

# Arquitectura del procesador MIPS R2000

Aula Virtual → IS09

**Sergio Barrachina Mir**

Área de Arquitectura y Tecnología de Computadores  
Dpt. de Ingeniería y Ciencia de los Computadores  
Universidad Jaume I

# Índice

<b>1. El lenguaje de la máquina</b>	<b>8</b>
1.1. Introducción . . . . .	9
1.2. Operaciones del hardware . . . . .	10
1.3. Operandos del hardware . . . . .	14
1.4. Representación de las instrucciones . . . . .	28
1.5. Instrucciones para la toma de decisiones . . . . .	33
1.6. Procedimientos (o funciones) . . . . .	48
1.7. Otros estilos de direccionamiento . . . . .	70
1.7.1. Constantes u operandos inmediatos . . . . .	71
1.7.2. Direccionamiento en saltos . . . . .	73
1.8. Consideraciones adicionales de la sección . . . . .	82
<b>2. Tipos de datos del R2000</b>	<b>83</b>

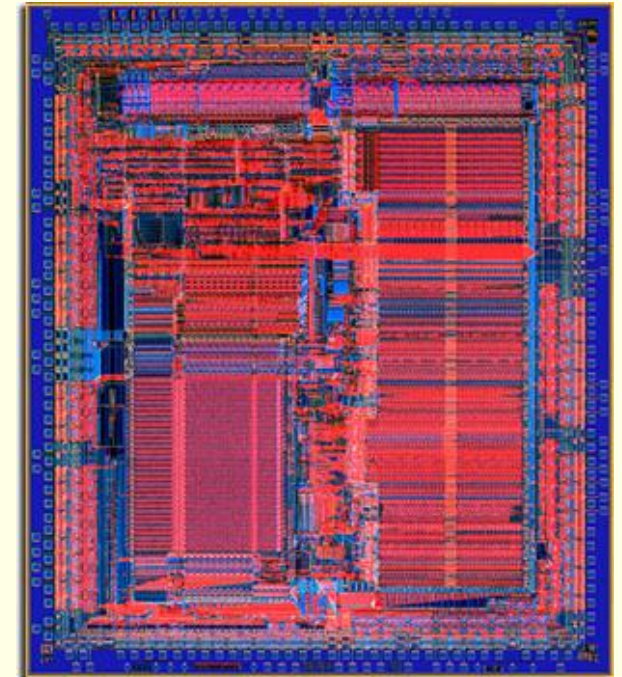
<b>3. Bancos de registros</b>	<b>84</b>
3.1. Banco de registros de números enteros . . . . .	85
3.2. Banco de registros de números reales . . . . .	86
3.3. Banco de registros para el manejo de excepciones . . . . .	87
<b>4. Organización de la memoria</b>	<b>88</b>
<b>5. Juego de instrucciones</b>	<b>91</b>
5.1. Instrucciones aritméticas . . . . .	93
5.2. Instrucciones lógicas . . . . .	95
5.3. Instrucciones de carga y almacenamiento . . . . .	97
5.4. Instrucciones de movimiento con registros HI y LO . . . . .	98
5.5. Instrucciones de comparación . . . . .	99
5.6. Instrucciones de salto condicional . . . . .	100
5.7. Instrucciones de salto incondicional . . . . .	101

<b>6. Programación en ensamblador</b>	<b>102</b>
6.1. Directivas . . . . .	103
6.1.1. Directivas de inicio de las zonas de datos e instrucciones . . . . .	104
6.1.2. Directivas de reserva de espacio . . . . .	105
6.1.3. Directivas de propósito variado . . . . .	107
6.2. Pseudo-instrucciones . . . . .	108
6.2.1. Pseudo-instrucciones de carga y almacenamiento . . . . .	109
6.2.2. Pseudo-instrucciones de salto condicional . . . . .	111

# Presentación del MIPS R2000

¿Sabías que... ?

- Si tienes un sintonizador de televisión por cable digital es probable que esté basado en MIPS.
- Si tienes una consola de vídeo juegos, probablemente esté basada en MIPS.
- Tu correo electrónico probablemente viaje a través de un *router* Cisco basado en MIPS.

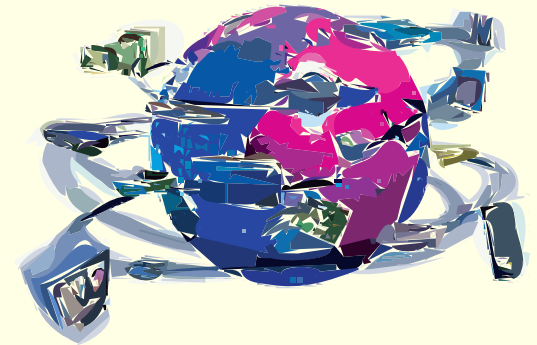


<http://www.mips.com/> (2005)

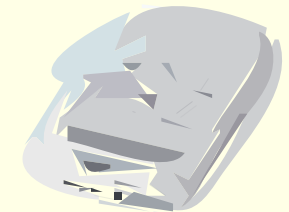
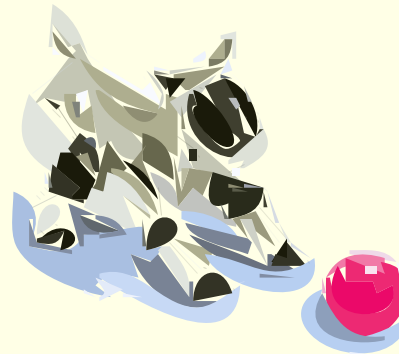
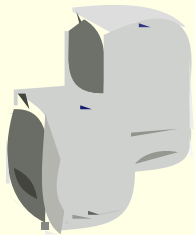
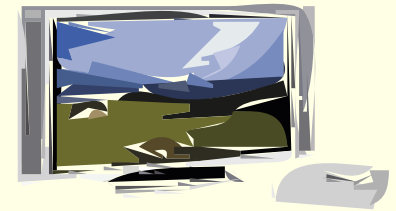
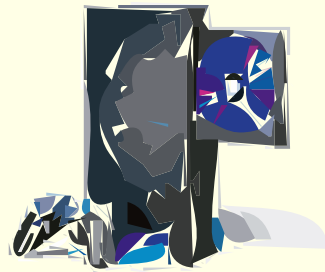
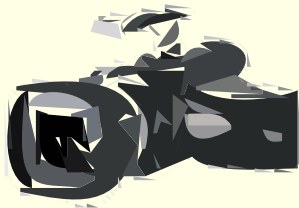
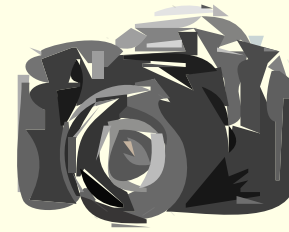
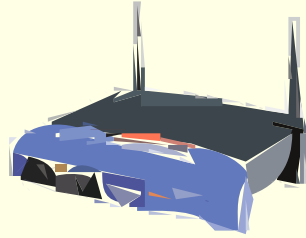
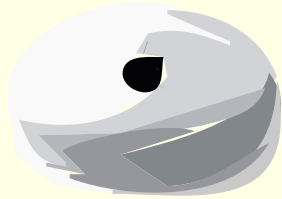
# Presentación del MIPS R2000 (II)

## ► ¿Porcentaje de equipos con MIPS?

Cable Modems	94 %
DSL Modems	40 %
VDSL Modems	93 %
IDTV	40 %
Cable STBs	76 %
Grabadores DVD	75 %
Consolas de Juego	76 %
Impresoras Laser Color	62 %
Fotocopiadoras comerciales color	73 %



# Presentación del MIPS R2000 (III)



# 1 El lenguaje de la máquina

- Esta parte del tema presenta el juego (o repertorio) de instrucciones de un computador real:
  - ⇒ comienza con una notación similar a un lenguaje de programación restringido,
  - ⇒ y llega al lenguaje real de un procesador real.
- Sigue el capítulo 3 del libro:
  - ⇒ David A. Patterson y John L. Hennessy (2000). *Estructura y Diseño de Computadores. Interficie, circuitería/programación*. Editorial Reverté. ISBN: 84-291-2619-8.



# 1.1 Introducción

- Los lenguajes máquina son bastante similares:
  - ⇒ Dialectos más que idiomas.
  - ⇒ Una vez se aprende uno, es fácil entender los demás.
- Un lenguaje máquina tiene como objetivo:
  - ⇒ Hacer fácil la construcción tanto del *hardware* como del compilador, a la vez que  $\uparrow$  rendimiento y  $\downarrow$  coste.

## 1.2 Operaciones del hardware

- Operaciones aritméticas (suma):

```
add a, b, c # a ← b+c
```

- Notación rígida (siempre tres variables), si queremos realizar  $a \leftarrow b+c+d+e$ , entonces:

## 1.2 Operaciones del hardware

- Operaciones aritméticas (suma):

```
add a, b, c # a ← b+c
```

- Notación rígida (siempre tres variables), si queremos realizar  $a \leftarrow b+c+d+e$ , entonces:

```
add a, b, c # a ← b+c
```

```
add a, a, d # a ← b+c+d
```

```
add a, a, e # a ← b+c+d+e
```

⇒ 3 instrucciones para sumar 4 variables.

## 1.2 Operaciones del hardware (II)

- Cada instrucción 3 operandos  $\rightarrow$  *hardware* más sencillo, número variable de operandos  $\rightarrow$  complica el hardware.
- **PRINCIPIO 1:** *La simplicidad favorece la uniformidad.*

### Lenguaje ensamblador MIPS

Categoría	Instrucción	Ejemplo	Significado	Comentario
Aritmética	suma	add a,b,c	$a \leftarrow b + c$	3 operandos
	resta	sub a,b,c	$a \leftarrow b - c$	3 operandos

## 1.2 Operaciones del hardware (III)

### ➤ Ejercicio 1:

Dado el siguiente fragmento de programa en C con 5 variables, escribir el equivalente en lenguaje ensamblador.

```
a=b+c ;  
d=a-e ;
```

### ➤ Solución:

## 1.2 Operaciones del hardware (III)

### ➤ Ejercicio 1:

Dado el siguiente fragmento de programa en C con 5 variables, escribir el equivalente en lenguaje ensamblador.

```
a=b+c ;  
d=a-e ;
```

### ➤ Solución:

```
add a , b , c  
sub d , a , e
```

## 1.2 Operaciones del hardware (IV)

### ➤ Ejercicio 2:

Dado la siguiente expresión más compleja en C con 5 variables, escribir el equivalente en lenguaje ensamblador.

```
f = (g+h) - (i+k);
```

### ➤ Solución:

## 1.2 Operaciones del hardware (IV)

### ➤ Ejercicio 2:

Dado la siguiente expresión más compleja en C con 5 variables, escribir el equivalente en lenguaje ensamblador.

```
f = (g+h) - (i+k);
```

### ➤ Solución:

```
add t0 , g , h # Variable temporal t0 ← g+h  
add t1 , i , k # Variable temporal t1 ← i+k  
sub f , t0 , t1 # f ← t0-t1 , f ← (g+h) - (i+k)
```



## 1.3 Operandos del hardware

- Los operandos no pueden ser variables cualesquiera (como en los lenguajes de alto nivel).
  - ⇒ Serie limitada de posiciones especiales: *registros*.
- MIPS tiene 32 registros de 32 bits para enteros:  $\$0, \$1, \dots, \$31$
- Convenio MIPS permite representarlos mediante dos caracteres precedidos por el símbolo  $\$$ . Algunos son  $\$s0, \$s1\dots$  y  $\$t0, \$t1\dots$
- ¿Por qué sólo 32? ¿Por qué no muchos más registros?
  - ⇒ **PRINCIPIO 2:** *Cuanto más pequeño, más rápido.*

## 1.3 Operandos del hardware (II)

➤ Es tarea del compilador asociar variables de programas a registros.

➤ **Ejercicio 3:**

Dada la sentencia en C « $f=(g+h)-(i+j)$ », y sabiendo que las variables  $f$ ,  $g$ ,  $h$ ,  $i$  y  $j$  han sido asignadas a los registros  $\$s0$ ,  $\$s1$ ,  $\$s2$ ,  $\$s3$  y  $\$s4$ , ¿cuál es el código en ensamblador MIPS?

➤ **Solución:**

## 1.3 Operandos del hardware (II)

➤ Es tarea del compilador asociar variables de programas a registros.

➤ **Ejercicio 3:**

Dada la sentencia en C « $f=(g+h)-(i+j)$ », y sabiendo que las variables  $f$ ,  $g$ ,  $h$ ,  $i$  y  $j$  han sido asignadas a los registros  $\$s0$ ,  $\$s1$ ,  $\$s2$ ,  $\$s3$  y  $\$s4$ , ¿cuál es el código en ensamblador MIPS?

➤ **Solución:**

```
add $t0 , $s1 , $s2 # registro $t0 contiene g+h
```

```
add $t1 , $s3 , $s4 # registro $t1 contiene i+j
```

```
sub $s0 , $t0 , $t1 #  $f \leftarrow \$t0 - \$t1$ , o  $(g+h) - (i+j)$ 
```

## 1.3 Operandos del hardware (III)

- Lenguajes de programación no sólo utilizan estructuras simples, también complejas (p.e. *vectores*).
- ¿Cómo puede un computador trabajar con vectores?  
Un vector no cabe en los registros  $\Rightarrow$  MEMORIA  
(las estructuras de datos más complejas deben utilizar la memoria).
- Transferencia desde la memoria a los registros y viceversa  
 $\Rightarrow$  instrucciones de carga y almacenamiento.

## 1.3 Operandos del hardware (IV)

- ▶ Podemos ver la memoria como si fuera un vector.
- ▶ Si llamamos Memoria a este vector, podemos referirnos a la posición 3 como:
  - ⇒ Memoria[2]
- ▶ ¿Qué hay en Memoria[2]?
  - ⇒ Unos y ceros que pueden interpretarse, p.e., como el número 2.

Dir.	Contenido
⋮	⋮
3	100
2	10
1	101
0	1

**Memoria**

# 1.3 Operandos del hardware (V)

➤ La instrucción de carga desde memoria **lw** (*load word*) especifica:

1. El registro destino.
2. La dirección de memoria:
  - ➡ Valor constante +  
Contenido de un registro.

➤ Ejemplo. Si \$s0 == 1:

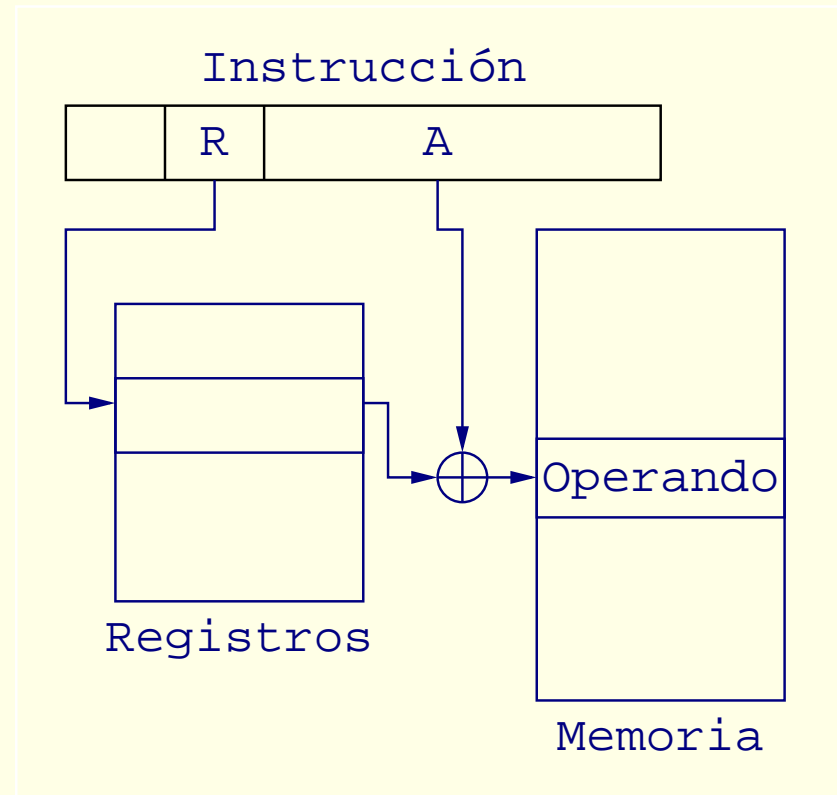
**lw** \$t0, 2(\$s0) # \$t0 ← Memoria[2+1]

Dir.	Contenido
:	:
3	<b>100</b>
2	10
1	101
0	1

**Memoria**

## 1.3 Operandos del hardware (VI)

- El modo de direccionamiento del operando fuente de la instrucción **lw** (p.e., **lw** \$t0, 2(\$s0)) es el **desplazamiento con registro base**.



