



Tema 3:

# Programación en ensamblador

Fundamentos de computadores II

**José Manuel Mendías Cuadros**

*Dpto. Arquitectura de Computadores y Automática  
Universidad Complutense de Madrid*





# Contenidos

- ✓ Introducción
- ✓ Lenguaje ensamblador
- ✓ Pseudo-instrucciones.
- ✓ Variables y constantes.
- ✓ Expresiones.
- ✓ Organización de código.
- ✓ Funciones.
- ✓ Flujo de desarrollo.

Transparencias basadas en los libros:

- S.L. Harris and D. Harris. *Digital Design and Computer Architecture. RISC-V Edition.*
- D.A. Patterson and J.L. Hennessy. *Computer Organization and Design. RISC-V Edition.*



# Introducción

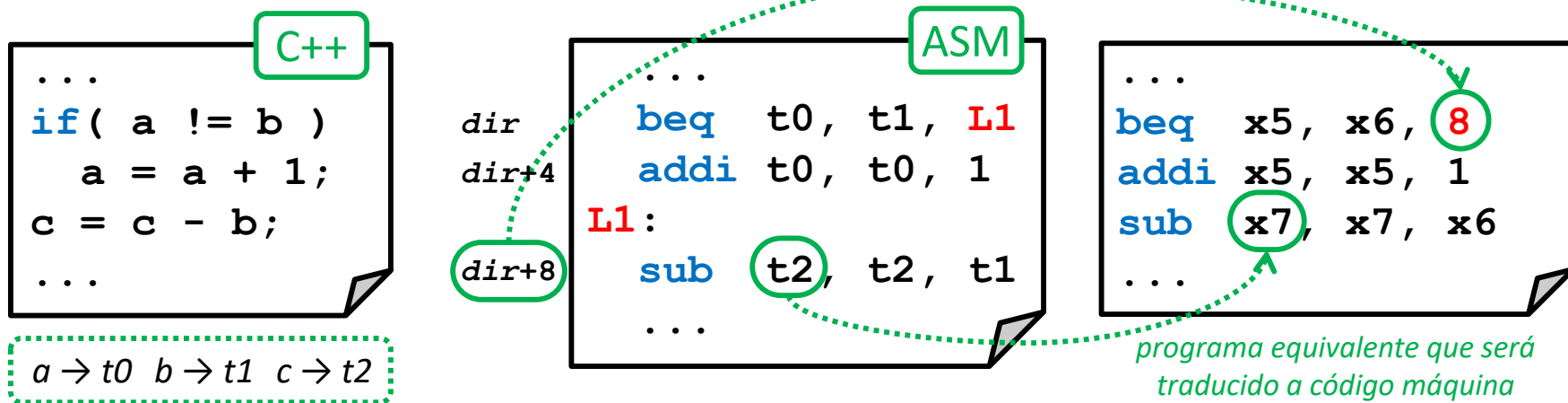
- Los computadores ejecutan código máquina, pero los programadores desarrollan software en lenguajes de alto nivel.
- El **código ensamblador** es el **punto intermedio** entre HW y SW:
  - Los programas en un lenguaje de alto nivel se compilan a ensamblador.
  - Los programas en lenguaje ensamblador se ensamblan a código máquina.
- Un **programa en lenguaje ensamblador** está formado **mayoritariamente** por **instrucciones en ensamblador**
  - Pero también incorpora otros elementos que facilitan la programación: **etiquetas, directivas, comentarios, pseudo-instrucciones**, etc...
- En la actualidad **prácticamente no se programa en ensamblador**, pero **es importante** tener nociones de cómo hacerlo porque ayuda a:
  - Entender la **capa de abstracción** que supone la **arquitectura del procesador**.
  - Comprender cómo los **compiladores traducen a ensamblador** los programas escritos en lenguaje de alto nivel.



# Lenguaje ensamblador

## Elementos de un programa (i)

- Un programa en lenguaje ensamblador:
  - Está compuesto por una **secuencia de instrucciones** que se ubicarán en memoria en el **mismo orden** para ser **ejecutadas en serie**.
    - Normalmente usando alias para referirse a los registros que utiliza.
  - **No indica explícitamente la dirección de memoria** de cada instrucción o dato.
    - Pero instrucciones consecutivas ocuparán direcciones consecutivas.
  - Si es necesario **saltar a una cierta instrucción**, permite **definir una etiqueta** para referirse simbólicamente a su dirección.
    - Las etiquetas liberan al programador del cálculo de los desplazamientos relativos al PC requeridos por las instrucciones de salto.

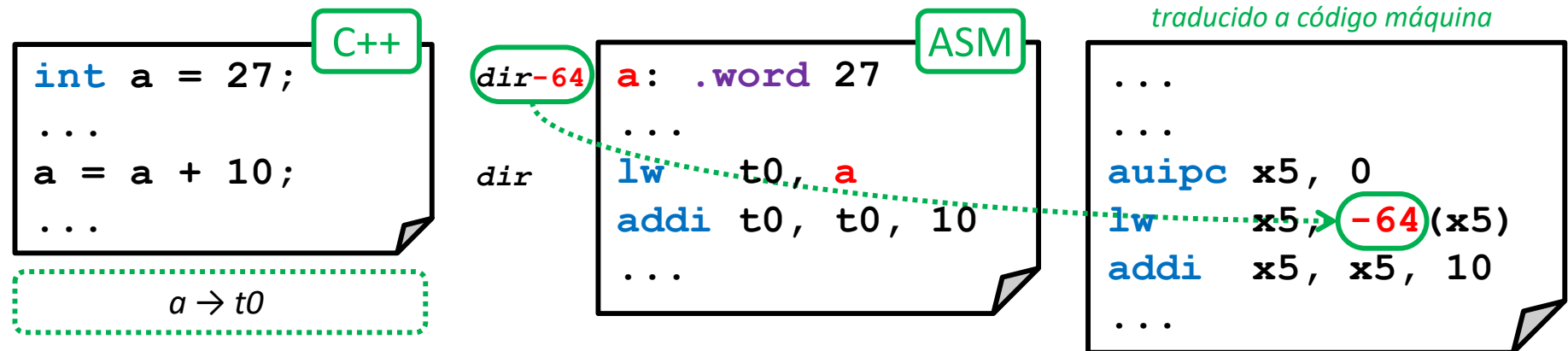




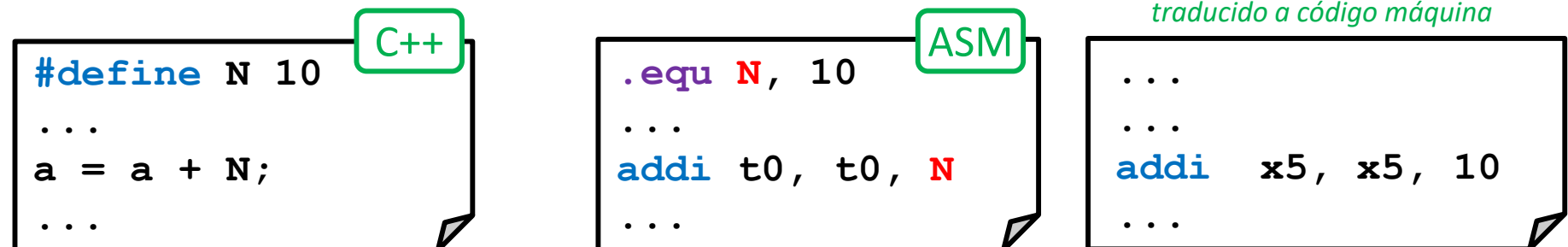
# Lenguaje ensamblador

## Elementos de un programa (ii)

- En un programa en lenguaje ensamblador también pueden definirse:
  - **Etiquetas** para referirse simbólicamente a la **dirección que ocupa un dato**.
    - Liberan al programador de la gestión de direcciones absolutas de 32b.



- **Símbolos** para referirse simbólicamente a **constantes inmediatas**.
  - Facilitan la legibilidad del código.

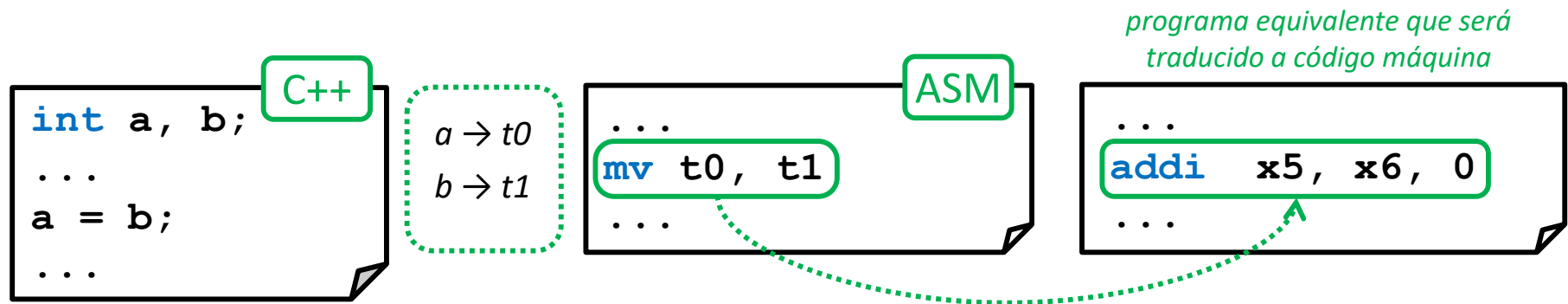




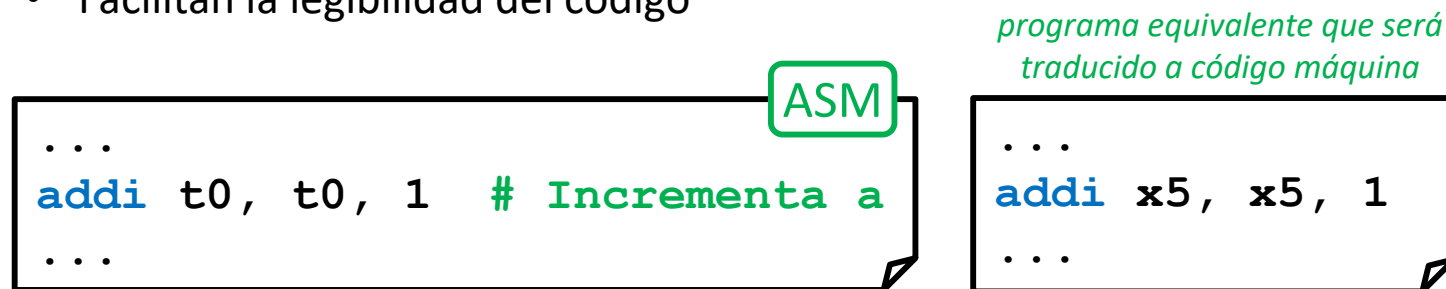
# Lenguaje ensamblador

## Elementos de un programa (iii)

- Además de instrucciones, un programa en ensamblador puede contener:
  - **Pseudo-instrucciones:** son alias de ciertas instrucciones.
    - Facilitan al programador el uso de instrucciones de uso recurrentes.



- **Comentarios:** texto aclarativo que no se traduce a código máquina.
  - Facilitan la legibilidad del código



- **Otras directivas:** que permiten controlar el proceso de ensamblado.
  - Ubicación de código y datos, alineamiento, etc...



# Lenguaje ensamblador

## Elementos de un programa (iv)

- Un **programa en ensamblador** es una **secuencia de líneas**, y cada línea contiene **como máximo uno** de cada uno de los siguientes elementos:
  - **Etiqueta**: referencia simbólica a la **dirección** de una instrucción o dato.
    - Toda etiqueta debe comenzar por una letra y terminar con dos puntos.
    - Para referirse a ella no se ponen los dos puntos.
    - Si está sola en la línea se refiere a la dirección ocupada por la primera instrucción o dato que le siga.
  - **Instrucción en ensamblador**: formada por una **instrucción y sus operandos**.
    - Los operandos pueden ser explícitos o simbólicos.
    - En lugar de una instrucción, puede ser haber una pseudo-instrucción.
  - **Directiva**: **indicación auxiliar** utilizada durante el ensamblado.
    - Toda directiva comienza por un punto.
  - **Comentario**: **texto libre** usado por el programador
    - Los comentarios pueden estar al final de una línea u ocupar la línea completa.
    - Comienzan por # pero también puede usarse // e incluso /\* \*/



# Lenguaje ensamblador

## Secciones

- Un programa en ensamblador se divide en **secciones**.
- Una **sección** representa una **región de memoria contigua**, en donde se ubicarán un conjunto de datos/instrucciones con un mismo propósito.
  - Instrucciones/datos consecutivos dentro de una sección tendrán direcciones consecutivas en memoria.
  - Durante el proceso de enlazado cada sección podrá ser ubicada en un lugar de la memoria distinto.
- En un programa en ensamblador existen 3 secciones:
  - **text**: contiene las instrucciones que forman el programa.
  - **data**: contiene las constantes y variables globales con valor inicial.
  - **bss**: contiene variables globales sin valor inicial.
  - Pueden declararse otras usando una directiva especial (**.section**)





# Lenguaje ensamblador

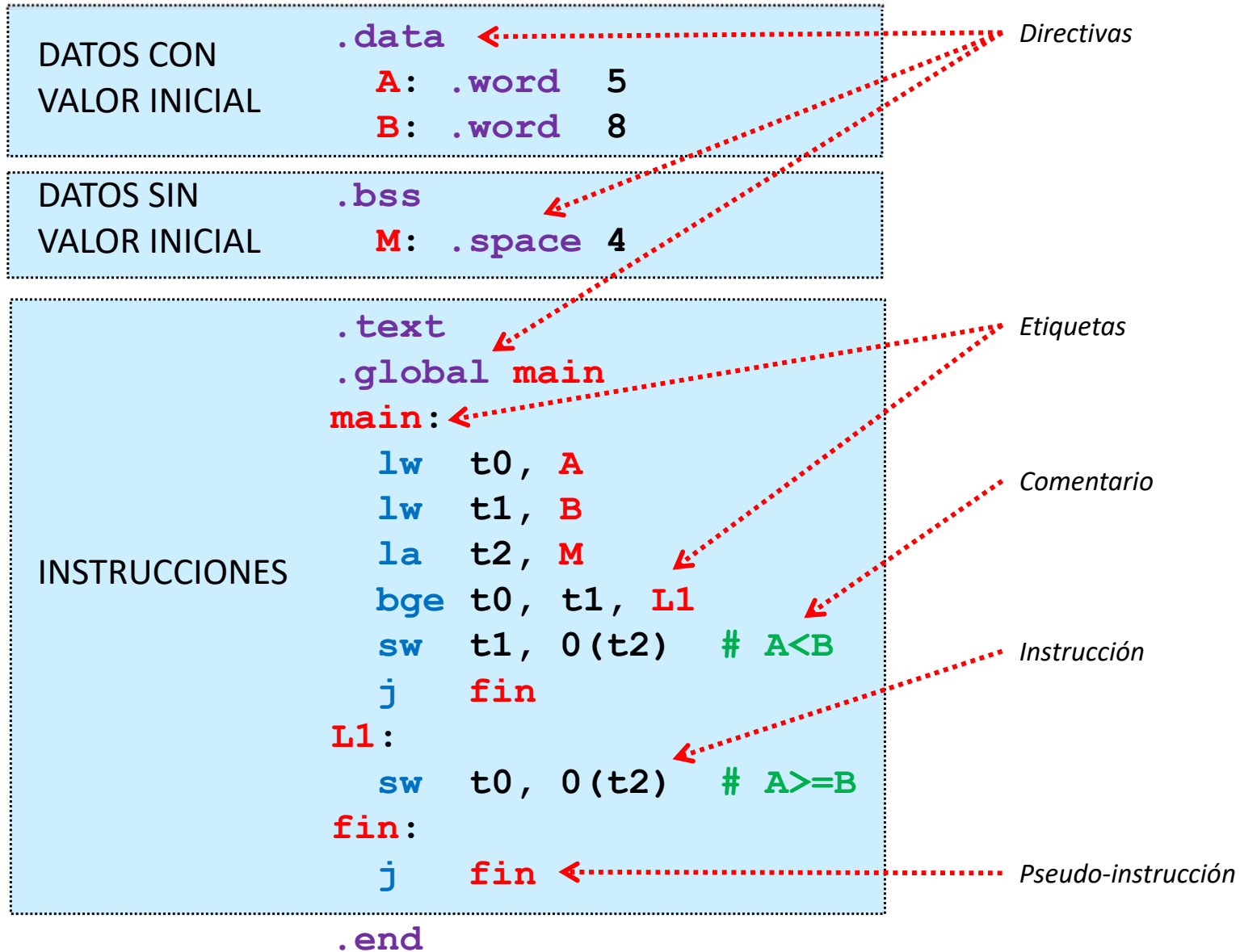
## Directivas

Directiva	Descripción
<code>.text</code>	Declara el comienzo de la sección de instrucciones
<code>.data</code>	Declara el comienzo de la sección de variables globales <b>con</b> valor inicial
<code>.bss</code>	Declara el comienzo de la sección de variables globales <b>sin</b> valor inicial
<code>.word</code> $w_1, \dots, w_n$	Reserva espacio en memoria para $n$ palabras inicializadas a $w_1 \dots w_n$
<code>.half</code> $h_1, \dots, h_n$	Reserva espacio en memoria para $n$ medias palabras inicializadas a $h_1 \dots h_n$
<code>.byte</code> $b_1, \dots, b_n$	Reserva espacio en memoria para $n$ bytes inicializados a $b_1 \dots b_n$
<code>.zero</code> $n$	Reserva espacio en memoria para $n$ bytes inicializados a 0
<code>.space</code> $n$	Reserva espacio en memoria para $n$ bytes sin inicializar
<code>.string</code> " <i>str</i> "	Reserva espacio en memoria inicializado con la cadena " <i>str</i> "
<code>.align</code> $n$	Alinea los datos/instrucciones a direcciones múltiplo de $2^n$
<code>.equ</code> <i>sym</i> , <i>val</i>	Define una constante simbólica llamada <i>sym</i> de valor <i>val</i>
<code>.global</code> <i>sym</i>	Hace visible la etiqueta <i>sym</i> fuera del archivo que la contiene (global)
<code>.extern</code> <i>sym</i>	Indica que un símbolo está definido en otro archivo del proyecto
<code>.end</code>	Declara el final del programa ensamblador



# Lenguaje ensamblador

## Ejemplo (i)





# Lenguaje ensamblador

## Ejemplo (ii)

```

DATOS CON VALOR INICIAL
    .data ←
    A: .word 5
    B: .word 8
  
```

Indica el inicio de la sección de datos de entrada

Datos de entrada inicializados de tamaño palabra

```

DATOS SIN VALOR INICIAL
    .bss ←
    M: .space 4 ←
  
```

Indica el inicio de la sección de datos de salida

Datos de salida de tamaño palabra

```

INSTRUCCIONES
    .text ←
    .global main ←
    main:
        lw t0, A
        lw t1, B
        la t2, M
        bge t0, t1, L1
        sw t1, 0(t2) # A<B
        j fin
    L1:
        sw t0, 0(t2) # A>=B
    fin:
        j fin
  
```

Indica el inicio de la sección de código

Hace visible esta etiqueta fuera de este archivo, en particular para que el simulador pueda conocer la dirección de inicio del programa

`.end` ← Indica el final del programa en ensamblador



# Lenguaje ensamblador

## Ejemplo (iii)

```
DATOS CON VALOR INICIAL      .data
                              A: .word 5
                              B: .word 8

DATOS SIN VALOR INICIAL     .bss
                              M: .space 4

INSTRUCCIONES                .text
                              .global main
                              main:
                              lw t0, A
                              lw t1, B
                              la t2, M
                              bge t0, t1, L1
                              sw t1, 0(t2)
                              j fin
                              L1:
                              sw t0, 0(t2)
                              fin:
                              j fin
                              .end
```

- Carga en t0 el valor contenido en la dirección A
- Carga en t1 el valor contenido en la dirección B
- Carga en t2 la dirección de M
- Compara los valores cargados
- Almacena en la dirección de M el valor de t1 (B)
- Salta a la última instrucción del programa
- Almacena en la dirección de M el valor de t0 (A)
- Indefinidamente ejecuta esta instrucción



# Pseudo-instrucciones

- Las **pseudo-instrucciones** son **alias** de casos muy habituales de uso de ciertas instrucciones.
  - Realizan una operación específica con **nemotécnico** y **operandos** propios.
  - Durante el ensamblado son **traducidas a instrucciones reales** de comportamiento equivalente.
  - Permiten **enriquecer el lenguaje** sin añadir complejidad al HW.
- Por ejemplo, en ensamblador es muy común querer **copiar el contenido de un registro en otro**:
  - Por ello, en RISC-V está definida la pseudo-instrucción de 2 operandos **mv**:  

---

```
mv rd, rs1          rd ← rs1
```

---
  - Durante el ensamblado, se traduce a la siguiente instrucción máquina:  

---

```
addi rd, rs1, 0     rd ← rs1 + 0
```

---
  - El programador puede usar indistintamente una opción u otra.



# Pseudo-instrucciones

## Aritmético-lógicas (i)

- En ensamblador, son muy comunes las **comparaciones con 0**.

Instrucción	Operación	Traducción	Descripción
<code>seqz rd, rs1</code>	$rd \leftarrow \text{if}(rs1 = 0) \text{ then } (1) \text{ else } (0)$	<code>sltiu rd, rs1, 1</code>	set if <b>e</b> qual to <b>z</b> ero "igual que" 0
<code>snez rd, rs1</code>	$rd \leftarrow \text{if}(rs1 \neq 0) \text{ then } (1) \text{ else } (0)$	<code>sltu rd, x0, rs1</code>	set if <b>n</b> ot <b>e</b> qual to <b>z</b> ero "distinto que" 0
<code>sltz rd, rs1</code>	$rd \leftarrow \text{if}(rs1 < 0) \text{ then } (1) \text{ else } (0)$	<code>slt rd, rs1, x0</code>	set if <b>l</b> ess <b>t</b> han to <b>z</b> ero "menor que" 0
<code>sgtz rd, rs1</code>	$rd \leftarrow \text{if}(rs1 > 0) \text{ then } (1) \text{ else } (0)$	<code>slt rd, x0, rs1</code>	set if <b>g</b> reater <b>t</b> han to <b>z</b> ero "mayor que" 0

- También es muy habitual **cambiar el signo** de un operando.

Instrucción	Operación	Traducción	Descripción
<code>neg rd, rs1</code>	$rd \leftarrow -rs1$	<code>sub rd, x0, rs1</code>	<b>n</b> egate opuesto aritmético



# Pseudo-instrucciones

## Aritmético-lógicas (ii)

- Así como **negar bit a bit** un operando.

Instrucción	Operación	Traducción	Descripción
<code>not rd, rs1</code>	$rd \leftarrow \sim rs1$	<code>xori rd, rs1, -1</code>	<b>not</b> "no lógica" bit a bit

- También es útil **renombrar las instrucciones** que operan con inmediato.
  - Evita que el programador tenga que usar nemotécnicos diferentes.

Instrucción	Operación	Traducción
<code>add rd, rs1, imm<sub>12b</sub></code>	$rd \leftarrow rs1 + sExt(imm)$	<code>addi rd, rs1, imm</code>
<code>slt rd, rs1, imm<sub>12b</sub></code>	$rd \leftarrow if ( rs1 <_s sExt(imm) )$ $then ( 1 ) else ( 0 )$	<code>slti rd, rs1, imm</code>
<code>sltu rd, rs1, imm<sub>12b</sub></code>	$rd \leftarrow if ( rs1 <_u sExt(imm) )$ $then ( 1 ) else ( 0 )$	<code>sltiu rd, rs1, imm</code>
<code>and rd, rs1, imm<sub>12b</sub></code>	$rd \leftarrow rs1 \& sExt(imm)$	<code>andi rd, rs1, imm</code>
<code>or rd, rs1, imm<sub>12b</sub></code>	$rd \leftarrow rs1   sExt(imm)$	<code>ori rd, rs1, imm</code>
<code>xor rd, rs1, imm<sub>12b</sub></code>	$rd \leftarrow rs1 \wedge sExt(imm)$	<code>xori rd, rs1, imm</code>



# Pseudo-instrucciones

## De desplazamiento

- Ídem para operaciones de desplazamiento con operando inmediato:

Instrucción	Operación	
<code>sll rd, rs1, imm<sub>5b</sub></code>	$rd \leftarrow rs1 \ll imm$	<code>slli rd, rs1, imm</code>
<code>srl rd, rs1, imm<sub>5b</sub></code>	$rd \leftarrow rs1 \gg imm$	<code>srlr rd, rs1, imm</code>
<code>sra rd, rs1, imm<sub>5b</sub></code>	$rd \leftarrow rs1 \ggg imm$	<code>srai rd, rs1, imm</code>





# Pseudo-instrucciones

## De transferencia de datos (i)

- Es común **cargar/almacenar datos con dirección absoluta** conocida
  - Lo habitual es usar direccionamiento relativo a PC para hacerlo.
  - Estas pseudo-instrucciones liberan al programador de hacer los cálculos.

Instrucción	Operación	Traducción
<b>lb</b> <i>rd</i> , <i>imm</i> <sub>32b</sub>	$rd \leftarrow \text{sExt}(\text{Mem}[\text{PC} + \text{imm}]_{7:0})$	<b>auipc</b> <i>rd</i> , <i>imm</i> <sub>31:12</sub> <sup>*</sup> <b>lb</b> <i>rd</i> , <i>imm</i> <sub>11:0</sub> ( <i>rd</i> )
<b>lh</b> <i>rd</i> , <i>imm</i> <sub>32b</sub>	$rd \leftarrow \text{sExt}(\text{Mem}[\text{PC} + \text{imm}]_{15:0})$	<b>auipc</b> <i>rd</i> , <i>imm</i> <sub>31:12</sub> <sup>*</sup> <b>lh</b> <i>rd</i> , <i>imm</i> <sub>11:0</sub> ( <i>rd</i> )
<b>lw</b> <i>rd</i> , <i>imm</i> <sub>32b</sub>	$rd \leftarrow \text{sExt}(\text{Mem}[\text{PC} + \text{imm}]_{31:0})$	<b>auipc</b> <i>rd</i> , <i>imm</i> <sub>31:12</sub> <sup>*</sup> <b>lw</b> <i>rd</i> , <i>imm</i> <sub>11:0</sub> ( <i>rd</i> )
<b>sb</b> <i>rs2</i> , <i>imm</i> <sub>32b</sub> , <i>rs1</i>	$\text{Mem}[\text{PC} + \text{imm}]_{7:0} \leftarrow rs2_{7:0}$	<b>auipc</b> <i>rs1</i> , <i>imm</i> <sub>31:12</sub> <sup>*</sup> <b>sb</b> <i>rs2</i> , <i>imm</i> <sub>11:0</sub> ( <i>rs1</i> )
<b>sh</b> <i>rs2</i> , <i>imm</i> <sub>32b</sub> , <i>rs1</i>	$\text{Mem}[\text{PC} + \text{imm}]_{15:0} \leftarrow rs2_{15:0}$	<b>auipc</b> <i>rs1</i> , <i>imm</i> <sub>31:12</sub> <sup>*</sup> <b>sh</b> <i>rs2</i> , <i>imm</i> <sub>11:0</sub> ( <i>rs1</i> )
<b>sw</b> <i>rs2</i> , <i>imm</i> <sub>32b</sub> , <i>rs1</i>	$\text{Mem}[\text{PC} + \text{imm}]_{31:0} \leftarrow rs2_{31:0}$	<b>auipc</b> <i>rs1</i> , <i>imm</i> <sub>31:12</sub> <sup>*</sup> <b>sw</b> <i>rs2</i> , <i>imm</i> <sub>11:0</sub> ( <i>rs1</i> )

(\*) Si *imm*<sub>11</sub> vale 1, incrementará en 1 el valor de *imm*<sub>31:12</sub> usado



# Pseudo-instrucciones

## De transferencia de datos (ii)

- Así como **copiar un dato** de un registro a otro.

Instrucción	Operación	Traducción	Descripción
<code>mv rd, rs1</code>	$rd \leftarrow rs1$	<code>addi rd, rs1, 0</code>	<b>move</b> copia registro

- Cargar una constante** en un registro.

Instrucción	Operación	Traducción	Descripción
<code>li rd, imm<sub>12b</sub></code>	$rd \leftarrow sExt(imm)$	<code>addi rd, x0, imm</code>	load <b>i</b> nmediate copia inmediato de 12 bits
<code>li rd, imm<sub>32b</sub></code>	$rd \leftarrow imm$	<code>lui rd, imm<sub>31:12</sub>*</code> <code>addi rd, rd, imm<sub>11:0</sub></code>	load <b>i</b> nmediate copia inmediato de 32 bits

- Cargar una dirección** en un dato ubicado en memoria.

