Tema 4

Un computador real

Arquitectura de Conjunto de Instrucciones (ISA) RISC-V

Índice

Introducción

- RISC-V es un conjunto de instrucciones (ISA) estándar abierto
- Diseñado en UC Berkeley (2010)
- Desarrollo coordinado por un organismo internacional (https://riscv.org)
- Gran apoyo por parte de universidades, empresas y gobierno

Google FOUNDING IBM FOUNDING Meta Microsoft NVIDIA

Berkeley UNIVERSITY OF CALIFORNIA · UNIVERSITY OF CAMBRIDGE · 東京大学 THE UNIVERSITY OF TOKYO · UNSW SYDNEY · Western UNIVERSITY · CANADA

Universidad de Alcalá · UNIVERSIDAD DE CANTABRIA · Deusto Universidad de Deusto Deustuko Unibertsitatea University of Deusto · BSC Barcelona Supercomputing Center Centro Nacional de Supercomputación

Arquitectura de conjunto de instrucciones (ISA)

- Una arquitectura de conjunto de instrucciones (ISA: *Instruction Set Architecture*) es una descripción funcional de un procesador que describe:
  - los registros de usuario (disponibles para el programador)
  - el conjunto de instrucciones disponibles y su codificación a nivel de máquina
  - la forma de direccionar la memoria

- Las especificaciones son públicas y su uso es gratuito. Se puede implementar y modificar libremente.

- Una ISA no detalla la implementación hardware (puede implementarse de muchas formas)

- En el tema 3 hemos implementado un computador simple basado en la ISA del RISC-V (RV32I) con muchas simplificaciones

Arquitectura de Conjunto de Instrucciones (ISA) RISC-V RV32I

- Vamos a estudiar el repertorio base RV32I (I: Integer) (Hay otros: RV32E,RV64I,RV128I)
  - Datos enteros de 32 bits, direcciones de 32 bits e instrucciones de 32 bits

- Es de tipo RISC (*Reduced Instruction Set Computer*):
  - Conjunto muy reducido de instrucciones simples
  - Instrucciones de tamaño fijo
  - Pocos modos de direccionamiento
  - Arquitectura LOAD/STORE
    - Las instrucciones operan con datos almacenados en registros nunca con memoria
  - Gran número de registros de propósito general (32) para evitar accesos a memoria

Arquitectura de Conjunto de Instrucciones (ISA) RISC-V RV32I

RISC vs CISC

- Los computadores tipo CISC (Complex Instruction Set Computer):
  - Conjunto grande de instrucciones complejas
  - Instrucciones de tamaño variable
  - Muchos modos de direccionamiento
  - Muchas instrucciones trabajan con datos en memoria
  - Hardware más complejo

Ventajas RISC	Desventajas RISC	Ventajas CISC	Desventajas CISC
- Instrucciones rápidas - Pipeline eficiente - Poco consumo - Hardware simple	- Necesita más instrucciones para tareas complejas - Compilar es complejo	- Instrucciones muy potentes - Código con menos instrucciones - Compilador más simple	- Difícil pipeline - Consumo mayor - Hardware complejo

Índice

Registros RISC-V RV32I

- La ALU de RISC-V sólo opera con datos almacenados en registros de usuario.

- La arquitectura RISC-V tiene 32 registros de propósito general de 32 bits
  - Se enumeran de x0 a x31
  - El registro x0 siempre está fijo a 0

  - Si se quiere escribir código que pueda reutilizarse hay que respetar un convenio de uso de los registros (ABI: Interfaz Binaria de Aplicaciones)
  - Cada registro tiene un nombre alternativo que recuerda el uso más habitual establecido en el convenio (ABI)

- Adicionalmente:
  - Registro contador de programa PC
  - Otros registros no visibles por el programador (p.ej: IR, MDR, MAR, etc.).

Registros RISC-V RV32I

Registro	Nombre alternativo	Descripción
x0	zero	Siempre vale 0 no se puede modificar
x1	ra	Dirección de retorno
x2	sp	Puntero de pila
x3	gp	Global pointer
x4	tp	Thread pointer
x5-x7	t0-t2	Temporales 0-2
x8	s0/fp	Saved register 0/frame pointer
x9	s1	Saved register 1
x10-x17	a0-a7	Argumentos de función
x18-x27	s2-s11	Saved register 2-11
x28-x31	t3-t6	Temporales 3-6
pc	pc	Contador de programa

Instrucciones y datos en RISC-V RV32I

- Todas las instrucciones de RISC-V ocupan 32 bits en memoria.
- Las instrucciones en RISC-V operan habitualmente con **datos y direcciones de 32 bits**
- Los datos son **enteros (con signo) codificados en Ca2**, o números enteros sin signo codificados en binario natural (base 2)
- También hay instrucciones para trabajar con datos tamaño 8 (byte) o tamaño 16 (half word)

Arquitectura de memoria RISCV RV32I

Capacidad de memoria:

2^{32}x8b = 2^{30}x32b $\rightarrow$ 4 GB = 1 GW

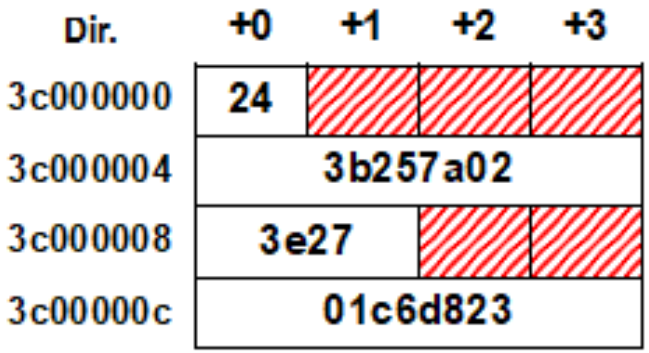
- Direccionable a nivel de bytes (cada byte tiene una dirección única)
- El procesador puede acceder a datos de 32b (Word), de 16b (Halfword) o de 8b (Byte)
- La ordenación de los diferentes bytes de una palabra (o media palabra) es little-endian
- Todos los datos deben estar alineados

Modelo de memoria: alineamiento

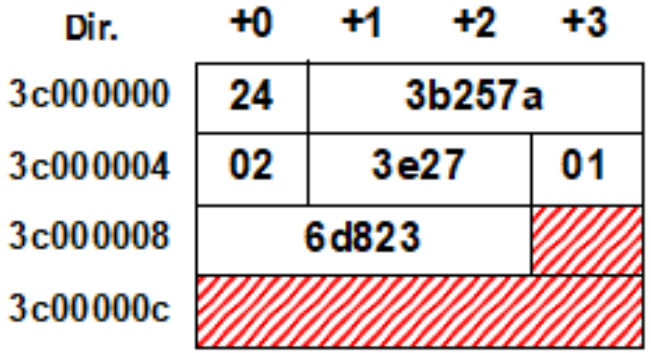
- En la memoria los datos deben estar alineados, es decir, existen restricciones de ubicación en función de su tamaño.
  - Byte: puede ubicarse en cualquier dirección
  - Media palabra: puede ubicarse solo en direcciones múltiplo de 2 (pares)
  - Palabra: puede ubicarse solo en direcciones múltiplo de 4

- Cuando se ubican consecutivamente en memoria datos de distinto tamaño quedan huecos vacíos

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



RISC-V: Datos alineados

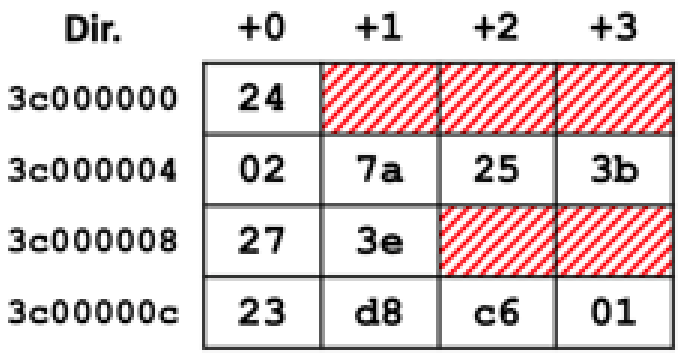


Datos no alineados

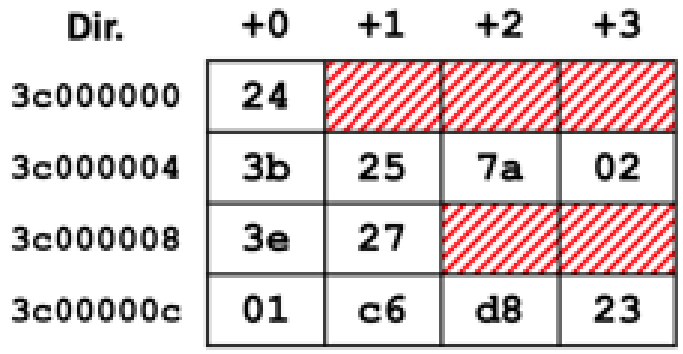
Modelo de memoria: ordenamiento (endianness)

- En la memoria los bytes de una palabra o de una media palabra tienen ordenamiento little-endian
  - En la dirección más baja se coloca el byte menos significativo
  - La dirección del dato coincidirá con la de su byte menos significativo

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



RISC-V: Little-Endian



Big-Endian

Índice

Tipos de instrucción RISC-V RV32I

- Las instrucciones pueden clasificarse en diferentes tipos:
  - Aritméticas
  - Lógicas
  - Desplazamiento
  - Salto
  - Privilegiadas: *permiten acceso a funcionalidades para control del sistema (no se verán en esta asignatura)*

- Muchas de ellas coinciden en sintaxis y operación con las del RISCY (CS3), pero los formatos de instrucción son más complejos

- También algunas instrucciones operan de modo ligeramente diferente que en el RISCY

Formatos de instrucción RISC-V RV32I

Todas las instrucciones son de 32 bits, con diferente formato según la información que requieren.
Los campos opcode, funct3 y funct7 indican a la unidad de control qué instrucción se debe ejecutar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TIPO
funct7							rs2					rs1					funct3			rd					opcode							R
imm [11:0]												rs1					funct3			rd					opcode							I
imm [11:5]							rs2					rs1					funct3			imm [4:0]					opcode							S
imm[12|10:5]							rs2					rs1					funct3			imm[4:1|11]					opcode							B
imm [31:12]																				rd					opcode							U
imm[20|10:1|11|19:12]																				rd					opcode							J

Modos de direccionamiento

- Direccionamiento de Registro: en la instrucción se indican los registros implicados (registros que contienen operandos y registro donde guardar el resultado).

```
add x20,x19,x18      # x20 ← x19 + x18
```

- Direccionamiento Inmediato: uno de los operandos es una constante (con signo) (el otro operando está almacenado en un registro)

```
addi x20,x18,4       # x20 ← x18 + 4
```

- Direccionamiento Indirecto con desplazamiento: la dirección de memoria implicada en la instrucción se indica mediante la suma de un registro (registro base) y una constante (con signo)

```
lw x20,8(x18)        # x20 ← MEM(x18+8)
sw x20,-4(x19)       # MEM(x19-4) ← x20
jal x3,32            # x3 ← pc + 4      pc ← pc + 32  (se estudiará más adelante)
```

