

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

# Procedimientos

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

```
factorial:  
    mv    t0 a0  
    li    v0 1  
b1: beq   t0 zero f1  
    mul   v0 v0 t0  
    addi  t0 t0 -1  
    beq   x0 x0 b1  
f1: jr    ra  
...  
li    a0 3  
jal   ra factorial  
...
```

- ▶ 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
- ▶ 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

# Pasos en la ejecución de una función

- ▶ Situar los parámetros en un lugar donde la función pueda accederlos
- ▶ Transferir el control a la función
- ▶ Adquirir los recursos de almacenamiento necesarios para la función
- ▶ Realizar la tarea deseada
- ▶ Poner el resultado en un lugar donde el programa o función que realiza la llamada pueda accederlo
- ▶ Devolver el control al punto de origen


# Funciones en un lenguaje de alto nivel

```
int main() {  
    int z;  
    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;  
}
```


# Funciones en un lenguaje de alto nivel

```
int main() {  
    int z;  
    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;  
}
```



# Funciones en un lenguaje de alto nivel

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

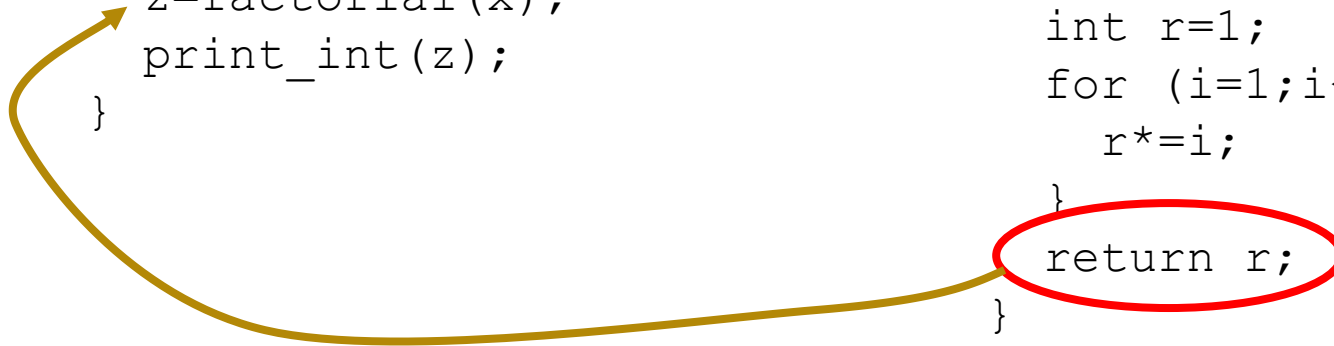


```
int factorial(int x) {  
    int i;  
    int r=1;  
    for (i=1;i<=x;i++) {  
        r*=i;  
    }  
    return r;  
}
```

# Funciones en un lenguaje de alto nivel

```
int main() {  
    int z;  
    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;  
}
```





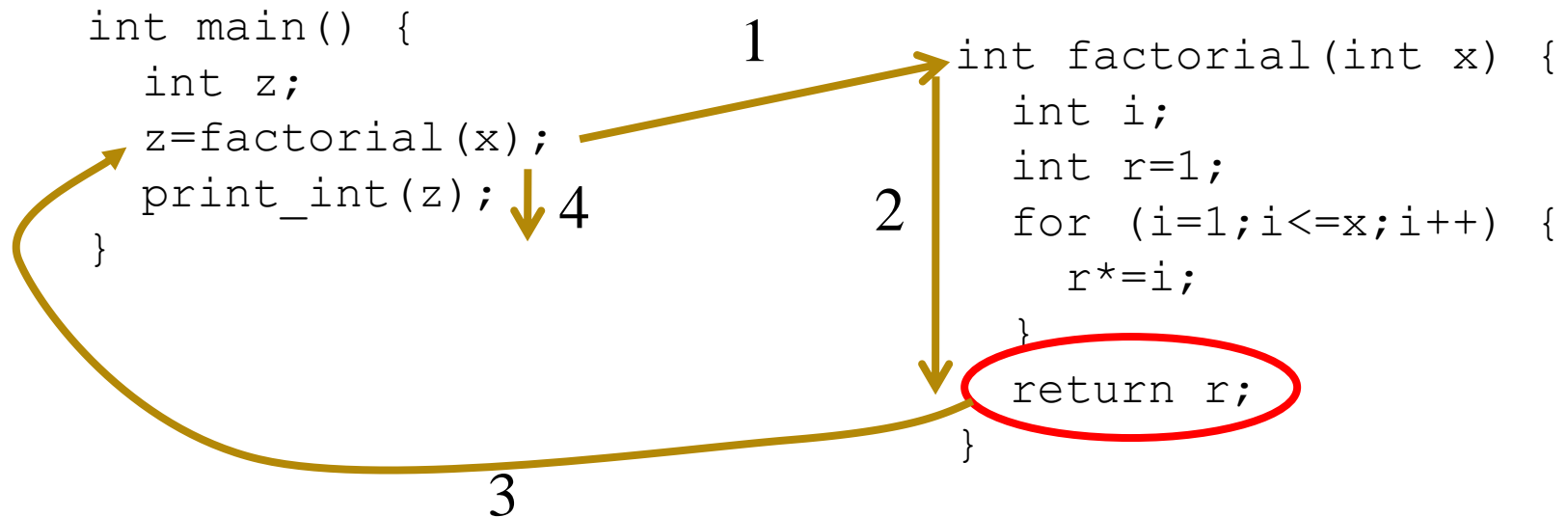
# Funciones en un lenguaje de alto nivel

```
int main() {  
    int z;  
    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;  
}
```

# Funciones en un lenguaje de alto nivel



# Funciones en un lenguaje de alto nivel

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

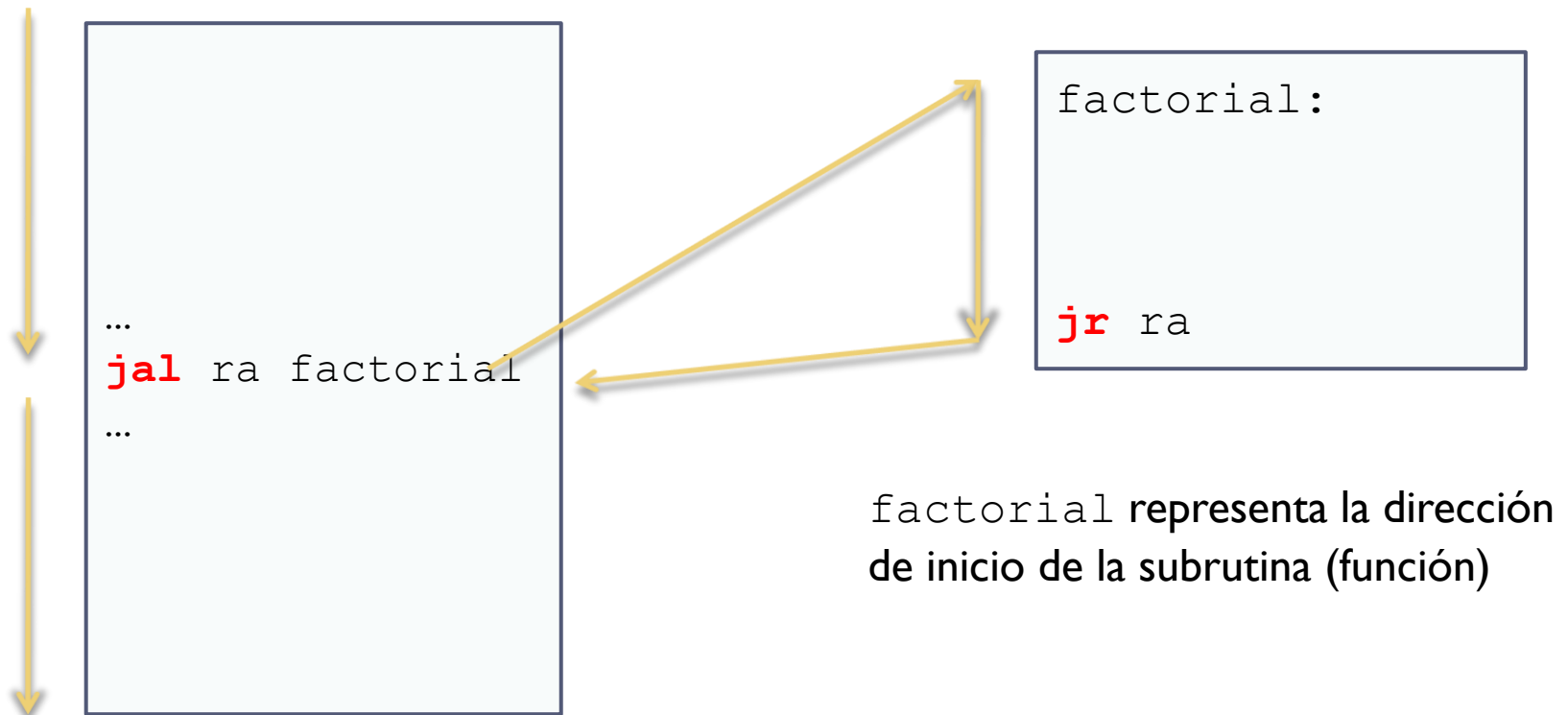
```
int factorial(int x) {  
    int i;  
    int r=1;  
    for (i=1;i<=x;i++) {  
        r*=i;  
    }  
    return r;  
}
```

