

Grupo ARCOS

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

# Tema 3: Fundamentos de la programación en ensamblador (III) **Estructura de Computadores**

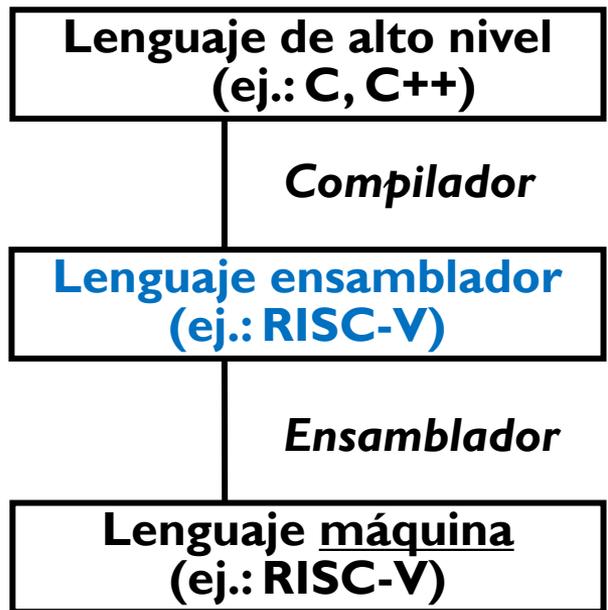
Grado en Ingeniería Informática  
Grado en Matemática aplicada y Computación  
Doble Grado en Ingeniería Informática y Administración de Empresas



# Contenidos

- ▶ Fundamentos básicos de la programación en ensamblador
- ▶ Ensamblador del RISC-V 32, modelo de memoria y representación de datos
- ▶ **Formato de las instrucciones**, modos de direccionamiento y juego de instrucciones
- ▶ Llamadas a procedimientos y uso de la pila

# Diferentes niveles de lenguajes



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

```
lw    t0, 0(x2)
lw    t1, 4(x2)
sw    t1, 0(x2)
sw    t0, 4(x2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

► Una **instrucción en ensamblador** se corresponde con una **instrucción máquina**

► Ejemplo:

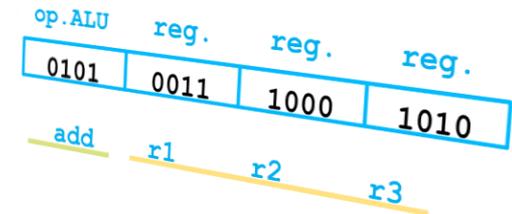
- Ensamblador: `add x5, x6, x2`
- Máquina: `0x002302B3`

# Instrucciones y pseudoinstrucciones RISC-V<sub>32</sub>

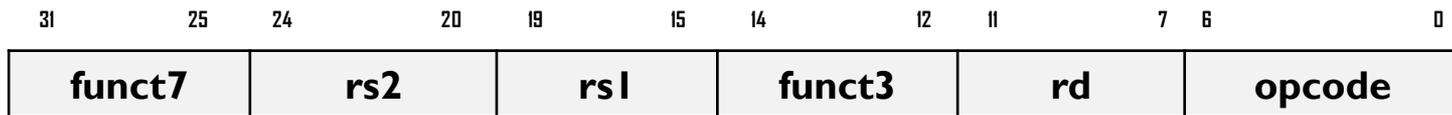
Recordatorio

- ▶ Una **pseudoinstrucción en ensamblador** se corresponde con **una o varias instrucciones de ensamblador**
  - ▶ Ejemplo 1:
    - ▶ La instrucción: `mv x2, x1`
    - ▶ Equivale a: `add x2, zero, x1`
  - ▶ Ejemplo 2:
    - ▶ La instrucción: `li t1, 0x00800010`
    - ▶ No cabe en 32 bits, pero puede usarse como pseudoinstrucción
    - ▶ Es equivalente a:
      - `lui t1, 0x00800`
      - `ori t1, t1, 0x010`
- ▶ Una **instrucción en ensamblador** se corresponde con una **instrucción máquina**
  - ▶ Ejemplo:
    - ▶ Ensamblador: `add x5, x6, x2`
    - ▶ Máquina: `0x002302B3`

# Formato de una instrucción

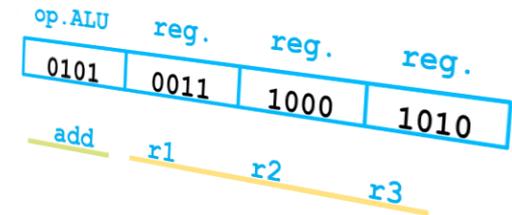


- ▶ Una **instrucción máquina se codifica en binario**:
  - ▶ Su tamaño se **ajusta** a **una** o **varias palabras**
  - ▶ Una instrucción se divide en **campos**
    - ▶ Cada campo codifica un elemento que incluya la instrucción
    - ▶ Puede haber elementos implícitos
  - ▶ Ejemplo de campos en una instrucción del RISC-V:



- ▶ El **formato** especifica, por cada campo de la instrucción:
  - ▶ El **significado** de cada **campo**
  - ▶ **Codificación usada** en cada campo
    - ▶ Binario, complemento a uno, a dos, etc.
  - ▶ El **número de bits** de cada campo
    - ▶ El tamaño de los campos limita el número de valores a codificar

# Formato de una instrucción



- ▶ Una **instrucción máquina** es autocontenida e **incluye**:
  - ▶ Código de operación
  - ▶ Operandos (valor o localización del valor a usar)
  - ▶ Resultado (localización donde guardar)
  - ▶ Dirección de la siguiente instrucción
    - ▶ Implícito:  $PC \leftarrow PC + '4'$  (apuntar a la siguiente instrucción de 32 bits)
    - ▶ Explícito: j 0x01004 (modifica el PC)
- ▶ Normalmente:
  - ▶ Una **arquitectura ofrece unos pocos formatos** de instrucción.
    - ▶ Simplicidad en el diseño de la unidad de control.
  - ▶ **Campos del mismo tipo siempre igual longitud.**
  - ▶ **Selección junto con el código de operación** (ej.: add, addi)
    - ▶ **Normalmente el primer campo.**

# Longitud de formato

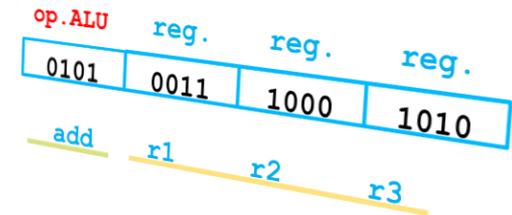
- ▶ La **longitud del formato** es el número de bits para codificar la instrucción
  - ▶ El tamaño habitual es una palabra (o múltiples palabras)
  - ▶ En RISC-V<sub>32</sub> el tamaño de todas las instrucciones es una palabra.
- ▶ Dos tipos:
  - ▶ **Longitud única:**
    - ▶ Todas las instrucciones tienen la misma longitud de formato.
    - ▶ Ejemplos:
      - MIPS32: 32 bits, PowerPC: 32 bits, ...
  - ▶ **Longitud variable:**
    - ▶ Distintas instrucciones tienen distinta longitud de formato.
    - ▶ ¿Cómo se sabe la longitud? → Código de operación
    - ▶ Ejemplos:
      - IA32 (Procesadores Intel): Número variable de bytes.

# Ejemplo: Formato de las instrucciones del RISC-V

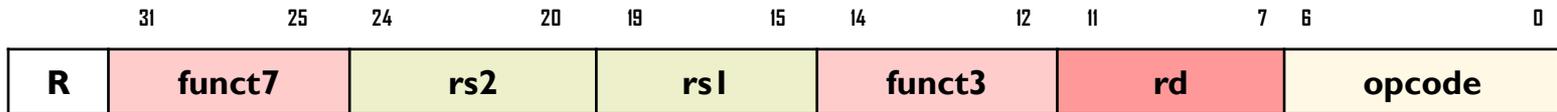
	31		25	24		20	19		15	14		12	11		7	6		0																	
<b>R</b>	funct7					rs2					rs1					funct3					rd					opcode									
<b>I</b>	imm[11:0]															rs1					funct3					rd					opcode				
<b>UI</b>	imm[31:12]															rd					opcode														
<b>S</b>	imm[11:5]					rs2					rs1					funct3					imm[4:0]					opcode									
<b>B</b>	[12]	imm[10:5]					rs2					rs1					funct3					imm[4:1]					[11]	opcode							
<b>J</b>	[20]	imm[10:1]					[11]	imm[19:12]					rd					opcode																	

- **opcode** (7 bits): parcialmente indica el tipo de formato de instrucción.
  - Register, Immediate, Upper Immediate, Store, Branch, Jump
- **funct7+funct3** (10 bits): junto con *opcode*, describen la operación a realizar.
- **rs1** (5 bits): especifica el registro con primer operando.
- **rs2** (5 bits): especifica el registro con segundo operando
- **rd** (5 bits): especifica el registro destino

# Código de operación



- ▶ **Tamaño fijo:**
  - ▶ n bits  $\rightarrow 2^n$  códigos de operación.
  - ▶ m códigos de operación  $\rightarrow \lceil \log_2 m \rceil$  bits.
- ▶ **Campos de extensión**
  - ▶ RISC-V (instrucciones aritméticas-lógicas)
  - ▶  $Op = 0$ ; la instrucción está codificada en `functX`



- ▶ **Tamaño variable:**
  - ▶ Instrucciones más frecuentes = Tamaños más cortos.

# Ubicaciones de los operandos

op. ALU	reg.	reg.	reg.
0101	0011	1000	1010
add	r1	r2	r3

## 1. En la propia instrucción

li t0 0x123

## 2. En registros (CPU)

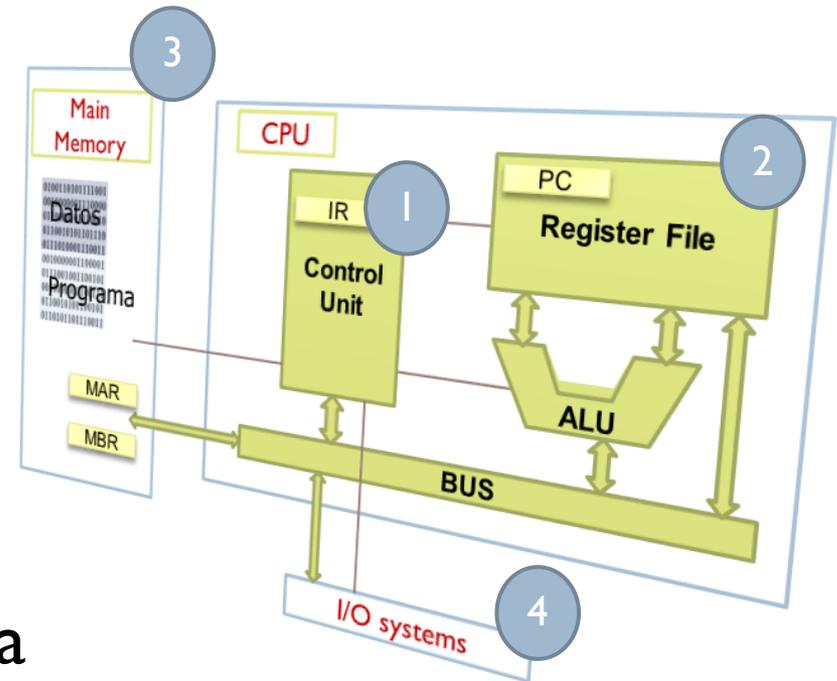
li t0 0x123

## 3. Memoria principal

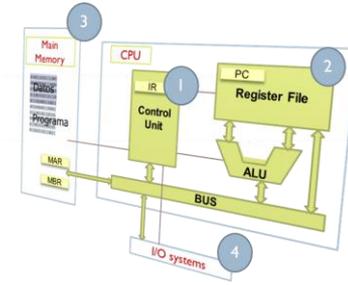
lw t0 address(x0)

## 4. Unidades de Entrada/Salida

in t0 0xFEB



# Ubicaciones de los operandos



## 1. En la propia instrucción