Índice

Instrucciones aritméticas y lógicas RV32I

mnemónico	Sintaxis	Tipo	Descripción
add	`add rd, rs1, rs2`	**R**	rd ← rs1 + rs2
sub	`sub rd, rs1, rs2`	**R**	rd ← rs1 – rs2
and	`and rd, rs1, rs2`	**R**	rd ← rs1 & rs2
or	`or rd, rs1, rs2`	**R**	rd ← rs1 | rs2
xor	`xor rd, rs1, rs2`	**R**	rd ← rs1 ^ rs2
addi	`addi rd, rs1, imm12`	**I**	rd ← rs1 + immext
andi	`andi rd, rs1, imm12`	**I**	rd ← rs1 & immext
ori	`ori rd, rs1, imm12`	**I**	rd ← rs1 | immext
xori	`xori rd, rs1, imm12`	**I**	rd ← rs1 ^ immext

`immext:`
`constante de 12 bits extendida en signo a 32 bits`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TIPO
funct7							**rs2**					**rs1**					**funct3**			**rd**					**opcode**							**R**
imm [11:0]												**rs1**					**funct3**			**rd**					**opcode**							**I**

Algunos ejemplos de uso de add y addi

- *addi* permite cargar una constante ”corta” [-2048,+2047] [-0x800,+0x7ff] en un registro (*“load inmediate”*)):

```
addi x19,x0,23          # x19= 23 =0x00000017
addi x20,x0,-2          # x20= -2 =0xfffffffe
```

Más adelante veremos cómo cargar una constante de 32 bits (lui)

- No existe *“subi”* , pero se consigue el mismo efecto sumando una constante negativa:

```
addi x19,x19,-2         # x19 ← x19 - 2
```

- *add* permite copiar el dato de un registro a otro (*”move”*)

```
add x21,x0,x19          # x21 ← x19
add x22,x20,x0          # x22 ← x20
```

- *add* y *addi* permiten generar retardos (*“nop”*)

```
add x0,x0,x0            # x0 ← x0
addi x0,x0,0            # x0 ← x0
```

Ejemplos de cálculo de Ca1 y Ca2

- Cálculo del complemento a 1 *("com")*

```
        xori x19,x19,-1        # x19 ← x19 ^ 0xffffffff

X19:          0001 1100 0000 0000 1111 0011 1100 1010
-1:      ⊕    1111 1111 1111 1111 1111 1111 1111 1111
-------------------------------------------------------
X19:          1110 0011 1111 1111 0000 1100 0011 0101
```

- Cálculo del complemento a 2 *("neg")*

```
        sub x19,x0,x19         # x19 ← x0 - x19

X0:           0000 0000 0000 0000 0000 0000 0000 0000
x19:     --   0001 1100 0000 0000 1111 0011 1100 1010
-------------------------------------------------------
X19:          1110 0011 1111 1111 0000 1100 0011 0110
```

No existen instrucciones not, nand, nor, xnor, nandi, nori, xnori

- Las operaciones *nand*, *nor*, *xnor*, *nandi*, *nori* y *xnori* pueden realizarse con parejas de instrucciones:

- Ej: x7 ← x5 nand x6

```
and x7,x5,x6          # x7 ← x5 & x6        Se hace la and
xori x7,x7,-1         # x7 ← ~x7            Se hace su complemento
```

Nota 1: -1 en Ca2 es 0xffffffff

Nota 2: `xori x7,x7,0xfff` no está permitido por estar el inmediato fuera de rango [-2048,+2047], esto es, [-0x800,0x7ff] pero si podemos hacer `xori x7,x7,-1`

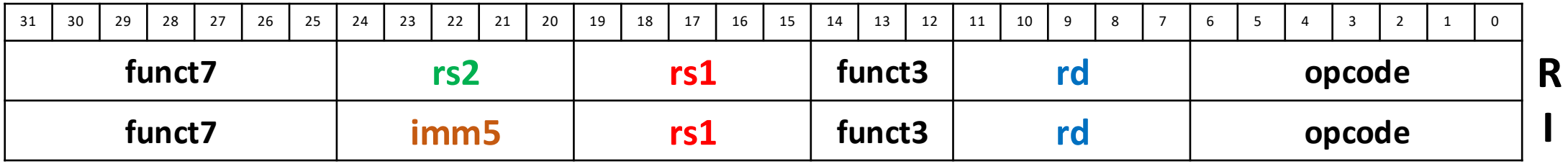
ATENCIÓN: esto es una diferencia con RISCY, donde sí estaba permitido

Índice

Operaciones de desplazamiento

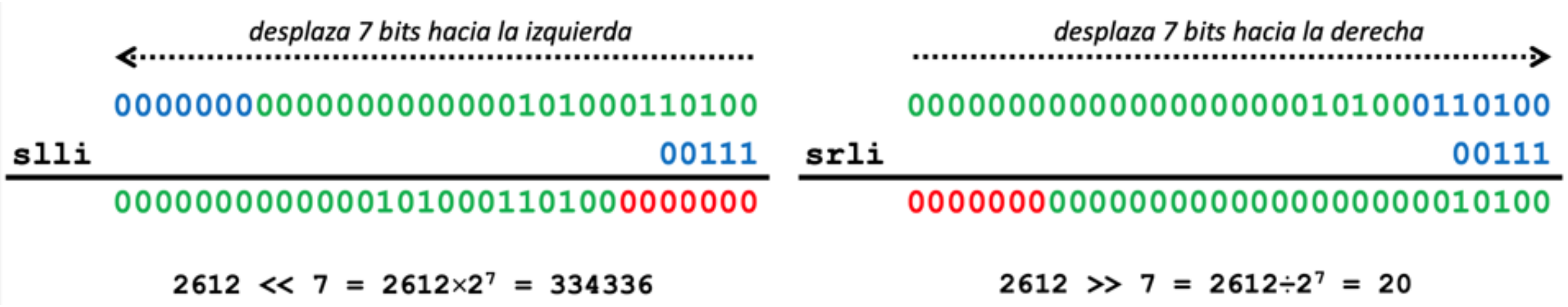
mnemónico	Sintaxis	Tipo	Descripción
sll	`sll rd, rs1, rs2`	**R**	rd ← rs1 << $rs2_{4:0}$
srl	`srl rd, rs1, rs2`	**R**	rd ← rs1 >> $rs2_{4:0}$
sra	`sra rd, rs1, rs2`	**R**	rd ← rs1 >>> $rs2_{4:0}$
slli	`slli rd, rs1, imm5`	**I**	rd ← rs1 << imm5
srli	`srli rd, rs1, imm5`	**I**	rd ← rs1 >> imm5
srai	`srai rd, rs1, imm5`	**I**	rd ← rs1 >>> imm5

Shift Right aritmético (insertando el bit de signo)



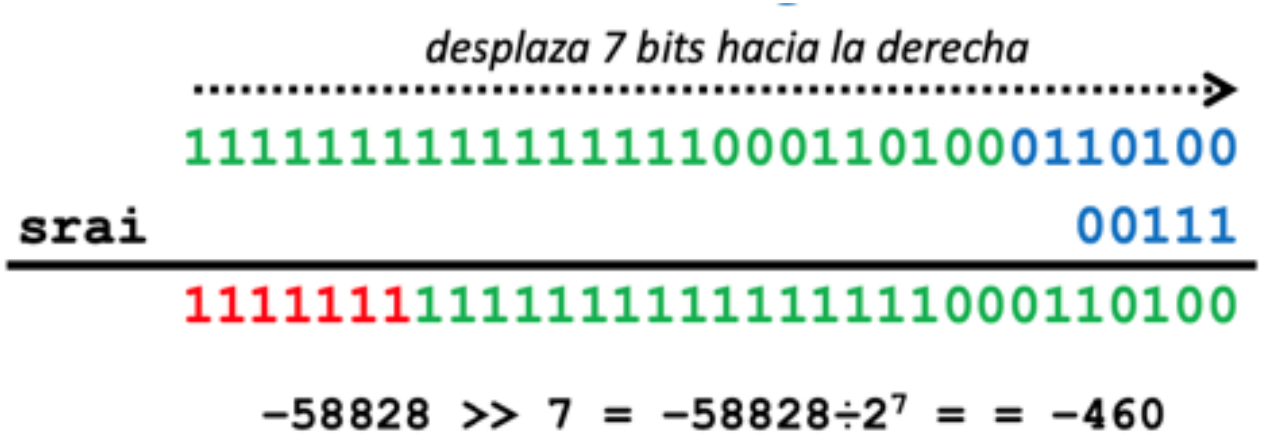
Ejemplos de instrucciones de desplazamiento

- Las instrucciones de desplazamiento lógico insertan 0 por un extremo y descartan los bits que salen por el otro extremo
  - Desplazar n bits a la izquierda equivale a multiplicar por 2^n
  - Desplazar n bits a la derecha equivale a dividir por 2^n



Ejemplos 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 dividir por 2^n datos con signo



Índice

Instrucciones de lectura de memoria (LOAD): lw,lb,lbu,lh,lhu (Formato I)

	Sintaxis	Tipo	Descripción	
lb	`lb rd, imm12(rs1)`	**I**	rd ← sExt(MEM[rs1+inmext]$_{7:0}$)	load byte (signo extendido)
lh	`lh rd, imm12(rs1)`	**I**	rd ← sExt(MEM[rs1+inmext]$_{15:0}$)	load half word (signo extendido)
lw	`lw rd, imm12(rs1)`	**I**	rd ← MEM[rs1+inmext]	load word
lbu	`lbu rd, imm12(rs1)`	**I**	rd ← zExt(MEM[rs1+inmext]$_{7:0}$)	load byte unsigned
lhu	`lhu rd, imm12(rs1)`	**I**	rd ← zExt(MEM[rs1+inmext]$_{15:0}$)	load half word unsigned

sExt: extendido en signo
zExt: extendido con 0

Sólo quedan libres 12 bits para indicar la constante (dato inmediato en Ca2) [-2048,+2047]

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TIPO
imm [11:0]												rs1					funct3			rd					opcode							I

Ejemplo: lw rd , inm12(rs1)

posición dec	posición hex	contenido
8000	0x1f40	08 01 0e 0a
8004	0x1f44	f0 f3 45 2a
8008	0x1f48	a0 b5 e1 14
8012	0x1f4c	0a 10 17 21

La palabra 0x0A0E0108 está almacenada en la memoria en la dirección 8000 en formato little endian:

8000 -> 0x08
8001 -> 0x01
8002 -> 0x0E
8003 -> 0x0A

a2 | 00 00 1f 40

El registro a2 contiene la dirección de la palabra que se va a leer de la memoria (0x1F40 = 8000_{10})