- ¿Cuál es el modo de direccionamiento del operando destino?

## 1.3 Operandos del hardware (VII)

### ➤ Ejercicio 4:

Dada la sentencia C: « $g=h+A[8]$ »,

obtener el equivalente en ensamblador sabiendo:

- Que el compilador ha asociado las variables  $g$  y  $h$  a los registros  $\$s1$  y  $\$s2$ .
- Que la dirección de comienzo del vector  $A$  está almacenada en el registro  $\$s3$ .

### ➤ Solución:



## 1.3 Operandos del hardware (VII)

### ► Ejercicio 4:

Dada la sentencia C: « $g=h+A[8]$ »,

obtener el equivalente en ensamblador sabiendo:

- ⇒ Que el compilador ha asociado las variables  $g$  y  $h$  a los registros  $\$s1$  y  $\$s2$ .
- ⇒ Que la dirección de comienzo del vector  $A$  está almacenada en el registro  $\$s3$ .

### ► Solución:

```
lw    $t0 , 8($s3)      # registro temporal $t0 ← A[8]
add   $s1 , $s2 , $t0   # g=h+A[8]
```

## 1.3 Operandos del hardware (VIII)

- El acceso a memoria es en realidad más complejo:
  - ➡ Las **palabras** en el R2000 son de 32 bits (4 bytes).
  - ➡ Sin embargo, se pueden direccionar bytes individuales.
  - ➡ Por consiguiente, las direcciones de **palabras** contiguas se diferencian en 4.

Dir.	Contenido
:	:
12	100
8	10
4	101
0	1

**Memoria**

## 1.3 Operandos del hardware (IX)

- ▶ Por lo tanto, si queremos leer el contenido de  $A[8]$ ,
- ▶ en lugar de:

```
lw $t0 , 8 ($s3)      # $t0 ← A[8/4]
```

- ▶ deberemos utilizar:

```
lw $t0 , 32 ($s3)     # $t0 ← A[8]
```

# 1.3 Operandos del hardware (X)

➤ La instrucción de almacenamiento en memoria **sw** (*store word*) tiene el formato:

1. Instrucción.
2. Registro fuente.
3. Dirección de memoria:
  - ➡ Valor constante +  
Contenido de un registro.

➤ Ej. Si  $\$s3 == 4$  y  $\$t0 == 110$ :

**sw**  $\$t0, 8(\$s3) \# Memoria[8+4] \leftarrow \$t0$

Dir.	Contenido
:	:
12	110
8	10
4	101
0	1

**Memoria**

## 1.3 Operandos del hardware (XI)

- **Ejercicio 5:** Dada la sentencia C:  $A[12]=h+A[8]$ , obtener el equivalente en ensamblador sabiendo:
  - Que el registro  $\$s3$  contiene la dirección de comienzo de **A**.
  - Que el compilador ha asociado la variable  $h$  al registro  $\$s2$ .
- **Solución:**

## 1.3 Operandos del hardware (XI)

- **Ejercicio 5:** Dada la sentencia C:  $A[12]=h+A[8]$ , obtener el equivalente en ensamblador sabiendo:
  - ➡ Que el registro  $\$s3$  contiene la dirección de comienzo de **A**.
  - ➡ Que el compilador ha asociado la variable  $h$  al registro  $\$s2$ .

### ➤ Solución:

```
lw    $t0 , 32($s3)    # registro temp. $t0 ← A[8]
add   $t0 , $s2 , $t0  # registro temp. $t0 ← h+A[8]
sw    $t0 , 48($s3)    # A[12] ← h+A[8]
```

## 1.3 Operandos del hardware (XII)

- ▶ ¿Como acceder a  $A[i]$ , con  $i$  variable?

## 1.3 Operandos del hardware (XII)

➤ ¿Como acceder a  $A[i]$ , con  $i$  variable?

1. Multiplicar  $i \times 4$  ( $i$  en  $\$s4$  y aún no sabemos multiplicar):



## 1.3 Operandos del hardware (XII)

➤ ¿Como acceder a  $A[i]$ , con  $i$  variable?

1. Multiplicar  $i \times 4$  ( $i$  en  $\$s4$  y aún no sabemos multiplicar):

```
add $t1 , $s4 , $s4      # $t1 ← 2 * i
add $t1 , $t1 , $t1      # $t1 ← 4 * i
```

## 1.3 Operandos del hardware (XII)

➤ ¿Como acceder a  $A[i]$ , con  $i$  variable?

1. Multiplicar  $i \times 4$  ( $i$  en  $\$s4$  y aún no sabemos multiplicar):

```
add $t1 , $s4 , $s4      # $t1 ← 2 * i
add $t1 , $t1 , $t1     # $t1 ← 4 * i
```

2. Calcular la dirección de  $A[i]$  (la dirección de comienzo en  $\$s3$ ):

## 1.3 Operandos del hardware (XII)

➤ ¿Como acceder a  $A[i]$ , con  $i$  variable?

1. Multiplicar  $i \times 4$  ( $i$  en  $\$s4$  y aún no sabemos multiplicar):

```
add $t1 , $s4 , $s4    # $t1 ← 2 * i
add $t1 , $t1 , $t1    # $t1 ← 4 * i
```

2. Calcular la dirección de  $A[i]$  (la dirección de comienzo en  $\$s3$ ):

```
add $t1 , $t1 , $s3    # $t1 ← dirección de A[i]
                        #          (4 * i + $s3)
lw  $t0 , 0($t1)       # $t0 ← A[i]
```

## 1.3 Operandos del hardware (XIII)

### Operandos MIPS

Nombre	Ejemplo	Comentario
32 registros	\$0, \$1, ..., \$31	Posiciones rápidas para datos
$2^{30}$ palabras de memoria	Memoria[0], Memoria[4], ..., Memoria[4294967292]	Accedidas sólo por instrucciones de transferencia

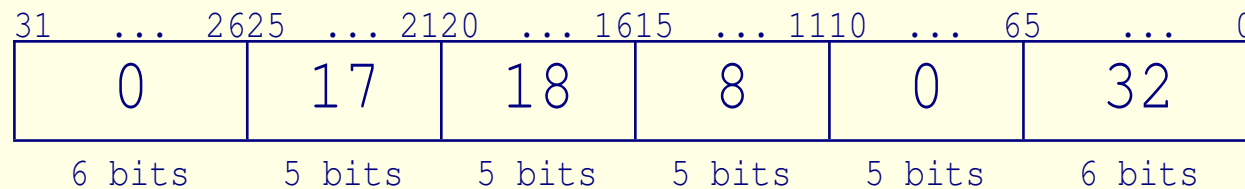
# 1.3 Operandos del hardware (XIV)

## Lenguaje ensamblador MIPS

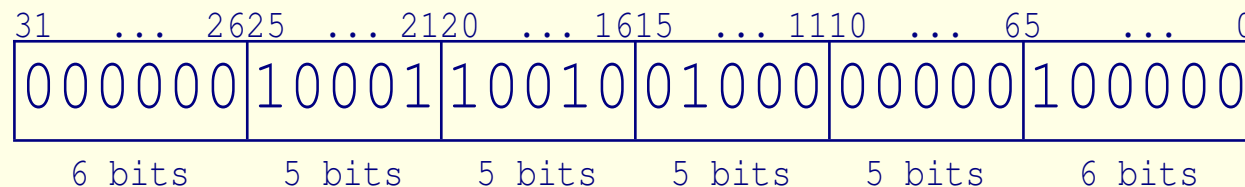
Categoría	Instrucción	Ejemplo	Significado
Aritmética	suma	<b>add</b> \$s1,\$s2,\$s3	$\$s1 \leftarrow \$s2 + \$s3$
	resta	<b>sub</b> \$s1,\$s2,\$s3	$\$s1 \leftarrow \$s2 - \$s3$
Transferencia	carga	<b>lw</b> \$s1,100(\$s2)	$\$s1 \leftarrow \text{Mem}[100 + \$s2]$
	almacena	<b>sw</b> \$s1,100(\$s2)	$\text{Mem}[100 + \$s2] \leftarrow \$s1$

# 1.4 Representación de las instrucciones

- Las instrucciones se encuentran en el computador como un conjunto de señales altas y bajas → números en base 2.
- Los registros  $\$s0$  a  $\$s7$  →  $\$16$  a  $\$23$ .
- Los registros  $\$t0$  a  $\$t7$  →  $\$8$  a  $\$15$ .
- La representación decimal de la instrucción **add**  $\$t0, \$s1, \$s2$ , sería:

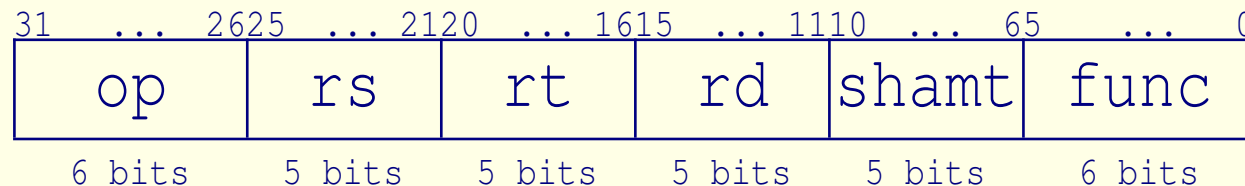


- que en binario es:



## 1.4 Representación de las instrucciones (II)

- Formato de instrucción → distribución de la información de la instrucción.
- La instrucción anterior ocupa 32 bits. Todas las instrucciones del R2000 ocupan 32 bits (PRINCIPIO 1: la simplicidad favorece la uniformidad).
- El formato visto (formato **R**) está formado por los **campos**:



# 1.4 Representación de las instrucciones (III)

## ➤ Problema:

⇒ Hay instrucciones que requieren campos mayores (p.e. **lw**).

## ➤ Soluciones:

⇒ Hacer estas instrucciones más largas

→ instrucciones de distinta longitud.

⇒ Utilizar un formato distinto para estas instrucciones

→ igual longitud pero formatos distintos.

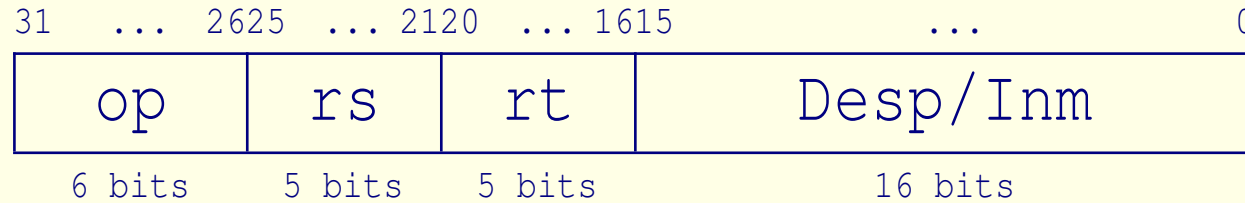
➤ **PRINCIPIO 3:** *Un buen diseño necesita soluciones de compromiso.*

➤ Opción escogida por MIPS: Instrucciones con distintos formatos.

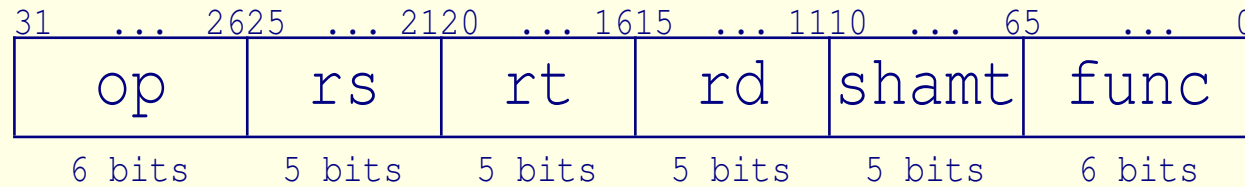


# 1.4 Representación de las instrucciones (IV)

## ► Formato I:



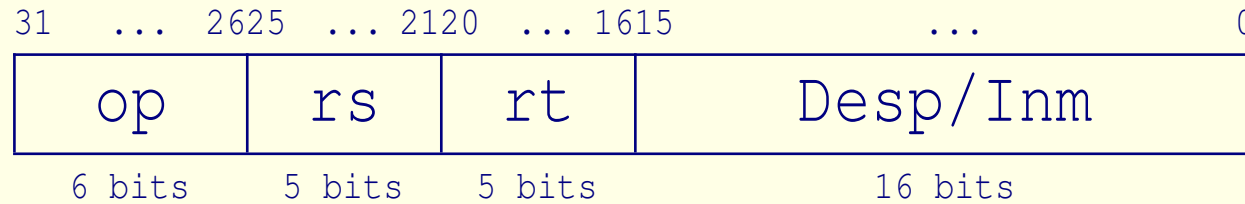
## ► Recordatorio formato R (comparar):



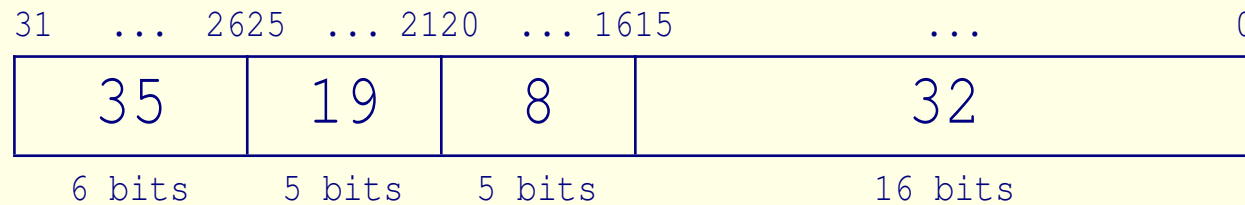
¿Cómo sabrá el procesador si la instrucción leída es de un formato u otro?

