  
    }  
    return r;  
}
```

5

1. Situar los parámetros en un lugar donde la función pueda accederlos
2. Transferir el control a la función
3. Adquirir los recursos de almacenamiento necesarios para la función
4. Realizar la tarea deseada
5. **Guardar el resultado donde la función llamante pueda acceder a él**
6. Devolver el control al punto de origen

# Funciones en un lenguaje de alto nivel

## Pasos en la ejecución de una función

```
int main() {  
    int z;  
    x=3;  
    z=factorial(x);  
    print_int(z);  
}
```

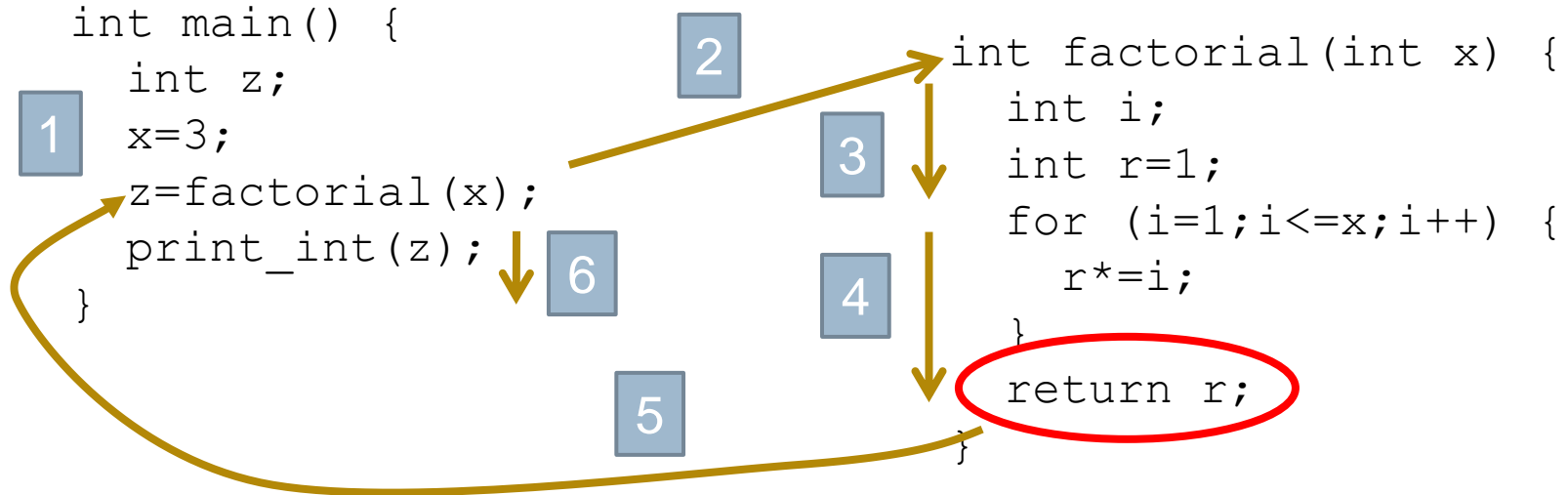
↓ 6

```
int factorial(int x) {  
    int i;  
    int r=1;  
    for (i=1;i<=x;i++) {  
        r*=i;  
    }  
    return r;  
}
```

1. Situar los parámetros en un lugar donde la función pueda accederlos
2. Transferir el control a la función
3. Adquirir los recursos de almacenamiento necesarios para la función
4. Realizar la tarea deseada
5. Guardar el resultado donde la función llamante pueda acceder a él
6. **Devolver el control al punto de origen**

# Pasos en la ejecución de una función de alto nivel

## resumen



1. Situar los parámetros en un lugar donde la función pueda accederlos
2. Transferir el control a la función
3. Adquirir los recursos de almacenamiento necesarios para la función
4. Realizar la tarea deseada
5. Guardar el resultado donde la función llamante pueda acceder a él
6. Devolver el control al punto de origen

# Procedimientos y funciones

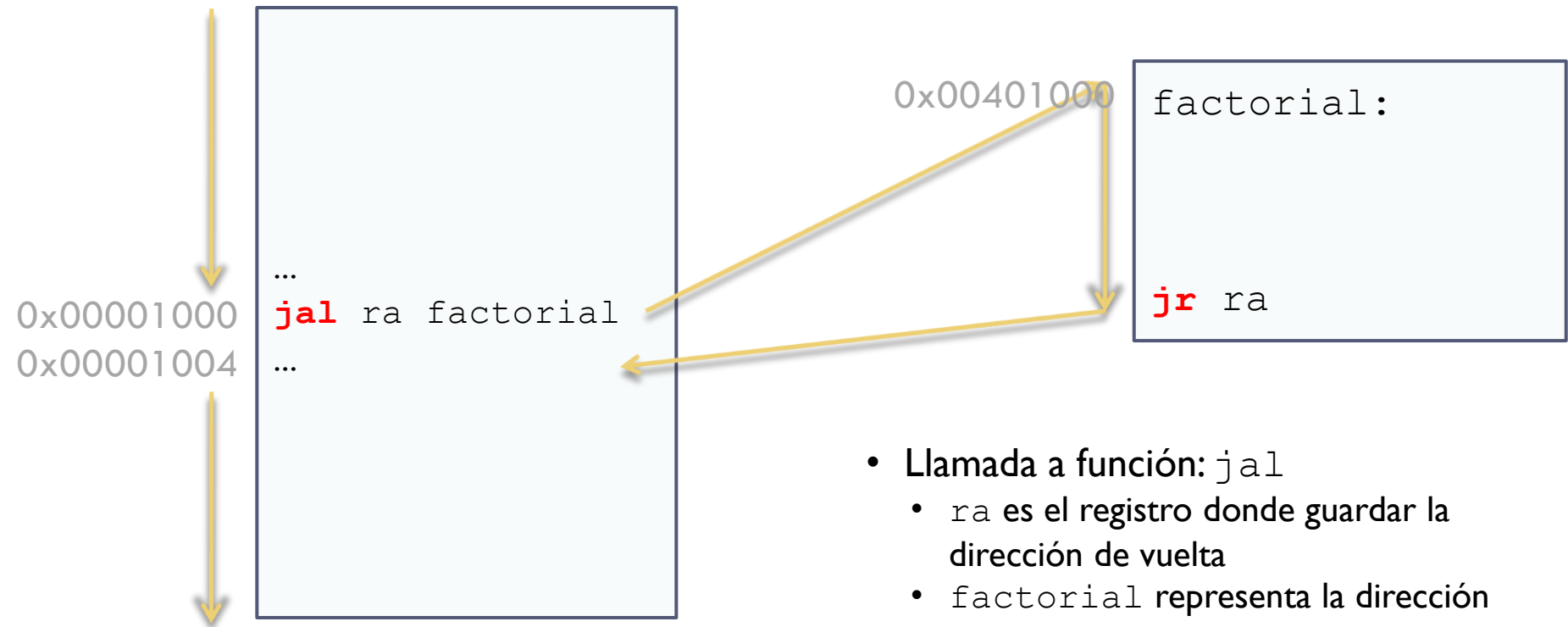
```
int factorial(int x) {
    int i;
    int r=1;
    for (i=1;i<=x;i++) {
        r*=i;
    }
    return r;
}
...
r1 = factorial(3) ;
...
```

- ▶ Un función (procedimiento, método) en alto nivel es un subprograma que realiza una tarea específica cuando se le invoca
  - ▶ Recibe argumentos o parámetros de entrada
  - ▶ Devuelve algún resultado