La instrucción `lw a0,0(a2)` carga en el registro a0 el dato tamaño palabra almacenado en la memoria en la dirección que indica a2 (0x1F40 = 8000_{10})

a0 | 0a 0e 01 08

Ejemplo: lw rd , inm12(rs1)

posición dec	posición hex	contenido
8000	0x1f40	08 01 0e 0a
8004	0x1f44	f0 f3 45 2a
8008	0x1f48	a0 b5 e1 14
8012	0x1f4c	0a 10 17 21

a2	00 00 1f 40

Para acceder a un vector o tabla completos, podemos usar el mismo registro base a2 y modificar el desplazamiento:

```
lw x4,0(a2)
lw x5,4(a2)
lw x6,8(a2)
lw x7,12(a2)
```

X4	0a 0e 01 08
X5	2a 45 f3 f0
X6	14 e1 b5 a0
X7	21 17 10 0a

El registro a2 contiene la dirección de la primera palabra de la tabla (0x1F40 = 8000_{10})

Ejemplo: lb rd, inm12(rs1) comparada con lbu rd, inm12(rs1)

posición dec	posición hex	contenido
8000	0x1f40	08 01 0e 0a
8004	0x1f44	fe f3 45 2a
8008	0x1f48	a0 b5 e1 14
8012	0x1f4c	0a 10 17 21

a2	00 00 1f 40
a3	00 00 1f 45

a0	0a 0e 01 08

```
lb a0,0(a2)
```

a0 `00 00 00 08`

```
lb a0,0(a3)
     o
lb a0,5(a2)
```

a0 `ff ff ff f3`

a0 ← signext(memdat)

se extiende el signo

```
lbu a0,0(a2)
```

a0 `00 00 00 08`

```
lbu a0,0(a3)
     o
lbu a0,5(a2)
```

a0 `00 00 00 f3`

a0 ← zext(memdat)

se copia el byte y

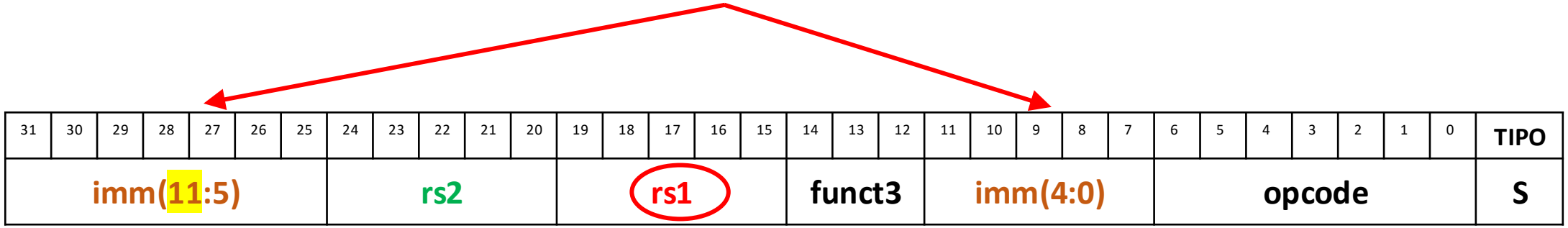
se rellena con ceros

lh y lhu funcionan de modo análogo pero con medias palabras (16 bits)

Instrucciones de escritura en memoria (STORE): sw, sh, sb (Formato S)

	Sintaxis	Tipo	Descripción	
sw	`sw rs2, imm12(rs1)`	**S**	MEM[rs1+immext] ← rs2	store word
sh	`sh rs2, imm12(rs1)`	**S**	MEM[rs1+immext] $_{15:0}$ ← rs2$_{15:0}$	store half word
sb	`sb rs2, imm12(rs1)`	**S**	MEM[rs1+immext] $_{7:0}$ ← rs2$_{7:0}$	store byte

El dato inmediato (offset de 12 bits en Ca2) está repartido [-2048,+2047]



Registro base
"puntero a memoria (rs1)"

Ejemplo: sw/sh/sb rs2, inm12(rs1)

posición dec	posición hex	contenido
8000	0x1f40	a0 b0 c0 d4
8004	0x1f44	f0 f3 45 2a
8008	0x1f48	a0 b5 e1 14
8012	0x1f4c	0a 10 17 21

a0	aa c5 f8 35
a2	00 00 1f 40
a3	00 00 1f 44

```
sw a0,0(a2)
```

```
sh a0,0(a2)
```

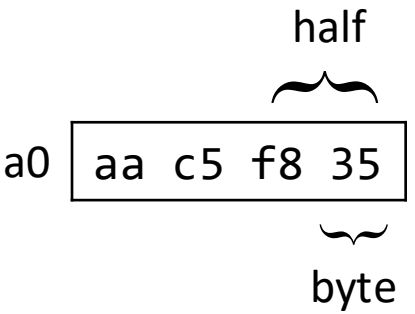
```
sb a0,0(a3)
```

contenido (hex)

8000	0x1f40	35 f8 c5 aa
8004	0x1f44	f0 f3 45 2a
8008	0x1f48	a0 b5 e1 14
8012	0x1f4c	0a 10 17 21

contenido (hex)

8000	0x1f40	35 f8 c0 d4
8004	0x1f44	35 f3 45 2a
8008	0x1f48	a0 b5 e1 14
8012	0x1f4c	0a 10 17 21



Índice

1. Introducción.
2. Arquitectura RISC-V RV32I
3. Conjunto de instrucciones
   - Formatos y modos de direccionamiento
   - Instrucciones aritméticas y lógicas
   - Instrucciones de desplazamiento
   - Instrucciones de acceso a memoria (LOAD/STORE)
   - Instrucciones de salto condicional
   - Instrucción LUI
   - Instrucciones de salto a subrutina (función)
4. Pseudo-instrucciones
5. Lenguaje ensamblador
6. Subrutinas no hojas y gestión de la pila
7. Estructuras de control

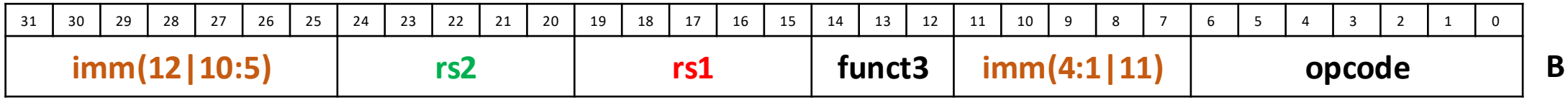
Instrucciones de salto condicional (ii) (Formato B)

mnemónico	Sintaxis	Tipo	Descripción
beq	`beq rs1, rs2, imm13`	**B**	if (rs1=rs2) => PC ← PC + sExt(imm12:1 <<1))
bne	`bne rs1, rs2, imm13`	**B**	if (rs1!=rs2) => PC ← PC + sExt(imm12:1 <<1))
blt	`blt rs1, rs2, imm13`	**B**	if (rs1<rs2) => PC ← PC + sExt(imm12:1 <<1))
bge	`bge rs1, rs2, imm13`	**B**	if (rs1≥rs2) => PC ← PC + sExt(imm12:1 <<1))
bltu	`bltu rs1, rs2, imm13`	**B**	if (rs1<rs2) => PC ← PC + sExt(imm12:1 <<1))
bgeu	`bgeu rs1, rs2, imm13`	**B**	if (rs1≥rs2) => PC ← PC + sExt(imm12:1 <<1))

Para datos sin signo

Son saltos relativos al contador de programa, no se puede saltar a una posición absoluta como en el RISCY

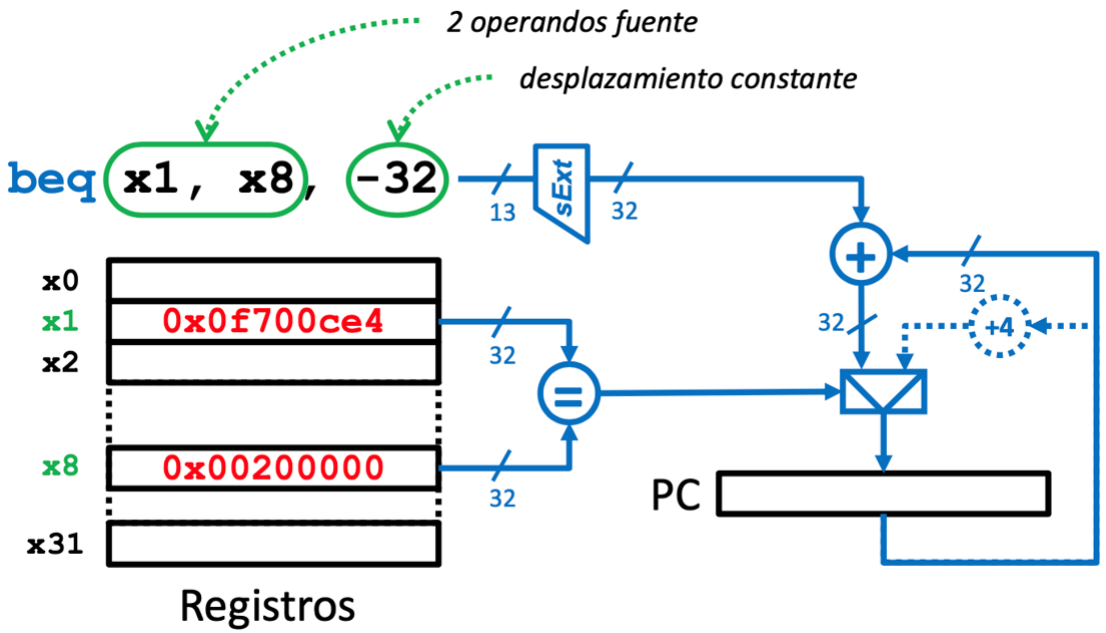
El desplazamiento que se suma a PC ha de ser múltiplo de 4 (palabra de instrucción alineada). Por ello se expresa como imm12:1 << 1. Los dos bits menos significativos son 0.