```
li t0 0x123
```

## 2. En registros (CPU)

```
li t0 0x123
```

## 3. Memoria principal

```
lw t0 address(x0)
```

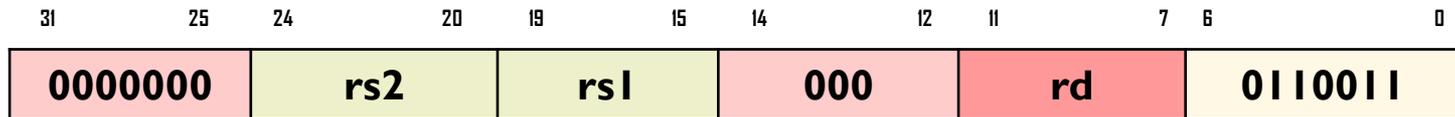
- **num(registro)**: representa la dirección que se obtiene de sumar num con la dirección almacenada en el registro

## 4. Unidades de Entrada/Salida

```
in t0 0xFEB
```

# Ejemplo instrucción y formato asociado en RISC-V

► `add rd rs1 rs2`



# Ejercicio

- ▶ Sea un computador de 16 bits de tamaño de palabra, que incluye un repertorio con 60 instrucciones máquina y con un banco de registros que incluye 8 registros.

Se pide:

Indicar el formato de la instrucción `ADDx R1 R2 R3`, donde R1, R2 y R3 son registros.

# Ejercicio (solución)

palabra -> 16 bits  
60 instrucciones  
8 registros (en BR)  
ADDx R1(reg.), R2(reg.), R3(reg.)

- ▶ Palabra de 16 bits define el tamaño de la instrucción

16 bits



# Ejercicio (solución)

palabra -> 16 bits  
60 instrucciones  
8 registros (en BR)  
ADDx R1(reg.), R2(reg.), R3(reg.)

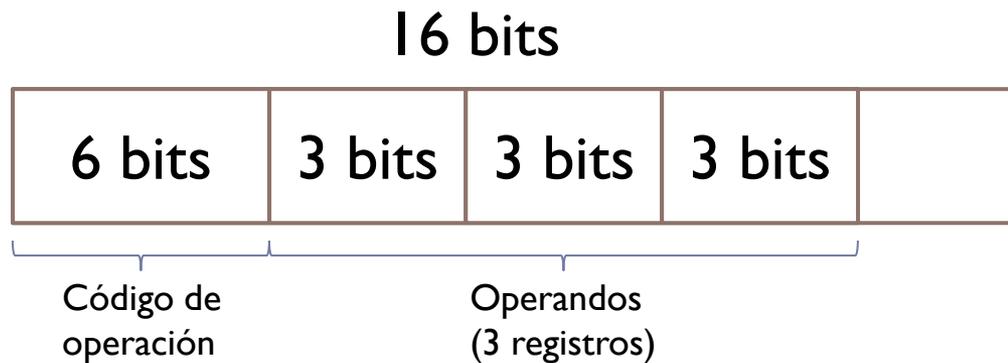
- ▶ Para 60 instrucciones se necesitan 6 bits (mínimo)



# Ejercicio (solución)

palabra -> 16 bits  
60 instrucciones  
8 registros (en BR)  
ADDx R1(reg.), R2(reg.), R3(reg.)

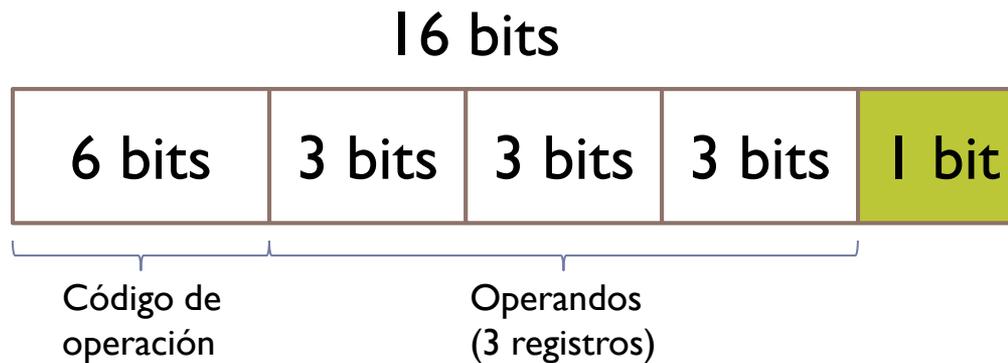
- ▶ Para 8 registros se necesitan 3 bits (mínimo)



# Ejercicio (solución)

palabra -> 16 bits  
60 instrucciones  
8 registros (en BR)  
ADDx R1(reg.), R2(reg.), R3(reg.)

- ▶ Sobra 1 bit ( $16 - 6 - 3 - 3 - 3 = 1$ ), usado de relleno



# Contenidos

- ▶ Fundamentos básicos de la programación en ensamblador
- ▶ Ensamblador del RISC-V<sub>32</sub>, modelo de memoria y representación de datos
- ▶ Formato de las instrucciones, **modos de direccionamiento** y juego de instrucciones
- ▶ Llamadas a procedimientos y uso de la pila

# Modos de direccionamiento

- ▶ El **modo de direccionamiento** es un procedimiento que permite determinar la **ubicación** de un operando, un resultado o una instrucción
  - ▶ Implícito
  - ▶ Inmediato
  - ▶ Directo
    - a registro
    - a memoria
  - ▶ Indirecto
    - a registro
    - a memoria
  - ▶ Relativo
    - a registro índice
    - a registro base
    - a PC
    - a Pila

# Modos de direccionamiento en RISC-V

- ▶ Inmediato value
- ▶ Directo
  - ▶ A memoria address
  - ▶ A registro xr
- ▶ Indirecto
  - ▶ A memoria
  - ▶ A registro (xr)
- ▶ Relativo a
  - registro offset(xr)
  - pila offset(sp)
  - PC beq ... labelI

# Modos de direccionamiento

- ▶ El **modo de direccionamiento** es un procedimiento que permite determinar la **ubicación** de un operando, un resultado o una instrucción

- ▶ **Implícito**

- ▶ **Inmediato**

- ▶ **Directo** {
  - a registro
  - a memoria

- ▶ **Indirecto** {
  - a registro
  - a memoria

- ▶ **Relativo** {
  - a registro índice
  - a registro base
  - a PC
  - a Pila

# Direccionamiento **implícito**

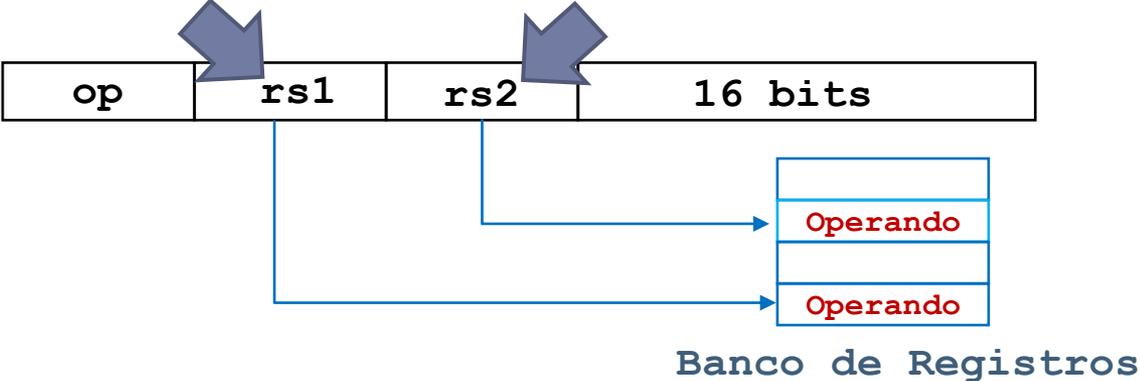
Características	<ul style="list-style-type: none"><li>▶ El operando no está codificado en la instrucción, pero forma parte de esta.</li></ul>
