

## ***Unidad 3. El procesador I: Diseño del juego de Instrucciones. El lenguaje máquina***

---

Escuela Politécnica Superior - UAM

# Índice

- **Arquitectura y Tecnología de Computadores**
- El lenguaje Ensamblador
- ISA MIPS. El juego de instrucciones
- Programación

# Arquitectura y Tecnología de Computadores

APLICACIÓN SOFTWARE	PROGRAMAS
SISTEMAS OPERATIVOS	DRIVERS
ARQUITECTURA	INSTRUCCIONES REGISTROS
MICRO-ARQUITECTURA	CAMINO DE DATOS CONTROLADORES
LÓGICA	SUMADORES MEMORIA
CIRCUITOS DIGITALES	PUERTAS LÓGICAS
CIRCUITOS ANALÓGICOS	AMPLIFICADORES FILTROS
DISPOSITIVOS	TRANSISTORES DIODOS
FÍSICA	ELECTRONES

- **Arquitectura:**

- Es la visión que desde el punto de vista del programador se tiene del sistema computador.
- Viene definida por el juego de instrucciones (operaciones) y por la ubicación de los operandos

- **Microarquitectura:**

- Es la implementación en hardware del computador (U4 y U5).

# Arquitectura y Tecnología de Computadores

- Para poder controlar un computador se debe comprender su lenguaje.
  - ✓ **Instrucciones:** son las palabras que forman el lenguaje de un computador
  - ✓ **Juego de instrucciones:** es el vocabulario del lenguaje de un computador
- Los computadores se diferencian por su juego (set) de instrucciones
- Una instrucción indica la operación a ejecutar y los operandos a utilizar.
  - ✓ **Lenguaje Máquina:** escrito con 1's y 0's es el único lenguaje que el computador es capaz de leer.
  - ✓ **Lenguaje Ensamblador:** muy cercano al lenguaje máquina, representa el primer nivel de abstracción del lenguaje máquina legible para el usuario.

**“Ambos lenguajes están relacionados de forma biunívoca”**

# Arquitectura y Tecnología de Computadores

- Arquitectura MIPS (*Microprocessor without Interlocking Pipeline Stages*)
  - ✓ MIPS es una arquitectura real, en la que se basan muchos de los procesadores actuales de compañías como Silicon Graphics, Sony o Cisco.
  - ✓ MIPS es una arquitectura del tipo RISC (***Reduced Instruction Set Computer***).
  - ✓ El concepto RISC fue desarrollado por Hennessy y Patterson en los años ochenta.
  - ✓ Un diseño RISC está basado en tres principios:
    1. La simplicidad favorece la regularidad
    2. Diseñar el caso común muy rápido
    3. Lo más pequeño es más rápido
- Una vez que se aprende una determinada arquitectura es más fácil entender cualquier otra.

# Índice

- Arquitectura y Tecnología de Computadores
- **El lenguaje Ensamblador**
- ISA MIPS. El juego de instrucciones
- Programación

# El lenguaje Ensamblador

## Primer Principio: La simplicidad favorece la regularidad

En el diseño de las instrucciones se utilizan, en la medida que es posible, formatos consistentes con dos fuentes y un destino. Esta medida facilita su decodificación e implementación en hardware.

Código en alto nivel (lenguaje C)	//	Código ensamblador de MIPS
a = b + c;		add a, b, c
a = b - c;		sub a, b, c

**add, sub:** mnemónicos, indican la operación a ejecutar (suma, resta)

**b, c:** operandos fuente, señalan los datos con los que ejecutar la operación

**a:** operando destino, señala dónde escribir el resultado de la operación

# El lenguaje Ensamblador

## Segundo Principio: Diseñar el caso común muy rápido

**MIPS** es una arquitectura RISC en contraposición a otras arquitecturas CISC (*Complex Instruction Set Computer*), como IA-32 de Intel.

- ✓ En un procesador RISC, como es MIPS, en el juego de instrucciones sólo se incluyen aquellas instrucciones que se usan de forma habitual (las más comunes).
- ✓ El hardware para la decodificación de instrucciones es sencillo y rápido.
- ✓ Las instrucciones más complejas, que son las menos usadas, se ejecutan por medio de instrucciones simples.

### Código en alto nivel (lenguaje C)

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

### // Código ensamblador de MIPS

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

```
sub a, t, d      # a = t - d
```

```
# comentario hasta final de línea
```



# El lenguaje Ensamblador

## Tercer Principio: Lo más pequeño es más rápido

- ✓ Buscar información en unos pocos libros que se encuentran encima de la mesa es más rápido que buscar la misma información en todos los libros de una gran biblioteca.
- ✓ De la misma forma buscar datos en unos pocos registros, es más rápido que encontrarlos entre miles o cientos de miles (memoria).

# Operandos y Registros

- ✓ Un operando es una palabra escrita en binario, que representa un dato variable o una constante denominada dato inmediato.
- ✓ Un computador necesita acceder a ubicaciones físicas, desde las cuales poder leer los operandos fuente (uno o dos) y escribir el resultado en el operando destino que se haya definido.
- ✓ Un computador puede leer/escribir operandos de/en:
  - La Memoria: mucha capacidad pero acceso lento.
  - Los Registros Internos: menor capacidad pero de acceso rápido.
    - MIPS tiene 32 registros de 32 bits.
    - **MIPS es una arquitectura de 32 bits porque opera en la ALU con datos de 32 bits.**

# Operandos y Registros

- Los operandos de las instrucciones hacen referencia a los registros internos (o a datos inmediatos).

Código en alto nivel (lenguaje C)	//	Código ensamblador de MIPS
a = b + c;		#\$s0=a, \$s1=b, \$s2=c
		add \$s0, \$s1, \$s2

Las variables b y c (operandos fuente) **se leen** respectivamente desde los registros \$s1 y \$s2 en donde están almacenadas.

La variable a (operando destino) **se escribe** en el registro \$s0 en donde queda almacenada para posteriores operaciones.

# El conjunto de registros en MIPS

Nombre	Nº Registro	Función
\$0	0	Constante de valor 0
\$at	1	Temporal de uso por el Ensamblador
\$v0-\$v1	2-3	Datos de retorno en procedimientos
\$a0-\$a3	4-7	Datos de entrada (argumentos) en procedimientos
\$t0-\$t7	8-15	Datos internos (variables internas temporales)
\$s0-\$s7	16-23	Datos globales (variables globales permanentes)
\$t8-\$t9	24-25	Datos internos (variables internas temporales)
\$k0-\$k1	26-27	Temporales de uso por el SO
\$gp	28	Puntero Global ( <i>global pointer</i> )
\$sp	29	Puntero de pila ( <i>stack pointer</i> )
\$fp	30	Puntero de página ( <i>frame pointer</i> )
\$ra	31	Dirección de enlace en procedimientos ( <i>link</i> )

# El conjunto de registros en MIPS

- Registros en MIPS:
  - ✓ Se identifican con \$ delante del nombre (o número).
- Aunque todos son equivalentes, algunos registros son de uso específico para ciertas operaciones. Por ejemplo:
  - ✓ \$0 sólo permite lectura y siempre contiene el valor 0.
  - ✓ Los registros \$s0-\$s7 se utilizan para variables.
  - ✓ Los registros temporales \$t0-\$t9 se utilizan para datos intermedios utilizados durante los procedimientos.
  - ✓ Los tres registros punteros (\$gp, \$sp y \$fp) y el registro de enlace, \$ra, señalan siempre a direcciones de memoria.
- En las primeras etapas del aprendizaje de MIPS, sólo se usarán los registros temporales (\$t0-\$t9) y los utilizados para almacenar variables (\$s0-\$s7)

# Operandos y Memoria

- ✓ Con tan sólo decenas de registros internos, no es posible acceder a todos los datos que se manejan en un sistema computador.
- ✓ También es preciso almacenar datos en memoria.
  - La memoria es una estructura física que permite almacenar una gran cantidad de datos.
  - La memoria es un elemento de acceso más lento que los registros
- ✓ Normalmente las variables más habituales (de uso más probable), se almacenan en los registros.
- ✓ Con la combinación adecuada de los registros y memoria, un programa puede acceder a un gran cantidad de datos de forma rápida y eficaz.
- ✓ Las arquitecturas para el acceso a memoria (jerarquías) serán objeto de estudio en cursos posteriores de arquitectura.

# Memoria byte-direccional

- MIPS maneja memoria direccionable por bytes, en donde cada byte se asocia a una única dirección.
- Señalando una dirección y en una única operación, MIPS puede leer/escribir (*load/store*) una palabra de 4 bytes ( $lw$ ,  $sw$ ) o sólo un byte ( $lb$ / $sb$ ).
- Como una palabra (32 bits) ocupa 4 bytes, las direcciones de dos palabras consecutivas se diferencian en 4 unidades.

Word Address	Data								
⋮	⋮								⋮
0000000C	4	0	F	3	0	7	8	8	Word 3
00000008	0	1	E	E	2	8	4	2	Word 2
00000004	F	2	F	1	A	C	0	7	Word 1
00000000	A	B	C	D	E	F	7	8	Word 0

← width = 4 bytes →

# Memoria byte-direccional: Lectura

- En MIPS, la dirección de una palabra tiene que ser un múltiplo de 4.

(DIRECCIÓN ALINEADA)

**Ejemplo:** Leer (*load*) la palabra de la dirección 4 y escribir el dato en el registro \$s3.

**Código ensamblador de MIPS:** `lw $s3, 4($0)`

- Tras la ejecución, \$s3 almacena el valor 0xF2F1AC07

Word Address	Data							
⋮	⋮							
0000000C	4	0	F	3	0	7	8	8
00000008	0	1	E	E	2	8	4	2
00000004	F	2	F	1	A	C	0	7
00000000	A	B	C	D	E	F	7	8

width = 4 bytes



# Memoria byte-direccional: Escritura

**Ejemplo:** Escribir (*store*) la palabra almacenada en el registro \$t7 en la dirección de memoria 0x2C (44<sub>10</sub>)

**Código ensamblador de MIPS:** `sw $t7, 44($0)`

- Tras la ejecución, se almacena en MEM[44] el valor \$t7 = 0xA2E5F0C3

Word Address

Data

0000002C

A2	E5	F0	C3
...	...	...	...
01	EE	28	42
F2	F1	AC	07
AB	CD	EF	78

Word 11

...

...

