MIPS Encoding Reference

| **Mnemonic** | **Meaning** | **Type** | **Opcode** | | **Funct** | |
| --- | --- | --- | --- | --- | --- | --- |
| add | Add | R | 00 | 000000 | 32 | 100000 |
| addu | Add Unsigned | R | 00 | 000000 | 33 | 100001 |
| and | Bitwise AND | R | 00 | 000000 | 36 | 100100 |
| div | Divide | R | 00 | 000000 | 26 | 011010 |
| divu | Unsigned Divide | R | 00 | 000000 | 27 | 011011 |
| jr | Jump to Address in Register | R | 00 | 000000 | 08 | 001000 |
| mfhi | Move from HI Register | R | 00 | 000000 | 16 | 010000 |
| mflo | Move from LO Register | R | 00 | 000000 | 18 | 010010 |
| mfc0 | Move from Coprocessor 0 | R | 16 | 010000 | NA | NA |
| mult | Multiply | R | 00 | 000000 | 24 | 011000 |
| multu | Unsigned Multiply | R | 00 | 000000 | 25 | 011001 |
| nor | Bitwise NOR (NOT-OR) | R | 00 | 000000 | 39 | 100111 |
| xor | Bitwise XOR (Exclusive-OR) | R | 00 | 000000 | 38 | 100110 |
| or | Bitwise OR | R | 00 | 000000 | 37 | 100101 |
| slt | Set to 1 if Less Than | R | 00 | 000000 | 42 | 101010 |
| sltu | Set to 1 if Less Than Unsigned | R | 00 | 000000 | 43 | 101011 |
| sll | Logical Shift Left | R | 00 | 000000 | 00 | 000000 |
| srl | Logical Shift Right (0-extended) | R | 00 | 000000 | 02 | 000010 |
| sra | Arithmetic Shift Right (sign-extended) | R | 00 | 000000 | 03 | 000011 |
| sub | Subtract | R | 00 | 000000 | 34 | 100010 |
| subu | Unsigned Subtract | R | 00 | 000000 | 35 | 100011 |
| j | Jump to Address | J | 02 | 000010 | NA | |
| jal | Jump and Link | J | 03 | 000011 | NA | |
| addi | Add Immediate | I | 08 | 001000 | NA | |
| addiu | Add Unsigned Immediate | I | 09 | 001001 | NA | |
| andi | Bitwise AND Immediate | I | 12 | 001100 | NA | |
| beq | Branch if Equal | I | 04 | 000100 | NA | |
| bne | Branch if Not Equal | I | 05 | 000101 | NA | |
| lbu | Load Byte Unsigned | I | 36 | 100100 | NA | |
| lhu | Load Halfword Unsigned | I | 37 | 100101 | NA | |
| lui | Load Upper Immediate | I | 15 | 001111 | NA | |
| lw | Load Word | I | 35 | 100011 | NA | |
| ori | Bitwise OR Immediate | I | 13 | 001101 | NA | |
| sb | Store Byte | I | 40 | 101000 | NA | |
| sh | Store Halfword | I | 41 | 101001 | NA | |
| slti | Set to 1 if Less Than Immediate | I | 10 | 001010 | NA | |
| sltiu | Set to 1 if Less Than Unsigned Immediate | I | 11 | 001011 | NA | |
| sw | Store Word | I | 43 | 101011 | NA | |

**Instruction Encodings**

Each MIPS instruction is encoded in exactly one word (32 bits). There are three encoding formats.

**Register Encoding (R-Type)**

The prototypical R-type instruction is:

add $rd, $rs, $rt

The semantics of the instruction are:

R[d] = R[s] + R[t]

This encoding is used for instructions which do not require any immediate data. These instructions receive all their operands in registers. Additionally, certain of the bit shift instructions use this encoding; their operands are two registers and a 5-bit shift amount.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **op (6 bits)** | | | | | | **rs (5 bits)** | | | | | | **rt (5 bits)** | | | | |  | **rd (5 bits)** | | | | | **shamt (5 bits)** | | | | | | **funct (6 bits)** | | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| o | o | o | o | o | o | s | s | s | s | s | t | t | t | t | t | d | d | d | d | d | a | a | a | a | a | f | f | f | f | f | f |

|  |  |  |
| --- | --- | --- |
| **Field** | **Width** | **Description** |
| o | 6 | Instruction opcode. This is 000000 for instructions using this encoding. |
| s | 5 | First source register, in the range 0-31. |
| t | 5 | Second source register, in the range 0-31. |
| d | 5 | Destination register, in the range 0-31. |
| a | 5 | Shift amount, for shift instructions. |
| f | 6 | Function. Determines which operation is to be performed. Values for this field are documented in the tables at the bottom of this page. |

**Immediate Encoding (I-Type)**

The prototypical I-type instruction looks like:

add $rt, $rs, immed

The semantics of the addi instruction are;

R[t] = R[s] + (IR15)16 IR15-0

where IR refers to the instruction register, the register where the current instruction is stored. (IR15)16 means that bit B15 of the instruction register (which is the sign bit of the immediate value) is repeated 16 times. This is then followed by IR15-0, which is the 16 bits of the immediate value.

