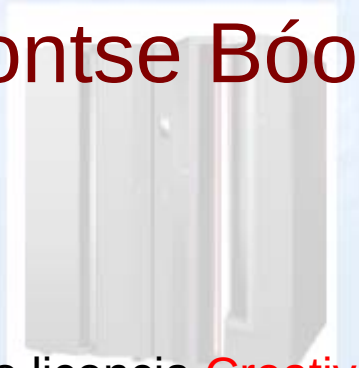


# El repertorio de instrucciones

Montse Bóo Cepeda





# Estructura del curso

1. Evolución y caracterización de los computadores.
2. Arquitectura del MIPS: Introducción.
3. Tipo de datos.
4. El repertorio de instrucciones.
5. Aritmética del computador.
6. El camino de datos.
7. Sección de control.
8. El camino de datos multiciclo.
9. Sección de control multiciclo.
10. Entrada/Salida (I/O).

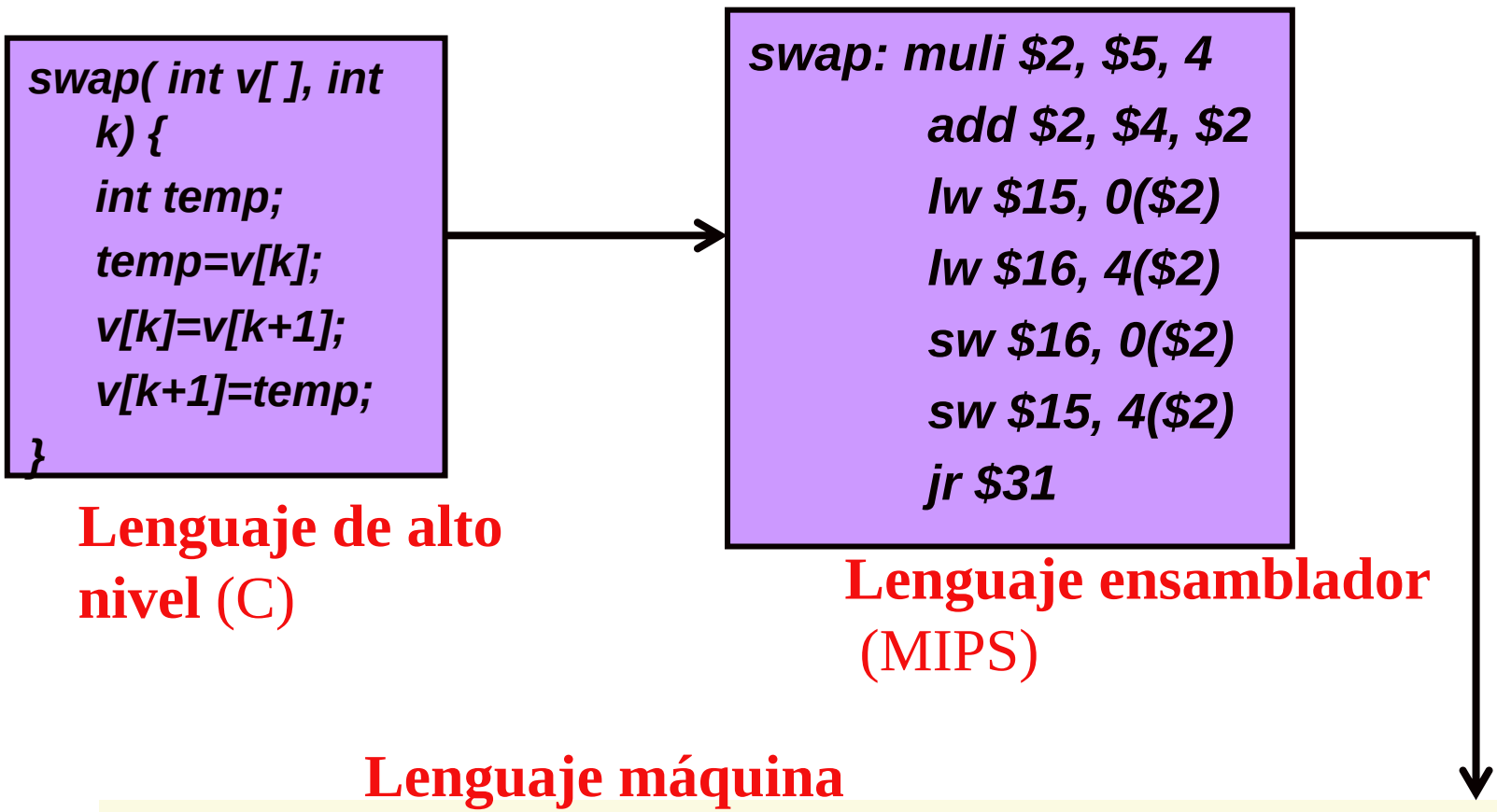
# Esquema de contenidos

1. Introducción: Visión del programador
2. Formato de instrucciones
3. Modos de direccionamiento
4. Instrucciones típicas
5. Llamadas a subrutinas
6. Pila o stack

# Introducción: La visión del programador

- El funcionamiento de la CPU está determinado por las instrucciones que ejecuta.
- El conjunto de instrucciones distintas que puede ejecutar se denomina **repertorio de instrucciones**
- En los computadores actuales las instrucciones se presentan como números  y se almacenan en memoria (**programa almacenado**)
- El lenguaje utilizado por el ordenador se denomina **lenguaje máquina**. El lenguaje **ensamblador** es una representación simbólica más cercana al humano.

# Visión software: Jerarquía de traducción



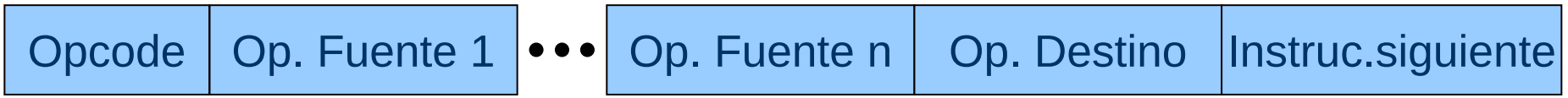
# Formato de instrucciones

Elementos de una instrucción máquina:

- Código de operación.
  - Especifica el tipo de operación a realizar: Suma, resta, movimiento de datos, …
- Referencia a operandos fuente
  - La instrucción puede involucrar ninguno, uno o varios operandos fuente.
  - Los operandos pueden especificarse utilizando diferentes modos de direccionamiento.
- Referencia al operando resultado
  - Si la instrucción produce un resultado debe especificar el operando destino.
  - El operando puede especificarse utilizando diferentes modos de direccionamiento.
- Referencia a la siguiente instrucción.
  - Normalmente la ejecución del programa es secuencial y la dirección de la siguiente instrucción está implícita.
  - En las instrucciones de salto se especifica la dirección de la siguiente instrucción.

# Formato de instrucciones

- Dentro del computador cada instrucción se representa por una secuencia de bits
- La instrucción se divide en campos correspondientes a los elementos constitutivos de la misma (código de operación, operandos, etc.)
- La descripción en campos y bits se denomina **formato de instrucción**



# Diseño del repertorio de instrucciones

Se debe decidir:

- Longitud de las instrucciones
  - Todas de la misma longitud.
  - De distinta longitud dependiendo del tipo de operación.
- Número de campos (dependiendo del tipo de instrucción).
- Número de bits por campo.
- Codificación de cada campo.

Factores a tener en cuenta para decidir el formato:

- Número de operaciones distintas
  - Aritméticas, lógicas, de control, de movimiento de datos.
- Número de operandos de cada instrucción.
  - Sin operandos, con 1,2,3… operandos.
- Modos de direccionamiento:
  - Inmediato, absoluto, de registro, indirecto…
- Tamaño y tipos de datos:
  - Bit, byte, palabra, doble palabra…
  - Caracteres, BCD, signo magnitud, complemento a 2, punto flotante…

# Repertorio de instrucciones escogido: MIPS

- Arquitectura MIPS
- Pionera de las arquitecturas RISC
- Muy sencilla y uniforme
- Creada en Stanford (John Hennessy)
- Utilizada por SGI, NEC, Toshiba

# MIPS: Ejemplo de sencillez

**SUMA de dos variables b y c, resultado en a.**

```
add a, b, c      # a = b + c
```

**SUMA de tres variables (b,c,d) y resultado en a.**

```
add a, b, c      # a = b + c
add a, a, d      # a = a + d= (b+c)+d
```

# Repertorio de instrucciones MIPS

- Longitud de instrucción fija: 32 bits
- La arquitectura MIPS realiza la mayor parte de los cálculos con datos almacenados en registros
- MIPS es sencilla y eficiente
- El acceso a memoria se hace a través de operaciones de carga/almacenamiento (transferencia de datos)
- MIPS direcciona bytes individuales, para indicar la dirección de una palabra en memoria hay que indicar su dirección (dirección de su primer byte).

# Registros del MIPS

- 32 registros de 32 bits

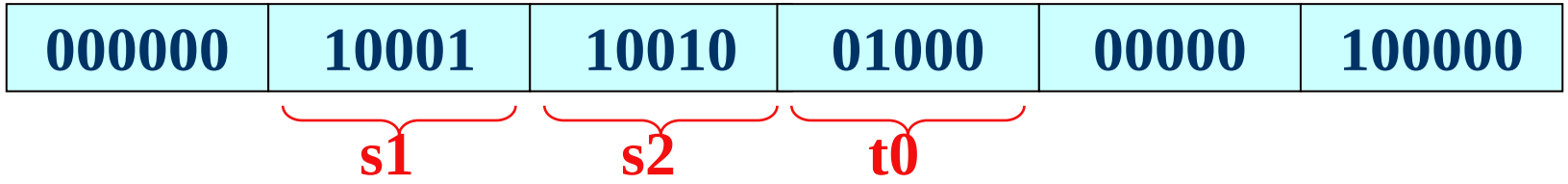
| Númeo | Nombre | Uso convencional |
|---|---|---|
| 0 | $zero | Valor constante 0 |
| 1 | $at | Reservado por el ensamblador |
| 2-3 | $v0-$v1 | Para resultados y evaluación de expresiones |
| 4-7 | $a0-$a3 | Argumentos |
| 8-15 | $t0-$t7 | Temporales |
| 16-23 | $s0-$s7 | Salvados |
| 24-25 | $t8-$t9 | Temporales |
| 26-27 | $k0-$k1 | Reservado para núcleo de SO |
| 28 | $gp | Puntero global |
| 29 | $sp | Puntero de pila |
| 30 | $fp | Puntero de bloque de activación |
| 31 | $ra | Dirección de retorno |

# Ejemplo de instrucción MIPS

Ejemplo:

**add $t0, $s1, $s2**

Se representa en lenguaje MIPS como:




En representación decimal:

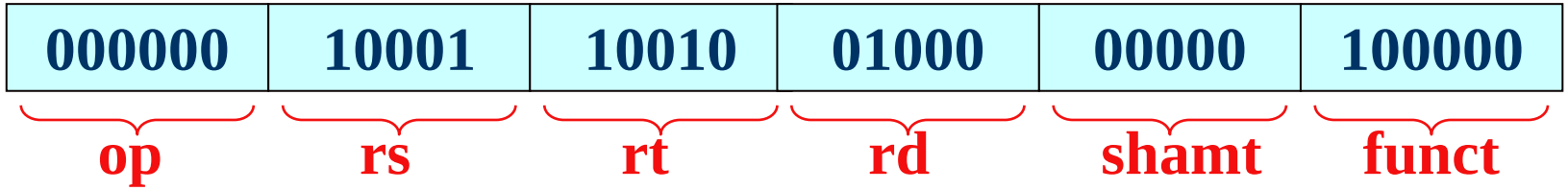
| 0 | 17 | 18 | 8 | 0 | 32 |
|---|---|---|---|---|---|

# Ejemplo de instrucción MIPS

Ejemplo:

**add $t0, $s1, $s2**

Se representa en lenguaje MIPS como:




Campos:

op: código de operación.

rs: primer registro operando fuente

rt: segundo registro operando fuente

rd: registro operando destino

shamt: tamaño de desplazamiento (*shift amount*)

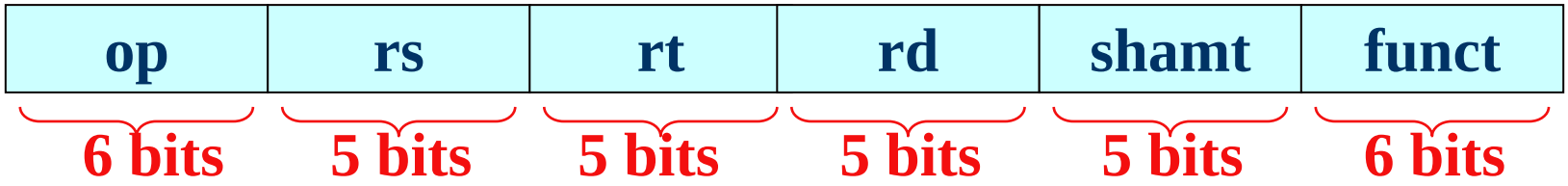
funct: código de función.

# Formatos de las instrucciones MIPS

MIPS tiene 3 tipos de formatos:

- TIPO R: Operaciones aritméticas y lógicas
- TIPO I: Transferencia de datos, salto condicional e instrucciones con operandos inmediatos.
- TIPO J: Instrucciones de bifurcación.

**TIPO R**



# Formatos de las instrucciones MIPS

MIPS tiene 3 tipos de formatos:

- TIPO R: Operaciones aritméticas y lógicas
- TIPO I: Transferencia de datos, salto condicional e instrucciones con operandos inmediatos.
- TIPO J: Instrucciones de bifurcación.

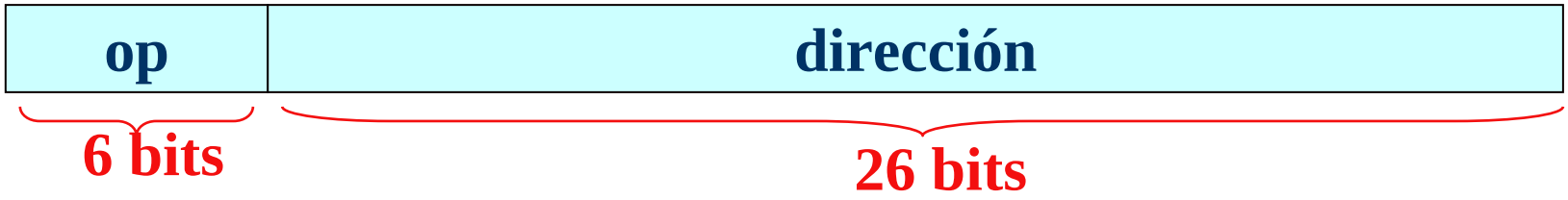
**TIPO I**

# Formatos de las instrucciones MIPS

MIPS tiene 3 tipos de formatos:

- TIPO R: Operaciones aritméticas y lógicas
- TIPO I: Transferencia de datos, salto condicional e instrucciones con operandos inmediatos.
- TIPO J: Instrucciones de bifurcación.

**TIPO J**



# ¿Dónde están los operandos?: Modos de direccionamiento

Los operandos pueden estar:

- En la instrucción:
  - Rápido y simple.
  - Sólo para constantes
- En la memoria principal:
  - Direccionamiento implica muchos bits.
  - Acceso lento
- En los registros de la CPU:
  - Hay pocos registros: el direccionamiento implica pocos bits
  - Acceso rápido

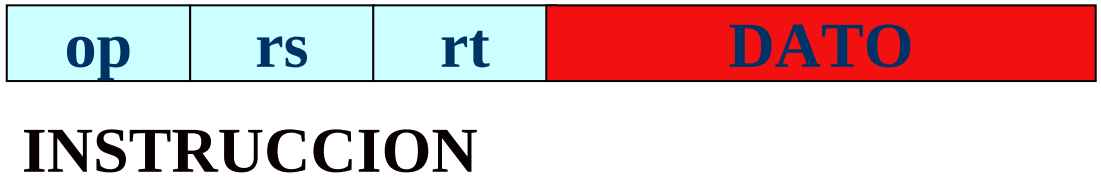
# ¿Dónde están los operandos?: Modos de direccionamiento

Modos de direccionamiento del MIPS:

- Modo de direccionamiento inmediato.
- Modo de direccionamiento registro.
- Modo de direccionamiento base con desplazamiento.
- Modo de direccionamiento relativo al PC.
- Modo de direccionamiento pseudodirecto.

# Direccionamiento inmediato

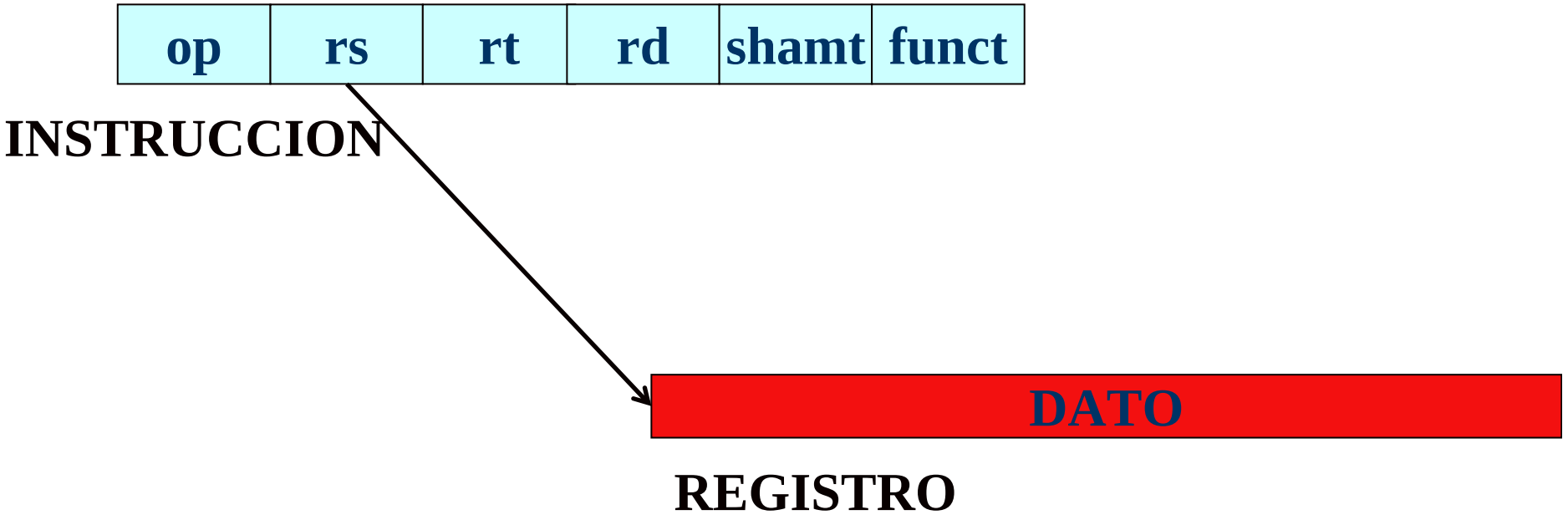
- El operando es una constante que aparece en la instrucción




