

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 MIPS 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

- ▶ Un procedimiento (función, subrutina) es un subprograma que realiza una tarea específica cuando se le invoca
  - ▶ Recibe argumentos o parámetros de entrada
  - ▶ Devuelve algún resultado
- ▶ En ensamblador un procedimiento se asocia con un nombre simbólico que denota su dirección de inicio (la dirección de memoria donde se encuentra la primera instrucción de la subrutina)

# Pasos en la ejecución de un procedimiento/función

- ▶ Situar los parámetros en un lugar donde el procedimiento pueda accederlos
- ▶ Transferir el control al procedimiento
- ▶ Adquirir los recursos de almacenamiento necesarios para el procedimiento
- ▶ Realizar la tarea deseada
- ▶ Poner el resultado en un lugar donde el programa o procedimiento 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);  
}
```

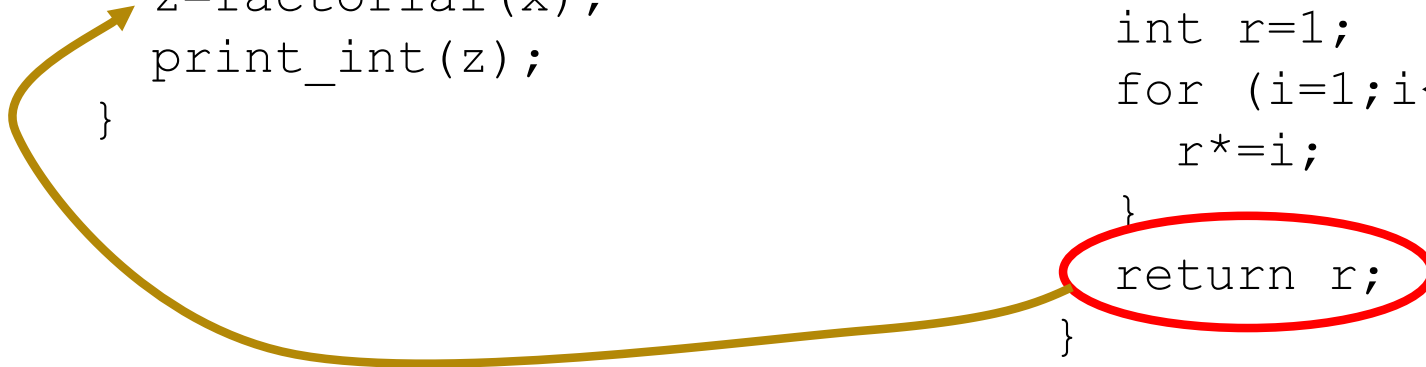


```
int factorial(int x) {  
    int i;  
    int r=1;  
    for (i=1;i<=x;i++) {  
        r*=i;  
    }  
    return r;  
}
```

# 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;  
}
```





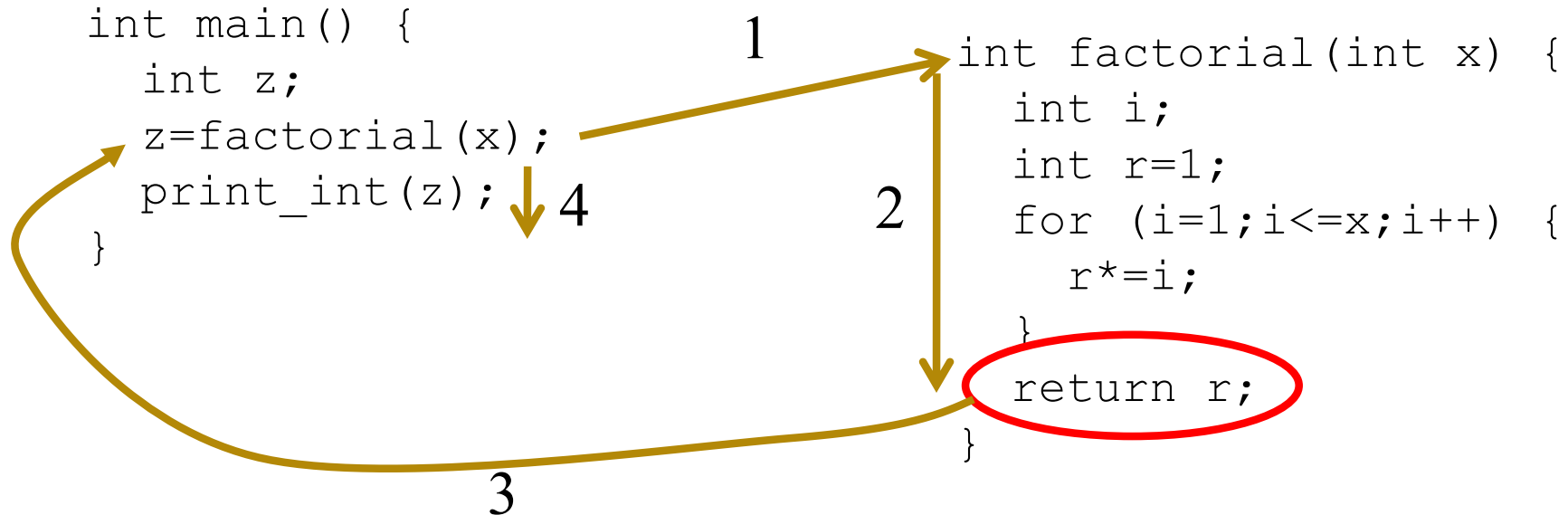
# Funciones en un lenguaje de alto nivel

```
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);  
}
```

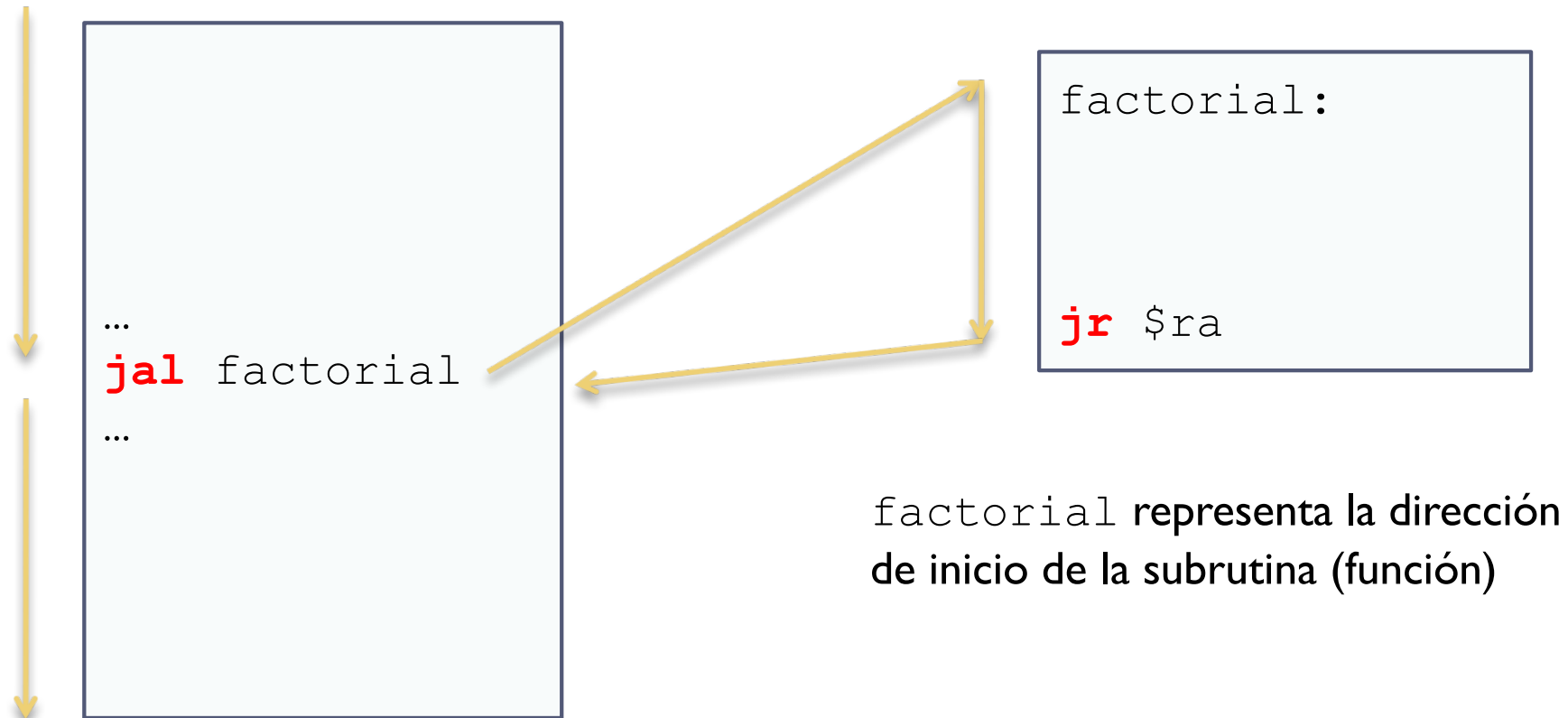
Variables locales



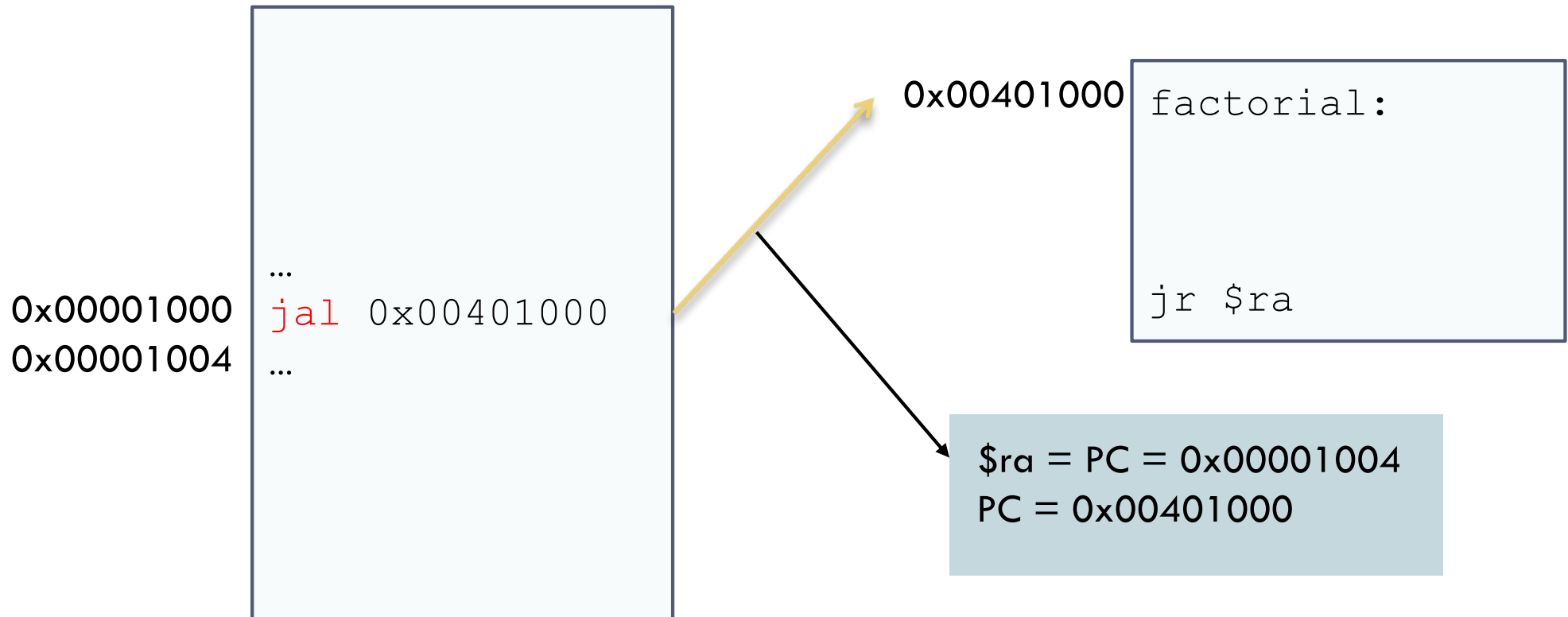
```
int factorial(int x) {  
    int i;  
    int r=1;  
    for (i=1;i<=x;i++) {  
        r*=i;  
    }  
    return r;  
}
```

# Llamadas a funciones en MIPS

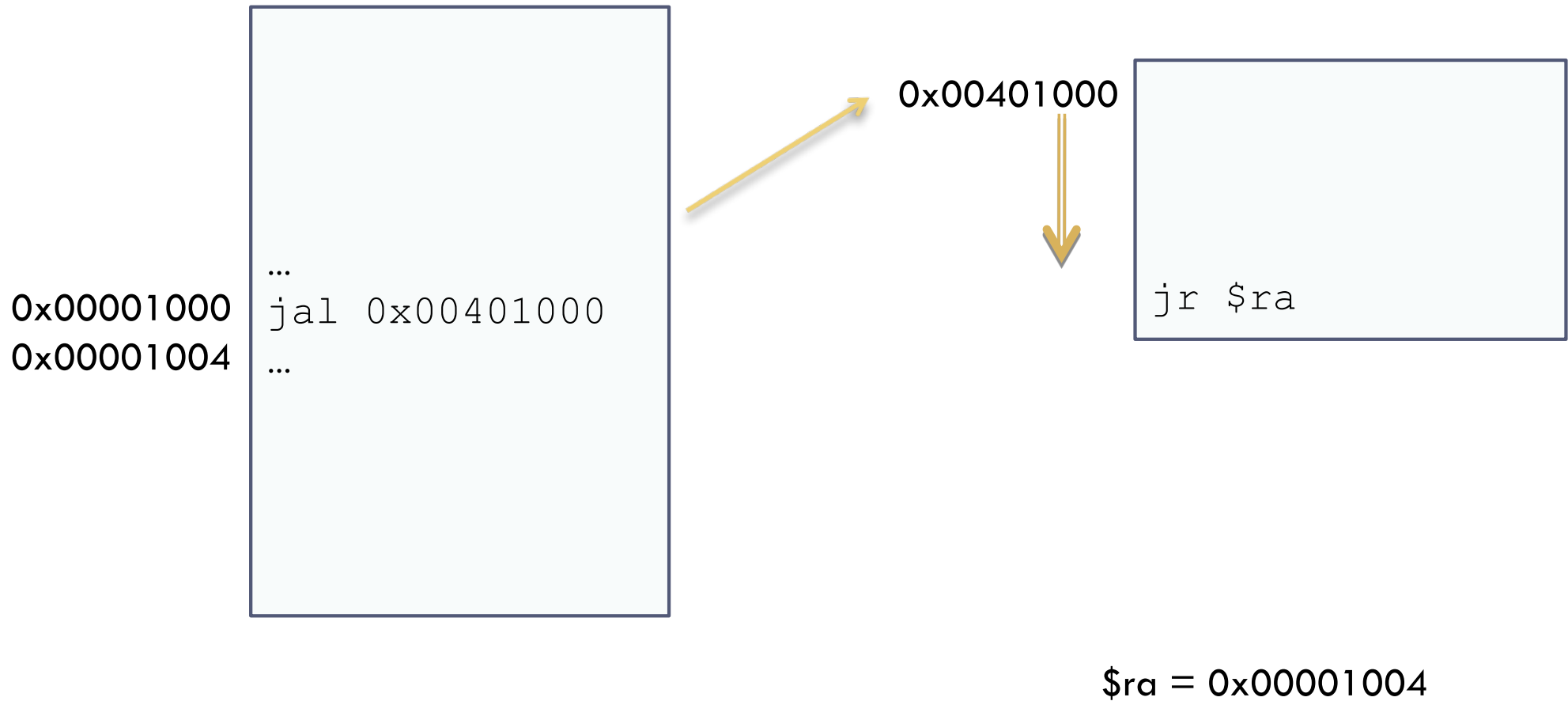
Llamada a función en el MIPS (instrucción `jal`)



# Llamadas a funciones en MIPS

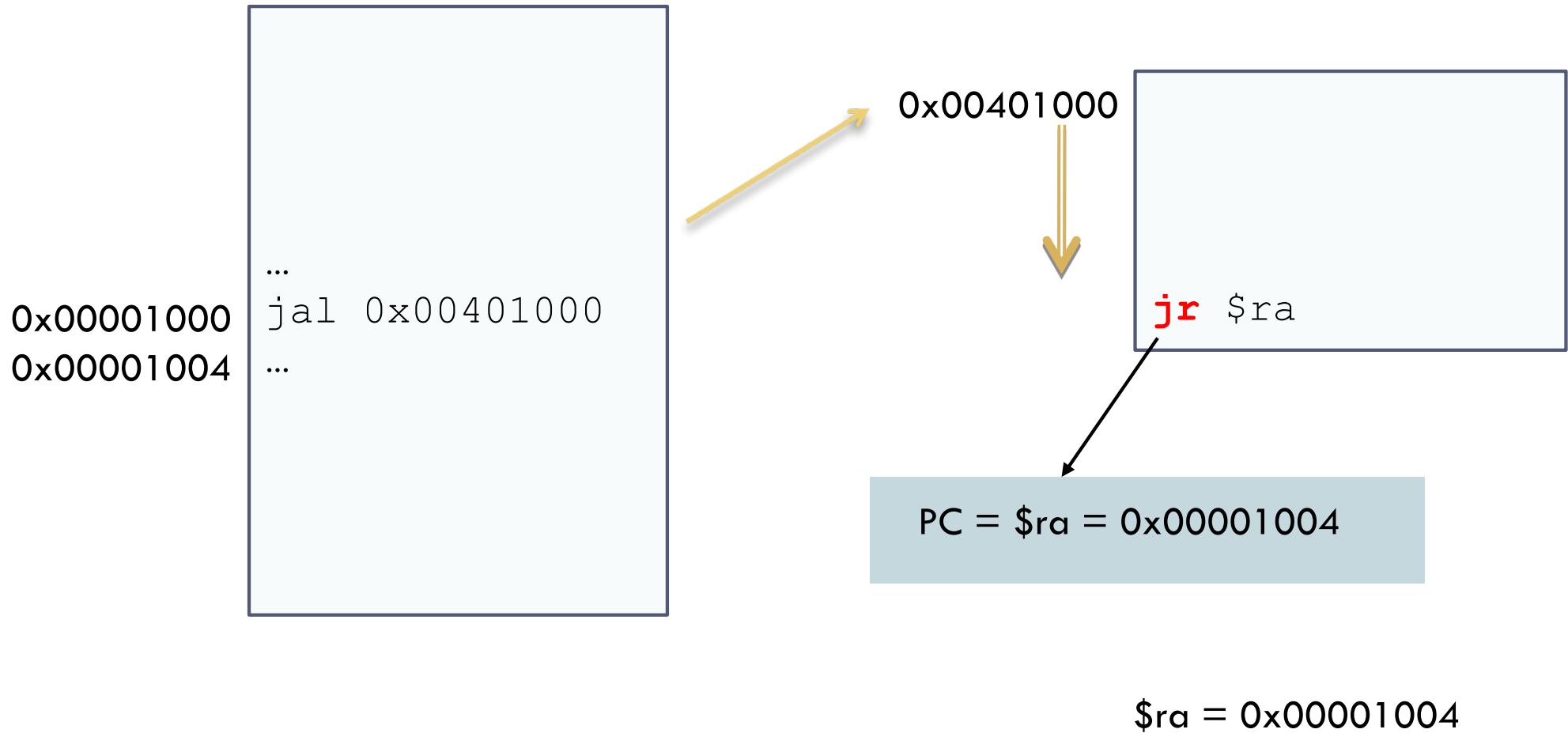


# Llamadas a funciones en MIPS

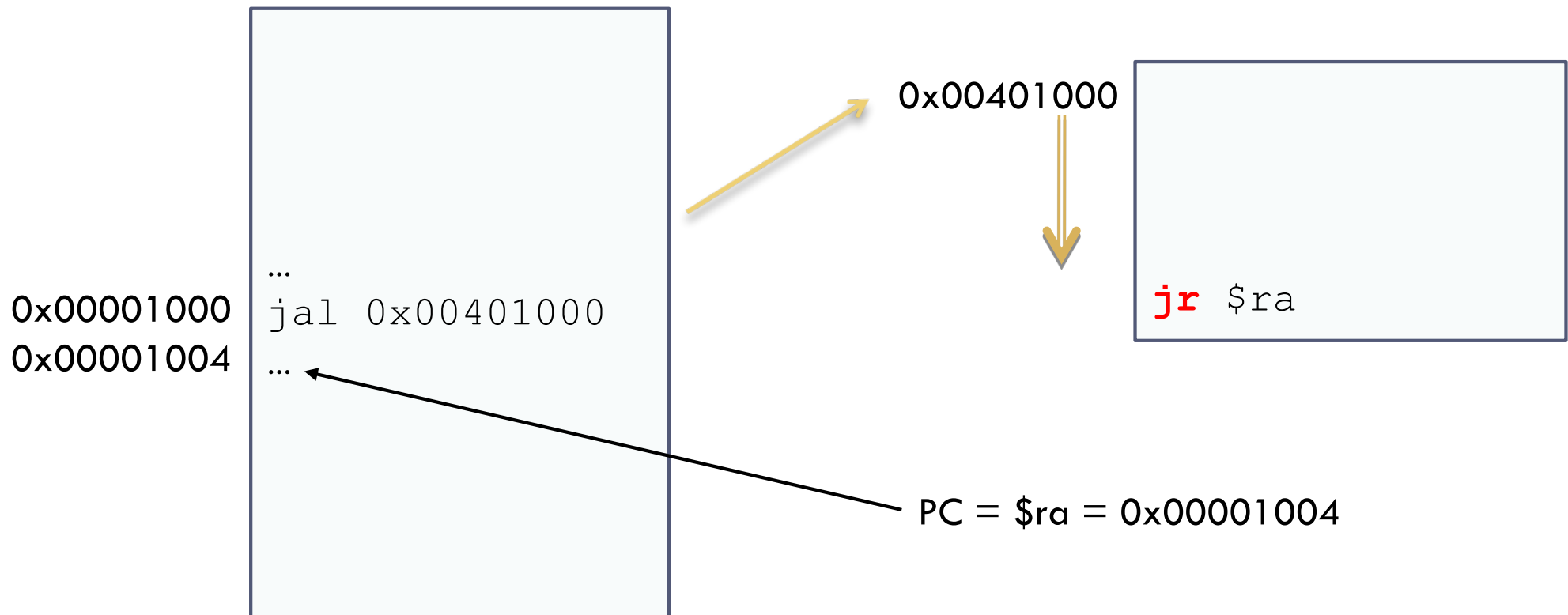


# Llamadas a funciones en MIPS

Retorno de subrutina (instrucción `jr` )



