







Assembler

Ing. Alejandro C. Rodríguez Costello
Ing. Walter Lozano

*“Todo programa hace algo perfectamente bien,
aunque no sea exactamente lo que nosotros queremos que haga.”
Roger Pressmann*

Objetivos

-  **Estudiar los conceptos fundamentales para la programación en lenguaje ensamblador usando como ejemplo el procesador MIPS R2000.**
-  **Introducir al alumno en la programación en assembly.**
-  **Entender que soportes provee la arquitectura para implementar estructuras básicas de datos.**
-  **Administrar correctamente la pila.**

Programación en ensamblador

 **Lenguaje ensamblador es la representación simbólica de la codificación binaria de un computador (lenguaje máquina)**

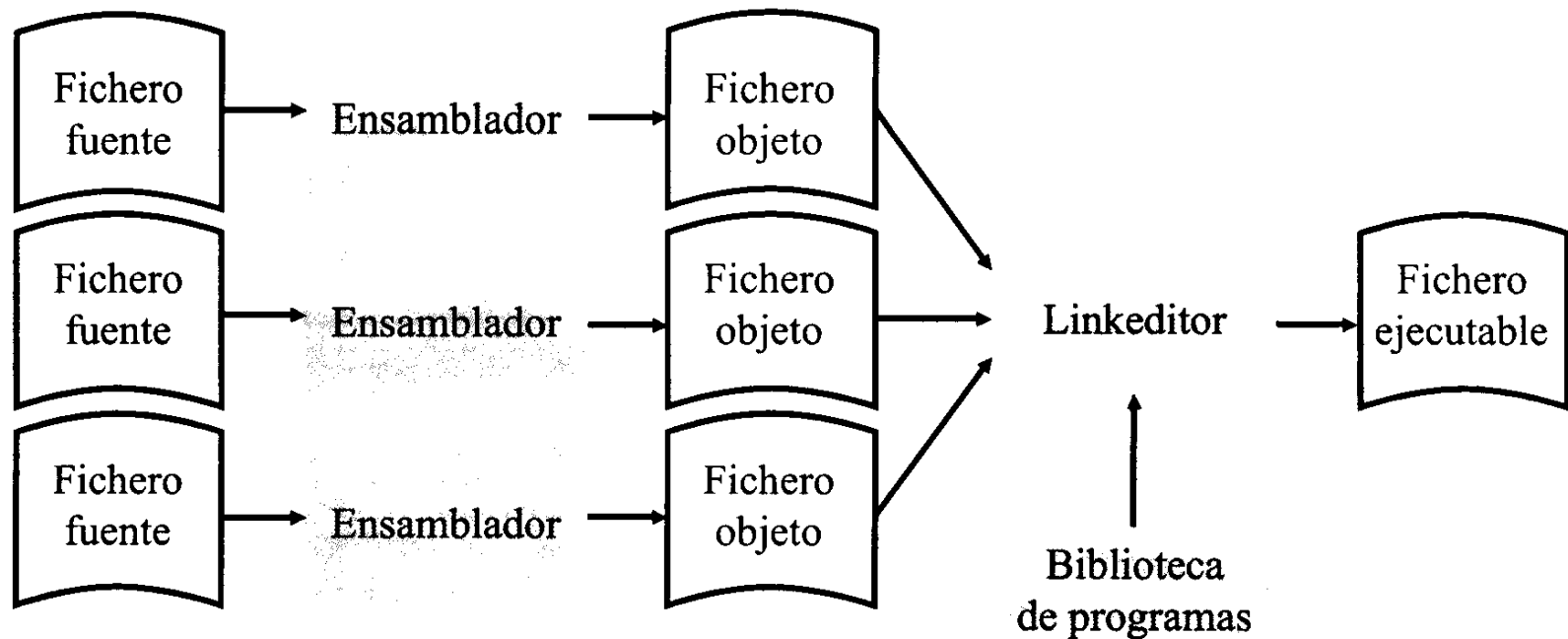
- ↳ Utilización de nemotécnicos para los códigos de operación y registros
- ↳ Utilización de etiquetas para identificar y dar nombre a palabras de memoria que almacenan datos e instrucciones
- ↳ Recursos de programación
 - ↳ Directivas
 - ↳ Pseudoinstrucciones
 - ↳ Definición de macros

 **Desventajas de la programación en ensamblador**

- ↳ Dependencia de la especificación de la máquina
- ↳ Lenguaje no estructurado
- ↳ No permite especificar diferentes tipos de datos

Programación en ensamblador

✍ **Ensamblador es un programa que traduce programas escritos en lenguaje ensamblador a lenguaje máquina**