Instrucción	Operación	Traducción	Descripción
<code>la rd, imm<sub>32b</sub></code>	$rd \leftarrow PC + imm$	<code>auipc rd, imm<sub>31:12</sub>*</code> <code>addi rd, rd, imm<sub>11:0</sub></code>	load <b>a</b> ddress copia dirección de 32 bits



# Pseudo-instrucciones

## De salto condicional (i)

- Es común hacer **cualquier tipo de comparación** en saltos condicionales:
  - En el **repertorio real** solo existen comparaciones de tipo: =, ≠, < y ≥
  - Para hacer otras, el programador debe **cambiar el orden de los operandos**.
  - Se añaden pseudo-instrucciones para comparaciones de tipo: ≤ o >

Instrucción	Operación	Traducción	Descripción
<b>ble</b> <i>rs1, rs2, imm<sub>13b</sub></i>	<i>if ( rs1 ≤<sub>s</sub> rs2 ) then</i> ( PC ← PC + sExt(imm <sub>12:1</sub> << 1) )	<b>bge</b> <i>rs2, rs1, imm</i>	<b>b</b> branch if <b>l</b> ess than or <b>e</b> qual salta si “menor o igual que” con signo
<b>bgt</b> <i>rs1, rs2, imm<sub>13b</sub></i>	<i>if ( rs1 &gt;<sub>s</sub> rs2 ) then</i> ( PC ← PC + sExt(imm <sub>12:1</sub> << 1) )	<b>blt</b> <i>rs2, rs1, imm</i>	<b>b</b> branch if <b>g</b> reater <b>t</b> han salta si “mayor que” con signo
<b>bleu</b> <i>rs1, rs2, imm<sub>13b</sub></i>	<i>if ( rs1 ≤<sub>u</sub> rs2 ) then</i> ( PC ← PC + sExt(imm <sub>12:1</sub> << 1) )	<b>bgeu</b> <i>rs2, rs1, imm</i>	<b>b</b> branch if <b>l</b> ess than or <b>e</b> qual <b>u</b> nsigned salta si “menor o igual que” sin signo
<b>bgtu</b> <i>rs1, rs2, imm<sub>13b</sub></i>	<i>if ( rs1 &gt;<sub>u</sub> rs2 ) then</i> ( PC ← PC + sExt(imm <sub>12:1</sub> << 1) )	<b>bltu</b> <i>rs2, rs1, imm</i>	<b>b</b> branch if <b>g</b> reater <b>t</b> han <b>u</b> nsigned salta si “mayor que” sin signo



# Pseudo-instrucciones

## De salto condicional (ii)

- En particular, las **comparaciones con 0** en saltos son las más comunes

Instrucción	Operación	Traducción	Descripción
<b>beqz</b> <i>rs1</i> , <i>imm</i> <sub>13b</sub>	<i>if</i> ( <i>rs1</i> = 0 ) <i>then</i> ( $PC \leftarrow PC + sExt(imm_{12:1} \ll 1)$ )	<b>beq</b> <i>rs1</i> , <b>x0</b> , <i>imm</i>	branch if <b>e</b> qual to <b>z</b> ero salta si "igual que" 0
<b>bnez</b> <i>rs1</i> , <i>imm</i> <sub>13b</sub>	<i>if</i> ( <i>rs1</i> ≠ 0 ) <i>then</i> ( $PC \leftarrow PC + sExt(imm_{12:1} \ll 1)$ )	<b>bne</b> <i>rs1</i> , <b>x0</b> , <i>imm</i>	branch if <b>n</b> ot <b>e</b> qual to <b>z</b> ero salta si "distinto que" 0
<b>bltz</b> <i>rs1</i> , <i>imm</i> <sub>13b</sub>	<i>if</i> ( <i>rs1</i> < 0 ) <i>then</i> ( $PC \leftarrow PC + sExt(imm_{12:1} \ll 1)$ )	<b>blt</b> <i>rs1</i> , <b>x0</b> , <i>imm</i>	branch if <b>l</b> ess <b>t</b> han to <b>z</b> ero salta si "menor que" 0
<b>bgez</b> <i>rs1</i> , <i>imm</i> <sub>13b</sub>	<i>if</i> ( <i>rs1</i> ≥ 0 ) <i>then</i> ( $PC \leftarrow PC + sExt(imm_{12:1} \ll 1)$ )	<b>bge</b> <i>rs1</i> , <b>x0</b> , <i>imm</i>	branch if <b>g</b> reater than or <b>e</b> qual to <b>z</b> ero salta si "mayor o igual que" 0
<b>blez</b> <i>rs1</i> , <i>imm</i> <sub>13b</sub>	<i>if</i> ( <i>rs1</i> ≤ 0 ) <i>then</i> ( $PC \leftarrow PC + sExt(imm_{12:1} \ll 1)$ )	<b>bge</b> <b>zero</b> , <i>rs1</i> , <i>imm</i>	branch if <b>l</b> ess than or <b>e</b> qual to <b>z</b> ero salta si "menor o igual que" 0
<b>bgtz</b> <i>rs1</i> , <i>imm</i> <sub>13b</sub>	<i>if</i> ( <i>rs1</i> > 0 ) <i>then</i> ( $PC \leftarrow PC + sExt(imm_{12:1} \ll 1)$ )	<b>blt</b> <b>zero</b> , <i>rs1</i> , <i>imm</i>	branch if <b>g</b> reater <b>t</b> han to <b>z</b> ero salta si "mayor que" 0



# Pseudo-instrucciones

## De salto a función (ii)

- Por convenio, el registro **ra** (alias de **x1**) se suele usar como registro en donde almacenar direcciones de retorno en saltos a función.
  - Se añaden pseudo-instrucciones que lo usan implícitamente.

Instrucción	Operación	Traducción	Descripción
<code>jalr rs1</code>	$PC \leftarrow rs1$ $ra \leftarrow PC+4$	<code>jalr ra, rs1, 0</code>	<b>jump and link register</b> salto a función con dirección relativa a registro base
<code>jal imm<sub>21b</sub></code>	$PC \leftarrow PC + sExt(imm_{20:1} \ll 1)$ $ra \leftarrow PC+4$	<code>jal ra, imm</code>	<b>jump and link</b> salto a función con dirección relativa a PC (cercana)
<code>call imm<sub>21b</sub></code>	$PC \leftarrow PC + sExt(imm_{20:1} \ll 1)$ $ra \leftarrow PC+4$	<code>jal ra, imm</code>	<b>call</b> salto a función con dirección relativa a PC (cercana)
<code>call imm<sub>32b</sub></code>	$PC \leftarrow PC + imm$ $ra \leftarrow PC+4$	<code>auipc ra, imm<sub>31:12</sub>*</code> <code>jalr ra, ra, imm<sub>11:0</sub></code>	<b>call</b> salto a función con dirección relativa a PC (lejana)
<code>ret</code>	$PC \leftarrow ra$	<code>jalr x0, ra, 0</code>	<b>return</b> retorno de función



# Pseudo-instrucciones

## Otras

- El **salto incondicional** es una funcionalidad muy útil que no existe como tal en el repertorio de instrucciones del RISC-V.

Instrucción	Operación	Traducción	Descripción
<code>j</code> <i>imm</i> <sub>21b</sub>	$PC \leftarrow PC + sExt(imm_{20:1} \ll 1)$	<code>jal</code> <code>x0</code> , <i>imm</i>	<b>j</b> ump salto incondicional (inmediato)
<code>jr</code> <i>rs1</i>	$PC \leftarrow rs1$	<code>jalr</code> <code>x0</code> , <i>rs1</i> , 0	<b>j</b> ump <b>r</b> egister salto incondicional (con registro)

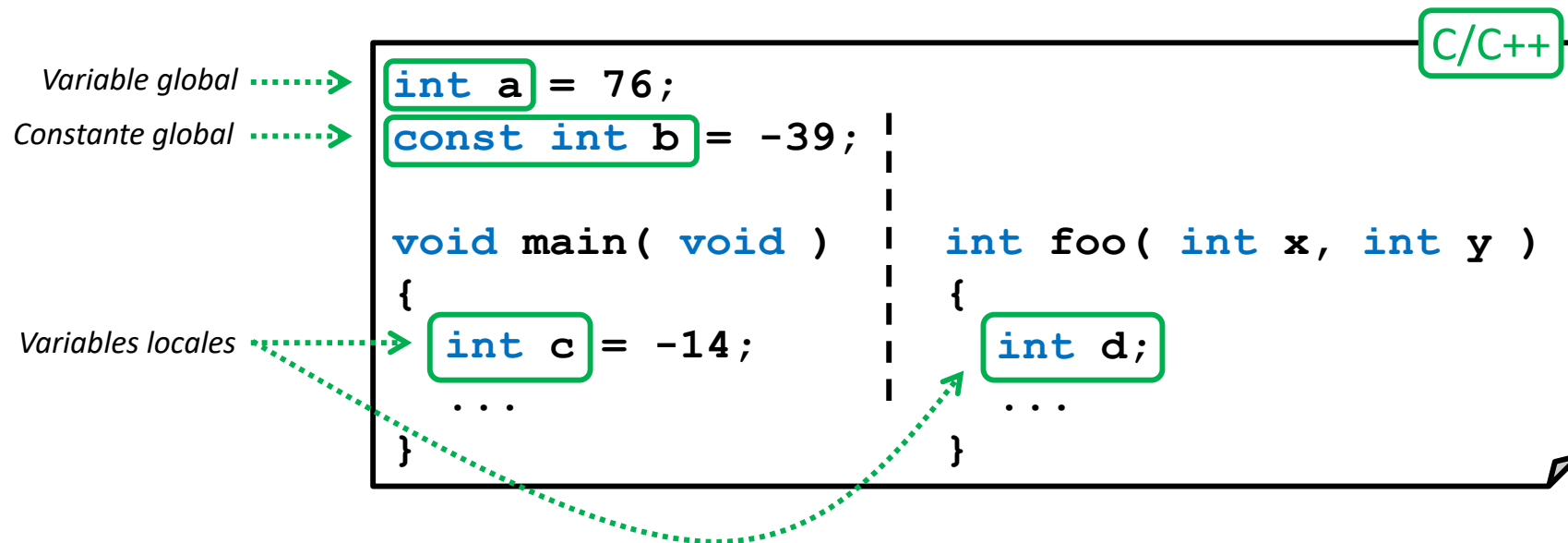
- Asimismo, en ocasiones, es útil **no hacer nada**.

Instrucción	Operación	Traducción	Descripción
<code>nop</code>	---	<code>addi</code> <code>x0</code> , <code>x0</code> , 0	<b>n</b> o <b>o</b> peration Instrucción sin efecto



# Variables y constantes

- En C/C++ una variable tiene un tipo y puede ser global o local.
  - Una **variable global** se declara fuera de las funciones
    - Es visible desde cualquier punto del programa.
    - Persiste durante toda la ejecución del programa (estática).
  - Una **variable local** se declara dentro de una función.
    - Es visible solo dentro del cuerpo de la función en donde se declara.
    - Por defecto, solo persiste durante la ejecución de la función (automáticas).
    - Los parámetros formales de una función se comportan como variables locales.





# Variables y constantes

- En ensamblador **no existen variables** como tales,
  - **Existen datos** que residen en **memoria**, en **registro** o **alternando** entre ambas ubicaciones.
    - Para operar con ellos, **siempre deben estar en registros** porque en el ensamblador de RISC-V **no existen instrucciones con operandos en memoria**.
  - Del mismo modo, la **dirección de memoria** o **registro** donde reside el dato **puede cambiar** a lo largo de la ejecución del programa.
    - El **programador debe llevar la traza del lugar** en donde se encuentra en cada momento el dato para operar con él.
- En ensamblador tampoco hace **distinción entre variables y constantes**
  - Si el dato **cambia durante la ejecución** del programa diremos que es **variable**.
  - Si no cambia, diremos que es **constante**.
- Además, en ensamblador las **constantes** pueden residir en **memoria** o en la **propia instrucción** (como operandos inmediatos).





# Variables y constantes

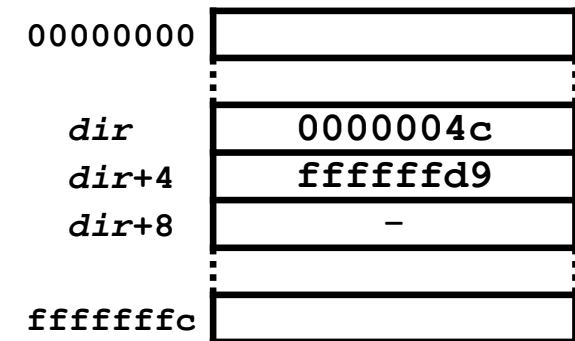
- En el caso de **variables/constantes globales**, se usan **etiquetas** para evitar el uso de direcciones explícitas en el código ensamblador.
  - Estas etiquetas juegan en ensamblador **el papel del nombre de la variable**.

C/C++

```
int a = 76;
const int b = -39;
int c;
...
```

ASM

```
a: .word 76
b: .word -39
c: .space 4
...
```



Memoria

- En ensamblador un **valor constante** puede expresarse indistintamente en
  - **Decimal**, tal cual: **109**
  - **Hexadecimal**, **anteponiendo 0x** a la secuencia de dígitos: **0x6d**
  - **Binario**, **anteponiendo 0b** a la secuencia de dígitos: **0b1101101**



# VARIABLES Y CONSTANTES

- Simplificando, los **datos de entrada y salida** de un programa podremos tratarlos como **variables globales**.
  - Inicialmente los **datos de entrada** de un programa **residen en memoria**.
    - Por haber sido **recibidos desde un periférico**.
    - Por tener un **valor inicial** (fijado por el propio programa o calculado por otro ejecutado con anterioridad).
  - Los **datos de salida** del programa deberán almacenarse también en **memoria**.
    - Para ser **transmitidos hacia un periférico** o ser usados con posterioridad.
  - Sus direcciones **serán fijas y conocidas** por el programador.
    - Por ello, los datos de entrada y salida **estarán identificados por una etiqueta**.
- Como el **número de registros es limitado**:
  - El resto de datos del programa mayoritariamente residen en memoria.
- Pero, como el **acceso a un registro es mucho más rápido que a memoria**.
  - Los datos deben mantenerse el mayor tiempo posible en registros.
  - Como no es posible mantenerlos todos, se mantienen los más usados.



# Variables y constantes

- Los datos que residen en memoria:
  - Deben cargarse en registros para operar con ellos.
  - Una vez calculado el resultado, se almacena en memoria.

**C/C++**

```
int a = 5;
...
a = a + 1;
...
```

$&a \rightarrow t0$   $a \rightarrow t1$

**ASM**

```
a: .word 5
...
la    t0, a
lw    t1, 0(t0)
addi  t1, t1, 1
sw    t1, 0(t0)
...
```

**ASM**

```
a: .word 5
...
lw    t1, a
addi  t1, t1, 1
sw    t1, a, t0
...
```

*programas equivalentes sin pseudo-instrucciones que serán traducidos a código máquina*

*Las constantes inmediatas serán calculadas durante el ensamblado*

```
a: .word 5
...
auipc t0, ...
addi  t0, t0, ...
lw    t1, 0(t0)
addi  t1, t1, 1
sw    t1, 0(t0)
...
```

```
a: .word 5
...
auipc t1, ...
lw    t1, ... (t1)
addi  t1, t1, 1
auipc t0, ...
sw    t1, ... (t0)
...
```



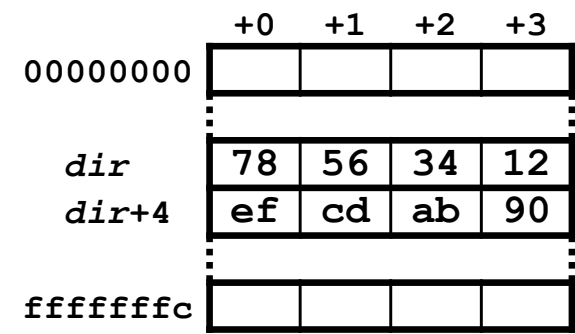
# VARIABLES Y CONSTANTES

- Al igual que las instrucciones, los **datos se ubican en memoria en el mismo orden** en que aparecen en el programa en ensamblador.
  - Al cargar (durante la ejecución) datos de distinto tamaño ubicados consecutivamente pueden producirse **errores de alineamiento**.
  - Para que queden **correctamente alineados** se usa la directiva **.align**

ASM

```

a: .word 0x12345678
b: .word 0x90abcdef
...
  
```



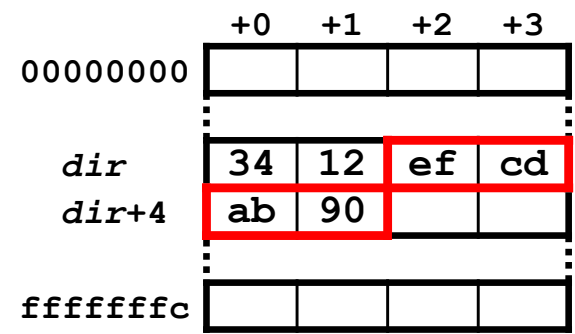
Memoria

INCORRECTO

ASM

```

a: .half 0x1234
b: .word 0x90abcdef
...
  
```



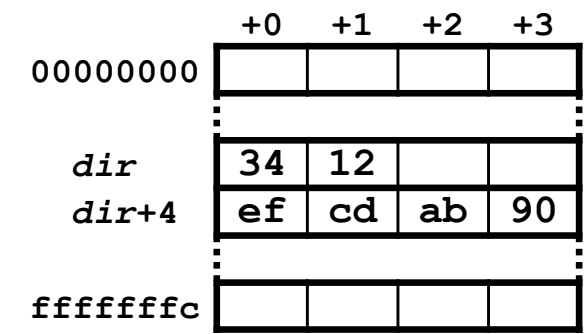
Memoria

CORRECTO

ASM

```

a: .half 0x1234
   .align 2
b: .word 0x90abcdef
...
  
```



Memoria



# Variables y constantes

## Tipos (i)

- En ensamblador las **variables tampoco tienen tipo** explícito.
  - Un **dato tiene cierta anchura en bytes** sin referencia explícita a su codificación.
  - El **programador debe mantener la coherencia** entre la codificación del dato y las instrucciones que usa para operar con él.
- La **equivalencia entre tipos** en C/C++ y anchuras en ensamblador es:

Tipo C/C++	Anchura	Declaración	Carga
<code>[signed] char</code>	8b = 1B	<code>.byte / .space 1</code>	<code>lb</code>
<code>unsigned char</code>	8b = 1B	<code>.byte / .space 1</code>	<code>lbu</code>
<code>[signed] short [int]</code>	16b = 2B	<code>.half / .space 2</code>	<code>lh</code>
<code>unsigned short [int]</code>	16b = 2B	<code>.half / .space 2</code>	<code>lhu</code>
<code>[signed] int</code>	32b = 4B	<code>.word / .space 4</code>	<code>lw</code>
<code>unsigned int</code>	32b = 4B	<code>.word / .space 4</code>	<code>lw</code>
puntero (dirección)	32b = 4B	<code>.word / .space 4</code>	<code>lw</code>



# Variables y constantes

## Tipos (ii)

- La **instrucción de carga** a usar es diferente según la anchura de los datos y de si estos tienen o no signo.

```
unsigned char a = 5;  
...  
a = a + 1;  
...
```

```
short a = 5;  
...  
a = a + 1;  
...
```

```
int a = 5;  
...  
a = a + 1;  
...
```

$&a \rightarrow t0$   
 $a \rightarrow t1$

```
a: .byte 5  
...  
la t0, a  
lbu t1, 0(t0)  
addi t1, t1, 1  
sb t1, 0(t0)  
...
```

```
a: .half 5  
...  
la t0, a  
lh t1, 0(t0)  
addi t1, t1, 1  
sh t1, 0(t0)  
...
```

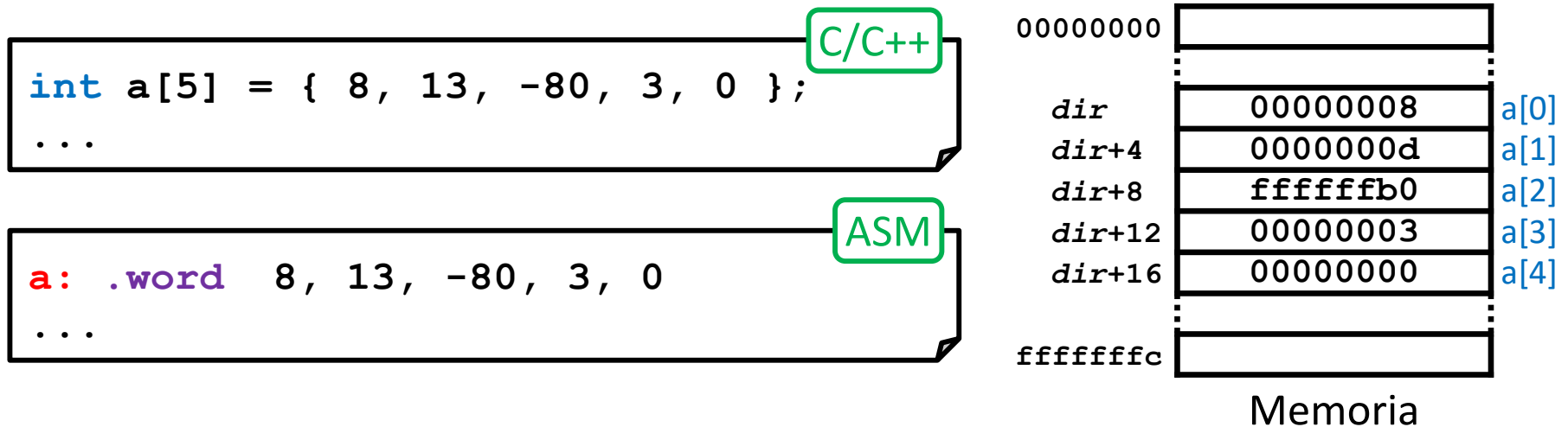
```
a: .word 5  
...  
la t0, a  
lw t1, 0(t0)  
addi t1, t1, 1  
sw t1, 0(t0)  
...
```



# Variables y constantes

## Arrays (i)

- Un **array** es una **colección de datos de la misma anchura** ubicados en **direcciones consecutivas** de memoria en orden creciente de índice.
  - El **índice** indica la **posición relativa del dato** respecto del primero.



- Para **acceder a un elemento del array** hay que calcular su dirección:
  - Es la suma de la **dirección base** del array y un **desplazamiento**
    - La dirección base del array es la dirección de su primera componente.
  - El **desplazamiento** en bytes se calcula:

$$\text{desplazamiento (bytes)} = \text{índice} \times \text{tamaño del dato (bytes)}$$



# Variables y constantes

## Arrays (ii)

- El **desplazamiento** lo calcula el **programador** si el **índice es constante**.
- Si el **índice es variable**, lo debe calcular el **programa**.

```

C/C++
int a[5];
...
a[0] = a[1] + a[2];
...

```

```

C/C++
int a[5], i;
...
a[i] = a[i] + 1;
...

```

$a \equiv \&a[0] \rightarrow t1$   
 $i \rightarrow t1$   
 $a[i] \rightarrow t2$

```

ASM
a: .space 20
...
la    t0, a
Carga a[1] -> lw    t1, 4(t0)
Carga a[2] -> lw    t2, 8(t0)
add   t1, t1, t2
Almacena a[0] -> sw   t1, 0(t0)
...

```

```

ASM
a: .space 20
...
la    t0, a
slli  t1, t1, 2
add   t0, t0, t1
lw    t2, 0(t0)
addi  t2, t2, 1
sw    t2, 0(t0)
...

```

Carga la dirección base del array  
Calcula el desplazamiento  $i*4$   
Suma base y desplazamiento  
Carga  $a[i]$   
Almacena  $a[i]$





# Variables y constantes

## Arrays (iii)

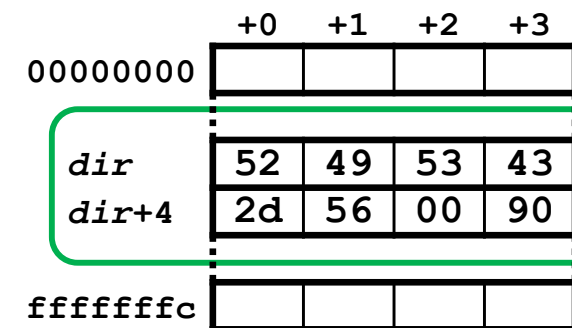
- Las **cadenas de caracteres** son un caso especial de arrays.
  - Almacena ordenadamente caracteres **codificados en ASCII**.
  - Cada carácter ASCII ocupa **un byte**.
  - El array finaliza con el **carácter '\0' (0x0)** que actúa como **centinela de fin de cadena** (permite saber cuando se acaba la cadena).

```
const char a[] = "RISC-V";  
...
```

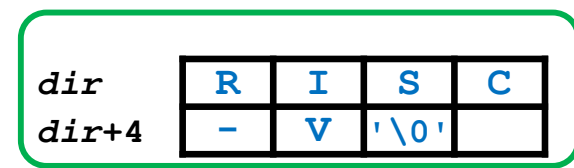
C/C++

```
a: .string "RISC-V"  
...
```

ASM



Memoria





# Variables y constantes

## Estructuras (i)

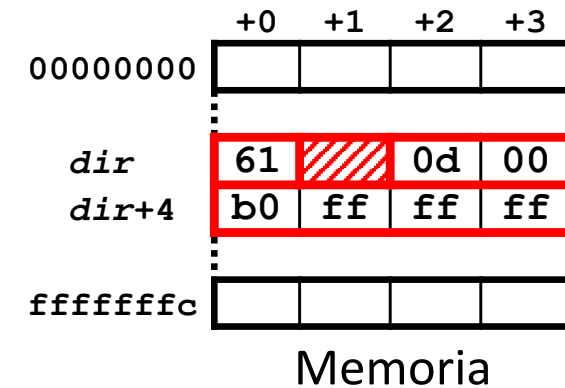
- Una **estructura** es una **colección de datos de la distinta anchura** ubicados en **direcciones consecutivas** de memoria.
  - En C/C++, cada miembro de una estructura se identifica por un nombre.
  - La estructura y sus miembros deben **estar alineados** según su tamaño.

C/C++

```
struct baz {  
    char c;  
    short int si;  
    int i;  
} a = { 'a', 13, -80 };  
...
```

ASM

```
a:  
    .align 2  
    .byte 'a'  
    .align 1  
    .half 13  
    .word -80  
...
```



- Para **acceder a un miembro de la estructura** hay que calcular su dirección:
  - Es la suma de la **dirección base** de la estructura y un **desplazamiento**.
    - La dirección base de la estructura es la dirección de su primera componente.
    - El **desplazamiento** en bytes se calcula en función de su posición relativa.



# Variables y constantes

## Estructuras (ii)

- El **desplazamiento** de cada miembro siempre es constante y lo calcula el **programador**.

C/C++

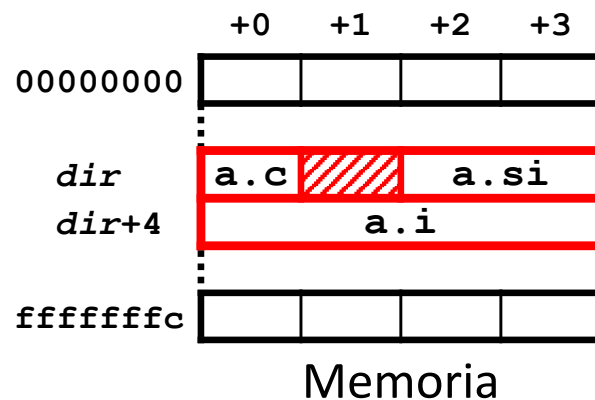
```
struct baz { char c; short int si; int i; } a;  
...  
a.i = a.c + a.si;  
...
```

$&a \rightarrow t0$   
 $a.c \rightarrow t1$   
 $a.si \rightarrow t2$   
 $a.i \rightarrow t3$

ASM

```
a: .space 8  
...  
la t0, a  
lb t1, 0(t0)  
lh t2, 2(t0)  
add t3, t1, t2  
sw t0, 4(t0)  
...
```

Carga la dirección base de la estructura  
Carga a.c  
Carga a.si  
Almacena a.i





# Expresiones

- Las **expresiones simples** en C/C++ requieren una **única instrucción**
  - Usando los registros en donde previamente se han cargado los datos.

<pre>int a, b, c; ... a = b + c; ...</pre>	<pre>a → t0 b → t1 c → t2</pre>	<pre>... add t0, t1, t2 ...</pre>
--	---	---

- Las **expresiones compuestas** requieren **más de una instrucción**.
  - Usando registros adicionales para almacenar los resultados intermedios .

<pre>int a, b, c, d; ... a = (b + c) - d; ...</pre>	<pre>a → t0 b → t1 c → t2 d → t3</pre>	<pre>... add t4, t1, t2 sub t0, t4, t3 ...</pre>
---	--	--



# Expresiones

- Las **constantes explícitas** pueden aparecer de manera **simbólica**:
  - Si la **constante es corta** ( $\leq 12b$ ), es decir, en el rango  $[-2048, +2047]$  puede usarse directamente como operando inmediato.

<div style="text-align: right; border: 1px solid green; border-radius: 5px; padding: 2px; color: green; font-weight: bold;">C/C++</div> <pre>int a, b; ... a = b + 5; ...</pre>	$a \rightarrow t0$ $b \rightarrow t1$	<div style="text-align: right; border: 1px solid green; border-radius: 5px; padding: 2px; color: green; font-weight: bold;">ASM</div> <pre>... addi t0, t1, 5 ...</pre>	<div style="text-align: right; border: 1px solid green; border-radius: 5px; padding: 2px; color: green; font-weight: bold;">ASM</div> <pre>.equ N, 5 ... addi t0, t1, N ...</pre>
---	--	---	---

- Las **constantes largas** ( $> 12b$ ) deben **cargarse previamente en un registro**.
  - Para evitar tener que dividir explícitamente la constante (y corregir la parte alta en caso de que el bit 11 sea 1), debe usarse la **pseudo-instrucción li**

<div style="text-align: right; border: 1px solid green; border-radius: 5px; padding: 2px; color: green; font-weight: bold;">ASM</div> <pre>... li t2, 50000 add t0, t1, t2 ...</pre>	<div style="text-align: right; border: 1px solid green; border-radius: 5px; padding: 2px; color: green; font-weight: bold;">ASM</div> <pre>.equ N, 50000 ... li t2, N add t0, t1, t2 ...</pre>	<p style="color: green; text-align: center;"><i>programa equivalente</i></p> <pre>... lui t2, 0xc addi t2, t2, 0x350 add t0, t1, t2 ...</pre>
--	--	---

$$50000_{10} = 0000c350_{16}$$



# Expresiones

- También **deben cargarse en registros las constantes** cuando se usan instrucciones que no permite operandos inmediatos.
  - El programador **no tiene que preocuparse del tamaño de la constante**, la pseudo-instrucción `li` será traducida convenientemente según su tamaño.

<pre>int a, b; ... a = b * 27; ...</pre>	<pre>a → t0 b → t1</pre>	<pre>... li    t2, 27 mul   t0, t1, t2 ...</pre>	<p><i>programa equivalente</i></p> <pre>... addi  t2, x0, 27 mul   t0, t1, t2 ...</pre>
<pre>int a, b; ... a = b * 50000; ...</pre>	<pre>a → t0 b → t1</pre>	<pre>... li    t2, 50000 mul   t0, t1, t2 ...</pre>	<p><i>programa equivalente</i></p> <pre>... lui   t2, 0xc addi  t2, t2, 0x350 add   t0, t1, t2 ...</pre>

$50000_{10} = 0000c350_{16}$



# Expresiones

- Las **constantes se pueden definir mediante expresiones** formadas por otras constantes (explícitas o simbólicas) u otras expresiones.
  - Durante el ensamblado **las expresiones se reducirán a una constante numérica explícita** que será traducida a código máquina.