Variables locales

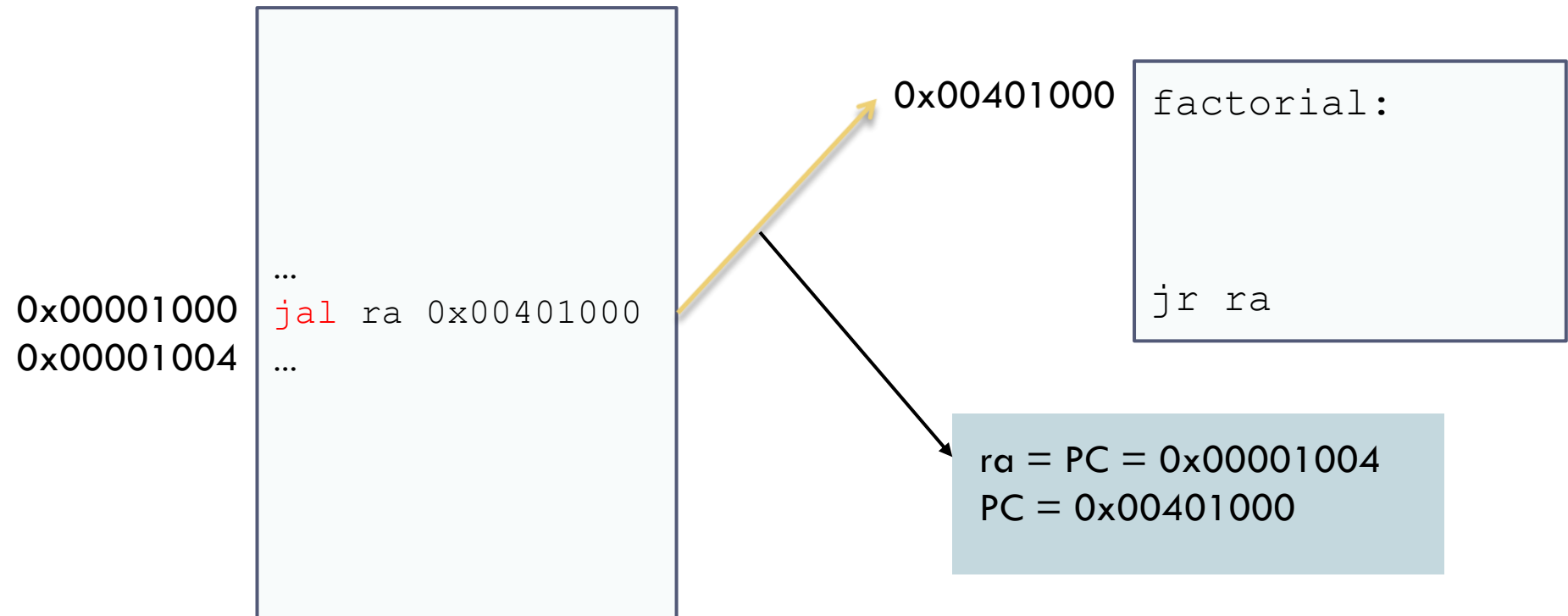


# Llamadas a funciones en RISC-V

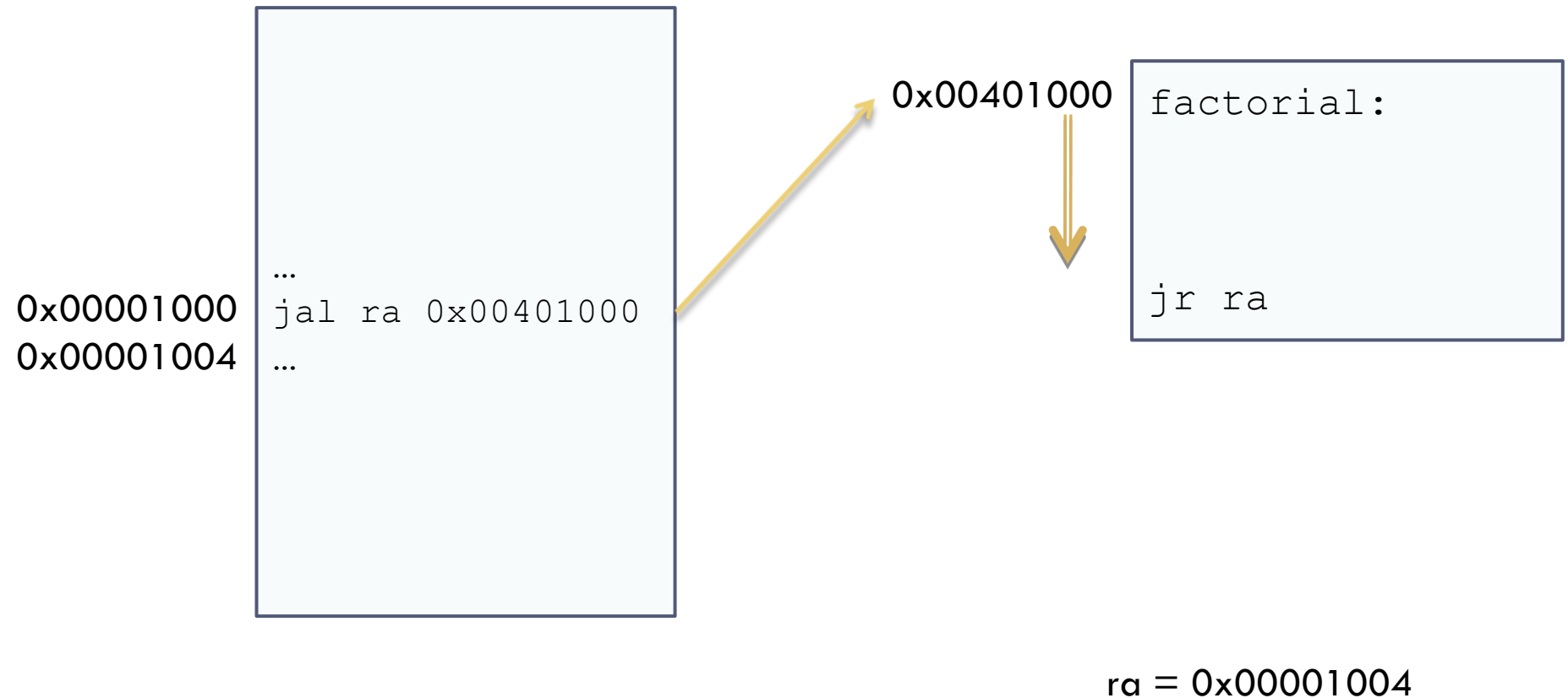
Llamada a función en el RISC-V (instrucción `jal`)



# Llamadas a funciones en RISC-V

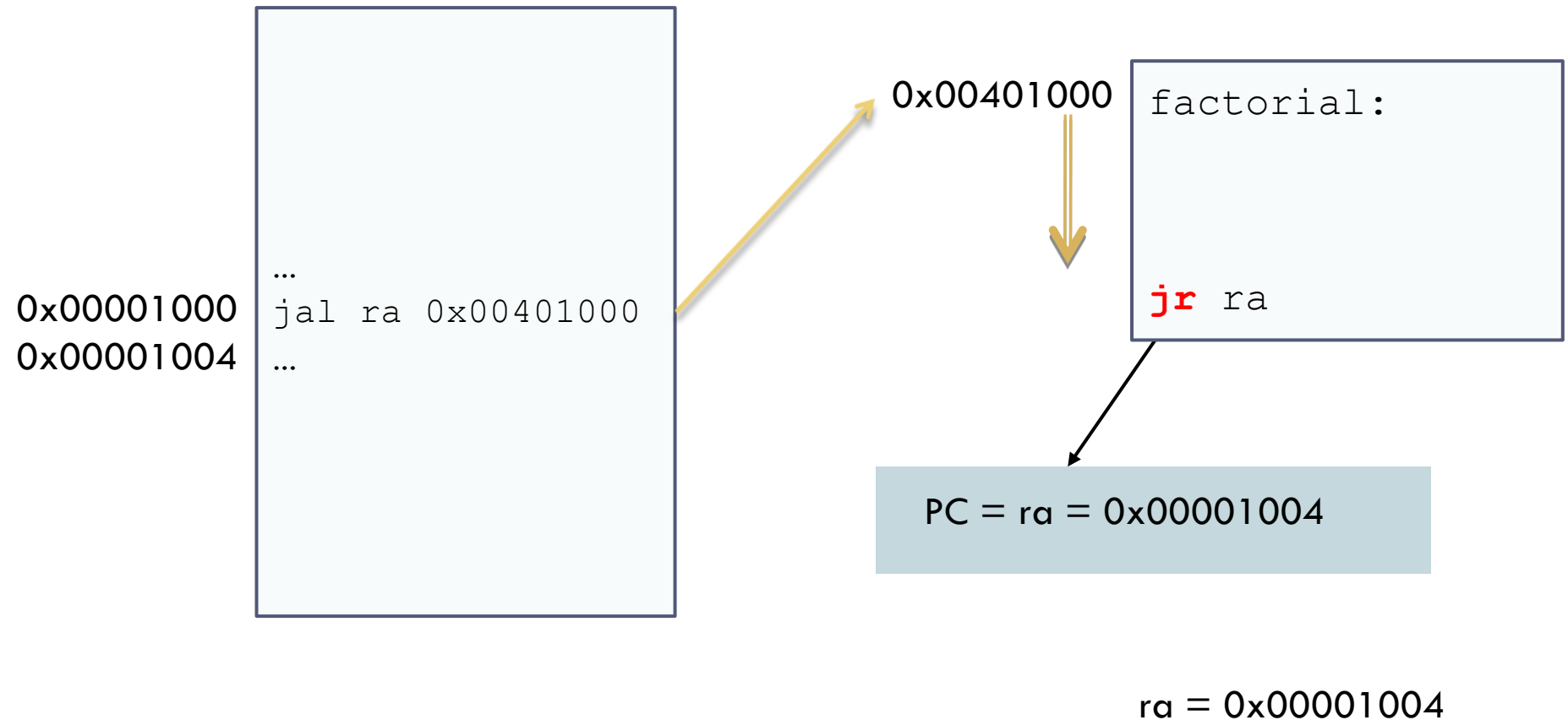


# Llamadas a funciones en RISC-V

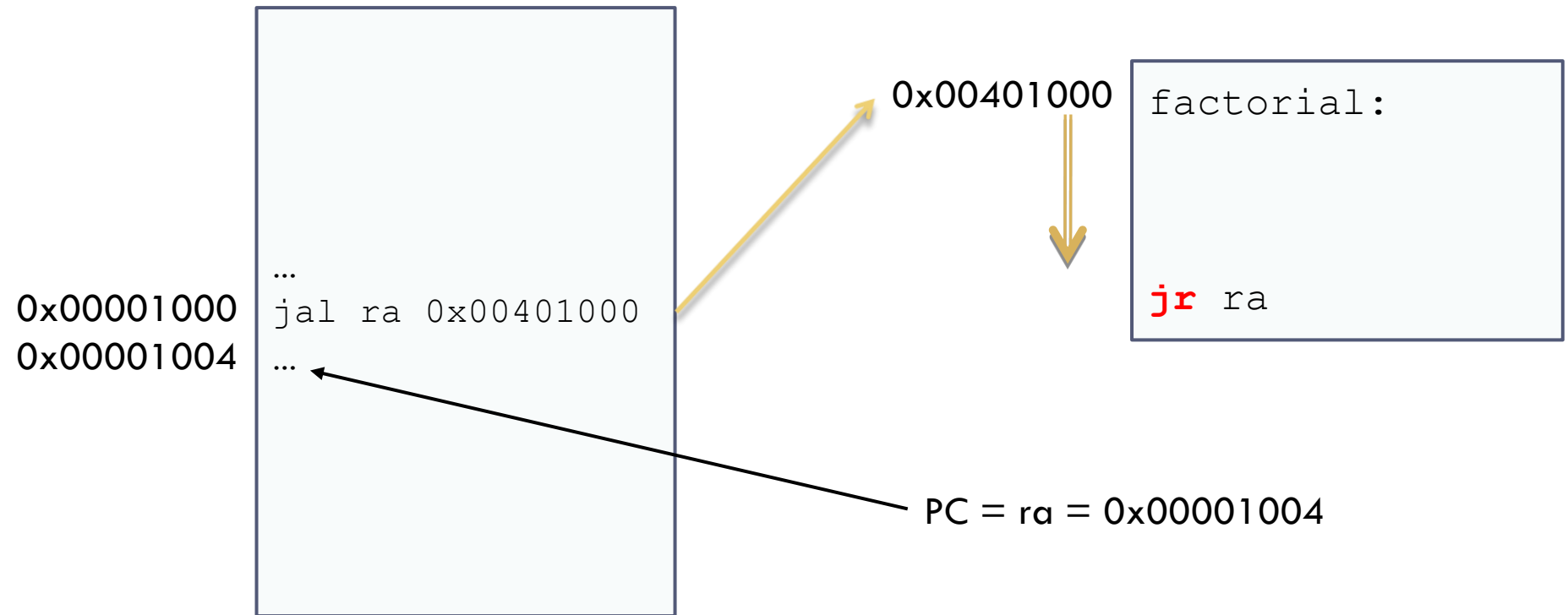


# Llamadas a funciones en RISC-V

Retorno de subrutina (instrucción `jr` )