Programación en ensamblador

El lenguaje ensamblador tiene dos funciones

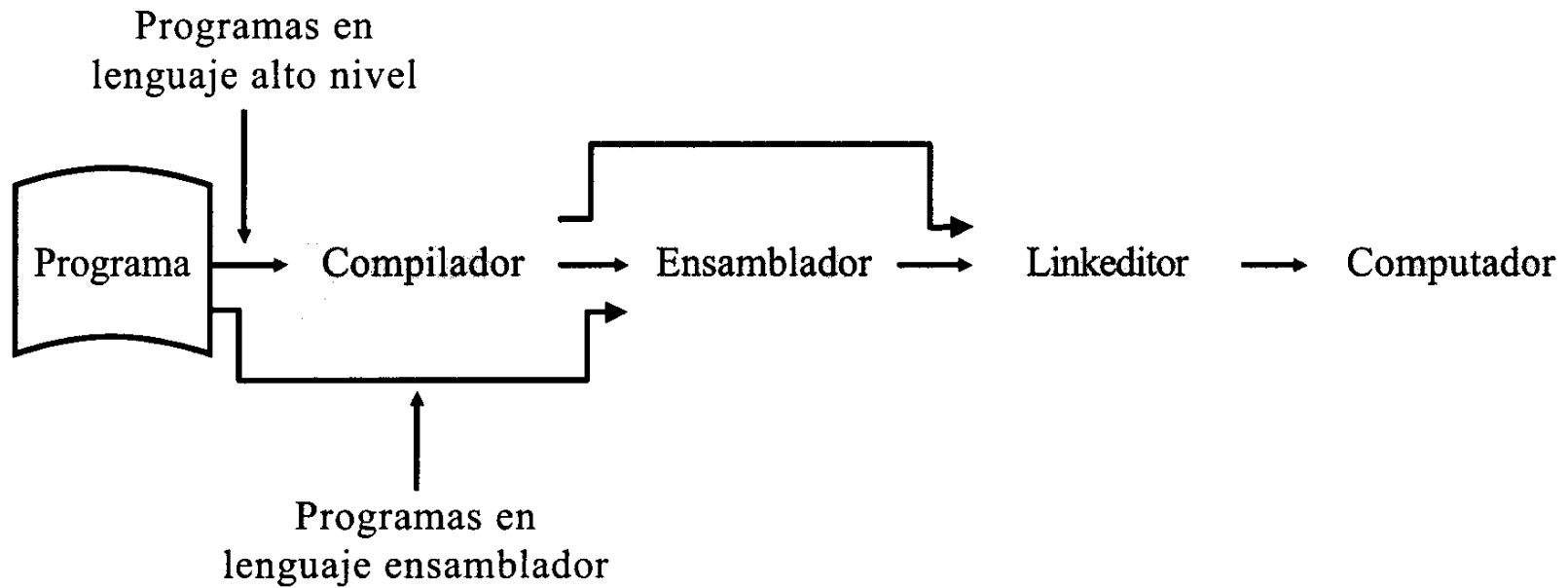
↳ Lenguaje de salida de los compiladores

↳ Escribir programas donde son críticos

↳ la velocidad de ejecución

↳ tamaño


↳ explotar recursos de la arquitectura



Directivas

 **Las directivas del ensamblador sirven para ubicar los datos y las instrucciones en la memoria del procesador**

 **Sintaxis:**

 Sus nombres empiezan con un punto y se emplean corchetes para indicar que uno o varios argumentos son opcionales

 **Se agrupan en tres tipos:**

 Directivas de propósito variado

 Directivas para reserva de posiciones de memoria

 Directivas para indicar el inicio del área de datos y de instrucciones

Directivas

Directivas de propósito variado:

Directiva *.globl*

 Sintaxis: **.globl etiqueta**

 Descripción:

- ✓ La función de esta directiva es declarar nombres de etiquetas o de funciones de forma global
- ✓ De esta manera desde un fichero se podrá acceder a etiquetas que se encuentran definidas en otros ficheros

Directiva *.end*

 Sintaxis: **.globl etiqueta**

 Descripción:

- ✓ Indica el final de un programa codificado en ensamblador

Directivas

Directivas para reserva de memoria:

Directiva `.space`

 Sintaxis: `.space n`

 Descripción:

- ✓ Reserva un número n de posiciones de memoria de tamaño de 8 bits (1 byte) inicializándolas a cero
- ✓ Reserva memoria a partir de cualquier posición

Directiva `.ascii` y `.asciiz`

 Sintaxis: `.ascii cadena [, cadena2, cadena3, ... cadenan]`

 Sintaxis: `.asciiz cadena [, cadena2, cadena3, ... cadenan]`

 Descripción:

- ✓ Se utiliza para almacenar una cadena de caracteres en memoria. (la cadena debe estar entre ")
- ✓ La cadena finaliza con un caracter nulo si se utiliza la directiva `.asciiz`
- ✓ Reservan memoria a partir de cualquier dirección

Directivas

Directivas para reserva de memoria:

Directiva `.byte`

 Sintaxis: `.byte b [, b2, b3, ..., bn]`

 Descripción:

- ✓ Inicializa posiciones consecutivas de memoria con enteros en complementos a 2 de 8 bits (1 byte)
- ✓ Reserva memoria a partir de cualquier posición

Directiva `.word`

 Sintaxis: `.word w [, w2, w3, ..., wn]`

 Descripción:

- ✓ Inicializa posiciones consecutivas de memoria con enteros en complementos a 2 de 32 bits (4 bytes)
- ✓ Reservan memoria a partir direcciones múltiplos de 4

Directivas

Directivas para inicio de datos e instrucciones:


Directiva *.data*

 Sintaxis: **.data [dir]**

 Descripción:

- ✓ Sirve para ubicar datos a partir de la dirección de memoria indicada por dir
- ✓ Si no se indica una dirección, por defecto se empieza en la dirección 0x10000000

Directiva *.text*

 Sintaxis: **. text [dir]**

 Descripción:

- ✓ Sirve para ubicar instrucciones a partir de la dirección de memoria indicada por dir
- ✓ Si no se indica una dirección, por defecto se empieza en la dirección 0x00400000

Directivas

Ejercicio:

```
.data 0x10000010
.space 6
.word 4, 5, 6, 7
.space 1
.byte 0x10, 0x20
.asciiz "hola"
.ascii "hola"
```

