MIPS Instruction Set (last revision: 11/05/2002)

Instruction	**Example**	**Meaning**	**Comments**
Arithmetic			
add	`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`	overflow detected
subtract	`sub $s1,$s2,$s3`	`$s1 = $s2 - $s3`	overflow detected
add imm	`addi $s1,$s2,100`	`$s1 = $s2 + 100`	overflow detected
add uns	`addu $s1,$s2,$s3`	`$s1 = $s2 + $s3`	overflow *not* detected
subtract uns	`subu $s1,$s2,$s3`	`$s1 = $s2 - $s3`	overflow *not* detected
add uns imm	`addiu $s1,$s2,$s3`	`$s1 = $s2 + 100`	overflow *not* detected
multiply	`mult $s2,$s3`	`Hi,Lo = $s2 × $s3`	64-bit signed product
multiply uns	`mult $s2,$s3`	`Hi,Lo = $s2 × $s3`	64-bit unsigned product
divide	`div $s2,$s3`	`Lo = $s2/$s3; Hi = $s2 mod $s3`	Lo,Hi = quotient,remainder
divide uns	`divu $s2,$s3`	`Lo = $s2/$s3; Hi = $s2 mod $s3`	unsigned Lo,Hi
move from Hi	`mfhi $s1`	`$s1 = Hi`	copy Hi
move from Lo	`mflo $s1`	`$s1 = Lo`	copy Lo
Logical			
and	`and $s1,$s2,$s3`	`$s1 = $s2 & $s3`	logical AND
or	`or $s1,$s2,$s3`	`$s1 = $s2 | $s3`	logical OR
and imm	`andi $s1,$s2,100`	`$s1 = $s2 & 100`	logical AND
or imm	`ori $s1,$s2,100`	`$s1 = $s2 | 100`	logical OR
shift left logical	`sll $s1,$s2,10`	`$s1 = $s2 << 10`	shift left by const
shift right logical	`srl $s1,$s2,10`	`$s1 = $s2 >> 10`	shift right by const
Data transfer			
load word	`lw $s1,100($s2)`	`$s1 = Mem[$s2+100]`	word from mem to reg
store word	`sw $s1,100($s2)`	`Mem[$s2+100] = $s1`	word from reg to mem
load byte	`lb $s1,100($s2)`	`$s1 = Mem[$s2+100]`	sign extended
load byte uns	`lbu $s1,100($s2)`	`$s1 = Mem[$s2+100]`	not sign extended
store byte	`sb $s1,100($s2)`	`Mem[$s2+100] = $s1`	low byte from reg to mem
load upper imm	`lui $s1,100`	`$s1 = 100 *` 2^{16}	load const, upper 16 bits
load imm†	`li $s1,100`	`$s1 = 100`	load const
load address†	`la $s1,L1`	`$s1 = address L1`	load address
move registers†	`move $s1,$s2`	`$s1 = $s2`	
Conditional branch			
branch on equal	`beq $s1,$s2,25`	`if ($s1 == $s2) goto PC + 4 + 4*25`	PC-relative branch
branch on not equal	`bne $s1,$s2,25`	`if ($s1 != $s2) goto PC + 4 + 4*25`	PC-relative branch
branch on greater than†	`bgt $s1,$s2,25`	`if ($s1 > $s2) goto PC + 4 + 4*25`	PC-relative branch
branch on less than†	`blt $s1,$s2,25`	`if ($s1 < $s2) goto PC + 4 + 4*25`	PC-relative branch
branch on greater or eq†	`bge $s1,$s2,25`	`if ($s1 >= $s2) goto PC + 4 + 4*25`	PC-relative branch
branch on less than or eq†	`ble $s1,$s2,25`	`if ($s1 <= $s2) goto PC + 4 + 4*25`	PC-relative branch
set on less than	`slt $s1,$s2,$s3`	`if ($s2<$s3) $s1=1; else $s1 = 0`	two's complement
set on less than imm	`slti $s1,$s2,100`	`if ($s2<100) $s1=1; else $s1 = 0`	two's complement
set on less than uns	`sltu $s1,$s2,$s3`	`if ($s2<$s3) $s1=1; else $s1 = 0`	natural numbers
set on less than imm uns	`sltiu $s1,$s2,100`	`if ($s2<100) $s1=1; else $s1 = 0`	natural numbers
Unconditional jump			
jump	`j 2500`	`go to 4*2500`	jump to target address
jump register	`jr $ra`	`go to addr in $ra`	for switch, proc return
jump and link	`jal 2500`	`$ra = PC + 4; go to 2500*4`	for proc call
Operating system			
syscall	`syscall`	`system call`	see A-48, A-49