Ejemplo	<ul style="list-style-type: none"><li>▶ <code>auipc a0 0x12345</code><ul style="list-style-type: none"><li>▶ <math>a0 = PC + (0x12345 \ll 12)</math>.</li><li>▶ <code>a0</code> es un operando, <b>PC</b> es el otro (implícito)</li></ul></li></ul>  <p>The diagram shows a horizontal box divided into four sections. The first section is labeled 'op', the second 'rs', and the third '16 bits'. A blue arrow points to the 'rs' section.</p>
(V/I) Ventajas / Inconvenientes	<ul style="list-style-type: none"><li>✓ Es rápido: no es necesario acceder a memoria.</li><li>✗ Pero solo es posible en unos pocos casos.</li></ul>

# Direccionamiento **inmediato**

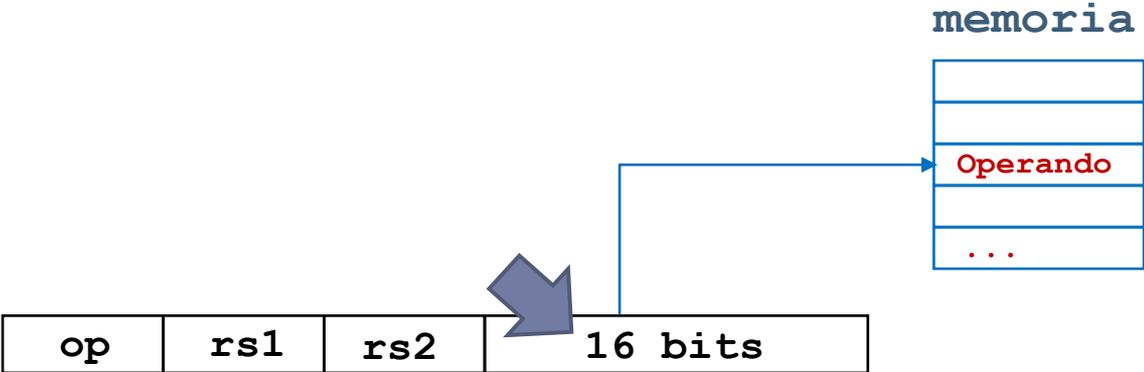
Características	<ul style="list-style-type: none"><li>▶ El operando forma parte de la instrucción.</li></ul>				
Ejemplo	<ul style="list-style-type: none"><li>▶ <code>li a0 0x4f5 l</code><ul style="list-style-type: none"><li>▶ Carga en el registro a0 el valor inmediato <code>0x4f5 l</code>.</li><li>▶ El valor <code>0x00004f5 l</code> está en un campo <b>inmediato</b>.</li></ul></li></ul> <div data-bbox="569 749 1725 821" style="border: 1px solid black; padding: 5px; margin: 10px 0;"><table border="1" style="width: 100%; text-align: center;"><tr><td style="width: 25%;"><code>op</code></td><td style="width: 25%;"><code>rs</code></td><td style="width: 25%;"></td><td style="width: 25%;"><code>16 bits</code></td></tr></table></div>	<code>op</code>	<code>rs</code>		<code>16 bits</code>
<code>op</code>	<code>rs</code>		<code>16 bits</code>		
(V/I) Ventajas / Inconvenientes	<ul style="list-style-type: none"><li>✓ Es rápido: no es necesario acceder a memoria.</li><li>✗ No siempre cabe el valor en una palabra:<ul style="list-style-type: none"><li>▶ No cabe en 32 bits, es equivalente a:<ul style="list-style-type: none"><li>□ <code>lui t1, 0x87654</code></li><li>□ <code>ori t1, t1, 0x321</code></li></ul></li></ul></li></ul>				

# Direccionamiento **directo a registro**

## direccionamiento de registro

Características	<ul style="list-style-type: none"><li>▶ El operando se encuentra en el registro.</li></ul>
Ejemplo	<ul style="list-style-type: none"><li>▶ <code>mv a0 a1</code></li><li>▶ Copia en el registro <code>a0</code> el valor que hay en el registro <code>a1</code>.</li><li>▶ El identificador de <code>a0</code> y <code>a1</code> está codificado en la instrucción.</li></ul>  <p style="text-align: right;">Banco de Registros</p>
(V/I) Ventajas / Inconvenientes	<ul style="list-style-type: none"><li>✗ El número de registros está limitado.</li><li>✓ Acceso a registros es rápido</li><li>✓ El número de registros es pequeño =&gt; pocos bits para su codificación, instrucciones más cortas</li></ul>

# Direccionamiento **directo a memoria**

Características	<ul style="list-style-type: none"><li>▶ El operando se encuentra en memoria, y la dirección está codificada en la instrucción (no disponible en RISC-V).</li></ul>
Ejemplo	<ul style="list-style-type: none"><li>▶ <b>LD .RI #0xFFF0 # IEEE 694</b><ul style="list-style-type: none"><li>▶ Carga en RI la palabra almacenada en <b>0xFFF0</b>.</li></ul></li></ul> 