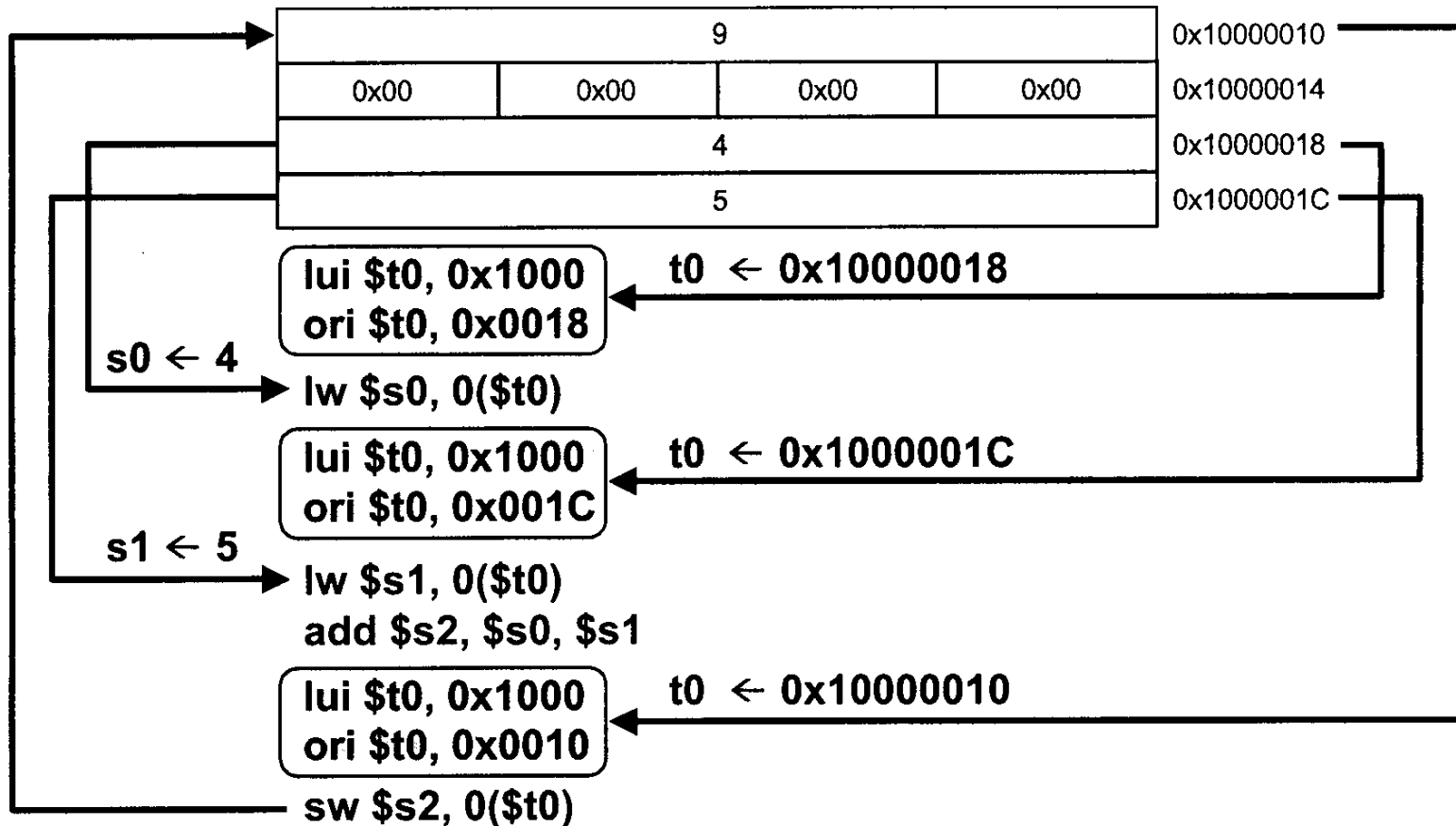
Contenido de la memoria				Dir Memoria
0x00	0x00	0x00	0x00	0x10000010
0x00	0x00	0x00	0x00	0x10000014
4				0x10000018
5				0x1000001C
6				0x10000020
7				0x10000024
"h"	0x20	0x10	0x00	0x10000028
0x00	"a"	"l"	"o"	0x1000002C
"a"	"l"	"o"	"h"	0x10000030

Escribir el fragmento de código de ensamblador del MIPS R2000 que permita realizar la siguiente operación:

