



# **Arquitectura de Computadores**

---

**Introducción básica al  
Lenguaje Máquina (Ensamblador)**

# Introducción

- Subamos algunos niveles de abstracción
- **Arquitectura:** La vista del programador del computador
  - Esta definida por la ubicación de los operandos & instrucciones
- **Micro-arquitectura:** como implementar una arquitectura en hardware.

|                      |                           |
|----------------------|---------------------------|
| Application Software | programs                  |
| Operating Systems    | device drivers            |
| Architecture         | instructions<br>registers |
| Micro-architecture   | datapaths<br>controllers  |
| Logic                | adders<br>memories        |
| Digital Circuits     | AND gates<br>NOT gates    |
| Analog Circuits      | amplifiers<br>filters     |
| Devices              | transistors<br>diodes     |
| Physics              | electrons                 |

# Lenguaje Ensamblador

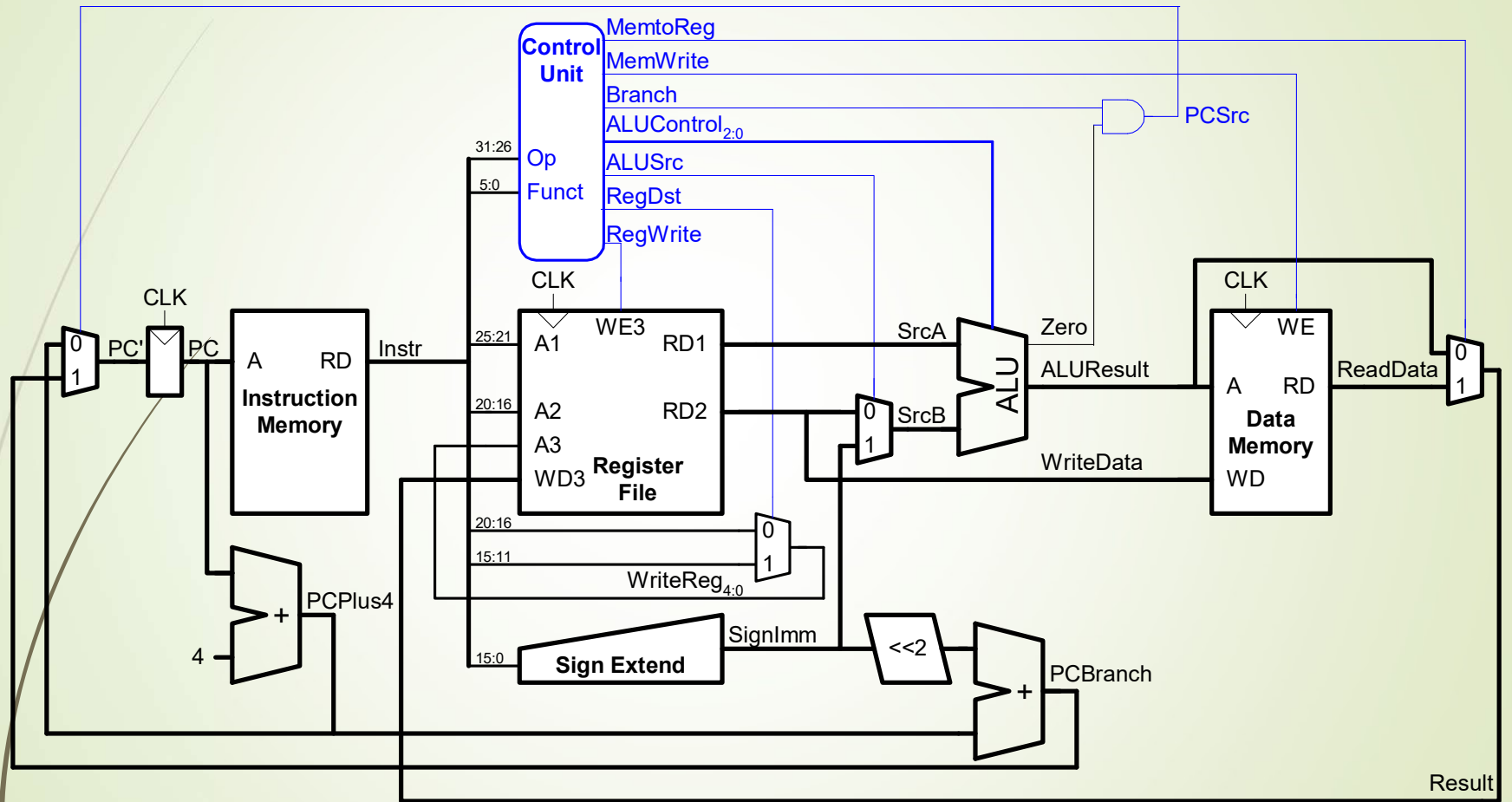
- **Instrucciones:** comandos de un lenguaje de un computador
  - **Lenguaje Ensamblador (Assembly):** instrucciones definidas en un formato comprensible por un ser humano
  - **Lenguaje de maquina:** formato comprensible para un computador (1's y 0's)
- **Arquitectura MIPS:**
  - Desarrollado por John Hennessy y sus colegas en Stanford en los 1980's.
  - Ha sido usado en sistemas comerciales, que incluyen a Silicon Graphics, Nintendo, y Cisco

# John Hennessy


- Rector de Stanford University
- Profesor de Ingeniería Eléctrica y Ciencias de la Computación en Stanford desde 1977
- Có-invento el set reducido de instrucciones para un computador (RISC) con David Patterson
- Desarrollo la arquitectura MIPS en Stanford en 1984 y cofundó la empresa MIPS Computer Systems
- Al año 2004, mas de 300 millones de microprocesadores MIPS han sido vendidos



# Procesador Mono-ciclo MIPS



# Principios de Diseño de una Arquitectura



Los principios de diseño definidos por Hennessy y Patterson son:

1. La simplicidad favorece la regularidad
2. Haz que el caso común opere con rapidez
3. Mientras mas pequeño es mas rápido
4. Un buen diseño demanda buenos compromisos

# Ejemplo de Instrucciones: Suma

## Código C

```
a = b + c;
```

## Código Ensamblador MIPS

```
add a, b, c
```

- **add:** indica la operación SUMA a realizar
- **b, c:** operandos de entrada (a los cuales se realizara la operación)
- **a:** operando destino (que almacenará el resultado)

# Instrucciones: Resta

- Similar a la suma – solo cambio mnemotécnico

## Código C

```
a = b - c;
```

## Código Ensamblador MIPS

```
sub a, b, c
```

- **sub:** indica la operación RESTA a realizar
- **b, c:** operandos de origen
- **a:** operando destino



# Principio de Diseño 1



## La simplicidad favorece la regularidad

- Formato de instrucción consistente
- El mismo numero de operandos (dos de origen y uno de destino)
- Facilita la codificación y el manejo en el hardware

# Múltiples Instrucciones

- Un código de varias operaciones es manejado por múltiples instrucciones MIPS. Ejemplo:

## Código C

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

## Código Ensamblador MIPS

```
add t, b, c    # t = b + c  
sub a, t, d    # a = t - d
```

# Principio de Diseño 2

Haz que el caso común opere con rapidez

- MIPS incluye solo instrucciones simples y que son comúnmente usadas
- El hardware para decodificar y ejecutar las instrucciones debe ser simple, pequeño y rápido
- Las instrucciones mas complejas (que son las menos comunes) son realizadas con múltiples instrucciones simples
- MIPS es un *reduced instruction set computer (RISC)*, con un pequeño numero de instrucciones simples
- Otras arquitecturas, tales como x86 de Intel, son *complex instruction set computers (CISC)*

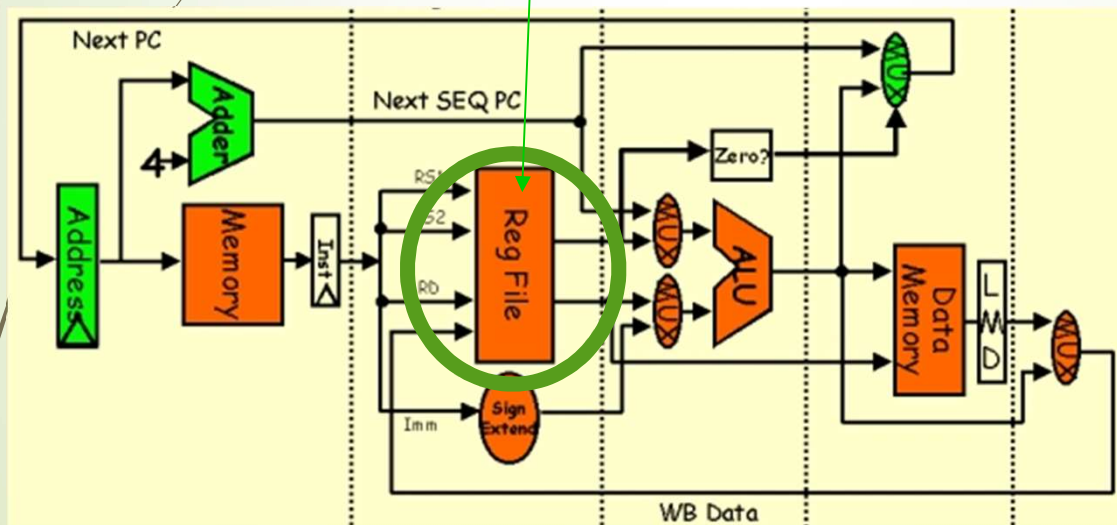
# Operandos



- La ubicación de un operando: ubicación física en el computador
  - Registros
  - Memoria
  - Constantes

# Operandos: Registros

- MIPS tiene 32 registros de 32-bit
- Los registros son mas rápidos que la memoria
- Se dice que MIPS es una “arquitectura de 32-bit” porque opera con datos de 32 bits