00000008

Word 2

00000004

Word 1

00000000

Word 0

# Memoria Big-Endian vs Little-Endian

## ¿Cómo se numeran los bytes en una palabra?

- ✓ La dirección de palabra es siempre la misma
- ✓ **Little-endian:** la dirección más baja corresponde al byte menos significativo
- ✓ **Big-endian:** la dirección más baja corresponde al byte más significativo

### Big-Endian

Byte Address			
⋮			
C	D	E	F
8	9	A	B
4	5	6	7
0	1	2	3
MSB		LSB	

### Little-Endian

Byte Address			
⋮			
F	E	D	C
B	A	9	8
7	6	5	4
3	2	1	0
MSB		LSB	

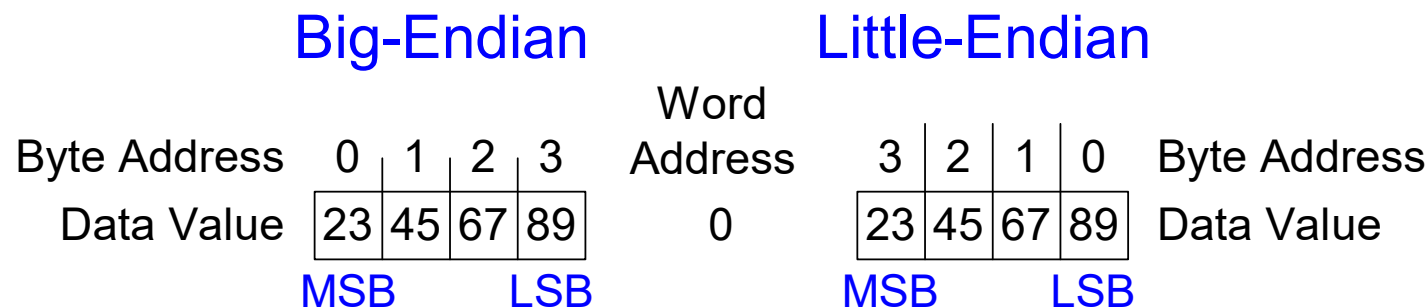
# Memoria Big-Endian vs Little-Endian

**Ejemplo:** Suponer que `$t0` contiene `0x23456789`. Señalar el contenido de `$s0`, después de ejecutar el programa adjunto:

**a)** En un sistema big-endian y **b)** En un sistema little-endian

```
sw $t0, 0($0) # Escribir el contenido de $t0 en MEM[0]
```

```
lb $s0, 1($0) # Leer un byte de MEM[1] y guardarlo en $s0
```



**a) Big-endian: `$s0 = 0x00000045`**

**b) Little-endian: `$s0 = 0x00000067`**

# Operandos: Constantes/Inmediatos

- En MIPS las constantes se denominan datos inmediatos porque se definen en la propia instrucción.
- Al no ser necesario leerlos ni de un registro ni desde memoria, están disponibles de forma inmediata.
- La operación suma inmediata (`addi`) suma un dato inmediato a una variable (almacenada en un registro).
- Un dato inmediato es una constante de 16 bits (positiva o negativa), escrita en complemento a 2.

**¿Es necesario implementar la resta inmediata (`subi`)?**

Código en alto nivel (lenguaje C)	// Código ensamblador de MIPS
	<code># \$s0 = a, \$s1 = b</code>
<code>a = a + 4;</code>	<code>addi \$s0, \$s0, 4</code>
<code>b = a - 12;</code>	<code>addi \$s1, \$s0, -12</code>

# Directivas en MIPS

Además del juego de instrucciones descrito para MIPS, existen una serie de Directivas o pseudo-instrucciones que facilitan la operación de ensamblado.

- **.text** <dirección>: Señala el comienzo de la sección (segmento) de código del usuario.
- **.data** <dirección>: Señala el comienzo de la sección (segmento) de datos del usuario.

Sólo en la sección de datos:

- **.space** n: Asigna n bytes de espacio en memoria

## El intérprete Ensamblador/Máquina. Tabla de símbolos

- ✓ En MIPS las directivas `.data` y `.text`, señalan al programa ensamblador (intérprete), la posición de memoria en donde ubicar los datos y las instrucciones respectivamente.
- ✓ En el proceso de ensamblado, se genera una tabla en donde a cada literal del código ensamblador, se le asocia una posición de memoria, denominada tabla de símbolos.

```
.text 0x0000  
main:  lw $t3, A($0)  
        add $s1, $t3, $t3  
write: sw $s1, B($0)
```

```
.data 0x2000  
A: 5  
.space 8  
B: 0
```

Tabla de símbolos	
Símbolo	Dirección
main	0x00000000
write	0x00000008
A	0x00002000
B	0x0000200C

# Índice

- Arquitectura y Tecnología de Computadores
- El lenguaje Ensamblador
- **ISA MIPS. El juego de instrucciones**
- Programación

# Los Formatos en MIPS

- Los computadores sólo entienden de 1's y 0's.
- El lenguaje Máquina es la representación de las instrucciones en binario.
- Como señala el primer principio “la simplicidad favorece la regularidad”, los datos y las instrucciones en MIPS son de 32 bits.
- Para las instrucciones, MIPS tiene tres tipos de formato:
  - ✓ **R-Type**: todos los operandos están en registros
  - ✓ **I-Type**: aparte de registros hay un operando inmediato (constante)
  - ✓ **J-Type**: instrucciones utilizadas para saltos



# El juego de instrucciones en MIPS (1)

op	Nombre	Descripción	Operación
000000	R-Type	Instrucciones con formato R-Type	Varías
000010	j	Salto incondicional	PC = JTA
000011	jal	Salto a Subrutina	\$ra = PC+4; PC = JTA
000100	beq	Bifurca si igual (Z = 1)	Si ([rs] == [rt]); PC =BTA
000101	bne	Bifurca si distinto (Z = 0)	Si ([rs] != [rt]); PC =BTA
001000	addi	Suma con dato inmediato	[rt] = [rs] + SigImm
001100	andi	AND con dato inmediato	[rt] = [rs] & ZeroImm
001101	ori	OR con dato inmediato	[rt] = [rs]   ZeroImm
001110	xori	XOR con dato inmediato	[rt] = [rs] $\oplus$ ZeroImm
001111	lui	Carga superior dato inmediato	[rt] <sub>31:16</sub> = [imm], [rt] <sub>15:0</sub> = [0...0]
100011	lw	Lee una palabra de memoria	MEM ([rs]+SigImm) => [rt]
101011	sw	Escribe una palabra en memoria	[rt] => MEM ([rs]+SigImm)

# El juego de instrucciones en MIPS (2)

## Formato R-Type (ordenadas por el campo funct):

Funct	Nombre	Descripción	Operación
000000	sll	Despl. Lógico Izquierda	$[rd] = [rt] \ll \text{shamt}$
000010	srl	Despl. Lógico Derecha	$[rd] = [rt] \gg \text{shamt}$
000011	sra	Despl. Aritmético Derecha	$[rd] = [rt] \ggg \text{shamt}$
000100	sllv	Despl. Lógico Izquierda Variable	$[rd] = [rt] \ll [rs]_{4:0}$
000110	srlv	Despl. Lógico Derecha Variable	$[rd] = [rt] \gg [rs]_{4:0}$
000111	srav	Despl. Aritmético Derecha Variable	$[rd] = [rt] \ggg [rs]_{4:0}$
001000	j r	Salta al valor dado por un registro	$PC = [rs]$
100000	add	Sumar	$[rd] = [rs] + [rt]$
100010	sub	Restar	$[rd] = [rs] - [rt]$
100100	and	Función AND	$[rd] = [rs] \& [rt]$
100101	or	Función OR	$[rd] = [rs] \mid [rt]$
100110	xor	Función OR EXCLUSIVA	$[rd] = [rs] \oplus [rt]$
100111	nor	Función NOR	$[rd] = \sim ([rs] \mid [rt])$
101010	slt	Set on less than	$[rs] < [rt] ? [rd]=1 : [rd]=0$

# ISA MIPS. Formato de Tipo-R

## Formato Tipo Registro

- Los operandos se encuentran en 3 registros identificados por 3 campos:
  - **rs, rt**: dos registros operandos fuente (5bits)
  - **rd**: un registro operando destino (5bits)
- Otros campos:
  - **op**: código de operación (*opcode*) (6bits). En las instrucciones de Tipo-R vale 0
  - **funct**: código función (*function*) (6 bits). Junto con op, señala al sistema la operación a ejecutar.
  - **shamt**: campo desplazamiento (5 bits). Señala el desplazamiento en las instrucciones de este tipo, en caso contrario vale 0.

## R-Type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

# Formato de Tipo-R. Ejemplos

	Código Ensamblador	Valores de los campos (decimal)					
		op	rs	rt	rd	shamt	funct
1	add \$s0, \$s1, \$s2	0	17	18	16	0	32
2	sub \$t0, \$t3, \$t5	0	11	13	8	0	34
	Código máquina (HEX)	Código máquina (binario)					
		op (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
1	0x02328020	000000	10001	10010	10000	00000	100000
2	0x016D4022	000000	01011	01101	01000	00000	100010

**Observar** el diferente orden de los registros en ensamblador (add rd, rs, rt) y el orden de los campos en el lenguaje máquina.

# ISA MIPS. Formato de Tipo-I

## Formato Tipo Inmediato

- Hay 3 operandos, 2 en registro y un dato inmediato:
  - **rs**: registro operando fuente (5bits).
  - **rt**: registro operando destino (5bits). **Atención** ahora **rt** es destino.
  - **imm**: dato inmediato (16 bits).
- Otros campos:
  - **op**: código de operación (*opcode*) (6bits). Sólo op señala al sistema la operación a ejecutar.

## I-Type



# Formato de Tipo-I. Ejemplos

	Código Ensamblador	Valores de los campos (decimal)			
		op	rs	rt	imm
1	addi \$s0, \$s1, 5	8	17	16	5
2	addi \$t0, \$s3, -12	8	19	8	-12
3	lw \$t2, 32(\$0)	35	0	10	32
4	sw \$s1, 4(\$t1)	43	9	17	4
	Código máquina (HEX)	Código máquina (binario)			
		op (6)	rs (5)	rt (5)	imm (16)
1	0x22300005	001000	10001	10000	00000000000000101
2	0x2268FFF4	001000	10011	01000	1111111111110100
3	0x8C0A0020	100011	00000	01010	0000000000100000
4	0xAD310004	101011	01001	10001	0000000000000100

**Observar** el diferente orden de los registros en ensamblador (`addi rt, rs, imm;`  
`lw rt, imm(rs);` `sw rt, imm(rs)`) y el orden de los campos en el lenguaje  
 máquina.

# ISA MIPS. Formato de Tipo-J

## *Formato Tipo Salto (jump)*

- Hay sólo dos campos:
  - **op**: código de operación (*opcode*) (6bits). Identifica el tipo de salto.
  - **addr**: dirección de salto (26 bits).

## J-Type



Ejemplos de este tipo de formato se verán más adelante.