C/C++

```
#define N 5
...
int a[N];
...
a[0] = a[1] + a[2];
...
```

$a \equiv \&a[0] \rightarrow t0$   
 $a[1] \rightarrow t1$   
 $a[2] \rightarrow t2$   
 $a[0] \rightarrow t3$

ASM

```
.equ N, 5
.equ LEN, 4
...
a: .space N*LEN
...
la t0, a
lw t1, 1*LEN(t0)
lw t2, 2*LEN(t0)
add t3, t1, t2
sw t3, 0*LEN(t0)
...
```

programa equivalente

```
...
la t0, ...
lw t1, 4(t0)
lw t2, 8(t0)
add t1, t1, t2
sw t1, 0(t0)
...
```



# Organización de código

- En ensamblador **no existen restricciones** sobre cómo debe **organizarse de código**:
  - Pero es recomendable realizar una **programación estructurada y procedural**.
  - **Evita** que el resultado sea un **programa “spaghetti”** plagado de saltos muy difícil de entender, depurar y mantener.
- La **programación estructurada** supone programar en bloques:
  - Cada bloque solo tiene un punto de entrada y uno de salida
  - Un bloque puede ser:
    - **Lineal**: formado por código sin saltos.
    - **Condicional** (tipo **if**, **switch**): formado por bloques que se ejecutan alternativamente según el valor de una condición.
    - **Iterativo** (tipo **for**, **while**): formado por un bloque que se ejecuta repetidamente según el valor de una condición.
- La **programación procedural** supone dividir el código en **funciones reutilizables** más pequeñas:
  - Que utilizan **datos locales** y se comunican entre sí usando **parámetros**.





# Organización de código

## Bloque lineal

- Formado por una sucesión de bloques sin saltos entre ellos.

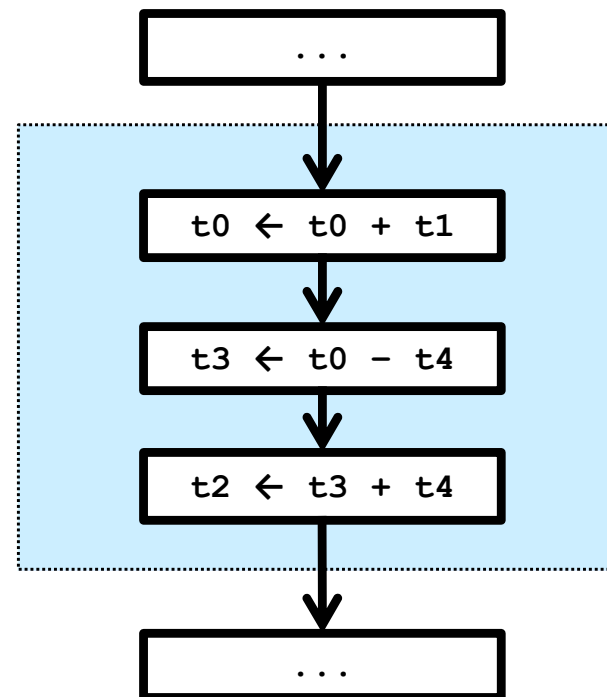
C/C++

```
...  
a = a + b;  
g = a - h;  
f = g + h;  
...
```

```
a → t0 b → t1  
f → t2 g → t3 h → t4
```

ASM

```
...  
add t0, t0, t1  
sub t3, t0, t4  
add t2, t3, t4  
...
```

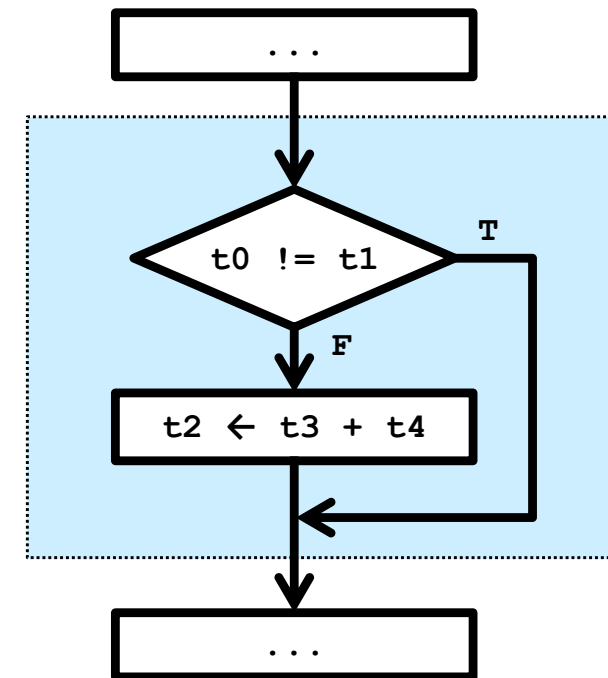
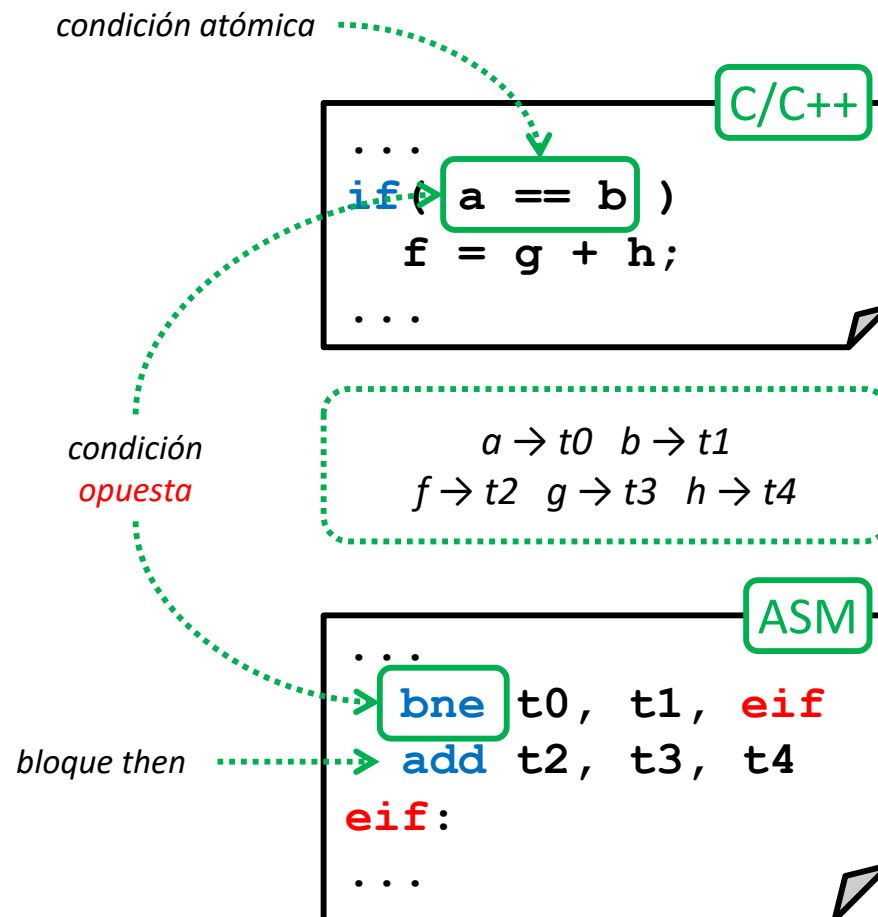




# Organización de código

## Bloque condicional simple *if-then* (i)

- Según el valor de una **condición** ejecuta o no un bloque.





# Organización de código

## Bloque condicional simple *if-then* (ii)

- Cuando la **condición es compuesta**, se chequean de una en una las condiciones atómicas que la forman.

condición compuesta *conjuntiva*

```
...  
if ( a == b && a > 0 )  
    f = g + h;  
...
```

C/C++

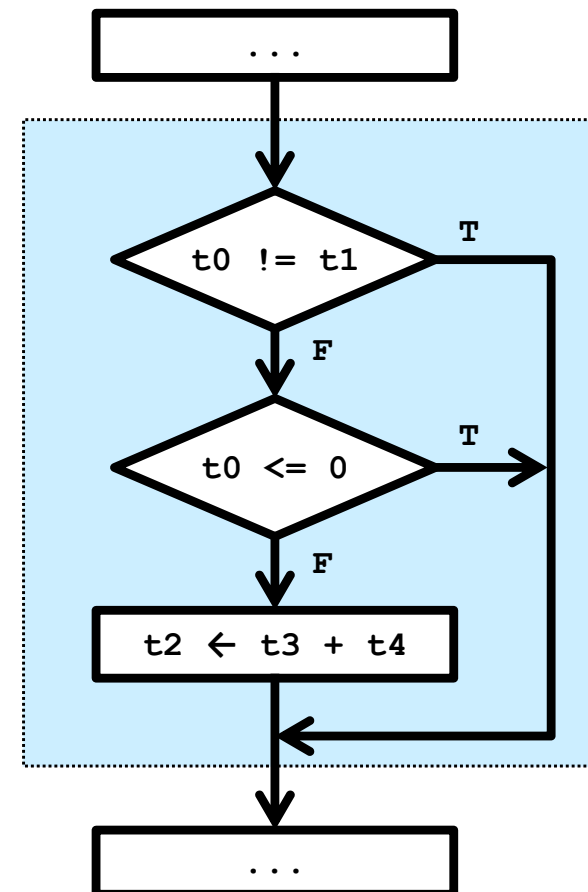
condiciones opuestas

```
a → t0  b → t1  
f → t2  g → t3  h → t4
```

```
...  
bne t0, t1, EIF  
blez t0, EIF  
add t2, t3, t4  
EIF:  
...
```

ASM

bloque then





# Organización de código

## Bloque condicional simple *if-then* (iii)

- Cuando la **condición es compuesta**, se chequean de una en una las condiciones atómicas que la forman.

condición compuesta *disyuntiva*

```
...  
if ( a == b || a > 0 )  
    f = g + h;  
...
```

C/C++

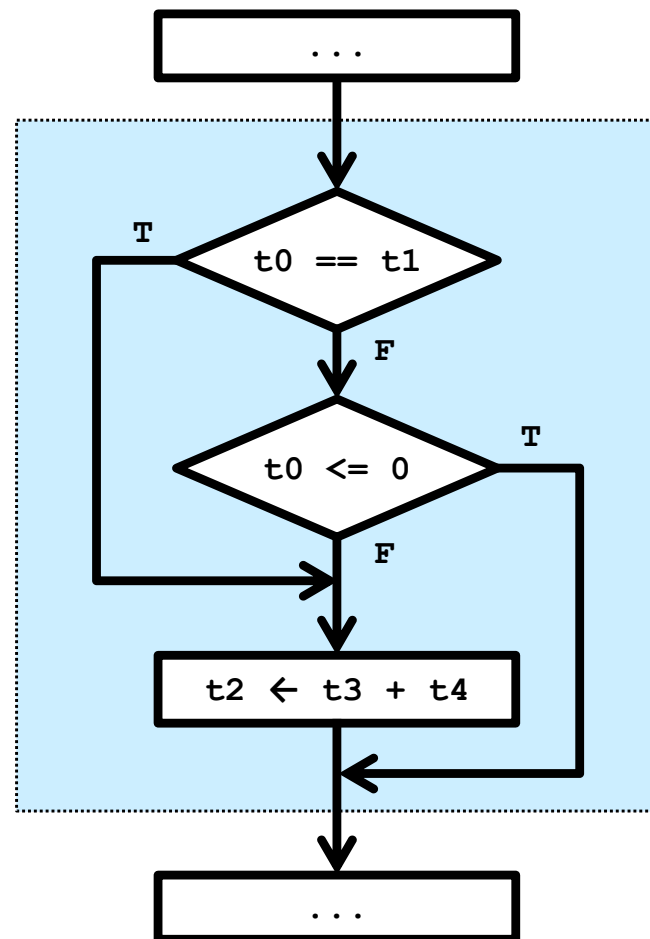
1º condición *igual*  
2º condición *opuesta*

```
a → t0  b → t1  
f → t2  g → t3  h → t4
```

```
...  
beq  t0, t1, then  
blez t0, eif  
then:  
add  t2, t3, t4  
eif:  
...
```

ASM

bloque then

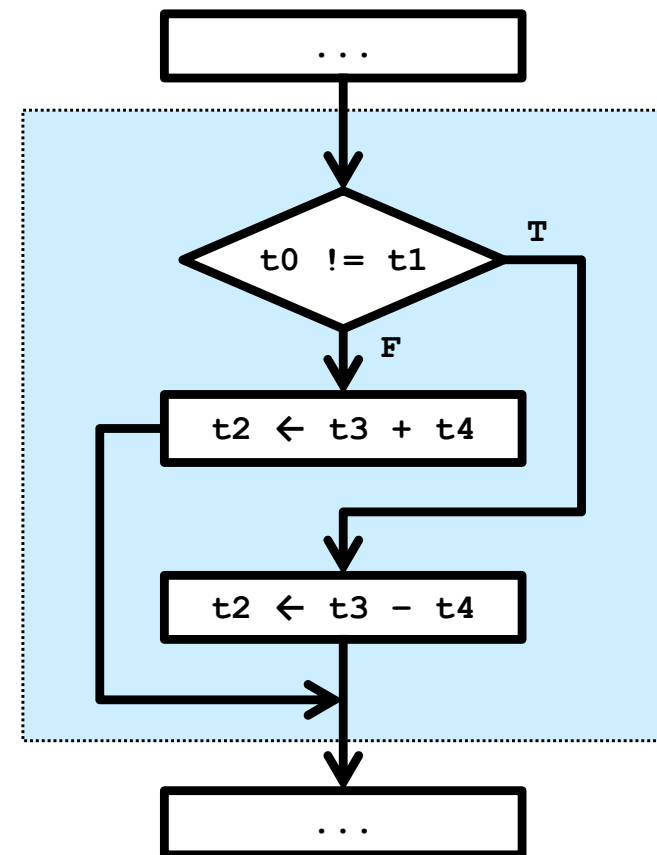
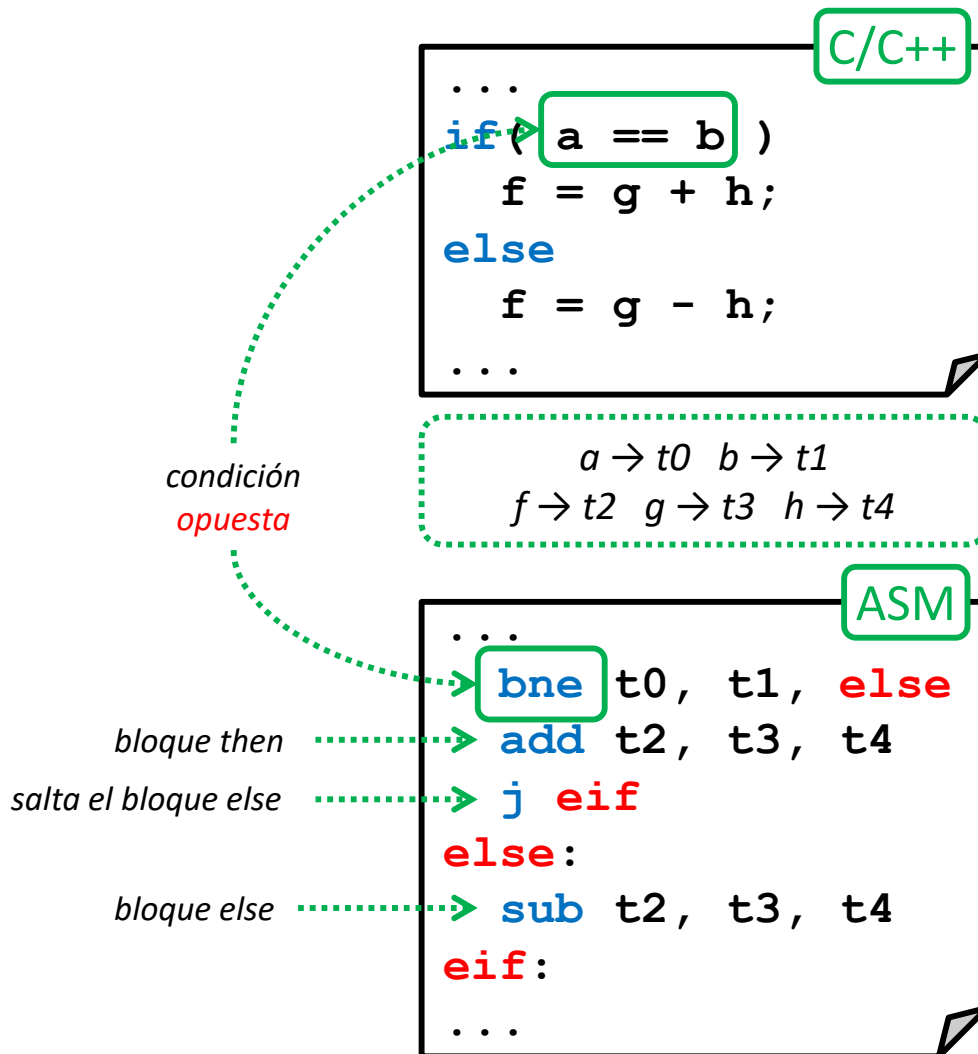




# Organización de código

## Bloque condicional doble *if-then-else*

- Según el valor de una condición ejecuta uno de entre dos bloques.





# Organización de código

## Bloque condicional múltiple *if-then-else*

- Según el valor de **varias condiciones** ejecuta uno de entre varios bloques.

```
...  
if ( a == b )  
    f = g + h;  
else if ( a > 0 )  
    f = g - h;  
...
```

C/C++

$a \rightarrow t0 \quad b \rightarrow t1$   
 $f \rightarrow t2 \quad g \rightarrow t3 \quad h \rightarrow t4$

condiciones  
opuestas

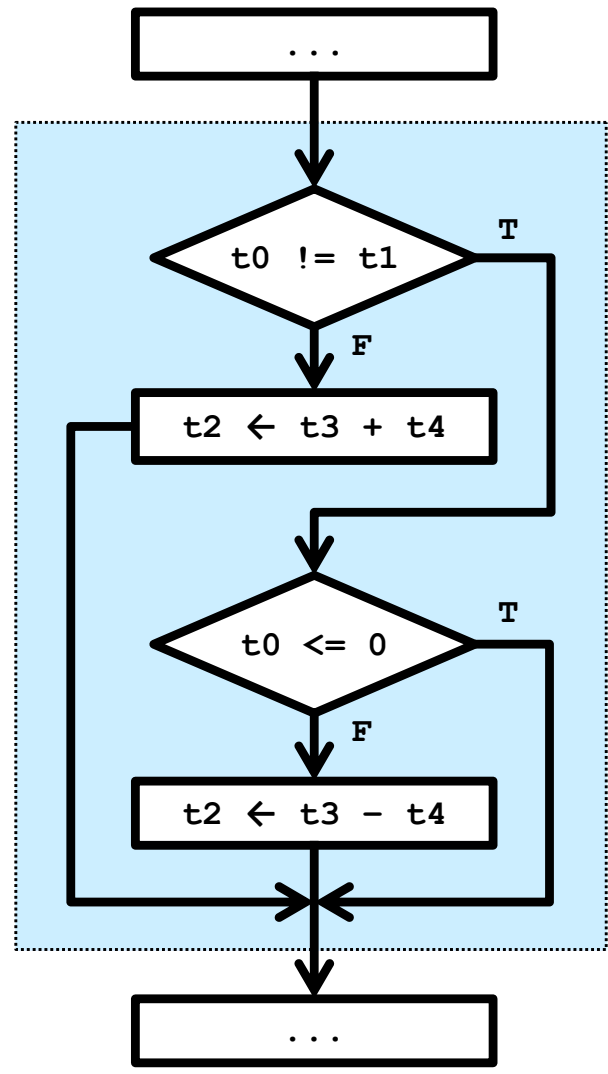
```
...  
bne t0, t1, else  
add t2, t3, t4  
j eif  
else:  
blez t0, eif  
sub t2, t3, t4  
eif:  
...
```

ASM

bloque then

salta el bloque else

bloque else





# Organización de código

## Bloque selectivo *switch*

- Según el valor de una variable ejecuta uno de entre varios bloques.

C/C++

```
...
switch( a )
{
    case 0:
        f = g + h;
        break;
    case 1:
        f = g - h;
        break;
    default:
        f = g;
}
...
```

$a \rightarrow t0$   $f \rightarrow t2$   
 $g \rightarrow t3$   $h \rightarrow t4$

ASM

```
switch: .word case0, case1
...
li    t5, 1
bgt  t0, t5, default
la    t5, switch
slli  t6, t0, 2
add   t5, t5, t6
lw    t5, 0(t5)
jr    t5
case0:
add   t2, t3, t4
j     eswitch
case1:
sub   t2, t3, t4
j     eswitch
default:
mv    t2, t3
eswitch:
...
```

Array de direcciones de comienzo de cada bloque

- Salta al bloque por defecto
- Carga la dirección base del array
- Calcula el desplazamiento
- Suma base y desplazamiento
- Carga dirección de salto
- Salta al bloque correspondiente



# Organización de código

## Bloque iterativo *while-do*

- Repite la ejecución de un bloque según el valor de una **condición** que se **evalúa al principio** del bloque.

C/C++

```
...  
while ( a != b )  
{  
    f = f + a;  
    a = a - b;  
}  
...
```

condición opuesta

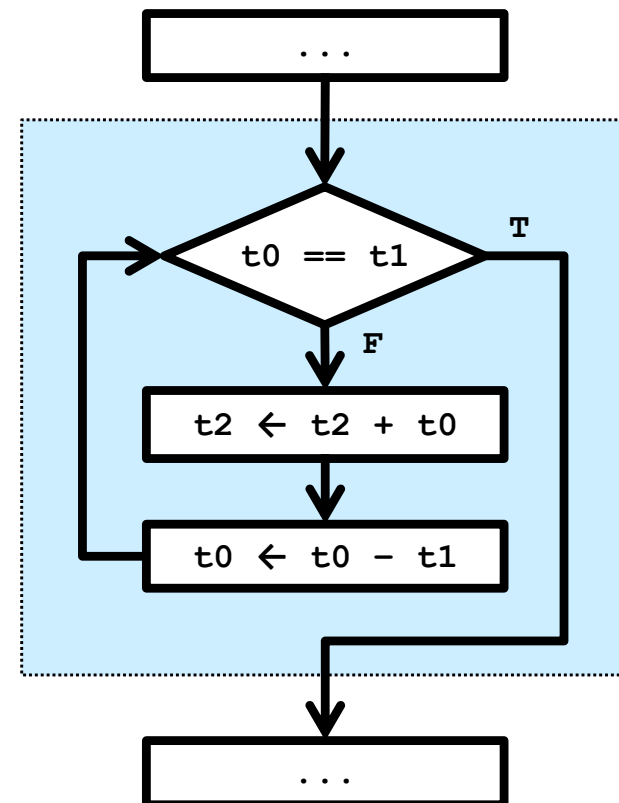
```
a → t0  
b → t1  
f → t2
```

ASM

```
...  
while:  
    beq t0, t1, ewhile  
    add t2, t2, t0  
    sub t0, t0, t1  
    j while  
ewhile:  
...
```

cuerpo del bucle

salta atrás para repetir







# Organización de código

## Bloque iterativo *do-while*

- Repite la ejecución de un bloque según el valor de una **condición** que se **evalúa al final** del bloque.

```
...  
do  
{  
    f = f + a;  
    a = a - b;  
} while ( a != b )  
...
```

C/C++

```
a → t0  b → t1  f → t2
```

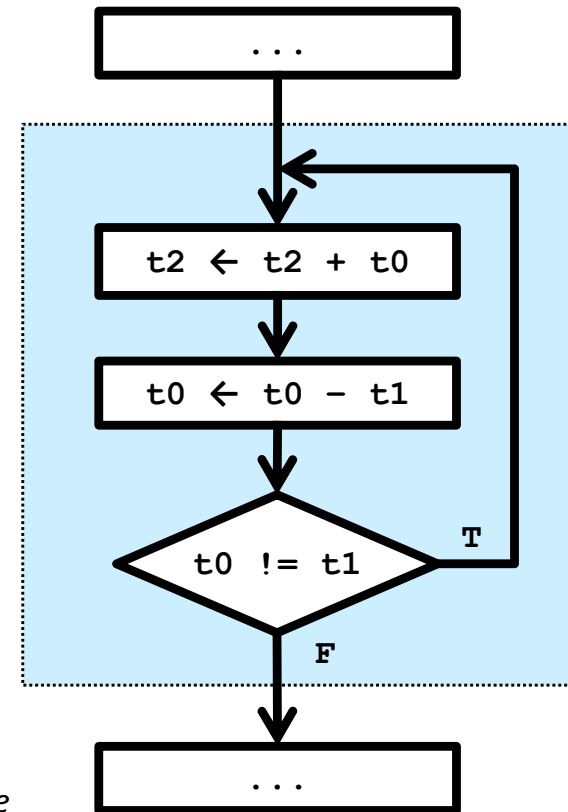
misma condición

```
...  
do:  
    add t2, t2, t0  
    sub t0, t0, t1  
    bne t0, t1, do  
...
```

ASM

cuerpo del bucle

salta atrás para repetir





# Organización de código

## Bloque iterativo *for*

- Repite la ejecución de un bloque un número determinado de veces.

```
...  
for ( a=0; a<10; a=a+1 )  
{  
    f = f + b;  
}  
...
```

C/C++

a → t0  
b → t1  
f → t2

condición opuesta

inicializa el índice

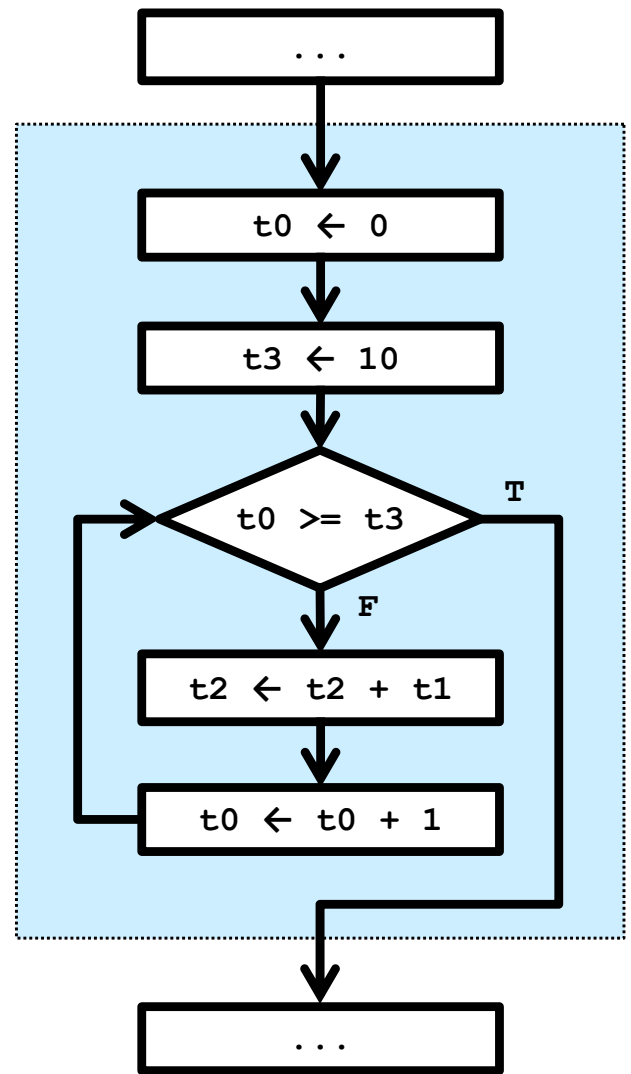
cuerpo del bucle

incrementa índice

salta atrás para repetir

```
...  
mv t0, x0  
li t3, 10  
for:  
bge t0, t3, efor  
add t2, t2, t1  
add t0, t0, 1  
j for  
efor:  
...
```

ASM





# Funciones

- Las **funciones**\* permiten **reutilizar porciones de código** y hacer más **modular** y **legible** un programa.
  - En ensamblador las funciones no se declaran.
  - Se identifican por la dirección de comienzo de su código con una etiqueta.
- Cuando una función (invocante) llama a otra (invocada):
  - La **función invocante** (*caller*) debe **pasar los argumentos** y **saltar al comienzo** de la función invocada.
  - La **función invocada** (*callee*) debe **devolver el resultado** y **saltar a la instrucción siguiente** a la que hizo la llamada en la función invocante.
  - Dado que los registros y la memoria son accesibles por ambas funciones, **la función invocada no debe alterar** nada que sea usado por la invocante.
- En ensamblador, toda **la gestión de argumentos y saltos es explícita**
  - Pero cada arquitectura define un **convenio de llamada a funciones estándar** que debe respetarse para garantizar la interoperabilidad.

(\*) También llamadas procedimientos, métodos o subrutinas



# Funciones

- Los registros del RISC-V pueden ser usados indistintamente, pero para facilitar la gestión de funciones en ensamblador:
  - Cada registro tiene asignado por convenio un cierto propósito y definido un **alias** para que el programador lo recuerde.