**INSTRUCCION**

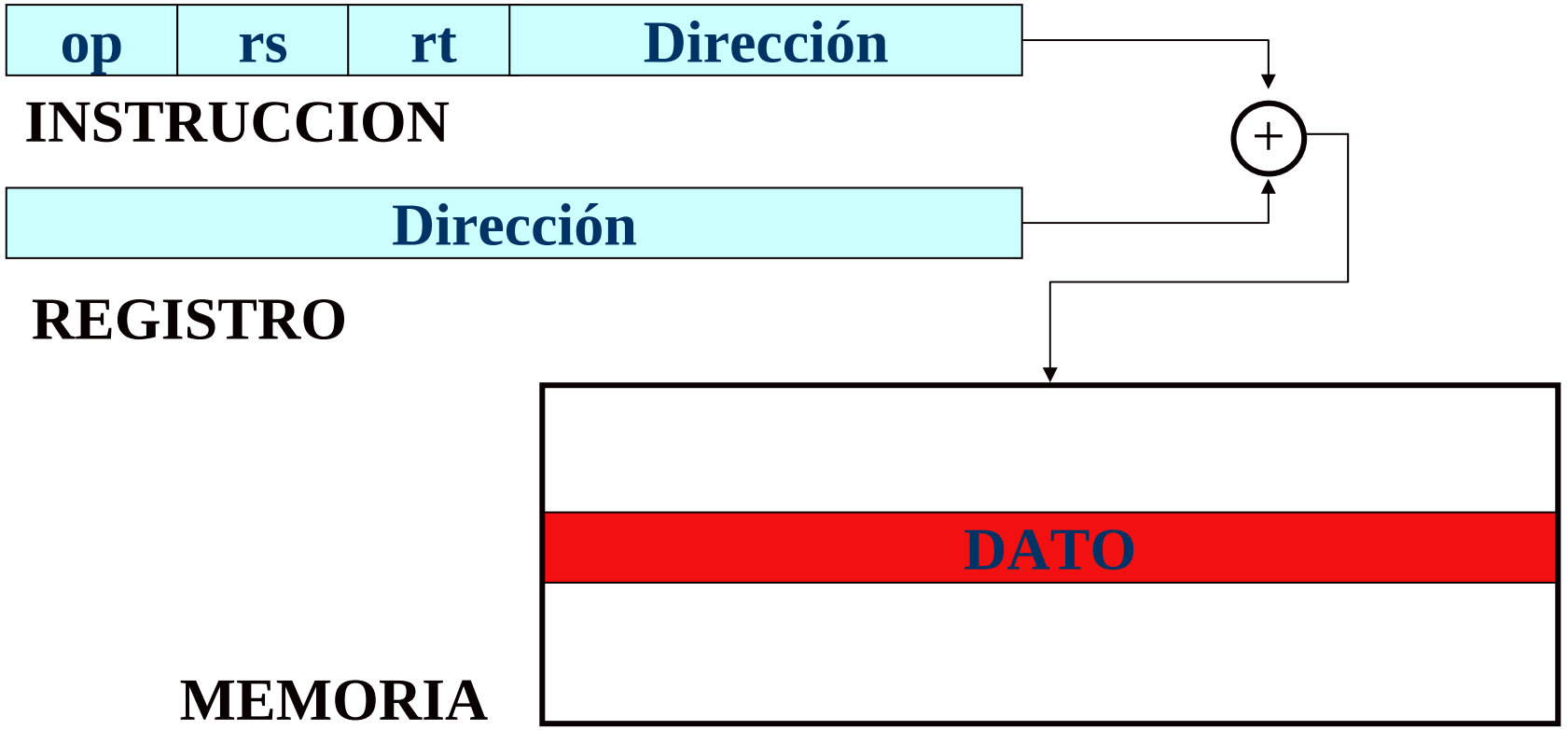
# Modo de direccionamiento registro

- El operando está en un registro



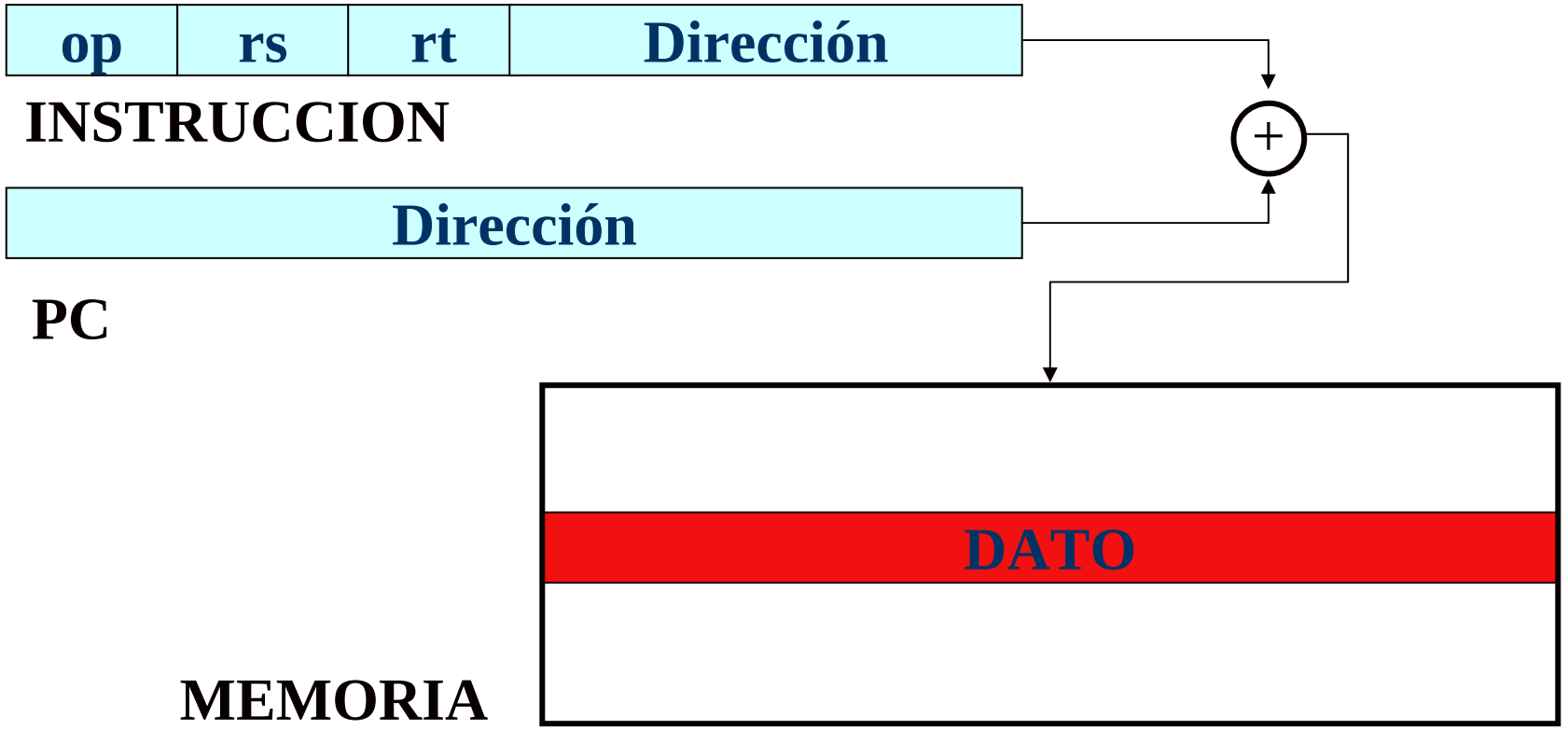
# Modo de direccionamiento base con desplazamiento

- El operando está en la memoria en la dirección: constante en la instrucción + contenido registro



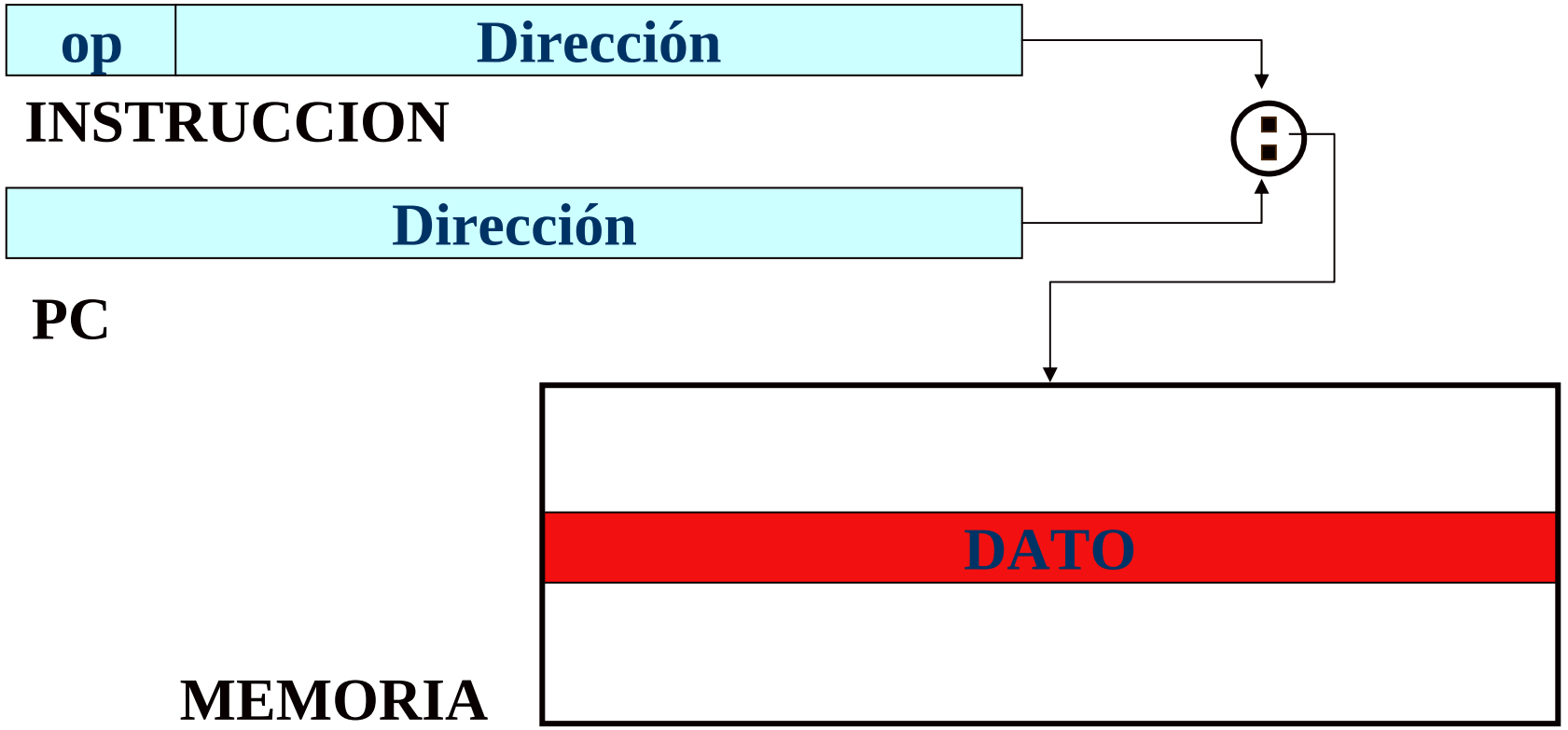
# Modo de direccionamiento relativo al PC

- El operando está en la memoria en la dirección: constante en la instrucción + contador de programa (PC).



# Modo de direccionamiento pseudodirecto

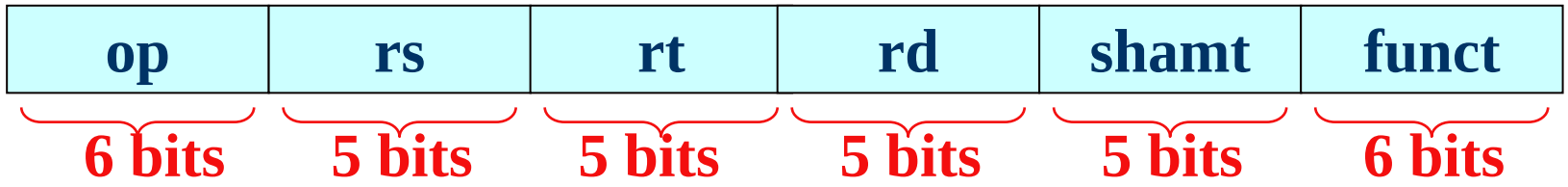
- El operando está en la memoria en la dirección: 26 bits de instrucción CONCATENADOS CON los bits de mayor peso del contador de programa (PC).