# Resumen: Los formatos en MIPS

## R-Type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

## I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

## J-Type

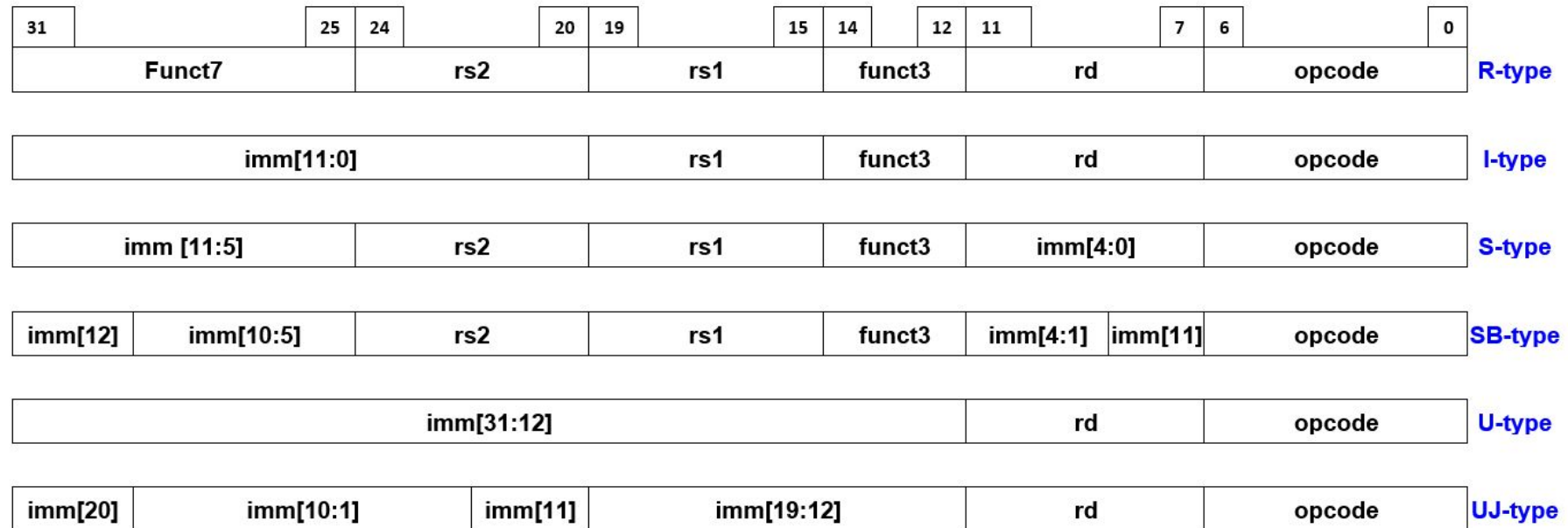
op	addr
6 bits	26 bits



# Una nueva Arquitectura: RISC-V

- ✓ RISC-V es una arquitectura ISA diseñada a partir de 2010 en la UC Berkeley<sup>(1)</sup> para su desarrollo e investigación (*open source*)<sup>(2)</sup> con fines académicos y comerciales.

## Formatos de instrucciones para ISA RV32I



<sup>(1)</sup>A.S. Waterman. PhD Thesis. "Design of the RISC-V Instruction Set Architecture". UC Berkeley. 2016

<sup>(2)</sup> <https://riscv.org/>

# Programas almacenados: ventajas

- Las instrucciones y los datos de 32-bits se encuentran almacenados en memoria.
- La única diferencia entre dos aplicaciones distintas es la secuencia de escritura de las instrucciones del programa.
- Para la ejecución de un nuevo programa:
  - No se precisa modificar el hardware.
  - Sólo es necesario almacenar en memoria el nuevo programa.
  - El procesador lee (captura, *fetch*) secuencialmente las instrucciones de memoria y ejecuta la operación especificada.
- El registro contador de programa (PC, *program counter*) lleva el control de la dirección de la instrucción que se está ejecutando.
- En MIPS, habitualmente los programas se ubican a partir de la dirección de memoria 0x00400000. **En el modo comprimido de MARS en 0x00000000.**

# Ejemplo de un programa almacenado

## Código Ensamblador

lw \$t2, 32(\$0)

add \$s0, \$s1, \$s2

addi \$t0, \$s3, -12

sub \$t0, \$t3, \$t5

## Código Máquina

0x8C0A0020 ← Se ejecuta

0x02328020 ← Se ejecuta

0x2268FFF4 ← Se ejecuta

0x016D4022 ← Se ejecuta

## Memoria Principal

### Dirección

### Instrucciones

...

0040000C

01 6D 40 22

← PC

00400008

22 68 FF F4

← PC

00400004

02 32 80 20

← PC

00400000

8C 0A 00 20

← PC

...

# Interpretación del Código Máquina

- Se comienza con el código de operación (*opcode*, *op*)
- El *op* indica cómo tratar el resto de los bits
  - Si *op* = “000000”
    - Se trata de una instrucción Tipo-R
    - Los bits del campo función (*funct*) indican qué instrucción es
  - Si *op* ≠ “000000”
    - Los bits del campo op indican qué instrucción es

Machine Code

(0x2237FFF1)

op	rs	rt	imm
001000	10001	10111	1111 1111 1111 0001
2	2	3	7 F F F 1

Field Values

op	rs	rt	imm
8	17	23	-15

Assembly Code

addi \$s7, \$s1, -15

(0x02F34022)

op	rs	rt	rd	shamt	funct
000000	10111	10011	01000	00000	100010
0	2	F	3	4	0 2 2

op	rs	rt	rd	shamt	funct
0	23	19	8	0	34

sub \$t0, \$s7, \$s3

# Instrucciones lógicas en MIPS

`and, or, xor, nor, andi, ori, xori`

- `and, andi`: Se utilizan para aplicar máscaras
- `or, ori`: Se utilizan para combinar campos de bits
- `xor, xori`: Se utilizan para comparar bits
- `nor`: Se utiliza para invertir bits ( $A \text{ nor } \$0 == \text{not } A$ )
- **ATENCIÓN:** En las instrucciones lógicas, el dato inmediato de 16-bits se extiende a 32 añadiendo ceros, **no se extiende el signo.**
- La instrucción `nori` no se utiliza (no se implementa)

# Instrucciones lógicas: Codificación

	Código Ensamblador	Valores de los campos (decimal)					
		op	rs	rt	rd	shamt	funct
1	and \$s3, \$s1, \$s2	0	17	18	19	0	36
2	or \$s4, \$s1, \$s2	0	17	18	20	0	37
3	xor \$s5, \$s1, \$s2	0	17	18	21	0	38
4	nor \$s6, \$s1, \$s2	0	17	18	22	0	39
	Código máquina (HEX)	Código máquina (binario)					
		op (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
1	0x02329024	0000 00	10 001	1 0010	1001 1	000 00	10 0100
2	0x0232A025	0000 00	10 001	1 0010	1010 0	000 00	10 0101
3	0x0232A826	0000 00	10 001	1 0010	1010 1	000 00	10 0110
4	0x0232B027	0000 00	10 001	1 0010	1011 0	000 00	10 0111

# Instrucciones lógicas: Codificación

	Código Ensamblador	Valores de los campos (decimal)			
		op	rs	rt	imm
1	andi \$s2, \$s1, 0xFA34	12	17	18	64052
2	ori \$s3, \$s1, 0xFA34	13	17	19	64052
3	xori \$s4, \$s1, 0xFA34	14	17	20	64052
	Código máquina (HEX)	Código máquina (binario)			
		op (6)	rs (5)	rt (5)	Imm (16)
1	0x3232FA34	0011 00	10 001	1 0010	1111 1010 0011 0100
2	0x3633FA34	0011 01	10 001	1 0011	1111 1010 0011 0100
3	0x3A34FA34	0011 10	10 001	1 0100	1111 1010 0011 0100

# Instrucciones lógicas: Ejemplos

## Operandos Fuente

\$s1	1111	1111	1111	1111	0000	0011	1100	0011
\$s2	0100	0110	1010	0001	1111	0000	1011	0111
imm	0000	0000	0000	0000	1111	1010	0011	0100

Instrucción		Resultado								
and \$s3, \$s1, \$s2		\$s3:	0100	0110	1010	0001	0000	0000	1000	0011
or \$s4, \$s1, \$s2		\$s4:	1111	1111	1111	1111	1111	0011	1111	0111
xor \$s5, \$s1, \$s2		\$s5:	1011	1001	0101	1110	1111	0011	0111	0100
nor \$s6, \$s1, \$s2		\$s6:	0000	0000	0000	0000	0000	1100	0100	1000
andi \$s2, \$s1, 0xFA34		\$s2:	0000	0000	0000	0000	0000	0010	0000	0000
ori \$s3, \$s1, 0xFA34		\$s3:	1111	1111	1111	1111	1111	1011	1111	0111
xori \$s4, \$s1, 0xFA34		\$s4:	1111	1111	1111	1111	1111	1001	1111	0111



# Desplazamientos en MIPS

**sll, srl, sra, sllv, srlv, srav**

- **sll**: desplazamiento lógico a la izquierda (*shift left logical*)
  - Ejemplo: `sll $t0, $t1, 5` # `$t0 <= $t1 << 5`
- **srl**: desplazamiento lógico a la derecha (*shift right logical*)
  - Ejemplo: `srl $t0, $t1, 5` # `$t0 <= $t1 >> 5`
- **sra**: desplazamiento aritmético a la derecha (*shift right arithmetic*)
  - Ejemplo: `sra $t0, $t1, 5` # `$t0 <= $t1 >>> 5`
- **sllv**: despl. variable lógico a la izquierda (*shift left logical variable*)
  - Ejemplo: `sllv $t0, $t1, $t2` # `$t0 <= $t1 << $t2`
- **srlv**: despl. variable lógico a la derecha (*shift right logical variable*)
  - Ejemplo: `srlv $t0, $t1, $t2` # `$t0 <= $t1 >> $t2`
- **srav**: despl. variable aritmético a la derecha (*shift right arithmetic variable*)
  - Ejemplo: `srav $t0, $t1, $t2` # `$t0 <= $t1 >>> $t2`

# Desplazamientos: Codificación

	Código Ensamblador	Valores de los campos (decimal)					
		op	rs	rt	rd	shamt	funct
1	sll \$t0, \$t1, 5	0	0	9 (\$t1)	8 (\$t0)	5	0
2	srl \$s2, \$s1, 12	0	0	17 (\$s1)	18 (\$s2)	12	2
3	sra \$s3, \$s1, 4	0	0	17 (\$s1)	19 (\$s3)	4	3
4	sllv \$t0, \$t1, \$t3	0	11 (\$t3)	9 (\$t1)	8 (\$t0)	0	4
	Código máquina (HEX)	Código máquina (binario)					
		op (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
1	0x00094140	00 0000	00 000	0 1001	0100 0	001 01	00 0000
2	0x00119302	00 0000	00 000	1 0001	1001 0	011 00	00 0010
3	0x00119903	00 0000	00 000	1 0001	1001 1	001 00	00 0011
4	0x01694004	00 0000	01 011	0 1001	0100 0	000 00	00 0100

**Observar** el diferente orden de los registros en ensamblador (sllv rd, rt, rs) y el orden de los campos en el lenguaje máquina.