# Reg.	Alias	Tipo	Propósito más habitual
x0	zero	N/A	Valor constante 0
x1	ra	preservado	Almacenar la dirección de retorno al función invocante
x2	sp	preservado	Almacenar la dirección de la cima de la pila
x3	gp	N/A	Almacenar la dirección de la región de datos globales de un programa
x4	tp	N/A	Almacenar la dirección de la región de datos locales a una hebra
x5...x7	t0...t2	temporal	Propósito general
x8	s0 fp	preservado	Propósito general Almacenar la dirección de la base del marco de una función
x9	s1	preservado	Propósito general
x10...x11	a0...a1	temporal	Pasar argumentos a la función invocada Devolver valor a la función invocante
x12...x17	a2...a7	temporal	Pasar argumentos a la función invocada
x18...x27	s2...s11	preservado	Propósito general
x28...x31	t3...t6	temporal	Propósito general



# Funciones

## Llamada y retorno (i)

- Por convenio, la **función invocante** en ensamblador de RISC-V usa:
  - Los registros: **a0 ... a7** para pasar hasta 8 argumentos a la invocada.
  - El registro **ra** para guardar la dirección a la que retornar desde la invocada.
  - La instrucción **jal/jalr** para llamar (saltar) a la invocada.

```
C/C++
int a;
...
a = inc( a );
...
int inc( int x )
{
    return x+1;
}
...
```

```
ASM
a: .space 4
...
la  t0, a
lw  a0, 0(t0)
jal ra, inc
sw  a0, 0(t0)
...
inc:
add a0, a0, 1
jalr x0, ra, 0
...
```

← Carga en a0 el argumento  
← Salta a la función invocada guardando en ra la dirección de retorno



# Funciones

## Llamada y retorno (ii)

- Por convenio, la **función invocada** en ensamblador de RISC-V usa:
  - El registro **a0** para devolver el resultado a la invocante (si el dato devuelto fuera de 64b se usaría también **a1** para la parte alta).
  - La instrucción **jalr** para retornar a la invocante.

```
C/C++
int a;
...
a = inc( a );
...
int inc( int x )
{
    return x+1;
}
...
```

```
ASM
a: .space 4
...
la  t0, a
lw  a0, 0(t0)
jal ra, inc
sw  a0, 0(t0)
...
inc:
add a0, a0, 1
jalr x0, ra, 0
...
```

La función invocante almacena el resultado devuelto en a0

Guarda en a0 el resultado

Retorna a la función invocante



# Funciones

## Llamada y retorno (iii)

- Dado que el salto/retorno a/desde funciones es muy común:
  - La **función invocante** suele usar la pseudo-instrucción **call**
  - La **función invocada** suele la pseudo-instrucción **ret**

C/C++

```
int a;  
...  
a = inc( a );  
...  
int inc( int x )  
{  
    return x+1;  
}  
...
```

ASM

```
a: .space 4  
...  
la    t0, a  
lw    a0, 0(t0)  
call  inc  
sw    a0, 0(t0)  
...  
inc:  
add   a0, a0, 1  
ret  
...
```

programa equivalente

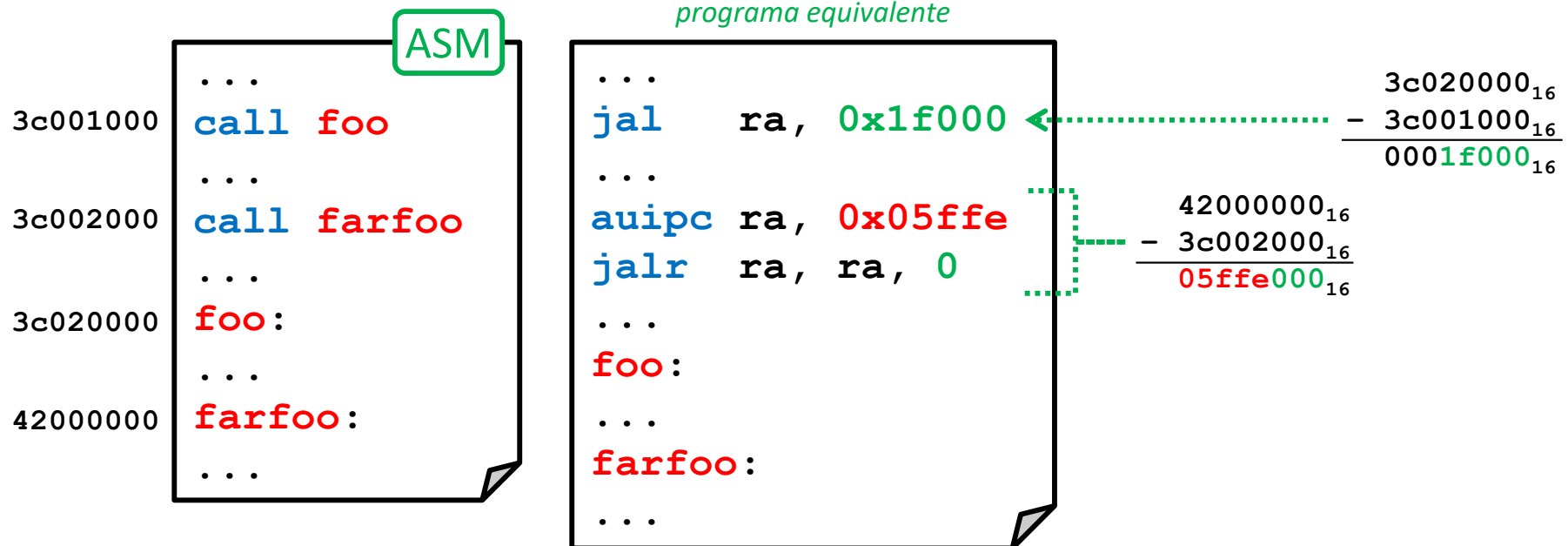
```
a: .space 4  
...  
la    t0, a  
lw    a0, 0(t0)  
jal   ra, inc  
sw    a0, 0(t0)  
...  
inc:  
addi  a0, a0, 1  
jalr  x0, ra, 0  
...
```



# Funciones

## Llamada y retorno (iv)

- Además, la **pseudo-instrucción call** durante el ensamblado es traducida convenientemente:
  - A una instrucción **jal**, si la función está dentro del ámbito de  $\pm 1\text{MiB}$ 
    - El desplazamiento de esta instrucción es de 21b en C2 relativo al PC.
  - A un par **auipc+jalr**, si la función está en un ámbito más lejano.
  - Liberando al programador de conocer la cercanía de la función invocada.







# Funciones

## Parámetros por valor vs. referencia (i)

- Los **parámetros de entrada** se pasan a la función invocada **por valor**:
  - La **función invocante** pasa como argumento a la función invocada una **copia del valor** de la variable.
  - Si el valor de la variable debe actualizarse, lo hace la función invocante.

```
C/C++
int a;
...
a = inc( a );
...
int inc( int x )
{
    return x+1;
}
...
```

```
ASM
a: .space 4
...
la    t0, a
lw    a0, 0(t0)
call  inc
sw    a0, 0(t0)
...
inc:
add   a0, a0, 1
ret
...
```

La función invocante copia en a0 el valor de a

La función invocante actualiza el valor de a

x se declara como parámetro de entrada



# Funciones

## Parámetros por valor vs. referencia (ii)

- Los **parámetros de salida** se pasan a la función invocada **por referencia**:
  - La **función invocante** pasa como argumento a la función invocada la **dirección de la variable** que debe modificarse.
  - La actualización del valor de la variable la hace la función invocada.

```
int a;  
...  
inc( a );  
...  
void inc( int & x )  
{  
    x = x + 1;  
}  
...
```

C++

```
a: .space 4  
...  
la    a0, a  
call  inc  
...  
inc:  
lw    t0, 0(a0)  
add   t0, t0, 1  
sw    t0, 0(a0)  
ret  
...
```

ASM

La función invocante  
copia en a0 la dirección de a

La función invocada  
actualiza el valor de a

x se declara como  
parámetro de salida



# Funciones

## Parámetros por valor vs. referencia (iii)

- El concepto de **puntero en C/C++** es una abstracción de la **dirección ocupada por una variable**.
  - Los parámetros de salida en C, son argumentos de tipo puntero.

C++

```
int a;  
...  
inc( a );  
...  
void inc( int & x )  
{  
    x = x + 1;  
}
```

C

```
int a;  
...  
inc( &a );  
...  
void inc( int *x )  
{  
    *x = *x + 1;  
}
```

ASM

```
a: .space 4  
...  
la    a0, a  
call  inc  
...  
inc:  
lw    t0, 0(a0)  
add   t0, t0, 1  
sw    t0, 0(a0)  
ret
```



# Funciones

## Parámetros por valor vs. referencia (iv)

- Los **arrays** se pasan a la función invocada **por referencia**.

C/C++

```
int a[10];
...
incArray( a, 10 );
...
void incArray( int x[], int n )
{
    int i;

    for( i=0; i<n; i=i+1 )
        x[i] = x[i] + 1;
}
```

C/C++

```
int a[10];
...
incArray( a, 10 );
...
void incArray( int *x, int n )
{
    int i;

    for( i=0; i<n; i=i+1 )
        x[i] = x[i] + 1;
}
```

ASM

```
a: .space 4*10
...
la    a0, a
li    a1, 10
call  incArray
...
incArray:
    mv    t0, zero
for:
    bge  t0, a1, efor
    sll  t1, t0, 2
    add  t1, a0, t1
    lw   t2, 0(t1)
    add  t2, t2, 1
    sw   t2, 0(t1)
    add  t0, t0, 1
    j    for
efor:
    ret
```



# Funciones

## Registros temporales vs. preservados (i)

- La función invocada puede usar los **mismos registros** que está usando la función invocante.
  - Si la función invocada **cambia alguno en uso** por la invocante, al retornar a ésta no encontrará el valor esperado y **el programa fallará**.

```
C/C++
int a;
...
a = inc( a );
...
int inc( int x )
{
    return x+1;
}
...
```

```
INCORRECTO
ASM
a: .space 4
...
la    t0, a
lw    a0, 0(t0)
call  inc
sw    a0, 0(t0)
...
inc:
add   t0, a0, 1
mv    a0, t0
ret
...
```

La función invocante usa t0 para almacenar temporalmente la dirección de a

Pero, el valor de t0 ha cambiado tras la llamada y ya no contiene la dirección de a

La función invocada usa t0 para almacenar temporalmente el resultado del cálculo

# Funciones

## Registros temporales vs. preservados (ii)



- Por convenio, los registros se clasifican en preservados y temporales.
- **Registro preservado** (*callee-saved*): Aquel que el programador debe garantizar que su contenido no varía tras ejecutar una función.
  - Su valor tras retornar de la función invocada debe ser el mismo que tenía cuando se saltó a ella.
  - Para ello, o no se modifica dentro de la función invocada o la función invocada salva su valor al principio y lo restaura al final.
  - Son registros preservados: **s1 ... s11, sp, ra, s0/fp**
- **Registro temporal** (*caller-saved*): Aquel cuyo contenido puede alterarse libremente al ejecutar una función.
  - Su valor tras retornar de la función invocada puede ser distinto del que tenía cuando se saltó a ella.
  - Si la función invocante quiere conservar su valor, debe salvar su valor antes de saltar a la función invocada y restaurarlo a su retorno.
  - Son registros temporales: **t0 ... t6, a0 ... a7**



# Funciones

## Registros temporales vs. preservados (iii)

- Según este convenio, lo correcto sería que la función invocante use registros preservados cuando quiera conservar un dato tras una llamada.
  - La invocada podrá seguir usando registros temporales y si usa preservados deberá salvarlos antes de modificarlos y restaurarlos antes de volver.

<div data-bbox="667 738 824 810" style="border: 1px solid green; border-radius: 10px; padding: 2px; display: inline-block;">C/C++</div>	<div data-bbox="929 715 1198 762" style="color: green;">CORRECTO</div>	<div data-bbox="1294 738 1417 810" style="border: 1px solid green; border-radius: 10px; padding: 2px; display: inline-block;">ASM</div>	
<pre>int a; ... a = inc( a ); ... int inc( int x ) {     return x+1; } ...</pre>	<pre>a: .space 4 ... la    s0, a lw    a0, 0(s0) call  inc sw    a0, 0(s0) ... inc: add   t0, a0, 1 mv    a0, t0 ret ...</pre>	<p>La función invocante usa s0 para almacenar temporalmente la dirección de a</p> <p>Si inc está correctamente programada, s0 seguirá conteniendo la dirección de a</p> <p>La función invocada no usa s0 por lo que su valor no cambiará durante su ejecución</p>	



# Funciones

## Gestión de pila (i)

- La **pila** (*stack*) es una **región de memoria** en donde se pueden **almacenar datos** temporalmente sin conocer la dirección efectiva que ocupan.
  - Registros que se deben preservar.
  - Argumentos de una función (cuando son más de 8).
  - Variables locales a una función cuando no hay registros suficientes.
- Sobre una pila se pueden realizar **2 operaciones**:
  - **Apilar** (*push*): **almacenar un dato** sobre la cima de la pila.
  - **Desapilar** (*pop*): **recuperar el dato** ubicado en la cima de la pila.
  - La **pila funciona como una LIFO** (Last-in First-out): los datos apilados en un cierto orden se recuperan desapilándolos en orden inverso.
- Por convenio, en ensamblador de RISC-V:
  - La pila es **descendente**: **crece de direcciones altas hacia bajas** de memoria.
  - Se usa el registro **sp** para almacenar la **dirección de la cima de la pila**, que siempre **contiene el último dato apilado**.



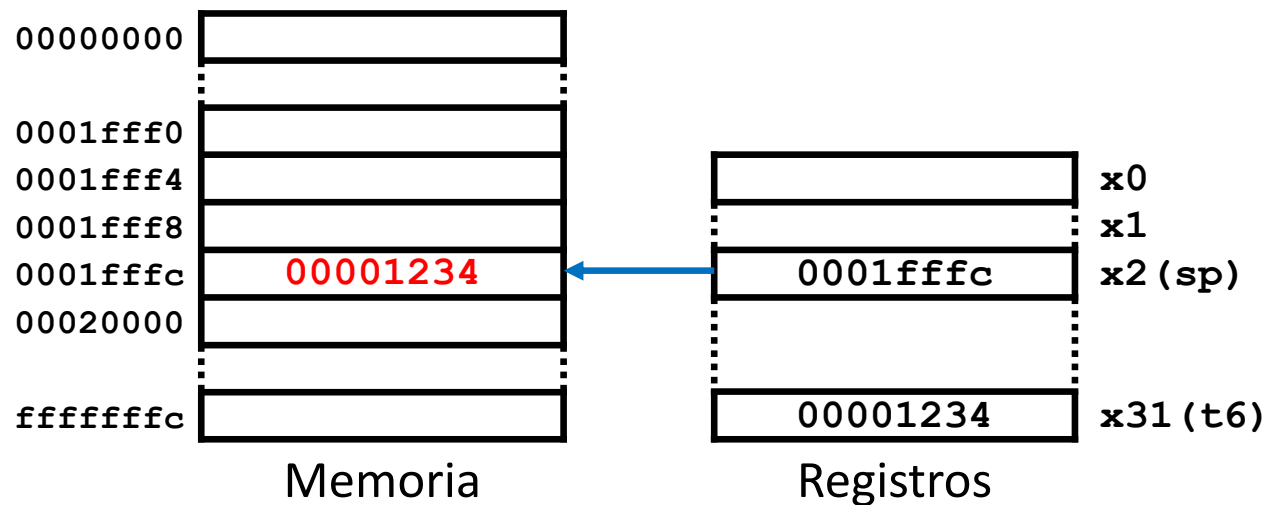


# Funciones

## Gestión de pila (ii)

- **Apilar un dato** supone:
  - Decrementar **sp** (el número de bytes que ocupe el dato, normalmente 4).
  - Almacenar el dato en la **cima** de la pila.

```
ASM
...
li sp, 0x20000
...
li t6, 0x1234
add sp, sp, -4
sw t6, 0(sp)
mv t6, zero
...
lw t6, 0(sp)
add sp, sp, 4
...
li t6, 0x9abc
add sp, sp, -4
sw t6, 0(sp)
...
```



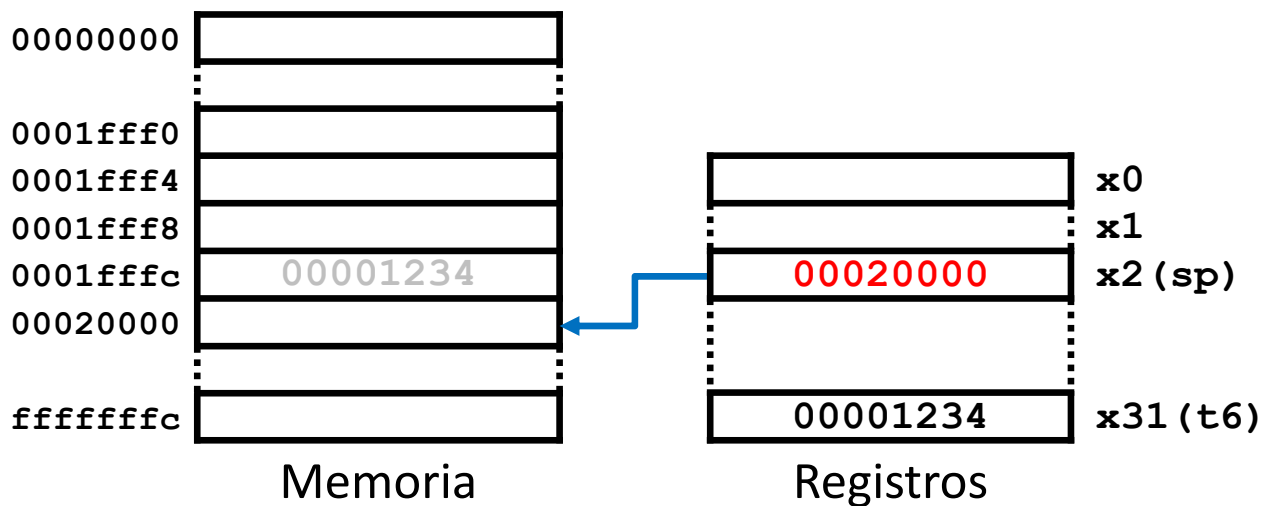


# Funciones

## Gestión de pila (iii)

- **Desapilar un dato** supone:
  - Cargar el dato ubicado en la **cima** de la pila.
  - Incrementar **sp** (el número de bytes que ocupe el dato, normalmente 4).

```
ASM
...
li    sp, 0x20000
...
li    t6, 0x1234
add   sp, sp, -4
sw    t6, 0(sp)
mv    t6, zero
...
lw    t6, 0(sp)
add   sp, sp, 4
...
li    t6, 0x9abc
add   sp, sp, -4
sw    t6, 0(sp)
...
```



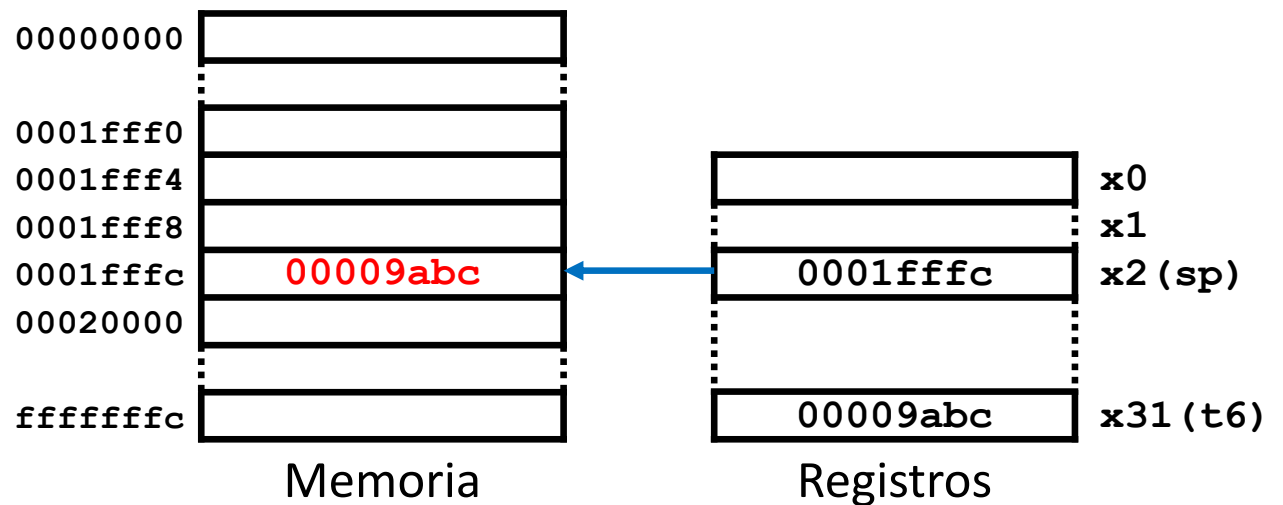


# Funciones

## Gestión de pila (iv)

- Los datos desapilados **permanecen en memoria**
  - Pero **no pueden usarse** porque **serán sobrescritos** por los datos que se apilen con posterioridad.

```
ASM
...
li    sp, 0x20000
...
li    t6, 0x1234
add  sp, sp, -4
sw   t6, 0(sp)
mv   t6, zero
...
lw   t6, 0(sp)
add  sp, sp, 4
...
li    t6, 0x9abc
add  sp, sp, -4
sw   t6, 0(sp)
...
```



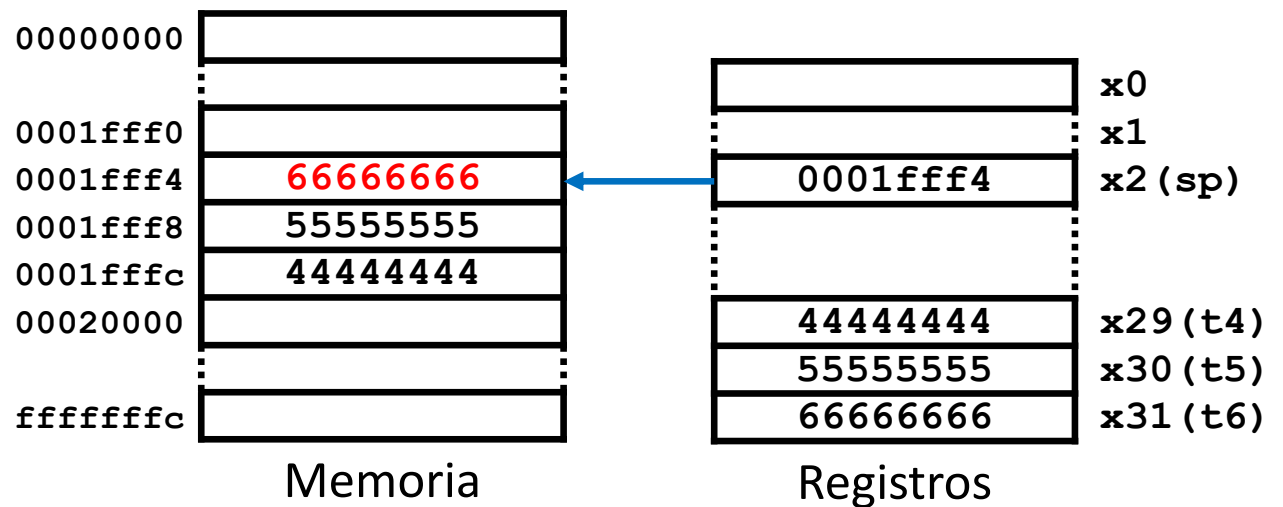


# Funciones

## Gestión de pila (v)

- **Apilar un conjunto de datos** supone:
  - Decrementar **sp** (el número de bytes que ocupen todos ellos).
  - Almacenar cada dato en **direcciones consecutivas desde la cima** de la pila.

```
...  
li    sp, 0x20000  
...  
add  sp, sp, -12  
sw   t4, 8(sp)  
sw   t5, 4(sp)  
sw   t6, 0(sp)  
...  
mv   t4, zero  
...  
lw   t4, 8(sp)  
lw   t5, 4(sp)  
lw   t6, 0(sp)  
add  sp, sp, 12  
...
```



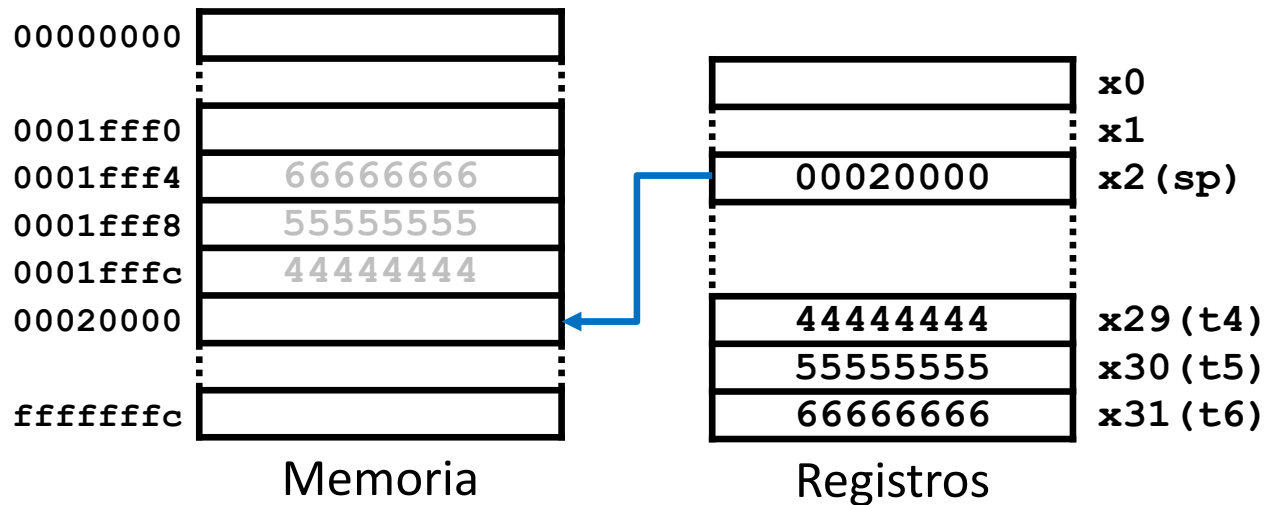


# Funciones

## Gestión de pila (vi)

- **Desapilar un conjunto de datos** supone:
  - Cargar los datos ubicados en direcciones consecutivas desde la cima.
  - Incrementar `sp` (el número de bytes que ocupen todos ellos).

```
...  
li    sp, 0x20000  
...  
add  sp, sp, -12  
sw   t4, 8(sp)  
sw   t5, 4(sp)  
sw   t6, 0(sp)  
...  
mv   t4, zero  
...  
lw   t4, 8(sp)  
lw   t5, 4(sp)  
lw   t6, 0(sp)  
→ add sp, sp, 12  
...
```

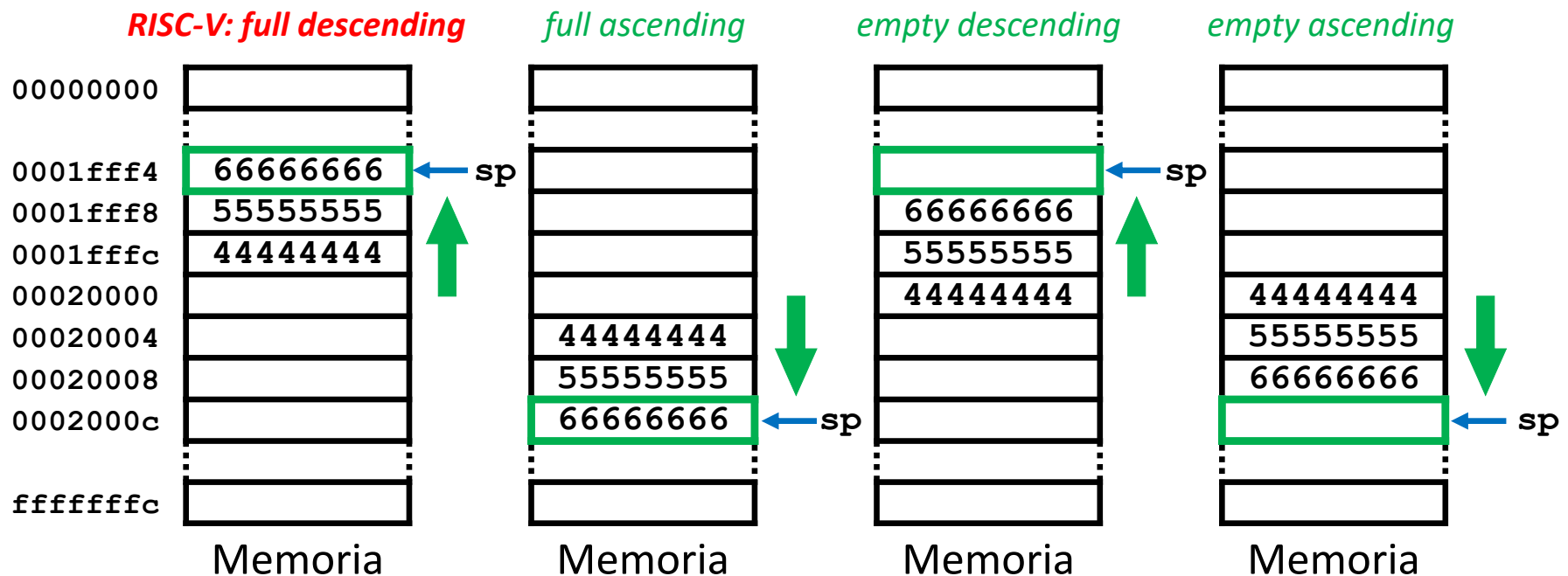




# Funciones

## Gestión de pila (vii)

- La pila del RISC-V es de tipo *full descending* porque:
  - Crece hacia direcciones bajas y apila pre-decrementando el `sp`, es decir, primero decrementa `sp` y después almacena el dato en la cima de la pila.
- En otras arquitecturas existen otros convenios:
  - *Full ascending*: crece hacia direcciones altas, apila pre-incrementando.
  - *Empty descending*: crece hacia direcciones bajas, apila post-decrementando.
  - *Empty ascending*: crece hacia direcciones altas, apila post-incrementando.





# Funciones

## Salvado de registros (i)

- **Regla de la función invocante.** La **función invocante** deberá:
  - **Apilar los registros temporales** que contengan valores que vaya a necesitar posteriormente **antes de saltar** a la función invocada.
  - **Desapilar dichos registros** tras el **retorno** de la función invocada.

```
C/C++
int a;
...
a = inc( a );
...
int inc( int x )
{
    return x+1;
}
...
```

```
ASM
a: .space 4
...
la    t0, a
lw    a0, 0(t0)
add   sp, sp, -4
sw    t0, 0(sp)
call  inc
lw    t0, 0(sp)
add   sp, sp, 4
sw    a0, 0(t0)
...
inc:
add   t0, a0, 1
mv    a0, t0
ret
...
```

La función invocante usa `t0` para almacenar la dirección de `a`

La función invocante apila el valor de `t0` antes de llamar a `inc`, porque lo necesitará tras su retorno

La función invocante despila el valor de `t0` antes de volver a usarlo

Al ser `t0` un registro temporal, la función invocada puede modificarlo libremente.



# Funciones

## Salvado de registros (ii)

- **Regla de la función invocada.** La función invocada deberá:
  - Apilar los registros preservados que use antes de cambiar su valor.
    - Esta colección de registros se denomina **contexto** de la función invocante.
  - Desapilar dichos registros antes de retornar a la función invocante.

```
C/C++
int a;
...
a = inc( a );
...
int inc( int x )
{
    return x+1;
}
...
```

```
ASM
a: .space 4
...
la    s0, a
lw    a0, 0(s0)
call  inc
sw    a0, 0(s0)
...
inc:
add   sp, sp, -4
sw    s0, 0(sp)
add   s0, a0, 1
mv    a0, s0
lw    s0, 0(sp)
add   sp, sp, 4
ret
...
```

La función invocante usa s0 para almacenar la dirección de a

La función invocante asume que el registro s0 conservar su valor tras la llamada a inc

Al ser s0 un registro preservado, la función invocada debe apilar su valor antes de modificarlo.