$$M[0x10000010] \leftarrow M[0x10000018] + M[0x1000001C]$$

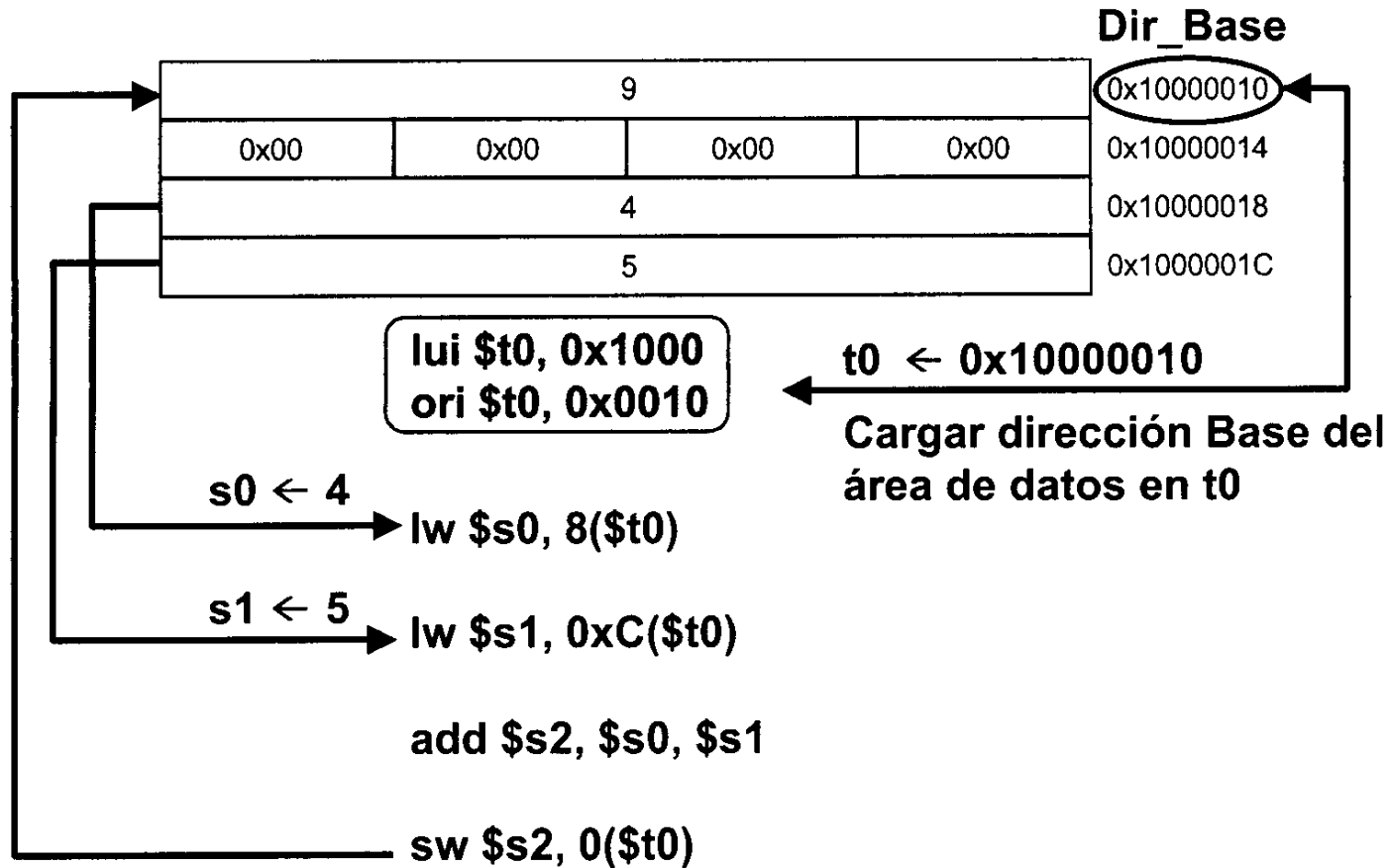
Directivas

**Solución A. Cargar la dirección efectiva de los operandos en un registro.
Referenciarlos a partir del modo base con desplazamiento: $EA = \$t0 + 0$**



Directivas

Solución B. Cargar la dirección base del área de datos en un registro (\$t0). Referenciarlos a partir del modo base con desplazamiento: EA=\$t0+desp



Directivas

Ejercicio:

.data 0x10000010
espacio: .space 6
tabla: .word 4, 5, 6, 7

Dir_base del área de datos → 0x10000000

0	0x10000010	+ 10
0	0x10000014	
4	0x10000018	+ 18
5	0x1000001C	+ 1C
6	0x10000020	
7	0x10000024	

Programa en
lenguaje ensamblador

```
lw    $s0, tabla($0)
lw    $s1, tabla+4($0)
add   $s2, $s0, $s1
sw    $s2, espacio($0)
```

```
lui    $1, 0x1000
lw     $s0, 0x18($1)
lw     $s0, 0x1C($1)
add    $s2, $s0, $s1
sw     $s0, 0x10($1)
```

\$1 ← 0x10000000
 \$s0 ← 4
 \$s1 ← 5
 M[0x1000001C] ← 9

Instrucciones ensambladas

Pseudoinstrucciones


- ✍ **Conjunto de instrucciones que el lenguaje ensamblador incorpora además de las instrucciones máquina del procesador**
- ✍ **El programador las puede usar como si se trataran de instrucciones máquina soportadas por el procesador**
- ✍ **El programa ensamblador se encarga de sustituirlas por las instrucciones máquina correspondientes (de una a cuatro por cada pseudoinstrucción)**

Pseudoinstrucciones

Pseudoinstrucciones de carga y almacenamiento

Pseudoinstrucción *li* (*load immediate*)


 Sintaxis: **li rd, inm**


 Descripción: ✓ Carga en el registro indicado por rd el entero con signo en complemento a 2 de 16 bits inm

 Ejemplo: **li \$t0, 32 ⇒ ori \$t0, \$s0, 32**

Pseudoinstrucción *la* (*load address*)

 Sintaxis: **la rd, etiqueta**

 Descripción: ✓ Carga en el registro rd la dirección de memoria a la que hace referencia etiqueta

 Ejemplo:


```
.data 0x10008000  
dato1: .word 1,2,3,4  
...  
.text  
main: la $s0, dato1 ⇒ lui $s0, 0x1000  
ori $s0, $s0, 0x8000
```



Pseudoinstrucciones

Pseudoinstrucciones de multiplicación y división

 Pseudoinstrucciones *mul* y *divu*

 Sintaxis: **mul rd, rs, rt** **divu rd, rs, rt**


 Descripción: ✓ Multiplica el contenido de los registros rs y rt y el resultado lo almacena en el registro rd (nº con signo)
✓ Pone el cociente de dividir el contenido del registro rs entre rt en el registro rd (nº con signo)

