

Grupo ARCOS

uc3m | Universidad **Carlos III** de Madrid

Tema 3 (I)

Fundamentos de la programación en ensamblador

Estructura de Computadores
Grado en Ingeniería Informática

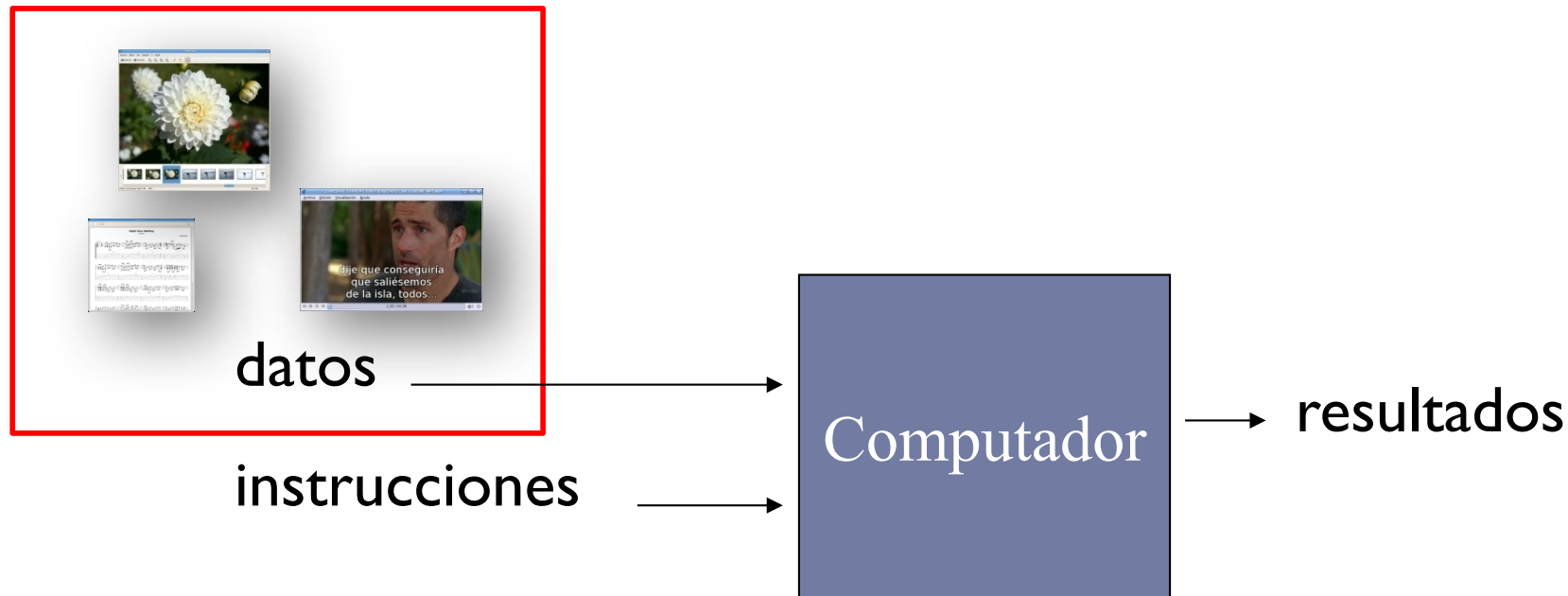


Contenidos

1. Fundamentos básicos de la programación en ensamblador
2. Ensamblador del MIPS 32, modelo de memoria y representación de datos
3. Formato de las instrucciones y modos de direccionamiento
4. Llamadas a procedimientos y uso de la pila

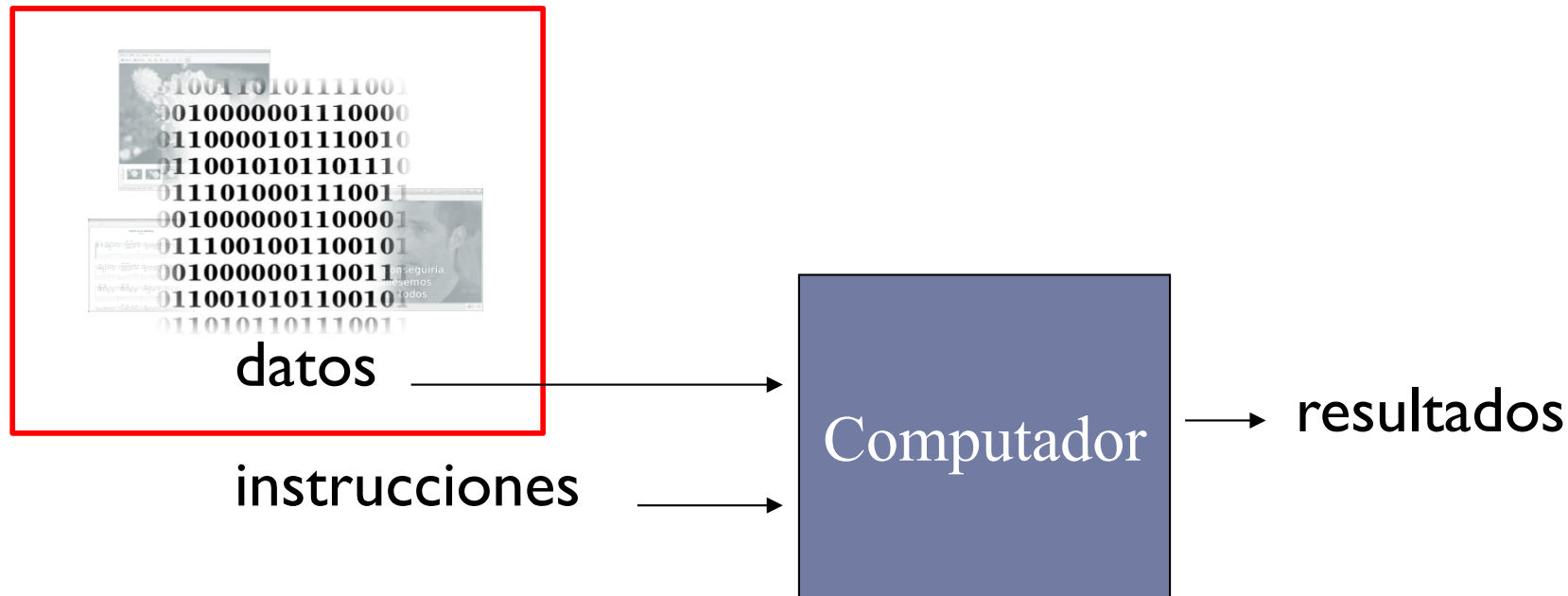
Tipos de información: instrucciones y datos

► Representación de **datos** ...



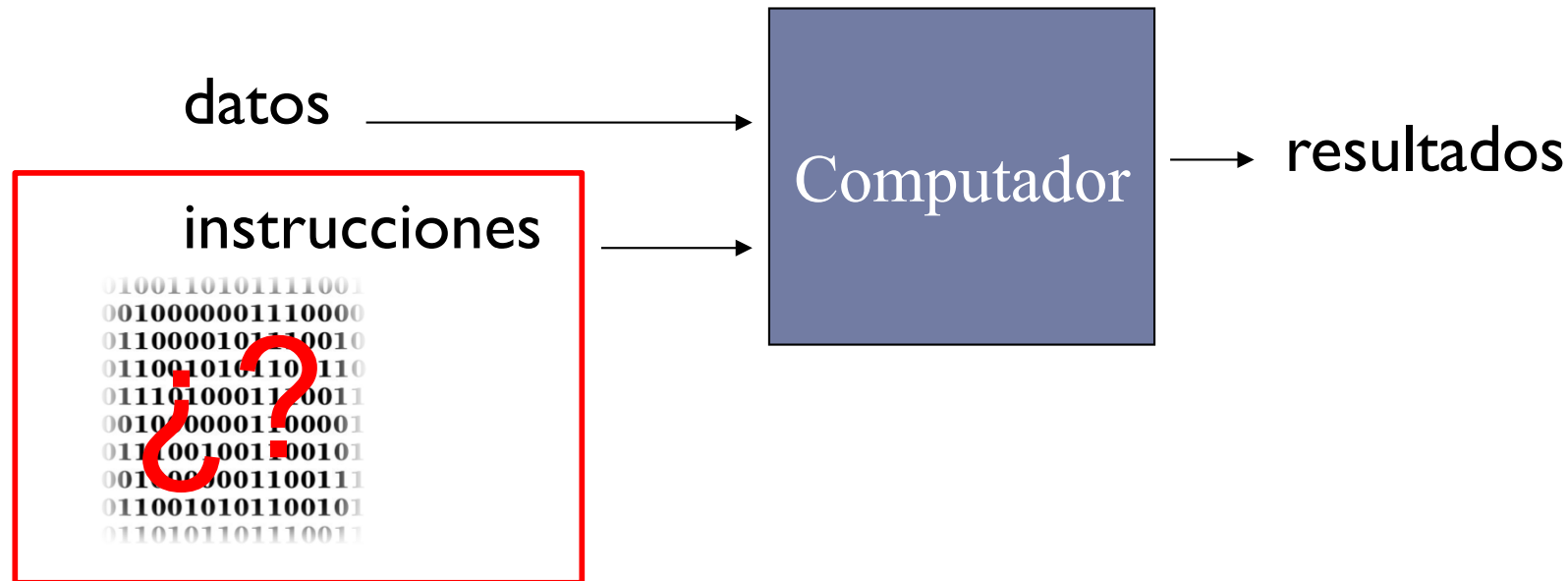
Tipos de información: instrucciones y datos

- Representación de **datos** en **binario**.



Tipos de información: instrucciones y datos

- ¿Qué sucede con las instrucciones?

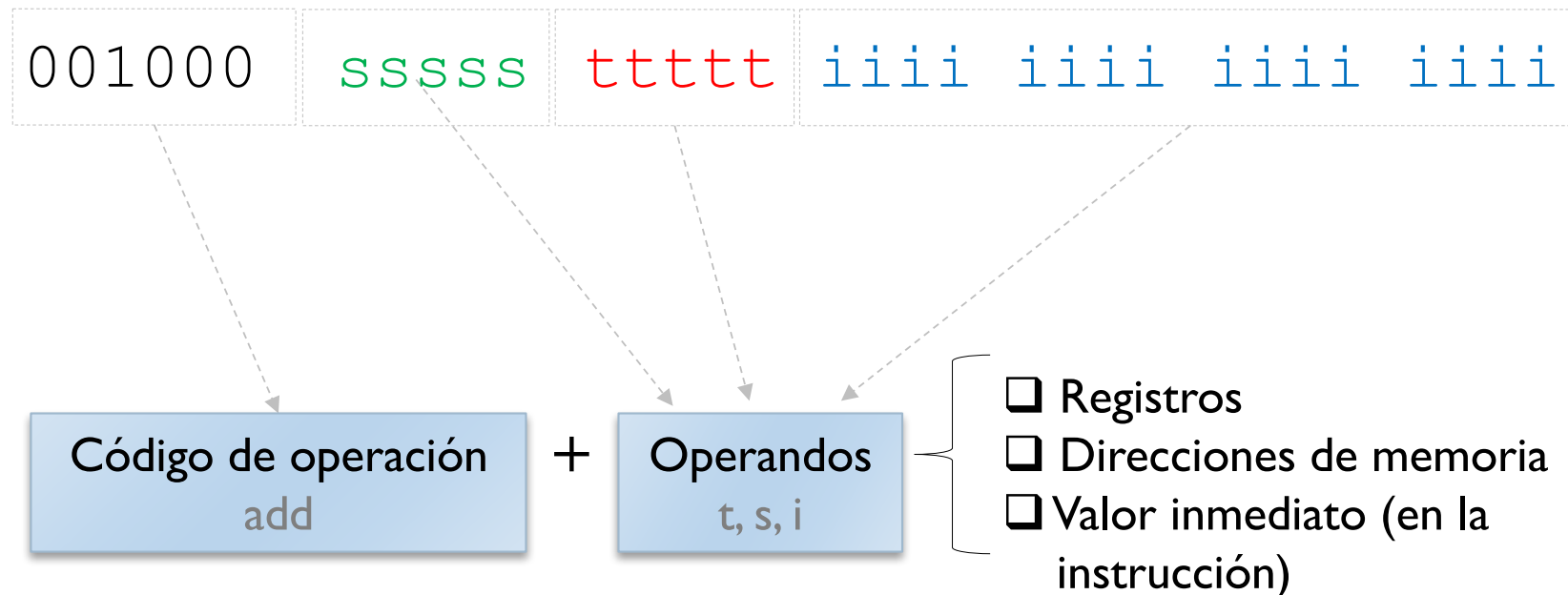


Modelo de programación de un computador

- ▶ Un computador ofrece un modelo de programación formando por:
 - ▶ **Juego de instrucciones (lenguaje ensamblador)**
 - ▶ ISA: Instruction set Architecture
 - ▶ Una instrucción incluye:
 - Código de operación
 - Otros elementos: identificadores de registros, direcciones de memoria o números
 - ▶ **Elementos de almacenamiento**
 - ▶ Registros
 - ▶ Memoria
 - ▶ Registros de los controladores de E/S
 - ▶ **Modos de ejecución**