# Llamadas a funciones en RISC-V

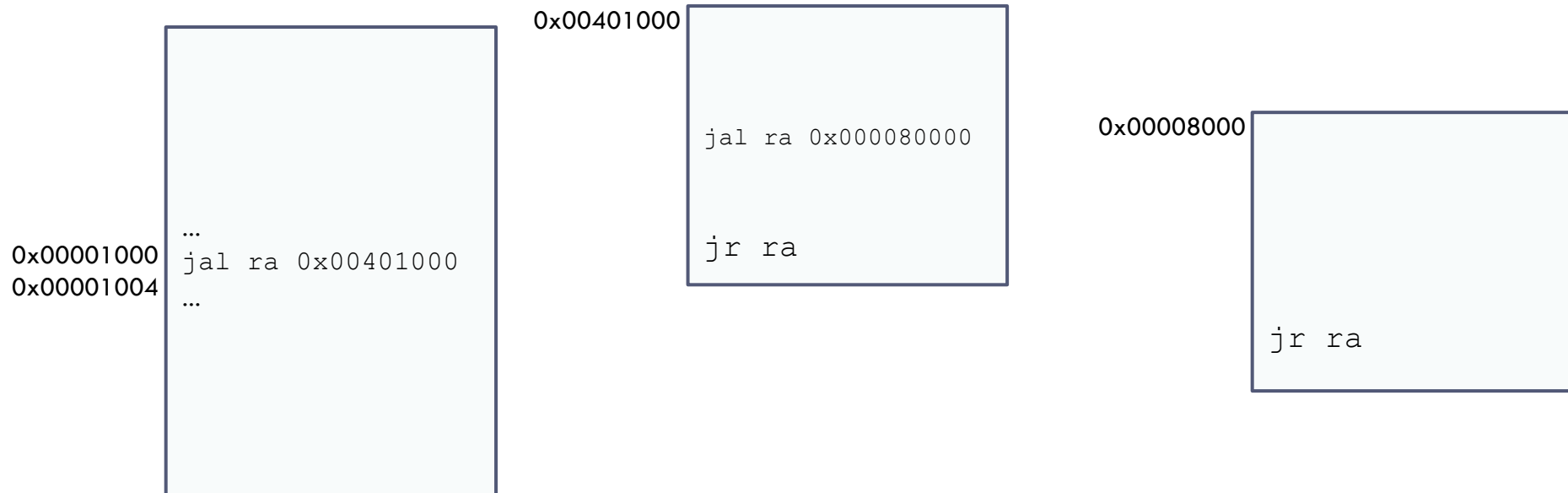




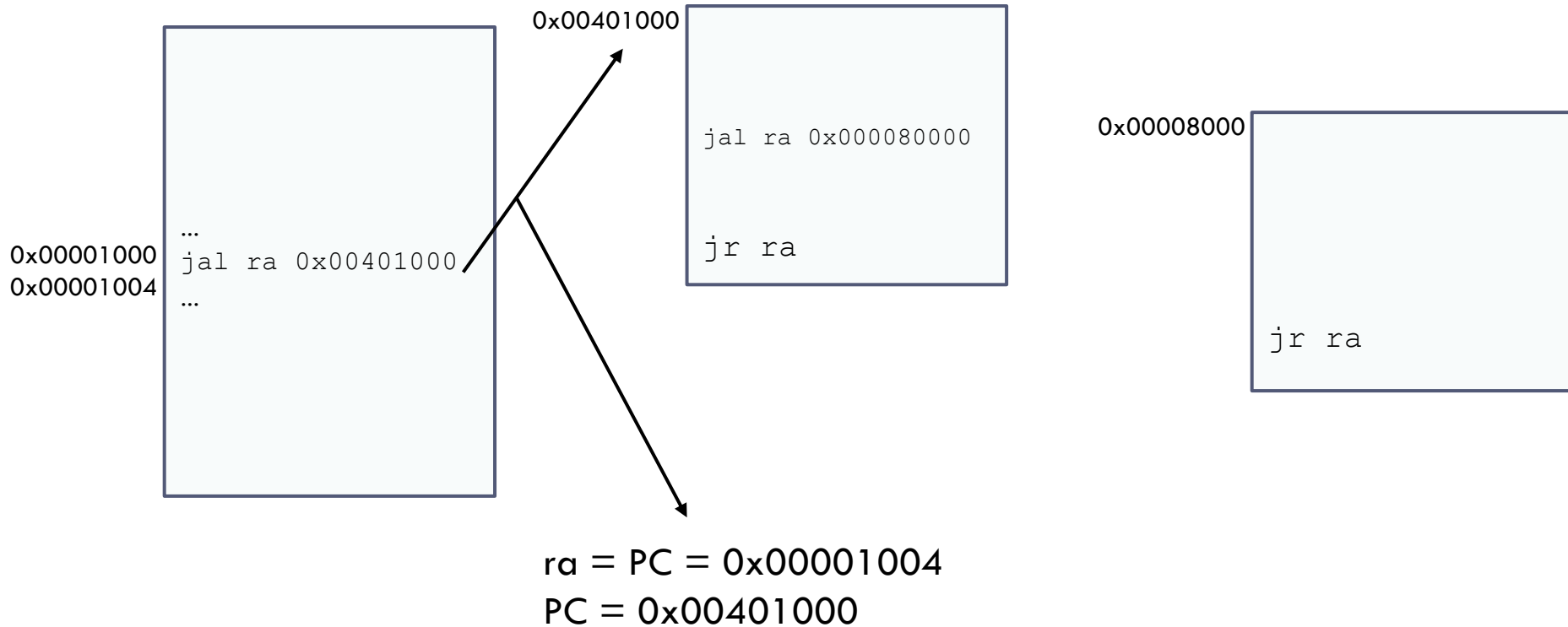
# Instrucciones jal/jr

- ▶ ¿Qué hace la instrucción jal?
  - ▶  $ra \leftarrow PC$
  - ▶  $PC \leftarrow \text{Dirección de salto}$
- ▶ ¿Qué hace la instrucción jr?
  - ▶  $PC \leftarrow ra$

# Llamadas anidadas

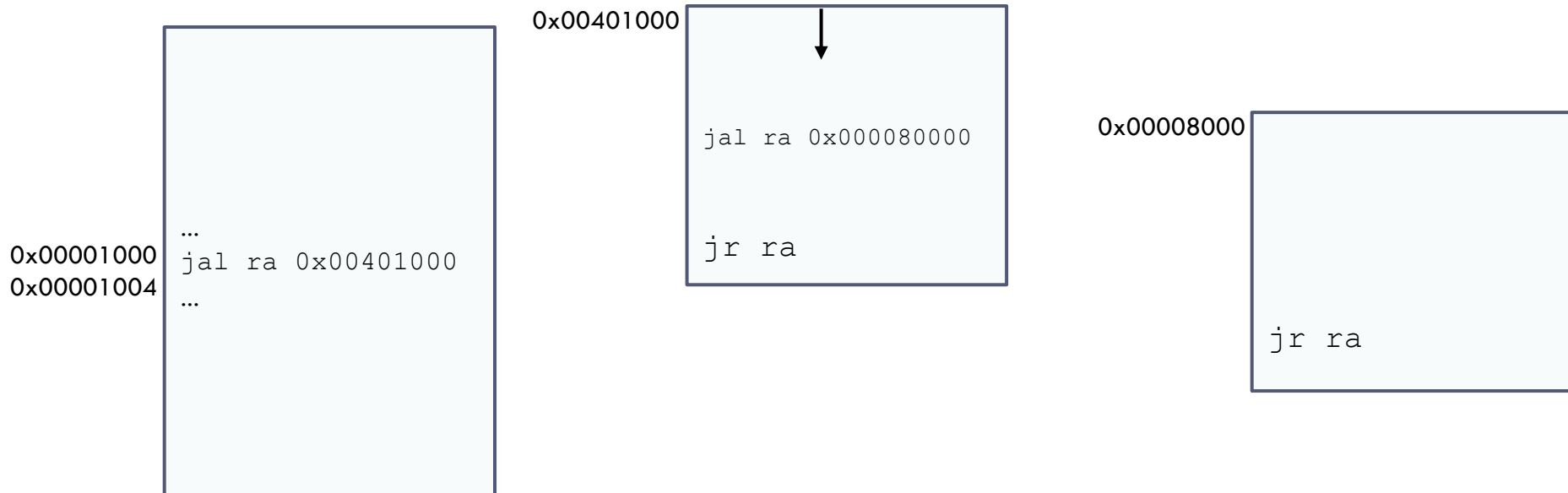


# Llamadas anidadas



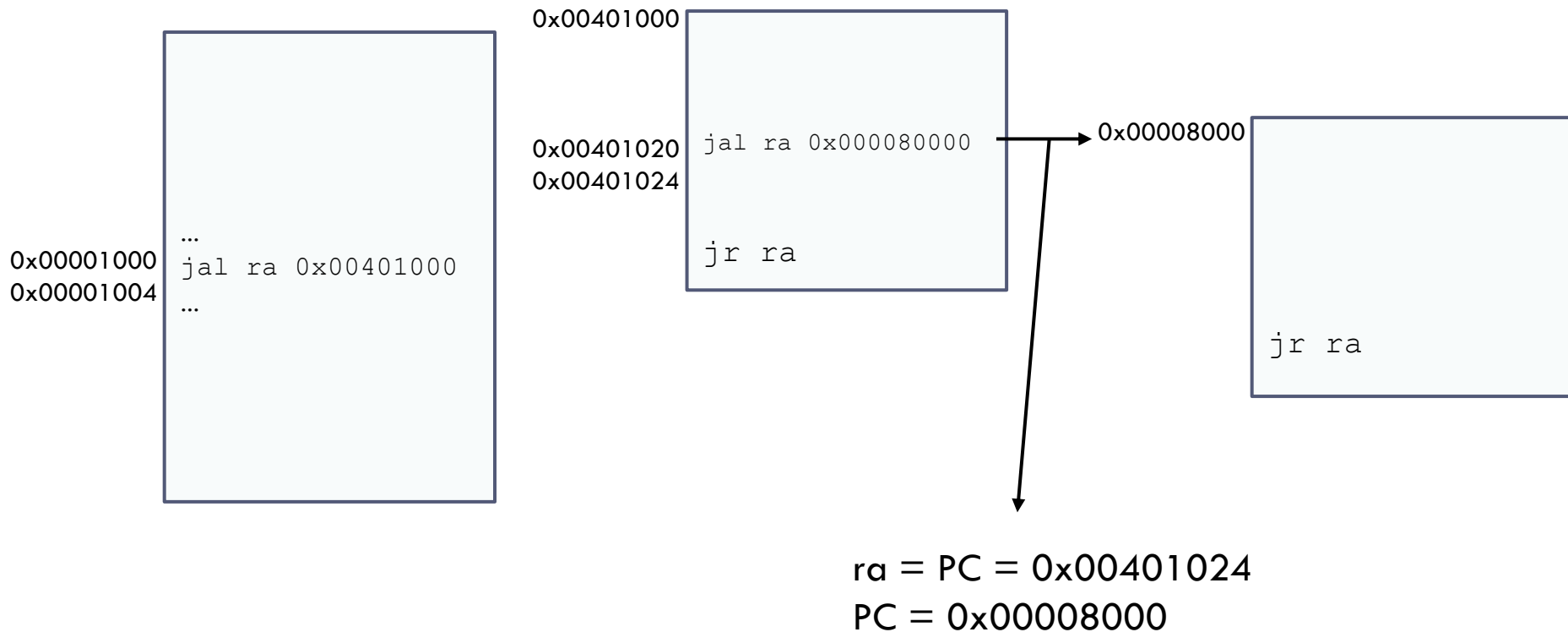
Dirección de retorno `ra = PC = 0x00001004`

# Llamadas anidadas



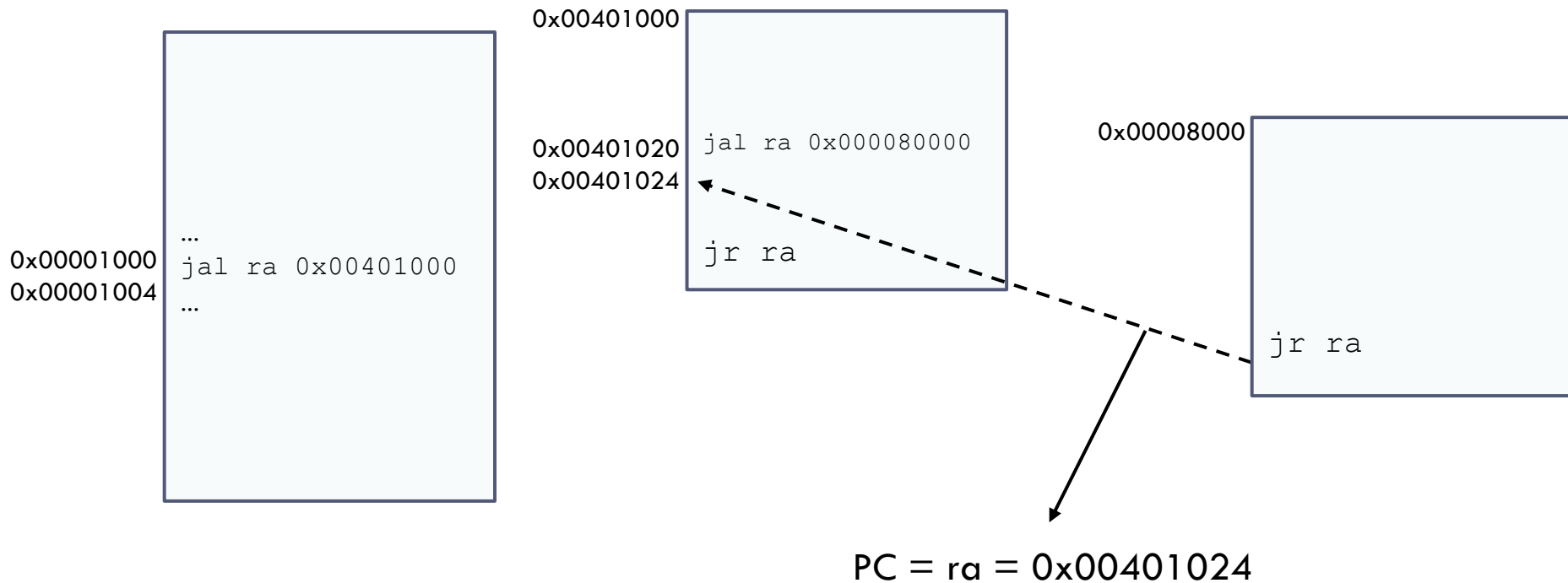
Dirección de retorno  $ra = PC = 0x00001004$