(V/I) Ventajas / Inconvenientes	<ul style="list-style-type: none"><li>✗ Acceso a memoria es más lento comparado con los registros</li><li>✗ Direcciones largas =&gt; instrucciones más largas</li><li>✓ Acceso a un gran espacio de direcciones (capacidad &gt; B.R.)</li></ul>

# Modos de direccionamiento

- ▶ El **modo de direccionamiento** es un procedimiento que permite determinar la **ubicación** de un operando, un resultado o una instrucción
  - ▶ Implícito
  - ▶ Inmediato
  - ▶ Directo
    - a registro
    - a memoria
  - ▶ Indirecto
    - **a registro**
    - **a memoria**
  - ▶ Relativo
    - a registro índice
    - a registro base
    - a PC
    - a Pila

# Direccionamiento directo vs. indirecto



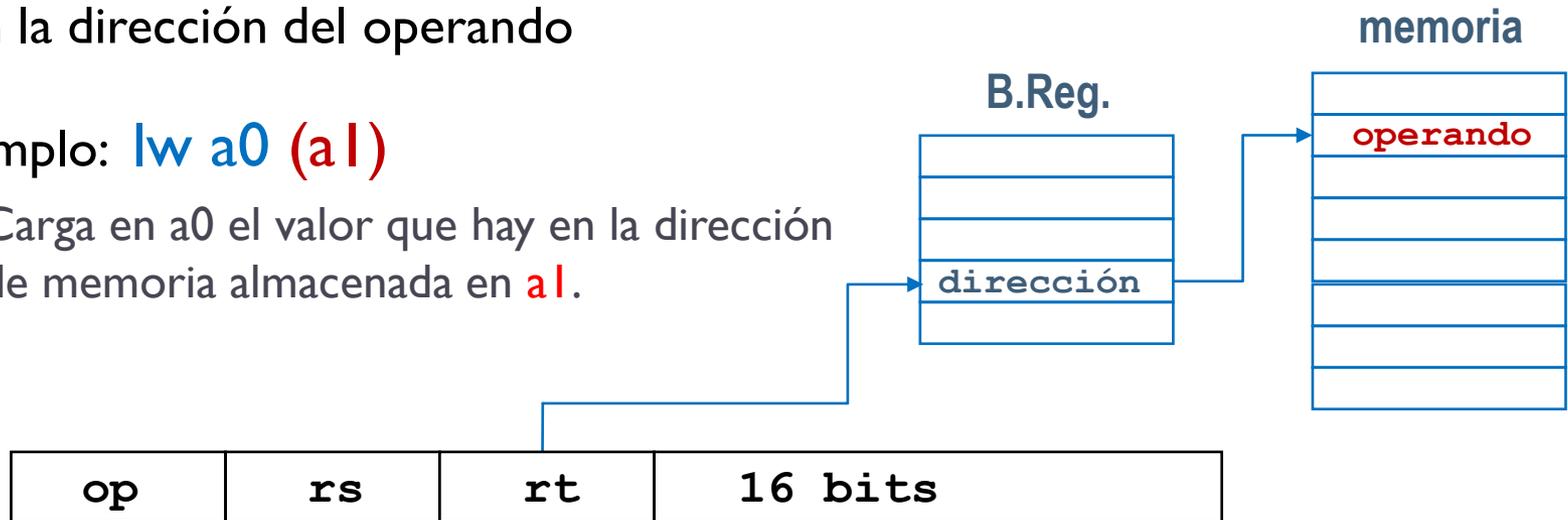
- ▶ En el direccionamiento directo se indica **dónde está el operando**:
  - ▶ En qué registro o en qué posición de memoria
- ▶ En el indirecto se indica **dónde está la dirección del operando**:
  - ▶ Hay que acceder a esa dirección en memoria
  - ▶ Se incorpora un nivel (o varios) de direccionamiento

# Direccionamiento indirecto de registro

- ▶ Se indica en la instrucción el registro con la dirección del operando

▶ Ejemplo: `lw a0 (a1)`

- ▶ Carga en a0 el valor que hay en la dirección de memoria almacenada en a1.



▶ V/I

✓ Amplio espacio de direcciones, instrucciones cortas

- ▶ Pseudo-instrucción equivalente a `lw a0 0(a1)`

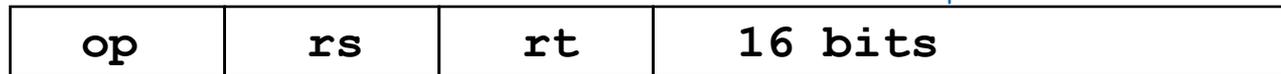
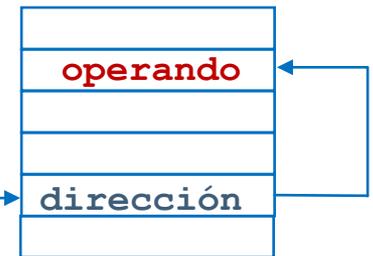
# Direccionamiento indirecto a memoria

- ▶ Se indica en la instrucción la dirección donde está la de la dirección del operando (no disponible en RISC-V)

▶ Ejemplo: `LD .RI [#DIR] # IEEE 694`

- ▶ Carga en RI el valor que hay en la dirección de memoria que está almacenada en la dirección de memoria `#DIR`.

memoria



- ▶ V/I

✓ Amplio espacio de direcciones

✓ El direccionamiento puede ser anidado, multinivel o en cascada

- ▶ Ejemplo: `LD .RI [[[.RI]]]`

✗ Puede requerir varios accesos memoria

✗ instrucciones más lentas de ejecutar

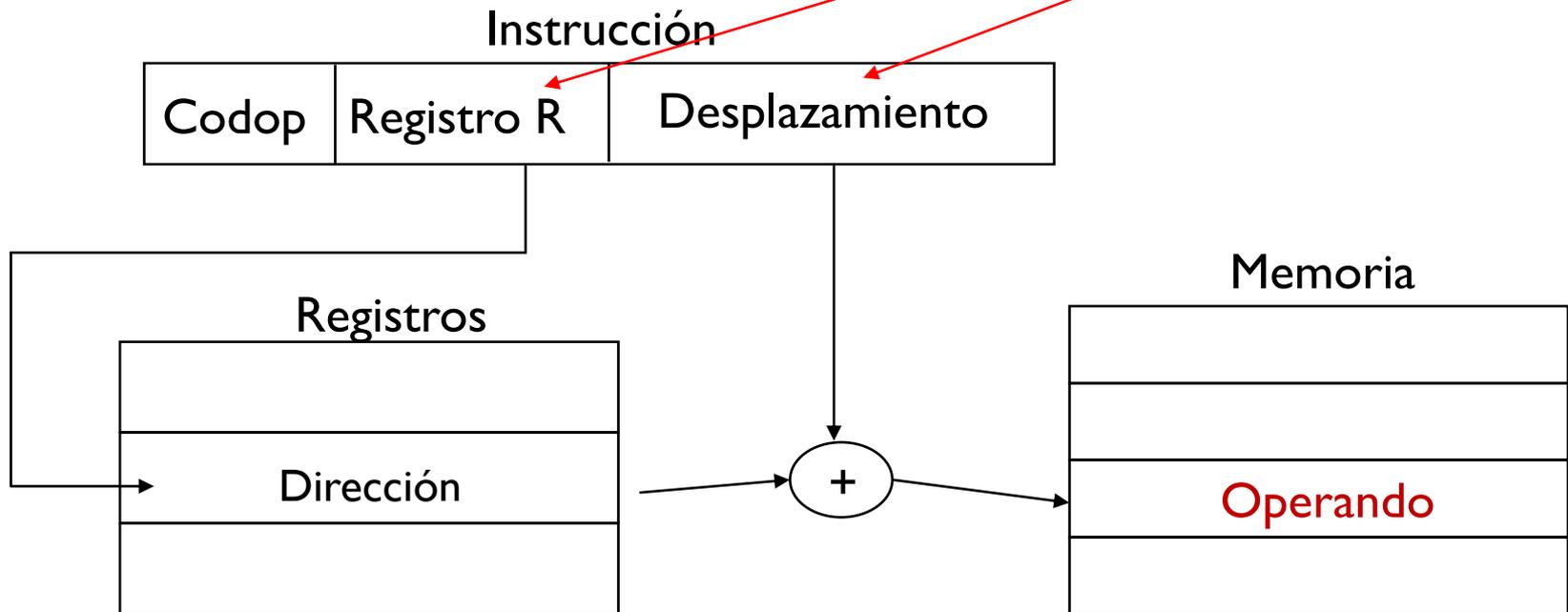
# Modos de direccionamiento

- ▶ El **modo de direccionamiento** es un procedimiento que permite determinar la **ubicación** de un operando, un resultado o una instrucción

- ▶ Implícito
- ▶ Inmediato
- ▶ Directo 
  - a registro
  - a memoria
- ▶ Indirecto 
  - a registro
  - a memoria
- ▶ **Relativo** 
  - **a registro índice**
  - **a registro base**
  - **a PC**
  - **a Pila**

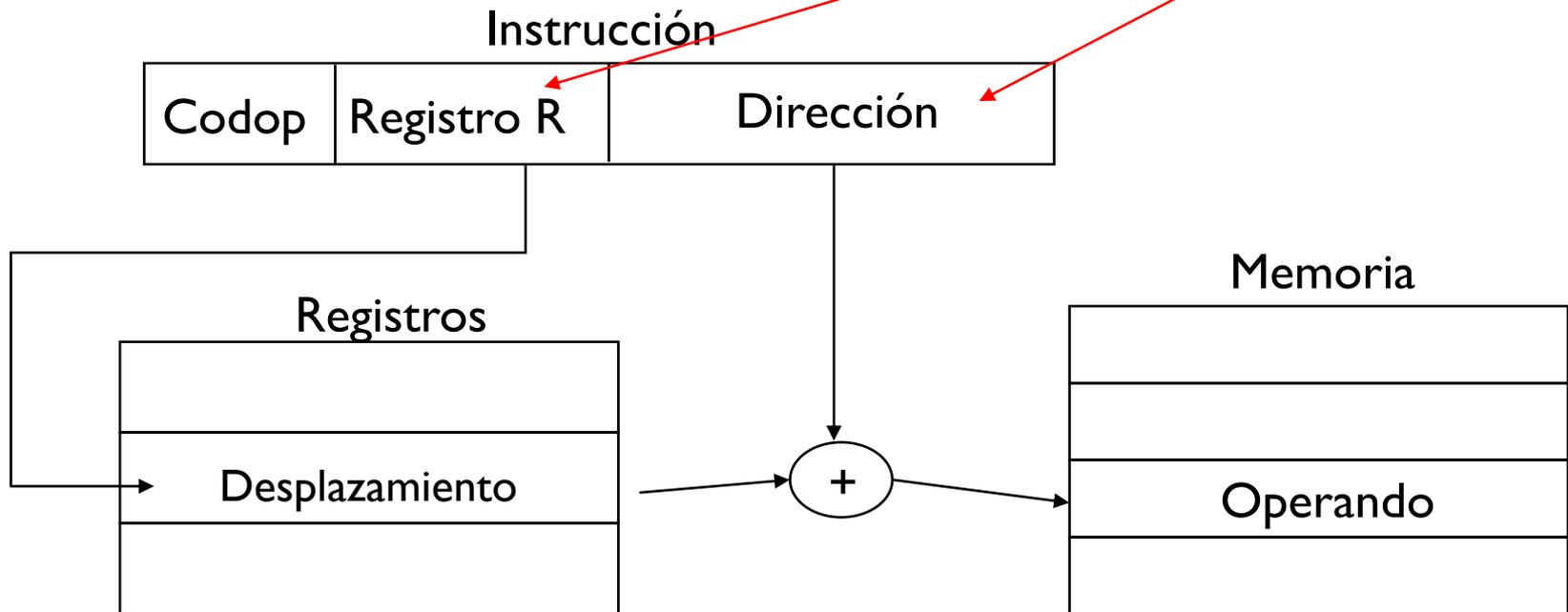
# Direccionamiento relativo a registro base

- Ejemplo: `lw a0 12(tl)`
  - Carga en `a0` el contenido de la posición de memoria dada por  $tl + 12$
  - Utiliza dos campos de la instrucción, **tl tiene la dirección base**



# Direccionamiento relativo a registro índice

- Ejemplo: `lw a0 dir(tl)`
  - Carga en `a0` el contenido de la posición de memoria dada por  $tl + dir$
  - Utiliza dos campos: **tl** representa el desplazamiento (índice) respecto a la dirección **dir**



# Utilidad: acceso a vectores