Microprocesador monociclo MIPS



# Principio de Diseño 3

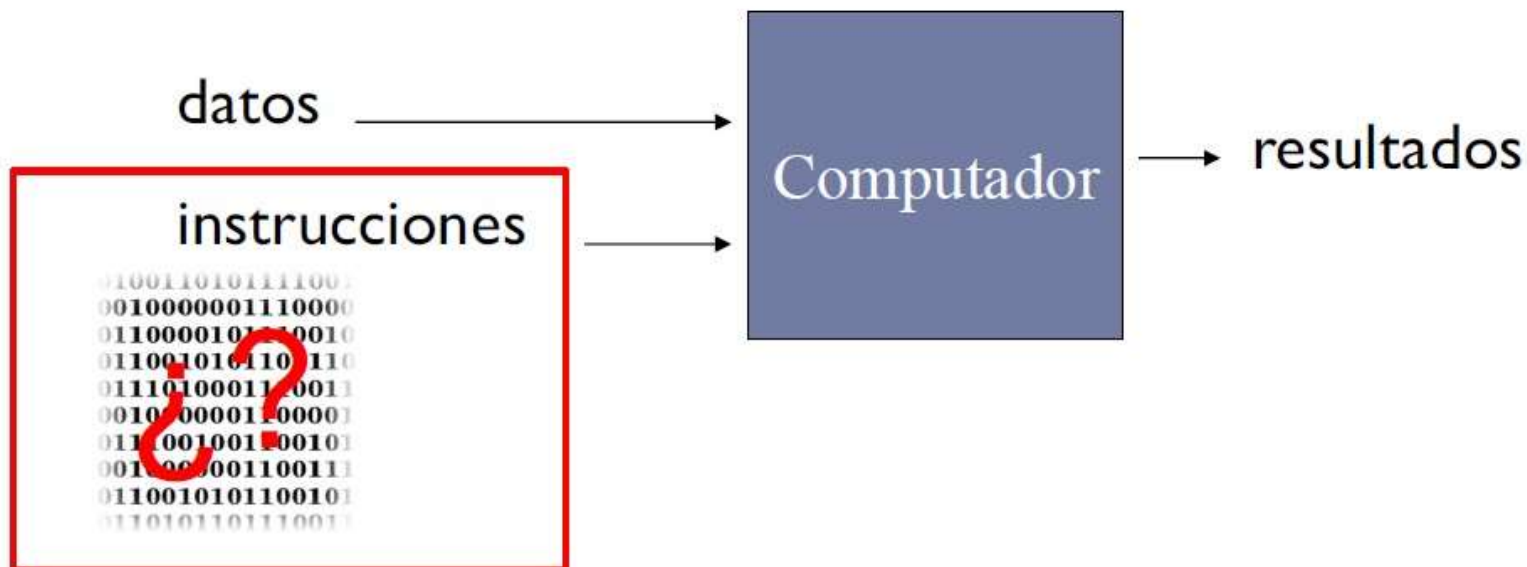


**Mientras mas pequeño es mas rápido**

- MIPS incluye solo un pequeño numero de registros

# Tipos de información: instrucciones y datos

- ▶ ¿Qué sucede con las instrucciones?



# Modelo de programación de un computador

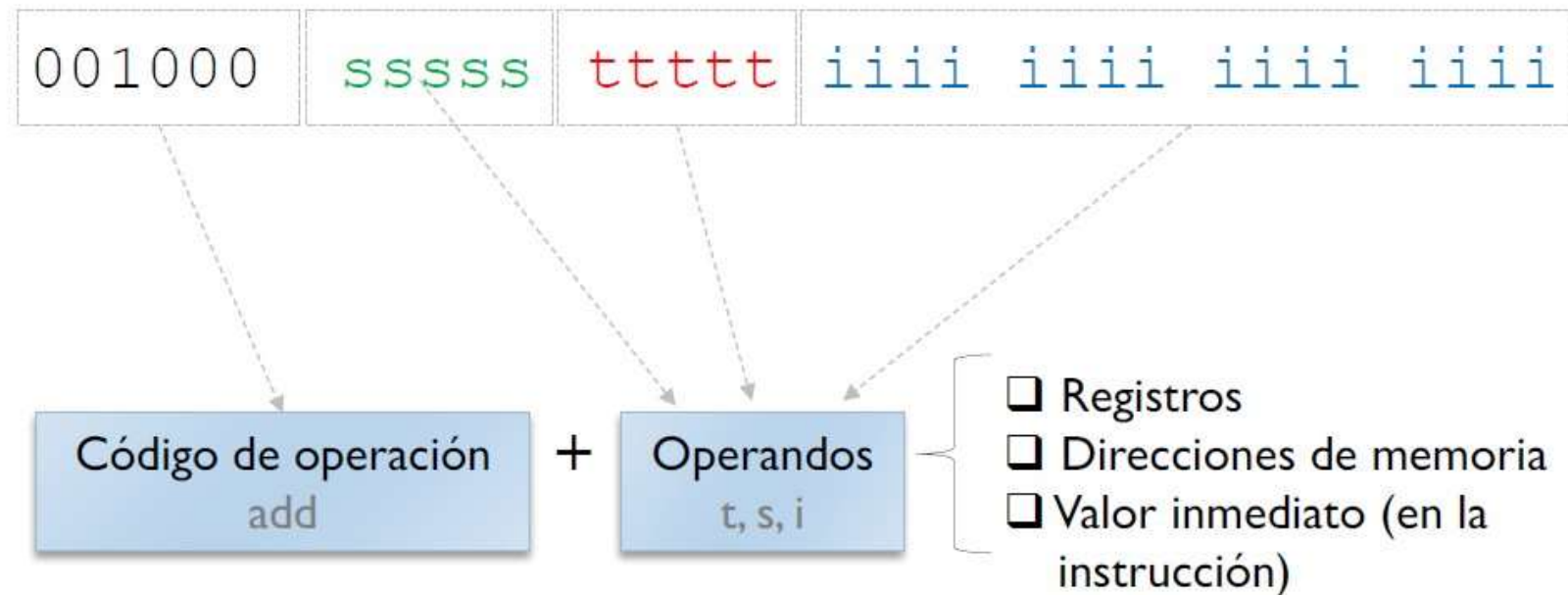
- ▶ Un computador ofrece un modelo de programación formando por:
  - ▶ **Juego de instrucciones (lenguaje ensamblador)**
    - ▶ Una instrucción incluye:
      - Código de operación
      - Otros elementos: identificadores de registros, direcciones de memoria o números
  - ▶ **Elementos de almacenamiento**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



# Instrucción máquina

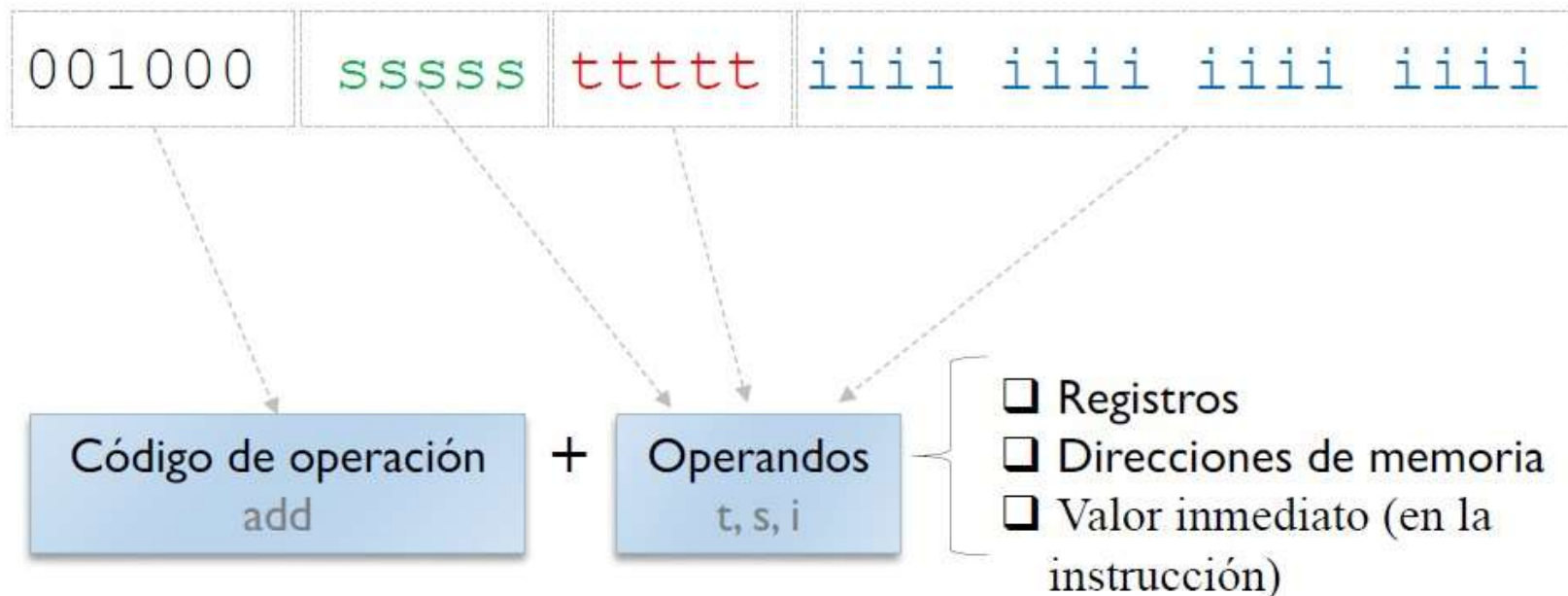
## ► Ejemplo de instrucción en MIPS:

- Suma de un registro (s) con un valor inmediato (i) y el resultado de la suma se almacena en registro (t)



