

MIPS32® Instruction Set

Quick Reference

R _D	—	DESTINATION REGISTER
RS, R _T	—	SOURCE OPERAND REGISTERS
RA	—	RETURN ADDRESS REGISTER (R31)
PC	—	PROGRAM COUNTER
ACC	—	64-BIT ACCUMULATOR
Lo, Hi	—	ACCUMULATOR LOW (ACC _{31:0}) AND HIGH (ACC _{63:32}) PARTS
±	—	SIGNED OPERAND OR SIGN EXTENSION
∅	—	UNSIGNED OPERAND OR ZERO EXTENSION
::	—	CONCATENATION OF BIT FIELDS
R2	—	MIPS32 RELEASE 2 INSTRUCTION
DOTTED	—	ASSEMBLER PSEUDO-INSTRUCTION

PLEASE REFER TO “*MIPS32 ARCHITECTURE FOR PROGRAMMERS VOLUME II: THE MIPS32 INSTRUCTION SET*” FOR COMPLETE INSTRUCTION SET INFORMATION.

ARITHMETIC OPERATIONS		
ADD	R _D , RS, R _T	R _D = R _S + R _T (OVERFLOW TRAP)
ADDI	R _D , RS, CONST16	R _D = R _S + CONST16 [±] (OVERFLOW TRAP)
ADDIU	R _D , RS, CONST16	R _D = R _S + CONST16 [±]
ADDU	R _D , RS, R _T	R _D = R _S + R _T
CLO	R _D , RS	R _D = COUNTLEADINGONES(RS)
CLZ	R _D , RS	R _D = COUNTLEADINGZEROS(RS)
<u>LA</u>	R _D , LABEL	R _D = ADDRESS(LABEL)
<u>LI</u>	R _D , IMM32	R _D = IMM32
LUI	R _D , CONST16	R _D = CONST16 << 16
<u>MOVE</u>	R _D , RS	R _D = R _S
<u>NEGU</u>	R _D , RS	R _D = −R _S
SEB ^{R2}	R _D , RS	R _D = R _S _{7:0} [±]
SEH ^{R2}	R _D , RS	R _D = R _S _{15:0} [±]
SUB	R _D , RS, R _T	R _D = R _S − R _T (OVERFLOW TRAP)
SUBU	R _D , RS, R _T	R _D = R _S − R _T

SHIFT AND ROTATE OPERATIONS		
ROTR ^{R2}	R _D , RS, BITS5	R _D = R _S _{BIT55−1:0} :: R _S _{31:BIT55}
ROTRV ^{R2}	R _D , RS, R _T	R _D = R _S _{RT40−1:0} :: R _S _{31:RT40}
SLL	R _D , RS, SHIFT5	R _D = R _S << SHIFT5
SLLV	R _D , RS, R _T	R _D = R _S << R _T _{4:0}
SRA	R _D , RS, SHIFT5	R _D = R _S [±] >>> SHIFT5
SRAV	R _D , RS, R _T	R _D = R _S [±] >>> R _T _{4:0}
SRL	R _D , RS, SHIFT5	R _D = R _S [∅] >>> SHIFT5
SRLV	R _D , RS, R _T	R _D = R _S [∅] >>> R _T _{4:0}

LOGICAL AND BIT-FIELD OPERATIONS		
AND	R _D , RS, R _T	R _D = R _S & R _T
ANDI	R _D , RS, CONST16	R _D = R _S & CONST16 [∅]
EXT ^{R2}	R _D , RS, P, S	R _S = R _S _{P+5−1:P} [∅]
INS ^{R2}	R _D , RS, P, S	R _D _{P+S−1:P} = R _S _{S−1:0}
<u>NOP</u>		No-OP
NOR	R _D , RS, R _T	R _D = ~(R _S R _T)
<u>NOT</u>	R _D , RS	R _D = ~R _S
OR	R _D , RS, R _T	R _D = R _S R _T
ORI	R _D , RS, CONST16	R _D = R _S CONST16 [∅]
WSBH ^{R2}	R _D , RS	R _D = R _S _{23:16} :: R _S _{31:24} :: R _S _{7:0} :: R _S _{15:8}
XOR	R _D , RS, R _T	R _D = R _S ⊕ R _T
XORI	R _D , RS, CONST16	R _D = R _S ⊕ CONST16 [∅]