# Llamadas a funciones en MIPS

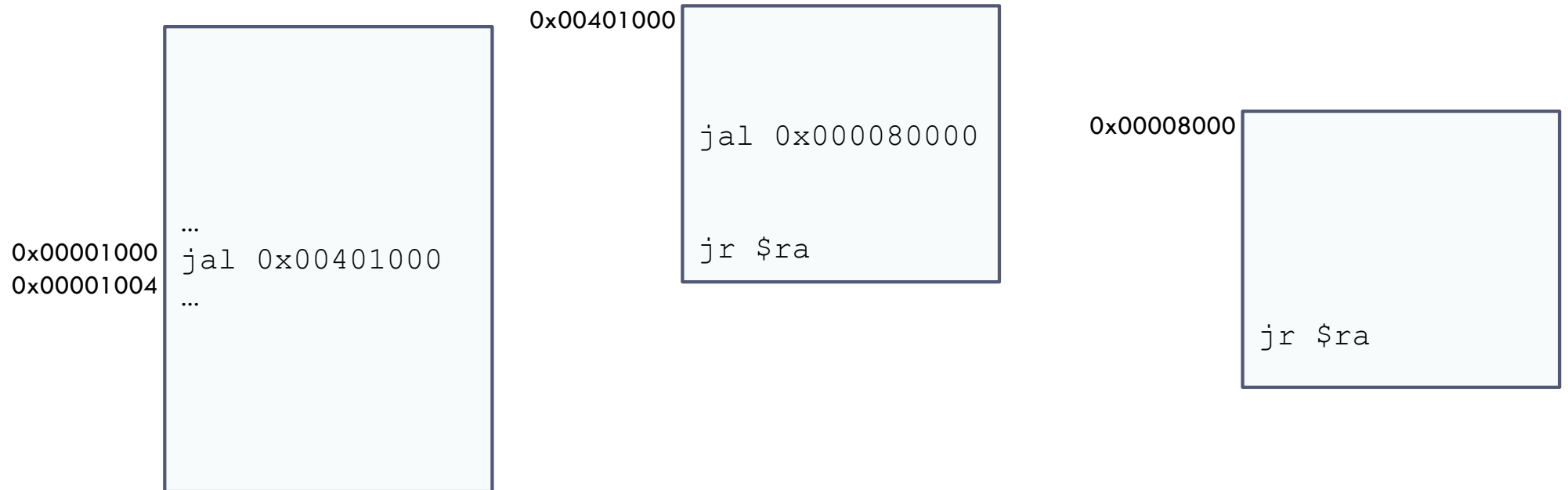




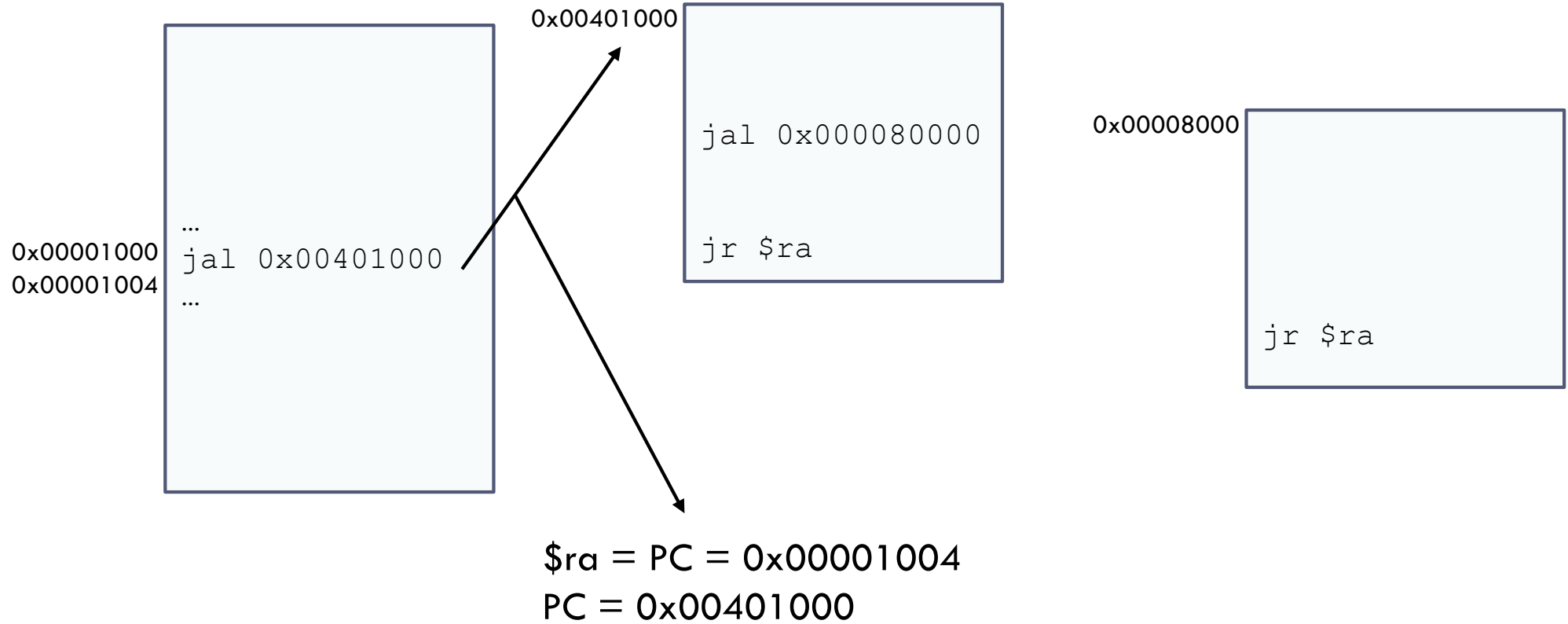
# Instrucciones jal/jr

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

# Llamadas anidadas

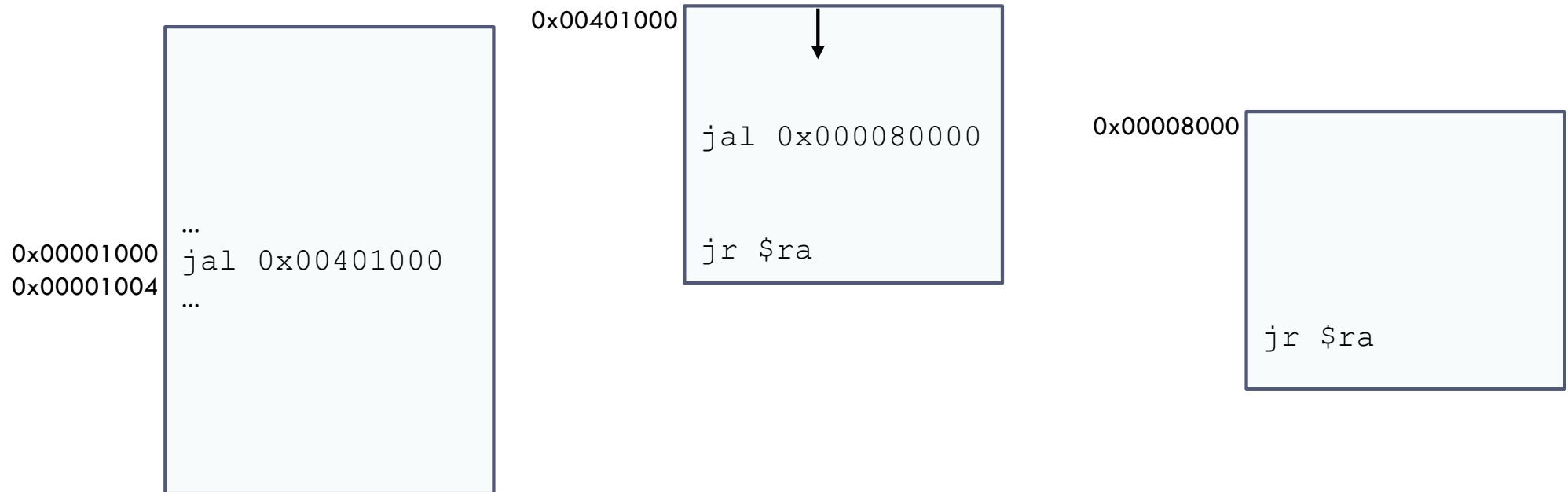


# Llamadas anidadas



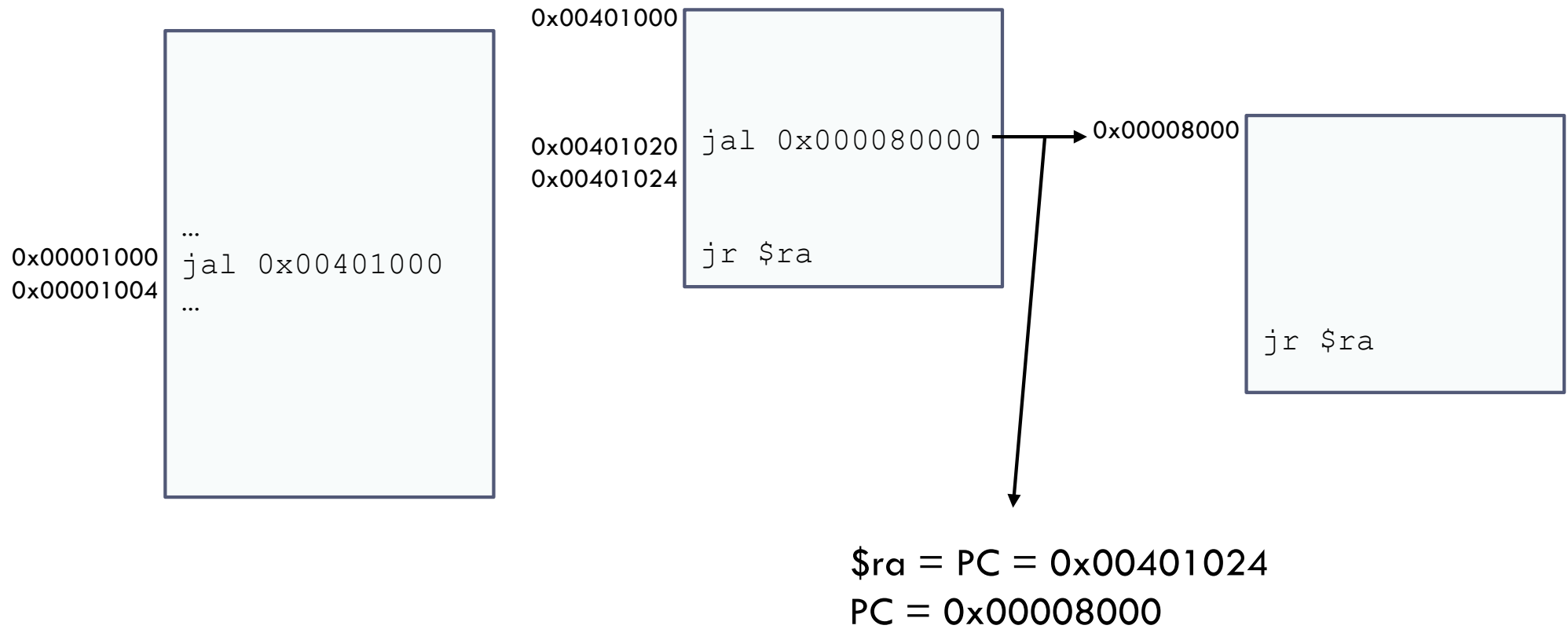
Dirección de retorno `$ra = PC = 0x00001004`

# Llamadas anidadas



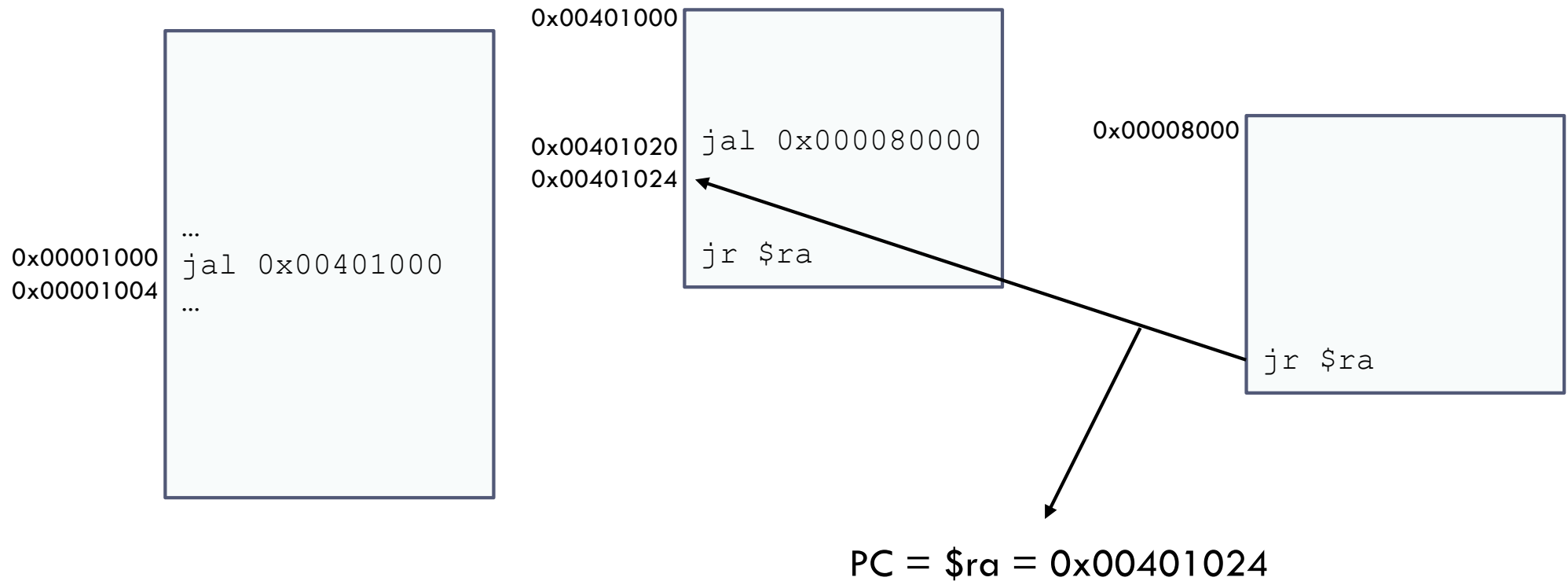
Dirección de retorno  $\$ra = PC = 0x00001004$

# Llamadas anidadas



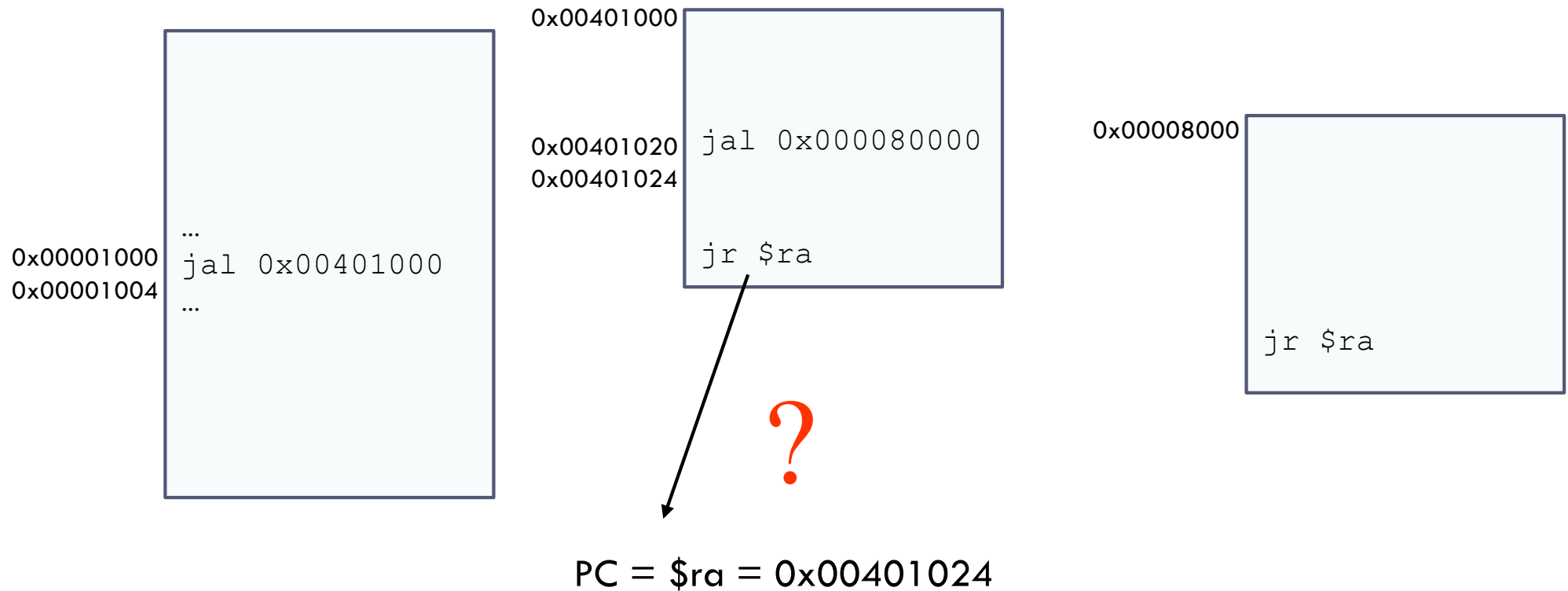
Dirección de retorno  ~~$\$ra = PC = 0x00001004$~~

# Llamadas anidadas

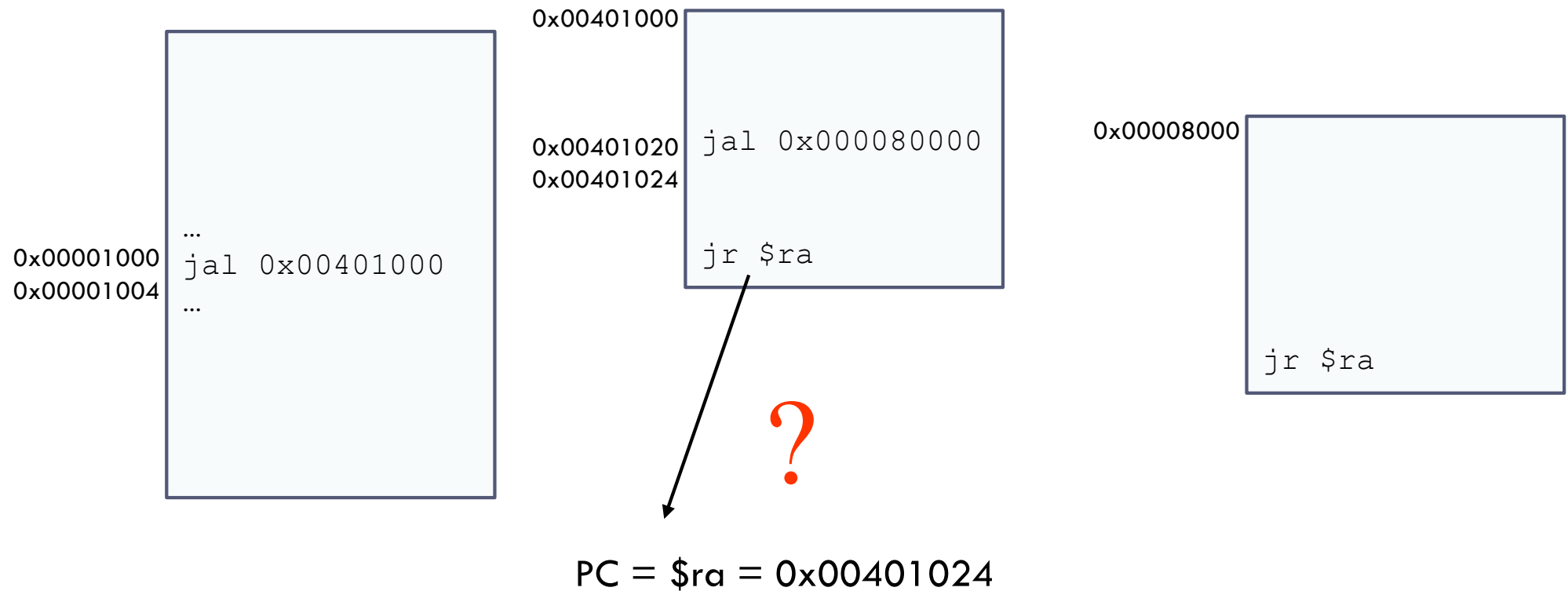


Dirección de retorno  ~~$\neq$~~   $\$ra = PC = 0x00001004$

# Llamadas anidadas



# Llamadas anidadas



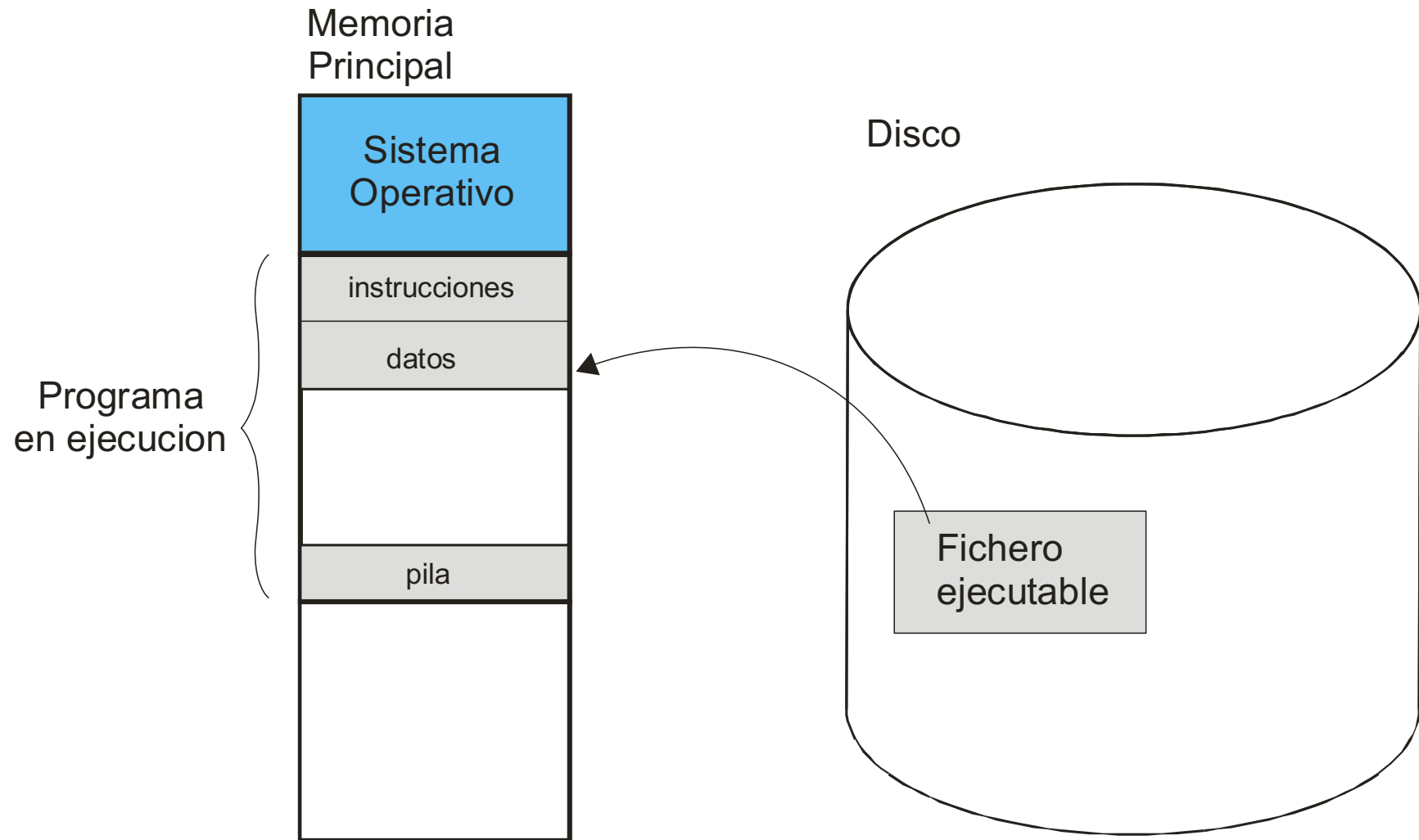
Se ha perdido la dirección de retorno



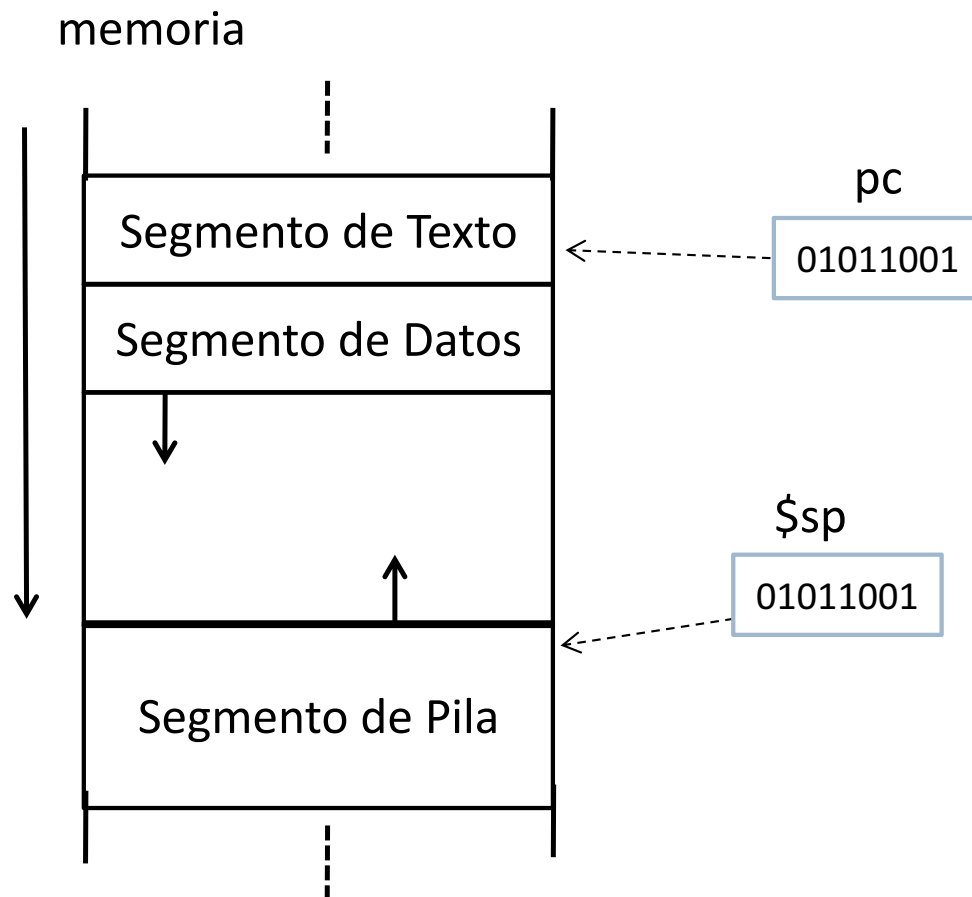
# ¿Dónde guardar la dirección de retorno?

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

# Ejecución de un programa



# Mapa de memoria de un proceso

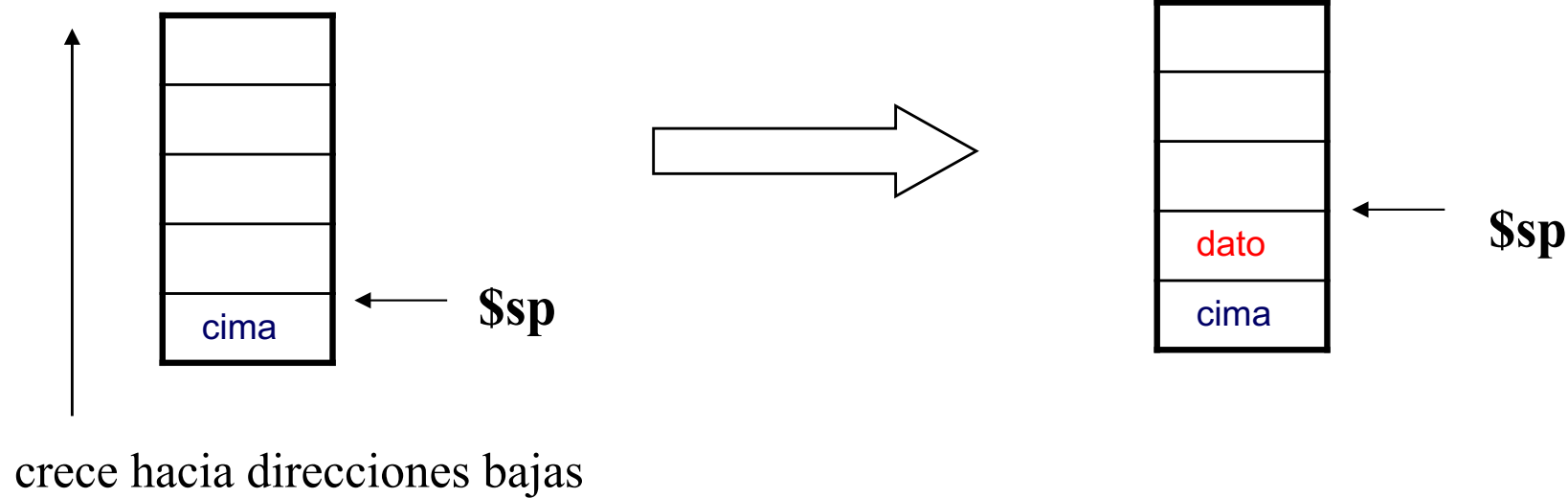


- ▶ El programa de usuario se divide en segmentos:
  - ▶ Segmento de código (texto)
    - ▶ Código, instrucciones máquina
  - ▶ Segmento de datos
    - ▶ Datos estáticos, variables globales
  - ▶ Segmento de pila
    - ▶ Variables locales
    - ▶ Contexto de funciones

# Pila

## PUSH Reg

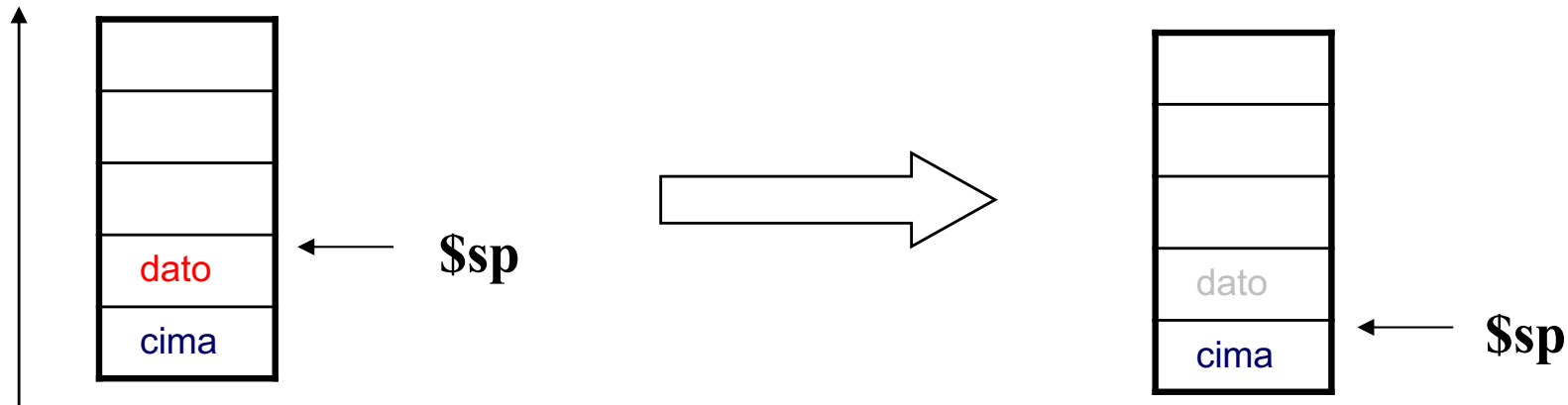
Apila el contenido del registro (dato)



# Pila

## POP Reg

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



crece hacia direcciones bajas

# Antes de empezar

- ▶ MIPS 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

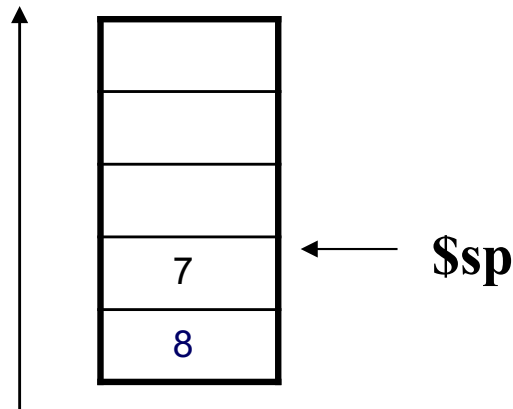
## PUSH $\$t0$

```
addu $sp, $sp, -4  
sw   $t0, ($sp)
```

## POP $\$t0$

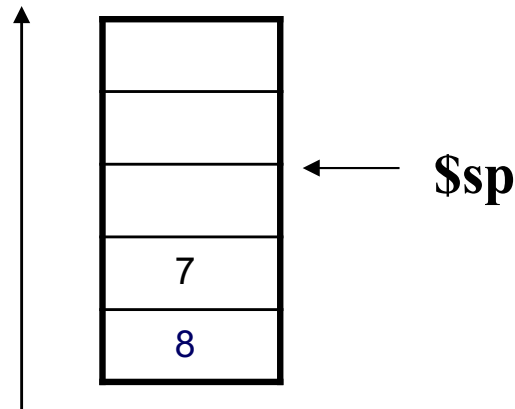
```
lw   $t0, ($sp)  
addu $sp, $sp, 4
```

# Operación PUSH en el MIPS



Estado inicial: el registro puntero de pila (\$sp) apunta al último elemento situado en la cima de la pila

# Operación PUSH en el MIPS

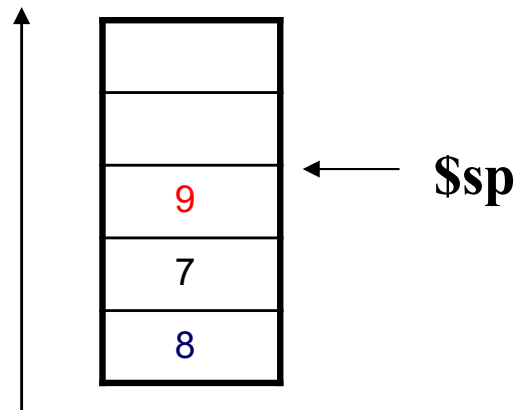


**addu \$sp, \$sp, -4**

Se resta 4 al registro puntero de pila para poder insertar una nueva palabra en la pila



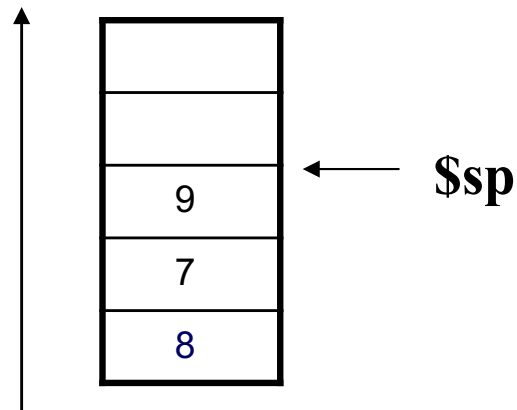
# Operación PUSH en el MIPS



```
li    $t2, 9  
sw    $t2 ($sp)
```

Se inserta el contenido del registro \$t2 en la cima de la pila

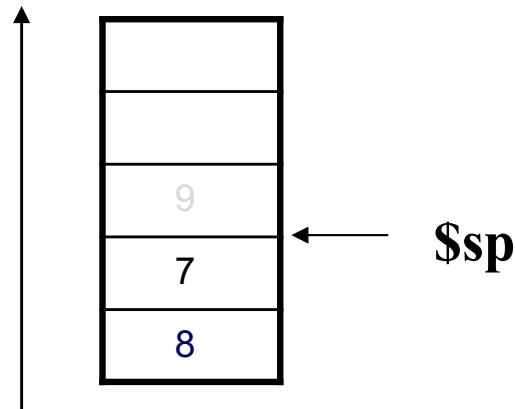
# Operación POP en el MIPS



**lw \$t2 (\$sp)**

Se copia en \$t2 el dato almacenado en la cima de la pila (9)

# Operación POP en el MIPS

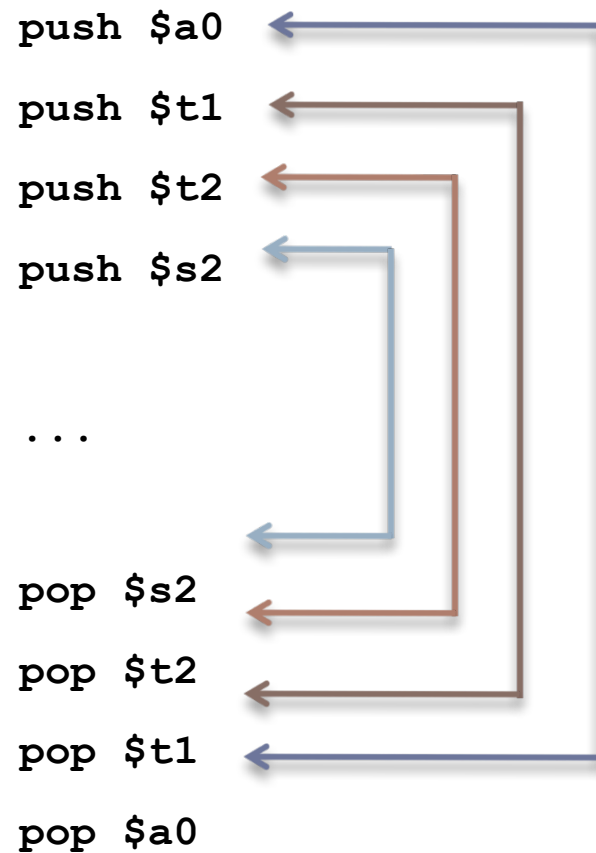


**`addu $sp, $sp, 4`**

Se actualiza el registro puntero de pila para apuntar a la nueva cima de la pila. El dato desapilado (9) sigue estando en memoria pero será sobrescrito en futuras operaciones PUSH

# 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
```

```
addu $sp $sp -4  
sw   $a0 ($sp)  
addu $sp $sp -4  
sw   $t1 ($sp)  
addu $sp $sp -4  
sw   $t2 ($sp)  
addu $sp $sp -4  
sw   $s2 ($sp)
```

...