```
factorial:
    mv    t0 a0
    li    v0 1
b1: beq   t0 zero f1
    mul   v0 v0 t0
    addi  t0 t0 -1
    j     b1
f1: jr   ra
...
li    a0 3
jal  ra factorial
...
```

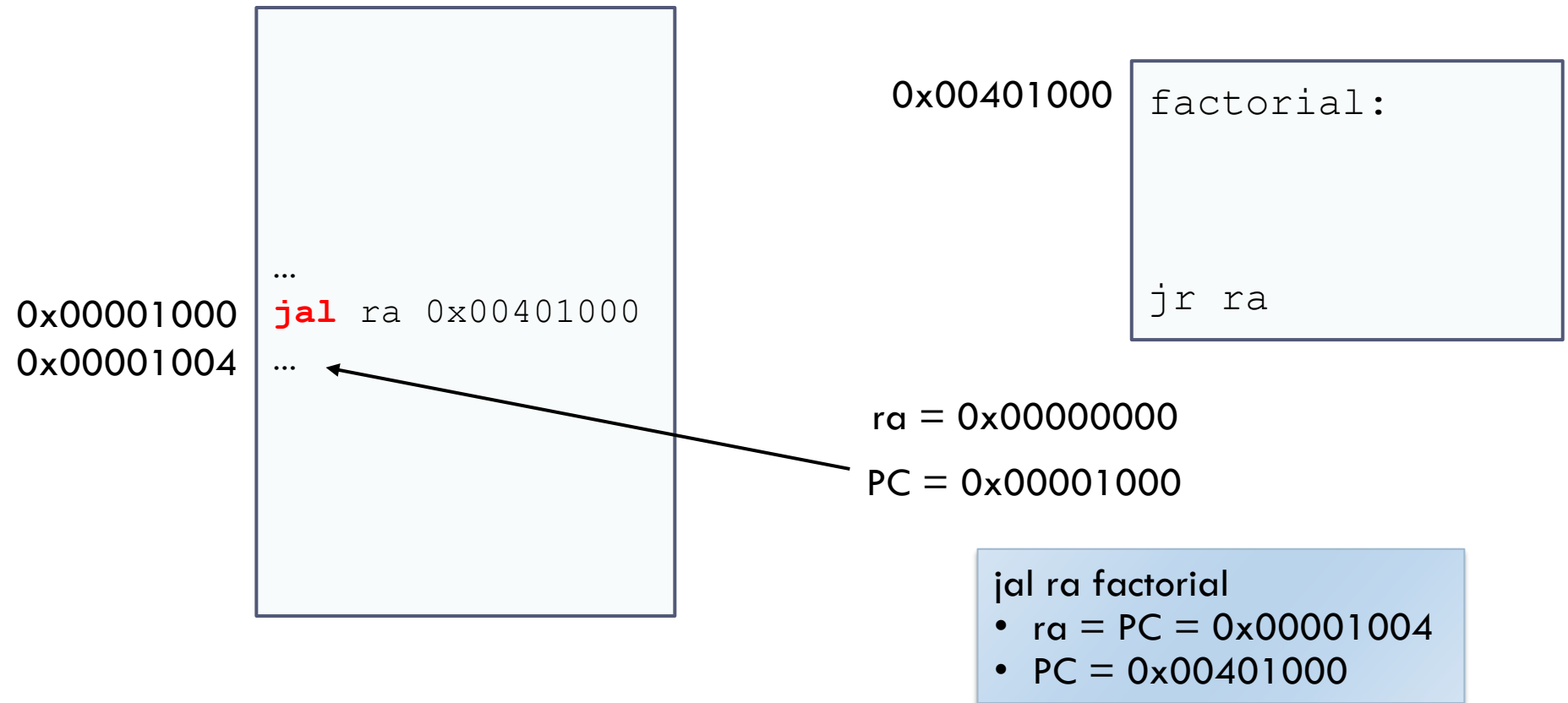
- ▶ En ensamblador una función (subrutina) se asocia con una etiqueta en la primera instrucción de la función
  - ▶ Nombre simbólico que denota su dirección de inicio
  - ▶ La dirección de memoria donde se encuentra la primera instrucción

# Llamadas a funciones en RISC-V

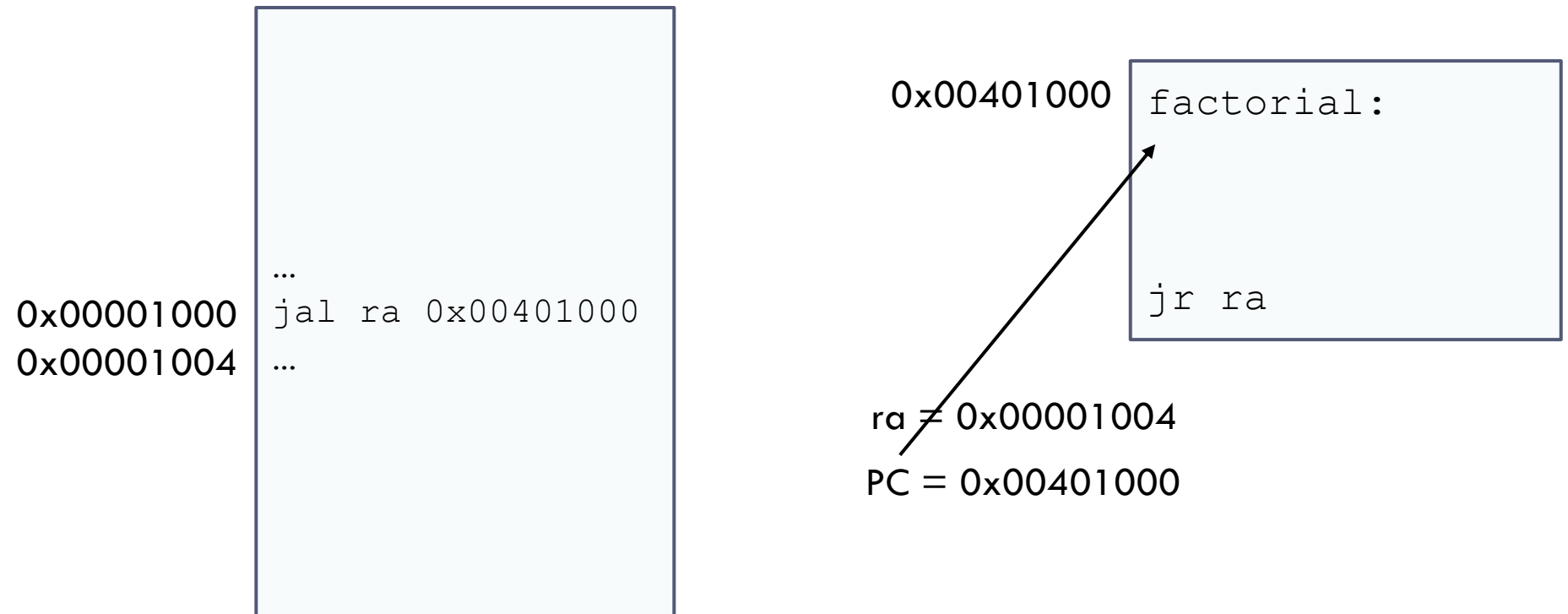


- **Llamada a función: `jal`**
  - `ra` es el registro donde guardar la dirección de vuelta
  - `factorial` representa la dirección de inicio de la subrutina/función
- **Retorno de subrutina: `jr`**

# Llamadas a funciones en RISC-V

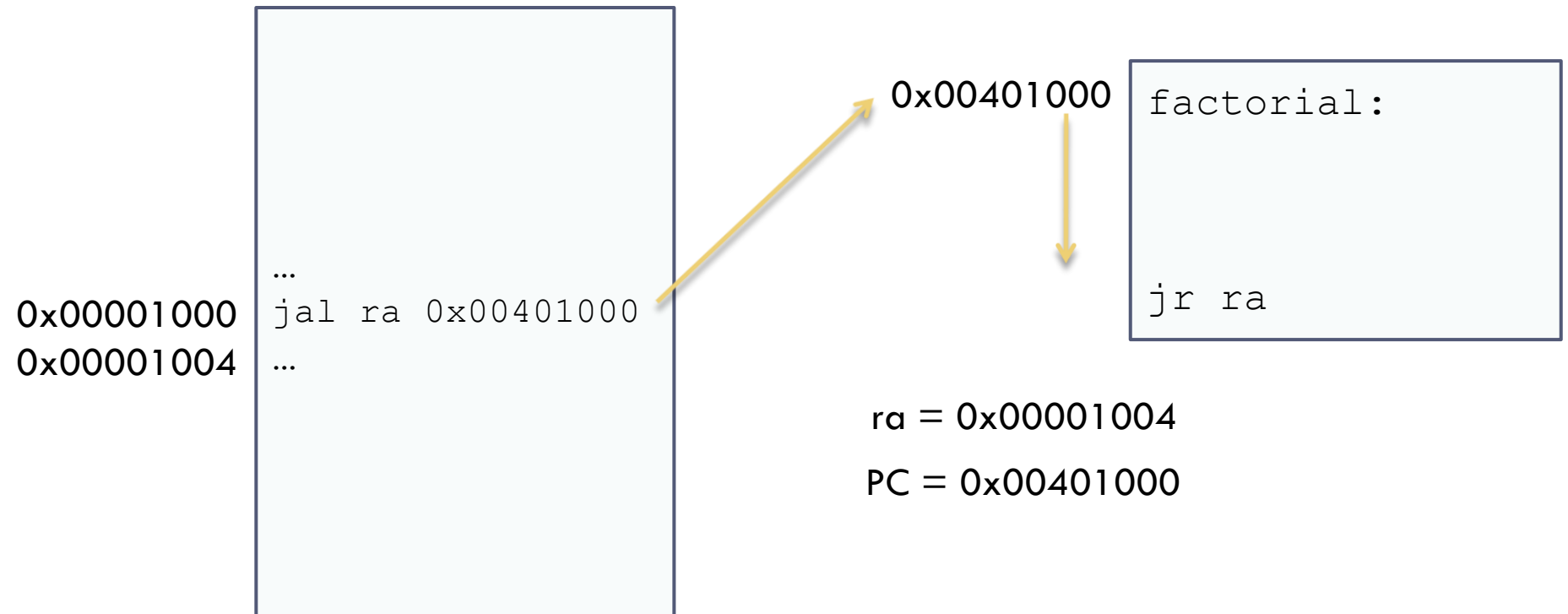


# Llamadas a funciones en RISC-V

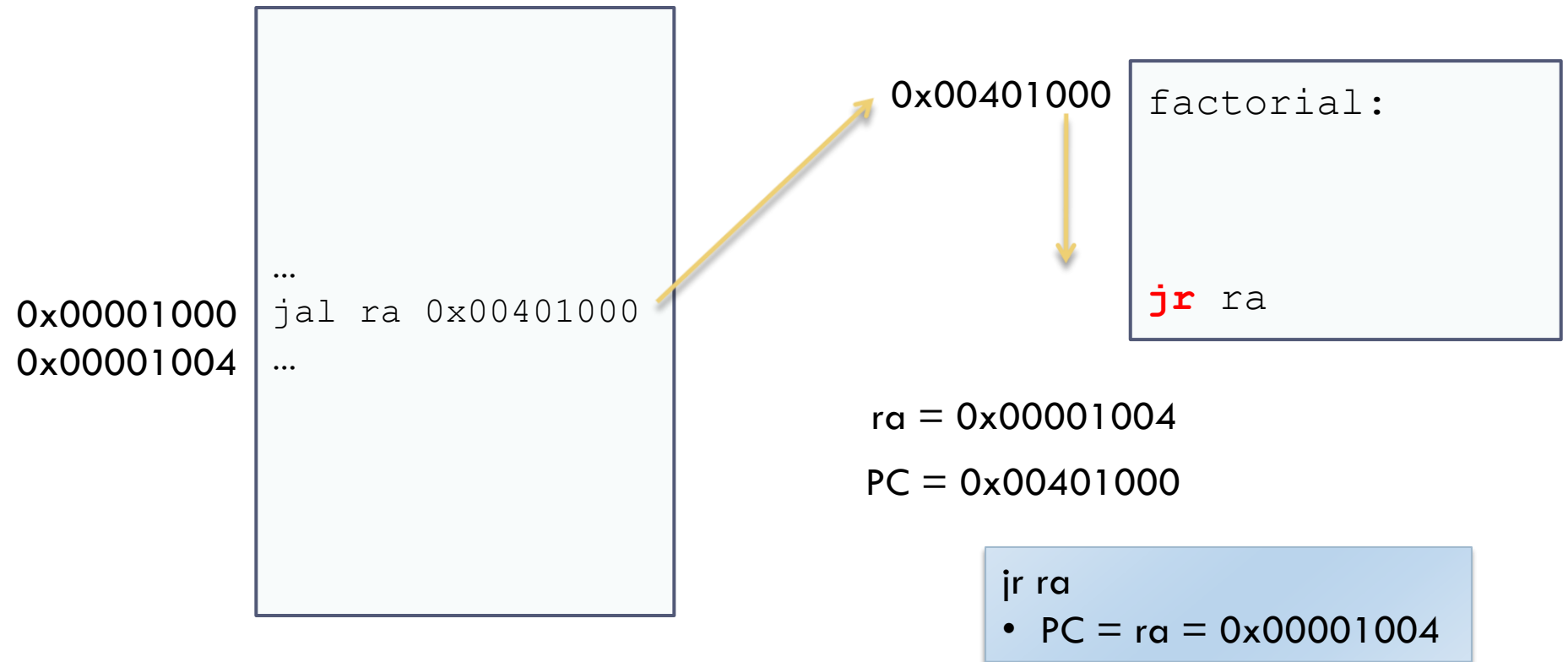




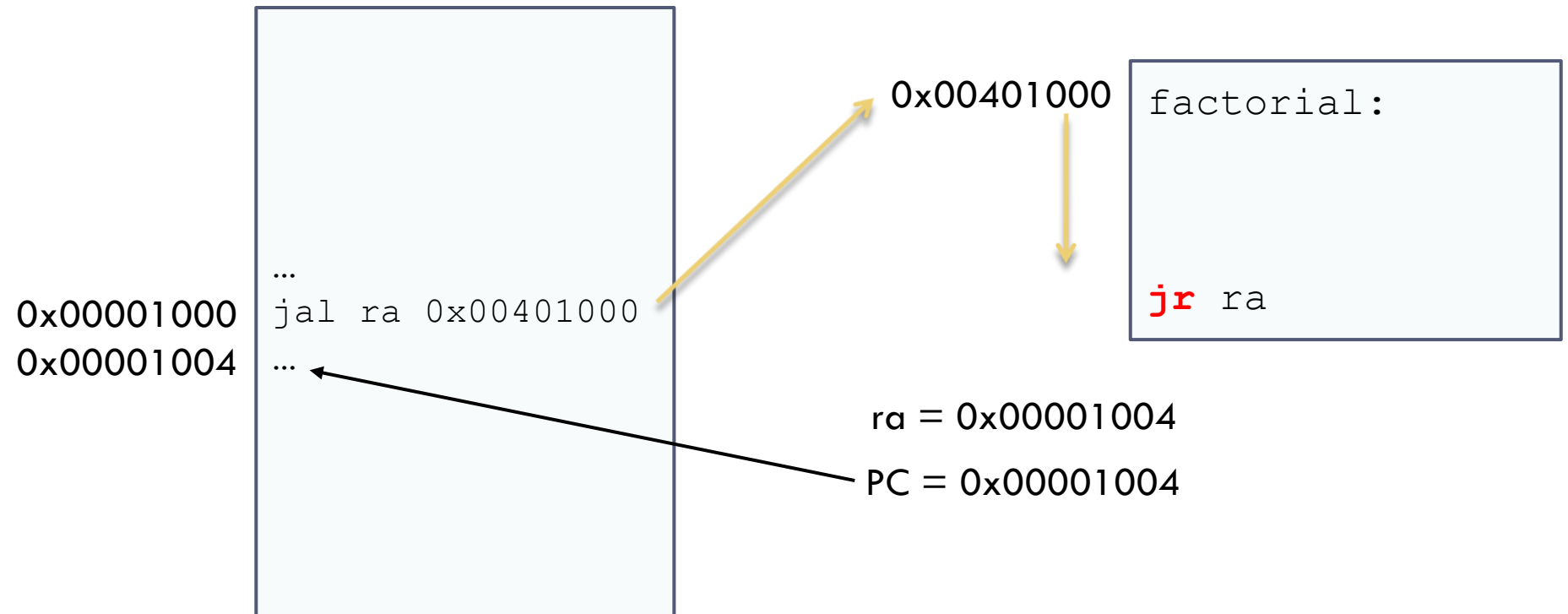
# Llamadas a funciones en RISC-V



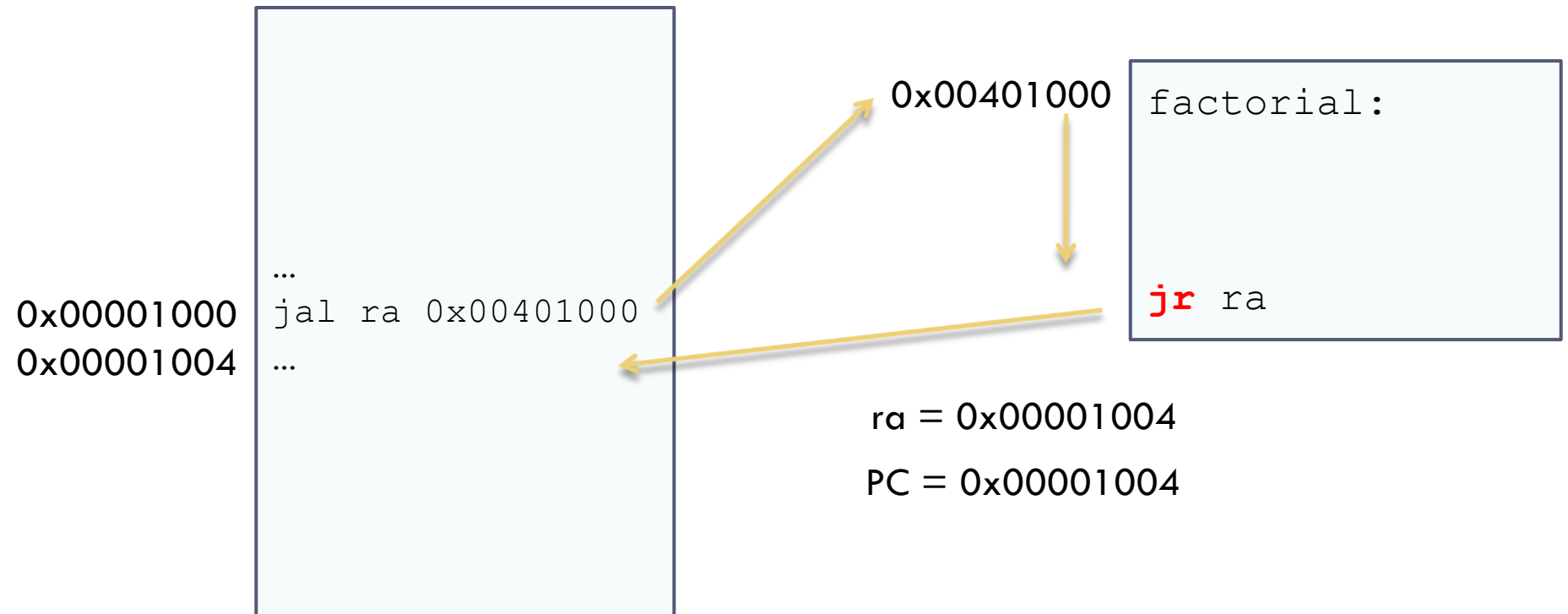
# Llamadas a funciones en RISC-V



# Llamadas a funciones en RISC-V



# Llamadas a funciones en RISC-V



# Instrucciones jal/jr

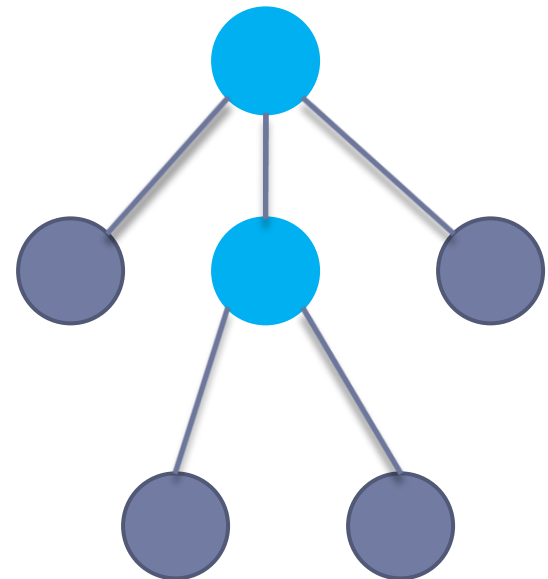
Subrutinas / Funciones		
<code>jal reg2, label</code>	<code>reg2 = PC</code> <code>PC = label</code>	<ul style="list-style-type: none"><li>• Carga en el registro reg2 el contenido de PC. Cuando se ejecuta la instrucción jal PC apunta al primer byte de la siguiente instrucción.</li><li>• Calcula y carga en PC la dirección de memoria que la etiqueta label representa. La siguiente instrucción a ejecutar será la apuntada por PC.</li></ul>
<code>jr reg1</code>	<code>PC = reg1</code>	<ul style="list-style-type: none"><li>• Guarda en PC el valor guardado en el registro reg1.</li></ul>

# Contenido

- ▶ Fundamentos básicos de la programación en ensamblador
- ▶ Ensamblador del RISC-V 32, modelo de memoria y representación de datos
- ▶ Formato de las instrucciones y modos de direccionamiento
- ▶ **Llamadas a procedimientos y uso de la pila**
  - ▶ ¿Cómo se llama a una función/subrutina?
  - ▶ ¿Dónde guardar la dirección de retorno en rutinas no terminales?
  - ▶ ¿Cuál es el convenio de paso de parámetros?
  - ▶ ¿Cuál es el convenio de uso de registros?
  - ▶ ¿Cómo son las variables locales?

# Tipos de subrutinas

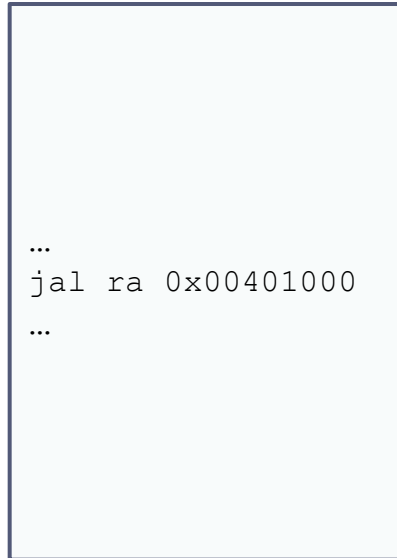
- **Subrutina terminal.**
  - ▶ **No** invoca a ninguna otra subrutina.
- **Subrutina no terminal.**
  - ▶ Sí invoca a alguna otra subrutina.



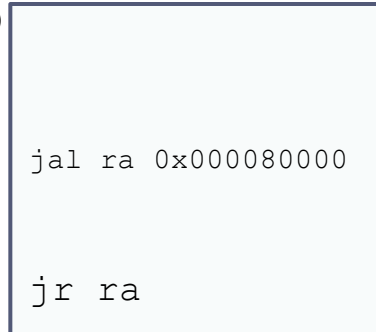
# Problema en subrutinas no terminales



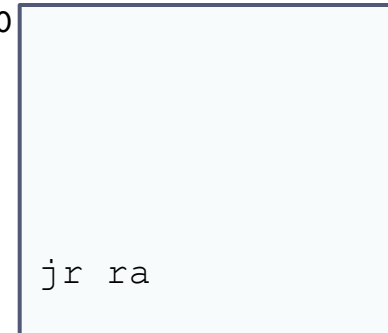
```
0x00001000 ...  
0x00001004 jal ra 0x00401000  
...
```