CONDITION TESTING AND CONDITIONAL MOVE OPERATIONS		
MOVN	R _D , RS, R _T	IF R _T ≠ 0, R _D = R _S
MOVZ	R _D , RS, R _T	IF R _T = 0, R _D = R _S
SLT	R _D , RS, R _T	R _D = (R _S [±] < R _T [±]) ? 1 : 0
SLTI	R _D , RS, CONST16	R _D = (R _S [±] < CONST16 [±]) ? 1 : 0
SLTIU	R _D , RS, CONST16	R _D = (R _S [∅] < CONST16 [∅]) ? 1 : 0
SLTU	R _D , RS, R _T	R _D = (R _S [∅] < R _T [∅]) ? 1 : 0

MULTIPLY AND DIVIDE OPERATIONS		
DIV	RS, R _T	Lo = R _S [±] / R _T [±] ; Hi = R _S [±] MOD R _T [±]
DIVU	RS, R _T	Lo = R _S [∅] / R _T [∅] ; Hi = R _S [∅] MOD R _T [∅]
MADD	RS, R _T	ACC += R _S [±] × R _T [±]
MADDU	RS, R _T	ACC += R _S [∅] × R _T [∅]
MSUB	RS, R _T	ACC −= R _S [±] × R _T [±]
MSUBU	RS, R _T	ACC −= R _S [∅] × R _T [∅]
MUL	R _D , RS, R _T	R _D = R _S [±] × R _T [±]
MULT	RS, R _T	ACC = R _S [±] × R _T [±]
MULTU	RS, R _T	ACC = R _S [∅] × R _T [∅]

ACCUMULATOR ACCESS OPERATIONS		
MFHI	R _D	R _D = Hi
MFLO	R _D	R _D = Lo
MTHI	RS	Hi = R _S
MTLO	RS	Lo = R _S

JUMPS AND BRANCHES (NOTE: ONE DELAY SLOT)		
<u>B</u>	OFF18	PC += OFF18 [±]
<u>BAL</u>	OFF18	RA = PC + 8, PC += OFF18 [±]
BEQ	RS, R _T , OFF18	IF R _S = R _T , PC += OFF18 [±]
<u>BEQZ</u>	RS, OFF18	IF R _S = 0, PC += OFF18 [±]
BGEZ	RS, OFF18	IF R _S ≥ 0, PC += OFF18 [±]
BGEZAL	RS, OFF18	RA = PC + 8; IF R _S ≥ 0, PC += OFF18 [±]
BGTZ	RS, OFF18	IF R _S > 0, PC += OFF18 [±]
BLEZ	RS, OFF18	IF R _S ≤ 0, PC += OFF18 [±]
BLTZ	RS, OFF18	IF R _S < 0, PC += OFF18 [±]
BLTZAL	RS, OFF18	RA = PC + 8; IF R _S < 0, PC += OFF18 [±]
BNE	RS, R _T , OFF18	IF R _S ≠ R _T , PC += OFF18 [±]
<u>BNEZ</u>	RS, OFF18	IF R _S ≠ 0, PC += OFF18 [±]
J	ADDR28	PC = PC _{31:28} :: ADDR28 [∅]
JAL	ADDR28	RA = PC + 8; PC = PC _{31:28} :: ADDR28 [∅]
JALR	R _D , RS	R _D = PC + 8; PC = R _S
JR	RS	PC = R _S

