

Grupo ARCOS

**uc3m** | Universidad **Carlos III** de Madrid

## Tema 3: Fundamentos de la programación en ensamblador (I) **Estructura de Computadores**

Grado en Ingeniería Informática  
Grado en Matemática aplicada y Computación  
Doble Grado en Ingeniería Informática y Administración de Empresas

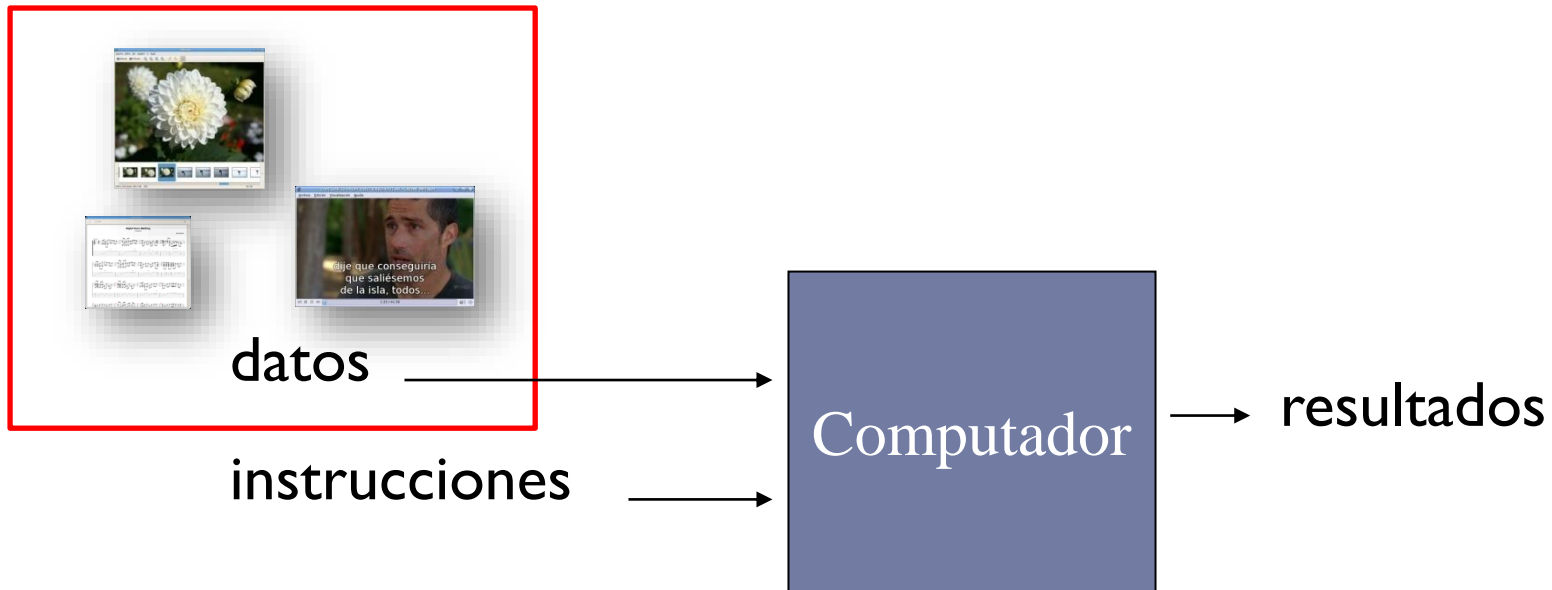


# Contenidos

1. Fundamentos básicos de la programación en ensamblador
  1. Motivación y objetivos
  2. Introducción a RISC-V32
2. Ensamblador del RISC-V 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

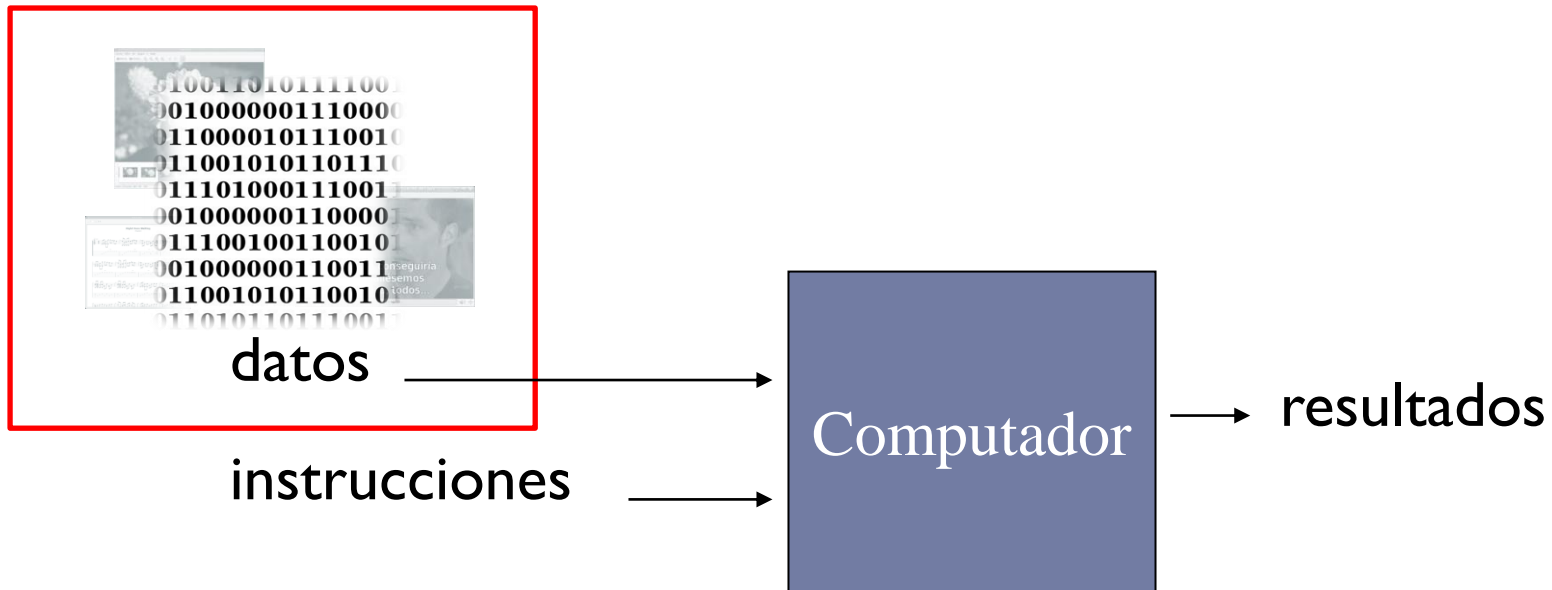
# Tipos de información: instrucciones y datos

## ► Representación de **datos** ...



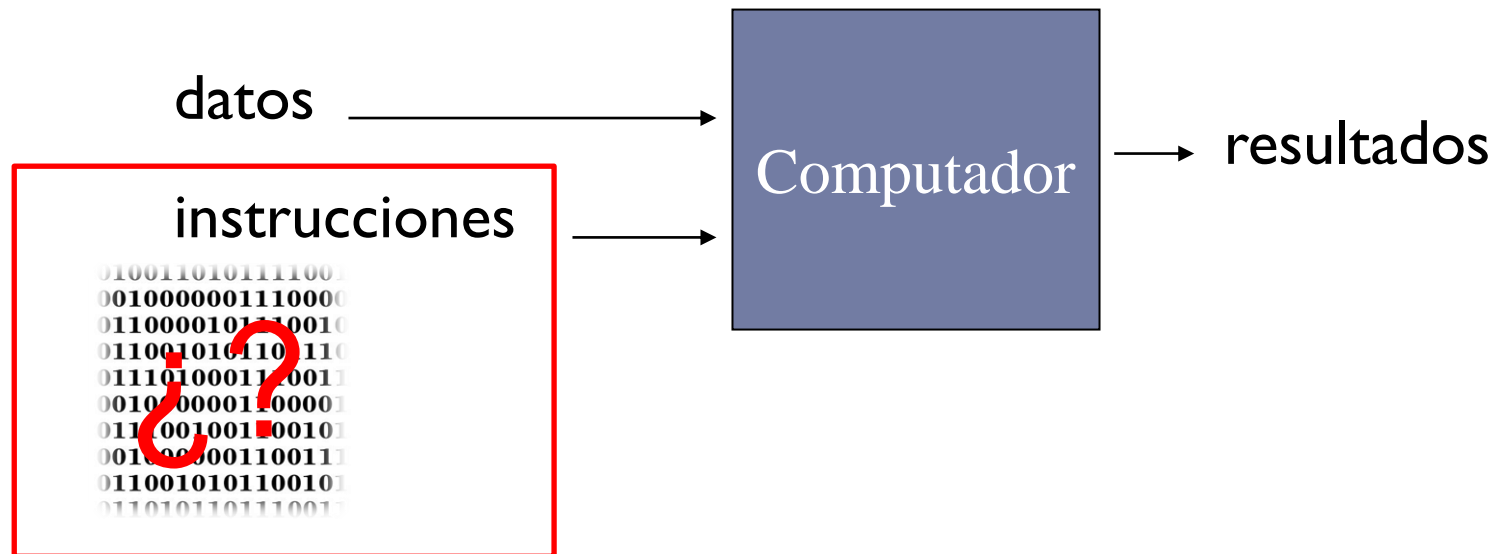
# Tipos de información: instrucciones y datos

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



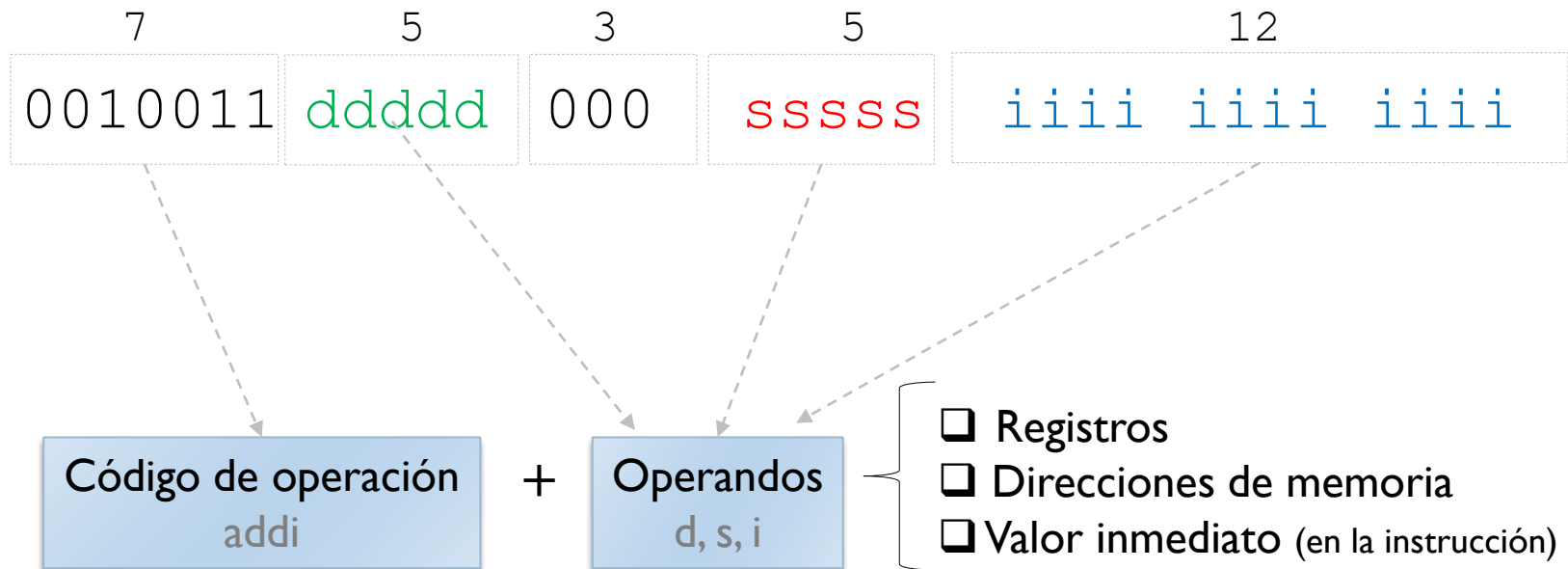
# Tipos de información: instrucciones y datos