# 1.4 Representación de las instrucciones (V)

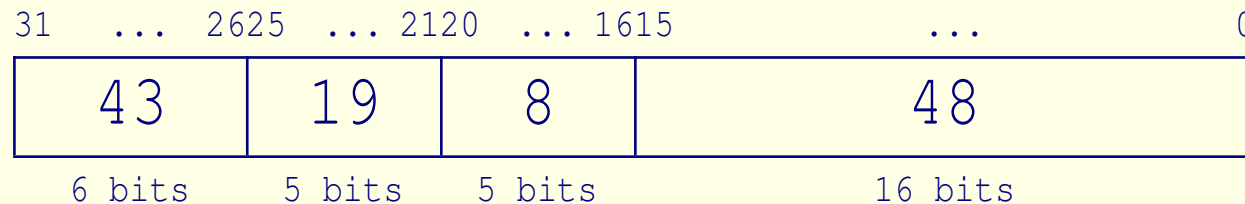
## ► Formato I:



**lw** \$t0,32(\$s3)



**sw** \$t0,48(\$s3)



# 1.5 Instrucciones para la toma de decisiones

- La capacidad de tomar decisiones es lo que distingue a un **computador** de una **calculadora**.
- Instrucciones MIPS de **bifurcación condicional**:
  - ⇒ **beq** registro1,registro2,L1 (salta si igual —*branch if equal*—)
  - ⇒ **bne** registro1,registro2,L1 (salta si no igual —*branch if not equal*—)
- Instrucción MIPS de **bifurcación incondicional**:
  - ⇒ **j** L1 (salta —*jump*—)

## 1.5 Instrucciones para la toma de decisiones (II)

- **Ejercicio 6:** Dado el siguiente código C (con **goto**):

```
    if ( i==j ) goto L1 ;  
    f=g+h ;  
L1 :    f=f-i ;
```

Obtener el equivalente en ensamblador sabiendo que las variables  $f$ ,  $g$ ,  $h$ ,  $i$  y  $j$  se han asociado a los registros  $\$s0$  al  $\$s4$ .

- **Solución:**

## 1.5 Instrucciones para la toma de decisiones (II)

- **Ejercicio 6:** Dado el siguiente código C (con **goto**):

```
    if ( i==j ) goto L1 ;  
    f=g+h ;  
L1 :    f=f-i ;
```

Obtener el equivalente en ensamblador sabiendo que las variables  $f$ ,  $g$ ,  $h$ ,  $i$  y  $j$  se han asociado a los registros  $\$s0$  al  $\$s4$ .

- **Solución:**

```
    beq $s3 , $s4 , L1    # ir a L1 si i==j  
    add $s0 , $s1 , $s2   #  $f \leftarrow g+h$  (no se ejecuta si  $i==j$ )  
L1 :    sub $s0 , $s0 , $s3 #  $f \leftarrow f-i$  (se ejecuta siempre)
```

## 1.5 Instrucciones para la toma de decisiones (III)

- Los compiladores crean saltos y etiquetas (*rótulos*) aunque no estén en el lenguaje de programación.
- **Ejercicio 7:** `if (i==j) f=g+h; else f=g-h;`  
(variables `f`, ..., `j`  $\Rightarrow$  registros del `$s0` al `$s4`.)
- **Solución:**

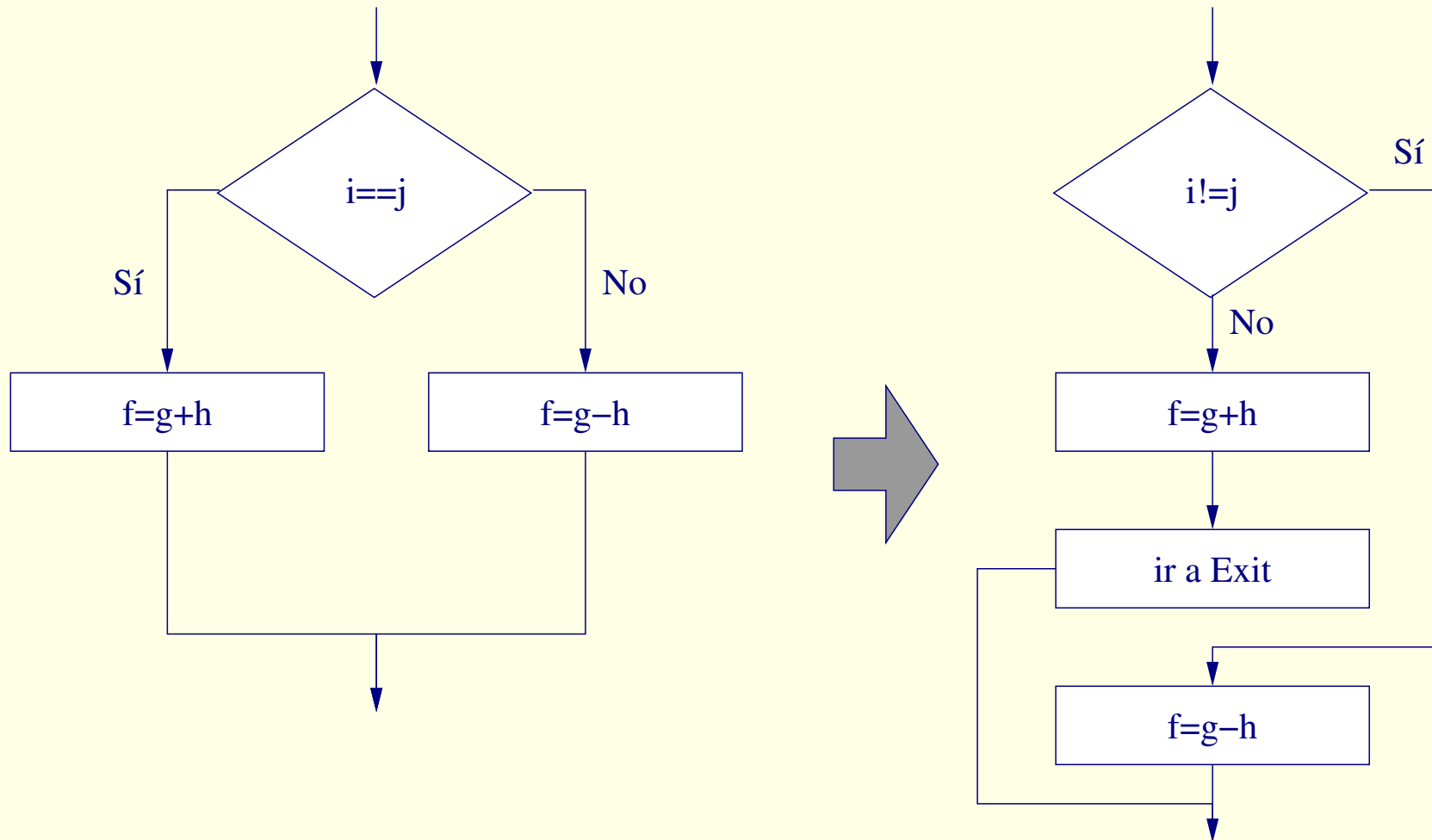
## 1.5 Instrucciones para la toma de decisiones (III)

- Los compiladores crean saltos y etiquetas (*rótulos*) aunque no estén en el lenguaje de programación.
- **Ejercicio 7:** **if** ( $i==j$ )  $f=g+h$ ; **else**  $f=g-h$ ;  
(variables  $f, \dots, j \Rightarrow$  registros del  $\$s0$  al  $\$s4$ .)
- **Solución:**

```
bne $s3, $s4, ELSE # ir a ELSE si  $i \neq j$   
add $s0, $s1, $s2 #  $f \leftarrow g+h$  (saltada si  $i \neq j$ )  
j EXIT # ir a EXIT  
ELSE: sub $s0, $s1, $s2 #  $f \leftarrow g-h$  (saltada si  $i == j$ )  
EXIT: ...
```

# 1.5 Instrucciones para la toma de decisiones (IV)

- La toma de decisiones se realiza mediante saltos:





## 1.5 Instrucciones para la toma de decisiones (V)

- También son importantes para **iterar** una computación.
- **Ejercicio 8:** Convertir el siguiente código a ensamblador:

```
Bucle : g=g+A[ i ];  
        i=i+j ;  
        if ( i != h ) goto Bucle ;
```

Sabiendo que:

- g, h, i y j se han asociado a los registros \$s1 al \$s4.
- \$s5 ← dirección de comienzo de A.

# 1.5 Instrucciones para la toma de decisiones (VI)

► **Resumen ejercicio 8:** ( $g, h, i$  y  $j \leftarrow \$s1$  al  $\$s4$ ;  $\$s5 \leftarrow \text{dir. A}$ ) :

```
Bucle : g=g+A[ i ];  
        i=i+j ;  
        if ( i != h ) goto Bucle ;
```

```
Bucle : add   $t1 , $s3 , $s3      # $t1 ← 2* i  
        add   $t1 , $t1 , $t1     # $t1 ← 4* i  
        add   $t1 , $t1 , $s5     # $t1 ← dirección de A[ i ]  
        lw    $t0 , 0( $t1 )      # $t0 ← A[ i ]  
        add   $s1 , $s1 , $t0     # g ← g+A[ i ]  
        add   $s3 , $s3 , $s4     # i ← i+j  
        bne   $s3 , $s2 , Bucle   # ir a Bucle si i!=h
```

## 1.5 Instrucciones para la toma de decisiones (VII)

- Un programador no escribe bucles con *gotos*, el compilador traduce los bucles tradicionales a saltos como los vistos.
- **Ejercicio 9:**

```
while ( nota [ i ] == k )  
    i = i + j ;
```

Obtener el equivalente en ensamblador sabiendo que:

➤  $i, j, y k \leftarrow \$s3 \text{ al } \$s5.$

➤  $\$s6 \leftarrow$  Dirección de comienzo del vector `nota`.

# 1.5 Instrucciones para la toma de decisiones (VIII)

- $(i, j, y k \leftarrow \$s3 \text{ al } \$s5; \$s6 \leftarrow \text{Dir. comienzo vector nota.})$

```
while (nota[i]==k) i=i+j;
```

- **Solución:**

```
Bucle : add $t1 , $s3 , $s3      # $t1 ← 2 * i
        add $t1 , $t1 , $t1     # $t1 ← 4 * i
        add $t1 , $t1 , $s6     # $t1 ← dirección de nota[i]
        lw  $t0 , 0($t1)        # $t0 ← nota[i]
        bne $t0 , $s5 , Fin     # ir a Fin si nota[i] != k
        add $s3 , $s3 , $s4     # i ← i + j
        j   Bucle              # ir a Bucle

Fin :   ...
```

## 1.5 Instrucciones para la toma de decisiones (IX)

- **beq** y **bne** permiten tomar una decisión en función de si son iguales o no los valores almacenados en dos registros.
- ¿Otras comparaciones? ¿ $\$s0 < \$s1$ ?
- **slt** (*set less than*) compara dos registros y pone a 1 un tercero si el primero es menor que el segundo (a 0 en caso contrario). Ej:

```
slt $t0 , $s0 , $s1 # si  $\$s0 < \$s1$  ,  $\$t0 \leftarrow 1$  , si no ,  $\$t0 \leftarrow 0$ 
```

# 1.5 Instrucciones para la toma de decisiones (X)

- Los compiladores utilizan
  - ⇒ las instrucciones **slt**, **beq** y **bne** y
  - ⇒ el hecho de que el registro `$zero` tiene siempre el valor 0 para generar **todas** las condiciones relativas.
- **Ejercicio 10:** `if (a<b) goto Menor;` (variables `a`, `b` → `$s0` y `$s1`.)
- **Solución:**

# 1.5 Instrucciones para la toma de decisiones (X)

- Los compiladores utilizan
  - ⇒ las instrucciones **slt**, **beq** y **bne** y
  - ⇒ el hecho de que el registro `$zero` tiene siempre el valor 0 para generar **todas** las condiciones relativas.
- **Ejercicio 10:** `if (a<b) goto Menor;` (variables `a`, `b` → `$s0` y `$s1`.)
- **Solución:**

```
slt $t0 , $s0 , $s1      # si a<b entonces $t0 ← -1  
bne $t0 , $zero , Menor  # ir a Menor si $t0 != 0 (a<b)
```

## 1.5 Instrucciones para la toma de decisiones (XI)

- El ejemplo anterior implementa un **salto si menor que**:

```
slt $t0 , $s0 , $s1
```

```
bne $t0 , $zero , Menor    # ir a Menor si $s0 < $s1
```

- El **ensamblador** proporciona una **pseudo-instrucción**, **blt** (*branch if less than*), que traduce por las dos instrucciones anteriores.
- El procesador no reconoce **blt** y similares: sólo sabe de **slt**, **beq** y **bne**.



## 1.5 Instrucciones para la toma de decisiones (XII)

- Otra instrucción de toma de decisiones común en lenguajes de alto nivel es **switch** que junto con **case** permite seleccionar una de varias alternativas. Por ejemplo:

```
switch (k) {  
    case 0:  f=i+j; break; /* k==0 */  
    case 1:  f=g+h; break; /* k==1 */  
    case 2:  f=g-h; break; /* k==2 */  
}
```

- En lenguaje máquina esta construcción se puede resolver por medio del equivalente a varios **if–then–else** encadenados.

## 1.5 Instrucciones para la toma de decisiones (XIII)

- ▶ A veces, se puede resolver de una forma más eficiente:
  - ⇒ Se crea un vector con las direcciones de comienzo de cada caso.
  - ⇒ La variable de la que depende la selección se utiliza como índice del vector anterior → se obtiene la dirección de salto.
- ▶ **jr** (**jump register**) salta a la dirección especificada en el registro indicado:

```
jr $t0 # salta a la dir. indicada por $t0
```

# 1.5 Instrucciones para la toma de decisiones (XIV)

```
switch (k) {  
  case 0: f=i+j; break;  
  case 1: f=g+h; break;  
  case 2: f=g-h; break;
```

$\$s0$  al  $\$s5 \leftarrow f, g, h, i, j, k$

TablaDeSalto  $\leftarrow [L0, L1, L2]$

$\$t4 \leftarrow \text{Dir. com. TablaDeSalto}$

$\$t2 \leftarrow 3$

```
Switch : slt    $t3, $s5, $zero  
         bne    $t3, $zero, Fin  
         slt    $t3, $s5, $t2  
         beq    $t3, $zero, Fin  
         add    $t1, $s5, $s5  
         add    $t1, $t1, $t1  
         add    $t1, $t1, $t4  
         lw     $t0, 0($t1)  
         jr     $t0  
L0 :     add    $s0, $s3, $s4  
         j      Fin  
L1 :     add    $s0, $s1, $s2  
         j      Fin  
L2 :     sub    $s0, $s1, $s2  
Fin :     ...
```

# 1.5 Instrucciones para la toma de decisiones (XV)

## Lenguaje ensamblador MIPS

Categoría	Instrucción	Ejemplo	Significado
Aritmética	suma	<b>add</b> \$s1,\$s2,\$s3	$\$s1 \leftarrow \$s2 + \$s3$
	resta	<b>sub</b> \$s1,\$s2,\$s3	$\$s1 \leftarrow \$s2 - \$s3$
Transferencia	carga palabra	<b>lw</b> \$s1,100(\$s2)	$\$s1 \leftarrow \text{Mem}[\$s2 + 100]$
	almacena palabra	<b>sw</b> \$s1,100(\$s2)	$\text{Mem}[\$s2 + 100] \leftarrow \$s1$
Salto condicional	salta si igual	<b>beq</b> \$s1,\$s2,L	Si $\$s1 == \$s2$ ir a L
	salta si no igual	<b>bne</b> \$s1,\$s2,L	Si $\$s1 \neq \$s2$ ir a L
	inic. si menor que	<b>slt</b> \$s1,\$s2,\$s3	Si $\$s2 < \$s3$ , $\$s1 \leftarrow 1$ ; si no, $\leftarrow 0$
Salto incondicional	salta	<b>j</b> 10000	Ir a 10000
	salta con registro	<b>jr</b> \$t1	Ir a la dirección indicada en \$t1

## 1.6 Procedimientos (o funciones)

- Un procedimiento es una herramienta para estructurar el código:
  - ⇒ Facilita su comprensión.
  - ⇒ Permite la reutilización de código.
- Los procedimientos permiten al programador concentrarse en una parte del código que:
  1. Recibe unos parámetros.
  2. Realiza su cometido.
  3. Devuelve el resultado.