# Desplazamientos: Ejemplos

Operandos Fuente								
\$s1	1111	0000	1111	1111	1111	0000	0000	0000
\$t3	0000	0000	0000	0000	0000	0000	0000	<b>1000</b>
\$t1	<b>0000</b>	<b>0000</b>	0000	0000	0000	0000	1111	1111

Instrucción	Resultado									
sll \$t0, \$s1, 5	\$t0:	0001	1111	1111	1110	0000	0000	0000	<b>0000</b>	<b>0000</b>
srl \$s3, \$s1, 12	\$s3:	<b>0000</b>	<b>0000</b>	<b>0000</b>	1111	0000	1111	1111	1111	1111
sra \$s5, \$s1, 4	\$s5:	<b>1111</b>	1111	0000	1111	1111	1111	0000	0000	0000
sllv \$t0, \$t1, \$t3	\$t0:	0000	0000	0000	0000	1111	1111	<b>0000</b>	<b>0000</b>	<b>0000</b>

En desplazamientos variables se utilizan los **5 lsb** del segundo registro fuente

# Generación de constantes

- Constante de 16 bits usando `addi`:

Código de alto nivel	//	Código Ensamblador MIPS
<code>//int es una palabra de 32-bit</code>		<code># \$s0 = a</code>
<code>// con signo</code>		<code>addi \$s0,\$0,0x4F3C</code>
<code>int a = 0x4F3C;</code>		

- Constante de 32 bits usando carga superior inmediata (`lui`) y `ori`:  
(`lui` carga el dato inmediato de 16 en la mitad superior del registro destino y pone los 16 bits de menos peso a 0)

Código de alto nivel	//	Código Ensamblador MIPS
<code>//int es una palabra de 32-bit</code>		<code># \$s0 = a</code>
<code>// con signo</code>		<code>lui \$s0, 0xFEDC</code>
<code>int a = 0xFEDC8765;</code>		<code>ori \$s0, \$s0, 0x8765</code>

# Instrucciones de Salto en MIPS

- Instrucciones que permiten al programa cambiar el orden de ejecución.

Tipos de saltos o bifurcaciones (*branches*):

## ✓ Saltos condicionales

Salta a la dirección indicada en la instrucción si se cumple la condición

- `beq rs, rt, target;` (*branch if equal*) saltar a target si `[rs] = [rt]`
- `bne rs, rt, target;` (*branch if not equal*) saltar a target si `[rs] ≠ [rt]`

**“LA DECISIÓN DEL SALTO SE TOMA EN TIEMPO DE EJECUCIÓN”**

## ✓ Saltos incondicionales

Salta a la dirección indicada en la instrucción **sin condición alguna**

- `j target,` (*jump*) saltar. La dirección se indica en la instrucción.
- `jal target,` (*jump and link*) saltar y enlazar. La dirección de salto se indica en la instrucción y la dirección actual (+4) se guarda en el registro \$ra.
- `jr,` (*jump register*) saltar registro. La dirección de salto se indica en un registro codificado en el campo rs. `[rs] = target.`

# Salto condicionales ejemplos (beq)

# Dirección	# Ensamblador de MIPS	
0x00400000	addi \$s0, \$0, 4	# \$s0 = 0 + 4 = 4
0x00400004	addi \$s1, \$0, 1	# \$s1 = 0 + 1 = 1
0x00400008	sll \$s1, \$s1, 2	# \$s1 = 1 << 2 = 4
0x0040000C	beq \$s0, \$s1, target	# <b>salta a target si \$s0 = \$s1</b>
0x00400010	addi \$s1, \$s1, 1	# <b>Si hay salto NO se ejecuta</b>
0x00400014	sub \$s1, \$s1, \$s0	# <b>Si hay salto NO se ejecuta</b>
. . .	. . .	
0x004000FC	target:	# <b>Si hay salto SÍ se ejecuta</b>
	add \$s1, \$s1, \$s0	# \$s1 = 4 + 4 = 8

Con la **Etiqueta** “target” se indica en el programa la dirección de la instrucción a la que se accede en caso de que el salto sea efectivo (**branch taken**).

Para una **Etiqueta**, no se pueden emplear palabras reservadas y se debe terminar por dos puntos (:).

# Salto condicionales ejemplos (bne)

# Dirección	# Ensamblador de MIPS	
0x00400000	addi \$s0, \$0, 4	# \$s0 = 0 + 4 = 4
0x00400004	addi \$s1, \$0, 1	# \$s1 = 0 + 1 = 1
0x00400008	sll \$s1, \$s1, 2	# \$s1 = 1 << 2 = 4
0x0040000C	bne \$s0, \$s1, target	# <b>salta a target si \$s0 ≠ \$s1</b>
0x00400010	addi \$s1, \$s1, 1	# \$s1 = 4 + 1 = 5
0x00400014	sub \$s1, \$s1, \$s0	# \$s1 = 5 - 4 = 1
. . .	. . .	
0x004000FC	target:	
	add \$s1, \$s1, \$s0	# <b>NO se ejecuta</b>

Si se suponen los mismos valores que en el caso anterior, al cambiar la condición para el salto, ahora no se produce (**branch not taken**) y el programa continúa con la ejecución secuencial.

# Salto condicionales: Codificación

	Código Ensamblador	Valores de los campos (decimal)			
		op	rs	rt	imm (BTA, Branch Target)
1	beq \$s0, \$s1, target	4	16 (\$s0)	17 (\$s1)	59
2	bne \$s0, \$s1, target	5	16 (\$s0)	17 (\$s1)	59
	Código máquina (HEX)	Código máquina (binario)			
		op (6)	rs (5)	rt (5)	imm (16)
1	0x1211003B	000100	10000	10001	0000000000111011
2	0x1611003B	000101	10000	10001	0000000000111011

Los saltos condicionales utilizan el formato I-Type

En el dato inmediato se señala la distancia entre la dirección de la siguiente instrucción a la de salto y la dirección de la instrucción objetivo (*target*) en palabras (bytes/4) y con signo (positivo si adelante, negativo si atrás).

Si se toma el salto:  $PC_{\text{nuevo}} = \underbrace{BTA}_{\text{Branch Target Address}} = PC_{\text{antiguo}} + 4 + (\text{SignImm} \ll 2)$



# Saltos condicionales: Codificación

# Dirección	# Ensamblador de MIPS		
0x00400000	addi	\$s0, \$0, 4	# \$s0 = 0 + 4 = 4
0x00400004	addi	\$s1, \$0, 1	# \$s1 = 0 + 1 = 1
0x00400008	sll	\$s1, \$s1, 2	# \$s1 = 1 << 2 = 4
<b>0x0040000C</b>	bne	\$s0, \$s1, target	# <b>salta a target si \$s0 ≠ \$s1</b>
0x00400010	addi	\$s1, \$s1, 1	# \$s1 = 4 + 1 = 5
0x00400014	sub	\$s1, \$s1, \$s0	# \$s1 = 5 - 4 = 1
. . .	. . .		
<b>0x004000FC</b>	target:		
	add	\$s1, \$s1, \$s0	# <b>NO se ejecuta</b>

$$\text{BTA} = \text{PC} + 4 + (\text{SignImm} \ll 2)$$

$$\text{SignImm} = (\text{BTA} - \text{PC} - 4) \gg 2$$

$$\text{SignImm} = (0x004000FC - 0x0040000C - 4) \gg 2$$

$$\text{SignImm} = (0x000000F0 - 4) \gg 2$$

$$\text{SignImm} = (0x000000EC) \gg 2$$

$$\text{SignImm} = 0x0000003B = 59$$

	op	rs	rt	imm (BTA, Branch Target)
beq \$s0, \$s1, target	4	16 (\$s0)	17 (\$s1)	59

# Salto condicionales: Codificación

# Dirección	#Código máquina	# Ensamblador de MIPS
0x00400000	0x20100004	addi \$s0, \$0, 4
0x00400004	0x20110001	addi \$s1, \$0, 1
0x00400008	0x16110001	<b>bne \$s0, \$s1, target</b>
0x0040000C	0x20110005	addi \$s1, \$0, 5
0x00400010	0x22310001	addi \$s1, \$s1, 1
0x00400014	0x02308822	sub \$s1, \$s1, \$s0

target:

¿Dónde estará escrita la etiqueta *target*?

bne \$s0, \$s1, target => 0x16110001 =>  
 000101 10000 10001 00000000000000001  
 OPCODE RS RT IMM

BTA = PC + 4 + (SignImm << 2)

BTA = 0x00400008 + 4 + (0x0001 << 2)

BTA = 0x0040000C + (0x0004)

BTA = 0x00400010

De forma intuitiva:  
 El dato inmediato  
 indicará cuántas  
 instrucciones hay que  
 saltar (arriba o abajo)  
 desde la siguiente  
 instrucción del *branch*.

# Salto condicionales: Codificación

# Dirección	#Código máquina	# Ensamblador de MIPS
0x00400000	0x20100004 <b>target:</b>	addi \$s0, \$0, 4
0x00400004	0x20110001	addi \$s1, \$0, 1
0x00400008	0x1611FFFD	<b>bne \$s0, \$s1, target</b>
0x0040000C	0x20110005	addi \$s1, \$0, 5
0x00400010	0x22310001	addi \$s1, \$s1, 1
0x00400014	0x02308822	sub \$s1, \$s1, \$s0

¿Dónde estará escrita la etiqueta *target*?

bne \$s0, \$s1, target => 0x1611FFFD =>  
 000101 10000 10001 1111111111111101  
 OPCODE      RS      RT      IMM

BTA = PC + 4 + (SignImm << 2)  
 BTA = 0x00400008 + 4 + (0xFFFD << 2)  
 BTA = 0x0040000C + (0xFFF4)  
 BTA = 0x00400000

De forma intuitiva:  
 El dato inmediato  
 indicará cuántas  
 instrucciones hay que  
 saltar (arriba o abajo)  
 desde la siguiente  
 instrucción del *branch*.

# Saltos incondicionales: (j y jal)

# Dirección	# Ensamblador de MIPS
0x00002000	addi \$s0, \$0, 4 # \$s0 = 4
0x00002004	addi \$s1, \$0, 1 # \$s1 = 1
0x00002008	j target # <b>salta (siempre) a target</b>
0x0000200C	sra \$s1, \$s1, 2 # <b>nunca se ejecuta</b>
0x00002010	addi \$s1, \$s1, 1 # <b>nunca se ejecuta</b>
. . .	. . .
0x00004000	target: add \$s1, \$s1, \$s0 # \$s1 = 1 + 4 = 5
0xF0002008	jal funcion # <b>salta (siempre) a funcion</b> # \$ra = 0xF000200C
0xF000200C	sra \$s1, \$s1, 2 # <b>se ejecuta al volver</b>
. . .	. . .
0xF0004000	funcion: add \$s1, \$s1, \$s0 # \$s1 = 1 + 4 = 5
. . .	. . .
0xF000401C	jr \$ra # <b>regresa a la rutina principal</b>

# Saltos incondicionales: codificación (j)

	Código Ensamblador	Valores de los campos (decimal)	
		op	addr
1	j target	2	4096
2	jal funcion	3	4096
	Código máquina (HEX)	Código máquina (binario)	
		op (6)	addr (26)
1	0x08001000	0000 10	00 0000 0000 0001 0000 0000 0000
2	0x0C001000	0000 11	00 0000 0000 0001 0000 0000 0000

En MIPS solo las instrucciones de salto incondicional *jump* (j) y *jump and link* (jal), utilizan el formato denominado J-Type.