Nota: en realidad, se permiten saltos de 2 bytes por compatibilidad con la ISA RVC. Por eso se especifican los bits del 12 al 1. En nuestro caso el bit 1 siempre es 0 y los saltos son de 4 bytes

Instrucciones de salto condicional (i) (Formato B)

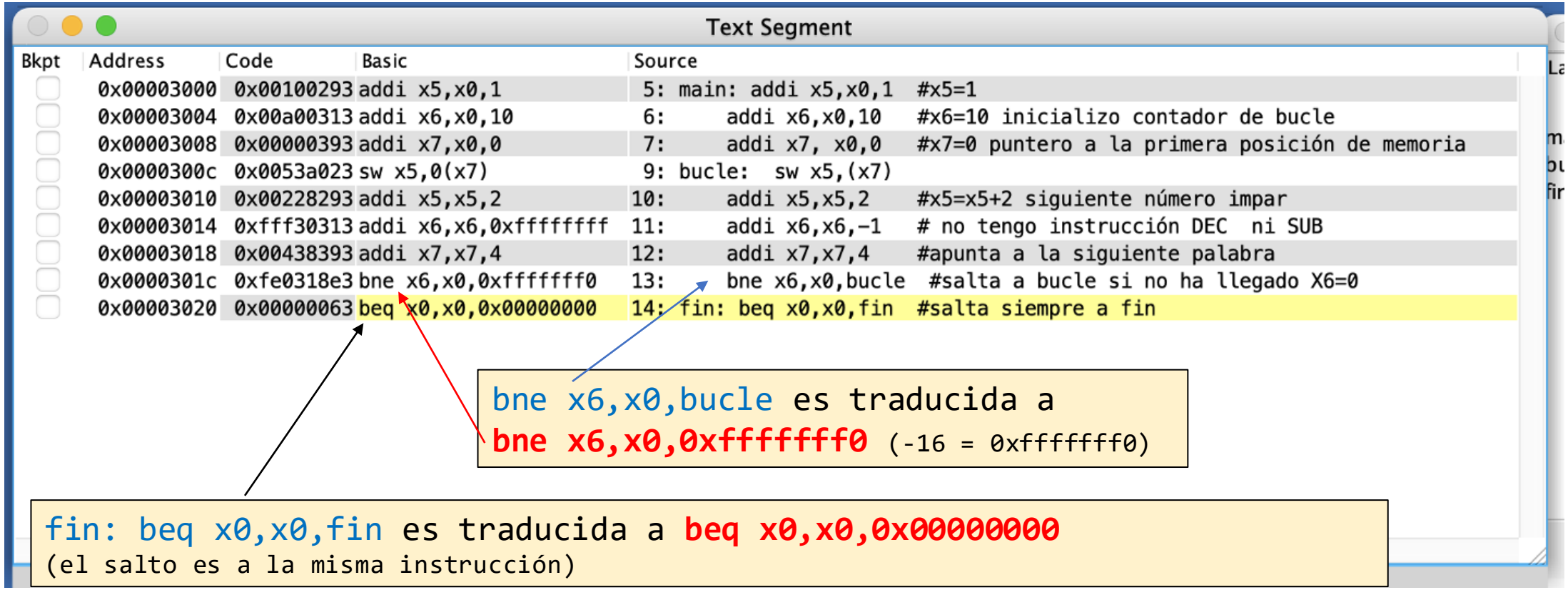
- Permiten romper la secuencia normal de ejecución saltando a una dirección cercana cuando se cumple cierta condición
  - Compara dos operandos fuente (contenidos en dos registros)
  - Si se cumple la condición actualiza el valor de PC sumándole una constante de 13b en Ca2 (desplazamiento) extendida en signo a 32b



En lenguaje ensamblador se usan etiquetas y al obtenerse el código máquina el programa ensamblador calcula el desplazamiento correspondiente

Instrucciones de salto condicional (i) (Ejemplo)

La etiqueta bucle está en 0x300c y el salto bne está en 0x301c,
por tanto, el salto es 4 instrucciones hacia atrás (16 bytes atrás)



Índice

Instrucción lui *(load upper inmediate)* (Formato U)

	Sintaxis	Tipo	Descripción	
lui	lui rd, inm20	**U**	rd ← {imm[31:12],12'b0}	load upper immediate

- carga una constante de 20b en la parte alta de un registro y rellena con 0

```
lui x10,0xff109          # x10 ← 0xff109000
```

- en combinación con addi permite cargar un registro con constantes de 32b

```
Operación:  x5 ← 0xabcde123

lui  x7,0xabcde          #x7 ← abcde000
addi x7,x7,0x123         #x7 ← abcde123
```

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TIPO
imm [31:12]																				rd					opcode								U

Instrucción lui *(load upper inmediate)* (Formato U)

- en combinación con addi permite cargar un registro con constantes de 32b

```
Operación:   x5 ← 0x12345987

lui   x7,0x12346          #x7 ← 12345000
addi x7,x7,0x987           ← error
```

(addi no permite constantes fuera del intervalo [-0x800,0x7ff])

Solución:

```
lui x7,0x12346
addi x7,x7,-0x679
```

0x987 =100110000111, pero si se interpreta como nº con signo en complemento a 2 es el -0x679

100110000111 = -(011001111001) = -0x679

addi extiende en signo el inmediato tendríamos que poner 6 en lugar de 5 ya que 0x987 (=-0x679) se extendería en signo y la operación sería:

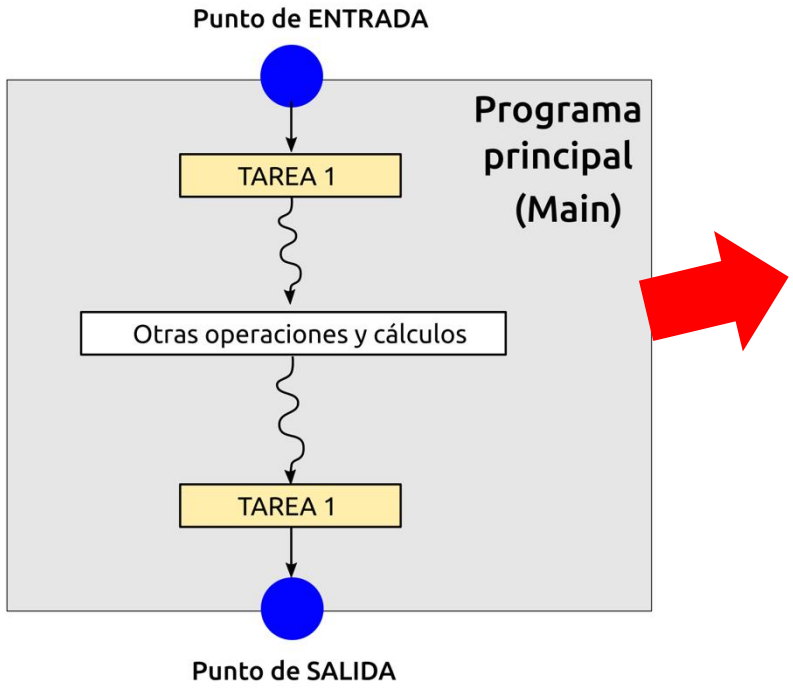
```
  12346000
+ ffffff987
  12345987
```

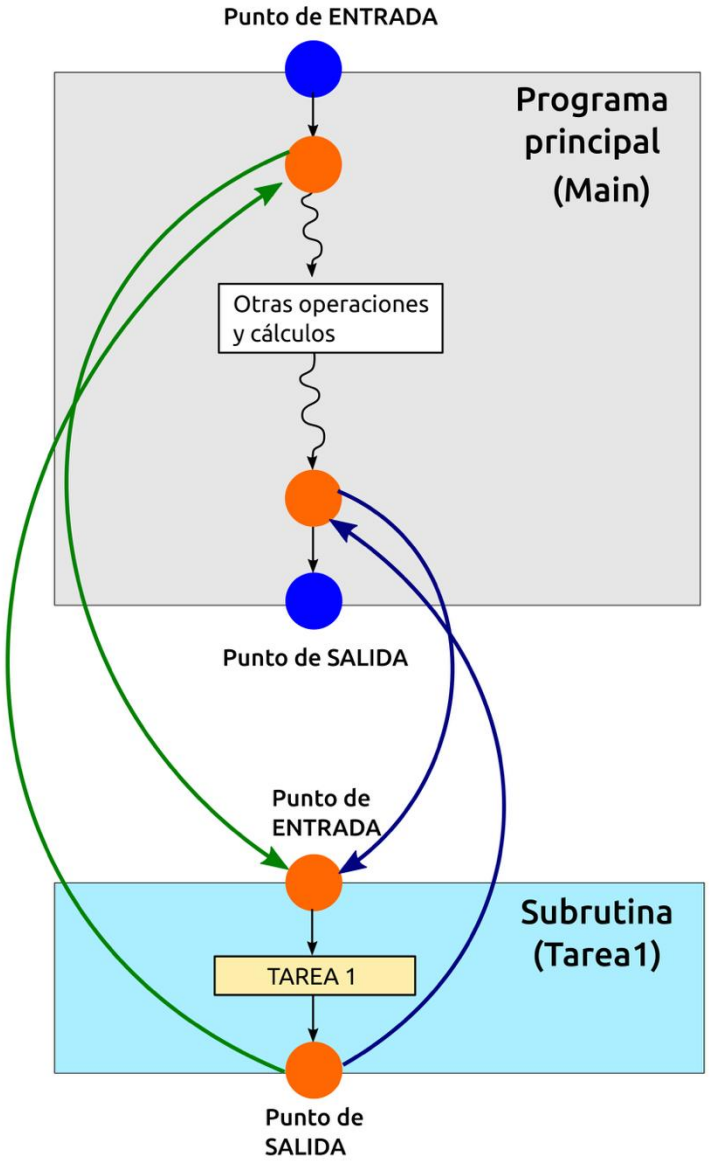
Índice

Concepto de subrutina (función)

- Una **subrutina** es un fragmento de código escrito para realizar una tarea concreta con un doble objetivo:
  - Dividir un problema complejo en problemas más pequeños
  - Poder reutilizarlo sin necesidad de reescribirlo

Cuando se salta a una subrutina es necesario <u>almacenar el valor de PC (+4)</u> para volver al programa principal (dirección de retorno)





Instrucciones de salto a subrutina (jal, jalr)

- RISCV tiene dos instrucciones para salto a subrutinas (dirección de retorno en rd: **rd← PC+4**)
- no tiene instrucciones específicas de retorno de subrutina (puede usarse jalr)

	Sintaxis	Tipo	Descripción	
jalr	jalr rd, rs1, imm12	**I**	PC ← rs1 + sExt(imm12) rd ← PC + 4	jump and link register salto con dirección relativa a registro base
jal	jal rd, imm21	**J**	PC ← PC + sExt($imm_{20:1}$<<1) rd ← PC + 4	jump and link salto con dirección relativa a PC

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TIPO
jalr	imm [11:0]												rs1					funct3			rd					opcode							I
	imm [11:0]												rs1					0	0	0	rd					1	1	0	1	1	1	1	I
jal	imm[20|10:1|11|19:12]																				rd					opcode							J
	imm[20|10:1|11|19:12]																				rd					1	1	0	0	1	1	1	J

Instrucciones de salto a subrutina (jal)

- *jal* permite saltar a una subrutina de modo que:
  - La dirección de salto se obtiene sumando al PC un desplazamiento de 21 bits en notación Ca2
  - La dirección de retorno se guarda en un registro: puede ser cualquiera, pero normalmente es x1 (ra)

```
jal rd, imm21

PC ← PC + sExt(imm20:1<<1), rd ← PC+4
```

```
jal x1,8          # PC ←PC+8 (se saltaría una instrucción)
                  # x1 ←PC+4 (x1 almacena la dirección de retorno, que
                             es la siguiente a jal)
```

Instrucciones de salto relativo a subrutina (jalr)

- *jalr* permite saltar a una subrutina de modo que:
  - La dirección de salto se obtiene sumando al contenido de un registro (registro base) un desplazamiento de 12 bits en Ca2
  - La dirección de retorno se guarda en un registro: puede ser cualquiera, pero normalmente es x1 (ra)

```
jalr rd, rs1, imm12

PC ← rs1 + sExt(imm), rd ← PC+4
```

```
jalr x1,x2,8          # PC ←x2+8 (saltaría a la instrucción en x2+8)
                      # x1 ←PC+4 (x1 almacena la dirección de retorno)
```