```
lw $s2 ($sp)  
addu $sp $sp 4  
lw $t2 ($sp)  
addu $sp $sp 4  
lw $t1 ($sp)  
addu $sp $sp 4  
lw $a0 ($sp)  
addu $sp $sp 4
```

# Pila

## uso de push y pop consecutivos

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

```
addu $sp $sp -16  
sw   $a0 12($sp)  
sw   $t1 8($sp)  
sw   $t2 4($sp)  
sw   $s2 ($sp)
```

...

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


...

```
lw   $s2 ($sp)  
lw   $t2 4($sp)  
lw   $t1 8($sp)  
lw   $a0 12($sp)  
addu $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;  
}
```



# Ejemplo

## (2) Pensar en el paso de parámetros

- ▶ Los **parámetros** en el MIPS se pasan en \$a0, \$a1, \$a2 y \$a3
- ▶ Los **resultados** en el MIPS se recogen en \$v0, \$v1
- ▶ Más adelante se verá con más detalle
- ▶ Si se necesita pasar más de cuatro parámetros, los cuatro primeros en los registros \$a0, \$a1, \$a2 y \$a3 y el resto en la pila
  
- ▶ En la llamada `z=factorial(5);`
- ▶ Un parámetro de entrada: en \$a0
- ▶ Un resultado en \$v0




# Ejemplo

(3) Se pasa a ensamblador cada función


El parámetro se pasa en \$a0  
El resultado se devuelve en \$v0

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



```
li $a0, 5      # argumento  
jal factorial  # llamada  
move $a0, $v0  # resultado  
li $v0, 1  
syscall        # llamada para  
               # imprimir un int  
...
```


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



```
factorial: 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  
           b      bucle  
fin:      move   $v0, $s1  #resultado  
           jr     $ra
```

# Ejemplo

## (4) Se analizan los registros que se modifican

<pre>int factorial(int x) {     int i;     int r=1;     for (i=1;i&lt;=x;i++) {         r*=i;     }     return r; }</pre>		<pre>factorial: 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            b      bucle fin:      move   \$v0, \$s1   #resultado            jr     \$ra</pre>
---	--	--

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: addu $sp, $sp, -8  
           sw   $s0, 4($sp)  
           sw   $s1, ($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  
           b     bucle  
fin:      move  $v0, $s1   #resultado  
           lw   $s1, ($sp)  
           lw   $s0, 4($sp)  
           addu $sp, $sp, 8  
           jr   $ra
```

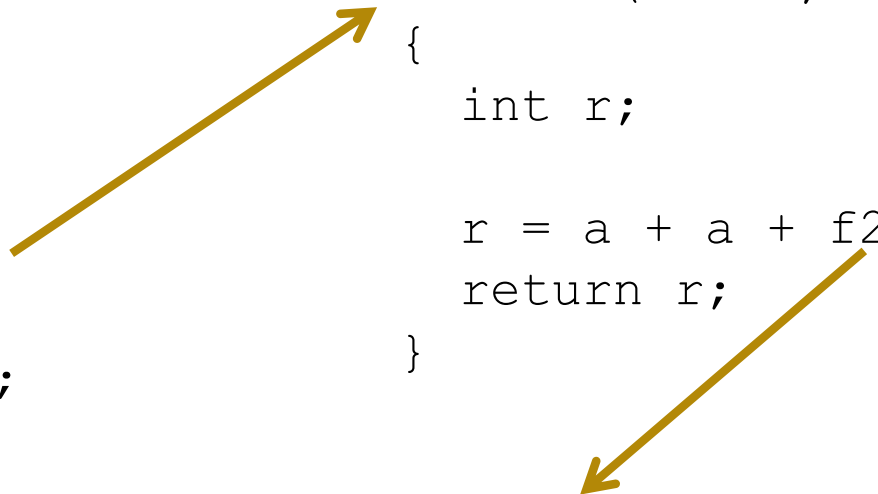
No es necesario guardar \$ra. La rutina factorial es terminal

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 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;  
}
```

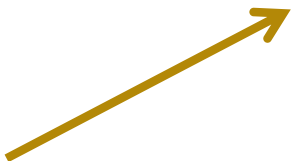


## Ejemplo 2. Invocación

```
int main()
{
    int z;

    z=f1(5, 2);

    print_int(z);
}
```




li	\$a0,	5	# primer argumento
li	\$a1,	2	# segundo argumento
jal	f1		# llamada
move	\$a0,	\$v0	# resultado
li	\$v0,	1	
syscall			# llamada para
			# imprimir un int

Los parámetros se pasan en \$a0 y \$a1  
El resultado se devuelve en \$v0

## 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

      move  $a0, $a1
      jal   f2
      add   $v0, $s0, $v0

      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

      move  $a0, $a1
      jal   f2
      add   $v0, $s0, $v0

      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 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: addu    $sp, $sp, -8
    sw     $s0, 4($sp)
    sw     $ra, ($sp)

    add    $s0, $a0, $a0

    move   $a0, $a1
    jal    f2
    add    $v0, $s0, $v0

    lw     $ra, ($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 $t0, $t0, $a0
     jr  $ra
```

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

# Convención en el uso de los registros en el MIPS

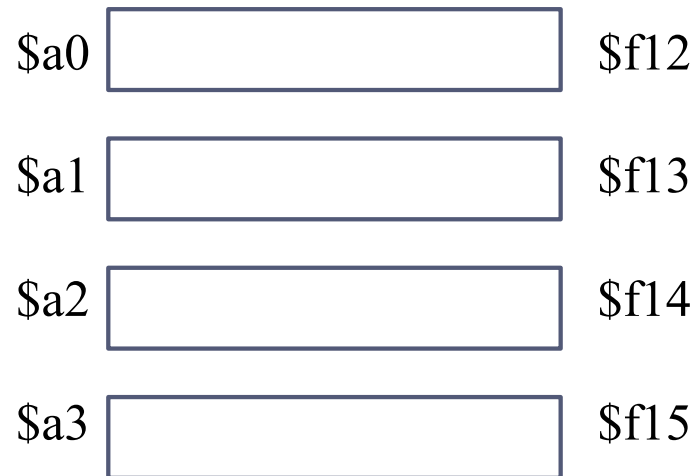
<b>Registro</b>	<b>Uso</b>	<b>Preservar el valor</b>
\$v0-\$v1	Resultados	No
\$a0..\$a3	Argumentos	No
\$t0..\$t9	Temporales	No
\$s0..\$s7	Temporales a Salvar	Si
\$sp	Puntero de pila	Si
\$fp	Puntero marco de pila	Si
\$ra	Dirección de retorno	Si

# Convención en el uso de los registros de coma flotante en el MIPS

<b>Registro</b>	<b>Uso</b>	<b>Preservar el valor</b>
\$f0-\$f3	Resultados	No
\$f4..\$f11	Temporales	No
\$f12-\$f15	Argumentos	No
\$f16-\$f19	Temporales	No
\$f20-\$f31	Temporales a salvar	Si

# Paso de argumentos detallado en MIPS

- ▶ Se utilizan los registros  $\$a_i$  y  $\$f_i$ , pero no se pueden pasar más de 16 bytes en registros teniendo en cuenta  $\$a$  y  $\$f$



- ▶ Si el primer argumento es entero:
  - ▶ Todos los argumentos se pasan en  $\$a0 \dots \$a3$  (aunque el resto sean de tipo float o double)
  - ▶ El resto en la pila

# Paso de argumentos detallado en MIPS

- ▶ Si el primer argumento es float o double
  - ▶ Se pasa en \$f12 (float) y \$f14-\$f15 (double)
  - ▶ Para el resto se utiliza \$a1 o \$f1 hasta llegar a 16 bytes, el resto en la pila



# Ejemplos de llamadas

Argumentos de la función	Asignación en registros y pila
d1, d2	\$f12-\$f13, \$f14-\$f15
s1, d2	\$f12, \$f14-\$f15
s1, s2	\$f12, \$f13
n1, n2, n3, n4	\$a0, \$a1, \$a2, \$a3
d1, n1, d2	\$f12-\$f13, \$a2, pila
d1, n1, n2	\$f12-\$f13, \$a2, \$a3
n1, n2, n3, d1	\$a0, \$a1, \$a2, pila
n1, n2, n3, s1	\$a0, \$a1, \$a2, \$a3
n1, n2, d1	\$a0, \$a1, (\$a2, \$a3)
d1, s1, s2	\$f12-\$f13, \$f14, \$a3

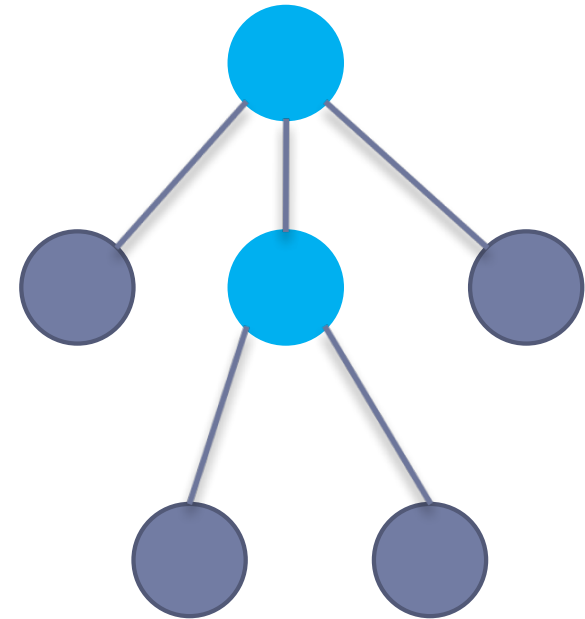
n=int, d=double, s=float

# Retorno de resultados en MIPS

- ▶ Se usa \$v0 y \$v1 para valores de tipo entero
- ▶ Se usa \$f0 para valores de tipo float
- ▶ Se usa \$f0-\$f1 para valores de tipo double
- ▶ En caso de estructuras o valores complejos han de dejarse en pila. El espacio lo reserva la función que realiza la llamada

# 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
- ▶ Su manipulación se hace a través de dos registros:
  - ▶  $\$sp$ : puntero de pila, que apunta siempre a la cima de la pila
  - ▶  $\$fp$ : puntero de marco de pila, que marca la zona de la pila que pertenece al procedimiento llamado
- ▶ El **registro marco de pila** ( $\$fp$ ) se utiliza en el procedimiento llamado para:
  - ▶ Acceder a los parámetros que se pasan en la pila
  - ▶ Acceder a las variables locales del procedimiento

# Marco de pila

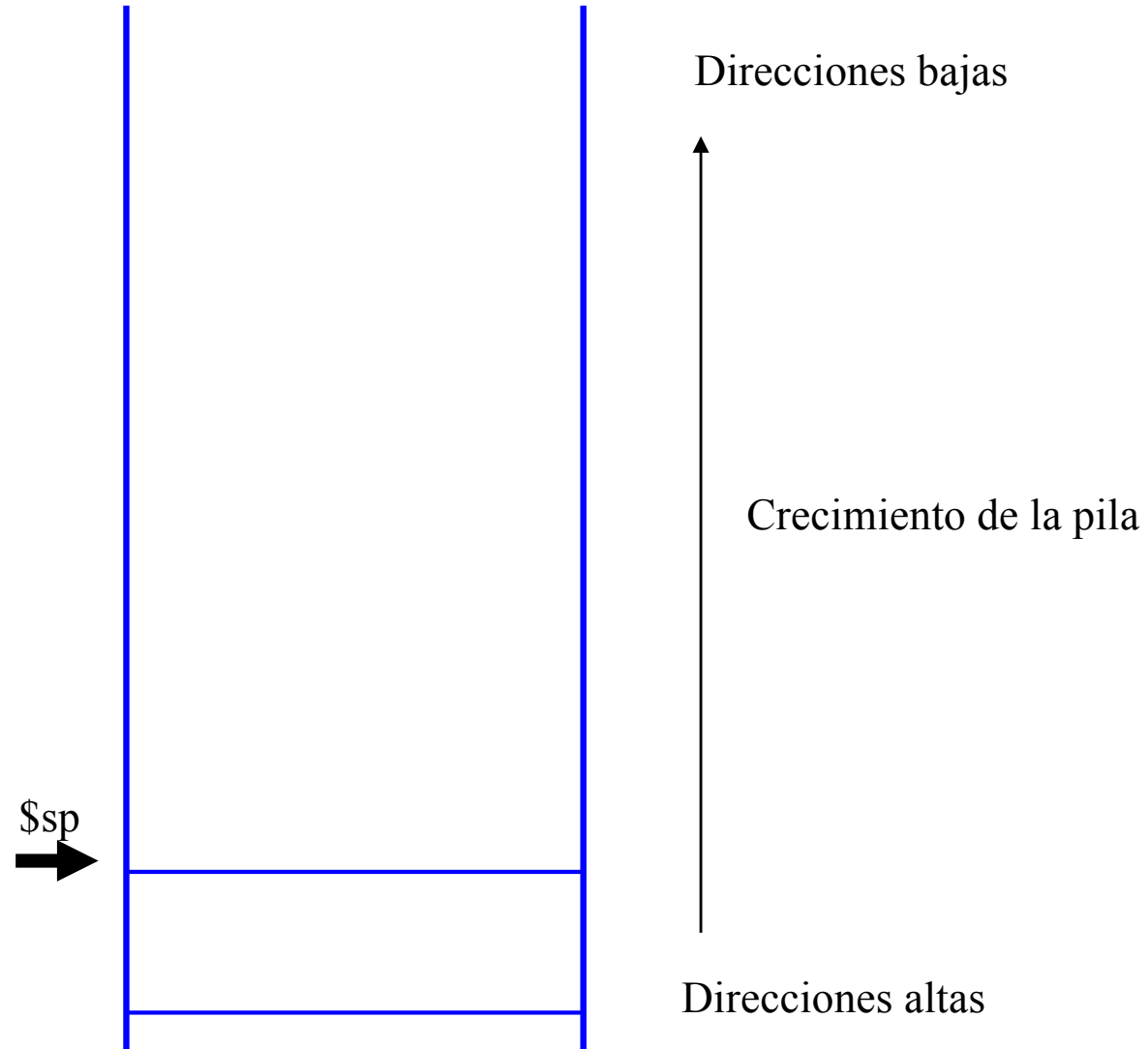
- ▶ El marco de pila almacena:
  - ▶ Los parámetros introducidos por el procedimiento llamante en caso de ser necesarios
  - ▶ El registro marco de pila del procedimiento llamante (antiguo  $\$fp$ )
  - ▶ Los registros guardados por la función (incluyen al registro  $\$ra$  en caso de procedimientos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

# Procedimiento general de llamadas a subrutinas

Subrutina llamante	Subrutina llamada
Salvaguarda de registros que no quiera que modifique la subrutina llamada (\$t, \$a, ..)	
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, \$fp, \$s)
	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

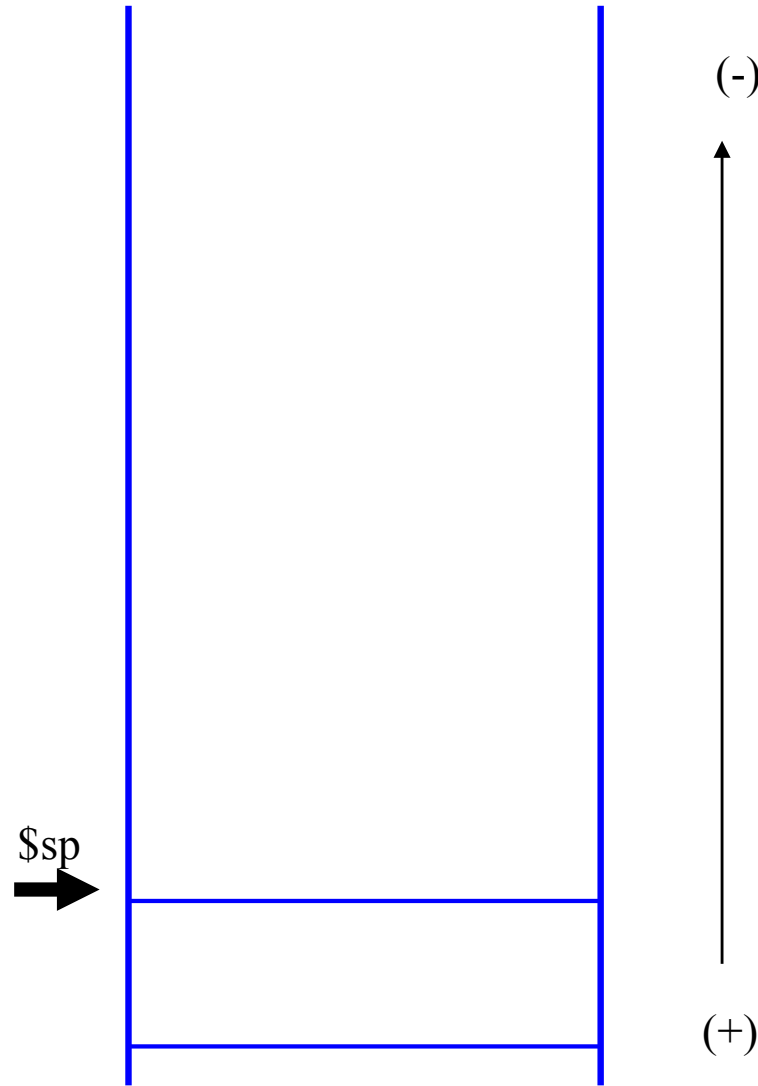
No se va a seguir el estrictamente el convenio del MIPS por simplicidad



# 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



# Construcción del marco de pila subrutina llamante

## Salvaguardar Registros

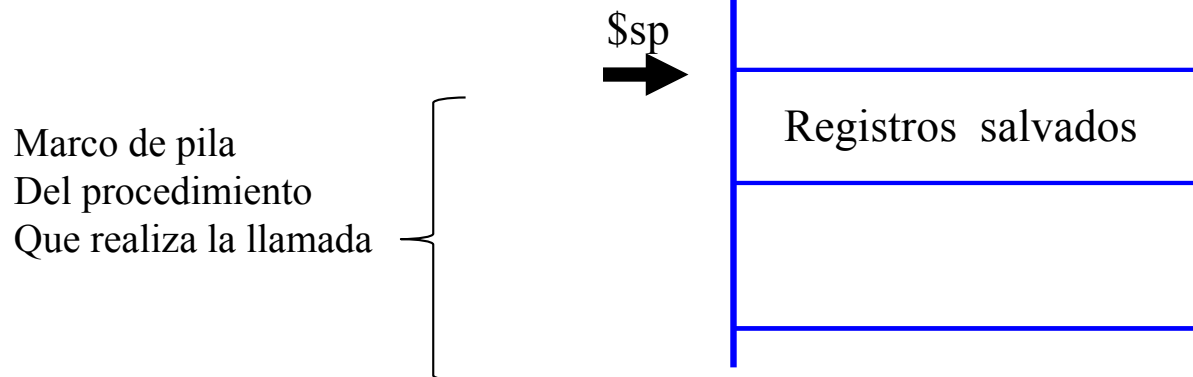
Una subrutina puede modificar cualquier registro **\$a0..\$a3** y **\$t0..\$t9**

## Ejemplo:

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

```
move $s2, $t0
```

¿Qué valor tiene \$t0 y \$t1?



# Construcción del marco de pila subrutina llamante

## Salvaguardar Registros

Una subrutina puede modificar cualquier registro **\$a0..\$a3** y **\$t0..\$t9**

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 funcion

move \$s2, \$t0

¿Qué valor tiene \$t0 y  
t1?

# Construcción del marco de pila subrutina llamante

## Salvaguardar Registros

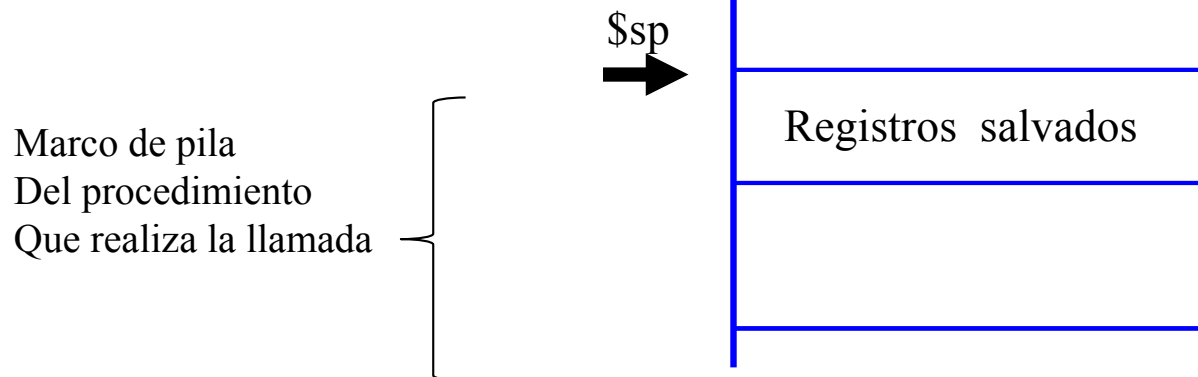
Una subrutina puede modificar cualquier registro **\$a0..\$a3** y **\$t0..\$t9**

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

## Ejemplo:

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

li   $a0, 5
jal  funcion
```





# Construcción del marco de pila subrutina llamante

## Salvaguardar Registros

Una subrutina puede modificar cualquier registro **\$a0..\$a3** y **\$t0..\$t9**

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:

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

