MIPS Encoding Reference

Instruction Encodings

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

Register Encoding

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.

```
oooooss ssstttt dddddaaa aaffffff
```

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

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.

```
oooooss ssstttt iiiiiiii iiiiiiii
```

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 -2^{15} and $2^{15}-1$.

Jump Encoding

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

```
oooooooii iiiiiiii iiiiiiii iiiiiiii
```

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 -2^{25} and $2^{25}-1$.

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	ArithLog	$d = $s + $t
addu	100001	ArithLog	$d = $s + $t
addi	001000	ArithLogI	$t = $s + SE(i)
addiu	001001	ArithLogI	$t = $s + SE(i)
and	100100	ArithLog	$d = $s & $t
andi	001100	ArithLogI	$t = $s & ZE(i)
div	011010	DivMult	lo = $s / $t; hi = $s % $t
divu	011011	DivMult	lo = $s / $t; hi = $s % $t
mult	011000	DivMult	hi:lo = $s * $t
multu	011001	DivMult	hi:lo = $s * $t
nor	100111	ArithLog	$d = ~($s | $t)
or	100101	ArithLog	$d = $s | $t
ori	001101	ArithLogI	$t = $s | ZE(i)
sll	000000	Shift	$d = $t << a
sllv	000100	ShiftV	$d = $t << $s
sra	000011	Shift	$d = $t >> a
srav	000111	ShiftV	$d = $t >> $s
srl	000010	Shift	$d = $t >>> a
srlv	000110	ShiftV	$d = $t >>> $s
sub	100010	ArithLog	$d = $s - $t
subu	100011	ArithLog	$d = $s - $t
xor	100110	ArithLog	$d = $s ^ $t
xori	001110	ArithLogI	$d = $s ^ ZE(i)
Constant-Manipulating Instructions			
Instruction	**Opcode/Function**	**Syntax**	**Operation**
lhi	011001	LoadI	`HH ($t) = i`
llo	011000	LoadI	`LH ($t) = i`
Comparison Instructions			
Instruction	**Opcode/Function**	**Syntax**	**Operation**
slt	101010	ArithLog	$d = ($s < $t)
sltu	101001	ArithLog	$d = ($s < $t)

slti	001010	ArithLogI	$t = ($s < SE(i))
sltiu	001001	ArithLogI	$t = ($s < SE(i))
Branch Instructions			
Instruction	**Opcode/Function**	**Syntax**	**Operation**
beq	000100	Branch	if ($s == $t) pc += i << 2
bgtz	000111	BranchZ	if ($s > 0) pc += i << 2
blez	000110	BranchZ	if ($s <= 0) pc += i << 2
bne	000101	Branch	if ($s != $t) pc += i << 2
Jump Instructions			
Instruction	**Opcode/Function**	**Syntax**	**Operation**
j	000010	Jump	pc += i << 2
jal	000011	Jump	$31 = pc; pc += i << 2
jalr	001001	JumpR	$31 = pc; pc = $s
jr	001000	JumpR	pc = $s
Load Instructions			
Instruction	**Opcode/Function**	**Syntax**	**Operation**
lb	100000	LoadStore	`$t = SE (MEM [$s + i]:1)`
lbu	100100	LoadStore	`$t = ZE (MEM [$s + i]:1)`
lh	100001	LoadStore	`$t = SE (MEM [$s + i]:2)`
lhu	100101	LoadStore	`$t = ZE (MEM [$s + i]:2)`
lw	100011	LoadStore	`$t = MEM [$s + i]:4`
Store Instructions			
Instruction	**Opcode/Function**	**Syntax**	**Operation**
sb	101000	LoadStore	`MEM [$s + i]:1 = LB ($t)`
sh	101001	LoadStore	`MEM [$s + i]:2 = LH ($t)`
sw	101011	LoadStore	`MEM [$s + i]:4 = $t`
Data Movement Instructions			
Instruction	**Opcode/Function**	**Syntax**	**Operation**
mfhi	010000	MoveFrom	$d = hi
mflo	010010	MoveFrom	$d = lo
mthi	010001	MoveTo	hi = $s
mtlo	010011	MoveTo	lo = $s
Exception and Interrupt Instructions			
Instruction	**Opcode/Function**	**Syntax**	**Operation**
trap	011010	Trap	Dependent on operating system; different values for immed26 specify different operations. See the list of 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								