 Ejemplo: **mul \$s0, \$s1, \$s2** \Rightarrow $\begin{cases} \text{mult } \$s1, \$s2 \\ \text{mflo } \$s0 \end{cases}$

Pseudoinstrucciones de comparación

 Pseudoinstrucciones *seq*, *sge*, *sgt*, *sle* y *sne*

 Sintaxis: **sxx rd, rs, rt**

 Descripción: ✓ Si se cumple la relación de equivalencia, indicada por xx (eq, ge, gt, le y ne), entre los registros rs y rt, se almacena un 1 en el registro rd, si no se cumple, se almacena un 0 en el registro rd (nº con signo)


Ejemplo: **sge \$s1, \$s2, \$s3** \Rightarrow $\begin{cases} \text{slt } \$s1, \$s3, \$s2 \\ \text{bne } \$s3, \$2, +2 \\ \text{ori } \$s1, \$s0, 1 \end{cases}$


Pseudoinstrucciones


Pseudoinstrucciones de salto condicional

 Pseudoinstrucciones *bge*, *bgt*, *ble* y *blt*


 Sintaxis: **bxx rs, rt, etiqueta**

 Descripción: ✓ Si se cumple la relación de equivalencia, indicada por xx (ge, gt, le, lt), entre los registros rs1 y rs2, se salta a la dirección a la que hace referencia la etiqueta (números con signo)

 Ejemplos: **bgt \$s1, \$s2, sig** \Rightarrow $\begin{cases} \text{slt } \$t0, \$s2, \$s1 \\ \text{bne } \$t0, \$0, \text{sig} \end{cases}$

 Pseudoinstrucciones *bgez*, *bgtz*, *blez*, *bltz*, *beqz* y *bnez*

 Sintaxis: **bxxz rs, etiqueta**

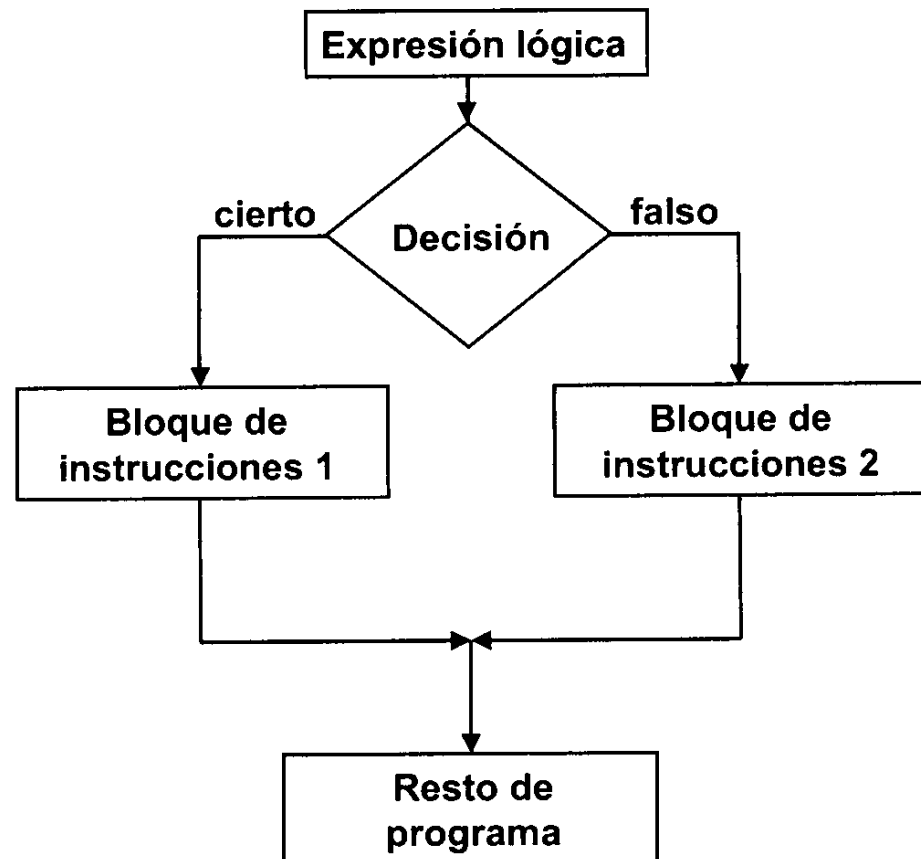
 Descripción: ✓ Si rs es (mayor o igual, mayor, menor o igual, menor, igual o distinto de cero) a cero según la condición indicada por xx (ge, gt, le, lt, eq, ne) el programa sigue su ejecución a partir de la etiqueta

 Ejemplos: **beqz \$s0, sig** \Rightarrow **beq \$s0, \$0, sig**

Estructuras de control

Estructura condicional if-then-else

➡ Evalúa una condición y si el resultados de la evaluación es cierto la ejecución del programa toma un camino, en caso contrario el programa ejecuta otro camino alternativo

