

Tema 1

ESTRUCTURA BÁSICA DE UN COMPUTADOR.
PROGRAMACIÓN EN ENSAMBLADOR.

Tecnología de Computadores



TEMA 1

1. Introducción

1. ¿Qué es un computador?
2. Programación de un procesador
3. Ejecución de un programa

2. Instrucciones de un procesador

1. Conjunto de instrucciones
2. Formato de instrucción
3. Modos de direccionamiento

3. Programación en ensamblador

1. Lenguaje ensamblador RISC-V
2. Traducción de estructuras de alto nivel a ensamblador



¿QUÉ ES UN COMPUTADOR? INTRODUCCIÓN

1. ¿QUÉ ES UN COMPUTADOR?

Máquina (sistema) en la que se distinguen dos elementos:

- HW: Circuitos y cableado.
- SW: Información que modifica la función del HW sin realizar una reconfiguración de la circuitería para la resolución del problema.

↓ ¿Cómo?

Controlando el HW ⇒ Indicando qué debe hacer

1. ¿QUÉ ES UN COMPUTADOR?

Máquina (sistema) en la que se distinguen dos elementos:

- HW: Circuitos y cableado.
- SW: Información que modifica la función del HW sin realizar una reconfiguración de la circuitería para la resolución del problema.

Controlando el HW \Rightarrow Indicando qué debe hacer
↓ ¿Cómo?

Mediante una colección de elementos que
instruyen al HW sobre la acción a realizar en
cada momento

↓ ¿Cómo? **INSTRUCCIONES**

1. ¿QUÉ ES UN COMPUTADOR?

Máquina (sistema) en la que se distinguen dos elementos:

- HW: Circuitos y cableado.
- SW: Información que modifica la función del HW sin realizar una reconfiguración de la circuitería para la resolución del problema.

Mediante una colección de **elementos** que instruyen al HW sobre la acción a realizar en cada momento



PROGRAMA



1. ¿QUÉ ES UN COMPUTADOR?

TIPOS DE PROPOSITO

- Específico: Diseñado para una tarea determinada.
- General: Máquina destinada a ejecutar cualquier tipo de algoritmo

1. ¿QUÉ ES UN COMPUTADOR?

TIPOS DE PROPOSITO

- Específico: Diseñado para una tarea determinada.
- General: Máquina destinada a ejecutar cualquier tipo de algoritmo

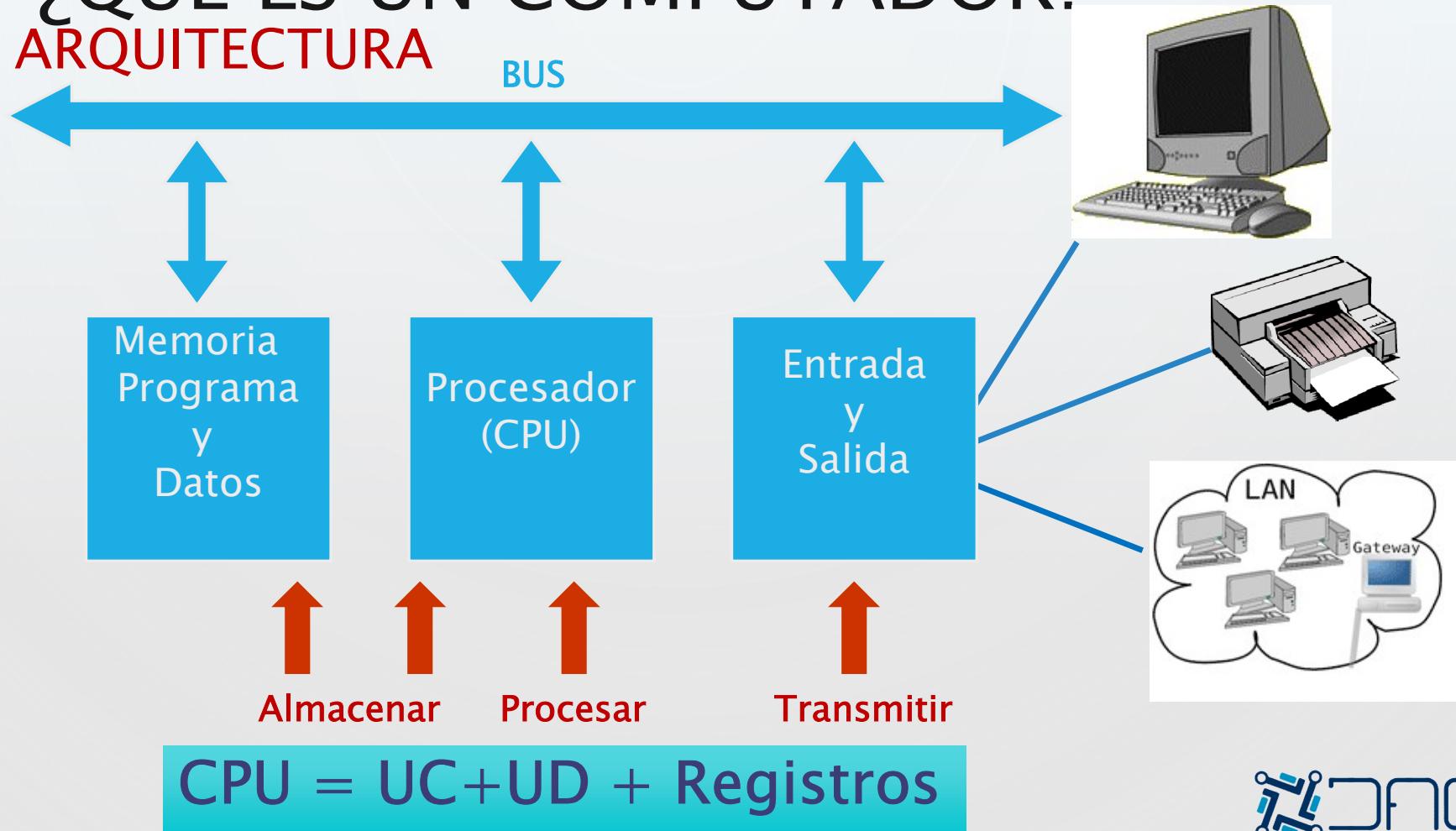


Programa almacenado: Establecido por John Von Neumann (1945)⇒

- El almacenamiento de los programas en memoria.
- Las instrucciones se codifican de forma numérica (secuencias binarias)

1. ¿QUÉ ES UN COMPUTADOR?

ARQUITECTURA



1. ¿QUÉ ES UN COMPUTADOR?

Por tanto:

Un computador es un sistema electrónico digital usado para la manipulación de información digital (datos binarios), de acuerdo a una lista de Instrucciones conocida como programa”

Tipo de manipulación:

- Almacenar
- Transmitir
- Procesar

INTRODUCCIÓN

PROGRAMACIÓN DE UN PROCESADOR

2 PROGRAMACIÓN DE UN COMPUTADOR

¿Cómo se define qué tarea debe realizar un computador? → **PROGRAMA**

- **Programa**: Algoritmo escrito mediante un lenguaje de programación.
- **Algoritmo**: Secuencia de pasos a seguir para resolver un problema.
- **Lenguaje de programación**: Conjunto de reglas y símbolos que permiten escribir un algoritmo formalmente para que un computador le entiende.

2 TIPOS DE LENGUAJES

Lenguaje de alto nivel

Lenguaje de Programación con instrucciones de alto contenido semántico comprensibles por los humanos. (ejm: C, C++, Java, Pascal).

```
#define vueltas 4
int suma = 0;
int V[4] = {0,1,2,3};
main () {
    register int suma_parcial = 0;
    register int i = 0;
    while (i < vueltas) {
        suma_parcial += V[i];
        i++;
    }
    suma = suma_parcial;
}
```

2 TIPOS DE LENGUAJES

Lenguaje ensamblador

Lenguaje de Programación con instrucciones de bajo contenido semántico comprensibles por los humanos. (ejm: Intel 386, Alpha, MIPS, ARM)

```
.text
main:
    MOVI R3,0          ; R3 es suma_parcial
    MOVI R1,0          ; R1 es i
    MOVI R4, LO(V)
    MOVHI R4, HI(V)   ; R4 carga 1er elem. vector V
    MOVI R0,0          ; para el salto incondicional
    MOVI R2,Vueltas
while:
    CMPLT R5,R1,r2    ; ¿i < Vueltas?
    BZ R5, fiwhile    ; si (R5==0) fin bucle
    ...
    ADDI R1,R1,1       ; i++
    ...
    BZ R0, while       ; salto incondicional bucle
Fiwhile:
```

2 TIPOS DE LENGUAJES

Lenguaje máquina

Lenguaje de Programación con instrucciones comprensible por el computador.

Ciclo de instrucción

- **Búsqueda de la instrucción (fetching):** el procesador lee de memoria la instrucción a ejecutar. Debe conocer la posición de memoria donde se encuentra la instrucción: PC (Program Counter, contador del programa).
- **Decodificación de la instrucción:** Debe determinar el tipo de instrucción y los operandos que utiliza.
- **Ejecución de la instrucción:** Ejecuta la acción expresada por la instrucción.



```
0010100101101001  
1010001001001011  
1111010100111010  
0010010101000010  
1110010100101001  
0101000001000011  
1110010010000010  
0001000010101110  
1001000101010000  
1110100100100100  
0010101001011110  
1100101000000010
```

2 PROCESO DE TRADUCCIÓN

- El procesador SOLO entiende lenguaje máquina (instrucciones máquina).
- Existe un proceso de “traducción” desde el lenguaje de alto nivel hasta el lenguaje máquina.

001000101001
00111110000
.....

Introducción

```
Main () {  
    int a, b, c  
    c=a+b  
    ....
```

Programa en Lenguaje de Alto nivel (ejem. C++, C, Fortran, etc)

Compilador

Programa en Lenguaje Ensamblador

```
ADD R1 , R0 , a  
ADD R1, R3, R2  
....
```

Ensamblador

Programa en Lenguaje maquina (unos y ceros)

INTRODUCCIÓN

EJECUCIÓN DE UN PROGRAMA

3. EJECUCIÓN DE UN PROGRAMA

- El procesador lee y ejecuta secuencialmente y en orden las instrucciones que forman un programa **almacenadas en memoria**.
- Ejecución de una instrucción (ciclo de instrucción):
 - **Búsqueda de la instrucción**: el procesador lee de memoria la instrucción a ejecutar. El PC (Program Counter, contador del programa) da la posición de memoria donde se encuentra dicha instrucción

3. EJECUCIÓN DE UN PROGRAMA

- El procesador lee y ejecuta secuencialmente y en orden las instrucciones que forman un programa **almacenadas en memoria**.
- Ejecución de una instrucción (ciclo de instrucción):
 - **Búsqueda de la instrucción**: el procesador lee de memoria la instrucción a ejecutar. El PC (Program Counter, contador del programa) da la posición de memoria donde se encuentra dicha instrucción
 - **Decodificación de la instrucción**: Debe determinar el tipo de instrucción y los operandos que utiliza.

3. EJECUCIÓN DE UN PROGRAMA

- El procesador lee y ejecuta secuencialmente y en orden las instrucciones que forman un programa **almacenadas en memoria**.
- Ejecución de una instrucción (ciclo de instrucción):
 - **Búsqueda de la instrucción**: el procesador lee de memoria la instrucción a ejecutar. El PC (Program Counter, contador del programa) da la posición de memoria donde se encuentra dicha instrucción
 - **Decodificación de la instrucción**: Debe determinar el tipo de instrucción y los operandos que utiliza.
 - **Ejecución de la instrucción**: Ejecuta la acción expresada por la instrucción.

TEMA 1

1. Introducción

1. ¿Qué es un computador?
2. Programación de un procesador
3. Ejecución de un programa

2. Instrucciones de un procesador

1. Conjunto de instrucciones
2. Formato de instrucción
3. Modos de direccionamiento

3. Programación en ensamblador

1. Lenguaje ensamblador RISC-V
2. Traducción de estructuras de alto nivel a ensamblador

INSTRUCCIONES DE UN PROCESADOR

CONJUNTO DE INSTRUCCIONES

1. CONJUNTO DE INSTRUCCIONES

- Cada procesador está diseñado para “entender” un conjunto de instrucciones
- Tipos de instrucciones
 - **Transferencia**: copian información de un lugar a otro (memoria, registros)
 - **Transformación**: transforman la información
 - Aritméticas, Lógicas, Desplazamiento, Comparación, ...
 - **Control**: controlan el flujo de instrucciones del programa

Instrucciones de un procesador

CONJUNTO DE INSTRUCCIONES, EJEMPLO MIPS

MIPS assembly language				
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1,\$s2,\$s3	\$s1 = \$s2 + \$s3	Three register operands
	subtract	sub \$s1,\$s2,\$s3	\$s1 = \$s2 - \$s3	Three register operands
	add immediate	addi \$s1,\$s2,20	\$s1 = \$s2 + 20	Used to add constants
Data transfer	load word	lw \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Word from memory to register
	store word	sw \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Word from register to memory
	load half	lh \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Halfword memory to register
	load half unsigned	lhu \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Halfword memory to register
	store half	sh \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Halfword register to memory
	load byte	lb \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Byte from memory to register
	store byte	sb \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	Byte from register to memory
	load linked word	ll \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc \$s1,20(\$s2)	Memory[\$s2+20] = \$s1; \$s1=0 or 1	Store word as 2nd half of atomic swap
Logical	load upper immed.	lui \$s1,20	\$s1 = 20 * 2 ¹⁶	Loads constant in upper 16 bits
	and	and \$s1,\$s2,\$s3	\$s1 = \$s2 & \$s3	Three reg. operands; bit-by-bit AND
	or	or \$s1,\$s2,\$s3	\$s1 = \$s2 \$s3	Three reg. operands; bit-by-bit OR
	nor	nor \$s1,\$s2,\$s3	\$s1 = ~(\$s2 \$s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi \$s1,\$s2,20	\$s1 = \$s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori \$s1,\$s2,20	\$s1 = \$s2 20	Bit-by-bit OR reg with constant
	shift left logical	sll \$s1,\$s2,10	\$s1 = \$s2 << 10	Shift left by constant
Conditional branch	shift right logical	srl \$s1,\$s2,10	\$s1 = \$s2 >> 10	Shift right by constant
	branch on equal	beq \$s1,\$s2,25	if (\$s1 == \$s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1,\$s2,25	if (\$s1 != \$s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt \$s1,\$s2,\$s3	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu \$s1,\$s2,\$s3	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than unsigned
	set less than immediate	slti \$s1,\$s2,20	if (\$s2 < 20) \$s1 = 1; else \$s1 = 0	Compare less than constant
Unconditional jump	set less than immediate unsigned	sltiu \$s1,\$s2,20	if (\$s2 < 20) \$s1 = 1; else \$s1 = 0	Compare less than constant unsigned
	jump	j 2500	go to 10000	Jump to target address
	jump register	jr \$ra	go to \$ra	For switch, procedure return
	jump and link	jal 2500	\$ra = PC + 4; go to 10000	For procedure call

CONJUNTO DE INSTRUCCIONES, EJEMPLO ARM

ARM assembly language				
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	ADD r1,r2,r3	$r1 = r2 + r3$	3 register operands
	subtract	SUB r1,r2,r3	$r1 = r2 - r3$	3 register operands
Data transfer	load register	LDR r1, [r2,#20]	$r1 = \text{Memory}[r2 + 20]$	Word from memory to register
	store register	STR r1, [r2,#20]	$\text{Memory}[r2 + 20] = r1$	Word from memory to register
	load register halfword	LDRH r1, [r2,#20]	$r1 = \text{Memory}[r2 + 20]$	Halfword memory to register
	load register halfword signed	LDRHS r1, [r2,#20]	$r1 = \text{Memory}[r2 + 20]$	Halfword memory to register
	store register halfword	STRH r1, [r2,#20]	$\text{Memory}[r2 + 20] = r1$	Halfword register to memory
	load register byte	LDRB r1, [r2,#20]	$r1 = \text{Memory}[r2 + 20]$	Byte from memory to register
	load register byte signed	LDRBS r1, [r2,#20]	$r1 = \text{Memory}[r2 + 20]$	Byte from memory to register
	store register byte	STRB r1, [r2,#20]	$\text{Memory}[r2 + 20] = r1$	Byte from register to memory
	swap	SWP r1, [r2,#20]	$r1 = \text{Memory}[r2 + 20], \text{Memory}[r2 + 20] = r1$	Atomic swap register and memory
Logical	mov	MOV r1, r2	$r1 = r2$	Copy value into register
	and	AND r1, r2, r3	$r1 = r2 \& r3$	Three reg. operands; bit-by-bit AND
	or	ORR r1, r2, r3	$r1 = r2 r3$	Three reg. operands; bit-by-bit OR
	not	MVN r1, r2	$r1 = \sim r2$	Two reg. operands; bit-by-bit NOT
	logical shift left (optional operation)	LSL r1, r2, #10	$r1 = r2 << 10$	Shift left by constant
	logical shift right (optional operation)	LSR r1, r2, #10	$r1 = r2 >> 10$	Shift right by constant
Conditional Branch	compare	CMP r1, r2	cond. flag = $r1 - r2$	Compare for conditional branch
	branch on EQ, NE, LT, LE, GT, GE, LO, LS, HI, HS, VS, VC, MI, PL	BEQ 25	if ($r1 == r2$) go to PC + 8 + 100	Conditional Test; PC-relative
Unconditional Branch	branch (always)	B 2500	go to PC + 8 + 10000	Branch
	branch and link	BL 2500	$r14 = \text{PC} + 4; \text{go to PC} + 8 + 10000$	For procedure call

INSTRUCCIONES DE UN PROCESADOR

FORMATOS DE INSTRUCCIONES

2. FORMATO DE INSTRUCCIÓN

- Las instrucciones se almacenan en memoria →
Codificación en binario (n-bits)

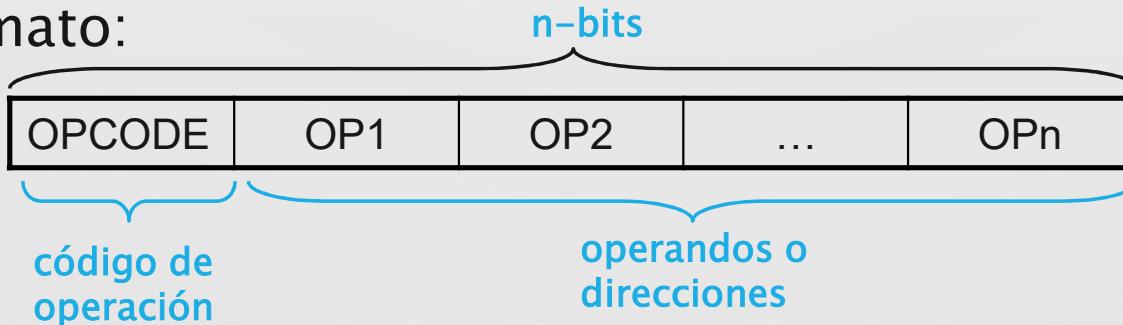
- **ADD A, B, C** → 0100100100100101

2. FORMATO DE INSTRUCCIÓN

- Las instrucciones se almacenan en memoria → **Codificación en binario (n-bits)**
 - ADD A, B, C → 0100100100100101
- Elementos de una instrucción:
 - **Código de operación (OPCODE):** especifica el tipo de instrucción. (ADD)
 - **Operandos (direcciones):** argumentos necesarios para la ejecución de la instrucción (entradas/salidas). (A, B, C)

2. FORMATO DE INSTRUCCIÓN

- Las instrucciones se almacenan en memoria → **Codificación en binario (n-bits)**
 - ADD A, B, C → 0100100100100101
- Elementos de una instrucción:
 - Código de operación (OPCODE): especifica el tipo de instrucción. (ADD)
 - Operandos (direcciones): argumentos necesarios para la ejecución de la instrucción (entradas/salidas). (A, B, C)
- Formato:



FORMATO DE INSTRUCCIÓN

- **OPCODE**

- Codificación en binario del tipo de instrucción (código binario)
- Formato fijo o variable

FORMATO DE INSTRUCCIÓN

- **OPCODE**
 - Codificación en binario del tipo de instrucción (código binario)
 - Formato fijo o variable
- **Operandos o direcciones**
 - Especifican los operandos fuente y destino de la instrucción (codificados en binario).
 - Valor, registro, dirección de memoria

FORMATO DE INSTRUCCIÓN

- **OPCODE**

- Codificación en binario del tipo de instrucción (código binario)
- Formato fijo o variable

- **Operandos o direcciones**

- Especifican los operandos fuente y destino de la instrucción (codificados en binario).
 - Valor, registro, dirección de memoria
- Hay operandos **explícitos e implícitos**
 - Máquina de **M-direcciones**; el formato de instrucción acepta como máximo M operandos explícitos.

FORMATO DE INSTRUCCIÓN

- **OPCODE**

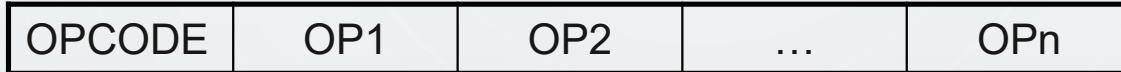
- Codificación en binario del tipo de instrucción (código binario)
- Formato fijo o variable

- **Operandos o direcciones**

- Especifican los operandos fuente y destino de la instrucción (codificados en binario).
 - Valor, registro, dirección de memoria
- Hay operandos **explícitos** e **implícitos**
 - Máquina de **M-direcciones**: el formato de instrucción acepta como máximo M operandos explícitos.
- **Modo de direccionamiento**: especifica como se debe interpretar el campo operando.
 - Cada operando posee su propio modo de direccionamiento

FORMATO DE INSTRUCCIÓN

- Formato de instrucción de longitud fija
 - Todas las instrucciones tienen mismo número de bits



- Formato de instrucción de longitud variable
 - La longitud de las instrucciones varía
 - Varias palabras de memoria



INSTRUCCIONES DE UN PROCESADOR

MODOS DE DIRECCIONAMIENTO

3. MODOS DE DIRECCIONAMIENTO

¿Cómo se interpretan los m-bits del campo operando para obtener el operando?

