# Resumen de MIPS32: Lenguaje Ensamblador

Angel Besteiro

## Introducción al Lenguaje Ensamblador MIPS32

El **lenguaje máquina** es un concepto fundamental para entender cómo funcionan las computadoras. Se compone de una serie de "instrucciones" o "pasos a seguir" que el equipo ejecuta para completar una tarea específica. Aunque existen muchos lenguajes de programación, a grandes rasgos no difieren demasiado entre sí, y los equipos que los emplean tienen requisitos y propósitos similares. Por supuesto, cada lenguaje tiene sus particularidades, ventajas y desventajas, y la elección del lenguaje adecuado depende de la pericia del programador.

El lenguaje que presentamos hoy es un **lenguaje de bajo nivel**, lo que significa que está muy cerca del lenguaje máquina. El procesador opera con instrucciones en binario; mediante este lenguaje, podemos darle instrucciones que son traducidas a binario y nos retornan resultados que son interpretados del binario para su comprensión. Como se mencionó, el lenguaje se basa en instrucciones, y el ensamblador MIPS32, al que se refiere este escrito, tiene una serie de instrucciones preestablecidas con el fin de agilizar el código y el flujo del algoritmo.

Se dispone de **memoria** y **registros**. La memoria es donde se almacenan los datos de distinto tipo. Los registros son también una forma de memoria, pero es donde el sistema puede operar directamente. Para ser utilizados en operaciones, los datos deben ser movilizados de la memoria a un registro y luego devueltos a su origen o a otra ubicación en la memoria. MIPS32 cuenta con 32 registros, algunos específicos y otros de uso más general.

## Operaciones

### Aritméticas

Estas operaciones son básicas en cualquier sistema: suma y resta.

- **add:** Formato: 'add($s1$,s2, $s3$) $\implies$ \$$s1$ = \$$s2$+\$$s3$'.$Tomaelprimerregistroycolocaelresultadodelasumadelosdosú$
  '$addi$(s1, $s2, 100$) $\implies$ \$$s1$ = \$$s2 + 100$'.

- **sub (subtract):** Funciona como una resta. Formato: 'sub($s1$,s2, $s3$) $\implies$ \$$s1$ = \$$s2$−\$$s3$', $calculandoladiferenciadelosi$

### Transferencia de Datos

Operaciones específicas para movilizar datos entre espacios de memoria y registros.

- **lw (load word):** Moviliza una palabra (32 bits) de una dirección de memoria a un registro.
- **sw (store word):** Moviliza una palabra (32 bits) de un registro a una dirección de memoria.
- **lh (load half):** Carga media palabra (16 bits) de memoria a un registro.
- **sh (store half):** Carga media palabra (16 bits) de un registro a memoria.
- **lb (load byte):** Carga un byte (8 bits) de memoria a un registro.
- **sb (store byte):** Carga un byte (8 bits) de un registro a memoria.
- **lui (load upper immediate):** Carga una constante de 16 bits en la parte superior (más significativa) de un registro.

### Lógicas

Instrucciones que realizan operaciones lógicas bit a bit.

- **and:** Opera bit a bit con tres registros, implementando una compuerta AND.
- **or:** Opera bit a bit con tres registros, implementando una compuerta OR.
- **not:** Opera bit a bit con un registro (pseudo-instrucción que usa 'nor' con '$zero$'), $implementandounacompuertaNO$

- **ori (or immediate):** Opera bit a bit con un registro y una constante, implementando una compuerta OR.
- **sll (shift left logical):** Desplazamiento lógico a la izquierda de bits por una constante.
- **srl (shift right logical):** Desplazamiento lógico a la derecha de bits por una constante.

### Salto Condicional

Instrucciones que alteran el flujo de ejecución basándose en una condición.

- **beq (branch on equal):** Comprueba si dos registros son iguales y salta si lo son.
- **bne (branch on not equal):** Comprueba si dos registros no son iguales y salta si no lo son.
- **slt (set on less than):** Compara si el primer registro es menor que el segundo, y pone 1 en un registro destino si es verdad, 0 si es falso.
- **slti (set on less than immediate):** Compara si un registro es menor que una constante, y pone 1 en un registro destino si es verdad, 0 si es falso.

### Salto Incondicional

Instrucciones que alteran el flujo de ejecución sin una condición.

- **j (jump):** Salta a una dirección de destino especificada.
- **jr (jump register):** Salta a la dirección contenida en un registro (comúnmente usado para retornar de un procedimiento).
- **jal (jump and link):** Salta a una dirección de destino y guarda la dirección de retorno (la siguiente instrucción) en el registro `$ra` (usado para llamadas a procedimientos).

## Registros de MIPS32

La arquitectura MIPS32 cuenta con 32 registros de propósito general, además de algunos registros especiales. Sus usos convencionales son los siguientes:

- `$zero` **o** `$0` **(Registro 0):** Siempre contiene el valor 0. No se puede escribir en él. Es útil para inicializar registros a cero o para operaciones que requieren un cero.
- `$at` **o** `$1` **(Assembler Temporary):** Reservado para el ensamblador. Lo usa para convertir pseudo-instrucciones. No debe ser utilizado por el programador.
- `$v0-$v1` **o** `$2-$3` **(Value):** Utilizados para retornar valores de funciones. `$v0` es el registro principal para el valor de retorno, y `$v1` se usa para un segundo valor si es necesario.
- `$a0-$a3` **o** `$4-$7` **(Argument):** Utilizados para pasar los primeros cuatro argumentos a una función. Si hay más argumentos, se pasan en la pila. Son "caller-saved".
- `$t0-$t9` **o** `$8-$15, $24-$25` **(Temporary):** Registros temporales. Se utilizan para almacenar valores que no necesitan preservarse entre llamadas a funciones. Son "caller-saved".
- `$s0-$s7` **o** `$16-$23` **(Saved):** Registros guardados. Se utilizan para almacenar valores que deben preservarse entre llamadas a funciones. Son "callee-saved".

- `$k0-$k1` o `$26-$27` **(Kernel):** Reservados para el sistema operativo (kernel). Los programas de usuario no deben usarlos.
- `$gp` o `$28` **(Global Pointer):** Puntero a la sección de datos globales. Facilita el acceso a variables globales y estáticas.
- `$sp` o `$29` **(Stack Pointer):** Puntero de pila. Apunta al tope de la pila y se utiliza para gestionar el marco de pila en llamadas a funciones.
- `$fp` o `$30` **(Frame Pointer):** Puntero de marco. Apunta a una posición fija dentro del marco de pila actual. No siempre se usa.
- `$ra` o `$31` **(Return Address):** Dirección de retorno. Almacena la dirección de la instrucción inmediatamente después de una llamada a función (`jal`).