## 1.6 Procedimientos (o funciones) (II)

- Para ejecutar un procedimiento es necesario:
  1. Situar los parámetros de entrada en un lugar adecuado.
  2. Transferir el control al procedimiento.
  3. Adquirir los recursos de almacenamiento necesarios para el procedimiento.
  4. Realizar la tarea deseada.
  5. Situar el valor del resultado en un lugar adecuado.
  6. Retornar el control al punto de origen.

## 1.6 Procedimientos (o funciones) (III)

- Registros para el paso de parámetros:
  - ⇒ De entrada:  $\$a0$ – $\$a3$ .
  - ⇒ De salida:  $\$v0$ – $\$v1$ .
- Registro para la dirección de retorno:
  - ⇒  $\$ra$  (*return address*).

## 1.6 Procedimientos (o funciones) (IV)

- ▶ La instrucción `jal Address` (*jump and link*) se encarga de:
  1. Almacenar en `$ra` la dirección de vuelta ( $PC_{actual}+4$ ).
  2. Saltar a la dirección de comienzo del procedimiento.
- ▶ ¿Qué instrucción podemos utilizar para volver desde el procedimiento?



## 1.6 Procedimientos (o funciones) (IV)

➤ La instrucción `jal Address` (*jump and link*) se encarga de:

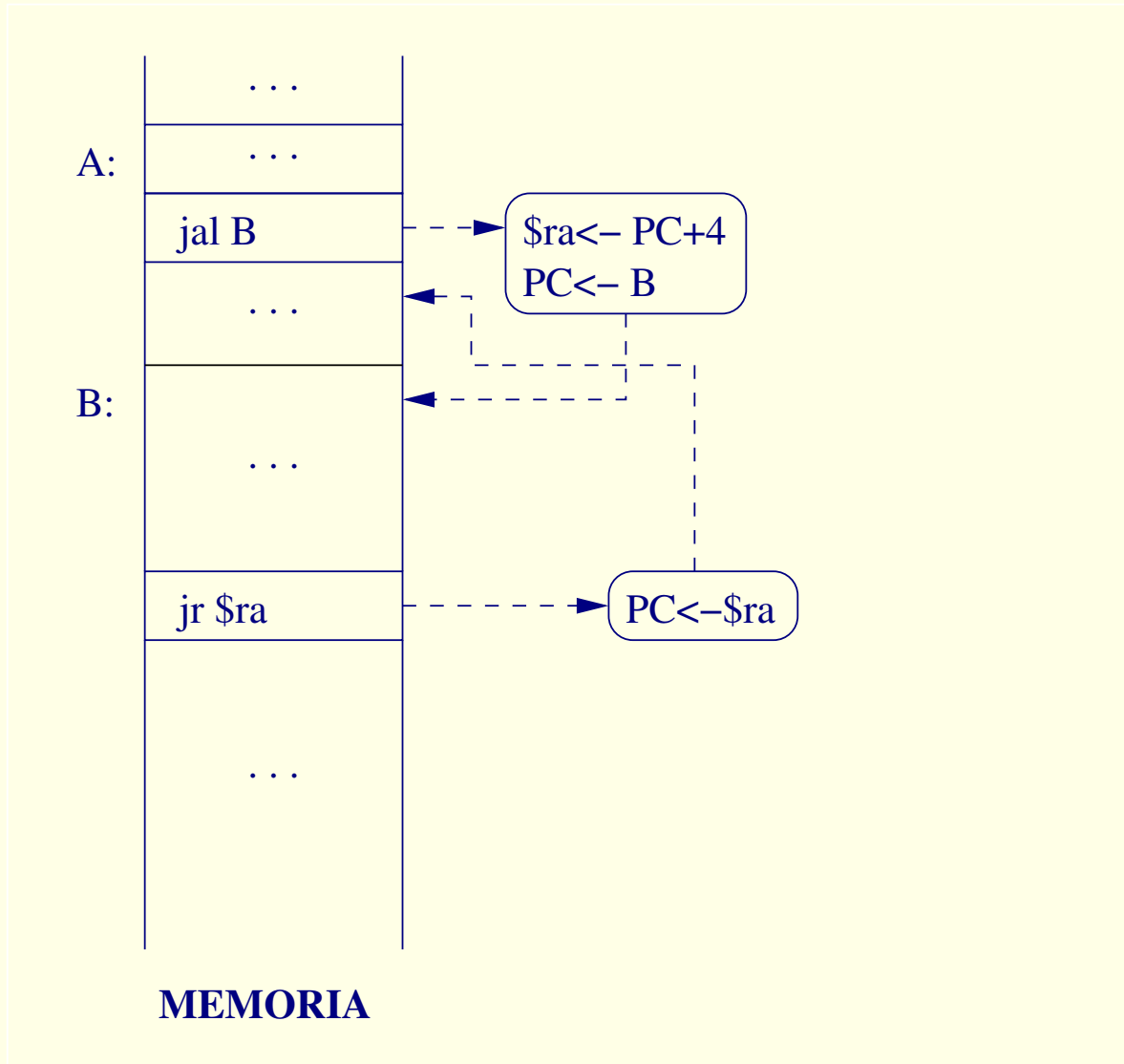
1. Almacenar en `$ra` la dirección de vuelta ( $PC_{actual}+4$ ).
2. Saltar a la dirección de comienzo del procedimiento.

➤ ¿Qué instrucción podemos utilizar para volver desde el procedimiento?

⇒ `jr $ra`

# 1.6 Procedimientos (o funciones) (V)

- La llamada y vuelta de un procedimiento queda:



## 1.6 Procedimientos (o funciones) (VI)

➤ Ejemplo de procedimiento:

```
int ejemplo(int g, int h, int i, int j)
{
    int f;
    f=(g+h)-(i+j);
    return f;
}
```

⇒ \$a0 al \$a3 ← g, h, i, j

⇒ \$v0 ← f

## 1.6 Procedimientos (o funciones) (VII)

- Si el compilador decide resolver el problema de la siguiente forma:

$t0 = g + h$

$t1 = i + j$

$f = t0 - t1$

no tendrá suficiente con los registros  $\$a0-\$a3$ ,  $\$v0$  y  $\$v1$ .

- Cualquier registro utilizado por el procedimiento debe ser restaurado a su valor original antes de retornar el control al programa invocador.

- ⇒ El valor original de dichos registros debe guardarse en memoria.

- ⇒ Estructura de datos idónea: la pila.

## 1.6 Procedimientos (o funciones) (VIII)

- Una **pila**:
  - ⇒ Permite almacenar y recuperar información.
  - ⇒ Es una cola LIFO (el último en entrar será el primero en salir).
- El registro  $\$sp$  (*stack pointer*) indica el tope de la pila.
- El uso de pilas es tan habitual que sus operaciones tienen nombres propios: apilar (*push*) y desapilar (*pop*).
- La pila «crece» de direcciones superiores a inferiores:
  - ⇒ Para apilar se resta del valor de  $\$sp$  el número adecuado.
  - ⇒ Para desapilar se añade al valor de  $\$sp$  el número adecuado.

## 1.6 Procedimientos (o funciones) (IX)

➤ Ejemplo de procedimiento (otra vez):

```
int ejemplo(int g, int h, int i, int j)
{
    int f;
    f=(g+h)-(i+j);
    return f;
}
```

➤ \$a0 al \$a3 ← g, h, i, j

➤ \$v0 ← f

➤ Dentro del procedimiento: \$t0 ← g+h, \$t1 ← i+j

## 1.6 Procedimientos (o funciones) (X)

```
ejemplo : sub $sp,$sp, 8    # reserva espacio para 2 elementos
sw  $t1,4($sp)    # apila $t1
sw  $t0,0($sp)    # apila $t0

add $t0,$a0,$a1    #  $t0 \leftarrow g+h$ 
add $t1,$a2,$a3    #  $t1 \leftarrow i+j$ 

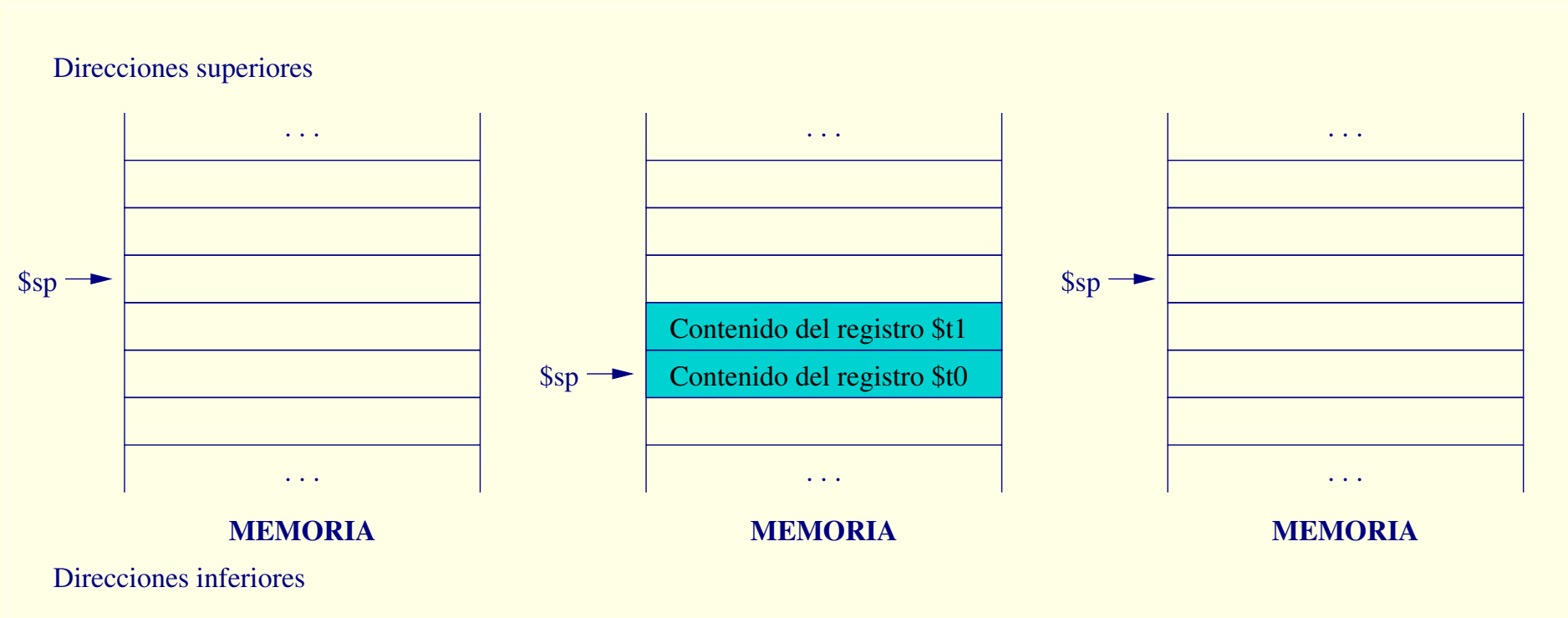
sub $v0,$t0,$t1    #  $v0 \leftarrow (g+h) - (i+j)$ 

lw  $t0,0($sp)    # restaura $t0
lw  $t1,4($sp)    # restaura $t1
add $sp,$sp, 8    # elimina de la pila 2 elementos

jr $ra            # retorno a la rutina invocadora
```

# 1.6 Procedimientos (o funciones) (XI)

- Estado de la pila antes, durante y después de la llamada al procedimiento:



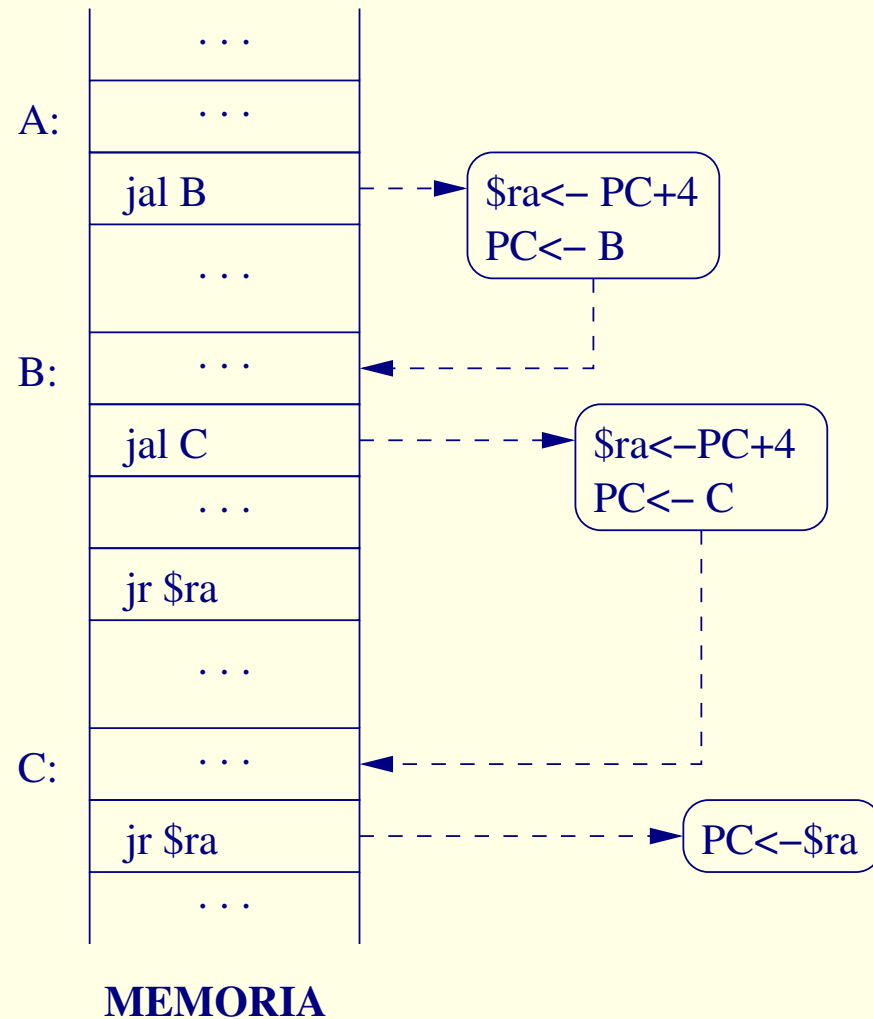


## 1.6 Procedimientos (o funciones) (XII)

- Para evitar tener que guardar y restaurar los registros utilizados en un procedimiento, MIPS diferencia entre dos clases de registros:
  - ⇒  $\$t0-\$t9$ : 10 registros temporales que no tienen por qué ser preservados por el programa invocado.
  - ⇒  $\$s0-\$s7$ : 8 registros salvados que deben ser preservados.
- En el procedimiento anterior no era necesario preservar los registros utilizados ( $\$t0$  y  $\$t1$ ).