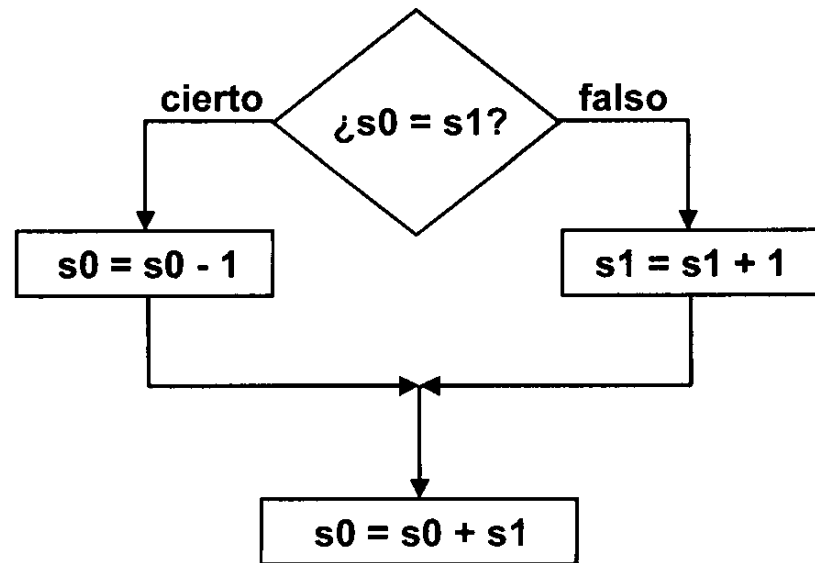


Estructuras de control

Ejemplo if-then-else

```
if (s0 == s1) then
    s0 = s0 - 1
else
    s1 = s1 + 1
endif
s0 = s0 + s1
```

```
beq $s0, $s1, cierto
bne $s0, $s1, falso
cierto: addi $s0, $s0, -1
        j finisi
falso:  addi $s0, $s0, 1
finisi: add $s0, $s0, $s1
```

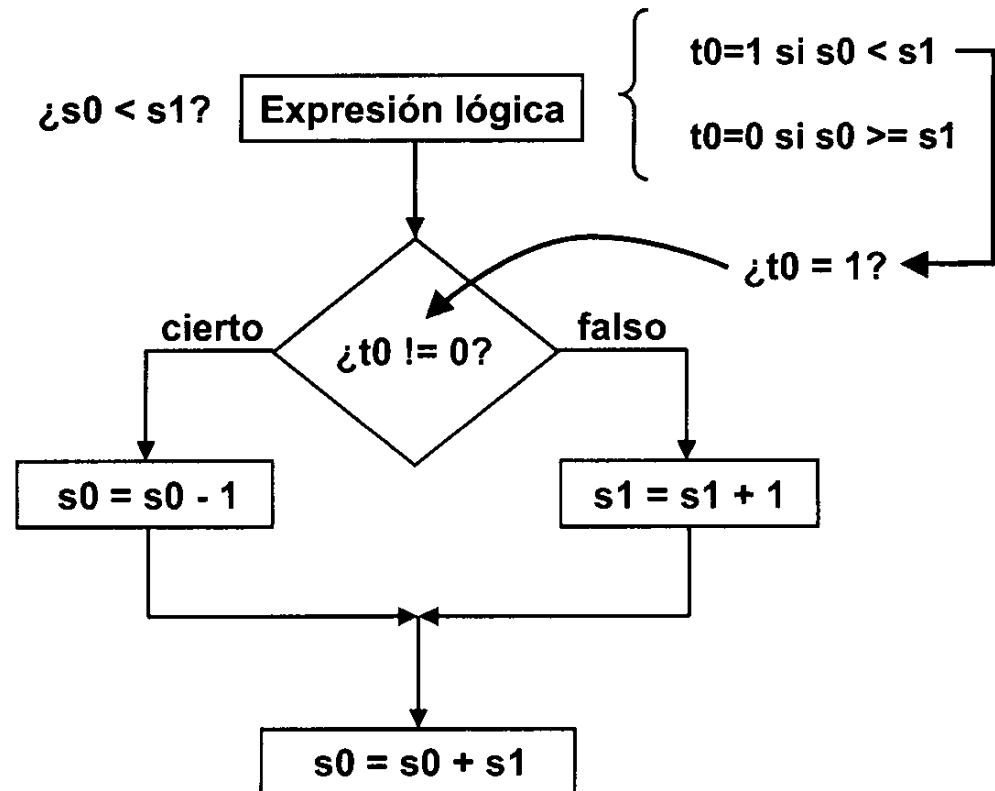


Estructuras de control

Ejemplo if-then-else

```
if (s0 < s1) then
    s0 = s0 - 1
else
    s1 = s1 + 1
endif
s0 = s0 + s1
```

```
slt $t0, $s0, $s1
bne $t0, $0, cierto
beq $t0, $0, falso
cierto: addi $s0, $s0, -1
j finisi
falso:  addi $s1, $s1, 1
finisi: add $s0, $s0, $s1
```