Además de saltar, en jal se hace  $\$ra = PC+4$  para guardar la dirección de retorno

Dirección de salto:  $PC_{nuevo} = JTA = (PC_{antiguo} + 4) [31:28] \& addr \& "00"$   
└──────────┘  
Jump Target Address

## Saltos incondicionales: codificación (j)

# Dirección	# Ensamblador de MIPS	
0x00002000	addi \$s0, \$0, 4	# \$s0 = 4
0x00002004	addi \$s1, \$0, 1	# \$s1 = 1
0x00002008	j target	# <b>salta (siempre) a target</b>
0x0000200C	sra \$s1, \$s1, 2	# <b>nunca se ejecuta</b>
0x00002010	addi \$s1, \$s1, 1	# <b>nunca se ejecuta</b>
. . .	. . .	
0x00004000	target: add \$s1, \$s1, \$s0	# \$s1 = 1 + 4 = 5

```
JTA = { (PC+4) [31:28], addr, "00" }
```

```
0x00004000 = (0x00002008+4)[31:28], addr, "00"
```

```
0x00004000 = (0x0000200C)[31:28], addr, "00"
```

```
0x00004000 = "0000", addr, "00"
```

**0000    00000000000000000000**1**000000000000    00 = 0000    addr 00**

$$\text{addr} = 00000000000000000001000000000000 = 2^{12} = 4096$$

# Saltos incondicionales: codificación (j)

# Dirección	#Código máquina	# Ensamblador de MIPS
0x00400000	0x20100004	addi \$s0, \$0, 4
0x00400004	0x20110001	addi \$s1, \$0, 1
0x00400008	0x08100005	<b>j target</b>
0x0040000C	0x00118883	sra \$s1, \$s1, 2
0x00400010	0x22310001	addi \$s1, \$s1, 1
0x00400014	0x02308820	add \$s1, \$s1, \$s0

**target:**

¿Dónde estará escrita la etiqueta *target*?

```
j target => 0x08100005 =>
000010 0000010000000000000000000000101
  OPCODE                addr
```

```
JTA = { (PC+4)[31:28], addr, "00" }
```

```
JTA = { (0x00400008+4)[31:28], 0000010000000000000000000000101, 00 }
```

```
JTA = { (0x0040000C)[31:28], 0000010000000000000000000000101, 00 }
```

```
JTA = { 0000, 0000010000000000000000000000101, 00 }
```

```
JTA = 0000000001000000000000000000000010100 = 0x00400014
```

# Saltos incondicionales: codificación (*jr*)

	Código Ensamblador	Valores de los campos (decimal)					
		op	rs	rt	rd	shamt	funct
1	jr \$s0	0	16	0	0	0	8
	Código máquina (HEX)	Código máquina (binario)					
		op (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
1	0x02000004	0000 00	10 000	0 0000	0000 0	000 00	00 1000

En MIPS la instrucción de salto incondicional *jump register* (*jr*) utiliza el formato denominado R-Type.

El contenido de registro indicado en el campo **rs** de 5 bits, señala a una dirección absoluta de 32 bits donde se encuentra la instrucción objetivo (*target*):

$$PC = \underbrace{[rs]}_{\text{Contenido del registro rs}}$$



# Salto incondicionales: ejemplos (jr)

## #Dirección

0x00002000

0x00002004

0x00002008

0x0000200C

0x00002010

## # Ensamblador de MIPS

addi \$s0, \$0, 0x2010

jr \$s0

addi \$s1, \$0, 1

sra \$s1, \$s1, 2

lw \$s3, 44(\$s1)

Salta siempre (*branch taken*) a la dirección contenida en el registro que acompaña a la instrucción.

**¿Qué instrucción se ejecuta tras el salto?**

**lw \$s3, 44(\$s1)**

# Los modos de direccionamiento

## Modos de direccionamiento:

Son las distintas formas que un procesador utiliza para obtener un operando. Los modos usados en MIPS:

✓ **Directo en Registro.** Los operandos se encuentran en registros.

➤ Ejemplos: `add $s0, $t2, $t3` // `sub $t8, $s1, $0`

✓ **Inmediato.** El operando es un dato de 16 bits que se encuentra en la instrucción.

➤ Ejemplos: `addi $s4, $t5, -73` // `ori $t3, $t7, 0xFF`

✓ **Relativo a Registro.** La dirección efectiva del operando se obtiene sumando una constante inmediata extendida en signo, al contenido de un registro que actúa como base.

➤ Ejemplos: `lw $s4, 72($0)` Dirección =  $\$0 + 72$   
`sw $t2, -25($t1)` Dirección =  $\$t1 - 25$

# Los modos de direccionamiento

- ✓ **Relativo a PC.** La dirección efectiva del operando se obtiene sumando una constante inmediata extendida en signo, al contenido del registro Contador de Programa (PC) + 4.

➤ **Ejemplo:** modo de direccionamiento utilizado en saltos condicionales.

# Dirección	# Ensamblador de MIPS
0x10	beq \$t0, \$0, else # Salta 3 instrucciones adelante
0x14	addi \$v0, \$0, 1
0x18	addi \$sp, \$sp, i
0x1C	else: addi \$a0, \$a0, -1
0x20	jal factorial

Código Ensamblador	Valores de los campos (decimal)			
	op	rs	rt	imm
beq \$t0, \$0, else	4	8	0	2

Ver instrucciones de saltos condicionales para el funcionamiento

# Los modos de direccionamiento

- ✓ **Direccionamiento Seudo-Directo.** La dirección efectiva del operando se encuentra casi-directamente en la instrucción y señala a una dirección de memoria.
  - **Ejemplo:** modo de direccionamiento utilizado en llamadas a subprogramas (`jal`) y `jump` incondicional `j`.

```
# Dirección          # Ensamblador de MIPS
0x0040005C          jal    sum          # llama a la rutina sum
...                . . .
0x004000A0    sum:  addi $a0, $a0, -1
```

Código Ensamblador	Valores de los campos (decimal, hexadecimal, binario)	
	op	addr
jal sum	3	1048616
0x0C100028	0000 11	00 0001 0000 0000 0000 0010 1000

Ver instrucciones de saltos incondicionales para el funcionamiento

# Ejemplo de Programa

## Código en C

```
int f, g, y; // global
int main(void)
{
    f = 2;
    g = 3;
    y = sum(f, g);
    return y;
}

int sum(int a, int b) {
    return (a + b);
}
```

Tabla de símbolos	
Símbolo	Dirección
f	0x00002000
g	0x00002004
y	0x00002008
main	0x00000000
sum	0x0000002C

## // Código Ensamblador MIPS

```
.data
f: 0x00
g: 0x00
y: 0x00
.text
main:
    addi $sp, $sp, -4    # stack frame
    sw   $ra, 0($sp)     # store $ra
    addi $a0, $0, 2      # $a0 = 2
    sw   $a0, f           # f = 2
    addi $a1, $0, 3      # $a1 = 3
    sw   $a1, g           # g = 3
    jal  sum              # call sum
    sw   $v0, y           # y = sum()
    lw   $ra, 0($sp)     # restore $ra
    addi $sp, $sp, 4      # restore $sp
    jr   $ra              # return to OS

sum:
    add  $v0, $a0, $a1    # $v0 = a + b
    jr   $ra              # return
```

# Ejemplo de Programa: Ejecutable

Executable file header	Text Size	Data Size
	0x34 (52 bytes)	0xC (12 bytes)
Text segment	Address	Instruction
	0x00000000	0x23BDFFFC
	0x00000004	0xAFBF0000
	0x00000008	0x20040002
	0x0000000C	0xAC042000
	0x00000010	0x20050003
	0x00000014	0xAC052004
	0x00000018	0x0C00000B
	0x0000001C	0xAC022008
	0x00000020	0x8FBF0000
	0x00000024	0x23BD0004
	0x00000028	0x03E00008
	0x0000002C	0x00851020
	0x00000030	0x03E00008
Data segment	Address	Data
	0x00002000	0x00000000
	0x00002004	0x00000000
	0x00002008	0x00000000

```

addi $sp, $sp, 0xFFFC
sw   $ra, 0x0000($sp)
addi $a0, $0, 0x0002
sw   $a0, 0x2000($0)
addi $a1, $0, 0x0003
sw   $a1, 0x2004($0)
jal  0x0000002C
sw   $v0, 0x2008($0)
lw   $ra, 0x0000($sp)
addi $sp, $sp, 0x0004
jr   $ra
add  $v0, $a0, $a1
jr   $ra
    
```

# El mapa de memoria de MIPS

## ¿Cuál es el tamaño de la memoria?

- ✓ Si el bus de direcciones tiene 32 bits, como máximo  $2^{32} = 4$  gigabytes (4 GB)
- ✓ El mapa de direcciones se extiende entre:  
**0x00000000** hasta **0xFFFFFFFF**

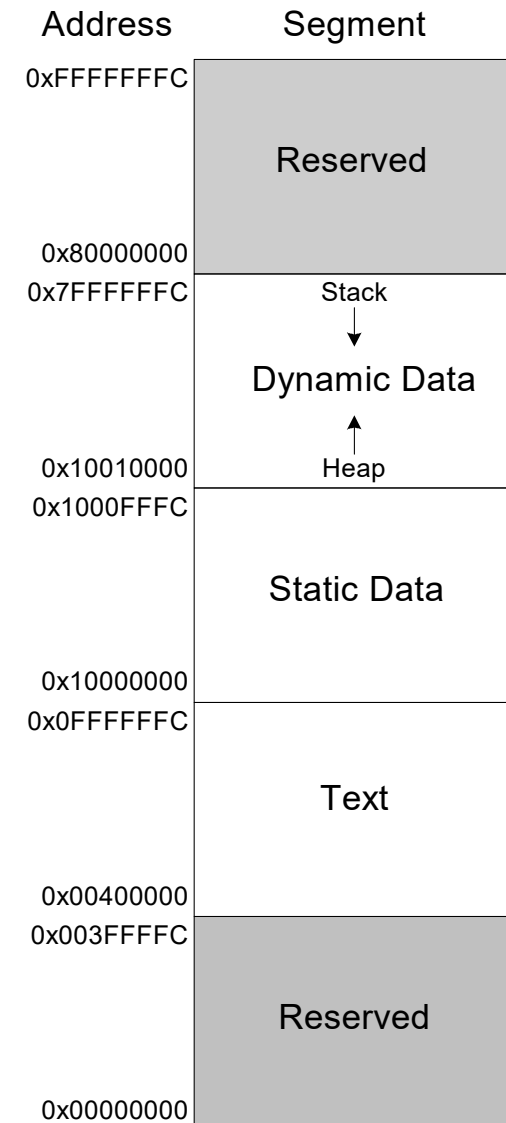
## En el simulador:

### ¿dónde se almacenan el código y los datos?

- ✓ Con las directivas **.text** y **.data**
  - Por defecto: código (0x00400000) y datos (0x10010000)
  - **Modo Texto 0: código (0x00000000) y datos (0x00002000)**
  - Modo Datos 0: código (0x00003000) y datos (0x00000000)

## Tipos de datos:

- ✓ Globales/Estáticos. Asignados antes de comenzar la ejecución
- ✓ Dinámicos. Asignados al cargar el programa