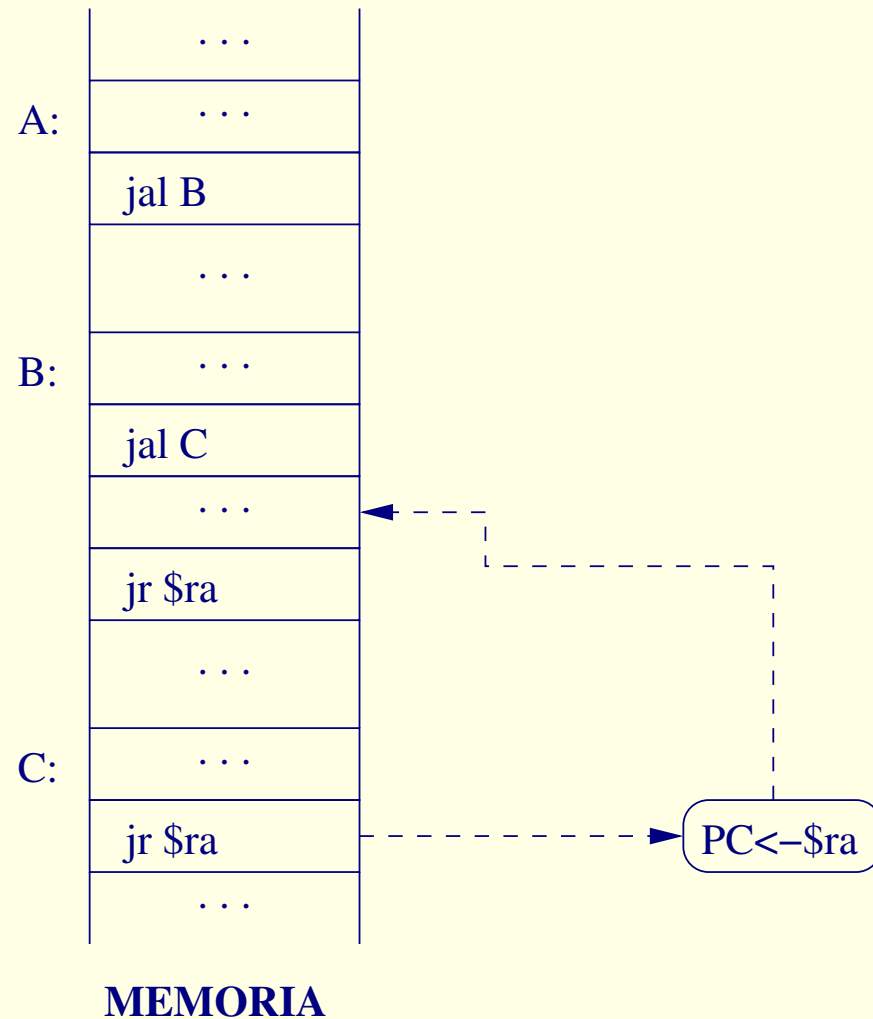
# 1.6 Procedimientos (o funciones) (XIII)

- ¿Qué ocurre si un procedimiento llama a su vez a otro?



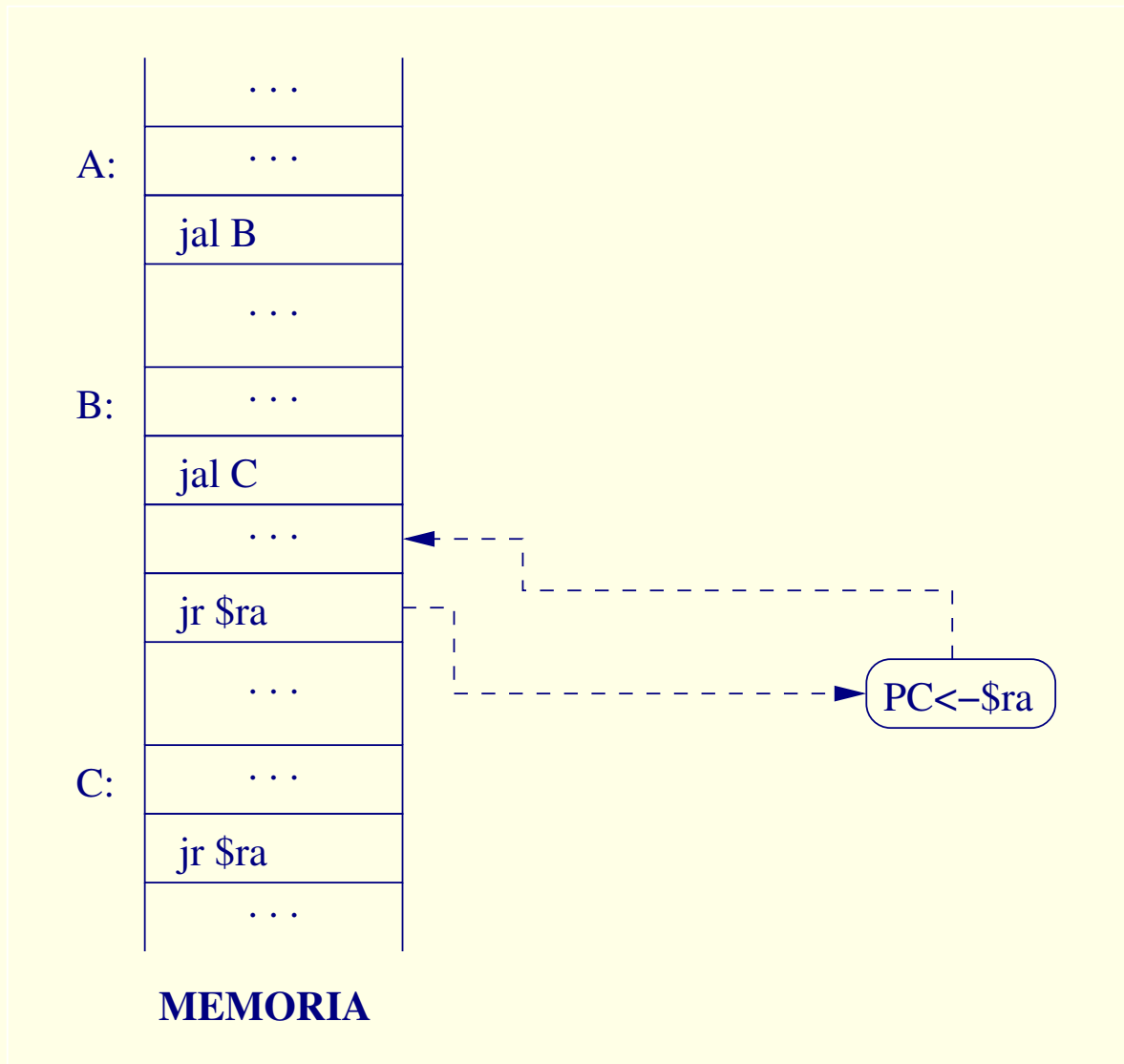
# 1.6 Procedimientos (o funciones) (XIV)

- ¿Qué ocurre si un procedimiento llama a su vez a otro?



# 1.6 Procedimientos (o funciones) (XV)

- ¿Qué ocurre si un procedimiento llama a su vez a otro?

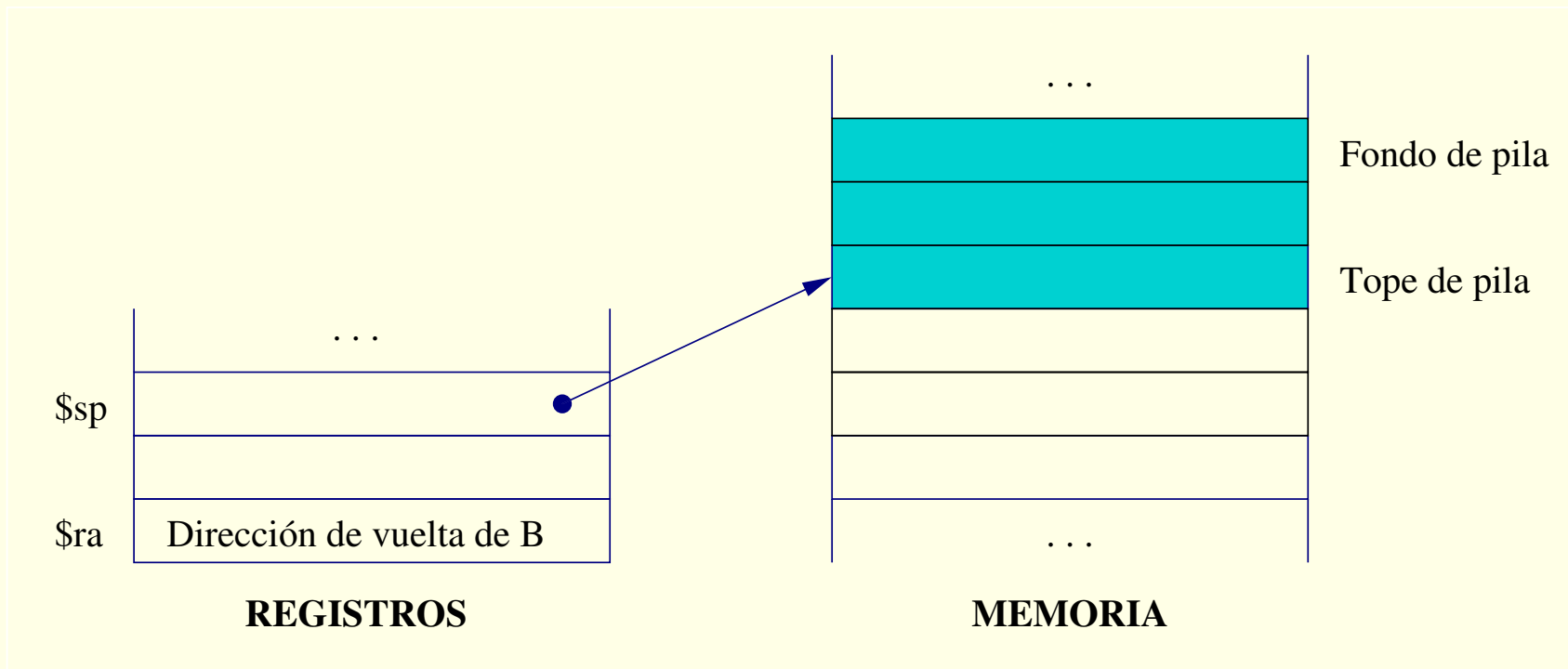


## 1.6 Procedimientos (o funciones) (XVI)

- ▶ ¿Qué ocurre si un procedimiento llama a su vez a otro?
  - ⇒ Que la dirección de vuelta del primero se pierde
    - ⇒ Es necesario almacenar el contenido de  $\$ra$  en memoria.
  - ⇒ No podemos utilizar una dirección fija de memoria
    - ⇒ Si una función se llama a si misma, tenemos el mismo problema.
- ▶ Solución: utilizar una **pila**.
  - ⇒ Antes de llamar a un procedimiento: apilar  $\$ra$ .
  - ⇒ Antes de volver: desapilar  $\$ra$ .

## 1.6 Procedimientos (o funciones) (XVII)

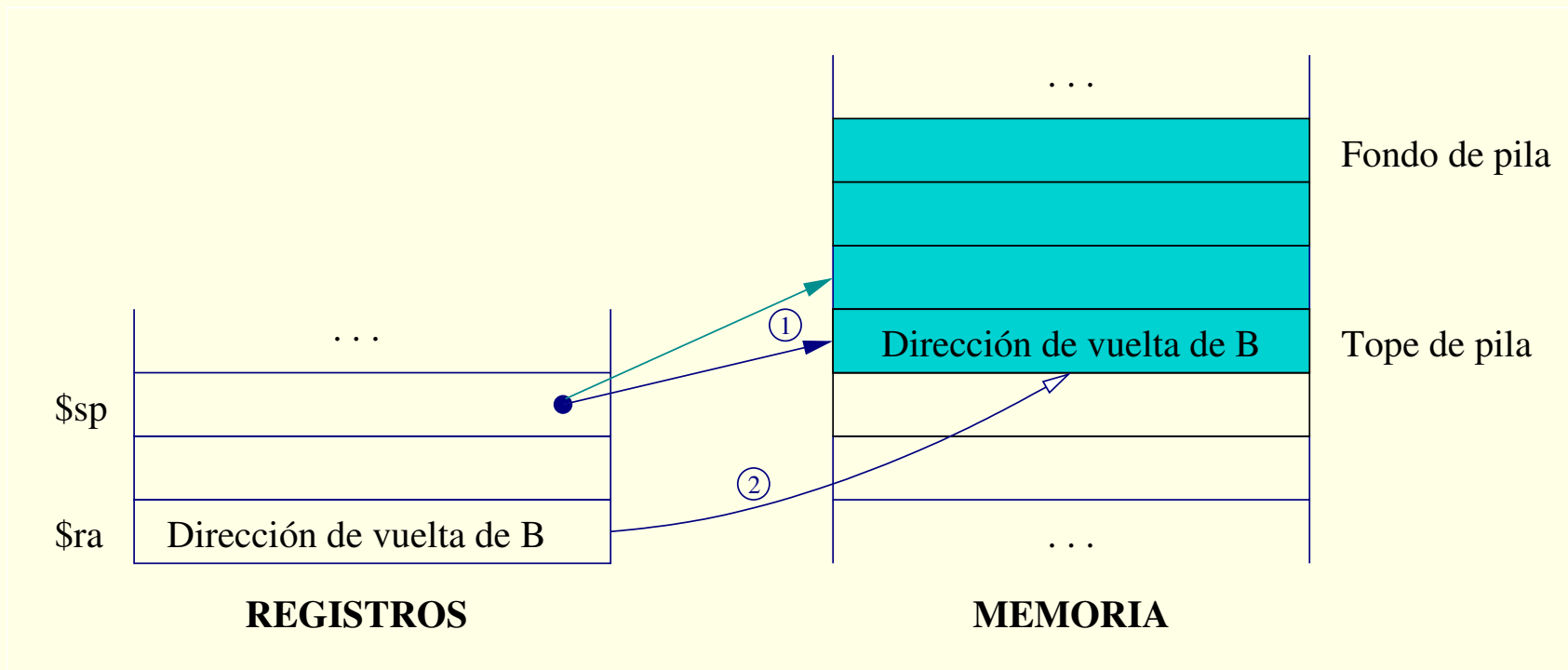
- Utilización de la pila para almacenar  $\$ra$  cuando sea necesario.
  - 1 Justo después de que A haya llamado a B:



# 1.6 Procedimientos (o funciones) (XVIII)

- Utilización de la pila para almacenar  $\$ra$  cuando sea necesario.