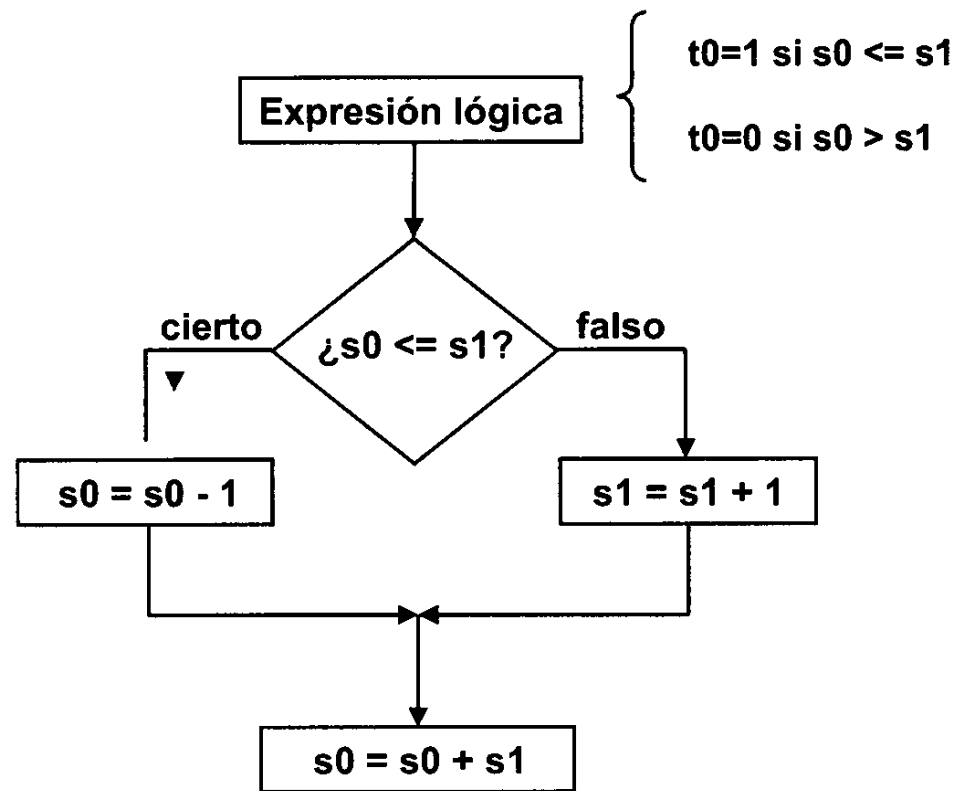


Estructuras de control

Ejemplo if-then-else

```
if (s0 <= s1) then
    s0 = s0 - 1
else
    s1 = s1 + 1
endif
s0 = s0 + s1
```

```
slt $t0, $s0, $s1
bne $s0, $s1, fineval
ori $t0, $t0, 1
fineval: bne $t0, $0, cierto
         beq $t0, $0, falso
cierto:  addi $s0, $s0, -1
         j  finisi
falso:   addi $s1, $s1, 1
finisi:  add $s0, $s0, $s1
```



Estructuras de control

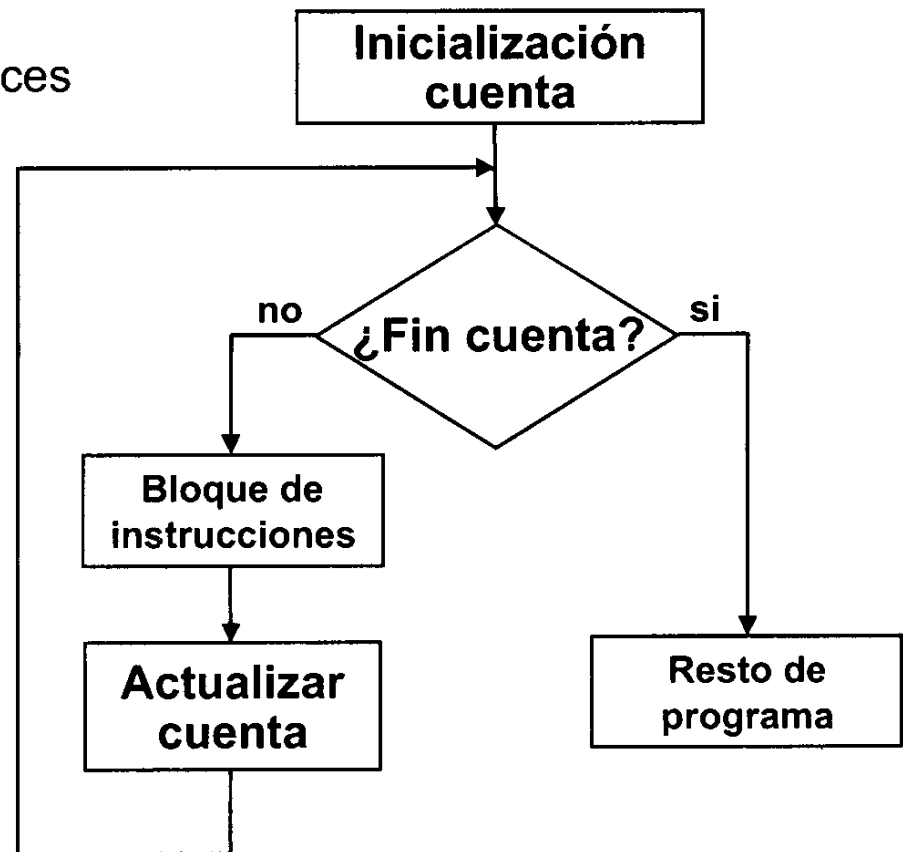
Estructura de repetición for

Un bloque de instrucciones se repite (bucle) un número de veces conocido a priori

Ejemplo

```
for i=1 to 10 do
    s0 = s0 + i
endfor
```

```
li $t0, 10
li $t1, 1
for:   bgt $t1, $t0, endfor
      add $s0, $s0, $t1
      addi $t1, $t1, 1
      j for
endfor:
```