# Ejemplo de Programa: Ubicación en memoria

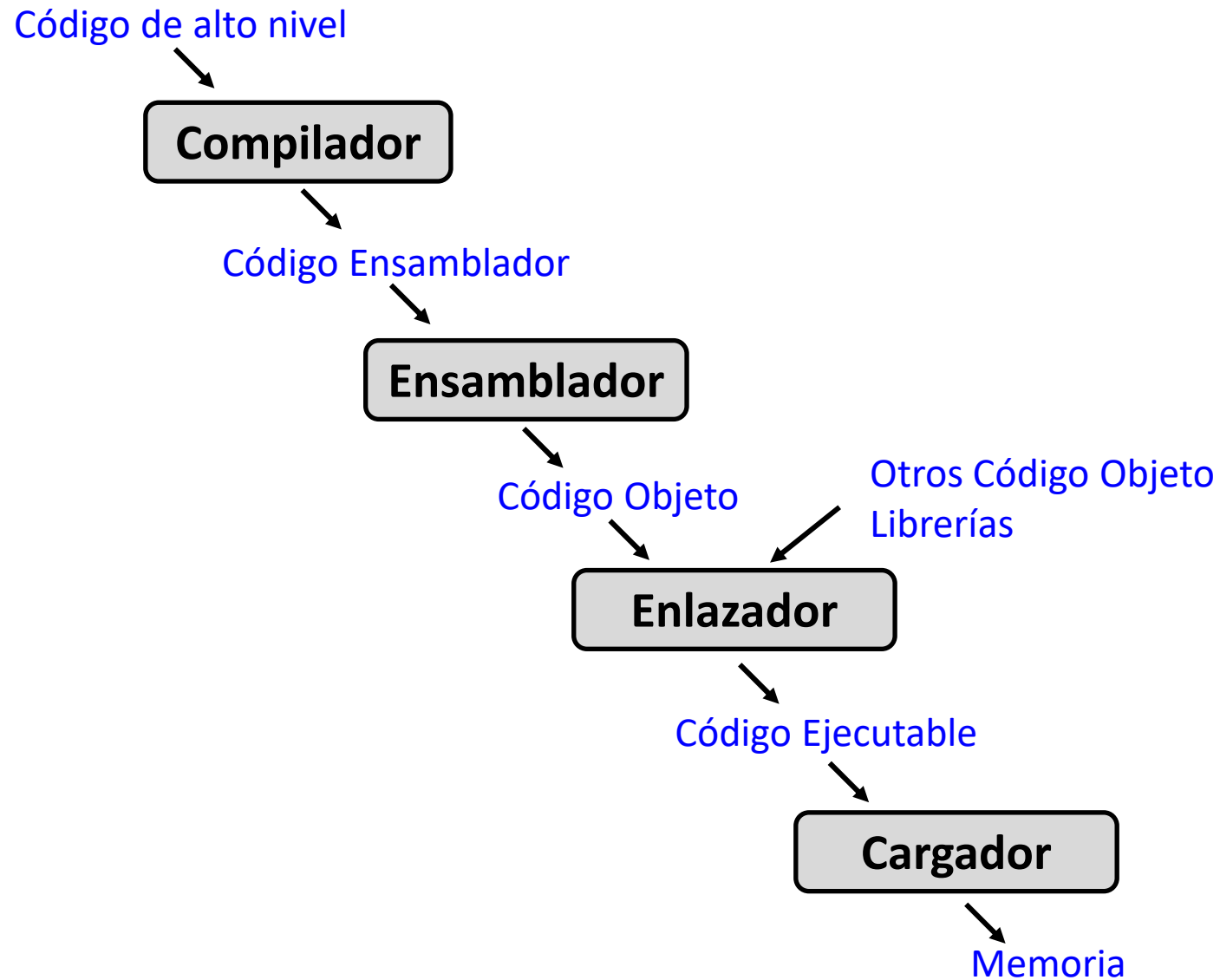
## Simulador MARS. Memoria en modo Compacto Texto 0

- Memoria de Código: (0x00000000:0x00001FFC)
- Memoria de Datos: (0x00002000:0x00003FFF)
  - Datos Dinámicos: (0x00003FFC:0x00002000)

Dirección	Memoria	
	Reservado	
0x00003FFC	Pila (stack) ↓ . . . .	← \$sp = 0x00003FFC
	0x00000000	
	0x00000000	
0x00002000	0x00000000 . . .	
	0x03E00008	
	0x00851020	
	0x03E00008	
	0x23BD0004	
	0x8FBF0000	
	0xAC022008	
	0x0C00000B	
	0xAC052004	
	0x20050003	
	0xAC042000	
	0x20040002	
	0xAFBF0000	
0x00000000	0x23BDFFFC	← PC = 0x00000000 <b>64</b>



# Compilar, enlazar y ejecutar un proceso



# Índice

- Arquitectura y Tecnología de Computadores
- El lenguaje Ensamblador
- ISA MIPS. El juego de instrucciones
- **Programación**

# Programar en MIPS

- Usando las instrucciones aritmético-lógicas vistas y con los diferentes saltos analizados, ya estamos en disposición de hacer programas en lenguaje ensamblador para MIPS.
- En MIPS se pueden generar las estructuras habituales de programación que se utilizan en lenguajes de alto nivel como C, Java, Python, etc....
- En los lenguajes de alto nivel, estas estructuras están descritas a un mayor nivel de abstracción.
- Las estructuras a revisar en MIPS son:
  - Llamadas a procedimientos, subrutinas o funciones
  - Ejecución condicional if/else
  - Bucles for
  - Bucles while
  - Trabajar con arrays de datos

# Llamadas a procedimientos

## Operaciones a realizar en la llamada a un procedimiento:

- En la rutina principal que hace la llamada (*caller*):
  - ✓ **Pasa los argumentos** a la rutina llamada (subrutina o rutina secundaria). En MIPS se utilizan los registros :  $\$a0-\$a3$  o también se puede utilizar la *pila*.
  - ✓ **Salta** a la subrutina. En MIPS se utiliza la instrucción `jal` (*jump and link*), para guardar la dirección de regreso en:  $\$ra$ .
- En la rutina llamada o subrutina (*callee*):
  - ✓ **Ejecuta el procedimiento.**
  - ✓ **Devuelve el resultado** a la rutina principal. En MIPS se utilizan los registros  $\$v0-\$v1$  o también se puede utilizar la *pila*.
  - ✓ **Retorna** a la rutina principal. En MIPS se utiliza la instrucción `jr $ra`, (*jump register*).
  - ✓ **ATENCIÓN:** la subrutina no debe sobrescribir en registros utilizados por la rutina principal.

# Llamadas a procedimientos

## Código de alto nivel

```
int main() {  
    simple();  
    a = b + c;  
}
```

```
void simple() {  
    return;  
}
```

//

## Código Ensamblador MIPS

### Dirección

```
0x00400200 main: jal simple  
0x00400204 add $s0, $s1, $s2  
...  
...  
0x00401020 simple: jr $ra
```

`void` significa que la función `simple` no retorna ningún valor

`jal simple:` salta a `simple` y guarda PC+4 (0x00400204) en el registro de enlace (`$ra = 0x00400204`)

`jr $ra:` salta a la dirección de retorno que se encuentra en `$ra`

# Argumentos y variables de retorno

## Código de alto nivel

```
int main()
{
    int y;
    ...
    y = diffofsums(2, 3, 4, 5);    //4 argumentos de entrada
                                   // 2=f, 3=g, 4=h, 5=i)
    ...
}

int diffofsums(int f, int g, int h, int i)
{
    int result;
    result = (f + g) - (h + i);
    return result;                // valor retorno (y)
}
```

# Argumentos y variables de retorno

## Código Ensamblador MIPS

```
# $s0 = y
main:
    ...
    addi $a0, $0, 2      # argument0 0 = 2
    addi $a1, $0, 3      # argument0 1 = 3
    addi $a2, $0, 4      # argument0 2 = 4
    addi $a3, $0, 5      # argument0 3 = 5
    jal  diffofsums      # llama a la subrutina diffofsums. $ra guarda la dir. de retorno
    add  $s0, $v0, $0     # y = valor de retorno
    ...
# $s0 = result
diffofsums:
    add $t0, $a0, $a1     # $t0 = f + g
    add $t1, $a2, $a3     # $t1 = h + i
    sub $s0, $t0, $t1     # result = (f + g) - (h + i)
                          # la subrutina diffofsums modifica: $t0, $t1 y $s0
    add $v0, $s0, $0      # escribe el valor de retorno en $v0
    jr  $ra              # retorna la rutina principal
```