# Llamadas anidadas



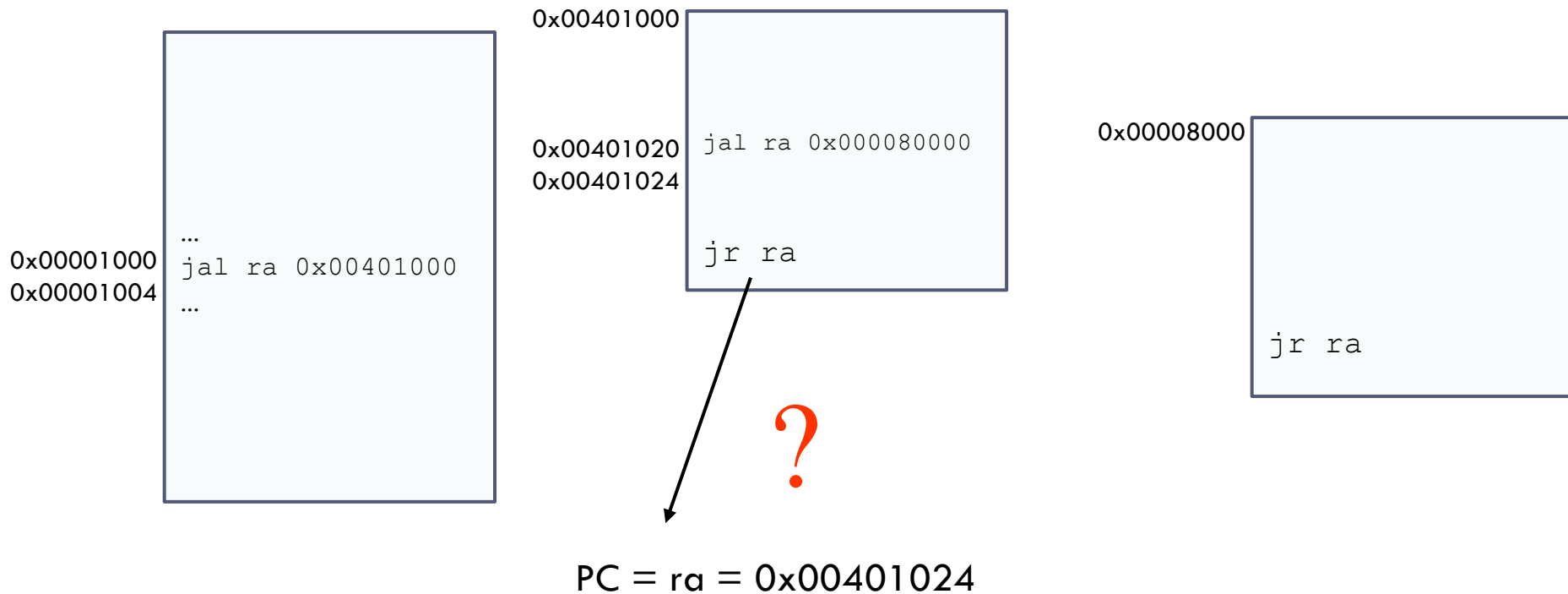
~~Dirección de retorno ra = PC = 0x00001004~~

# Llamadas anidadas



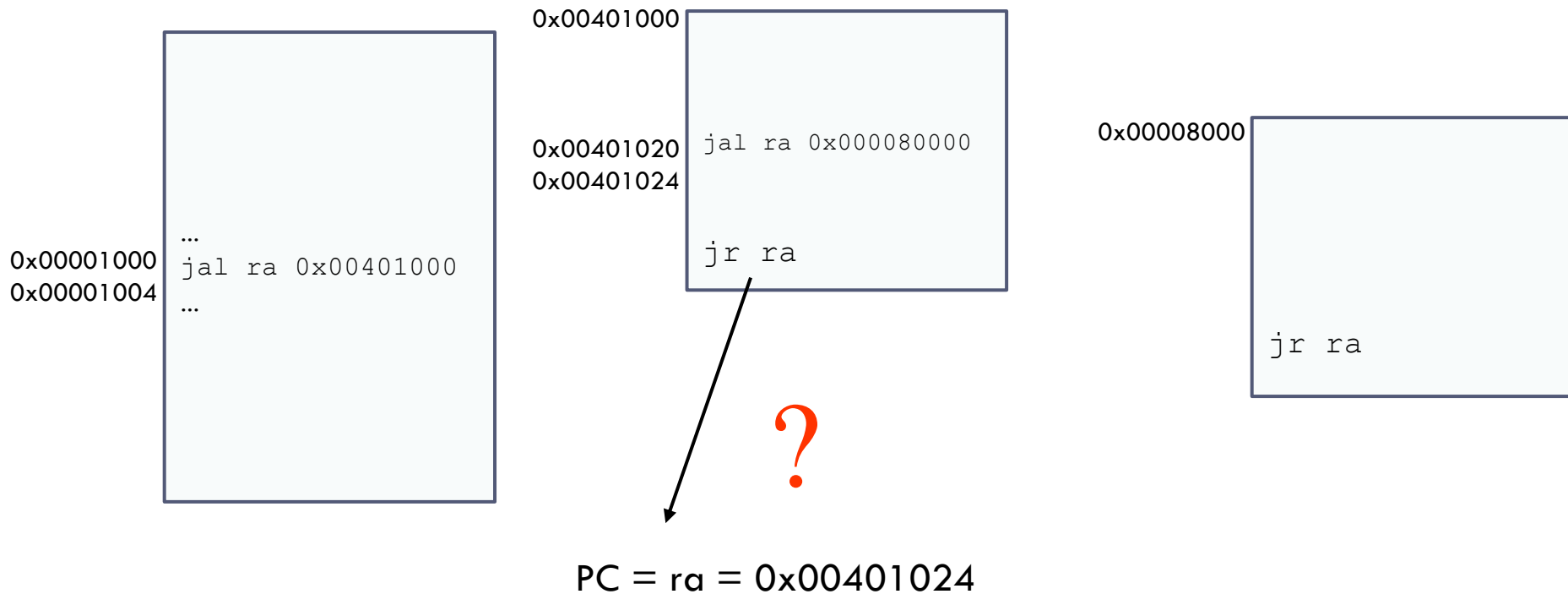
~~Dirección de retorno ra = PC = 0x00001004~~

# Llamadas anidadas



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

# Llamadas anidadas



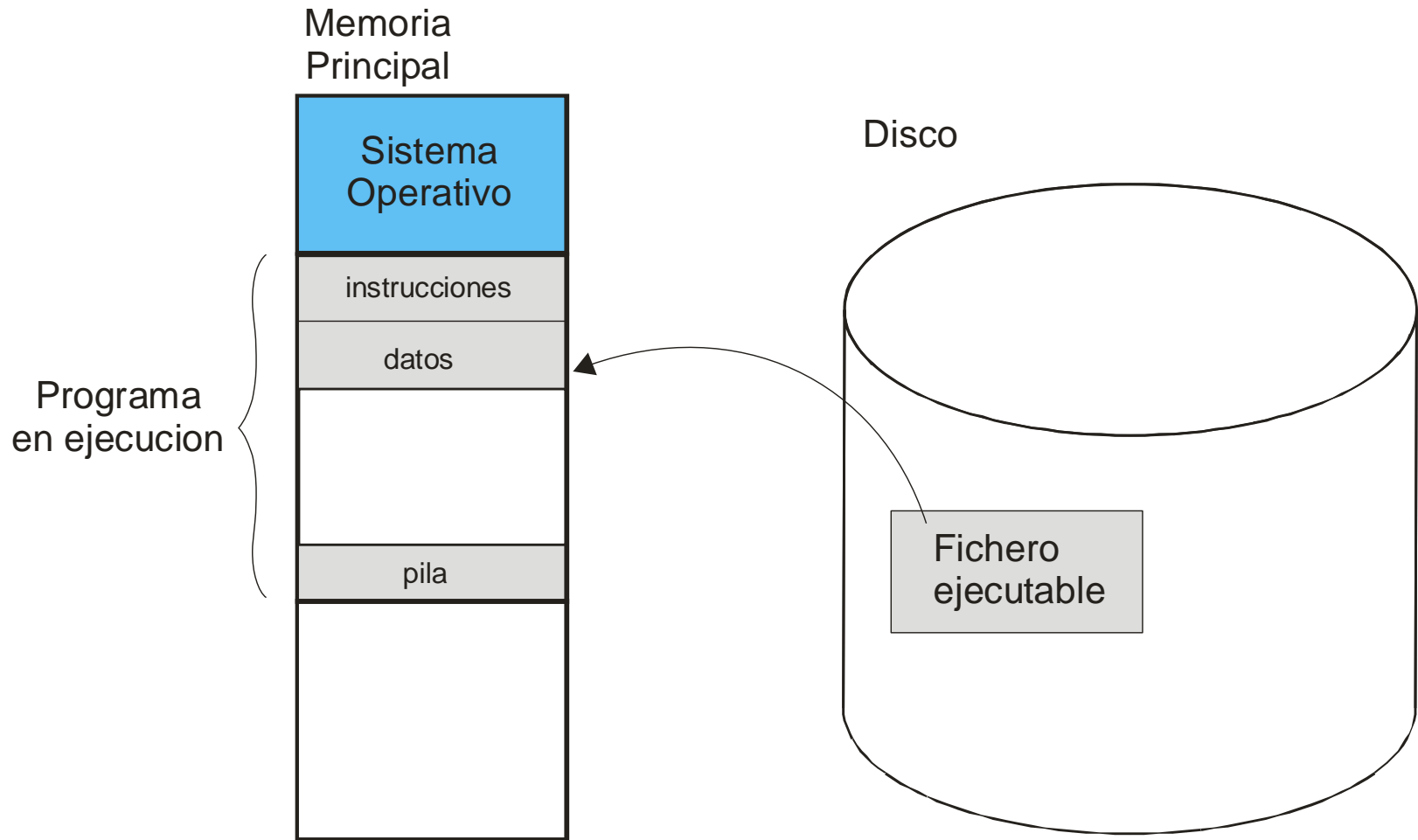
Se ha perdido la dirección de retorno



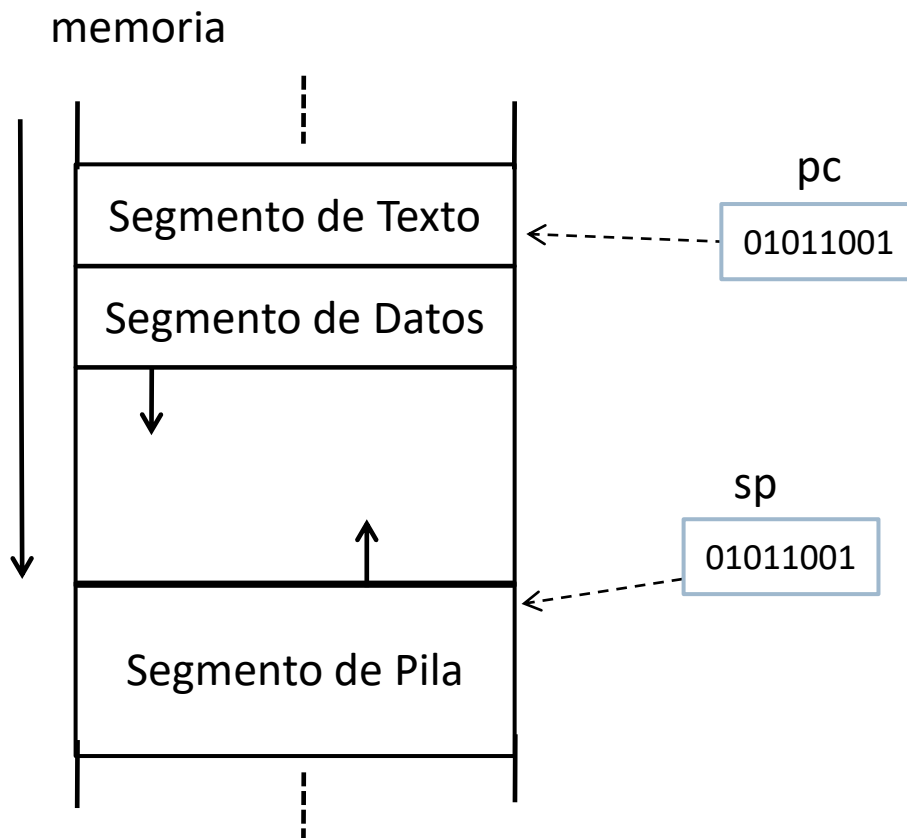
# ¿Dónde guardar la dirección de retorno?

