Category	Name	Instruction syntax	Meaning	Format/opcode/funct			Notes/Encoding
Arithmetic	Add	`add $d,$s,$t`	`$d = $s + $t`	R	0	20_{16}	adds two registers, executes a trap on overflow `000000ss ssstttt ddddd--- --100000`
	Add unsigned	`addu $d,$s,$t`	`$d = $s + $t`	R	0	21_{16}	as above but ignores an overflow `000000ss ssstttt ddddd--- --100001`
	Subtract	`sub $d,$s,$t`	`$d = $s - $t`	R	0	22_{16}	subtracts two registers, executes a trap on overflow `000000ss ssstttt ddddd--- --100010`
	Subtract unsigned	`subu $d,$s,$t`	`$d = $s - $t`	R	0	23_{16}	as above but ignores an overflow `000000ss ssstttt ddddd000 00100011`
	Add immediate	`addi $t,$s,C`	`$t = $s + C (signed)`	I	8_{16}	-	Used to add sign-extended constants (and also to copy one register to another: addi $1, $2, 0), executes a trap on overflow `001000ss ssstttt CCCCCCCC CCCCCCCC`
	Add immediate unsigned	`addiu $t,$s,C`	`$t = $s + C (unsigned)`	I	9_{16}	-	as above but ignores an overflow `001001ss ssstttt CCCCCCCC CCCCCCCC`
							Multiplies two registers and puts the 64-bit result in two special memory spots - LO and HI. Alternatively, one could say the result of this operation is:

	Multiply	`mult $s,$t`	`LO = (($s * $t) << 32) >> 32;` `HI = ($s * $t) >> 32;`	R	0	18_{16}	`(int HI,int LO) = (64-bit) $s * $t` . See mfhi and mflo for accessing LO and HI regs.
	Multiply unsigned	`multu $s,$t`	`LO = (($s * $t) << 32) >> 32;` `HI = ($s * $t) >> 32;`	R	0	19_{16}	Multiplies two registers and puts the 64-bit result in two special memory spots - LO and HI. Alternatively, one could say the result of this operation is: `(int HI,int LO) = (64-bit) $s * $t` . See mfhi and mflo for accessing LO and HI regs.
	Divide	`div $s, $t`	`LO = $s / $t  HI = $s % $t`	R	0	$1A_{16}$	Divides two registers and puts the 32-bit integer result in LO and the remainder in HI.[36]
	Divide unsigned	`divu $s, $t`	`LO = $s / $t  HI = $s % $t`	R	0	$1B_{16}$	Divides two registers and puts the 32-bit integer result in LO and the remainder in HI.
	Load word	`lw $t,C($s)`	`$t = Memory[$s + C]`	I	23_{16}	-	loads the word stored from: MEM[$s+C] and the following 3 bytes.
	Load halfword	`lh $t,C($s)`	`$t = Memory[$s + C] (signed)`	I	21_{16}	-	loads the halfword stored from: MEM[$s+C] and the following byte. Sign is extended to width of register.
	Load halfword unsigned	`lhu $t,C($s)`	`$t = Memory[$s + C] (unsigned)`	I	25_{16}	-	As above without sign extension.
	Load byte	`lb $t,C($s)`	`$t = Memory[$s + C] (signed)`	I	20_{16}	-	loads the byte stored from: MEM[$s+C].
	Load byte unsigned	`lbu $t,C($s)`	`$t = Memory[$s + C] (unsigned)`	I	24_{16}	-	As above without sign extension.
							stores a word into: MEM[$s+C]

Data Transfer	Store word	`sw $t,C($s)`	`Memory[$s + C] = $t`	I	$2B_{16}$	-	and the following 3 bytes. The order of the operands is a large source of confusion.
	Store half	`sh $t,C($s)`	`Memory[$s + C] = $t`	I	29_{16}	-	stores the least-significant 16-bit of a register (a halfword) into: MEM[$s+C].
	Store byte	`sb $t,C($s)`	`Memory[$s + C] = $t`	I	28_{16}	-	stores the least-significant 8-bit of a register (a byte) into: MEM[$s+C].
	Load upper immediate	`lui $t,C`	`$t = C << 16`	I	F_{16}	-	loads a 16-bit immediate operand into the upper 16-bits of the register specified. Maximum value of constant is 2^{16}-1
	Move from high	`mfhi $d`	`$d = HI`	R	0	10_{16}	Moves a value from HI to a register. Do not use a multiply or a divide instruction within two instructions of mfhi (that action is undefined because of the MIPS pipeline).
	Move from low	`mflo $d`	`$d = LO`	R	0	12_{16}	Moves a value from LO to a register. Do not use a multiply or a divide instruction within two instructions of mflo (that action is undefined because of the MIPS pipeline).
	Move from Control Register	`mfcZ $t, $d`	`$t = Coprocessor[Z].ControlRegister[$d]`	R	0		Moves a 4 byte value from Coprocessor Z Control register to a general purpose register. Sign extension.
	Move to Control Register	`mtcZ $t, $d`	`Coprocessor[Z].ControlRegister[$d] = $t`	R	0		Moves a 4 byte value from a general purpose register to a Coprocessor Z Control register. Sign extension.
	And	`and $d,$s,$t`	`$d = $s & $t`	R	0	24_{16}	Bitwise and `000000ss ssstttt ddddd--- --100100`