uns = Unsigned; *imm* = Immediate; † = Pseudo-instruction

Floating-point

Instruction	**Example**	**Meaning**	**Comments**
Arithmetic			
FP add single	`add.s $f2,$f4,$f6`	`$f2 = $f4 + $f6`	
FP add double	`add.d $f2,$f4,$f6`	`$f2 = $f4 + $f6`	
FP subtract single	`sub.s $f2,$f4,$f6`	`$f2 = $f4 - $f6`	
FP subtract double	`sub.d $f2,$f4,$f6`	`$f2 = $f4 - $f6`	
FP multiply single	`mul.s $f2,$f4,$f6`	`$f2 = $f4 × $f6`	
FP multiply double	`mul.d $f2,$f4,$f6`	`$f2 = $f4 × $f6`	
FP divide single	`div.s $f2,$f4,$f6`	`$f2 = $f4 / $f6`	
FP divide double	`div.d $f2,$f4,$f6`	`$f2 = $f4 / $f6`	
Data transfer and conversion			
load single†	`l.s $f0,`*addr*	`$f0 = Mem[`*addr*`]`	memory (32 bits) to FP reg
load double†	`l.d $f0,`*addr*	`$f0 = Mem[`*addr*`]`	memory (64 bits) to FP reg
load word coproc 1	`lwc1 $f1,100($s2)`	`$f1 = Mem[$s2 + 100]`	32-bit data to FP reg
store word coproc 1	`swc1 $f1,100($s2)`	`Mem[$s2 + 100] = $s2`	32-bit data to memory
convert single to double	`cvt.d.s $f2,$f4`	`$f2 = (double) $f4`	also .d.s, .w.d, etc.; w=int
move FP regs, single	`mov.s $f0,$f2`	`$f0 = $f2`	copy FP reg, single
move FP regs, double	`mov.d $f0,$f2`	`$f0 = $f2`	copy FP reg, double
move to FP, single	`mtc1 $s0,$f0`	`$f0 = $s0`	32-bit value, not converted
move from FP, single	`mfc1 $s0,$f0`	`$s0 = $f0`	32-bit value, not converted
Conditional branch			
branch on FP true	`bc1t 25`	`if (cond == 1) goto PC + 4 + 4*25`	PC relative branch
branch on FP false	`bc1f 25`	`if (cond == 0) goto PC + 4 + 4*25`	PC relative branch
FP compare single	`c.lt.s $f2,$f4`	`if ($f2<$f4) cond=1; else cond=0`	also eq,ne,lt,le,gt,ge; also .d
FP compare double	`c.lt.d $f2,$f4`	`if ($f2<$f4) cond=1; else cond = 0`	also eq,ne,lt,le,gt,ge; also .d

Registers	**Usage**
$zero	constant 0 (enforced by hardware)
$v0, $v1	expression evaluation, function return value
$a0 ... $a3	first four function arguments (rest on stack)
$t0 ... $t9	temporary; caller-saved
$s0 ... $s7	callee-saved
$sp	stack pointer; last location on stack; grow by subtracting
$fp	frame pointer; first word in current stack frame
$ra	return address, written by `jal` instruction
$f0 ... $f31	floating-point registers; for double precision even numbers refer to pairs, i.e.,$f0 uses $f0 *and* $f1.

Compiler directive	**Description**
`.asciiz "`*string*`"`	Store *string* in memory, add null terminator
`.data`	Subsequent items stored in data segment
`.double 3.1415926535898`	Store a double-precision floating-point number in memory
`.float 3.141593`	Store a single-precision floating-point number in memory
`.globl` *symbol*	Declare label *symbol* to be global
`.space` *n*	Allocate *n* bytes of space (must be in data segment)
`.text`	Subsequent items stored in text segment
`.word` *q*	Store 32-bit quantity *q* in memory

Based on Computer Organization and Design (Hennessy and Patterson), p. 274, p. 291 and Appendix A.