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

# Ejecución de un programa



# Mapa de memoria de un proceso

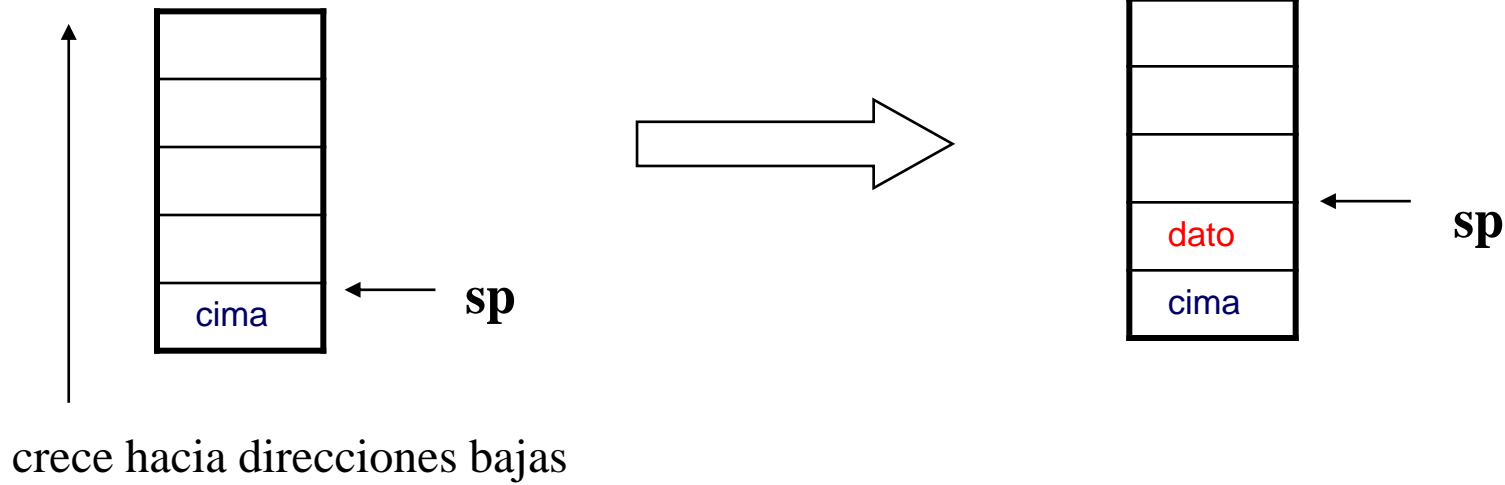


- ▶ El programa de usuario se divide en segmentos:
  - ▶ Segmento de código (texto)
    - ▶ Código, instrucciones máquina
  - ▶ Segmento de datos
    - ▶ Datos estáticos, variables globales
  - ▶ Segmento de pila
    - ▶ Variables locales
    - ▶ Contexto de funciones

# Pila

## PUSH Reg

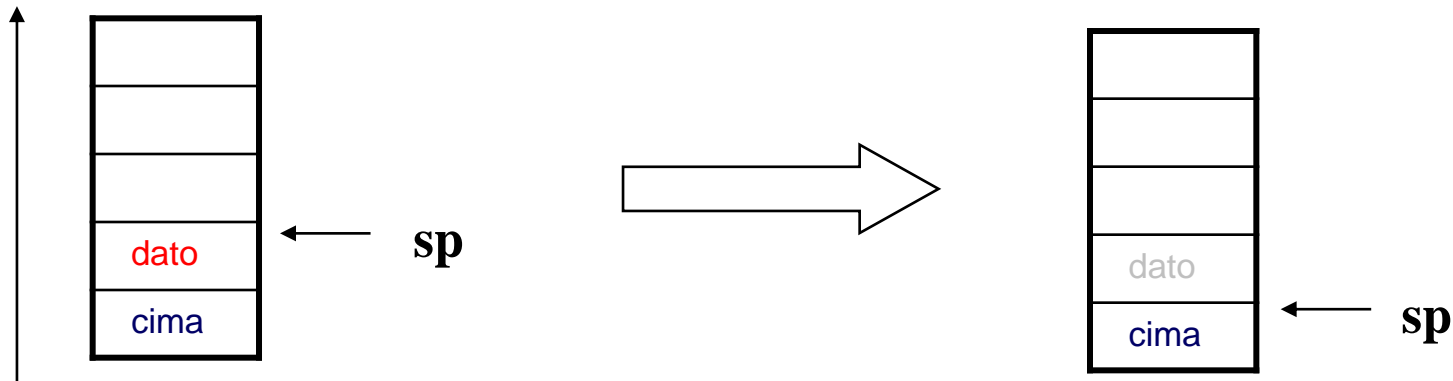
Apila el contenido del registro (dato)



# Pila

## POP Reg

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



crece hacia direcciones bajas

# Antes de empezar

- ▶ RISC-V no dispone de instrucciones PUSH o POP.
- ▶ El registro puntero de pila (`sp`) es visible al programador.
  - ▶ Se va a asumir que el puntero de pila apunta al último elemento de la pila

## PUSH `t0`

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

## POP `t0`

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

# Operación PUSH en el RISC-V

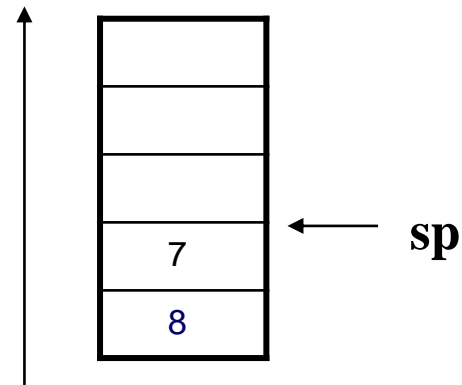
...

```
li    t2, 9
```

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

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

...

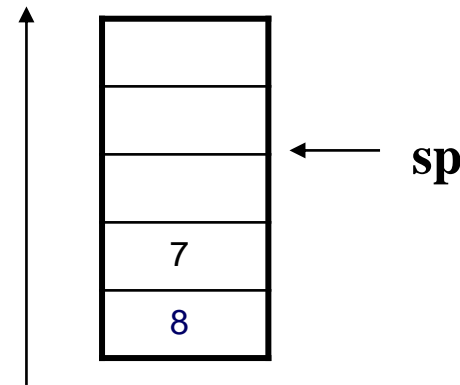


## ► Estado inicial:

- 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)  
...
```

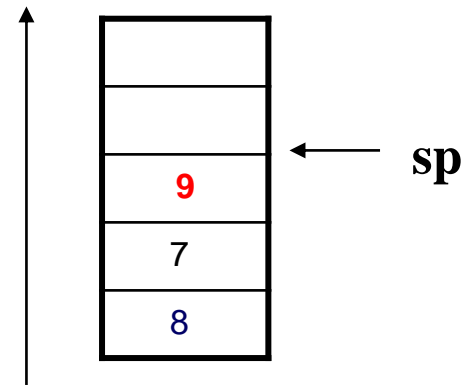


- ▶ Se resta 4 al registro puntero de pila para poder insertar una nueva palabra en la pila
  - ▶ `addi sp, sp, -4`



# 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)`

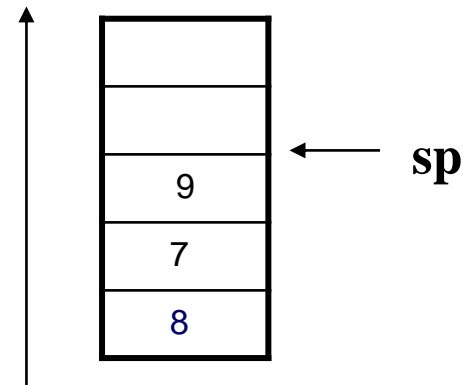
# Operación POP en el RISC-V<sub>32</sub>

...

```
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)

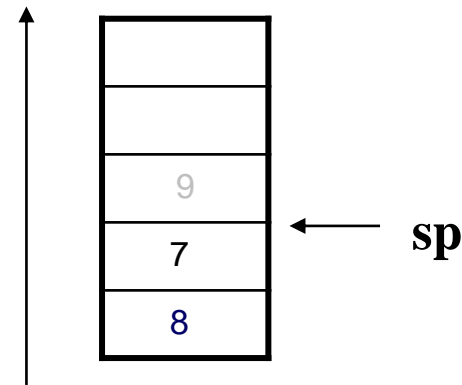
# Operación POP en el RISC-V

...

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

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

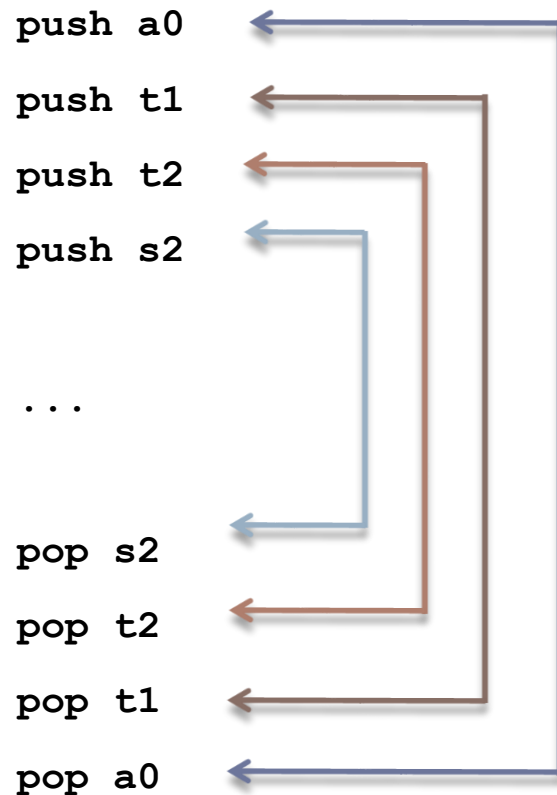
...



- ▶ 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



# Pila

## uso de push y pop consecutivos

```
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

```
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
```

# 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;  
}
```



# 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



# Ejemplo

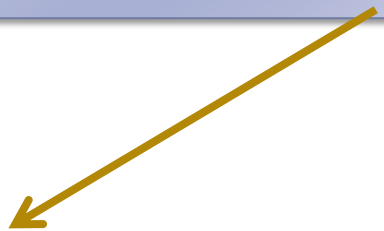
## (2) Pensar en el paso de parámetros

- ▶ Los **parámetros** en RISC-V se pasarán en a0 ... a7
- ▶ Los **resultados** en RISC-V se recogen en a0, a1
  - ▶ Más adelante se verá con más detalle
- ▶ Si se necesita pasar más de ocho parámetros, los ocho primeros en los registros a0...a7 y el resto en la pila
- ▶ En la llamada `z=factorial(5);`
  - ▶ Un parámetro de entrada: en a0
  - ▶ Un resultado en a0

# Ejemplo

## (3) Se pasa a ensamblador cada función

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



```
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 v0          # result  
# print_int(z)  
li a7, 1  
ecall  
...  
  
→ factorial:
```

```
li s1, 1          #s1 for r  
li s0, 1          #s0 for i  
loop: bgt s0, a0, end  
mul s1, s1, s0  
addi s0, s0, 1  
beq x0, x0, loop  
end: mv v0, s1    #result  
jr ra
```

# Ejemplo

(4) Se analizan los registros que se modifican

```
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  
loop:      bgt   s0, a0, end  
           mul   s1, s1, s0  
           addi  s0, s0, 1  
           beq   x0, x0, loop  
end:       move  v0, s1    #result  
           jr    ra
```

- 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

# Ejemplo

(5) Se guardan los registros en la pila

```
int factorial(int x) {  
    int i;  
    int r=1;  
    for (i=1;i<=x;i++) {  
        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  
bucle:    bgt      s0, a0, fin  
           mul      s1, s1, s0  
           addi     s0, s0, 1  
           beq      x0, x0, bucle  
fin:      mv       a0, s1      # resultado  
           lw       s1, 0(sp)  
           lw       s0, 4(sp)  
           addi     sp, sp, 8  
           jr       ra
```

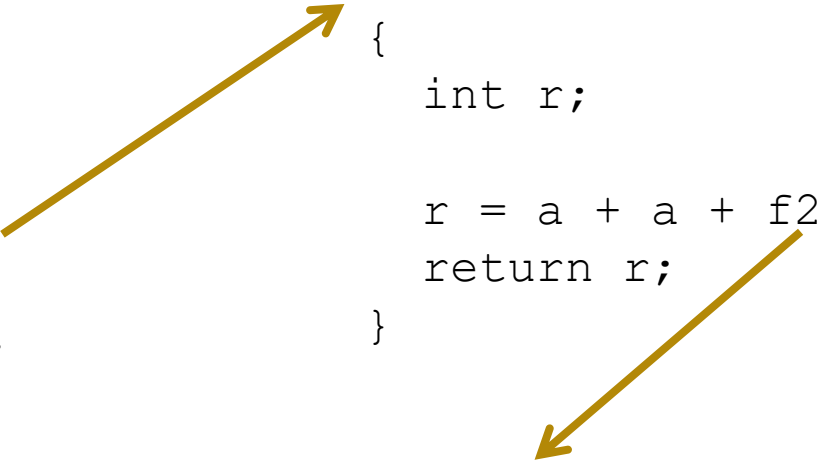
No es necesario guardar ra. La rutina factorial es terminal

Se guarda en la pila s0 y s1 porque se modifican

Si se hubiera utilizado t0 y t1 no habría hecho falta hacerlo (los registros  $t_x$  no se preservan)

# Ejemplo 2

```
int main()  
{  
    int z;  
  
    z=f1(5, 2);  
  
    print_int(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;  
}
```




## Ejemplo 2. Invocación

```
int main()
{
    int z;

    z=f1(5, 2);

    print_int(z);
}
```



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


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

Los parámetros se pasan en a0 y a1  
El resultado se devuelve en a0

## Ejemplo 2. Cuerpo de f1

```
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. Se analizan los registros que se modifican en f1