# Propiedades de las instrucciones máquina

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



# Información incluida en instrucción máquina

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



# Juego de instrucciones

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

# Características de un juego de instrucciones

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

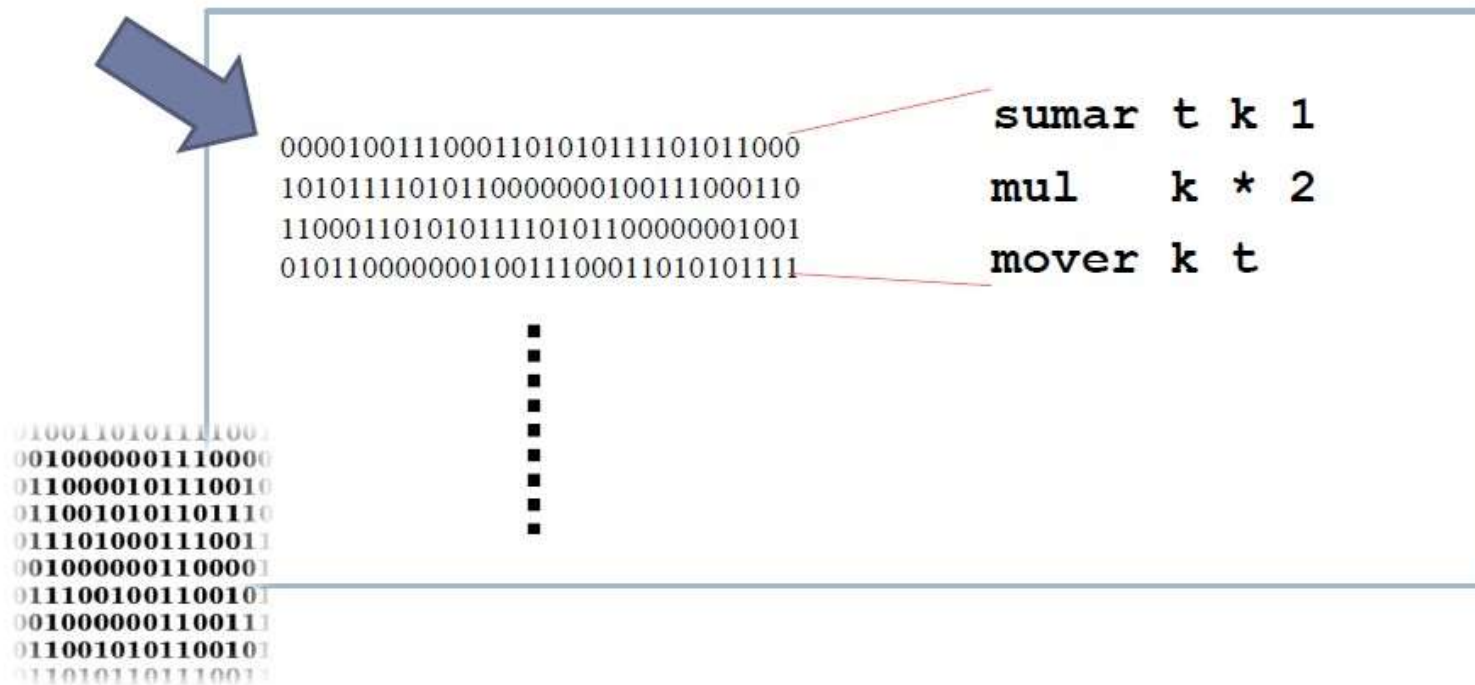


# Características de un juego de instrucciones

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

# Definición de programa

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



# Fases de ejecución de una instrucción

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

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

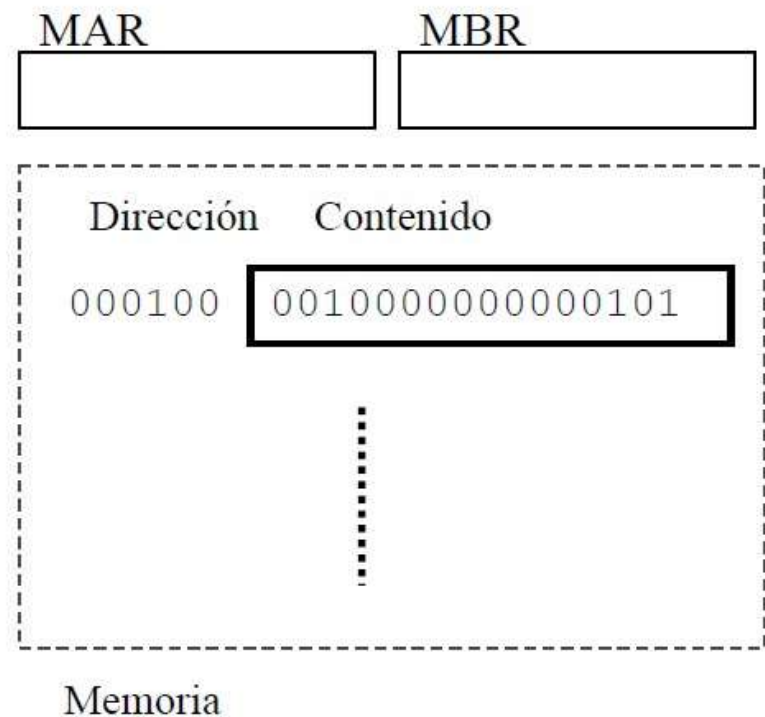
## ▶ Decodificación de la instrucción

## ▶ Ejecución de la instrucción

## ▶ Volver a *fetch*

PC 000100

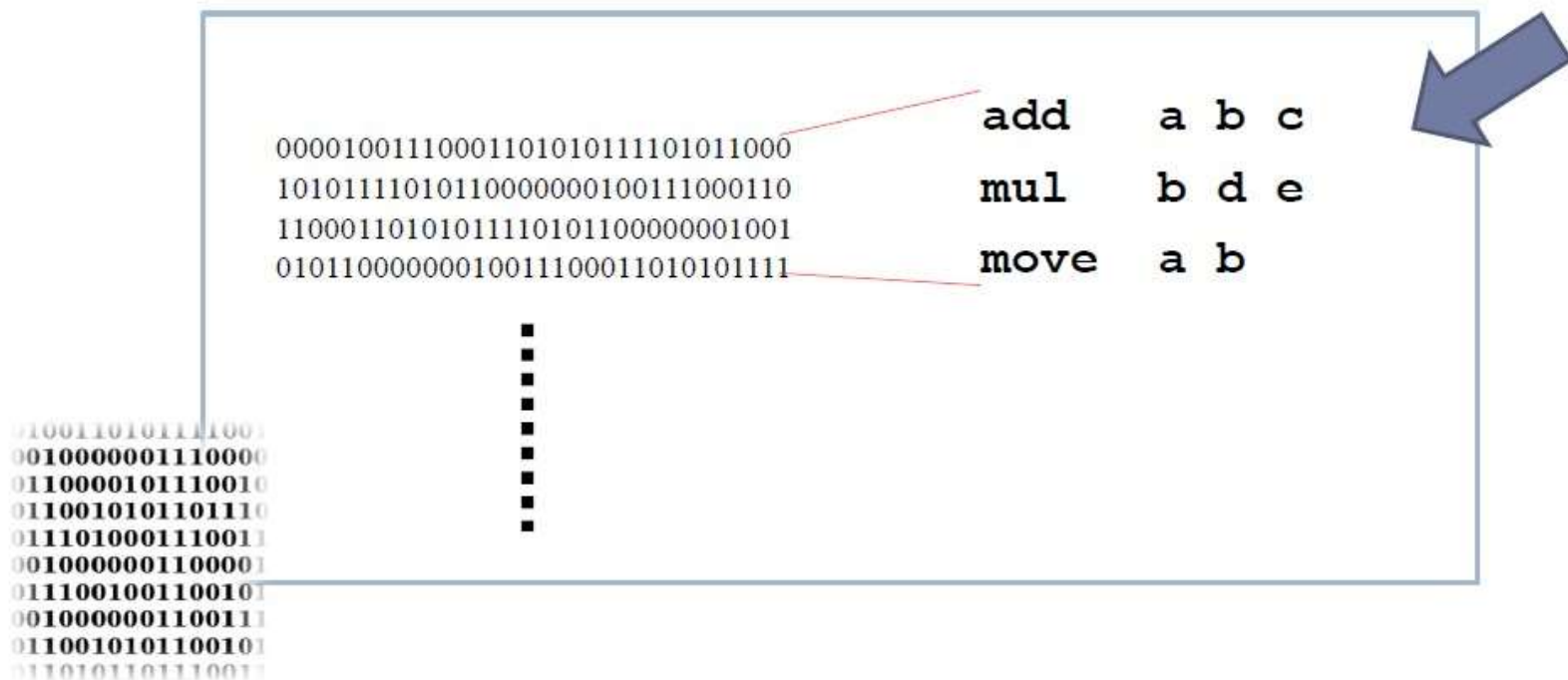
RI 0010000000000101





# Definición de lenguaje ensamblador

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



# Definición de lenguaje ensamblador

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

# Diferentes niveles de lenguajes

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

*Compilador*

Lenguaje ensamblador  
(Ej: MIPS)

*Ensamblador*

Lenguaje Máquina  
(MIPS)

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

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

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

# Proceso de compilación

Lenguaje de alto nivel

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 $v0 1
    syscall
```



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



# Compilación: ejemplo

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

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

`hola.c`

```
#include <stdio.h>  
#define PI 3.1416  
#define RADIO 20  
  
int main ( )  
{  
    int i;  
  
    i=2*PI*RADIO;  
    return (0);  
}
```

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

```
MZ      yy      @  
€  º  Í!,LÍ!This program cannot be run in DOS  
mode.
```