ADD A, B, C → 0100100100100101 ($A \leftarrow B + C$)

- ¿Qué dato concreto hay que sumar a **C** para calcular el valor a almacenar en **A**?
- La respuesta la da el modo de direccionamiento del operando **B**
 - Significado de **0010**

3. MODOS DE DIRECCIONAMIENTO

- Inmediato
- Directo
 - Absoluto
 - A registro
 - A memoria
 - Relativo
 - Registro implícito (PC, base, índice, SP, ...)
 - Registro explícito
- Indirecto
 - Por registro
 - Por memoria

3. MODOS DE DIRECCIONAMIENTO (I)

Inmediato

- “El campo operando es el propio **VALOR** del operando”
- El operando está en la propia instrucción
- Codificación en binario del tipo de dato del operando
(natural, entero, real, ...)

3. MODOS DE DIRECCIONAMIENTO (I)

Inmediato

- “El campo operando es el propio **VALOR** del operando”
- El operando está en la propia instrucción
- Codificación en binario del tipo de dato del operando (natural, entero, real, ...)
- Ejemplo: B operando Natural con direccionamiento inmediato:
 - $A = ? + C$

ADDI A, B, C → 0011001000100101

3. MODOS DE DIRECCIONAMIENTO (I)

Inmediato

- “El campo operando es el propio **VALOR** del operando”
- El operando está en la propia instrucción
- Codificación en binario del tipo de dato del operando (natural, entero, real, ...)
- Ejemplo: B operando Natural con direccionamiento inmediato:

$$A = 2 + C$$

$$00010 = 2$$

ADDI A, B, C → 0011001000100101

Dato

3. MODOS DE DIRECCIONAMIENTO (II)

Directo

- El campo operando indica **DONDE** está el operando
- El operando puede estar en un registro o en una posición de memoria
- Puede indicar donde está el operando de dos modos distintos:
 - **ABSOLUTO**: Se proporciona la dirección completa del operando
 - **RELATIVO**: Se incluye información parcial de direccionamiento
⇒ calcular la dirección final donde está el operando

3. MODOS DE DIRECCIONAMIENTO (II)

Directo Absoluto

- Directo a **Registro**: el operando está en un registro (código binario del registro)
- Directo a **Memoria**: el operando está en una posición de memoria (dirección de memoria)

Ejemplo Directo a Registro

$$A = ? + C$$

ADD2 A, B, C → 0101000100101010

Memoria	
M(00000)	00010
M(00001)	00100
M(00010)	00000
...	...
M(11111)	00101

Registros	
R0 (0000)	00110
R1 (0001)	00001
R2 (0010)	00111
...	...
R15 (1111)	00000

3. MODOS DE DIRECCIONAMIENTO (III)

Directo Absoluto

- Directo a **Registro**: el operando está en un registro (código binario del registro)
- Directo a **Memoria**: el operando está en una posición de memoria (dirección de memoria)

Ejemplo Directo a Registro

$$A = 7 + C$$

00111 = 7

ADD2 A, B, C → 0101000100100101

Registro

Memoria	
M(00000)	00010
M(00001)	00100
M(00010)	00000
...	...
M(11111)	00101

Registros	
R0 (0000)	00110
R1 (0001)	00001
R2 (0010)	00111
...	...
R15 (1111)	00000

3. MODOS DE DIRECCIONAMIENTO (III)

Directo Absoluto

- Directo a **Registro**: el operando está en un registro (código binario del registro)
- Directo a **Memoria**: el operando está en una posición de memoria (dirección de memoria)

Ejemplo Directo a Memoria

$$A = ? + C$$

ADD3 A, B, C → 0111001000100101

Memoria	
M(00000)	00010
M(00001)	00100
M(00010)	00000
...	...
M(11111)	00101

Registros	
R0 (0000)	00110
R1 (0001)	00001
R2 (0010)	00111
...	...
R15 (1111)	00000

3. MODOS DE DIRECCIONAMIENTO (III)

Directo Absoluto

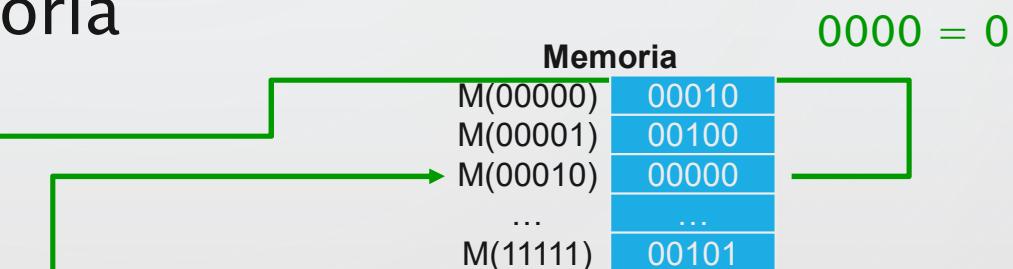
- Directo a **Registro**: el operando está en un registro (código binario del registro)
- Directo a **Memoria**: el operando está en una posición de memoria (dirección de memoria)

Ejemplo Directo a Memoria

$$A = 0 + C$$

ADD3 A, B, C → 0111001000100101

Dir. Memoria



Registros	
R0 (0000)	00110
R1 (0001)	00001
R2 (0010)	00111
...	...
R15 (1111)	00000

3. MODOS DE DIRECCIONAMIENTO

Directo Relativo

- El operando está en una posición de memoria
- **Dirección Efectiva (DE)**: Dirección de memoria donde se encuentra el operando
- La DE se calcula sumando un **desplazamiento** al contenido de un **registro**:
 - Con **registro explícito**: Hay que especificar en la instrucción el desplazamiento y el registro.
 - Con **registro implícito**: El registro no se especifica explícitamente.

3. MODOS DE DIRECCIONAMIENTO (IV)

Directo Relativo con registro implícito:

- El registro no se especifica explícitamente en el formato de la instrucción
- Sólo se especifica el desplazamiento
 - Relativo a PC (Program Counter), Relativo a Base, Indexado (relativo a registro índice), Relativo a SP (Stack Pointer) ...

Ejemplo Relativo a Base

$$A = ? + C$$

Base
11101

ADD4 A, B, C → 1001001000100101

Memoria	
M(00000)	00010
M(00001)	00100
M(00010)	00000
...	...
M(11111)	00101

Registros	
R0 (0000)	00110
R1 (0001)	00001
R2 (0010)	00111
...	...
R15 (1111)	00000

3. MODOS DE DIRECCIONAMIENTO (IV)

Directo Relativo con registro implícito:

- El registro no se especifica explícitamente en el formato de la instrucción
- Sólo se especifica el desplazamiento
 - Relativo a PC (Program Counter), Relativo a Base, Indexado (relativo a registro índice), Relativo a SP (Stack Pointer) ...

Ejemplo Relativo a Base

$$A = 5 + C$$

Base
11101

DE = 11111

ADD4 A, B, C → 1001001000100101

Desplazamiento

00101 = 5

Memoria

M(00000)	00010
M(00001)	00100
M(00010)	00000
...	...
M(11111)	00101

Registros

R0 (0000)	00110
R1 (0001)	00001
R2 (0010)	00111
...	...
R15 (1111)	00000

3. MODOS DE DIRECCIONAMIENTO (V)

Directo Relativo con registro explícito:

- El registro se especifica explícitamente en el formato de la instrucción
- En el formato de instrucción al operando se le asocian dos campos: registro y desplazamiento

Ejemplo relativo a registro:

$$A = ? + C$$

Desplazamiento
ADD5 A, B(D), C → 1011001111000010101
Registro

Memoria	
M(00000)	00010
M(00001)	00100
M(00010)	00000
...	...
M(11111)	00101

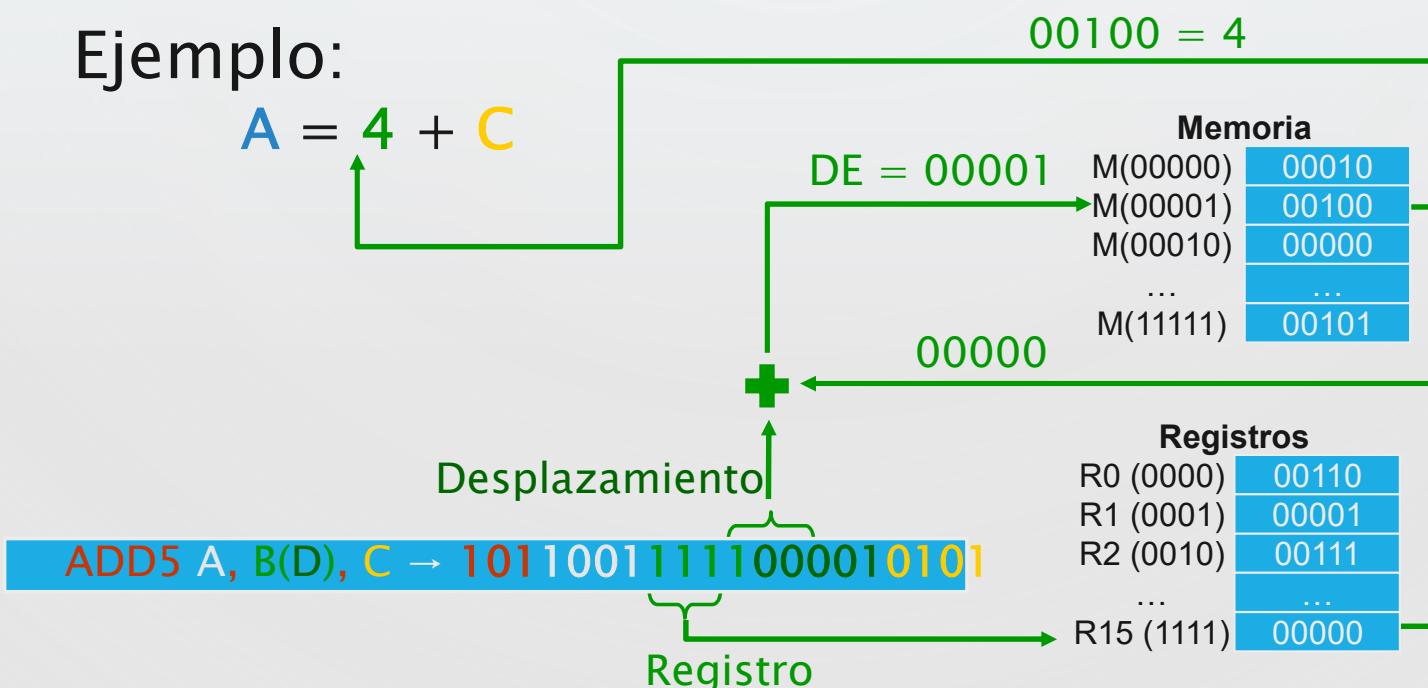
Registros	
R0 (0000)	00110
R1 (0001)	00001
R2 (0010)	00111
...	...
R15 (1111)	00000

3. MODOS DE DIRECCIONAMIENTO (V)

Directo Relativo con registro explícito:

- El registro se especifica explícitamente en el formato de la instrucción
- En el formato de instrucción al operando se le asocian dos campos: registro y desplazamiento

Ejemplo:



3. MODOS DE DIRECCIONAMIENTO (V)

Directo Relativo

Principal utilidad: recorrer conjunto de direcciones de memoria con una misma instrucción.

```
Loop:  
...  
ADD A,R0(0)  
INC R0  
...  
JUMP Loop
```

Memoria	
M(00000)	00010
M(00001)	00100
M(00010)	00000
...	...
M(11111)	00101

3. MODOS DE DIRECCIONAMIENTO (V)

Directo Relativo

Principal utilidad: recorrer conjunto de direcciones de memoria con una misma instrucción.



3. MODOS DE DIRECCIONAMIENTO (V)

Directo Relativo

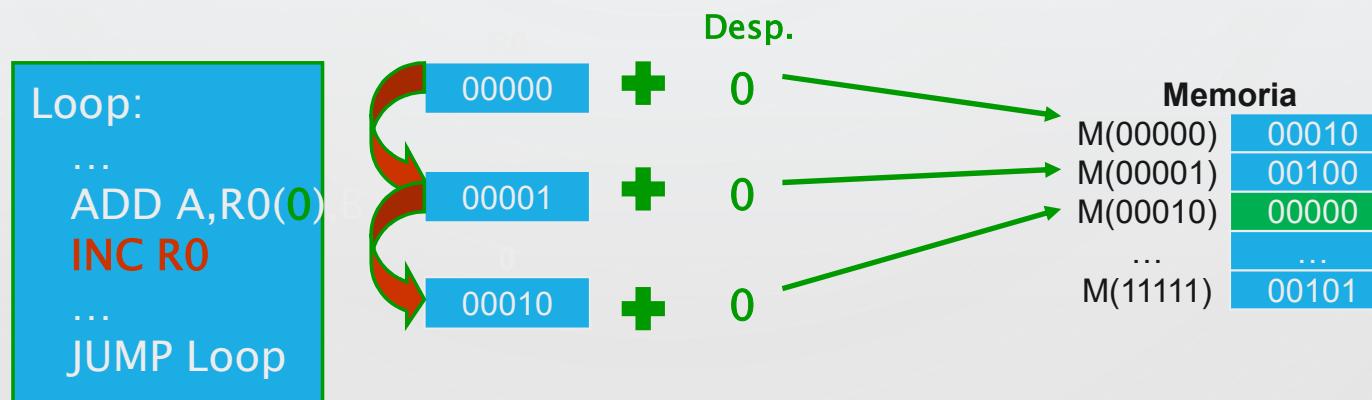
Principal utilidad: recorrer conjunto de direcciones de memoria con una misma instrucción.



3. MODOS DE DIRECCIONAMIENTO (V)

Directo Relativo

Principal utilidad: recorrer conjunto de direcciones de memoria con una misma instrucción.



3. MODOS DE DIRECCIONAMIENTO (V)

Directo Relativo Autoindexado

- Incremento/decremento automático del registro
- Útil en bucles como el anterior (ahorro instrucción INC)
- ADD A,R0(0),B
 - Preautoincrementado: $R0=R0+1$; $DE=R0+0$;
 - Preautodecrementado: $R0=R0-1$; $DE=R0+0$;
 - Postautoincrementado: $DE=R0+0$; $R0=R0+1$;
 - Postautodecrementado: $DE=R0+0$; $R0=R0-1$;

3. MODOS DE DIRECCIONAMIENTO

Indirecto

- El campo operando indica **DONDE** está la **DIRECCIÓN** del operando
- El operando está en una posición de memoria
- La dirección de memoria donde está el operando puede estar en:
 - **Registro**: el campo operando indica el código del registro
 - **Memoria**: el campo operando indica la dirección de memoria

3. MODOS DE DIRECCIONAMIENTO (VI)

Indirecto por registro:

- El formato de instrucción indica el código del registro donde se almacena la posición de memoria donde está el operando.
- Ejemplo:

$$A = ? + C$$

ADD6 A, B, C → 1101000100100101

Memoria	
M(00000)	00010
M(00001)	00100
M(00010)	00000
...	...
M(11111)	00101

Registros	
R0 (0000)	00110
R1 (0001)	00001
R2 (0010)	11111
...	...
R15 (1111)	00000

3. MODOS DE DIRECCIONAMIENTO (VI)

Indirecto por registro:

- El formato de instrucción indica el código del registro donde se almacena la posición de memoria donde está el operando.
- Ejemplo:

$$A = 5 + C$$

00101 = 5 (en C2)

Memoria

M(00000)	00010
M(00001)	00100
M(00010)	00000
...	...
M(11111)	00101

ADD6 A, B, C → 1101000100100101

Registro

Registros

R0 (0000)	00110
R1 (0001)	00001
R2 (0010)	11111
...	...
R15 (1111)	00000

3. MODOS DE DIRECCIONAMIENTO (VII)

Indirecto por memoria:

- El formato de instrucción indica la dirección de la memoria donde se almacena la posición de memoria donde está el operando.
- Ejemplo:

$$A = ? + C$$

ADD7 A, B, C → 1111001000100101

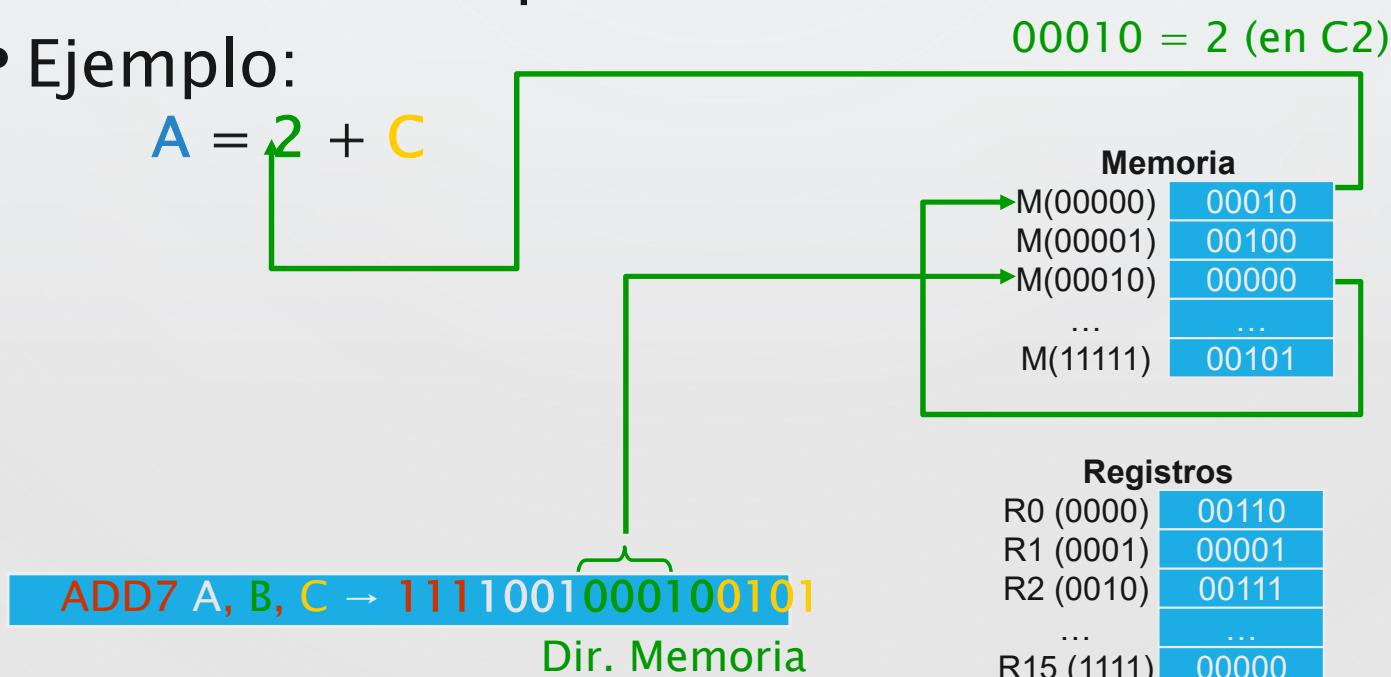
Memoria	
M(00000)	00010
M(00001)	00100
M(00010)	00000
...	...
M(11111)	00101

Registros	
R0 (0000)	00110
R1 (0001)	00001
R2 (0010)	00111
...	...
R15 (1111)	00000

3. MODOS DE DIRECCIONAMIENTO (VII)

Indirecto por memoria:

- El formato de instrucción indica la dirección de la memoria donde se almacena la posición de memoria donde está el operando.
- Ejemplo:



4. EJEMPLO FORMATO INSTRUCCIÓN

Diseñar el formato de instrucción para un procesador con las siguientes características:

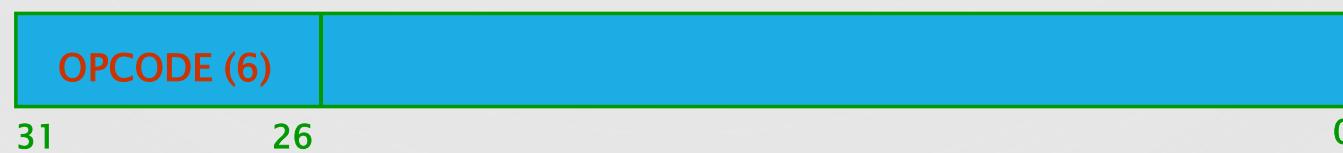
- 64 instrucciones de 32 bits
- 32 registros de 32 bits
- 3 tipos con modo de direccionamiento: Directo a registro, directo a memoria y relativo a PC
- Instrucciones con 2 operandos:
 - 1^{er} operando especifica un registro
 - 2º operando una posición de memoria



4. EJEMPLO FORMATO INSTRUCCIÓN

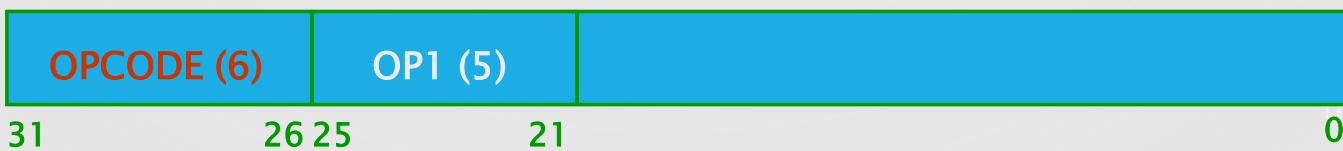
Diseñar el formato de instrucción para un procesador con las siguientes características:

- 64 instrucciones de 32 bits $\Rightarrow 64 = 2^6 \Rightarrow 6$ bits (OPCODE)
- 32 registros de 32 bits
- 3 tipos con modo de direccionamiento: Directo a registro, directo a memoria y relativo a PC
- Instrucciones con 2 operandos:
 - 1^{er} operando especifica un registro
 - 2º operando una posición de memoria



4. EJEMPLO FORMATO INSTRUCCIÓN

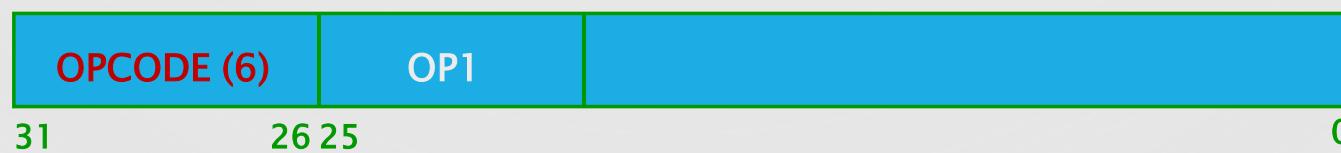
- Diseñar el formato de instrucción para un procesador con las siguientes características:
 - 64 instrucciones de 32 bits $\Rightarrow 64 = 2^6 \Rightarrow 6$ bits (OPCODE)
 - 32 registros de 32 bits
 - 3 modos de direccionamiento: Directo a registro, directo a memoria y relativo a PC
 - Instrucciones con 2 operandos:
 - 1^{er} operando especifica un registro $\Rightarrow 32 = 2^5 \Rightarrow 5$ bits (OP1)
 - 2º operando una posición de memoria



4. EJEMPLO FORMATO INSTRUCCIÓN

Diseñar el formato de instrucción para un procesador con las siguientes características:

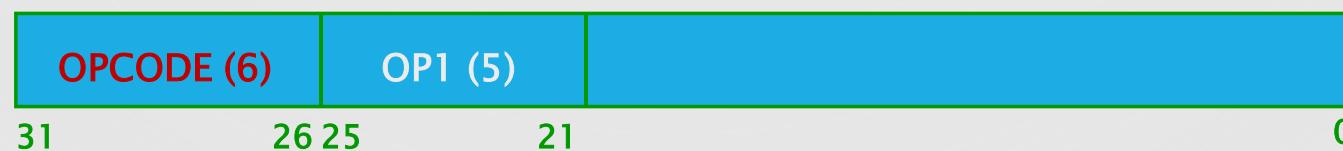
- 64 instrucciones de 32 bits => $64 = 2^6 \Rightarrow 6$ bits (OPCODE)
- 32 registros de 32 bits
- 3 tipos con modo de direccionamiento: Directo a registro, directo a memoria y relativo a PC
- Instrucciones con 2 operandos:
 - 1^{er} operando especifica un registro
 - 2º operando una posición de memoria



4. EJEMPLO FORMATO INSTRUCCIÓN

Diseñar el formato de instrucción para un procesador con las siguientes características:

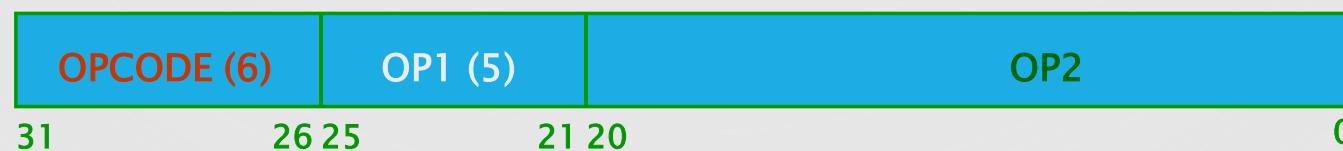
- 64 instrucciones de 32 bits $\Rightarrow 64 = 2^6 \Rightarrow 6$ bits (OPCODE)
- 32 registros de 32 bits
- 3 tipos con modo de direccionamiento: Directo a registro, directo a memoria y relativo a PC
- Instrucciones con 2 operandos:
 - 1^{er} operando especifica un registro
 - 2º operando una posición de memoria



4. EJEMPLO FORMATO INSTRUCCIÓN

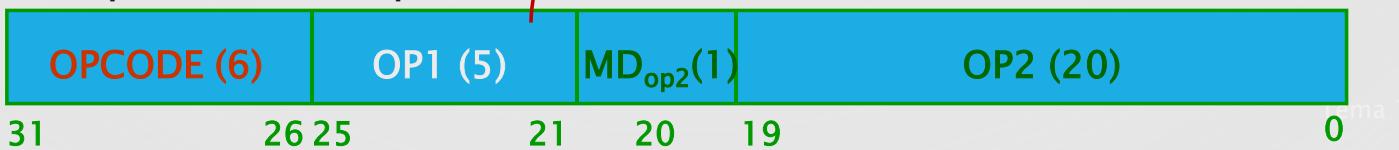
Diseñar el formato de instrucción para un procesador con las siguientes características:

- 64 instrucciones de 32 bits => $64 = 2^6 \Rightarrow 6$ bits (OPCODE)
- 32 registros de 32 bits
- 3 tipos con modo de direccionamiento: Directo a registro, directo a memoria y relativo a PC → tipo definido por el **opcode**: OP2 es desplazamiento o dirección de memoria
- Instrucciones con 2 operandos:
 - 1^{er} operando especifica un registro
 - 2º operando una posición de memoria



4. EJEMPLO FORMATO INSTRUCCIÓN

- Diseñar el formato de instrucción para un procesador con las siguientes características:
 - 64 instrucciones de 32 bits $\Rightarrow 64 = 2^6 \Rightarrow 6$ bits (OPCODE)
 - 32 registros de 32 bits
 - 3 modos de direccionamiento: Directo a registro, directo a memoria y relativo a PC
- Instrucciones con 2 operandos: $2 = 2^1 \Rightarrow 1$ bit (MD_{op2})
 - 1^{er} operando especifica un registro $\Rightarrow 32 = 2^5 \Rightarrow 5$ bits (OP1)
 - 2º operando una posición de memoria



TEMA 1

1. Introducción

1. ¿Qué es un computador?
2. Programación de un procesador
3. Ejecución de un programa

2. Instrucciones de un procesador

1. Conjunto de instrucciones
2. Formato de instrucción
3. Modos de direccionamiento

3. Programación en ensamblador

1. Lenguaje ensamblador RISC-V
2. Traducción de estructuras de alto nivel a ensamblador

TEMA 1

PROGRAMACIÓN EN ENSAMBLADOR

LENGUAJE ENSAMBLADOR RISC-V

Transparencias basadas en:

- José Manuel Mendías Cuadros. Fundamentos de los Computadores II. UCM.
- Chris Terman. Computation Structures. MIT Open Courseware



ARQUITECTURA RISC-V

- RISC-V ISA (Instruction Set Architecture) es una arquitectura:
 - Abierta, no propietaria y en evolución
- Es de tipo RISC (Reduced Instruction Set Computer) por lo que:
 - Tiene un repertorio reducido de instrucciones simples
 - Sólo las instrucciones de carga y almacenamiento acceden a memoria
 - El resto de instrucciones trabajan con datos almacenados en registros
 - Dispone de un gran número de registros de propósito general
 - Tienen un conjunto reducido de modos de direccionamiento
 - Instrucciones de tamaño fijo y con un número reducido de formatos
- Trabajaremos con el repertorio base RV32I con la extensión RVM
 - Datos enteros de 32 bits e instrucciones de 32 bits (RV32I)
 - Con operaciones de multiplicación y división sobre enteros (RVM)

INSTRUCCIONES Y DATOS

- Toda **instrucción** en RISC-V ocupa **32 bits**
- Las instrucciones operan con **datos** o **direcciones** de **32 bits**
 - Los **datos** son números **enteros** (con signo, codificados en **complemento a 2**) o **naturales** (sin signo, codificados en **binario puro**)
 - Las **direcciones** son números **naturales** codificadas en **binario puro**
- Se puede trabajar con números de menor anchura
 - Son extendidos a 32 bits (signo los enteros (**sExt**) y 0s los naturales (**zExt**))
- Tamaños comunes de datos:
 - Palabra (**word**): 32 bits
 - Media palabra (**half**): 16 bits
 - Byte (**byte**): 8 bits

MODELO DE MEMORIA

- Memoria RAM direccionable de **4GiB** ($2^{32} \times 8\text{bytes} = 2^{30} \times 32\text{bytes}$)
 - Bus de datos y direcciones de 32 bits
 - Direccionable a nivel de byte Cada byte tiene una única dirección de memoria
 - Contiene datos de 8, 16 y 32 bits y direcciones de 32 bits
 - Todos ellos alineados y con ordenación little-endian

MODELO DE MEMORIA: ALINEAMIENTO

- La información almacenada en memoria está **alineada** (restricciones de ubicación según su tamaño):
 - **Byte**: puede ubicarse en cualquier dirección de memoria
 - **Media palabra**: sólo puede ubicarse en dir. múltiplo de 2
 - **Palabra**: sólo puede ubicarse en dir. múltiplo de 4
- Si se ubican consecutivamente datos de distinto tamaño quedan huecos vacíos

Tamaño	Dato
byte	0x24
palabra	0x3b257a02
½ palabra	0x3e27
palabra	0x01c6d823

Dir.	+0	+1	+2	+3
3c000000	24			
3c000004		3b257a02		
3c000008	3e27			
3c00000c		01c6d823		

RISC-V: Datos alineados

Dir.	+0	+1	+2	+3
3c000000	24		3b257a	
3c000004	02	3e27		01
3c000008		6d823		
3c00000c				

Datos no alineados

MODELO DE MEMORIA: ORDENAMIENTO

- Los bytes de una palabra tienen ordenamiento **little-endian**
 - En la **dirección más baja** se ubica el **byte menos significativo**
 - La dirección de un dato coincide con la dirección de su byte menos significativo
- Otros procesadores tienen ordenamiento **big-endian**
 - En la dirección más baja se ubica el byte más significativo

Tamaño	Dato
byte	0x24
palabra	0x3b257a02
½ palabra	0x3e27
palabra	0x01c6d823

Dir.	+0	+1	+2	+3
3c000000	24			
3c000004	02	7a	25	3b
3c000008	27	3e		
3c00000c	23	d8	c6	01

RISC-V: Little-Endian

Dir.	+0	+1	+2	+3
3c000000	24			
3c000004	3b	25	7a	02
3c000008	3e	27		
3c00000c	01	c6	d8	23

Big-Endian

REGISTROS

- Todos los **datos** residen en memoria, pero para ser operados deben **cargarse en registros**
- Un RISC-V dispone de **32 registros de 32 bits** de propósito general
 - Se numeran del x_0 al x_{31}
 - El **registro x_0** contiene la **constante 0** (no se puede sobrescribir)
 - Cada registro tiene un **alias** (para facilitar la programación)
- Adicionalmente dispone del registro especial, **PC (Program Counter)**
 - Contiene la **dirección** de memoria de la **instrucción a ejecutar**
 - Al terminar de ejecutarla **se incrementa en 4** (cada instrucción ocupa 4 bytes)
 - Excepto en instrucciones de salto

REGISTROS

# Reg.	Alias	Descripción
x0	zero	zero – cero
x1	ra	return address – dirección de retorno
x2	sp	stack pointer – puntero de pila
x3	gp	global pointer – puntero global
x4	tp	thread pointer – puntero de hebra
x5...x7	t0...t2	temporary register – registro temporal
x8	s0/fp	saved register / frame pointer – registro preservado / puntero de marco
x9	s1	saved register – registro preservado
x10...x17	a0...a7	argument register – registro de argumento
x18...x27	s2...s11	saved register – registro preservado
x28...x31	t3...t6	temporary register – registro temporal

programando en ensamblador deben **usarse siempre los alias**

MODOS DE DIRECCIONAMIENTO

- Indican **dónde** se encuentran los operandos de una instrucción
- Los operandos pueden estar ubicados en:
 - La propia **instrucción**
 - Un **registro** del procesador (se indicará cuál)
 - La **memoria** del computador (se indicará la dirección que ocupa)
- En RISC-V sólo existen 4 modos de direccionamiento:
 - **Inmediato**: el operando es una **constante** ubicada en la propia **instrucción**
 - **Directo a registro**: el operando se encuentra en un **registro** del procesador, se indica cual
 - **Relativo a registro**: el operando se encuentra en **memoria**, su dirección se obtiene sumando el contenido de un registro y un inmediato
 - **Relativo a PC**: el operando es una **dirección** (de salto), se obtiene sumando el contenido del PC y un inmediato

MODOS DE DIRECCIONAMIENTO: INMEDIATO

El operando es una **constante** ubicada en la propia **instrucción**

- En **ensamblador** se indica explícitamente la **constante** con la que operar:

`addi x1, x1, 14`

- La codificación de la instrucción contiene un **campo** que almacena la **constante**:

14

addi

Las instrucciones son de 32 bits → los operandos inmediatos son de menor anchura

- **Inmediatos sin signo de 5 bits**: se usan sin extender
- **Inmediatos con signo de 12/13 bits**: se extienden a 32 bits antes de usarlos
- **Inmediatos de 20 bits**: se usan sin extender pero desplazados
- **Inmediatos de 21 bits**: se extienden a 32 bits antes de usarlos

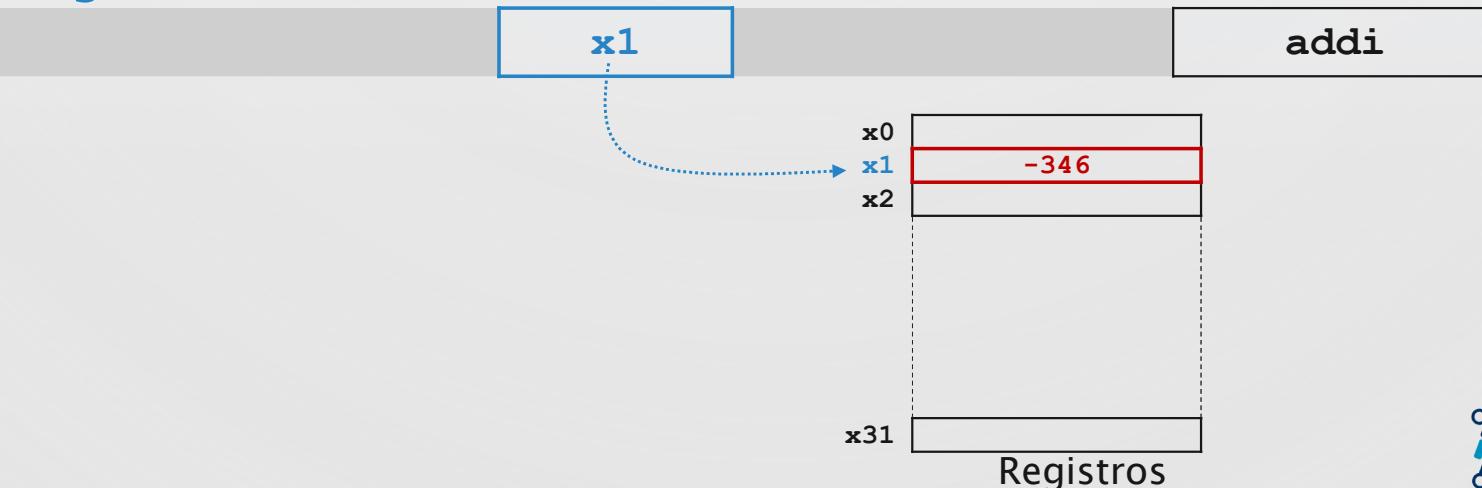
MODOS DE DIRECCIONAMIENTO: DIRECTO A REGISTRO

El operando está almacenado en un **registro del procesador**

- En **ensamblador** se indica el nombre del registro que contiene el dato:

addi x1, x1, 14

- La codificación de la instrucción contiene un **campo que indica el número del registro**:



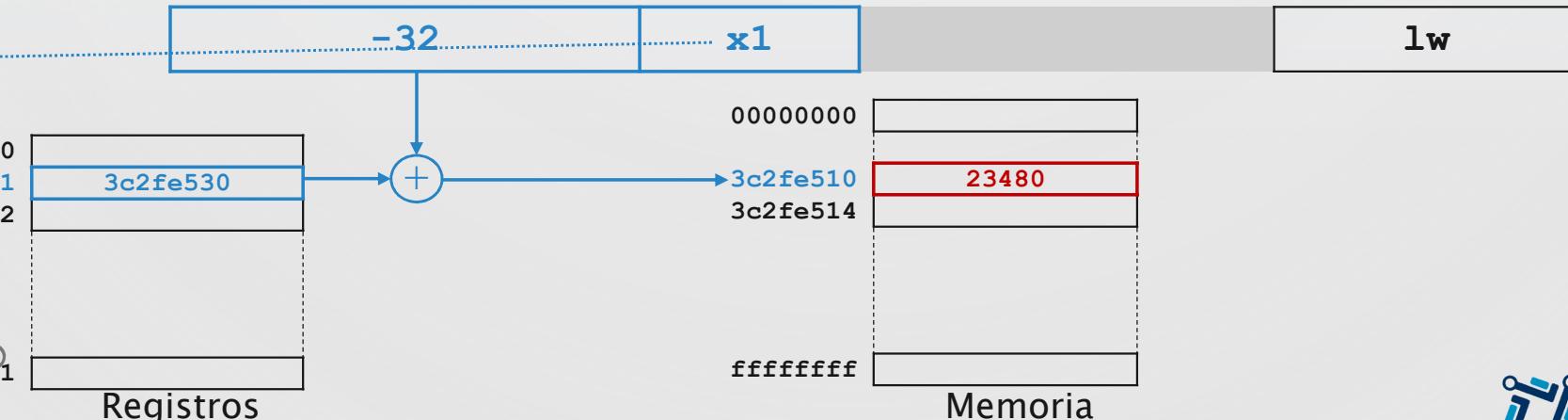
MODOS DE DIRECCIONAMIENTO: RELATIVO A REGISTRO

El operando está en una posición de memoria cuya dirección se calcula:

- Sumando el contenido de un registro y un desplazamiento constante contenido en la instrucción
- En ensamblador se indica explícitamente el desplazamiento y el registro:

`lw x3, -32(x1)`

- La codificación de la instrucción contiene campos para indicar ambos elementos:



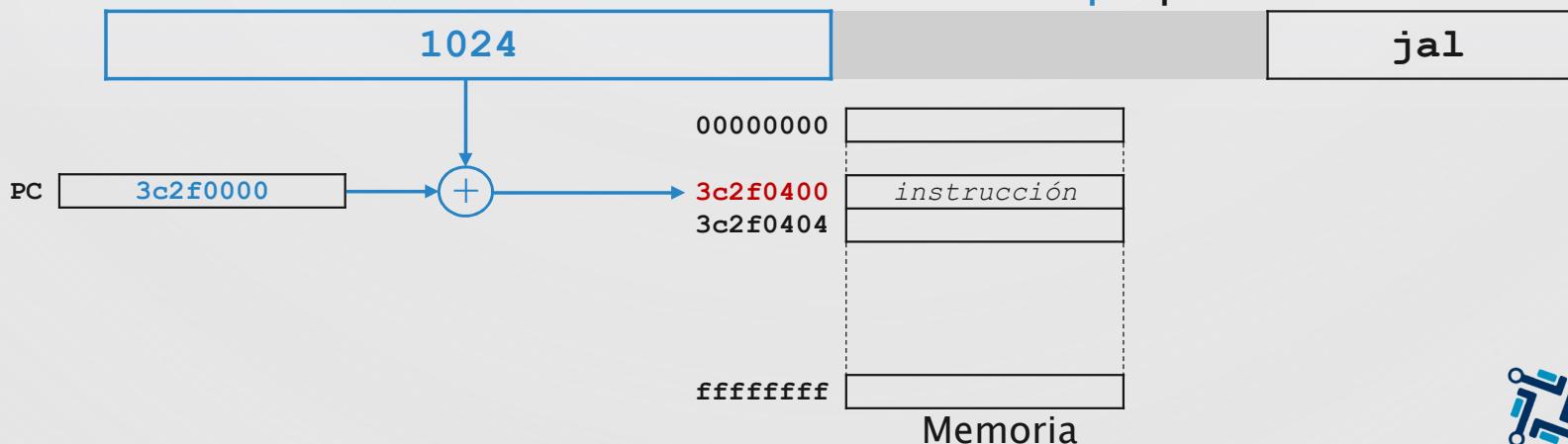
MODOS DE DIRECCIONAMIENTO: RELATIVO A PC

El operando es una dirección (de salto) que se calcula:

- Sumando el contenido del PC y un desplazamiento constante contenido en la instrucción
- En ensamblador se indica explícitamente el desplazamiento:

`jal x3, 1024`

- La codificación de la instrucción contiene un campo para indicar el desplazamiento:



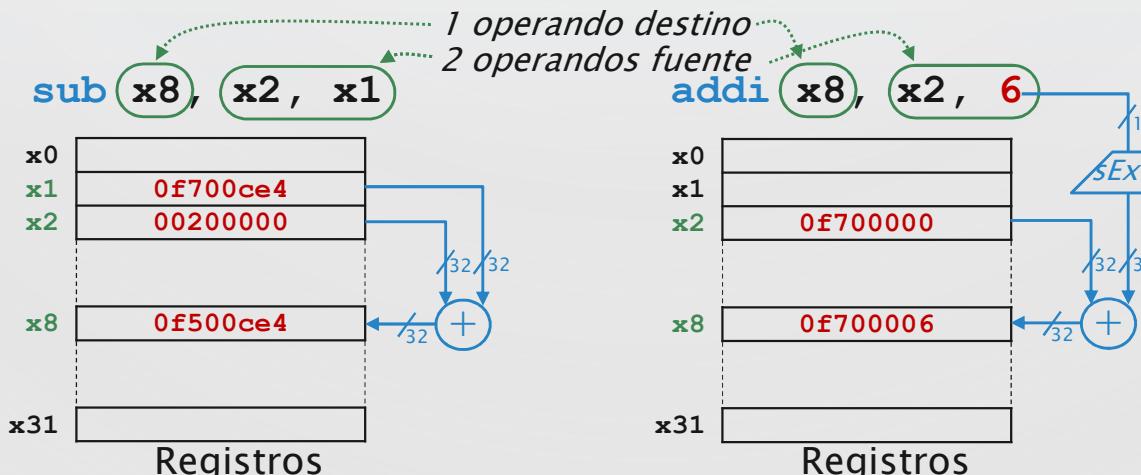
REPERTORIO DE INSTRUCCIONES

- El **repertorio de instrucciones** es el conjunto de todas las instrucciones que puede ejecutar un procesador
- Hay diferentes tipos de instrucciones:
 - **Transformación de datos**: realizan operaciones aritméticas, lógicas bit a bit y de desplazamiento de bits.
 - **Transferencia de datos**: copian datos entre registros y memoria
 - **Control o salto**: rompen el orden implícito de ejecución (modifican PC)
 - **Privilegiadas**: permiten acceso a funcionalidades para control del sistema
- El repertorio de instrucciones del RISC-V es extremadamente reducido

REPERTORIO DE INSTRUCCIONES: ARITMÉTICAS

Realizan operaciones aritméticas con 2 operandos fuente y 1 operando destino, todos de 32 bits

- El **operando izquierdo** se encuentra SIEMPRE en un **registro**
- El **operando derecho** se encuentra en un **registro** o es un **inmediato corto**
 - La constante inmediata es de 12 bits en C2 ($[-2048, +2048]$) pero se extiende a 32 bits
- El **resultado** se almacena SIEMPRE en un **registro**



REPERTORIO DE INSTRUCCIONES: ARITMÉTICAS

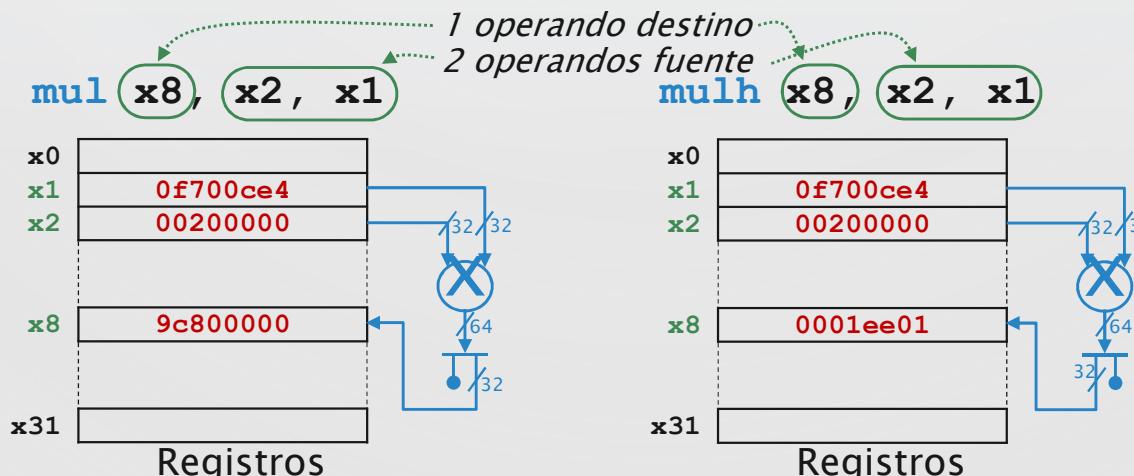
	Operación	Descripción
<code>add rd, rs1, rs2</code>	$rd \leftarrow rs1 + rs2$	<code>add</code> <code>suma</code>
<code>sub rd, rs1, rs2</code>	$rd \leftarrow rs1 - rs2$	<code>subtract</code> <code>resta</code>
<code>slt rd, rs1, rs2</code>	$rd \leftarrow \text{if } (rs <_S rs2) \text{ then } (1) \text{ else } (0)$	<code>set if less than</code> “menor que” con signo
<code>sltu rd, rs1, rs2</code>	$rd \leftarrow \text{if } (rs <_U rs2) \text{ then } (1) \text{ else } (0)$	<code>set if less than unsigned</code> “menor que” sin signo
<code>addi rd, rs1, imm_{12b}</code>	$rd \leftarrow rs1 + sExt(imm)$	Instrucción
<code>slti rd, rs1, imm_{12b}</code>	$rd \leftarrow \text{if } (rs <_S sExt(imm)) \text{ then } (1) \text{ else } (0)$	<code>set if less than immediate</code> “menor que constante” con signo
<code>sltiu rd, rs1, imm_{12b}</code>	$rd \leftarrow \text{if } (rs <_U sExt(imm)) \text{ then } (1) \text{ else } (0)$	<code>set if less than immediate unsigned</code> “menor que constante” sin signo

Existen **instrucciones diferentes** de comparación para interpretar operandos fuente como **datos con y sin signo**

REPERTORIO DE INSTRUCCIONES: MULTIPLICACIÓN Y DIVISIÓN

El resultado de multiplicar 2 datos de 32 bits requiere 64 bits

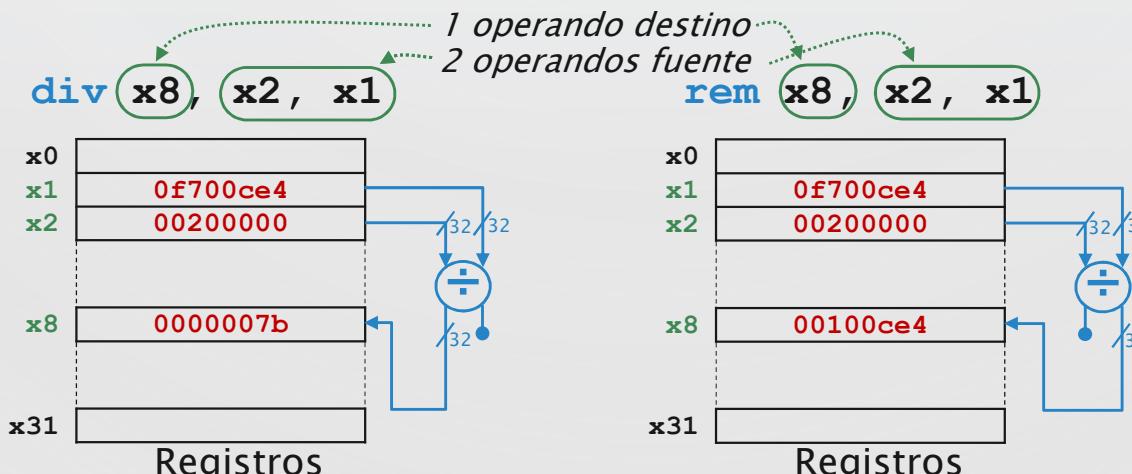
- Por ello existen **2 tipos de instrucciones de multiplicación**: una para la **parte alta** (más significativa) del resultado y otra para la **parte baja** (menos significativa)
 - Sólo existe una instrucción para obtener la parte baja del resultado.
 - Existen dos instrucciones para la parte alta según los operandos fuente sean con o sin signo
- Todos los **operандos** de estas instrucciones se encuentran en **registros**



REPERTORIO DE INSTRUCCIONES: MULTIPLICACIÓN Y DIVISIÓN

La **división entera de 2 datos de 32 bits** da lugar a dos resultados: el cociente y el resto, ambos de 32 bits

- Existen **dos tipos de instrucciones**: una para obtener el **cociente** y otra para el **resto**
- Cada una con variantes para operar con **datos con y sin signo**
- Todos los **operандos** de estas instrucciones se encuentran en **registros**



REPERTORIO DE INSTRUCCIONES: MULTIPLICACIÓN Y DIVISIÓN

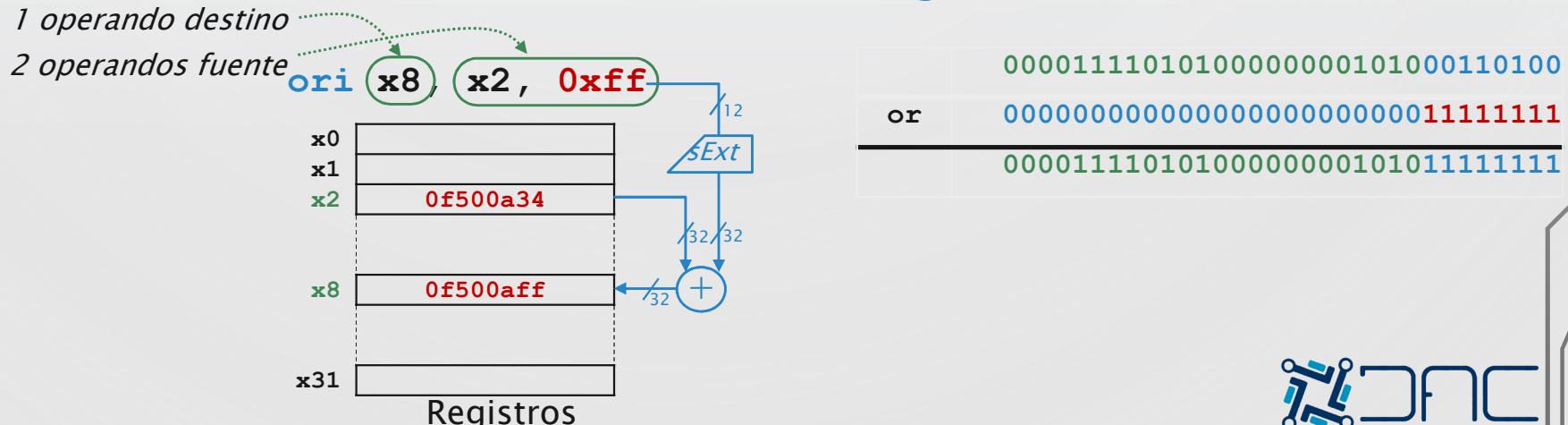
Instrucción	Operación	Descripción
<code>mul rd, rs1, rs2</code>	$rd \leftarrow (rs1 * rs2)_{31:0}$	multiply multiplicación entera (32 bits menos significativos)
<code>mulh rd, rs1, rs2</code>	$rd \leftarrow (rs1_S * rs2_S)_{63:32}$	multiply high multiplicación entera con signo (32 bits más significativos)
<code>mulhsu rd, rs1, rs2</code>	$rd \leftarrow (rs1_S * rs2_U)_{63:32}$	multiply high signed unsigned multiplicación entera mixta (32 bits más significativos)
<code>mulhu rd, rs1, rs2</code>	$rd \leftarrow (rs1_U * rs2_U)_{63:32}$	multiply high unsigned multiplicación entera sin signo (32 bits más significativos)
<code>div rd, rs1, rs2</code>	$rd \leftarrow rs1 /_S rs2$	divide división entera con signo
<code>divu rd, rs1, rs2</code>	$rd \leftarrow rs1 /_U rs2$	divide unsigned división entera sin signo
<code>rem rd, rs1, rs2</code>	$rd \leftarrow rs1 \%_S rs2$	remainder resto entero con signo
<code>remu rd, rs1, rs2</code>	$rd \leftarrow rs1 \%_U rs2$	remainder unsigned resto entero sin signo

Estas instrucciones no forman parte del repertorio RV32I pero sí de su extensión RVM

REPERTORIO DE INSTRUCCIONES: LÓGICAS

Realizan operaciones de tipo lógico bit a bit (bitwise) con 2 operandos fuente y 1 operando destino, todos de 32 bits

- El **operando izquierdo** se encuentra SIEMPRE en un **registro**
- El **operando derecho** se encuentra en un **registro** o es un **inmediato corto**
 - La **constante inmediata** es de 12 bits en C2 pero se extiende su signo a 32 bits
- El **resultado** se almacena SIEMPRE en un **registro**



REPERTORIO DE INSTRUCCIONES: LÓGICAS

Las operaciones de tipo lógico bit a bit se usan para **manipular los bits individuales** de un dato

- Un **operando** contiene el **dato a manipular**
- El otro **operando** contiene una **máscara** que indica los bits a cambiar
- Se utilizará una operación distinta según el cambio que se desee

*La instrucción **or** pone a 1 aquellos bits del dato cuyos correspondientes en la máscara estén a 1*

*La instrucción **and** pone a 0 aquellos bits del dato cuyos correspondientes en la máscara estén a 0*

*La instrucción **xor** complementa aquellos bits del dato cuyos correspondientes en la máscara estén a 1*

or	0000111010100000000101000110100 000000000000000000000000000000001111111 0000111010100000000101011111111	----- dato máscara dato manipulado
and	0000111010100000000101000110100 000000000000000000000000000000001111111 0000000000000000000000000000000000110100	
xor	0000111010100000000101000110100 000000000000000000000000000000001111111 0000111010100000000101011001011	

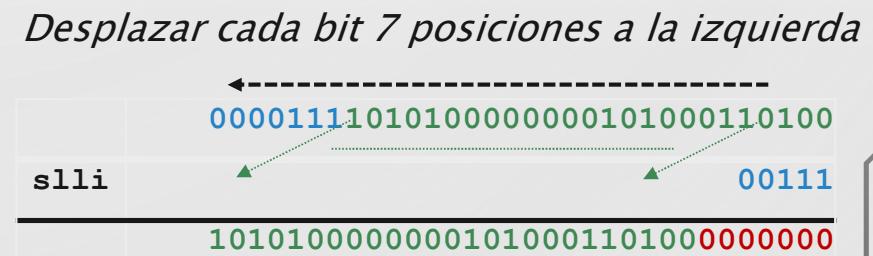
REPERTORIO DE INSTRUCCIONES: LÓGICAS

Instrucción	Operación	Descripción
<code>and rd, rs1, rs2</code>	$rd \leftarrow rs1 \& rs2$	and "y lógica" bit a bit
<code>or rd, rs1, rs2</code>	$rd \leftarrow rs1 rs2$	or "o lógica" bit a bit
<code>xor rd, rs1, rs2</code>	$rd \leftarrow rs1 \wedge rs2$	xor "o-exclusiva lógica" bit a bit
<code>andi rd, rs1, imm_{12b}</code>	$rd \leftarrow rs1 \& sExt(imm)$	and immediate "y lógica" bit a bit con constante
<code>ori rd, rs1, imm_{12b}</code>	$rd \leftarrow rs1 sExt(imm)$	or immediate "o lógica" bit a bit con constante
<code>xori rd, rs1, imm_{12b}</code>	$rd \leftarrow rs1 \wedge sExt(imm)$	xor immediate "o-exclusiva lógica" bit a bit con constante

REPERTORIO DE INSTRUCCIONES: DE DESPLAZAMIENTO

Permiten desplazar los bits de un operando fuente un número de posiciones indicadas por otro y almacenarlo en un operando destino

- El **operando izquierdo de 32 bits** se encuentra SIEMPRE en un **registro**
- El **operando derecho de 5 bits** se encuentra en un **registro** o es un **inmediato**
 - Del **registro** se toman los **5 bits menos significativos**
 - La **constante inmediata** son **5 bits en binario puro** que no se extienden
- El **resultado** se almacena SIEMPRE en un **registro**



REPERTORIO DE INSTRUCCIONES: DE DESPLAZAMIENTO

Insertan 0's por un extremo del dato y descartan los bits que salen por el otro

Permiten el **reescalamiento de datos sin signo**:

- Desplazar **n bits a la izquierda** equivale a **multiplicar por 2^n**
- Desplazar **n bits a la derecha** equivale a **dividir por 2^n**

Desplazar cada bit 7 posiciones a la izquierda →

slli	00000000000000000000000000000000101000110100 00111	00000000000000000000000000000000101000110100 0000000
	2612 << 7 = 2612 × 2 ⁷ = 334336	

Desplazar cada bit 7 posiciones a la derecha →

srlt	00000000000000000000000000000000101000110100 00111	00000000000000000000000000000000101000110100 0000000
	2612 >> 7 = 2612 ÷ 2 ⁷ = 20	

- Aplicadas tras una lógica bit a bit permiten **extraer campos** de un dato:

and	0000111101000000001010001101000 00000000111100000000000000000000 00000000101000000000000000000000
slli	10100 0000000000000000000000000000000010100

Aísla los bits 20 al 23 del dato

Los desplaza al extremo derecho
bits menos significativos

REPERTORIO DE INSTRUCCIONES: DE DESPLAZAMIENTO

La instrucción de **desplazamiento aritmético a la derecha** propaga el bit de signo por la izquierda y descarta los bits que salen por la derecha

- Permite el **reescalamiento** de datos **con signo**:

Desplazar cada bit 7 posiciones a la derecha

srai 
111111111111110001101000110100
00111
11111111111111111111000110100

$$-58828 \gg 7 = -58828 \div 2^7 = -460$$

Desplazar cada bit 7 posiciones a la derecha

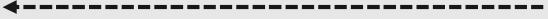
srai 
011111111111110001101000110100
00111
0000000011111111111111000110100

$$2147424820 \gg 7 = 2147424820 \div 2^7 = 16776756$$

No existe instrucción de desplazamiento aritmético a la izquierda

- Para resultados válidos (el dato reescalado con signo puede representarse con 32 bits) el desplazamiento lógico es equivalente

Desplazar cada bit 7 posiciones a la izquierda

slli 
111111111111110001101000110100
00111
1111111100011010001101000000000

$$-58828 \ll 7 = -58828 \times 2^7 = -7529984$$

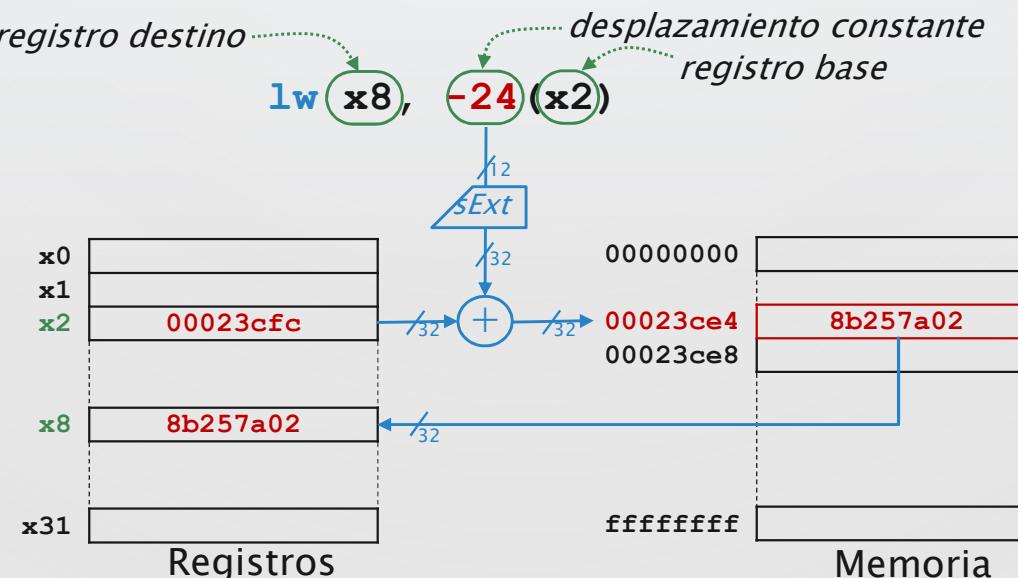
REPERTORIO DE INSTRUCCIONES: DE DESPLAZAMIENTO

Instrucción	Operación	Descripción
sll rd, rs1, rs2	$rd \leftarrow rs1 \ll rs2_{4:0}$	shift left logical desplazamiento lógico a izquierda
srl rd, rs1, rs2	$rd \leftarrow rs1 \gg rs2_{4:0}$	shift right logical desplazamiento lógico a derecha
sra rd, rs1, rs2	$rd \leftarrow rs1 \ggg rs2_{4:0}$	shift right arithmetical desplazamiento aritmético a derecha
slli rd, rs1, imm_{5b}	$rd \leftarrow rs1 \ll imm$	shift left logical immediate desplazamiento lógico a izquierda con constante
srlti rd, rs1, imm_{5b}	$rd \leftarrow rs1 \gg imm$	shift right logical immediate desplazamiento lógico a derecha con constante
srai rd, rs1, imm_{5b}	$rd \leftarrow rs1 \ggg imm$	shift right arithmetical immediate desplazamiento aritmético a derecha con constante

REPERTORIO DE INSTRUCCIONES: DE TRANSFERENCIA DE DATOS: CARGA

Permiten **copiar** un dato de **memoria** a un **registro**

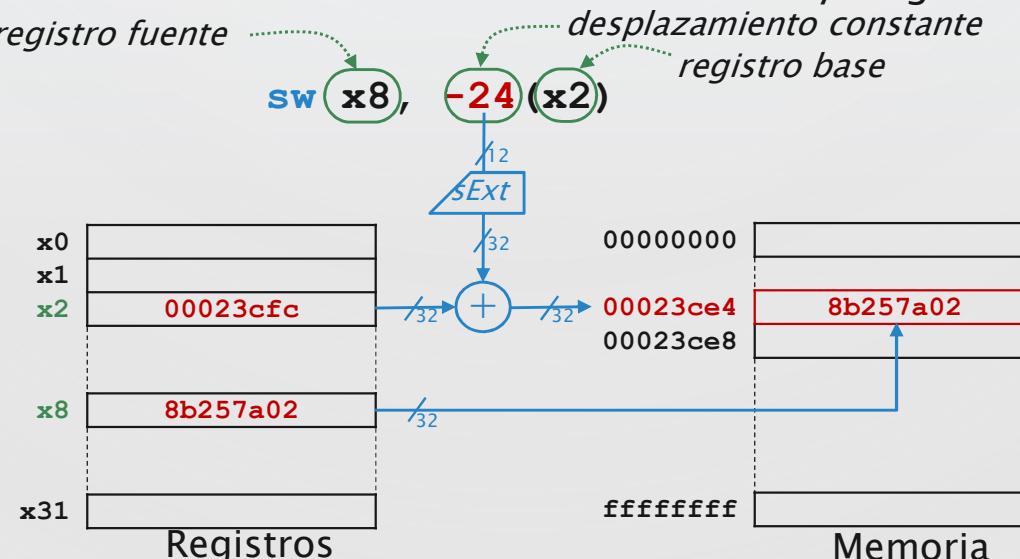
- Utiliza **direcciónamiento relativo a registro base** para indicar la dirección de memoria que ocupa el dato (la dirección es la suma de una dirección base y un desplazamiento)
 - La **dirección base** se encuentra en un **registro**
 - El **desplazamiento** es un **inmediato** de 12 bits en C2 cuyo signo se extiende a 32 bits
- El **dato leído de memoria** se carga en un **registro**