```
int v[5] ;
```

```
main ( )
```

```
{
```

```
    v[3] = 5 ;
```

```
    v[4] = 8 ;
```

```
    v[1] = 3 ;
```

```
}
```

```
.data
```

```
    v: .zero 20    # 5int*4bytes/int
```

```
.text
```

```
main:
```

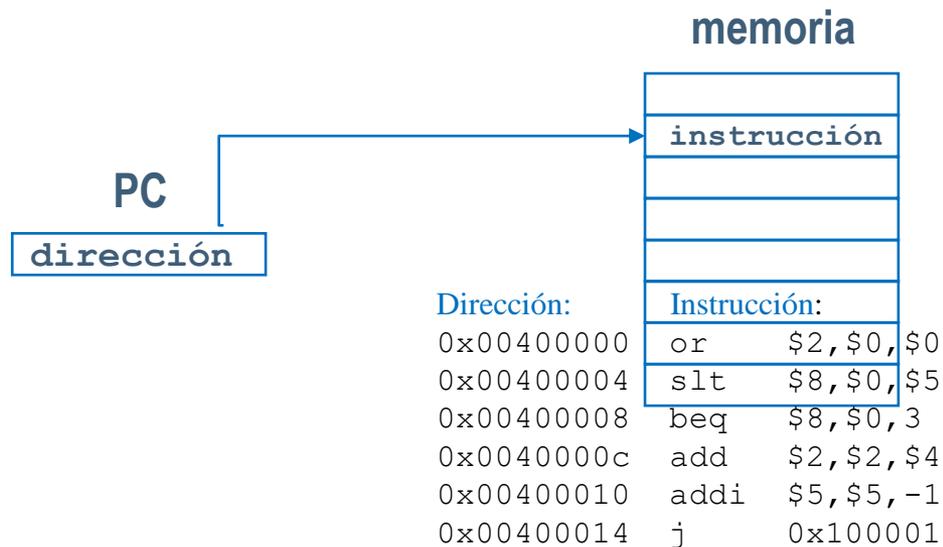
```
    la    t0, v
    li    t1, 5
    sw    t1, 12(t0)
```