```
$      PE  L ,UŽI  ù  à  
8  
@  ^ .text  º  
@  @.bss  @  0
```

`hola`

```
11001101011100  
01000000111000  
11000010111001  
11001010110111  
11101000111001  
01000000110000  
11100100110010  
01000000110011  
11001010110010  
11010110111001
```



# Compilación: ejemplo

- Desensamblar **hola**:
- `objdump -d hola`



hola.exe: formato del fichero pei-i386

Desensamblado de la sección .text:

00401000 <\_WinMainCRTStartup>:

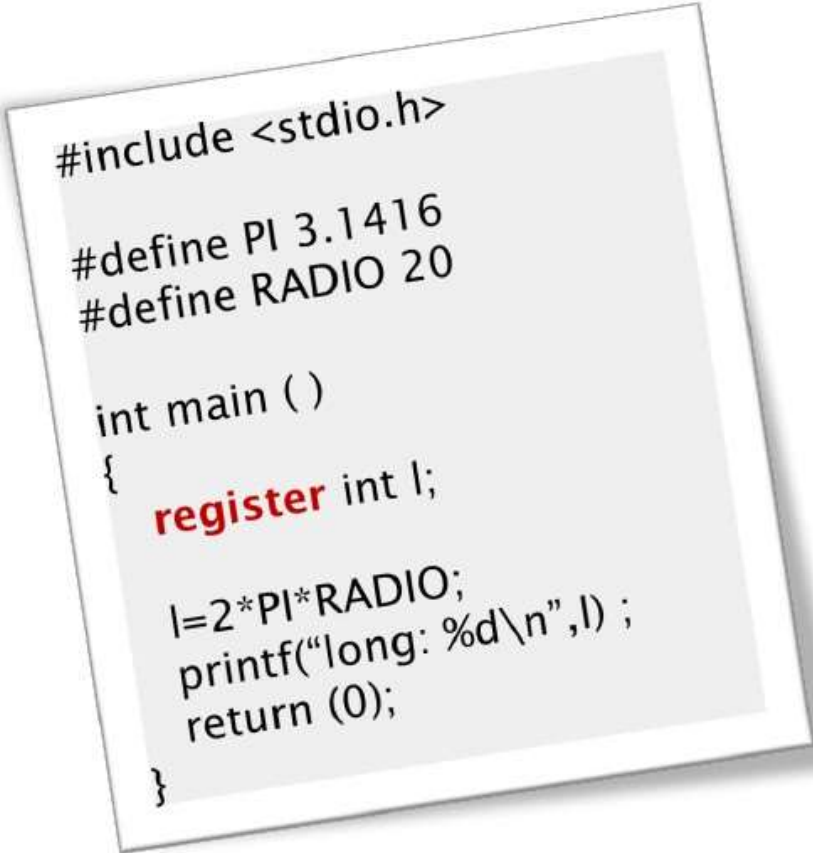
```
401000: 55          push    %ebp
...
40103f: c9          leave   %ebp
401040: c3          ret
```

00401050 <\_main>:

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

```
.data
PI: word 3.14156
RADIO: word 20
.text
li $a0 2
la $t0 PI
lw $t0 ($t0)
la $t1 RADIO
lw $t1 ($t1)
li $v0 1
syscall
```

# Motivación para aprender ensamblador



```
#include <stdio.h>
#define PI 3.1416
#define RADIO 20

int main ()
{
    register int l;

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

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

# Objetivos

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

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

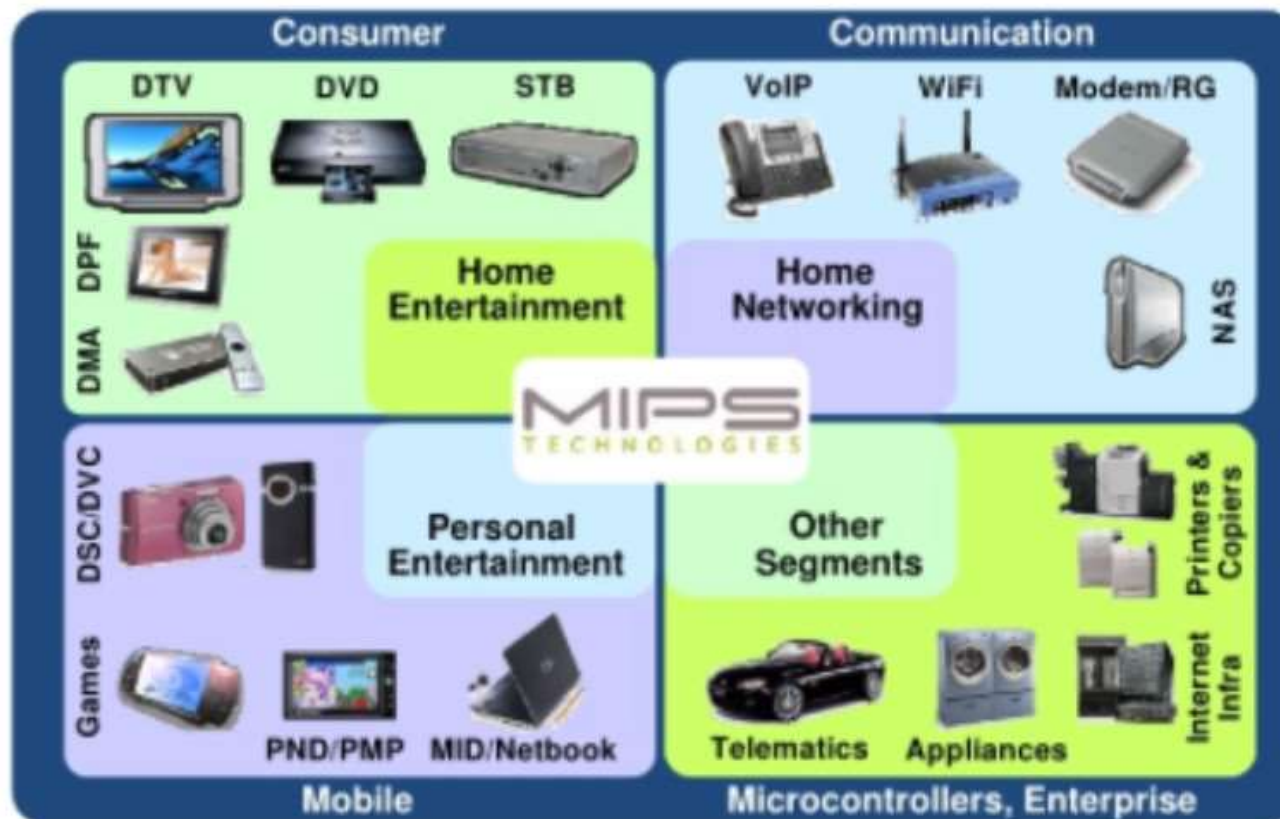
.text
li $a0 2
la $t0 PI
lw $t0 ($t0)
la $t1 RADIO
lw $t1 ($t1)
mul $a0 $a0 $t0
mul $a0 $a0 $t1

