Instrucciones y Registros en MIPS 32

Rolando Suarez
V- 30.445.947

Resumen

Este documento proporciona un resumen detallado y completo de la arquitectura de conjuntos de instrucciones MIPS de 32 bits, centrándose en la organización de sus registros de propósito general y especial, así como en los formatos y categorías principales de sus instrucciones. MIPS, una arquitectura RISC fundamental, es explorada en su simplicidad y eficiencia para el procesamiento de datos.

1. Introducción a MIPS 32

MIPS (Microprocessor without Interlocked Pipeline Stages) es una arquitectura de conjunto de instrucciones reducido (RISC) de 32 bits que se caracteriza por su simplicidad, regularidad y diseño optimizado para el *pipelining*. Esto significa que las instrucciones son generalmente de longitud fija, tienen un formato sencillo y se ejecutan en un solo ciclo de reloj (cuando la pipeline está llena).

2. Registros MIPS 32

Los registros son pequeñas ubicaciones de almacenamiento de alta velocidad directamente en la CPU, utilizadas para almacenar datos sobre los que se está operando activamente. MIPS-32 tiene 32 registros de propósito general (GPRs), cada uno de 32 bits de ancho, más algunos registros especiales.

2.1. Registros de Propósito General (GPRs)

Aunque los registros son genéricos en su funcionalidad, existe una convención de uso establecida para facilitar la programación y la interoperabilidad de las funciones (llamada convención de llamada o *calling convention*). Se nombran como `$0` a `$31`, o con un nombre simbólico que indica su propósito:

2.2. Registros de Propósito Especial

- **PC (Program Counter):** Contiene la dirección de memoria de la próxima instrucción a ejecutar. Se actualiza automáticamente.
- **HI (High) y LO (Low):** Registros utilizados para almacenar los resultados de operaciones de multiplicación (los 64 bits de un producto de 32x32 se dividen entre HI y LO) y división (LO para el cociente, HI para el resto).

3. Instrucciones MIPS 32

Todas las instrucciones MIPS son de **32 bits de longitud fija**, lo que simplifica enormemente la fase de *fetch* y *decode* de la CPU. Se agrupan en tres formatos principales:

3.1. Formatos de Instrucción

1. **Formato R (Register):** Usado para operaciones que involucran únicamente registros.
   - **Opcode (6 bits):** Código de operación principal.
   - **rs (5 bits):** Primer registro fuente.
   - **rt (5 bits):** Segundo registro fuente.

Cuadro 1: Convención de uso de los Registros de Propósito General (GPRs) en MIPS 32

Número	Nombre Simbólico	Propósito Convencional
R0	`$zero`	**Siempre cero**. No se puede escribir en él; leer de él siempre devuelve 0. Muy útil.
R1	`$at`	**Ensamblador temporal**. Usado por el ensamblador para traducir pseudo-instrucciones. No debe ser usado por el programador directamente.
R2-R3	`$v0-$v1`	**Valores de retorno de funciones**. Usados para devolver resultados de una función (e.g., `$v0` para el resultado principal, `$v1` para un segundo resultado o código de error).
R4-R7	`$a0-$a3`	**Argumentos de funciones**. Usados para pasar los primeros cuatro argumentos a una función.
R8-R15	`$t0-$t7`	**Temporales**. No se espera que se preserven a través de llamadas a funciones (la función que llama puede modificar su valor).
R16-R23	`$s0-$s7`	**Temporales guardados**. Se espera que se preserven a través de llamadas a funciones. Si una función llamada los necesita, debe guardarlos en la pila y restaurarlos antes de retornar.
R24-R25	`$t8-$t9`	**Más temporales**. Similar a `$t0-$t7`.
R26-R27	`$k0-$k1`	**Reservado para el kernel del sistema operativo**. No deben ser usados por programas de usuario.
R28	`$gp`	**Puntero global (Global Pointer)**. Usado para acceder a variables globales de manera eficiente.
R29	`$sp`	**Puntero de pila (Stack Pointer)**. Apunta al tope de la pila de llamadas. Fundamental para la gestión de la memoria de las funciones.
R30	`$fp / $s8`	**Puntero de marco (Frame Pointer)** o como `$s8` (otro temporal guardado). Se usa para acceder a argumentos y variables locales en la pila.
R31	`$ra`	**Dirección de retorno (Return Address)**. Al realizar una llamada a función (`jal`), la dirección de la siguiente instrucción se guarda aquí para saber dónde volver.

- **rd (5 bits):** Registro destino donde se guarda el resultado.
- **shamt (5 bits):** Cantidad de desplazamiento (para instrucciones de shift).
- **funct (6 bits):** Código de función, para especificar la operación exacta cuando el opcode es el mismo (por ejemplo, para `ADD` vs `SUB` tienen el mismo opcode principal, pero diferente `funct`).