Basically, the semantics says to sign-extend the immediate value to 32 bits, add it (using signed addition) to register R[s], and store the result in register $rt.

This encoding is used for instructions which require a 16-bit immediate operand. These instructions typically receive one operand in a register, another as an immediate value coded into the instruction itself, and place their results in a register. This encoding is also used for load, store, branch, and other instructions so the use of the fields is different in some cases.

Note that the "first" and "second" registers are not always in this order in the assembly language; see "Instruction Syntax" for details.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **op (6 bits)** | | | | | | **rs (5 bits)** | | | | | | **rt (5 bits)** | | | | |  | **Immediate data (16 bits)** | | | | | | | | | | | | | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| o | o | o | o | o | o | s | s | s | s | s | t | t | t | t | t | i | i | i | i | i | i | i | i | i | i | i | i | i | i | i | i |

|  |  |  |
| --- | --- | --- |
| **Field** | **Width** | **Description** |
| o | 6 | Instruction opcode. Determines which operation is to be performed. Values for this field are documented in the tables at the bottom of this page. |
| s | 5 | First register, in the range 0-31. |
| t | 5 | Second register, in the range 0-31. |
| i | 16 | Immediate data. These 16 bits of immediate data are interpreted differently for different instructions. 2's-complement encoding is used to represent a number between -215 and 215-1. |

**Jump Encoding (J-Type)**

The prototypical I-type instruction looks like:

j target

The semantics of the **j** instruction (**j** means jump) are:

PC <- PC31-28 IR25-0 00

where PC is the program counter, which stores the current address of the instruction being executed. You update the PC by using the upper 4 bits of the program counter, followed by the 26 bits of the target (which is the lower 26 bits of the instruction register), followed by two 0's, which creates a 32 bit address. The jump instruction will be explained in more detail in a future set of notes.

This encoding is used for jump instructions, which require a 26-bit immediate offset. It is also used for the trap instruction.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **op (6 bits)** | | | | | | **Immediate data (26 bits)** | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| o | o | o | o | o | o | i | i | i | i | i | i | i | i | i | i | i | i | i | i | i | i | i | i | i | i | i | i | i | i | i | i |

|  |  |  |
| --- | --- | --- |
| **Field** | **Width** | **Description** |
| o | 6 | Instruction opcode. Determines which operation is to be performed. Values for this field are documented in the tables at the bottom of this page. |
| i | 26 | Immediate data. These 26 bits of immediate data are interpreted differently for different instructions. 2's-complement encoding is used to represent a number between -225 and 225-1. |

**Instruction Syntax**

This is a table of all the different types of instruction as they appear in the assembly listing. Note that each syntax is associated with exactly one encoding which is used to encode all instructions which use that syntax.

|  |  |  |  |
| --- | --- | --- | --- |
| **Encoding** | **Syntax** | **Template** | **Comments** |
| Register | ArithLog | f $d, $s, $t |  |
| DivMult | f $s, $t |  |
| Shift | f $d, $t, a |  |
| ShiftV | f $d, $t, $s |  |
| JumpR | f $s |  |
| MoveFrom | f $d |  |
| MoveTo | f $s |  |
| Immediate | ArithLogI | o $t, $s, i |  |
| LoadI | o $t, immed32 | i is high or low 16 bits of immed32 |
| Branch | o $s, $t, label | i is calculated as (label - (current + 4)) >> 2 |
| BranchZ | o $s, label | i is calculated as (label - (current + 4)) >> 2 |
| LoadStore | o $t, i ($s) |  |
| Jump | Jump | o label | i is calculated as (label - (current + 4)) >> 2 |
| Trap | o i |  |

**Opcode Table**

These tables list all of the available operations in MIPS. For each instruction, the 6-bit opcode or function is shown. The syntax column indicates which syntax is used to write the instruction in assembly text files. Note that which syntax is used for an instruction also determines which encoding is to be used. Finally the operation column describes what the operation does in pseudo-Java plus some special notation as follows:

"MEM [*a*]:*n*" means the *n* bytes of memory starting with address *a*.  
The address must always be aligned; that is, *a* must be divisible by *n*, which must be a power of 2.

"LB (*x*)" means the least significant 8 bits of the 32-bit location *x*.  
"LH (*x*)" means the least significant 16 bits of the 32-bit location *x*.  
"HH (*x*)" means the most significant 16 bits of the 32-bit location *x*.