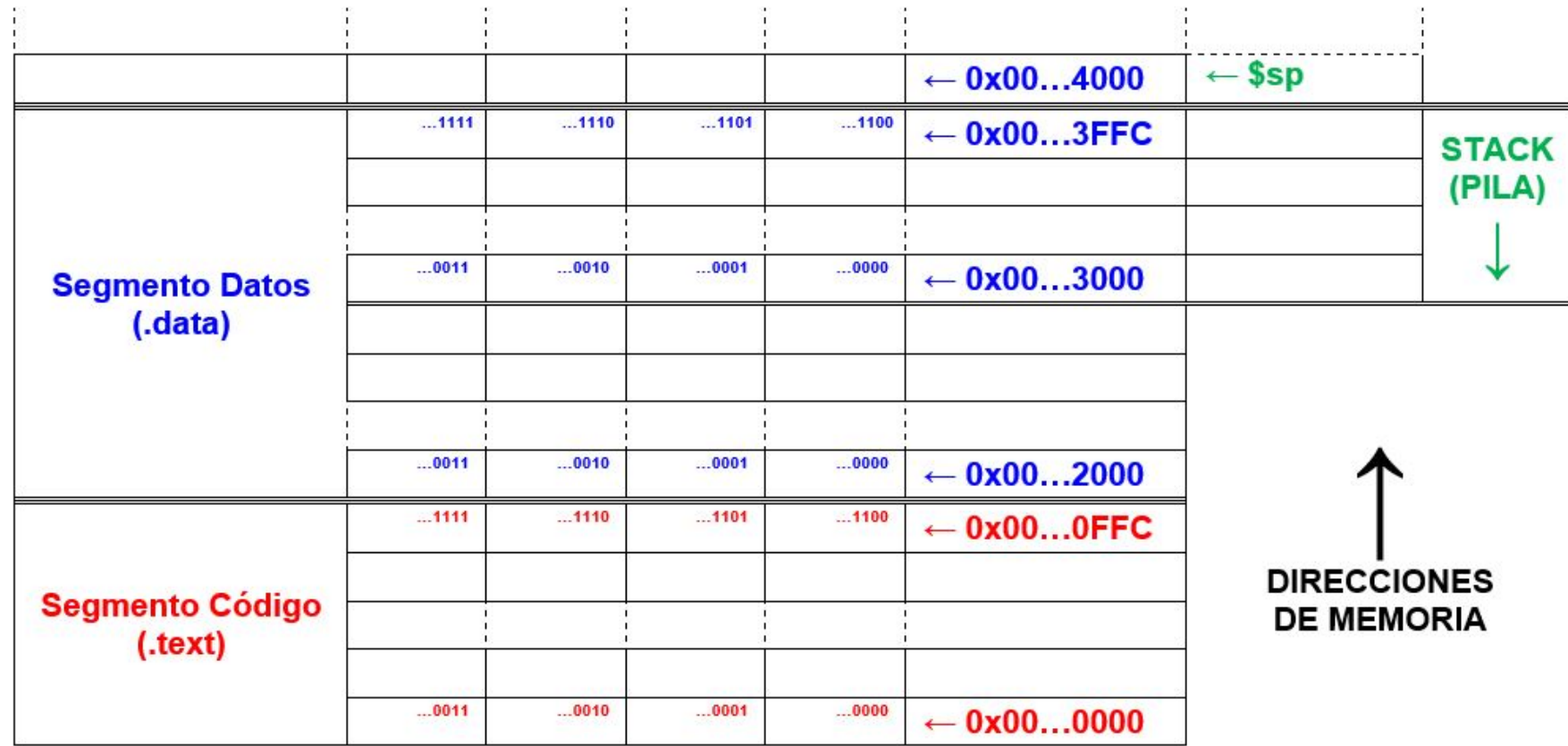
# La pila (*stack*) del sistema

- Zona de la memoria que se usa temporalmente para almacenar variables.
- Es memoria RAM pero se utiliza como una memoria LIFO (*last-in-first-out*), utilizando un registro específico como puntero, `$sp` (*stack pointer*).
- *La pila se expande*: utiliza más memoria (bytes) según necesita.
- *La pila se contrae*: reduce la memoria cuando deja de necesitarse.
- La pila tiene múltiples funciones tales como paso de parámetros entre funciones, almacenar copias de registros, almacenar variables locales, etc.



# La pila (*stack*) del Sistema en MIPS

## (MARS modo Compact text Address 0)



# La pila (*stack*) del sistema

- La pila crece desde direcciones de memoria altas hacia direcciones más bajas.
- El registro puntero de pila (*stack pointer*)  $\$sp$ , guarda la dirección de la última posición de memoria ocupada de la pila.
- Al inicio del programa el puntero  $\$sp$  indica la última posición ocupada por el anterior programa (sistema operativo si lo hubiera, o el inicio de la pila si no hay SSOO).

Address	Data
00003FFC	
00003FF8	
00003FF4	
00003FF0	
⋮	⋮

←  $\$sp$

# La pila (*stack*) del sistema

- La pila crece desde direcciones de memoria altas hacia direcciones más bajas.
- El registro puntero de pila (*stack pointer*)  $\$sp$ , guarda la dirección de la última posición de memoria ocupada de la pila.

Address	Data
00003FFC	12345678 ← $\$sp$
00003FF8	
00003FF4	
00003FF0	
⋮	⋮

Address	Data
00003FFC	12345678
00003FF8	AABBCCDD
00003FF4	11223344 ← $\$sp$
00003FF0	
⋮	⋮

# La pila (*stack*) del sistema

Después de cada operación (envío y posterior recogida de datos) con la pila, el puntero de pila  $\$sp$  debe recuperar el valor de inicio.

Address	Data	
00003FFC	12345678	← \$sp
00003FF8	AABBCCDD	
00003FF4	11223344	
00003FF0		
⋮	⋮	

# Los registros y la pila del sistema

Un procedimiento no puede usar cualquier registro sin haberlo preservado anteriormente.

En MIPS existe un acuerdo que señala qué registros deben preservarse.

<b>Preservados</b> <i>En la subrutina</i>	<b>No Preservados</b> <i>En la subrutina</i>
\$s0 - \$s7	\$t0 - \$t9
\$ra	\$a0 - \$a3
\$sp	\$v0 - \$v1
La pila por encima de \$sp	La pila por debajo de \$sp

La rutina debe preservarlo.

El llamador debe preservarlo si lo desea.

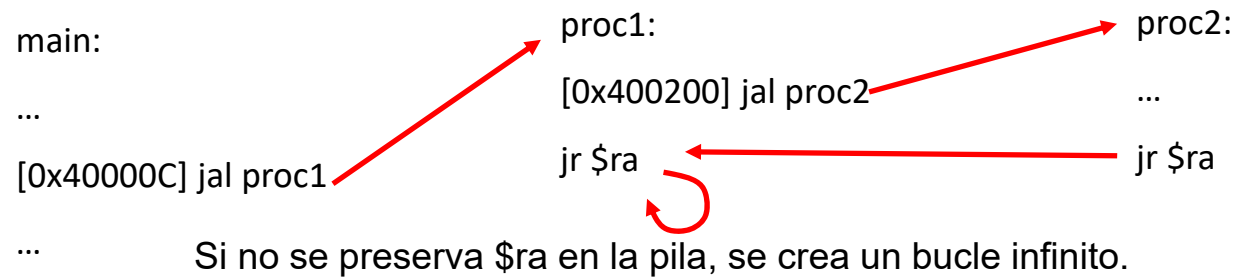
# Los registros y la pila del sistema

La pila sirve para preservar los contenidos de los registros usados en el procedimiento

```
# $s0 = result
diffofsums:
    addi $sp, $sp, -4    # reserva espacio en la pila para guardar el registro
                        # $s0. Los registros $t0 y $t1 no se necesitan preservar

    sw    $s0, 0($sp)    # guarda $s0 en la pila
    add   $t0, $a0, $a1   # $t0 = f + g
    add   $t1, $a2, $a3   # $t1 = h + i
    sub   $s0, $t0, $t1   # result = (f + g) - (h + i)
    add   $v0, $s0, $0     # escribe el valor de retorno en $v0
    lw    $s0, 0($sp)    # recupera $s0 desde la pila
    addi  $sp, $sp, 4     # libera el espacio de pila usado
    jr    $ra            # regresa a la rutina principal
```

# Llamadas anidadas



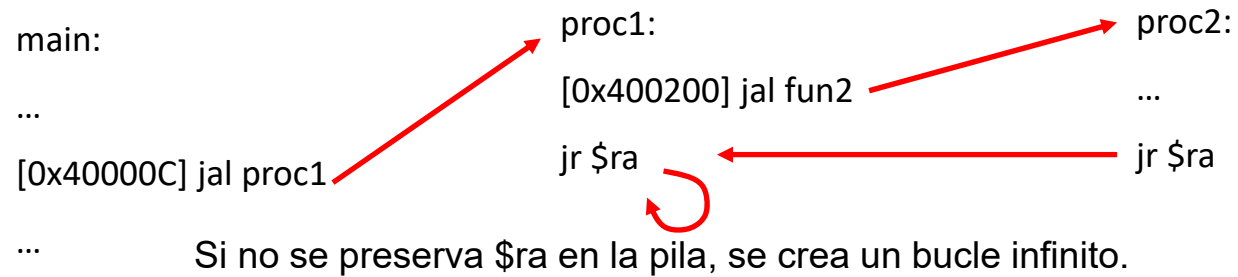
# Llamadas anidadas

En llamadas a procedimientos múltiples (llamadas anidadas), cada subrutina debe preservar en la pila su dirección de retorno `$ra`, antes de saltar.

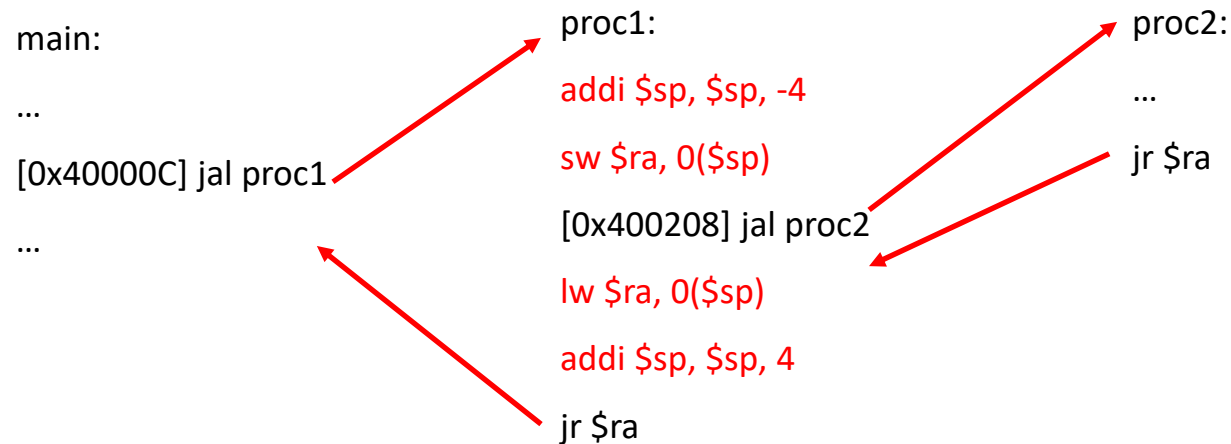
```
proc1:                                # comienza la subrutina proc1
    addi $sp, $sp, -4                 # reserva espacio en la pila
    sw   $ra, 0($sp)                 # guarda $ra en la pila
    jal  proc2                       # salta a una nueva subrutina
    ...
    ...
    lw   $ra, 0($sp)                 # regresa de proc2 y restaura $ra desde la pila
    addi $sp, $sp, 4                 # libera espacio en la pila
    jr   $ra                         # regresa a la rutina principal
```



# Llamadas anidadas



## Solución:



# Uso de la pila


Ya se ha visto que la pila:

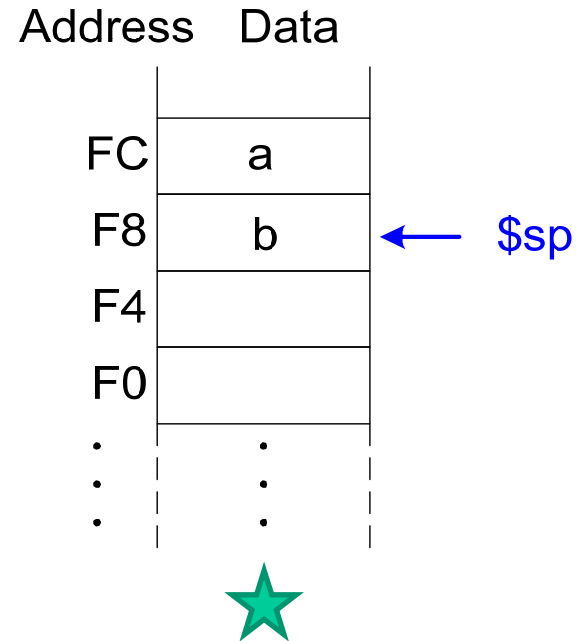
- Sirve para preservar los registros dentro de una función.
- Sirve para preservar el registro de enlace ( 'w) para realizar llamadas anidadas a funciones.

También sirve para:

- Guardar variables locales de una función.
- Pasar argumentos a funciones.
- Retorno de funciones.