```
li   $a0, 5  
jal  funcion
```

```
lw  $t0 ($sp)  
lw  $t1 4($sp)  
addu $sp $sp 8
```

# Construcción del marco de pila subrutina llamante

## Paso de parámetros:

Antes de realizar la llamada el  
procedimiento llamante:  
Deja los parámetros en **\$a1 (\$f)**  
El resto de parámetros en la pila

## Ejemplo (6 parámetros):

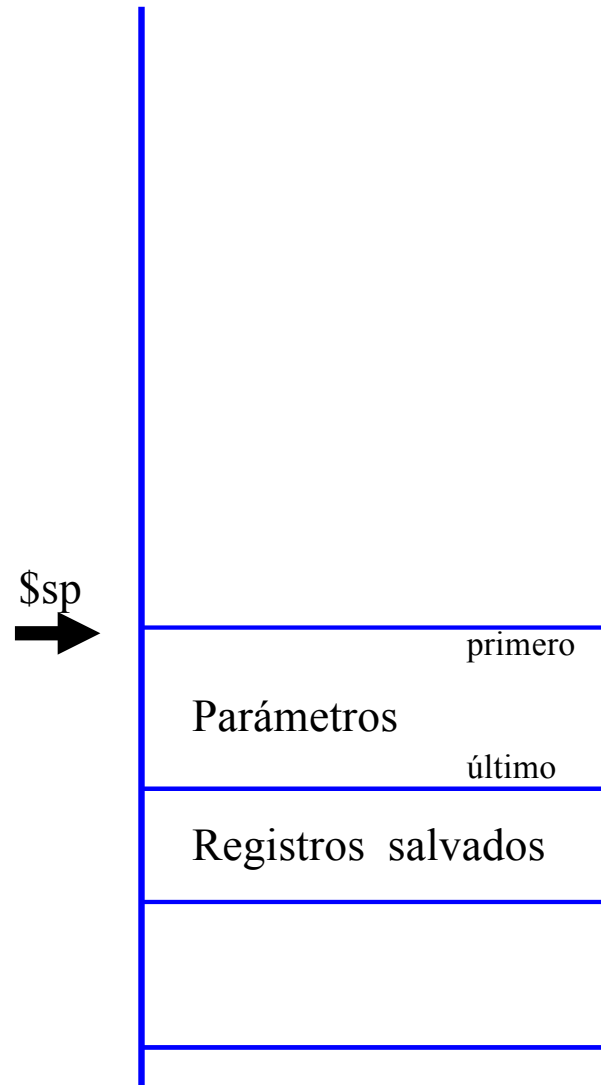
```
li    $a0, 1
li    $a1, 2
li    $a3, 3
li    $a4, 4

subu  $sp, $sp, 8

li    $t0, 5
sw    $t0, 4($sp)

li    $t0, 6
sw    $t0, ($sp)
```

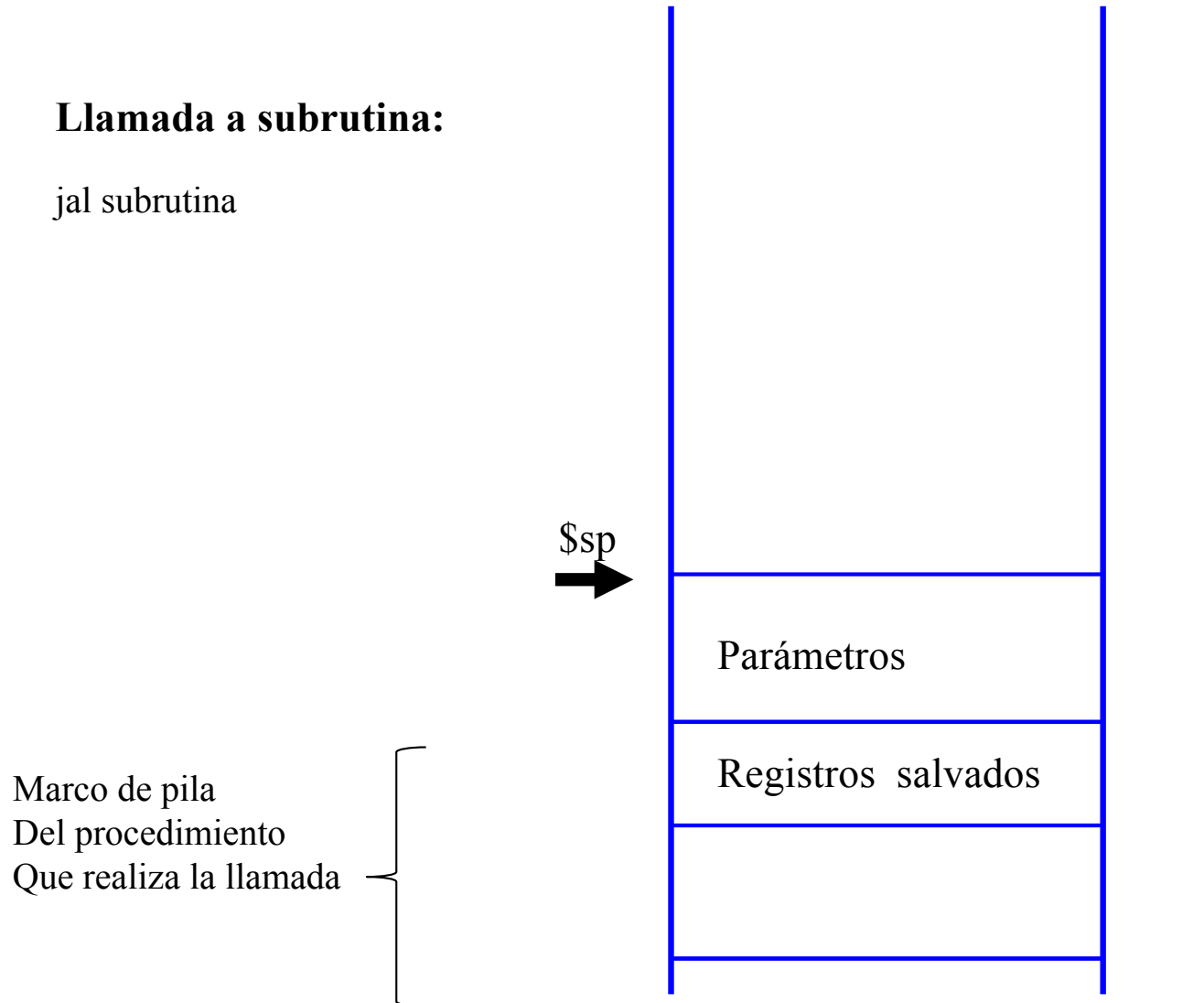
Marco de pila  
Del procedimiento  
Que realiza la llamada



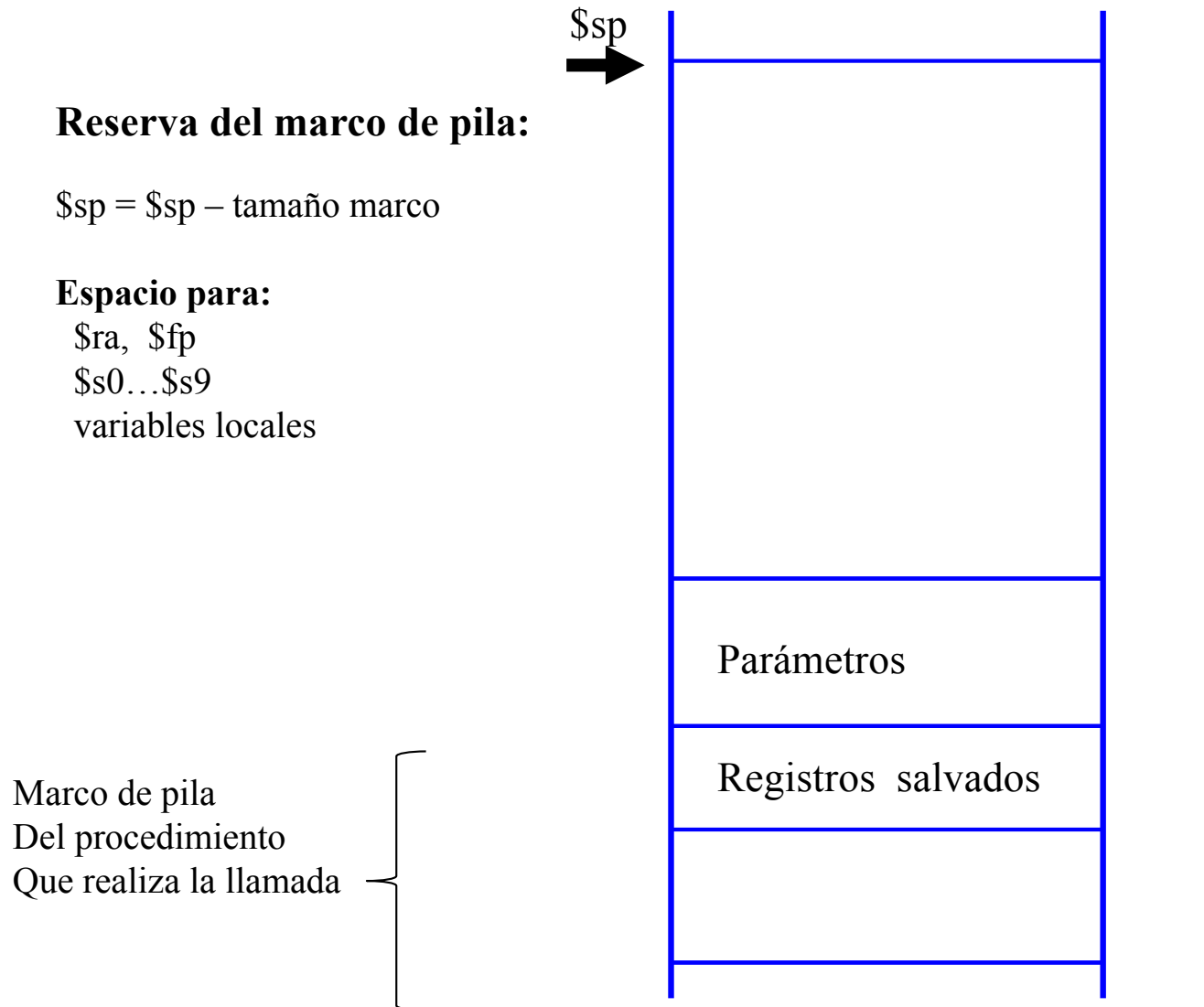
# Construcción del marco de pila subrutina llamante

## Llamada a subrutina:

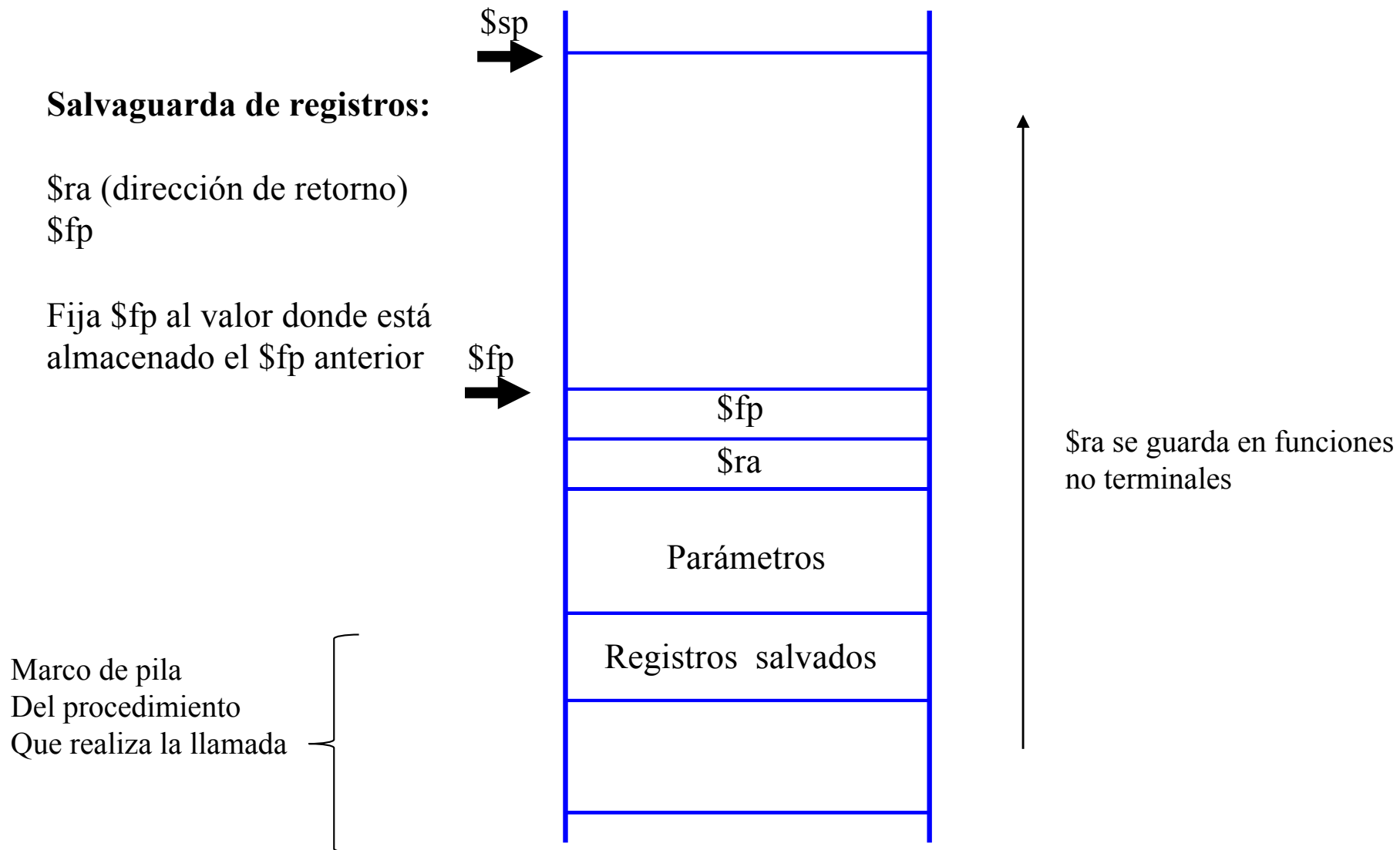
jal subrutina



# Construcción del marco de pila subrutina llamada



# Construcción del marco de pila subrutina llamada

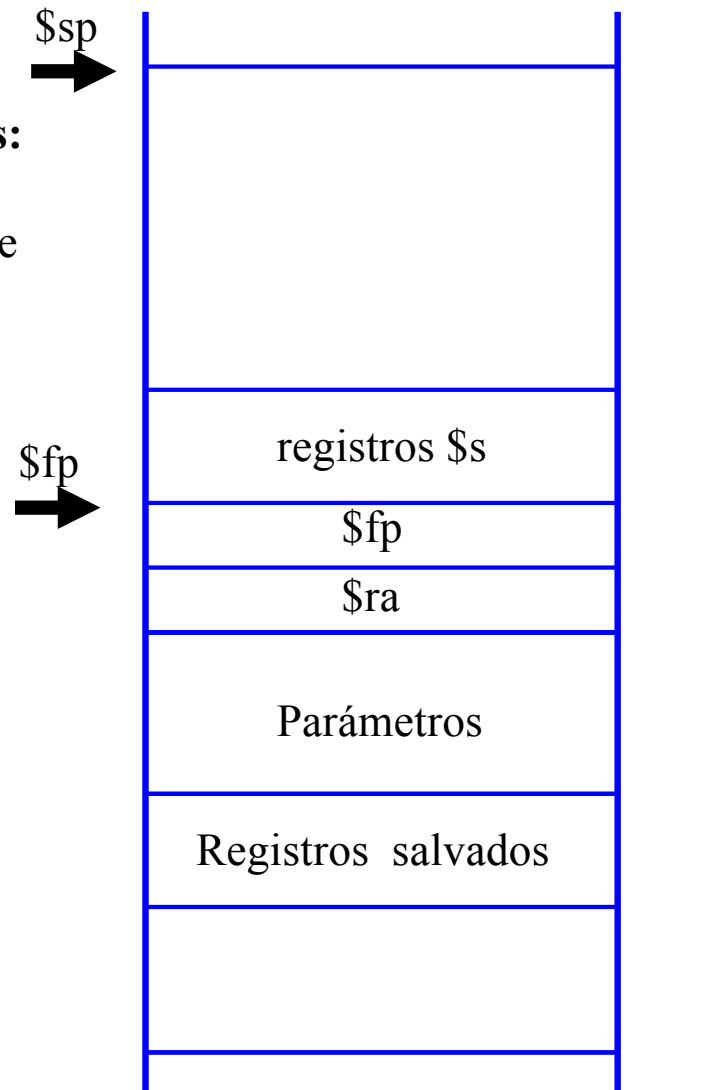


# Construcción del marco de pila subrutina llamada

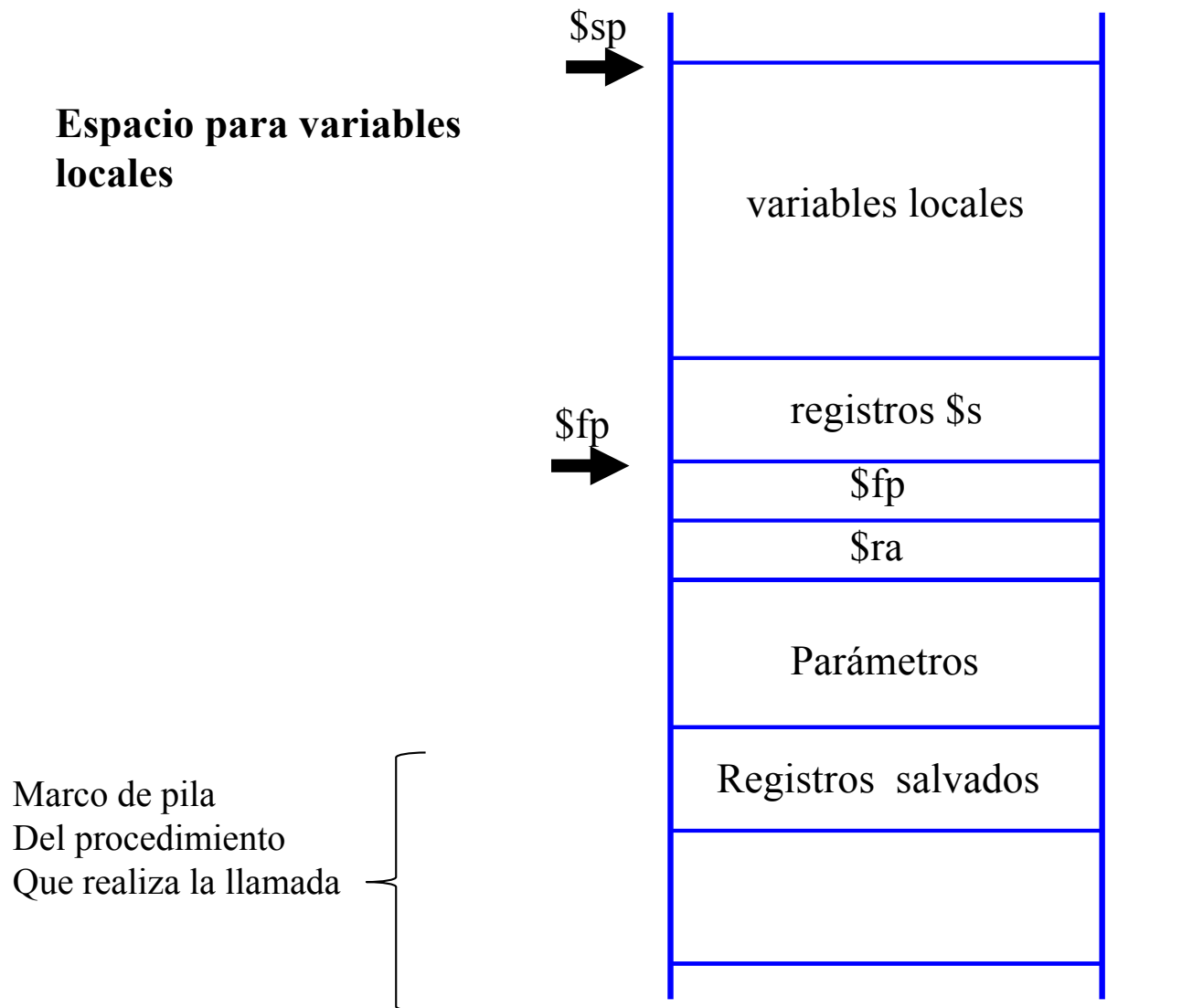
## Salvaguarda de registros \$s:

Se guarda los registros \$s que se vayan a modificar  
Una función no puede por convenio modificar los registros \$s (sí lo \$t y los \$a)

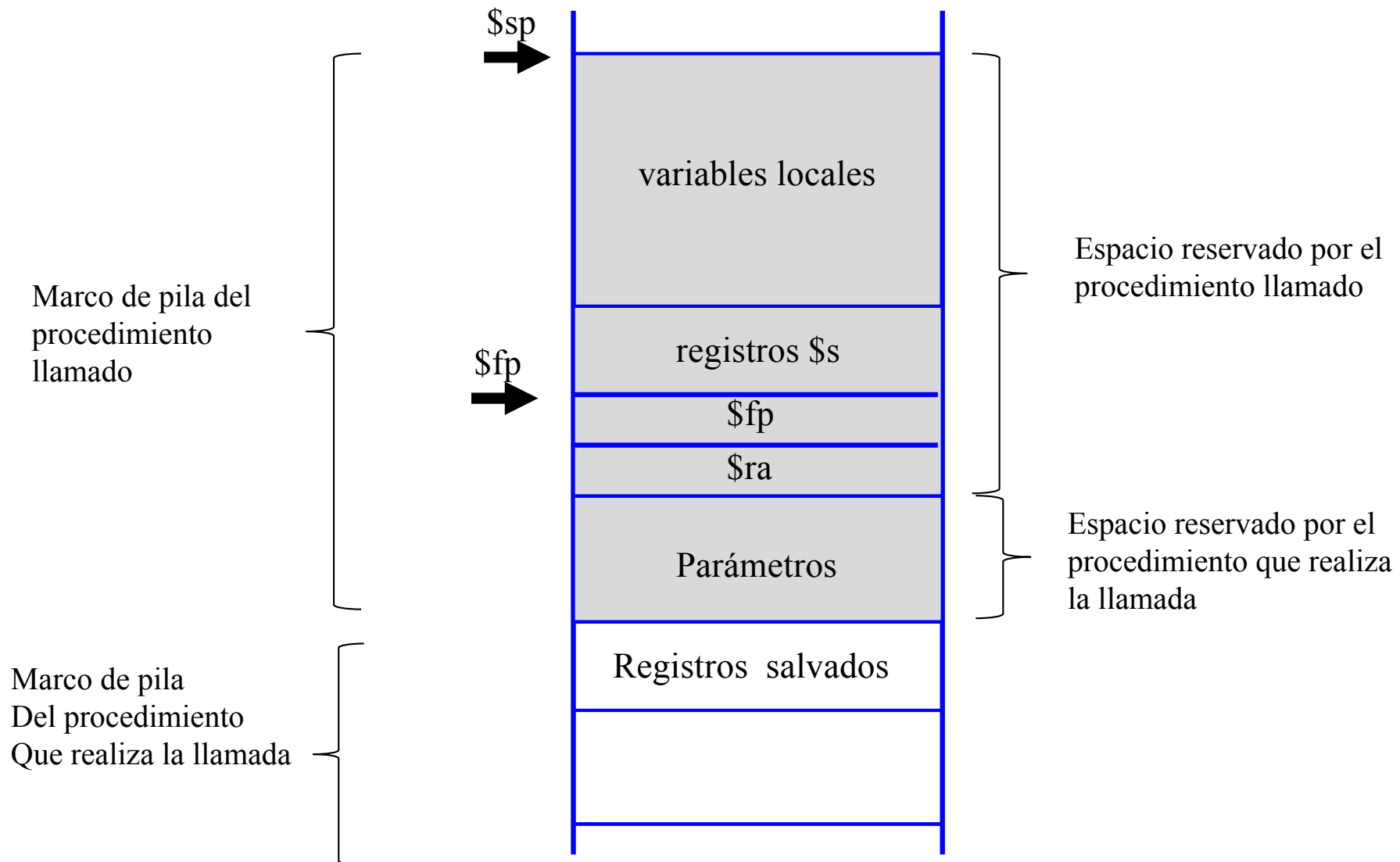
Marco de pila  
Del procedimiento  
Que realiza la llamada



# Construcción del marco de pila subrutina llamada

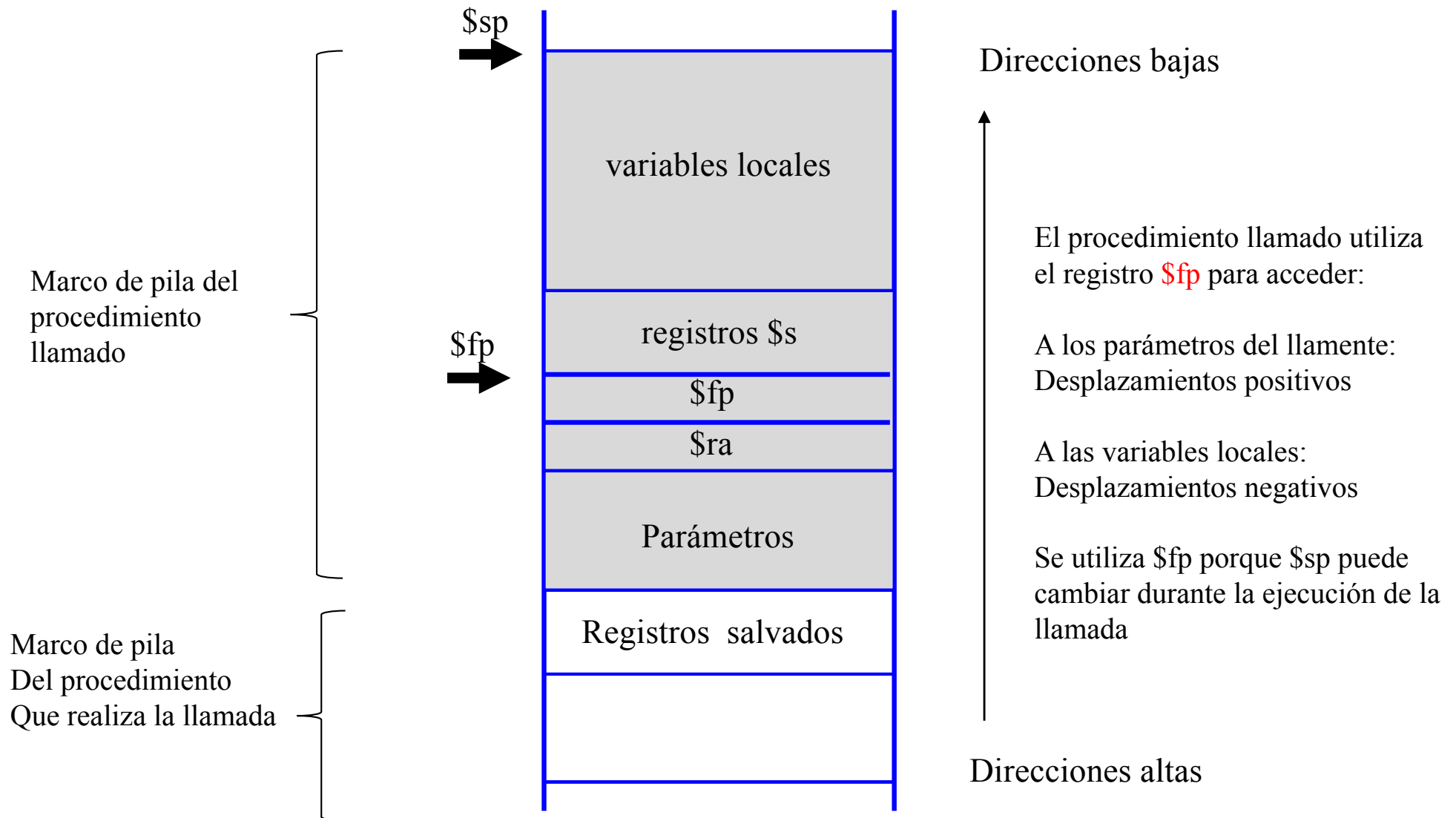


# Construcción del marco de pila





# Marco de pila

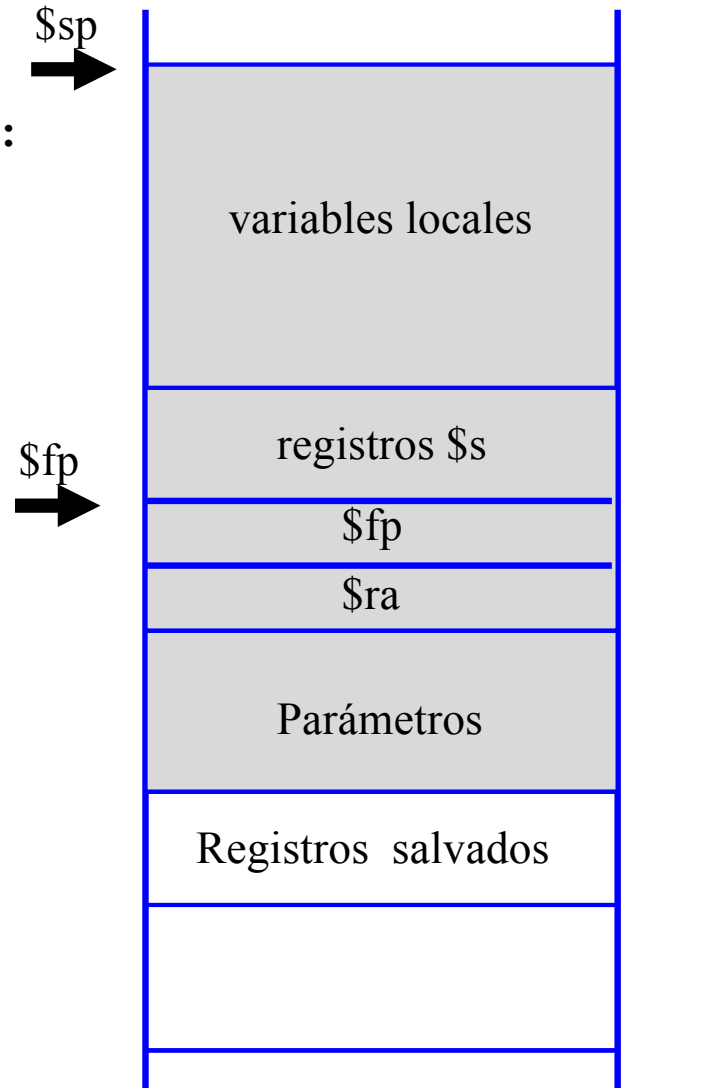


# Finalización de la subrutina subrutina llamada

**Se devuelven los resultados:**

$\$v0$ ,  $\$v1$ , ( $\$f0$ )

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



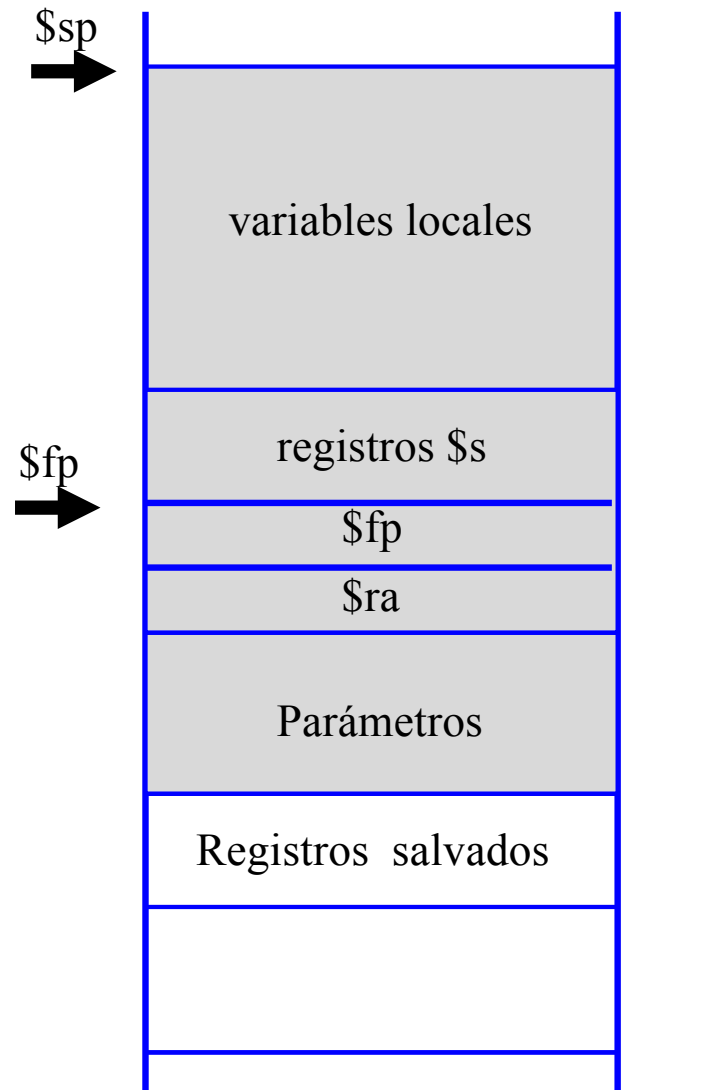
Marco de pila  
Del procedimiento  
Que realiza la llamada

# Finalización de la subrutina subrutina llamada

**Se restauran los registros salvados:**

registros \$s  
registro \$fp  
registro \$ra

Marco de pila  
Del procedimiento  
Que realiza la llamada

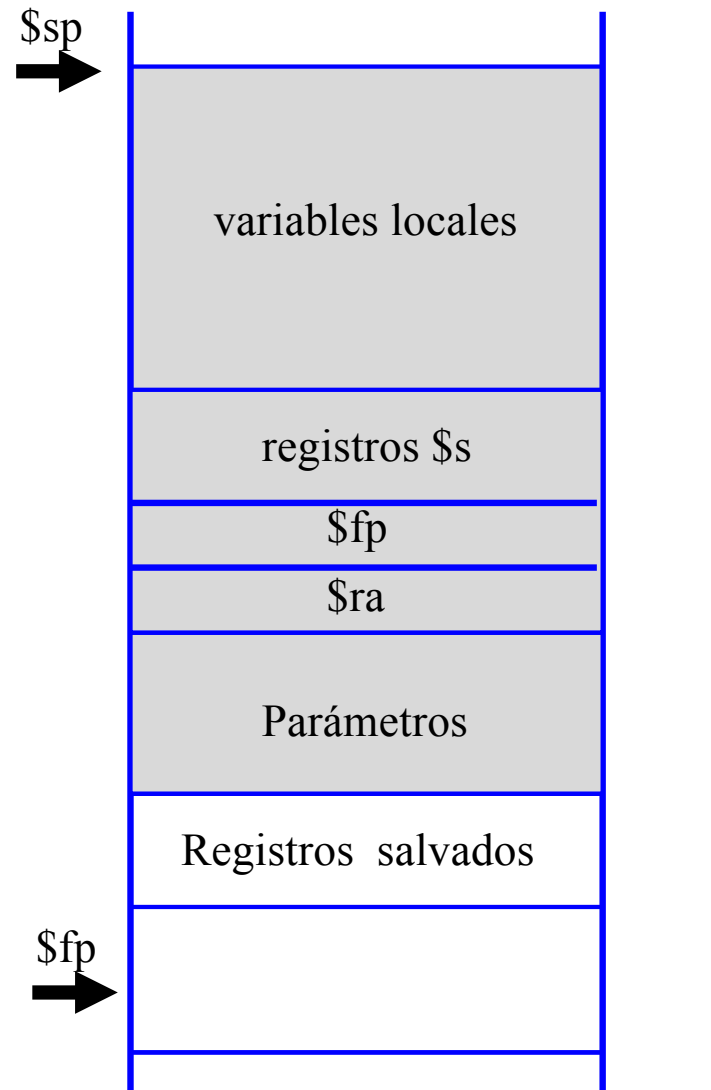


# Finalización de la subrutina subrutina llamada

**Se restauran los registros salvados:**

registros \$s  
registro \$fp  
registro \$ra

Marco de pila  
Del procedimiento  
Que realiza la llamada

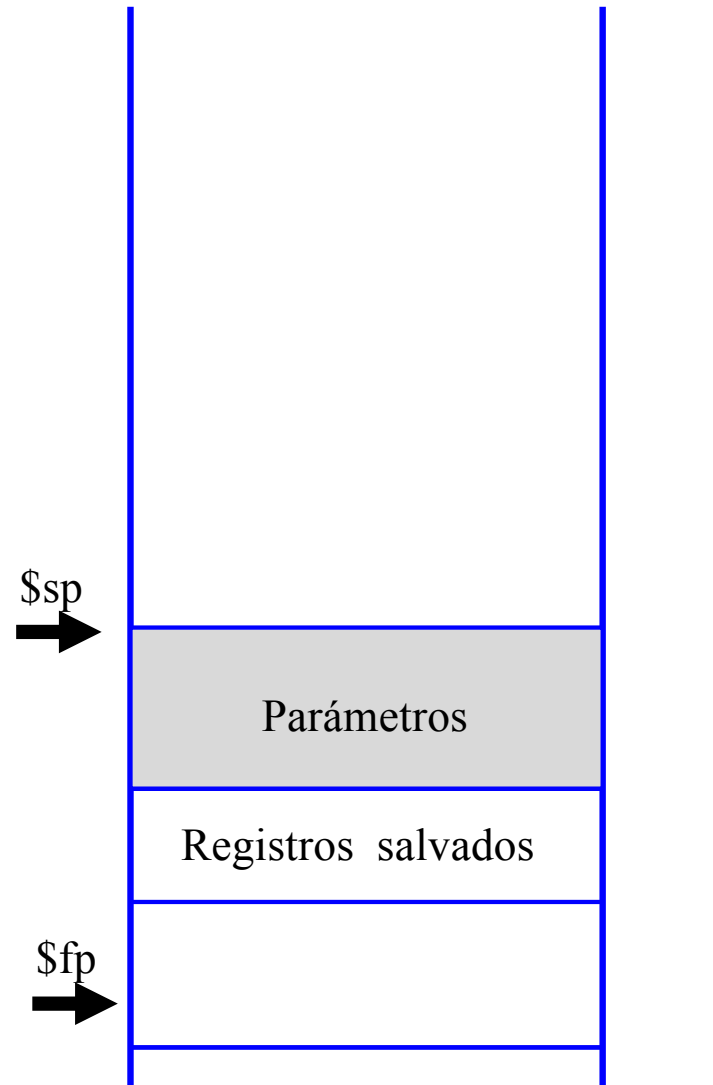


# Finalización de la subrutina subrutina llamada

**Se libera el espacio del  
marco:**

$$\text{\$sp} = \text{\$sp} + \text{tamaño marco}$$

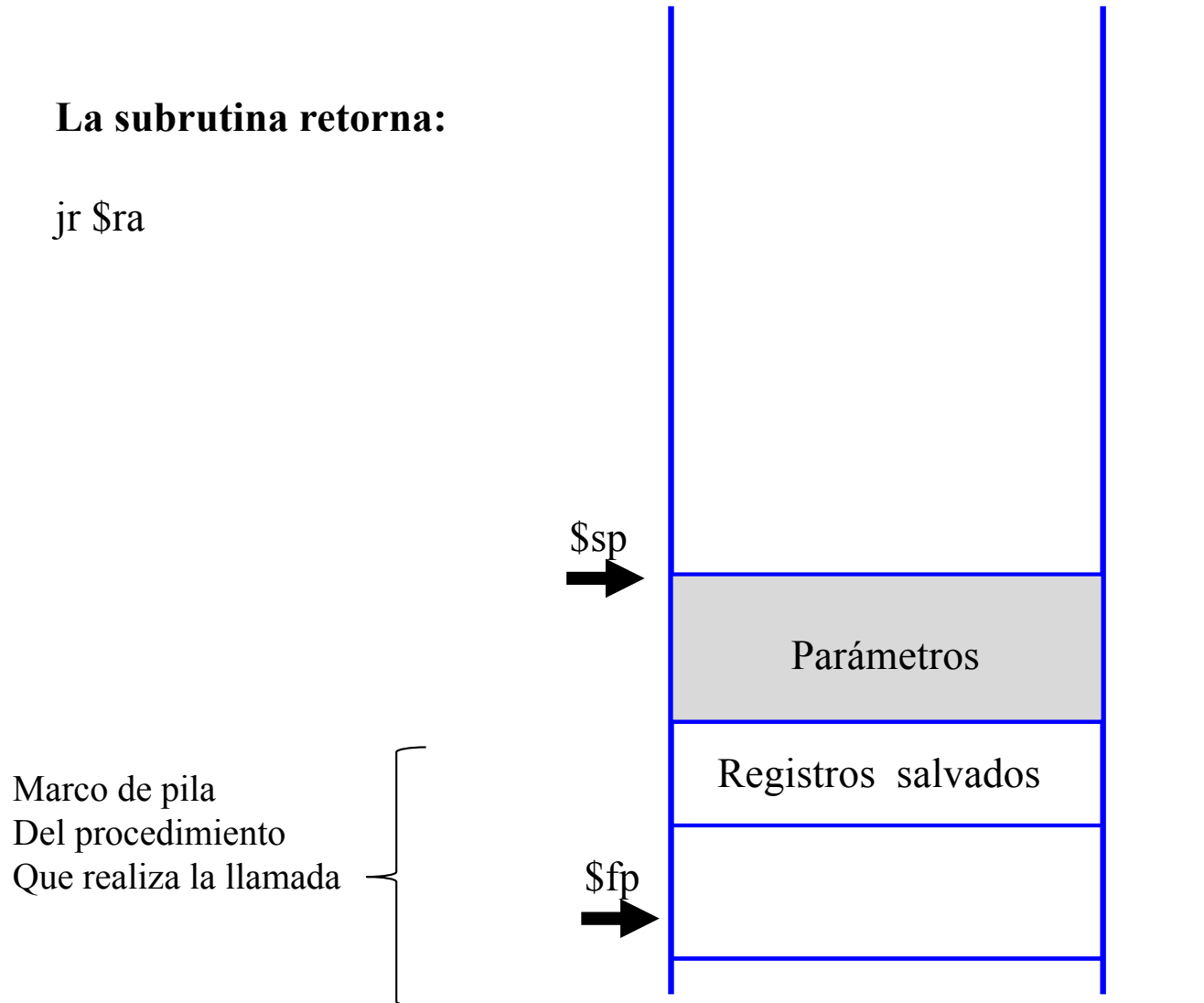
Marco de pila  
Del procedimiento  
Que realiza la llamada



# Finalización de la subrutina subrutina llamada

**La subrutina retorna:**

jr \$ra

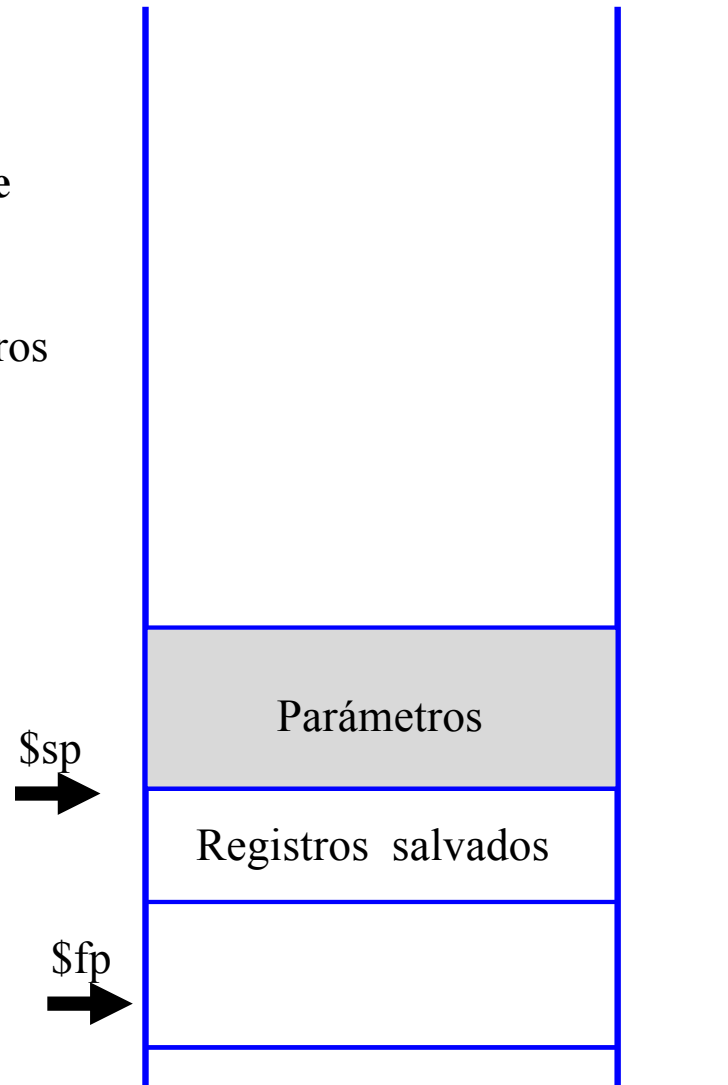


# Finalización de la subrutina subrutina llamante

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

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

Marco de pila  
Del procedimiento  
Que realiza la llamada

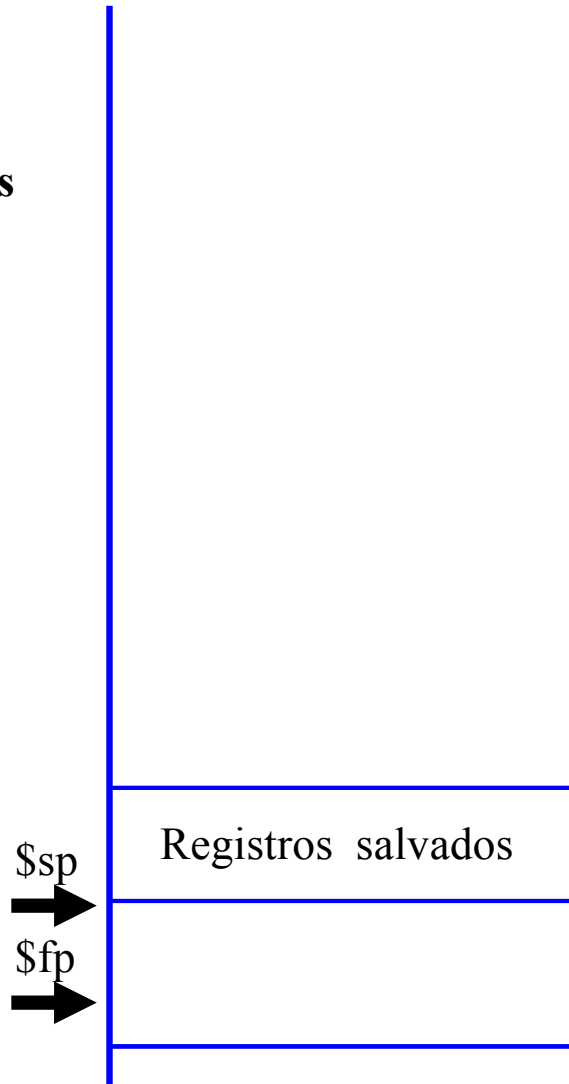


# Finalización de la subrutina subrutina llamante

La rutina que realizó la  
llamada restaura los registros  
que salvó

Restaura \$sp

Marco de pila  
Del procedimiento  
Que realiza la llamada



Ejemplo:

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

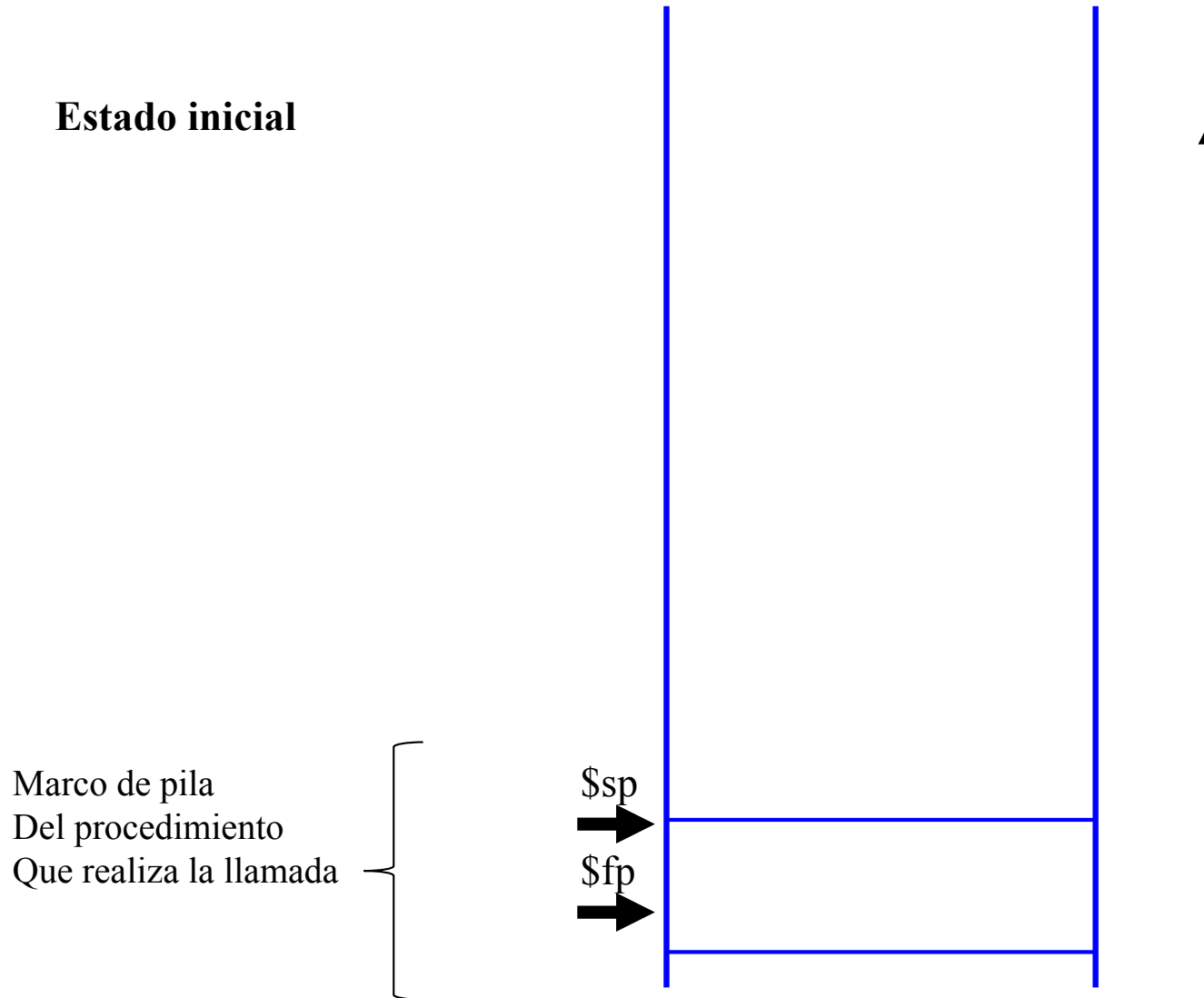
```
li    $a0, 5  
jal   funcion
```