La función invocada desapila el valor de s0 antes de volver a la invocante

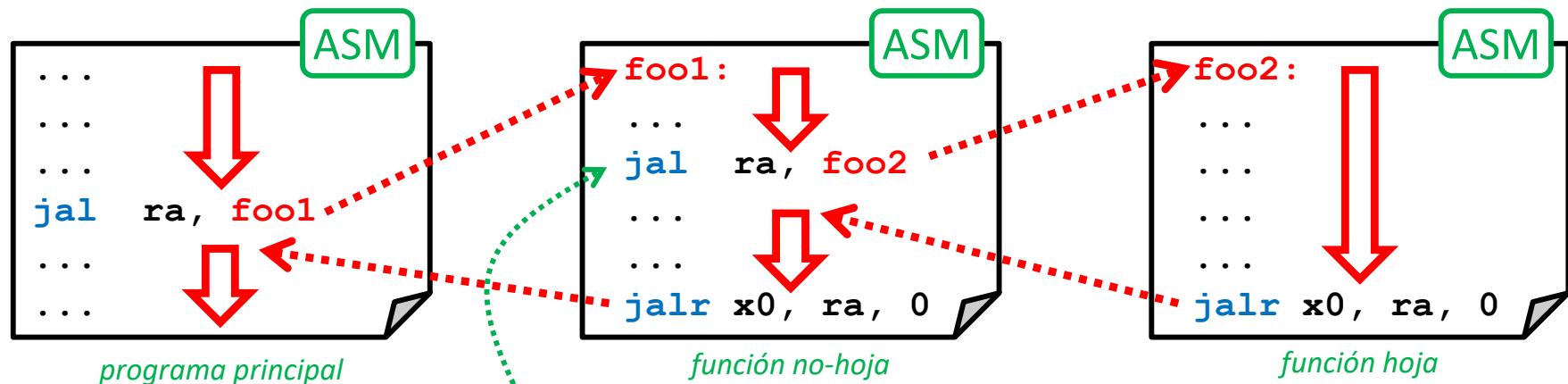




# Funciones

## Anidamiento y recursividad (i)

- Si una **función no llama a otra** (tipo hoja) solo debe **apilar aquellos registros preservados que use**.
  - Si solo usa registros temporales, no debe apilar nada.
- Pero si la **función llama a otras** (tipo no-hoja) además de **apilar los registros preservados que use** también deben apilar el **registro ra**.
  - Ya que este **almacena la dirección de retorno** a la función invocante.



Esta instrucción salta a `foo2` y sobrescribe `ra` con la dirección de retorno a `foo1`.  
Si `ra` no se salva previamente al salto, `foo1` no volverá al programa principal (volverá siempre a sí mismo)



# Funciones

## Anidamiento y recursividad (ii)

C/C++

```

int a[10];
...
incArray( a, 10 );
...
void incArray( int x[], int n )
{
    int i;

    for( i=0; i<n; i=i+1 )
        x[i] = inc( x[i] );
}

int inc( int x )
{
    return x+1;
}
...

```

$x[] \rightarrow s0, n \rightarrow s1, i \rightarrow s2, \&x[i] \rightarrow s3$

ASM

```

a: .space 4*10
...
la    a0, a
li    a1, 10
jal   ra, incArray
...
incArray:
add   sp, sp, -20
sw    ra, 16(sp)
sw    s0, 12(sp)
...
mv    s0, a0
mv    s1, a1
mv    s2, zero
for:
bge   s2, s1, efor
sll   t0, s2, 2
add   s3, s0, t0
lw    a0, 0(s3)
jal   ra, inc
sw    a0, 0(s3)
add   s2, s2, 1
j     for
efor:
lw    ra, 16(sp)
lw    s0, 12(sp)
...
add   sp, sp, 20
jalr  x0, ra, 0
inc:
add   a0, a0, 1
jalr  x0, ra, 0
...

```

*incArray es una función no-hoja, por lo que debe apilar su dirección de retorno antes de llamar a inc.*

*también debe apilar los 4 registros preservados usados*

*Sobrescribe el registro ra*

*incArray restaurara el registro ra para poder volver al programa principal*

*también debe restaurar los 4 registros preservados usados*



# Funciones

## Anidamiento y recursividad (iii)

- Las **funciones recursivas** son un caso extremo de anidamiento (se invocan a sí mismas).

```
int a;
...
a = fact( 4 );
...
int fact( int x )
{
    if( x <= 1 )
        return 1;
    else
        return x*fact( x-1 );
}
```

C/C++

```
a: .space 4
...
li    sp, 0x20000
li    a0, 4
call  fact
la    t6, a
sw    a0, 0(t6)
...
fact:
add   sp, sp, -8
sw    ra, 4(sp)
sw    s1, 0(sp)
mv    s1, a0
li    t6, 1
bgt   s1, t6, else
li    a0, 1
j     eif
else:
add   a0, s1, -1
call  fact
mul   a0, s1, a0
eif:
lw    ra, 4(sp)
lw    s1, 0(sp)
add   sp, sp, 8
ret
```

ASM

caso base

caso general

apila contexto

¿x > 1?

retorna 1

fact(x-1)

retorna x\*fact(x-1)

desapila contexto

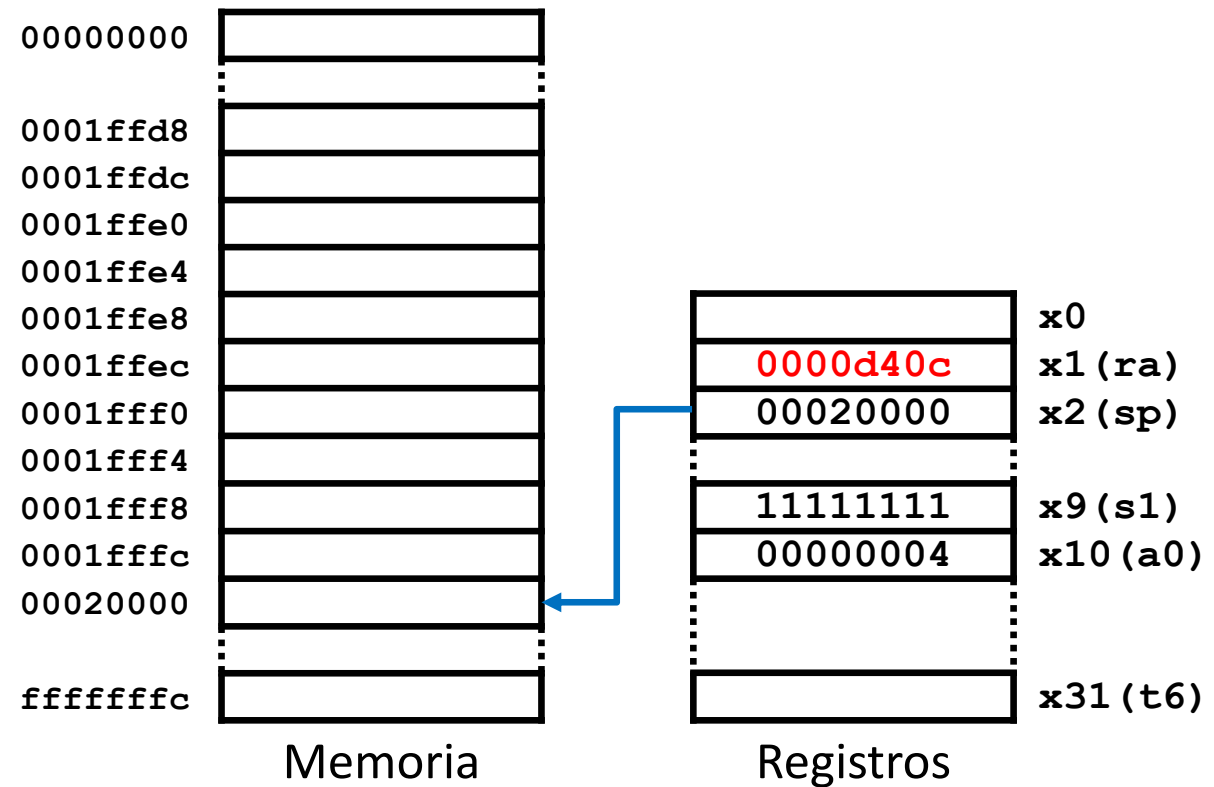


# Funciones

## Anidamiento y recursividad (iv)

ASM

```
a: .space 4
...
li    sp, 0x20000
li    a0, 4
call  fact
0000d40c la    t6, a
      sw    a0, 0(t6)
...
fact:
add   sp, sp, -8
sw    ra, 4(sp)
sw    s1, 0(sp)
mv    s1, a0
li    t6, 1
bgt   s1, t6, else
li    a0, 1
j     eif
else:
add   a0, s1, -1
call  fact
0000fa28 mul   a0, s1, a0
eif:
lw    ra, 4(sp)
lw    s1, 0(sp)
add   sp, sp, 8
ret
```





# Funciones

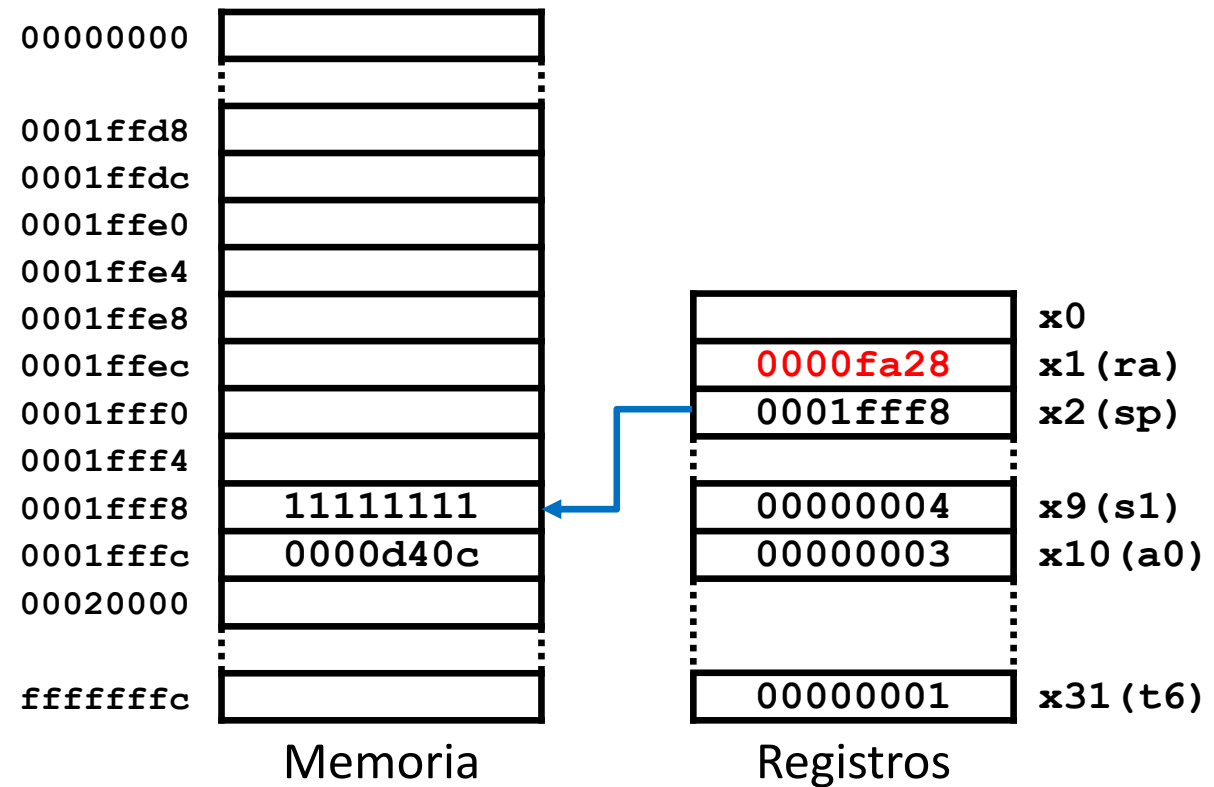
## Anidamiento y recursividad (iv)

ASM

```

a: .space 4
...
li    sp, 0x20000
li    a0, 4
call  fact
0000d40c la    t6, a
      sw    a0, 0(t6)
...
fact:
      add   sp, sp, -8
      sw    ra, 4(sp)
      sw    s1, 0(sp)
      mv    s1, a0
      li    t6, 1
      bgt   s1, t6, else
      li    a0, 1
      j     eif
else:
      add   a0, s1, -1
      call  fact
0000fa28 mul   a0, s1, a0
eif:
      lw    ra, 4(sp)
      lw    s1, 0(sp)
      add   sp, sp, 8
      ret

```





# Funciones

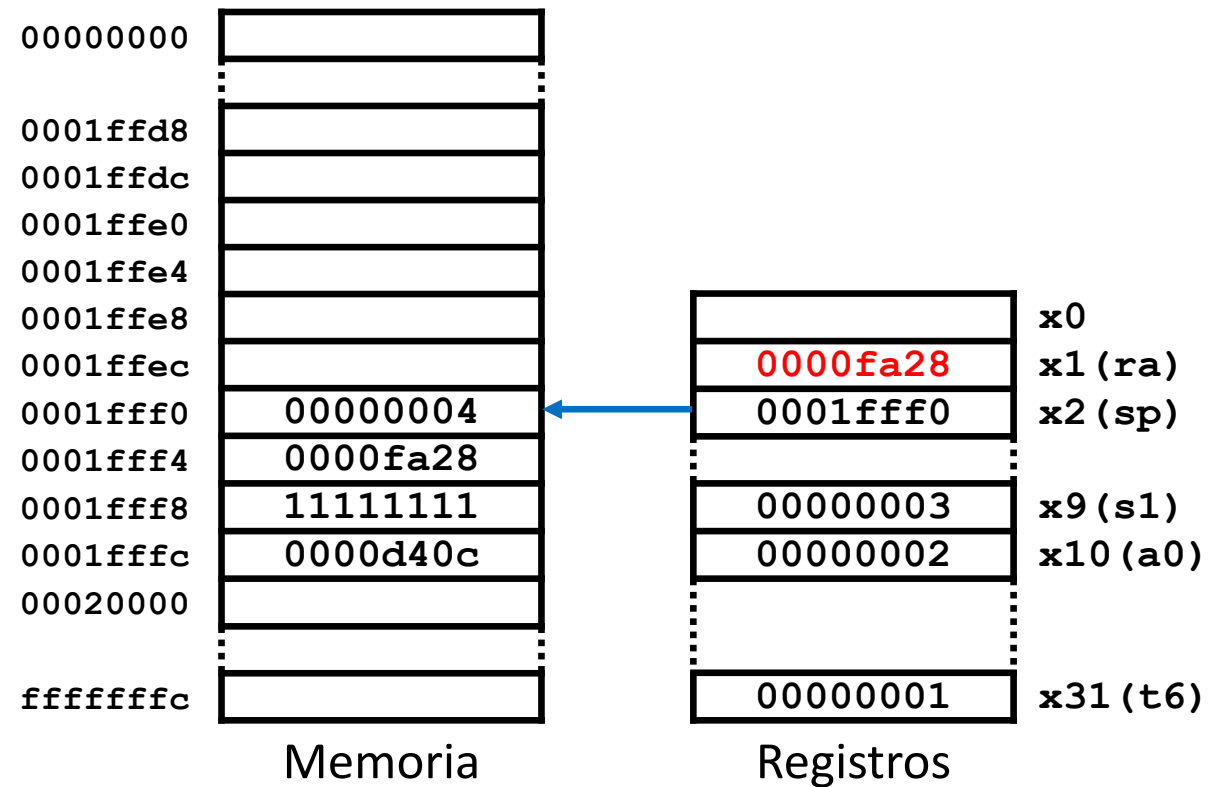
## Anidamiento y recursividad (iv)

ASM

```

a: .space 4
...
li    sp, 0x20000
li    a0, 4
call  fact
0000d40c la  t6, a
sw    a0, 0(t6)
...
fact:
add   sp, sp, -8
sw    ra, 4(sp)
sw    s1, 0(sp)
mv    s1, a0
li    t6, 1
bgt   s1, t6, else
li    a0, 1
j     eif
else:
add   a0, s1, -1
call  fact
0000fa28 mul a0, s1, a0
eif:
lw    ra, 4(sp)
lw    s1, 0(sp)
add   sp, sp, 8
ret

```



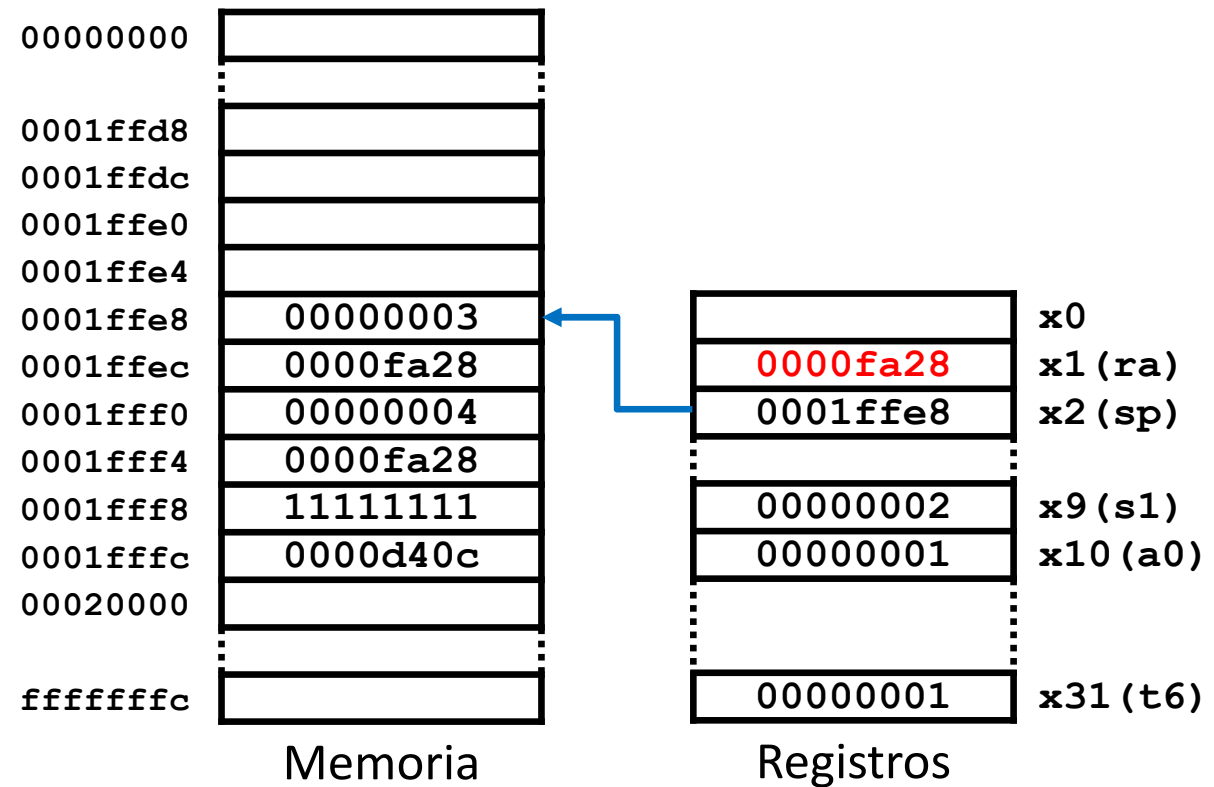


# Funciones

## Anidamiento y recursividad (iv)

ASM

```
a: .space 4
...
li    sp, 0x20000
li    a0, 4
call  fact
0000d40c la    t6, a
      sw    a0, 0(t6)
...
fact:
add   sp, sp, -8
sw    ra, 4(sp)
sw    s1, 0(sp)
mv    s1, a0
li    t6, 1
bgt   s1, t6, else
li    a0, 1
j     eif
else:
add   a0, s1, -1
call  fact
0000fa28 mul  a0, s1, a0
eif:
lw    ra, 4(sp)
lw    s1, 0(sp)
add   sp, sp, 8
ret
```



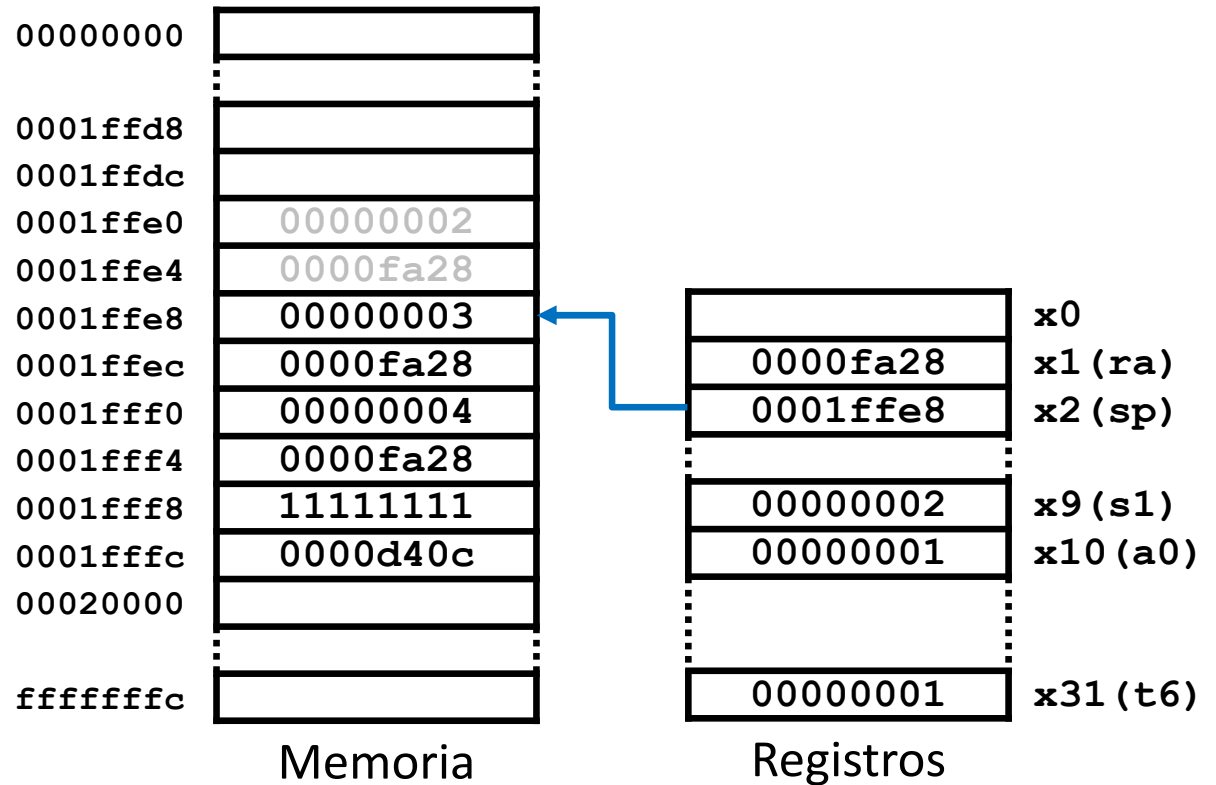


# Funciones

## Anidamiento y recursividad (iv)

ASM

```
a: .space 4
...
li    sp, 0x20000
li    a0, 4
call  fact
0000d40c la    t6, a
      sw    a0, 0(t6)
...
fact:
      add   sp, sp, -8
      sw    ra, 4(sp)
      sw    s1, 0(sp)
      mv    s1, a0
      li    t6, 1
      bgt   s1, t6, else
      li    a0, 1
      j     eif
else:
      add   a0, s1, -1
      call  fact
0000fa28 mul   a0, s1, a0
eif:
      lw    ra, 4(sp)
      lw    s1, 0(sp)
      add   sp, sp, 8
      ret
```







# Funciones

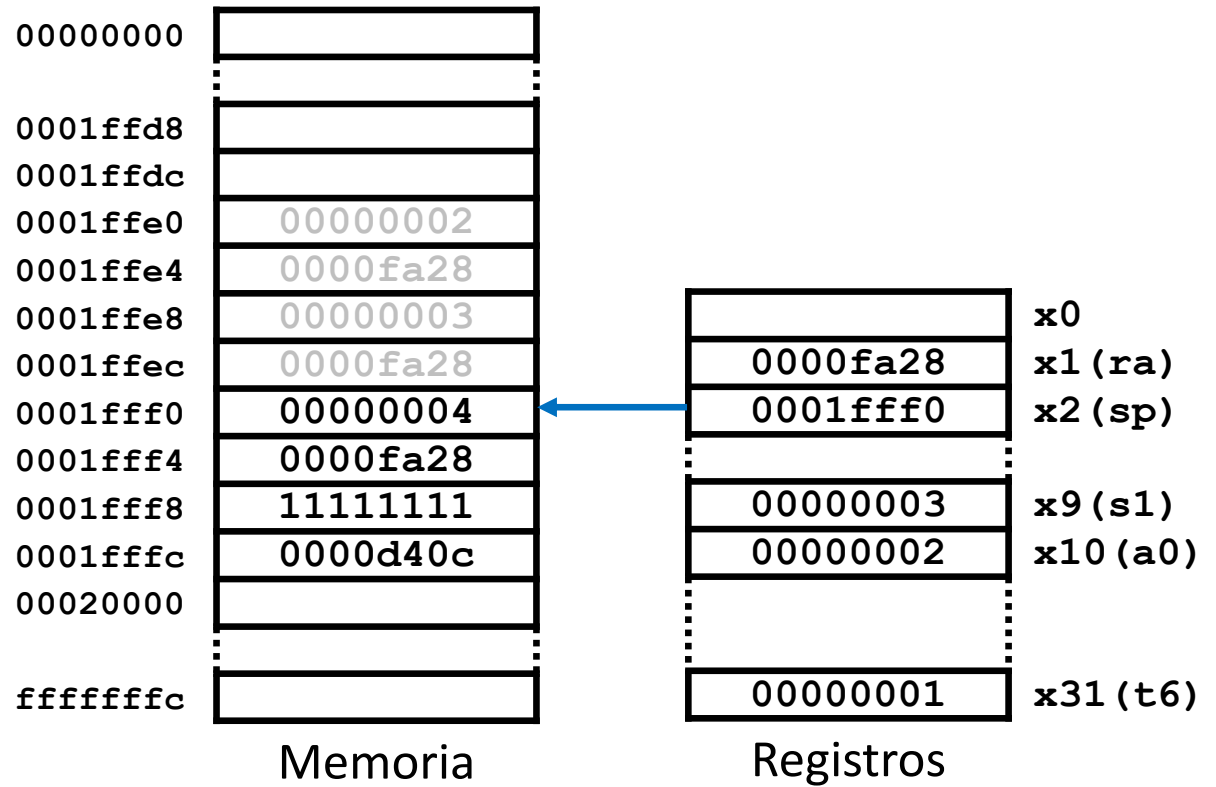
## Anidamiento y recursividad (iv)

ASM

```

a: .space 4
...
li    sp, 0x20000
li    a0, 4
call  fact
0000d40c la    t6, a
      sw    a0, 0(t6)
...
fact:
      add   sp, sp, -8
      sw    ra, 4(sp)
      sw    s1, 0(sp)
      mv    s1, a0
      li    t6, 1
      bgt   s1, t6, else
      li    a0, 1
      j     eif
else:
      add   a0, s1, -1
      call  fact
0000fa28 mul   a0, s1, a0
eif:
      lw    ra, 4(sp)
      lw    s1, 0(sp)
      add   sp, sp, 8
      ret

```



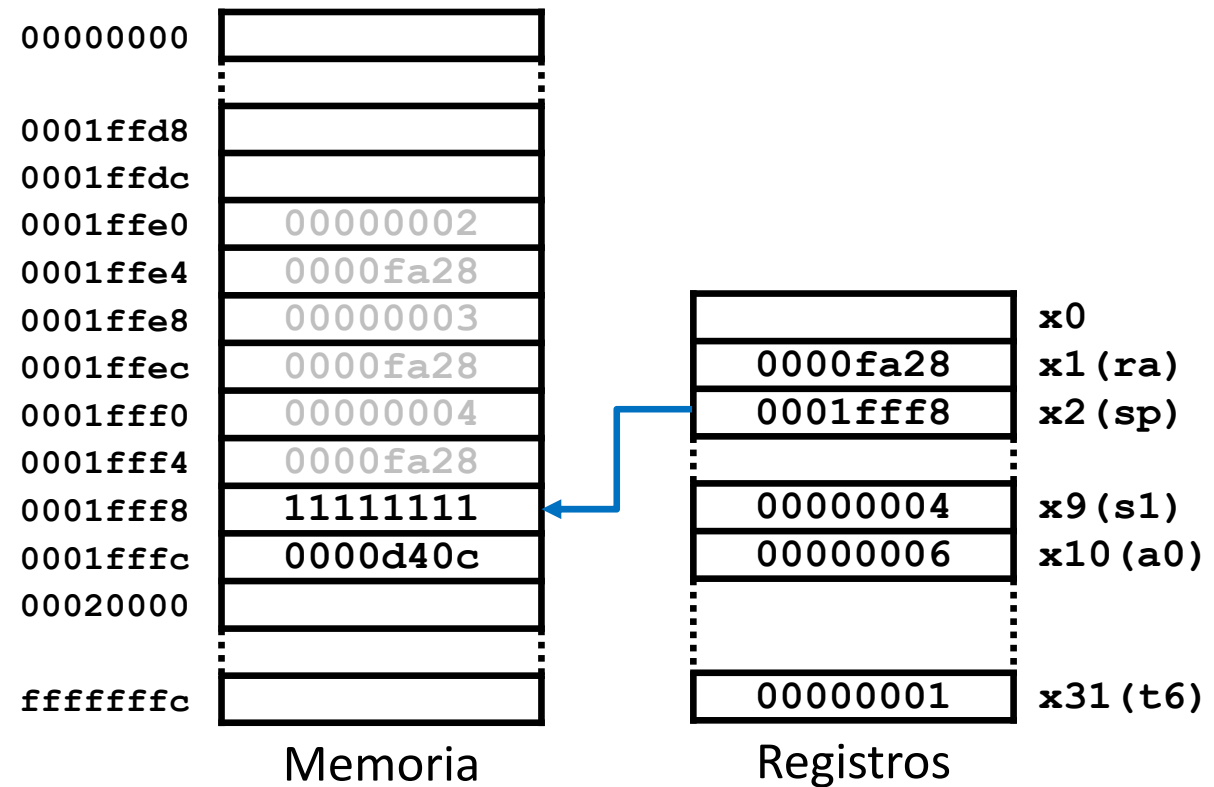


# Funciones

## Anidamiento y recursividad (iv)

ASM

```
a: .space 4
...
li    sp, 0x20000
li    a0, 4
call  fact
0000d40c la    t6, a
      sw    a0, 0(t6)
...
fact:
      add   sp, sp, -8
      sw    ra, 4(sp)
      sw    s1, 0(sp)
      mv    s1, a0
      li    t6, 1
      bgt  s1, t6, else
      li    a0, 1
      j    eif
else:
      add   a0, s1, -1
      call  fact
0000fa28 mul   a0, s1, a0
eif:
      lw    ra, 4(sp)
      lw    s1, 0(sp)
      add   sp, sp, 8
      ret
```