# Pila: Variables locales.

```
int main() {  
  int a, b;   
  ...  
}
```



a y b se guardan en la pila como variable temporal. Pueden ser leídas sin cambiar el puntero a pila, es decir, se puede leer a sin hacer *pop* en la pila: *lw <reg>, \$sp(4)*.

# Pila: Argumentos de función.

```
int main() {
```

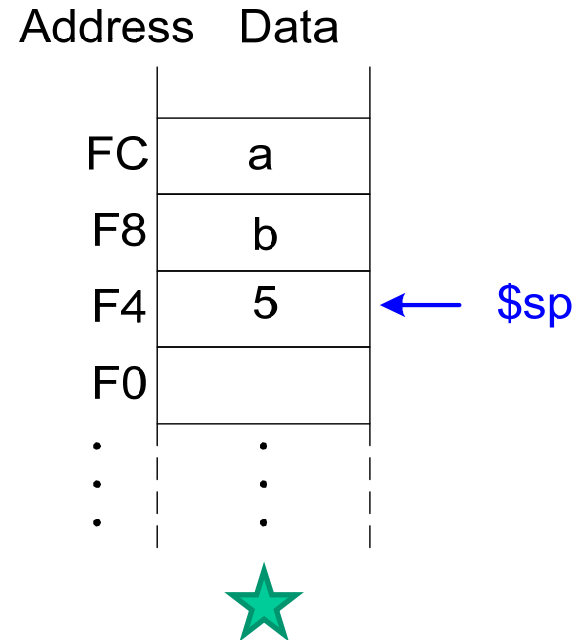
```
  int a, b;
```

```
  ...
```

```
  funct(5);
```

```
  ...
```

```
}
```



El argumento se introduce en pila sin borrar *a* y *b*, ya que quizá sean útiles después.

# Pila: Argumentos de función.

```
int main() {
```

```
  int a, b;
```

```
  ...
```

```
  funct(5);
```

```
  ...
```

```
}
```

Address	Data
FC	a
F8	b
F4	5
F0	
⋮	⋮
⋮	⋮
⋮	⋮

Address	Data
FC	a
F8	b
F4	5
F0	c
⋮	⋮
⋮	⋮
⋮	⋮

Address	Data
FC	a
F8	b
F4	10
F0	c
⋮	⋮
⋮	⋮
⋮	⋮

```
int funct(int x)
```

```
{
```

```
  int c;
```

```
  return x+5;
```

```
}
```



La función *funct* puede devolver el resultado por pila, y sobrescribe el parámetro que recibió, además de desechar la variable local (temporal) que utilizó.

# Ejecución condicional If

## Código de alto nivel

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

//

## Código Ensamblador MIPS

```
# $s0=f, $s1=g, $s2=h
# $s3 = i, $s4 = j

    bne $s3, $s4, L1
    add $s0, $s1, $s2

L1: sub $s0, $s0, $s3
```

**Observar** que en Ensamblador la condición se prueba para el caso opuesto ( $i \neq j$ ) que en el código de alto nivel ( $i == j$ ).

# Ejecución condicional If / Else

## Código de alto nivel

```
if (i == j)
    f = g + h;
else
    f = f - i;
```

//

## Código Ensamblador MIPS

```
# $s0=f, $s1=g, $s2=h
# $s3 = i, $s4 = j

    bne $s3, $s4, L1
    add $s0, $s1, $s2
    j    done
L1:   sub $s0, $s0, $s3
done:
```

**Observar** que en Ensamblador la condición se prueba para el caso opuesto ( $i \neq j$ ) que en el código de alto nivel ( $i == j$ ).

# Bucle While

## Código de alto nivel

//

```
// determina la potencia
// de x que cumple  $2^x = 128$ 
int pow = 1;
int x = 0;

while (pow != 128)
{
    pow = pow * 2;
    x = x + 1;
}
```

## Código Ensamblador MIPS

```
# $s0 = pow, $s1 = X

addi $s0, $0, 1
add  $s1, $0, $0
addi $t0, $0, 128
while: beq  $s0, $t0, done
sll  $s0, $s0, 1
addi $s1, $s1, 1
j    while
done:
```

**Observar** que en Ensamblador la condición se prueba para el caso opuesto ( $\text{pow} = 128$ ) que en el código de alto nivel ( $\text{pow} \neq 128$ ).



# Bucle For

La estructura general de un bucle for es:

```
for (inicialización; condición; operación del bucle)  
    Cuerpo del bucle
```

- **Inicialización:** se ejecuta antes de iniciar el bucle
- **Condición:** se comprueba al principio de cada iteración
- **Operación del bucle:** se ejecuta al final de cada iteración
- **Cuerpo del bucle:** se ejecuta cada vez que se cumple la condición

En ensamblador, esta estructura se codifica en cinco pasos:

1. Inicialización.
2. Condición.
3. Cuerpo del bucle.
4. Operación del bucle.
5. Salto al paso 2.

# Bucle For

## Código de alto nivel

```
// suma los números de
// 0 hasta 9
int sum = 0 ;
int i ;

for (i=0; i!=10; i = i+1)
{
    sum = sum + i;
}
```

//

## Código Ensamblador MIPS

```
# $s0 = i, $s1 = sum

        addi $s1, $0, 0
1       add  $s0, $0, $0
        addi $t0, $0, 10
2 for:   beq  $s0, $t0, done
        add  $s1, $s1, $s0
3       addi $s0, $s0, 1
4       j    for
5
done:
```

**Observar** que en Ensamblador la condición se prueba para el caso opuesto ( $i == 10$ ) que en el código de alto nivel ( $i != 10$ ).

# Comparación Menor Que (*Less Than*)

## Código de alto nivel

//

```
// suma las potencias de 2
// de 1 a 100
int sum = 0 ;
int i ;

for (i=1; i<101; i = i*2)
{
    sum = sum + i;
}
```

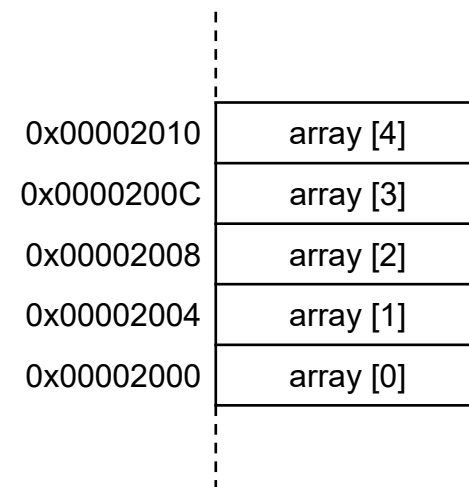
## Código Ensamblador MIPS

```
# $s0 = i, $s1 = sum
        addi $s1, $0, 0
        addi $s0, $0, 1
        addi $t0, $0, 101
loop:    slt  $t1, $s0, $t0
        beq  $t1, $0, done
        add  $s1, $s1, $s0
        sll  $s0, $s0, 1
        j    loop
done:
```

$\$t1 = 1$  if  $i < 101$

# Trabajar con Arrays de datos

- ✓ Un array se utiliza para acceder a un gran número de datos de características similares.
- ✓ El tamaño del array viene dado por el número de elementos
- ✓ Un elemento del array se caracteriza y diferencia de otro por su índice
  - Sea por ejemplo un array de 5 elementos de 32 bits
  - Dirección base (*base address*): 0x00002000, es la dirección de memoria del primer elemento array[0]
  - Las instrucciones lw y sw acceden a la posición de memoria resultante de sumar un dato inmediato y el contenido de un registro.
  - Si la dirección base cabe en 16 bits, ésta se puede usar como dato inmediato.
  - El registro actuaría como el índice dentro del array (0, 1, 2,...).



# Trabajar con Arrays de datos

## Código de alto nivel

```
int array [5];  
array[0] = array[0] * 2;  
array[1] = array[1] * 2;  
:
```

Suponiendo que array empieza en la posición  
0x00002000 (**DIRECCIÓN BASE DE 16b**).  
Cada posición del array ocupa 4 bytes

## Código Ensamblador de MIPS

```
# índice = $s0, la base fija en el dato inmediato.  
addi $s0, $0, 0          # $s0 = índice 0  
lw   $t1, array($s0)     # $t1 = array[0]  
sll  $t1, $t1, 1         # $t1 = $t1 * 2  
sw   $t1, array($s0)     # array[0] = $t1  
addi $s0, $0, 4          # $s0 = índice 1 (byte +4)  
lw   $t1, array($s0)     # $t1 = array[1]  
sll  $t1, $t1, 1         # $t1 = $t1 * 2  
sw   $t1, array($s0)     # array[1] = $t1  
:
```

# Trabajar con Arrays de datos

## Código de alto nivel

```
int array [5];  
array[0] = array[0] * 2;  
array[1] = array[1] * 2;  
:
```

Suponiendo que array empieza en la posición 0x12348000 (**DIRECCIÓN BASE DE 32b**). La dirección base no cabe en el dato inmediato.

Cada posición del array ocupa 4 bytes

## Código Ensamblador de MIPS

```
# dirección base fija = $s0, el índice en el dato inmediato  
lui    $s0, 0x1234           # 0x1234 en los 16 msb de $s0  
ori    $s0, $s0, 0x8000      # 0x8000 en los 16 lsb de $s0  
lw     $t1, 0($s0)           # $t1 = array[0]  
sll    $t1, $t1, 1           # $t1 = $t1 * 2  
sw     $t1, 0($s0)           # array[0] = $t1  
lw     $t1, 4($s0)           # $t1 = array[1]  
sll    $t1, $t1, 1           # $t1 = $t1 * 2  
sw     $t1, 4($s0)           # array[1] = $t1  
:
```

# Trabajar con Arrays de datos

## Código de alto nivel (usando bucles)

```
int array [1000];  
int i ;  
for (i=0; i < 1000; i = i + 1)  
    array[i] = array[i] * 8;
```

El bucle tiene que ejecutarse 1000 veces porque el array tiene 1000 elementos.

Si cada elemento del array ocupa 4 bytes, en MIPS, la dirección en el registro equivalente a *i* para la lectura y escritura de memoria tiene que incrementarse de 4 en 4, por lo que la condición de permanencia en el bucle debe ser modificada.

# Trabajar con Arrays de datos

## # Código Ensamblador de MIPS (usando bucles)

Suponiendo que array empieza en la posición 0x00002000.

```
# $s1 = i
# Código de inicialización
    addi $s1, $0, 0      # i = 0;
    addi $t2, $0, 4000   # $t2 = 4000.
                        # (Si i se incrementa en 4 en
                        # cada iteración, el valor
                        # 1000 debe multiplicarse por 4)

loop:
    slt  $t0, $s1, $t2   # i < 4000?
    beq  $t0, $0, done   # if not then done
    lw   $t1, array($s1) # $t1 = array[i]
    sll  $t1, $t1, 3      # $t1 = array[i] * 8
    sw   $t1, array($s1) # array[i] = array[i] * 8
    addi $s1, $s1, 4      # i = i + 4
    j    loop            # repeat
done:
```



## ***Unidad 3. El procesador I: Diseño del juego de Instrucciones. El lenguaje máquina***

---

Escuela Politécnica Superior - UAM