The MIPS R2000 Instruction Set

Arithmetic and Logical Instructions

In all instructions below, Src2 can either be a register or an immediate value (a 16 bit integer). The immediate forms of the instructions are only included for reference. The assembler will translate the more general form of an instruction (e.g., add) into the immediate form (e.g., addi) if the second argument is constant. Instructions marked with a dagger (†) are pseudoinstructions.

`abs Rdest, Rsrc` — *Absolute Value* †

Put the absolute value of the integer from register Rsrc in register Rdest.

`add Rdest, Rsrc1, Src2` — *Addition (with overflow)*
`addi Rdest, Rsrc1, Imm` — *Addition Immediate (with overflow)*
`addu Rdest, Rsrc1, Src2` — *Addition (without overflow)*
`addiu Rdest, Rsrc1, Imm` — *Addition Immediate (without overflow)*

Put the sum of the integers from register Rsrc1 and Src2 (or Imm) into register Rdest.

`and Rdest, Rsrc1, Src2` — *AND*
`andi Rdest, Rsrc1, Imm` — *AND Immediate*

Put the logical AND of the integers from register Rsrc1 and Src2 (or Imm) into register Rdest.

`div Rsrc1, Rsrc2` — *Divide (with overflow)*
`divu Rsrc1, Rsrc2` — *Divide (without overflow)*

Divide the contents of the two registers. Leave the quotient in register lo and the remainder in register hi. Note that if an operand is negative, the remainder is unspecified by the MIPS architecture and depends on the conventions of the machine on which SPIM is run.

`div Rdest, Rsrc1, Src2` — *Divide (with overflow)* †
`divu Rdest, Rsrc1, Src2` — *Divide (without overflow)* †

Put the quotient of the integers from register Rsrc1 and Src2 into register Rdest.

`mul Rdest, Rsrc1, Src2` — *Multiply (without overflow)* †
`mulo Rdest, Rsrc1, Src2` — *Multiply (with overflow)* †
`mulou Rdest, Rsrc1, Src2` — *Unsigned Multiply (with overflow)* †

Put the product of the integers from register Rsrc1 and Src2 into register Rdest.

`mult Rsrc1, Rsrc2` — *Multiply*
`multu Rsrc1, Rsrc2` — *Unsigned Multiply*

Multiply the contents of the two registers. Leave the low-order word of the product in register lo and the high-word in register hi.

`neg Rdest, Rsrc` *Negate Value (with overflow)* †

`negu Rdest, Rsrc` *Negate Value (without overflow)* †

Put the negative of the integer from register Rsrc into register Rdest.

`nor Rdest, Rsrc1, Src2` *NOR*

Put the logical NOR of the integers from register Rsrc1 and Src2 into register Rdest.

`not Rdest, Rsrc` *NOT* †

Put the bitwise logical negation of the integer from register Rsrc into register Rdest.

`or Rdest, Rsrc1, Src2` *OR*

`ori Rdest, Rsrc1, Imm` *OR Immediate*

Put the logical OR of the integers from register Rsrc1 and Src2 (or Imm) into register Rdest.

`rem Rdest, Rsrc1, Src2` *Remainder* †

`remu Rdest, Rsrc1, Src2` *Unsigned Remainder* †

Put the remainder from dividing the integer in register Rsrc1 by the integer in Src2 into register Rdest. Note that if an operand is negative, the remainder is unspecified by the MIPS architecture and depends on the conventions of the machine on which SPIM is run.

`rol Rdest, Rsrc1, Src2` *Rotate Left* †

`ror Rdest, Rsrc1, Src2` *Rotate Right* †

Rotate the contents of register Rsrc1 left (right) by the distance indicated by Src2 and put the result in register Rdest.

`sll Rdest, Rsrc1, Src2` *Shift Left Logical*

`sllv Rdest, Rsrc1, Rsrc2` *Shift Left Logical Variable*

`sra Rdest, Rsrc1, Src2` *Shift Right Arithmetic*

`srav Rdest, Rsrc1, Rsrc2` *Shift Right Arithmetic Variable*

`srl Rdest, Rsrc1, Src2` *Shift Right Logical*

`srlv Rdest, Rsrc1, Rsrc2` *Shift Right Logical Variable*

Shift the contents of register Rsrc1 left (right) by the distance indicated by Src2 (Rsrc2) and put the result in register Rdest.