```
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 porque 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

```
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 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

# Convenio simplificado de 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
  - ▶ Se toman los registros de dos en dos: f10-f11, f12-f13, ...  
Si se necesita pasar más de cuatro parámetros, el resto en la pila

# Retorno de resultados en RISC-V

- ▶ 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

# Convenio de registros

```
li    t0, 8
li    s0, 9

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



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

# Convención en el uso de los registros (RISC-V)

Nombre registro	Número	Uso	Preservar el valor
zero	0	Constante 0	No
ra	1	Dirección de retorno (rutinas)	<b>Si</b>
sp	2	Puntero a pila	<b>Si</b>
gp	3	Puntero al área global	No
tp	4	Puntero al hilo	No
t0...t2	5...7	Temporal	No
s0/fp	8	Temporal / Puntero a marco de pila	<b>Si</b>
sl	9	Temporal	<b>Si</b>
a0...a7	10...17	Argumento de entrada para rutinas	No
s2... s11	18...27	Temporal	<b>Si</b>
t3...t6	28...31	Temporal	No

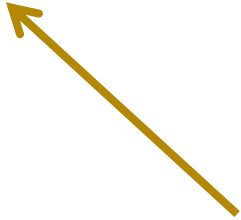
# Convención en el uso de los registros de coma flotante (RISC-V)