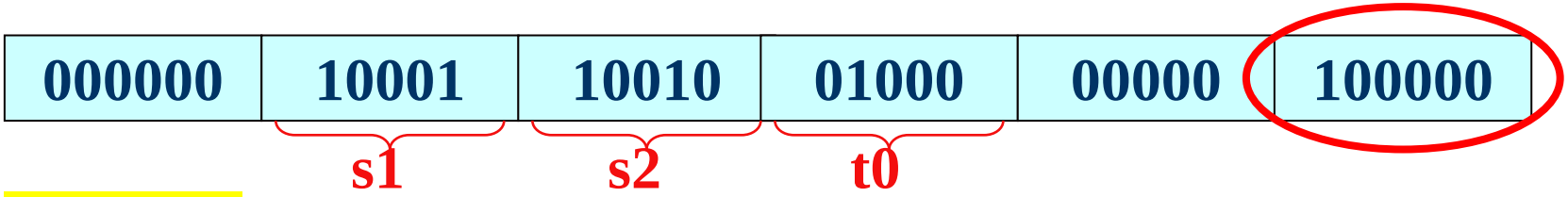
# Instrucciones aritmético-lógicas

**TIPO R**

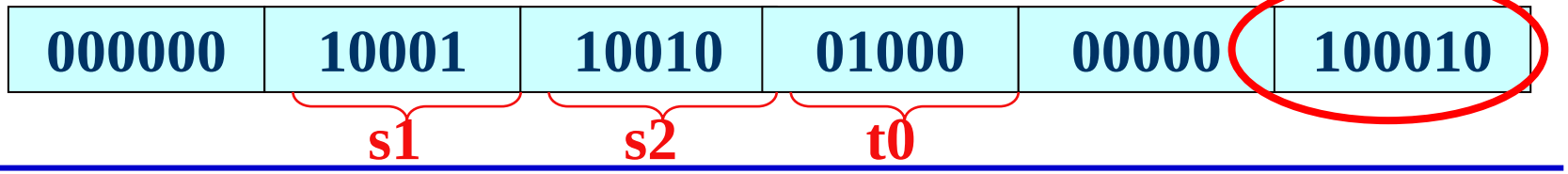
Direccionamiento registro




**SUMA** **add $t0, $s1, $s2** # **$t0 = $s1+$s2**



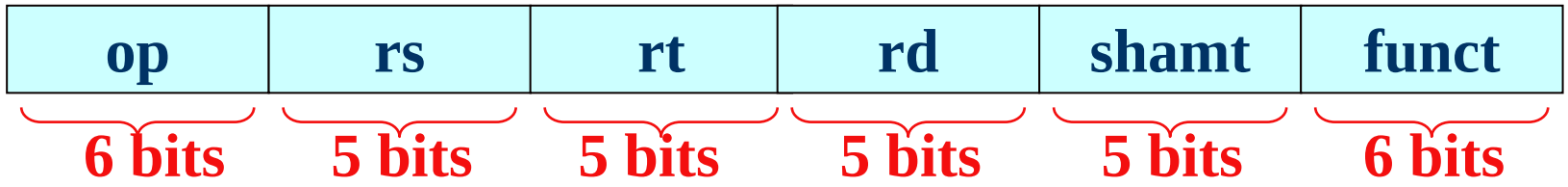

**RESTA** **sub $t0, $s1, $s2** # **$t0 = $s1-$s2**



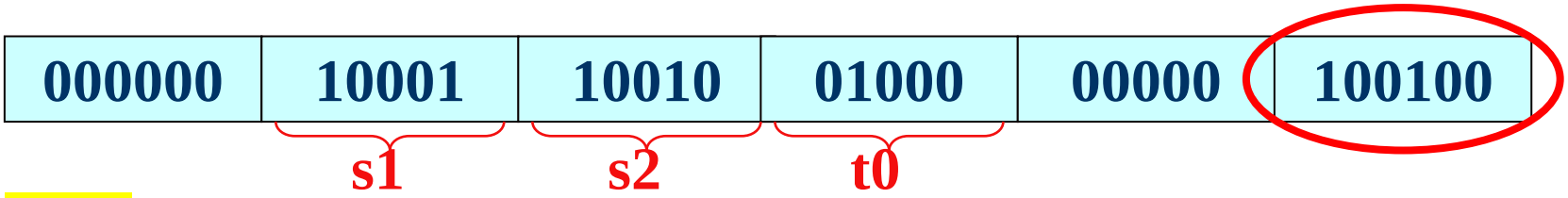
# Instrucciones aritmético-lógicas

**TIPO R**

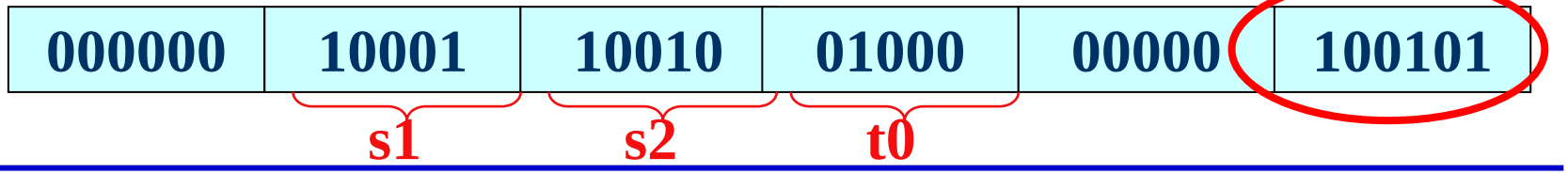
Direccionamiento registro




**AND** `and $t0, $s1, $s2` # $t0 = $s1 AND $s2




**OR** `or $t0, $s1, $s2` # $t0 = $s1 OR $s2



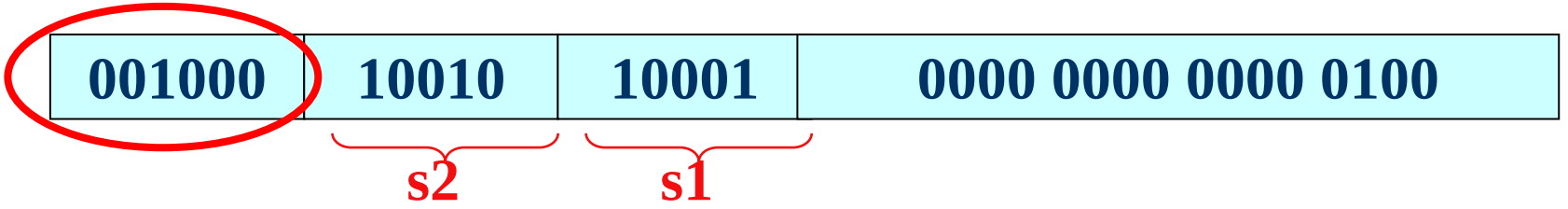
# Instrucciones aritmético-lógicas

- Hay instrucciones aritméticas de tipo I (modo direccionamiento inmediato)

**TIPO I**

Direccionamiento inmediato


**SUMA** **addi $s1, $s2,4** **# $s1 = $s2 + 4**



# Instrucciones de transferencia

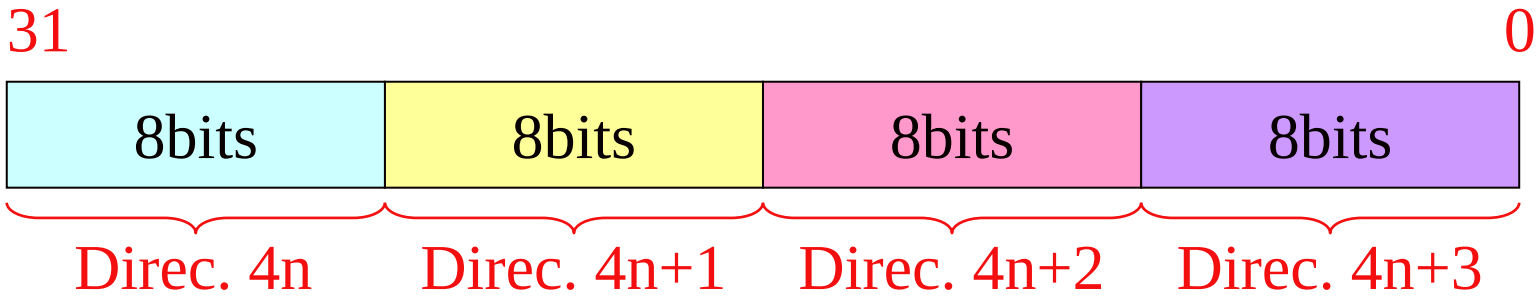
- Almacenamiento de información en la memoria.
- Instrucciones MIPS de acceso a la memoria:
  - lw (load word) carga una palabra desde memoria
  - sw (store word) almacena una palabra en memoria
- Memoria con direccionamiento de bytes (8 bits):
  - En MIPS las direcciones son múltiplos de 4 (restricción de alineación).

Computador de 32 bits: 4 bytes/palabra

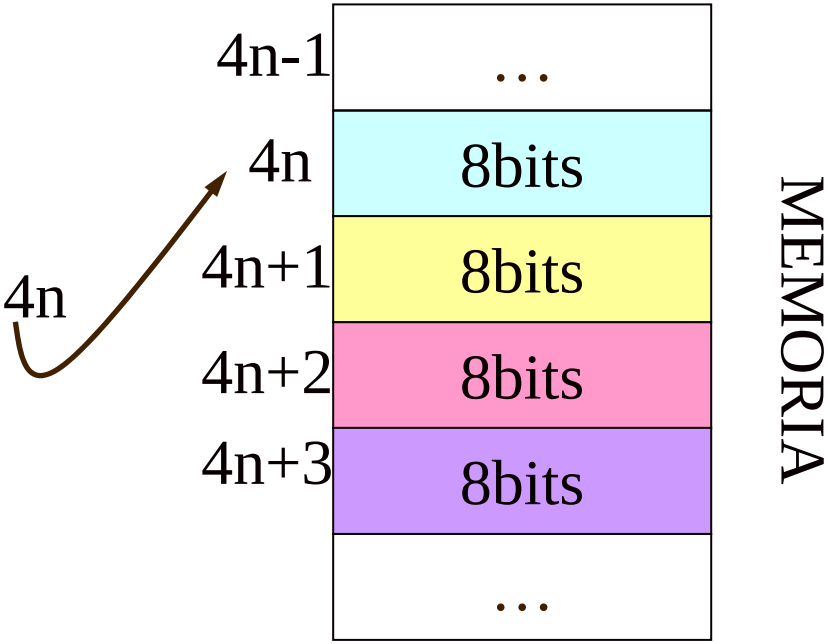
MIPS: Big Endian/Little Endiam

# Instrucciones de transferencia: Direccionamiento de la memoria

Palabra de 32 bits:




Ejemplo Big Endian: DIRECCION 4n



# Instrucciones de transferencia: lw

Ejemplo:

**lw $s1,1200($s2)** **# $s1 = Mem [1200+$s2]**

Explicación:

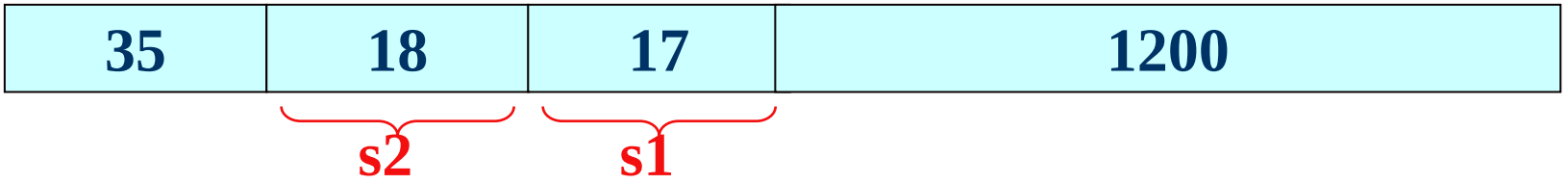
- Carga una palabra de 32 bits desde la memoria
- La pone en un registro ($s1)
- La dirección de memoria se encuentra en un registro ($s2) **más un desplazamiento** (1200)
- Cuidado!: Sólo las direcciones múltiplo de 4 son válidas (palabras de 32 bits)