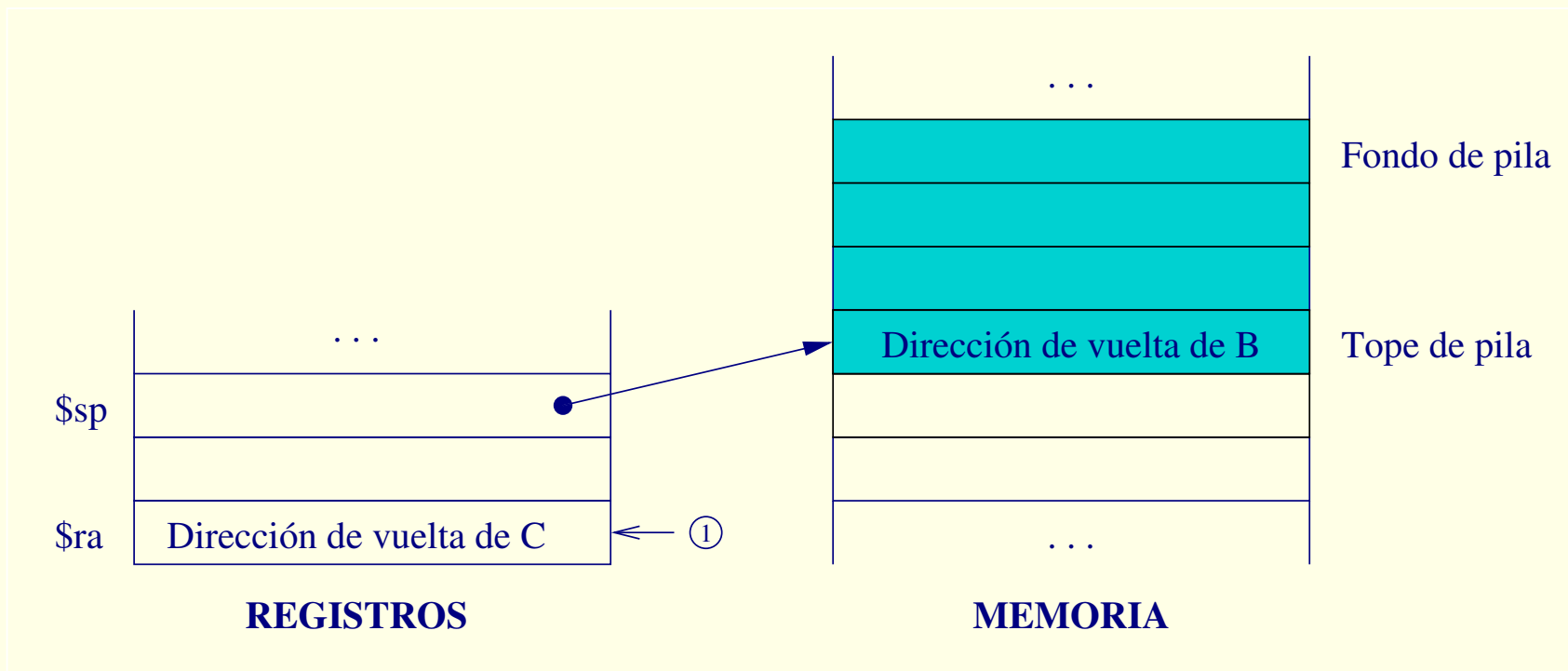
2 Antes de que B llame a C:



# 1.6 Procedimientos (o funciones) (XIX)

➤ Utilización de la pila para almacenar  $\$ra$  cuando sea necesario.

3 Justo después de que B haya llamado a C:

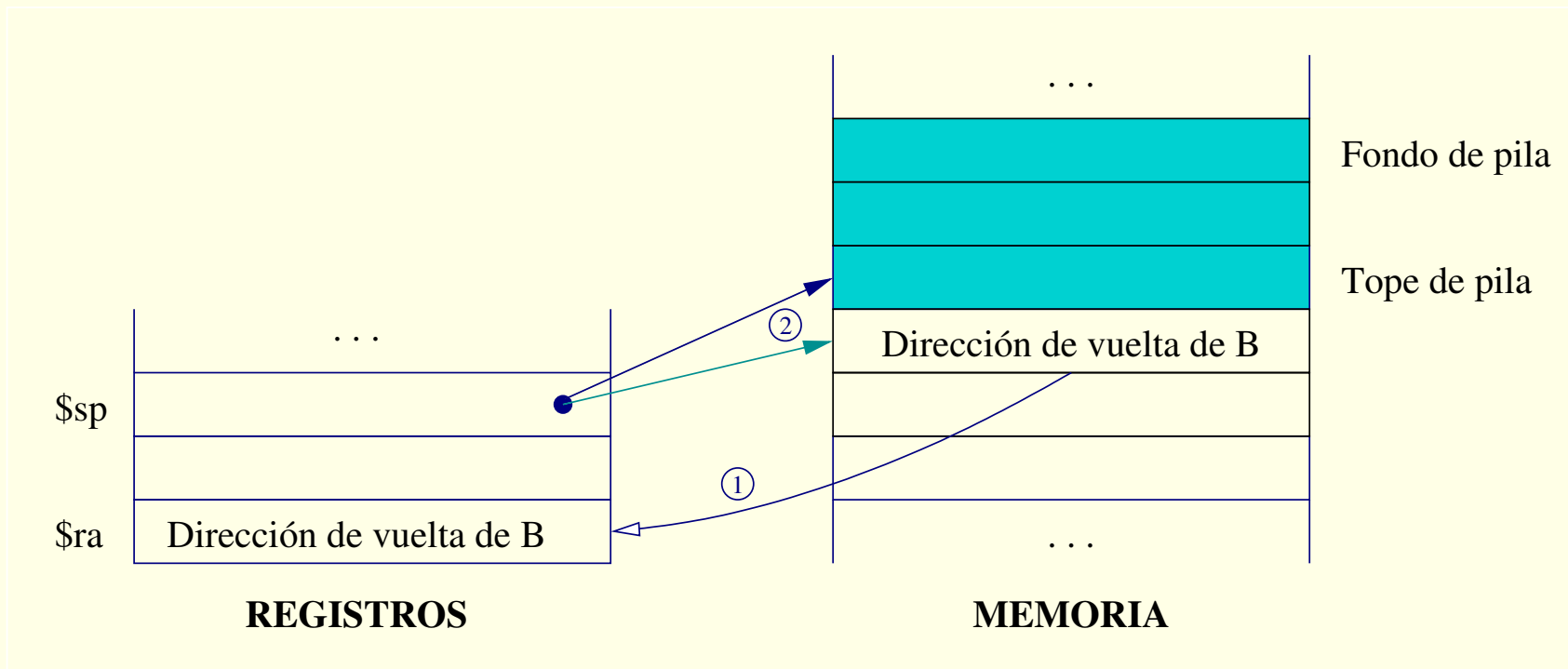




# 1.6 Procedimientos (o funciones) (XX)

- Utilización de la pila para almacenar  $\$ra$  cuando sea necesario.

4 Antes de la vuelta de B:



## 1.6 Procedimientos (o funciones) (XXI)

```
A:  ...
    jal B           # llama a B y guarda la dirección de vuelta en $ra
    ...
B:  ...
    sub $sp,$sp,4   # ajusta el puntero de pila para hacer sitio
    sw  $ra,0($sp)  # guarda la dirección de vuelta de B en la pila
    jal C           # llama a C y guarda la dirección de vuelta en $ra
    ...
    lw  $ra,0($sp)  # restaura la dirección de vuelta de B
    add $sp,$sp,4   # ajusta el puntero de pila para sacar elemento
    jr  $ra         # vuelve a la rutina que invocó a B
    ...
C:  ...
    jr  $ra         # vuelve a la rutina que invocó a C
```

# 1.6 Procedimientos (o funciones) (XXII)

Categoría	Instrucción	Ejemplo	Significado
Aritmética	suma	<b>add</b> \$s1,\$s2,\$s3	$\$s1 \leftarrow \$s2 + \$s3$
	resta	<b>sub</b> \$s1,\$s2,\$s3	$\$s1 \leftarrow \$s2 - \$s3$
Transferencia	carga palabra	<b>lw</b> \$s1,100(\$s2)	$\$s1 \leftarrow \text{Mem}[\$s2 + 100]$
	almacena palabra	<b>sw</b> \$s1,100(\$s2)	$\text{Mem}[\$s2 + 100] \leftarrow \$s1$
Salto condicional	salta si igual	<b>beq</b> \$s1,\$s2,L	Si $\$s1 == \$s2$ ir a L
	salta si no igual	<b>bne</b> \$s1,\$s2,L	Si $\$s1 \neq \$s2$ ir a L
	inic. si menor que	<b>slt</b> \$s1,\$s2,\$s3	Si $\$s2 < \$s3$ , $\$s1 \leftarrow 1$ ; si no, $\leftarrow 0$
Salto incondicional	salta	<b>j</b> 10000	Ir a 10000
	salta con registro	<b>jr</b> \$ra	Ir a la dirección indicada por \$ra
	salta y enlaza	<b>jal</b> 1000	$\$ra \leftarrow PC + 4$ ; $\$PC \leftarrow 1000$

## 1.7 Otros estilos de direccionamiento

- ▶ MIPS proporciona dos formas más de acceder a los datos para:
  - ⇒ acceder más rápido a pequeñas constantes.
  - ⇒ hacer más eficientes los saltos.

## 1.7.1 Constantes u operandos inmediatos

- Se utilizan con mucha frecuencia: incrementar el índice de un bucle, siguiente elemento de un vector, ajustar la pila...

(**gcc**: 52 %, **SPICE**: 69 % de operaciones aritméticas con ctes.)

- **PRINCIPIO 4**: *Hacer rápido el caso más común.*

⇒ Instrucciones con datos inmediatos: **add** → **addi**, **slt** → **slti**, etc.

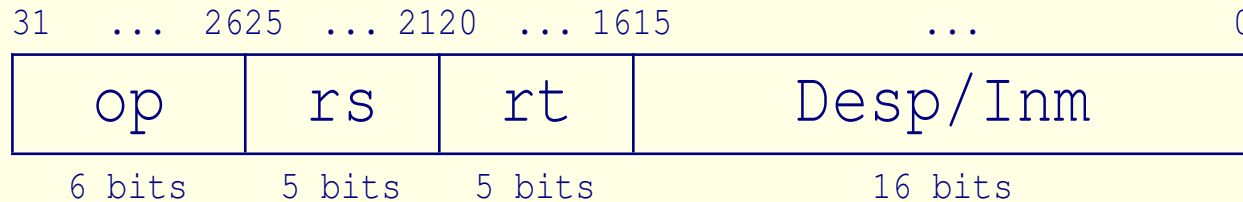
```
addi $sp, $sp, 4 # $sp ← $sp + 4
```

```
addi $sp, $sp, -4 # $sp ← $sp - 4
```

```
slti $t0, $s0, 10 # si $s0 < 10, $t0 ← 1; si no, ← 0
```

## 1.7.1 Constantes u operandos inmediatos (II)

- Estas instrucciones se codifican con el **Formato I**:



→ Para la constante se utiliza un campo de 16 bits

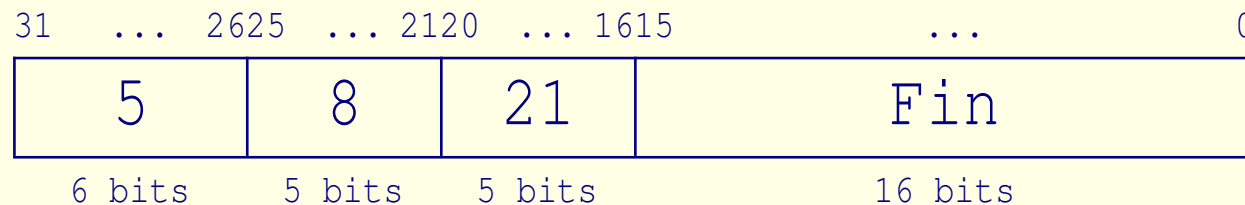
- ¿Podemos emplear constantes de más de 16 bits?  
Sí, utilizando un registro y la instrucción **lui** (*load upper immediate*).
- Ejemplo: cargar la constante `0xABCD0123` en `$s0`:

```
lui    $s0, 0xABCD          # $s0 ← 0xABCD0000
addi   $s0, $s0, 0x0123     # $s0 ← 0xABCD0123
```

## 1.7.2 Direccionamiento en saltos

- Las instrucciones de salto condicional utilizan el **Formato I**:

**bne** \$t0,\$s3,Fin



- **Fin** se almacena sólo con 16 bits:
  - ➡ Si las direcciones de salto fueran de 16 bits → los programas estarían limitados a los primeros 64KB ( $2^{16}$ ) de memoria.
  - ➡ Solución: sumar a la dirección de salto el contenido de un registro  $\Rightarrow PC \leftarrow \text{registro} + \text{Fin}$ .
  - ➡ ¿Qué registro? *El PC*.


## 1.7.2 Direccionamiento en saltos (II)

- El modo de direccionamiento utilizado por MIPS para las instrucciones de salto condicional es, por tanto, el *relativo al PC*:

$$PC \leftarrow PC + Fin$$

- Puesto que durante la primera fase del ciclo de instrucción se incrementa el PC en 4, en realidad:

$$PC \leftarrow (PC+4) + Fin$$

 La información que realmente se almacena en el campo `Desp/Inm` se optimiza aún más (se verá más adelante).




## 1.7.2 Direccionamiento en saltos (III)

- Las instrucciones de salto incondicional (**j** y **jal**) utilizan un formato de instrucción propio: el **formato J**.



- De esta forma, la dirección de salto se especifica con 26 bits.

 Una dirección consta de 32 bits, faltan 6, ¿de dónde salen? (se verá más adelante).

## 1.7.2 Direccionamiento en saltos (IV)

### ► Ejercicio 11:

```
        lui    $s0, 0x1001
Bucle : lw     $t0, 0($s0)
        bne   $t0, $s1, Fin
        addi  $s0, $s0, 4
        j     Bucle
```

Fin :

¿Cómo codificarías las instrucciones **bne** y **j** sabiendo:

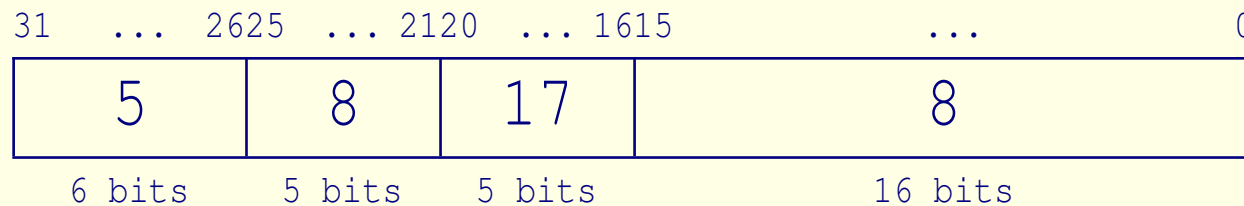
- ⇒ Que los códigos de operación de **bne** y **j** son: 5 y 2.
- ⇒ Que el programa anterior comienza en la posición 80.000 de memoria.

## 1.7.2 Direccionamiento en saltos (V)

➤ **Solución:** (según lo visto hasta ahora, ¡no es correcta!)