Retorno de subrutina con jalr

- *jalr* se utiliza también para retornar de una subrutina, cargando en el PC la dirección de retorno que previamente se había guardado en un registro.

```
jalr rd, rs1, imm12

PC ← rs1 + sExt(imm), rd ← PC+4
```

```
jalr x0,x1,0          # x0 ←PC+4 (se pone x0 porque x0 no se modifica)
                      # PC ←x1+0 (esto es lo que interesa para
                                  retornar al programa principal)
```

Ejemplo de salto y retorno de subrutina (jal, jalr)

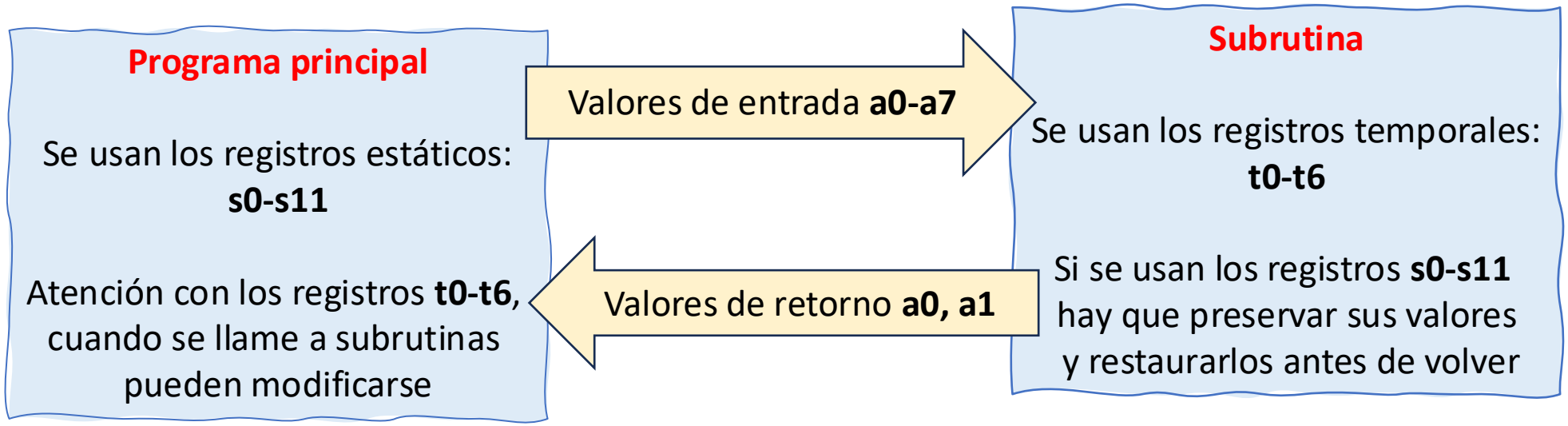
Bucle infinito que incrementa el valor de x10 en cada iteración

```
addi x10,x0,1      →   x10 se inicializa a 1

jal x1,8           →   se salta a dos palabras (2 instrucciones) por debajo, PC ← PC+8
                                                                  y x1 ← PC+4
beq x0,x0,-4       →   salto incondicional a una palabra (1 instrucción) por encima

addi x10,x10,1     →   suma 1 a x10
jalr x0,x1,0       →   salta a dirección de retorno pues PC←x1
```

en ensamblador usaríamos etiquetas

Salto a subrutina (convenio ABI)

- Por convenio:
  - La **dirección de retorno** se guarda en el registro **ra** (return address)
  - Los **argumentos** se pasan a la subrutina en orden en los registros **a0.. a7**
  - La subrutina **devuelve el valor** en orden **a0, a1**



Índice

Pseudoinstrucciones

- La ISA RISC-V incluye, por definición, un conjunto (muy reducido) de instrucciones

- Para facilitar la programación en ensamblador se han estandarizado pseudoinstrucciones que el programa ensamblador se encarga de traducir al conjunto nativo de instrucciones

- Hacen el código más legible

- Sólo **usaremos las más habituales**

Algunas de las pseudoinstrucciones más usadas

Pseudoinstrucción	Operación	Equivalente RISCV
li x8,0x7fe	s8 ← 0x7fe	addi s8, x0, 0x7fe (si dato <12b)
la t1, dato	t1 ← dir dato	auipc t1,imm20 addi t1,t1,imm12
li s8,0x56789def	s8 ← 0x56789def	lui s8,0x5678A addi s8,s8,0xdef
mv x7,x2	x7 ← x2	addi x7,x2,0
not x7,x2	x7 ← Ca1(x2)	xori x7,x2,-1
neg x7,x2	x7 ← Ca2(x2)	sub x7,0,x2
b label	PC ← dir label	beq x0,x0,label
j label	PC ← dir label (más alcance que b)	jal x0,label
call label	PC ← dir label, ra ← PC+4	jal x1,label
ret	PC ← ra	jalr x0,x1,0

Ver explicación de "la" dos páginas más adelante

Son equivalentes salvo por el alcance

dir_label = PC_actual + offset

Esta equivalencia para call solo es válida si el salto es cercano (±1MB)
Si no es así, el ensamblador lo resuelve de otra forma (con auipc y jalr)

salto sin retorno pues x0 no se escribe

Algunas de las pseudoinstrucciones más usadas

Pseudoinstrucción	Operación	Equivalente RISCV
nop	No operación	addi x0,x0,0
bgt x1,x3,label	if(x1>x3) PC ← dir label	blt x3,x1,label
ble x1,x3,label	if(x1≤x3) PC ← dir label	bge x3,x1,label
bgtu x1,x3,label	if(x1>x3) PC ← dir label	bltu x3,x1,label
bleu x1,x3,label	if(x1≤x3) PC ← dir label	bgeu x3,x1,label
beqz x1,label	if(x1=0) PC ← dir label	beq x1,x0,label
bnez x1,label	if(x1!=0) PC ← dir label	bne x1,x0,label

Aclaración de la pseudoinstrucción “la” y la instrucción “auipc”

`la t1, dato`	`t1← dir dato`	`auipc t1,imm20  addi t1,t1,imm12`

Carga en el registro t1 la dirección de memoria de dato, no el contenido, para ello el ensamblador la descompone en dos instrucciones:

auipc t1,imm20
addi t1,t1,imm12

auipc t1,imm20

auipc es una instrucción muy poco usada por el programador (formato U),
y su función es t1 <- PC + (imm20<<12)

En la pseudoinstrucción la t1,dato es el ensamblador el que calcula imm20 e imm12 para que en t1 se almacene la dirección de 32 bits correcta en dos pasos, primero los 20 bits más significativos (con auipc) y después los 12 bits menos significativos (con addi).
No hay ninguna instrucción que permita hacerlo en un paso.

El programa ensamblador calcula el desplazamiento desde el valor de PC hasta el dato y lo divide en dos trozos: imm20 e imm12.

Índice

Lenguaje ensamblador

- Los computadores ejecutan código máquina, pero los programadores desarrollan casi todo el software en lenguajes de alto nivel

- El código ensamblador es el punto intermedio entre el HW y el SW
  - Los programas en un lenguaje de alto nivel se compilan a ensamblador
  - Los programas en lenguaje ensamblador se "ensamblan" a código máquina

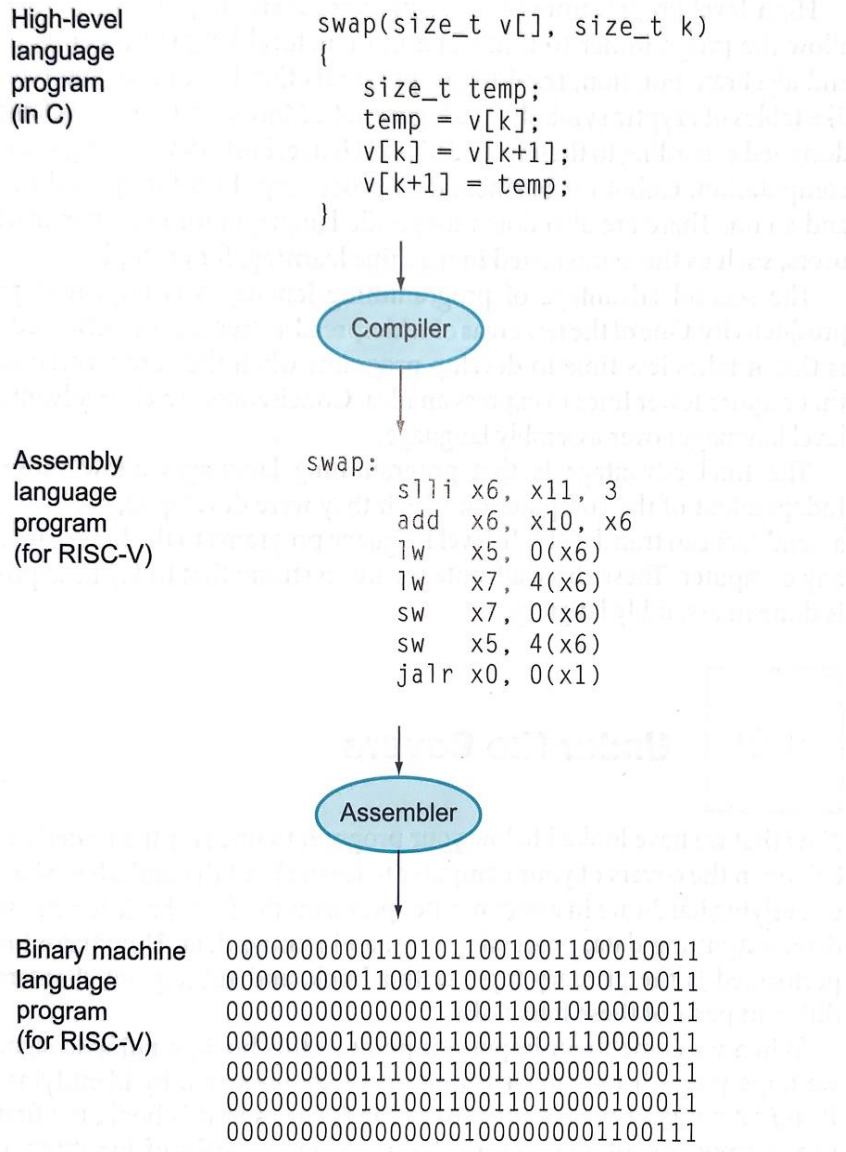
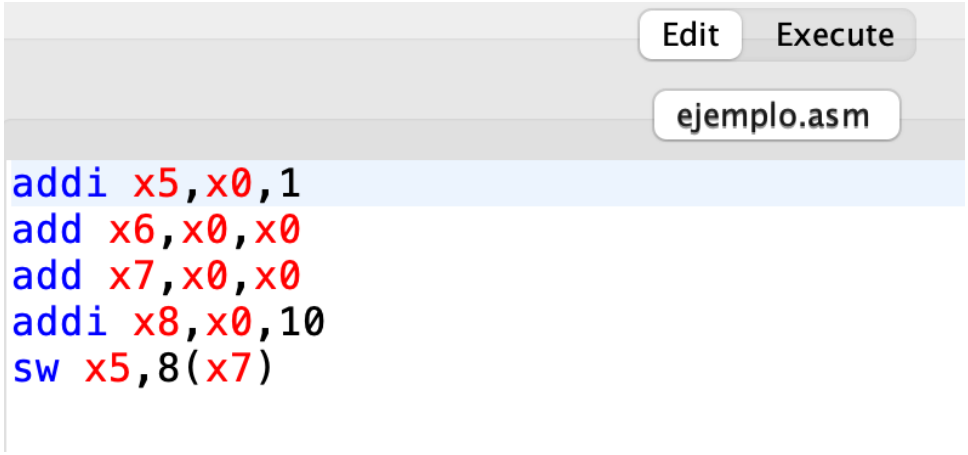
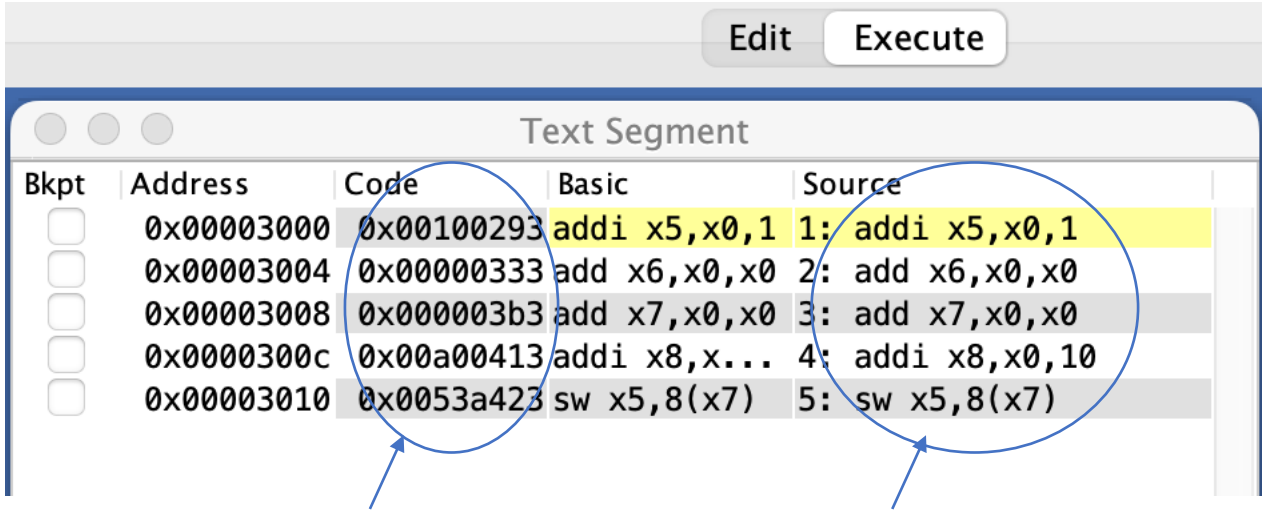