- ▶ ¿Qué sucede con las instrucciones?
  - ▶ Instrucción máquina, propiedades y formato



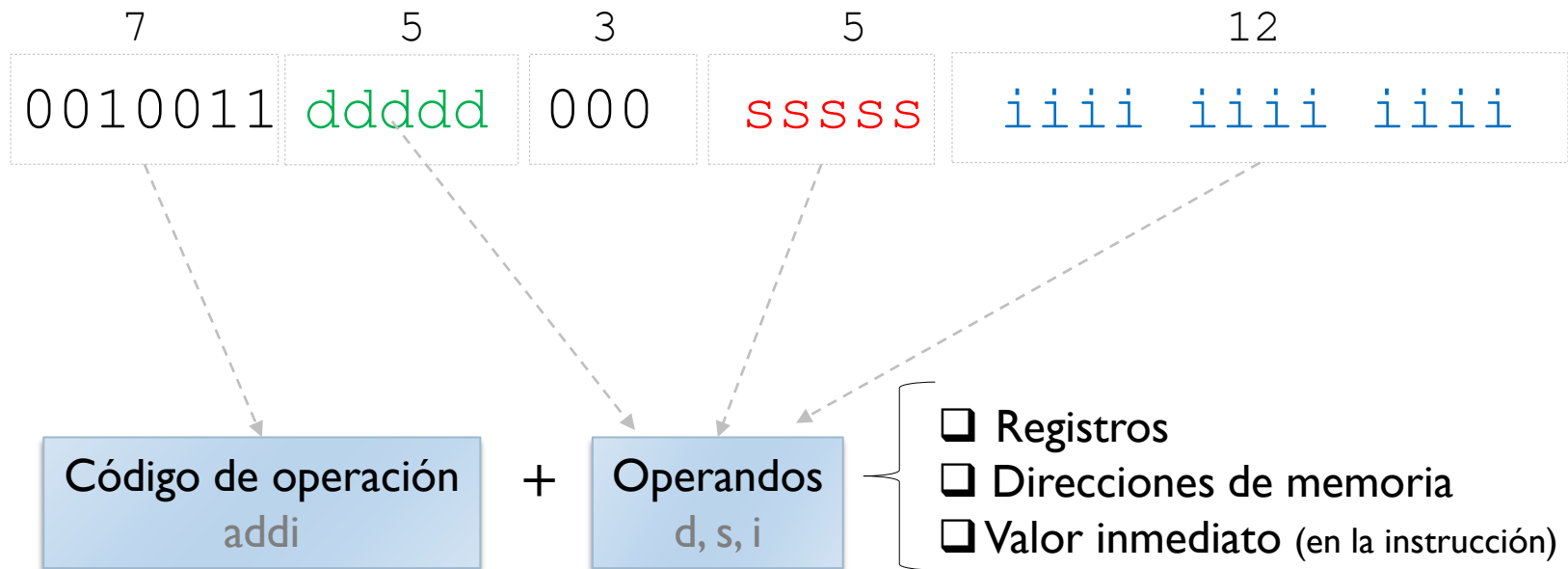
# Instrucción máquina: definición

- ▶ **Instrucción máquina:** operación elemental que puede ejecutar directamente el procesador
- ▶ Ejemplo: instrucción de suma inmediata (addi) en 32 bits
  - ▶  $(d) = \text{registro } (s) + \text{valor inmediato } (i)$



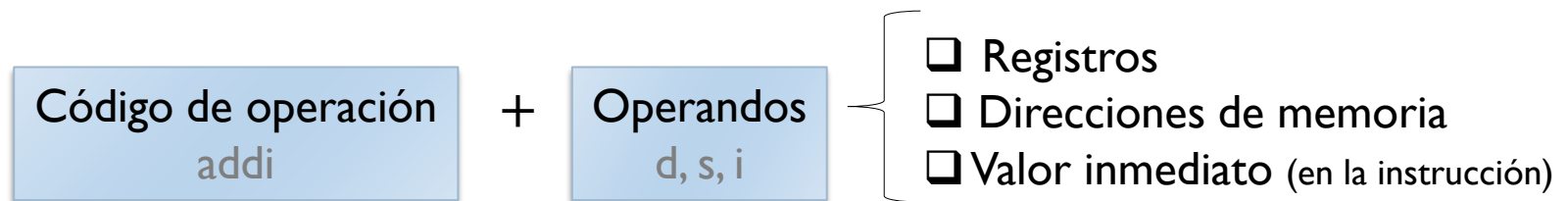
# Instrucciones máquina: propiedades

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



# Instrucción máquina: información incluida

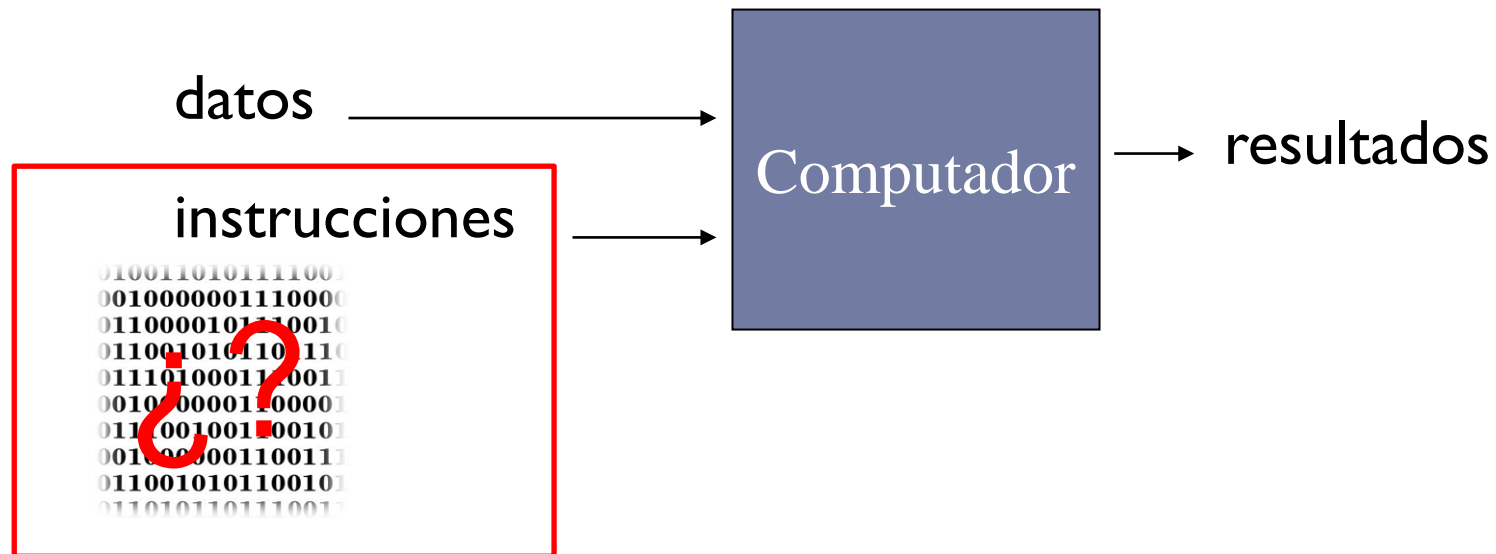
- ▶ La **operación a realizar**.
- ▶ Dónde se encuentran los **operandos**:
  - ▶ En registros
  - ▶ En memoria
  - ▶ En la propia instrucción (inmediato)
- ▶ Dónde dejar los resultados (como operando)
- ▶ Una referencia a la siguiente instrucción a ejecutar
  - ▶ De forma implícita, la siguiente instrucción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





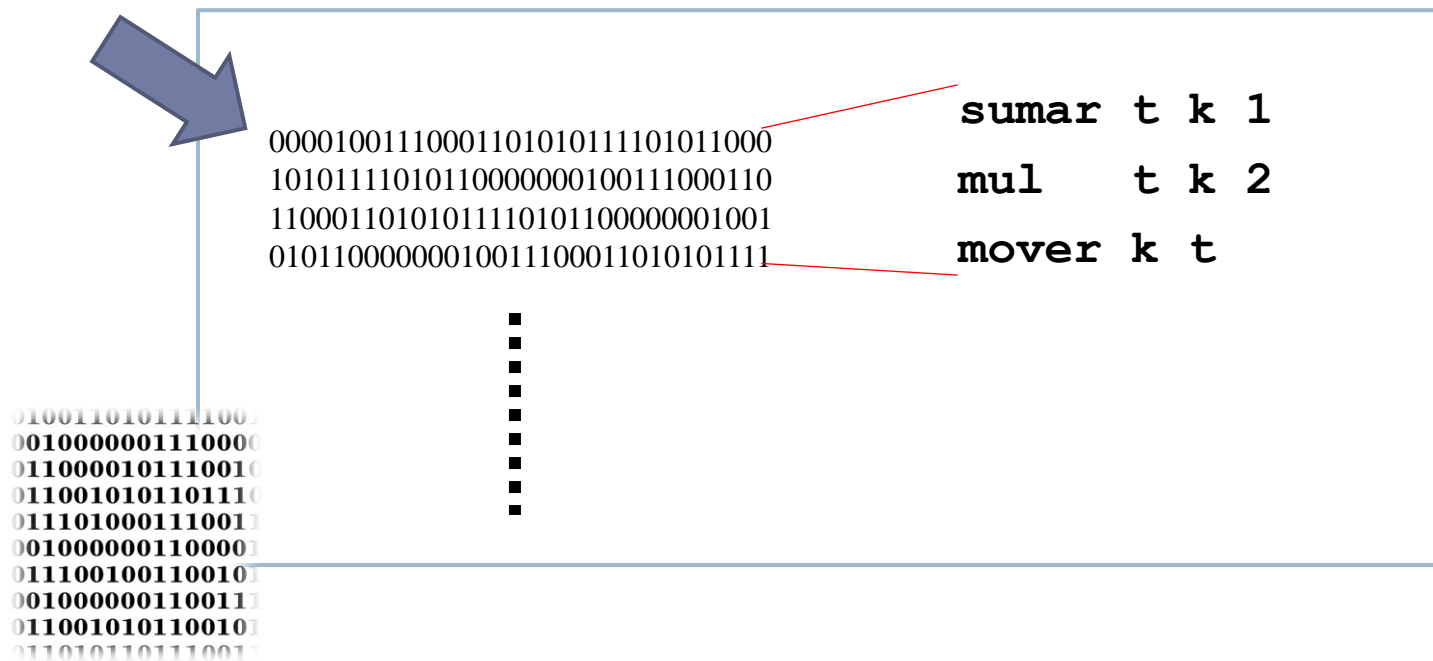
# Tipos de información: instrucciones y datos