REPERTORIO DE INSTRUCCIONES: DE TRANSFERENCIA DE DATOS: ALMACENAMIENTO

Permiten copiar un dato de un registro a memoria

- El dato se encuentra en un registro
- Utiliza direccionamiento relativo a registro base para indicar la dirección de memoria en donde almacenar el dato (la dirección es la suma de una dir. base y un desplazamiento)
 - La dirección base se encuentra en un registro
 - El desplazamiento es un inmediato de 12 bits en C2 cuyo signo se extiende a 32 bits



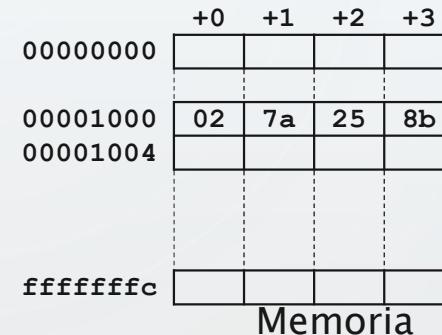
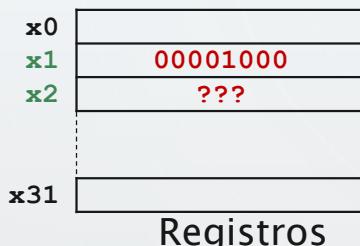
REPERTORIO DE INSTRUCCIONES: DE TRANSFERENCIA DE DATOS (I)

Instrucción	Operación	Descripción
lw rd, imm_{12b}(rs1)	$rd \leftarrow \text{Mem}[rs1 + sExt(imm)]$	load word carga palabra
lh rd, imm_{12b}(rs1)	$rd \leftarrow sExt(\text{Mem}[rs1 + sExt(imm)]_{15:0})$	load half carga media palabra con signo
lhu rd, imm_{12b}(rs1)	$rd \leftarrow zExt(\text{Mem}[rs1 + sExt(imm)]_{15:0})$	load half unsigned carga media palabra sin signo
lb rd, imm_{12b}(rs1)	$rd \leftarrow sExt(\text{Mem}[rs1 + sExt(imm)]_{7:0})$	load byte carga byte con signo
lbu rd, imm_{12b}(rs1)	$rd \leftarrow zExt(\text{Mem}[rs1 + sExt(imm)]_{7:0})$	load half unsigned carga byte sin signo
sw rs2, imm_{12b}(rs1)	$\text{Mem}[rs1 + sExt(imm)] \leftarrow rs2$	store word almacena palabra
sh rs2, imm_{12b}(rs1)	$\text{Mem}[rs1 + sExt(imm)]_{15:0} \leftarrow rs2_{15:0}$	store whalf almacena media palabra
sb rs2, imm_{12b}(rs1)	$\text{Mem}[rs1 + sExt(imm)]_{7:0} \leftarrow rs2_{7:0}$	store byte almacena byte

- Existen instrucciones diferentes para copiar datos de 8, 16 o 32 bits
- Existen instrucciones diferentes para cargar datos extendiendo el signo o completando con ceros

REPERTORIO DE INSTRUCCIONES: DE TRANSFERENCIA DE DATOS (II)

Los datos en memoria están **alineados** y con orden **Little-Endian**

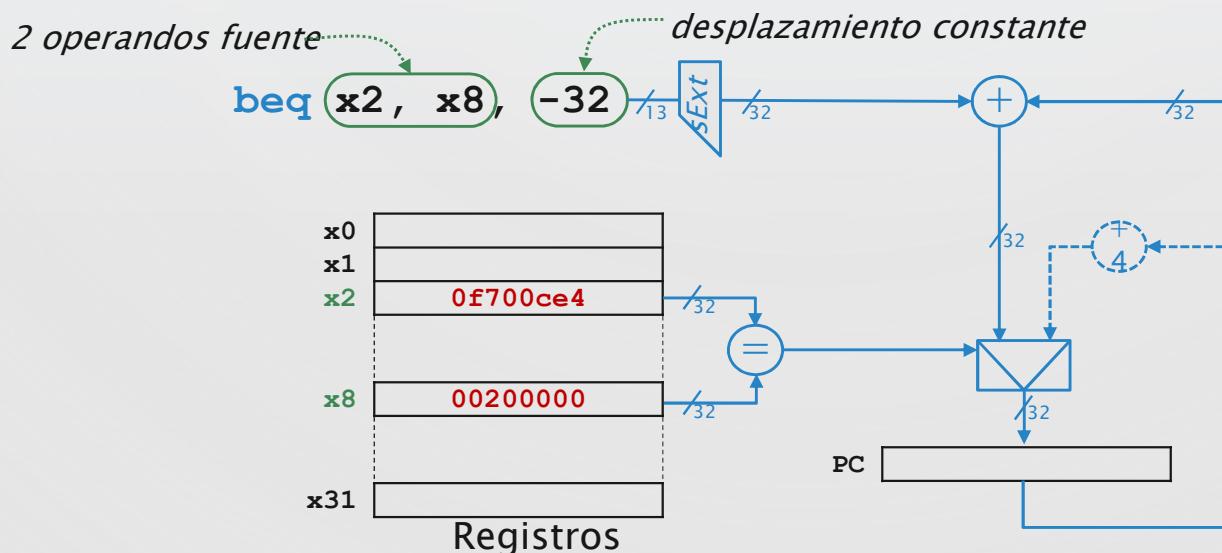


Instrucción	Operación	Descripción
lw x2, 0(x1)	$x2 \leftarrow \text{Mem}[x1 + sExt(0)]$	carga 8b257a02 en x2
lhu x2, 0(x1)	$x2 \leftarrow zExt(\text{Mem}[x1 + sExt(0)]_{15:0})$	carga 00007a02 en x2 ($7a02 =_2 +31234$)
lhu x2, 2(x1)	$x2 \leftarrow zExt(\text{Mem}[x1 + sExt(2)]_{15:0})$	carga 00008b25 en x2 ($8b25 =_2 +35621$)
lh x2, 0(x1)	$x2 \leftarrow sExt(\text{Mem}[x1 + sExt(0)]_{15:0})$	carga 00007a02 en x2 ($7a02 =_{C2} +31234$)
lh x2, 2(x1)	$x2 \leftarrow sExt(\text{Mem}[x1 + sExt(2)]_{15:0})$	carga ffff8b25 en x2 ($8b25 =_{C2} -29915$)
lbu x2, 3(x1)	$x2 \leftarrow zExt(\text{Mem}[x1 + sExt(3)]_{7:0})$	carga 0000008b en x2 ($8b =_2 +139$)
lb x2, 3(x1)	$x2 \leftarrow sExt(\text{Mem}[x1 + sExt(3)]_{7:0})$	carga ffffff8b en x2 ($8b =_{C2} -177$)
lh x2, 3(x1)	$x2 \leftarrow sExt(\text{Mem}[x1 + sExt(3)]_{15:0})$	<i>Al ejecutarse provoca error de alineamiento</i>

REPERTORIO DE INSTRUCCIONES: DE SALTO CONDICIONAL (I)

Permite romper la secuencia normal de ejecución saltando a una dirección cercana cuando se cumple cierta condición

- Compara dos operandos fuente que se encuentran en registros
- Utiliza direccionamiento relativo al PC para indicar la dirección con que actualizar el PC en caso de ser cierta la condición
 - Dicha dirección es la suma del contenido del PC y un desplazamiento corto
 - El desplazamiento es una constante de 13 bits en C2 cuyo signo se extiende a 32 bits



REPERTORIO DE INSTRUCCIONES: DE SALTO CONDICIONAL

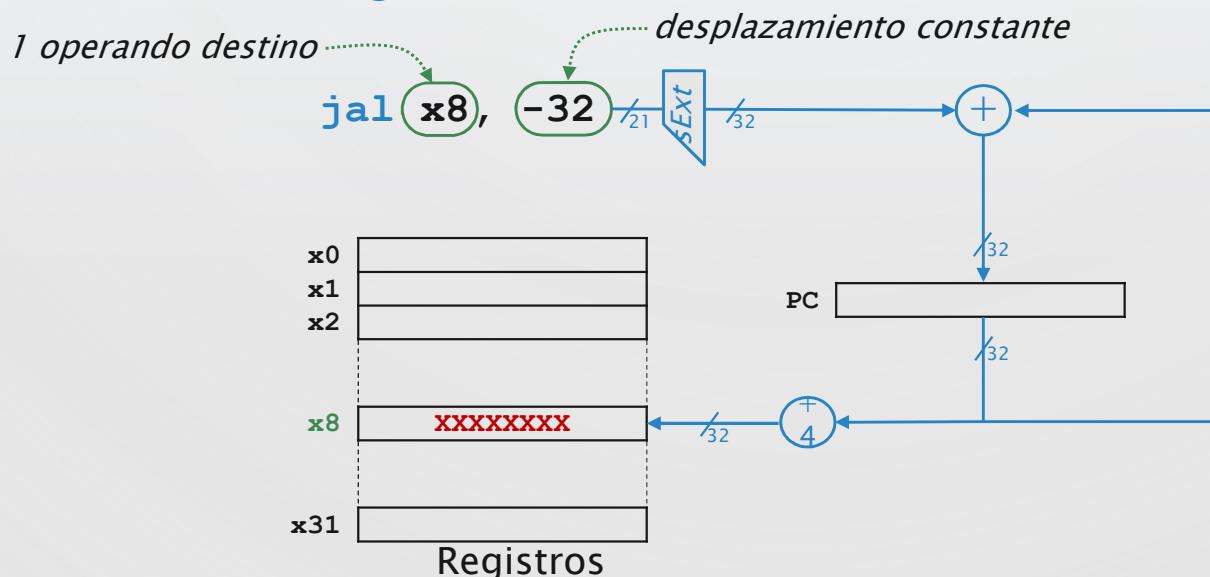
Instrucción	Operación	Descripción
<code>beq rs1, rs2, imm_{13b}</code>	<i>if(rs1 = rs2) then (PC ← PC + sExt(imm_{12:1} << 1))</i>	branch if equal salta si “igual que”
<code>bne rs1, rs2, imm_{13b}</code>	<i>if(rs1 ≠ rs2) then (PC ← PC + sExt(imm_{12:1} << 1))</i>	branch if not equal salta si “disintino que”
<code>blt rs1, rs2, imm_{13b}</code>	<i>if(rs1 <_S rs2) then (PC ← PC + sExt(imm_{12:1} << 1))</i>	branch if less than salta si “menor que” con signo
<code>bge rs1, rs2, imm_{13b}</code>	<i>if(rs1 ≥_S rs2) then (PC ← PC + sExt(imm_{12:1} << 1))</i>	branch if greater than or equal salta si “mayor o igual que” con signo
<code>bltu rs1, rs2, imm_{13b}</code>	<i>if(rs1 <_U rs2) then (PC ← PC + sExt(imm_{12:1} << 1))</i>	branch if less than unsigned salta si “menor que” sin signo
<code>bgeu rs1, rs2, imm_{13b}</code>	<i>if(rs1 ≥_U rs2) then (PC ← PC + sExt(imm_{12:1} << 1))</i>	branch if greater than or equal unsigned salta si “mayor o igual que” sin signo

- Las instrucciones de salto condicional se usan para implementar en ensamblador **estructuras de control** tipo *if, while, for, ...*
- Existen instrucciones diferentes para interpretar los operandos fuente que se comparan como **datos con y sin signo**

REPERTORIO DE INSTRUCCIONES: DE SALTO A FUNCIÓN: JAL

Permite romper la secuencia normal de ejecución saltando a una dirección, pero guardando la dirección de retorno

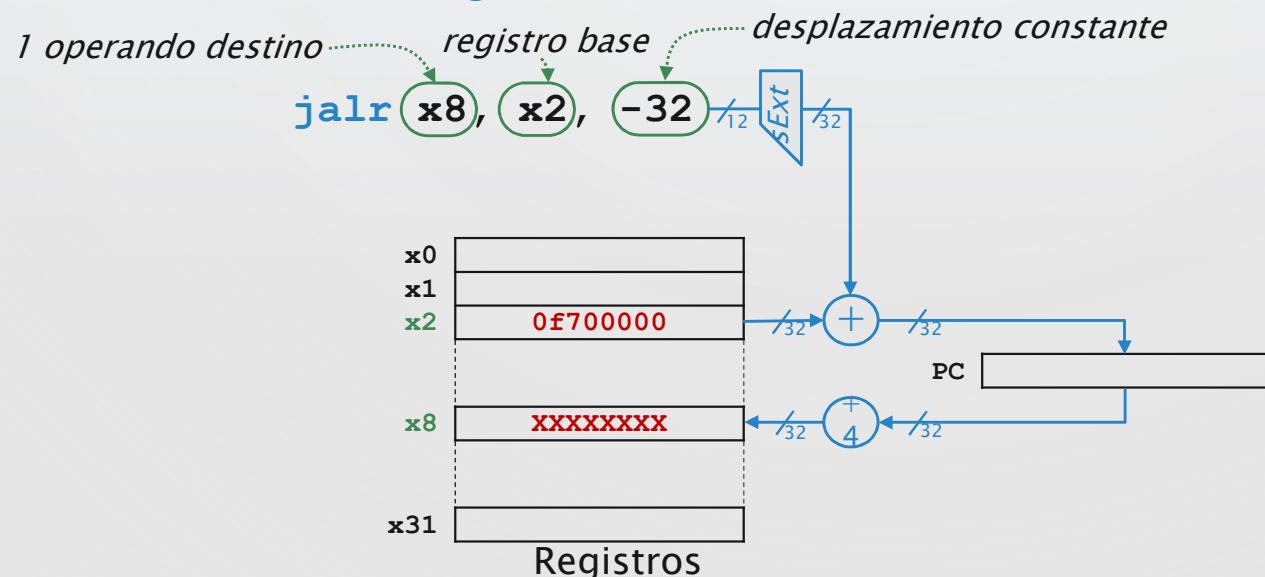
- Utiliza **direcciónamiento relativo al PC** para indicar la dirección con que actualizar el PC
 - Dicha dirección es la suma del **contenido del PC** y un **desplazamiento largo**
 - El **desplazamiento** es una constante de 21 bits en C2 cuyo signo se extiende a 32 bits
- La **dirección de la siguiente instrucción (retorno)** se almacena en un **registro**



REPERTORIO DE INSTRUCCIONES: DE SALTO A FUNCIÓN: JALR

Permite romper la secuencia normal de ejecución saltando a una dirección, pero guardando la dirección de retorno

- Utiliza **direcccionamiento relativo a base** para indicar la dirección con que actualizar el PC
 - Dicha dirección es la suma del **contenido del registro** y un desplazamiento corto
 - El **desplazamiento** es una constante de 12 bits en C2 cuyo signo se extiende a 32 bits
- La **dirección de la siguiente instrucción (retorno)** se almacena en un **registro**



REPERTORIO DE INSTRUCCIONES: DE SALTO A FUNCIÓN (II)

Instrucción	Operación	Descripción
<code>jal rd, imm_{21b}</code>	$PC \leftarrow PC + sExt(imm_{20:1} << 1),$ $rd \leftarrow PC+4$	jump and link salto a función con dirección relativa a PC
<code>jalr rd, rs1, imm_{12b}</code>	$PC \leftarrow rs1 + sExt(imm),$ $rd \leftarrow PC+4$	jump and link register salto a función con dirección relativa a registro base

En el repertorio del RISC-V **no existen** instrucciones de **retorno desde función**, ni de **salto incondicional** a una instrucción, pero pueden usarse éstas para hacerlo:

- Suponiendo que la instrucción de retorno está almacenada en el registro **xn**, el retorno se hace con:

`jalr x0, xn, 0`

- El salto incondicional (relativo a PC) a la dirección que ocupa cierta instrucción se hace con:

`jal x0, imm21b`

PROGRAMACIÓN ENSAMBLADOR

- Los computadores ejecutan código máquina pero los programadores desarrollan software en lenguajes de alto nivel
- El lenguaje ensamblador es el **punto intermedio entre HW y SW**:
 - Los programas de alto nivel se compilan a ensamblador
 - Los programas en lenguaje ensamblador se ensamblan a código máquina
- Un programa ensamblador está formado por:
 - Instrucciones ensamblador
 - Etiquetas
 - Directivas
 - Pseudo-instrucciones
 - Comentarios, ...

PROGRAMACIÓN ENSAMBLADOR

ELEMENTOS DE UN PROGRAMA

Un programa en lenguaje ensamblador:

- Está compuesto por una **secuencia de instrucciones** que se ubicarán en memoria en el mismo orden para **ejecutarse en serie**
 - Normalmente utiliza alias para referirse a los registros
- No indica explícitamente la dirección de memoria de cada instrucción o dato
 - Pero instrucciones consecutivas ocuparán direcciones consecutivas
- Se necesita **saltar a una cierta instrucción**, permite definir una **etiqueta** para referirse simbólicamente a su dirección
 - Las etiquetas liberan al programador del cálculo de desplazamientos relativos

C++

```
...  
if (a != b)  
    a = a + 1;  
c = c - b;  
...
```

a → t0, b → t1, c → t2

ASM

```
dir  
dir+4  
dir+8  
...  
beq t0, t1, L1  
addi t0, t0, 1  
L1:  
    sub t2, t2, t1  
...
```

...
beq x5, x6, 8
addi x5, x5, 1
sub x7, x7, x6
...

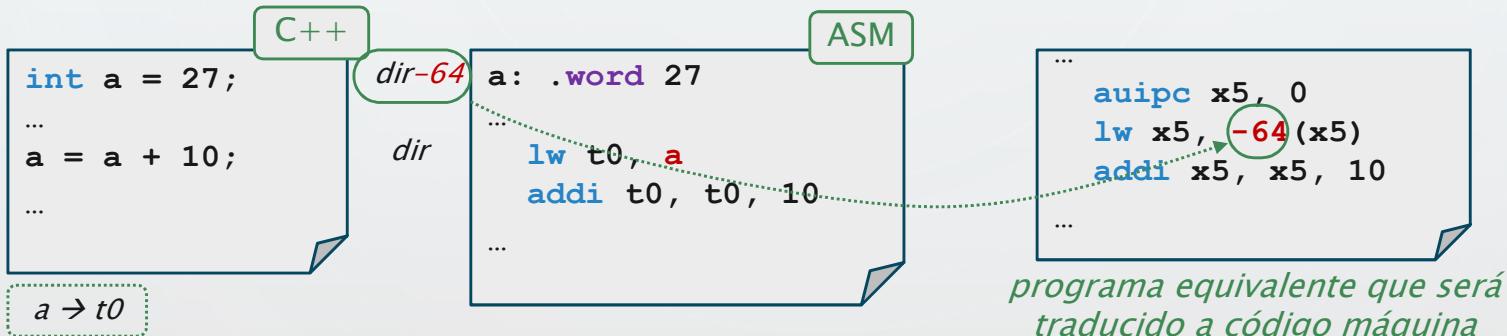
programa equivalente que será traducido a código máquina

PROGRAMACIÓN ENSAMBLADOR

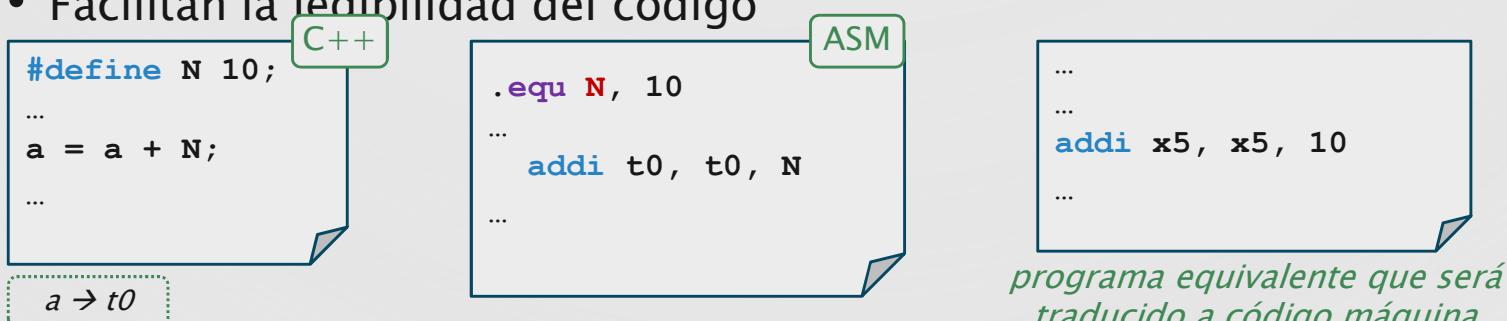
ELEMENTOS DE UN PROGRAMA

Un programa en lenguaje ensamblador se pueden definir:

- **Etiquetas** para referirse simbólicamente a la dirección que ocupa un dato
 - Liberan al programador de la gestión de direcciones absolutas de 32 bits



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



PROGRAMACIÓN ENSAMBLADOR

ELEMENTOS DE UN PROGRAMA

Además de instrucciones, un programa ensamblador puede contener:

- **Pseudo-instrucciones**: son alias de ciertas instrucciones

- Facilitan al programador el uso de instrucciones recurrentes
programa equivalente que será traducido a código máquina



- **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.