"SE (*x*)" means the 32-bit quantity obtained by extending the value *x* on the left with its most significant bit.  
"ZE (*x*)" means the 32-bit quantity obtained by extending the value *x* on the left with 0 bits.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Arithmetic and Logical Instructions** | | | | |
| **Instruction** | **Opcode/Function** | | **Syntax** | **Operation** |
| add | 100000 | 32 | ArithLog | $d = $s + $t |
| addu | 100001 | 33 | ArithLog | $d = $s + $t |
| addi | 001000 | 8 | ArithLogI | $t = $s + SE(i) |
| addiu | 001001 | 9 | ArithLogI | $t = $s + SE(i) |
| and | 100100 | 36 | ArithLog | $d = $s & $t |
| andi | 001100 | 12 | ArithLogI | $t = $s & ZE(i) |
| div | 011010 | 26 | DivMult | lo = $s / $t; hi = $s % $t |
| divu | 011011 | 27 | DivMult | lo = $s / $t; hi = $s % $t |
| mult | 011000 | 24 | DivMult | hi:lo = $s \* $t |
| multu | 011001 | 25 | DivMult | hi:lo = $s \* $t |
| nor | 100111 | 39 | ArithLog | $d = ~($s | $t) |
| or | 100101 | 37 | ArithLog | $d = $s | $t |
| ori | 001101 | 13 | ArithLogI | $t = $s | ZE(i) |
| sll | 000000 | 0 | Shift | $d = $t << a |
| sllv | 000100 | 4 | ShiftV | $d = $t << $s |
| sra | 000011 | 3 | Shift | $d = $t >> a |
| srav | 000111 | 7 | ShiftV | $d = $t >> $s |
| srl | 000010 | 2 | Shift | $d = $t >>> a |
| srlv | 000110 | 6 | ShiftV | $d = $t >>> $s |
| sub | 100010 | 34 | ArithLog | $d = $s - $t |
| subu | 100011 | 35 | ArithLog | $d = $s - $t |
| xor | 100110 | 38 | ArithLog | $d = $s ^ $t |
| xori | 001110 | 14 | ArithLogI | $d = $s ^ ZE(i) |
| **Constant-Manipulating Instructions** | | | | |
| **Instruction** | **Opcode/Function** | | **Syntax** | **Operation** |
| lhi | 011001 | 25 | LoadI | HH ($t) = i |
| llo | 011000 | 24 | LoadI | LH ($t) = i |
| **Comparison Instructions** | | | | |
| **Instruction** | **Opcode/Function** | | **Syntax** | **Operation** |
| slt | 101010 | 42 | ArithLog | $d = ($s < $t) |
| sltu | 101001 | 41 | ArithLog | $d = ($s < $t) |
| slti | 001010 | 10 | ArithLogI | $t = ($s < SE(i)) |
| sltiu | 001001 | 9 | ArithLogI | $t = ($s < SE(i)) |
| **Branch Instructions** | | | | |
| **Instruction** | **Opcode/Function** | | **Syntax** | **Operation** |
| beq | 000100 | 4 | Branch | if ($s == $t) pc += i << 2 |
| bgtz | 000111 | 7 | BranchZ | if ($s > 0) pc += i << 2 |
| blez | 000110 | 6 | BranchZ | if ($s <= 0) pc += i << 2 |
| bne | 000101 | 5 | Branch | if ($s != $t) pc += i << 2 |
| **Jump Instructions** | | | | |
| **Instruction** | **Opcode/Function** | | **Syntax** | **Operation** |
| j | 000010 | 2 | Jump | pc += i << 2 |
| jal | 000011 | 3 | Jump | $31 = pc; pc += i << 2 |
| jalr | 001001 | 9 | JumpR | $31 = pc; pc = $s |
| jr | 001000 | 8 | JumpR | pc = $s |
| **Load Instructions** | | | | |
| **Instruction** | **Opcode/Function** | | **Syntax** | **Operation** |
| lb | 100000 | 32 | LoadStore | $t = SE (MEM [$s + i]:1) |
| lbu | 100100 | 36 | LoadStore | $t = ZE (MEM [$s + i]:1) |
| lh | 100001 | 33 | LoadStore | $t = SE (MEM [$s + i]:2) |
| lhu | 100101 | 37 | LoadStore | $t = ZE (MEM [$s + i]:2) |
| lw | 100011 | 35 | LoadStore | $t = MEM [$s + i]:4 |
| **Store Instructions** | | | | |
| **Instruction** | **Opcode/Function** | | **Syntax** | **Operation** |
| sb | 101000 | 40 | LoadStore | MEM [$s + i]:1 = LB ($t) |
| sh | 101001 | 41 | LoadStore | MEM [$s + i]:2 = LH ($t) |
| sw | 101011 | 43 | LoadStore | MEM [$s + i]:4 = $t |
| **Data Movement Instructions** | | | | |
| **Instruction** | **Opcode/Function** | | **Syntax** | **Operation** |
| mfhi | 010000 | 16 | MoveFrom | $d = hi |
| mflo | 010010 | 18 | MoveFrom | $d = lo |
| mthi | 010001 | 17 | MoveTo | hi = $s |
| mtlo | 010011 | 19 | MoveTo | lo = $s |
| **Exception and Interrupt Instructions** | | | | |
| **Instruction** | **Opcode/Function** | | **Syntax** | **Operation** |
| trap | 011010 | 26 | Trap | Dependent on operating system; different values for immed26 specify different operations. See the [list of traps](https://www.student.cs.uwaterloo.ca/~isg/res/mips/traps) for information on what the different trap codes do. |