```
lw   $t0 ($sp)  
lw   $t1 4($sp)  
addu $sp $sp 8
```



# Estado después de finalizar la llamada

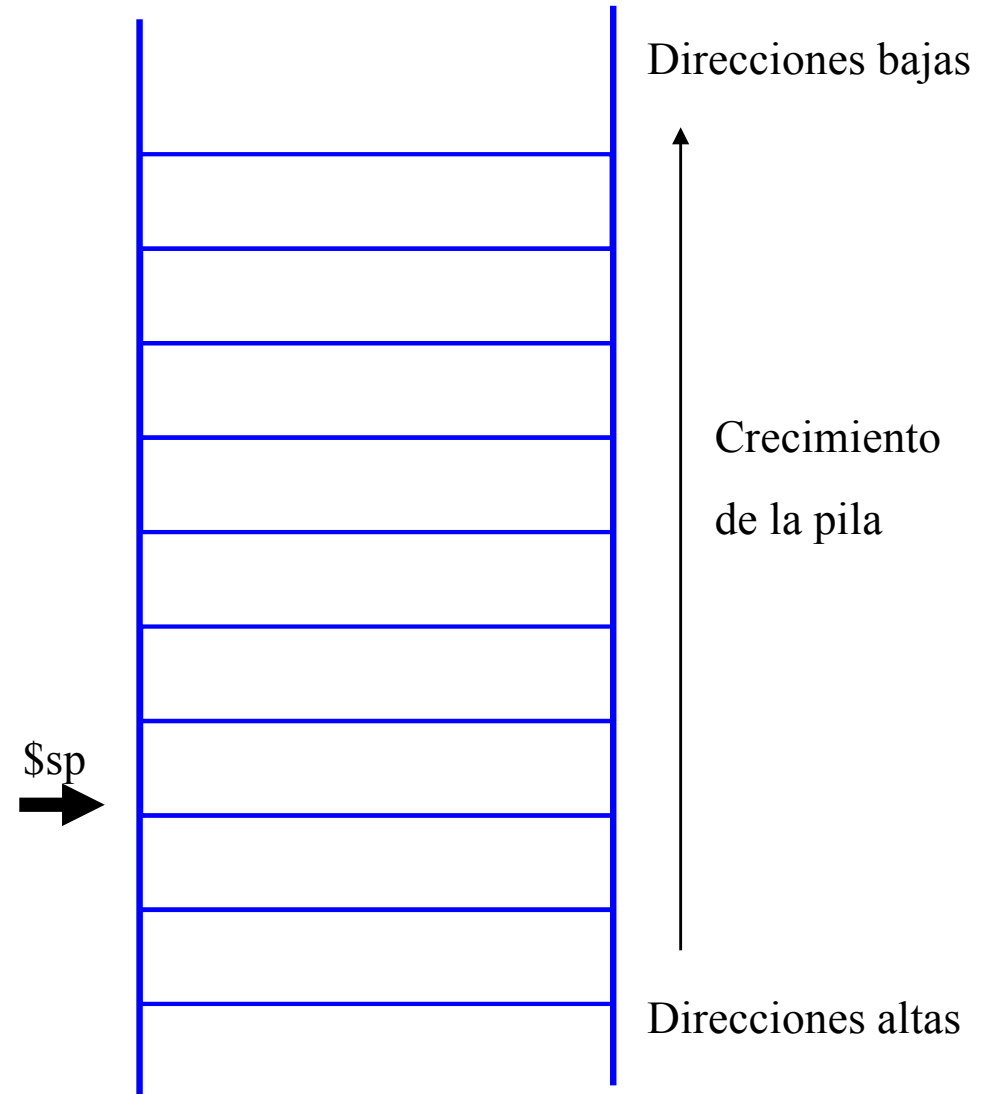
**Estado inicial**



# Acceso a parámetros y variables locales usando el marco de pila

```
int f (int n1, n2, n3,  
      n4, n5, n6)  
{  
    int v[4];  
    int k;  
  
    for (k= 0; k <3; k++)  
        v[i] = n1+n2+n3+n4+n5+n6;  
  
    return (v[1]);  
}
```

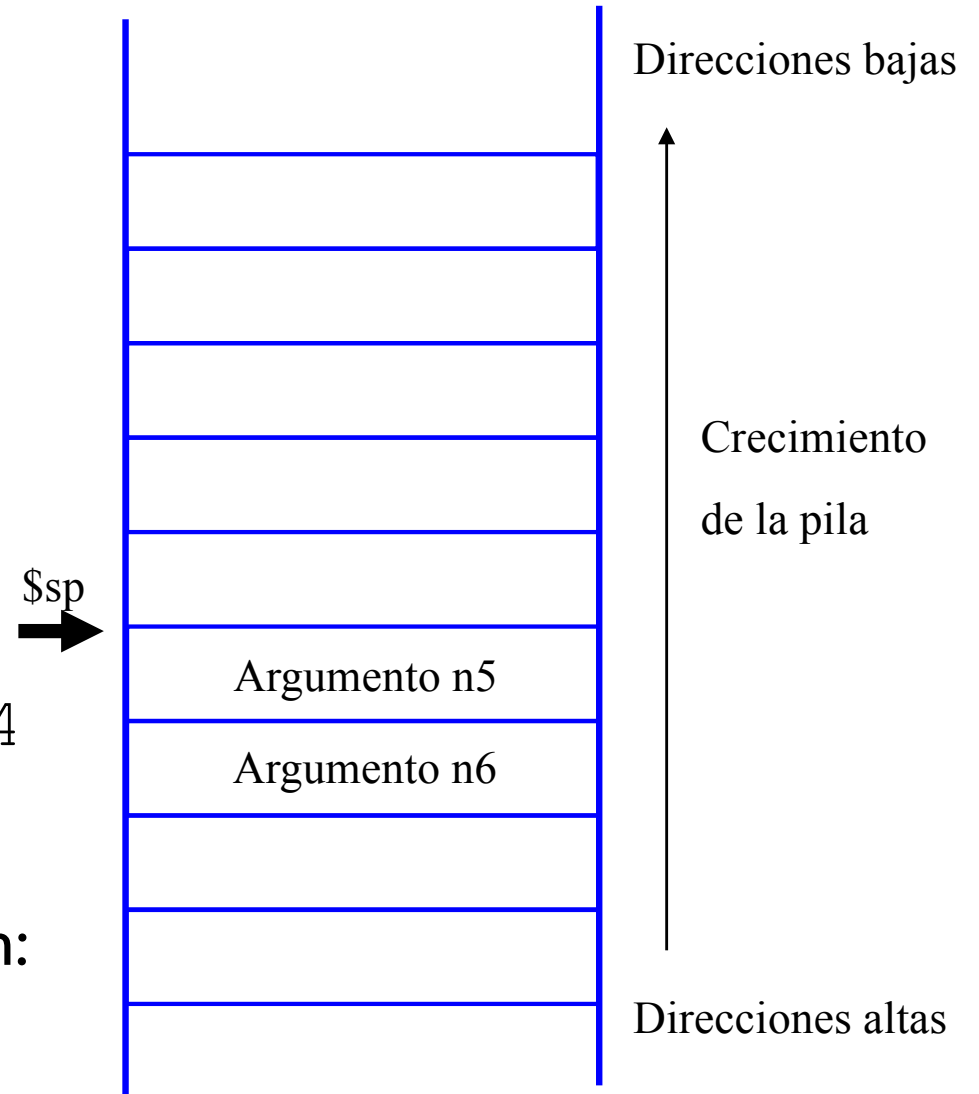
Si se realiza una llamada a f( )



# Acceso a parámetros y variables locales usando el marco de pila

```
int f (int n1, n2, n3,  
      n4, n5, n6)  
{  
    int v[4];  
    int k;  
  
    for (k= 0; k <3; k++)  
        v[i] = n1+n2+n3+n4+n5+n6;  
  
    return (v[1]);  
}
```

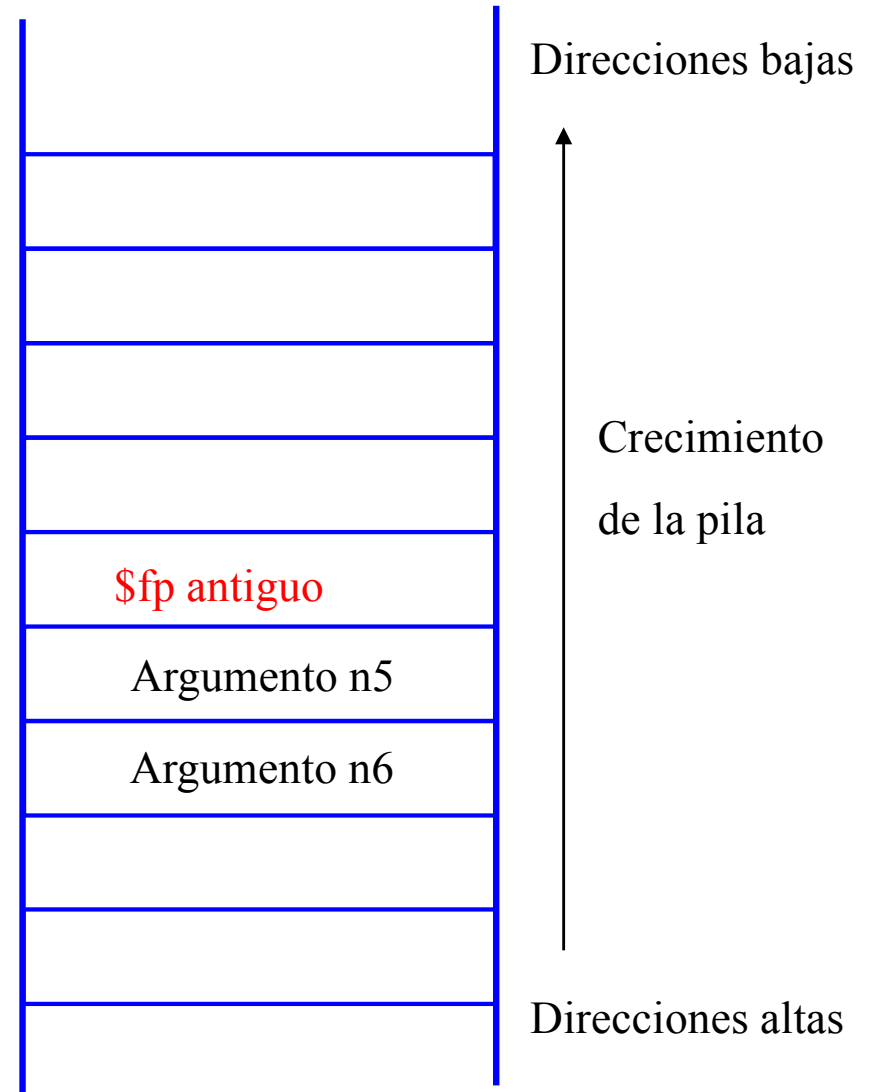
- ▶ Los parámetros  $n1, n2, n3$  y  $n4$  se pasan:
  - ▶ En  $\$a0, \$a1, \$a2, \$a3$
- ▶ Los parámetros  $n5, n6$  se pasan:
  - ▶ En la pila



# Acceso a parámetros y variables locales usando el marco de pila