PROGRAMACIÓN ENSAMBLADOR

ELEMENTOS DE UN PROGRAMA

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 una 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 implícitos
 - En lugar de una instrucción, puede haber una pseudo-instrucción
- **Directiva:** **indicación auxiliar** utilizada durante el ensamblado
 - Toda directiva comienza por un punto
- **Comentario:** **texto libre** usado por el programador
 - Comienzan con # y pueden estar al final de una línea o ocuparla completamente

PROGRAMACIÓN ENSAMBLADOR

SECCIONES

- Un programa 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**

PROGRAMACIÓN 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 con valor inicial
<code>.bss</code>	Declara el comienzo de la sección de variables globales sin valor inicial
<code>.word w₁, ... w_n</code>	Reserva espacio en memoria para n palabras inicializadas a w_1, \dots, w_n
<code>.half h₁, ... h_n</code>	Reserva espacio en memoria para n medias palabras inicializadas a h_1, \dots, h_n
<code>.byte b₁, ... b_n</code>	Reserva espacio en memoria para n bytes inicializadas a b_1, \dots, b_n
<code>.zero n</code>	Reserva espacio en memoria para n bytes inicializados a 0
<code>.space n</code>	Reserva espacio en memoria para n bytes sin inicializar
<code>.string "str"</code>	Reserva espacio en memoria inicializado con la cadena "str"
<code>.align n</code>	Alinea los datos/instrucciones a direcciones múltiplo de 2^n
<code>.equ sym, val</code>	Define una constante simbólica llamada <i>sym</i> de valor <i>val</i>
<code>.global sym</code>	Hace visible la etiqueta <i>sym</i> fuera del archivo que la contiene (global)
<code>.extern sym</code>	Indica que un símbolo está definido en otro archivo del proyecto
<code>.end</code>	Declara el final del programa ensamblador

PROGRAMACIÓN ENSAMBLADOR

EJEMPLO

DATOS CON
VALOR INICIAL

```
.data
A: .word 5
B: .word 8
```

Directivas

DATOS SIN
VALOR INICIAL

```
.bss
M: .space 4
```

Etiquetas

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
.end
```

Comentario

Instrucción

Pseudo-instrucción

PROGRAMACIÓN ENSAMBLADOR

EJEMPLO

DATOS CON VALOR INICIAL

```
.data
A: .word 5
B: .word 8
```

Indica el inicio de la sección de datos inicializados
Datos inicializados de tamaño palabra

DATOS SIN VALOR INICIAL

```
.bss
M: .space 4
```

Indica el inicio de la sección de datos no inicializados
Datos no inicializados 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
.end
```

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

Indica el final del programa ensamblador

PROGRAMACIÓN ENSAMBLADOR

EJEMPLO

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)      # A<B
    j   fin
L1:   sw  t0, 0(t2)      # A>=B
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 M el valor de t1 (B)
Salta a la última instrucción del programa
Almacena en la dirección M el valor de t0 (A)
Indefinidamente ejecuta esta instrucción

PROGRAMACIÓN ENSAMBLADOR

VARIABLES Y CONSTANTES

- En ensamblador **no existe variables** como tales
 - Existen **datos** que **residen en memoria o en registro**
 - Para **operar** con ellos **siempre deben estar en registros** porque en el ensamblador de RISC-V no existen instrucciones con operandos en memoria
- No hay distinción entre **variables y constantes** (si cambia su valor es variable)
- Las constantes pueden residir en la propia instrucción (**operandos inmediatos**)
 - Los valores constantes se pueden expresar indistintamente en:
 - **Decimal**, tal cual: 109
 - **Hexadecimal**, anteponiendo **0x** a la secuencia de dígitos: 0x6d
 - **Binario**, anteponiendo **0b** a la secuencia de dígitos: 0b1101101

PROGRAMACIÓN ENSAMBLADOR

VARIABLES Y CONSTANTES

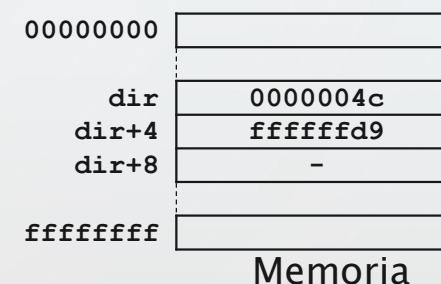
- En el caso de **variables/constantes globales**, se **usan etiquetas** para evitar el uso de direcciones explícitas en el código ensamblador
 - Estas etiquetas juegan en ensamblador **el papel del nombre de la variable**

C++

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

ASM

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



- Como el **número de registros es limitado** los datos mayoritariamente residen en memoria
- Como el **acceso a 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

PROGRAMACIÓN ENSAMBLADOR

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++

```
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  
...
```

PROGRAMACIÓN ENSAMBLADOR

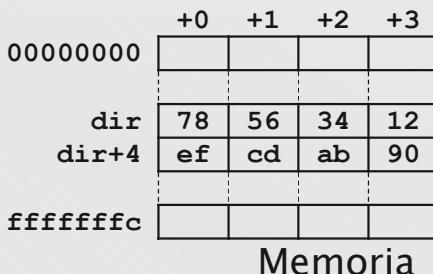
VARIABLES Y CONSTANTES

Al igual que las instrucciones, los datos **se ubican en memoria en el mismo orden** en que aparecen en el programa 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`

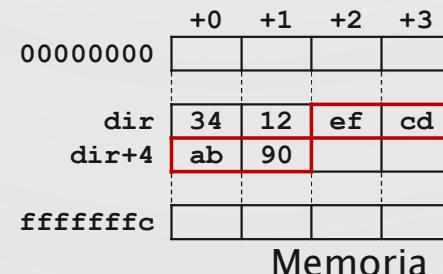
INCORRECTO ASM

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



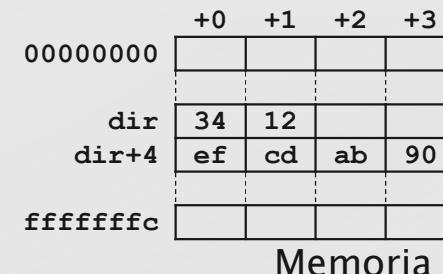
INCORRECTO ASM

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



CORRECTO ASM

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



PROGRAMACIÓN ENSAMBLADOR

TIPOS

En ensamblador las variables **no 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 de dato y las instrucciones que usa para operar con él (carga, almacenamiento, comparación ...)

La **equivalencia entre tipos C/C++ y anchuras en ensamblador es:**

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

PROGRAMACIÓN ENSAMBLADOR

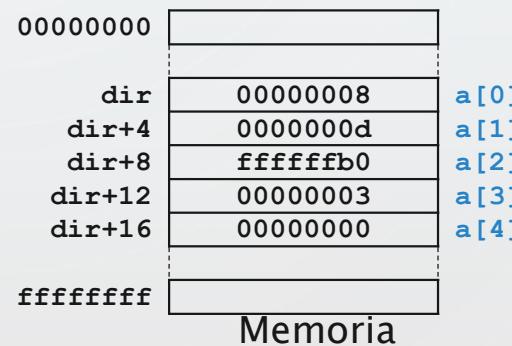
ARRAYS

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

C/C++
`int a[5] = {8, 13, -80, 3, 0};
...`

ASM
`a: .word 8, 13, -80, 3, 0
...`



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 del primer elemento
- El **desplazamiento** en bytes se calcula:
 - desplazamiento (bytes) = índice × tamaño de los datos (bytes)

PROGRAMACIÓN ENSAMBLADOR ARRAYS

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;
...
```

ASM

carga la dirección
base del array
carga a[1]
carga a[2]
almacena a[0]

```
a: .space 20
...
la t0, a
lw t1, 4(t0)
lw t2, 8(t0)
add t1, t1, t2
sw t1, 0(t0)
...
```

$a \equiv \&a[0] \rightarrow t0$
 $i \rightarrow t1$
 $a[i] \rightarrow t2$

ASM

carga la dirección base del array
calcula el desplazamiento $i*4$
suma base y desplazamiento
carga a[i]
almacena a[i]

```
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)
...
```

PROGRAMACIÓN ENSAMBLADOR

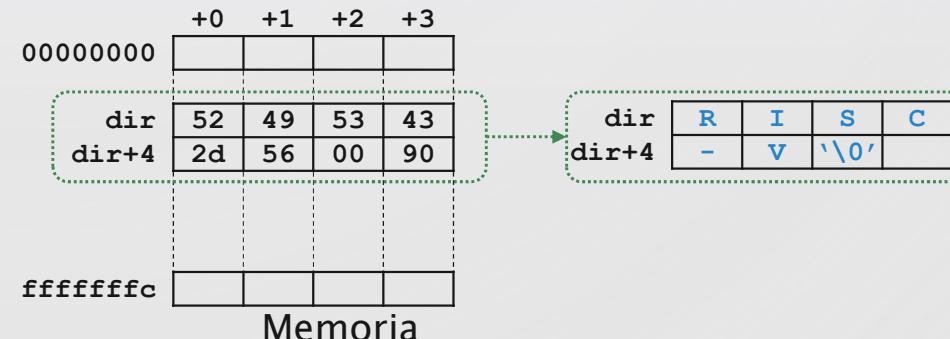
ARRAYS

Las cadenas de caracteres son un caso especial de arrays

- Almacenan 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

C/C++

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

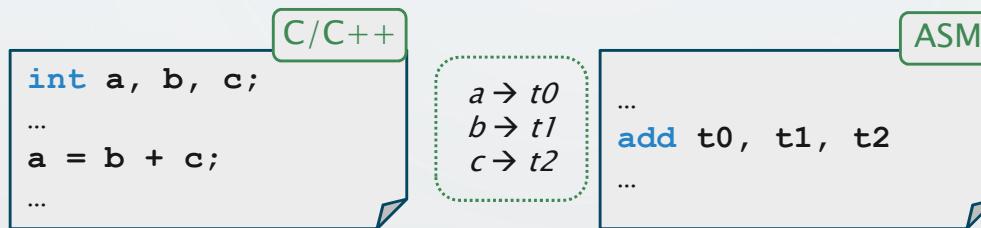


PROGRAMACIÓN ENSAMBLADOR

EXPRESIONES

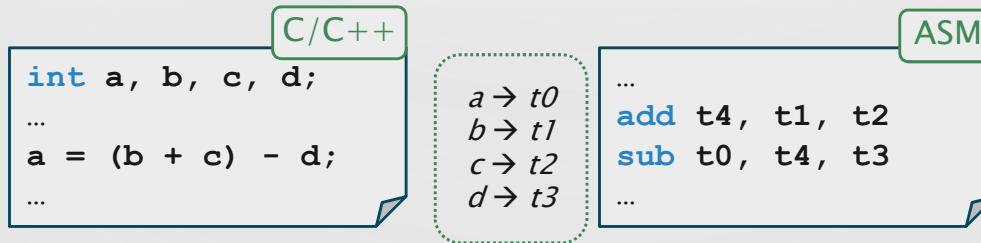
Las expresiones simples en C/C++ requieren una única instrucción

- Usando registros en donde previamente se han cargado los datos



Las expresiones compuestas requieren más de una instrucción

- Usando registros adicionales para almacenar los resultados intermedios

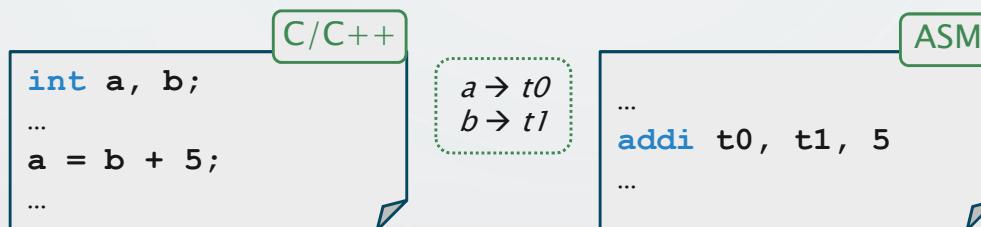


PROGRAMACIÓN ENSAMBLADOR

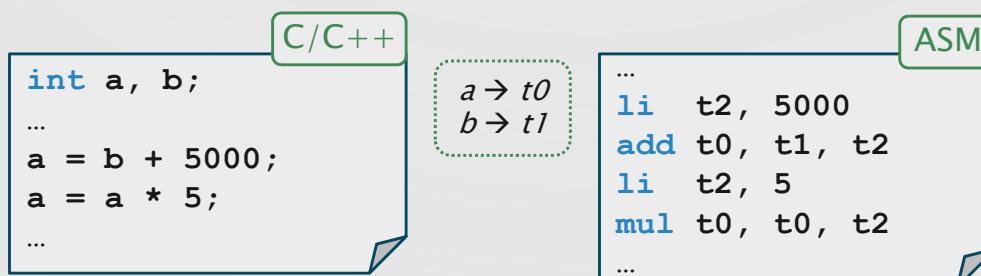
EXPRESIONES

Las **constantes explícitas** pueden aparecer de manera simbólica:

- Si la **constante es corta** (≤ 12 bits [-2048, +2047]) puede usarse directamente como operando inmediato



- Deben cargarse previamente en un **registro**:
 - Las **constantes largas** (> 12 bits)
 - Constantes que se usan en instrucciones que **no permiten operandos inmediatos**

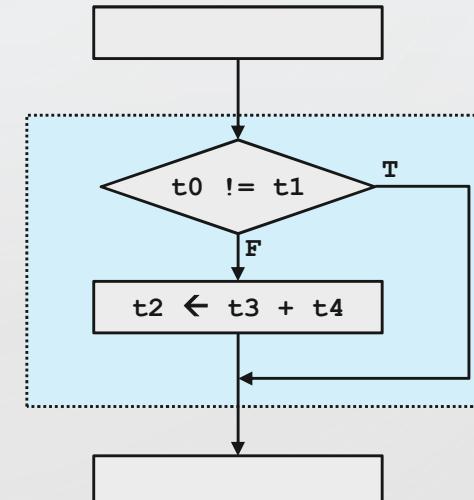
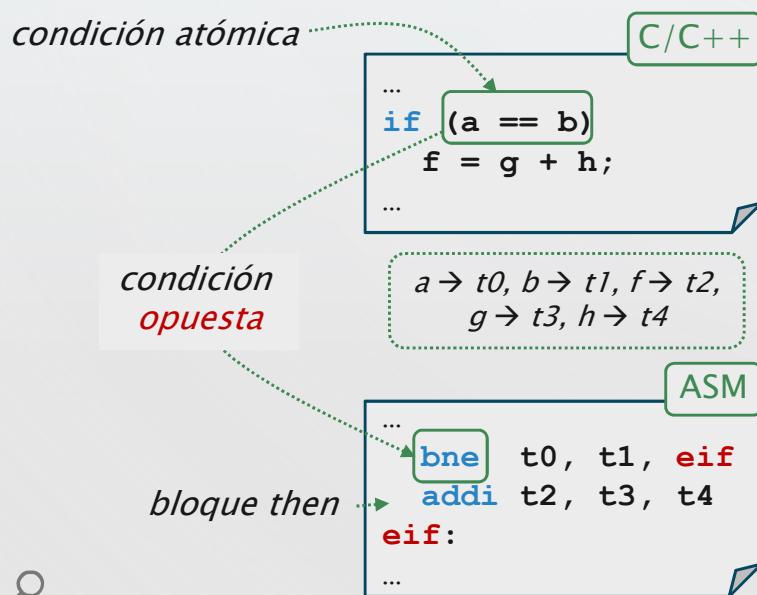


PROGRAMACIÓN ENSAMBLADOR

ESTRUCTURA IF-THEN

Se utiliza un **salto condicional** para chequear la condición **opuesta** y saltar bloque THEN

Si la **condición es compuesta**, se chequean de **una en una** las condiciones

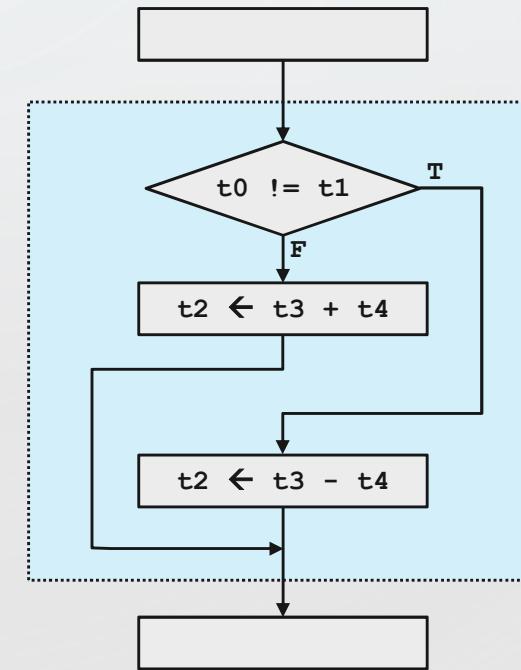
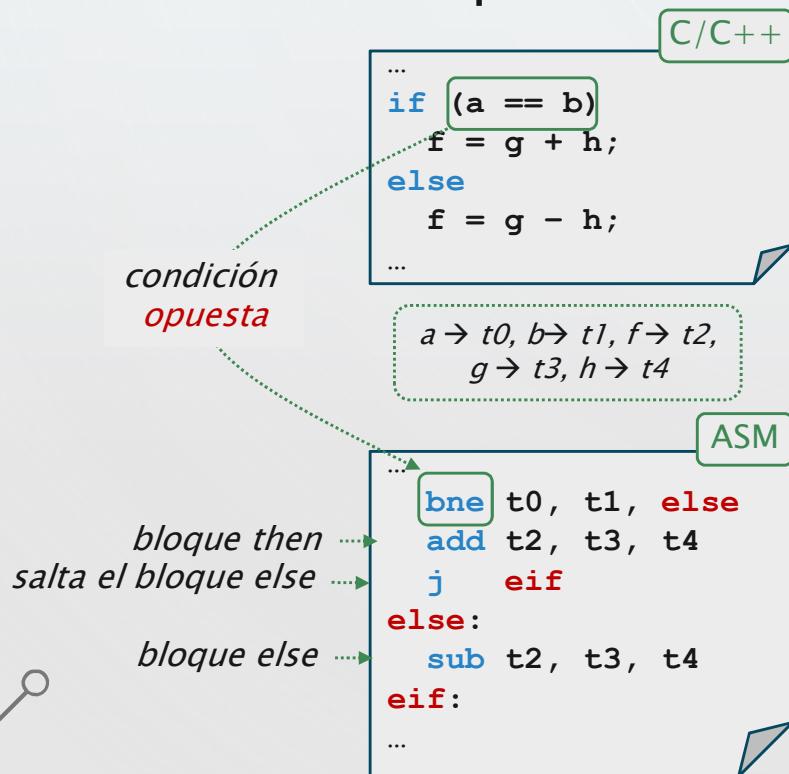


PROGRAMACIÓN ENSAMBLADOR

ESTRUCTURA IF-THEN-ELSE

Se utiliza un **salto condicional** para chequear la condición **opuesta** y saltar a bloque ELSE

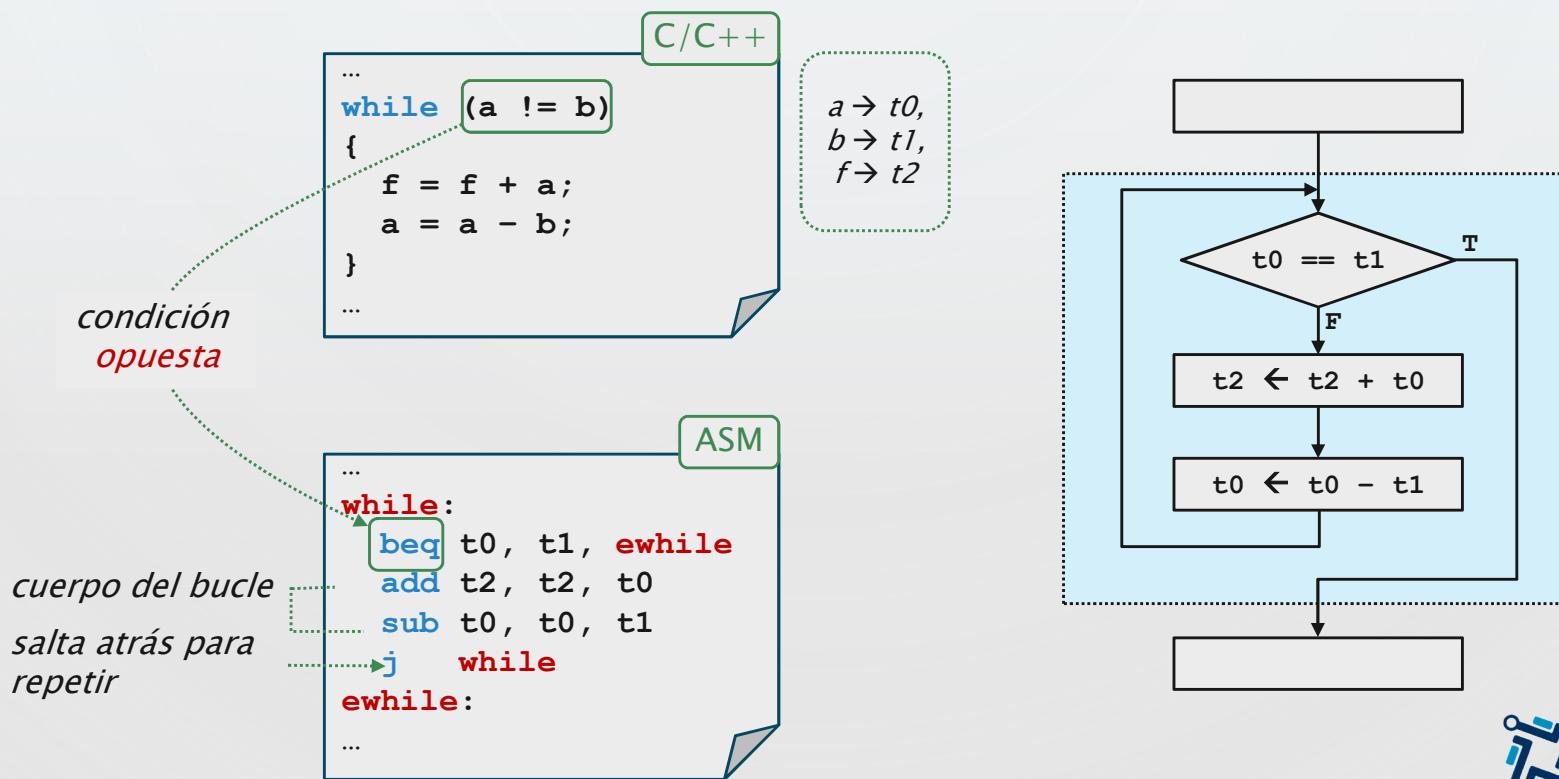
Al final del bloque THEN **salto incondicional** para saltar bloque ELSE



PROGRAMACIÓN ENSAMBLADOR

ESTRUCTURA WHILE-DO

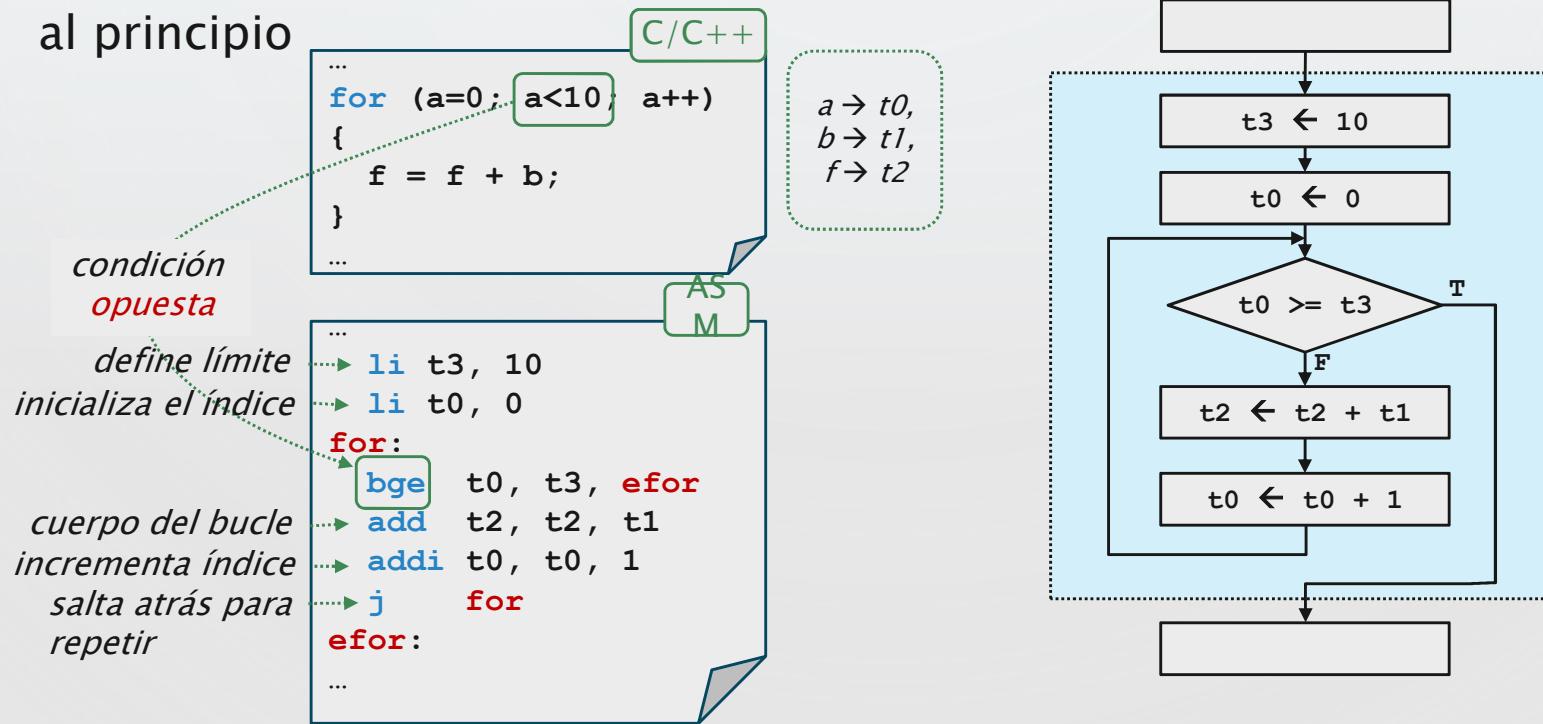
- Se utiliza un **salto condicional** para chequear la condición **opuesta** y saltar fuera
- Al final del bloque del cuerpo del bucle **salto incondicional** para saltar al principio



PROGRAMACIÓN ENSAMBLADOR

ESTRUCTURA FOR

- Se inicializa variable índice del bucle fuera
- Se utiliza un **salto condicional** para chequear la condición **opuesta** y saltar fuera
- Al final del cuerpo del bucle se modifica índice y **salto incondicional** para saltar al principio



PROGRAMACIÓN ENSAMBLADOR

EJEMPLO BUCLE QUE MODIFICA ARRAY

Bucle que multiplica por 2 todos los elementos de un array