- ▶ ¿Qué sucede con las instrucciones?
  - ▶ Programa, lenguaje ensamblador, ISA



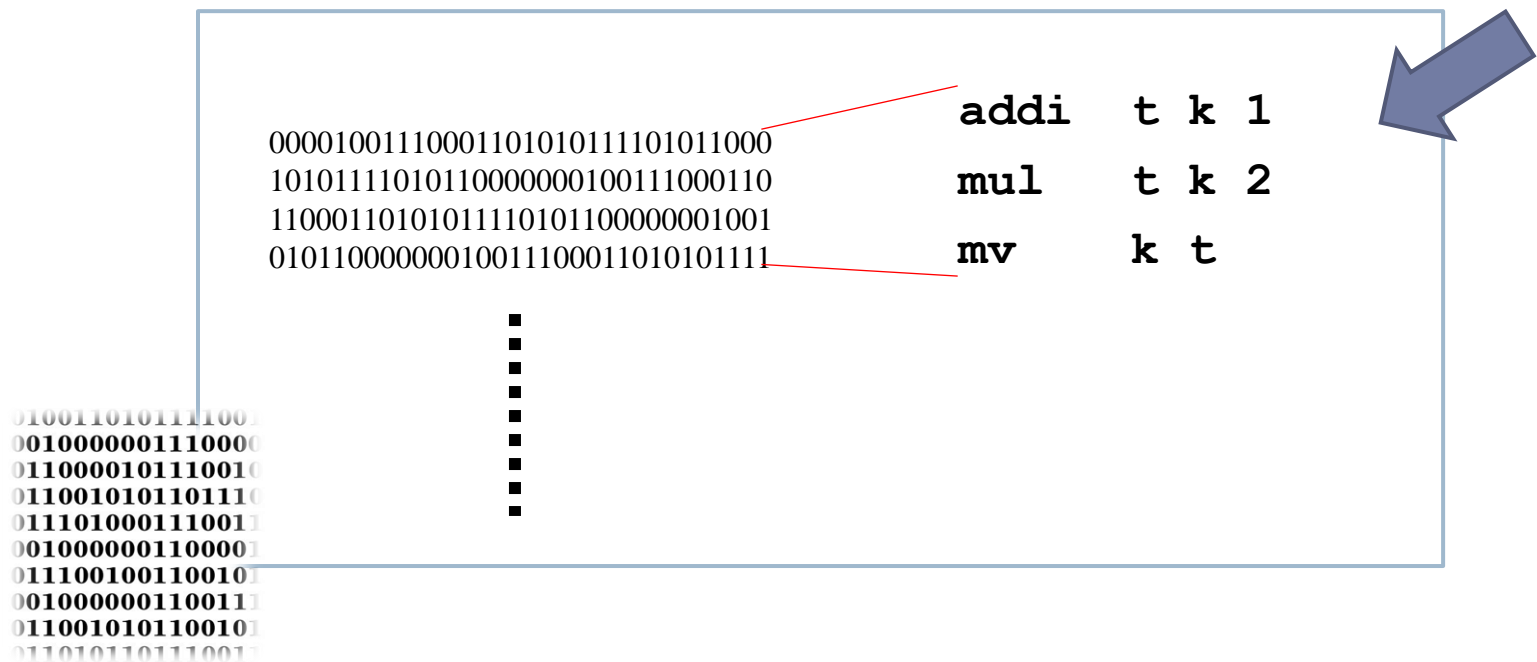
# Definición de programa

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



# 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: RISC-V)

*Ensamblador*

**Lenguaje Máquina**  
(RISC-V)

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

```
lw    t0, 0(x2)  
lw    t1, 4(x2)  
sw    t1, 0(x2)  
sw    t0, 4(x2)
```

```
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
```

# Juego de instrucciones

- ▶ *Instruction Set Architecture* (ISA)
  - ▶ Conjunto de instrucciones de un procesador
  - ▶ Frontera entre el hardware y el software
- ▶ Ejemplos:
  - ▶ 80x86
  - ▶ ARM
  - ▶ MIPS
  - ▶ RISC-V
  - ▶ PowerPC
  - ▶ Etc.

# Características de un juego de instrucciones (1/2)

- ▶ **Formato y codificación del juego de instrucciones**
  - ▶ Instrucciones de longitud fija o variable
    - ▶ 80x86: variable de 1 a 18 bytes
    - ▶ RISC-V, ARM: fijo
- ▶ **Operandos:**
  - ▶ Registros, memoria, la propia instrucción
- ▶ **Tipo y tamaño de los operandos**
  - ▶ bytes: 8 bits
  - ▶ enteros: 16, 32, 64 bits
  - ▶ números en coma flotante: simple precisión, doble,...
- ▶ **Modos de direccionamiento**
  - ▶ Especifican el lugar y la forma de acceder a los operandos (registro, memoria o la propia instrucción)

# Características de un juego de instrucciones (2/2)

- ▶ **Operaciones:**
  - ▶ Aritméticas, lógicas, de transferencia, control, ...
- ▶ **Instrucciones de control de flujo**
  - ▶ Saltos incondicionales
  - ▶ Saltos condicionales
  - ▶ Llamadas a procedimientos
- ▶ **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



# 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: id. de registros, direcciones de memoria o números
  - ▶ **Elementos de almacenamiento**
    - ▶ Registros
    - ▶ Memoria
    - ▶ Registros de los controladores de E/S
  - ▶ **Modos de ejecución**

# ¿Por qué aprender ensamblador?

Lenguaje de alto nivel

Lenguaje ensamblador

Lenguaje binario

```
#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);
```

```
}
```



```
.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 a7 1
```

```
ecall
```



```
0100110101111001
```

```
0010000001110000
```

```
0110000101110010
```

```
0110010101101110
```

```
0111010001110011
```

```
0010000001100001
```

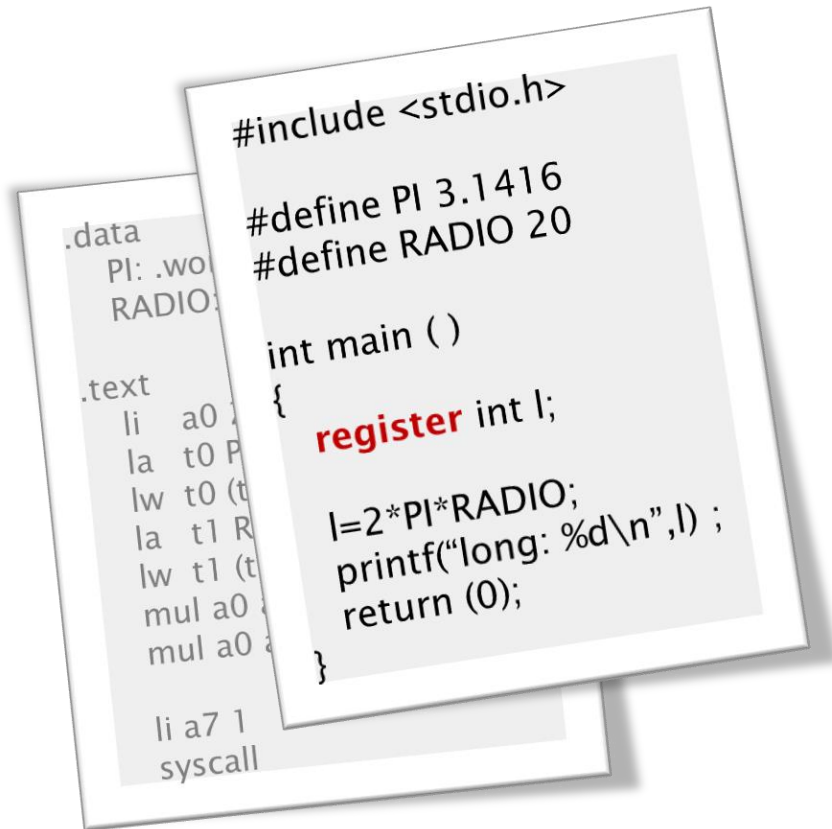
```
0111001001100101
```

```
0010000001100111
```

```
0110010101100101
```

```
0110101101110011
```

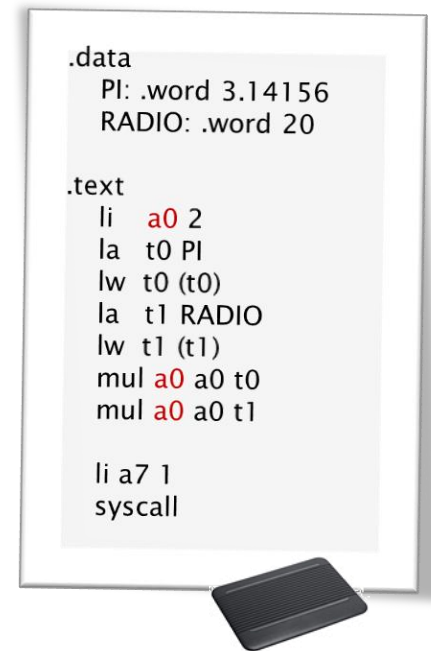
# Motivación para aprender ensamblador



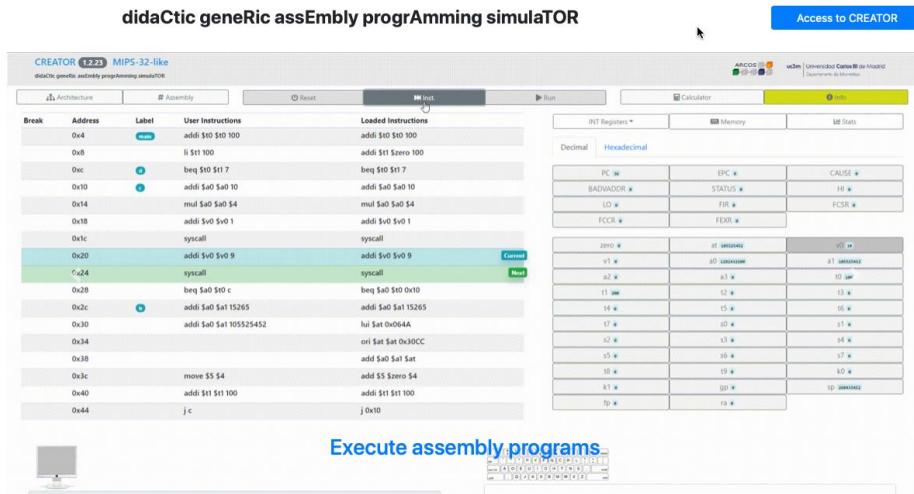
- ▶ Comprender qué ocurre cuando un computador ejecuta una sentencia de un lenguaje de alto nivel.
  - ▶ C, C++, Java, Python,...
- ▶ Poder determinar el impacto en tiempo de ejecución de una instrucción de alto nivel.
- ▶ Útil en dominios específicos:
  - ▶ Compiladores
  - ▶ Sistemas Operativos
  - ▶ 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 y entender pequeños programas en ensamblador