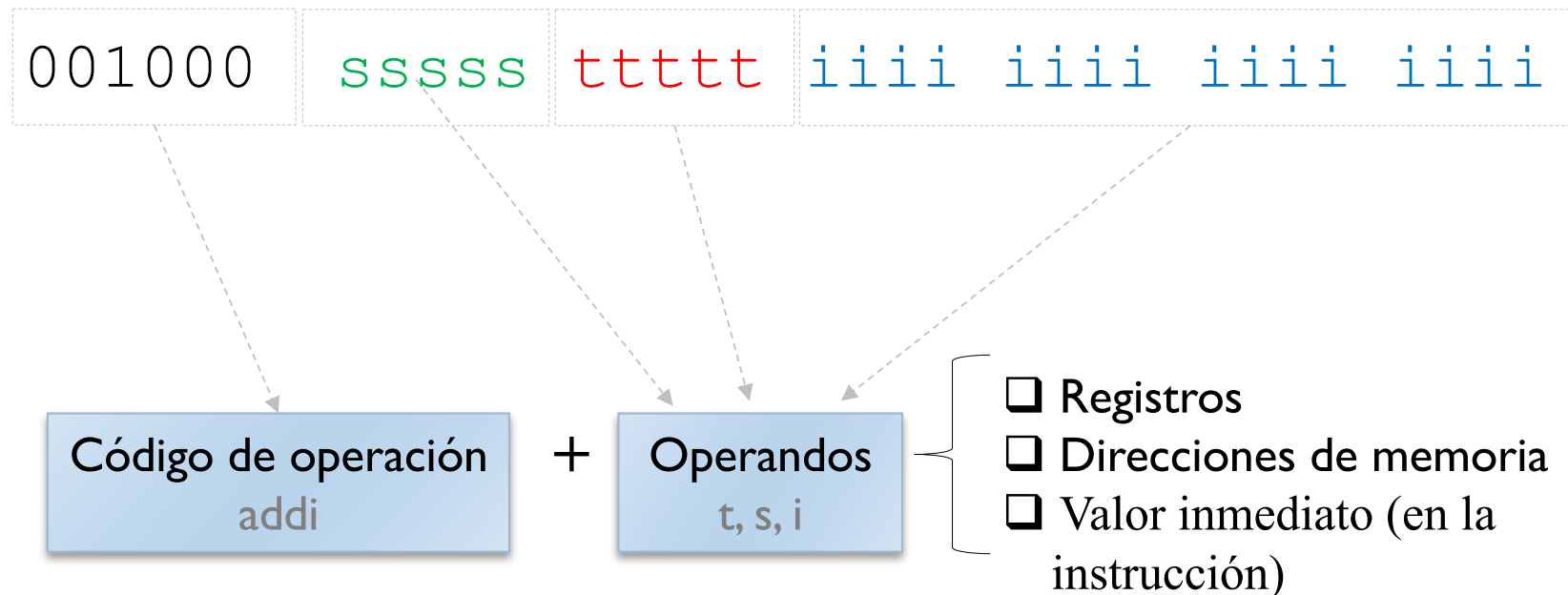
Instrucción máquina

- ▶ Instrucción máquina: operación elemental que puede ejecutar directamente el procesador
- ▶ Ejemplo de instrucción en MIPS:
 - ▶ Suma de un registro (s) con un valor inmediato (i) y el resultado de la suma se almacena en registro (t)



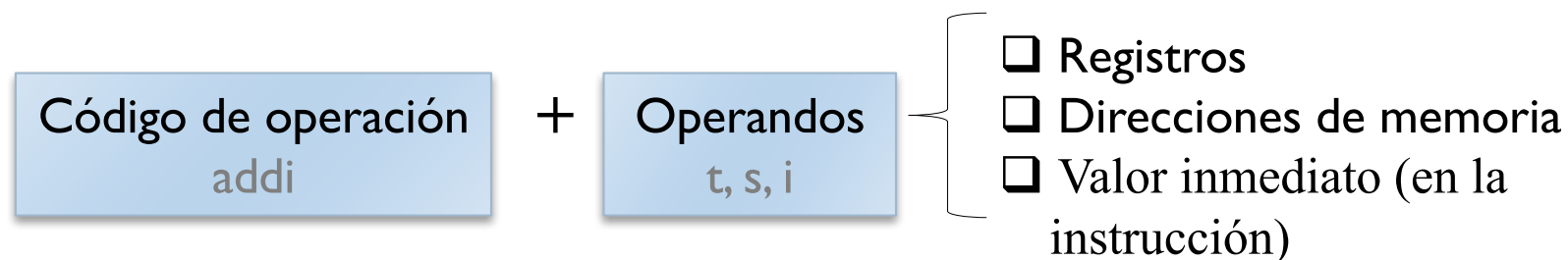
Propiedades de las instrucciones máquina

- ▶ Realizan una **única y sencilla tarea**
- ▶ Operan sobre un **número fijo de operandos**
- ▶ **Incluyen toda la información necesaria para su ejecución**



Información incluida en una instrucción máquina

- ▶ La **operación a realizar**.
- ▶ Dónde se encuentran los **operandos**:
 - ▶ En registros
 - ▶ En memoria
 - ▶ En la propia instrucción (inmediato)
- ▶ Dónde dejar los resultados (como operando)
- ▶ Una referencia a la siguiente instrucción a ejecutar
 - ▶ De forma implícita, la siguiente instrucción
 - ▶ Un programa es una secuencia consecutiva de instrucciones máquina
 - ▶ De forma explícita en las instrucciones de bifurcación (como operando)



Juego de instrucciones

- ▶ **Instruction Set Architecture (ISA)**
 - ▶ Conjunto de instrucciones de un procesador
 - ▶ Frontera entre el HW y el SW
- ▶ **Ejemplos:**
 - ▶ 80x86
 - ▶ MIPS
 - ▶ ARM
 - ▶ Power

Características de un juego de instrucciones

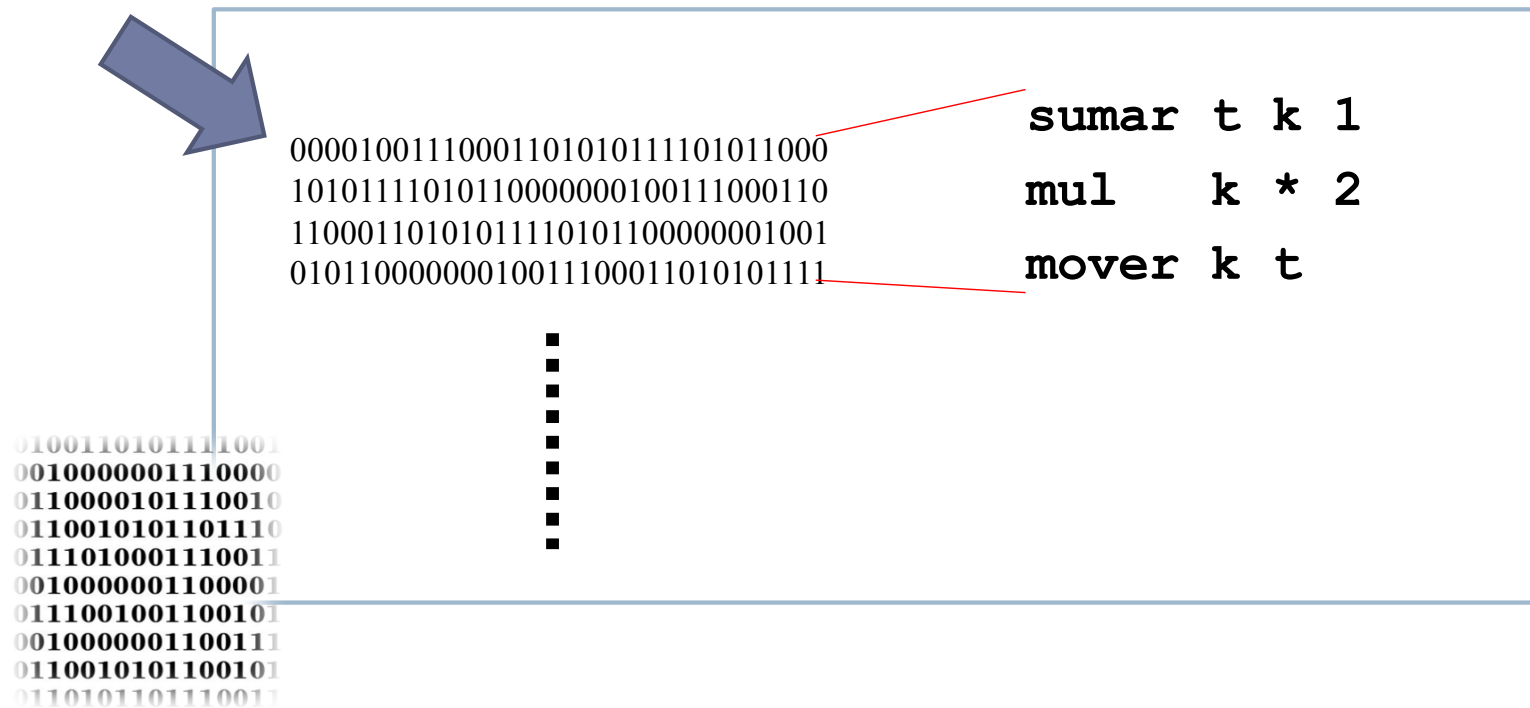
- ▶ **Operandos:**
 - ▶ Registros, memoria, la propia instrucción
- ▶ **Direccionamiento de la memoria**
 - ▶ La mayoría utilizan direccionamiento por bytes
 - ▶ Ofrecen instrucciones para acceder a elementos de varios bytes a partir de una determinada posición
- ▶ **Modos de direccionamiento**
 - ▶ Especifican el lugar y la forma de acceder a los operandos (registro, memoria o la propia instrucción)
- ▶ **Tipo y tamaño de los operandos**
 - ▶ bytes: 8 bits
 - ▶ enteros: 16, 32, 64 bits
 - ▶ números en coma flotante: simple precisión, doble,...

Características de un juego de instrucciones

- ▶ **Operaciones:**
 - ▶ Aritméticas, lógicas, de transferencia, control, ...
- ▶ **Instrucciones de control de flujo**
 - ▶ Saltos incondicionales
 - ▶ Saltos condicionales
 - ▶ Llamadas a procedimientos
- ▶ **Formato y codificación del juego de instrucciones**
 - ▶ Instrucciones de longitud fija o variable
 - ▶ 80x86: variable de 1 a 18 bytes
 - ▶ MIPS, ARM: fijo

Definición de programa

- **Programa:** lista ordenada de instrucciones máquina que se ejecutan en secuencia (por defecto).