Ejemplo: `add $rd, $rs, $rt` (`$rd = $rs + $rt`)

2. **Formato I (Immediate):** Usado para operaciones con un operando inmediato (constante) o para instrucciones de carga/almacenamiento.
   - **Opcode (6 bits):** Código de operación principal.
   - **rs (5 bits):** Registro fuente.
   - **rt (5 bits):** Registro destino (para resultado inmediato) o registro fuente/destino (para load/store).
   - **Immediate (16 bits):** Valor constante, dirección de memoria con signo (offset), o dirección de rama.

   Ejemplo: `addi $rt, $rs, immediate` (`$rt = $rs + immediate`)
   Ejemplo Load/Store: `lw $rt, offset($rs)` (`$rt = Mem[$rs + offset]`)

3. **Formato J (Jump):** Usado para saltos incondicionales.
    - **Opcode (6 bits):** Código de operación principal.
    - **Target Address (26 bits):** Parte de la dirección de destino del salto.

    Ejemplo: `j target_address`

3.2. Categorías de Instrucciones Comunes

1. **Aritméticas:** Realizan operaciones matemáticas con enteros.
    - `add rd, rs, rt`: Suma (con detección de overflow).
    - `addu rd, rs, rt`: Suma sin signo (sin detección de overflow).
    - `sub rd, rs, rt`: Resta (con detección de overflow).
    - `subu rd, rs, rt`: Resta sin signo.
    - `addi rt, rs, immediate`: Suma inmediata.
    - `mult rs, rt`: Multiplica `$rs` por `$rt`, resultado en `HI/LO`.
    - `div rs, rt`: Divide `$rs` por `$rt`, cociente en `LO`, resto en `HI`.
2. **Lógicas:** Realizan operaciones bit a bit.
    - `and rd, rs, rt`: AND lógico.
    - `or rd, rs, rt`: OR lógico.
    - `xor rd, rs, rt`: XOR lógico.
    - `nor rd, rs, rt`: NOR lógico.
    - `andi rt, rs, immediate`: AND inmediato.
    - `ori rt, rs, immediate`: OR inmediato.
    - `xori rt, rs, immediate`: XOR inmediato.
3. **Desplazamiento (Shift):** Mueven bits dentro de un registro.
    - `sll rd, rt, shamt`: Desplazamiento lógico a la izquierda.
    - `srl rd, rt, shamt`: Desplazamiento lógico a la derecha.
    - `sra rd, rt, shamt`: Desplazamiento aritmético a la derecha (preserva el signo).
4. **Transferencia de Datos (Load/Store):** Mueven datos entre registros y memoria.
    - `lw rt, offset(base)`: Carga una palabra (32 bits) de memoria a un registro.
    - `sw rt, offset(base)`: Almacena una palabra de un registro a memoria.
    - `lb rt, offset(base)`: Carga un byte (sign-extended).
    - `lbu rt, offset(base)`: Carga un byte sin signo.
    - `sb rt, offset(base)`: Almacena un byte.
    - `li rt, immediate`: Pseudo-instrucción para cargar un valor inmediato (el ensamblador lo traduce a `lui` + `ori` o `addi`).
5. **Control de Flujo (Branches y Jumps):** Cambian la secuencia de ejecución del programa.
    - `beq rs, rt, label`: Salto si `$rs` es igual a `$rt`.
    - `bne rs, rt, label`: Salto si `$rs` no es igual a `$rt`.
    - `j label`: Salto incondicional.
    - `jal label`: Salto y enlace (Jump and Link). Guarda la dirección de retorno en `$ra` antes de saltar. Se usa para llamadas a funciones.
    - `jr rs`: Salto de registro (Jump Register). Salta a la dirección contenida en `$rs`. Se usa para retornar de funciones (con `$ra`).

6. **Comparación y Establecimiento (Set on Less Than):**
   - `slt rd, rs, rt`: Establece `$rd` a 1 si `$rs <$rt`, 0 en caso contrario (con signo).
   - `sltu rd, rs, rt`: Establece `$rd` a 1 si `$rs <$rt`, 0 en caso contrario (sin signo).
   - `slti rt, rs, immediate`: Establece si es menor que un inmediato (con signo).
   - `sltiu rt, rs, immediate`: Establece si es menor que un inmediato (sin signo).

4. Consideraciones Adicionales

- **Pseudo-instrucciones:** El ensamblador MIPS proporciona "pseudo-instrucciones"que son más fáciles de usar para el programador (ej. `move $rd, $rs`, `li $rt, immediate`). El ensamblador las convierte internamente en una o más instrucciones MIPS reales.
- **Endianness:** MIPS puede configurarse para ser *Big-Endian* (el byte más significativo en la dirección de memoria más baja) o *Little-Endian* (el byte menos significativo en la dirección de memoria más baja). Esto es crucial para la portabilidad de datos.
- **Pipeline:** El diseño de MIPS está fuertemente enfocado en permitir una ejecución eficiente de instrucciones en un pipeline (una cadena de etapas de procesamiento). La uniformidad de las instrucciones y el acceso a los registros en posiciones fijas facilitan esto.