# Motivación para usar CREATOR

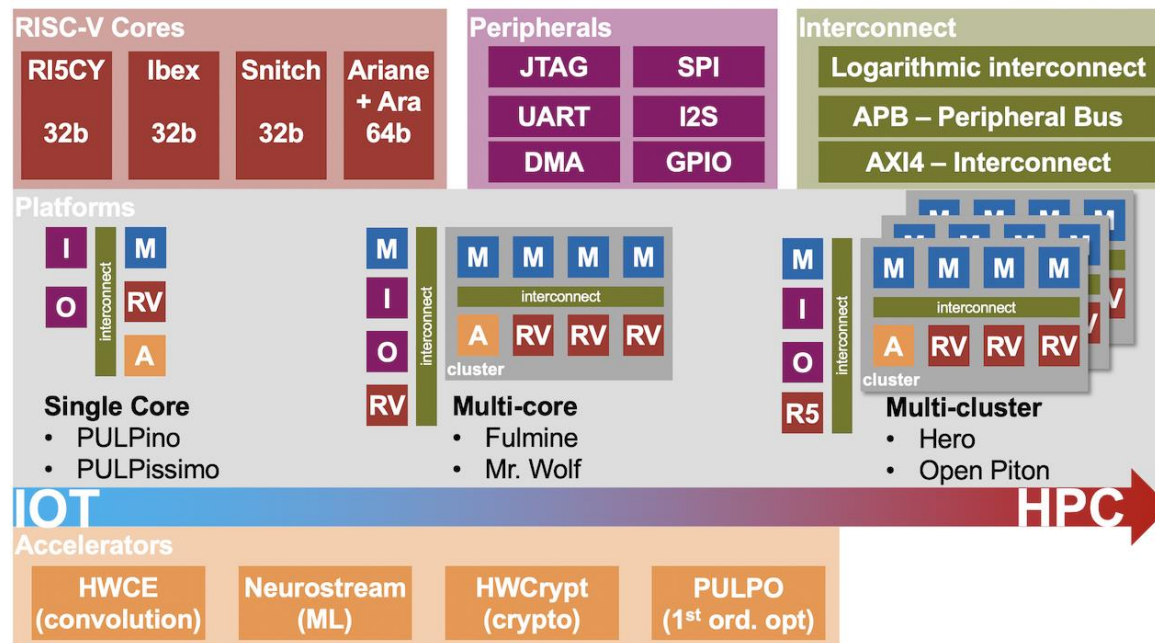


<https://creatorsim.github.io/>

- ▶ CREATOR: *didaCtic geneRic assEmbly progrAmming simulaTOR*
- ▶ CREATOR puede simular las arquitecturas RISC-V y MIPS<sub>32</sub>
- ▶ CREATOR puede ejecutarse desde Firefox, Chrome o Edge

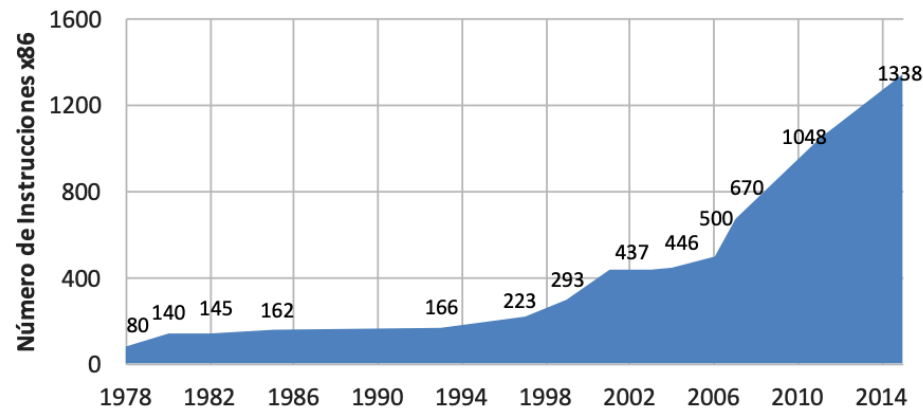
# Motivación para usar RISC-V

- Procesador RISC (*Reduced Instruction Set Computer*)
- Ejemplos de procesadores RISC:
  - RISC-V, ARM, MIPS



# Ventajas de usar RISC-V

- ▶ Arquitectura de hardware libre:
  - ▶ Permite que cualquiera diseñe, fabrique y venda chips y software de RISC-V
- ▶ Conjunto de instrucciones pequeño y sencillo
  - ▶ RV32I -> ~47 instrucciones, RV32IMAF -> ~76
  - ▶ Diferencia con instrucciones de la arquitectura x86



Guía Práctica de RISC-V.  
David Patterson y Andrew Waterman

# Contenidos

1. Fundamentos básicos de la programación en ensamblador
  1. Motivación y objetivos
  2. Introducción a RISC-V32
2. Ensamblador del RISC-V 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



# Juegos de instrucciones RISC-V

- ▶ Juegos de instrucciones:
  - ▶ RV32I: Juego de instrucciones sobre enteros. 32 bits
  - ▶ RV64I: Juego de instrucciones sobre enteros. 64 bits
  - ▶ RV128I: Juego de instrucciones sobre enteros. 128 bits
- ▶ Sobre cada uno de ellos hay diferentes extensiones:
  - ▶ M: instrucciones para multiplicación y división de enteros
  - ▶ F: instrucciones para coma flotante de simple precisión
  - ▶ D: instrucciones para coma flotante de doble precisión
  - ▶ G: Incluye M, F y D
  - ▶ Q: instrucciones de coma flotante de cuádruple precisión
  - ▶ Etc.
- ▶ Ejemplo: RV64IF: procesador RISC-V de 64 bits con instrucciones de coma flotante de simple precisión

# Juegos de instrucciones RISC-V que se van a describir

- ▶ Juegos de instrucciones:
  - ▶ RV32I: Juego de instrucciones sobre enteros. 32 bits
  - ▶ RV64I: Juego de instrucciones sobre enteros. 64 bits
  - ▶ RV128I: Juego de instrucciones sobre enteros. 128 bits
- ▶ Sobre cada uno de ellos hay diferentes extensiones:
  - ▶ M: instrucciones para multiplicación y división de enteros
  - ▶ F: instrucciones para coma flotante de simple precisión
  - ▶ D: instrucciones para coma flotante de doble precisión

# CREATOR

didaCtic geneRc assEmbly progrAMming simulaTOR

Banco de registros

CREATOR 1.2.23 MIPS-32-like  
didaCtic geneRc assEmbly progrAMming simulaTOR

Architecture # Assembly Reset Inst. Run Calculator Info

Break	Address	Label	User Instructions	Loaded Instructions
	0x4	start	addi \$t0 \$t0 100	addi \$t0 \$t0 100
	0x8		li \$t1 100	addi \$t1 \$zero 100
	0xc		beq \$t0 \$t1 7	beq \$t0 \$t1 7
	0x10		addi \$a0 \$a0 10	addi \$a0 \$a0 10
	0x14		mul \$a0 \$a0 \$4	mul \$a0 \$a0 \$4
	0x18		addi \$v0 \$v0 1	addi \$v0 \$v0 1
	0x1c		syscall	syscall
	0x20		addi \$v0 \$v0 9	addi \$v0 \$v0 9
	0x24		syscall	syscall
	0x28		beq \$a0 \$t0 c	beq \$a0 \$t0 0x10
	0x2c		addi \$a0 \$a1 15265	addi \$a0 \$a1 15265
	0x30		addi \$a0 \$a1 105525452	lui \$at 0x064A
	0x34			ori \$at \$at 0x30CC
	0x38			add \$a0 \$a1 \$at
	0x3c		move \$5 \$4	add \$5 \$zero \$4
	0x40		addi \$t1 \$t1 100	addi \$t1 \$t1 100
	0x44		j c	j 0x10

INT Registers Memory Inf Stats

Decimal Hexadecimal

PC	EPC	CAUSE
BADVADDR	STATUS	HI
LO	FIR	FCSR
FCR	FEXR	
zero	a1	v0
v1	a0	a1
a2	a3	t0
t1	t2	t3
t4	t5	t6
t7	s0	s1
s2	s3	s4
s5	s6	s7
t8	t9	k0
k1	gp	sp
fp	ra	

<https://creatorsim.github.io/>

# Banco de registros (enteros)

Nombre ABI	Número	Uso
zero	x0	Constante 0
ra	x1	Dirección de retorno (rutinas)
sp	x2	Puntero a pila
gp	x3	Puntero al área global
tp	x4	Puntero al hilo
t0...t2	x5...x7	Temporal ( <u>NO</u> se conserva entre llamadas)
s0/fp	x8	Temporal (se conserva entre llamadas) / Puntero a marco de pila
s1	x9	Temporal (se conserva entre llamadas)
a0...a1	x10...x11	Argumento de entrada para rutinas/valores de retorno
a2...a7	x12...x17	Argumento de entrada para rutinas
s2...s11	x18...x27	Temporal (se conserva entre llamadas)
t3...t6	x28...x31	Temporal ( <u>NO</u> se conserva entre llamadas)

- ▶ Hay 32 registros
  - ▶ 4 bytes de tamaño (una palabra)
  - ▶ Doble nombrado:
    - ▶ Lógico: nombre ABI (*Application Binary Interface*)
    - ▶ Numérico: con **x** al principio
- ▶ Convenio de uso
  - ▶ Reservados
  - ▶ Argumentos
  - ▶ Resultados
  - ▶ Temporales
  - ▶ Punteros