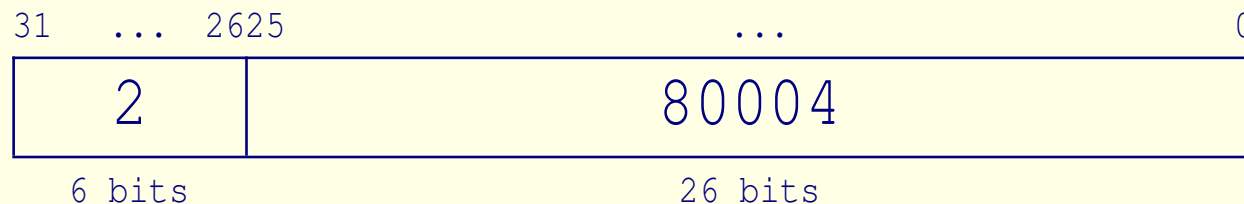
⇒ Entre la instrucción siguiente a **bne** y la etiquetada con `Fin` hay 8 posiciones de memoria, por tanto:

**bne** \$st0,\$s1,Fin



⇒ El programa comienza en 80.000, por tanto:

**j** Bucle



## 1.7.2 Direccionamiento en saltos (VI)

### ➤ EN REALIDAD:

Puesto que todas las instrucciones ocupan 4 bytes, las instrucciones de salto que utilizan el **formato I** almacenan en el campo Desp/Inm:

- ➡ No el número de posiciones de memoria que se deben saltar.
- ➡ Si no, el **número de instrucciones** que se deben saltar.
  - ➡ Se ganan 2 bits: el rango del salto se incrementa de  $2^{16}$  a  $2^{18}$ :
    - ◇ De 65.536 bytes (64KB) se pasa a 262.144 (256KB).
    - ◇ O, de 16.384 instrucciones se pasa a 65.536.

## 1.7.2 Direccionamiento en saltos (VII)

### ➤ Ejercicio 12:

En el anterior ejercicio, entre la instrucción siguiente a **bne** y la etiquetada con `Fin` habían 8 posiciones de memoria.

¿Qué valor se almacenará en el campo `Desp/Inm` cuando se codifique la instrucción **bne**?

### ➤ Solución:

## 1.7.2 Direccionamiento en saltos (VII)

### ➤ Ejercicio 12:

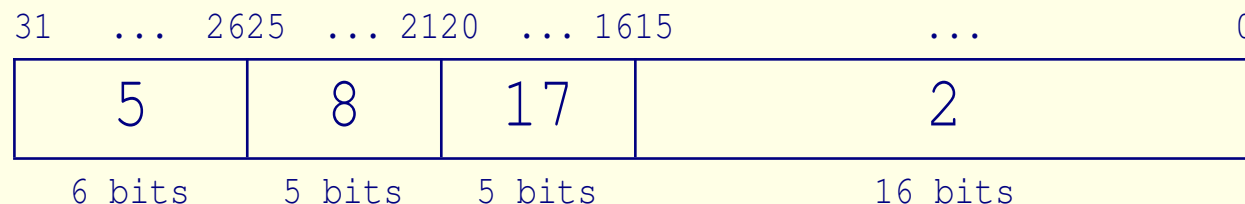
En el anterior ejercicio, entre la instrucción siguiente a **bne** y la etiquetada con `Fin` habían 8 posiciones de memoria.

¿Qué valor se almacenará en el campo `Desp/Inm` cuando se codifique la instrucción **bne**?

### ➤ Solución:

2, ya que  $8/4 = 2$  (hay que saltar 2 instrucciones).

**bne** \$t0,\$s1,Fin



## 1.7.2 Direccionamiento en saltos (VIII)

### ➤ EN REALIDAD:

Las instrucciones de salto que utilizan el **formato J** también se refieren a palabras en lugar de a posiciones de memoria: habrá que dividir por 4 la dirección de memoria antes de almacenarla en el campo correspondiente.

- Por lo tanto, tenemos 28 bits de la dirección:
  - ➡ 26 bits codificados en la instrucción.
  - ➡ +2 bits de menor peso a 0 (instrucciones en múltiplos de 4).
- ¿De dónde salen los 4 de mayor peso? del PC actual (PC+4).
- Rango de **j** y **jal**: 256MB ( $2^{28}$ ) —espacio dir.: 4GB ( $2^{32}$ )—.

## 1.7.2 Direccionamiento en saltos (IX)

### ➤ **Ejercicio 13:**

En un ejercicio anterior, la instrucción **j** Bucle saltaba a la dirección de memoria 80.004 (etiquetada con Bucle), ¿qué valor se almacenará en el campo Dirección cuando se codifique dicha instrucción?

### ➤ **Solución:**



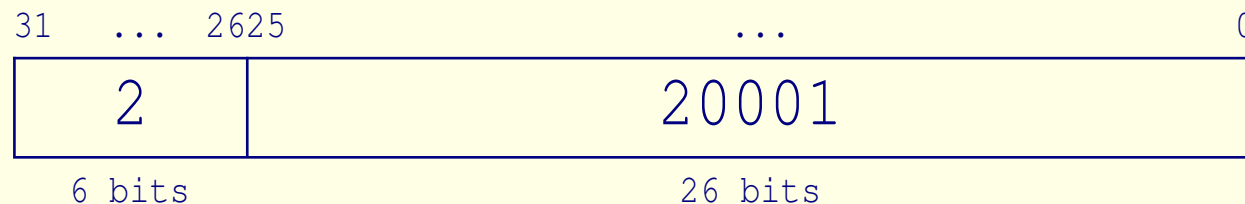
## 1.7.2 Direccionamiento en saltos (IX)

### ➤ Ejercicio 13:

En un ejercicio anterior, la instrucción **j** Bucle saltaba a la dirección de memoria 80.004 (etiquetada con Bucle), ¿qué valor se almacenará en el campo Dirección cuando se codifique dicha instrucción?

➤ **Solución:** 20.001, ya que  $80.004/4 = 20.001$ .

### **j** Bucle



## 1.8 Consideraciones adicionales de la sección

- Esta sección ha presentado el capítulo 3 de:
  - ⇒ David A. Patterson y John L. Hennessy (2000). *Estructura y Diseño de Computadores. Interficie, circuitería/programación*. Editorial Reverté. ISBN: 84-291-2619-8.
- Además de lo visto hasta ahora, debes estudiar los siguientes apartados de dicho capítulo:
  - 3.7 Más allá de los números
  - 3.9 Inicio de un programa
  - 3.10 Un ejemplo para unirlo todo
  - 3.11 Tablas frente a punteros

## 2 Tipos de datos del R2000

Tipo	Tamaño	Datos representados
ASCII	8 bits	Caracteres
Byte	8 bits	Números enteros con y sin signo
Half	16 bits	(con signo: complemento a 2)
Word	32 bits	
Float	32 bits	Números reales de simple precisión
Double	64 bits	Números reales de doble precisión


# 3 Bancos de registros

- MIPS R2000 posee 3 bancos de registros:
  - Banco de registros de números enteros.
  - Banco de registros de números reales.
  - Banco de registros para el manejo de excepciones.

# 3.1 Banco de registros de números enteros

➤ Está formado por:

⇒ 32 registros de 32 bits de propósito general. Se identifican por el carácter '\$' seguido del número de registro: del \$0 al \$31.

 El \$0 está cableado a 0

⇒ 3 registros de propósito específico de 32 bits: HI, LO y PC.

⇒ HI (*High*). Parte alta del resultado de una multiplicación de 64 bits o el resto de una división entera.

⇒ LO (*Low*). Parte baja del resultado de una multiplicación de 64 bits o el cociente de una división entera.

⇒ PC (*Program Counter*). Almacena la dirección de memoria de la siguiente instrucción que se debe ejecutar.

## 3.2 Banco de registros de números reales

- Está formado por:
  - ⇒ 32 registros de 32 bits de propósito general para operaciones en coma flotante (IEEE 754 de simple precisión). Se identifican mediante la cadena “\$f” seguida del número del registro: del \$f0 al \$f31.
- Los registros anteriores pueden combinarse dos a dos para obtener 16 registros de 64 bits para operaciones en coma flotante (IEEE 754 de doble precisión). Se identifican por el primer elemento de cada par: así, \$f0 hace referencia al par formado por \$f0 y \$f1.  
(El primer elemento tiene que ser un registro par.)

## 3.3 Banco de registros para el manejo de excepciones

- El simulador SPIM implementa sólo los siguientes:
  - ⇒ `BadVAddress` (*bad virtual address*). Dirección virtual errónea; asociada a ciertos tipos de excepciones.
  - ⇒ `Status` (estado). Máscara de interrupción y bits de habilitación de interrupciones.
  - ⇒ `Cause` (causa). Tipo de la excepción actual y bits de interrupciones pendientes.
  - ⇒ `EPC` (*exception PC*). Dirección de la instrucción que ha provocado la excepción.

# 4 Organización de la memoria

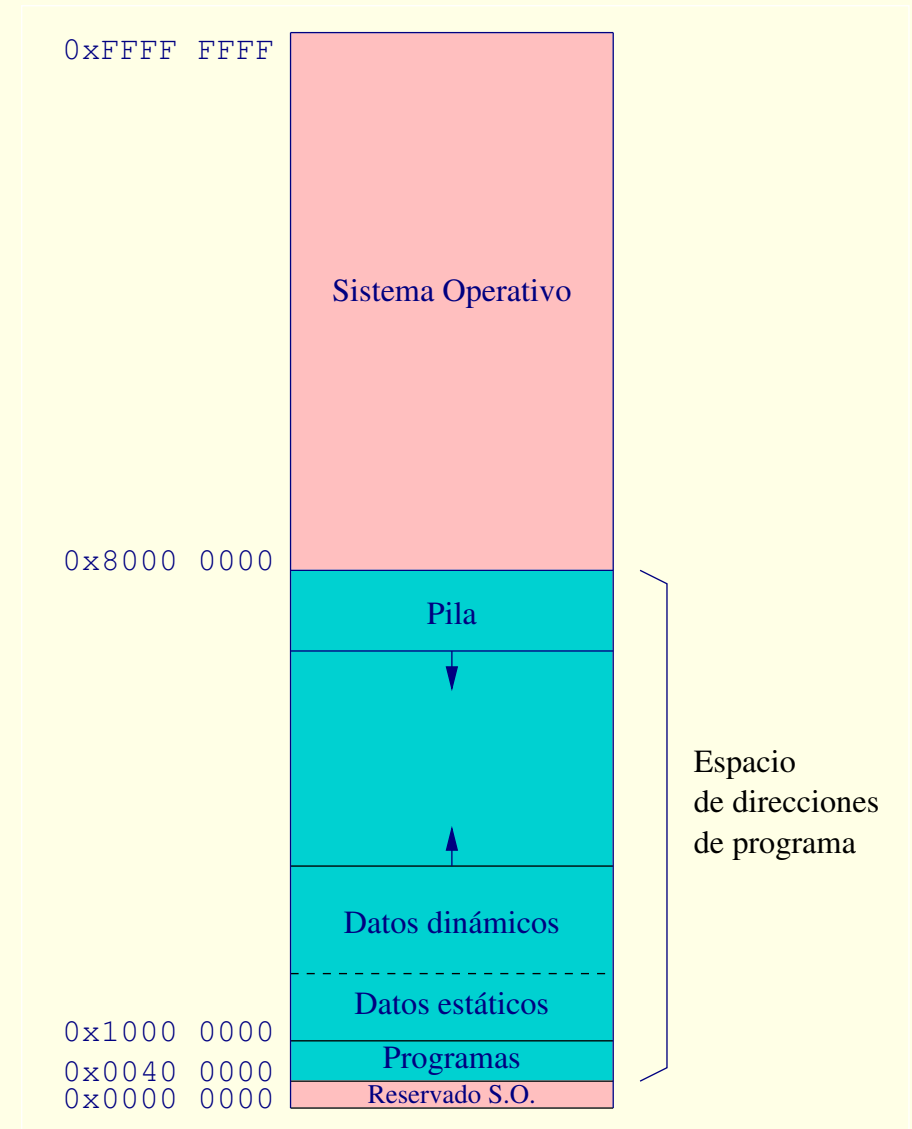
➤ 4 GB de espacio direccionable:

⇒  $2^{32}$  bytes:  $[0..2^{32} - 1]$

⇒  $2^{30}$  palabras:  $[0..2^{32} - 4]$

➤ Espacio accesible por el usuario:

$[0x0040\ 0000..0x7FFF\ FFFF]$



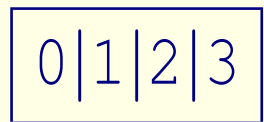


# 4 Organización de la memoria (II)

- Convenios para almacenar palabras (4 bytes) en memoria:

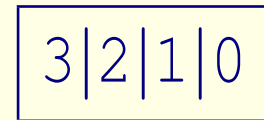
- ↳ Formato *Big Endian*.

- ➡ El byte de **mayor** peso va en la dirección más baja.



- ↳ Formato *Little Endian*.

- ➡ El byte de **menor** peso va en la dirección más baja.



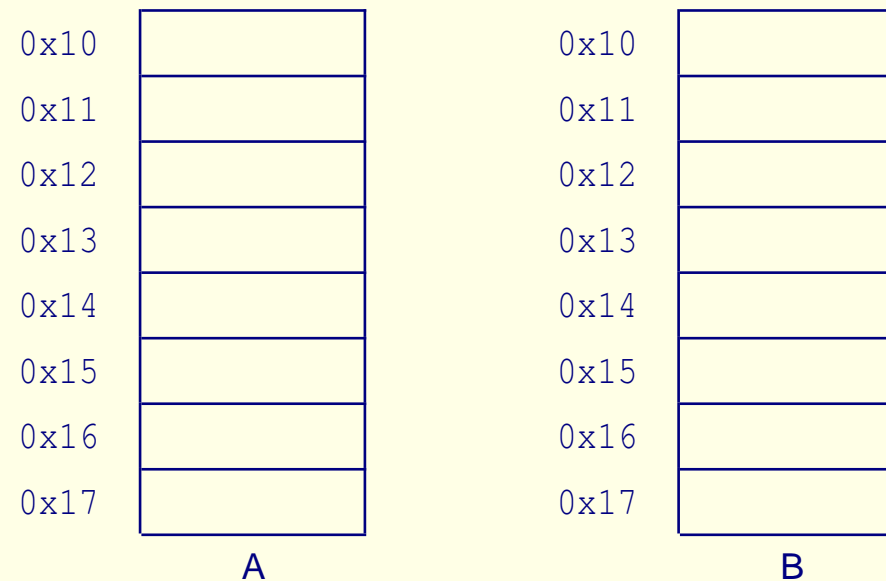
- Los procesadores MIPS R2000 pueden ser *Big Endian* o *Little Endian*.

- El simulador SPIM adopta el convenio utilizado por el procesador en el que se ejecuta: si es INTEL, *Little Endian*.

# 4 Organización de la memoria (III)

- **Ejercicio 14:** Las palabras `0x1020 3040` y `0x5060 7080` deben almacenarse en memoria en las direcciones `0x10` y `0x14`, respectivamente.

Escribe dichas palabras en los bancos de memoria A y B, utilizando el formato *Big Endian* y el *Little Endian*, respectivamente.