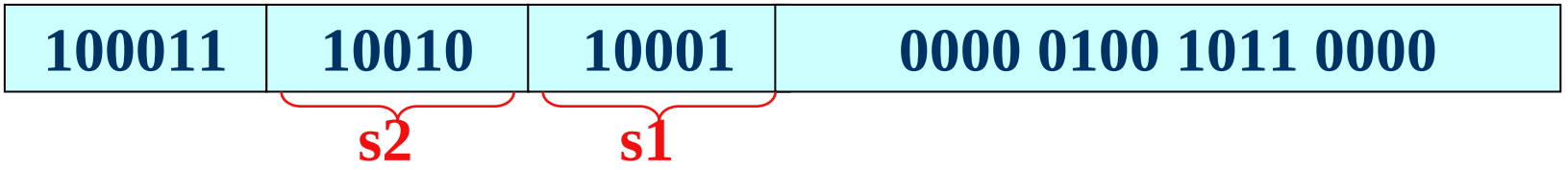
# Instrucciones de transferencia: lw

**TIPO I**

Direccionam. base con desplazamiento


**lw** **lw $s1,1200($s2)** **# $s1 = Mem [1200+$s2]**



# Instrucciones de transferencia: sw

Ejemplo:

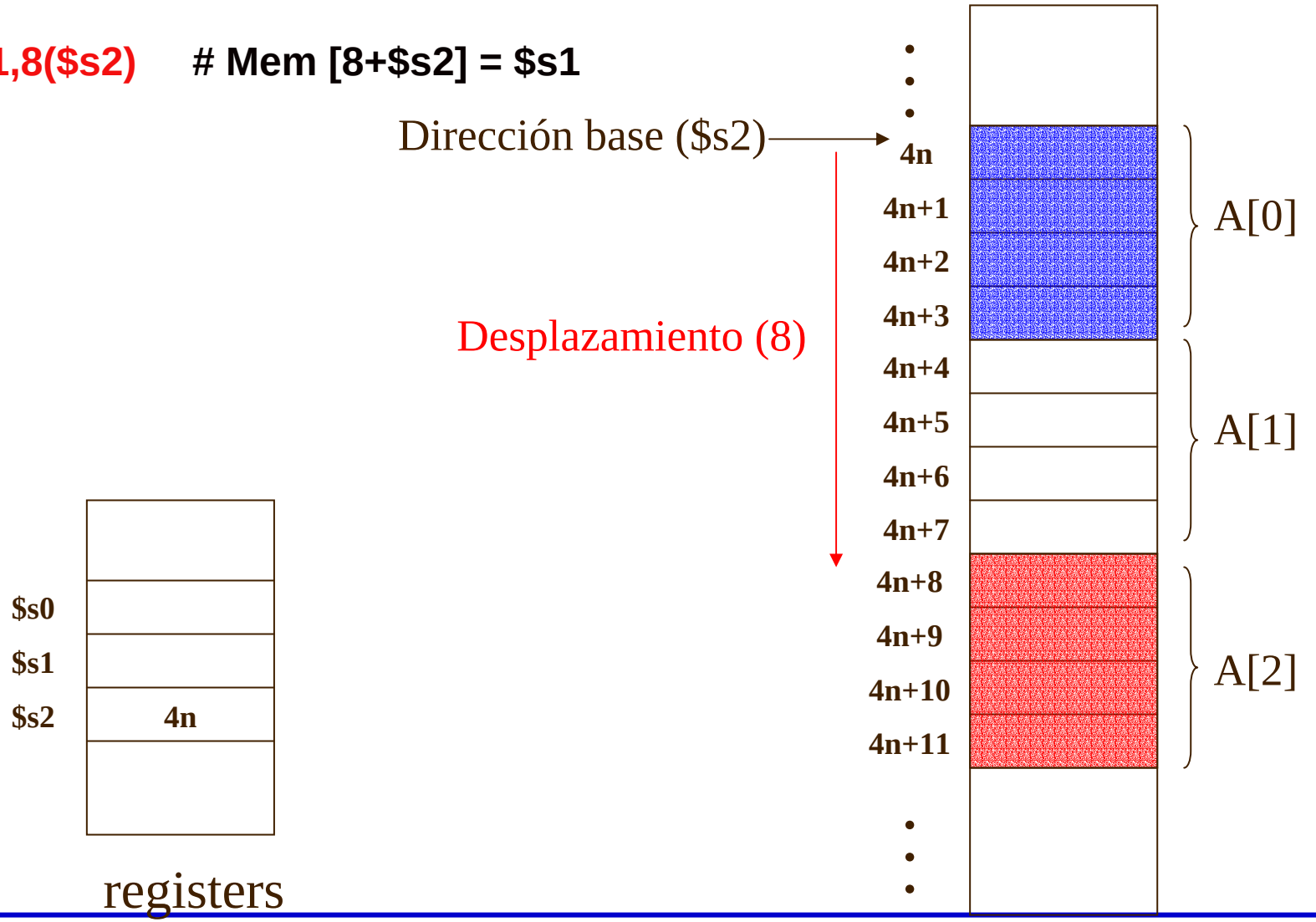
**sw $s1,8 ($s2)** # **Mem [8+$s2] = $s1**

Explicación:

- Guarda una palabra de 32 bits en la memoria
- La palabra esta en el registro ($s1)
- La dirección de memoria se encuentra en un registro ($s2) **más un desplazamiento** (8)
- Cuidado!: Sólo las direcciones múltiplo de 4 son válidas (palabras de 32 bits)

# Instrucciones de transferencia: sw

**sw $s1,8($s2)** **# Mem [8+$s2] = $s1**

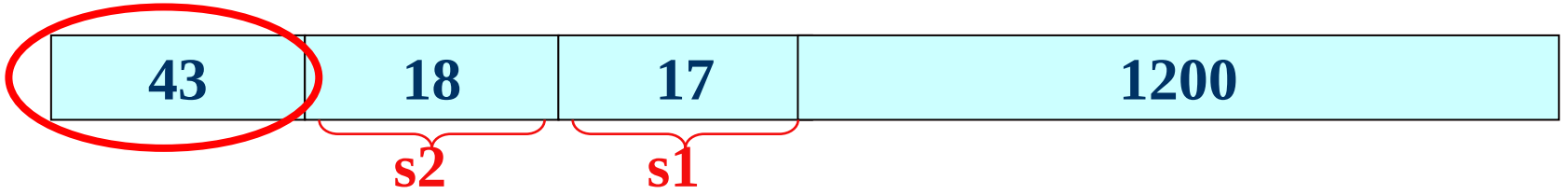


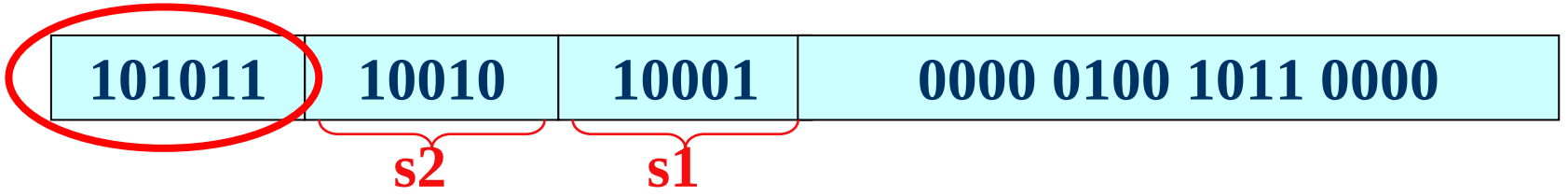
# Instrucciones de transferencia: sw

**TIPO I**

Direccionam. base con desplazamiento


**sw** **sw $s1,1200($s2)** **# Mem [1200+$s2] = $s1**



# Carga de constantes en registros: load upper inmediate

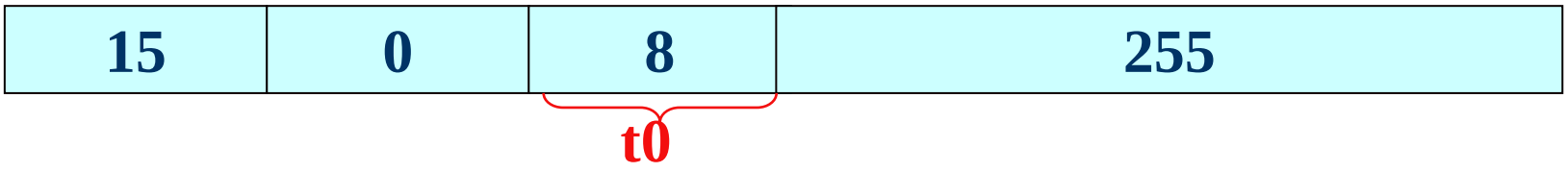
**TIPO I**

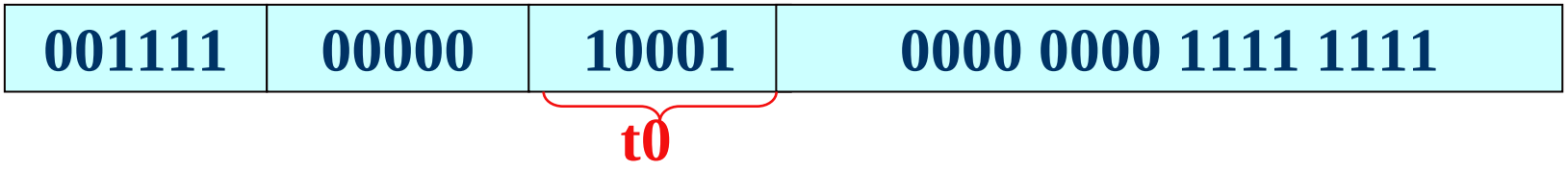
Direccionamiento inmediato


**lui**

**lui $t0, 255**

**# Carga 255 en los 16 bits más significativos de $t0**



# Carga de constantes en registros: carga de 32 bits

- Es necesario utilizar otra instrucción para cargar los 16 bits menos significativos
- Supongamos que queremos cargar en $s0:

**Constante = 0000 0000 0011 1101 0000 1001 0000 0000**

**61** **2304**

- DOS PASOS:
  - Carga de los 16 bits más significativos

    **lui $s0, 61**
  - Suma de los 16 bits menos significativos

    **addiu $s0, $s0, 2304**

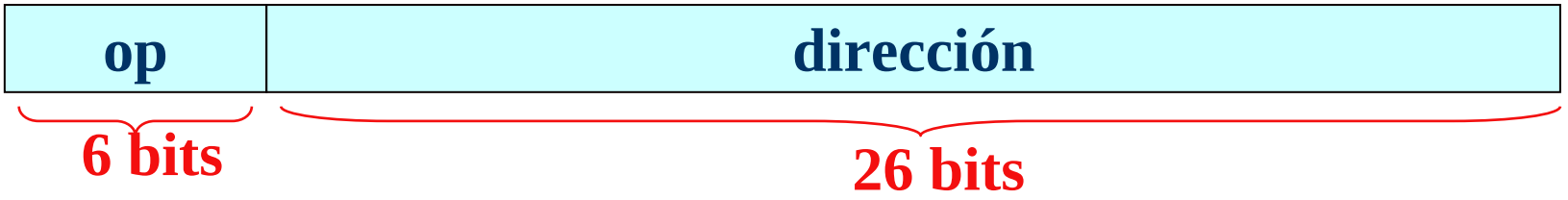
# Instrucciones de cambio de flujo

- Hacen que el programa salte a otra instrucción distinta de la siguiente
- Pueden saltar siempre o dependiendo de una condición
- La instrucción debe indicar la dirección de la nueva instrucción
  - De forma absoluta
  - De forma relativa
- MIPS dispone de:
  - Salto incondicional: **`j, jr`**
  - Salto condicional: **`beq, bne`**
  - Apoyo al salto condicional: **`slt`**