# Transferencia de datos (registros enteros)

## ► Copia **datos**:

- entre **registros**
- entre **registros y memoria**

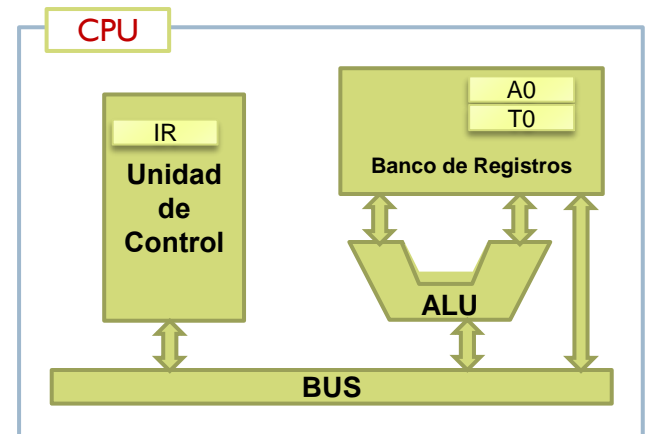
## ► Ejemplos:

- Copia de registro a registro

**mv** a0 t0

- Carga inmediata

**li** t0 5



```
mv    a0 t0    # a0 ← t0
li    t0 5      # t0 ← 000....00101
```

# Aritméticas (registros enteros)

- ▶ Realiza operaciones aritméticas de enteros en Complementos a dos

- ▶ Ejemplos (ALU):

- ▶ Sumar

`add t0 t1 t2`       $\# t0 \leftarrow t1 + t2$

`addi t0 t1 5`       $\# t0 \leftarrow t1 + 5$

- ▶ Restar

`sub t0 t1 t2`       $\# t0 \leftarrow t1 - t2$

- ▶ Multiplicar

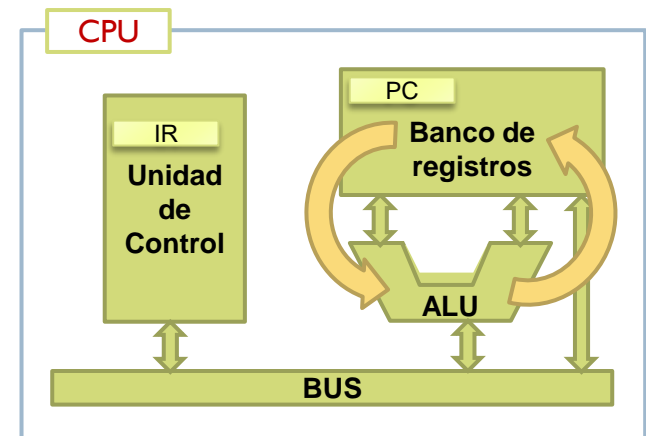
`mul t0 t1 t2`       $\# t0 \leftarrow t1 * t2$

- ▶ División entera (5 / 2=2)

`div t0 t1 t2`       $\# t0 \leftarrow t1 / t2$

- ▶ Resto de la división (5 % 2=1)

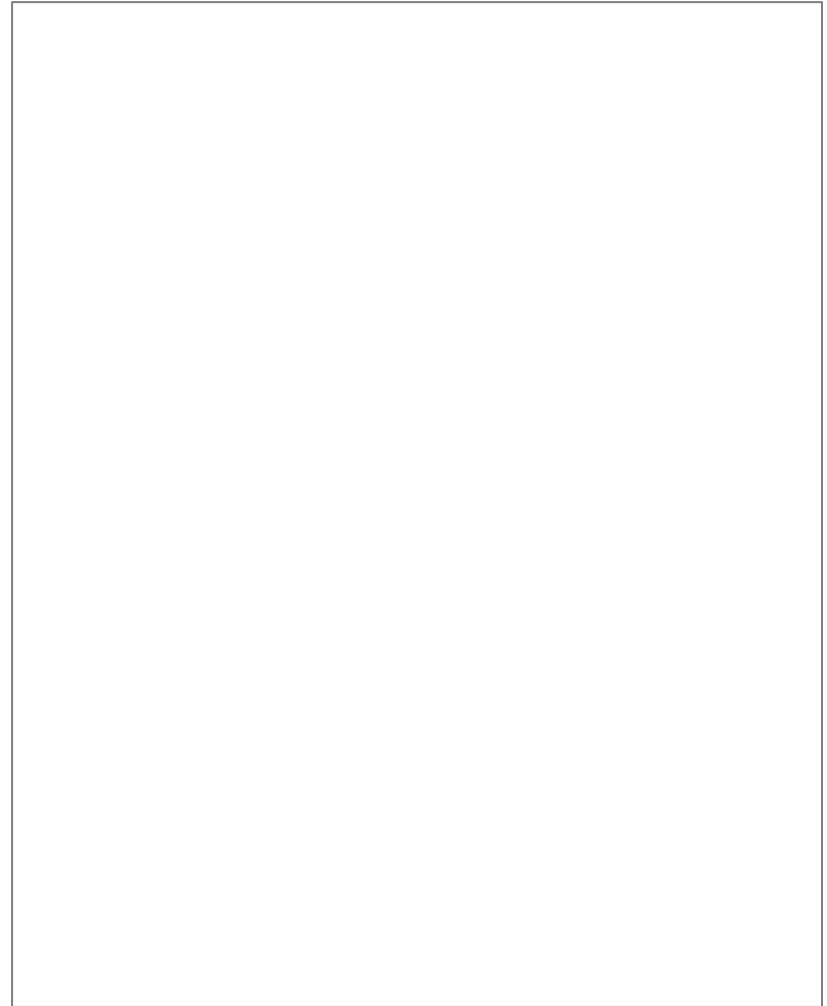
`rem t0 t1 t2`       $\# t0 \leftarrow t1 \% t2$



# 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

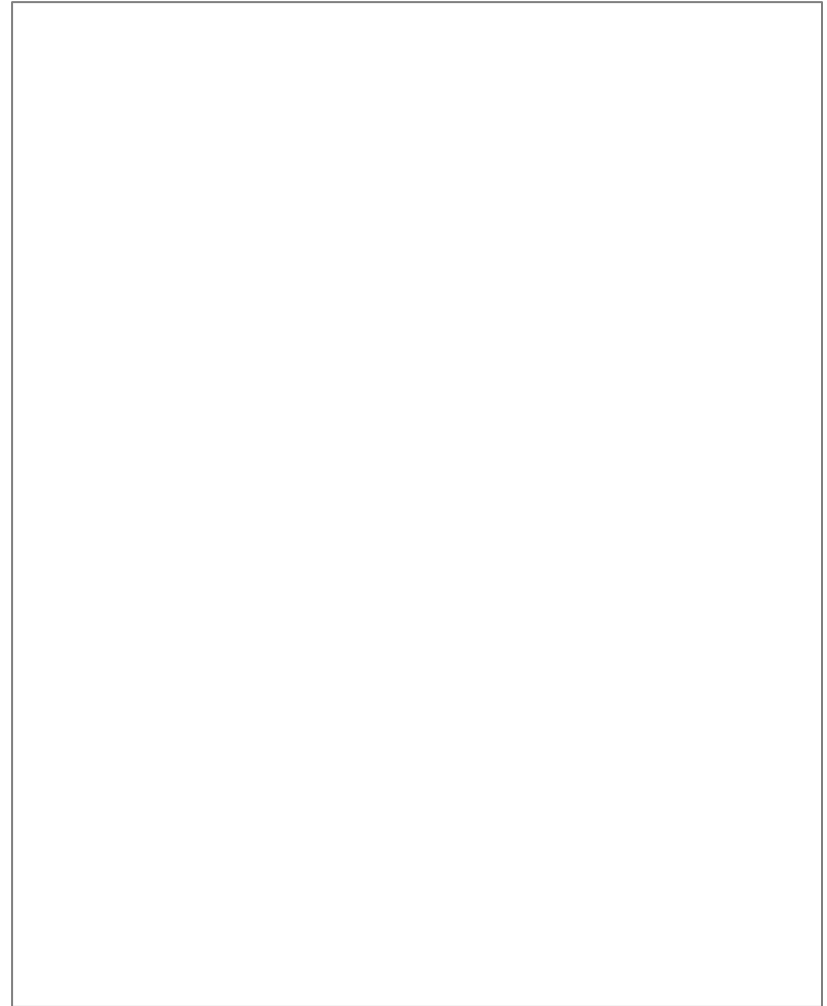
```
int a = 5;
```

```
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)
```

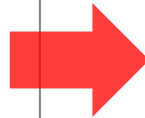
```
li  t1 5  
li  t2 7  
li  t3 8
```

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

# 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
```

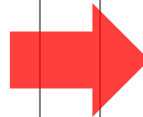


**¿Y usando menos  
instrucciones?**

# 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  
add     t4 x0 t4
```

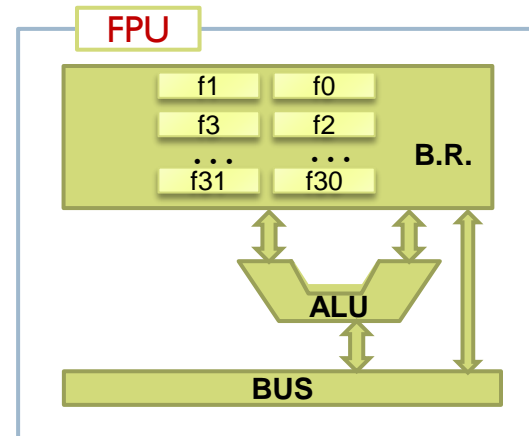
# Banco de registros (coma flotante)

Nombre ABI	Nombre	Uso
ft0...ft7	f0 ... f7	Temporales (como los t...)
fs0...fs1	f8 ... f9	Se guardan (como los s...)
fa0...fa1	f10 ... f11	Argumentos/retorno (como los a...)
fa2...fa7	f12 ... f17	Argumentos (como los a...)
fs2...fs11	f18 ... f27	Se guardan (como los s...)
ft8...ft11	f28 ... f31	Temporales (como los t...)