Gestión de la pila

Pila o segmento de pila

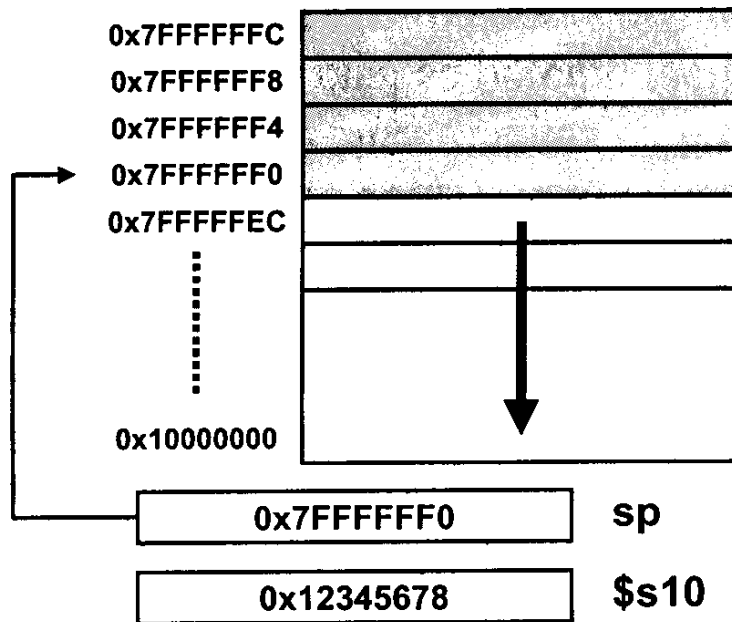
- ↳ Zona de memoria donde se almacenan datos siguiendo una estructura LIFO
- ↳ La pila siempre crece hacia posiciones decrecientes de memoria
- ↳ Cada dato ocupa 4 bytes en la pila
- ↳ La pila se gestiona mediante el registro \$29 (que también se puede referenciar como \$sp)
- ↳ El SP contiene, en cualquier instante, la dirección de memoria correspondiente a la posición que ocupa en la pila el último dato almacenado.
- ↳ Existen dos operaciones básicas sobre la pila
 - ↳ **Inserción:** Almacenar un nuevo dato en la pila (PUSH)
 - ↳ **Extracción:** Extraer un dato de la pila (POP)

Gestión de la pila

✍ Operación de inserción (Apilar/push)

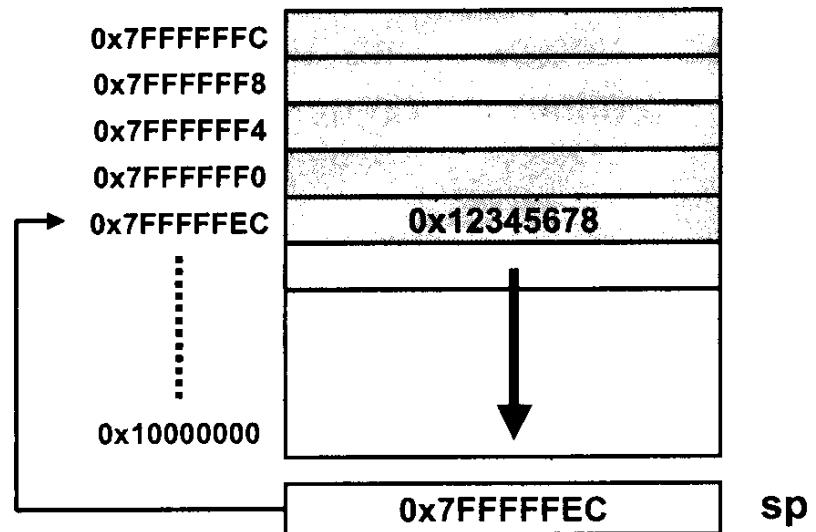
- ✍ Consiste en almacenar un nuevo dato en la pila
- ✍ El dato a almacenar siempre ocupará 4 bytes (1 palabra)
- ✍ Pasos a realizar:
 - ✍ Decrementar el puntero de pila en 4 bytes
 - ✍ Almacenar el dato en la nueva posición a la que apunta el puntero

Situación **antes de apilar** el registro \$s10 en la pila



Situación **después de apilar** el registro \$s10 en la pila, ejecutando el código:

```
addi $sp, -4  
sw $s10, 0($sp)
```

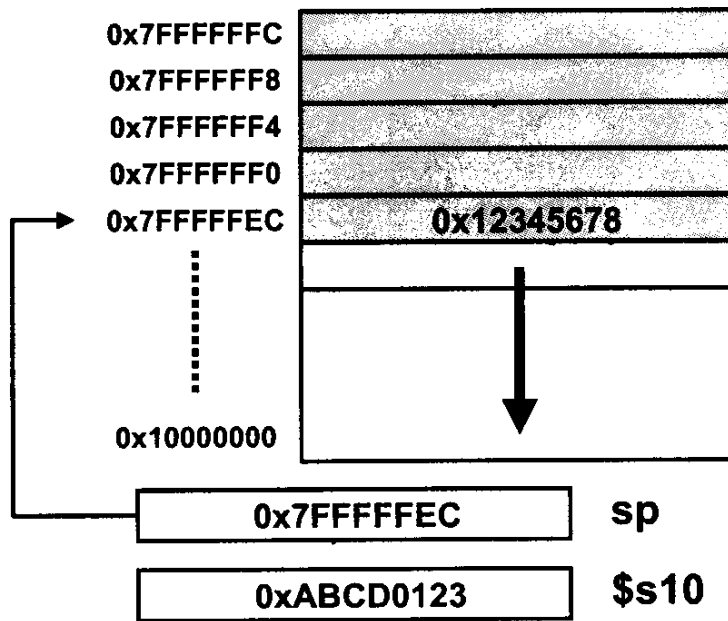


Gestión de la pila

Operación de extracción (Desapilar/pop)

- Consiste en almacenar un dato desde la pila en un registro
- Pasos a realizar:
 - Cargar el dato almacenado en la pila, en la posición apuntada por el puntero de pila, en un registro de propósito general.
 - Incrementar el puntero de pila en 4 unidades

Situación **antes de desapilar** la información en el registro \$s10



Situación **después de desapilar**,
ejecutando el código: `lw $s10, 0($sp)`
`addi $sp, 4`