Fases de ejecución de una instrucción

▶ Lectura de la instrucción (ciclo de *fetch*)

- ▶ $MAR \leftarrow PC$
- ▶ Lectura
- ▶ $MBR \leftarrow \text{Memoria}$
- ▶ $PC \leftarrow PC + I$
- ▶ $RI \leftarrow MBR$

▶ Decodificación de la instrucción

▶ Ejecución de la instrucción

▶ Volver a *fetch*

PC

000100

RI

0010000000000101

MAR

MBR

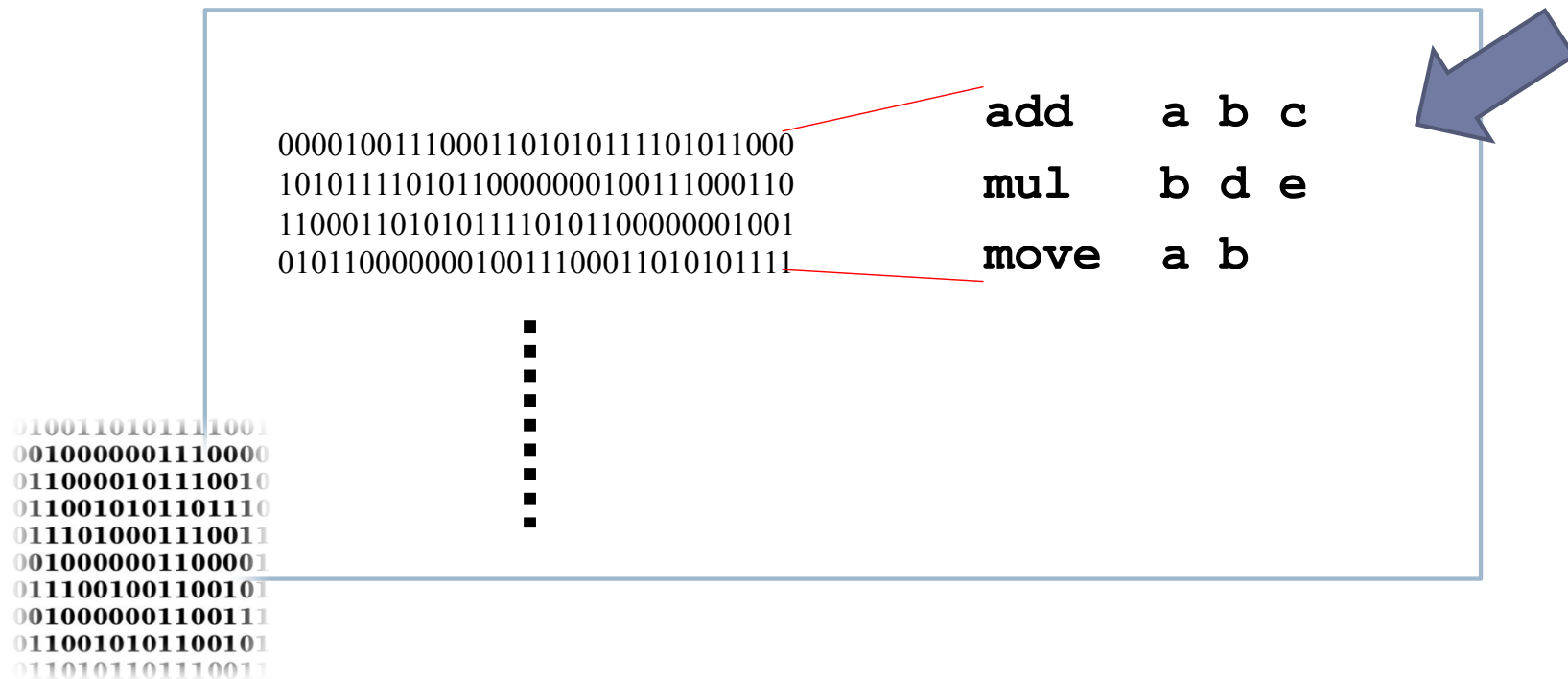
| | |
|--|--|
| | |
|--|--|

| Dirección | Contenido |
|-----------|------------------|
| 000100 | 0010000000000101 |
| | ⋮ |

Memoria

Definición de lenguaje ensamblador

- **Lenguaje ensamblador:** lenguaje legible por un programador que constituye **la representación más directa del código máquina específico de una arquitectura**



Definición de lenguaje ensamblador

- ▶ **Lenguaje ensamblador:** lenguaje legible por un programador que constituye **la representación más directa del código máquina específico de una arquitectura de computadoras.**
- ▶ Emplea códigos nemónicos para representar instrucciones
 - ▶ `add` – suma
 - ▶ `lw` – carga un dato de memoria
- ▶ Emplea nombres simbólicos para designar a datos y referencias
 - ▶ `$t0` – identificador de un registro
- ▶ Cada instrucción en ensamblador se corresponde con una instrucción máquina
 - ▶ `add $t1, $t2, $t3`

Diferentes niveles de lenguajes

Lenguaje de alto nivel
(ej: C, C++)

Compilador

Lenguaje ensamblador
(Ej: MIPS)

Ensamblador

Lenguaje Máquina
(MIPS)

```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

```
lw    $t0, 0($2)  
lw    $t1, 4($2)  
sw    $t1, 0($2)  
sw    $t0, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000  
1010 1111 0101 1000 0000 1001 1100 0110  
1100 0110 1010 1111 0101 1000 0000 1001  
0101 1000 0000 1001 1100 0110 1010 1111
```

Proceso de compilación

Lenguaje de alto nivel

```
#include <stdio.h>

#define PI 3.1416
#define RADIO 20

int main ( )
{
    int l;

    l=2*PI*RADIO;
    printf("long: %d\n",l) ;
    return (0);
}
```



Lenguaje ensamblador

```
.data
    PI: .word 3.14156
    RADIO: .word 20

.text
    li $a0 2
    la $t0 PI
    lw $t0 ($t0)
    la $t1 RADIO
    lw $t1 ($t1)
    mul $a0 $a0 $t0
    mul $a0 $a0 $t1

    li $v0 1
    syscall
```



Lenguaje binario

```
0100110101111001
0010000001110000
0110000101110010
0110010101101110
0111010001110011
0010000001100001
0111001001100101
0010000001100111
0110010101100101
0110101101110011
```

Compilación: ejemplo

- ▶ Edición de `hola.c`
 - ▶ `gedit hola.c`

```
int main ( )
{
    printf("Hola mundo...\n") ;
}
```

`hola.c`

```
#include <stdio.h>

#define PI 3.1416
#define RADIO 20

int main ( )
{
    int i;

    l=2*PI*RADIO;
    return (0);
}
```

- ▶ Generación del programa `hola`:
 - ▶ `gcc hola.c -o hola`

```
MZ
€  º  '  ¨  ,  Í  !  ,  L  Í  !  This program cannot be run in DOS
mode.
```

```
$      PE  L  ,  U  Ž  I  ¨  à
8
@  ^  .  text  º
@  @  .  bss  @  0
```

`hola`

```
P  ³  Ð
`  .  rdata
€  À  .  idata  ^
```

```
1100110101111001
0100000011100001
1100001011100011
1100101011011111
1110100011100011
0100000011000001
1110010011000101
0100000011000111
1100101011000101
1101011011100011
```

Compilación: ejemplo

► Desensamblar **hola**:

► **objdump -d hola**

```
hola.exe:      formato del fichero pei-i386

Desensamblado de la sección .text:

00401000 <_WinMainCRTStartup>:
  401000:      55                      push   %ebp
  ...
  40103f:      c9                      leave
  401040:      c3                      ret

00401050 <_main>:
  401050:      55                      push   %ebp
  401051:      89 e5                   mov     %esp,%ebp
  401053:      83 ec 08                sub     $0x8,%esp
  401056:      83 e4 f0                and     $0xffffffff0,%esp
  401059:      b8 00 00 00 00         mov     $0x0,%eax
  40105e:      83 c0 0f                add     $0xf,%eax
  401061:      83 c0 0f                add     $0xf,%eax
  401064:      c1 e8 04                shr     $0x4,%eax
  401067:      c1 e0 04                shl     $0x4,%eax
  40106a:      89 45 fc                mov     %eax,0xffffffffc(%ebp)
  40106d:      8b 45 fc                mov     0xffffffffc(%ebp),%eax
  401070:      e8 1b 00 00 00         call    401090 <__chkstk>
  401075:      e8 a6 00 00 00         call    401120 <_main>
  40107a:      c7 04 24 00 20 40 00   movl    $0x402000,(%esp)
  401081:      e8 aa 00 00 00         call    401130 <_printf>
  401086:      c9                      leave
  401087:      c3                      ret
  ...
```

```
.data
Pl: .word
3.14156
RADIO: .word
20

.text
li $a0 2
la $t0 Pl
lw $t0 ($t0)
lw $t1 RADIO
li $v0 1
syscall
```

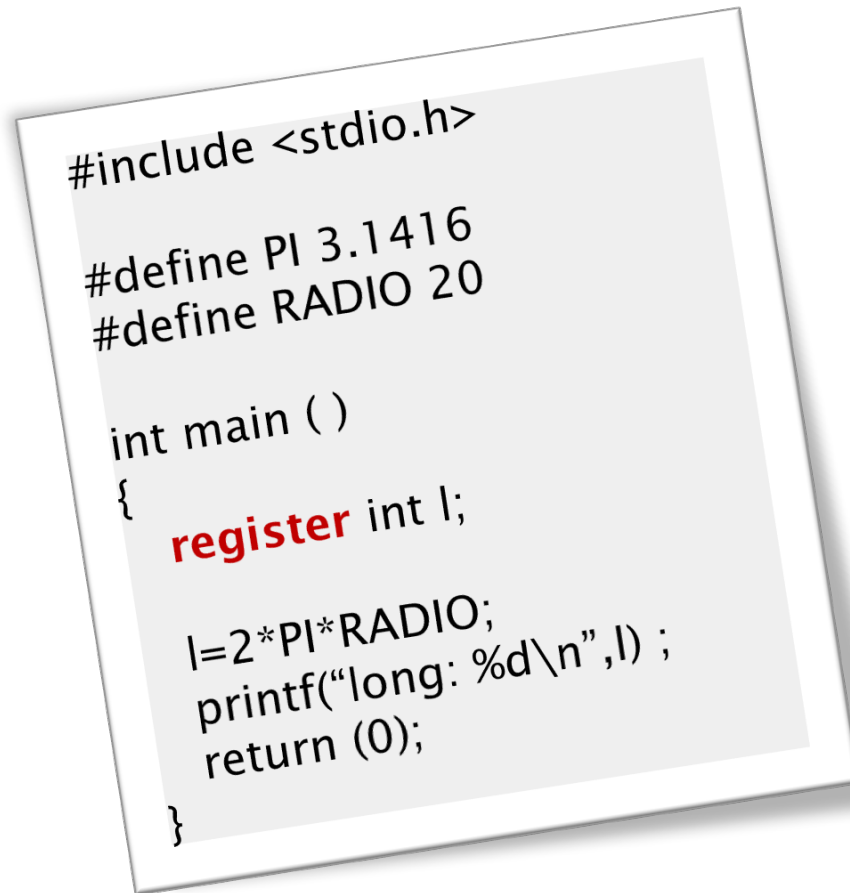
```
]10011010111100[
]01000000111000[
]11000010111001[
]11001010110111[
]11101000111001[
]01000000110000[
]11100100110010[
]01000000110011[
]11001010110010[
]11010110111001[
```



```
.data
Pl: .word 3.14156
RADIO: .word 20

.text
li $a0 2
la $t0 Pl
lw $t0 ($t0)
la $t1 RADIO
lw $t1 ($t1)
li $v0 1
syscall
```

Motivación para aprender ensamblador