```
    li    t0, 16
    li    t1, 8
    sw    t1, v(t0)
```

```
    la    t0, v
    addi  t0, t0, 4
    li    t1, 3
    sw    t1, (t0)
```

# El contador de programa PC...

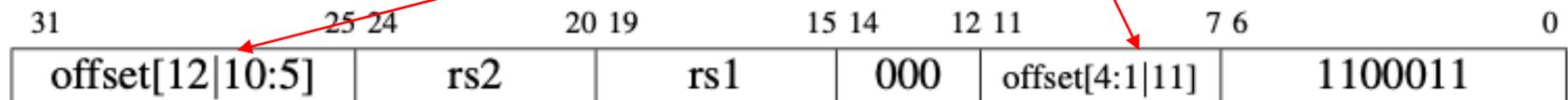
- ▶ Es un registro de 32 bits (4 bytes) en un computador de 32-bits
- ▶ Almacena la dirección de la siguiente instrucción a ejecutar
  - ▶ Apunta a una palabra (4 bytes) con la instrucción a ejecutar
  - ▶ PC en un computador de 32-bits se actualiza por defecto como  $PC = PC + 4$





# Direccionamiento relativo a PC en el RISC-V

- ▶ La instrucción `beq t0, x1, offset` se codifica en la instrucción:



- ▶ Etiqueta tiene que codificarse en el campo “offset” como desplazamiento relativo a la instrucción `beq` en el momento de ejecutarse (PC apunta al primer byte de la siguiente instrucción)
  - ▶ El valor de `offset` puede ser positivo o negativo
- ▶ ¿Cómo se actualiza el PC si `t0 == x1`?
  - ▶ Si se cumple la condición:
    - ▶  $PC = PC + offset$
  - ▶ Si no se cumple:
    - ▶  $PC = PC + 4$

# Direccionamiento relativo a PC en el RISC-V

- ▶ ¿Cuánto vale **fin** cuando se genera código máquina?

```
bucle:  beq    t0, x1, fin
        add    t8, t4, t4
        addi   t0, x0, -1
        j     bucle
fin:    mv     t1 x0
        . . .
```

- ▶ El valor de **fin** es:

- ▶ `fin == 12`
- ▶ Cuando se ejecuta una instrucción, el PC apunta a la siguiente
- ▶ Hay que saltar 3 instrucciones (“addi”, “j” y “mv”)
- ▶ Cada instrucción a saltar son 4 bytes

# Utilidad: desplazamientos en bucles

```
        li    t0 8
        li    t1 4
        li    t2 1
        li    t4 0
while:  bge   t4 t1 fin
        mul   t2 t2 t0
        addi  t4 t4 1
        j     while
fin:    mv    t2 t4
```

- ▶ **fin** representa la dirección donde se encuentra la instrucción `mv`
- ▶ **while** representa la dirección donde se encuentra la instrucción `bge`

# Utilidad: desplazamientos en bucles

```
li    t0 8
li    t1 4
li    t2 1
li    t4 0
while: bge  t4 t1 fin
      mul  t2 t2 t0
      addi t4 t4 1
      j    while
fin:   mv   t2 t4
```

Dirección	Contenido
0x0000100	li    t0 8
0x0000104	li    t1 4
0x0000108	li    t2 1
0x000010C	li    t4 0
0x0000110	bge   t4 t1 fin
0x0000114	mul   t2 t2 t0
0x0000118	addi  t4 t4 1
0x000011C	j     while
0x0000120	mv    t2 t4

# Utilidad: desplazamientos en bucles

```
li    t0 8
li    t1 4
li    t2 1
li    t4 0
while: bge  t4 t1 fin
      mul  t2 t2 t0
      addi t4 t4 1
      j    while
fin:   mv   t2 t4
```

En `bge t4 t1 fin`

`fin` representa un desplazamiento  
respecto al PC actual => **12**

$PC = PC + 12$

En `j while`

`while` representa un desplazamiento  
respecto al PC actual => **-16**

$PC = PC - 16$

Dirección	Contenido
0x0000100	li t0 8
0x0000104	li t1 4
0x0000108	li t2 1
0x000010C	li t4 0
0x0000110	bge t4 t1 fin
0x0000114	mul t2 t2 t0
0x0000118	addi t4 t4 1
0x000011C	j while
0x0000120	mv t2 t4

# Utilidad: desplazamientos en bucles