`sub Rdest, Rsrc1, Src2` *Subtract (with overflow)*

`subu Rdest, Rsrc1, Src2` *Subtract (without overflow)*

Put the difference of the integers from register Rsrc1 and Src2 into register Rdest.

`xor Rdest, Rsrc1, Src2` *XOR*

`xori Rdest, Rsrc1, Imm` *XOR Immediate*

Put the logical XOR of the integers from register Rsrc1 and Src2 (or Imm) into register Rdest.

Constant-Manipulating Instructions

`li Rdest, Imm` *Load Immediate* †

Move the immediate imm into register Rdest.

`lui Rdest, Imm` *Load Upper Immediate*

Load the lower halfword of the immediate imm into the upper halfword of register Rdest. The lower bits of the register are set to 0.

Comparison Instructions

In all instructions below, Src2 can either be a register or an immediate value (a 16 bit integer).

`seq Rdest, Rsrc1, Src2` *Set Equal* †

Set register Rdest to 1 if register Rsrc1 equals Src2 and to be 0 otherwise.

`sge Rdest, Rsrc1, Src2` *Set Greater Than Equal* †

`sgeu Rdest, Rsrc1, Src2` *Set Greater Than Equal Unsigned* †

Set register Rdest to 1 if register Rsrc1 is greater than or equal to Src2 and to 0 otherwise.

`sgt Rdest, Rsrc1, Src2` *Set Greater Than* †

`sgtu Rdest, Rsrc1, Src2` *Set Greater Than Unsigned* †

Set register Rdest to 1 if register Rsrc1 is greater than Src2 and to 0 otherwise.

`sle Rdest, Rsrc1, Src2` *Set Less Than Equal* †

`sleu Rdest, Rsrc1, Src2` *Set Less Than Equal Unsigned* †

Set register Rdest to 1 if register Rsrc1 is less than or equal to Src2 and to 0 otherwise.

`slt Rdest, Rsrc1, Src2` *Set Less Than*

`slti Rdest, Rsrc1, Imm` *Set Less Than Immediate*

`sltu Rdest, Rsrc1, Src2` *Set Less Than Unsigned*

`sltiu Rdest, Rsrc1, Imm` *Set Less Than Unsigned Immediate*

Set register Rdest to 1 if register Rsrc1 is less than Src2 (or Imm) and to 0 otherwise.

`sne Rdest, Rsrc1, Src2` *Set Not Equal* †

Set register Rdest to 1 if register Rsrc1 is not equal to Src2 and to 0 otherwise.

Branch and Jump Instructions

In all instructions below, Src2 can either be a register or an immediate value (integer). Branch instructions use a signed 16-bit offset field; hence they can jump 2^{15} - 1 instructions (not bytes) forward or 2^{15} instructions backwards. The jump instruction contains a 26 bit address field.

`b label` *Branch instruction* †

Unconditionally branch to the instruction at the label.

`bal label` *Branch And Link* †

Unconditionally branch to the instruction at the label. Save the address of the next instruction in register `$ra`.

`bczt label` *Branch Coprocessor* **z** *True*

`bczf label` *Branch Coprocessor* **z** *False*

Conditionally branch to the instruction at the label if coprocessor z condition flag is true (false).

`beq Rsrc1, Src2, label` *Branch on Equal*

Conditionally branch to the instruction at the label if the contents of register Rsrc1 equals Src2.

`beqz Rsrc, label` *Branch on Equal Zero* †

Conditionally branch to the instruction at the label if the contents of Rsrc equals 0.

`bge Rsrc1, Src2, label` *Branch on Greater Than Equal* †

`bgeu Rsrc1, Src2, label` *Branch on GTE Unsigned* †

Conditionally branch to the instruction at the label if the contents of register Rsrc1 are greater than or equal to Src2.

`bgez Rsrc, label` *Branch on Greater Than Equal Zero*

Conditionally branch to the instruction at the label if the contents of Rsrc are greater than or equal to 0.

`bgezal Rsrc, label` *Branch on Greater Than Equal Zero And Link*

Conditionally branch to the instruction at the label if the contents of Rsrc are greater than or equal to 0. Save the address of the next instruction in register `$ra`.