```
#include <stdio.h>

#define PI 3.1416
#define RADIO 20

int main ( )
{
    register int l;

    l=2*PI*RADIO;
    printf("long: %d\n",l) ;
    return (0);
}
```

- ▶ Comprender qué ocurre cuando un computador ejecuta una sentencia de un lenguaje de alto nivel.
 - ▶ C, C++, Java, ...
- ▶ Poder determinar el impacto en tiempo de ejecución de una instrucción de alto nivel.
- ▶ Útil en dominios específicos:
 - ▶ Compiladores
 - ▶ Sistemas Operativos
 - ▶ Juegos
 - ▶ Sistemas empujados
 - ▶ Etc.

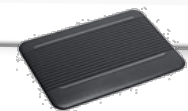
Objetivos

- ▶ Saber cómo se representan los elementos de un lenguaje de alto nivel en ensamblador:
 - ▶ Tipos de datos (int, char, ...)
 - ▶ Estructuras de control (if, while, ...)
- ▶ Poder escribir pequeños programas en ensamblador

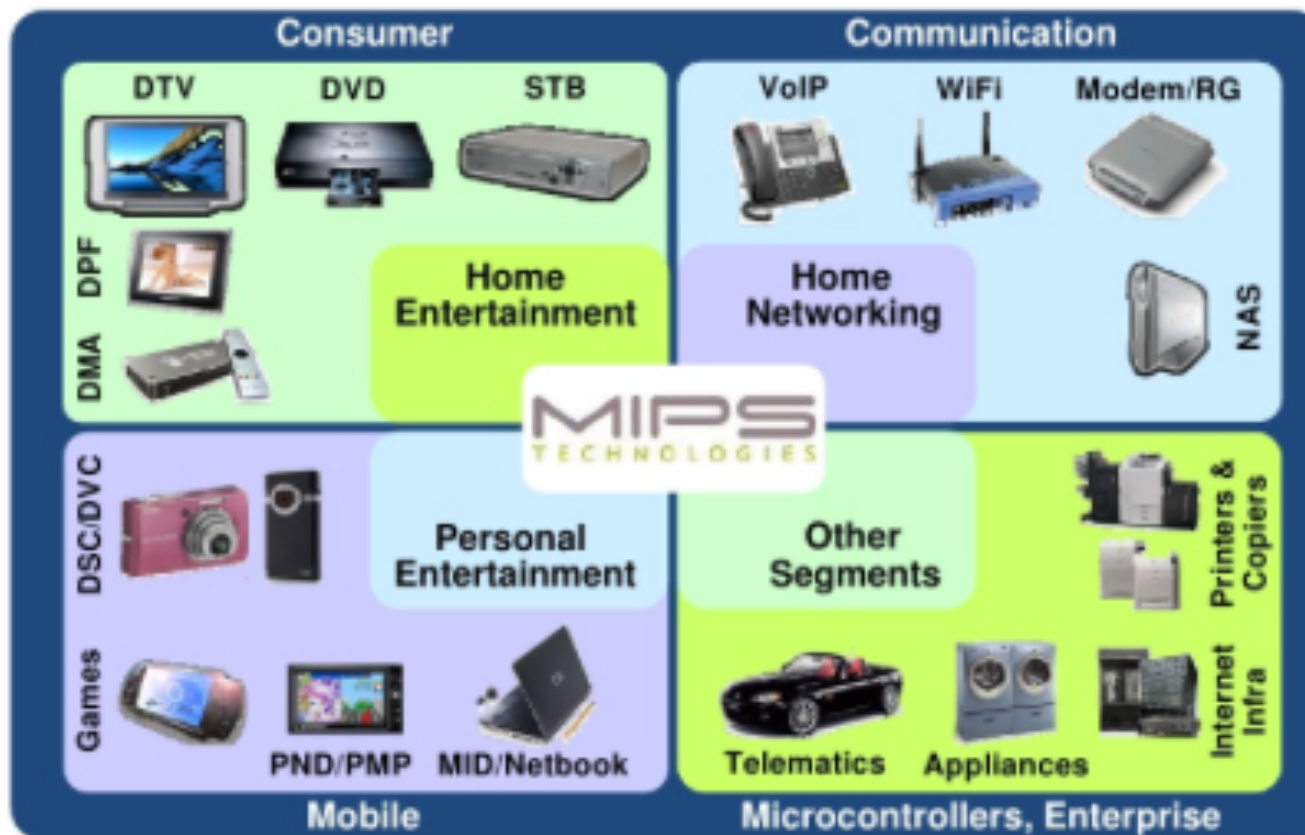
```
.data
PI: .word 3.14156
RADIO: .word 20

.text
li $a0 2
la $t0 PI
lw $t0 ($t0)
la $t1 RADIO
lw $t1 ($t1)
mul $a0 $a0 $t0
mul $a0 $a0 $t1

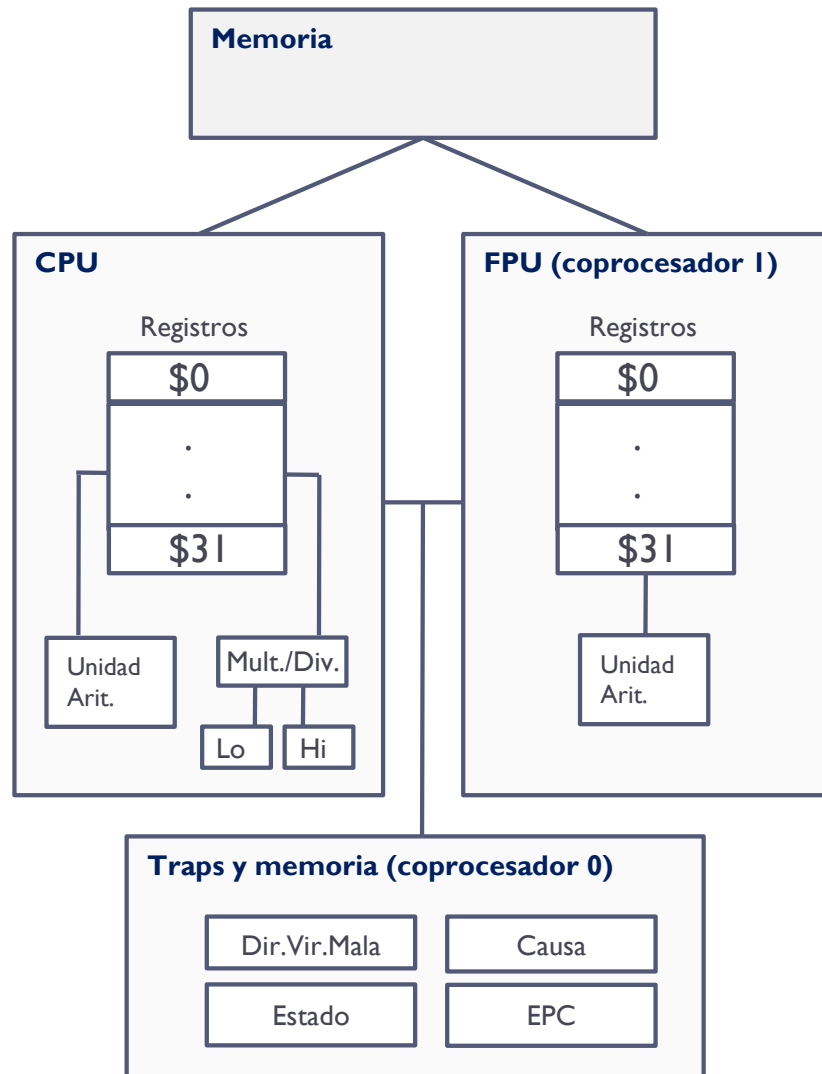
li $v0 1
syscall
```



Ejemplo de ensamblador: MIPS 32



Arquitectura del MIPS 32



- ▶ **MIPS 32**
 - ▶ Procesador de 32 bits
 - ▶ Tipo RISC
 - ▶ CPU + coprocesadores auxiliares
- ▶ **Coprocesador 0**
 - ▶ excepciones, interrupciones y sistema de memoria virtual
- ▶ **Coprocesador 1**
 - ▶ FPU (Unidad de Punto Flotante)

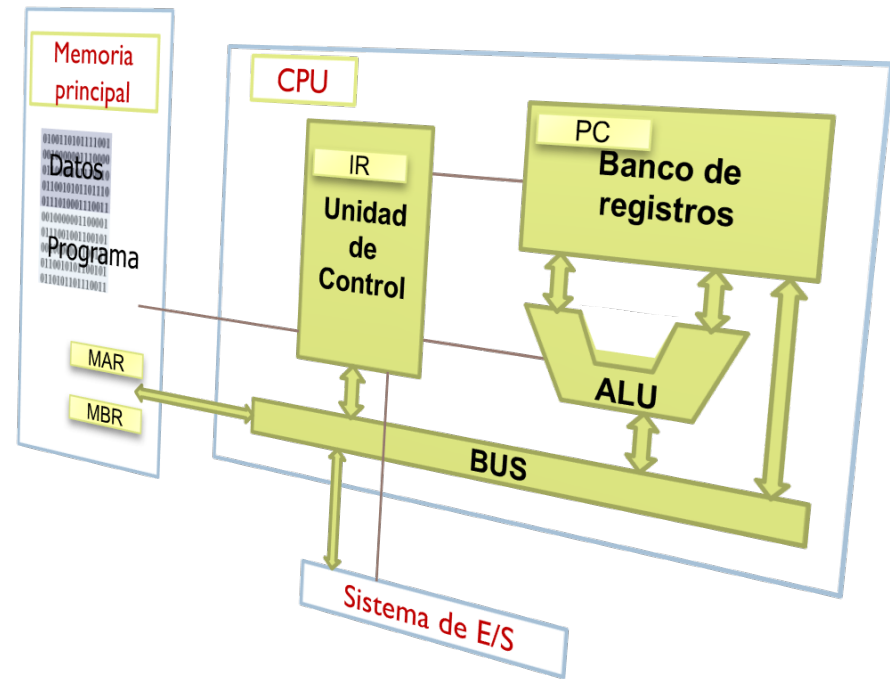
Banco de registros (enteros)

| Nombre registro | Número | Uso |
|-----------------|-------------|--|
| zero | 0 | Constante 0 |
| at | 1 | Reservado para el ensamblador |
| v0, v1 | 2, 3 | Resultado de una rutina (o expresión) |
| a0, ..., a3 | 4, ..., 7 | Argumento de entrada para rutinas |
| t0, ..., t7 | 8, ..., 15 | Temporal (<u>NO</u> se conserva entre llamadas) |
| s0, ..., s7 | 16, ..., 23 | Temporal (se conserva entre llamadas) |
| t8, t9 | 24, 25 | Temporal (<u>NO</u> se conserva entre llamadas) |
| k0, k1 | 26, 27 | Reservado para el sistema operativo |
| gp | 28 | Puntero al área global |
| sp | 29 | Puntero a pila |
| fp | 30 | Puntero a marco de pila |
| ra | 31 | Dirección de retorno (rutinas) |

- ▶ Hay 32 registros
 - ▶ 4 bytes de tamaño (una palabra)
 - ▶ Se nombran con un \$ al principio
- ▶ Convenio de uso
 - ▶ Reservados
 - ▶ Argumentos
 - ▶ Resultados
 - ▶ Temporales
 - ▶ Punteros

Tipo de instrucciones

- ▶ Transferencias de datos
- ▶ Aritméticas
- ▶ Lógicas
- ▶ De desplazamiento, rotación
- ▶ De comparación
- ▶ Control de flujo (bifurcaciones, llamadas a procedimientos)
- ▶ De conversión
- ▶ De Entrada/salida
- ▶ Llamadas al sistema

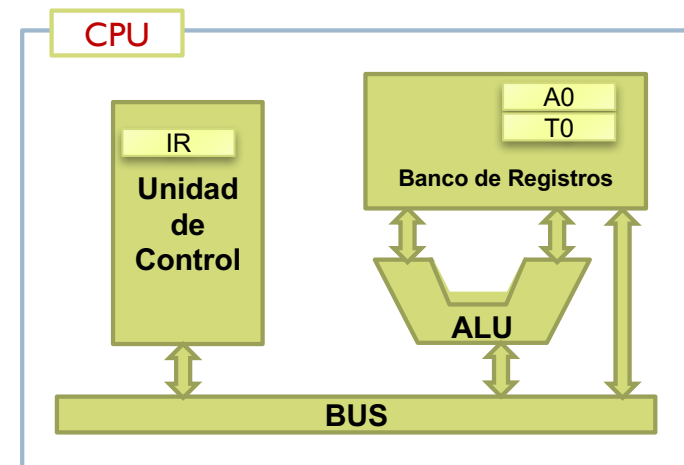


Transferencia de datos

- ▶ Copia **datos**:
 - ▶ entre **registros**
 - ▶ entre **registros y memoria**

- ▶ Ejemplos:

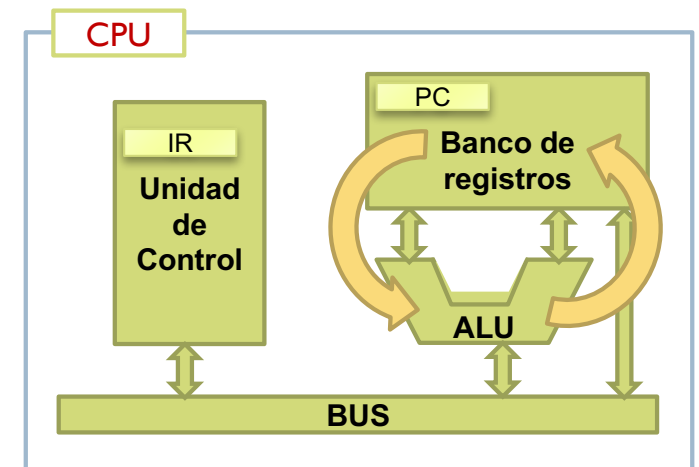
- ▶ Registro a registro
move \$a0 \$t0
- ▶ Carga inmediata
li \$t0 5



| | |
|-----------------------|-----------------------|
| move \$a0 \$t0 | # \$a0 ← \$t0 |
| li \$t0 5 | # \$t0 ← 000....00101 |

Aritméticas

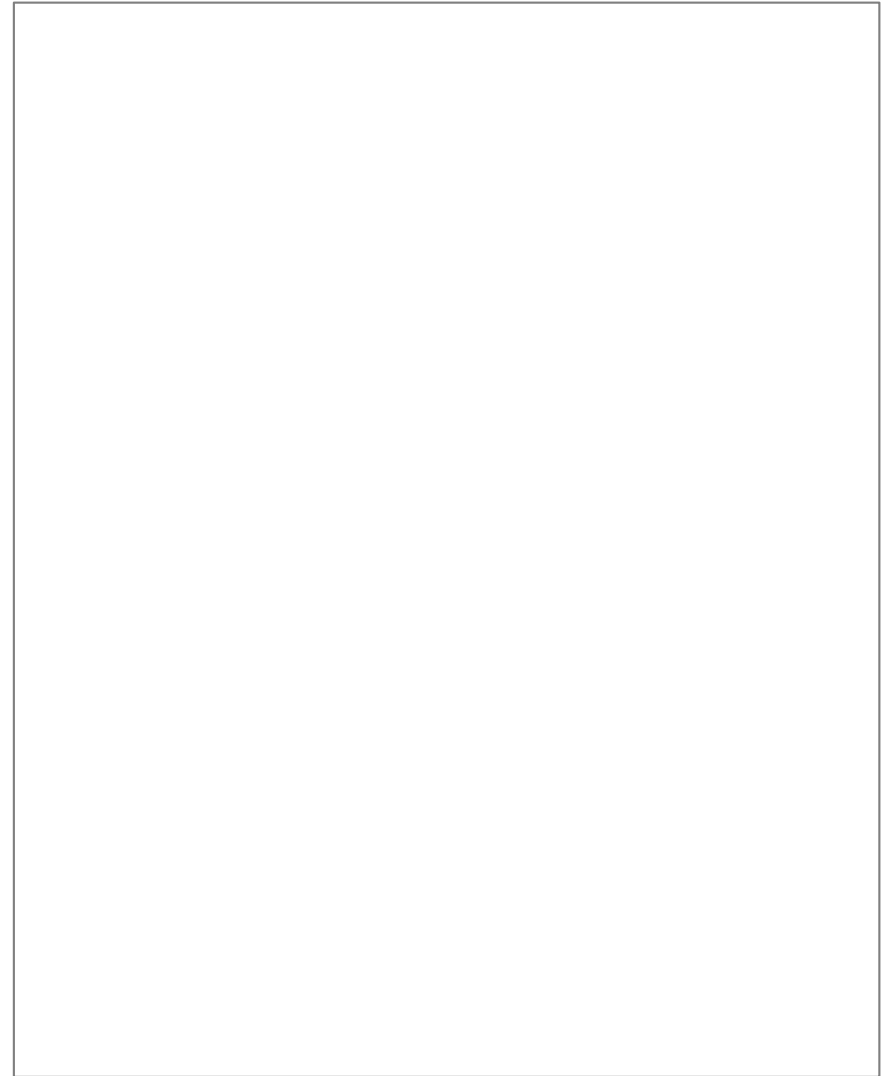
- ▶ Realiza operaciones **aritméticas de enteros** en la ALU o **aritméticas de coma flotante (FPU)**
- ▶ Ejemplos (ALU):
 - ▶ Sumar
`add $t0 $t1 $t2` $\$t0 \leftarrow \$t1 + \$t2$
`addi $t0 $t1 5` $\$t0 \leftarrow \$t1 + 5$
 - ▶ Restar
`sub $t0 $t1 $t2`
 - ▶ Multiplicar
`mul $t0 $t1 $t2`
 - ▶ División entera (5 / 2=2)
`div $t0 $t1 $t2`
 - ▶ Resto de la división (5 % 2=1)
`rem $t0 $t1 $t2` $\$t0 \leftarrow \$t1 \% \$t2$



Ejemplo