# Salto incondicional: jump

**TIPO J**

Direccionamiento pseudodirecto




**j** **j 2500** **# ir a dirección 10000**

**PC anterior = 0000 xx xxxx xxxx xxxx xxxx xxxx xxxx xx**

**PC nuevo = 0000 00 0000 0000 0000 1001 1100 0100 00**

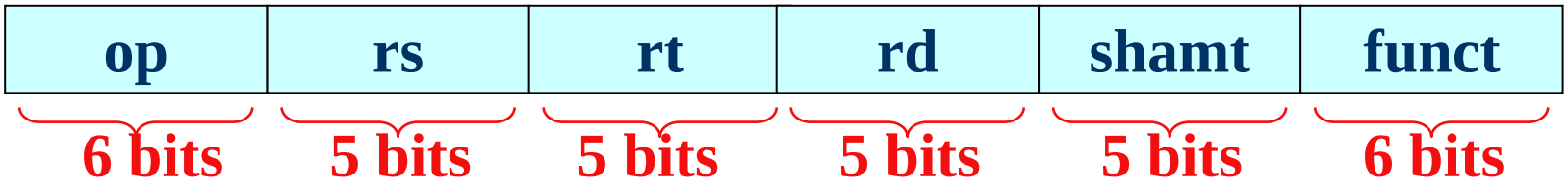
| 2 | 2500 |
|---|---|

| 000010 | 00 0000 0000 0000 1001 1100 0100 |
|---|---|

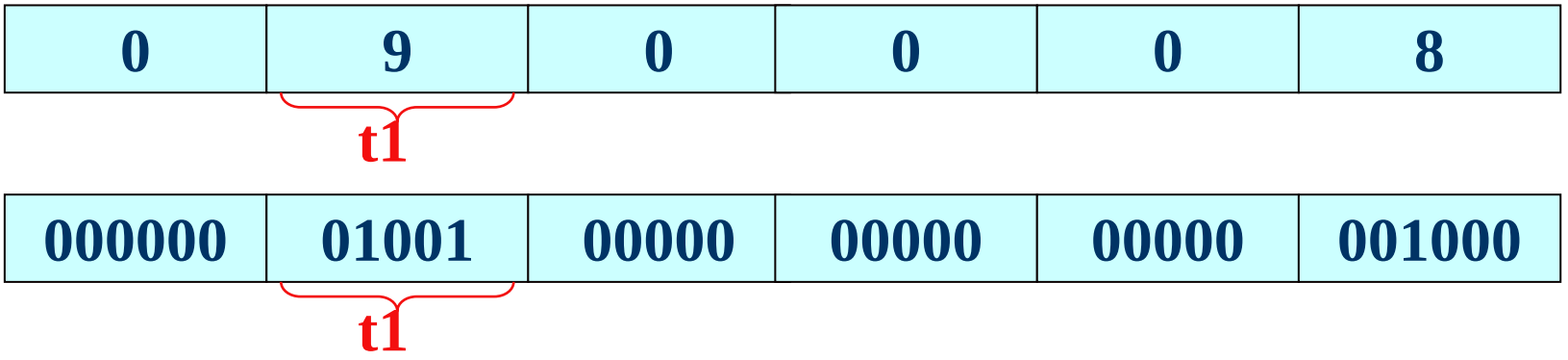
# Salto incondicional: jump register

**TIPO R**

Direccionamiento registro




**jr** **jr $t1** **# salto a la dirección almacenada en el registro t1**



# Salto condicional: branch if equal

**TIPO I** **beq $s1, $s2, 25** **# si ($s1 == $s2) avanzar 100**


**Salto = 0000 0000 0001 1001**

**PC anterior = 0100 1101 0000 0011 1010 1100 0101 1000**

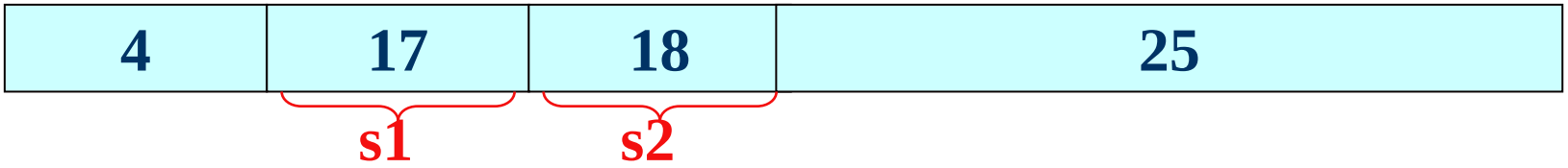
**PC nuevo = 0100 1101 0000 0011 1010 1100 0101 1000**
**0000 0000 0000 0000 0000 0000 0110 0100**
**= 0100 1101 0000 0011 1010 1100 1011 1100**

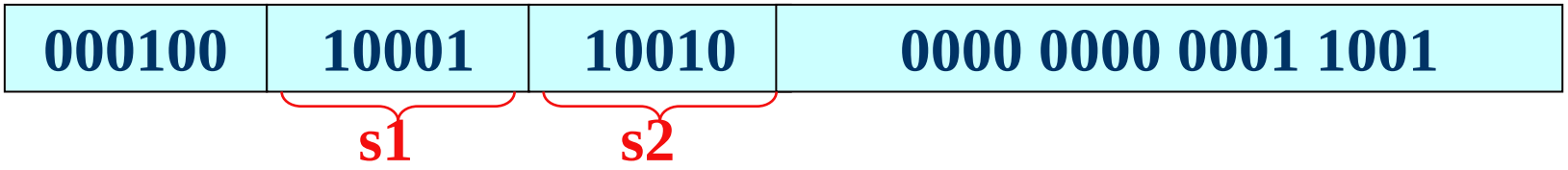
# Salto condicional: branch if equal

**TIPO I**

Direccionamiento relativo al PC


**beq** **beq $s1, $s2, 25** **# si ($s1 == $s2) avanzar 100**



# Salto condicional (hacia adelante)

```
      beq reg1, reg2, salto

Salto = 0000 0000 0010 0111
PC    = 0100 1101 0000 0011 1010 1100 0101 1000


        0100 1101 0000 0011 1010 1100 0101 1000
      + 0000 0000 0000 0000 0000 0000 1001 1100
      ---------------------------------------
        0100 1101 0000 0011 1010 1100 1111 0100
```

# Salto condicional (hacia atrás)

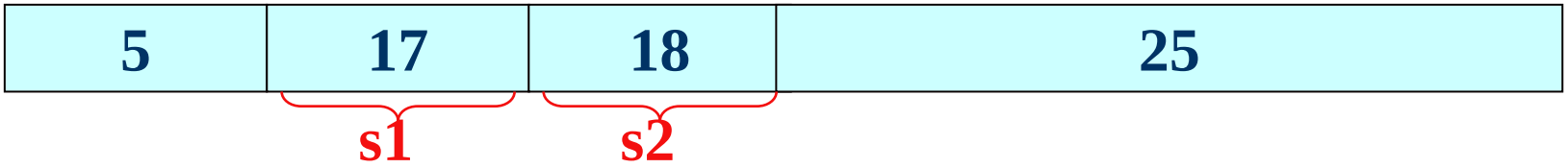
```
      beq reg1, reg2, salto

Salto = 1111 0000 0010 0111
PC    = 0100 1101 0000 0011 1010 1100 0101 1000


        0100 1101 0000 0011 1010 1100 0101 1000
      + 1111 1111 1111 1111 1100 0000 1001 1100
        ---------------------------------------
        0100 1101 0000 0011 0110 1100 1111 0100
```

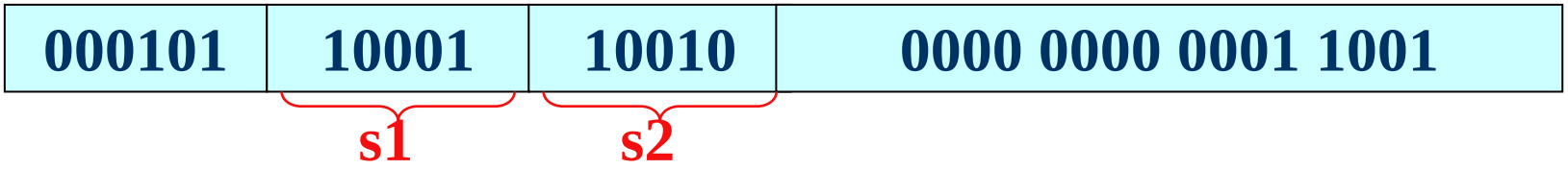
# Salto condicional: branch if not equal