- ▶ Hay 32 registros
- ▶ El registro ft0 no tiene su valor a 0
- ▶ En la extensión de simple precisión los registros son de 32 bits (4 bytes)
- ▶ En la extensión de doble precisión los registros son de 64 bits (8 bytes) y pueden almacenar:
  - ▶ Valores de simple precisión en los 32 bits inferiores del registro
  - ▶ Valores de do le precisión en los 64 bits del registro

# Banco de registros de coma flotante

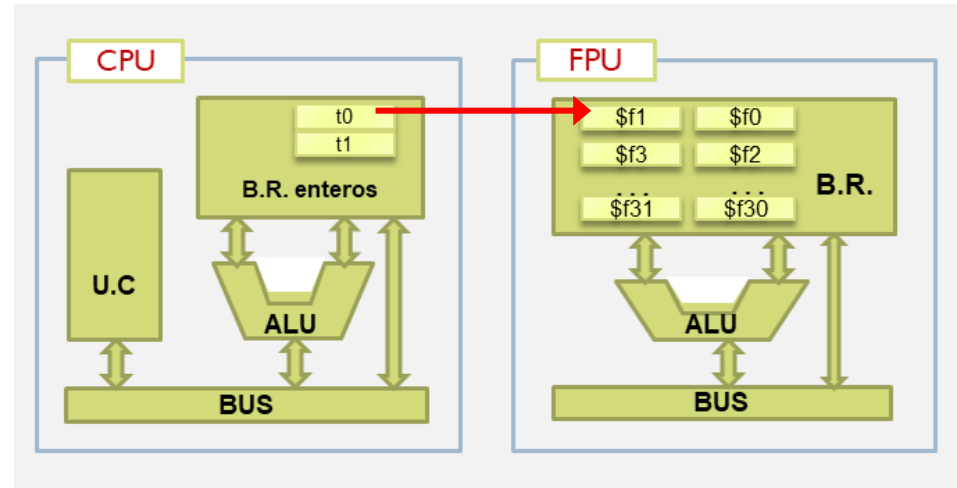
- ▶ Se tiene 32 registros de coma flotante adicionales a los de entero:
  - ▶ De f0 a f31
- ▶ Copia de registros (.s .d):
  - ▶ `fmv.s rd rs` #  $rd = rs$
- ▶ Operaciones aritmética comunes (.s .d):
  - ▶ `fadd.s rd rs1 rs2` #  $rd = rs1 + rs2$
  - ▶ `fsub.s rd rs1 rs2` #  $rd = rs1 - rs2$
  - ▶ `fmul.s rd rs1 rs2` #  $rd = rs1 * rs2$
  - ▶ `fdiv.s rd rs1 rs2` #  $rd = rs1 / rs2$
  - ▶ `fmin.s rd rs1 rs2` #  $rd = \min(rs1, rs2)$
  - ▶ `fmax.s rd rs1 rs2` #  $rd = \max(rs1, rs2)$
  - ▶ `fsqrt.s rd rs1` #  $rd = \sqrt{rs1}$
  - ▶ `fmadd.s rd rs1 rs2 rs3` #  $rd = rs1 \times rs2 + rs3$
  - ▶ `fmsub.s rd rs1 rs2 rs3` #  $rd = rs1 \times rs2 - rs3$
  - ▶ `fabs.s rd rs` #  $rd = |rs|$
  - ▶ `fneg.s rd rs` #  $rd = -rs$



# Operaciones de copia (registros enteros <-> registros coma flotante)

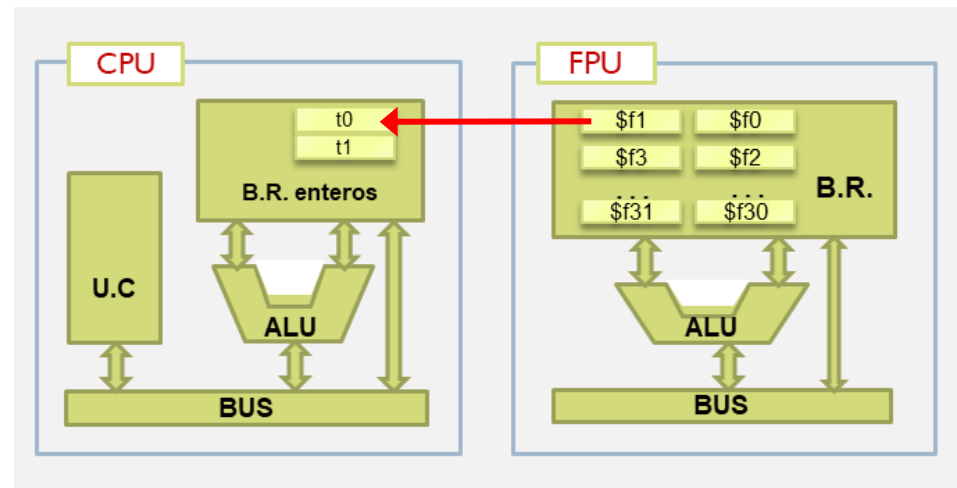
## `fmv.w.x rd rs`

- Copia desde reg. entero `rs` a reg. flotante `rd` (*single precision*)



## `fmv.x.w rd rs`

- Copia desde reg. flotante `rs` (*single precision*) a reg. entero `rd`



# Operaciones de conversión (1 / 3)

entero <-> simple precisión

## ▶ `fcvt.w.s rd, rs`

- ▶ Convierte de simple precisión (valor en registro flotante rs) a entero de 32 bits **con** signo (registro entero rd).

## ▶ `fcvt.wu.s rd, rs`

- ▶ Convierte de simple precisión (valor en registro flotante rs) a entero de 32 bits **sin** signo (registro entero rd).

## ▶ `fcvt.s.w rd, rs`

- ▶ Convierte de entero de 32 bits **con** signo (valor en registro entero rs) a simple precisión (registro flotante rd).

## ▶ `fcvt.s.wu rd, rs`

- ▶ Convierte de entero de 32 bits **sin** signo (valor en registro entero rs) a simple precisión (registro flotante rd).



# Operaciones de conversión (2/3)

entero <-> doble precisión

## ► `fcvt.w.d rd, rs`

- Convierte de doble precisión (valor en registro flotante rs) a entero de 32 bits **con** signo (registro entero rd).

## ► `fcvt.wu.d rd, rs`

- Convierte de doble precisión (valor en registro flotante rs) a entero de 32 bits **sin** signo (registro entero rd).

## ► `fcvt.d.w rd, rs`

- Convierte de entero de 32 bits **con** signo (valor en registro entero rs) a doble precisión (registro flotante rd).

## ► `fcvt.d.wu rd, rs1`

- Convierte de entero de 32 bits **sin** signo (valor en registro entero rs) a doble precisión (registro flotante rd).

# Operaciones de conversión (3/3)

doble precisión  $\leftrightarrow$  simple precisión

## ► `fcvt.s.d rd, rs1`

- Convierte de doble precisión (valor en registro flotante rs) a simple precisión (registro flotante rd).

## ► `fcvt.d.s rd, rs`

- Convierte de simple precisión (valor en registro flotante rs) a doble precisión (registro flotante rd).

# Clasificación de números en coma flotante

- ▶ `fclass.s rd, rs1` (simple precisión)
- ▶ `fclass.d rd, rs1` (doble precisión)
- ▶ Escribe en `rd` el tipo de número de coma flotante del registro `rs1`:

Valor en rd	Significado
0	-Inf
1	Número normalizado negativo
2	Número no normalizado negativo
3	-0
4	+0
5	Numero no normalizado positivo
6	Número normalizado positivo
7	+Inf
8	NaN (no silencioso)
9	NaN (silencioso)

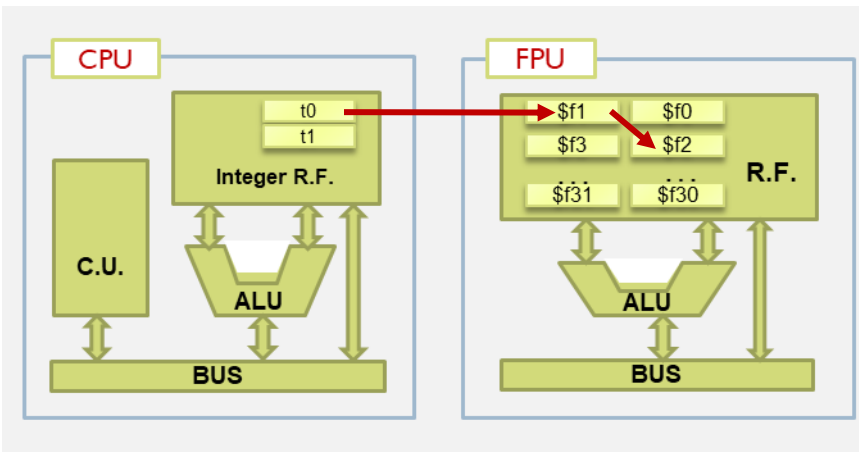
# Ejemplo

```
float PI    = 3,1415;
int  radio = 4;
float length;

length = PI * radio;
```

```
.text
main:
```

```
li t0, 0x40490E56
   # no existe li.s
   # 0x40490E56 es la
   # representación IEEE754
   # en hexadecimal de 3.1415
fmv.w.x ft0, t0  # ft0 ← t0
li      t1  4    # 4 en Ca2
fcvt.s.w ft1, t1  # 4 en ieee754
fmul.s  ft0, ft0, ft1
```



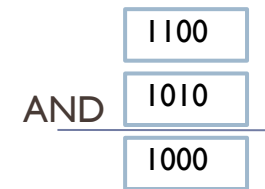
# Lógicas

- ▶ Operaciones **booleanas**

- ▶ Ejemplos:

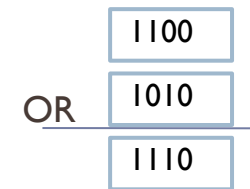
- ▶ AND

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



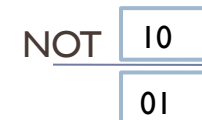
- ▶ OR

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



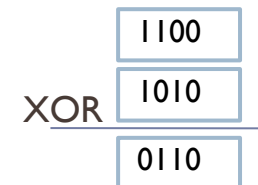
- ▶ NOT

`not t0 t1` ( $t0 = ! t1$ )  
`xori t0 t1 -1`



- ▶ XOR

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

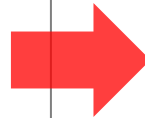


# Ejercicio

```
li t0, 5
```

```
li t1, 8
```

```
and t2, t1, t0
```



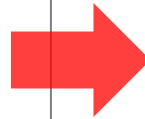
¿Cuál será el valor  
almacenado en t2?

# Ejercicio (solución)

li t0, 5

li t1, 8

and t2, t1, t0



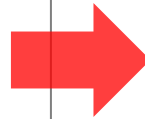
00...0101 t0

00...1000 t1

and 00...0000 t2

# Ejercicio

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



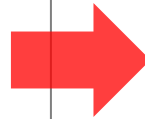
¿Qué permite hacer un and  
con 0x007FFFFFFF?



# Ejercicio (solución)

```
li t0, 5
li t1, 0x007FFFFFFF

and t2, t1, t0
```



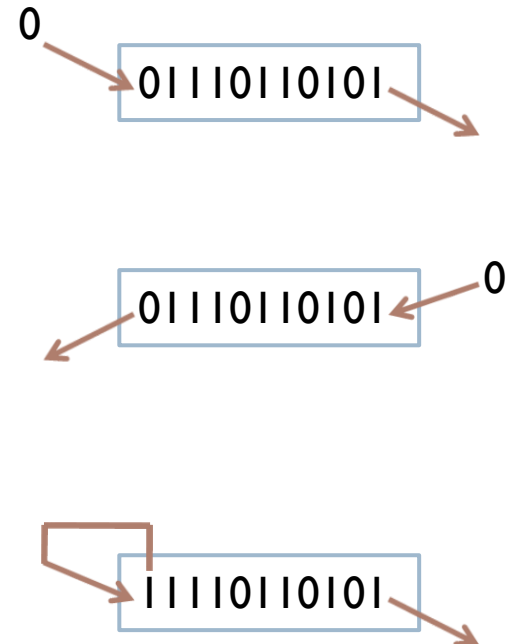
¿Qué permite hacer un and con 0x007FFFFFFF?