```
int a = 5;  
int b = 7;  
int c = 8;  
int i;
```

```
i = a * (b + c)
```



Ejemplo

```
int a = 5;  
int b = 7;  
int c = 8;  
int i;
```

```
i = a * (b + c)
```

```
li $t1 5  
li $t2 7  
li $t3 8
```

```
add $t4 $t2 $t3  
mul $t4 $t4 $t1
```

Ejercicio

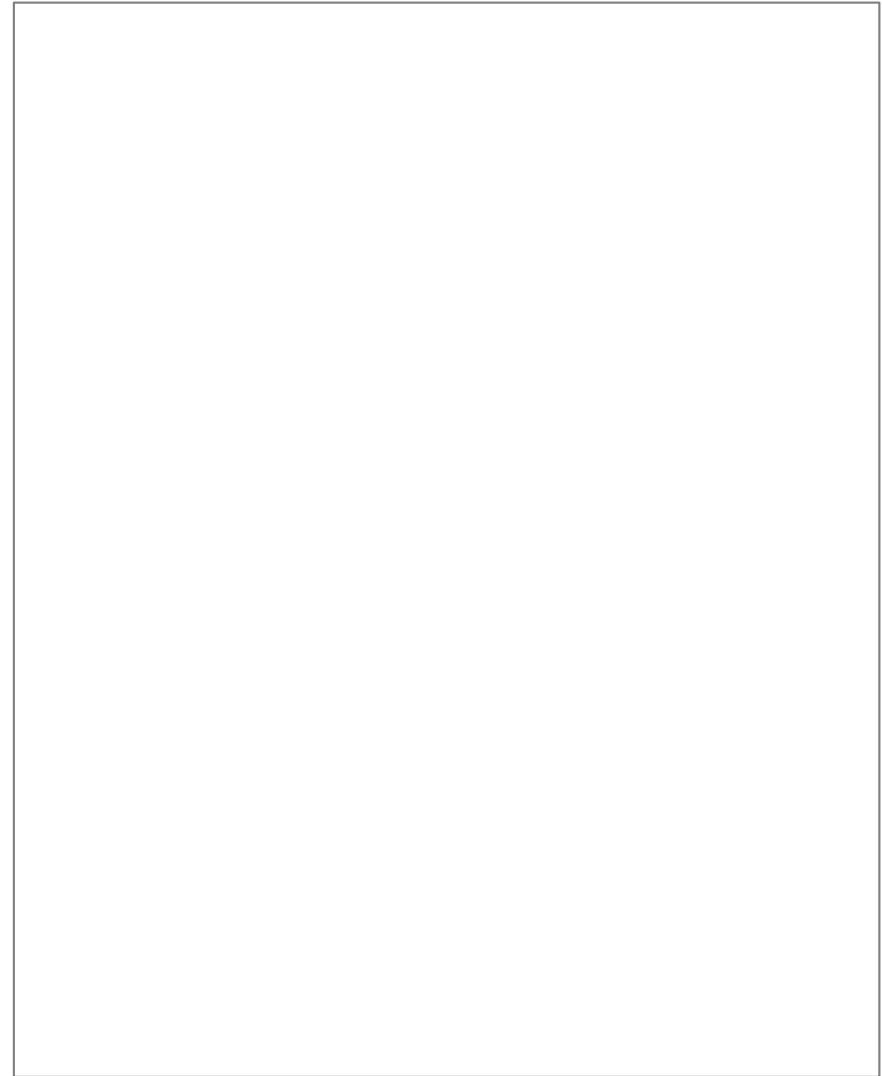
```
int a = 5;
```

```
int b = 7;
```

```
int c = 8;
```

```
int i;
```

```
i = -(a * (b - 10) + c)
```



Ejercicio (solución)

```
int a = 5;  
int b = 7;  
int c = 8;  
int i;
```

```
i = -(a * (b - 10) + c)
```

```
li $t1 5  
li $t2 7  
li $t3 8
```

```
li    $t0 10  
sub   $t4 $t2 $t0  
mul   $t4 $t4 $t1  
add   $t4 $t4 $t3  
li    $t0 -1  
mul   $t4 $t4 $t0
```


Tipos de operaciones aritméticas

- ▶ Aritmética en **binario puro** o en **complemento a dos**

- ▶ Ejemplos:

- ▶ Suma con signo (ca2)
`add $t0 $t1 $t2`
- ▶ Suma inmediata con signo
`addi $t0 $t1 -5`
- ▶ Suma sin signo (binario puro)
`addu $t0 $t1 $t2`
- ▶ Suma inmediata sin signo
`addiu $t0 $t1 2`

- ▶ No **overflow**:

```
li $t0 0x7FFFFFFF
li $t1 5
addu $t0 $t0 $t1
```

- ▶ Con **overflow**:

```
li $t0 0x7FFFFFFF
li $t1 1
add $t0 $t0 $t1
```

Ejercicio

```
li $t1 5
```

```
li $t2 7
```

```
li $t3 8
```

```
li $t0 10
```

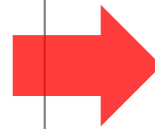
```
sub $t4 $t2 $t0
```

```
mul $t4 $t4 $t1
```

```
add $t4 $t4 $t3
```

```
li $t0 -1
```

```
mul $t4 $t4 $t0
```

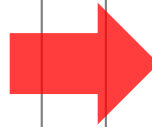


**¿Y usando las nuevas
instrucciones?**

Ejercicio (solución)

```
li $t1 5  
li $t2 7  
li $t3 8
```

```
li    $t0 10  
sub   $t4 $t2 $t0  
mul   $t4 $t4 $t1  
add   $t4 $t4 $t3  
li    $t0 -1  
mul   $t4 $t4 $t0
```