`bgt Rsrc1, Src2, label` *Branch on Greater Than* †

`bgtu Rsrc1, Src2, label` *Branch on Greater Than Unsigned* †

Conditionally branch to the instruction at the label if the contents of register Rsrc1 are greater than Src2.

`bgtz Rsrc, label` *Branch on Greater Than Zero*

Conditionally branch to the instruction at the label if the contents of Rsrc are greater than 0.

`ble Rsrc1, Src2, label` *Branch on Less Than Equal* †

`bleu Rsrc1, Src2, label` *Branch on LTE Unsigned* †

Conditionally branch to the instruction at the label if the contents of register Rsrc1 are less than or equal to Src2.

`blez Rsrc, label` *Branch on Less Than Equal Zero*

Conditionally branch to the instruction at the label if the contents of Rsrc are less than or equal to 0.

`bgezal Rsrc, label` *Branch on Greater Than Equal Zero And Link*

`bltzal Rsrc, label` *Branch on Less Than And Link*

Conditionally branch to the instruction at the label if the contents of Rsrc are greater or equal to 0 or less than 0, respectively. Save the address of the next instruction in register 31.

`blt Rsrc1, Src2, label` *Branch on Less Than* †

`bltu Rsrc1, Src2, label` *Branch on Less Than Unsigned* †

Conditionally branch to the instruction at the label if the contents of register Rsrc1 are less than Src2.

`bltz Rsrc, label` *Branch on Less Than Zero*

Conditionally branch to the instruction at the label if the contents of Rsrc are less than 0.

`bne Rsrc1, Src2, label` *Branch on Not Equal*

Conditionally branch to the instruction at the label if the contents of register Rsrc1 are not equal to Src2.

`bnez Rsrc, label` *Branch on Not Equal Zero* †

Conditionally branch to the instruction at the label if the contents of Rsrc are not equal to 0.

`j label` *Jump*

Unconditionally jump to the instruction at the label.

`jal label` *Jump and Link*

`jalr Rsrc` *Jump and Link Register*

Unconditionally jump to the instruction at the label or whose address is in register Rsrc. Save the address of the next instruction in register `$ra`.

`jr Rsrc` *Jump Register*

Unconditionally jump to the instruction whose address is in register Rsrc.

Load Instructions

`la Rdest, address` *Load Address* †

Load computed address, not the contents of the location, into register Rdest.

`lb` Rdest, address *Load Byte*

`lbu` Rdest, address *Load Unsigned Byte*

Load the byte at address into register Rdest. The byte is sign-extended by the `lb`, but not the `lbu`, instruction.

`ld` Rdest, address *Load Double-Word* †

Load the 64-bit quantity at address into registers Rdest and Rdest + 1.

`lh` Rdest, address *Load Halfword*

`lhu` Rdest, address *Load Unsigned Halfword*

Load the 16-bit quantity (halfword) at address into register Rdest. The halfword is sign-extended by the `lh`, but not the `lhu`, instruction

`lw` Rdest, address *Load Word* †

Load the 32-bit quantity (word) at address into register Rdest.

`lwcz` Rdest, address *Load Word Coprocessor*

Load the word at address into register Rdest of coprocessor z (0-3).

`lwl` Rdest, address *Load Word Left*

`lwr` Rdest, address *Load Word Right*

Load the left (right) bytes from the word at the possibly-unaligned address into register Rdest.

`ulh` Rdest, address *Unaligned Load Halfword* †

`ulhu` Rdest, address *Unaligned Load Halfword Unsigned* †

Load the 16-bit quantity (halfword) at the possibly-unaligned address into register Rdest. The halfword is sign-extended by the ulh, but not the \bf ulhu, instruction

`ulw` Rdest, address *Unaligned Load Word* †

Load the 32-bit quantity (word) at the possibly-unaligned address into register Rdest.

Store Instructions

`sb` Rsrc, address *Store Byte*

Store the low byte from register Rsrc at address .

`sd` Rsrc, address *Store Double-Word* †

Store the 64-bit quantity in registers Rsrc and Rsrc + 1 at address .

`sh` Rsrc, address *Store Halfword*

Store the low halfword from register Rsrc at address .