Logical	And immediate	`andi $t,$s,C`	`$t = $s & C`	I	C_{16}	-	Leftmost 16 bits are padded with 0s `001100ss ssstttt CCCCCCCC CCCCCCCC`
	Or	`or $d,$s,$t`	`$d = $s | $t`	R	0	25_{16}	Bitwise or
	Or immediate	`ori $t,$s,C`	`$t = $s | C`	I	D_{16}	-	Leftmost 16 bits are padded with 0s
	Exclusive or	`xor $d,$s,$t`	`$d = $s ^ $t`	R	0	26_{16}	
	Nor	`nor $d,$s,$t`	`$d = ~ ($s | $t)`	R	0	27_{16}	Bitwise nor
	Set on less than	`slt $d,$s,$t`	`$d = ($s < $t)`	R	0	$2A_{16}$	Tests if one register is less than another.
	Set on less than unsigned	`sltu $d,$s,$t`	`$d = ($s < $t)`	R	0	$2B_{16}$	Tests if unsigned integer in one register is less than another.
	Set on less than immediate	`slti $t,$s,C`	`$t = ($s < C)`	I	A_{16}	-	Tests if one register is less than a constant.
Bitwise Shift	Shift left logical immediate	`sll $d,$t,shamt`	`$d = $t << shamt`	R	0	0	shifts shamt number of bits to the left (multiplies by $2^{\mathtt{shamt}}$)
	Shift right logical immediate	`srl $d,$t,shamt`	`$d = $t >> shamt`	R	0	2_{16}	shifts shamt number of bits to the right - zeros are shifted in (divides by $2^{\mathtt{shamt}}$). Note that this instruction only works as division of a two's complement number if the value is positive.
	Shift right arithmetic immediate	`sra $d,$t,shamt`	$\mathtt{\$d = \$t >> shamt} + \left(\sum_{n=1}^{\mathtt{shamt}} 2^{32-n}\right) \cdot (\mathtt{\$t >> 31})$	R	0	3_{16}	shifts shamt number of bits - the sign bit is shifted in (divides a positive or even 2's complement number by $2^{\mathtt{shamt}}$)
	Shift left logical	`sllv $d,$t,$s`	`$d = $t << $s`	R	0	4_{16}	shifts `$s` number of bits to the left (multiplies by $2^{\$s}$)

	Shift right logical	`srlv $d,$t,$s`	`$d = $t >> $s`	R	0	6_{16}	shifts `$s` number of bits to the right - zeros are shifted in (divides by $2^{\$s}$). Note that this instruction only works as division of a two's complement number if the value is positive.
	Shift right arithmetic	`srav $d,$t,$s`	$\$d = \$t >> \$s + \left(\sum_{n=1}^{\$s} 2^{32-n}\right) \cdot (\$t >> 31)$	R	0	7_{16}	shifts `$s` number of bits - the sign bit is shifted in (divides a positive or even 2's complement number by $2^{\$s}$)
Conditional branch	Branch on equal	`beq $s,$t,C`	`if ($s == $t) go to PC+4+4*C`	I	4_{16}	-	Goes to the instruction at the specified address if two registers are equal. `000100ss ssssssss CCCCCCCC CCCCCCCC`
	Branch on not equal	`bne $s,$t,C`	`if ($s != $t) go to PC+4+4*C`	I	5_{16}	-	Goes to the instruction at the specified address if two registers are *not* equal.
Unconditional jump	Jump	`j C`	`PC = PC+4[31:28] . C*4`	J	2_{16}	-	Unconditionally jumps to the instruction at the specified address.
	Jump register	`jr $s`	goto address `$s`	R	0	8_{16}	Jumps to the address contained in the specified register
	Jump and link	`jal C`	`$31 = PC + 4; PC = PC+4[31:28] . C*4`	J	3_{16}	-	For procedure call - used to call a subroutine, $31 holds the return address; returning from a subroutine is done by: jr $31. Return address is PC + 8, not PC + 4 due to the use of a branch delay slot which forces the instruction after the jump to be executed

Note: In MIPS assembly code, the offset for branching instructions can be represented by a label elsewhere in the code.