Part 5: MIPS Instruction Set

In this section, we will describe the encoding format of MIPS assembly instructions, list the most common MIPS instructions, and discuss the anatomy of pseudo-instructions.

MIPS Instruction Formats

In Part 1: Introduction to MIPS Assembly, we discussed that assembly instructions are mnemonics for the combination of 1's and 0's that are defined as machine code instructions.

MIPS Instructions are always 4 bytes (32 bits) in size. To distinguish one instruction from another, several bits out of the 32 are assigned to represent the operation code (opcode), while other bits are assigned to represent the source and destination registers.

These combinations of bits make up several different types of MIPS instruction formats.

They are:

- R Instructions
- I Instructions
- J Instructions

Each instruction format follows a different syntax and encoding which will be described below in **big-endian** format. This is also described in greater detail from the MIPS Assembly Wikibook here

R Instruction Format

[**R**]**egister** instructions have operands that are registers.

An R instruction has the machine-code format:

[**opcode** (6 bits)] [**Rs** (5 bits)] [**Rt** (5 bits)] [**Rd** (5 bits)] [**shift** (5 bits)] [**function code** (6 bits)]

The **opcode** is the binary representation of the instruction. Related instructions can have the same opcode to which the **function code** bits of the instruction are used to tell the difference.

For example, add and addu have the same opcode but different function codes.

Rs, **Rt**, and **Rd** represent **source register**, **target register**, and **destination registers** respectively.

shift bits are used with the shift instructions and determine the number of shifts to be performed.

R Instructions that do not directly fit the machine-code format and omit bits for example **Rd** or **shift** bits will have those bits as all 0's.

For example, **jr Rs** has the encoding **0000 00ss sss0 0000 0000 0000 0000 1000**

An incomplete list of R-type instructions is

- add
- addu
- and
- or
- sll
- jr

An example of an R-type instruction in binary format would be: **0000 0010 0011 0010 1000 0000 0010 0000**

And the equivalent assembly instruction is: **add $s0**, **$s1**, **$s2**

I Instruction Format

[**I**]**mmediate** instructions have an operand that is an immediate value to be operated onto a register.

An I instruction has the machine-code format:

[**opcode** (6 bits)] [**Rs** (5 bits)] [**Rt** (5 bits)] [**Immediate** (16 bits)]

Just as in R-type instructions, the **opcode** is the binary representation of the instruction and **Rs** and **Rt** represent **source register** and **target register** respectively.

The **immediate** value is also called the **offset** when it comes to the load instructions.

An incomplete list of I-type instructions are

- beq
- bne
- addi
- lb
- lui

An example of an I instruction in binary format would be: **0010 0011 1011 1101 0000 0000 0000 0100**

And the equivalent assembly instruction is: **addi $sp**, **$sp**, **4**

J Instruction Format

[**J**]**ump** instructions describe the format for an instruction where a jump is being performed. Jumps and branches will be described in greater detail in Part 6: Jumps and Branches

A J instruction has the machine-code format:

[**opcode** (6 bits)] [**absolute-address** (26 bits)]

The **absolute address** is a 26-bit shortened memory address that is the destination of the jump.

An example of a J instruction in binary format would be: **0000 1000 0001 0000 0000 0000 0000 0011**

And the equivalent assembly instruction is: **j 0x0040000c**

MIPS Registers Encoding

Rs, Rt, and Rd are to be substituted with the corresponding MIPS registers $0-$31 in binary.

For example, the instruction: add $s0, $s1, $s2

Using the equivalent register numbers which can be viewed in the table from Part 3: MIPS Registers, the instruction can be read as: add $16, $17, $18

So the encoding will be to convert decimal 16, 17, and 18 to binary to get the encoding for Rd, Rs, and Rt.

MIPS Instruction Set Table

Below is a table with the most common MIPS-32 instructions adapted from the MIPS Assembly Wikibook and from here

To learn the instruction set, I recommend setting up a lab (either MARS, SPIM, or qemu) to test instructions and see what they do.

Instruction Name	**Description**	**Syntax**	**Operation**	**Instruction Type**	**Encoding**
add	add (with overflow)	add **Rd**, **Rs**, **Rt**	**Rd** = **Rs** + **Rt**	R	0000 00**ss** **ssst tttt** **dddd d**000 00**10 0000**
addi	add immediate (with overflow)	addi **Rt**, **Rs**, Immediate	**Rt** = **Rs** + Immediate	I	0010 00**ss** **ssst tttt** iiii iiii iiii iiii
addiu	add immediate unsigned (no overflow)	addiu **Rt**, **Rs**, Immediate	**Rt** = **Rs** + Immediate	I	0010 01**ss** **ssst tttt** iiii iiii iiii iiii
addu	add unsigned (no overflow)	addu **Rd**, **Rs**, **Rt**	**Rd** = **Rs** + **Rt**	R	0000 00**ss** **ssst tttt** **dddd d**000 00**10 0001**
and	bitwise AND	and **Rd**, **Rs**, **Rt**	**Rd** = **Rs** & **Rt**	R	0000 00**ss** **ssst tttt** **dddd d**000 00**10 0100**