```
0x00401000  
jal ra 0x00008000  
jr ra
```

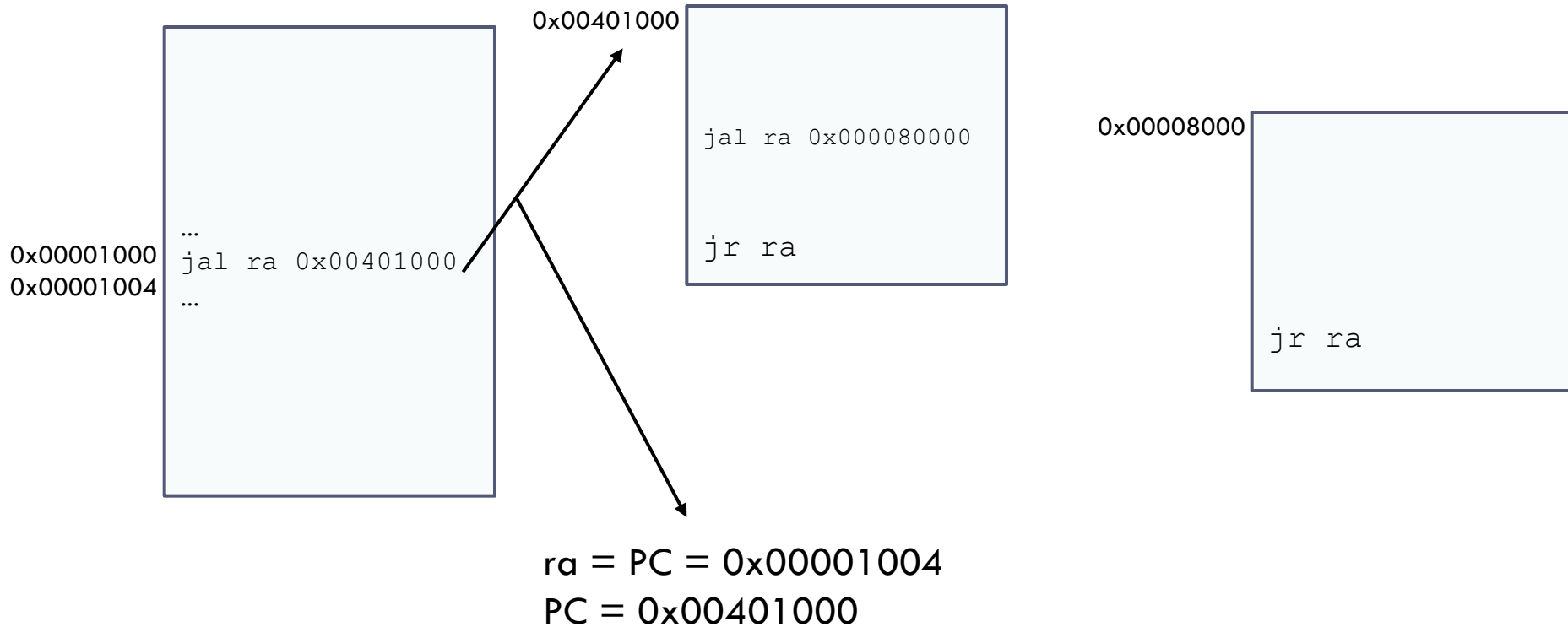


```
0x00008000  
jr ra
```



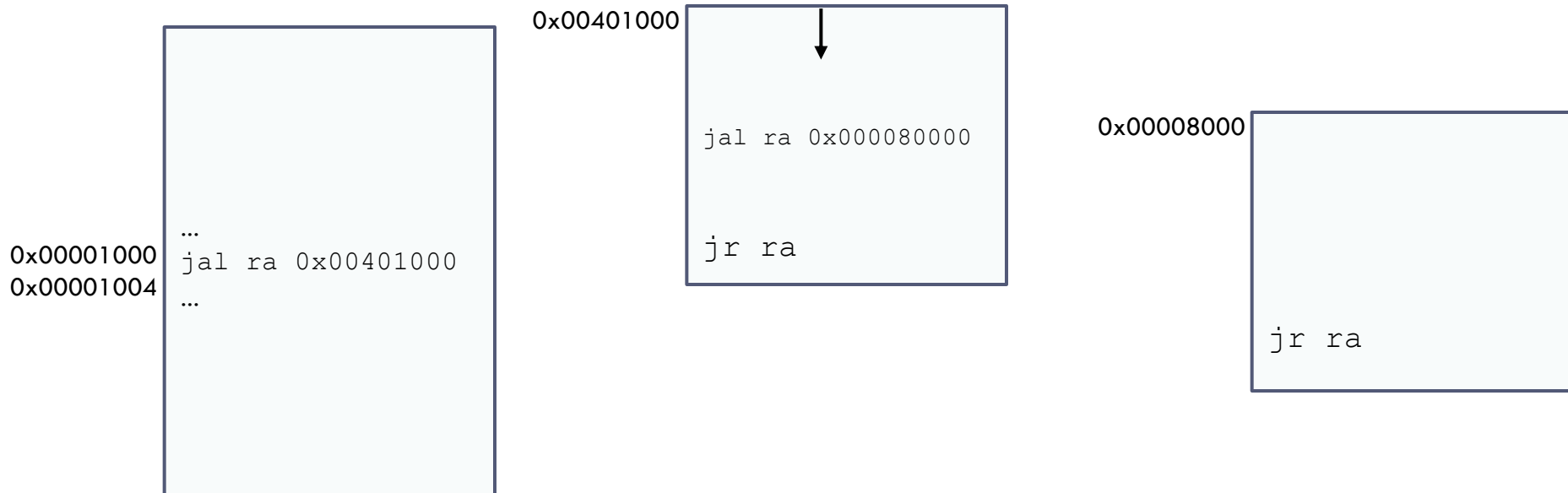


# Problema en subrutinas no terminales



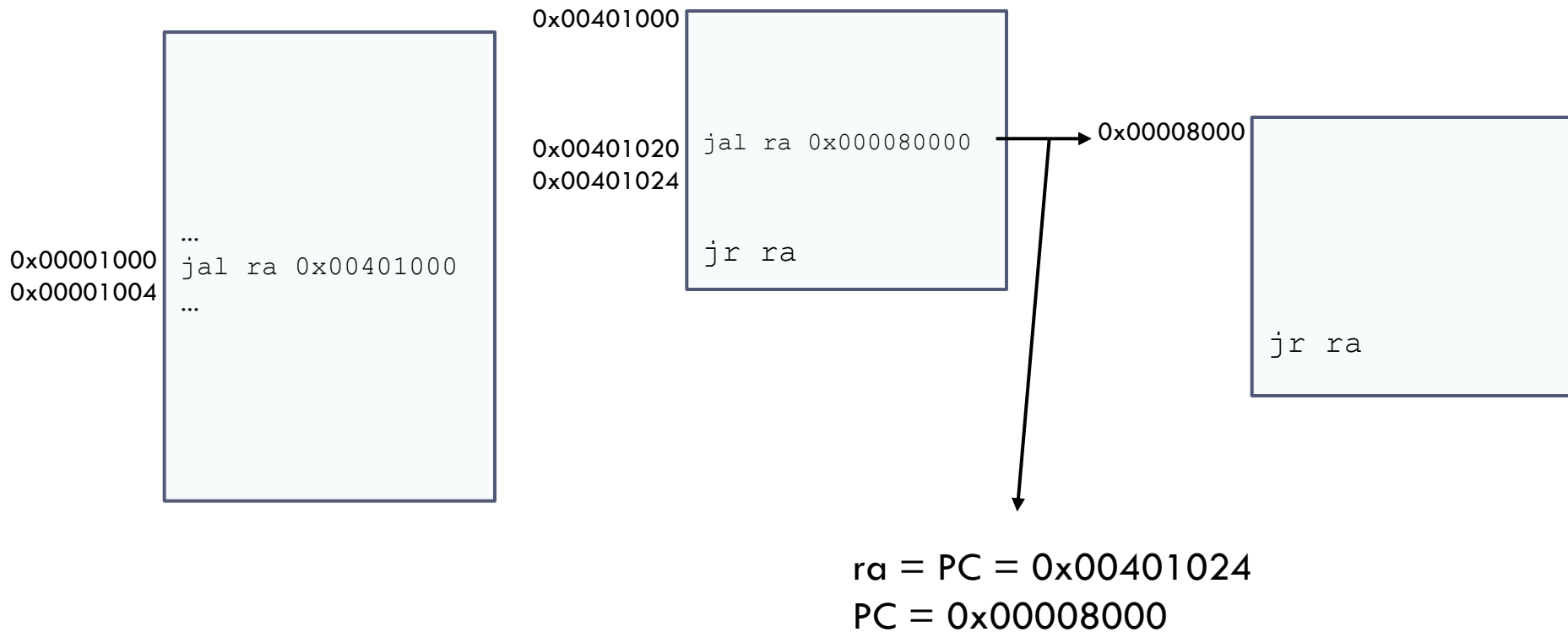
Dirección de retorno ra = PC = 0x00001004

# Problema en subrutinas no terminales



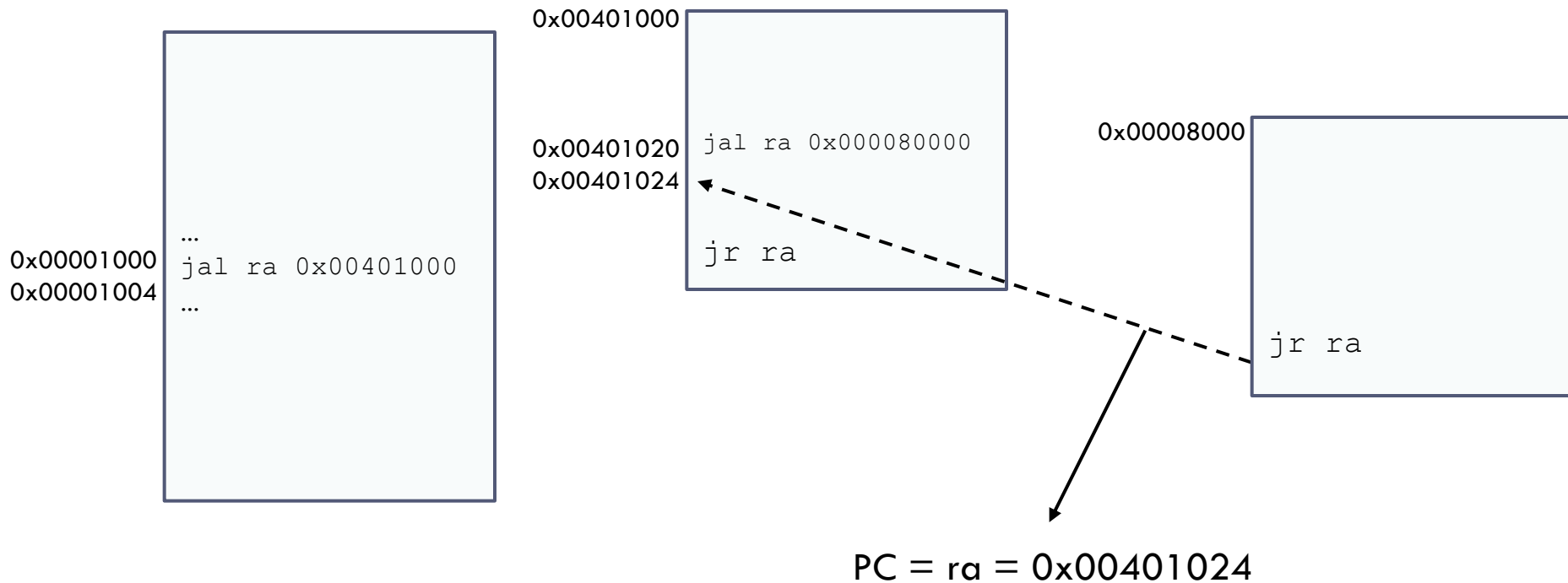
Dirección de retorno  $ra = PC = 0x00001004$

# Problema en subrutinas no terminales



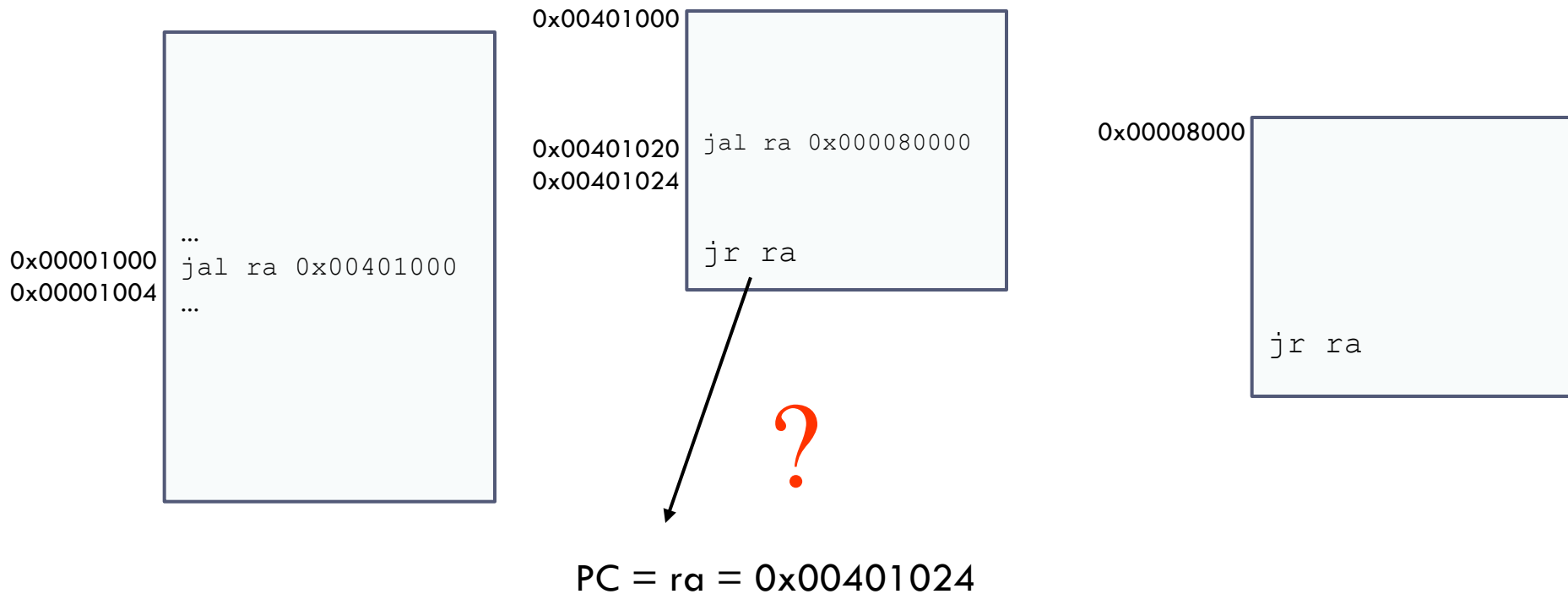
~~Dirección de retorno ra = PC = 0x00001004~~

# Problema en subrutinas no terminales

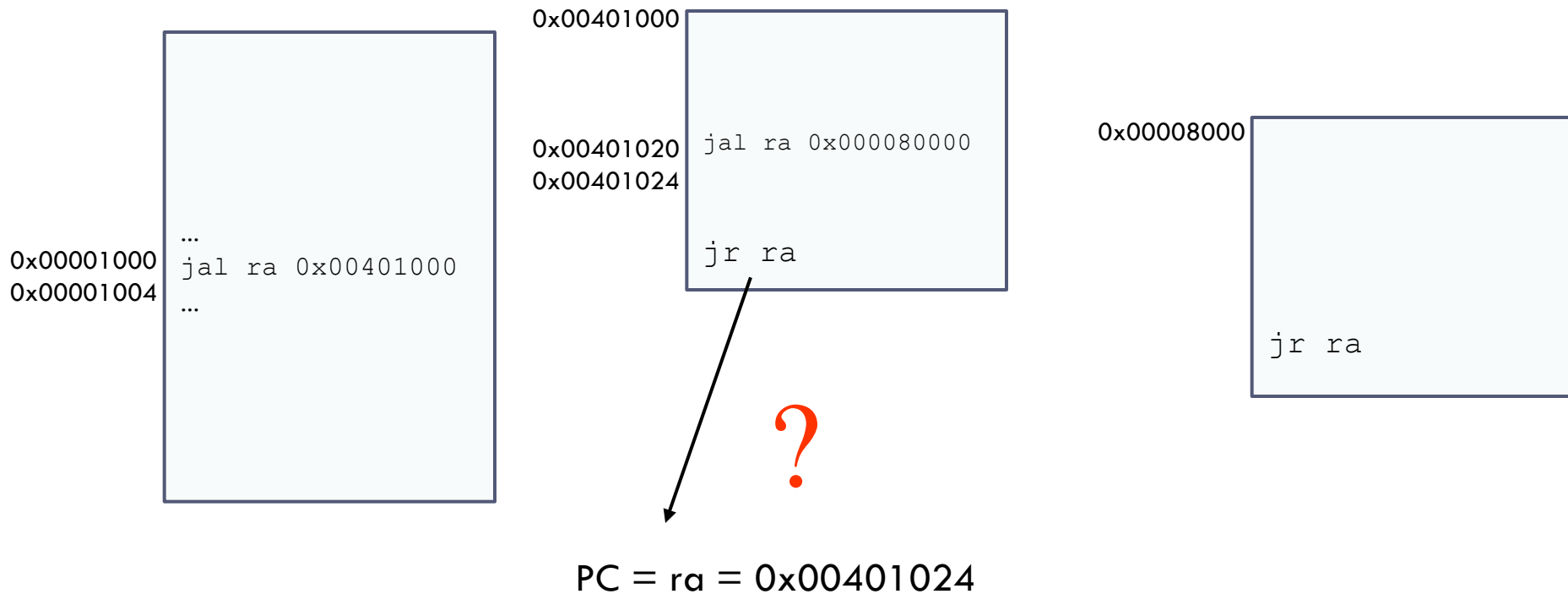


~~Dirección de retorno ra = PC = 0x00001004~~

# Problema en subrutinas no terminales



# Problema en subrutinas no terminales



Se ha perdido la dirección de retorno

# ¿Dónde guardar la dirección de retorno?

- ▶ El computador dispone para almacenamiento de:
  - ▶ Registros
  - ▶ Memoria
- ▶ Registros: No se pueden utilizar los registros porque su número es limitado (ej.: llamadas recursivas)
- ▶ Memoria: Se guarda en memoria principal
  - ▶ En una zona del programa que se denomina **pila**

# Pila, jal y jr...

IMPORTANTE

`no_terminal:`

```
addi sp sp -4  
sw ra 0(sp)
```

← Se guarda ra en la pila al principio

```
li t0, 8  
li s0, 9
```

...