`sw Rsrc, address` *Store Word*

Store the word from register Rsrc at address .

`swcz Rsrc, address` *Store Word Coprocessor*

Store the word from register Rsrc of coprocessor z at address .

`swl Rsrc, address` *Store Word Left*

`swr Rsrc, address` *Store Word Right*

Store the left (right) bytes from register Rsrc at the possibly-unaligned address.

`ush Rsrc, address` *Unaligned Store Halfword* †

Store the low halfword from register Rsrc at the possibly-unaligned address.

`usw Rsrc, address` *Unaligned Store Word* †

Store the word from register Rsrc at the possibly-unaligned address.

Data Movement Instructions

`move Rdest, Rsrc` *Move* †

Move the contents of Rsrc to Rdest.

The multiply and divide unit produces its result in two additional registers, hi and lo. These instructions move values to and from these registers. The multiply, divide, and remainder instructions described above are pseudoinstructions that make it appear as if this unit operates on the general registers and detect error conditions such as divide by zero or overflow.

`mfhi Rdest` *Move From hi*

`mflo Rdest` *Move From lo*

Move the contents of the hi (lo) register to register Rdest.

`mthi Rsrc` *Move To hi*

`mtlo Rsrc` *Move To lo*

Move the contents register Rsrc to the hi (lo) register.

Coprocessors have their own register sets. These instructions move values between these registers and the CPU's registers.

`mfcz Rdest, CPsrc` *Move From Coprocessor z*

Move the contents of coprocessor z's register CPsrc to CPU register Rdest.

`mfc1.d Rdest, FRsrc1` *Move Double From Coprocessor 1* †
Move the contents of floating point registers FRsrc1 and FRsrc1 + 1 to CPU registers Rdest and Rdest + 1.

`mtc1.d Rsrc, FRdest1` *Move Double To Coprocessor 1* †
Move the contents of CPU registers Rsrc and Rsrc + 1 to floating point registers FRdest1 and FRdest1 + 1.

`mtcz Rsrc, CPdest` *Move To Coprocessor z*
Move the contents of CPU register Rsrc to coprocessor z's register CPdest.

Floating Point Instructions

The MIPS has a floating point coprocessor (numbered 1) that operates on single precision (32-bit) and double precision (64-bit) floating point numbers. This coprocessor has its own registers, which are numbered `$f0` - `$f31`. Because these registers are only 32-bits wide, two of them are required to hold doubles. To simplify matters, floating point operations only use even-numbered registers - including instructions that operate on single floats.
Values are moved in or out of these registers a word (32-bits) at a time by `lwc1`, `swc1`, `mtc1`, and `mfc1` instructions described above or by the `l.s`, `l.d`, `s.s`, and `s.d` pseudoinstructions described below. The flag set by floating point comparison operations is read by the CPU with its `bc1t` and `bc1f` instructions. In all instructions below, FRdest, FRsrc1, FRsrc2, and FRsrc are floating point registers (e.g., `$f2`).

`abs.d FRdest, FRsrc` *Floating Point Absolute Value Double*
`abs.s FRdest, FRsrc` *Floating Point Absolute Value Single*
Compute the absolute value of the floating float double (single) in register FRsrc and put it in register FRdest.

`add.d FRdest, FRsrc1, FRsrc2` *Floating Point Addition Double*
`add.s FRdest, FRsrc1, FRsrc2` *Floating Point Addition Single*
Compute the sum of the floating float doubles (singles) in registers FRsrc1 and FRsrc2 and put it in register FRdest.

`c.eq.d FRsrc1, FRsrc2` *Compare Equal Double*
`c.eq.s FRsrc1, FRsrc2` *Compare Equal Single*
Compare the floating point double (single) in register FRsrc1 against the one in FRsrc2 and set the floating point condition flag true if they are equal.

`c.le.d` FRsrc1, FRsrc2 *Compare Less Than Equal Double (w/ exept.)*
`c.le.s` FRsrc1, FRsrc2 *Compare Less Than Equal Single (w/ exept.)*
Compare the floating point double (single) in register FRsrc1 against the one in FRsrc2 and set the floating point condition flag true if the first is less than or equal to the second. Cause exeption 14 if FRsrc1 or FRsrc2 contains an invalid floating point number.