**TIPO I**

Direccionamiento relativo al PC


**bne** **bne $s1, $s2, 25** **# si ($s1 != $s2) avanzar 100**

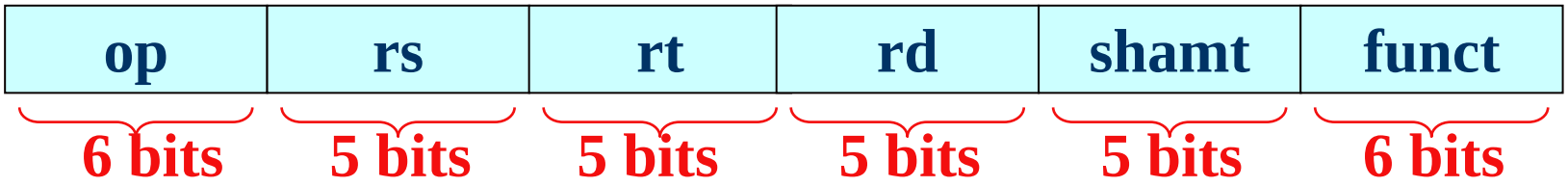


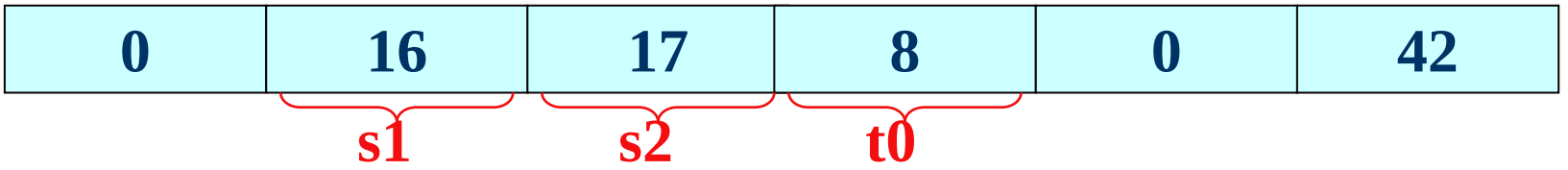
# Salto condicional: set on less than

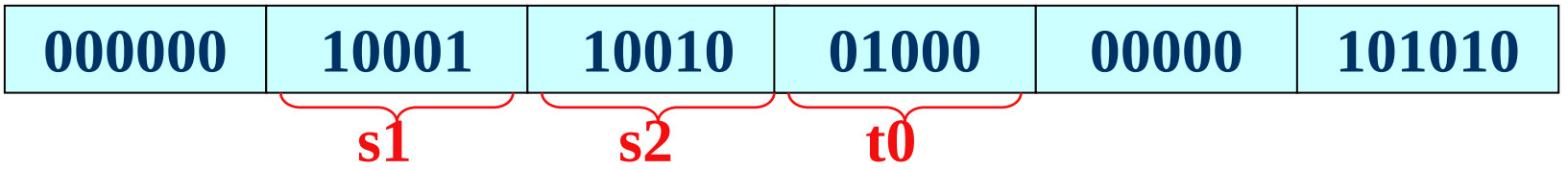
**TIPO R**

Direccionamiento registro




**slt** **slt $t0, $s1, $s2** # **if $s1 < $s2 then $t0=1 else $t0 = 0**

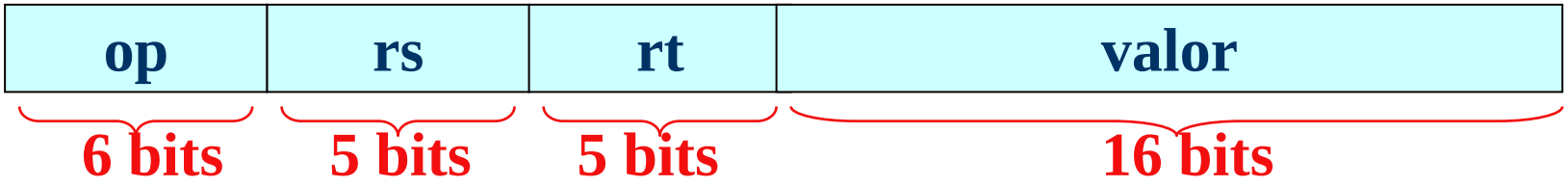


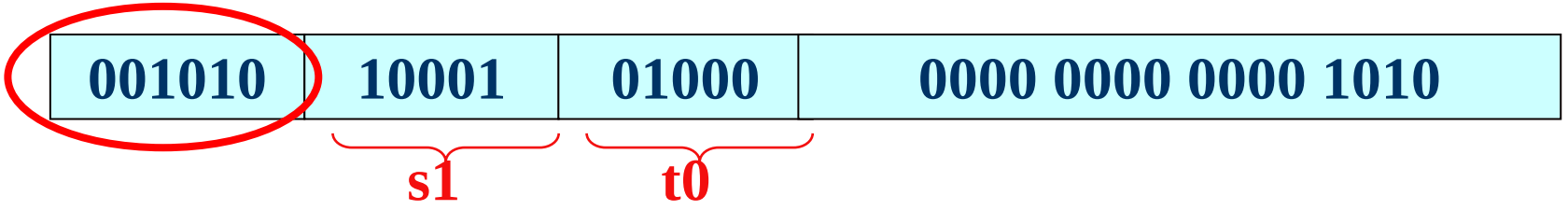
# Salto condicional: set on less than

- También hay una versión tipo I (modo direccionamiento inmediato)

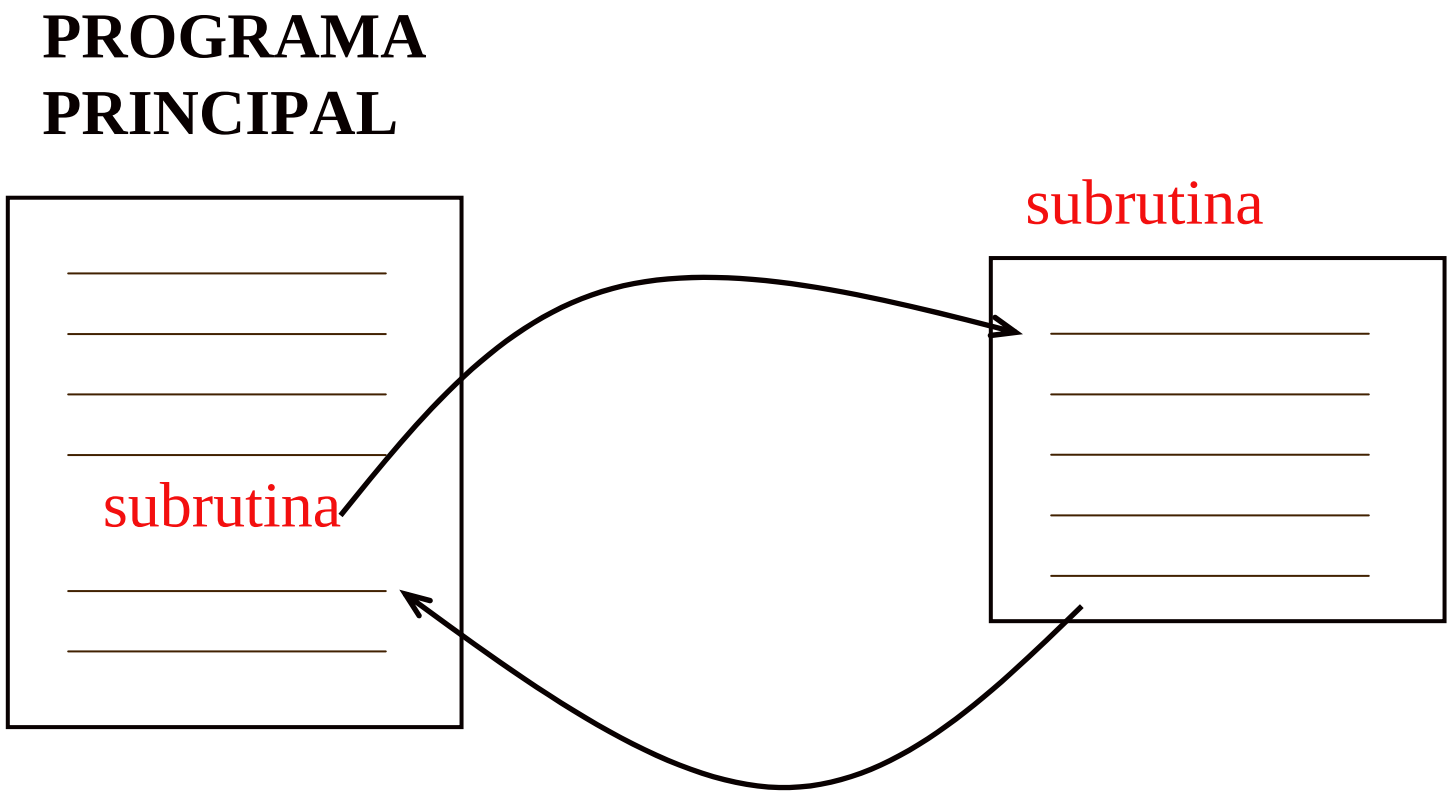

**TIPO I**




**slti** **slti $t0, $s1, 10** # **if $s1 < 10 then $t0=1 else $t0 = 0**



# Llamadas a subrutinas



# Llamadas a subrutinas

El programa debe realizar los siguientes pasos:

1. Facilitar parámetros de entrada a la subrutina.
2. Transferir el control a la subrutina (jump-and-link)
3. Ejecutar la subrutina
4. Situar resultados en un lugar accesible por el programa
5. Retornar el control al punto de origen.
6. Leer/utilizar los resultados

# Llamadas a subrutinas:jump-and-link (jal)

- jal funciona como la instrucción **`jump`**
- Guarda la *dirección de retorno (dirección siguiente a la dirección de la instrucción de salto)* en el registro 31 (**`$ra`**)
- Para pasar valores y recoger resultados:
  - **`$a0 - $a3`** parámetros de entrada
  - **`$v0 - $v1`** resultados

# Llamadas a subrutinas: jump-and-link

**TIPO J**

Direccionamiento pseudodirecto


**jal** **jal 2500** **# $ra=PC + 4;**

**Ir a subrutina en 10000**

**PC anterior = 0000 xx xxxx xxxx xxxx xxxx xxxx xxxx xx**

**PC nuevo = 0000 00 0000 0000 0000 1001 1100 0100 00**

| 3 | 2500 |
|---|---|

| 000011 | 00 0000 0000 0000 1001 1100 0100 |
|---|---|

# Llamadas a subrutinas

El programa debe realizar los siguientes pasos:

1. Facilitar parámetros de entrada a la subrutina ($a0-$a3).
2. Transferir el control a la subrutina (jal dirección_subrutina)
3. Ejecutar la subrutina
4. Situar resultados en un lugar accesible por el programa ($v0-$v1)
5. Retornar el control al punto de origen (jr $ra)
6. Leer/utilizar los resultados

**Si no son suficientes los 4 registros como argumentos de entrada y los 2 registros de retorno de valores : PILA o STACK**.

# Pila o Stack

- Es una parte de la memoria que implementa una estructura LIFO (last in first out)
- El registro 29 (**$sp**) es el puntero de pila
- La pila crece de direcciones de memoria superiores a inferiores:
  - Al poner un dato (PUSH) se debe decrementar **$sp** en 4 bytes
  - Al quitar un dato (POP) se debe incrementar **$sp** en 4 bytes

- Push (apilar)
  - **addi $sp, $sp, -4 # Ajustar el puntero de pila**
  - **sw $t0, 0($sp) # Guardo el registro $t0 en la pila**

- Pop (desapilar)
  - **lw $t0, 0($sp) # Contenido de la pila al registro $t0**
  - **addi $sp, $sp, 4 # Ajustar el puntero de pila**

# Pila o Stack: Guardando datos al llamar a una subrutina.

**Recuperando el estado del sistema al llamar a una subrutina:**

- **Al invocar desde un programa a una subrutina:**
  - **Guardar los valores de los registros en una pila.**
- **Al regresar al programa:**
  - **Recuperar los valores de los registros de la pila.**

**Convenios para guardar/restaurar registros:**

- **Guarda el invocador: el procedimiento invocador es el responsable de guardar/restaurar los registros a conservar.**
- **Guarda el invocado: el invocado es el responsable**

# Pila o Stack. Ejemplo de utilización

- Vamos a ver como funcionan los dos convenios para guardar/restaurar registros con el siguiente ejemplo:

```
clear_array(int A[], int n){
  int j;
  for(j=0; j<n; ++j)
     A[j]=0;
}
```