```
int f (int n1, n2, n3,  
      n4, n5, n6)  
{  
    int v[4];  
    int k;  
  
    for (k= 0; k <3; k++)  
        v[i] = n1+n2+n3+n4+n5+n6;  
  
    return (v[1]);  
}
```

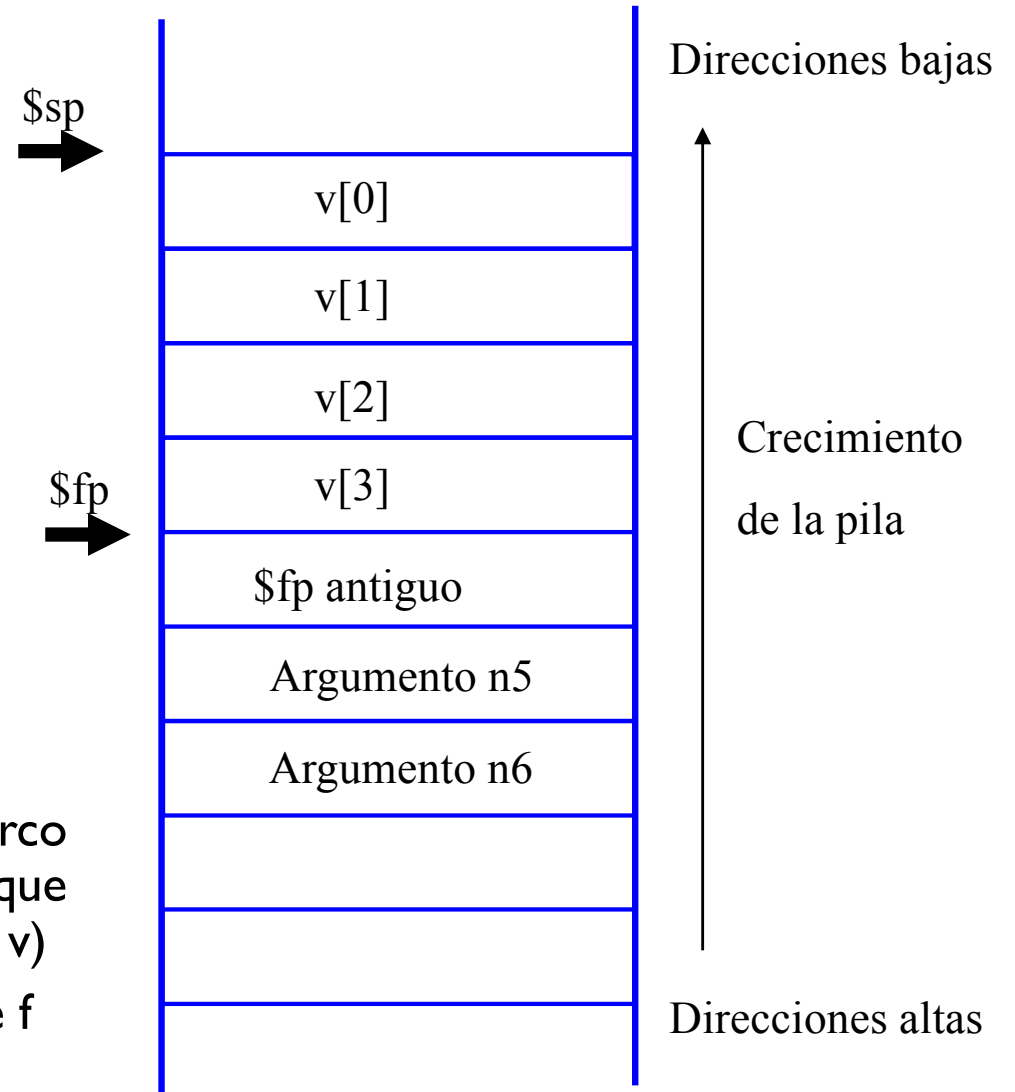
\$sp → \$fp



- ▶ Una vez invocada la función, f debe
  - ▶ Guardar una copia de \$fp
  - ▶ Guardar una copia de los registros a preservar
  - ▶ \$ra no, porque es terminal

# Acceso a parámetros y variables locales usando el marco de pila

```
int f (int n1, n2, n3,  
      n4, n5, n6)  
{  
    int v[4];  
    int k;  
  
    for (k= 0; k <3; k++)  
        v[i] = n1+n2+n3+n4+n5+n6;  
  
    return (v[1]);  
}
```

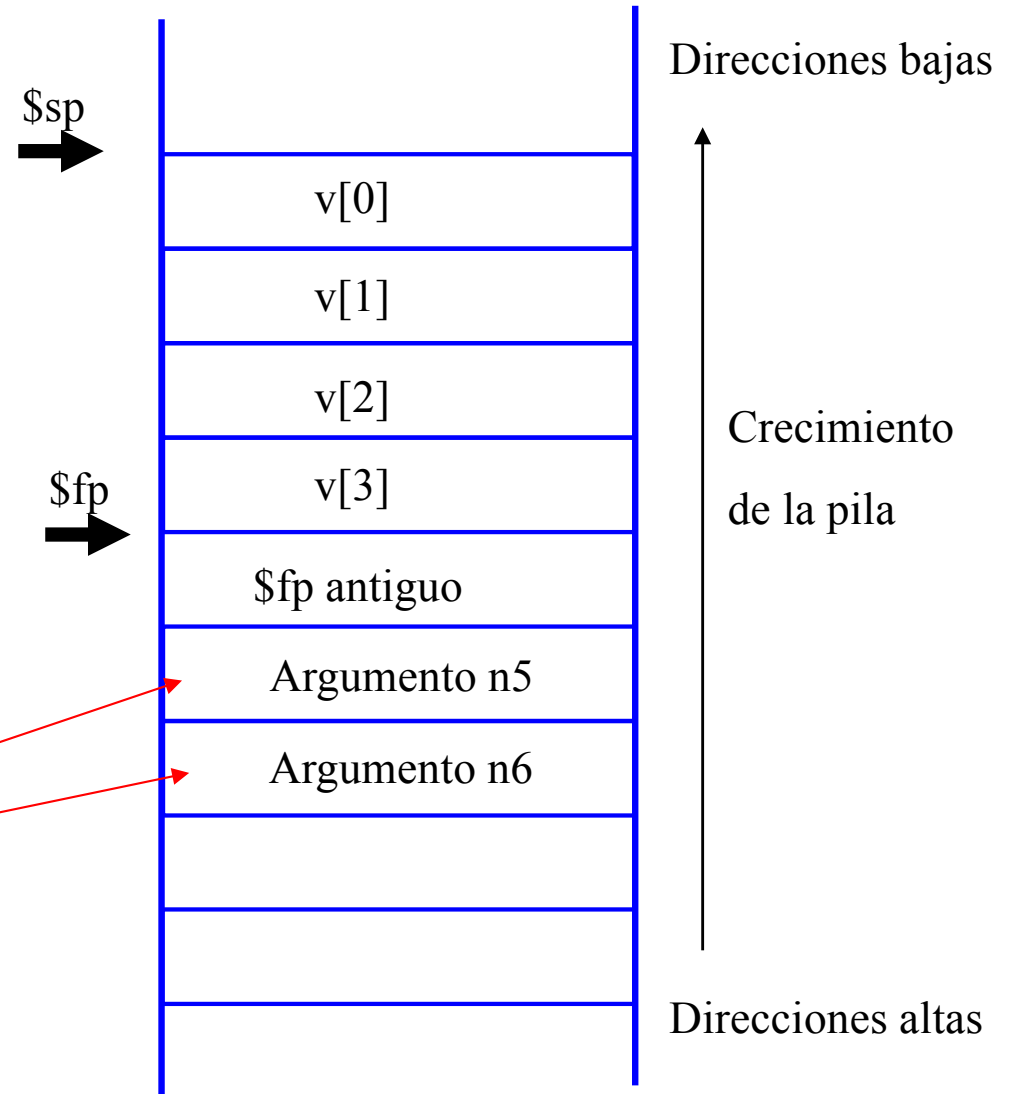


- ▶ A continuación, `f` debe **reservar** en el marco de pila **espacio** para las variables locales que no quepan en registros (en este ejemplo `v`)
- ▶ Para este ejemplo se ha considerado que `f` no modifica ningún registro

# Acceso a parámetros y variables locales usando el marco de pila

```
int f (int n1, n2, n3,  
      n4, n5, n6)  
{  
    int v[4];  
    int k;  
  
    for (k= 0; k <3; k++)  
        v[i] = n1+n2+n3+n4+n5+n6;  
  
    return (v[1]);  
}
```

- ▶ El valor de n1 está en \$a0
- ▶ El valor de n5 está en 4 (\$fp)
- ▶ El valor de n6 está en 8 (\$fp)
- ▶ El valor de v[3] está en -4 (\$fp)
- ▶ El valor de v[0] está en -16 (\$fp)



# Código para realizar la llamada a `f(int n1, n2, n3, n4, n5, n6)`

**Para la llamada:** `f (3, 4, 23, 12, 6, 7) ;`

<code>li</code>	<code>\$a0, 3</code>	}	Los cuatro primeros en registros \$ai	
<code>li</code>	<code>\$a1, 4</code>			
<code>li</code>	<code>\$a2, 23</code>			
<code>li</code>	<code>\$a3, 12</code>			
<code>addu</code>	<code>\$sp, \$sp, -8</code>	}	El resto en la pila	
<code>li</code>	<code>\$t0, 6</code>			
<code>sw</code>	<code>\$t0, (\$sp)</code>			
<code>li</code>	<code>\$t0, 7</code>			
<code>sw</code>	<code>\$t0, 4(\$sp)</code>	}		
<code>jal</code>	<code>f</code>			

# Código de la función

`f(int n1, n2, n3, n4, n5, n6)`

```
int f (int n1, n2, n3,          f:
      n4, n5, n6)
{
    int v[4];
    int k;

    for (k= 0; k <3; k++)
        v[i] = n1+n2+n3+n4+n5+n6;

    return (v[1]);
}
```



# Código de la función

`f(int n1, n2, n3, n4, n5, n6)`

```
int f (int n1, n2, n3,          f:      addu      $sp, $sp, -4
        n4, n5, n6)              sw        $fp, ($sp)
{
    int v[4];
    int k;

    for (k= 0; k <3; k++)
        v[i] = n1+n2+n3+n4+n5+n6;

    return (v[1]);
}
```

Se guarda el valor de \$fp (\$fp antiguo)

# Código de la función

`f(int n1, n2, n3, n4, n5, n6)`

```
int f (int n1, n2, n3,
      n4, n5, n6)
{
    int v[4];
    int k;

    for (k= 0; k <3; k++)
        v[i] = n1+n2+n3+n4+n5+n6;

    return (v[1]);
}
```

```
f:      addu      $sp, $sp, -4
        sw        $fp, ($sp)
        move      $fp, $sp
```

Se fija \$fp a la posición donde está guardado el actual \$fp

# Código de la función

`f(int n1, n2, n3, n4, n5, n6)`

<pre>int f (int n1, n2, n3,         n4, n5, n6) {     int v[4];     int k;      for (k= 0; k &lt;3; k++)         v[i] = n1+n2+n3+n4+n5+n6;      return (v[1]); }</pre>	<pre>f:      addu      \$sp, \$sp, -4         sw        \$fp, (\$sp)         move      \$fp, \$sp         addu      \$sp, \$sp, -16</pre>
--	---

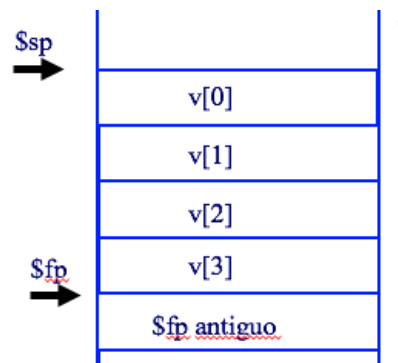
Se deja hueco para el vector v (16 bytes  
para 4 elementos de tipo int)

# Código de la función

`f(int n1, n2, n3, n4, n5, n6)`

```
int f (int n1, n2, n3,  
      n4, n5, n6)  
{  
    int v[4];  
    int k;  
  
    for (k= 0; k <3; k++)  
        v[i] = n1+n2+n3+n4+n5+n6;  
  
    return (v[1]);  
}
```

```
f:      addu      $sp, $sp, -4  
        sw       $fp, ($sp)  
        move     $fp, $sp  
        addu     $sp, $sp, -16  
        add      $t0, $a0, $a1  
        add      $t0, $t0, $a2  
        add      $t0, $t0, $a3  
        lw       $t1, 4($fp)  
        add      $t0, $t0, $t1  
        lw       $t1, 8($fp)  
        add      $t0, $t0, $t1
```



Se calcula la suma  $n1+n2+n3+n4+n5+n6$

# Código de la función

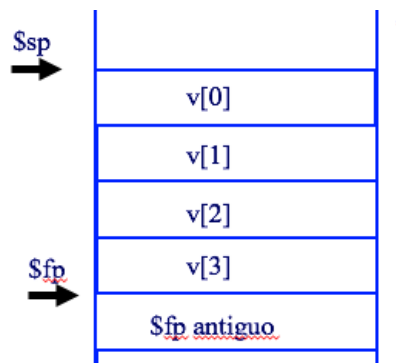
f(int n1, n2, n3, n4, n5, n6)

```
int f (int n1, n2, n3,
      n4, n5, n6)
{
    int v[4];
    int k;

    for (k= 0; k <3; k++)
        v[i] = n1+n2+n3+n4+n5+n6;

    return (v[1]);
}
```

Bucle



```
f:      addu      $sp, $sp, -4
        sw        $fp, ($sp)
        move      $fp, $sp
        addu      $sp, $sp, -16
        add       $t0, $a0, $a1
        add       $t0, $t0, $a2
        add       $t0, $t0, $a3
        lw        $t1, 4($fp)
        add       $t0, $t0, $t1
        lw        $t1, 8($fp)
        add       $t0, $t0, $t1
        li        $t1, 0    # indice
        move      $t2, $fp
        addi      $t2, $t2, -16 # desplaz.
        li        $t3, 3
bucle:  bgt       $t1, $t3, fin
        sw        $t0, ($t2)
        addi      $t2, $t2, 4
        addi      $t1, $t1, 1
        b         bucle
```

# Código de la función

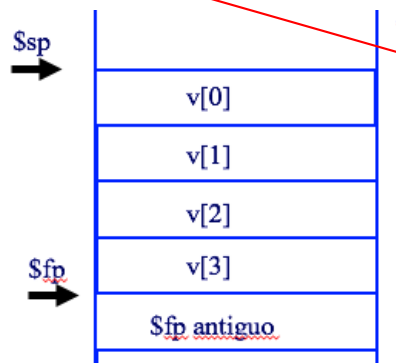
f(int n1, n2, n3, n4, n5, n6)

```
int f (int n1, n2, n3,
      n4, n5, n6)
{
    int v[4];
    int k;

    for (k= 0; k <3; k++)
        v[i] = n1+n2+n3+n4+n5+n6;

    return (v[1]);
}
```

```
f:      addu      $sp, $sp, -4
        sw        $fp, ($sp)
        move      $fp, $sp
        addu      $sp, $sp, -16
        add       $t0, $a0, $a1
        add       $t0, $t0, $a2
        add       $t0, $t0, $a3
        lw        $t1, 4($fp)
        add       $t0, $t0, $t1
        lw        $t1, 8($fp)
        add       $t0, $t0, $t1
        li        $t1, 0      # indice
        move      $t2, $fp
        addi      $t2, $t2, -16 # desplaz.
        li        $t3, 3
bucle:  bgt       $t1, $t3, fin
        sw        $t0, ($t2)
        addi      $t2, $t2, 4
        addi      $t1, $t1, 1
        b         bucle
fin:    lw        $v0, -12($fp)
```



Preparar  
el valor  
a retornar  
en \$v0

# Código de la función

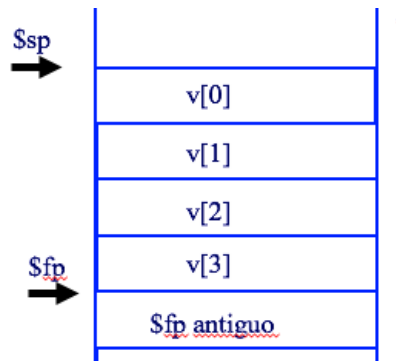
f(int n1, n2, n3, n4, n5, n6)

```
int f (int n1, n2, n3,
      n4, n5, n6)
{
    int v[4];
    int k;

    for (k= 0; k <3; k++)
        v[i] = n1+n2+n3+n4+n5+n6;

    return (v[1]);
}
```

```
f:      addu      $sp, $sp, -4
        sw        $fp, ($sp)
        move      $fp, $sp
        addu      $sp, $sp, -16
        add       $t0, $a0, $a1
        add       $t0, $t0, $a2
        add       $t0, $t0, $a3
        lw        $t1, 4($fp)
        add       $t0, $t0, $t1
        lw        $t1, 8($fp)
        add       $t0, $t0, $t1
        li        $t1, 0      # indice
        move      $t2, $fp
        addi      $t2, $t2, -16 # desplaz.
        li        $t3, 3
bucle:  bgt       $t1, $t3, fin
        sw        $t0, ($t2)
        addi      $t2, $t2, 4
        addi      $t1, $t1, 1
        b         bucle
fin:     lw        $v0, -12($fp)
        lw        $fp, ($fp)
```



Se restaura  
el valor de \$fp

# Código de la función

f(int n1, n2, n3, n4, n5, n6)

```
int f (int n1, n2, n3,
      n4, n5, n6)
{
    int v[4];
    int k;

    for (k= 0; k <3; k++)
        v[i] = n1+n2+n3+n4+n5+n6;

    return (v[1]);
}
```

```
f:      addu      $sp, $sp, -4
        sw        $fp, ($sp)
        move      $fp, $sp
        addu      $sp, $sp, -16
        add       $t0, $a0, $a1
        add       $t0, $t0, $a2
        add       $t0, $t0, $a3
        lw        $t1, 4($fp)
        add       $t0, $t0, $t1
        lw        $t1, 8($fp)
        add       $t0, $t0, $t1
        li        $t1, 0      # indice
        move      $t2, $fp
        addi      $t2, $t2, -16 # desplaz.
        li        $t3, 3
bucle:  bgt       $t1, $t3, fin
        sw        $t0, ($t2)
        addi      $t2, $t2, 4
        addi      $t1, $t1, 1
        b         bucle
fin:     lw        $v0, -12($fp)
        lw        $fp, ($fp)
        addu      $sp, $sp, 20
        jr        $ra
```

Se restaura  
la pila  
y se retorna



# Código posterior a la llamada a `f(int n1, n2, n3, n4, n5, n6)`

**Para la llamada:** `f (3, 4, 23, 12, 6, 7) ;`

<code>li</code>	<code>\$a0, 3</code>	}	Los cuatro primeros en registros \$ai
<code>li</code>	<code>\$a1, 4</code>		
<code>li</code>	<code>\$a2, 23</code>		
<code>li</code>	<code>\$a3, 12</code>		
<code>addu</code>	<code>\$sp, \$sp, -8</code>	}	El resto en la pila
<code>li</code>	<code>\$t0, 6</code>		
<code>sw</code>	<code>\$t0, 0(\$sp)</code>		
<code>li</code>	<code>\$t0, 7</code>		
<code>sw</code>	<code>\$t0, 4(\$sp)</code>		
<code>jal</code>	<code>f</code>	}	Deja el puntero de pila en el estado anterior En \$v0 está el valor devuelto: lo imprime
<code>addu</code>	<code>\$sp, \$sp, 8</code>		
<code>move</code>	<code>\$a0, \$v0</code>		
<code>li</code>	<code>\$v0, 1</code>		
<code>syscall</code>			

# Variables locales en registros

- Siempre que se puede, las variables locales (int, double, char, ...) se almacenan en registros

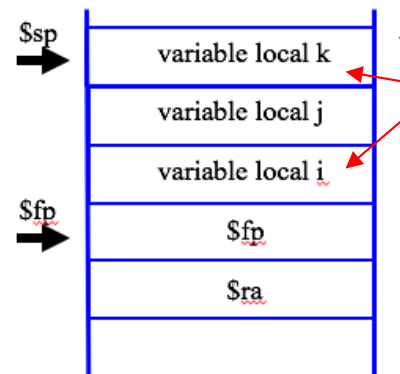
```
int f(...)  
{  
    int i, j, k;  
  
    i = 0;  
    j = 1;  
    k = i + j;  
    . . .  
}
```

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

# Variables locales en pila

- ▶ 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: . . .  
    #espacio para variables loc.  
    addu $sp, $sp, 12  
  
    li $t0, 0  
    sw $t0, -4($fp)  
    li $t1, 1  
    sw $t1, -8($fp)  
  
    li $t0, -4($fp)  
    li $t1, -8($fp)  
    add $t2, $t0, $t1  
    sw $t2, -12($fp)  
    . . .
```

# Convenio de paso de parámetros

- ▶ **Convenio que describe:**
  - ▶ Uso del banco de registros generales.
  - ▶ Uso del banco de registros FPU.
  - ▶ Uso de la pila.
  - ▶ Afecta a código llamante y código llamado.
- ▶ **Distintos compiladores usan distintos convenios.**
  - ▶ ABI → *Application Binary Interface*.

# Convenio del MIPS

- ▶ El puntero de pila siempre alineado a doble palabra (múltiplo de 8)
- ▶ El procedimiento llamado reserva espacio para los registros `$a0..$a3`
  - ▶ El mínimo marco de pila ocupa 24 bytes

```
f:      addu $sp, $sp, -24
        sw   $ra, 20($sp)
        sw   $fp, 16($sp)
```
- ▶ Durante el tema se ha utilizado un convenio más simplificado y ligiramente distinto al empleado realmente en el MIPS

# 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
```

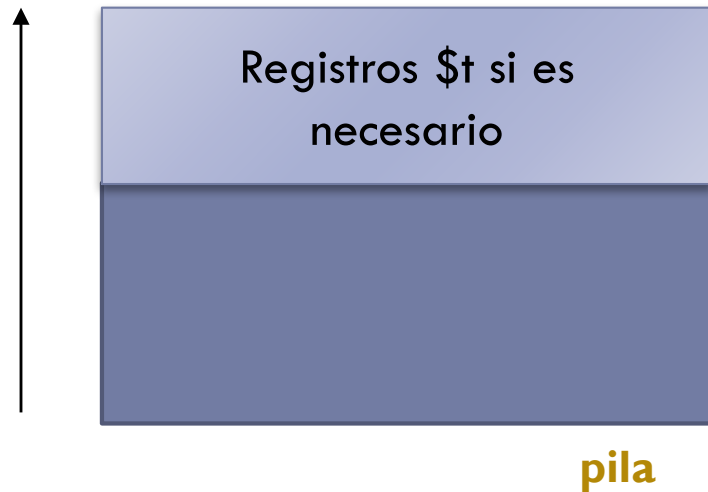
```
    b: .word 7
```

```
    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

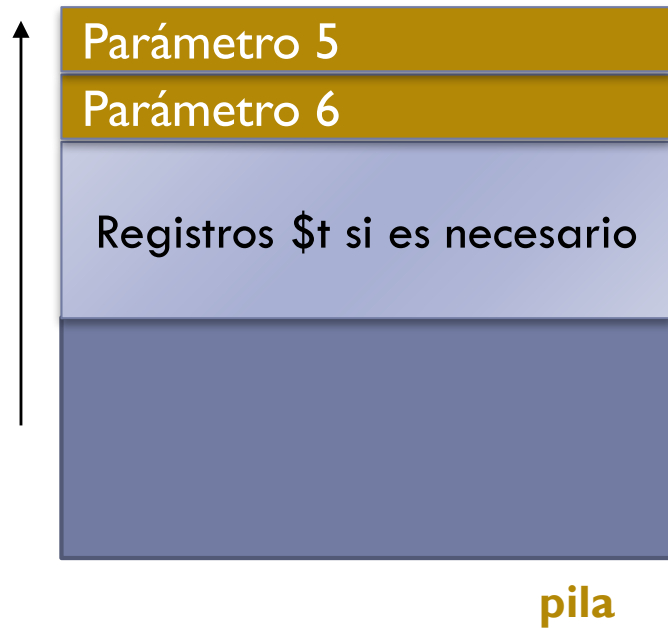
\$a0	Parámetro 1
\$a1	Parámetro 2
\$a2	Parámetro 3
\$a3	Parámetro 4

```
li $a0, 5    // param 1
li $a1, 8    // param 2

jal func

addu $sp, $sp, 16
```

# Paso de 6 parámetros



## Banco de registros

\$a0	Parámetro 1
\$a1	Parámetro 2
\$a2	Parámetro 3
\$a3	Parámetro 4

```
li $a0, 5      // param 1
li $a1, 8      // param 2
li $a2, 7      // param 3
li $a3, 9      // param 4
```

```
addu $sp, $sp, -8
li $t0, 10     // param 6
sw $t0, 4($sp)
li $t0, 7
s2 $t0, ($sp)  // param 5
```

```
jal func
```

```
addu $sp, $sp, 8
```



# Llamada a subrutina

## Subrutina llamante

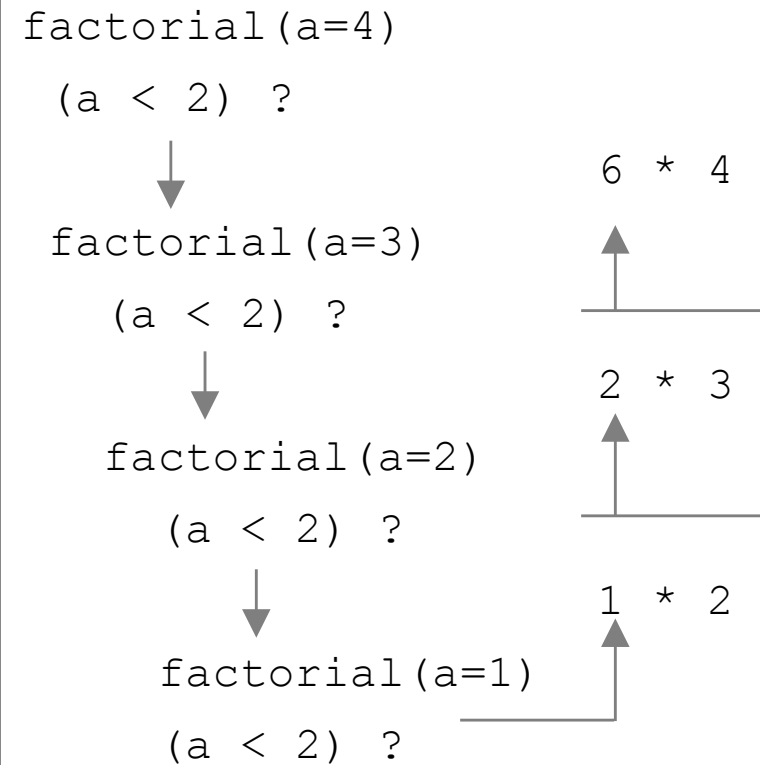
- ▶ Instrucción de salto “*and link*”
  - ▶ jal etiqueta
  - ▶ bal etiqueta
  - ▶ bltzal \$reg, etiqueta
  - ▶ bgezal \$reg, etiqueta
  - ▶ jalr \$reg
  - ▶ jalr \$reg, \$reg