FIGURE 1.4 C program compiled into assembly language and then assembled into binary machine language. Although the translation from high-level language to binary machine language is shown in two steps, some compilers cut out the middleman and produce binary machine language directly. These languages and this program are examined in more detail in Chapter 2.

Lenguaje ensamblador

- El procesador busca y ejecuta instrucciones codificadas en binario almacenadas en memoria (lenguaje máquina)
- El lenguaje ensamblador es una representación legible del lenguaje máquina
- Un programa ensamblador (assembler), por ejemplo: RARS, es un software que "ensambla" (traduce) instrucciones de lenguaje ensamblador a código máquina (binario)





Lenguaje ensamblador

- El lenguaje ensamblador depende de la arquitectura del procesador (x86, ARM, RISC V, …). Un programa escrito en ensamblador para una determinada arquitectura no funciona en otra. En cambio los lenguajes de alto nivel son portables.
- Tradicionalmente la programación en lenguaje ensamblador estaba dirigida a la optimización, los compiladores no generaban un código tan eficiente como el que se podía obtener programando en ensamblador.
- Actualmente el hardware es más potente (la optimización extrema no es tan importante), además los compiladores generan código más eficiente (algunos hacen optimizaciones avanzadas). Para realizar un buen compilador hay que conocer el ensamblador.
- Sigue siendo interesante programar en ensamblador para sistemas empotrados, bootloaders (arranque del sistema), seguridad y otras partes muy específicas donde la optimización es un aspecto crítico.
- Con fines académicos es muy interesante ya que ayuda a entender la arquitectura del procesador.

Lenguaje ensamblador (elementos)

- Un programa en ensamblador puede contener los siguientes elementos:
  - Directivas de ensamblado son indicaciones para el ensamblador y no generan código
  - Etiquetas
  - Instrucciones ejecutables (con sus operandos)
  - Pseudoinstrucciones (con sus operandos)
  - Comentarios

```
    .data
X:  .word   8
Y:  .word   4
Z:  .word   0


    .text
    la      t0, X
    lw      s1, 0(t0)
    lw      s2, 4(t0)
    add     s1, s1, s2 # Suma
    srli    s1, s1, 1 # Divide
    sw      s1, 8(t0)


    li      a7, 10
```

Lenguaje ensamblador: directivas

.data:
indica el inicio de la sección del programa en que se declaran las constantes y variables globales, con o sin un valor inicial

.text:
indica el inicio de la sección del programa donde escribiremos el código

.word:
escribe valores tamaño palabra

(también existen .byte y .half para escribir valores tamaño byte o media palabra)

```
        .data
X:      .word   8
Y:      .word   4
Z:      .word   0


        .text
        la      t0, X
        lw      s1, 0(t0)
        lw      s2, 4(t0)
        add     s1, s1, s2 # Suma
        srli    s1, s1, 1 # Divide
        sw      s1, 8(t0)


        li      a7, 10
```

También es posible definir varios valores en una sola línea:
`X: .word 8, 4, 0`

Lenguaje ensamblador: directivas

.eqv: define constantes

```
    .text
    .eqv    VALOR, 7
    .eqv    TAM, 10


    li      t1, VALOR
    add     t2, t3, t1


    li      s1, TAM
Loop:
    sub     t2, t2, t1
    addi    s1, s1, -1
    bne     s1, zero, Loop
```

.space: reserva N bytes consecutivos

.align: alinea la siguiente dirección, 1:half, 2:word

```
    .data
A:  .byte 0x01
    .align 1
B:  .half 0x0304
C:  .space  12

    .text
    la      t0, C
    sw      x0, 0(t0)
    li      s1, 10
    sw      s1, 4(t0)
    addi    s2, s1, s1
    sw      s2, 8(t0)
```

Dir.	0	1	2	3
Cont.	**01**	-	**04**	**03**

Dir.	4	5	6	7
Cont.	**00**	**00**	**00**	**00**

Dir.	8	9	10	11
Cont.	**0A**	**00**	**00**	**00**

Dir.	12	13	14	15
Cont.	**14**	**00**	**00**	**00**

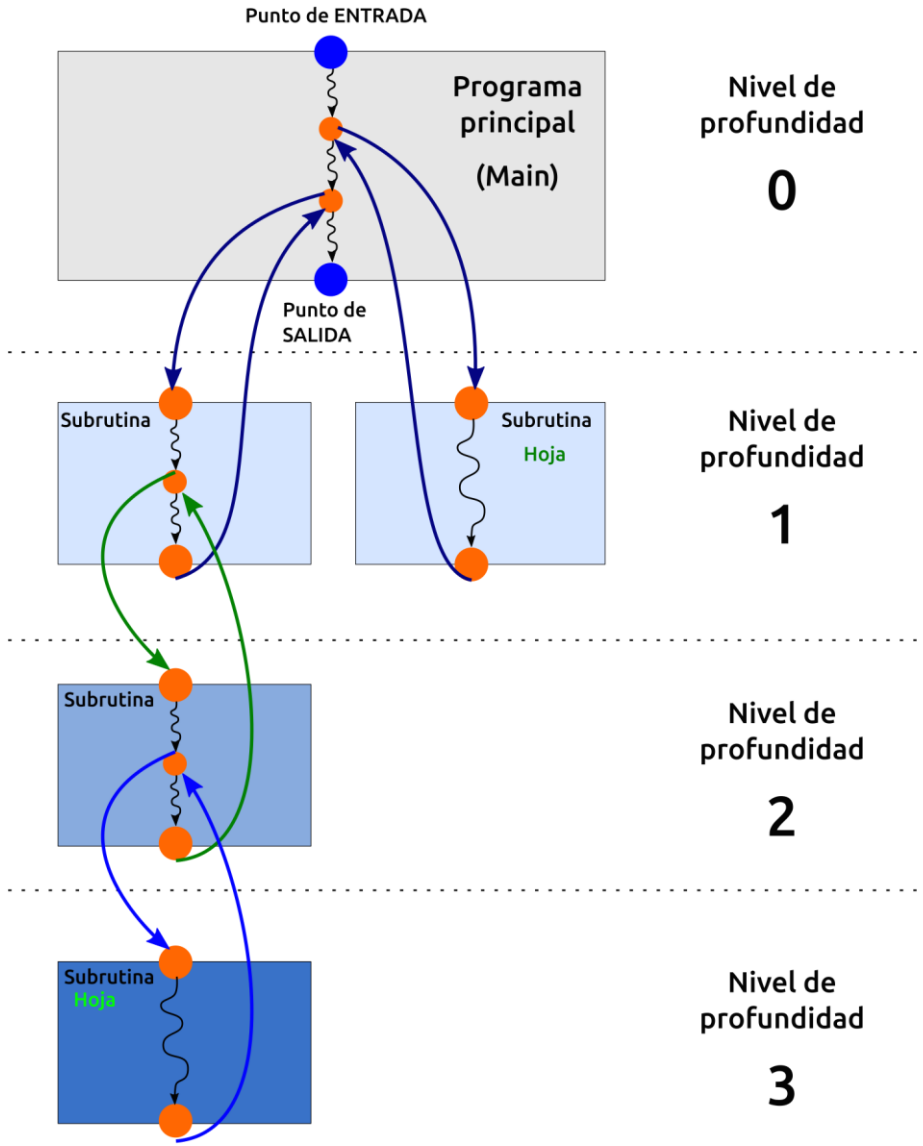
Índice

Subrutinas hojas y subrutinas no hojas

Se pueden distinguir dos tipos de subrutinas:

- subrutina hoja: Es una subrutina que no llama a ninguna otra subrutina.

- subrutina no hoja: Es una subrutina que sí llama a otra subrutina.
  - debe guardar el registro ra en la pila para no perder la dirección de retorno.
  - también debe guardar otros registros temporales (t0..t6) y de argumentos (a0..a7) que pudieran ser sobreescritos.



Gestión de la pila

- La pila (stack) es una región de la memoria RAM donde se pueden almacenar datos temporalmente sin conocer la dirección efectiva que ocupan.