li $v0 1
syscall
```

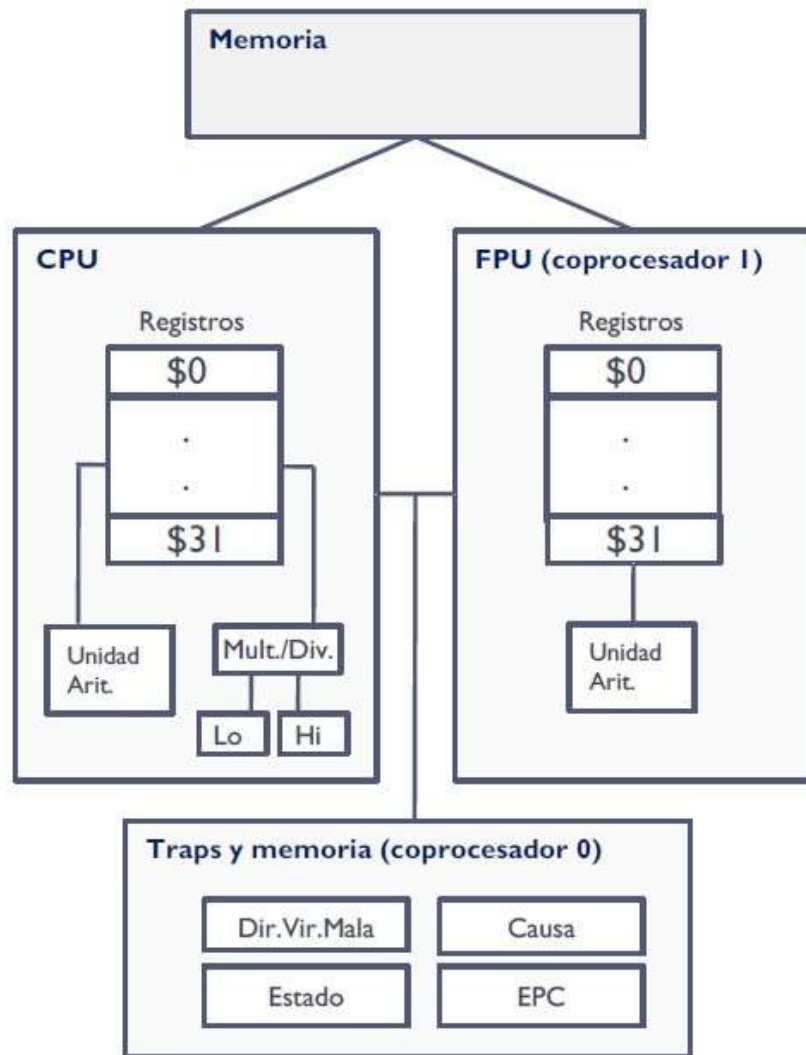




# Ejemplo de ensamblador: MIPS32



# Arquitectura MIPS



- ▶ **MIPS R2000/R3000**
  - ▶ Procesador de 32 bits
  - ▶ Tipo RISC
  - ▶ CPU + coprocesadores auxiliares
- ▶ **Coprocesador 0**
  - ▶ excepciones, interrupciones y sistema de memoria virtual
- ▶ **Coprocesador 1**
  - ▶ FPU (Unidad de Punto Flotante)

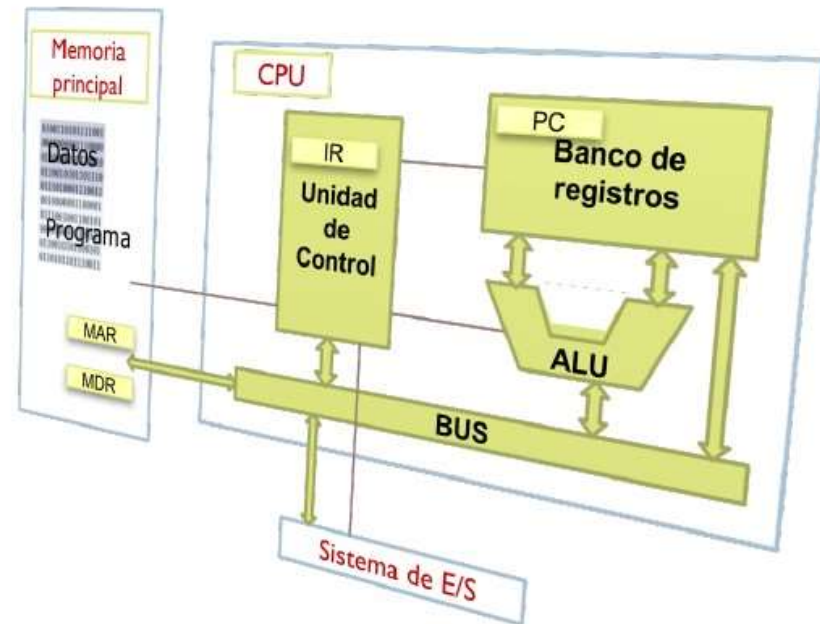
# Banco de registros (enteros)

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

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

# Tipo de instrucciones

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



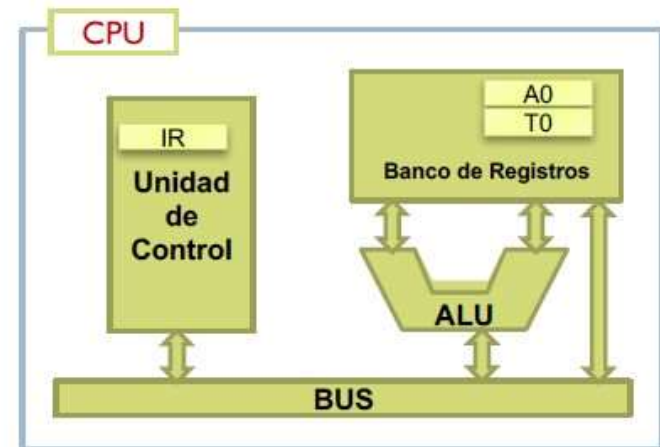
# Transferencia de datos

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

- ▶ Ejemplos:

- ▶ Registro a registro  
`move $a0 $t0`

- ▶ Carga inmediata  
`li $t0 5`

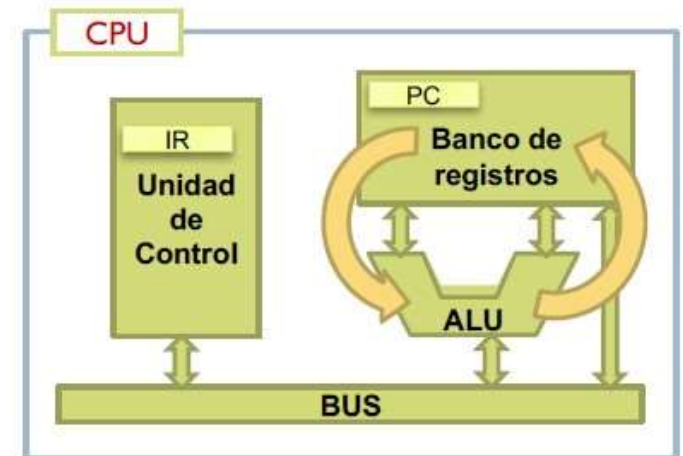


|                             |                                    |
|-----------------------------|------------------------------------|
| <code>move \$a0 \$t0</code> | <code># \$a0 ← \$t0</code>         |
| <code>li \$t0 5</code>      | <code># \$t0 ← 000....00101</code> |



# Aritméticas

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



# Operandos: Registros

- Registros:
  - \$ antes del nombre
  - Ejemplo: \$0, “registro cero”, “dolar cero”
- Los registros son usados para propósitos específicos:
  - \$0 siempre guarda el valor constante 0.
  - Los *registros de guarda*, \$s0–\$s7, se usan para almacenar variables
  - Los *registros temporales*, \$t0 - \$t9, se usan para guardar valores intermedios durante un gran computo

# Instrucciones con Registros

- Revisemos la instrucción add

## Código C

```
a = b + c
```

## Código ensamblador MIPS

```
# $s0 = a, $s1 = b, $s2 = c  
add $s0, $s1, $s2
```



# Operandos: Políticas de Memoria

- Se tienen tan solo 32 registros para almacenar muchos datos
- Se debe almacenar la mayoría de los datos en memoria
- Si bien la memoria es grande, pero es lenta
- Mantenga las variables de mayor uso en los registros

# Memoria direccionable por palabras

- Cada palabra de datos de 32 bits tiene una dirección única

| Word Address | Data            |        |
|--------------|-----------------|--------|
| ⋮            | ⋮               | ⋮      |
| 00000003     | 4 0 F 3 0 7 8 8 | Word 3 |
| 00000002     | 0 1 E E 2 8 4 2 | Word 2 |
| 00000001     | F 2 F 1 A C 0 7 | Word 1 |
| 00000000     | A B C D E F 7 8 | Word 0 |

**Nota:** MIPS usa memoria direccionable por bytes.

# Leyendo una Memoria Direccionable por Palabras

- A la lectura de memoria se le llama *load*
  - **Mnemotecnia:** *load word* ( $lw$ )
  - **Formato:**  
 $lw \$s0, 5(\$t1)$
  - **Calculo dirección:**
    - sume a la **dirección base** ( $\$t1$ ) la compensación (*offset*) (5)
    - $dirección = (\$t1 + 5)$
  - **Resultado:**
    - $\$s0$  guarda el valor leído de la dirección ( $\$t1 + 5$ )
- Cualquier registro** podría ser usado para la dirección base

# Leyendo una Memoria Direccionable por Palabra

- **Ejemplo:** Lea una palabra de datos de la dirección de memoria 1 y guárdelo en el registro `$s3`
  - dirección = `($0 + 1) = 1`
  - `$s3 = 0xF2F1AC07` luego de cargar/leer

## Código Ensamblador

```
lw $s3, 1($0) # lee de dirección 1 una palabra que se  
guarda en $s3
```

| Word Address | Data            |        |
|--------------|-----------------|--------|
| ⋮            | ⋮               | ⋮      |
| 00000003     | 4 0 F 3 0 7 8 8 | Word 3 |
| 00000002     | 0 1 E E 2 8 4 2 | Word 2 |
| 00000001     | F 2 F 1 A C 0 7 | Word 1 |
| 00000000     | A B C D E F 7 8 | Word 0 |

## Escribiendo una Memoria Direccionable por Palabra

- Las escrituras de memoria se llaman *store*
- **Mnemotecnia:** *store word* (sw)

# Escribiendo una Memoria Direccionable por Palabra

- **Ejemplo:** Escriba (guarde) el valor de  $\$t4$  en la dirección de memoria 7
  - Sume a la dirección base ( $\$0$ ) el offset (0x7)  
dirección:  $(\$0 + 0x7) = 7$

El offset puede ser escrito en decimal (por defecto) o hexadecimal

## Código Ensamblador

```
sw $t4, 0x7($0)    # escriba la palabra en $t4  
                   # a la dirección de memoria 7
```

| Word Address | Data            |        |
|--------------|-----------------|--------|
| ⋮            | ⋮               | ⋮      |
| 00000003     | 4 0 F 3 0 7 8 8 | Word 3 |
| 00000002     | 0 1 E E 2 8 4 2 | Word 2 |
| 00000001     | F 2 F 1 A C 0 7 | Word 1 |
| 00000000     | A B C D E F 7 8 | Word 0 |

# Memoria Direccionable por Byte

- Cada byte de datos tiene una direccion única
- Load/store palabras o bytes: load byte (lb) y store byte (sb)
- Palabra de 32-bit = 4 bytes, luego las direcciones de palabras se incrementan en 4

| 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 →



## Leyendo una Memoria Direccionable por Byte

- La dirección de palabra en memoria debe ser ahora multiplicada por 4.
- Ejemplo,
  - La dirección de memoria de la palabra 2 es  $2 \times 4 = 8$
  - La dirección de memoria de la palabra 10 es  $10 \times 4 = 40$  (0x28)
- **MIPS es direccionable por BYTE, no es direccionable por palabra (word)**

# Leyendo una Memoria Direccionable por Byte

- **Ejemplo:** Cargue la palabra de datos ubicada en la dirección de memoria 4 en \$s3.
- \$s3 guarda el valor 0xF2F1AC07 al momento de la carga

## Código Ensamblador MIPS

```
lw $s3, 4($0) # lee palabra de la dirección 4 y la  
escribe en $s3
```

| 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 →

# Operandos: Constantes/Inmediatos

- `lw` y `sw` usan constantes o *inmediatos*
- Están *inmediatamente* disponibles en la instrucción
- Números en complemento dos de 16 bits
- `addi`: suma inmediata
- ¿Es restar inmediatamente (`subi`) necesario?

## Código C