```
C/C++  
int a[5] = {8, 13, -80, 3, 0};  
...  
for (i=0; i<5; i++)  
{  
    a[i] = 2*a[i];  
}  
...  
  
&a[0] → t0, i → t1
```

```
ASM  
.data  
a: .word 8, 13, -80, 3, 0  
...  
.text  
...  
    la t0, a           carga dirección de array a en t0  
    li t2, 5            define límite del bucle en t2  
    li t1, 0            inicializa i (t1) a 0  
for:  
    bge t1, t2, efor    si i (t1) es mayor o igual que (t2), salir  
    slli t3, t1, 2       calcula desplazamiento i (t1) *4 en t3  
    add t3, t3, t0       suma el desp. (t3) a la dirección de a (t0)  
    lw t4, 0(t3)         carga en t4 el dato de la dir. calculada (t3)  
    slli t4, t4, 1       multiplica por 2 el dato leído  
    sw t4, 0(t3)         almacena el nuevo valor en la misma dir (t3)  
    addi t1, t1, 1       incrementa i (t1)  
    j for               salta atrás para repetir  
efor:  
...
```

carga dirección de array a en t0
define límite del bucle en t2
inicializa i (t1) a 0
si i (t1) es mayor o igual que (t2), salir
calcula desplazamiento i (t1) *4 en t3
suma el desp. (t3) a la dirección de a (t0)
carga en t4 el dato de la dir. calculada (t3)
multiplica por 2 el dato leído
almacena el nuevo valor en la misma dir (t3)
incrementa i (t1)
salta atrás para repetir

PROGRAMACIÓN ENSAMBLADOR

EJEMPLO BUCLE QUE MODIFICA ARRAY

Bucle que multiplica por 2 todos los elementos de un array

ASM

```
.data
a: .word 8, 13, -80, 3, 0
...
.text
...
    la t0, a
    li t2, 5
    li t1, 0
for:
    bge t1, t2, efor
    slli t3, t1, 2
    add t3, t3, t0
    lw t4, 0(t0)
    slli t4, t4, 1
    sw t4, 0(t3)
    addi t1, t1, 1
    j for
efor:
...
```

ASM

```
.data
a: .word 8, 13, -80, 3, 0
...
.text
...
    la t0, a
    li t1, 5
for:
    beqz t1, efor
    lw t4, 0(t0)
    slli t4, t4, 1
    sw t4, 0(t0)
    addi t0, t0, 4
    addi t1, t1, -1
    j for
efor:
...
```

carga dirección de array a en t0
inicializa contador del bucle con el
número de elementos en t1
si contador es 0, salir
carga en t4 el dato actual (t0)
multiplica por 2 el dato leído
almacena el nuevo valor en la misma dir. (t0)
calcula (+4) siguiente posición a leer (t0)
decrementa número de elementos restantes
salta atrás para repetir

PROGRAMACIÓN ENSAMBLADOR

FUNCIONES

- 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 **invocante (*caller*)** debe **pasar los argumentos** y **saltar al comienzo** de la función invocada
 - La **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 LLAMADAS A FUNCIONES** estándar que debe respetarse para garantizar la interoperabilidad

PROGRAMACIÓN ENSAMBLADOR

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 a la 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
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	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 de retorno 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

PROGRAMACIÓN ENSAMBLADOR

FUNCIONES: LLAMADA

Por **CONVENIO**, la función invocante usa:

Los registros: **a0...a7** para pasar hasta 8 **argumentos** a la invocada

La pseudo-instrucción **jal** para llamar (saltar) a la invocada:

Esta pseudo-instrucción de traduce en una instrucción **jal/jalr** y el uso del registro **ra** para guardar la dirección de retorno

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 inc
sw a0, 0(t0)
...
inc:
    add a0, a0, 1
    ret
...
```

carga en a0 el argumento
salta a la función
invocada guardando en
ra la dirección de retorno

PROGRAMACIÓN ENSAMBLADOR

FUNCIONES: RETORNO

Por **CONVENIO**, la función invocada usa:

El registro **a0** para **devolver el resultado** a la invocante. Si el dato fuera de 64 bits se usaría también **a1** para la parte alta.

La pseudo-instrucción **ret** para retornar a la invocante. Salta a la dirección almacenada en **ra**.

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 inc
sw a0, 0(t0)
...
inc:
add a0, a0, 1
ret
...
```

la función invocante almacena
el resultado devuelto en a0

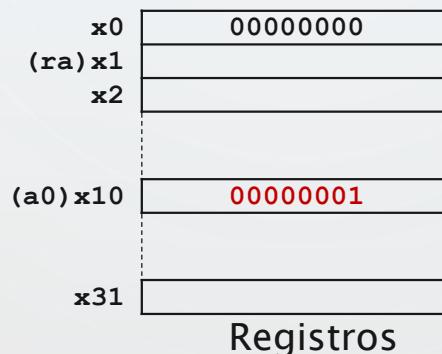
guarda en a0 el resultado
retorno a la función invocante

PROGRAMACIÓN ENSAMBLADOR

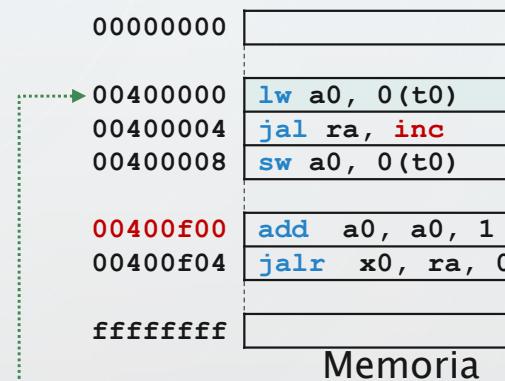
FUNCIONES: LLAMADA Y RETORNO

ASM

```
a: .word 1
...
    la    t0, a
    lw    a0, 0(t0)
    jal   inc
    sw    a0, 0(t0)
...
inc:
    add  a0, a0, 1
    ret
...
```



PC 00400000

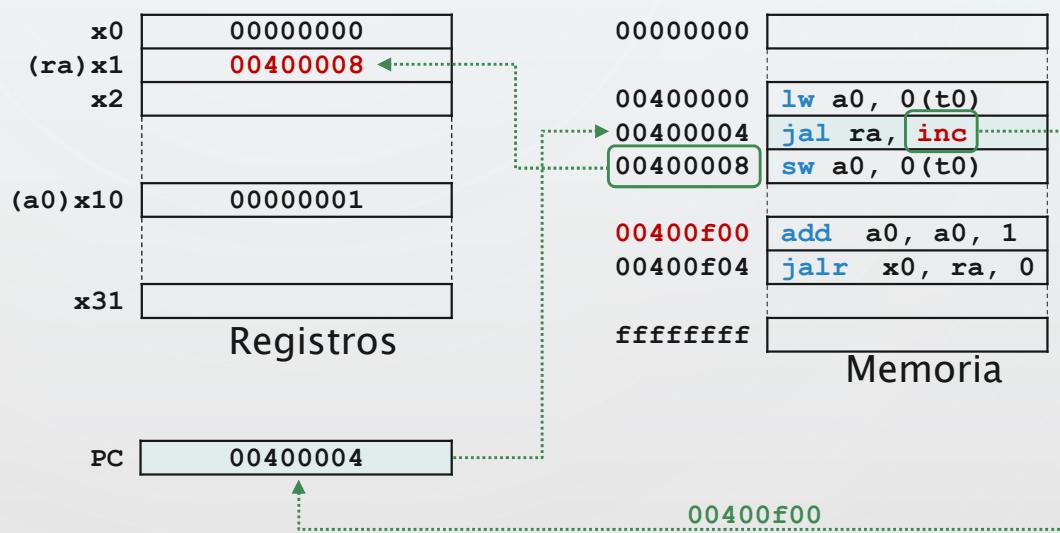


PROGRAMACIÓN ENSAMBLADOR

FUNCIONES: LLAMADA Y RETORNO

ASM

```
a: .word 1
...
    la    t0, a
    lw    a0, 0(t0)
    jal   inc
    sw    a0, 0(t0)
...
inc:
    add  a0, a0, 1
    ret
...
```



PROGRAMACIÓN ENSAMBLADOR

FUNCIONES: LLAMADA Y RETORNO

ASM

```
a: .word 1
...
    la t0, a
    lw a0, 0(t0)
    jal inc
    sw a0, 0(t0)
...
inc:
    add a0, a0, 1
    ret
...
```

x0	00000000
(ra)x1	00400008
x2	
(a0)x10	00000002
x31	

Registros

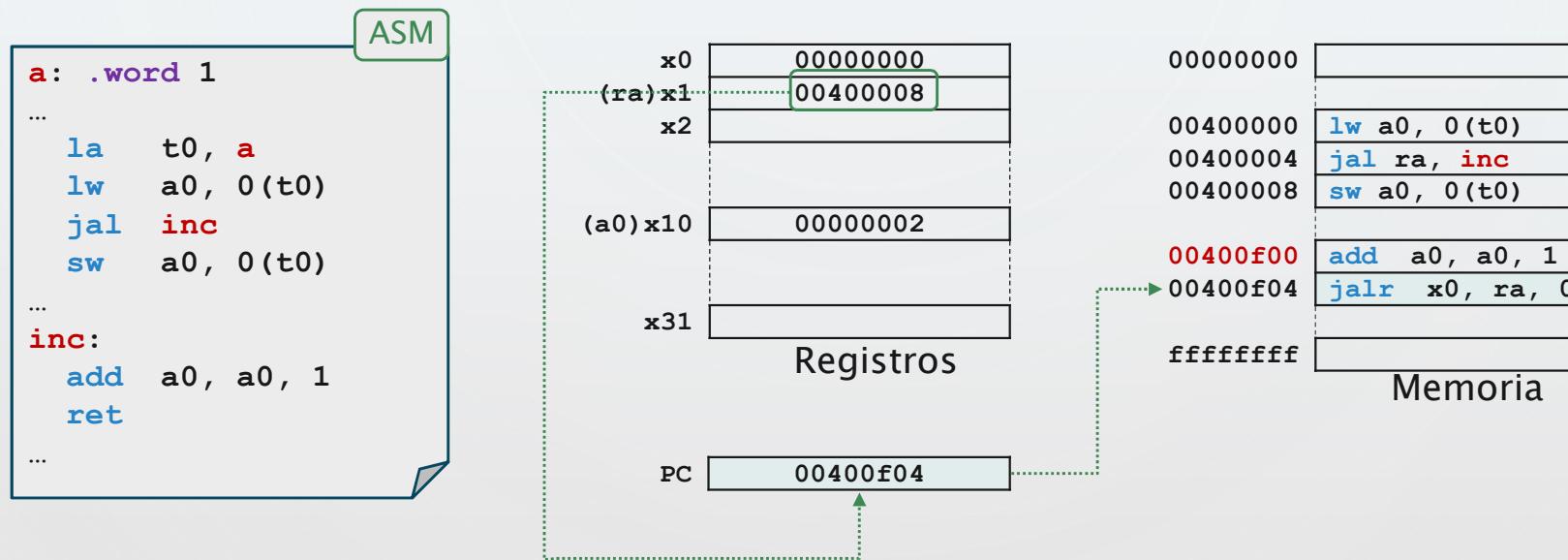
00000000	
00400000	
00400004	
00400008	
00400f00	lw a0, 0(t0)
00400f04	jal ra, inc
00400f08	sw a0, 0(t0)
add a0, a0, 1	
jalr x0, ra, 0	
ffffffffff	

Memoria

PC 00400f00

PROGRAMACIÓN ENSAMBLADOR

FUNCIONES: LLAMADA Y RETORNO



PROGRAMACIÓN ENSAMBLADOR

FUNCIONES: LLAMADA Y RETORNO

ASM

```
a: .word 1
...
    la t0, a
    lw a0, 0(t0)
    jal inc
    sw a0, 0(t0)
...
inc:
    add a0, a0, 1
    ret
...
```

x0	00000000
(ra)x1	00400008
x2	
(a0)x10	00000002
x31	

Registros

00000000	
00400000	
00400004	
00400008	
00400f00	lw a0, 0(t0)
00400f04	jal ra, inc
00400f08	sw a0, 0(t0)
add a0, a0, 1	
jalr x0, ra, 0	
ffffffffff	

Memoria

PC 00400008

PROGRAMACIÓN ENSAMBLADOR

FUNCIONES: REGISTROS TEMPORALES VS PRESERVADOS

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)
jal 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

PROGRAMACIÓN ENSAMBLADOR

FUNCIONES: REGISTROS TEMPORALES VS PRESERVADOS

- 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 en 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/sp**
- **Registro temporal (*caller-saved*)**: aquel cuyo **contenido puede alterarse libremente** al ejecutar una función
 - Su **valor tras retornar** de una 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**

PROGRAMACIÓN ENSAMBLADOR

FUNCIONES: REGISTROS TEMPORALES VS PRESERVADOS

Según este convenio, lo correcto sería que la función invocante use registros preservados cuando quiere 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.

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

C/C++