```
jal ra, función
```

...

```
lw ra, 0(sp)  
addi sp, sp, 4  
jr ra
```

← Se recupera el valor antes de “jr ra”



# Ejecución de un programa

Recordatorio

`no_terminal:`

```
addi sp sp -4
sw   ra 0(sp)

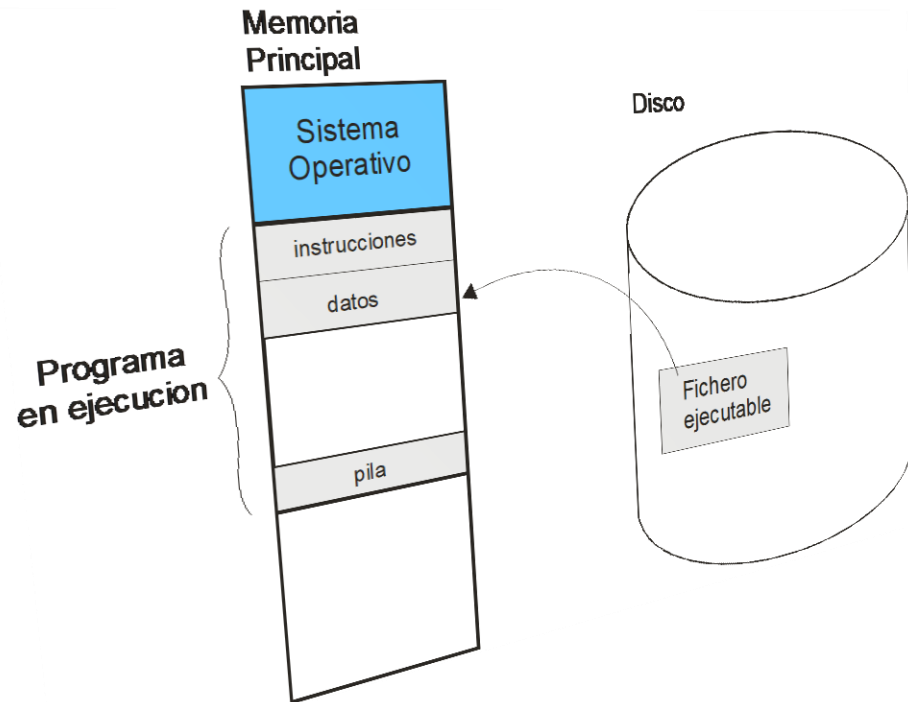
li   t0, 8
li   s0, 9

...

jal  ra, función

...

lw   ra, 0(sp)
addi sp, sp, 4
jr  ra
```



# Ejecución de un programa

Recordatorio

no\_terminal:

```
addi sp sp -4
sw   ra 0(sp)

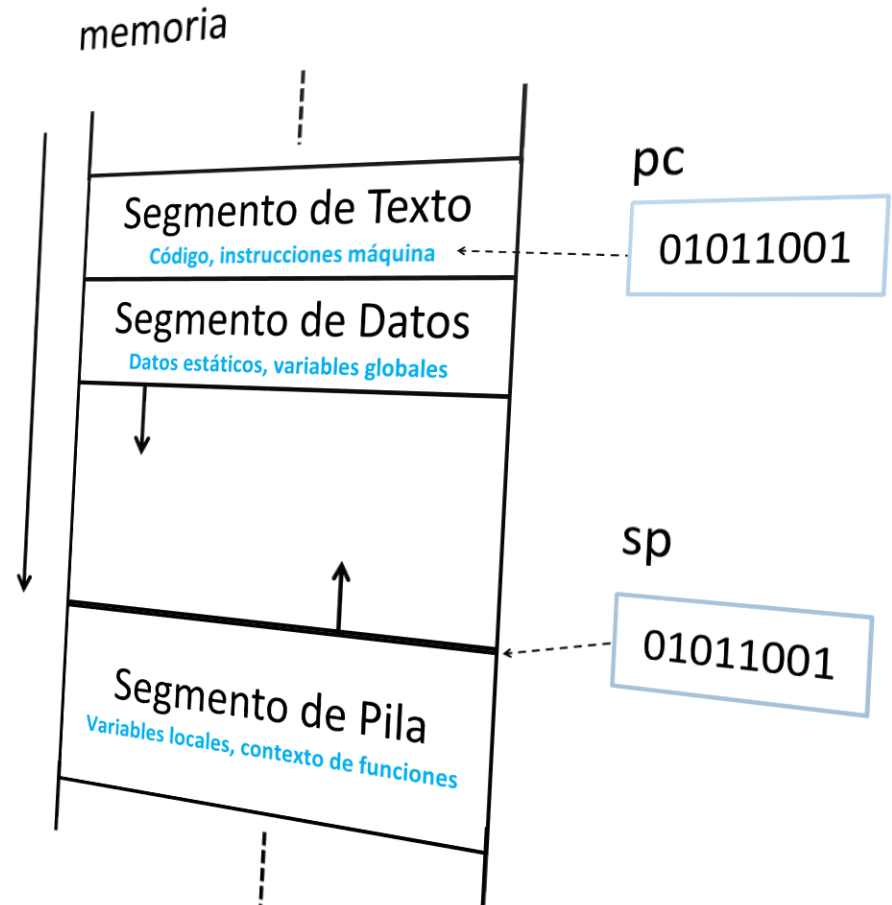
li   t0, 8
li   s0, 9

...

jal  ra, función

...

lw   ra, 0(sp)
addi sp, sp, 4
jr  ra
```



# Ejecución de un programa

`no_terminal:`

```
addi sp sp -4  
sw ra 0(sp)
```

```
li t0, 8  
li s0, 9
```

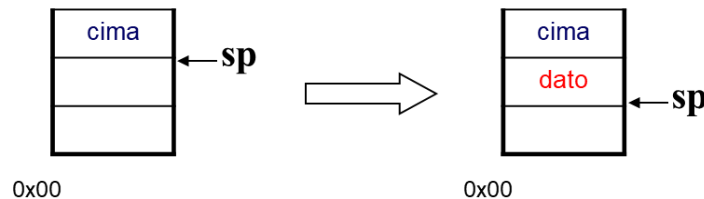
...

```
jal ra, función
```

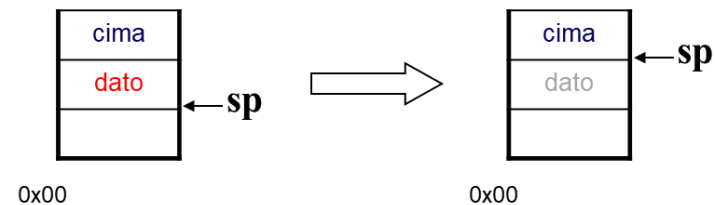
...

```
lw ra, 0(sp)  
addi sp, sp, 4  
jr ra
```

**PUSH Reg** Apila el contenido del registro (dato)



**POP Reg** Desapila el contenido del registro (dato)  
Copia dato en el registro Reg



# Operación PUSH en el RISC-V

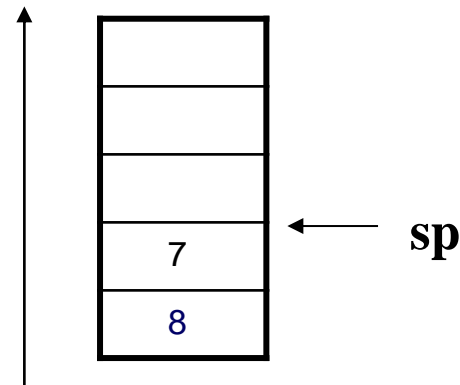
...

```
li t2, 9
```

```
addi sp, sp, -4
```

```
sw t2 0(sp)
```

...

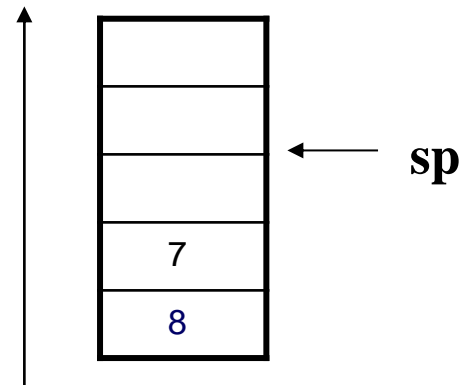


## ▶ Estado inicial:

- ▶ El registro puntero de pila (sp) apunta al último elemento situado en la cima de la pila
- ▶ El registro t2 almacena el valor 9

# Operación PUSH en el RISC-V

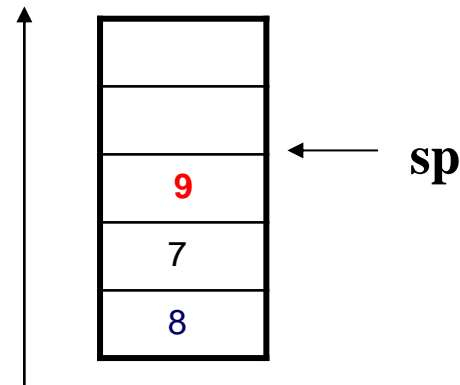
```
...  
li    t2, 9  
addi sp, sp, -4  
sw    t2 0(sp)  
...
```



- ▶ Se resta 4 al registro puntero de pila para poder insertar una nueva palabra en la pila
  - ▶ `addi sp, sp, -4`

# Operación PUSH en el RISC-V

```
...  
li    t2, 9  
addi sp, sp, -4  
sw    t2 0(sp)  
...
```



- ▶ Se inserta el contenido del registro t2 en la cima de la pila:
  - ▶ sw t2 0(sp)

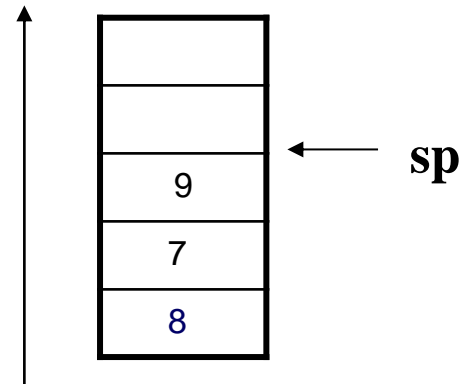
# Operación POP en el RISC-V<sub>32</sub>

...

```
lw t2 0(sp)
```

```
addi sp, sp, 4
```

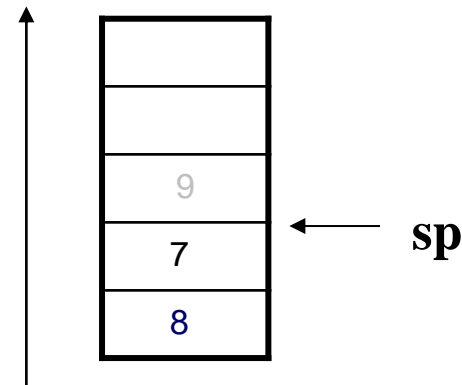
...



- ▶ Se copia en t2 el dato almacenado en la cima de la pila (9)
  - ▶ lw t2 0(sp)

# Operación POP en el RISC-V

```
...  
lw    t2 0(sp)  
addi sp, sp, 4  
...
```

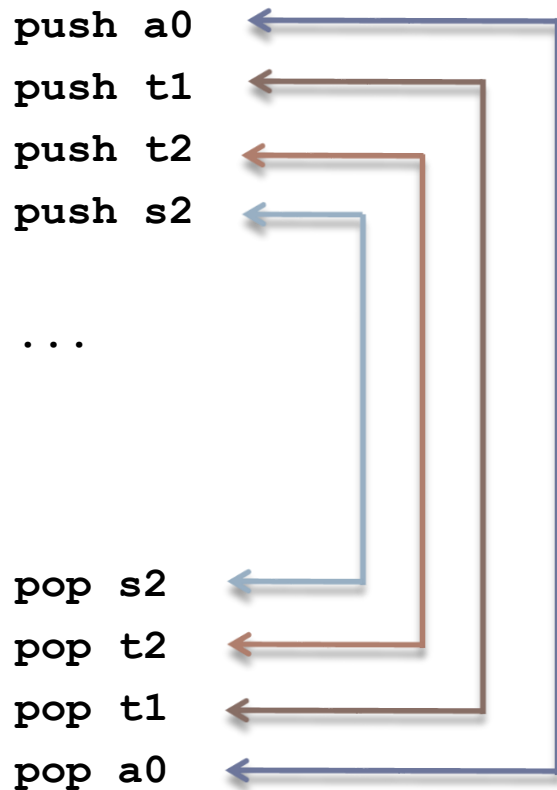


- ▶ Se actualiza el registro `sp` para apuntar a la nueva cima de la pila.
  - ▶ `addi sp, sp, 4`
- ▶ El dato desapilado (9) sigue estando en memoria pero será sobrescrito en futuras operaciones PUSH (o similar de acceso a memoria)



# Pila: uso de push y pop consecutivos

## a) desapilar en orden inverso al apilado



# Pila: uso de push y pop consecutivos

## b) es posible sumas por operación o juntar sumas

```
push a0  
push t1  
push t2  
push s2
```

...

```
pop s2  
pop t2  
pop t1  
pop a0
```

```
addi sp sp -4  
sw a0 0(sp)  
addi sp sp -4  
sw t1 0(sp)  
addi sp sp -4  
sw t2 0(sp)  
addi sp sp -4  
sw s2 0(sp)
```

...

```
lw s2 0(sp)  
addi sp sp 4  
lw t2 0(sp)  
addi sp sp 4  
lw t1 0(sp)  
addi sp sp 4  
lw a0 0(sp)  
addi sp sp 4
```

# Pila: uso de push y pop consecutivos

**b) es posible sumas por operación o juntar sumas**

```
push a0  
push t1  
push t2  
push s2
```

...

```
pop s2  
pop t2  
pop t1  
pop a0
```

```
addi sp sp -16  
sw a0 12(sp)  
sw t1 8(sp)  
sw t2 4(sp)  
sw s2 0(sp)
```

...