# Ejemplo: factorial

```
int factorial ( int a )
{
    if (a < 2) then
        return 1 ;
    return a * factorial(a-1) ;
}

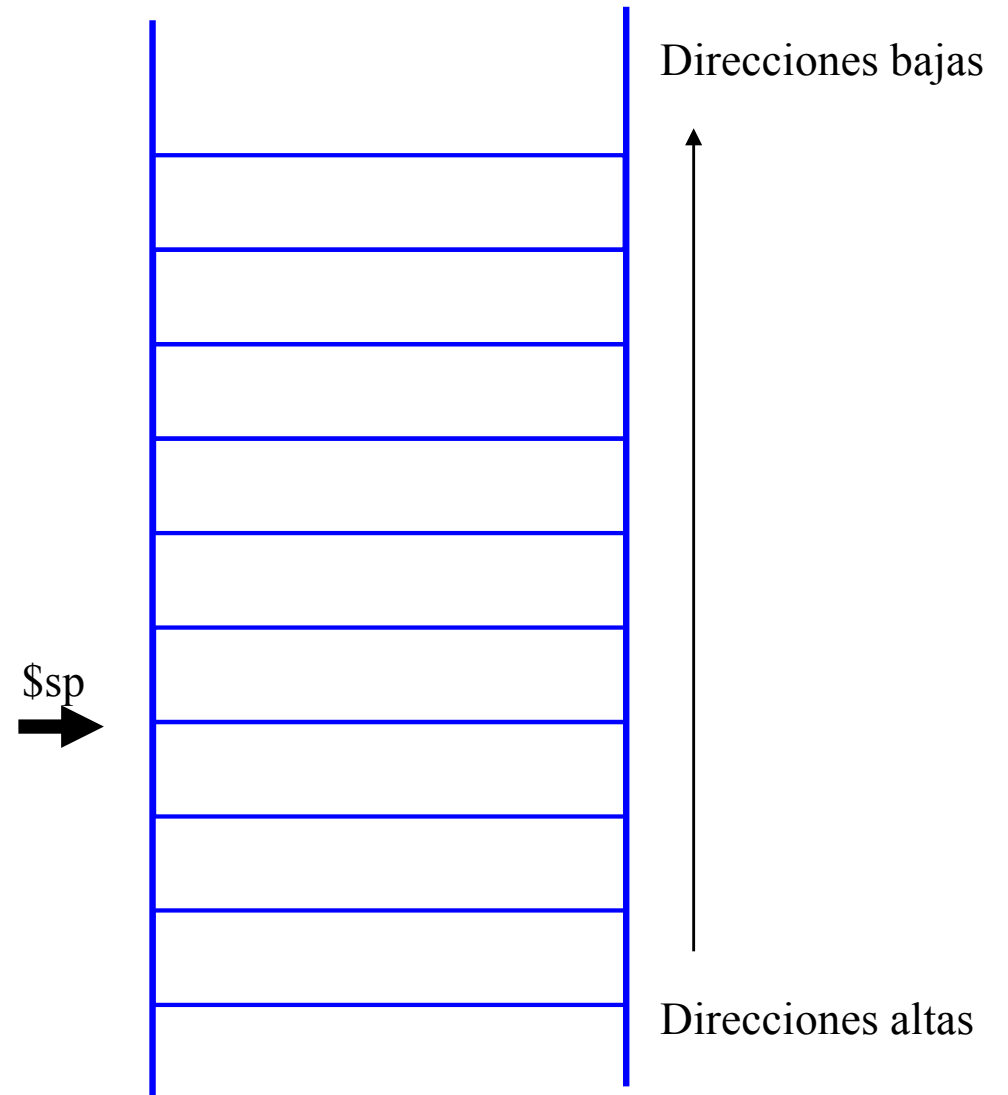
void main () {
    int resultado ;
    resultado=factorial(4) ;

    printf("f(4)=%d",resultado) ;
}
```



# Ejemplo: factorial

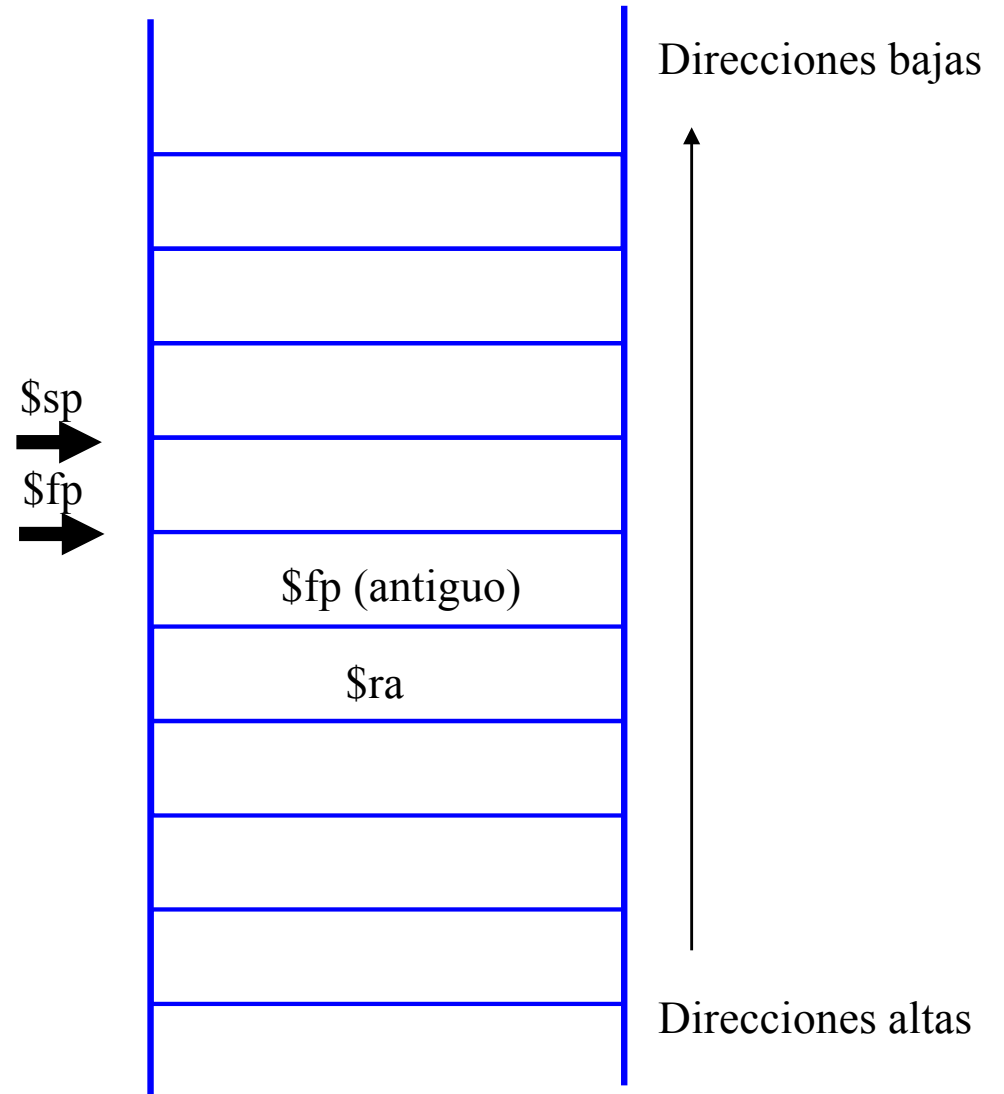
Justo antes de la llamada



# Ejemplo: factorial

factorial:

```
# frame stack
subu $sp $sp 12
sw   $ra 8($sp)
sw   $fp 4($sp)
addu $fp $sp 4
```



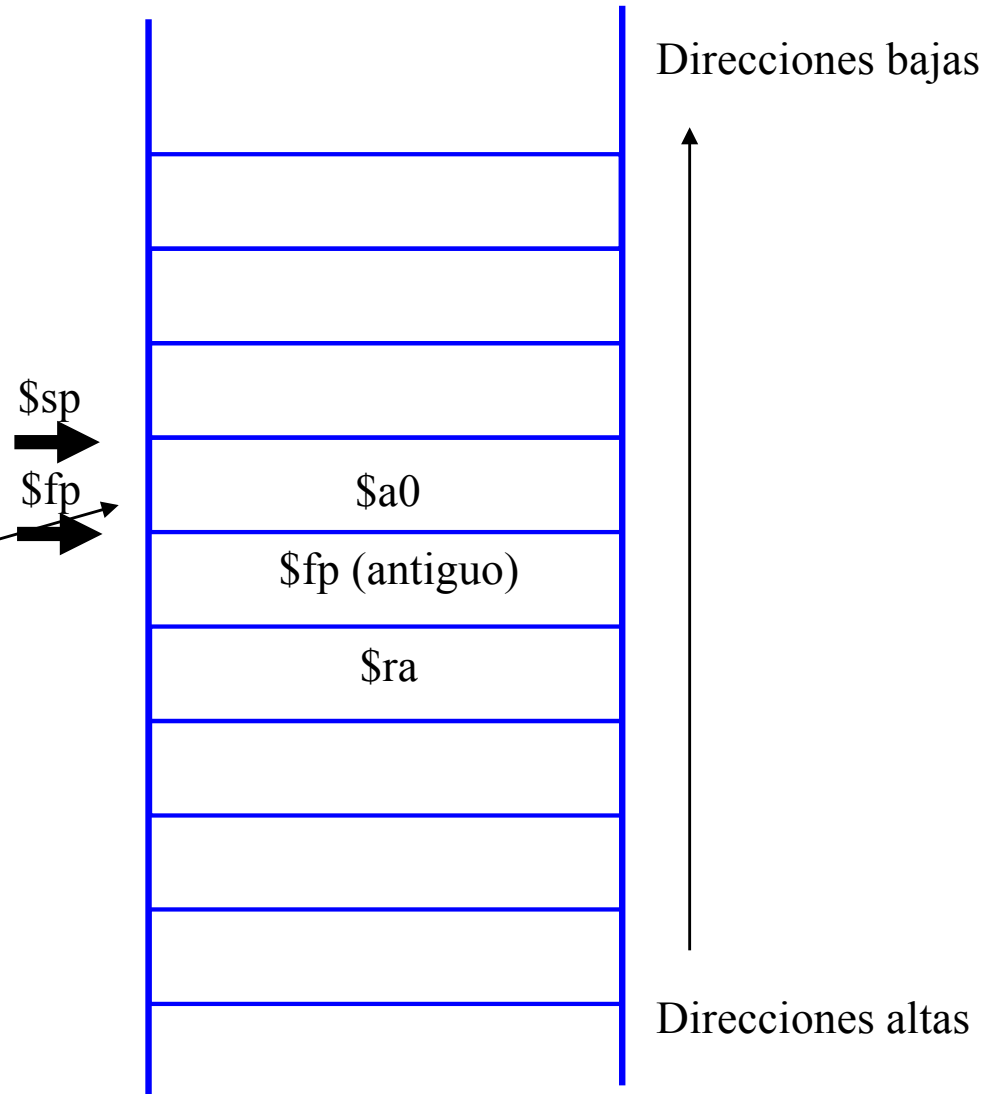
# Ejemplo: factorial

factorial:

```
# frame stack
subu $sp $sp 12
sw   $ra 8($sp)
sw   $fp 4($sp)
addu $fp $sp 4
```

```
bge $a0 2 b_else
li  $v0 1
b   b_efs
```

```
b_else: sw $a0 -4($fp)
        addi $a0 $a0 -1
        jal factorial
        lw $v1 -4($fp)
        mul $v0 $v0 $v1
```



# Ejemplo: factorial

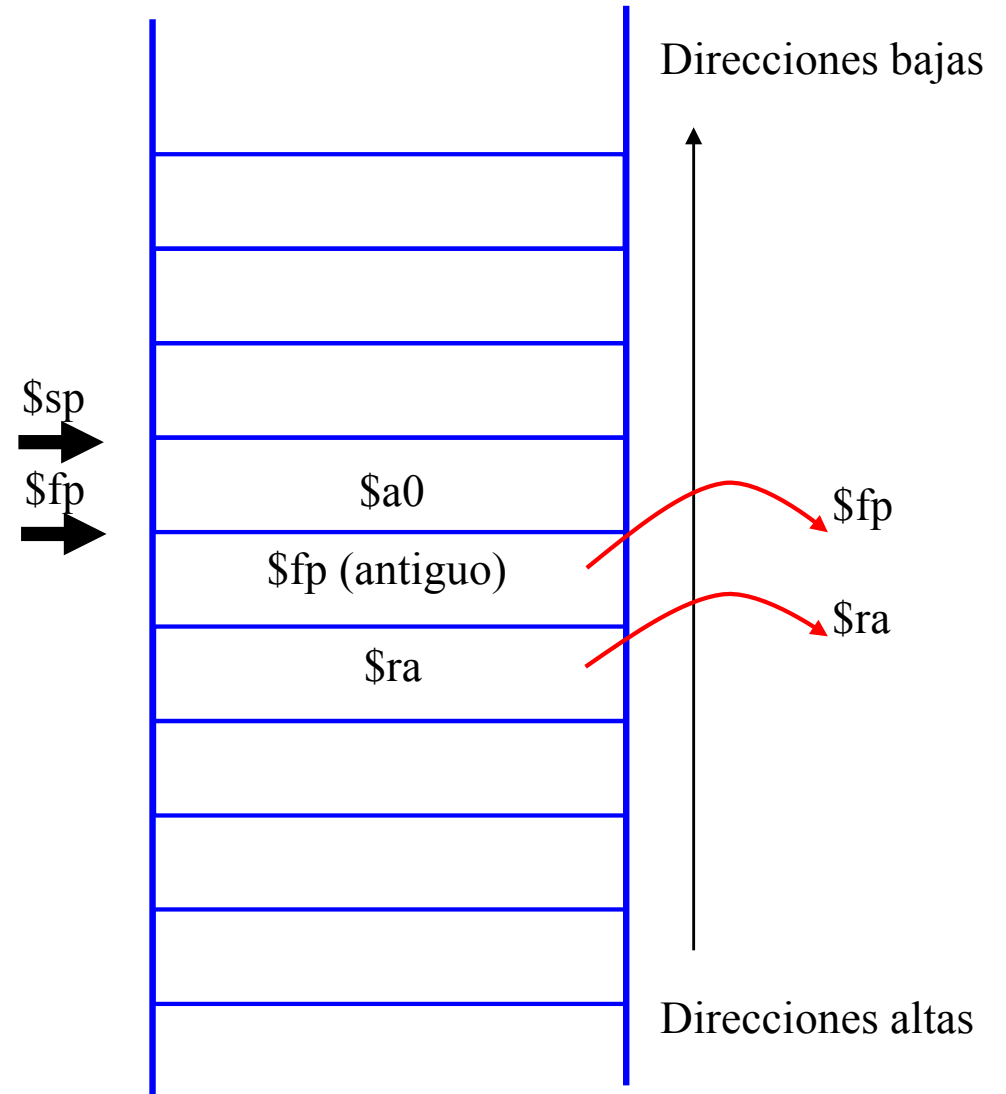
factorial:

```
# frame stack
subu $sp $sp 12
sw   $ra 8($sp)
sw   $fp 4($sp)
addu $fp $sp 8

bge $a0 2 b_else
li  $v0 1
b   b_efs

b_else: sw $a0 -4($fp)
        addi $a0 $a0 -1
        jal factorial
        lw $v1 -4($fp)
        mul $v0 $v0 $v1

# end frame stack
b_efs:  lw $ra 8($sp)
        lw $fp 4($sp)
```



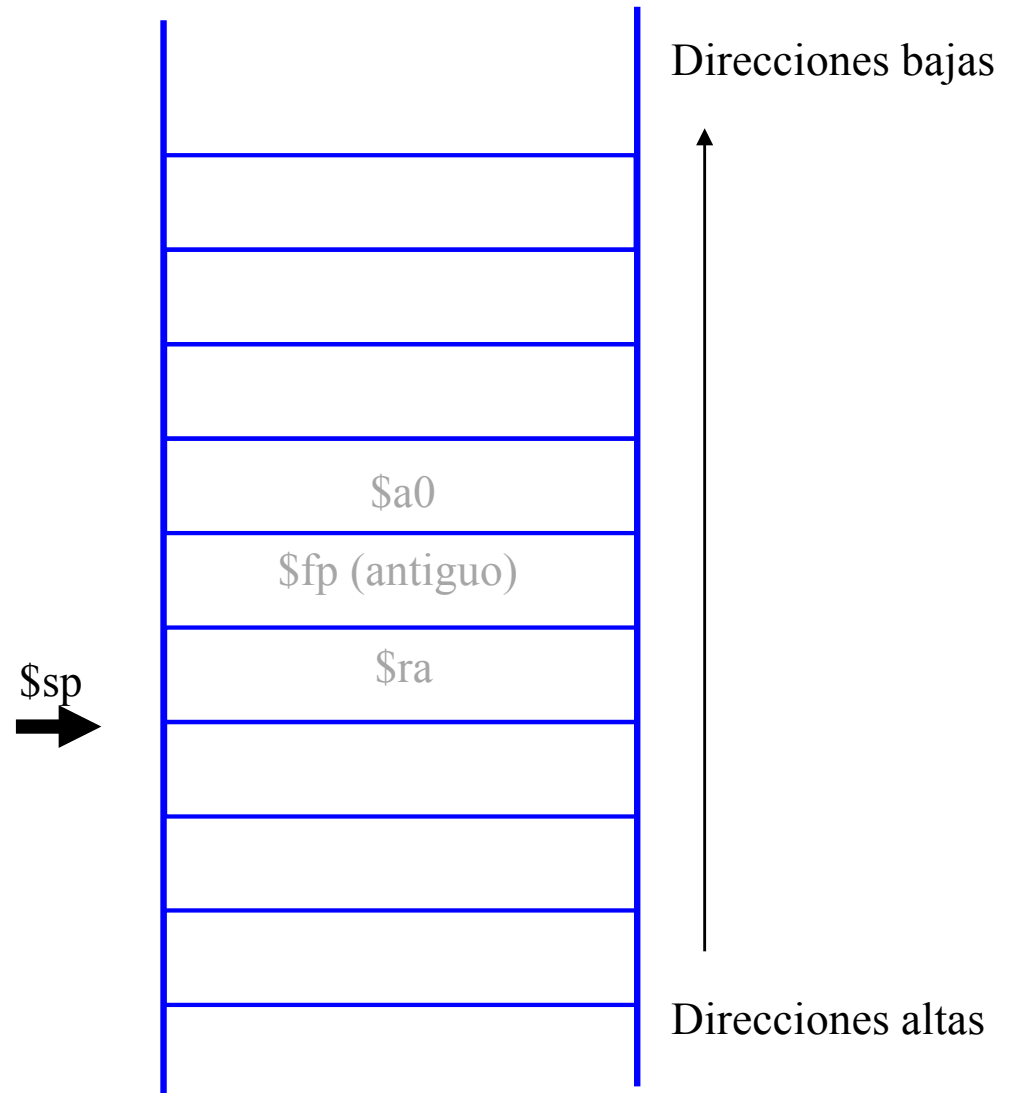
# Ejemplo: factorial

factorial:

```
# frame stack
subu $sp $sp 12
sw    $ra 8($sp)
sw    $fp 4($sp)
addu $fp $sp 8

bge $a0 2 b_else
li   $v0 1
b    b_efs

b_else: sw $a0 -4($fp)
        addi $a0 $a0 -1
        jal factorial
        lw $v1 -4($fp)
        mul $v0 $v0 $v1
# end frame stack
b_efs:  lw $ra 8($sp)
        lw $fp 4($sp)
        addu $sp $sp 12
        jr $ra
```



# Paso de parámetros en C

```
int f1(int a, int b) {  
    int z;  
  
    z = a + b;  
}
```

```
f1:  add $v0, $a0, $a1  
     jr $ra
```

En C, todos los parámetros se pasan por valor



# Paso de parámetros en C

```
int f2(int a, int *b) {  
    int z;  
  
    z = a + *b;  
    *b = z;  
}
```

```
f2: lw    $t0, ($a1)  
     add  $v0, $a0, $t0  
     sw   $v0, ($a1)  
     jr   $ra
```

En C, todos los parámetros se pasan por valor

- En este caso b es una dirección de memoria, pero se pasa por valor
- La función puede modificar el contenido de esa posición de memoria, puesto que se pasa su dirección

# Paso de parámetros en C

```
.data:
    a: .word 8

    f1:  add $v0, $a0, $a1
        jr $ra

.text:

main:
    . . .
    # llamada:  f1(a, 9)
    lw      $a0, a      # se pasa el valor de a
    li      $a1, 9      # se pasa el valor 9
    jal     f1
    move     $a0, $v0
    li      $v0, 1
    syscall          # se imprime el valor que devuelve
    . . .
```

# Llamadas a las funciones anteriores

```
.data:                                f2: lw    $t0, ($a1)
      a: .word 8                      add    $v0, $a0, $t0
                                      sw     $v0, ($a1)
.text:                                jr     $ra

main:
    . . .
    # llamada: f2(1, &a)
    li     $a0, 1      # se pasa el valor 1
    la     $a1, a      # se pasa la dirección de a
    jal    f2
    move   $a0, $v0
    li     $v0, 1
    syscall                # se imprime el valor que devuelve
    lw     $a0, a
    syscall                # se imprime el valor de a
```

# Asignación dinámica de memoria en SPIM

- ▶ Llamada al sistema `sbrk()` en SPIM
  - ▶ `$a0`: número de bytes a reservar
  - ▶ `$v0 = 9` (código de llamada al sistema)
  - ▶ Devuelve en `$v0` la dirección del bloque reservado
  - ▶ En SPIM no hay una llamada al sistema para liberar memoria (`free`)

```
int *p;  
  
p = malloc(20*sizeof(int));  
  
p[0] = 1;  
p[1] = 4;
```

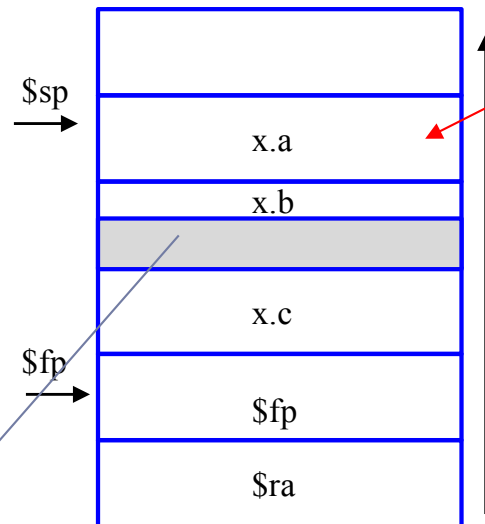
```
# se reservan 80 bytes  
li $a0, 80  
li $v0, 9 # código de  
           # llamada  
syscall  
  
move $a0, $v0  
li $t0, 1  
sw $t0, ($a0)  
li $t0, 4  
sw $t0, 4($a0)
```

# Uso de estructuras de C (structs)

- ▶ Las variables locales de tipo struct se asignan en la pila
- ▶ C puede pasar estructuras completas a las funciones
  - ▶ Se pasan en la pila
- ▶ Una función en C puede devolver una estructura
  - ▶ La función que llama reserva espacio en la pila para que la función llamada deje allí el resultado a devolver

# Variables locales de tipo struct

```
struct S {  
    int a;  
    char b;  
    int c;  
}  
  
int f(...)  
{  
    struct S x;  
  
    x.a = 2;  
    x.b = 'a';  
    x.c = 3;  
    . . .  
}
```

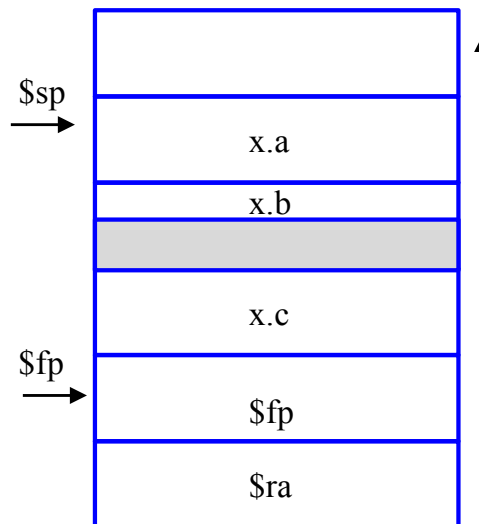


hueco para alineamiento de x.c

```
# se reserva el struct  
addu $sp, $sp, -12  
  
addu $t0, $fp, -12  
  
li    $t1, 2  
sw    $t1, 0($t0) #x.a  
  
li    $t1, 'a'  
sb    $t1, 4($t0) #x.b  
  
li    $t1, 3  
sw    $t1, 8($t0) #x.c
```

# Llamadas a funciones con struct

```
struct S {  
    int a;  
    char b;  
    int c;  
}  
  
void f1(struct S p) {  
    . . .  
}  
  
int f2(...)  
{  
    struct S x;  
  
    x.a = 2;  
    x.b = 'a';  
    x.c = 3;  
    f1(x);  
    . . .  
}
```



```
# se reserva el struct  
  
addu $sp, $sp, -12  
addu $t0, $fp, -12  
li    $t1, 2  
sw    $t1, 0($t0)  
li    $t1, 'a'  
sb    $t1, 4($t0)  
li    $t1, 3  
sw    $t1, 8($t0)
```

# Llamadas a funciones con struct

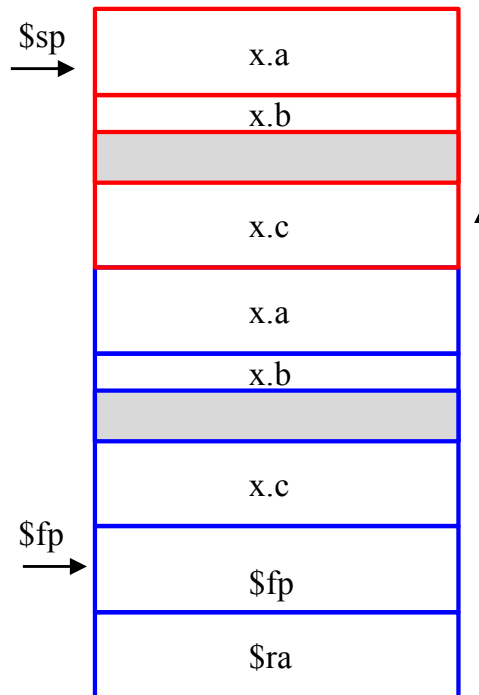
```
struct S {
    int a;
    char b;
    int c;
}

void f1(struct S p) {
    . . .
}

int f2(...)
{
    struct S x;

    x.a = 2;
    x.b = 'a';
    x.c = 3;
    f1(x);
    . . .
}
```

se copian los  
argumentos de f1  
en la pila (paso por  
valor)



# se reserva el struct

```
addu $sp, $sp, -12
addu $t0, $fp, -12
li    $t1, 2
sw    $t1, 0($t0)
li    $t1, 'a'
sb    $t1, 4($t0)
li    $t1, 3
sw    $t1, 8($t0)
```

# proceso de llamada

```
addu $sp, $sp, -12
lw    $t1, 0($t0) #x.a
sw    $t1, 0($sp)

lb    $t1, 4($t0) #x.b
sb    $t1, 4($sp)

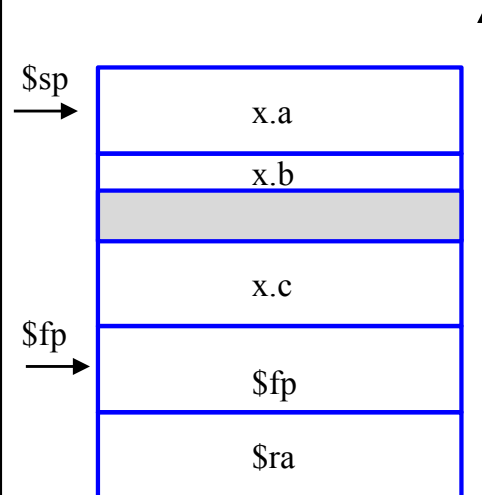
lw    $t1, 8($t0) #x.c
sw    $t1, 8($sp)
jal   f1
addu $sp, $sp, 12
```



# Llamadas a funciones con punteros a struct

```
struct S {  
    int a;  
    char b;  
    int c;  
}  
  
void f1(struct S *p) {  
    . . .  
}  
  
int f2(...)  
{  
    struct S x;  
  
    x.a = 2;  
    x.b = 'a';  
    x.c = 3;  
    f1(&x);  
    . . .  
}
```