```
a = a + 4;  
b = a - 12;
```

## Código Ensamblador MIPS

```
# $s0 = a, $s1 = b  
addi $s0, $s0, 4  
addi $s1, $s0, -12
```



# Arquitectura de Computadores

---

Lenguaje Ensamblador  
Lenguaje de Máquina

Basado en texto: "*Digital Design and Computer Architecture, 2<sup>nd</sup> Edition*",  
David Money Harris and Sarah L. Harris

# Lenguaje de Maquina

- Representación binaria de las instrucciones
- Un computador solo entiende 1's y 0's
- Instrucciones de 32-bit
  - Simplicidad favorece regularidad: instrucciones & datos de 32-bit data
- 3 formatos de instrucciones:
  - **Tipo R:** operandos de registros
  - **Tipo I:** operando inmediato
  - **Tipo J:** para saltos

# Tipo R

- *Tipo-Registros*
- 3 operandos de registros:
  - rs, rt: registros de origen
  - rd: registros de destino
- Otros campos:
  - op: el código de operación o *opcode* (0 para instrucciones tipo R)
  - funct: la *función*  
con opcode, le dice al computador que operación realizar
  - shamt: la *cantidad de desplazamiento* para las instrucciones shift, si no hay se coloca 0

## R-Type

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



# Ejemplo Tipo R

## Assembly Code

```
add $s0, $s1, $s2
```

```
sub $t0, $t3, $t5
```

## Field Values

| op     | rs     | rt     | rd     | shamt  | funct  |
|--------|--------|--------|--------|--------|--------|
| 0      | 17     | 18     | 16     | 0      | 32     |
| 0      | 11     | 13     | 8      | 0      | 34     |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

## Machine Code

| op     | rs     | rt     | rd     | shamt  | funct  |              |
|--------|--------|--------|--------|--------|--------|--------------|
| 000000 | 10001  | 10010  | 10000  | 00000  | 100000 | (0x02328020) |
| 000000 | 01011  | 01101  | 01000  | 00000  | 100010 | (0x016D4022) |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |              |

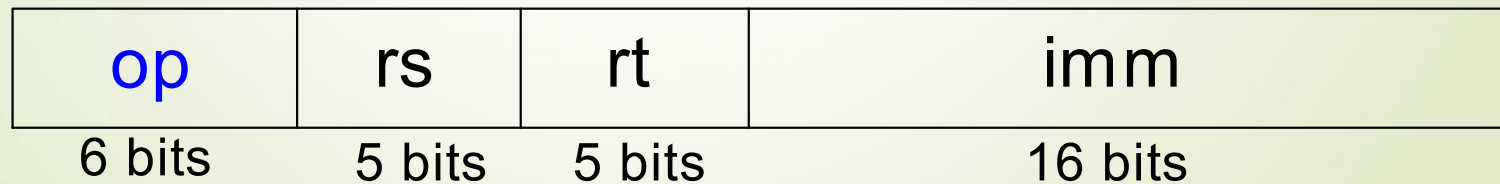
**Note** el orden de los registros en el código ensamblador:

```
add rd, rs, rt
```

# Tipo I

- *Tipo Inmediato*
- 3 operandos:
  - rs, rt: operandos de registros
  - imm: inmediato de 16-bit en complemento dos
- Otros campos:
  - op: el opcode
  - La simplicidad favorece la regularidad: todas las instrucciones tienen opcode
  - Una operación esta completamente definida por su opcode

## I-Type



# Ejemplos Tipo I

## Assembly Code

```
addi $s0, $s1, 5
addi $t0, $s3, -12
lw    $t2, 32($0)
sw    $s1, 4($t1)
```

## Field Values

| op | rs | rt | imm |
|----|----|----|-----|
| 8  | 17 | 16 | 5   |
| 8  | 19 | 8  | -12 |
| 35 | 0  | 10 | 32  |
| 43 | 9  | 17 | 4   |

6 bits      5 bits    5 bits    16 bits

**Note** la diferencia en el orden de los registros en el assembler y en los códigos de maquina:

```
addi rt, rs, imm
lw    rt, imm(rs)
sw    rt, imm(rs)
```

## Machine Code

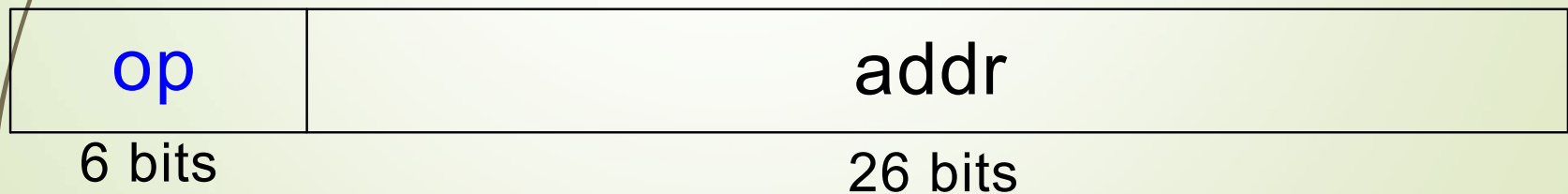
| op     | rs    | rt    | imm                 |              |
|--------|-------|-------|---------------------|--------------|
| 001000 | 10001 | 10000 | 0000 0000 0000 0101 | (0x22300005) |
| 001000 | 10011 | 01000 | 1111 1111 1111 0100 | (0x2268FFF4) |
| 100011 | 00000 | 01010 | 0000 0000 0010 0000 | (0x8C0A0020) |
| 101011 | 01001 | 10001 | 0000 0000 0000 0100 | (0xAD310004) |

6 bits      5 bits    5 bits    16 bits

# Lenguaje de Maquina: Tipo J

- *Tipo Salto (Jump)*
- El operando es una direccion de 26-bit (addr)
- Se usa para instrucciones “jump” (j)

## J-Type



# Resumen: Formatos de Instrucción

## 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 |

# Potencia de un Programa Almacenado

- Las instrucciones & datos de 32-bit instrucciones & son almacenados en memoria
- La secuencia de instrucciones: es la única diferencia entre dos aplicaciones
- Para ejecutar un programa:
  - No se requiere de un re-cableado
  - Simplemente guarde el nuevo programa en memoria
- Ejecución de un programa:
  - El Procesador *busca* (fetch/lee) las instrucciones de la memoria en secuencia
  - El procesador realiza la operación especificada



# Programa Almacenado

## Assembly Code

```
lw    $t2, 32($0)
add   $s0, $s1, $s2
addi  $t0, $s3, -12
sub   $t0, $t3, $t5
```

## Machine Code

```
0x8C0A0020
0x02328020
0x2268FFF4
0x016D4022
```

## Stored Program

| Address  | Instructions         |
|----------|----------------------|
| ⋮        | ⋮                    |
| 0040000C | 0 1 6 D 4 0 2 2      |
| 00400008 | 2 2 6 8 F F F 4      |
| 00400004 | 0 2 3 2 8 0 2 0      |
| 00400000 | 8 C 0 A 0 0 2 0 ← PC |
| ⋮        | ⋮                    |

Main Memory

**Program Counter/Contador de Programa (PC):**  
hace seguimiento de la instrucción actual que esta en ejecución

# Interpretando el código de maquina

- Comience por el opcode: nos dice como parsear el resto
- Si el opcode es todo 0's
  - Instrucción tipo R
  - Los bits de función nos dice la operación
- Sino
  - El opcode nos dirá que operación es operación

Machine Code

Field Values

Assembly Code

(0x2237FFF1)

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

|    |    |    |     |
|----|----|----|-----|
| op | rs | rt | imm |
| 8  | 17 | 23 | -15 |

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



# Arquitectura de Computadores

---

## Lenguaje Ensamblador Programación en Assembly MIPS

Basado en texto: "*Digital Design and Computer Architecture, 2<sup>nd</sup> Edition*",  
David Money Harris and Sarah L. Harris

# Programación



- Lenguajes de alto nivel:
  - Ejemplos: C, Java, Python
  - Escritos a un nivel de abstracción mas alto
- El software de alto nivel comúnmente esta construido con:
  - Sentencias if/else
  - Ciclos for
  - Ciclos while
  - arreglos
  - Llamados de función

# Programación

Programa  
en lenguaje  
de alto nivel  
(en C)

```
swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Compilador de C

Programa  
en lenguaje  
ensamblador  
(para MIPS)

```
swap:
    muli $2, $5, 4
    add  $2, $4, $2
    lw   $15, 0($2)
    lw   $16, 4($2)
    sw   $16, 0($2)
    sw   $15, 4($2)
    jr   $31
```

Ensamblador

Programa  
en lenguaje  
máquina  
binario  
(para MIPS)

```
00000000101000010000000000011000
000000000000110000001100000100001
10001100011000100000000000000000
100011001111001000000000000000100
10101100111100100000000000000000
101011000110001000000000000000100
00000011111000000000000000001000
```

# Instrucciones lógicas

- **and, or, xor, nor**

- and: util para **enmascar** bits
  - Enmascarar todos excepto el byte menos significativo de un valor:  
 $0xF234012F \text{ AND } 0x000000FF = 0x0000002F$
- or: util para **combinar** campos de bit
  - Combine 0xF2340000 con 0x000012BC:  
 $0xF2340000 \text{ OR } 0x000012BC = 0xF23412BC$
- nor: util para **invertir** bits:
  - $A \text{ NOR } \$0 = \text{NOT } A$

- **andi, ori, xori**

- El inmediato de 16-bit es extendido con cero (*no* es extendido con signo)
- `nor` no se necesitado



# Instrucciones Lógicas Ejemplo 1

## Assembly Code

```
and $s3, $s1, $s2  
or  $s4, $s1, $s2  
xor $s5, $s1, $s2  
nor $s6, $s1, $s2
```

## Source Registers

|      |      |      |      |      |      |      |      |      |
|------|------|------|------|------|------|------|------|------|
| \$s1 | 1111 | 1111 | 1111 | 1111 | 0000 | 0000 | 0000 | 0000 |
| \$s2 | 0100 | 0110 | 1010 | 0001 | 1111 | 0000 | 1011 | 0111 |

## Result

|      |  |  |  |  |  |  |  |  |
|------|--|--|--|--|--|--|--|--|
| \$s3 |  |  |  |  |  |  |  |  |
| \$s4 |  |  |  |  |  |  |  |  |
| \$s5 |  |  |  |  |  |  |  |  |
| \$s6 |  |  |  |  |  |  |  |  |