```
lw s2 0(sp)  
lw t2 4(sp)  
lw t1 8(sp)  
lw a0 12(sp)  
addi sp sp 16
```

# Contenido

- ▶ Fundamentos básicos de la programación en ensamblador
- ▶ Ensamblador del RISC-V 32, modelo de memoria y representación de datos
- ▶ Formato de las instrucciones y modos de direccionamiento
- ▶ **Llamadas a procedimientos y uso de la pila**
  - ▶ ¿Cómo se llama a una función/subrutina?
  - ▶ ¿Dónde guardar la dirección de retorno en rutinas no terminales?
  - ▶ ¿Cuál es el convenio de paso de parámetros?
  - ▶ ¿Cuál es el convenio de uso de registros?
  - ▶ ¿Cómo son las variables locales?

# Convenio de parámetros y registros

```
no_terminal:
```

```
addi sp sp -4  
sw ra 0(sp)
```

```
li t0, 8  
li s0, 9
```


```
...
```

```
jal ra, función
```

```
...
```

```
lw ra, 0(sp)  
addi sp, sp, 4  
jr ra
```

¿En qué registros se pasa los parámetros y devuelve los resultados?



# Convenio de paso de parámetros

- ▶ Cuando se programa en ensamblador se define un convenio que especifica cómo se pasan los argumentos y cómo se tratan los registros
- ▶ Los compiladores definen este convenio para una determinada arquitectura
- ▶ En la asignatura se va a utilizar una versión simplificada de los convenios que utilizan los compiladores

# Convenio simplificado (RISC-V)

IMPORTANTE

## ▶ Paso de parámetros

- ▶ Los parámetros **enteros** (char, int) se pasan en **a0 ... a7**
  - ▶ Si se necesita pasar más de ocho parámetros, los ocho primeros en los registros a0 ... a7 y el resto en la pila
- ▶ Los parámetros **float** se pasan en **fa0 ... fa7**
  - ▶ Si se necesita pasar más de ocho parámetros, el resto en la pila
- ▶ Los parámetros **double** se pasan en **fa0 ... fa7**
  - ▶ Si se necesita pasar más de ocho parámetros, el resto en la pila

## ▶ Retorno de resultados

- ▶ Se usa **a0** y **a1** para valores de tipo entero
- ▶ Se usa **fa0** y **fa1** para valores de tipo float y double
- ▶ En caso de estructuras o valores complejos han de dejarse en pila. El espacio lo reserva la función que realiza la llamada (llamante)

# Contenido

- ▶ Fundamentos básicos de la programación en ensamblador
- ▶ Ensamblador del RISC-V 32, modelo de memoria y representación de datos
- ▶ Formato de las instrucciones y modos de direccionamiento
- ▶ **Llamadas a procedimientos y uso de la pila**
  - ▶ ¿Cómo se llama a una función/subrutina?
  - ▶ ¿Dónde guardar la dirección de retorno en rutinas no terminales?
  - ▶ ¿Cuál es el convenio de paso de parámetros?
  - ▶ ¿Cuál es el convenio de uso de registros?
  - ▶ ¿Cómo son las variables locales?



# Convenio de parámetros y registros

```
no_terminal:
```

```
    addi sp, sp, -4  
    sw   ra, 0(sp)
```

```
    li   t0, 8  
    li   s0, 9
```


```
    ...
```

```
    jal ra, función
```

```
    ...
```

```
    lw   ra, 0(sp)  
    addi sp, sp, 4  
    jr  ra
```

¿Qué valores tienen los registros t0 y s0 a la vuelta?



# Convención uso de registros (RISC-V)

**IMPORTANTE**

Nombre	Uso	Preservar el valor
zero	Constante 0	No
ra	Dirección de retorno (rutinas)	<b>Si</b>
sp	Puntero a pila	<b>Si</b>
gp	Puntero al área global	No
tp	Puntero al hilo	No
t0 ... t6	Temporal	No
s0/fp	Temporal / Puntero a marco de pila	<b>Si</b>
s1 ... s11	Temporal	<b>Si</b>
a0 ... a7	Argumento de entrada para rutinas	No

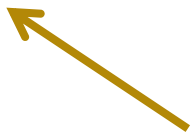
Nombre	Uso	Preservar el valor
ft0 ... ft11	Temporales	No
fs0 ... fs11	Temporales a guardar	<b>Si</b>
fa0 ... fa1	Argumentos/retorno	No
fa2 ... fa7	Argumentos	No

# Convenio de registros

```
li    t0, 8
li    s0, 9

li    a0, 7    # parámetro
jal   ra, función
```

...



De acuerdo al convenio:

- **s0** seguirá valiendo 9,
  - no hay garantía de que **t0** valga 8
  - ni que **a0** valga 7 tras la ejecución de la función.
- Si queremos que **t0** siga valiendo 8 habrá que guardarse en la pila antes de cada llamada a función.

# Convenio de registros

```
li    t0, 8  
li    s0, 9
```

```
addi  sp, sp, -4  
sw    t0, 0(sp)
```



Se guarda en la pila antes de la llamada

```
li    a0, 7    # parámetro  
jal   ra, función
```

```
lw    t0, 0(sp)  
addi  sp, sp, 4
```



Se recupera el valor después

...

# Convenio de parámetros y registros

**resumen**

no\_terminal:

```
li s0, 9  
li t0, 8
```

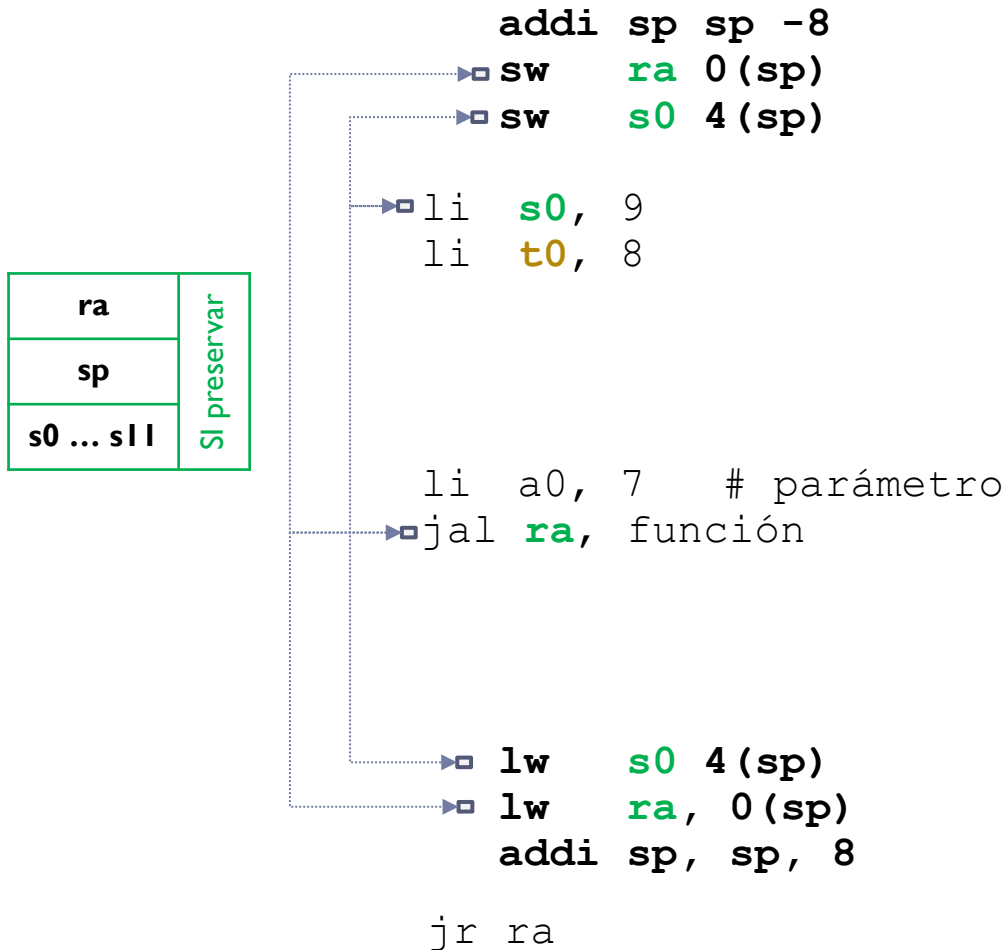
```
li a0, 7 # parámetro  
jal ra, función
```

```
jr ra
```

# Convenio de parámetros y registros

resumen

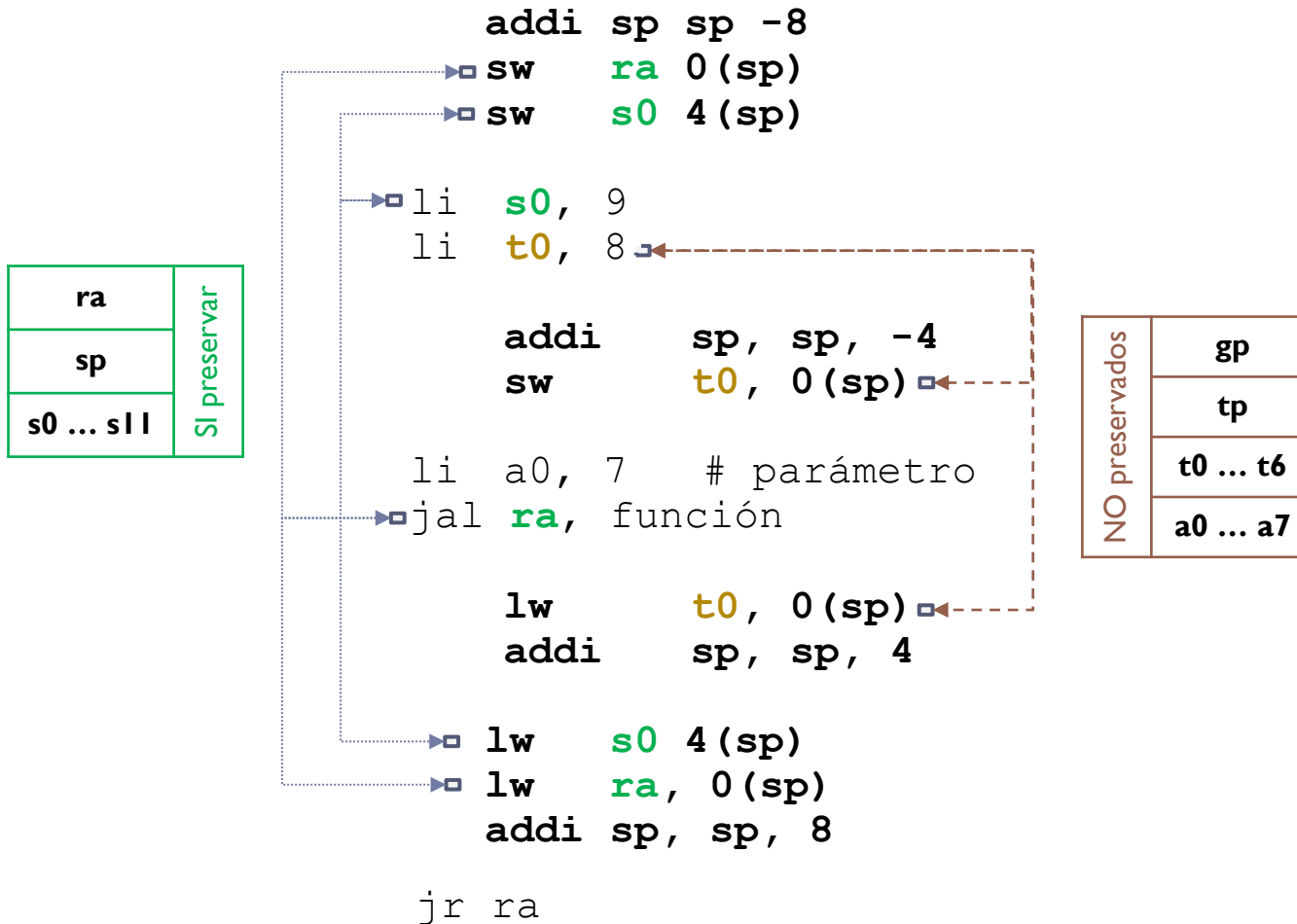
no\_terminal:



# Convenio de parámetros y registros

resumen

no\_terminal:



# Convenio de parámetros y registros

**resumen**

no\_terminal:

- Variables locales en pila.
- Ejemplo que reserva de 4 bytes (un entero)

ra	Si preservar
sp	
s0 ... s11	

```

    addi sp, sp, -12
    sw ra, 0(sp)
    sw s0, 4(sp)

    li s0, 9
    li t0, 8

    addi sp, sp, -4
    sw t0, 0(sp)

    li a0, 7 # parámetro
    jal ra, función

    lw t0, 0(sp)
    addi sp, sp, 4

    lw s0, 4(sp)
    lw ra, 0(sp)
    addi sp, sp, 12

    jr ra
  
```


NO preservados	gp
	tp
	t0 ... t6
	a0 ... a7



# Ejemplo

(1) Se parte de un código en lenguaje de alto nivel

```
int main() {  
    int z;  
    z=factorial(5);  
    print_int(z);  
    .  
    .  
    .  
}  
  
int factorial(int x) {  
    int i;  
    int r=1;  
    for (i=1;i<=x;i++) {  
        r*=i;  
    }  
    return r;  
}
```



# Ejemplo

## (2) Pensar en el paso de parámetros

- ▶ Los **parámetros** en RISC-V se pasarán en `a0 ... a7`
  - ▶ `z=factorial(5)` tiene un parámetro de entrada en `a0`
- ▶ Los **resultados** en RISC-V se recogen en `a0, a1`
  - ▶ `z=factorial(5)` devuelve un resultado en `a0`
- ▶ Si se necesita pasar más de ocho parámetros enteros,
  - (1) los ocho primeros en los registros `a0...a7` y
  - (2) el resto en la pila
  - ▶ No se precisa más de ocho parámetros

# Ejemplo

(3) Se pasa a ensamblador cada función

```
int main() {  
    int z;  
    z=factorial(5);  
    print_int(z);  
    . . .  
}  
  
int factorial(int x) {  
    int i;  
    int r=1;  
    for (i=1;i<=x;i++) {  
        r*=i;  
    }  
    return r;  
}
```

→ main: # factorial(5)  
li a0, 5 # arg.  
jal ra factorial # invoke  
mv a0 a0 # result  
# print\_int(z)  
li a7, 1  
ecall  
...

→ factorial: li s1, 1 #s1 for r  
li s0, 1 #s0 for i  
loop1: bgt s0, a0, end1  
mul s1, s1, s0  
addi s0, s0, 1  
j loop1  
end1: mv a0, s1 #result  
jr ra

- El parámetro se pasa en a0
- El resultado se devuelve en a0

# Ejemplo

(4) Se analizan los registros que se modifican (1/2)

```
int main() {
  int z;
  z=factorial(5);
  print_int(z);
}
main:
# factorial(5)
li a0, 5      # arg.
jal ra factorial # invoke
mv a0 a0      # result
# print_int(z)
li a7, 1
ecall
...

int factorial(int x) {
  int i;
  int r=1;
  for (i=1;i<=x;i++) {
    r*=i;
  }
  return r;
}
factorial: li s1, 1 #s1 for r
           li s0, 1 #s0 for i
loop1:    bgt s0, a0, end1
           mul s1, s1, s0
           addi s0, s0, 1
           j loop1
end1:    mv a0, s1 #result
           jr ra
```

- Main es no terminal (hay un jal... que llama a otra función/subrutina)
- Se modifica por tanto ra

# Ejemplo

(4) Se analizan los registros que se modifican (2/2)

```
int main
  int z;
  z=facto
  print_
  . . .
}
```

- La función factorial trabaja (modifica) con los registros s0, s1
- Si estos registros se modifican dentro de la función, podría afectar a la función que realizó la llamada (la función main)
- Por tanto, la función factorial debe guardar el valor de estos registros en la pila al principio y restaurarlos al final

```
int factorial(int x) {
  int i;
  int r=1;
  for (i=1;i<=x;i++) {
    r*=i;
  }
  return r;
}
```



```
...
factorial: li    s1, 1    #s1 for r
           li    s0, 1    #s0 for i
loop1:    bgt   s0, a0, end1
           mul   s1, s1, s0
           addi  s0, s0, 1
           j     loop1
end1:    mv    a0, s1    #result
           jr   ra
```

# Ejemplo

(5) Se guardan los registros necesarios en la pila (1/2)

```
int main() {  
    int z;  
    z=factorial(5);  
}
```

- Si es necesario guardar ra.
- La rutina main es no terminal
- No hay que guardar s0...s11 ni t0...t6

```
int factorial(int x) {  
    int i;  
    int r=1;  
    for (i=1;i<=x;i++) {  
        r*=i;  
    }  
    return r;  
}
```

main:

```
addi sp sp -4  
sw ra 0(sp)  
# factorial(5)  
li a0, 5 # arg.  
jal ra factorial # invoke  
mv a0 a0 # result  
# print_int(z)  
li a7, 1  
ecall  
...  
lw ra 0(sp)  
add sp sp 4  
jr ra
```

factorial:

```
addi sp, sp, -8  
sw s0, 4(sp)  
sw s1, 0(sp)  
li s1, 1 # s1 para r  
li s0, 1 # s0 para i  
loop1: bgt s0, a0, endl  
mul s1, s1, s0  
addi s0, s0, 1  
j loop1  
endl: mv a0, s1 # resultado  
lw s1, 0(sp)  
lw s0, 4(sp)  
addi sp, sp, 8  
jr ra
```

# Ejemplo

(5) Se guardan los registros necesarios en la pila (2/2)

```
int main() {  
    int z;  
    z=factorial(5);  
    print_int(z);  
    . . .  
}
```

main:

```
addi sp sp -4  
sw ra 0(sp)  
# factorial(5)  
li a0, 5 # arg.  
jal ra factorial # invoke  
mv a0 a0 # result  
# print_int(z)  
li a7, 1  
ecall  
...  
lw ra 0(sp)  
add sp sp 4  
jr ra
```

- No es necesario guardar ra.
- La rutina factorial es terminal
- Se guarda en la pila s0 y s1 (se modifican)
- Usando t0 y t1 no habría hecho falta

```
    r*=i;  
}  
return r;  
}
```

```
factorial:  
addi sp, sp, -8  
sw s0, 4(sp)  
sw s1, 0(sp)  
li s1, 1 # s1 para r  
li s0, 1 # s0 para i  
loop1: bgt s0, a0, end1  
mul s1, s1, s0  
addi s0, s0, 1  
j loop1  
end1: mv a0, s1 # resultado  
lw s1, 0(sp)  
lw s0, 4(sp)  
addi sp, sp, 8  
jr ra
```

# Ejemplo 2

```
int main()
{
    int z;

    z=f1(5, 2);

    pint(z);
}

int f1(int a, int b)
{
    int r;

    r = a+a+f2(b);
    return r;
}

int f2(int c)
{
    int s;

    s = c * c * c;
    return s;
}
```

The diagram illustrates the flow of control between functions. A box highlights the function call `f1(5, 2)` in the `main` function. A yellow arrow points from this call to the definition of `f1`. Inside the `f1` function, a box highlights the call `f2(b)`. A yellow arrow points from this call to the definition of `f2`.




## Ejemplo 2. Cuerpo de main (1/3)

```
int main()
{
    int z;

    z=f1(5, 2);

    pint(z);
}
```



```
main:
    li    a0, 5      # primer argumento
    li    a1, 2      # segundo argumento
    jal   ra, f1     # llamada
                                # resultado (a0)

    li    a7, 1
    ecall                                # llamada para
                                # imprimir un int

    jr   ra
```

## Ejemplo 2. Análisis de main (2/3)

```
int main()
{
    int z;

    z=f1(5, 2);

    pint(z);
}
```

main:

```
li    a0, 5    # primer argumento
li    a1, 2    # segundo argumento
jal   ra, f1   # llamada
                        # resultado (a0)

li    a7, 1    # llamada para
ecall                          # imprimir un int
```

- Los parámetros se pasan en a0 y a1
- El resultado se devuelve en a0
- Rutina no terminal (llama a otra rutina)

## Ejemplo 2. Ajuste de main (3/3)

```
int main()
{
    int z;

    z=f1(5, 2);

    pint(z);
}
```

main:

```
addi sp sp -4
sw ra 0(sp)
```

```
li a0, 5 # primer argumento
li a1, 2 # segundo argumento
jal ra, f1 # llamada
# resultado (a0)
```


```
li a7, 1
ecall # llamada para
# imprimir un int
```

```
lw ra 0(sp)
addi sp sp 4
jr ra
```

## Ejemplo 2. Cuerpo de f1 (1/3)

```
int f1 (int a, int b)
{
    int r;

    r = a + a + f2(b);
    return r;
}
```



```
f1: add    s0, a0, a0

     mv    a0, a1
     jal   ra f2
     add   a0, s0, a0

     jr    ra
```

```
int f2(int c)
{
    int s;

    s = c * c * c;
    return s;
}
```

## Ejemplo 2. Análisis de f1 (2/3)

```
int f1 (int a, int b)
{
    int r;

    r = a + a + f2(b);
    return r;
}
```

```
f1: add    s0, a0, a0

      mv    a0, a1
      jal   ra f2
      add   a0, s0, a0

      jr    ra
```

```
int f2(int c)
{
    int s;


    s = c * c * c;
    return s;
}
```

- f1 modifica s0 y ra, por lo tanto se guardan en la pila
- El registro ra se modifica en la instrucción “jal ra f2”
- El registro a0 se modifica al pasar el argumento a f2, pero por convenio la función f1 no tiene que guardarlo en la pila solo si lo utiliza después de realizar la llamada a f2

## Ejemplo 2. Cuerpo de f1 guardando en la pila los registros que se modifican (3/3)

```
int f1 (int a, int b)
{
    int r;

    r = a + a + f2(b);
    return r;
}
```



```
int f2(int c)
{
    int s;

    s = c * c * c;
    return s;
}
```

```
f1: addi    sp, sp, -8
     sw     s0, 4(sp)
     sw     ra, 0(sp)
```

```
     add   s0, a0, a0
     mv    a0, a1
     jal   ra f2
     add   a0, s0, a0
```

```
     lw    ra, 0(sp)
     lw    s0, 4(sp)
     addu  sp, sp, 8
```

```
     jr    ra
```

# Ejemplo 2. Cuerpo y análisis de f2

```
int f1 (int a, int b)
{
    int r;

    r = a + a + f2(b);
    return r;
}
```

```
int f2(int c)
{
    int s;

    s = c * c * c;
    return s;
}
```

```
f2: mul t0, a0, a0
     mul a0, t0, a0
     jr  ra
```

- La función f2 no modifica el registro ra porque no llama ninguna otra función.
- El registro t0 no es necesario guardarlo porque no se ha de preservar su valor según convenio

# Contenido

- ▶ Fundamentos básicos de la programación en ensamblador
- ▶ Ensamblador del RISC-V 32, modelo de memoria y representación de datos
- ▶ Formato de las instrucciones y modos de direccionamiento
- ▶ Llamadas a procedimientos y uso de la pila
  - ▶ ¿Cómo se llama a una función/subrutina?
  - ▶ ¿Dónde guardar la dirección de retorno en rutinas no terminales?
  - ▶ ¿Cuál es el convenio de paso de parámetros?
  - ▶ ¿Cuál es el convenio de uso de registros?
  - ▶ ¿Cómo son las variables locales? (registro de activación)



# Activación de procedimientos

## Marco de pila

- ▶ El **marco de pila o registro de activación** es el mecanismo que utiliza el compilador para activar los procedimientos (subrutinas) en los lenguajes de alto nivel
- ▶ El marco de pila lo construyen en la pila el procedimiento llamante y el llamado

# Marco de pila

- ▶ El marco de pila almacena:
  - ▶ Los **parámetros introducidos** por el procedimiento llamante **en caso de ser necesarios**
  - ▶ Los **registros guardados** por la función (incluyen al registro `ra` en caso de procedimientos no terminales)
  - ▶ **Variables locales**

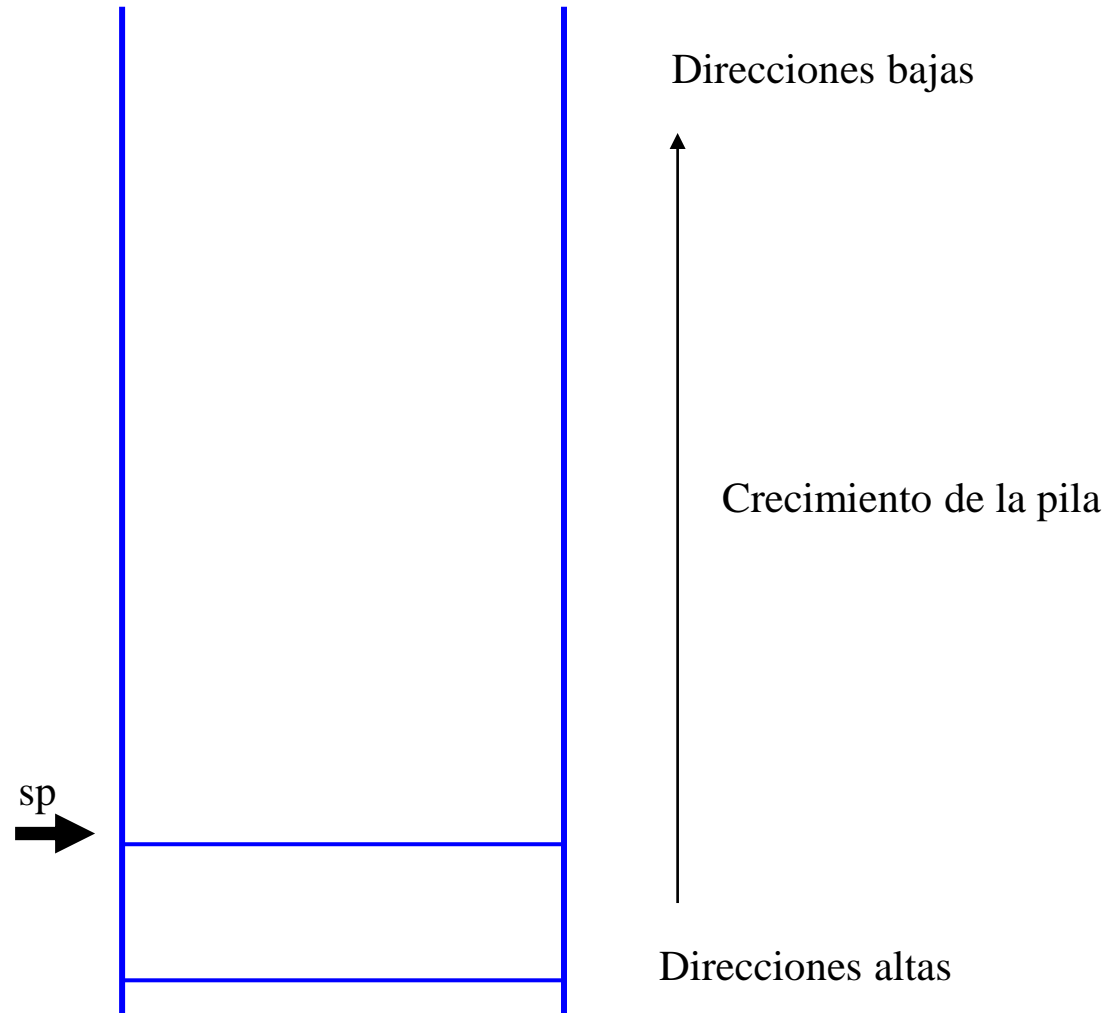
# Procedimiento general de llamadas a funciones

## versión simplificada

Subrutina llamante	Subrutina llamada
Salvaguarda de registros que no quiera que modifique la subrutina llamada ( $t_x, a_x, \dots$ )	
Paso de parámetros, reserva de espacio para valores a devolver si es necesario	
<b>Llamada a subrutina (jal)</b>	
	Reserva del marco de pila
	Salvaguarda de registros ( $ra, s_x$ )
	<b>Ejecución de subrutina</b>
	Restauración de valores guardados
	Copiar valores a devolver en el espacio reservado por el llamante
	Liberación de marco de pila
	<b>Salida de subrutina (jr ra)</b>
Recuperar valores devueltos	
Restauración de registros guardados, liberación del espacio de pila reservado	

# Construcción del marco de pila subrutina llamante

No se va a seguir el  
estrictamente el  
convenio del RISC-V  
por simplicidad



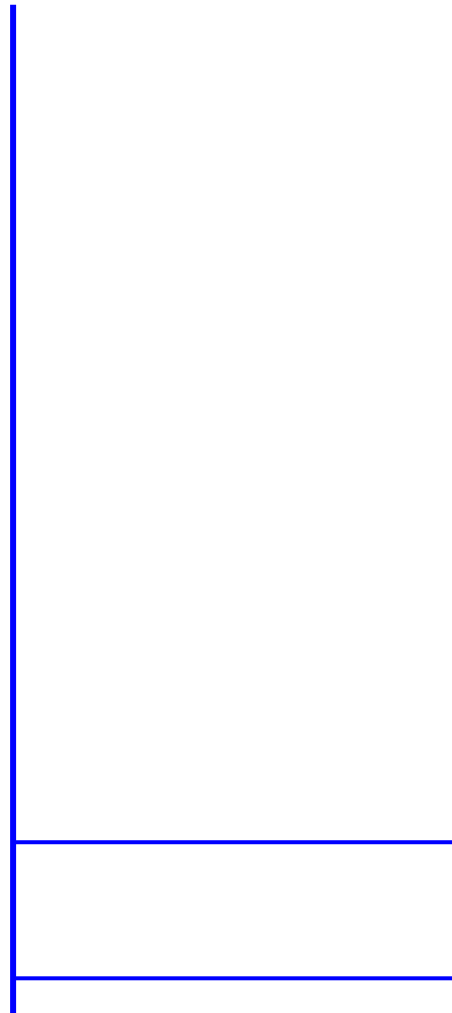
# Construcción del marco de pila subrutina llamante

Situación **inicial** antes de realizar la **llamada** a un procedimiento

Marco de pila del procedimiento que realiza la llamada



sp  
→



(-)

(+)

# Construcción del marco de pila subrutina llamante

## Salvaguardar Registros

Una subrutina puede modificar cualquier registro  $a_x$  y  $t_x$

Marco de pila  
Del procedimiento  
Que realiza la llamada

sp  
→

Registros salvados

## Ejemplo:

```
li  t0, 4  
li  t1, 8  
li  a0, 5  
jal ra, funcion
```

```
mv s2, t0
```

**¿Qué valor tiene t0  
y t1?**

# Construcción del marco de pila subrutina llamante

## Salvaguardar Registros

Una subrutina puede modificar cualquier registro **a0** y **t0**

Para preservar su valor, la subrutina que realiza la llamada debe guardar en la pila los valores de esos registros

Marco de pila  
Del procedimiento  
Que realiza la llamada

sp  
→

Registros salvados

## Ejemplo:

```
li t0, 4
li t1, 8
li a0, 5
jal ra, funcion
```

```
mv s2, t0
```

**¿Qué valor tiene t0 y t1?**

# Construcción del marco de pila subrutina llamante

## Salvaguardar Registros

Una subrutina puede modificar cualquier registro **a0** y **t0**

Para preservar su valor, la subrutina que realiza la llamada debe guardar en la pila los valores de esos registros

Marco de pila  
Del procedimiento  
Que realiza la llamada

sp →

Registros salvados

## Ejemplo:

```
subu sp sp 8
sw  t0 0(sp)
sw  t1 4(sp)

li  a0, 5
jal ra, funcion
```



# Construcción del marco de pila subrutina llamante

## Salvaguardar Registros

Una subrutina puede modificar cualquier registro **a0** y **t0**

Para preservar su valor, la subrutina que realiza la llamada debe guardar en la pila los valores de esos registros  
(habrá que restaurarlos después)

Marco de pila  
Del procedimiento  
Que realiza la llamada

sp →

Registros salvados

## Ejemplo:

```
sub sp sp 8
sw t0 0(sp)
sw t1 4(sp)

li a0, 5
jal ra, funcion

lw t0 0(sp)
lw t1 4(sp)
add sp sp 8
```

# Construcción del marco de pila subrutina llamante

Ejemplo (10 parámetros):

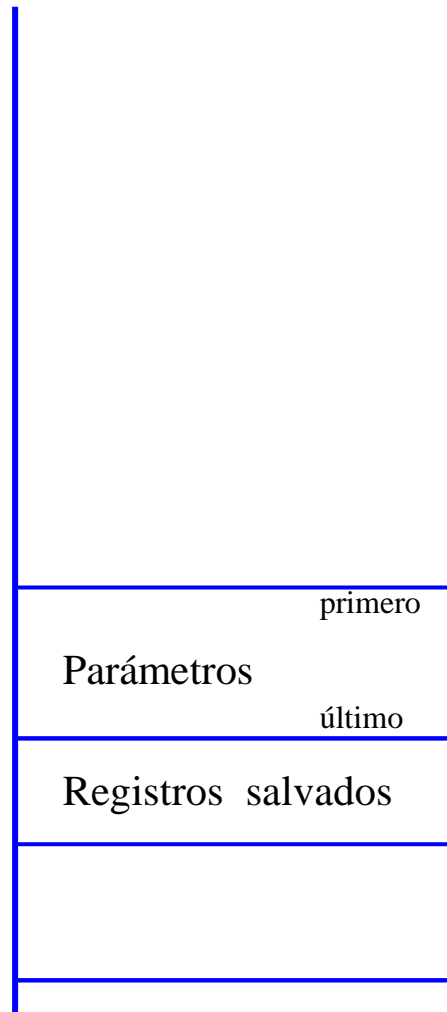
## Paso de parámetros:

Antes de realizar la llamada el procedimiento llamante:

- Deja los parámetros en  $a_x$  ( $f_x$ )
- El resto de parámetros en la pila

Marco de pila  
Del procedimiento  
Que realiza la llamada

sp  
→