**Opcode Map**

**ROOT**

Table of opcodes for all instructions:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **000** | **001** | **010** | **011** | **100** | **101** | **110** | **111** |
| **000** | REG |  | j | jal | beq | bne | blez | bgtz |
| **001** | addi | addiu | slti | sltiu | andi | ori | xori |  |
| **010** |  |  |  |  |  |  |  |  |
| **011** | llo | lhi | trap |  |  |  |  |  |
| **100** | lb | lh |  | lw | lbu | lhu |  |  |
| **101** | sb | sh |  | sw |  |  |  |  |
| **110** |  |  |  |  |  |  |  |  |
| **111** |  |  |  |  |  |  |  |  |

**REG**

Table of function codes for register-format instructions:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **000** | **001** | **010** | **011** | **100** | **101** | **110** | **111** |
| **000** | sll |  | srl | sra | sllv |  | srlv | srav |
| **001** | jr | jalr |  |  |  |  |  |  |
| **010** | mfhi | mthi | mflo | mtlo |  |  |  |  |
| **011** | mult | multu | div | divu |  |  |  |  |
| **100** | add | addu | sub | subu | and | or | xor | nor |
| **101** |  |  | slt | sltu |  |  |  |  |
| **110** |  |  |  |  |  |  |  |  |
| **111** |  |  |  |  |  |  |  |  |

|  |  |  |
| --- | --- | --- |
| **Register Name** | **Common Name** | **Description** |
| $0 | zero | Always has the value 0. Any writes to this register are ignored. |
| $1 | at | Assembler temporary. |
| $2-$3 | v0-v1 | Function result registers.  Functions return integer results in v0, and 64-bit integer results in v0 and v1 when using 32-bit registers.  In cases where floating-point hardware is not present, or when compiler options enable floating-point emulation, functions return single precision floating-point results in v0 and double precision floating-point results in v0 and v1 when using 32-bit registers.  v0 and v1 can be temporary registers.  Not preserved across function calls. |
| $4-$7 | a0-a3 | Function argument registers that hold the first four words of integer type arguments.  Functions use these registers to hold floating-point arguments.  When floating-point hardware is not present, or compiler options enable floating-point emulation, functions use a0 to hold the first single precision floating-point argument and a1 to hold the second single precision floating-point argument.  Functions use a0-a1 for the first double precision floating-point argument, and a2-a3 to hold the second double precision floating-point argument.  Not preserved across function calls. |
| $8-$15,  $24-$25 | t0-t9 | Temporary registers you can use as you want. Not preserved across function calls. |
| $16-$23, $30 | s0-s8 | Saved registers to use freely.  Preserved across function calls. These registers must be saved before use by the called function. |
| $26-$27 | k0-k1 | Reserved for use by the operating system kernel and for exception return. |
| $28 | gp | Global pointer.  Not used in Windows CE and may be used as save register for called functions. |
| $29 | sp | Stack pointer. |
| $31 | ra | Return address register, saved by the calling function. Available for use after saving. |
| $f0 | n/a | Function return register used to return float and double values from function calls. |
| ($f12, $f13)  and  ($f14, $f15) | n/a | Two pairs of registers used to pass float and double valued parameters to functions.  Pairs of registers are parenthesized because they have to pass double values.  To pass float values, only $f12 and $f14 are used. |

The following list contains additional information on floating-point registers:

* In MIPS ISAs I, II, and in MIPS III and up ISAs running in 32-bit mode, only $f4, $f6, $f8, $f10, $f16, and $f18 temporary registers are available.   
  When manipulating these registers with double precision instructions, the high-order 32-bits are in the implied odd register. The odd registers are not directly accessible.
* Permanent registers $f20, $f22, $f24, $f26, $f28, and $f30 are registers where values are preserved across function calls.
* In MIPS architectures III and up running in 64-bit mode, the following registers are also available as temporary registers: $f1, $f3, $f5, $f7, $f9, $f11, $f17, $f19, $f21, $f23, $f25, $f27, $f29, $f31.