# Instrucciones Lógicas Ejemplo 1

## Assembly Code

```
and $s3, $s1, $s2  
or  $s4, $s1, $s2  
xor $s5, $s1, $s2  
nor $s6, $s1, $s2
```

## Source Registers

|      |      |      |      |      |      |      |      |      |
|------|------|------|------|------|------|------|------|------|
| \$s1 | 1111 | 1111 | 1111 | 1111 | 0000 | 0000 | 0000 | 0000 |
| \$s2 | 0100 | 0110 | 1010 | 0001 | 1111 | 0000 | 1011 | 0111 |

## Result

|      |      |      |      |      |      |      |      |      |
|------|------|------|------|------|------|------|------|------|
| \$s3 | 0100 | 0110 | 1010 | 0001 | 0000 | 0000 | 0000 | 0000 |
| \$s4 | 1111 | 1111 | 1111 | 1111 | 1111 | 0000 | 1011 | 0111 |
| \$s5 | 1011 | 1001 | 0101 | 1110 | 1111 | 0000 | 1011 | 0111 |
| \$s6 | 0000 | 0000 | 0000 | 0000 | 0000 | 1111 | 0100 | 1000 |

# Instrucciones Lógicas Ejemplo 2

## Assembly Code

```
andi $s2, $s1, 0xFA34  
ori  $s3, $s1, 0xFA34  
xori $s4, $s1, 0xFA34
```

## Source Values

\$s1

|      |      |      |      |      |      |      |      |
|------|------|------|------|------|------|------|------|
| 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 1111 | 1111 |
|------|------|------|------|------|------|------|------|

imm

|      |      |      |      |      |      |      |      |
|------|------|------|------|------|------|------|------|
| 0000 | 0000 | 0000 | 0000 | 1111 | 1010 | 0011 | 0100 |
|------|------|------|------|------|------|------|------|

← zero-extended →

## Result

|      |  |  |  |  |  |  |  |
|------|--|--|--|--|--|--|--|
| \$s2 |  |  |  |  |  |  |  |
| \$s3 |  |  |  |  |  |  |  |
| \$s4 |  |  |  |  |  |  |  |

# Instrucciones Lógicas Ejemplo 2

## Assembly Code

```
andi $s2, $s1, 0xFA34  
ori  $s3, $s1, 0xFA34  
xori $s4, $s1, 0xFA34
```

Source Values

\$s1

|      |      |      |      |      |      |      |      |
|------|------|------|------|------|------|------|------|
| 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 1111 | 1111 |
|------|------|------|------|------|------|------|------|

imm

|      |      |      |      |      |      |      |      |
|------|------|------|------|------|------|------|------|
| 0000 | 0000 | 0000 | 0000 | 1111 | 1010 | 0011 | 0100 |
|------|------|------|------|------|------|------|------|

zero-extended

Result

\$s2

|      |      |      |      |      |      |      |      |
|------|------|------|------|------|------|------|------|
| 0000 | 0000 | 0000 | 0000 | 0000 | 0000 | 0011 | 0100 |
|------|------|------|------|------|------|------|------|

\$s3

|      |      |      |      |      |      |      |      |
|------|------|------|------|------|------|------|------|
| 0000 | 0000 | 0000 | 0000 | 1111 | 1010 | 1111 | 1111 |
|------|------|------|------|------|------|------|------|

\$s4

|      |      |      |      |      |      |      |      |
|------|------|------|------|------|------|------|------|
| 0000 | 0000 | 0000 | 0000 | 1111 | 1010 | 1100 | 1011 |
|------|------|------|------|------|------|------|------|

# Instrucciones Shift (Desplazamiento)

- **sll**: desplazamiento lógico a la izquierda
  - **Ejemplo:** `sll $t0, $t1, 5` # `$t0 <= $t1 << 5`
- **srl**: desplazamiento lógico a la derecha
  - **Ejemplo:** `srl $t0, $t1, 5` # `$t0 <= $t1 >> 5`
- **sra**: desplazamiento aritmetico a la derecha
  - **Ejemplo:** `sra $t0, $t1, 5` # `$t0 <= $t1 >>> 5`

# Instrucciones de Desplazamiento Variable

- **sllv**: desplazamiento variable lógico a la izquierda
  - **Ejemplo:** `sllv $t0, $t1, $t2 # $t0 <= $t1 << $t2`
- **srlv**: desplazamiento variable lógico a la derecha
  - **Ejemplo:** `srlv $t0, $t1, $t2 # $t0 <= $t1 >> $t2`
- **srav**: desplazamiento variable aritmético a la derecha
  - **Ejemplo:** `srav $t0, $t1, $t2 # $t0 <= $t1 >>> $t2`

# Instrucciones de Desplazamiento

## Assembly Code

sll \$t0, \$s1, 2

srl \$s2, \$s1, 2

sra \$s3, \$s1, 2

## Field Values

| op     | rs     | rt     | rd     | shamt  | funct  |
|--------|--------|--------|--------|--------|--------|
| 0      | 0      | 17     | 8      | 2      | 0      |
| 0      | 0      | 17     | 18     | 2      | 2      |
| 0      | 0      | 17     | 19     | 2      | 3      |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

## Machine Code

| op     | rs     | rt     | rd     | shamt  | funct  |              |
|--------|--------|--------|--------|--------|--------|--------------|
| 000000 | 00000  | 10001  | 01000  | 00010  | 000000 | (0x00114080) |
| 000000 | 00000  | 10001  | 10010  | 00010  | 000010 | (0x00119082) |
| 000000 | 00000  | 10001  | 10011  | 00010  | 000011 | (0x00119883) |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |              |



# Generando Constantes

- `addi` usa constantes de 16-bit:

## Código C

```
// int es una palabra con signo  
// de 32-bit  
int a = 0x4f3c;
```

## Código Ensamblador MIPS

```
# $s0 = a  
addi $s0, $0, 0x4f3c
```

- `load upper immediate (lui)` y `ori` usan constantes de 32-bit:

## Código C

```
int a = 0xFEDC8765;
```

## Código Ensamblador MIPS

```
# $s0 = a  
lui $s0, 0xFEDC  
ori $s0, $s0, 0x8765
```

# Multiplicación, División

- Registros especiales: `lo`, `hi`
- Multiplicación de 32 bits x 32 bits, Resultado de 64 bit
  - `mult $s0, $s1`
  - Resultado en `{hi, lo}`
- División de 32-bit, cociente de 32-bit, resto
  - `div $s0, $s1`
  - Cociente en `lo`
  - Resto in `hi`
- Mueve desde los registros especiales `lo/hi`
  - `mflo $s2`
  - `mfhi $s3`

# Bifurcación (Branching)

- Ejecute instrucciones fuera de la secuencia
- Tipos de bifurcaciones:
  - **Condicional**
    - branch if equal (b<sub>eq</sub>)
    - branch if not equal (b<sub>ne</sub>)
  - **Incondicional**
    - jump (j)
    - jump con registro (j<sub>r</sub>)
    - jump y link (j<sub>al</sub>)

# Revisión: Programa Almacenado

## Assembly Code

```
lw    $t2, 32($0)
add   $s0, $s1, $s2
addi  $t0, $s3, -12
sub   $t0, $t3, $t5
```

## Machine Code

```
0x8C0A0020
0x02328020
0x2268FFF4
0x016D4022
```

## Stored Program

| Address  | Instructions         |
|----------|----------------------|
| ⋮        | ⋮                    |
| 0040000C | 0 1 6 D 4 0 2 2      |
| 00400008 | 2 2 6 8 F F F 4      |
| 00400004 | 0 2 3 2 8 0 2 0      |
| 00400000 | 8 C 0 A 0 0 2 0 ← PC |
| ⋮        | ⋮                    |

Main Memory

# Bifurcación condicional (beq)

## # Assembler MIPS

```
addi $s0, $0, 4           # $s0 = 0 + 4 = 4
addi $s1, $0, 1           # $s1 = 0 + 1 = 1
sll  $s1, $s1, 2          # $s1 = 1 << 2 = 4
beq  $s0, $s1, target     # esta rama es elegida
addi $s1, $s1, 1          # no se ejecuta
sub  $s1, $s1, $s0        # no se ejecuta

target:                   # label/etiqueta
add  $s1, $s1, $s0        # $s1 = 4 + 4 = 8
```

**La etiqueta o label** indica la ubicación de la instrucción **No se pueden usar palabras reservadas** y deben ser seguidas por dos puntos (:)

# La rama no elegida (bne)

## # Assembler MIPS

```
addi    $s0, $0, 4           # $s0 = 0 + 4 = 4
addi    $s1, $0, 1           # $s1 = 0 + 1 = 1
sll     $s1, $s1, 2           # $s1 = 1 << 2 = 4
bne     $s0, $s1, target     # rama no elegida
addi    $s1, $s1, 1           # $s1 = 4 + 1 = 5
sub     $s1, $s1, $s0         # $s1 = 5 - 4 = 1

target:
add     $s1, $s1, $s0         # $s1 = 1 + 4 = 5
```

# Bifurcación incondicional (j)

## # Assembler MIPS

```
addi $s0, $0, 4           # $s0 = 4
addi $s1, $0, 1           # $s1 = 1
j      target             # salte a target
sra    $s1, $s1, 2         # no se ejecuta
addi   $s1, $s1, 1         # no se ejecuta
sub     $s1, $s1, $s0      # no se ejecuta

target:
add     $s1, $s1, $s0      # $s1 = 1 + 4 = 5
```




# Bifurcación incondicional (j r)

## # Assembler MIPS

|            |                        |
|------------|------------------------|
| 0x00002000 | addi \$s0, \$0, 0x2010 |
| 0x00002004 | jr \$s0                |
| 0x00002008 | addi \$s1, \$0, 1      |
| 0x0000200C | sra \$s1, \$s1, 2      |
| 0x00002010 | lw \$s3, 44(\$s1)      |

jr es una instrucción **tipo-R**.