- La pila funciona como una **memoria LIFO** (Last-in First-out): los datos apilados en un cierto orden, se recuperan desapilándolos en orden inverso. Es como si se hiciera una pila de libros, el primero que quito es el último que he puesto.

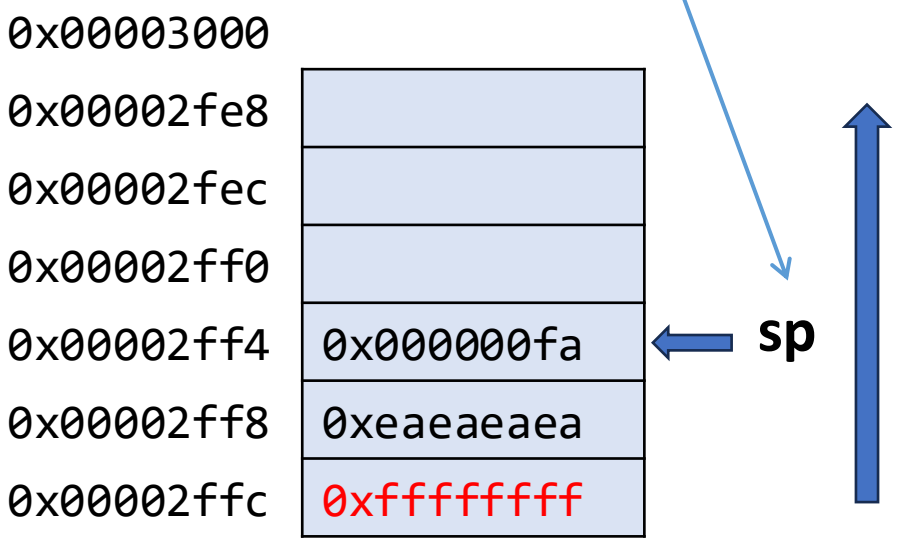
- Cada dato se guarda en una dirección determinada y el siguiente se guarda en la dirección inmediatamente inferior. La pila es descendente: crece hacia direcciones decrecientes de la memoria.

- Se usa el registro sp (x2) para almacenar la dirección de la cima de la pila, que siempre contiene la dirección del último dato apilado (”full”).

- Sobre una pila se pueden realizar **dos operaciones**:
  - **PUSH** (apilar): guardar el dato de un registro en la pila
  - **POP** (desapilar): recuperar un dato de la pila

Ejemplo de uso de la PILA: escritura (push)

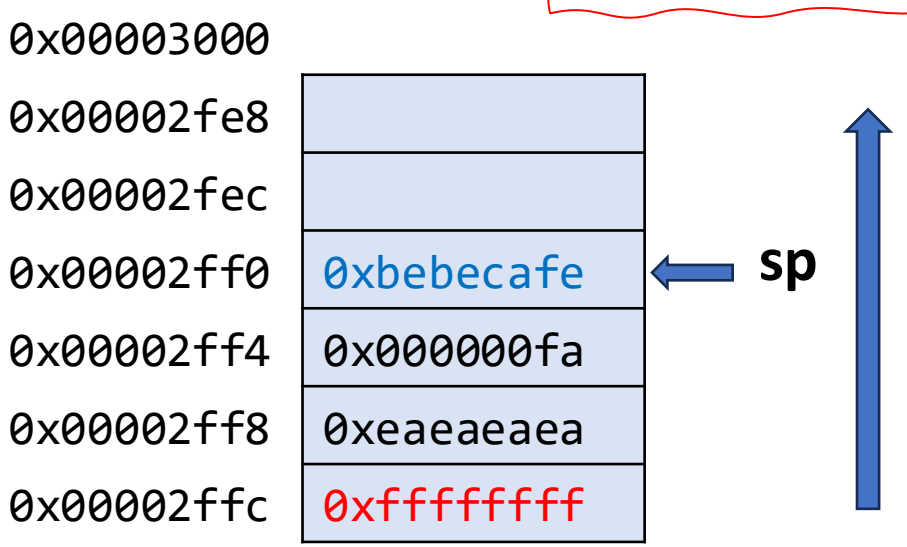
sp apunta siempre al último dato escrito en la pila



antes

Escribir en la pila (PUSH):
sp ← sp - 4
mem(sp) ← reg

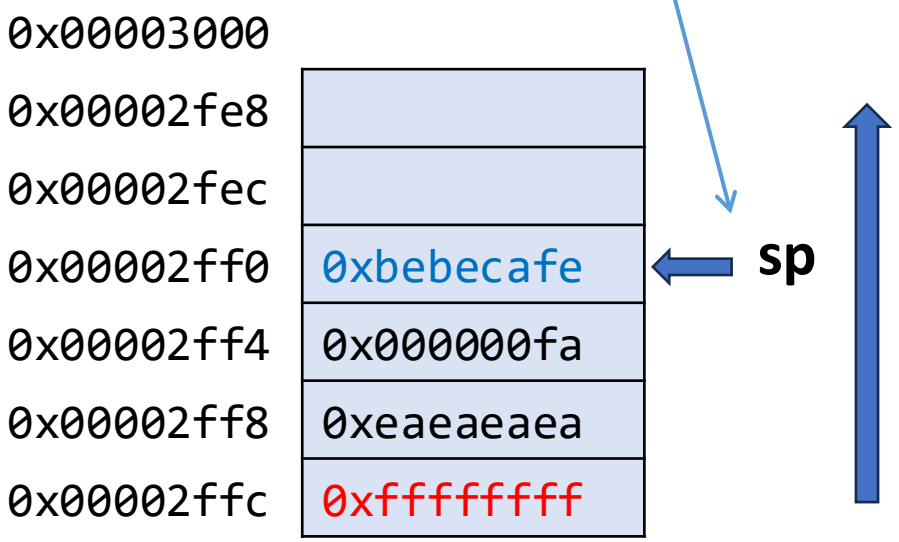
```
#cargo valores iniciales
    li s1,0xbebecafe
#PUSH del registro s1
    addi sp,sp,-4
    sw s1,0(sp)
```



después

Ejemplo de uso de la PILA: lectura (pop)

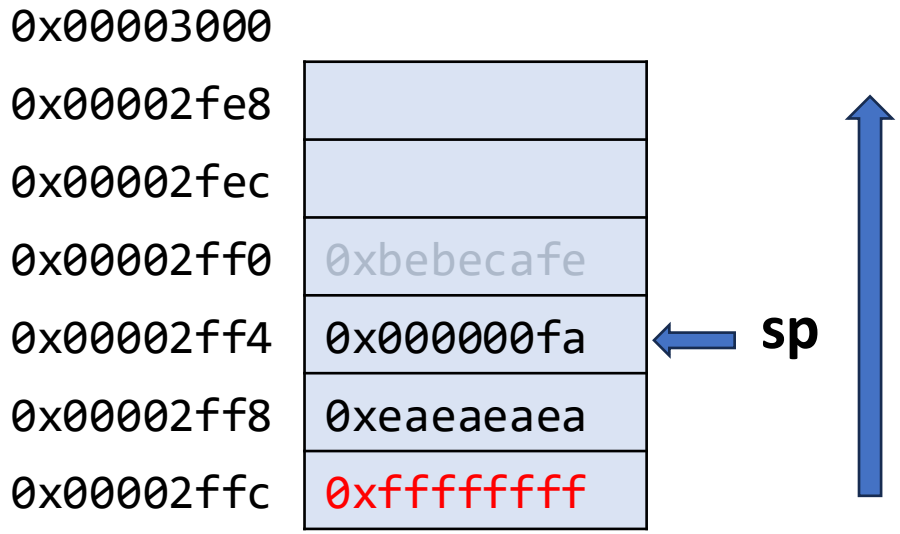
sp apunta siempre al último dato escrito en la pila



antes

Leer de la pila (pop):
reg ← mem(sp)
sp ← sp + 4

```
#POP del registro s1
    lw s1,0(sp)
    addi sp,sp,4
```

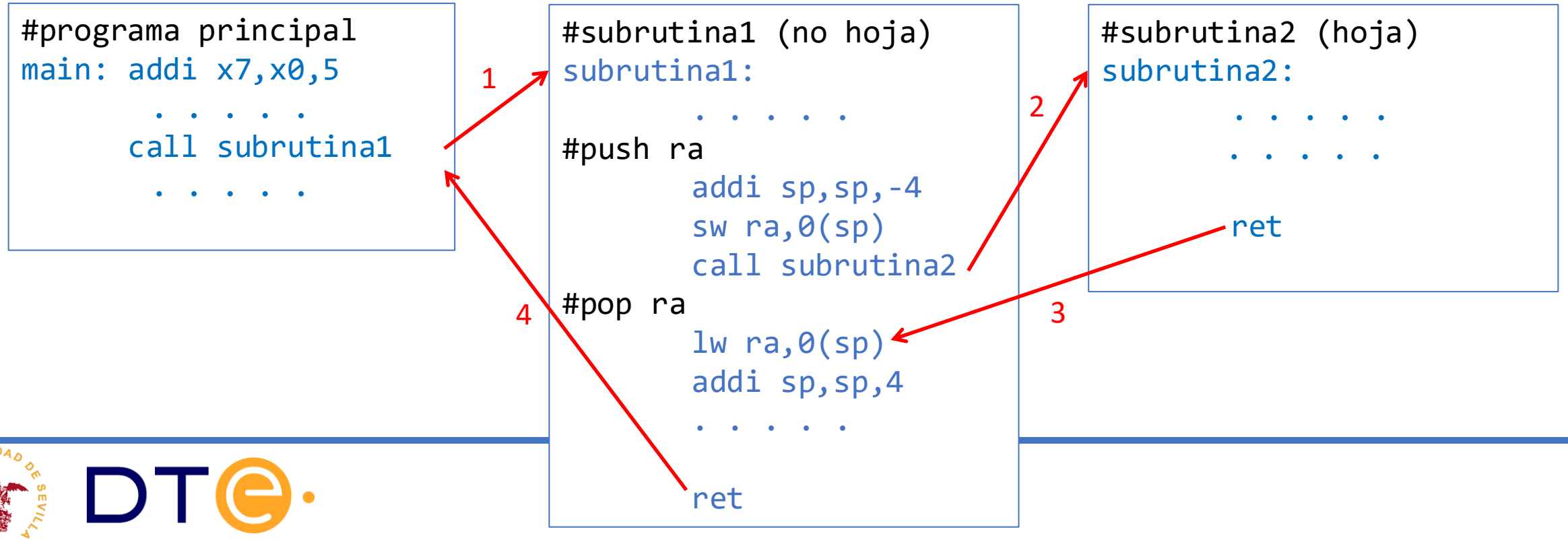


después

Subrutinas que llaman a subrutinas (”subrutinas no hoja”):

Si el programa principal llama a una subrutina (call), la dirección de retorno se guarda en ra

- Si una subrutina llama a otra subrutina, ra debe preservarse en la pila para no perder la dirección de retorno.
- La nueva subrutina puede sobreescribir los registros no preservados, esto es, los registros temporales **(t0..t6) y de argumentos (a0..a7)**