En este caso se pasa  
una dirección  
en \$a0  
no se copia nada en la  
pila



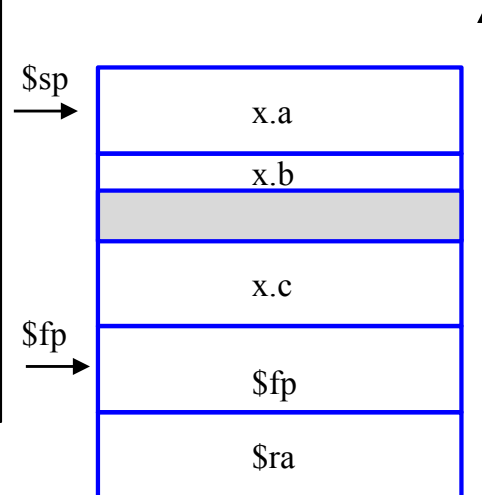
# se reserva el struct

```
addu $sp, $sp, -12  
addu $t0, $fp, -12  
li    $t1, 2  
sw    $t1, 0($t0)  
li    $t1, 'a'  
sb    $t1, 4($t0)  
li    $t1, 3  
sw    $t1, 8($t0)
```

```
addi  $a0, $fp, -12  
jal   f1
```

# Funciones que devuelve structs

```
struct S {  
    int a;  
    char b;  
    int c;  
}  
  
struct S f1() {  
    . . .  
}  
  
int f2(...)  
{  
    struct S x;  
  
    x= f1();  
    . . .  
}
```

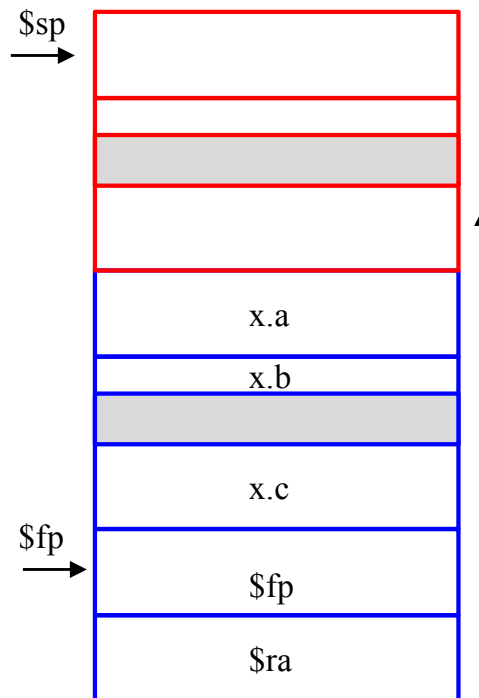


```
# se reserva el struct  
  
addu $sp, $sp, -12  
addu $t0, $fp, -12  
li    $t1, 2  
sw    $t1, 0($t0)  
li    $t1, 'a'  
sb    $t1, 4($t0)  
li    $t1, 3  
sw    $t1, 8($t0)
```

# Funciones que devuelve structs

```
struct S {  
    int a;  
    char b;  
    int c;  
}  
  
struct S f1() {  
    struct S p;  
  
    return p;  
}  
  
int f2(...)  
{  
    struct S x;  
  
    x= f1();  
    . . .  
}
```

la función que llama  
reserva espacio en la  
pila para el resultado



# se reserva el struct

```
addu $sp, $sp, -12  
addu $t0, $fp, -12  
li    $t1, 2  
sw    $t1, 0($t0)  
li    $t1, 'a'  
sb    $t1, 4($t0)  
li    $t1, 3  
sw    $t1, 8($t0)
```

```
addu $sp, $sp, -12  
jal   f1  
#recupera el valor  
lw    $t0, 0($sp)  
sw    $t0, -12($fp)
```

```
lb    $t0, 4($sp)  
sb    $t0, -8($sp)
```

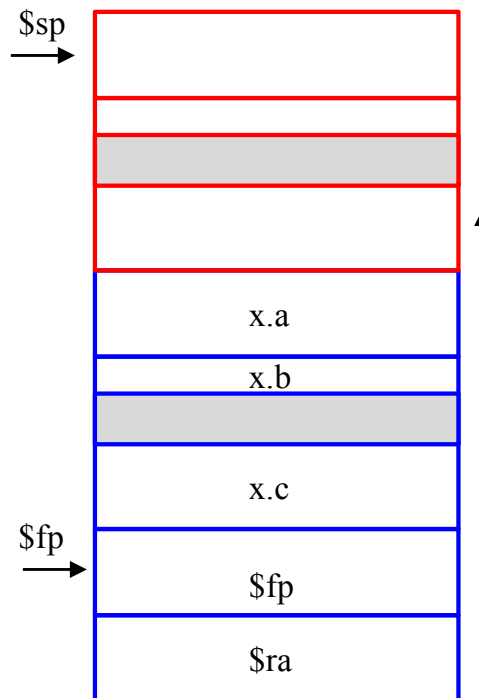
```
lw    $t0, 8($sp)  
sw    $t0, -4($fp)
```

```
addu $sp, $sp, 12
```

# Ejercicio

```
struct S {  
    int a;  
    char b;  
    int c;  
}  
  
struct S f1() {  
    struct S p;  
  
    return p;  
}  
  
int f2(...)  
{  
    struct S x;  
  
    x= f1();  
    . . .  
}
```

Escriba el código necesario para que f1()  
devuelva la estructura y la copie en la pila



# Traducción y ejecución de programas

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

# Código compilado frente a interpretado

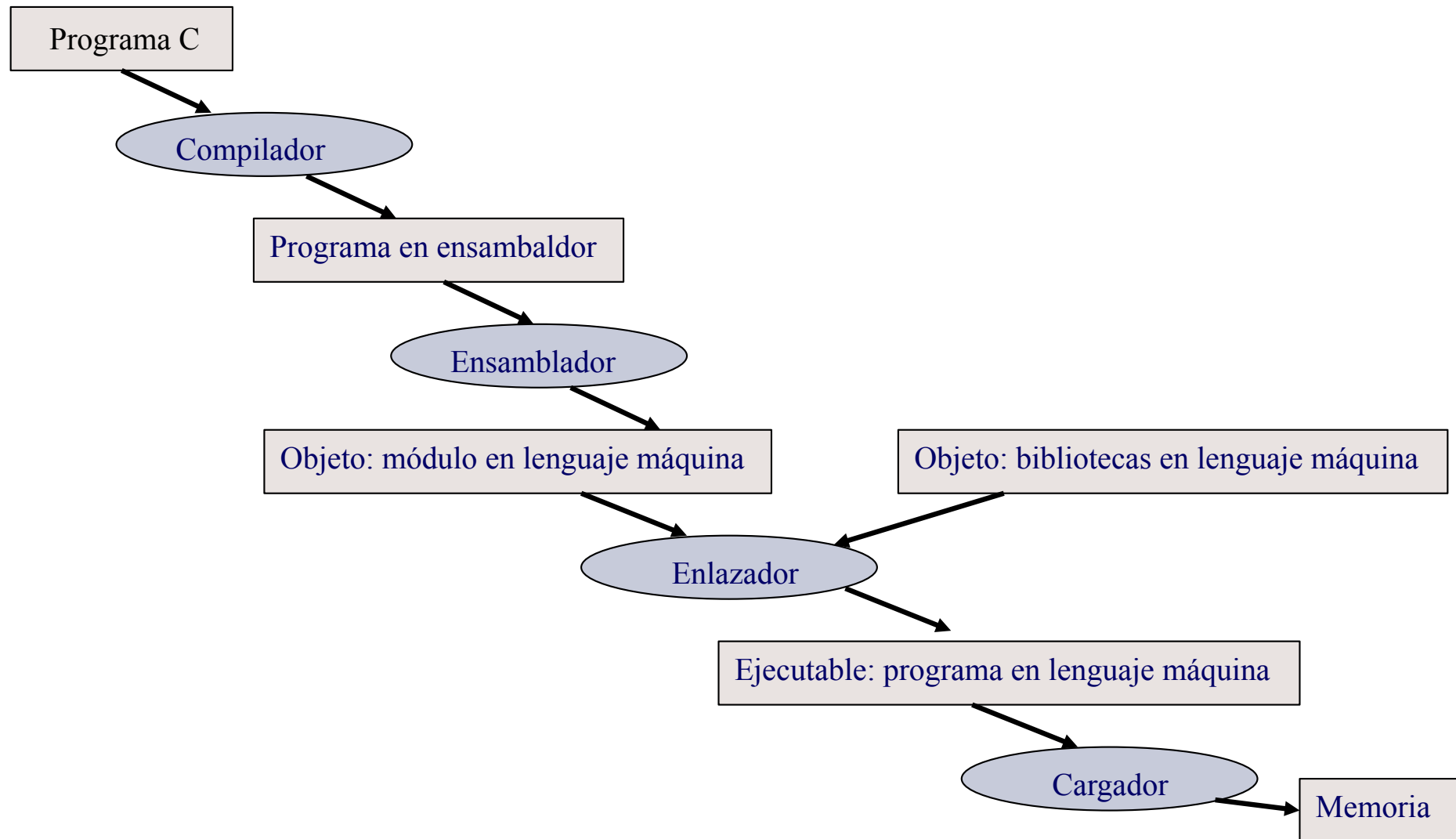
## ▶ Código compilado:

- ▶ Los programas son traducidos a código máquina de un computador
  - ▶ El código es ejecutado directamente por el computador
  - ▶ Generalmente más eficiente

## ▶ Código interpretado:

- ▶ Un interprete es un programa que ejecuta otros programas
- ▶ Un interprete ejecuta un conjunto de instrucciones independientes de la máquina. Las instrucciones son ejecutadas por un programa
- ▶ Ejemplo: Java es traducido a un *byte code* que es ejecutado por un interprete (*Java Virtual Machine*)
- ▶ Generalmente es más fácil escribir un interprete. Mayor portabilidad

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



# Compilador

- ▶ Entrada: lenguaje de alto nivel (C, C++, ...)
- ▶ Salida: código en lenguaje ensamblador
- ▶ Puede contener pseudoinstrucciones
- ▶ Una **pseudoinstrucción** es una instrucción que entiende el ensamblador pero que no tiene correspondencia directa con una instrucción en lenguaje máquina
  - ▶ `move $t1, $t2`  $\Rightarrow$  `or $t1, $t2, $zero`



# Ensamblador

- ▶ Entrada: código en lenguaje ensamblador
- ▶ Salida: **Código objeto** escrito en lenguaje máquina
- ▶ En ensamblador convierte las pseudoinstrucciones a instrucciones máquina
- ▶ Analiza las sentencias en ensamblador de forma independiente, sentencia a sentencia
- ▶ El ensamblador produce un **fichero objeto (.o)**

# Análisis de sentencias en ensamblador

- ▶ Se comprueba si la instrucción es correcta (código de operación, operandos, direccionamientos válidos, ...)
- ▶ Se comprueba si la sentencia tiene etiqueta. Si la tiene comprueba que el código simbólico no está repetido y le asigna el valor correspondiente a la posición de memoria que habrá de ocupar la instrucción o el dato.
- ▶ Construye una **tabla de símbolos** con todas las etiquetas simbólicas
  - ▶ En una primera fase o pasada se determinan todos los valores que no conllevan referencias adelantadas
  - ▶ En una segunda fase o pasada se resuelven aquellas etiquetas que han quedado pendientes

# Formato de un fichero objeto

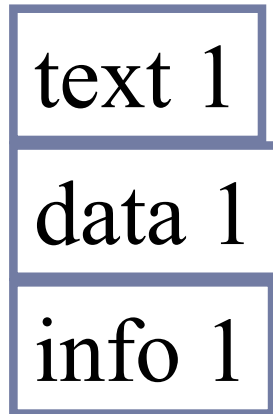
- ▶ **Cabecera del fichero.** Describe el tamaño y posición de los elementos dentro del fichero
- ▶ **Segmento de texto:** contiene el código máquina
- ▶ **Segmento de datos:** contiene los datos de las variables globales
- ▶ **Información de reubicación:** identifica instrucciones o palabras de datos que dependen de una dirección absoluta cuando el programa se cargue en memoria
  - ▶ Cualquier etiqueta de `j` or `jal` (internas o externas)
  - ▶ Direcciones de datos
- ▶ **Tabla de símbolos:** etiquetas no definidas en este módulo (referencias externas)
- ▶ **Información de depuración.** Permite asociar instrucciones máquina con código C e interpretar las estructuras de datos

# Enlazador

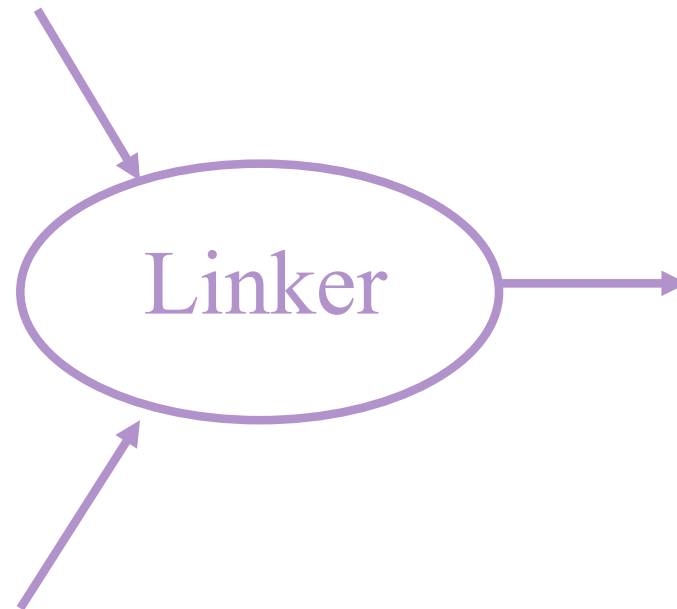
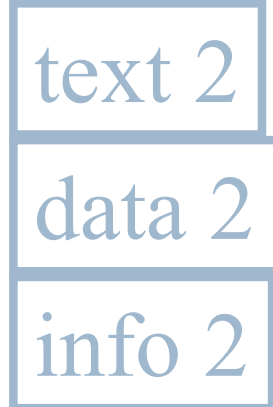
- ▶ Entrada: ficheros en código objeto
- ▶ Salida: Código ejecutable
- ▶ Combina varios archivos objeto (.o) en un único fichero ejecutable
- ▶ Resuelve todas las referencias (instrucciones de salto y direcciones de datos)
- ▶ En enlazador asume que la primera palabra del segmento de texto está en la dirección 0x00000000
- ▶ Permite la compilación separada de ficheros
  - ▶ El cambio en un fichero no implica recompilar todo el programa completo
  - ▶ Permite el uso de funciones de biblioteca (.a)

# Enlazador

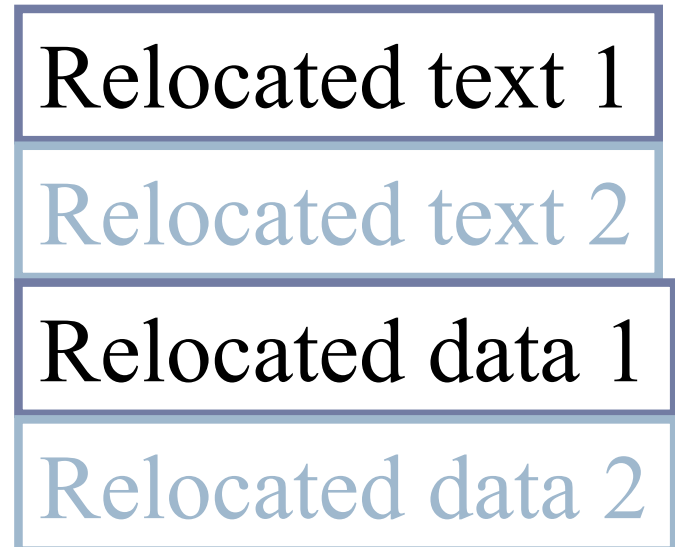
.o file 1



.o file 2



**a.out**

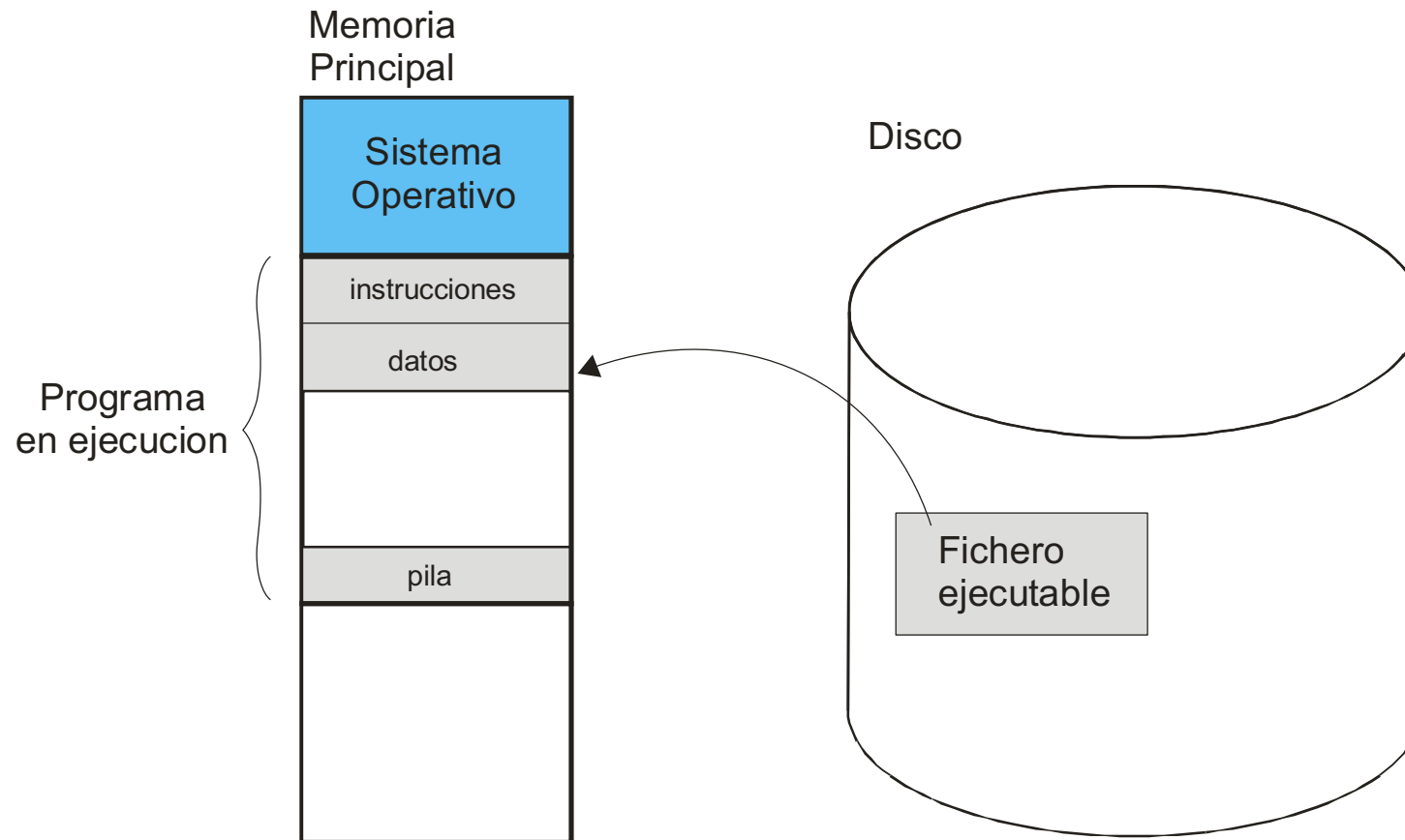


# Formato de un fichero ejecutable

- ▶ **Cabecera del fichero.** Describe el tamaño y posición de los elementos dentro del fichero. Incluye la dirección de inio del programa
- ▶ **Segmento de texto:** contiene el código máquina
- ▶ **Segmento de datos:** contiene los datos de las variables globales con valor inicial
- ▶ **Información de reubicación:** en caso de utilizar bibliotecas dinámicas

# Cargador

- Lee un fichero ejecutable (a.out) y lo carga en memoria



# Cargador

- ▶ Forma parte del sistema operativo
- ▶ Lee la cabecera del ejecutable para determinar el tamaño de los segmentos de texto y datos
- ▶ Crea un nuevo espacio de direcciones en memoria para ubicar el segmento de texto, datos y pila
- ▶ Copia las instrucciones y los datos con valor inicial del fichero ejecutable (disco) a memoria
- ▶ Copia los argumentos que se pasan al programa en la pila
- ▶ Inicializa los registros. Fija el PC y el SP a sus posiciones



# Bibliotecas

- ▶ Una **biblioteca** es una colección de objetos normalmente relacionados entre sí
- ▶ Los módulos objetos de los programas pueden incluir referencias a símbolos definidos en alguno de los objetos de una biblioteca (funciones o variables exportadas)
- ▶ Las **bibliotecas del sistema** son un conjunto de bibliotecas predefinidas que ofrecen servicios a las aplicaciones
- ▶ Tipos:
  - ▶ Bibliotecas estáticas: se enlazan con los ficheros objeto para producir un fichero ejecutable que incluye todas las referencias resueltas. Un cambio en la biblioteca implica volver a enlazar y generar el ejecutable
  - ▶ Bibliotecas dinámicas (DLL, *dynamically linked library*)

# Bibliotecas dinámicas

- ▶ Las rutinas de las bibliotecas no se enlazan en el archivo ejecutable y no se cargan hasta que el programa se ejecuta
- ▶ El programa incluye información para la localización de las bibliotecas y la actualización de las referencias externas durante la ejecución
- ▶ Ventajas:
  - ▶ Da lugar a ejecutables más pequeños.
  - ▶ Solo se carga de la biblioteca aquello que se utiliza durante la ejecución.
  - ▶ El cambio en una biblioteca no afecta al ejecutable. No se necesita volver a generar un nuevo ejecutable.

# Ejemplo

$C \Rightarrow ASM \Rightarrow Obj \Rightarrow Exe \Rightarrow Ejecución$

## Programa C: ejemplo.c

```
#include <stdio.h>
int main (int argc, char *argv[])
{
    int i, sum = 0;
    for (i = 1; i <= 10; i++)
        sum = sum + i + i;

    printf ("La suma 1 + ... +10 es %d\n", sum);
}
```

`printf()`: función de biblioteca en `libc.a`

# Compilación

```
— .text
  .align    2
  .globl    main
main:
  subu $sp,$sp,24
  sw  $ra, 20($sp)
  sw  $a0, 4($sp)
  sw  $a1, 8($sp)

  li $t0, 0
  li $t1, 0
bucle:
  bgt $t0, 10, fin
  add $t1, $t1, $t0
  addi $t0, $t0, 1
  b bucle
```

```
fin:
  la  $a0, str
  li  $a1, $t1
  jal printf
  move $v0, $0
  lw  $ra, 20($sp)
  lw  $a0, 4($sp)
  lw  $a1, 8($sp)
  addiu $sp,$sp,24
  jr  $ra
  .data
  .align    0

str:
  .asciiz "+10 La suma es %d\n"
```

# Compilación

```
.text
    .align    2
    .globl    main
main:
    subu $sp,$sp,24
    sw  $ra, 20($sp)
    sw  $a0, 4($sp)
    sw  $a1, 8($sp)

    li $t0, 0
    li $t1, 0
bucle:
    bgt $t0, 10, fin
    add $t1, $t1, $t0
    addi $t0, $t0, 1
    b bucle
```

```
fin:
    la  $a0, str
    li  $a1, $t1
    jal printf
    move $v0, $0
    lw  $ra, 20($sp)
    lw  $a0, 4($sp)
    lw  $a1, 8($sp)
    addiu $sp,$sp,24
    jr  $ra
    .data
    .align    0
    str:.asciiz "La
suma 1 + ... +10
es %d\n"
```

**7 pseudo-  
instrucciones**

# Compilación

## Eliminación de pseudoinstrucciones

```
— .text
  .align    2
  .globl    main
main:
  addiu $29,$29,-24
  sw $31, 20($29)
  sw $4, 4($29)
  sw $5, 8($29)
  ori $8, $0, 0
  ori $9, $0, 0
bucle:
  slti $1, $8, 11
  beq $1, $0, fin
  add $9, $9, $8
  addi $8, $8, 1
  bgez $0, bucle
```

```
fin:
  lui $4, l.str
  ori $4, $4, r.str
  addu $4, $0, $9
  jal printf
  addu $2, $0, $0
  lw $31, 20($29)
  lw $4, 4($29)
  lw $5, 8($29)
  addiu $29,$29,24
  jr $31
```

# Compilación

## Asignación de direcciones

```
00    addiu $29,$29,-24
04    sw     $31, 20($29)
08    sw     $4, 4($29)
0c    sw $5, 8($29)
10    ori $8, $0, 0
14    ori $9, $0, 0
18    slti $1, $8, 11
1c    beq  $1, $0, fin
20    add  $9, $9, $8
24    addi $8, $8, 1
28    bgez $0, bucle
```

```
2c    lui $4, l.str
30    ori $4, $4, r.str
34    addu $4, $0, $9
38    jal printf
3c    addu $2, $0, $0
40    lw     $31, 20($29)
44    lw $4, 4($29)
48    lw $5, 8($29)
4c    addiu $29,$29,24
50    jr $31
```

# Compilación

## Creación de la tabla de símbolos y de reubicación

### ■ Tabla de símbolos

Etiqueta	dirección (en modulo)	tipo
main:	0x00000000	global text
bucle:	0x0000001c	local text
str:	0x00000000	local data

### ■ Información de reubicación

Dirección	tipo Instr.	Dependencia
0x0000002c	lui	l.str
0x00000030	ori	r.str
0x00000038	jal	printf



# Compilación

## Resolver etiquetas relativas a PC

```
00    addiu $29,$29,-24
04    sw      $31, 20($29)
08    sw      $4, 4($29)
0c    sw $5, 8($29)
10    ori $8, $0, 0
14    ori $9, $0, 0
18    slti $1, $8, 11
1c    beq $1, $0, 3
20    add $9, $9, $8
24    addi $8, $8, 1
28    bgez $0, -4
```

```
2c    lui $4, l.str
30    ori $4, $4, r.str
34    addu $4, $0, $9
38    jal printf
3c    addu $2, $0, $0
40    lw      $31, 20($29)
44    lw $4, 4($29)
48    lw $5, 8($29)
4c    addiu $29,$29,24
50    jr $31
```

# Segmento de texto en el fichero objeto

0x000000	0010011110111101111111111111101000
0x000004	1010111110111111100000000000010100
0x000008	10101111101001000000000000000100
0x00000c	101011111010010100000000000001000
0x000010	10001101000000000000000000000000
0x000014	1010110100100000000000000000011100
0x000018	0010100000010100000000000000001011
0x00001c	0001000000010000000000000000000011
0x000020	00000000100101000001001000000100000
0x000024	00100000100000100000000000000000001
0x000028	00000010000000000000111111111111100
0x00002c	0011110000000010000000000000000000
0x000030	0011010010000010000000000000000000
0x000034	0000000010010010000000000000000100001
0x000038	0000110000000000000000000000000000
0x00003c	000000000000000000000000010000001000001
0x000040	10001111101111111000000000000010100
0x000044	100011111010010000000000000000000100
0x000048	100011111010010100000000000000001000
0x00004c	000000011111000000000000000000011000
0x000050	0000000000000000000000011101000000001000

# Enlazado

- ▶ Combinar ejemplo.o y libc.a
- ▶ Crear las direcciones de memoria absolutas
- ▶ Modificar y mezclar las tablas de símbolos y de reubicación

- ▶ **Symbol Table**

Label	Address
main:	0x00000000
loop:	0x0000001c
str:	0x10000430
printf:	0x000003b0 ...

- ▶ **Relocation Information**

Address	Instr.Type	Dependency
	lui	0x0000002c l.str
0x00000030	ori	r.str
0x00000038	jal	printf ...

# Enlazado

## Resolver las direcciones

```
00    addiu $29,$29,-24
04    sw     $31, 20($29)
08    sw     $4, 4($29)
0c    sw $5, 8($29)
10    ori $8, $0, 0
14    ori $9, $0, 0
18    slti $1, $8, 11
1c    beq  $1, $0, 3
20    add  $9, $9, $8
24    addi $8, $8, 1
28    bgez $0, -4
```

```
2c    lui $4, 4096
30    ori $4, $4, 1072
34    addu $4, $0, $9
38    jal 812
3c    addu $2, $0, $0
40    lw     $31, 20($29)
44    lw $4, 4($29)
48    lw $5, 8($29)
4c    addiu $29,$29,24
50    jr $31
```

# Enlazado

- ▶ **Generación del fichero ejecutable**
  - ▶ Único segmento de texto
  - ▶ Único segmento de datos
  - ▶ Cabecera con información sobre las secciones