Obtener los 23 bits menos significativos

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

# Desplazamientos

- ▶ De movimiento de **bits**. Solo sobre registros enteros
- ▶ Ejemplos:
  - ▶ Desplazamiento **lógico** a la derecha  
`srli t0 t0 4` ( $t0 = t0 \gg 4$  bits)
  - ▶ Desplazamiento **lógico** a la izquierda  
`slli t0 t0 5` ( $t0 = t0 \ll 5$  bits)
  - ▶ Desplazamiento **aritmético**  
`srai t0 t0 2` ( $t0 = t0 \gg 2$  bits)

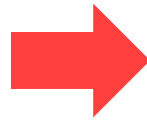


# Ejercicio (solución)

```
li t0, 5
```

```
li t1, 6
```

```
srai t0, t1, 1
```



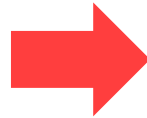
- ¿Cuál es el valor de t0?

000 .... 0110 t1

Se desplaza 1 bit a la derecha (/2)

000 ..... 0011 t0

```
slli t0, t1, 1
```



- ¿Cuál es el valor de t0?

000 .... 0110 t1

Se desplaza 1 bit a la izquierda (x2)

000 ..... 1100 t0

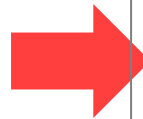
# Ejercicio

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  
srli  t1 t0 31
```

# Instrucciones de comparación (registros enteros)

- ▶ `slt rd, rs1, rs2`      if ( $s(rs1) < s(rs2)$ )     $rd = 1$ ; else  $rd = 0$
- ▶ `sltu rd, rs1, rs2`      if ( $u(rs1) < u(rs2)$ )     $rd = 1$ ; else  $rd = 0$
- ▶ `slti rd, rs1, 5`        if ( $s(rs1) < s(5)$ )      $rd = 1$ ; else  $rd = 0$
- ▶ `sltiu rd, rs1, 5`       if ( $u(rs1) < u(5)$ )      $rd = 1$ ; else  $rd = 0$
- ▶ `seqz rd, rs1`            if ( $rs1 == 0$ )             $rd = 1$ ; else  $rd = 0$
- ▶ `snez rd, rs1`            if ( $rs1 != 0$ )             $rd = 1$ ; else  $rd = 0$
- ▶ `sgtz rd, rs1`            if ( $rs1 > 0$ )             $rd = 1$ ; else  $rd = 0$
- ▶ `sltz rd, rs1`            if ( $rs1 < 0$ )             $rd = 1$ ; else  $rd = 0$

# Instrucciones de comparación (registros en coma flotante)

## ► Simple precisión

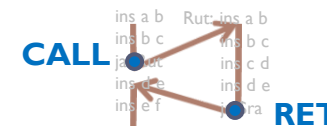
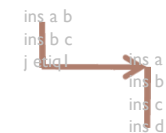
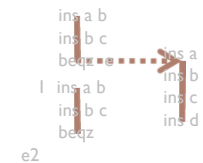
- `feq.s rd, rs1, rs2`      if (rs1 == rs2) rd = 1; else rd = 0
- `fle.s rd, rs1, rs2`      if (rs1 <= rs2) rd = 1; else rd = 0
- `flt.s rd, rs1, rs2`      if (rs1 < rs2) rd = 1; else rd = 0

## ► Doble precisión:

- `feq.d rd, rs1, rs2`      if (rs1 == rs2) rd = 1; else rd = 0
- `fle.d rd, rs1, rs2`      if (rs1 <= rs2) rd = 1; else rd = 0
- `flt.d rd, rs1, rs2`      if (rs1 < rs2) rd = 1; else rd = 0

# 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  $t0 \neq t1$
    - ▶ Ej: `bne t0 t1 etiqueta`
  - ▶ Bifurcación o salto incondicional:
    - ▶ El salto se realiza siempre
    - ▶ Ej: `j etiqueta`
  - ▶ Llamada a procedimiento:
    - ▶ Ej: `jal ra subrutina ..... jr ra`





# Instrucciones de bifurcación

## ► Condicional (solo con registros enteros):

- `beq t0 t1 etiq` # salta a etiq1 si `t0 == t1`
- `bne t0 t1 etiq` # salta a etiq1 si `t0 != t1`
- `blt t0 t1 etiq` # salta a etiq1 si `t0 < t1`
- `bltu t0 t1 etiq` # salta a etiq1 si `t0 < t1` (unsigned)
- `bge t0 t1 etiq` # salta a etiq1 si `t0 >= t1`
- `bgeu t0 t1 etiq` # salta a etiq1 si `t0 >= t1` (unsigned)

(como pseudoinstrucciones)

- `bgt t0 t1 etiq` # salta a etiq1 si `t0 > t1`
- `ble t0 t1 etiq` # salta a etiq1 si `t0 <= t1`

# Instrucciones de bifurcación

## etiqueta de salto

### ► Condicional (solo con registros enteros):


- `beq t0 t1 etiq` # salta a etiq1 si `t0 == t1`
- `bne t0 t1 etiq` # salta a etiq1 si `t0 != t1`
- `blt t0 t1 etiq` # salta a etiq1 si `t0 < t1`
- `bltu t0 t1 etiq` # salta a etiq1 si `t0 < t1` (unsigned)
- `bge t0 t1 etiq` # salta a etiq1 si `t0 >= t1`
- `bgeu t0 t1 etiq` # salta a etiq1 si `t0 >= t1` (unsigned)
- `bgt t0 t1 etiq` # salta a etiq1 si `t0 > t1`
- `ble t0 t1 etiq` # salta a etiq1 si `t0 <= t1`

### ► Incondicional:

- `j etiq` # salta a etiq. Equivalente a `beq x0 x0 etiq`

`etiq` hace referencia una instrucción (representa a una dirección de memoria donde se encuentra la instrucción) a la que se salta:

```
add    t1, t2, t3
j      dir_salto
add    t2, t3, t4
li     t4, 1
dir_salto: li t0, 4
```



# Estructuras de control if

beq	t1 = t0
bne	t1 != t0
bge	t1 >= t0
ble	t0 <= t1
blt	t1 < t0
bgt	t0 > t1

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

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

# Estructuras de control if

**CREATOR**

beq	t1 = t0
bne	t1 != t0
bge	t1 >= t0
ble	t0 <= t1
blt	t1 < t0
bgt	t0 > t1

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

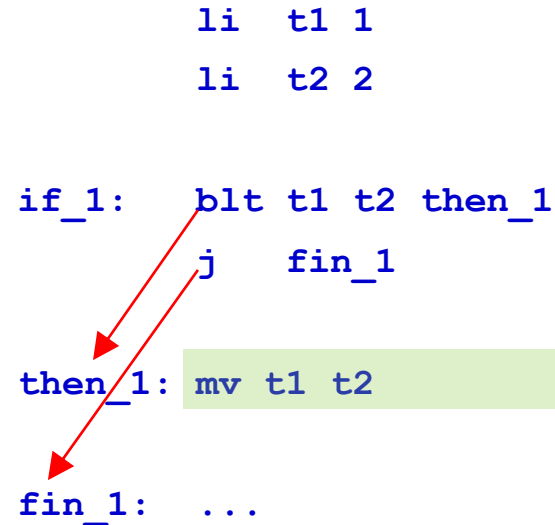
main ()
{
    if (a < b) {
        a = b;
    }
    ...
}
```

```
li    t1 1
li    t2 2

if_1:  blt t1 t2 then_1
      j    fin_1

then_1: mv t1 t2

fin_1:  ...
```



# Estructuras de control if

**CREATOR**

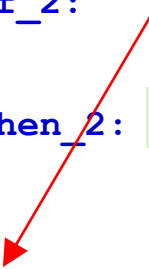
beq	t1 = t0
bne	t1 != t0
bge	t1 >= t0
ble	t0 <= t1
blt	t1 < t0
bgt	t0 > t1

```
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: mv t1 t2
fin_2: ...
```



# Estructuras de control if-else

**CREATOR**

beq	t1 = t0
bne	t1 != t0
bge	t1 >= t0
ble	t0 <= t1
blt	t1 < t0
bgt	t0 > t1

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

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

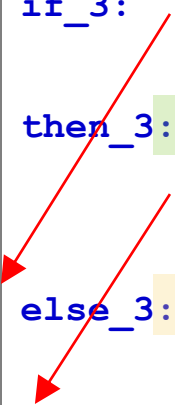
```
li t1 1
li t2 2

if_3: bge t1 t2 else_3

then_3: # acción 1
        j fi_3

else_3: # acción 2

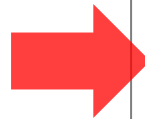
fi_3: ...
```



# Ejercicio

```
int b1 = 4;  
int b2 = 2;
```

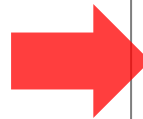
```
if (b2 == 8) {  
    b1 = 1;  
}  
...
```



# Ejercicio (solución)

```
int b1 = 4;  
int b2 = 2;
```

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



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



# Bifurcaciones con números en coma flotante

Saltar a etiqueta si  
ft1 < ft2

```
    flt t0, ft1, ft2
    bne t0, x0, etiqueta
    . . .
etiqueta:
```

# Estructuras de control

## while

beq	t1 = t0
bne	t1 != t0
bge	t1 >= t0
ble	t0 <= t1
blt	t1 < t0
bgt	t0 > t1

```
int i;

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

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

# Estructuras de control while

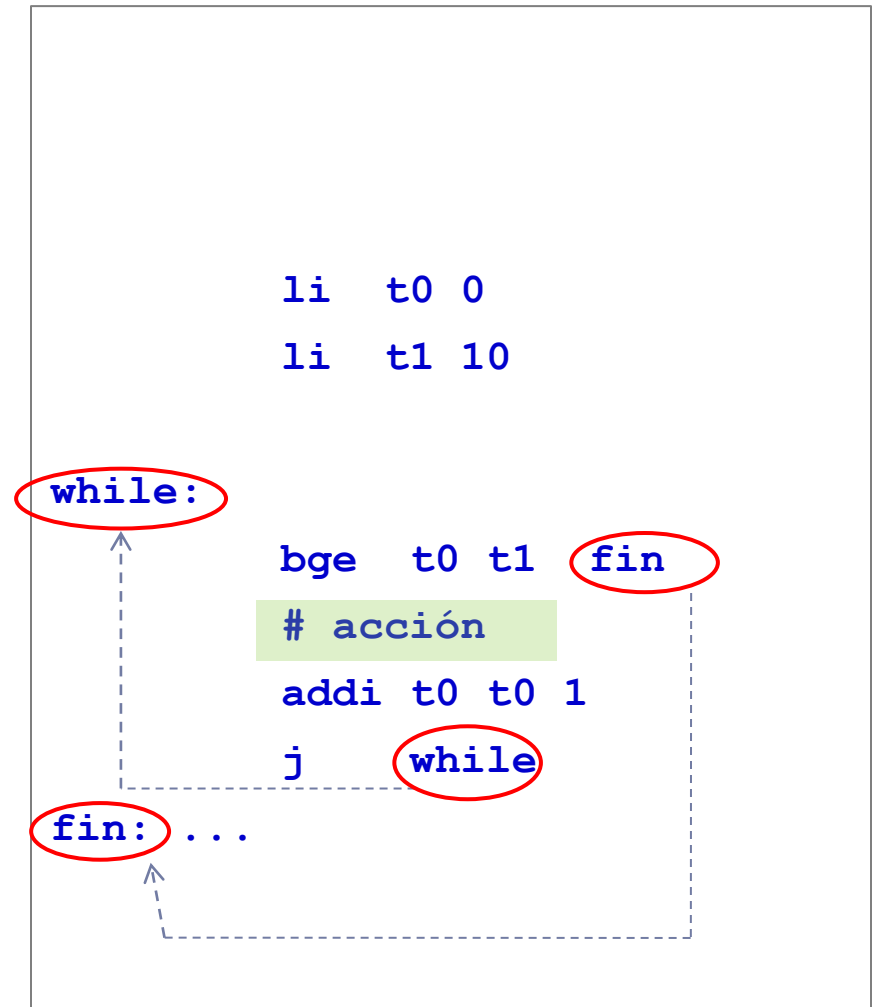
CREATOR

beq	t1 = t0
bne	t1 != t0
bge	t1 >= t0
ble	t0 <= t1
blt	t1 < t0
bgt	t0 > t1