# Componentes de código de alto nivel

- 
- Sentencias `if`
  - Sentencias `if/else`
  - Ciclos `while`
  - Ciclos `for`

# Sentencia If

## Código C

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

## Código Assembler MIPS

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

# Sentencia If

## Código C

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

f = f - i;
```

## Código Assembler MIPS

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j
    bne $s3, $s4, L1
    add $s0, $s1, $s2

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

# Sentencia If/Else

## Código C

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

## Código Assembler 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:
```

# Ciclos While

## Código C

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


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

## Código Assembler 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:
```

# Ciclos For



```
for (inicialización; condición; operación del  
loop)  
    sentencia
```

- **inicialización:** lo ejecuta antes que parta el loop
- **condición:** es testeada al comienzo de cada iteración
- **Operación del ciclo:** se ejecuta al final de cada iteración
- **sentencia:** se ejecuta cada vez que la condición se cumpla



# Ciclos For

## Código C

```
// sume los números del 0 al 9
int sum = 0;
int i;

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

## Código assembler MIPS

```
# $s0 = i, $s1 = sum
    addi $s1, $0, 0
    add  $s0, $0, $0
    addi $t0, $0, 10
for:  beq  $s0, $t0, done
    add  $s1, $s1, $s0
    addi $s0, $s0, 1
    j    for
done:
```

# Comparacion menor que

## Código C

```
// sume las potencias de 2
// desde 1 hasta 100
int sum = 0;
int i;

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

## Código assembler 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:
```

# Arreglos

- Queremos acceder a grandes volúmenes de datos similares
- **Indice**: accede a cada elemento
- **Tamaño**: numero de elementos

# Arreglos

- Arreglo de 5 elementos
- **Dirección base** = 0x12348000 (dirección del primer elemento, `array[0]`)
- El primer paso al acceder a un arreglo: cargar dirección base en un registro

|            |          |
|------------|----------|
| 0x12340010 | array[4] |
| 0x1234800C | array[3] |
| 0x12348008 | array[2] |
| 0x12348004 | array[1] |
| 0x12348000 | array[0] |

# Accediendo a un Arreglo

## // Código C

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

## # Código assembler MIPS

# \$s0 = dirección base del arreglo

```
    lui    $s0, 0x1234          # 0x1234 en la mitad superior de $s0  
    ori    $s0, $s0, 0x8000     # 0x8000 en la mitad inferior 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
```

# Usando un For para acceder a un Arreglo

## // Código C

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

## # Código Assembler MIPS

```
# $s0 = dirección base del arreglo, $s1 = i
```

# Usando un For para acceder a un Arreglo

## # Código asembler MIPS

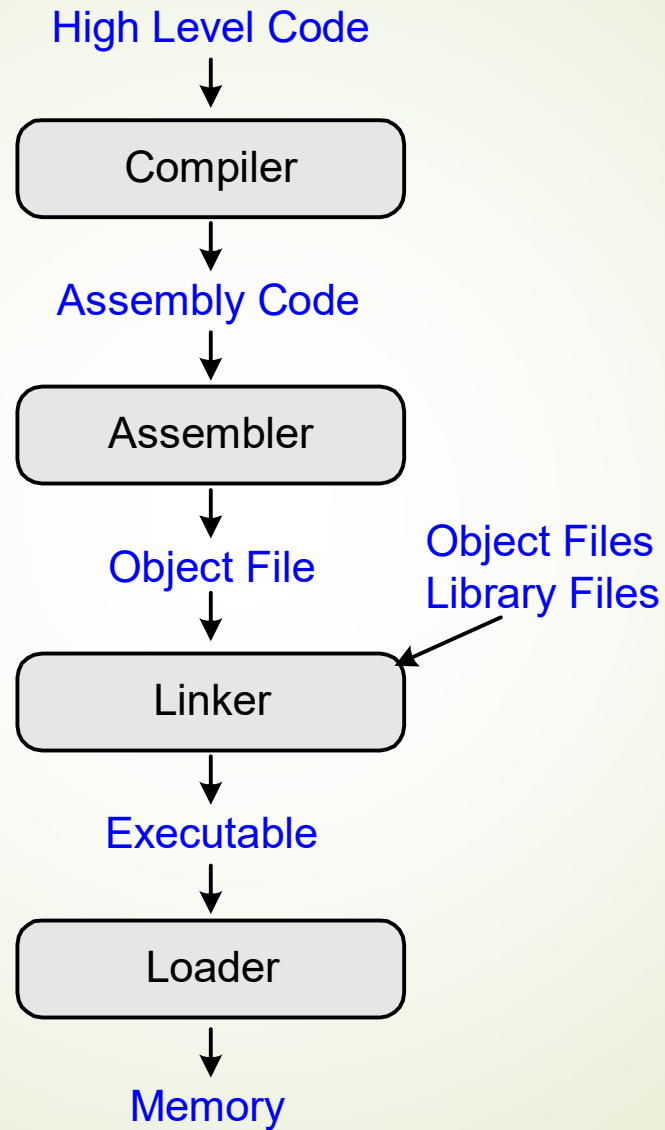
```
# $s0 = direccion base del arreglo, $s1 = i
# codigo de inicialización
lui   $s0, 0x23B8           # $s0 = 0x23B80000
ori   $s0, $s0, 0xF000      # $s0 = 0x23B8F000
addi  $s1, $0, 0            # i = 0
addi  $t2, $0, 1000         # $t2 = 1000

loop:
    slt  $t0, $s1, $t2      # i < 1000?
    beq  $t0, $0, done      # en caso contrario fin (done)
    sll  $t0, $s1, 2         # $t0 = i * 4 (offset del
    byte)
    add  $t0, $t0, $s0       # dirección de array[i]
    lw   $t1, 0($t0)         # $t1 = array[i]
    sll  $t1, $t1, 3         # $t1 = array[i] * 8
    sw   $t1, 0($t0)         # array[i] = array[i] * 8
    addi $s1, $s1, 1         # i = i + 1
    j    loop               # repetir

done:
```



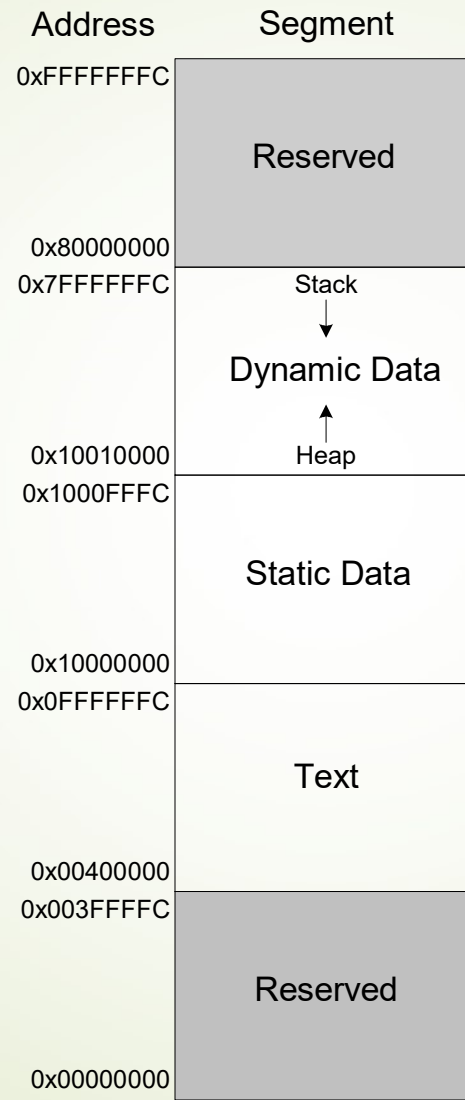
# ¿Como compilar & Correr un Programa?



# ¿Que se guarda en la Memoria?

- Instrucciones (también llamado *texto (text)*)
- Datos
  - Globales/estáticas: se asignan antes que el programa parta
  - Dinámica: asignada al ejecutar el programa
- ¿Cuan grande es la memoria?
  - A lo mas  $2^{32} = 4$  gigabytes (4 GB)
  - De la dirección 0x00000000 a la 0xFFFFFFFF

# Mapa de Memoria MIPS



# Ejemplo Programa: en C y Assembler MIPS

```
int f, g, y; // Var global
```

```
int main(void)
```

```
{
```

```
    f = 2;
```

```
    g = 3;
```

```
    y = suma(f, g);
```

```
    return y;
```

```
}
```

```
int suma(int a, int b) {
```

```
    return (a + b);
```

```
}
```

```
.data
```

```
f:
```

```
g:
```

```
y:
```

```
.text
```

```
main:
```

```
    addi $sp, $sp, -4    # stack frame
```

```
    sw   $ra, 0($sp)    # guarde $ra
```

```
    addi $a0, $0, 2     # $a0 = 2
```

```
    sw   $a0, f         # f = 2
```

```
    addi $a1, $0, 3     # $a1 = 3
```

```
    sw   $a1, g         # g = 3
```

```
    jal  suma          # llamar suma
```

```
    sw   $v0, y         # y = suma()
```

```
    lw   $ra, 0($sp)    # restaure $ra
```

```
    addi $sp, $sp, 4    # restaure $sp
```


```
    jr   $ra           # retorne al SO
```

```
suma:
```

```
    add  $v0, $a0, $a1  # $v0 = a + b
```

```
    jr   $ra           # retornar
```

# Programa Ejemplo: Tabla de Símbolos



| Símbolo | Dirección   |
|---------|-------------|
| f       | 0x100000000 |
| g       | 0x100000004 |
| y       | 0x100000008 |
| main    | 0x00400000  |
| sum     | 0x0040002C  |

# Fin de Aspectos Fundamentales

- Manejo básico del assembler MIPS
  - Creación de variables a través de registros
  - Creación de arreglo de memoria
  - Instrucciones lógico & matemáticas
  - Instrucción de bucle: for
  - Instrucción de bifurcación: If, If-else
  - Llamada de funciones



Fin Tema