$a0= dirección array A
$a1= valor de n

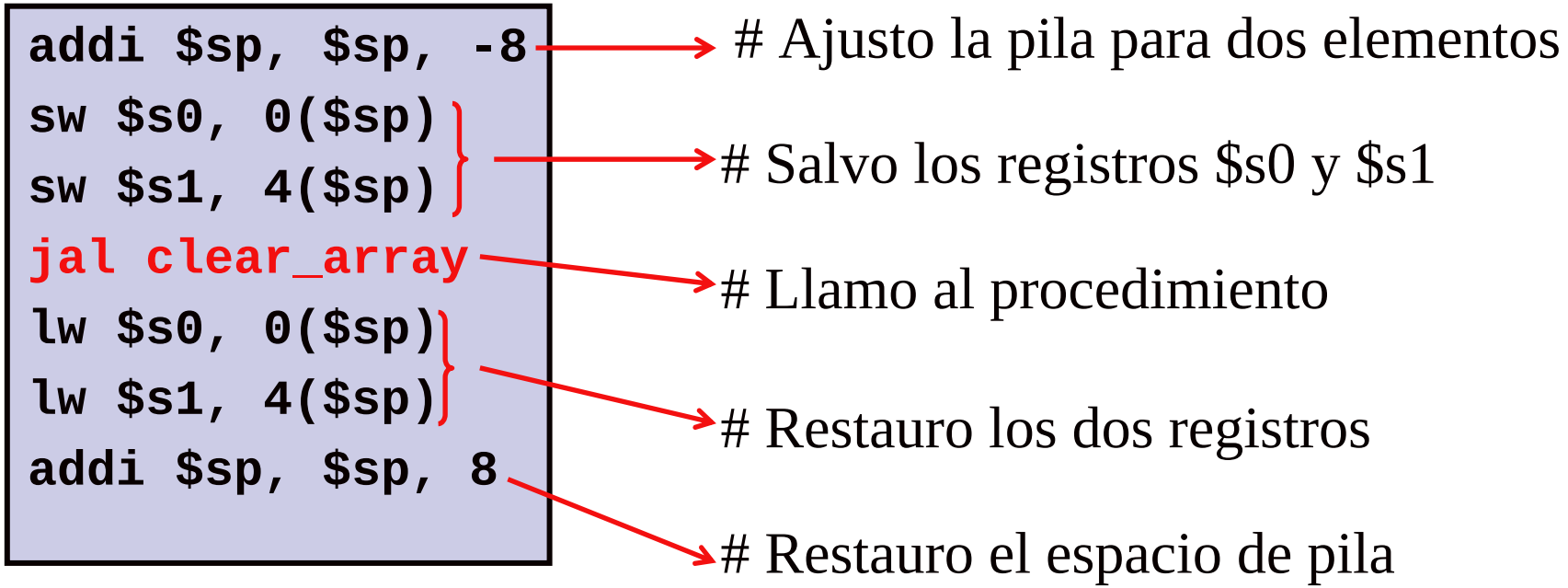
# Pila o Stack. Ejemplo de utilización. Guarda el invocador

- Salvamos 2 registros en la pila, saltamos a la subrutina y recuperamos al volver

```
addi $sp, $sp, -8
sw $s0, 0($sp)
sw $s1, 4($sp)
jal clear_array
lw $s0, 0($sp)
lw $s1, 4($sp)
addi $sp, $sp, 8
```

```
clear_array:
   add $s0, $zero, $zero
   add $s1, $zero, $a0
for:
   beq $s0, $a1, exit
   sw $zero, 0($s1)
   addi $s1, $s1, 4
   addi $s0, $s0, 1
   j for
exit:
        jr $31
```

# Pila o Stack. Ejemplo de utilización. Guarda el invocador



# Pila o Stack. Ejemplo de utilización. Guarda el invocador

```
clear_array:
   add $s0, $zero, $zero
   add $s1, $zero, $a0
for:
   beq $s0, $a1, exit
   sw $zero, 0($s1)
   addi $s1, $s1, 4
   addi $s0, $s0, 1
   j for
exit:
      jr $ra
```

# $s0= j=0

# $s1 dirección A[0]

# Saltamos a exit si j=N

# A[j] = 0

# Dirección A[j+1]

# j++

# retorno al programa invocador

# Pila o Stack. Ejemplo de utilización. Guarda el invocado

```
clear_array:
   addi $sp, $sp, -8
   sw $s0, 0($sp)
   sw $s1, 4($sp)

   add $s0, $zero, $zero
   add $s1, $zero, $a0
```

```
for:
   beq $s0, $a1, exit
   sw $zero, 0($s1)
   addi $s1, $s1, 4
   addi $s0, $s0, 1
   j for

exit:
   lw $s0, 0($sp)
   lw $s1, 4($sp)
   addi $sp, $sp, 8
   jr $ra
```

# Pila o Stack: Guardando datos al llamar a una subrutina.

**Llamada a subrutina:**
Los registros **$t0-$t9** son libres de ser modificados
Los registros **$s0-$s7** deben ser preservados
**PILA o STACK**.

## Programas MIPS:

- **$t0-$t9: “registros temporales” que NO son preservados por el invocado. Si los quiere preservar, lo tiene que hacer el invocador.**
- **$s0-$s7: “registros salvados” que SI deben ser preservados por el invocado en el caso de que los use.**

# Subrutinas anidadas

- Problemas en los anidamientos:
  - Dirección de retorno en **`$ra`**
  - Valores de los registros: argumentos ($a0-$a3), temporales ($t0-t9), salvados ($s0-s7).
- Solución: Guardar en pilas.
- Es necesario un convenio único para coordinación entre programadores.
- La subrutina invocada:
  - La subrutina invocada guarda los registros **`$s0-$s7`** que vaya a utilizar
  - La subrutina invocada debe guardar en la pila el registro **`$ra`**
  - Así puede llamar a otra subrutina con **`jal`**
  - Antes de retornar al procedimiento invocador es necesario recuperar **`$ra`** de la pila
- El invocador:
  - El invocador guarda los registros **`$t0-$t9`** y **`$a0-$a3`** que necesite

# Subrutinas anidadas. Ejemplo de utilización

- Vamos a analizar el siguiente ejemplo:

```
factorial(n){
        int tmp;
        if(n<2)
                tmp = 1;
        else
                tmp = n * factorial(n-1);
        return tmp;
}
```

$a0= valor de n
$t0= tmp
Resultado estará en $v0$

# Subrutinas anidadas. Ejemplo de utilización

```
factorial:
   addi $sp, $sp, -8
   sw $a0, 0($sp)
   sw $ra, 4($sp)

   slti $t0, $a0, 2
   beq $t0, $zero, else
if:
   addi $v0, $zero, 1
   addi $sp, $sp, 8
   jr $ra
```

```
else:
   addi $a0, $a0, -1
   jal factorial

   lw $a0, 0($sp)
   lw $ra, 4($sp)
   addi $sp, $sp, 8

   mult $v0, $v0, $a0

   jr $ra
```

# Subrutinas anidadas. Ejemplo de utilización

```
factorial:
   addi $sp, $sp, -8
   sw $a0, 0($sp)
   sw $ra, 4($sp)

   slti $t0, $a0, 2
   beq $t0, $zero, else
if:
   addi $v0, $zero, 1
   addi $sp, $sp, 8
   jr $ra
```

#ajusto la pila para 2 elementos

# salvo el argumento n

# salvo la dirección de retorno

# comprueba si n<2

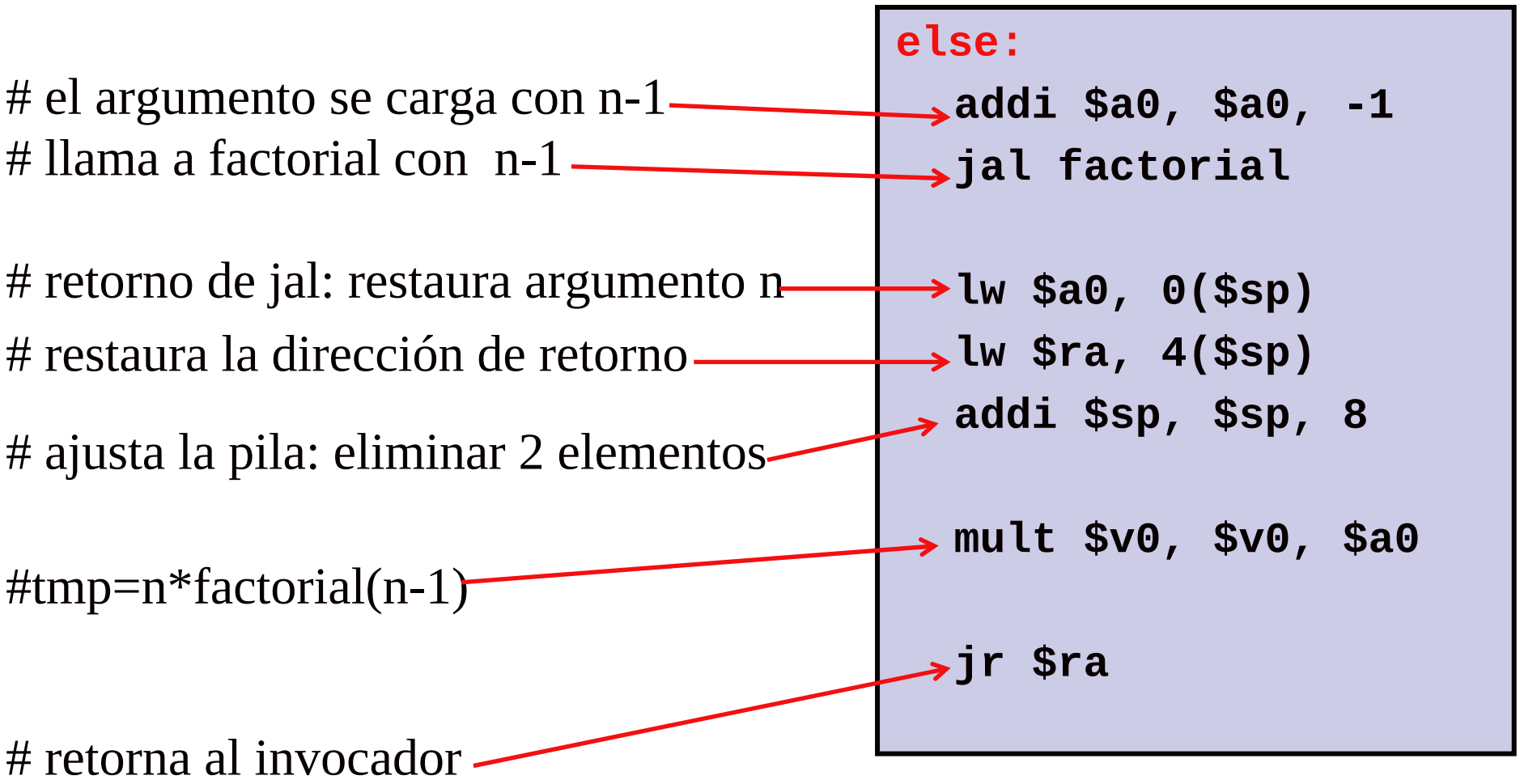
# si no se cumple: ir a “else”

# tmp=1

# elimina dos elementos de la pila

Retornar al punto después de jal.

# Subrutinas anidadas. Ejemplo de utilización



# Almacenamiento de caracteres: Representación ASCII, 8 bits

- Las instrucciones **lb** y **sb** permiten cargar/almacenar 8 bits
- **lb** carga 1 byte de memoria y lo sitúa en los 8 bits más a la derecha de un registro (hace extensión de signo en los bits más significativos)
- **sb** almacena en memoria los 8 bits menos significativos del registro
- Existe **lbu**, sin extensión de signo

```
lb      lb $t0, 0($s0)      # Leer byte de la fuente
sb      sb $t0, 0($s0)      # Escribir byte en destino
```