```
int i;

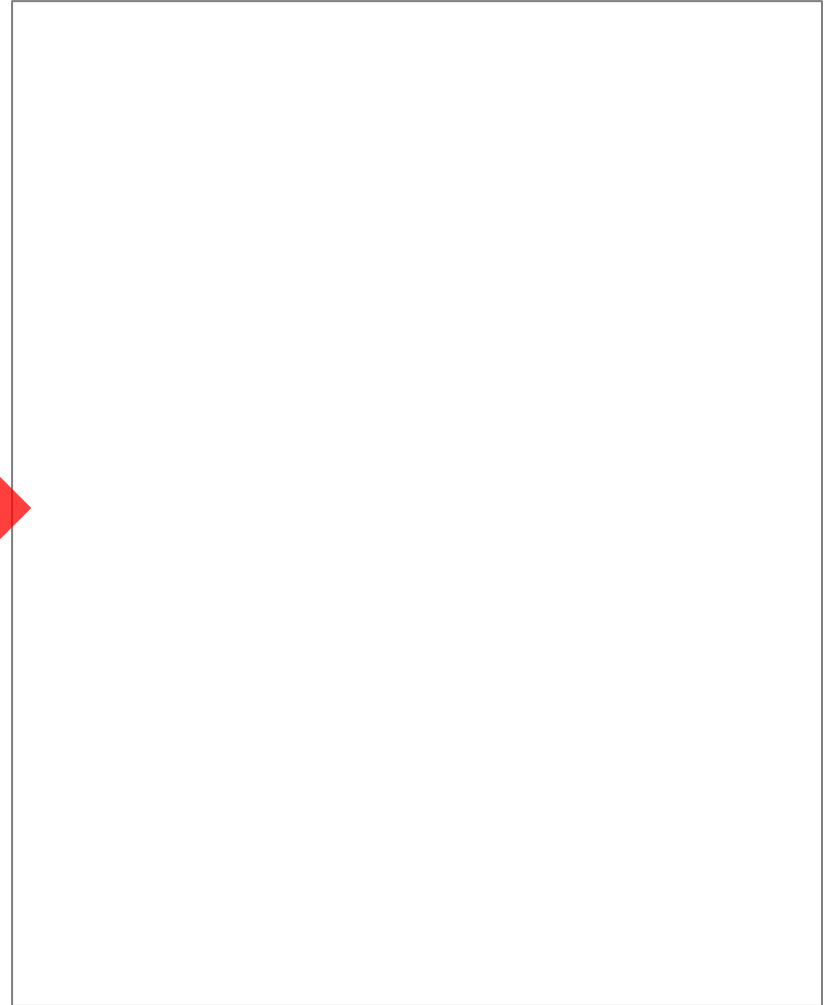
main ()
{
    i=0;
    while (i < 10) {

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



# Ejercicio

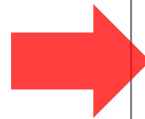
Realice un programa que calcule la suma de los diez primeros números y deje este valor en el registro a0



# Ejercicio (solución)

Realice un programa que calcule la suma de los diez primeros números y deje este valor en el registro a0

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



```
li    a0 0
add   a0 a0 1
add   a0 a0 2
add   a0 a0 3
add   a0 a0 4
add   a0 a0 5
add   a0 a0 6
add   a0 a0 7
add   a0 a0 8
add   a0 a0 9
```

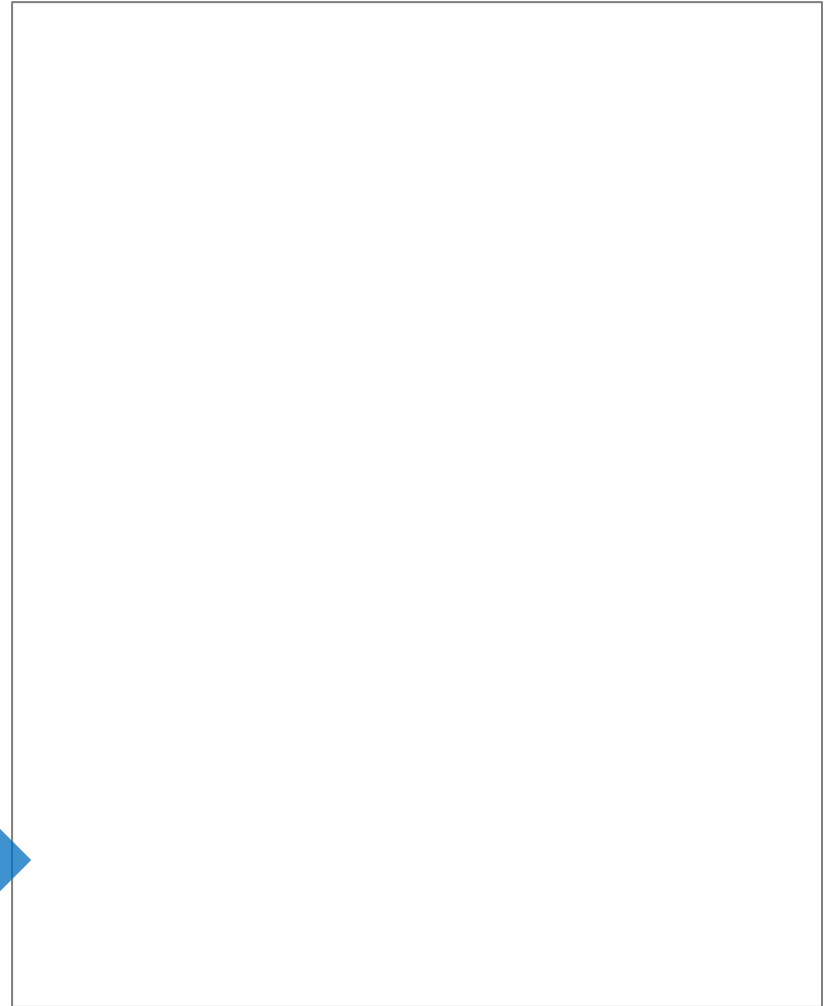
**NO vale  
;-)**

# Ejercicio (solución)

Realice un programa que calcule la suma de los diez primeros números y deje este valor en el registro a0



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

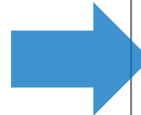


# Ejercicio (solución)

Realice un programa que calcule la suma de los diez primeros números y deje este valor en el registro a0



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



```
li t0 0  
li a0 0  
li t2 10  
  
while1:  
    bgt t0 t2 fin1  
    add a0 a0 t0  
    addi t0 t0 1  
    j 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;  # número
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;  # número
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;  # número
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

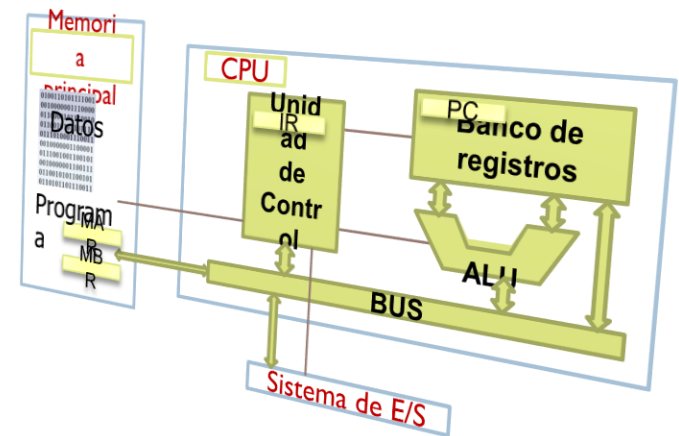
```
i = 0;
n = 45;  # número
s = 0;
while (i < 32)
{
    b = n & 1;
    s = s + b;
    n = n >> 1;
    i = i + 1 ;
}
```

```
li    t0, 0    #i
li    t1, 45   #n
li    t2, 32
li    t3, 0    #s
while: bge    t0, t2, fin
      andi    t4, t1, 1
      add     t3, t3, t4
      srli    t1, t1, 1
      addi    t0, t0, 1
      j       while
fin:   ...
```

# Tipo de instrucciones

## resumen

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



# 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

Grupo ARCOS

**uc3m** | Universidad **Carlos III** de Madrid

# Tema 3: Fundamentos de la programación en ensamblador (I) **Estructura de Computadores**

Grado en Ingeniería Informática  
Grado en Matemática aplicada y Computación  
Doble Grado en Ingeniería Informática y Administración de Empresas



# 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

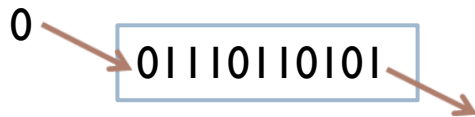
- ▶ Obtener los 16 bits superiores de un registro (t0) y dejarlos en los 16 bits inferiores de otro (t1)



# Ejercicio (solución)

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

```
srli  t1,  t0,  16
```



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

# 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 RISC-V 32 que permita evaluar la expresión anterior. El resultado ha de almacenarse en el registro t5.

# 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,  x0,  else  # cond.
then: li    t1,  1
      j     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   t3,  t2,  t1    # se obtiene el resto
li    t1,  0          # suponer impar
bne   t3,  x0,  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, x0 then     # cond.
else:    li    t1, 0
        j     fin            # incond.
then:    li    t1, 1
fin:     ...
```

# Ejercicio

- ▶ Calcular  $a^n$ 
  - ▶ a en t0
  - ▶ n en t1
  - ▶ El resultado en a0

```
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 a0

```
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
        j      while
fin:    move   a0, t2
```