LOAD AND STORE OPERATIONS		
LB	R _D , OFF16(RS)	R _D = MEM8(R _S + OFF16 [±]) [±]
LBU	R _D , OFF16(RS)	R _D = MEM8(R _S + OFF16 [±]) [∅]
LH	R _D , OFF16(RS)	R _D = MEM16(R _S + OFF16 [±]) [±]
LHU	R _D , OFF16(RS)	R _D = MEM16(R _S + OFF16 [±]) [∅]
LW	R _D , OFF16(RS)	R _D = MEM32(R _S + OFF16 [±])
LWL	R _D , OFF16(RS)	R _D = LOADWORDLEFT(R _S + OFF16 [±])
LWR	R _D , OFF16(RS)	R _D = LOADWORDRIGHT(R _S + OFF16 [±])
SB	RS, OFF16(RT)	MEM8(R _T + OFF16 [±]) = R _S _{7:0}
SH	RS, OFF16(RT)	MEM16(R _T + OFF16 [±]) = R _S _{15:0}
SW	RS, OFF16(RT)	MEM32(R _T + OFF16 [±]) = R _S
SWL	RS, OFF16(RT)	STOREWORDLEFT(R _T + OFF16 [±] , R _S)
SWR	RS, OFF16(RT)	STOREWORDRIGHT(R _T + OFF16 [±] , R _S)
<u>ULW</u>	R _D , OFF16(RS)	R _D = UNALIGNED_MEM32(R _S + OFF16 [±])
<u>USW</u>	RS, OFF16(RT)	UNALIGNED_MEM32(R _T + OFF16 [±]) = R _S

ATOMIC READ-MODIFY-WRITE OPERATIONS		
LL	R _D , OFF16(RS)	R _D = MEM32(R _S + OFF16 [±]); LINK
SC	R _D , OFF16(RS)	IF ATOMIC, MEM32(R _S + OFF16 [±]) = R _D ; R _D = ATOMIC ? 1 : 0

REGISTERS		
0	zero	Always equal to zero
1	at	Assembler temporary; used by the assembler
2-3	v0-v1	Return value from a function call
4-7	a0-a3	First four parameters for a function call
8-15	t0-t7	Temporary variables; need not be preserved
16-23	s0-s7	Function variables; must be preserved
24-25	t8-t9	Two more temporary variables
26-27	k0-k1	Kernel use registers; may change unexpectedly
28	gp	Global pointer
29	sp	Stack pointer
30	fp/s8	Stack frame pointer or subroutine variable
31	ra	Return address of the last subroutine call

DEFAULT C CALLING CONVENTION (O32)	
<p>Stack Management</p> <ul style="list-style-type: none"> The stack grows down. <ul style="list-style-type: none"> Subtract from \$sp to allocate local storage space. Restore \$sp by adding the same amount at function exit. The stack must be 8-byte aligned. <ul style="list-style-type: none"> Modify \$sp only in multiples of eight. 	
<p>Function Parameters</p> <ul style="list-style-type: none"> Every parameter smaller than 32 bits is promoted to 32 bits. First four parameters are passed in registers \$a0–\$a3. <ul style="list-style-type: none"> 64-bit parameters are passed in register pairs: <ul style="list-style-type: none"> Little-endian mode: \$a1:\$a0 or \$a3:\$a2. Big-endian mode: \$a0:\$a1 or \$a2:\$a3. Every subsequent parameter is passed through the stack. <ul style="list-style-type: none"> First 16 bytes on the stack are not used. Assuming \$sp was not modified at function entry: <ul style="list-style-type: none"> The 1st stack parameter is located at 16(\$sp). The 2nd stack parameter is located at 20(\$sp), etc. 64-bit parameters are 8-byte aligned. 	
<p>Return Values</p> <ul style="list-style-type: none"> 32-bit and smaller values are returned in register \$v0. 64-bit values are returned in registers \$v0 and \$v1: <ul style="list-style-type: none"> Little-endian mode: \$v1:\$v0. Big-endian mode: \$v0:\$v1. 	