```
li    t0  8
li    t1  4
li    t2  1
li    t4  0
while: bge  t4  t1  fin
      mul  t2  t2  t0
      addi t4  t4  1
      j    while
fin:   mv   t2  t4
```

En `bge t4 t1 fin`

`fin` representa un desplazamiento  
respecto al PC actual => **12**

$PC = PC + 12$

En `j while`

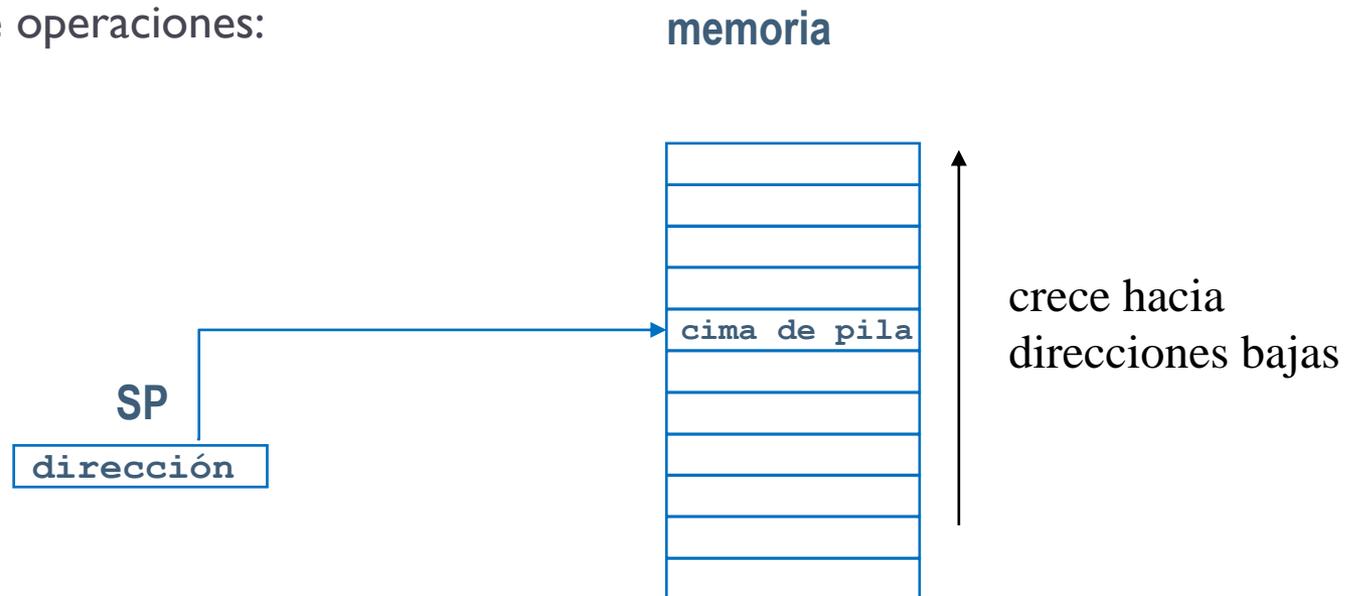
`while` representa un desplazamiento  
respecto al PC actual => **-16**

$PC = PC - 16$

Dirección	Contenido
0x0000100	li    t0  8
0x0000104	li    t1  4
0x0000108	li    t2  1
0x000010C	li    t4  0
0x0000110	bge   t4  t1 <b>12</b>
0x0000114	mul   t2  t2  t0
0x0000118	addi  t4  t4  1
0x000011C	j <b>-16</b>
0x0000120	mv    t2  t4

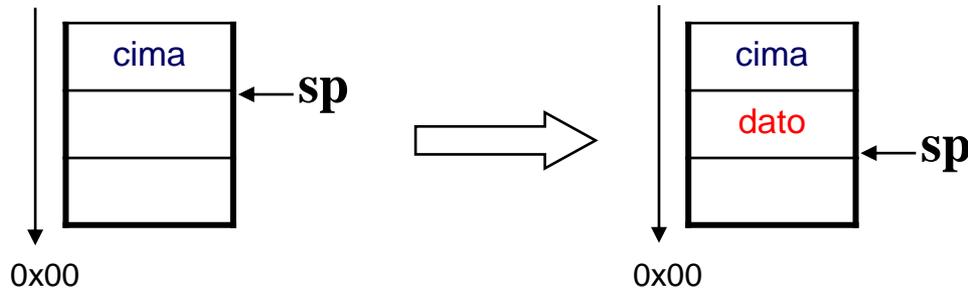
# Direccionamiento relativo a pila

- ▶ El puntero de pila SP (*Stack Pointer*):
  - ▶ Es un registro de 32 bits (4 bytes) en el RISC-V<sub>32</sub>
  - ▶ Almacena la dirección de la cima de pila
    - ▶ Apunta a una palabra (4 bytes)
- ▶ Dos tipos de operaciones:
  - ▶ push
  - ▶ pop

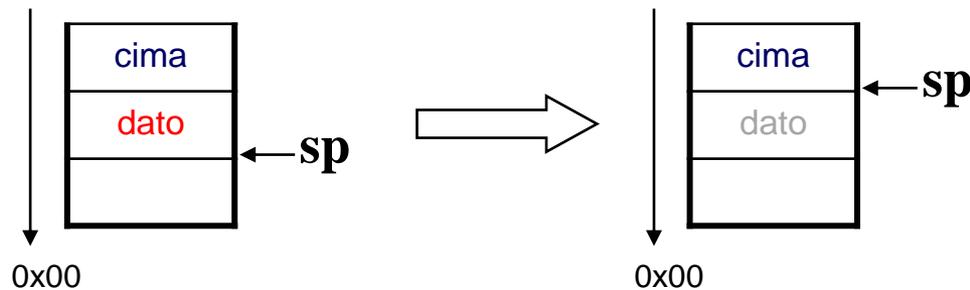


# Pila crece hacia direcciones bajas

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



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

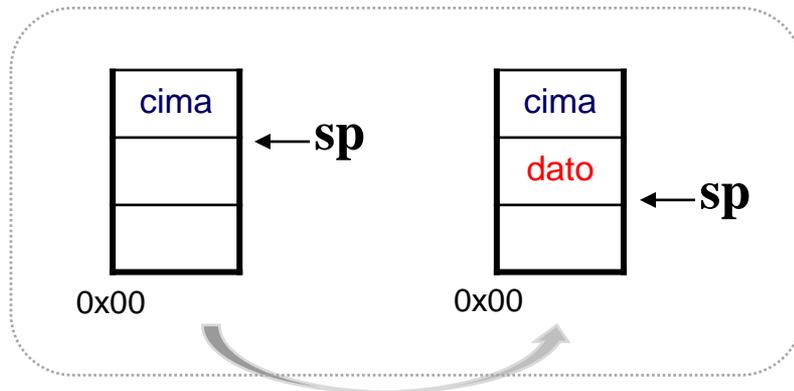


# Direccionamiento de pila en el RISC-V

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

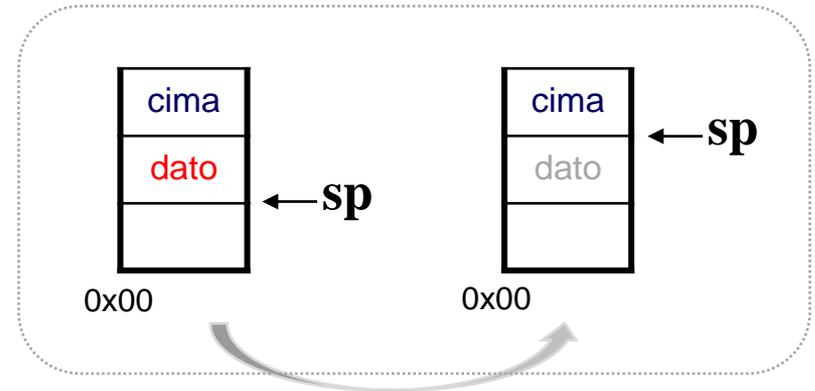
## PUSH $t0$

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



## POP $t0$

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



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

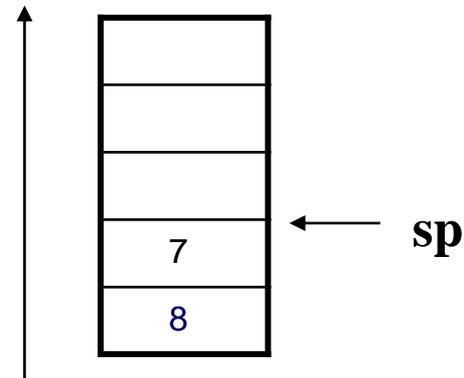
...