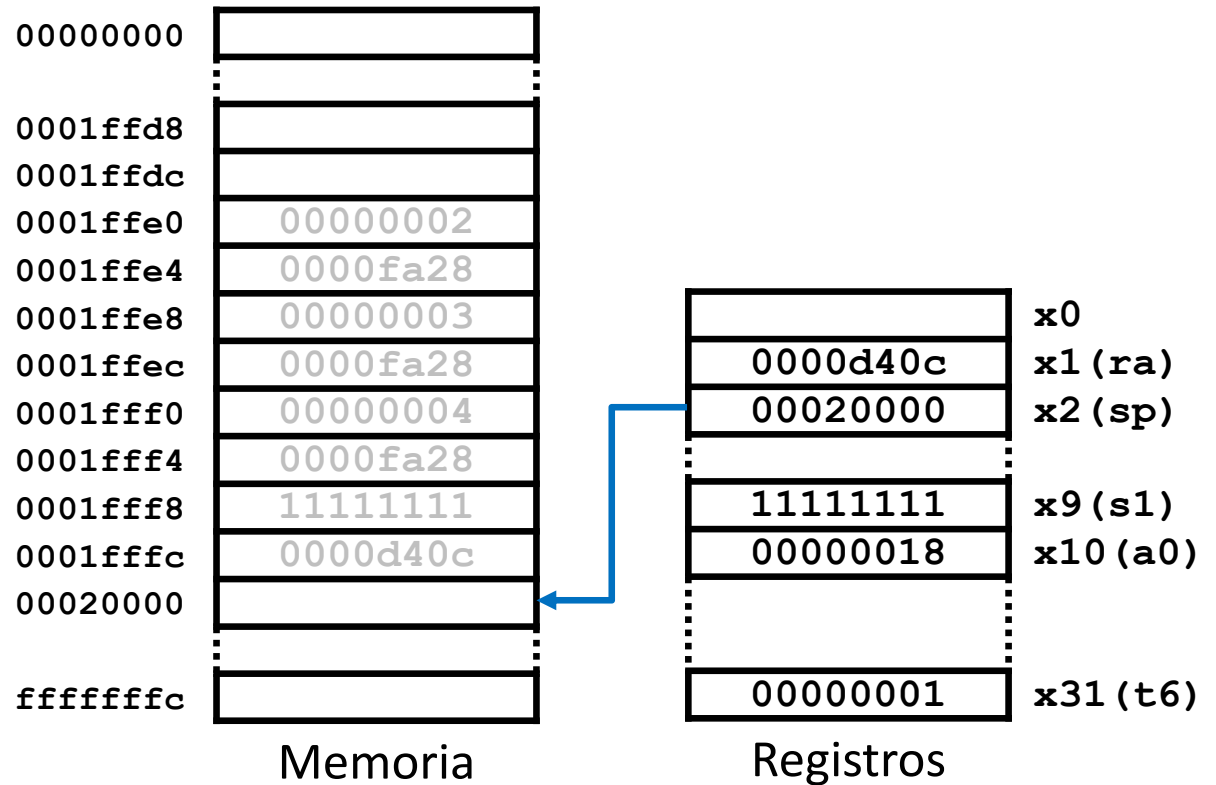


# Funciones

## Anidamiento y recursividad (iv)

ASM

```
a: .space 4
...
li    sp, 0x20000
li    a0, 4
call  fact
0000d40c la    t6, a
      sw    a0, 0(t6)
...
fact:
      add   sp, sp, -8
      sw    ra, 4(sp)
      sw    s1, 0(sp)
      mv    s1, a0
      li    t6, 1
      bgt   s1, t6, else
      li    a0, 1
      j     eif
else:
      add   a0, s1, -1
      call  fact
0000fa28 mul   a0, s1, a0
eif:
      lw    ra, 4(sp)
      lw    s1, 0(sp)
      add   sp, sp, 8
      ret
```

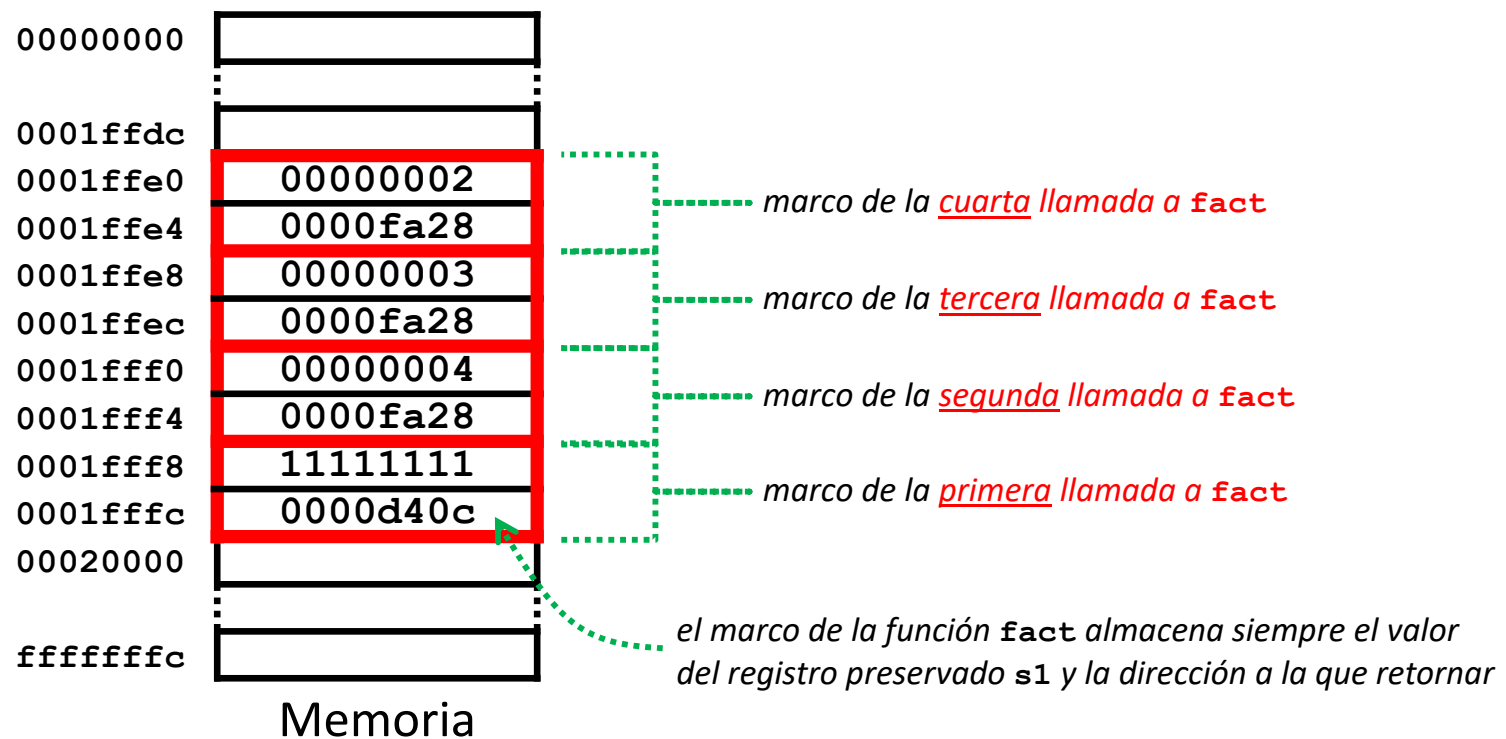




# Funciones

## Marco

- Se denomina **marco** (*frame*) a la **región de la pila** en donde se **ubican los datos que pertenecen** a cada activación de una **función**.
  - El **marco** de toda función **tiene una estructura fija**
    - Contiene el **mismo tipo de información** y en la **misma posición relativa**.
  - Pero en cada activación de la función el marco puede estar ubicado en direcciones efectivas de memoria distintas.





# Funciones

## Paso de gran número de argumentos

- Si el número de parámetros de una función es superior a 8, la función invocante debe apilar el 9º argumento y sucesivos antes de saltar.

C/C++

```
...
y = foo( 1, 2, 3,
        4, 5, 6,
        7, 8, 9 );
...
int foo( int a, int b, int c,
        int d, int e, int f,
        int g, int h, int i )
{
    return a + b + c + d +
           e + f + g + h + i;
}
...
```

ASM

```
...
li    a0, 1
...
li    a7, 8
add   sp, sp, -4
li    t0, 9
sw    t0, 0(sp)
call  foo
add   sp, sp, 4
...
foo:
add   t0, a0, a1
add   t1, a2, a3
add   t2, a4, a5
add   t3, a6, a7
add   t0, t0, t1
add   t2, t2, t3
add   t0, t0, t2
lw    a0, 0(sp)
add   a0, a0, t0
ret
...
```

la función invocante copia los 8 primeros argumentos en los registros a0-a7

la función invocante apila el 9º argumento

la función invocante restaura la cima de la pila

la función invocada lee sus 8 primeros argumentos de los registros a0-a7

la función invocada lee de la pila el 9º argumento



# Funciones

## Variables locales (i)

- Cuando no hay registros suficientes disponibles, las **variables locales** a una función se **ubican en pila** (dentro del marco de la función) por ello:
  - Solo están vivas durante la ejecución de la función.
  - Al no tener direcciones efectivas fijas, **no pueden usarse etiquetas** para referirse a ellas, y se usan **desplazamientos inmediatos** relativos a registro.

C/C++

```
int baz( int a, int b,  
        int c, int d )  
{  
    int sum1, sum2;  
  
    sum1 = (a + b);  
    sum2 = (c + d);  
    ...  
    return sum1 - sum2;  
}
```

$sum1 \rightarrow 4(sp)$     $sum2 \rightarrow 0(sp)$

ASM

```
baz:  
add  sp, sp, -8  
add  t6, a0, a1  
sw   t6, 4(sp)  
add  t6, a2, a3  
sw   t6, 0(sp)  
...  
lw   t5, 4(sp)  
lw   t6, 0(sp)  
sub  a0, t5, t6  
add  sp, sp, 8  
ret
```

reserva espacio en pila para 2 variables locales

almacena a+b en sum1

almacena c+d en sum2

carga sum1

carga sum2

calcula el valor de retorno

libera espacio en pila



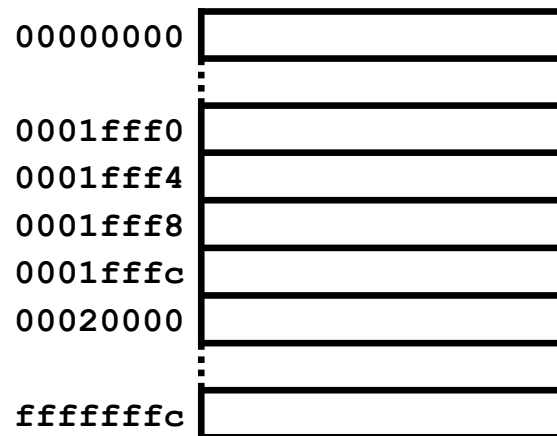
# Funciones

## Variables locales (ii)

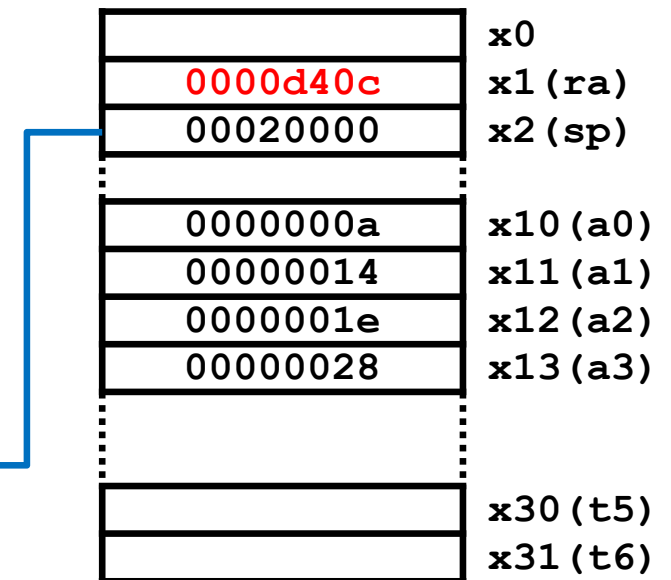
```
...  
li    sp, 0x20000  
li    a0, 10  
li    a1, 20  
li    a2, 30  
li    a3, 40  
call  baz  
...  
baz:  
add   sp, sp, -8  
add   t6, a0, a1  
sw    t6, 4(sp)  
add   t6, a2, a3  
sw    t6, 0(sp)  
...  
lw    t5, 4(sp)  
lw    t6, 0(sp)  
sub   a0, t5, t6  
add   sp, sp, 8  
ret  
...
```

ASM

0000d40c



Memoria



Registros

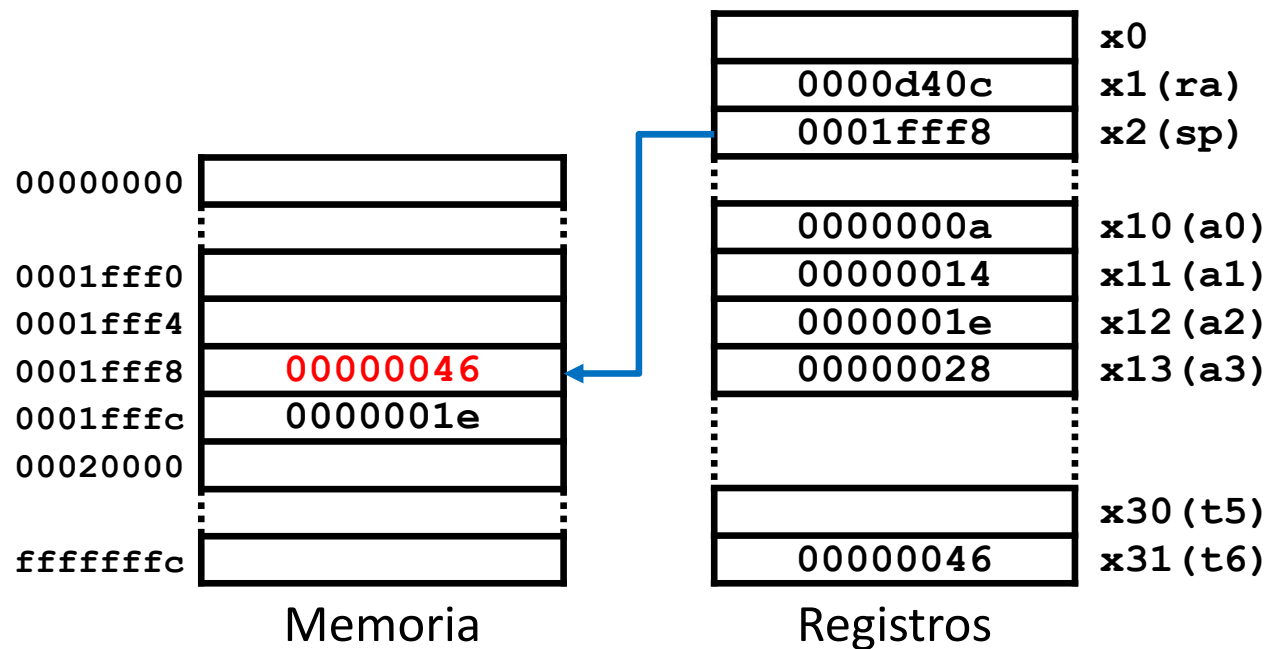


# Funciones

## Variables locales (ii)

```
...  
li    sp, 0x20000  
li    a0, 10  
li    a1, 20  
li    a2, 30  
li    a3, 40  
call  baz  
0000d40c ...  
baz:  
add   sp, sp, -8  
add   t6, a0, a1  
sw    t6, 4(sp)  
add   t6, a2, a3  
sw    t6, 0(sp)  
...  
lw    t5, 4(sp)  
lw    t6, 0(sp)  
sub   a0, t5, t6  
add   sp, sp, 8  
ret  
...
```

ASM





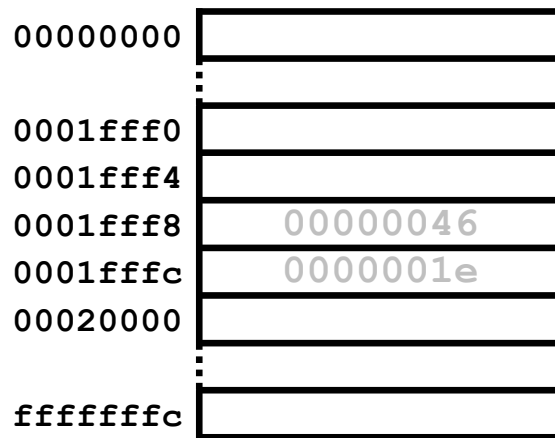


# Funciones

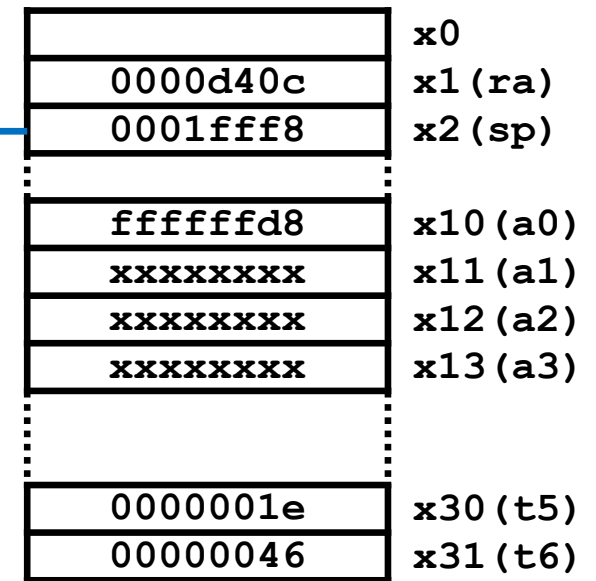
## Variables locales (ii)

ASM

```
...  
li    sp, 0x20000  
li    a0, 10  
li    a1, 20  
li    a2, 30  
li    a3, 40  
call  baz  
0000d40c ...  
baz:  
add   sp, sp, -8  
add   t6, a0, a1  
sw    t6, 4(sp)  
add   t6, a2, a3  
sw    t6, 0(sp)  
...  
lw    t5, 4(sp)  
lw    t6, 0(sp)  
sub   a0, t5, t6  
add   sp, sp, 8  
ret  
...
```



Memoria



Registros





# Funciones

## Variables locales (iii)

- Puede usarse **sp** como registro base, pero tiene riesgos si la pila cambia durante la ejecución de la función:
  - Referencias a una misma variable local en el código de la función podrían tener desplazamientos relativos a **sp** diferentes, haciéndolo poco legible.

```
int baz( int a, int b,  
        int c, int d )  
{  
    int sum1, sum2;  
  
    sum1 = (a + b);  
    sum2 = (c + d);  
    sum1 = foo( 1, 2, 3,  
               4, 5, 6,  
               7, 8, sum1 );  
  
    ...  
    return sum1 - sum2;  
}
```

C/C++

```
baz:  
    add    sp, sp, -8  
    add    t6, a0, a1  
    sw     t6, 4(sp) ←  
    add    t6, a2, a3  
    sw     t6, 0(sp) ←  
    li     a0, 1  
    ...  
    li     a7, 8  
    add    sp, sp, -4 ←  
    lw     t0, 8(sp) ←  
    sw     t0, 0(sp) ←  
    call   foo  
    add    sp, sp, 4 ←  
    ...  
    ret
```

ASM

sum1 está en 4 (sp)  
sum2 está en 0 (sp)  
se copian los 8 primeros argumentos en a0-a7  
se hace hueco para el 9º argumento  
se apila sum1 que ahora está en 8 (sp) si se usara 4 (sp) se apilaría sum2  
tras restaurar la cima de la pila sum1 volverá a estar en 4 (sp)



# Funciones

## Variables locales (iv)

- Para evitar riesgos, se usa **fp** como registro base.
  - Así toda **variable local** podrá tener un **desplazamiento constante y único** que la identifique dentro de la función.
  - **fp** es un **registro preservado** que se inicializa al comienzo de la función a la dirección de cima de la pila y no cambia durante la ejecución de la misma.

ASM

```

baz:
  add sp, sp, -8
  add t6, a0, a1
  sw t6, 4(sp)
  add t6, a2, a3
  sw t6, 0(sp)
  ...
  lw t5, 4(sp)
  lw t6, 0(sp)
  sub a0, t5, t6
  add sp, sp, 8
  ret
  
```

sum1 → 4(sp)  
 sum2 → 0(sp)

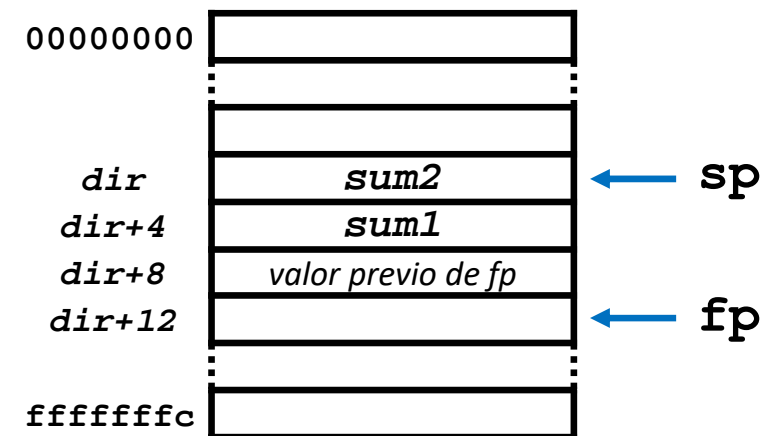
ASM

```

baz:
  add sp, sp, -12
  sw fp, 8(sp)
  add fp, sp, 12
  add t6, a0, a1
  sw t6, -8(fp)
  add t6, a2, a3
  sw t6, -12(fp)
  ...
  lw t5, -8(fp)
  lw t6, -12(fp)
  sub a0, t5, t6
  add sp, sp, 12
  ret
  
```

sum1 → -8(fp)  
 sum2 → -12(fp)

..... reserva espacio para **fp** y 2 variables locales  
 ..... apila **fp** por ser un registro preservado  
 ..... inicializa **fp**



Memoria



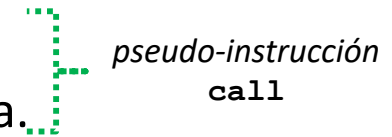
# Funciones

## Gestión del marco (i)

■ La llamada a una función supone:

función invocante

1. Apilar los registros temporales usados por la función invocante.
2. Pasar parámetros de entrada.
3. Salvar la dirección de retorno.
4. Saltar a la dirección de comienzo de la función invocada.



función invocada

5. Apilar de los registros preservados usados por la función invocada.
6. Reservar espacio para variables locales a la función invocada.
7. Inicializar las variables locales.

**prólogo**  
(construcción del marco)

8. Procesar y actualizar parámetros de salida.
9. Salvar el valor de retorno.

**cuerpo**

10. Liberar el espacio ocupado por las variables locales.
11. Desapilar los registros preservados.

**epílogo**  
(destrucción del marco)

12. Saltar a la dirección de retorno. ← pseudo-instrucción `ret`

función invocante

13. Recuperar el valor de retorno.
14. Desapilar los registros temporales.

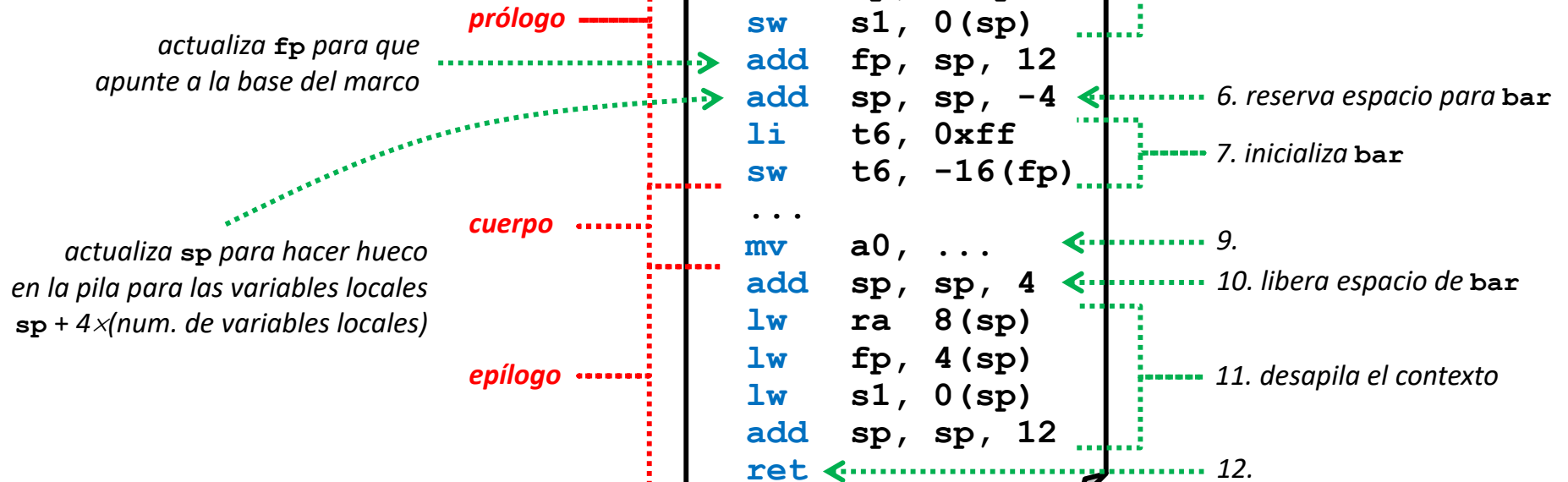


# Funciones

## Marco (ii)

```
C/C++  
int y;  
...  
y = foo( 10, 30 );  
...  
int foo( int a, int b )  
{  
    int bar = 0xff;  
    ...  
}
```

```
ASM  
y: .space 4  
...  
li    a0, 10  
li    a1, 30  
call  foo  
la    t6, y  
sw    a0, 0(t6)  
...  
foo:  
add   sp, sp, -12  
sw    ra, 8(sp)  
sw    fp, 4(sp)  
sw    s1, 0(sp)  
add   fp, sp, 12  
add   sp, sp, -4  
li    t6, 0xff  
sw    t6, -16(fp)  
...  
mv    a0, ...  
add   sp, sp, 4  
lw    ra, 8(sp)  
lw    fp, 4(sp)  
lw    s1, 0(sp)  
add   sp, sp, 12  
ret
```



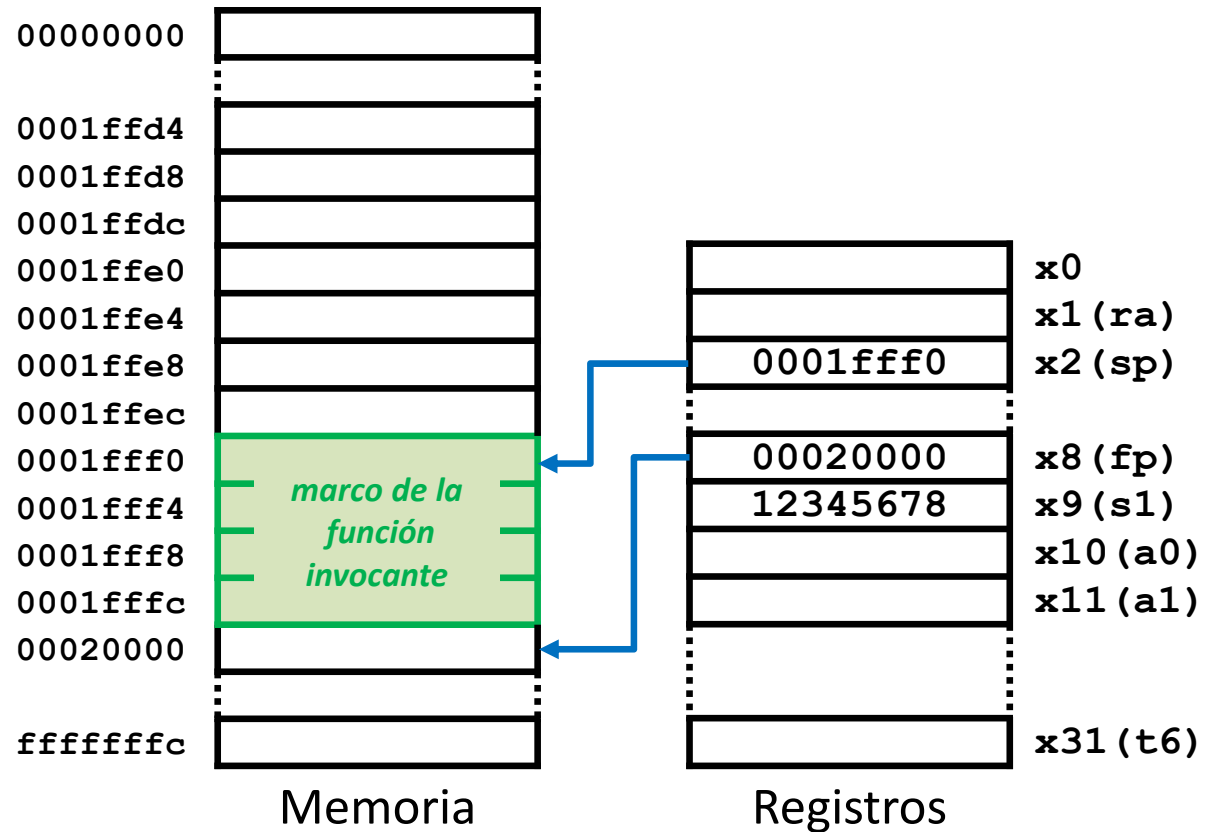


# Funciones

## Gestión del marco (iii)

ASM

```
y: .space 4
...
li a0, 10
li a1, 30
call foo
0000d40c la t6, y
sw a0, 0(t6)
...
foo:
add sp, sp, -12
sw ra 8(sp)
sw fp, 4(sp)
sw s1, 0(sp)
add fp, sp, 12
add sp, sp, -4
li t6, 0xff
sw t6, -16(fp)
...
mv a0, ...
add sp, sp, 4
lw ra 8(sp)
lw fp, 4(sp)
lw s1, 0(sp)
add sp, sp, 12
ret
```