`c.lt.d` FRsrc1, FRsrc2 *Compare Less Than Double (w/ exept.)*
`c.lt.s` FRsrc1, FRsrc2 *Compare Less Than Single (w/ exept.)*
Compare the floating point double (single) in register FRsrc1 against the one in FRsrc2 and set the condition flag true if the first is less than the second. Cause exeption 14 if FRsrc1 or FRsrc2 contains an invalid floating point number.

`c.nge.d` FRsrc1, FRsrc2 *Compare Not Greater Than Equal Double (w/ exept.)*
`c.nge.s` FRsrc1, FRsrc2 *Compare Not Greater Than Equal Single (w/ exept.)*
Compare the floating point double (single) in register FRsrc1 against the one in FRsrc2 and set the condition flag true if the first is not greater or equal than the second or if FRsrc1 or FRsrc2 is an invalid floating point number. Cause exeption 14 if FRsrc1 or FRsrc2 contains an invalid floating point number.

`c.ngle.d` FRsrc1, FRsrc2 *Compare Not Greater or Less Than Equal Double (w/ exept.)*
`c.ngle.s` FRsrc1, FRsrc2 *Compare Not Greater or Less Than Equal Single (w/ exept.)*
Set the condition flag true if the floating point double (single) in register FRsrc1 or FRsrc2 contains an invalid floating point number. Cause exeption 14 if FRsrc1 or FRsrc2 contains an invalid floating point number.

`c.ngl.d` FRsrc1, FRsrc2 *Compare Not Greater or Less Than Double (w/ exept.)*
`c.ngl.s` FRsrc1, FRsrc2 *Compare Not Greater or Less Than Single (w/ exept.)*
Compare the floating point double (single) in register FRsrc1 against the one in FRsrc2 and set the condition flag true if the first is not greater or less than the second or if FRsrc1 or FRsrc2 contains an invalid floating point number. Cause exeption 14 if FRsrc1 or FRsrc2 contains an invalid floating point number.

`c.ngt.d` FRsrc1, FRsrc2 *Compare Not Greater Than Double (w/ exept.)*
`c.ngt.s` FRsrc1, FRsrc2 *Compare Not Greater Than Single (w/ exept.)*
Compare the floating point double (single) in register FRsrc1 against the one in FRsrc2 and set the condition flag true if the first is not greater than the second or if FRsrc1 or FRsrc2 contains an invalid floating point number. Cause exeption 14 if FRsrc1 or FRsrc2 contains an invalid floating point number.

`c.ole.d` FRsrc1, FRsrc2 *Compare Less Than Equal Double*
`c.ole.s` FRsrc1, FRsrc2 *Compare Less Than Equal Single*
Compare the floating point double (single) in register FRsrc1 against the one in FRsrc2 and set the floating point condition flag true if the first is less than or equal to the second.

`c.olt.d` FRsrc1, FRsrc2 *Compare Less Than Double*
`c.olt.s` FRsrc1, FRsrc2 *Compare Less Than Single*
Compare the floating point double (single) in register FRsrc1 against the one in FRsrc2 and set the condition flag true if the first is less than the second.

`c.seq.d` FRsrc1, FRsrc2 *Compare Equal Double (w/ exept.)*
`c.seq.s` FRsrc1, FRsrc2 *Compare Equal Single (w/ exept.)*
Compare the floating point double (single) in register FRsrc1 against the one in FRsrc2 and set the floating point condition flag true if they are equal. Cause exeption 14 if FRsrc1 or FRsrc2 contains an invalid floating point number.

`c.sf.d` FRsrc1, FRsrc2 *Set False Double (w/ exept.)*
`c.sf.s` FRsrc1, FRsrc2 *Set False Single (w/ exept.)*
Set the condition flag false. Cause exeption 14 if FRsrc1 or FRsrc2 contains an invalid floating point number.

`c.f.d` FRsrc1, FRsrc2 *Set False Double*
`c.f.s` FRsrc1, FRsrc2 *Set False Single*
Set the condition flag false. The contents of FRsrc1 or FRsrc2 are meaningless.