```
CORRECTO  
a: .space 4  
...  
la s0, a  
lw a0, 0(s0)  
jal inc  
sw a0, 0(s0)  
...  
inc:  
addi t0, a0, 1  
mv a0, t0  
ret
```

ASM

la función invocante usa *s0* para almacenar temporalmente la dirección de *a*
si *inc* está correctamente programada, *s0* seguirá conteniendo la dirección de *a*
la función invocada no usa *s0* por lo que su valor no cambiará durante su ejecución

PROGRAMACIÓN ENSAMBLADOR

FUNCIONES: REGISTROS TEMPORALES VS PRESERVADOS

Resumen del uso de los registros por **CONVENIO**



PROGRAMACIÓN ENSAMBLADOR

FUNCIONES: GESTIÓN DE PILA

- 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 cierto orden se recuperan desapilándolos en orden inverso
- Por **CONVENIO**, en ensamblador 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**

PROGRAMACIÓN ENSAMBLADOR

FUNCIONES: GESTIÓN DE PILA

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 (*store*)

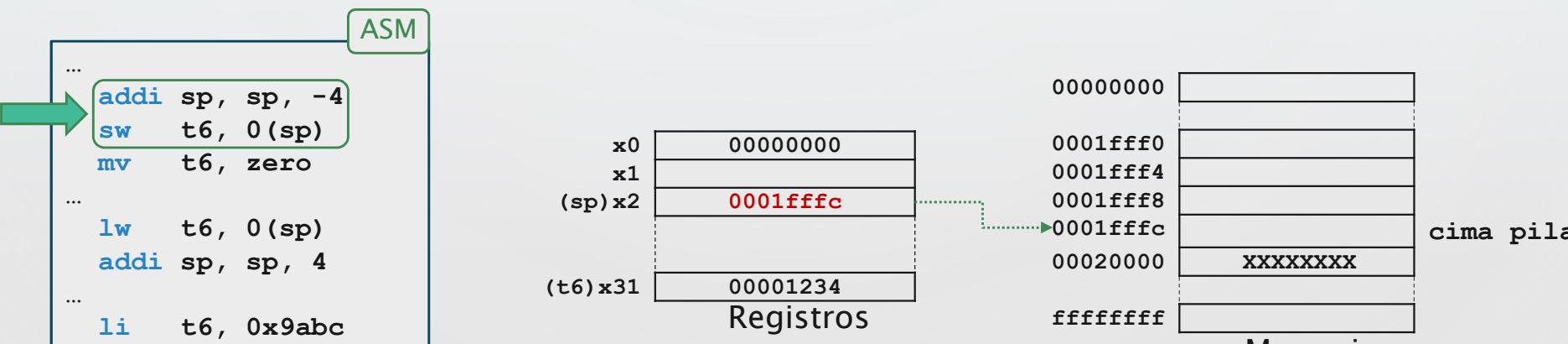


PROGRAMACIÓN ENSAMBLADOR

FUNCIONES: GESTIÓN DE PILA

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 (*store*)

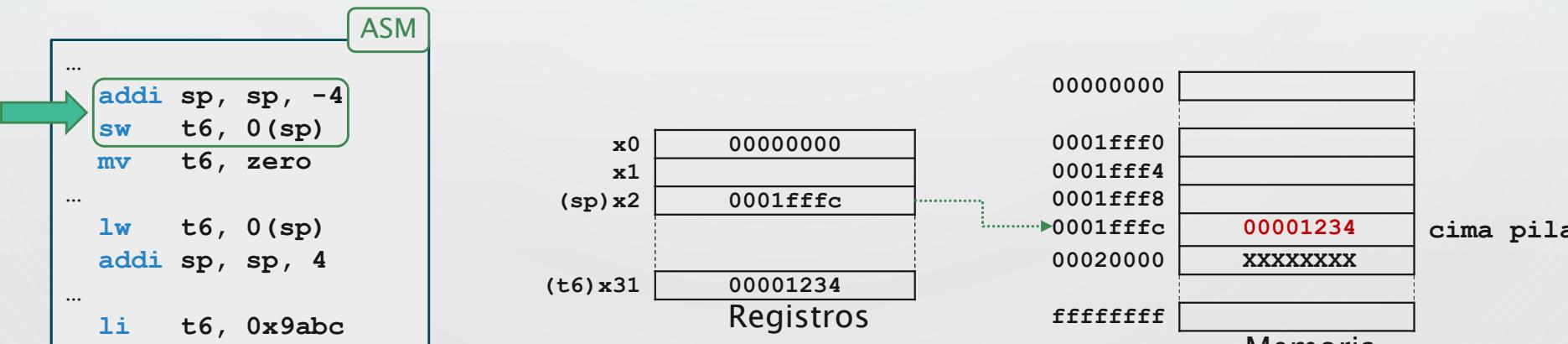


PROGRAMACIÓN ENSAMBLADOR

FUNCIONES: GESTIÓN DE PILA

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 (*store*)

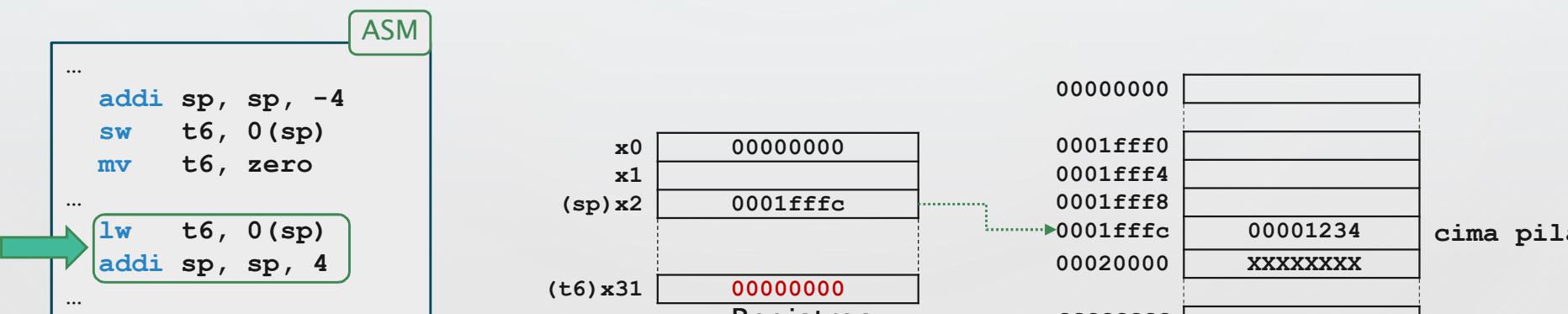


PROGRAMACIÓN ENSAMBLADOR

FUNCIONES: GESTIÓN DE PILA

Desapilar un dato supone:

- Cargar el dato ubicado en la **cima** de la pila (*load*)
- Incrementar **sp** el número de bytes que ocupe el dato, normalmente 4

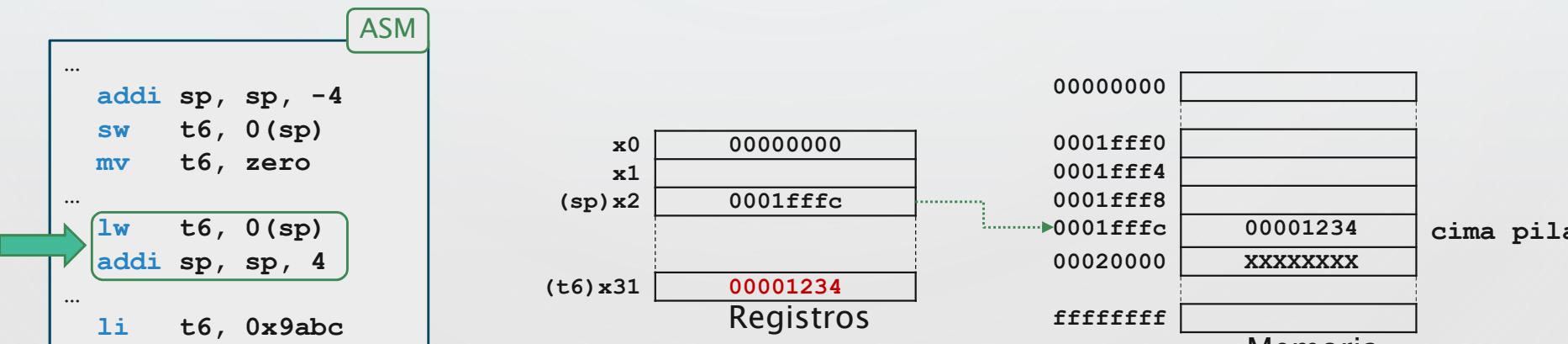


PROGRAMACIÓN ENSAMBLADOR

FUNCIONES: GESTIÓN DE PILA

Desapilar un dato supone:

- Cargar el dato ubicado en la **cima** de la pila (*load*)
- Incrementar **sp** el número de bytes que ocupe el dato, normalmente 4

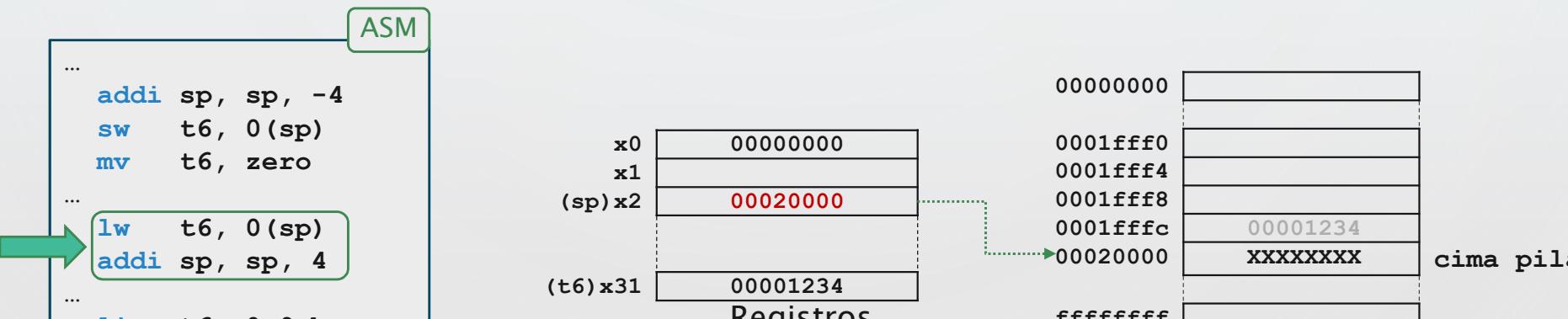


PROGRAMACIÓN ENSAMBLADOR

FUNCIONES: GESTIÓN DE PILA

Desapilar un dato supone:

- Cargar el dato ubicado en la **cima** de la pila (*load*)
- Incrementar **sp** el número de bytes que ocupe el dato, normalmente 4



PROGRAMACIÓN ENSAMBLADOR

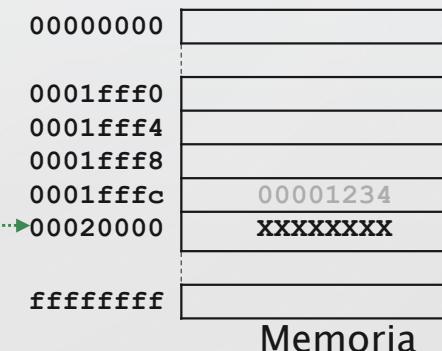
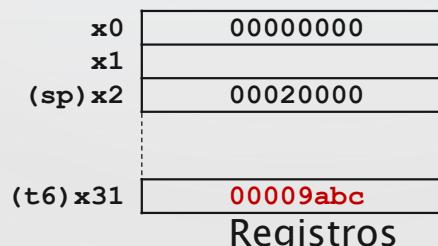
FUNCIONES: GESTIÓN DE PILA

Los datos desapilados permanecen en memoria

Pero **no pueden usarse** porque **serán sobrescritos** por los datos que se apilen con posterioridad

ASM

```
...
    addi sp, sp, -4
    sw t6, 0(sp)
    mv t6, zero
...
    lw t6, 0(sp)
    addi sp, sp, 4
...
    li t6, 0x9abc
    addi sp, sp, -4
    sw t6, 0(sp)
```

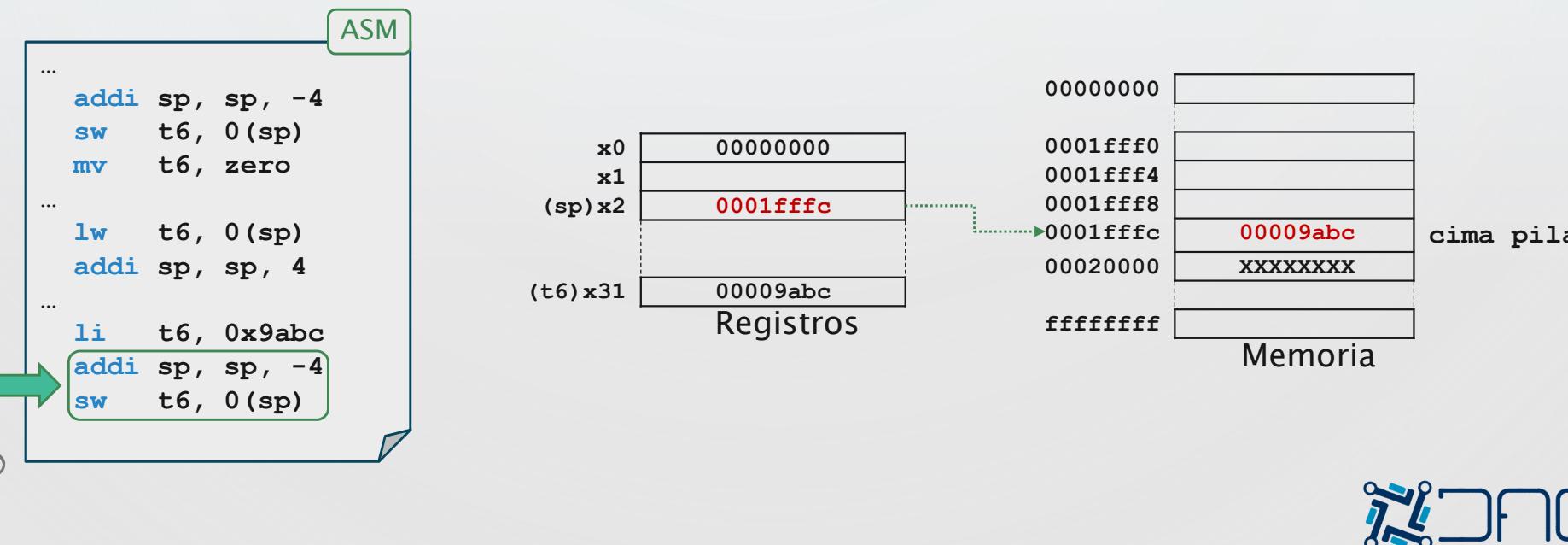


PROGRAMACIÓN ENSAMBLADOR

FUNCIONES: GESTIÓN DE PILA

Los datos desapilados permanecen en memoria

Pero **no pueden usarse** porque **serán sobrescritos** por los datos que se apilen con posterioridad

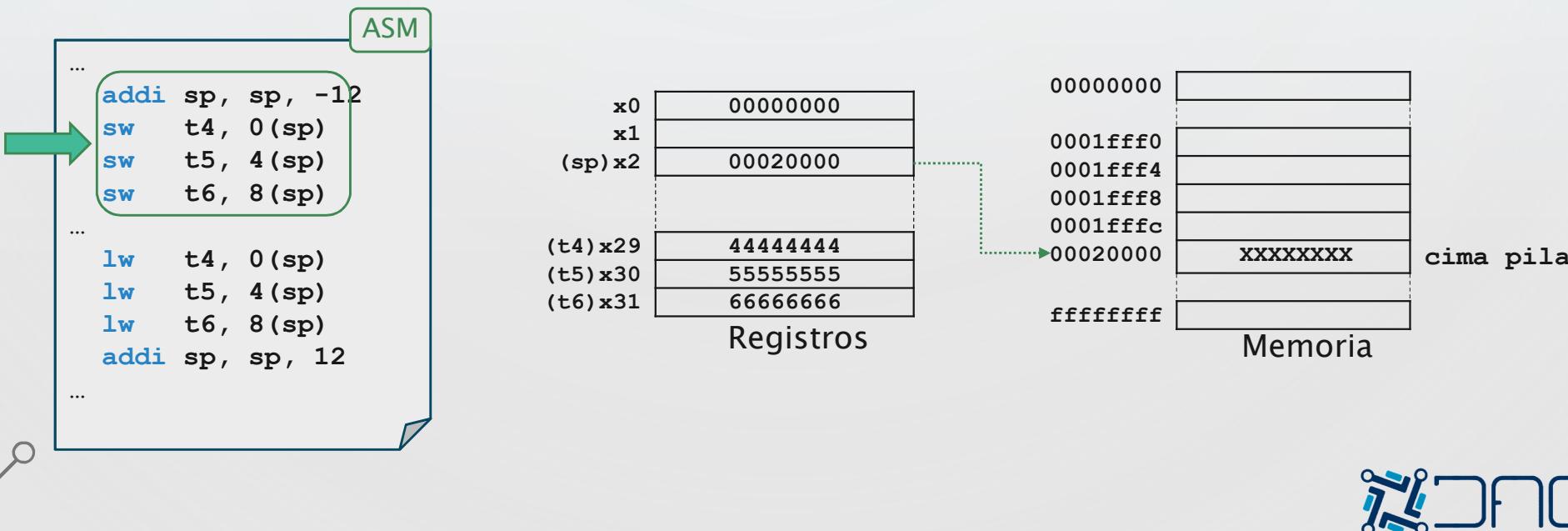


PROGRAMACIÓN ENSAMBLADOR

FUNCIONES: GESTIÓN DE PILA

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 (+0, +4, ...)

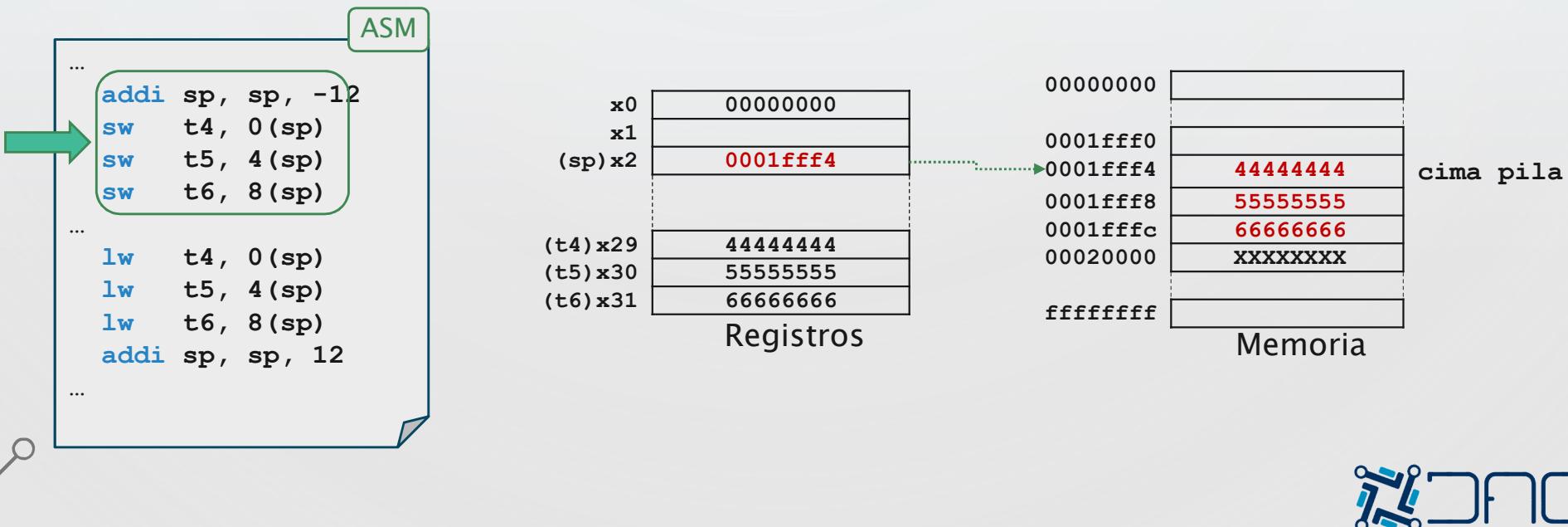


PROGRAMACIÓN ENSAMBLADOR

FUNCIONES: GESTIÓN DE PILA

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 (+0, +4, ...)



PROGRAMACIÓN ENSAMBLADOR

FUNCIONES: GESTIÓN DE PILA

Desapilar un conjunto de datos supone:

- Cargar los datos ubicados en **direcciones consecutivas desde la cima** (+0, +4, ...)
- Incrementar **sp** (el número de bytes que ocupen todos ellos)

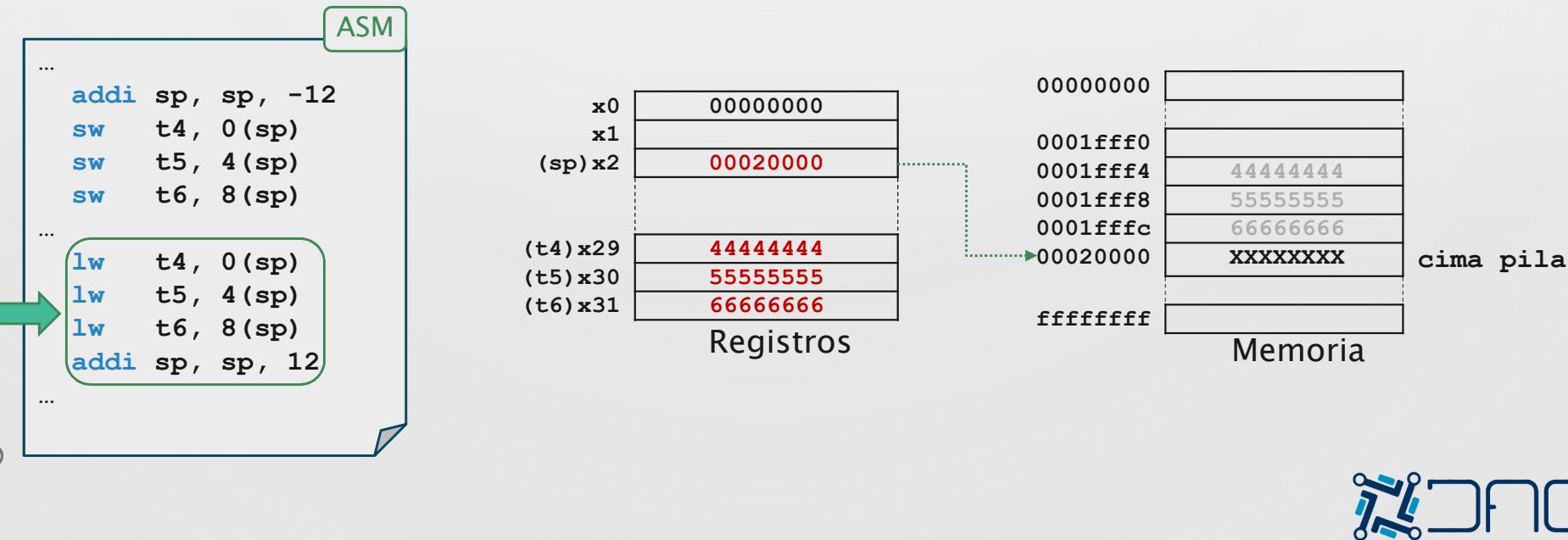


PROGRAMACIÓN ENSAMBLADOR

FUNCIONES: GESTIÓN DE PILA

Desapilar un conjunto de datos supone:

- Cargar los datos ubicados en **direcciones consecutivas desde la cima** (+0, +4, ...)
- Incrementar **sp** (el número de bytes que ocupen todos ellos)



PROGRAMACIÓN ENSAMBLADOR

FUNCIONES: SALVADO DE REGISTROS

Para salvar los registros, que por **CONVENIO** una función debe salvar, se usa la **PILA**

PROGRAMA

ASM

Usar preferiblemente registros preservados

Apilar registros **TEMPORALES** que no se quieran perder al llamar a la función **funcA**

jal funcA

Desapilar registros **TEMPORALES** salvados antes de la llamada a **funcA**

FUNCION NO-HOJA

(llama a otra función)

ASM

...

funcA:
Apilar registros **PRESERVADOS** que se vayan a modificar en el cuerpo de la función **funcA**

...

Apilar registros **TEMPORALES** que no se quieran perder al llamar a la función **funcB**

jal funcB

Desapilar registros **TEMPORALES** salvados antes de la llamada a **funcB**

...

Desapilar registros **PRESERVADOS** que se salvaron al comienzo de la función **funcA**

ret

FUNCION HOJA

(no llama a otra función)

ASM

Usar preferiblemente registros temporales

Desapilar registros **PRESERVADOS** que se salvaron al comienzo de la función **funcB**

ret