```
li a0, 1
li a1, 2
li a2, 3
li a3, 4
li a4, 5
li a5, 6
li a6, 7
li a7, 8
```

```
addi sp, sp, -8
```

```
li t0, 10
sw t0, 4(sp)
```

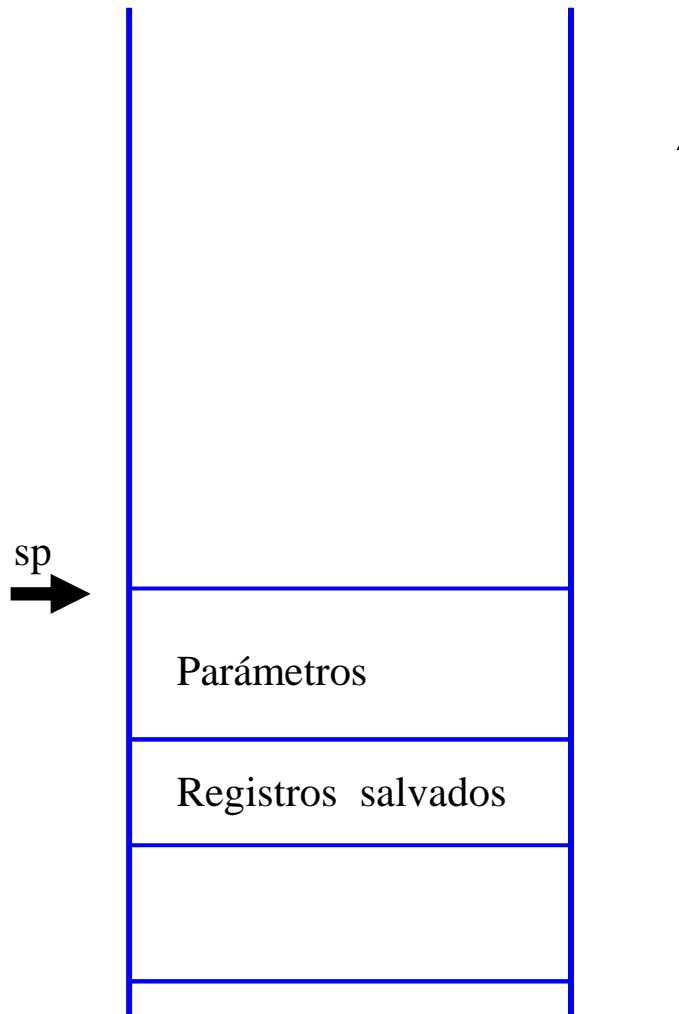
```
li t0, 9
sw t0, 0(sp)
```

# Construcción del marco de pila subrutina llamante

## Llamada a subrutina:

jal ra subrutina

Marco de pila  
Del procedimiento  
Que realiza la llamada



```
li a0, 1
li a1, 2
li a2, 3
li a3, 4
li a4, 5
li a5, 6
li a6, 7
li a7, 8
```

```
addi sp, sp, -8
```

```
li t0, 10
sw t0, 4(sp)
```

```
li t0, 9
sw t0, 0(sp)
```

```
jal ra subrutina
```

# Construcción del marco de pila subrutina llamada

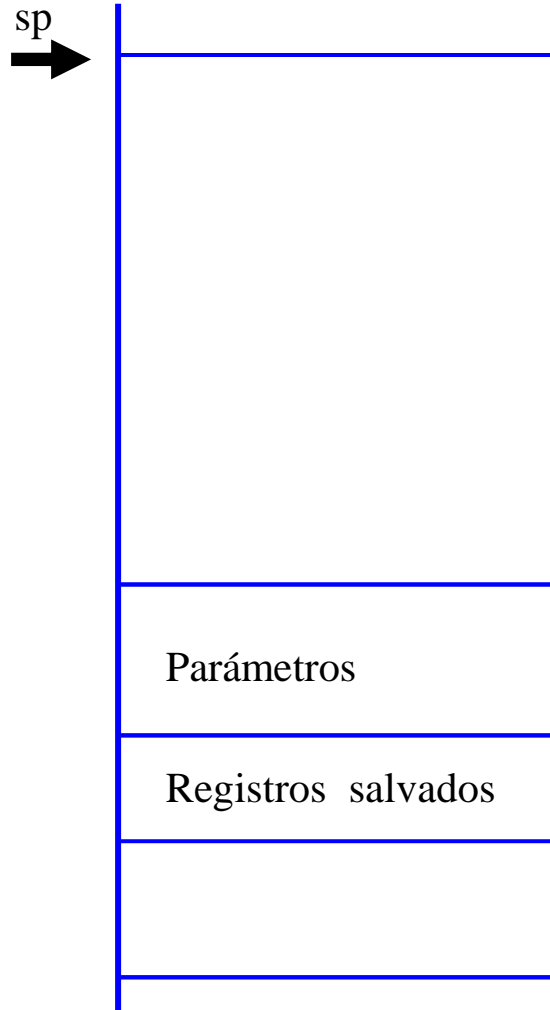
## Reserva del marco de pila:

$sp = sp - \text{tamaño marco}$

## Espacio para:

ra,  
s0...s7  
variables locales

Marco de pila  
Del procedimiento  
Que realiza la llamada



## Example:

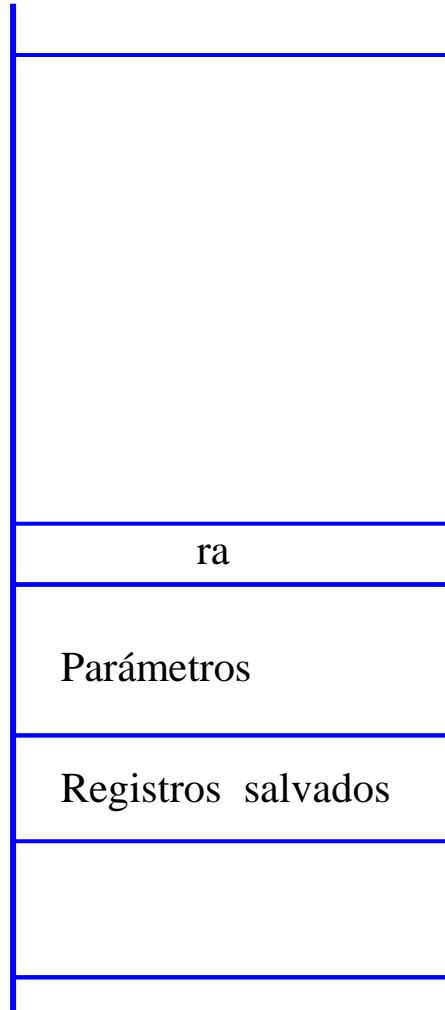
```
function:  
subu sp sp <fr.sz.>
```

# Construcción del marco de pila subrutina llamada

**Salvaguada de registros:**

ra (dirección de retorno)

sp  
→



**ra** se guarda en funciones  
no terminales

Marco de pila  
Del procedimiento  
Que realiza la llamada

# Construcción del marco de pila subrutina llamada

## Salvaguada de registros $s_x$ :

Se guarda los registros  $s_x$  que se vayan a modificar  
Una función no puede por convenio modificar los registros  $s_x$  (sí lo  $t_x$  y los  $a_x$ )

Marco de pila  
Del procedimiento  
Que realiza la llamada

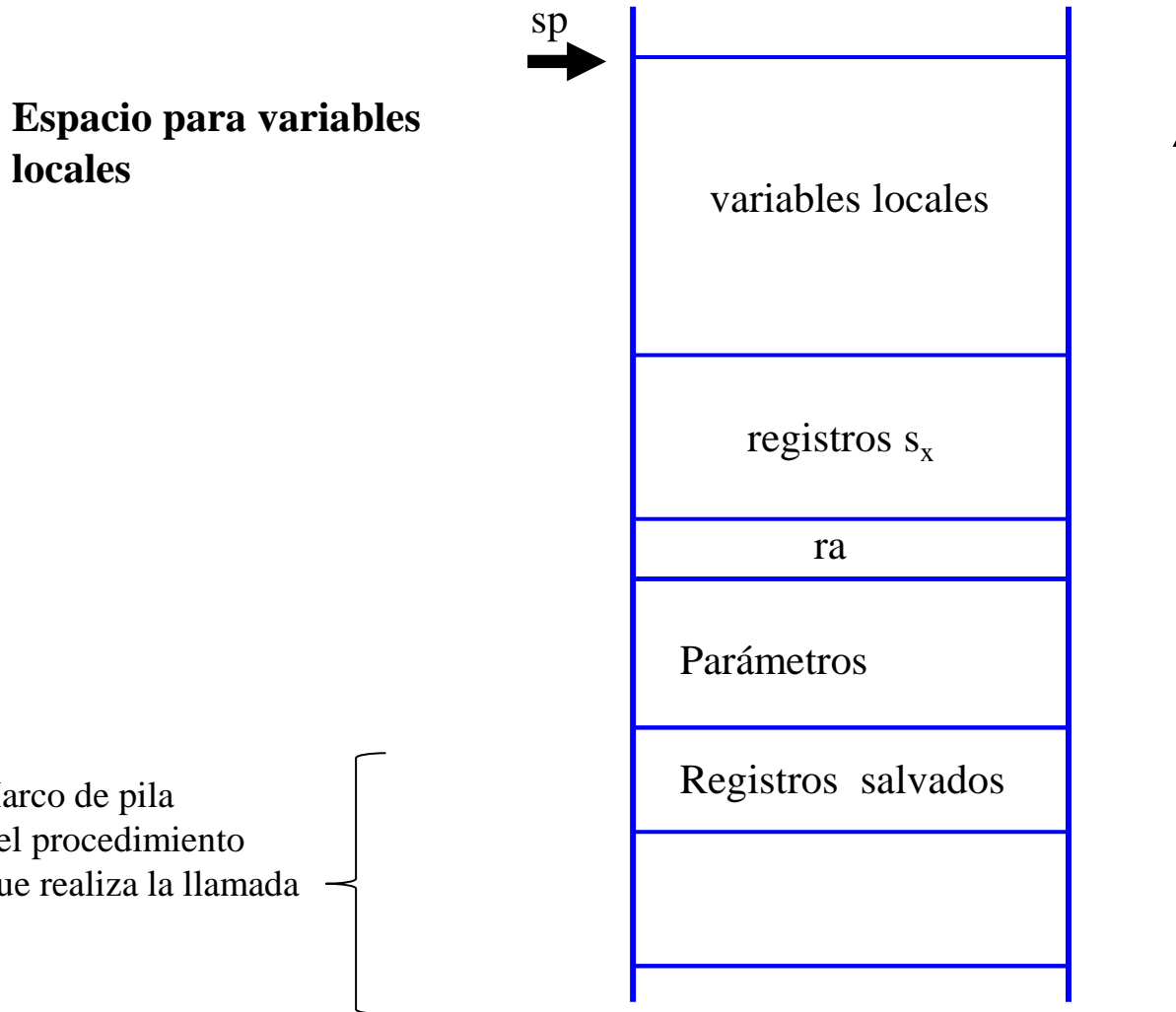


## Example:

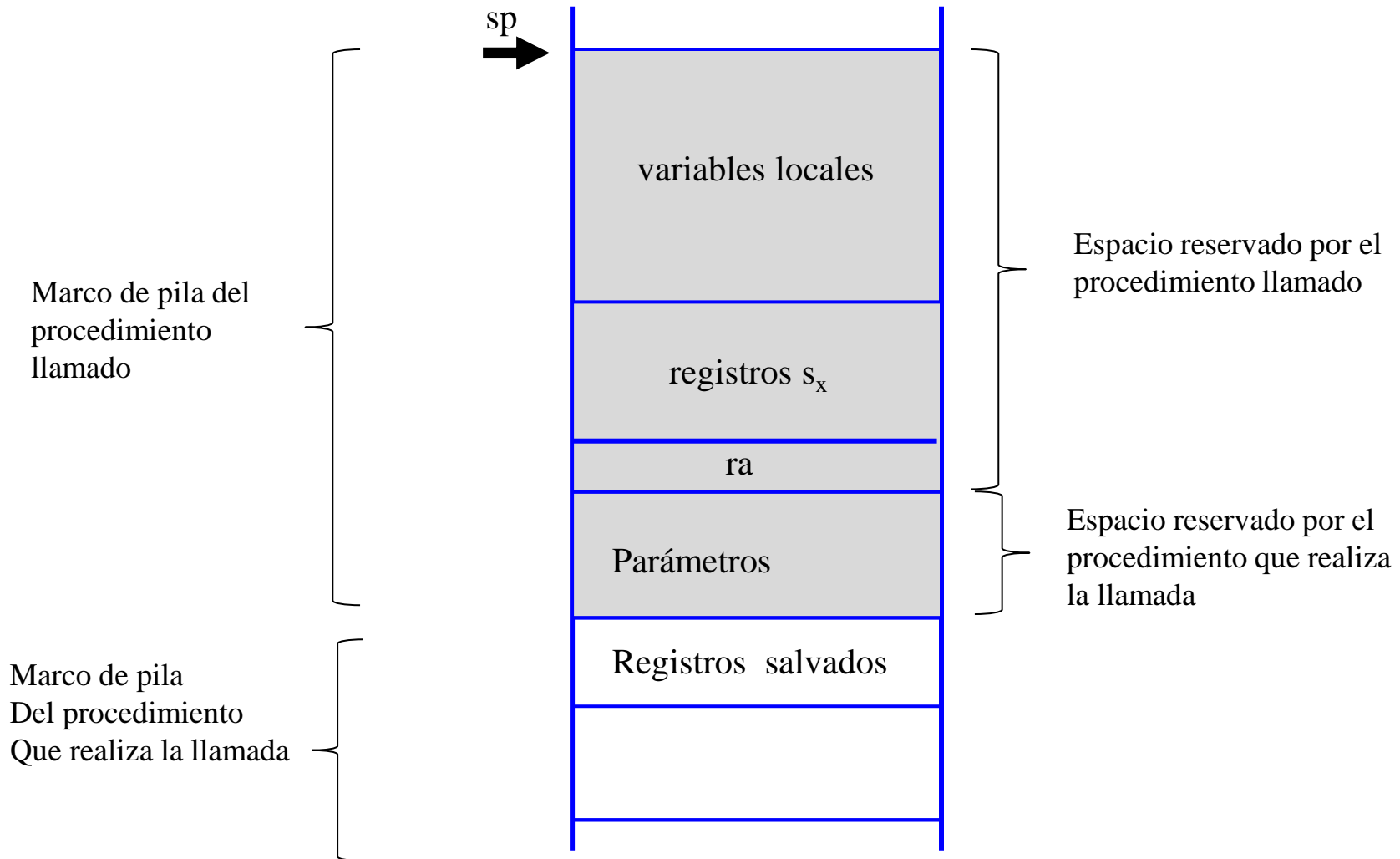
function:

```
subu sp sp <fr.sz.>
sw ra <fr.sz-4>(sp)
sw s0 <fr.sz-8>(sp)
sw s1 <...>(sp)
...
```

# Construcción del marco de pila subrutina llamada



# Construcción del marco de pila





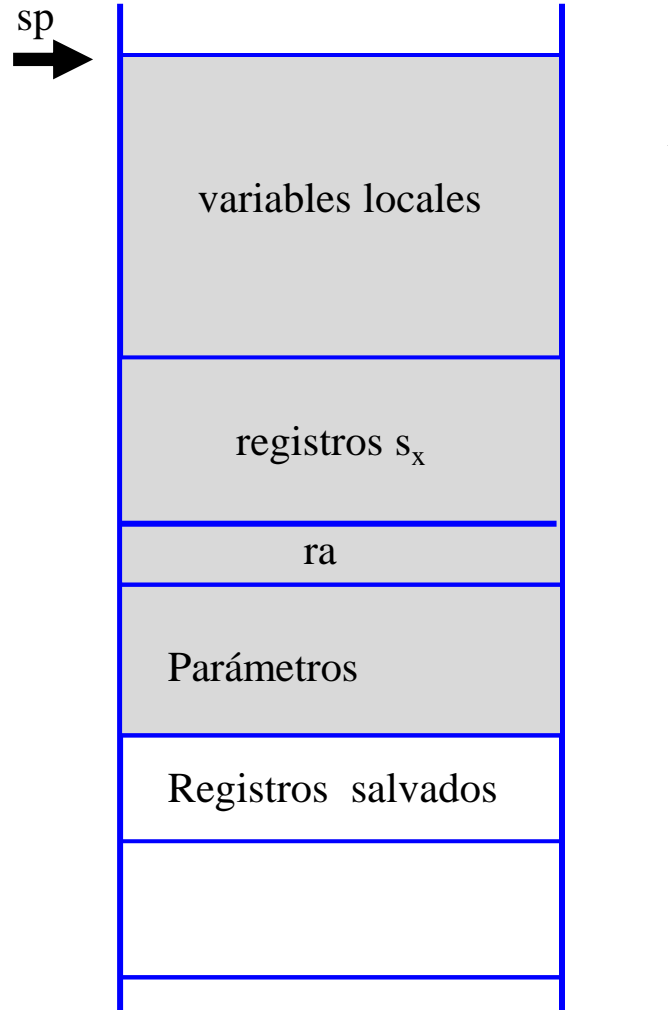
# Finalización de la subrutina subrutina llamada

**Se devuelven los resultados:**

a0, a1, (fa0, fa1)

Si devuelve estructuras más complejas se dejan en la pila (el llamante habrá dejado hueco)

Marco de pila  
Del procedimiento  
Que realiza la llamada

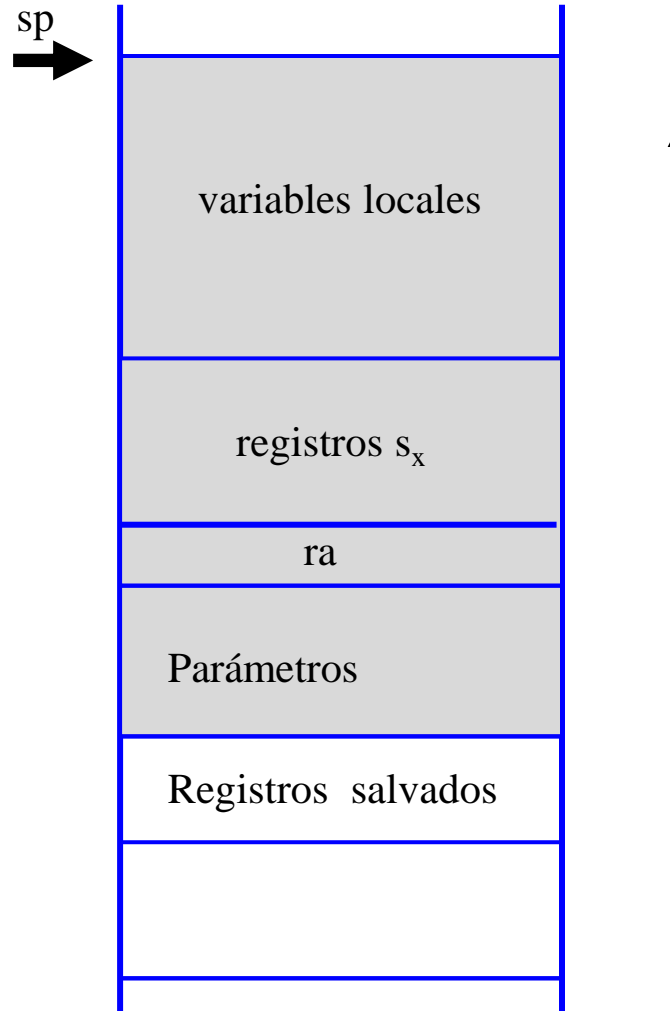


# Finalización de la subrutina subrutina llamada

**Se restauran los registros salvados:**

registros  $s_x$   
registro ra

Marco de pila  
Del procedimiento  
Que realiza la llamada

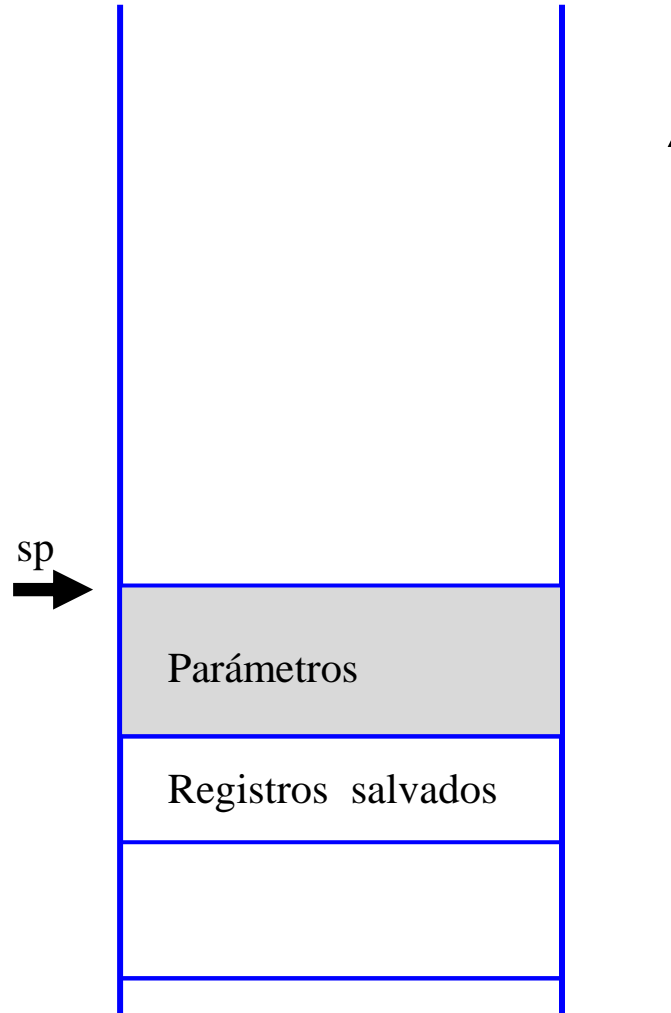


# Finalización de la subrutina subrutina llamada

**Se libera el espacio del marco:**

$sp = sp + \text{tamaño marco}$

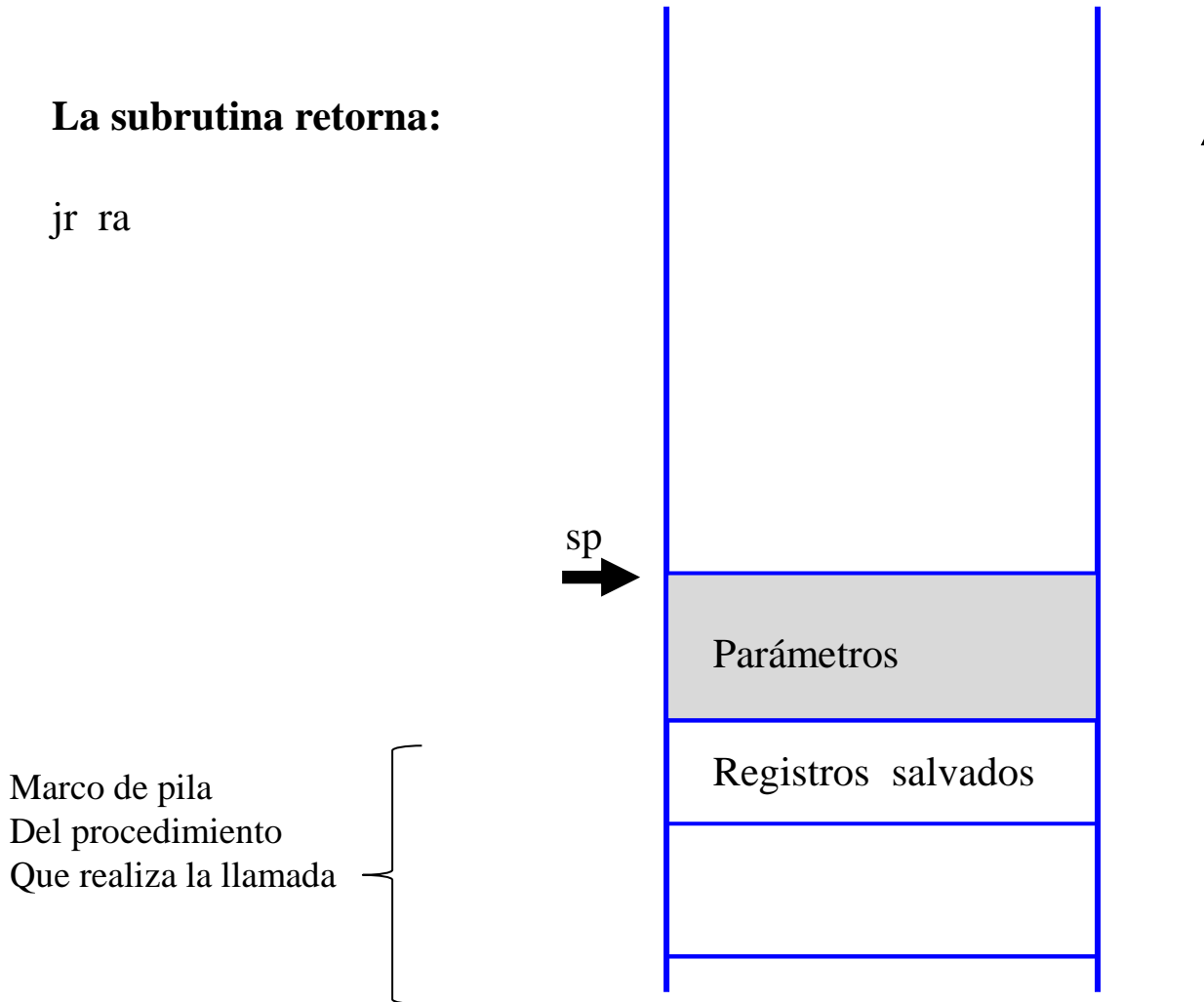
Marco de pila  
Del procedimiento  
Que realiza la llamada



# Finalización de la subrutina subrutina llamada

**La subrutina retorna:**

jr ra



# Finalización de la subrutina subrutina llamante

**La rutina que realizó la llamada libera el espacio de los parámetros**

$sp = sp + \langle \text{espacio parámetros} \rangle$

Marco de pila  
Del procedimiento  
Que realiza la llamada

sp →

Parámetros

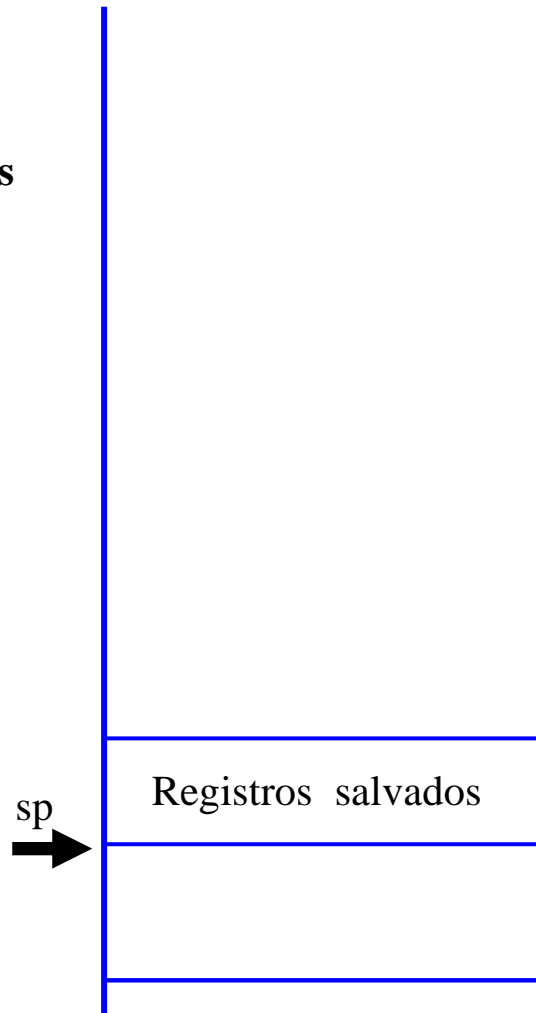
Registros salvados

# Finalización de la subrutina subrutina llamante

La rutina que realizó la llamada restaura los registros que salvó

Restaura sp

Marco de pila  
Del procedimiento  
Que realiza la llamada



**Ejemplo:**

```
addi sp sp -8  
sw  t0 0(sp)  
sw  t1 4(sp)
```

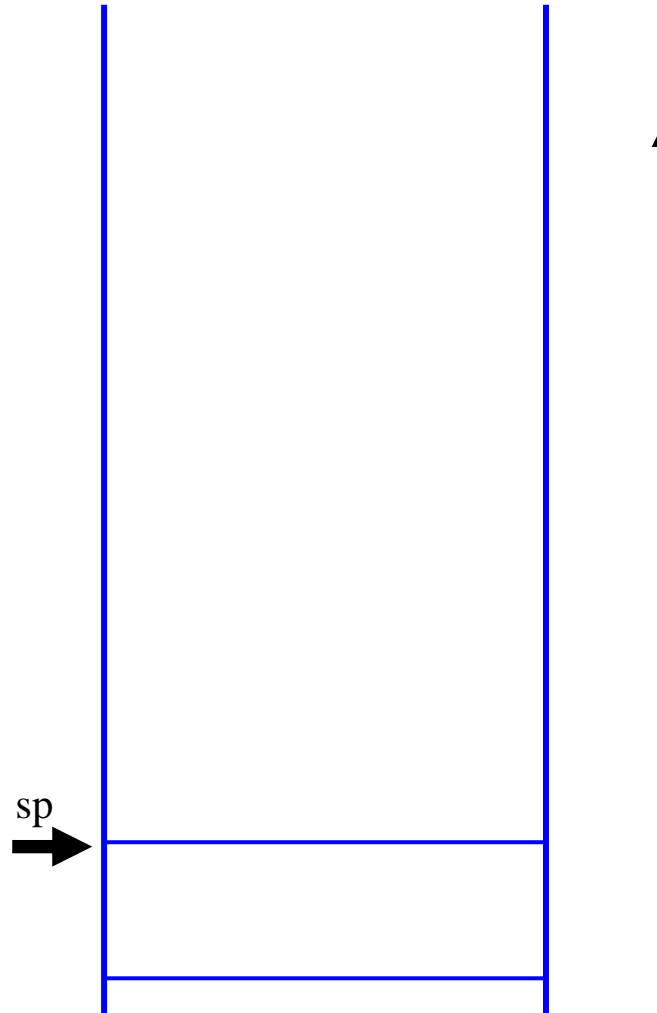
```
li  a0, 5  
jal ra, funcion
```

```
lw  t0 0(sp)  
lw  t1 4(sp)  
add sp sp 8
```

# Estado después de finalizar la llamada

**Estado inicial**

Marco de pila  
Del procedimiento  
Que realiza la llamada



# Variables locales en registros

- ▶ Siempre que se puede, las variables locales (int, double, char, ...) se almacenan en registros
  - ▶ Si no se pueden utilizar registros (no hay suficientes) se usa la pila

```
int f(...)  
{  
    int i, j, k;  
  
    i = 0;  
    j = 1;  
    k = i + j;  
    . . .  
}
```

```
f:  . . .  
    li  t0, 0  
    li  t1, 1  
    add t2, t0, t1  
    . . .
```



Grupo ARCOS

**uc3m** | Universidad **Carlos III** de Madrid

# Tema 3 (IV)

## Fundamentos de la programación en ensamblador

Estructura de Computadores  
Grado en Ingeniería Informática