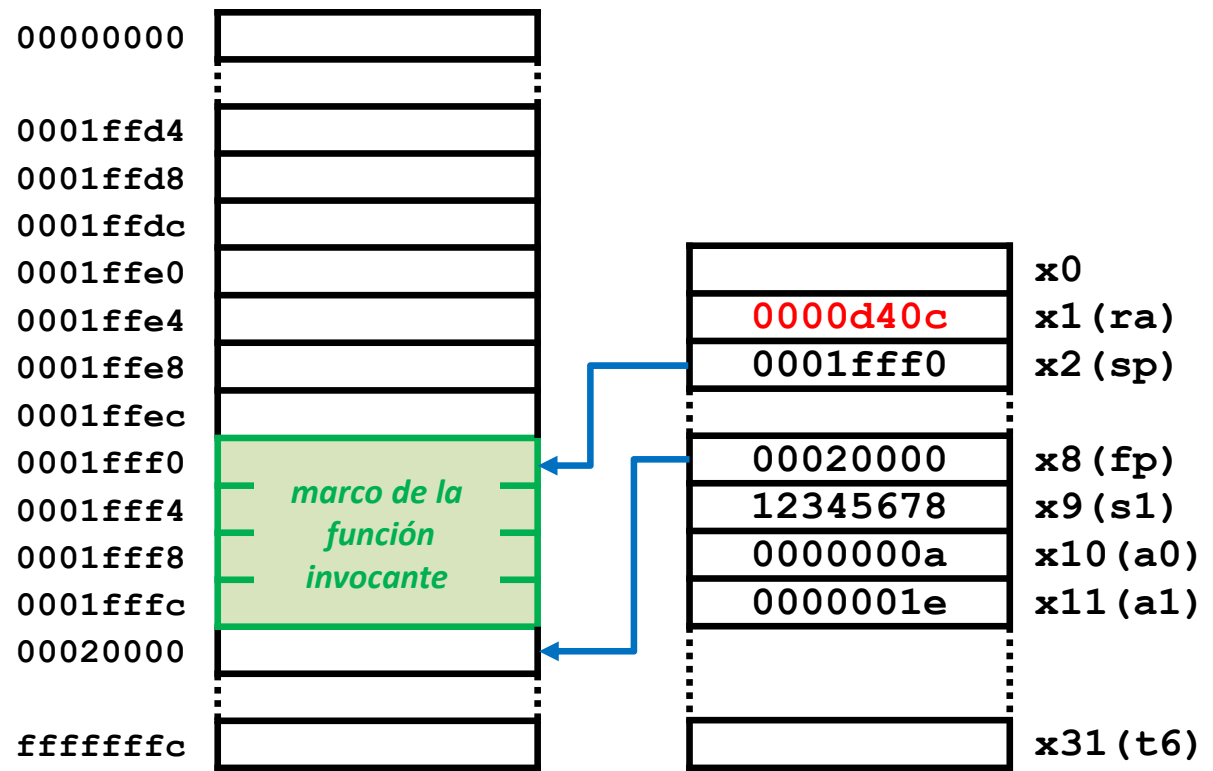
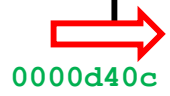


# Funciones

## Gestión del marco (iii)

ASM

```
y: .space 4
...
li a0, 10
li a1, 30
call foo
la t6, y
sw a0, 0(t6)
...
foo:
add sp, sp, -12
sw ra 8(sp)
sw fp, 4(sp)
sw s1, 0(sp)
add fp, sp, 12
add sp, sp, -4
li t6, 0xff
sw t6, -16(fp)
...
mv a0, ...
add sp, sp, 4
lw ra 8(sp)
lw fp, 4(sp)
lw s1, 0(sp)
add sp, sp, 12
ret
```



Memoria

Registros

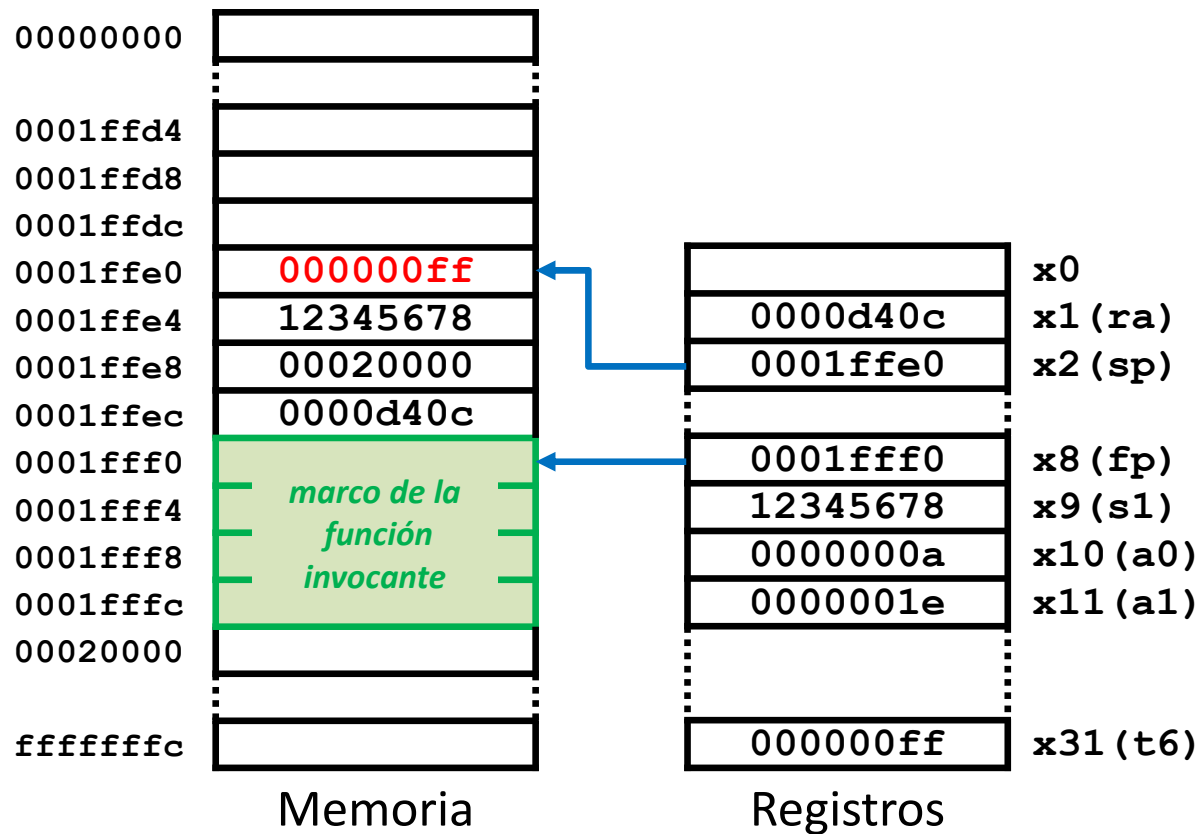


# Funciones

## Gestión del marco (iii)

ASM

```
y: .space 4
...
li a0, 10
li a1, 30
call foo
0000d40c la t6, y
sw a0, 0(t6)
...
foo:
add sp, sp, -12
sw ra 8(sp)
sw fp, 4(sp)
sw s1, 0(sp)
add fp, sp, 12
add sp, sp, -4
li t6, 0xff
sw t6, -16(fp)
...
mv a0, ...
add sp, sp, 4
lw ra 8(sp)
lw fp, 4(sp)
lw s1, 0(sp)
add sp, sp, 12
ret
```





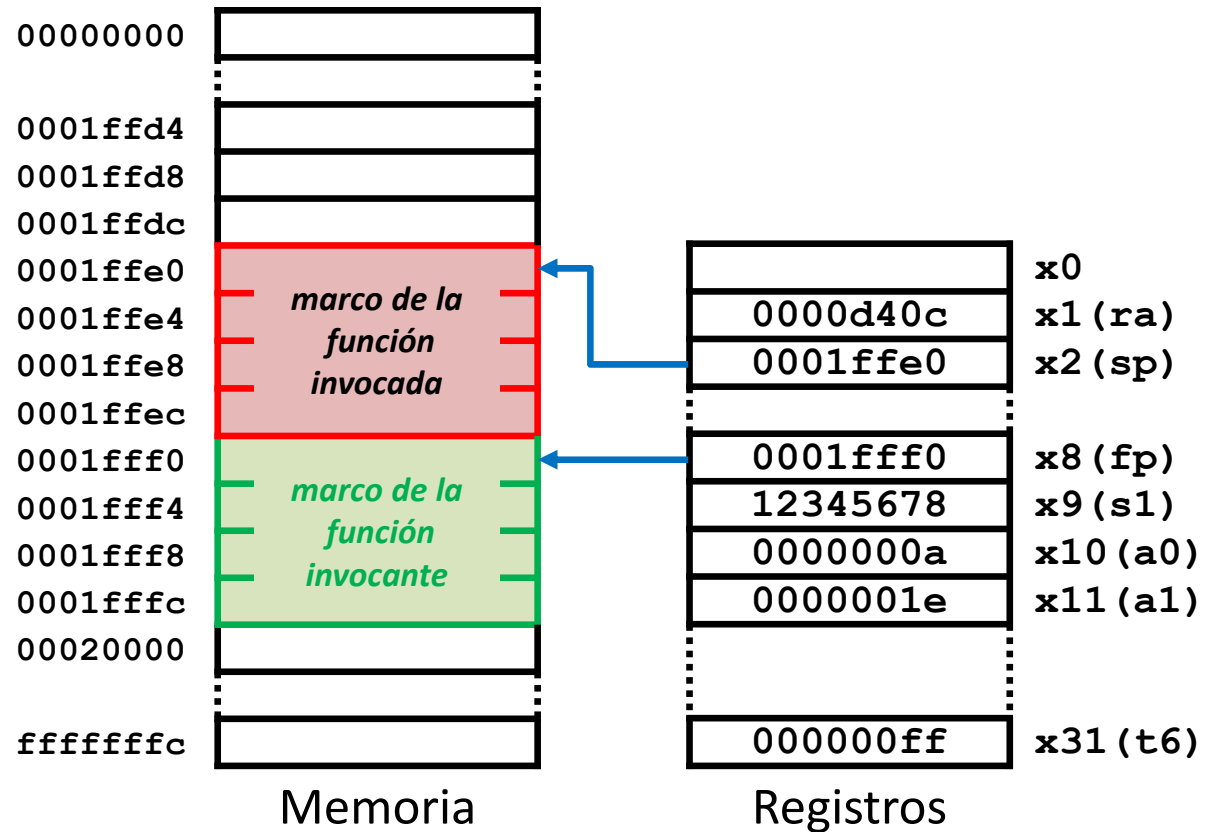


# Funciones

## Gestión del marco (iii)

ASM

```
y: .space 4
...
li a0, 10
li a1, 30
call foo
0000d40c la t6, y
sw a0, 0(t6)
...
foo:
add sp, sp, -12
sw ra 8(sp)
sw fp, 4(sp)
sw s1, 0(sp)
add fp, sp, 12
add sp, sp, -4
li t6, 0xff
sw t6, -16(fp)
...
mv a0, ...
add sp, sp, 4
lw ra 8(sp)
lw fp, 4(sp)
lw s1, 0(sp)
add sp, sp, 12
ret
```





# Funciones

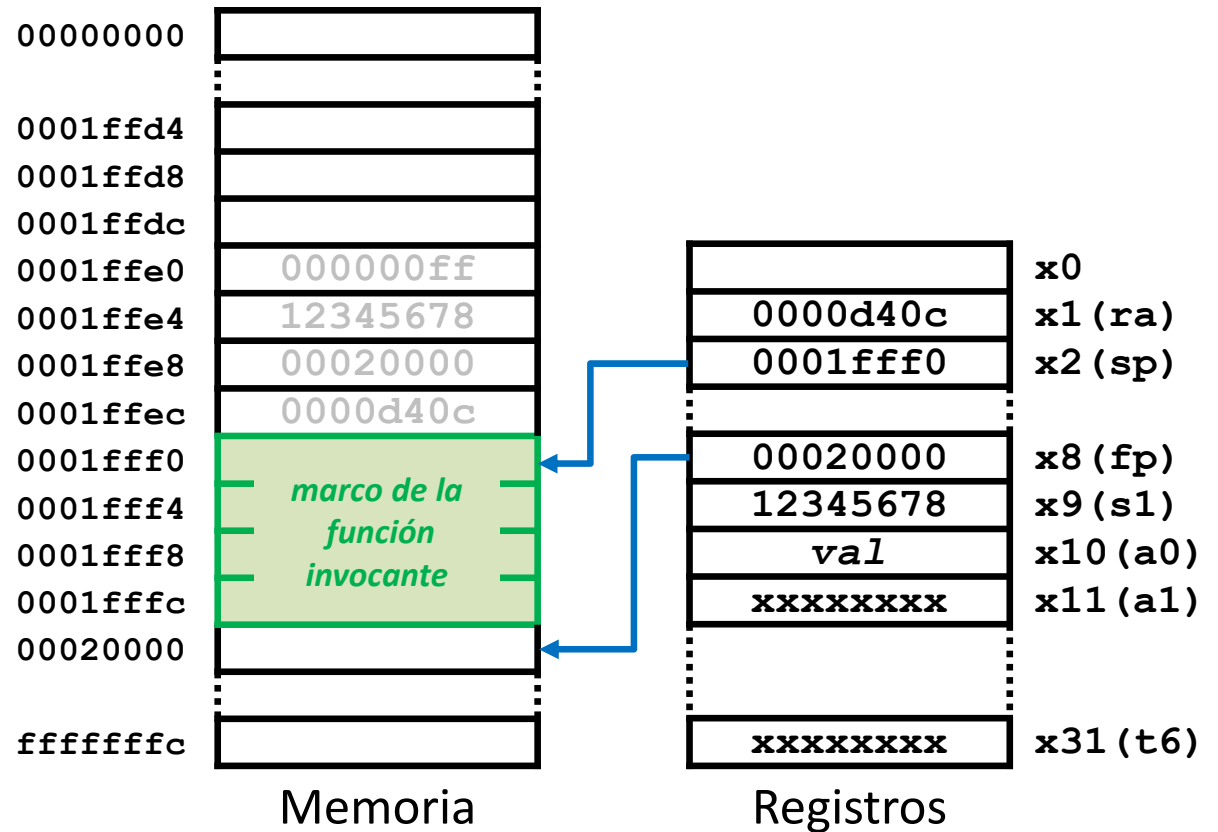
## Gestión del marco (iii)

ASM

```

y: .space 4
...
li a0, 10
li a1, 30
call foo
0000d40c la t6, y
sw a0, 0(t6)
...
foo:
add sp, sp, -12
sw ra 8(sp)
sw fp, 4(sp)
sw s1, 0(sp)
add fp, sp, 12
add sp, sp, -4
li t6, 0xff
sw t6, -16(fp)
...
mv a0, ...
add sp, sp, 4
lw ra 8(sp)
lw fp, 4(sp)
lw s1, 0(sp)
add sp, sp, 12
ret

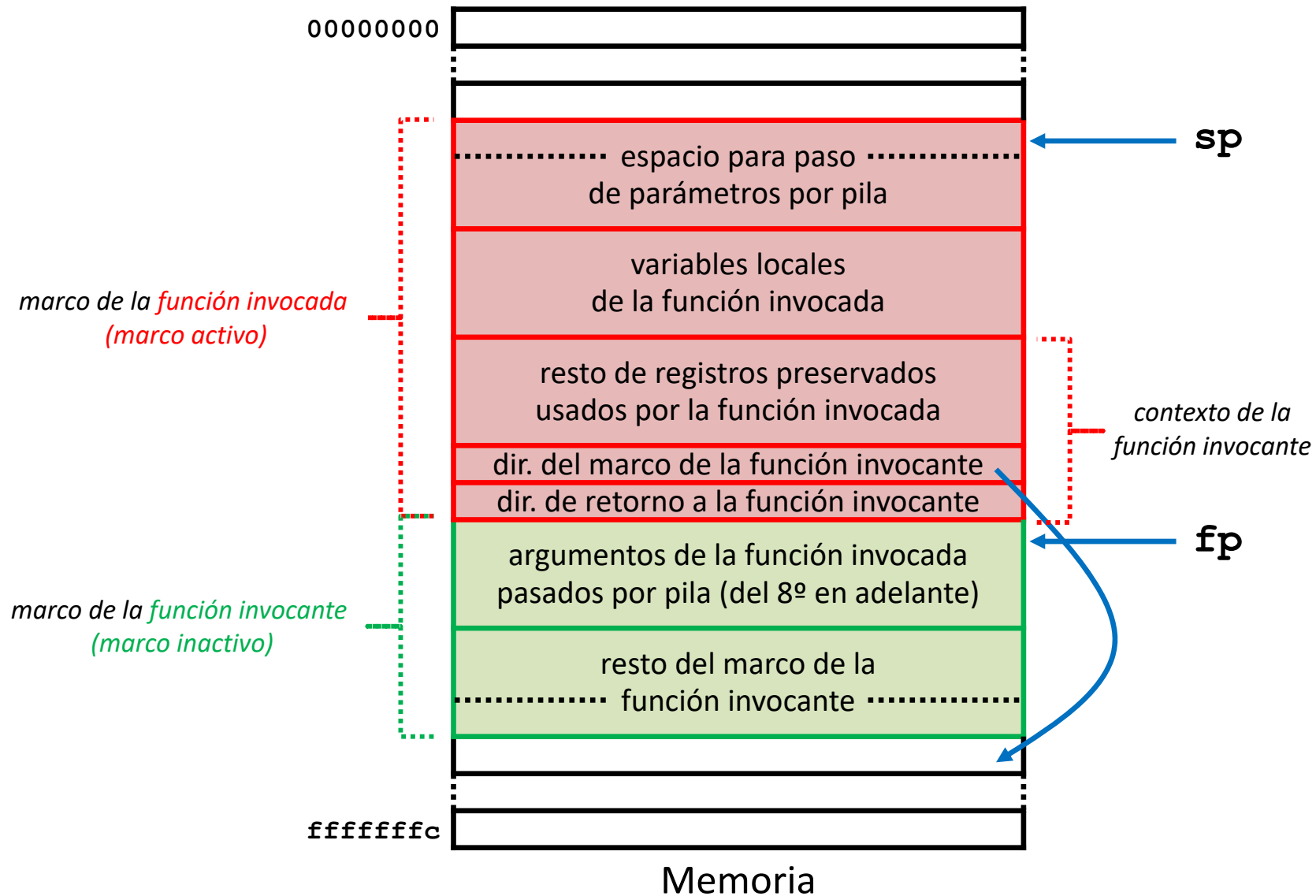
```





# Funciones

## Gestión del marco (iv)





# Funciones

## Gestión del marco (v)

- Según este esquema de llamada:
  - Los **límites de un marco** activo están marcados por **fp** y **sp**.
  - Los **argumentos** a partir del 8º (que la función invocante pasa por pila) tienen siempre **desplazamientos positivos** respecto del **fp**.
  - El **contexto** de la función invocante y las **variables locales** de la función invocada tienen siempre **desplazamientos negativos** respecto del **fp**.
- No obstante, **el esquema se simplifica** según sea la función invocada:
  - Si es de **tipo-hoja**, **ra** no se almacena en su marco.
  - Si **no usa registros preservados**, no hay que salvarlos en su marco.
  - Si **tiene menos de 8 parámetros**, no hay argumentos en el marco de la invocante.
  - Si además, **aloja todas sus variables locales en registros**, es innecesario salvar y usar el **fp**.



# Funciones

## Ejemplo (i)

- Algoritmo de la burbuja para ordenar de los elementos de un array:

C/C++

```
void sort( int v[], int n )
{
    int i, j;

    for( i=0; i<n; i++ )
        for( j=i-1; j>=0 && v[j]>v[j+1]; j-- )
            swap( v, j );
}

void swap( int v[], int k )
{
    int temp;

    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```



# Funciones

## Ejemplo (ii)

- La **función swap** es una **función hoja** y solo **usará registros temporales**.
  - No es necesario que salve el contexto.
  - Almacenará en registro la variable local por lo que no usará **fp**.
- **Recibe 2 argumentos** y **no retorna resultado**.
  - Por **a0** recibe la dirección de la base del array que se está ordenando.
  - Por **a1** recibe el índice del elemento a intercambiar con su siguiente.
  - No debe devolver nada por **a0**.

C/C++

```
void swap( int v[],  
          int k  )  
{  
    int temp;  
  
    temp = v[k];  
    v[k] = v[k+1];  
    v[k+1] = temp;  
}
```

ASM

```
swap:  
    slli t0, a1, 2  ←····· Calcula el desplazamiento k×4  
    add  t0, a0, t0 ←····· Suma base y desplazamiento  
    lw   t1, 0(t0) ←····· Carga v[k] en t1  
    lw   t2, 4(t0) ←····· Carga v[k+1] en t2  
    sw   t2, 0(t0) ←····· Almacena t2 en v[k]  
    sw   t1, 4(t0) ←····· Almacena t1 en v[k+1]  
    ret
```



# Funciones

## Ejemplo (iii)

- La **función sort** es una **función no-hoja** y usará **registros preservados**.
  - Será necesario que salve la dirección de retorno y el contexto.
  - Almacenará en registros las variables locales por lo que no usará **fp**.
- **Recibe 2 argumentos** y **no retorna resultado**.
  - Por **a0** recibe la dirección de la base del array que se quiere ordenar.
  - Por **a1** recibe el número de elementos del array.
  - No debe devolver nada por **a0**.

```
void sort( int v[], int n )
{
    int i, j;

    for( i=0; i<n; i++ )
        for( j=i-1; j>=0 && v[j]>v[j+1]; j-- )
            swap( v, j );
}
```

C/C++



# Funciones

## Ejemplo (iv)

C/C++

```

void sort( int v[], int n )
{
    int i, j;

    for( i=0; i<n; i++ )
        for( j=i-1;
              j>=0 && v[j]>v[j+1];
              j-- )
            swap( v, j );
}

```

$v[] (a0) \rightarrow s1$   
 $n (a1) \rightarrow s2$   
 $i \rightarrow s3$   
 $j \rightarrow s4$

ASM

```

sort:
    ... ← PROLOGO
    mv    s1, a0      ← Se copian los argumentos
    mv    s2, a1
    mv    s3, zero    ← i = 0
fori:
    bge   s3, s2, efori ← ¿ i ≥ n?
    add   s4, s3, -1   ← j = i-1
forj:
    blt   s4, zero, eforj ← ¿ j < 0?
    sll   t0, s4, 2    ← Calcula el desplazamiento j×4
    add   t0, s1, t0   ← Suma base y desplazamiento
    lw    t1, 0(t0)    ← Carga v[j] en t1
    lw    t2, 4(t0)    ← Carga v[j+1] en t2
    ble   t1, t2, eforj ← ¿ v[j] ≤ v[j+1]?
    mv    a0, s1      ← swap( v, j )
    mv    a1, s4
    call  swap
    add   s4, s4, -1   ← Decrementa j
    j     forj
eforj:
    add   s3, s3, 1    ← Incrementa i
    j     fori
efori:
    ... ← EPILOGO
    ret

```





# Funciones

## Ejemplo (v)

C/C++

```

void sort( int v[], int n )
{
    int i, j;

    for( i=0; i<n; i++ )
        for( j=i-1;
              j>=0 && v[j]>v[j+1];
              j-- )
            swap( v, j );
}

```

$v[] (a0) \rightarrow s1$   
 $n (a1) \rightarrow s2$   
 $i \rightarrow s3$   
 $j \rightarrow s4$

ASM

```

sort:
    add sp, sp, -4*5
    sw ra, 4*4(sp)
    sw s1, 3*4(sp)
    sw s2, 2*4(sp)
    sw s3, 1*4(sp)
    sw s4, 0*0(sp)
    ...
fori:
    ...
forj:
    ...
eforj:
    ...
efori:
    lw ra, 4*4(sp)
    lw s1, 3*4(sp)
    lw s2, 2*4(sp)
    lw s3, 1*4(sp)
    lw s4, 0*0(sp)
    add sp, sp, 4*5
    ret

```

PROLOGO:  
Apila contexto

Cuerpo de la función

EPILOGO:  
Desapila contexto



# Variables locales vs. globales

- Acceder a una variable global mediante etiqueta requiere ejecutar entre 2 y 3 instrucciones.

```
a: .word 5
...
la    t0, a
lw    s1, 0(t0)
...
```

```
a: .word 5
...
lw    s1, a
...
```

```
a: .word 5
...
auipc t0, ...
addi  t0, t0, ...
lw    s1, 0(t0)
...
```

```
a: .word 5
...
auipc s1, ...
lw    s1, ...(s1)
...
```

- Para que el **acceso sea más rápido** suelen usarse desplazamientos inmediatos relativos a registro.
  - De manera similar a como se referencian variables locales.

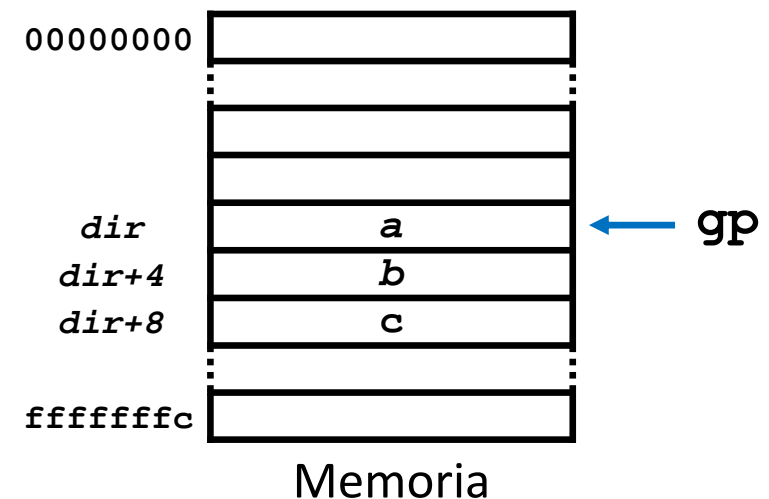


# Variables locales vs. globales

- Para **variables globales** se usa el registro **gp** como base.
  - Al iniciar el programa, **gp** se inicializa para que apunte a la región de memoria donde se ubican las variables globales y nunca cambia.
  - Toda **variable global** podrá identificarse por un **desplazamiento constante y único** relativo a **gp**.