```
li t2, 9
```

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

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

...

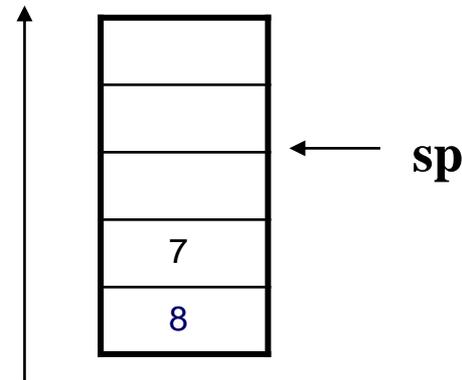


## ► Estado inicial:

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

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

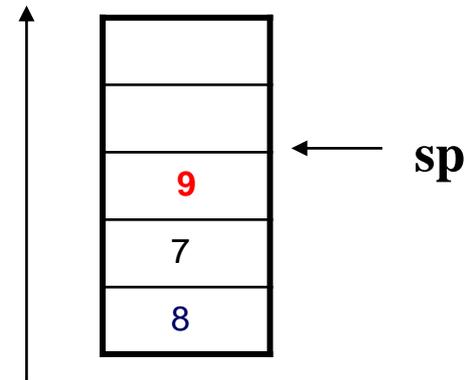
```
...  
li    t2, 9  
addi sp, sp, -4  
sw    t2 0(sp)  
...
```



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

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

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



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

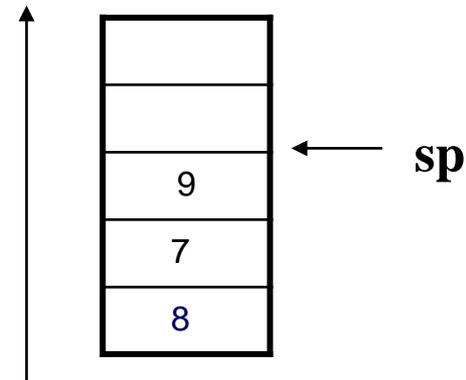
# Operación POP en el RISC-V<sub>32</sub>

...

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

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

...



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

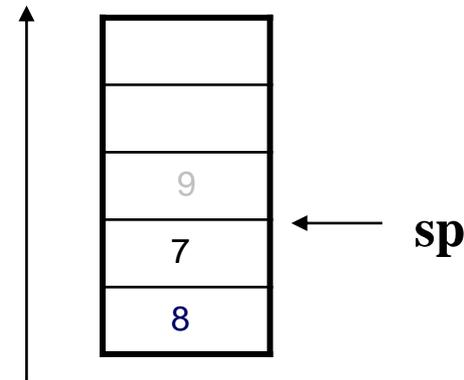
# Operación POP en el RISC-V<sub>32</sub>

...

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

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

...

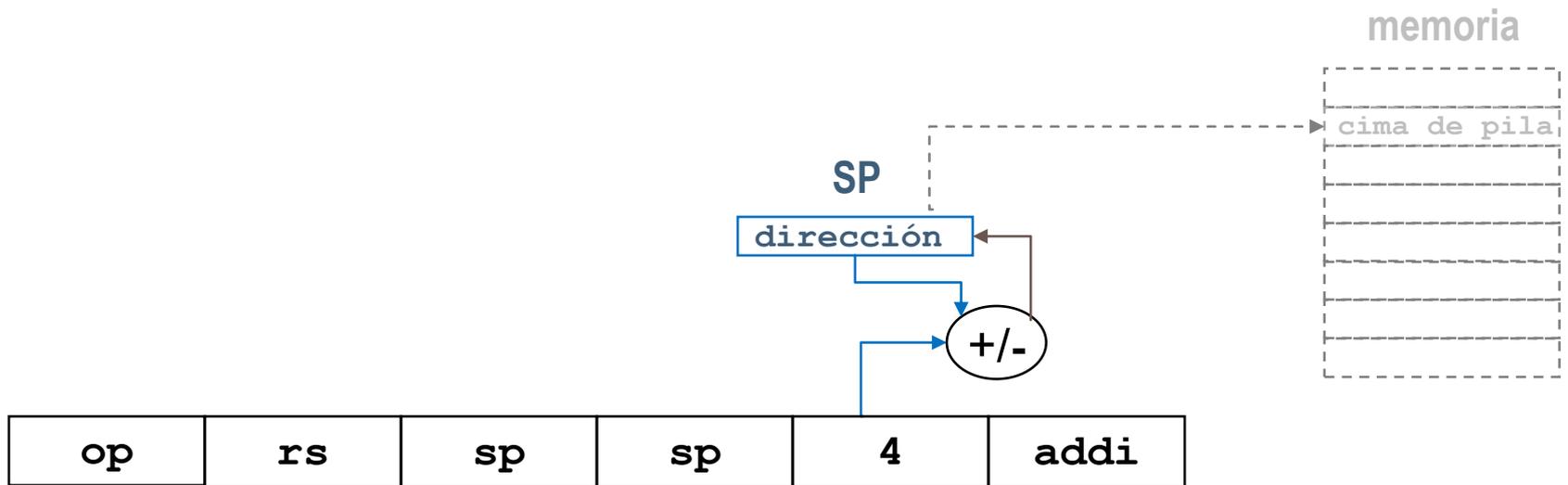


- ▶ Se actualiza el registro `sp` para apuntar a la nueva cima de la pila.
  - ▶ `addi sp, sp, 4`
- ▶ El dato desapilado (9) sigue estando en memoria pero será sobrescrito en futuras operaciones PUSH (o similar de acceso a memoria)

# Direccionamiento de pila en el RISC-V

## ▶ Ejemplo: **push a0**

- ▶ `addi sp sp -4` #  $SP = SP - 4$
- ▶ `sw a0 0(sp)` #  $\text{memoria}[SP] = a0$



# Ejercicio

► Indique el tipo de direccionamiento usado en las siguientes instrucciones RISC-V:

1. `li t1 4`
2. `lw t0 4(a0)`
3. `bne x0 a0 etiqueta`

# Ejercicio (solución)

## 1. `li t1 4`

- ▶ `t1` -> directo a registro
- ▶ `4` -> inmediato

## 1. `lw t0 4(a0)`

- ▶ `t0` -> directo a registro
- ▶ `4(a0)` -> relativo a registro base

## 1. `bne x0 a0 etiqueta`

- ▶ `a0` -> directo a registro
- ▶ `etiqueta` -> relativo a contador de programa

# Ejemplos de tipos de direccionamiento

- ▶ **la t0 label**                      **inmediato**
  - ▶ El segundo operando de la instrucción es una dirección
  - ▶ PERO no se accede a esta dirección, la propia dirección es el operando
  
- ▶ **lw t0 label**                      **directo a memoria (no  $\exists$  en RV32)**
  - ▶ El segundo operando de la instrucción es una dirección
  - ▶ Hay que acceder a esta dirección para tener el valor con el que trabajar
  
- ▶ **bne t0 t1 label**                      **relativo a registro PC**
  - ▶ El tercer operando de la instrucción es desplazamiento respecto al PC
  - ▶ label se codifica como un número en complemento a dos que representa el desplazamiento (como palabras) relativo al registro PC

# Ejemplos de instrucciones

- ▶ `la t0, 0x0F000002`
  - ▶ Direccionamientos directo a registro + inmediato.  
Se carga en `t0` el valor `0x0F000002`
- ▶ `lbu t0, etiqueta(x0)`
  - ▶ Direccionamientos directo a reg. + relativo a registro base.  
Se carga en `t0` el byte en la dirección de memoria `etiqueta`
- ▶ `lb t0, 0(t1)`
  - ▶ Direccionamientos directo a reg. + relativo a registro base.  
Se carga en `t0` el byte en la posición de memoria almacenada en `t1+0`

# Contenidos

- ▶ Fundamentos básicos de la programación en ensamblador
- ▶ Ensamblador del RISC-V<sub>32</sub>, modelo de memoria y representación de datos
- ▶ Formato de las instrucciones, modos de direccionamiento y **juego de instrucciones**
- ▶ Llamadas a procedimientos y uso de la pila

# Juego de instrucciones

- ▶ Queda **definido** por:
  - ▶ Conjunto de instrucciones
  - ▶ Formato de las instrucciones
  - ▶ Registros
  - ▶ Modos de direccionamiento
  - ▶ Tipos de datos y formatos

# Juego de instrucciones

- ▶ Distintas formas para la **clasificación** de un juego de instrucciones:
  - ▶ Complejidad del juego de instrucciones
    - ▶ CISC vs RISC
  - ▶ Modo de ejecución
    - ▶ Pila
    - ▶ Registro
    - ▶ Registro-Memoria, Memoria-Registro, ...

# CISC vs RISC

## ▶ *Complex Instruction Set Computer*

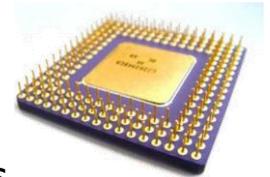
- ▶ Muchas instrucciones
- ▶ Instrucciones complejas
  - ▶ Más de una palabra
  - ▶ Unidad de control más compleja
  - ▶ Mayor tiempo de ejecución
- ▶ Diseño irregular

## ▶ *Reduced Instruction Set Computer*

- ▶ Instrucciones simples y ortogonales
  - ▶ Ocupan una palabra
  - ▶ Instrucciones sobre registros
  - ▶ Uso de los mismos modos de direccionamiento para todas las instrucciones (alto grado de ortogonalidad)
- ▶ Diseño más compacto:
  - ▶ Unidad de control más sencilla y rápida
  - ▶ Espacio sobrante para más registros y memoria caché



- ▶ Alrededor del 20% de las instrucciones ocupa el 80% del tiempo total de ejecución de un programa
- ▶ El 80% de las instrucciones no se utilizan casi nunca
- ▶ 80% del silicio infrutilizado, complejo y costoso



# Modos de ejecución

- ▶ Los modos de ejecución indican el número de operandos y el tipo de operandos que pueden especificarse en una instrucción.
  - ▶ 0 direcciones → Pila.
    - PUSH 5; PUSH 7; ADD
  - ▶ 1 dirección → Registro acumulador.
    - ADD RI →  $AC \leftarrow AC + RI$
  - ▶ 2 direcciones → Registros, Registro-memoria, Memoria-memoria.
    - ADD .R0, .RI ( $R0 \leftarrow R0 + RI$ )
  - ▶ 3 direcciones → Registros, Registro-memoria, Memoria-memoria.
    - ADD .R0, .RI, .R2

Grupo ARCOS

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

# Tema 3: Fundamentos de la programación en ensamblador (III) **Estructura de Computadores**

Grado en Ingeniería Informática  
Grado en Matemática aplicada y Computación  
Doble Grado en Ingeniería Informática y Administración de Empresas