```
li $t1 5  
li $t2 7  
li $t3 8
```

```
addi   $t4 $t2 -10  
mul    $t4 $t4 $t1  
add    $t4 $t4 $t3  
mul    $t4 $t4 -1
```

Lógicas

- ▶ Operaciones booleanas

- ▶ Ejemplos:

- ▶ AND

`and $t0 $t1 $t2` ($\$t0 = \$t1 \ \& \ \$t2$)

| | |
|-----|------|
| | 1100 |
| AND | 1010 |
| | 1000 |

- ▶ OR

`or $t0 $t1 $t2` ($\$t0 = \$t1 \ | \ \$t2$)
`ori $0 $t1 80` ($\$t0 = \$t1 \ | \ 80$)

| | |
|----|------|
| | 1100 |
| OR | 1010 |
| | 1110 |

- ▶ NOT

`not $t0 $t1` ($\$t0 = ! \$t1$)

| | |
|-----|----|
| NOT | 10 |
| | 01 |

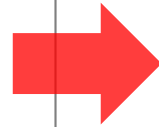
- ▶ XOR

`xor $t0 $t1 $t2` ($\$t0 = \$t1 \ ^ \ \$t2$)

| | |
|-----|------|
| | 1100 |
| XOR | 1010 |
| | 0110 |

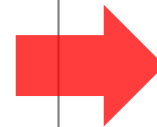
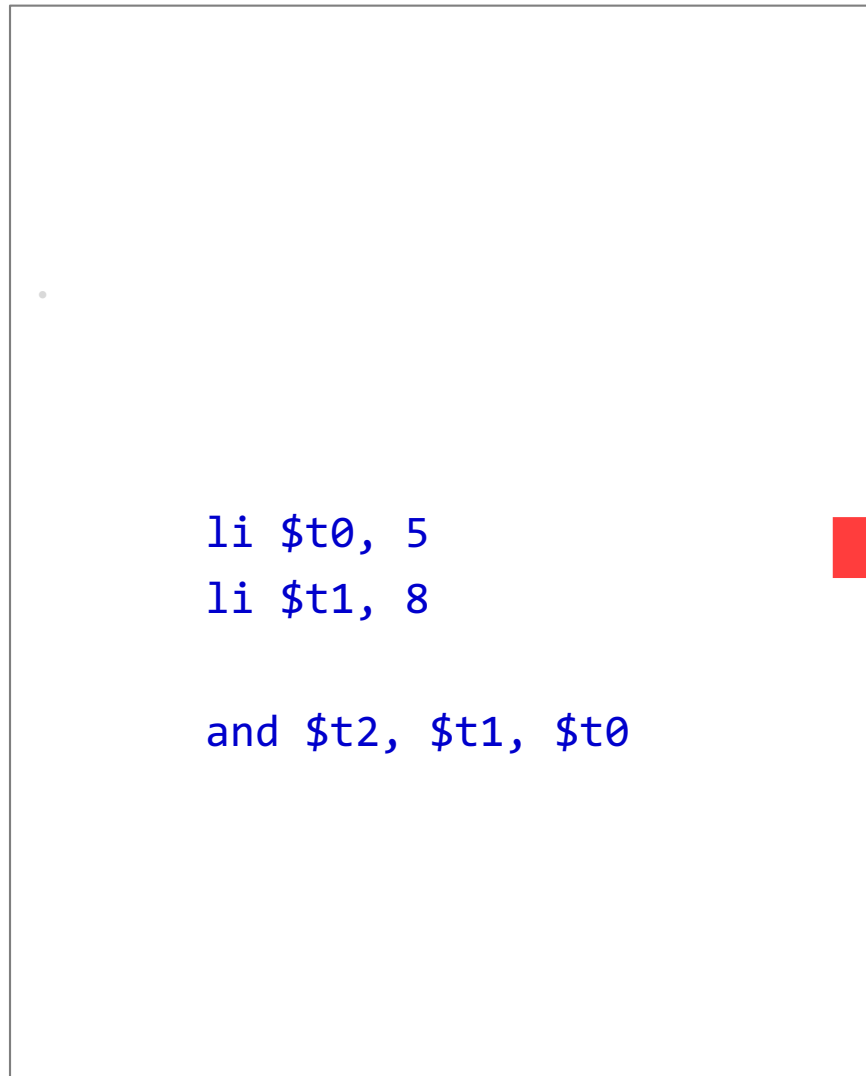
Ejercicio

```
li $t0, 5  
li $t1, 8  
  
and $t2, $t1, $t0
```



¿Cuál será el valor
almacenado en \$t2?

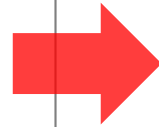
Ejercicio (solución)



```
00...0101    $t0
00...1000    $t1
and 00...0000 $t2
```

Ejercicio

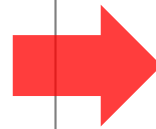
```
li $t0, 5  
li $t1, 0x007FFFFFFF  
  
and $t2, $t1, $t0
```



¿Qué permite hacer un and
con 0x007FFFFFFF?

Ejercicio (solución)

```
li $t0, 5  
li $t1, 0x007FFFFFFF  
  
and $t2, $t1, $t0
```



¿Qué permite hacer un and con 0x007FFFFFFF?

Obtener los 23 bits menos significativos

La constante usada para la selección de bits se denomina **máscara**.

Desplazamientos

- ▶ De movimiento de **bits**
- ▶ Ejemplos:
 - ▶ Desplazamiento **lógico** a la derecha
`srl $t0 $t0 4` (\$t0 = \$t0 >> 4 bits)
 - ▶ Desplazamiento **lógico** a la izquierda
`sll $t0 $t0 5` (\$t0 = \$t0 << 5 bits)
 - ▶ Desplazamiento **aritmético**
`sra $t0 $t0 2` (\$t0 = \$t0 >> 2 bits)



Ejercicio

```
li $t0, 5
```

```
li $t1, 6
```

```
sra $t0, $t1, 1
```

¿Cuál es el valor de \$t0?



Ejercicio (solución)

```
li $t0, 5
```

```
li $t1, 6
```

```
sra $t0, $t1, 1
```

¿Cuál es el valor de \$t0?

000 0110 \$t1

Se desplaza 1 bit a la derecha



000 0011 \$t0

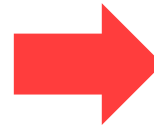
Ejercicio

```
li $t0, 5
```

```
li $t1, 6
```

```
sll $t0, $t1, 1
```

¿Cuál es el valor de \$t0?



Ejercicio (solución)

```
li $t0, 5
```

```
li $t1, 6
```

```
sll $t0, $t1, 1
```

¿Cuál es el valor de \$t0?

000 0110 \$t1

Se desplaza 1 bit a la izquierda



000 1100 \$t0

Rotaciones

- ▶ De movimiento de bits (2)

- ▶ Ejemplos:

- ▶ Rotación a la izquierda

`rol $t0 $t0 4` (\$t0 = \$t0 >> 4 bits)



- ▶ Rotación a la derecha

`ror $t0 $t0 5` (\$t0 = \$t0 << 5 bits)



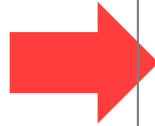
Ejercicio

Realice un programa que detecte el signo de un número almacenado \$t0 y deje en \$t1 un 1 si es negativo y un 0 si es positivo



Ejercicio (solución)

Realice un programa que detecte el signo de un número almacenado \$t0 y deje en \$t1 un 1 si es negativo y un 0 si es positivo



```
li    $t0 -3  
  
move  $t1 $t0  
rol   $t1 $t1 1  
and   $t1 $t1 0x0000 0001
```


Instrucciones de comparación

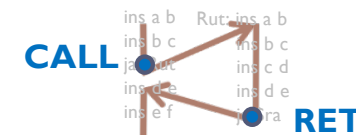
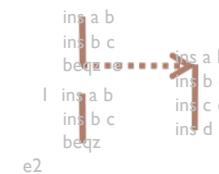
- ▶ `seq $t0, $t1, $t2` if ($t1 == t2$) $t0 = 1$; else $t0 = 0$
- ▶ `sneq $t0, $t1, $t2` if ($t1 != t2$) $t0 = 1$; else $t0 = 0$
- ▶ `sge $t0, $t1, $t2` if ($t1 \geq t2$) $t0 = 1$; else $t0 = 0$
- ▶ `sgt $t0, $t1, $t2` if ($t1 > t2$) $t0 = 1$; else $t0 = 0$
- ▶ `sle $t0, $t1, $t2` if ($t1 \leq t2$) $t0 = 1$; else $t0 = 0$
- ▶ `slt $t0, $t1, $t2` if ($t1 < t2$) $t0 = 1$; else $t0 = 0$

Instrucciones de comparación

- ▶ `seq $t0, $t1, $t2` Set if equal
- ▶ `sneq $t0, $t1, $t2` Set if no equal
- ▶ `sge $t0, $t1, $t2` Set if greater or equal
- ▶ `sgt $t0, $t1, $t2` Set if greater than
- ▶ `sle $t0, $t1, $t2` Set if less or equal
- ▶ `slt $t0, $t1, $t2` Set if less than

Control de Flujo

- ▶ Cambio de la secuencia de instrucciones a ejecutar (instrucciones de bifurcación)
- ▶ Distintos tipos:
 - ▶ Bifurcación o salto condicional:
 - ▶ Saltar a la posición etiqueta , si $\$t0 \leq \$t1$
 - ▶ Ej: `bne $t0 $t1 etiqueta`
 - ▶ Bifurcación o salto incondicional:
 - ▶ El salto se realiza siempre
 - ▶ Ej: `j etiqueta`
`b etiqueta`
 - ▶ Llamada a procedimiento:
 - ▶ Ej: `jal subrutina jr $ra`



Instrucciones de bifurcación condicional

| | | | | |
|--------|------|------|-------|---------------------------------|
| ▶ beq | \$t0 | \$t1 | etiq1 | # salta a etiq1 si \$t1 == \$t0 |
| ▶ bne | \$t0 | \$t1 | etiq1 | # salta a etiq1 si \$t1 != \$t0 |
| ▶ beqz | \$t1 | | etiq1 | # salta a etiq1 si \$t1 == 0 |
| ▶ bnez | \$t1 | | etiq1 | # salta a etiq1 si \$t1 != 0 |
| ▶ bgt | \$t0 | \$t1 | etiq1 | # salta a etiq1 si \$t0 > \$t1 |
| ▶ bge | \$t0 | \$t1 | etiq1 | # salta a etiq1 si \$t0 >= \$t1 |
| ▶ blt | \$t0 | \$t1 | etiq1 | # salta a etiq1 si \$t0 < \$t1 |
| ▶ ble | \$t0 | \$t1 | etiq1 | # salta a etiq1 si \$t0 <= \$t1 |

Ejercicio

Dada la siguiente expresión de un lenguaje de alto nivel

```
int a = 6;
```

```
int b = 7;
```

```
int c = 3;
```

```
int d;
```

```
d = (a+b) * (a+b);
```

Indique un fragmento de código en ensamblador del MIPS 32 que permita evaluar la expresión anterior. El resultado ha de almacenarse en el registro \$t5.

Estructuras de control

while

| | | | |
|------|------|----|------|
| beq | \$t1 | = | \$t0 |
| bnez | \$t1 | = | 0 |
| bne | \$t1 | != | \$t0 |
| bgt | \$t1 | > | \$t0 |
| bge | \$t1 | >= | \$t0 |
| blt | \$t1 | < | \$t0 |
| ble | \$t1 | <= | \$t0 |

```
int i;
```

```
main ()
```

```
{
```

```
    i=0;
```

```
    while (i < 10) {
```

```
        /* acción */
```

```
        i = i + 1 ;
```

```
    }
```

```
}
```

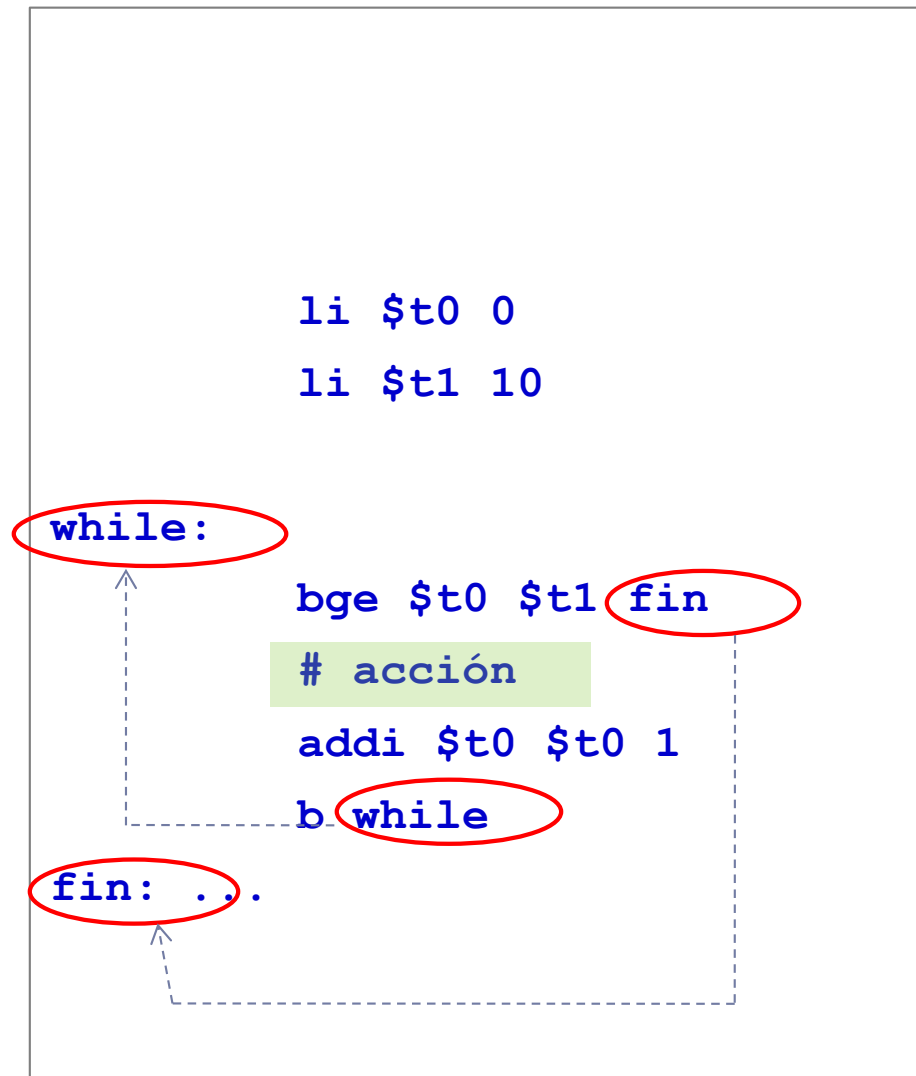
Estructuras de control while...

| | |
|------|--------------|
| beq | \$t1 = \$t0 |
| bnez | \$t1 = 0 |
| bne | \$t1 != \$t0 |
| bgt | \$t1 > \$t0 |
| bge | \$t1 >= \$t0 |
| blt | \$t1 < \$t0 |
| ble | \$t1 <= \$t0 |