```
ASM
a: .word 76
b: .word -39
c: .space 4
...
la t0, a
lw s1, 0(t0)
la t0, b
lw s2, 0(t0)
add s1, s1, s2
la t0, c
sw s1, 0(t0)
...
```

```
ASM
a: .word 76
b: .word -39
c: .space 4
...
la gp, a
...
lw s1, 0(gp)
lw s2, 4(gp)
add s1, s1, s2
sw s1, 8(gp)
...
```

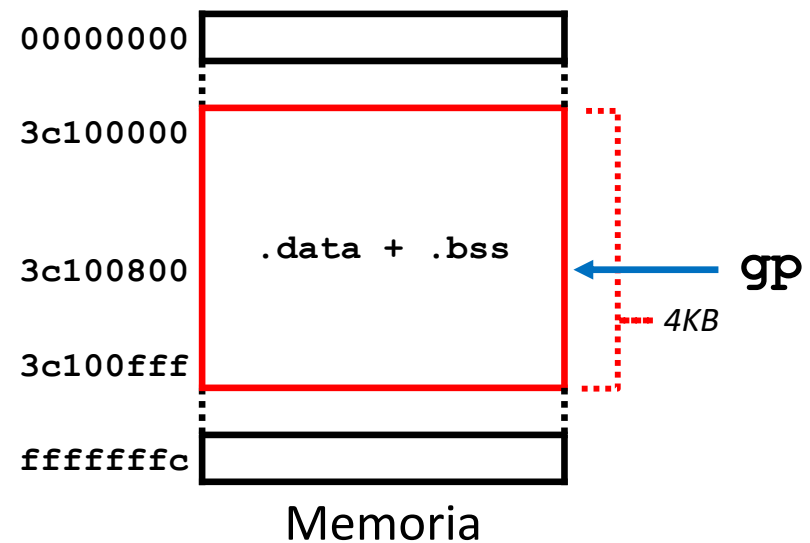
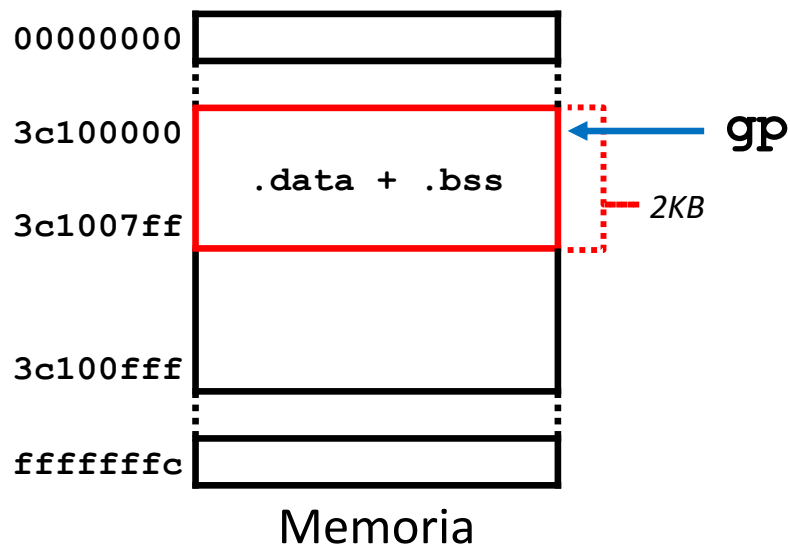


- En **aplicaciones multihebra**, el **tp** se usa de manera análoga para que cada hebra pueda acceder a su espacio local de variables.



# VARIABLES LOCALES VS. GLOBALES

- Los desplazamientos en instrucciones `lw/sw` son de 12b en C2
  - Si `gp` apunta al comienzo de la sección de datos globales, esta región como máximo podrá ser de 2KiB:  $gp + [0..2^{11}-1]$
  - Por ello, lo habitual es inicializar el `gp` para que apunte a la mitad de dicha sección para que pueda direccionarse una región de 4KiB:  $gp \pm [0..2^{11}-1]$ 
    - Es decir, sumando 0x800 a la dirección de inicio de la sección.





# Variables locales vs. globales

- **Variables globales** (estáticas)
  - Se **ubican en memoria principal** en la sección **.data** o **.bss**
  - Tienen **una dirección fija** durante toda la ejecución del programa.
    - Para referirse a ellas se usa una **etiqueta** o un **desplazamiento relativo al gpr**.
  - **Persisten** (están vivas) durante **toda la ejecución del programa**.
    - Se crean (están disponibles) cuando el programa arranca.
    - Se destruyen (su dirección se reutiliza) cuando el programa finaliza.
  
- **Variables locales** (automáticas)
  - Se **ubican en pila**, dentro del **marco de activación** de la función.
    - La pila es una región de memoria principal distinta a código y variables globales.
  - Tienen una **dirección distinta** en cada llamada a la función.
    - Para referirse a ellas se usan **desplazamientos relativos al sp** o al **fp**.
  - **Persisten** (están vivas) solo durante **la ejecución del cuerpo de la función**.
    - Se crean tras la llamada a la función (en el prólogo de la función).
    - Se destruyen antes de volver (en el epílogo de la función).



# Variables locales vs. globales

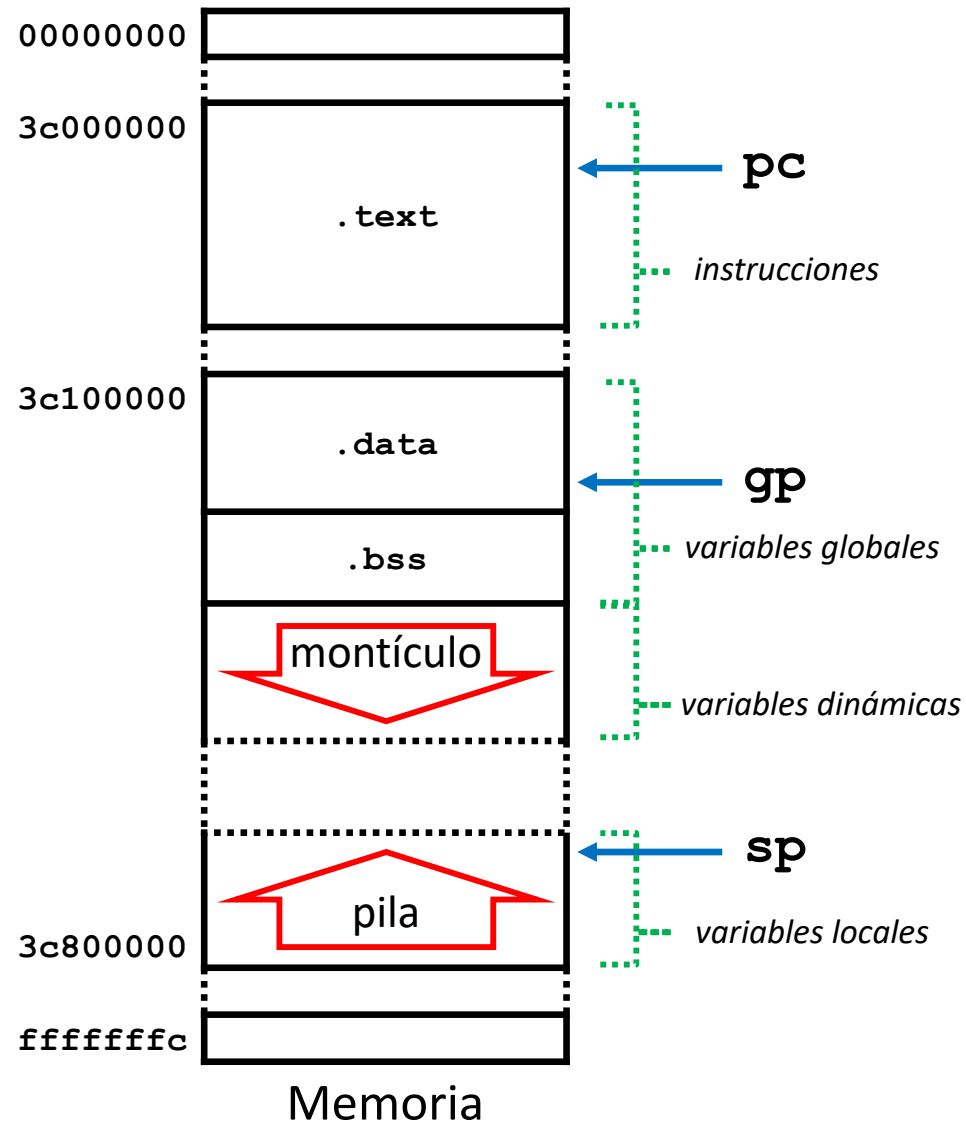
## Variables dinámicas

- En C/C++, además, existen **variables dinámicas**
  - Que el **programador crea y destruye explícitamente** durante la ejecución de un programa haciendo llamadas a **malloc/free** (C) o **new/delete** (C++).
- El **montículo** (*heap*) es una **región de memoria** en donde se ubican las variables dinámicas de un programa.
  - Suele ubicarse en el extremo de memoria opuesto a la pila y crecer en sentido contrario.
- Existen muchas alternativas para gestionar la memoria dinámica en un computador, pero llamadas a las funciones:
  - **malloc/new** devuelven la dirección de una región contigua de memoria libre ubicada en el montículo del tamaño solicitado.
    - A ella se accede con desplazamientos relativos a un registro base en donde previamente se ha almacenando la dirección devuelta por **malloc/new**.
  - **free/delete** marcan como libre la región cuya dirección se indica para que pueda ser reutilizada.



# VARIABLES LOCALES VS. GLOBALES

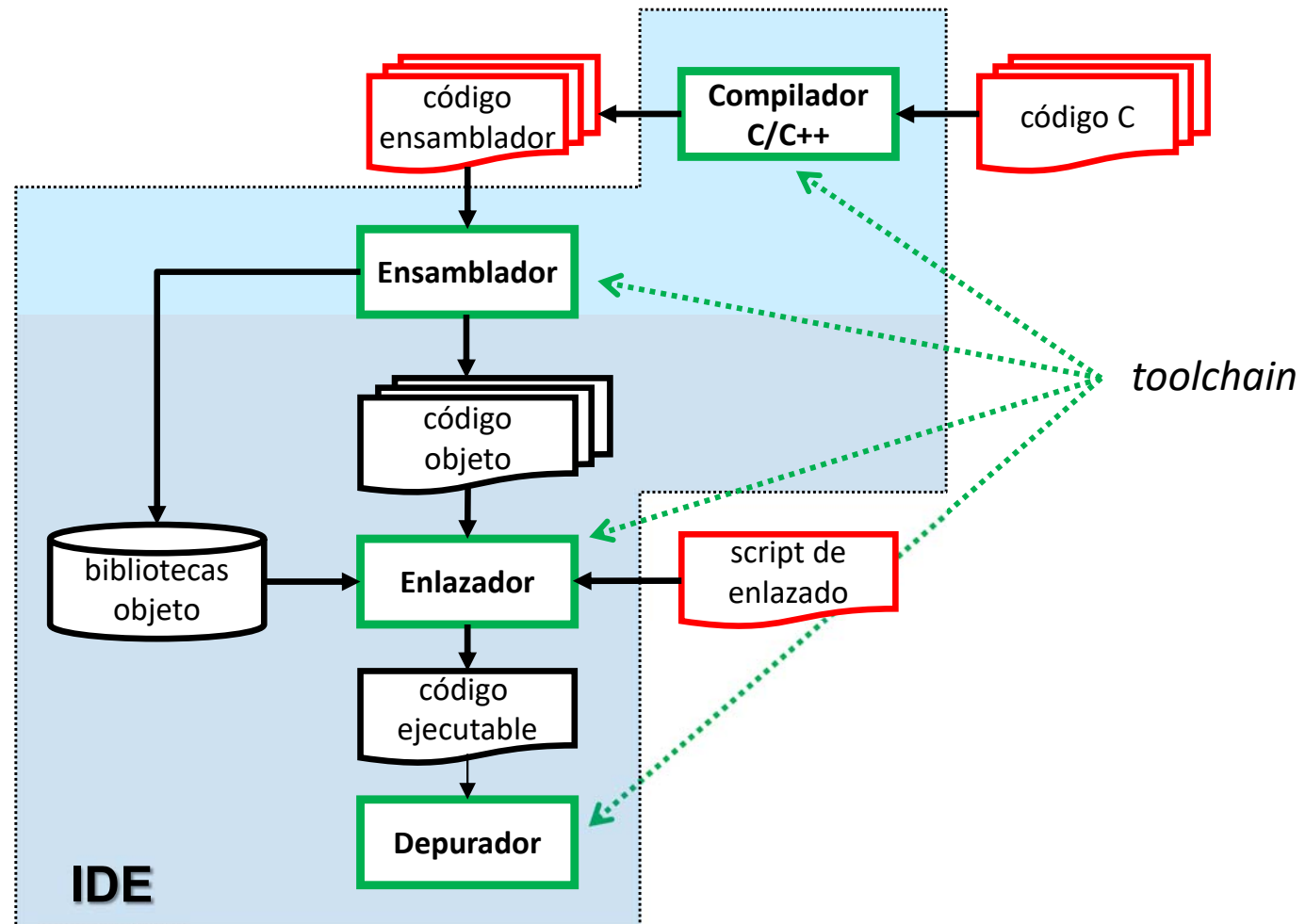
## Mapa de memoria





# Flujo de desarrollo

- Para desarrollar aplicaciones se usa un **IDE** (*Integrated Development Environment*) que actúa como **interfaz de un toolchain**.







# Flujo de desarrollo

- El **ensamblador** (*assembler*):
  - Interpreta las directivas de ensamblaje.
  - Expande pseudo-instrucciones y macros.
  - Transforma etiquetas locales de instrucciones y datos a:
    - Saltos y desplazamientos relativos al PC para que el código sea reubicable.
  - Transforma constantes y expresiones constantes a su representación binaria.
  - Traduce las instrucciones a código máquina.
  - Crea un archivo de código objeto **por cada archivo fuente** conteniendo:
    - Cabecera, secciones, tablas de símbolos e información de depuración.
- El **enlazador** (*linker*) siguiendo las indicaciones de un guión (*script*):
  - Combina secciones de entrada de códigos objeto en secciones de salida.
  - Asigna una región contigua de memoria a cada sección de salida.
  - Resuelve las referencias cruzadas transformando las etiquetas globales.
  - Crea **un único archivo** de código máquina ejecutable.

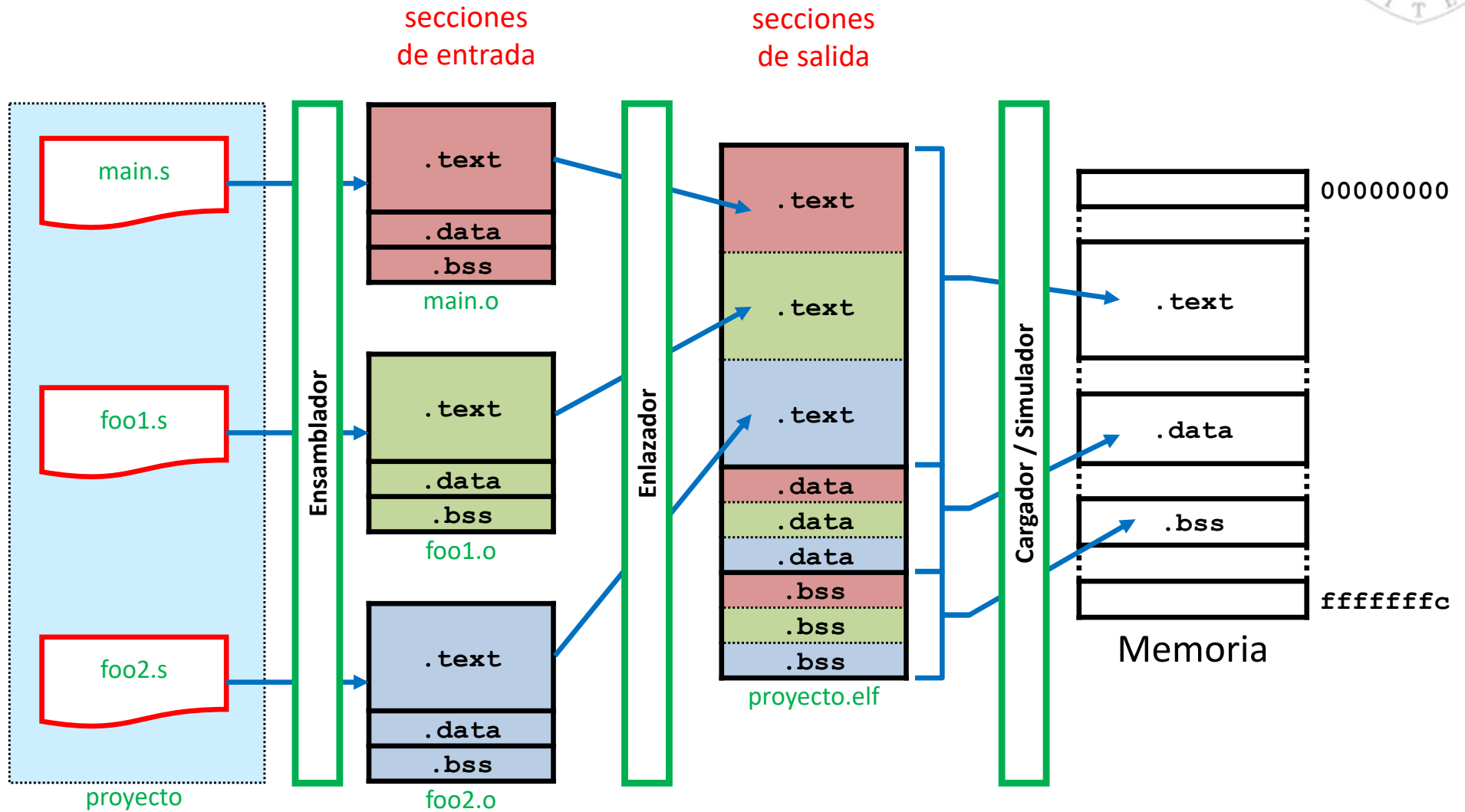


# Flujo de desarrollo

- El **cargador** (loader):
  - Copia en memoria un archivo ejecutable que reside en un dispositivo de almacenamiento secundario y salta la su primera instrucción.
  
- El **depurador** (debugger):
  - Permite ejecutar instrucción a instrucción un programa cargado en memoria e inspeccionar código y datos.
  
- Las aplicaciones pueden desarrollarse en 2 tipos de escenarios
  - **Desarrollo directo**: la aplicación se compila, ensambla y enlaza en el mismo computador en el que se ejecuta y depura.
    - O se ejecuta en un computador con la misma arquitectura.
  - **Desarrollo cruzado**: la aplicación se compila, ensambla y enlaza en un computador distinto del computador donde se ejecuta y depura.
    - Típicamente porque las arquitecturas de ambos computadores son diferentes.



# Flujo de desarrollo

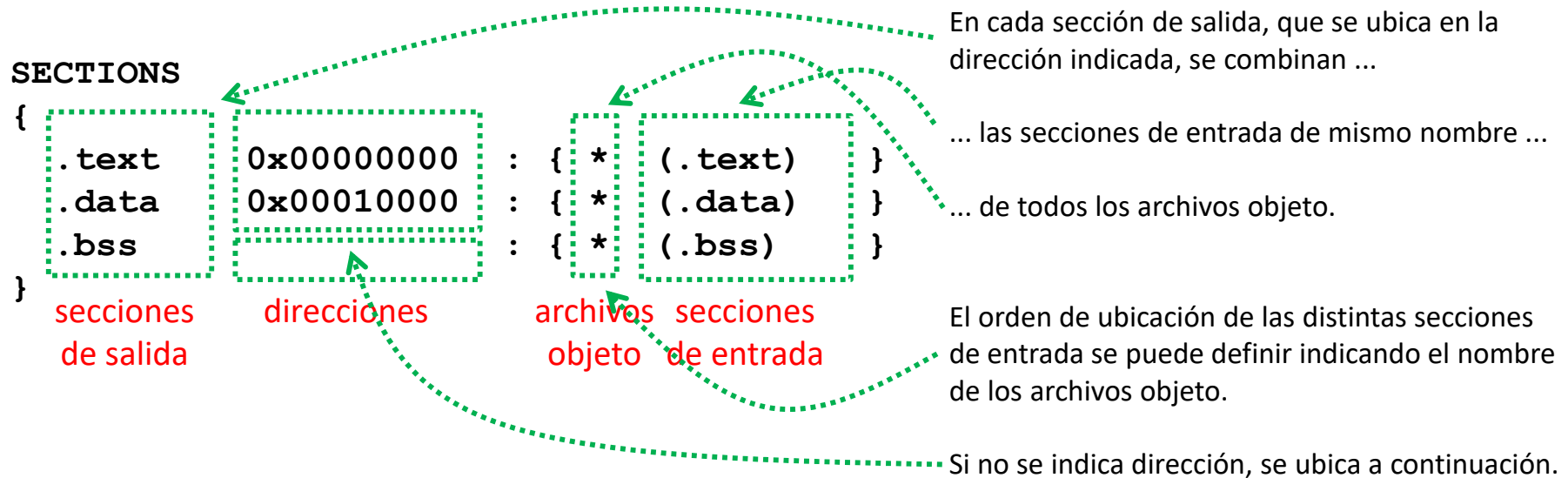




# Flujo de desarrollo

## Guión de enlazado (i)

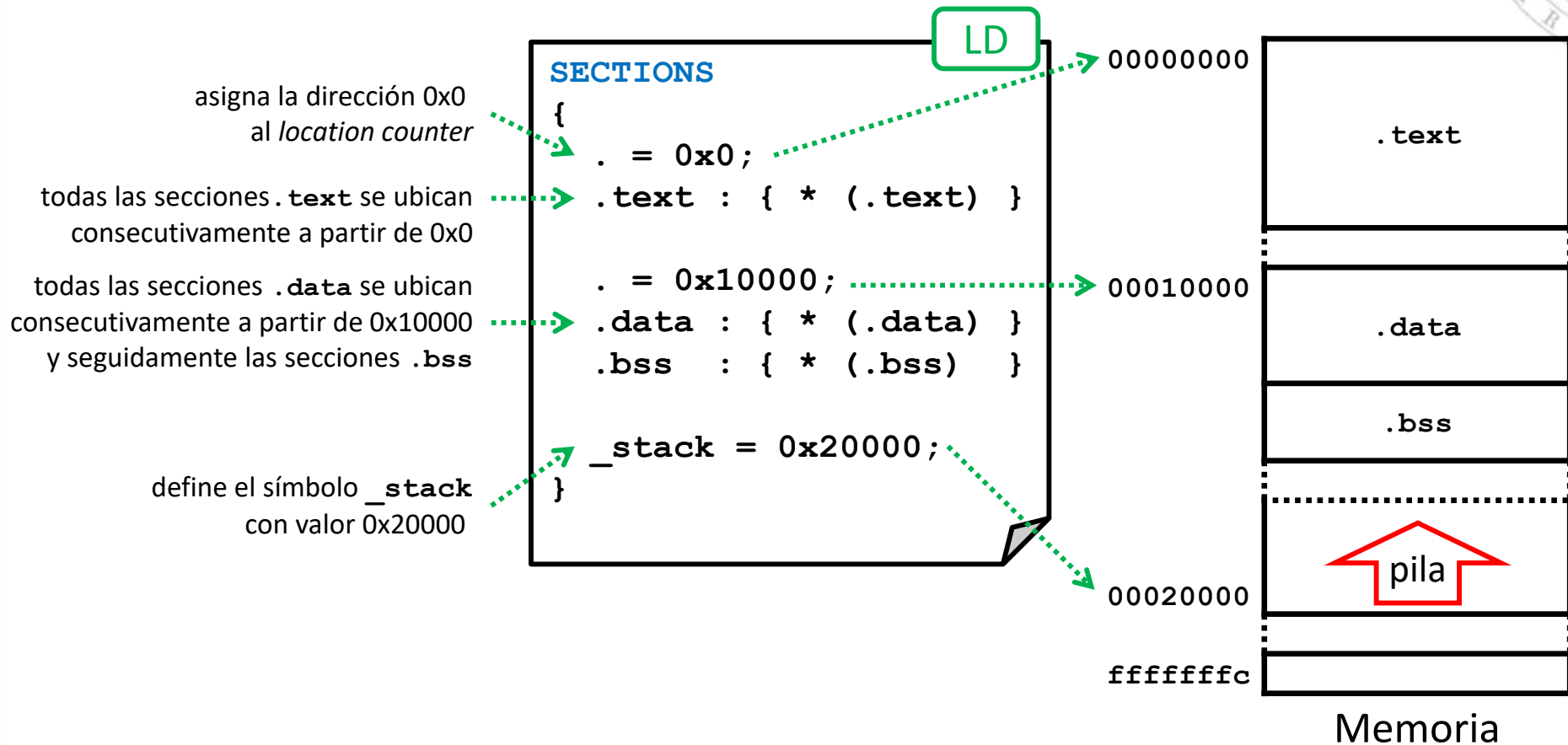
- Un **guión de enlazado** (*linker script*) está formado por una **colección de comandos** que permiten al programador **dirigir el proceso de enlazado**.
  - En el pueden definirse variables referenciables desde código fuente.
  - Está predefinida la variable *location counter* (.) que contiene la dirección actual de enlazado de las secciones de salida.
- El comando más importante es **SECTIONS** que permite definir el **nombre**, **ubicación** y **contenido** de las secciones de salida del ejecutable.





# Flujo de desarrollo

## Guión de enlazado (ii)



- Las direcciones deben elegirse cuidadosamente para evitar errores
  - Desbordamiento de pila (*stack overflow*) sucede cuando en ejecución la pila crece demasiado y sobrescribe otras secciones del programa.



# Flujo de desarrollo

## Combinando ensamblador y C/C++ (i)

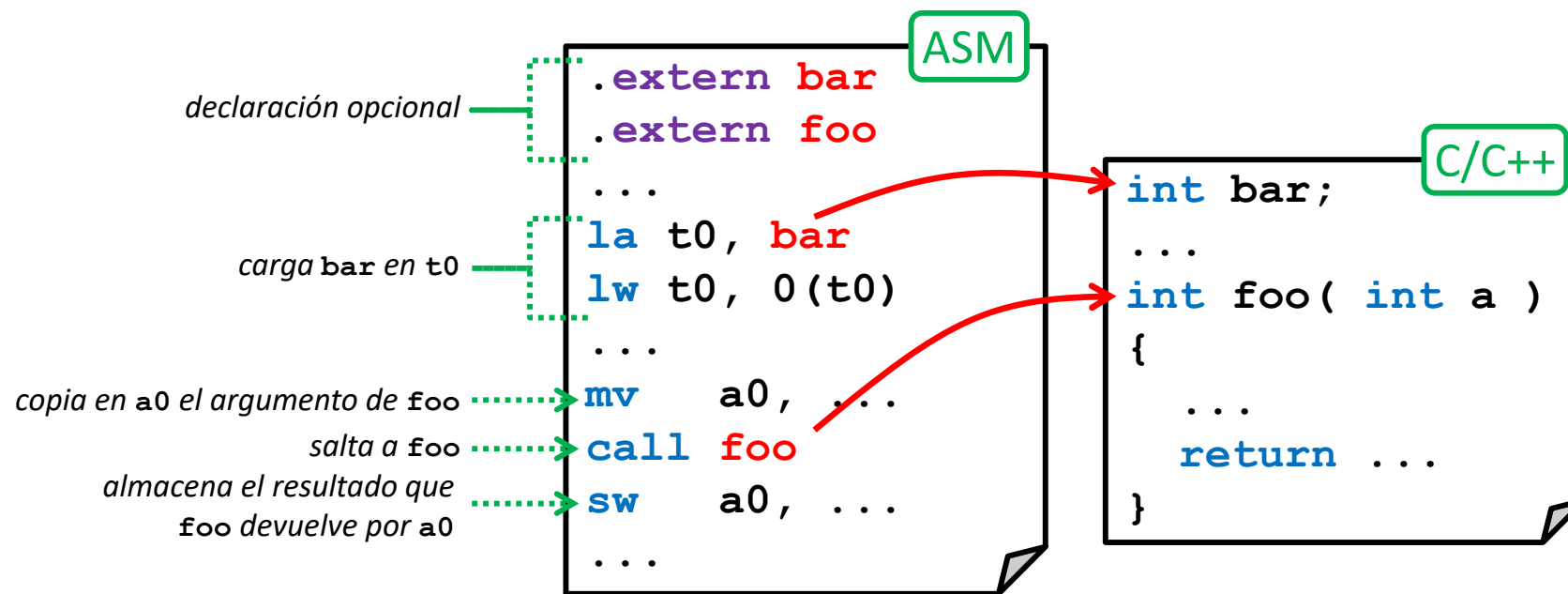
- Lo más común es que un programa este **mayoritariamente escrito en C/C++** y solo una **muy pequeña porción en ensamblador**.
  - Por ello, desde el código ensamblador será necesario poder llamar a funciones y acceder a variables globales declaradas en C/C++ y viceversa.
- Por defecto:
  - Las **funciones y variables globales en C/C++** son visibles desde cualquier archivo del proyecto.
    - No obstante, por legibilidad sus nombres suelen declararse también en el código ensamblador usando la directiva `.extern`
  - Las **funciones y variables globales en ensamblador** son solo visibles dentro del archivo en donde se definen.
    - Para que sean visibles fuera, debe usarse la directiva `.global`
  - El **compilador de C/C++ respeta el convenio de llamadas** definido en RISC-V
    - Si el programador de ensamblador también respeta ese convenio, los códigos C/C++ y ensamblador podrán interoperar entre sí.



# Flujo de desarrollo

## Combinando ensamblador y C/C++ (ii)

- Desde ensamblador se accede a una variable global declarada en C/C++:
  - Usando su identificador como operando.
- Desde ensamblador se llama a una función definida en C/C++:
  - Saltando a su identificador.
  - Los argumentos a la función C/C++ deben pasarse según el convenio RISC-V:
    - Usando los registros a0...a7 para los 8 primeros, y los restantes por pila.
  - La función C/C++ compilada devolverá el resultado en el registro a0.

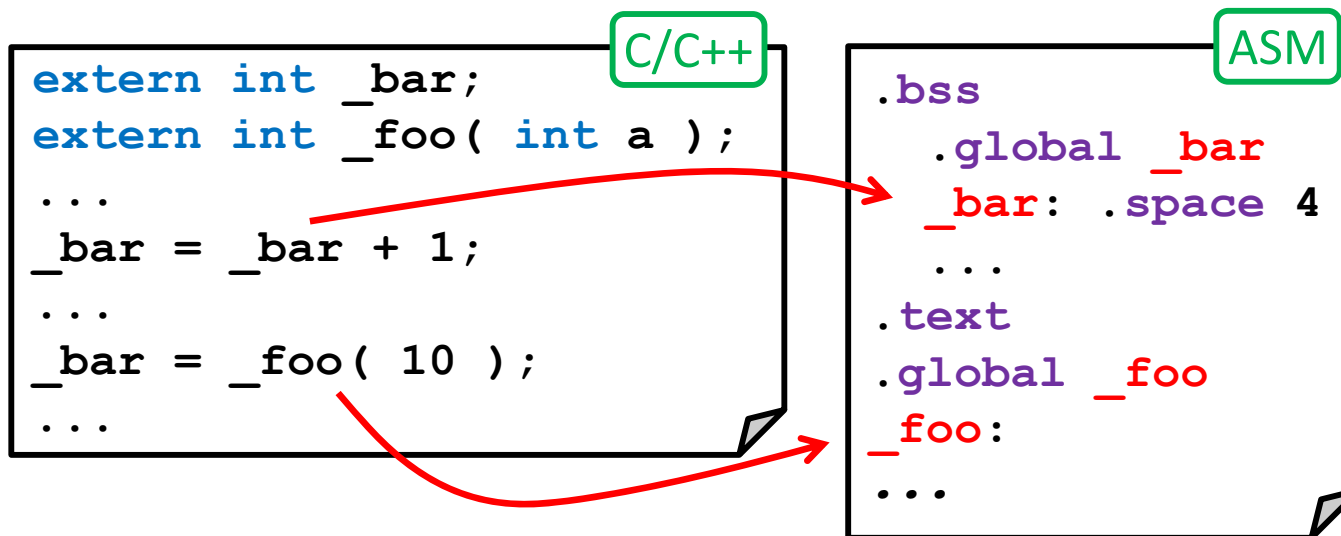




# Flujo de desarrollo

## Combinando ensamblador y C/C++ (iii)

- Desde C/C++ se accede a una **variable global** definida en ensamblador:
  - Declarándola como variable **extern** y usándola normalmente.
- Desde C/C++ se llama a una **función definida en ensamblador**:
  - Declarando su prototipo como **extern** e invocándola normalmente.
  - La llamada en C/C++ compilada pasa los argumentos según el convenio RISC-V:
    - La función en ensamblador encontrará los 8 primeros en los registros **a0...a7** y los restantes en la pila.
  - La función en ensamblador debe devolver el resultado en el registro **a0**.







# Flujo de desarrollo

## Arranque de un programa

- En C/C++ el programa principal `main` se trata como una función mas.
  - Con sus propios argumentos, su valor de retorno y sus variables locales.
  - Es el sistema operativo el que, para ejecutar un programa, llama a su función `main` pasándole los argumentos y recupera el valor que devuelve.
  - Aparte, el sistema operativo ha inicializado previamente el sistema.
- En computadores sin SO (*bare metal*), la inicialización del sistema y el salto a la función `main` lo realiza el código de arranque (*startup code*).

ASM

```
.extern main
.extern _stack
.text
.global start
start:
    la    sp, _stack
    mv    fp, sp
    call main
    j     .
.end
```

Por defecto, el enlazador reconoce la etiqueta `start` como dirección de la primera instrucción del programa

Inicializa `sp`

Inicializa `fp`

Llama a `main` sin argumentos

Si retorna de `main`, queda aquí indefinidamente

# Acerca de *Creative Commons*



## ■ Licencia CC (**Creative Commons**)

- Ofrece algunos derechos a terceras personas bajo ciertas condiciones. Este documento tiene establecidas las siguientes:



**Reconocimiento** (*Attribution*):

En cualquier explotación de la obra autorizada por la licencia hará falta reconocer la autoría.



**No comercial** (*Non commercial*):

La explotación de la obra queda limitada a usos no comerciales.



**Compartir igual** (*Share alike*):

La explotación autorizada incluye la creación de obras derivadas siempre que mantengan la misma licencia al ser divulgadas.

**Más información:** <https://creativecommons.org/licenses/by-nc-sa/4.0/>