`c.ueq.d` FRsrc1, FRsrc2 *Compare Unordered or Equal Double*
`c.ueq.s` FRsrc1, FRsrc2 *Compare Unordered or Equal Single*
Compare the floating point double (single) in register FRsrc1 against the one in FRsrc2 and set the floating point condition flag true if they are equal or if FRsrc1 or FRsrc2 contains an invalid floating point number.

`c.ule.d` FRsrc1, FRsrc2 *Compare Unordered or Less Than Equal Double*
`c.ule.s` FRsrc1, FRsrc2 *Compare Unordered or Less Than Equal Single*
Compare the floating point double (single) in register FRsrc1 against the one in FRsrc2 and set the floating point condition flag true if the first is less than or equal to the second or if FRsrc1 or FRsrc2 contains an invalid floating point number.

`c.ult.d` FRsrc1, FRsrc2 *Compare Unordered or Less Than Double*
`c.ult.s` FRsrc1, FRsrc2 *Compare Unordered or Less Than Single*
Compare the floating point double (single) in register FRsrc1 against the one in FRsrc2 and set the condition flag true if the first is less than the second or if FRsrc1 or FRsrc2 contains an invalid floating point number.

`c.un.d FRsrc1, FRsrc2` *Compare Unordered Double*
`c.un.s FRsrc1, FRsrc2` *Compare Unordered Single*
Set the condition flag true if FRsrc1 or FRsrc2 contains an invalid floating point number.

`cvt.d.s FRdest, FRsrc` *Convert Single to Double*
`cvt.d.w FRdest, FRsrc` *Convert Integer to Double*
Convert the single precision floating point number or integer in register FRsrc to a double precision number and put it in register FRdest.

`cvt.s.d FRdest, FRsrc` *Convert Double to Single*
`cvt.s.w FRdest, FRsrc` *Convert Integer to Single*
Convert the double precision floating point number or integer in register FRsrc to a single precision number and put it in register FRdest.

`cvt.w.d FRdest, FRsrc` *Convert Double to Integer*
`cvt.w.s FRdest, FRsrc` *Convert Single to Integer*
Convert the double or single precision floating point number in register FRsrc to an integer and put it in register FRdest.

`div.d FRdest, FRsrc1, FRsrc2` *Floating Point Divide Double*
`div.s FRdest, FRsrc1, FRsrc2` *Floating Point Divide Single*
Compute the quotient of the floating float doubles (singles) in registers FRsrc1 and FRsrc2 and put it in register FRdest.

`l.d FRdest, address` *Load Floating Point Double* †
`l.s FRdest, address` *Load Floating Point Single* †
Load the floating float double (single) at address into register FRdest.

`li.d FRdest, Imm` *Load Floating Point Double Immediate* †
`li.s FRdest, Imm` *Load Floating Point Single Immediate* †
Load the floating float double (single) immediate Imm (e.g. 3.141) into register FRdest.

`mov.d FRdest, FRsrc` *Move Floating Point Double*
`mov.s FRdest, FRsrc` *Move Floating Point Single*
Move the floating float double (single) from register FRsrc to register FRdest.

`mul.d FRdest, FRsrc1, FRsrc2` *Floating Point Multiply Double*
`mul.s FRdest, FRsrc1, FRsrc2` *Floating Point Multiply Single*
Compute the product of the floating float doubles (singles) in registers FRsrc1 and FRsrc2 and put it in register FRdest.

`neg.d` FRdest, FRsrc *Negate Double*

`neg.s` FRdest, FRsrc *Negate Single*

Negate the floating point double (single) in register FRsrc and put it in register FRdest.

`s.d` FRdest, address *Store Floating Point Double* †

`s.s` FRdest, address *Store Floating Point Single* †

Store the floating float double (single) in register FRdest at address.

`sub.d` FRdest, FRsrc1, FRsrc2 *Floating Point Subtract Double*

`sub.s` FRdest, FRsrc1, FRsrc2 *Floating Point Subtract Single*

Compute the difference of the floating float doubles (singles) in registers FRsrc1 and FRsrc2 and put it in register FRdest.

Exception and Trap Instructions

`rfe` *Return From Exception*

Restore the Status register.

`syscall` *System Call*

Register `$v0` contains the number of the system call provided by SPIM.

`break n` *Break*

Cause exception n. Exception 1 is reserved for the debugger.

`nop` *No operation* †

Do nothing.