MIPS32 VIRTUAL ADDRESS SPACE				
kseg3	0xE000.0000	0xFFFF.FFFF	Mapped	Cached
ksseg	0xC000.0000	0xDFFF.FFFF	Mapped	Cached
kseg1	0xA000.0000	0xBFFF.FFFF	Unmapped	Uncached
kseg0	0x8000.0000	0x9FFF.FFFF	Unmapped	Cached
useg	0x0000.0000	0x7FFF.FFFF	Mapped	Cached

READING THE CYCLE COUNT REGISTER FROM C
<pre> unsigned mips_cycle_counter_read() { unsigned cc; asm volatile("mfc0 %0, \$9" : "=r" (cc)); return (cc << 1); } </pre>

ASSEMBLY-LANGUAGE FUNCTION EXAMPLE
<pre> # int asm_max(int a, int b) # { # int r = (a < b) ? b : a; # return r; # } .text .set nomacro .set noreorder .global asm_max .ent asm_max asm_max: move \$v0, \$a0 # r = a slt \$t0, \$a0, \$a1 # a < b ? jr \$ra # return movn \$v0, \$a1, \$t0 # if yes, r = b .end asm_max </pre>

C / ASSEMBLY-LANGUAGE FUNCTION INTERFACE
<pre> #include <stdio.h> int asm_max(int a, int b); int main() { int x = asm_max(10, 100); int y = asm_max(200, 20); printf("%d %d\n", x, y); } </pre>

INVOKING MULT AND MADD INSTRUCTIONS FROM C
<pre> int dp(int a[], int b[], int n) { int i; long long acc = (long long) a[0] * b[0]; for (i = 1; i < n; i++) acc += (long long) a[i] * b[i]; return (acc >> 31); } </pre>

ATOMIC READ-MODIFY-WRITE EXAMPLE
<pre> atomic_inc: ll \$t0, 0(\$a0) # load linked addiu \$t1, \$t0, 1 # increment sc \$t1, 0(\$a0) # store cond'1 beqz \$t1, atomic_inc # loop if failed nop </pre>

ACCESSING UNALIGNED DATA			
NOTE: ULW AND USW AUTOMATICALLY GENERATE APPROPRIATE CODE			
LITTLE-ENDIAN MODE		BIG-ENDIAN MODE	
LWR	RD, OFF16(Rs)	LWL	RD, OFF16(Rs)
LWL	RD, OFF16+3(Rs)	LWR	RD, OFF16+3(Rs)
SWR	RD, OFF16(Rs)	SWL	RD, OFF16(Rs)
SWL	RD, OFF16+3(Rs)	SWR	RD, OFF16+3(Rs)

ACCESSING UNALIGNED DATA FROM C
<pre> typedef struct { int u; } __attribute__((packed)) unaligned; int unaligned_load(void *ptr) { unaligned *uptr = (unaligned *)ptr; return uptr->u; } </pre>

MIPS SDE-GCC COMPILER DEFINES	
__mips	MIPS ISA (= 32 for MIPS32)
__mips_isa_rev	MIPS ISA Revision (= 2 for MIPS32 R2)
__mips_dsp	DSP ASE extensions enabled
_MIPSEB	Big-endian target CPU
_MIPSEL	Little-endian target CPU
_MIPS_ARCH_CPU	Target CPU specified by -march= <i>CPU</i>
_MIPS_TUNE_CPU	Pipeline tuning selected by -mtune= <i>CPU</i>

NOTES
<ul style="list-style-type: none"> Many assembler pseudo-instructions and some rarely used machine instructions are omitted. The C calling convention is simplified. Additional rules apply when passing complex data structures as function parameters. The examples illustrate syntax used by GCC compilers. Most MIPS processors increment the cycle counter every other cycle. Please check your processor documentation.