# 5 Juego de instrucciones

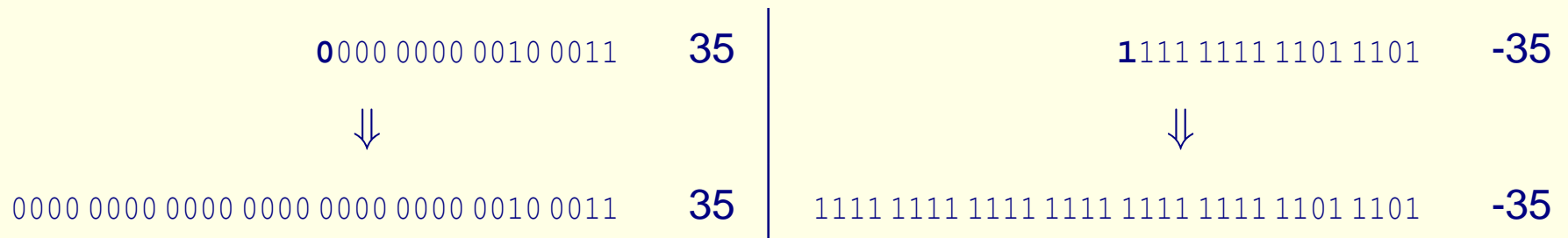
- El *juego de instrucciones* define el conjunto de operaciones que puede realizar el procesador (es el lenguaje del computador).
- Está formado por instrucciones:
  - ⇒ Aritméticas.
  - ⇒ Lógicas.
  - ⇒ De desplazamiento.
  - ⇒ De carga y almacenamiento.
  - ⇒ De movimiento entre registros especiales.
  - ⇒ De comparación.
  - ⇒ De salto condicional y de salto incondicional.

# 5 Juego de instrucciones (II)

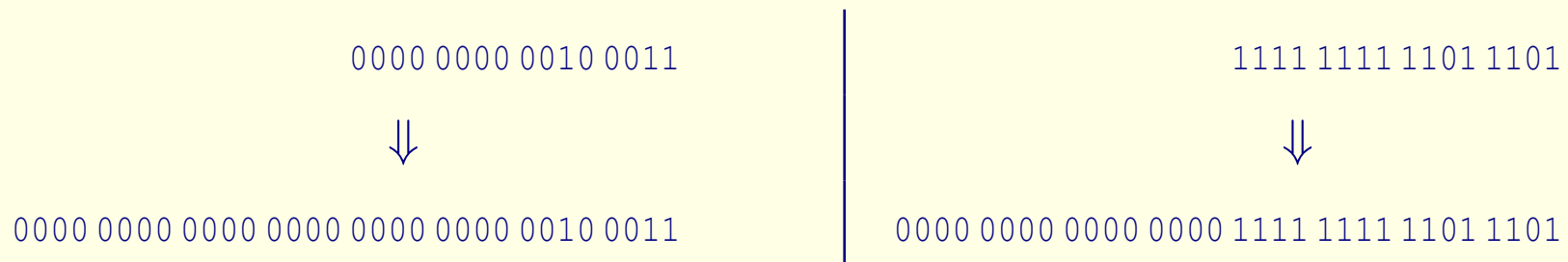
## EXTENSIÓN DEL SIGNO

- Los datos inmediatos ocupan 16 bits. Es necesario convertirlos a 32 bits. ¿Qué se pone en los 16 bits más altos?, depende:

- ➡ Instrucciones aritméticas: se extiende el signo.



- ➡ Instrucciones lógicas: se rellena con 0.



# 5.1 Instrucciones aritméticas

## Suma y resta

Sintaxis	Formato	Descripción
<b>add</b> rd,rs,rt	R	$r_d \leftarrow r_s + r_t$
<b>addi</b> rd,rs,inm	I	$r_d \leftarrow r_s + inm$
<b>addu</b> rd,rs,rt	R	$r_d \leftarrow r_s + r_t$ (suma sin signo)
<b>addiu</b> rd,rs,inm	I	$r_d \leftarrow r_s + inm$ (suma con cte. sin signo)
<b>sub</b> rd,rs,rt	R	$r_d \leftarrow r_s - r_t$
<b>subu</b> rd,rs,rt	R	$r_d \leftarrow r_s - r_t$ (resta sin signo)

(Nota: los datos inm se extienden a 32 bits rellenando los 16 bits de mayor peso con el bit de signo: tanto en **addi** como en **addiu**.)

## 5.1 Instrucciones aritméticas (II)

### Multiplicación y división

Sintaxis	Formato	Descripción
<b>mult</b> rs,rt	R	$HI \leftarrow (r_s \times r_t)_{63..32}, LO \leftarrow (r_s \times r_t)_{31..0}$
<b>multu</b> rs,rt	R	$HI \leftarrow (r_s \times r_t)_{63..32}, LO \leftarrow (r_s \times r_t)_{31..0}$ (sin signo)
<b>div</b> rs,rt	R	$HI \leftarrow (r_s \text{ mód } r_t), LO \leftarrow (r_s / r_t)$
<b>divu</b> rs,rt	R	$HI \leftarrow (r_s \text{ mód } r_t), LO \leftarrow (r_s / r_t)$ (sin signo)

( $r_s / r_t$  es la división entera de  $r_s$  entre  $r_t$ .)

## 5.2 Instrucciones lógicas

### Operaciones lógicas

Sintaxis	Formato	Descripción
<b>and</b> rd,rs,rt	R	$r_d \leftarrow r_s \text{ and } r_t$
<b>andi</b> rd,rs,inm	I	$r_d \leftarrow r_s \text{ and } inm$
<b>or</b> rd,rs,rt	R	$r_d \leftarrow r_s \text{ or } r_t$
<b>ori</b> rd,rs,inm	I	$r_d \leftarrow r_s \text{ or } inm$
<b>xor</b> rd,rs,rt	R	$r_d \leftarrow r_s \text{ XOR } r_t$
<b>xori</b> rd,rs,inm	I	$r_d \leftarrow r_s \text{ XOR } inm$
<b>nor</b> rd,rs,rt	R	$r_d \leftarrow r_s \text{ nor } r_t$

(Las operaciones se realizan bit a bit. Los datos inm se extienden a 32 bits con ceros.)

## 5.2 Instrucciones lógicas (II)

### Operaciones de desplazamiento

Sintaxis	Formato	Descripción
<b>sll</b> rd,rs,desp	R	$r_d \leftarrow r_s \ll desp$ desplazamiento a izquierdas, se rellena con 0 ( <i>shift left logic</i> )
<b>srl</b> rd,rs,desp	R	$r_d \leftarrow r_s \gg desp$ desplazamiento a derecha, se rellena con 0 ( <i>shift right logic</i> )
<b>sra</b> rd,rs,desp	R	$r_d \leftarrow r_s \ggg desp$ desplazamiento a derechas, se rellena con el valor del bit de signo ( <i>shift right arithmetic</i> )

(**sll** rd,rs,2 equivale a  $r_d \leftarrow r_s \times 4$ : se usa para multiplicar por 4 el índice de un vector.)



## 5.3 Instrucciones de carga y almacenamiento

### Carga y almacenamiento

Sintaxis	Formato	Descripción
<b>lw</b> rt, desp(rs)	I	$r_t \leftarrow M[desp + r_s]$
<b>lb</b> rt, desp(rs)	I	$r_t \leftarrow M[desp + r_s]$ carga 1 byte y extiende el signo ( <i>load byte</i> )
<b>lbu</b> rt, desp(rs)	I	$r_t \leftarrow M[desp + r_s]$ carga 1 byte y extiende con 0 ( <i>load byte unsigned</i> )
<b>sw</b> rt, desp(rs)	I	$M[desp + rs] \leftarrow r_t$
<b>sb</b> rt, desp(rs)	I	$M[desp + rs] \leftarrow r_t$ almacena 1 byte
<b>lui</b> rt, inm	I	$r_{t31...16} \leftarrow inm, r_{t15...0} \leftarrow 0$

## 5.4 Instrucciones de movimiento con registros HI y LO

### Transferencia de datos

Sintaxis	Formato	Descripción
<b>mfhi</b> rd	R	$r_d \leftarrow HI$ (move from HI)
<b>mflo</b> rd	R	$r_d \leftarrow LO$ (move from LO)
<b>mthi</b> rs	R	$HI \leftarrow r_s$ (move to HI)
<b>mtlo</b> rs	R	$LO \leftarrow r_s$ (move to LO)

## 5.5 Instrucciones de comparación

Poner a 1 si menor que

Sintaxis	Formato	Descripción
<b>slt</b> rd,rs,rt	R	Si $r_s < r_t$ entonces $r_d \leftarrow 1$ , si no $r_d \leftarrow 0$
<b>slti</b> rd,rs,inm	I	Si $r_s < inm$ entonces $r_d \leftarrow 1$ , si no $r_d \leftarrow 0$ ; inm se extiende a 32 bits con su signo ( <i>set less than immediate</i> )
<b>sltu</b> rd,rs,rt	R	Si $r_s < r_t$ entonces $r_d \leftarrow 1$ , si no $r_d \leftarrow 0$ ( <i>set less than unsigned</i> )

## 5.6 Instrucciones de salto condicional

### Salto condicional

Sintaxis	Formato	Descripción
<b>beq</b> rs,rt,etiqueta	I	Si $r_s = r_t$ entonces salta a la dir. etiqueta
<b>bne</b> rs,rt,etiqueta	I	Si $r_s \neq r_t$ entonces salta a la dir. etiqueta
<b>bgez</b> rs,etiqueta	I	Si $r_s \geq 0$ entonces salta a la dir. etiqueta ( <i>branch if greater or equal than zero</i> )
<b>bgtz</b> rs,etiqueta	I	Si $r_s > 0$ entonces salta a la dir. etiqueta
<b>blez</b> rs,etiqueta	I	Si $r_s \leq 0$ entonces salta a la dir. etiqueta ( <i>branch if less or equal than zero</i> )
<b>bltz</b> rs,etiqueta	I	Si $r_s < 0$ entonces salta a la dir. etiqueta

## 5.7 Instrucciones de salto incondicional

### Salto incondicional

Sintaxis	Formato	Descripción
<b>j</b> etiqueta	J	Salta a la dirección etiqueta
<b>jal</b> etiqueta	J	$\$31 \leftarrow PC + 4$ y salta a la dirección etiqueta
<b>jr</b> rs	R	Salta a la dir. contenida en el registro $r_s$

# 6 Programación en ensamblador

- El lenguaje ensamblador además de:
  - ⇒ proporcionar nemotécnicos para los códigos de operación y registros, y
  - ⇒ permitir usar etiquetas para identificar posiciones de memoria;
- proporciona los siguientes recursos de programación:
  - ⇒ Directivas.  
Indican cómo debe traducirse el programa.
  - ⇒ Pseudo-instrucciones.  
Instrucciones extra que facilitan la programación en ensamblador.

# 6.1 Directivas

- Las directivas permiten definir ciertos aspectos que le indican al programa ensamblador cómo debe traducir el código.
- Se pueden clasificar en:
  - ⇒ Directivas de inicio de las zonas de datos e instrucciones.
  - ⇒ Directivas de reserva de espacio.
  - ⇒ Directivas de propósito variado.



## 6.1.1 Directivas de inicio de las zonas de datos e instrucciones

### ➤ **.data** [dirección]

Indica el comienzo de la zona de datos. (Si no se especifica la dirección se toma la siguiente:  $0x1001\ 0000$ .)

### ➤ **.text** [dirección]

Indica el comienzo de la zona de código.

(Si no se especifica la dirección se toma la siguiente:  $0x0040\ 0000$ ; ahora bien, puesto que el simulador carga por defecto un código de arranque que pone en la dirección  $0x0040\ 0000$ , la dirección en este caso será la  $0x0040\ 0024$ —.)



## 6.1.2 Directivas de reserva de espacio

➤ **.space** n

Reserva n bytes y los inicializa a 0.

➤ **.byte** b1 [, b2]...

Reserva e inicializa los bytes indicados.

➤ **.half** h1 [, h2]...

Reserva e inicializa las medias palabras indicadas. (En posiciones de memoria pares.)

➤ **.word** w1 [, w2]...

Reserva e inicializa las palabras indicadas. (En posiciones de memoria múltiplo de 4.)

## 6.1.2 Directivas de reserva de espacio (II)

➤ **.ascii** cadena1 [, cadena2]...

Almacena una cadena de caracteres en memoria. Cada cadena debe estar entrecomillada (p.e. **.ascii** "¡Hola mundo!").

➤ **.asciiz** cadena1 [, cadena2]...

Igual que la anterior pero reserva un byte adicional al final de cada cadena y lo pone a 0.

## 6.1.3 Directivas de propósito variado

### ➤ **.globl** ETIQUETA

Sirve para indicar que ETIQUETA tiene ámbito global (si una etiqueta ha sido declarada en un fichero y queremos utilizarla en otro, utilizaremos la directiva **.globl** en este último para hacerlo).

### ➤ **.end**

Indica que se ha alcanzado el final del programa en ensamblador.

### ➤ **.align** $N$ , donde $N = 1, 2, 3 \dots$

Sirve para alinear el siguiente dato a una dirección múltiplo de  $2^N$ .

## 6.2 Pseudo-instrucciones

- Instrucciones extra proporcionadas por el ensamblador.
- El programador las ve como instrucciones normales.
- El ensamblador se encarga de traducirlas a instrucciones sí soportadas por el procesador (de una a cuatro por cada pseudo-instrucción).



## 6.2.1 Pseudo-instrucciones de carga y almacenamiento

### ► **li** rd, inm (*load immediate*)

⇒ Carga en el registro rd el dato indicado por inm (generalmente será un número, pero puede ser un carácter).

⇒ Ejemplos:

```
li $t0,32 # $t0 ← 32
```

```
li $t1,'A' # $t1 ← 65 (65 es el código ASCII de la A)
```

⇒ Código máquina equivalente:

```
li $t0,32 ⇒ ori $t0,$zero,32
```

```
li $t0,65538 ⇒ lui $at,1
```

```
ori $t0,$at,2 # 65538 es 0x0001 0002
```

## 6.2.1 Pseudo-instrucciones de carga y almacenamiento (II)

### ► **la** rd, ETIQUETA (*load address*)

⇒ Carga en el registro rd la dirección de memoria referenciada por ETIQUETA.

⇒ Ejemplo:

```
        .data 0x10008000
datos : .word 1 , 2 , 3 , 4
        .text
main :  la $t0 , datos      # $t0 ← 0x10008000
```

⇒ Código máquina equivalente:

**la** \$t0,datos ⇒ **lui** \$at,0x1000

**ori** \$t0,\$at,0x8000

## 6.2.2 Pseudo-instrucciones de salto condicional

➤ **bge** rs,rt,ETIQUETA (*branch if greater or equal*)

**bgt** rs,rt,ETIQUETA (*branch if greater than*)

**ble** rs,rt,ETIQUETA (*branch if less or equal*)

**blt** rs,rt,ETIQUETA (*branch if less than*)

⇒ Saltan a la dirección de memoria referenciada por ETIQUETA en caso de cumplirse la relación indicada por cada instrucción.

⇒ Ejemplo:

**bgt** \$t0,\$t1,ETIQUETA #Si  $\$t0 > \$t1$ ,  $PC \leftarrow ETIQUETA$

⇒ Código máquina equivalente:

**bgt** \$t0,\$t1,ETIQUETA  $\Rightarrow$  **slt** \$at,\$t0,\$t1

**bne** \$at,\$zero,ETIQUETA

# Fin

Copyright © 2006 Sergio Barrachina Mir

Área de Arquitectura y Tecnología de Computadores  
Dpt. de Ingeniería y Ciencia de los Computadores  
Universidad Jaume I

Realizada con `ujislides` © 2002-5 Sergio Barrachina ([barrachi@icc.uji.es](mailto:barrachi@icc.uji.es))