```
int i;

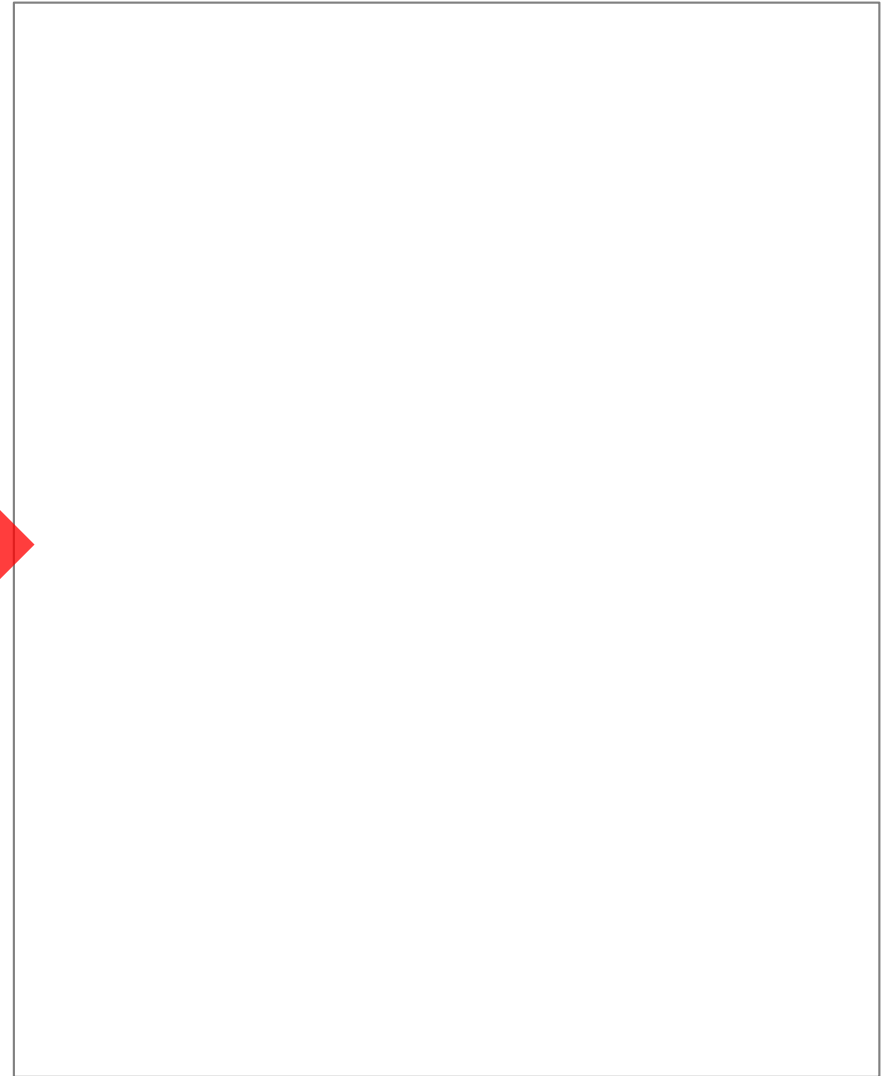
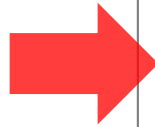
main ()
{
    i=0;
    while (i < 10) {

        /* acción */
        i = i + 1 ;
    }
}
```



Ejercicio

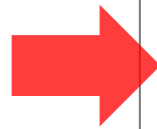
Realice un programa que calcule la suma de los diez primeros números y deje este valor en el registro \$v0



Ejercicio (solución)

Realice un programa que calcule la suma de los diez primeros números y deje este valor en el registro \$v0

$1 + 2 + 3 + \dots + 10$



```
li $v0 0
add $v0 $v0 1
add $v0 $v0 2
add $v0 $v0 3
add $v0 $v0 4
add $v0 $v0 5
add $v0 $v0 6
add $v0 $v0 7
add $v0 $v0 8
add $v0 $v0 9
```

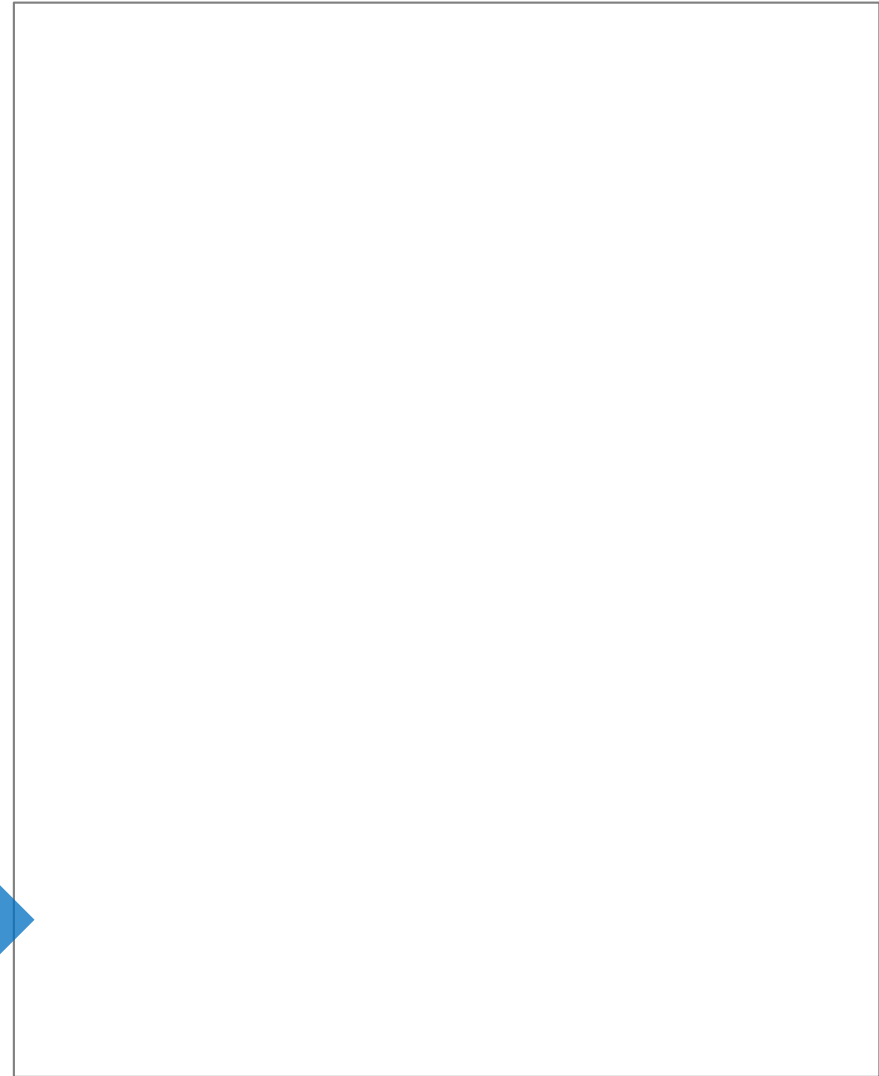
**NO vale
;-)**

Ejercicio (solución)

Realice un programa que calcule la suma de los diez primeros números y deje este valor en el registro \$v0



```
int i, s;  
  
s=0;  
i=0;  
while (i <= 10)  
{  
    s = s + i ;  
    i = i + 1 ;  
}
```

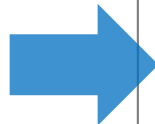


Ejercicio (solución)

Realice un programa que calcule la suma de los diez primeros números y deje este valor en el registro \$v0



```
int i, s;  
  
s=0;  
i=0;  
while (i <= 10)  
{  
    s = s + i ;  
    i = i + 1 ;  
}
```



```
li $t0 0  
li $v0 0  
li $t2 10  
  
while1:  
    bgt $t0 $t2 fin1  
    add $v0 $v0 $t0  
    add $t0 $t0 1  
    b while1  
  
fin1:
```

Ejercicio

- ▶ Calcular el número de 1's que hay en un registro (\$t0).
Resultado en \$t3

Ejercicio (solución)

- Calcular el número de 1's que hay en un registro (\$t0).
Resultado en \$t3

```
i = 0;
n = 45; #numero
s = 0;
while (i < 32)
{
    b = primer bit de n
    s = s + b;
    desplazar el contenido
    de n un bit a la
    derecha
    i = i + 1 ;
}
```

Ejercicio (solución)

- Calcular el número de 1's que hay en un registro (\$t0).
Resultado en \$t3

```
i = 0;
n = 45;  #numero
s = 0;
while (i < 32)
{
    b = primer bit de n
    s = s + b;
    desplazar el contenido
    de n un bit a la
    derecha
    i = i + 1 ;
}
```

```
i = 0;
n = 45;  #numero
s = 0;
while (i < 32)
{
    b = n & 1;
    s = s + b;
    n = n >> 1;
    i = i + 1 ;
}
```

Ejercicio (solución)

- ▶ Calcular el número de 1's que hay en un registro (\$t0).
Resultado en \$t3

```
i = 0;
n = 45;  #numero
s = 0;
while (i < 32)
{
    b = n & 1;
    s = s + b;
    n = n >> 1;
    i = i + 1 ;
}
```

```
li    $t0, 0    #i
li    $t1, 45   #n
li    $t2, 32
li    $t3, 0    #s
while: bge    $t0, $t2, fin
and    $t4, $t1, 1
add    $t3, $t3, $t4
srl    $t1, $t1, 1
addi   $t0, $t0, 1
b      while
fin:   ...
```

Ejemplo

Calcular el número de 1's que hay en un `int` en C/Java

```
int n = 45;
int b;
int i;
int s = 0;

for (i = 0; i < 32; i++) {
    b = n & 1;
    s = s + b;
    n = n >> 1;
}

printf("Hay %d\n", s);
```


Ejemplo

- Calcular el número de 1's que hay en un `int` en C/Java

Otra solución :

```
int count[256] = {0,1,1,2,1,2,2,3,1, . . .8};  
int i;  
int c = 0;  
  
for (i = 0; i <4; i++) {  
    c = count[n & 0xFF];  
    s = s + c;  
    n = n >> 8;  
}  
printf("Hay %d\n", c);
```

Ejercicio

- ▶ Obtener los 16 bits superiores de un registro (\$t0) y dejarlos en los 16 bits inferiores de otro (\$t1)

Ejercicio (solución)

- Obtener los 16 bits superiores de un registro (\$t0) y dejarlos en los 16 bits inferiores de otro (\$t1)