Convenio ABI para manejo de la pila

- La ABI RISCV exige que el sp esté siempre alineado en múltiplos de 16 bytes (16, 32, 64…)
- Como mínimo, reservaremos espacio para apilar 4 registros, aunque solo apilemos 1.

```
#subrutina1 (no hoja)
subrutina1:
        . . . . .
# push ra
        addi sp,sp,-16 #reserva espacio para 4 words
        sw ra,12(sp)   #guardo ra en la pila

        call subrutina2

# pop ra
        lw ra,12(sp) #recupero ra de la pila

        addi sp,sp,+16 #libero espacio de la pila
        . . . . .

        ret
```

Cómo preservar varios registros en la pila

- Si necesitamos preservar varios registros en la pila, además de ra, por ejemplo, porque necesitemos llamar a una subrutina que no preserva sus contenidos, podemos hacerlo de la siguiente forma:

```
#subrutina1 (no hoja)
subrutina1:
        . . . . .
        addi sp,sp,-16   #reservo la pila
        sw ra,12(sp)     # push ra
        sw t0,8(sp)      # push t0
        sw t1,4(sp)      # push t1
        sw t2,0(sp)      # push t2

        call subrutina2

        lw ra,12(sp)     #pop ra
        lw t0,8(sp)      #pop t0
        lw t1,4(sp)      #pop t1
        lw t2,0(sp)      #pop t1
        addi sp,sp,+16  #libero la pila
        . . . . .
        ret
```

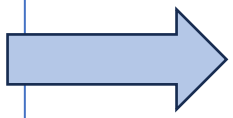
Índice

Estructuras de control

- Estructuras típicas de lenguaje de alto nivel y su traslación a lenguaje ensamblador con instrucciones de RISCV

If-Then-Else

```
// Codigo fuente C:

if (t1 == t2)
  s1 = 10;
else
  s1 = 20;
```



```
If:
    beq   t1, t2, Then
Else:
    li    s1, 20
    j     End
Then:
    li    s1, 10
End:
```

Estructuras de control

- Estructuras típicas de lenguaje de alto nivel y su traslación a lenguaje ensamblador con instrucciones de RISCV

For

```
// Codigo fuente C:

for (t6=0; t6<t1; t6=t6+1)
  s1 = s1 - s2;
```

```
For:
    li      t6, 0
Loop:
    bge     t6, t1, End # *
    sub     s1, s1, s2
    addi    t6, t6, 1
    j       Loop
End:


# * Condicion inversa
```

Estructuras de control

Switch

```
// Codigo fuente C:


switch(s1) {
  case 1:   s2 = 10;
            break;
  case 2:   s2 = 20;
            break;
  case 3:   s2 = 30;
            break;
  default:  s2 = 0;
}
```

```
    li      t1, 1
    beq     s1, t1, Case1
    li      t1, 2
    beq     s1, t1, Case2
    li      t1, 3
    beq     s1, t1, Case3
    j       Default
Case1:
    li      s2, 10
    j       End
Case2:
    li      s2, 20
    j       End
Case3:
    li      s2, 30
    j       End
Default:
    li      s2, 0
End:
```

Estructuras de control

While

```
// Codigo fuente C:


while (t1!= t2) {
    s1 = s1 + 1;
}
```

```
While:
    beq     t1, t2, End # *
    addi    s1, s1, 1
    j       While
End:


# * Condicion inversa
```

Estructuras de control

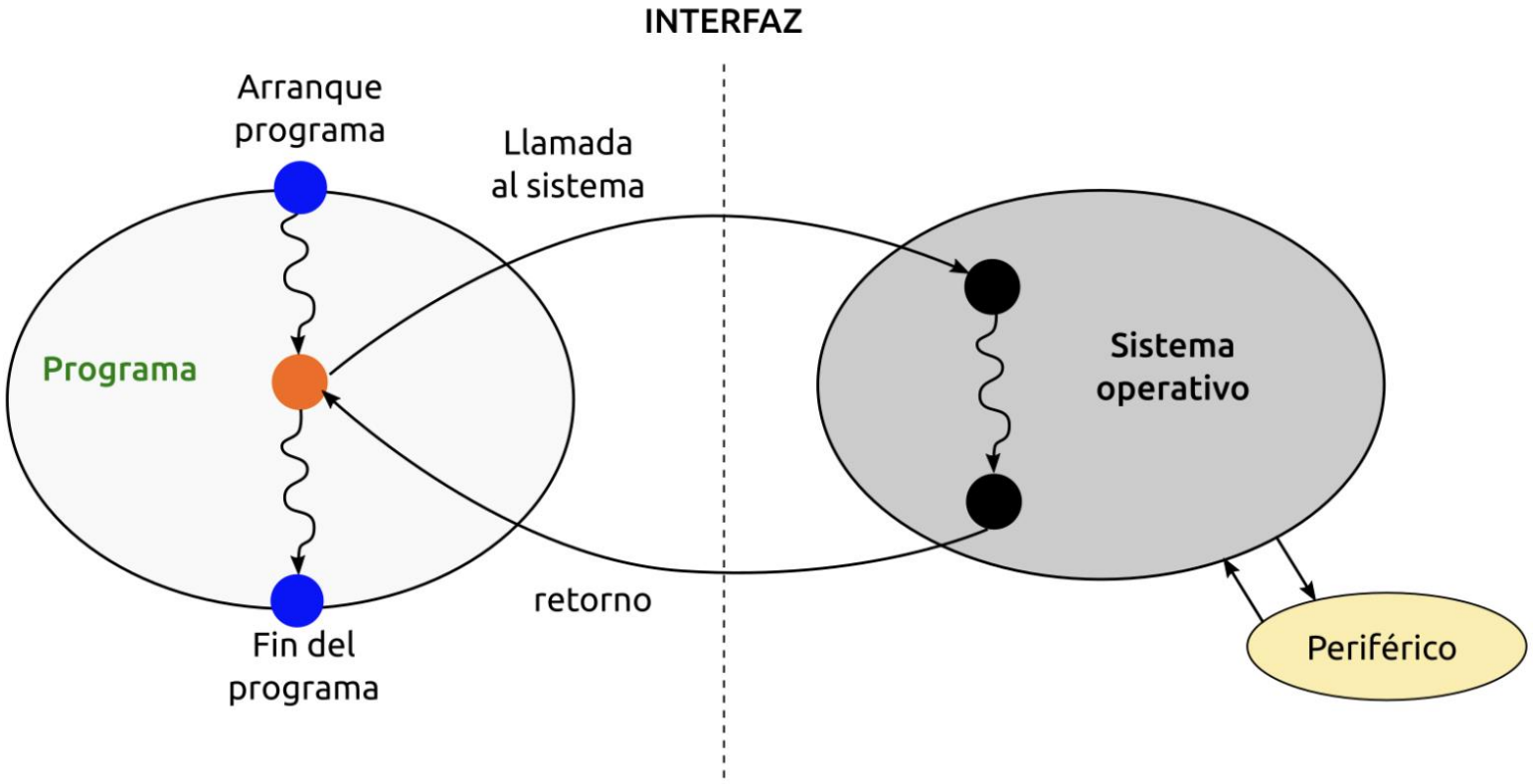
Do-While

```
// Codigo fuente C:


do {
     s1 = s1 + 1;
} while (t1 != t2);
```

```
Do:
     addi     s1, s1, 1
     bne      t1, t2, Do
End:
```

ANEXO: Llamadas al sistema

Los programas pueden interactuar con el sistema operativo invocando servicios (llamadas al sistema). Ejemplos: leer del teclado, imprimir un carácter en la consola o en el display, etc.



ANEXO: Llamadas al sistema

Las llamadas al sistema se realizan a través del registro **a7** y la instrucción **ecall**:

li a7, ECALL_PRINTINT ↔ li a7,1 #seleccionamos la función a realizar
ecall

Nombre	a7=	Descripción	Entradas	Salidas
PRINTINT	1	Imprime un entero	a0 = entero a imprimir	N/A
PRINTSTRING	4	Imprime en consola una cadena terminada en carácter nulo	a0= dirección de la cadena	N/A
READINT	5	Lee un entero de la entrada de consola	N/A	a0
READSTRING	8	Lee una cadena de la consola	a0= dirección del buffer de entrada a1= número máximo de caracteres a leer	N/A
EXIT	10	Salida del programa (devuelve el control al Sistema Operativo)		N/A
PRINTCHAR	11	Imprime un carácter ASCII	a0= carácter a imprimir	N/A
READCHAR	12	Lee un carácter de la consola	N/A	a0= carácter leído

ANEXO: Llamadas al sistema. Ejemplo.

```
                .include “hr1bs3.S”      #incluye los alias de las llamadas al sistema
                .data
X:              .word 7
Y:              .word 8
Z:              .space 4

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

#PRINTINT
                addi a0,t1,0     #el dato a imprimir debe estar en a0
                li a7,ECALL_PRINTINT
                ecall
#EXIT
                li a7,ECALL_EXIT
                ecall
```

códigos de operación RV32I (parte 1)

			funct3		opdoce		
31 25	24 20	19 15	14 12	11 7	6 0		
imm[31:12]				rd	0110111	U	lui
imm[31:12]				rd	0010111	U	auipc
imm[20|10:1|11|19:12]				rd	1101111	J	jal
imm[11:0]		rs1	000	rd	1100111	I	jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B	beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B	bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B	blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B	bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B	bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B	bgeu
imm[11:0]		rs1	0	rd	0000011	I	lb
imm[11:0]		rs1	001	rd	0000011	I	lh
imm[11:0]		rs1	010	rd	0000011	I	lw
imm[11:0]		rs1	100	rd	0000011	I	lbu
imm[11:0]		rs1	101	rd	0000011	I	lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S	sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S	sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S	sw

códigos de operación RV32I (parte 2)

imm[11:0]		rs1	000	rd	0010011	I	addi
imm[11:0]		rs1	010	rd	0010011	I	slti
imm[11:0]		rs1	11	rd	0010011	I	sltiu
imm[11:0]		rs1	100	rd	0010011	I	xori
imm[11:0]		rs1	110	rd	0010011	I	ori
imm[11:0]		rs1	111	rd	0010011	I	andi
0000000	shamt	rs1	001	rd	0010011	I	slli
0000000	shamt	rs1	101	rd	0010011	I	srli
0100000	shamt	rs1	101	rd	0010011	I	srai
0000000	rs2	rs1	000	rd	0110011	R	add
0100000	rs2	rs1	000	rd	0110011	R	sub
0000000	rs2	rs1	001	rd	0110011	R	sll
0000000	rs2	rs1	010	rd	0110011	R	slt
0000000	rs2	rs1	011	rd	0110011	R	sltu
0000000	rs2	rs1	100	rd	0110011	R	xor
0000000	rs2	rs1	101	rd	0110011	R	srl
0100000	rs2	rs1	101	rd	0110011	R	sra
0000000	rs2	rs1	110	rd	0110011	R	or
0000000	rs2	rs1	111	rd	0110011	R	and
31 25	24 20	19 15	14 12	11 7	6 0		
			funct3		opdoce		