Nombre registro	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, funcion
```



De acuerdo al convenio, s0 seguirá valiendo 9, pero no hay garantía de que t0 valga 8 ni que a0 valga 7.

Si queremos que t0 siga valiendo 8 habrá que guardarse en la pila antes de llamar a la 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

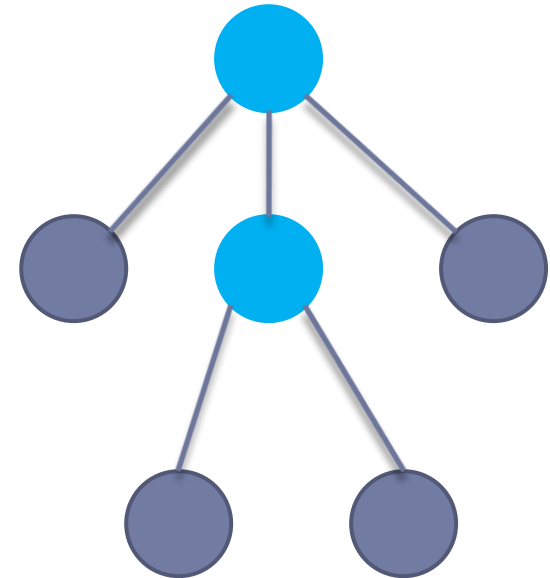
```
li    a0, 7    # parámetro
jal   ra, función
```

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

← Se recupera el valor después

# Tipos de subrutinas

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



# 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

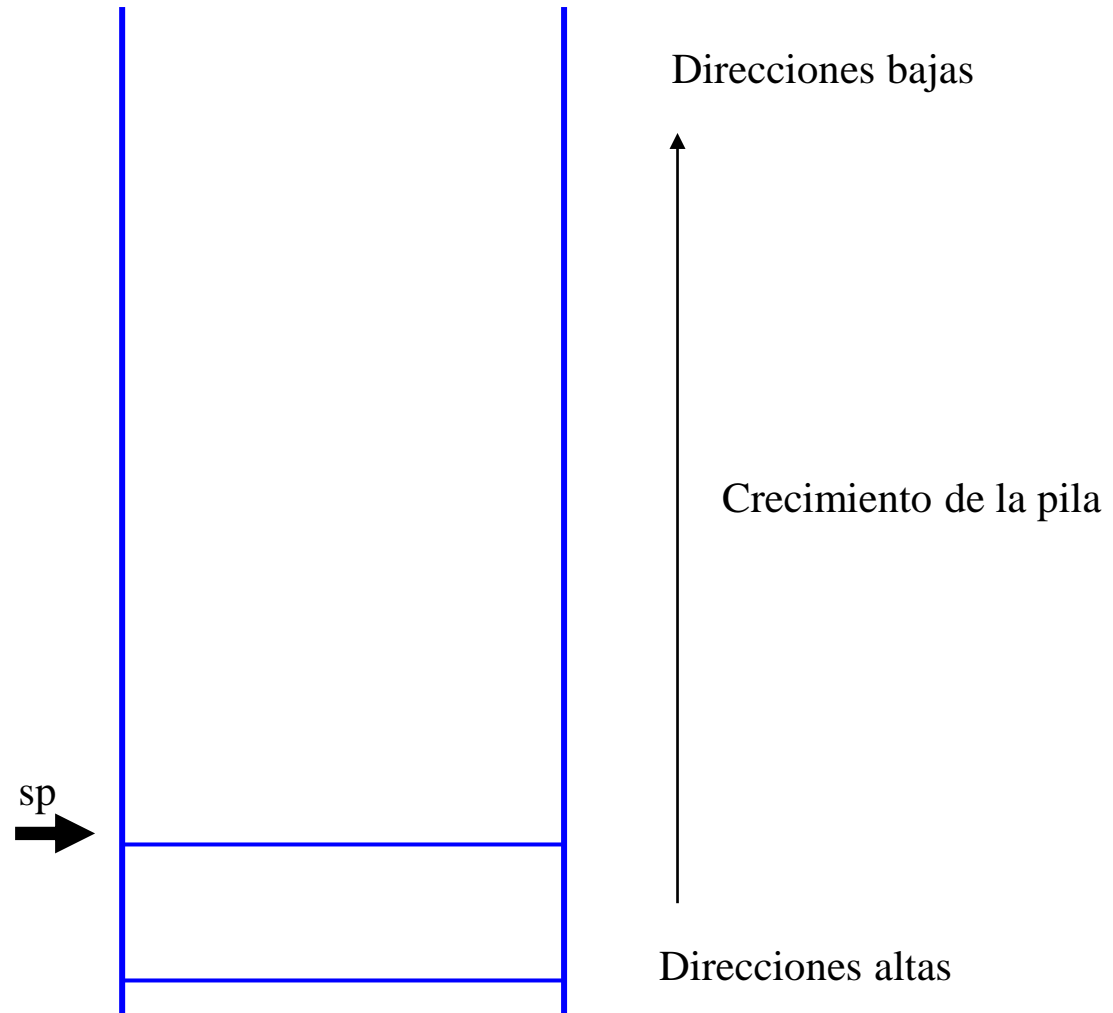
- ▶ El marco de pila almacena:
  - ▶ Los parámetros introducidos por el procedimiento llamante en caso de ser necesarios
  - ▶ Los registros guardados por la función (incluyen al registro `ra` en caso de procedimientos no terminales)
  - ▶ Variables locales

# 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	
Llamada a subrutina (jal)	
	Reserva del marco de pila
	Salvaguarda de registros ( $ra, s_x$ )
	Ejecución de subrutina
	Restauración de valores guardados
	Copiar valores a devolver en el espacio reservado por el llamante
	Liberación de marco de pila
	Salida de subrutina (jr ra)
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

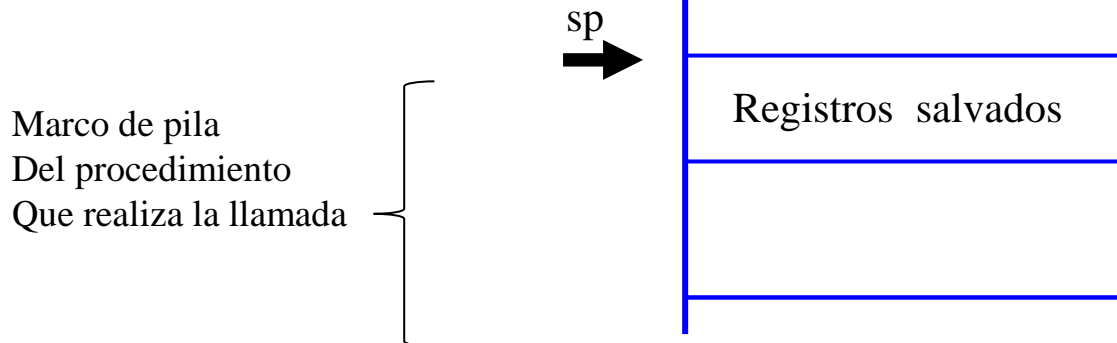
Una subrutina puede modificar cualquier registro **a** y **t**

## 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  $\mathbf{a_x}$  ( $\mathbf{f_x}$ )

El resto de parámetros en la pila

li a0, 1

li a1, 2

li a3, 3

li a4, 4

li a5, 1

li a6, 2

li a7, 3

**addi sp, sp, -8**

**li t0, 5**

**sw t0, 4(sp)**

**li t0, 6**

**sw t0, 0(sp)**

sp  
→

primero

Parámetros

último

Registros salvados

Marco de pila

Del procedimiento

Que realiza la llamada

# Construcción del marco de pila subrutina llamante

**Llamada a subrutina:**

llamada a subrutina

Marco de pila  
Del procedimiento  
Que realiza la llamada

sp  
→

Parámetros

Registros salvados



# Construcción del marco de pila subrutina llamada

## Reserva del marco de pila:

$sp = sp - \text{tamaño marco}$

## Espacio para:

ra,  
s0...s7  
variables locales

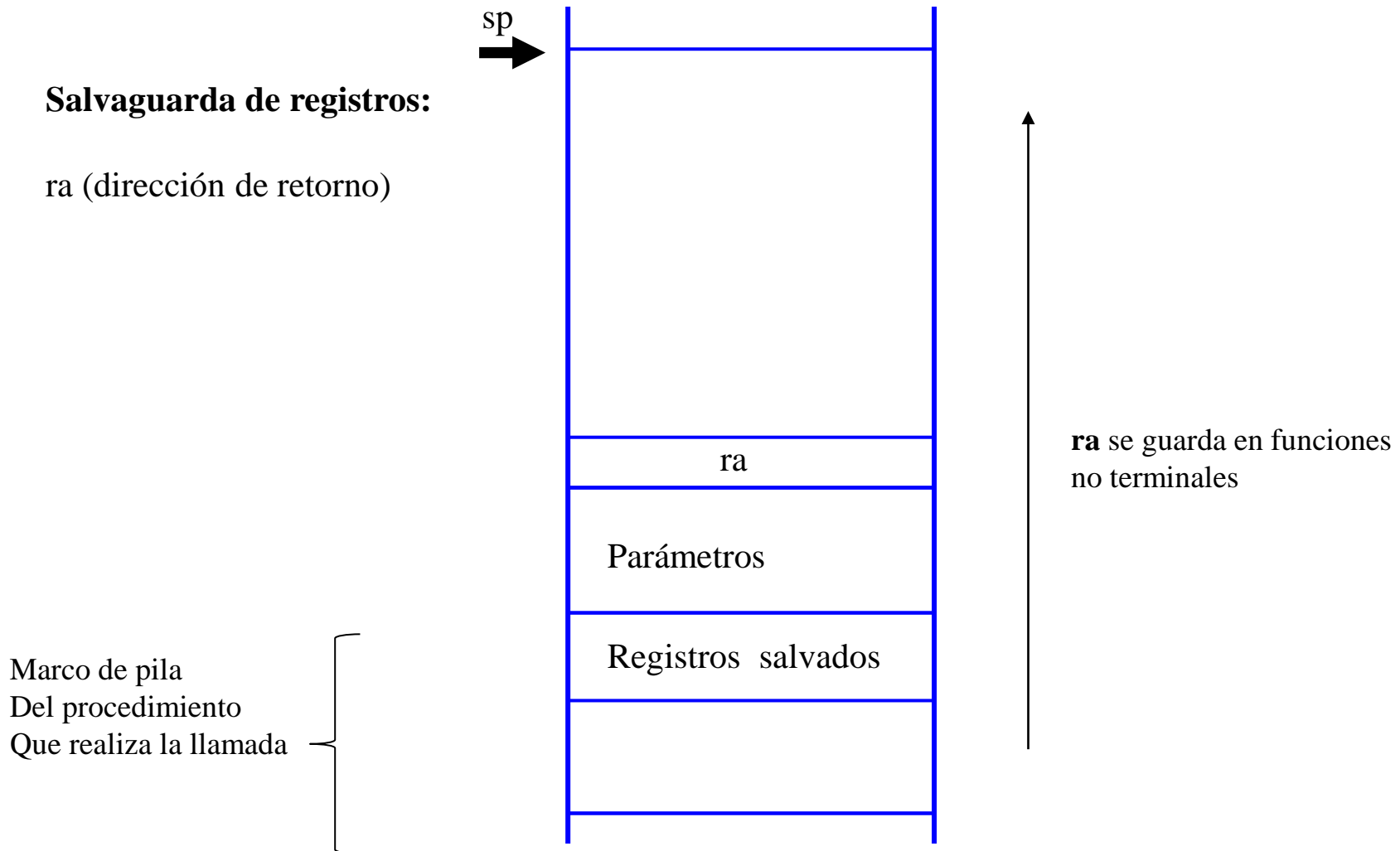
sp  
→

Parámetros

Registros salvados

Marco de pila  
Del procedimiento  
Que realiza la llamada

# Construcción del marco de pila subrutina llamada



# Construcción del marco de pila subrutina llamada

## Salvaguarda de registros $s_x$ :

Se guarda los registros  $s_x$  que se vayan a modificar

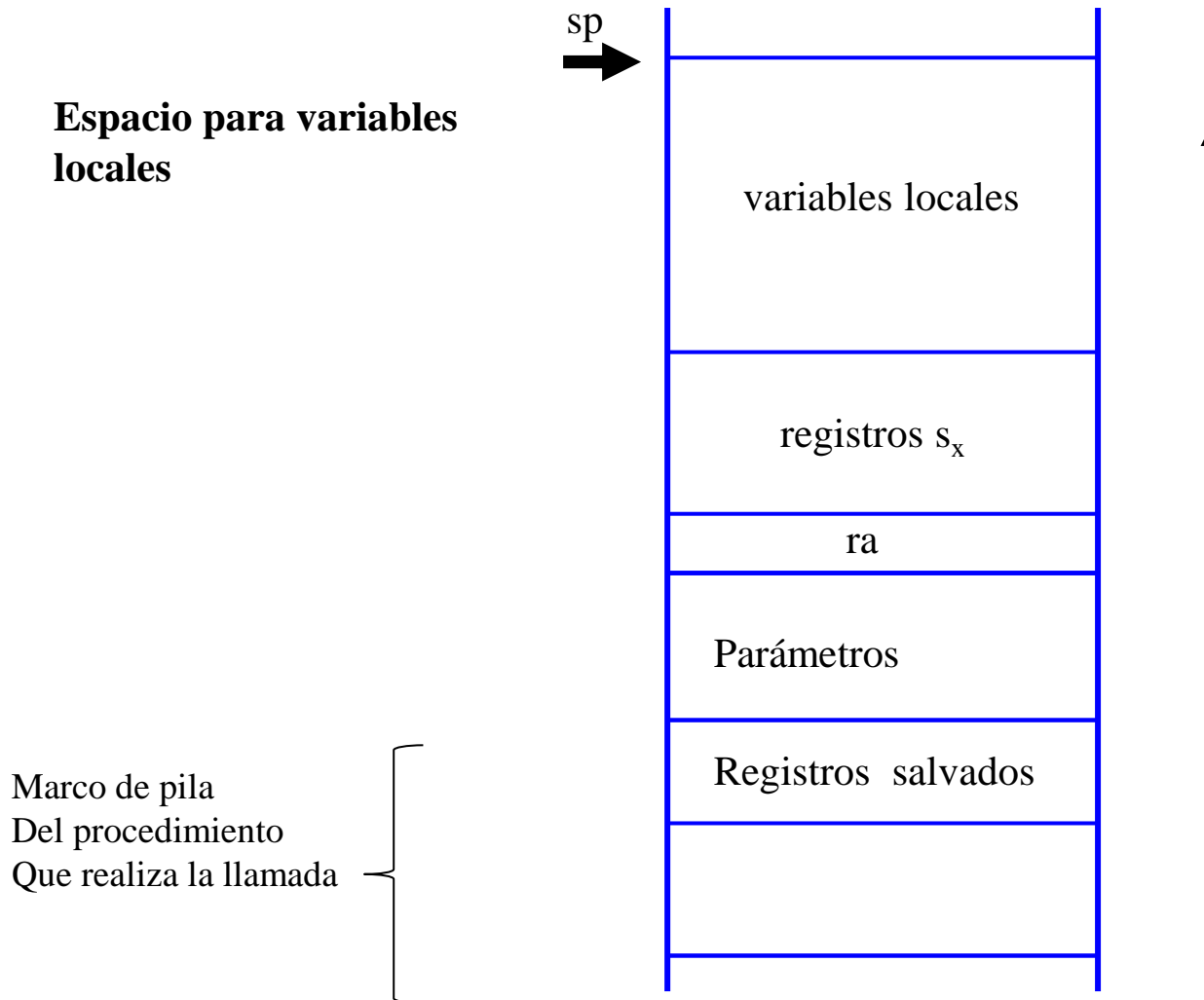
Una función no puede por convenio modificar los registros  $s_x$  (sí lo  $t_x$  y los  $a_x$ )

Marco de pila  
Del procedimiento  
Que realiza la llamada

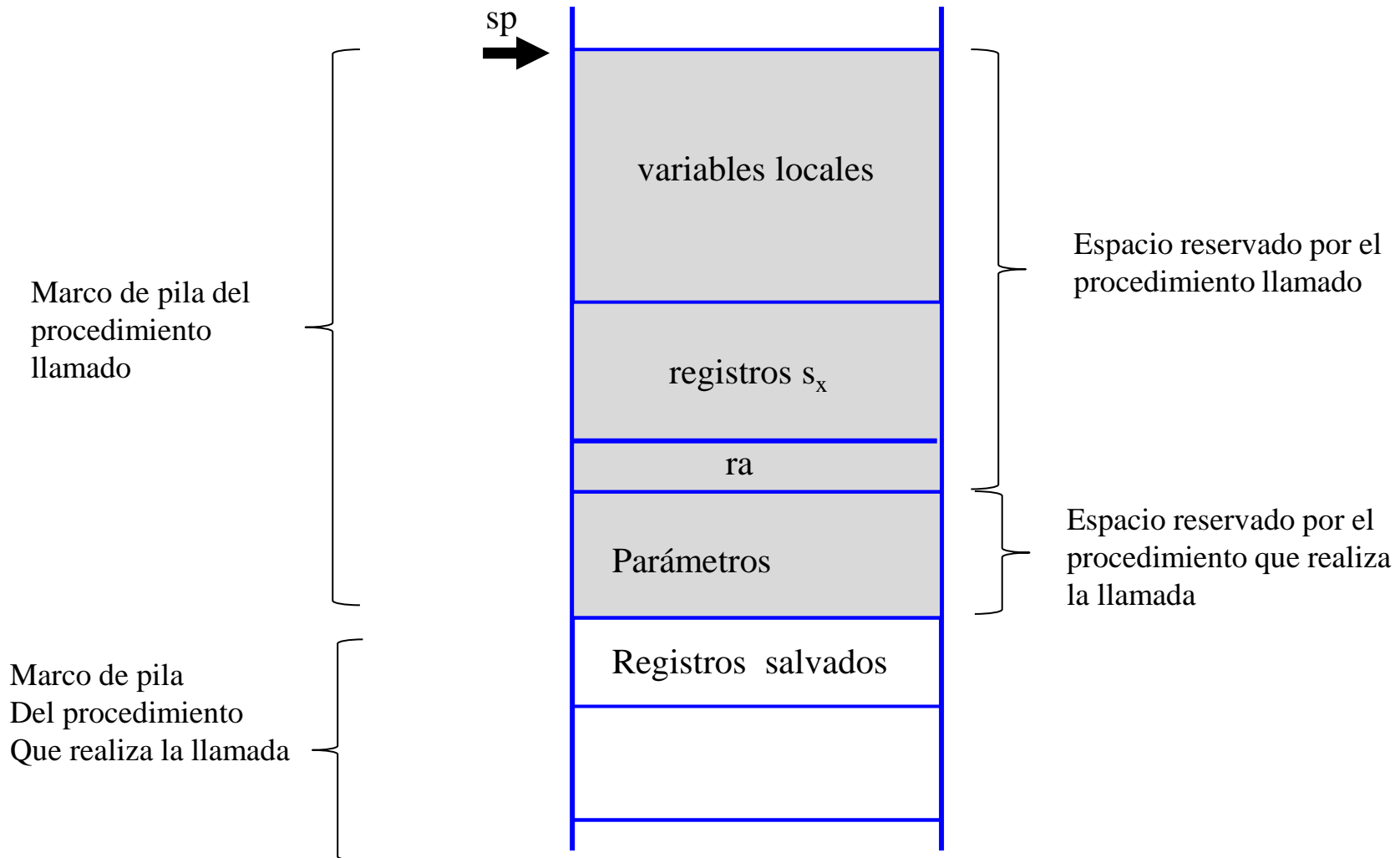




# Construcción del marco de pila subrutina llamada



# Construcción del marco de pila



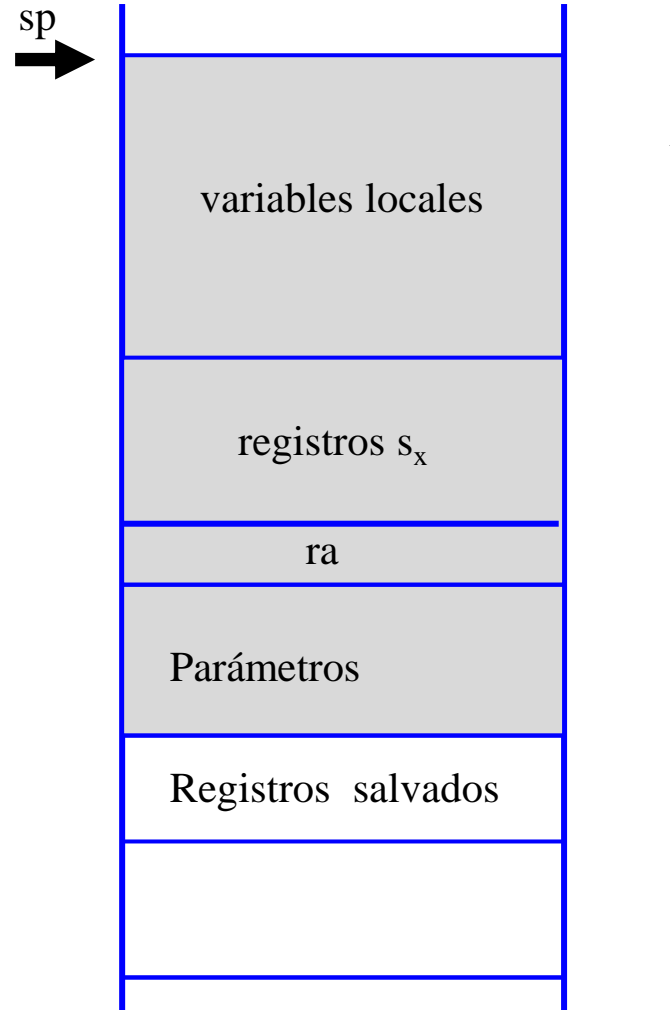
# Finalización de la subrutina subrutina llamada

**Se devuelven los resultados:**

a0, a1, (a1, 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

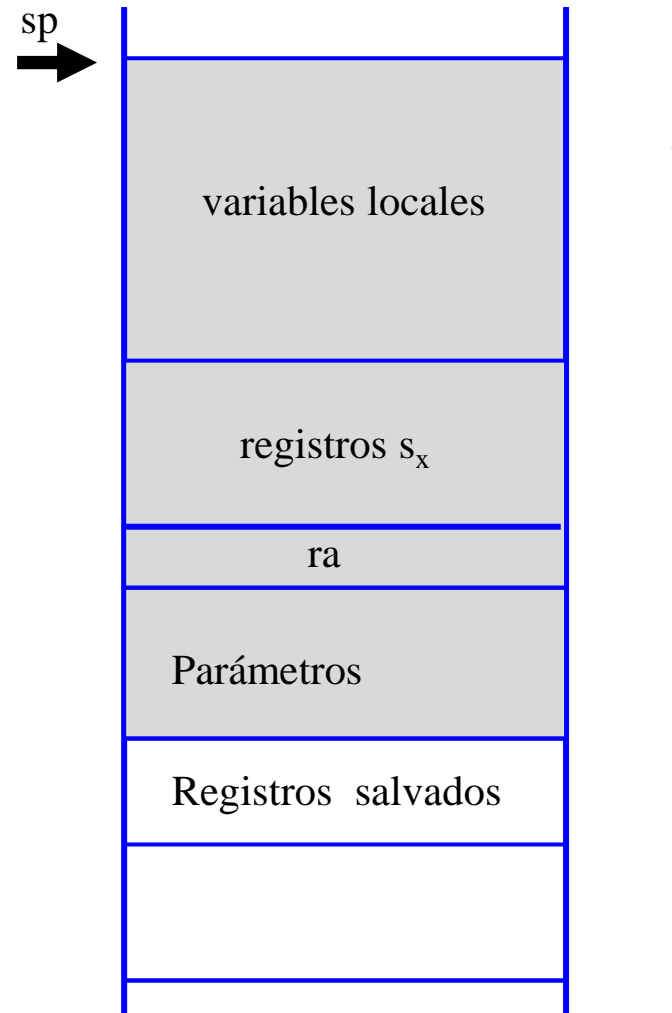


# Finalización de la subrutina subrutina llamada

**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

sp  
→

Parámetros

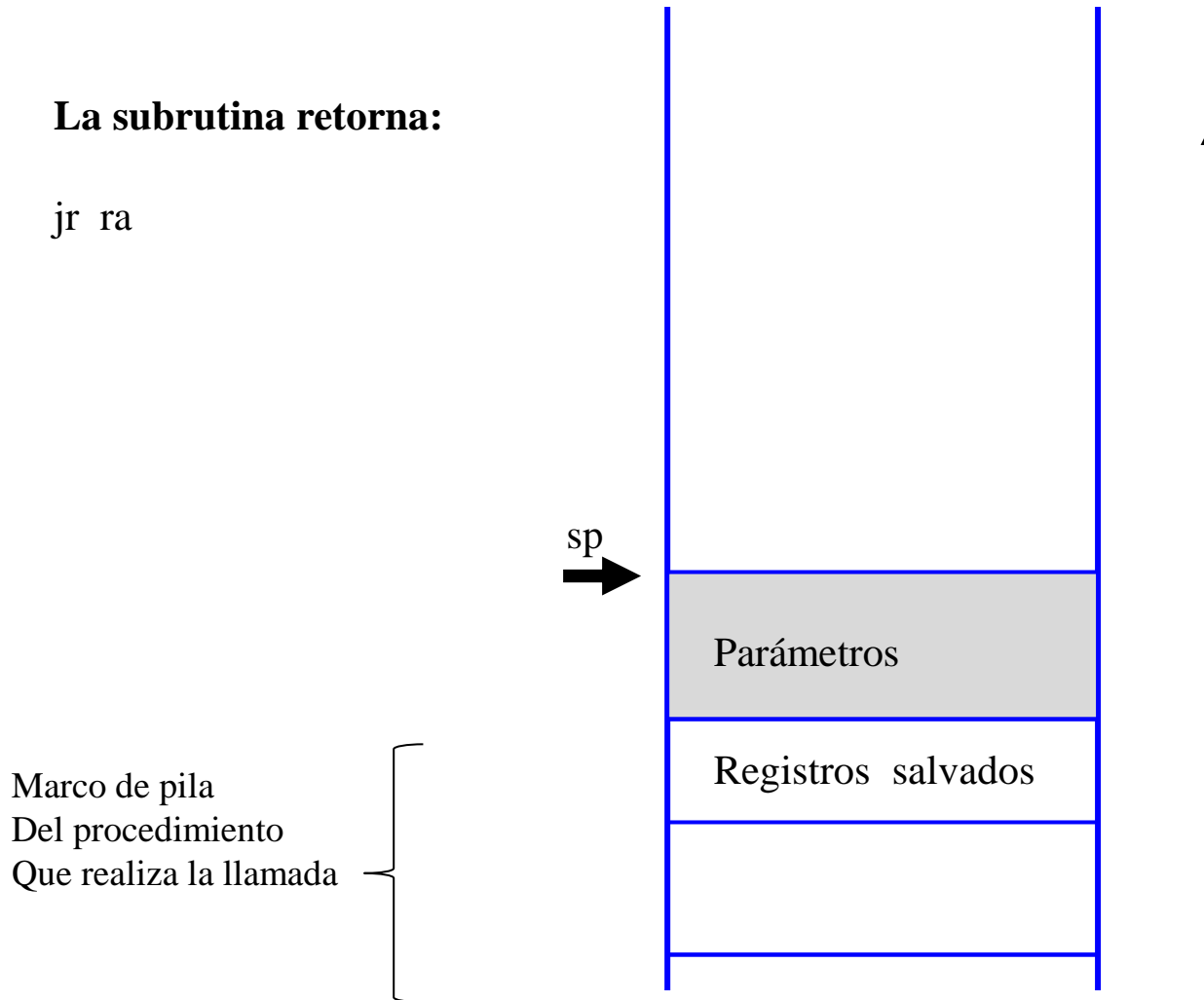
Registros salvados



# 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**

$sp = sp + \text{espacio parámetros}$

Marco de pila  
Del procedimiento  
Que realiza la llamada

sp  
→

Parámetros

Registros salvados

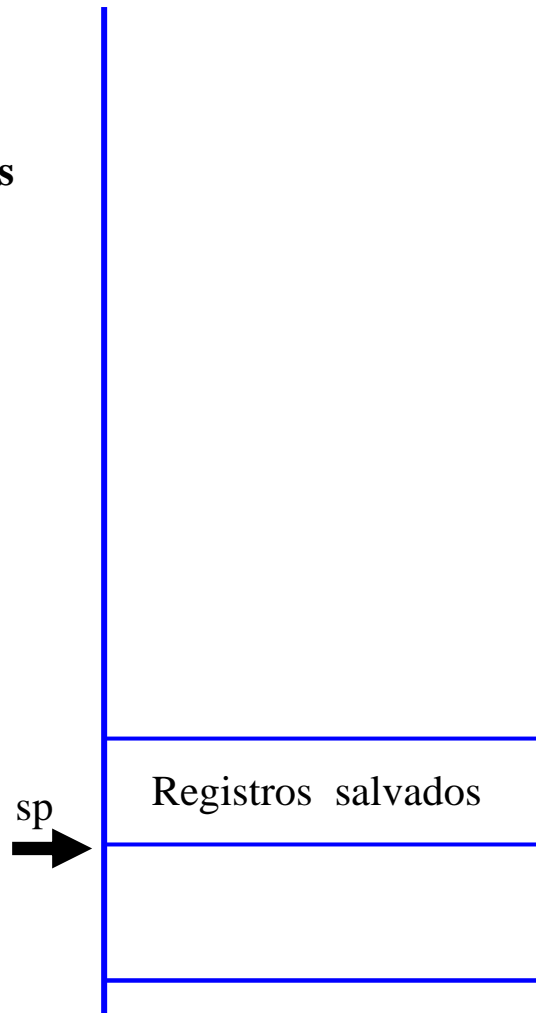


# Finalización de la subrutina subrutina llamante

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 funcion
```