```
srl $t1, $t0, 16
```



Se desplaza a la derecha 16
Posiciones (de forma lógica)

Estructuras de control

if

| | |
|------|--------------|
| beq | \$t1 = \$t0 |
| bnez | \$t1 = 0 |
| bne | \$t1 != \$t0 |
| bgt | \$t1 > \$t0 |
| bge | \$t1 >= \$t0 |
| blt | \$t1 < \$t0 |
| ble | \$t1 <= \$t0 |

```
int a=1;
int b=2;

main ()
{
    if (a < b) {
        a = b;
    }
    ...
}
```

Estructuras de control if

| | |
|------|--------------|
| beq | \$t1 = \$t0 |
| bnez | \$t1 = 0 |
| bne | \$t1 != \$t0 |
| bgt | \$t1 > \$t0 |
| bge | \$t1 >= \$t0 |
| blt | \$t1 < \$t0 |
| ble | \$t1 <= \$t0 |

```
int a=1;
int b=2;
```

```
main ()
```

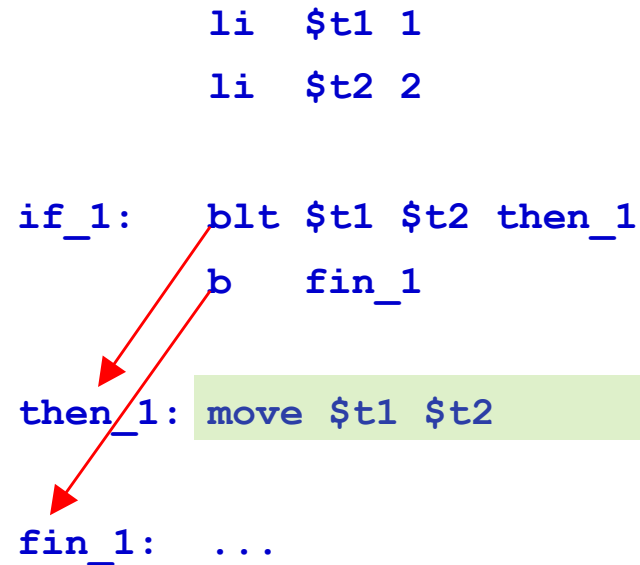
```
{
    if (a < b) {
        a = b;
    }
    ...
}
```

```
li $t1 1
li $t2 2

if_1: blt $t1 $t2 then_1
      b   fin_1

then_1: move $t1 $t2

fin_1: ...
```



The diagram illustrates the control flow of the assembly code. Red arrows show the execution path: from the 'if_1' label to the 'blt' instruction, then to the 'then_1' label, and finally to the 'fin_1' label. The 'b' instruction is shown without a target, indicating it falls through to the next instruction.

Estructuras de control

if

| | |
|------|--------------|
| beq | \$t1 = \$t0 |
| bnez | \$t1 = 0 |
| bne | \$t1 != \$t0 |
| bgt | \$t1 > \$t0 |
| bge | \$t1 >= \$t0 |
| blt | \$t1 < \$t0 |
| ble | \$t1 <= \$t0 |

```
int a=1;
int b=2;
```

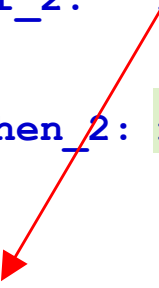
```
main ()
{
    if (a < b) {
        a = b;
    }
    ...
}
```

```
li $t1 1
li $t2 2

if_2: bge $t1 $t2 fin_2

then_2: move $t1 $t2

fin_2: ...
```



Estructuras de control

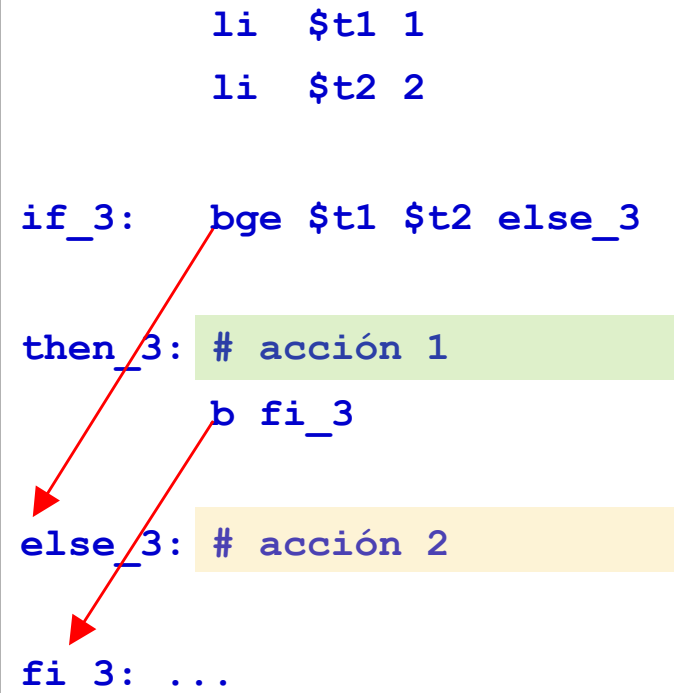
if-else

| | |
|------|--------------|
| beq | \$t1 = \$t0 |
| bnez | \$t1 = 0 |
| bne | \$t1 != \$t0 |
| bgt | \$t1 > \$t0 |
| bge | \$t1 >= \$t0 |
| blt | \$t1 < \$t0 |
| ble | \$t1 <= \$t0 |

```
int a=1;  
int b=2;
```

```
main ()  
{  
    if (a < b){  
        // acción 1  
    } else {  
        // acción 2  
    }  
}
```

```
        li $t1 1  
        li $t2 2  
  
if_3:    bge $t1 $t2 else_3  
  
then_3:  # acción 1  
        b fi_3  
  
else_3:  # acción 2  
  
fi_3:    ...
```



Ejercicio

```
int b1 = 4;
```

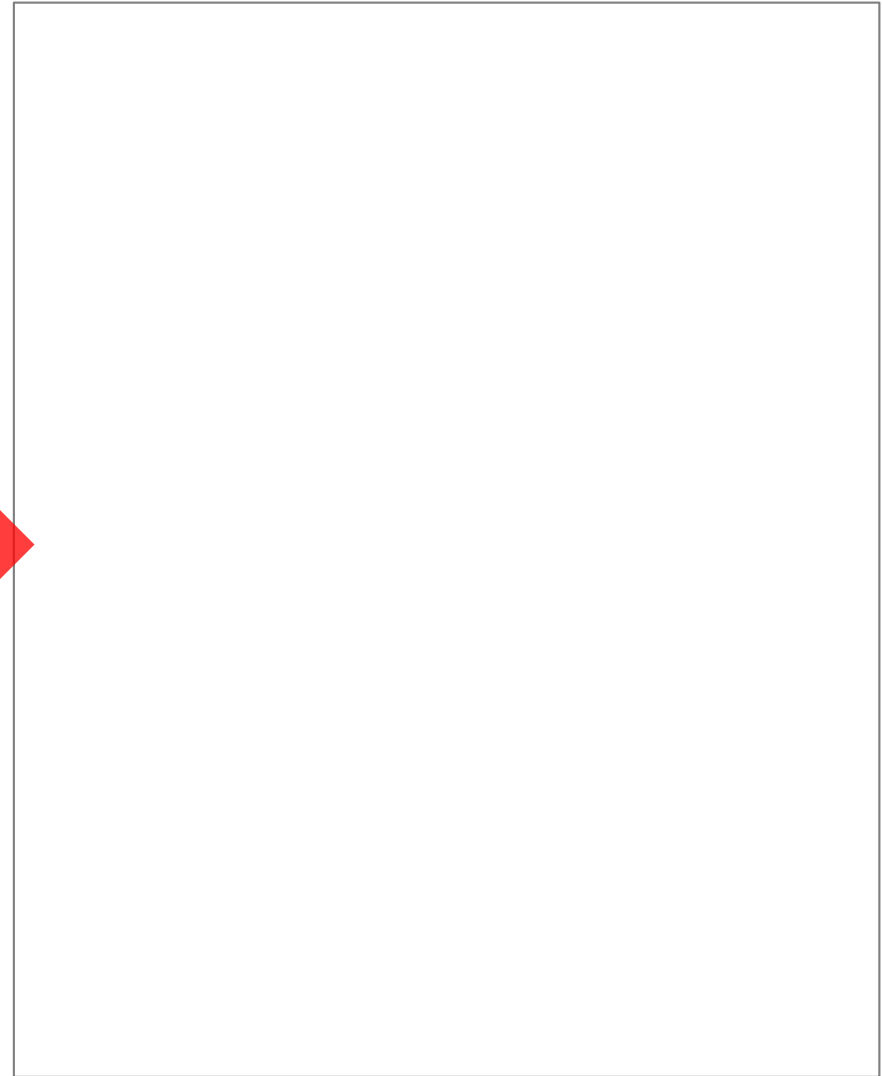
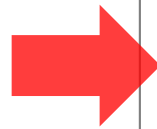
```
int b2 = 2;
```

```
if (b2 == 8) {
```

```
    b1 = 1;
```

```
}
```

```
...
```



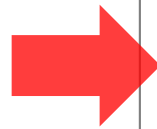
Ejercicio (solución)

```
int b1 = 4;
```

```
int b2 = 2;
```

```
if (b2 == 8) {  
    b1 = 1;  
}
```

```
...
```



```
li    $t0 4  
li    $t1 2  
li    $t2 8  
  
bneq  $t0 $t2  fin1  
li    $t1 1  
fin1:  ...
```

Ejercicio

- Determinar si el contenido de un registro (\$t2) es par.
Si es par se almacena en \$t1 un 1, sino se almacena un 0

Ejercicio (solución)

- Determinar si el contenido de un registro (\$t2) es par.

Si es par se almacena en \$t1 un 1, sino se almacena un 0

```
li    $t2, 9
li    $t1, 2
rem   $t1, $t2, $t1    # se obtiene el resto
bne   $t1, $0, else    # cond.
then: li    $t1, 1
      b     fin         # incond.
else: li    $t1, 0
fin: ...
```

Ejercicio (otra solución)

- Determinar si el contenido de un registro (\$t2) es par.

Si es par se almacena en \$t1 un 1, sino se almacena un 0

```
li    $t2, 9
      li    $t1, 2
      rem   $t1, $t2, $t1    # se obtiene el resto
      li    $t1, 0           # suponer impar
      bne   $t1, $0, fin     # si suposición ok, fin
      li    $t1, 1
fin:  ...
```

Ejercicio

- Determinar si el contenido de un registro (\$t2) es par. Si es par se almacena en \$t1 un 1, sino se almacena un 0. En este caso consultando el último bit

Ejercicio (solución)

- Determinar si el contenido de un registro (\$t2) es par. Si es par se almacena en \$t1 un 1, sino se almacena un 0. En este caso consultando el último bit

```
        li    $t2, 9
        li    $t1, 1
        and   $t1, $t2, $t1    # se obtiene el último bit
        beq   $t1, $0 then     # cond.
else:    li    $t1, 0
        b     fin              # incond.
then:    li    $t1, 1
fin:     ...
```

Ejercicio

- ▶ Calcular a^n
 - ▶ a en \$t0
 - ▶ n en \$t1
 - ▶ El resultado en \$t2

```
a=8
n=4;
i=0;
p = 1;
while (i < n)
{
    p = p * a
    i = i + 1 ;
}
}
```

Ejercicio (solución)

- ▶ Calcular a^n
 - ▶ a en \$t0
 - ▶ n en \$t1
 - ▶ El resultado en \$t2

```
a=8
n=4;
i=0;
p = 1;
while (i < n)
{
    p = p * a
    i = i + 1 ;
}
```

```
li    $t0, 8
li    $t1, 4
li    $t2, 1
li    $t4, 0

while: bge    $t4, $t1, fin
mul    $t2, $t2, $t0
addi   $t4, $t4, 1
b      while
fin:   move   $t2, $t4
```


Fallos típicos

1) Programa mal planteado

- ▶ No hace lo que se pide
- ▶ Hace incorrectamente lo que se pide

2) Programar directamente en ensamblador

- ▶ No codificar en pseudo-código el algoritmo a implementar

3) Escribir código ilegible

- ▶ No tabular el código
- ▶ No comentar el código ensamblador o
no hacer referencia al algoritmo planteado inicialmente