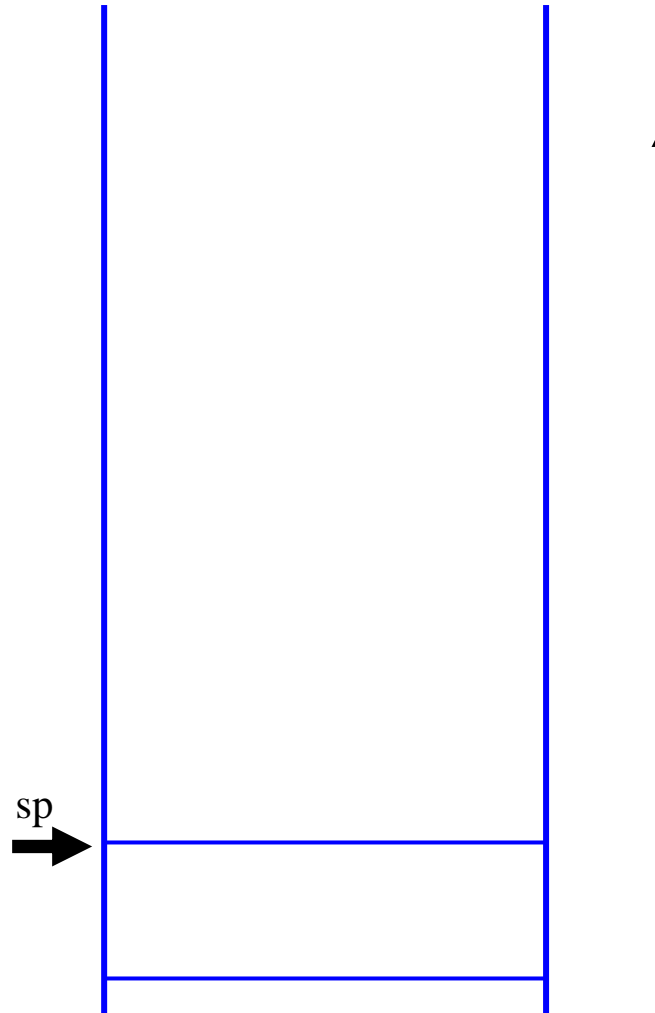
```
lw  t0 0(sp)  
lw  t1 4(sp)  
add sp sp 8
```



# 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  
      . . .
```

# Ejercicio

Considere una función denominada `func` que recibe tres parámetros de tipo entero y devuelve un resultado de tipo entero, y considere el siguiente fragmento del segmento de datos:

```
.data
```

```
    a: .word 5
```

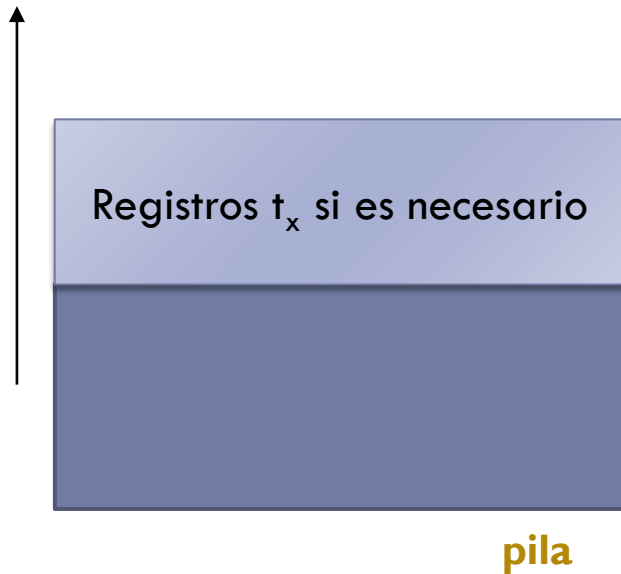
```
    b: .word 7
```

```
    c: .word 9
```

```
.text
```

Indique el código necesario para poder llamar a la función anterior pasando como parámetros los valores de las posiciones de memoria `a`, `b` y `c`. Una vez llamada a la función deberá imprimirse el valor que devuelve la función.

# Paso de 2 parámetros



## Banco de registros

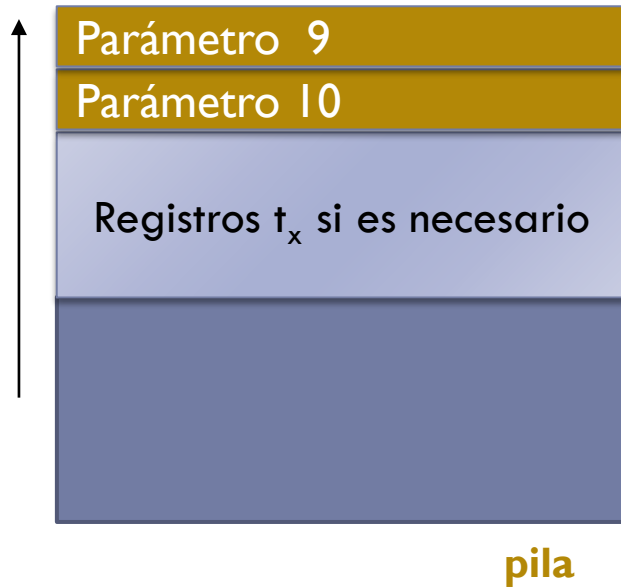
a0	Parámetro 1
a1	Parámetro 2
...	...
a7	Parámetro 8

```
li a0, 5    # param 1
li a1, 8    # param 2

jal ra, func

addi sp, sp, 16
```

# Paso de 10 parámetros



## Banco de registros

a0	Parámetro 1
a1	Parámetro 2
...	...
a7	Parámetro 8

```
li a0, 5      # param 1
li a1, 8      # param 2
...
li a7, 9      # param 8

addi sp, sp, -8
li t0, 10     # param 10
sw t0, 4(sp)
li t0, 7
s2 t0, 0(sp)  # param 9

jal ra, func

addi sp, sp, 8
```

# Asignación dinámica de memoria

- ▶ Llamada al sistema `sbrk()` en RISC-V
  - ▶ `a0`: número de bytes a reservar
  - ▶ `a7 = 9` (código de llamada al sistema)
  - ▶ Devuelve en `v0` la dirección del bloque reservado
  - ▶ En algunos casos para hacer free hay que usar `sbrk` con número negativo

```
int *p;
```

```
p = malloc(20*sizeof(int));
```

```
p[0] = 1;
```

```
p[1] = 4;
```

```
# se reservan 80 bytes
```

```
li a0, 80
```

```
li a7, 9 # código de llamada
```

```
ecall
```

```
mv a0, v0
```

```
li t0, 1
```

```
sw t0, 0(a0)
```

```
li t0, 4
```

```
sw t0, 4(a0)
```

# Traducción y ejecución de programas

- ▶ Elementos que intervienen en la traducción y ejecución de un programa:
  - ▶ Compilador
  - ▶ Ensamblador
  - ▶ Enlazador
  - ▶ Cargador

# Etapas en la traducción y ejecución de un programa (programa en C)