andi	bitwise AND immediate	andi **Rt**, **Rs**, immediate	**Rt** = **Rs** & immediate	I	0011 00**ss** **ssst tttt** iiii iiii iiii iiii
beq	branch on equal	beq **Rs**, **Rt**, offset	(**Rs** == **Rt**) ? $pc + (offset << 2) : $pc + 4	I	0001 00**ss** **ssst tttt** iiii iiii iiii iiii
bne	branch on not equal	bne **Rs**, **Rt**, offset	(**Rs** != **Rt**) ? $pc + (offset << 2) : $pc + 4	I	0001 01**ss** **ssst tttt** iiii iiii iiii iiii
blez	branch on less than or equal to zero	blez **Rs**, offset	(**Rs** <= 0) ? $pc + (offset << 2) : $pc + 4	I	0001 10**ss** **sss**0 0000 iiii iiii iiii iiii
bltz	branch on less than zero	bltz **Rs**, offset	(**Rs** < 0) ? $pc + (offset << 2) : $pc + 4	I	0000 01**ss** **sss**0 0000 iiii iiii iiii iiii
bltzal	branch on less than zero and link (saves return address)	bltzal **Rs**, offset	(**Rs** < 0) ? $ra = $pc + 8; $pc + (offset << 2) : $pc + 4	I	0000 01**ss** **sss**1 0000 iiii iiii iiii iiii
bgez	branch on greater than or equal to zero	bgez **Rs**, offset	(**Rs** >= 0) ? $pc + (offset << 2) : $pc + 4	I	0000 01**ss** **sss**0 0001 iiii iiii iiii iiii
bgtz	branch on greater than zero	bgtz **Rs**, offset	(**Rs** > 0) ? $pc + (offset << 2) : $pc + 4	I	0001 11**ss** **sss**0 0000 iiii iiii iiii iiii
bgezal	branch on greater than or equal to zero and link (saves return address)	bgezal **Rs**, offset	(**Rs** >= 0) ? $ra = $pc + 4; $pc + (offset << 2) : $pc + 4	I	0000 01**ss** **sss**1 0001 iiii iiii iiii iiii
div	divides **Rs** by Rt and stores quotient in $Lo and remainder in $Hi	div **Rs**, **Rt**	$Lo = **Rs** / **Rt**; $Hi = **Rs** % **Rt**	R	0000 00**ss** **ssst tttt** 0000 0000 00**01 1010**
divu	divides (unsigned) **Rs** by **Rt** and stores quotient in	divu **Rs**, **Rt**	$Lo = **Rs** / **Rt**; $Hi = **Rs** % **Rt**	R	0000 00**ss** **ssst tttt**

	$Lo and remainder in $Hi				0000 0000 00**01 1011**
j	jump to 26 bit absolute-address	j **absolute-addr**	$pc = next-$pc; next-$pc = ($pc & 0xf0000000) | (**absolute-addr** << 2);	J	0000 10**aa aaaa aaaa aaaa aaaa aaaa**
jal	jump and link (stores return address)	jal **absolute-addr**	$ra = $pc + 8; $pc = next-$pc; next-$pc = ($pc & 0xf0000000) | (**absolute-addr** << 2);	J	0000 11**aa aaaa aaaa aaaa aaaa aaaa**
jr	(jump register) jump to 4-byte address contained in register **Rs**	jr **Rs**	$pc = next-$pc; next-$pc = **Rs**;	R	0000 00**ss sss**0 0000 0000 0000 00**00 1000**
lb	(load byte) load one byte into target register from specified address	lb **Rt**, offset(**Rs**)	**Rt** = Memory[**Rs** + offset]	I	1000 00**ss ssst tttt** iiii iiii iiii iiii
lui	(load upper immediate) load 2 byte immediate value into upper 2 bytes of a register. Lower 2 bytes are zeroe'd out	lui **Rt**, immediate	**Rt** = immediate << 16	I	0011 11-- ---**t tttt** iiii iiii iiii iiii
lw	(load word) load 4 bytes into target register from memory	lw **Rt**, offset(**Rs**)	**Rt** = Memory[**Rs** + offset]	I	1000 11**ss ssst tttt** iiii iiii iiii iiii
mfhi	(move from $Hi) contents of register $Hi are moved to **destination register**	mfhi **Rd**	**Rd** = $Hi	R	0000 0000 0000 0000 **dddd d**000 00**01 0000**
mflo	(move from $Lo) contents of register $Lo are moved to **destination register**	mflo **Rd**	**Rd** = $Lo	R	0000 0000 0000 0000 **dddd d**000 00**01 0010**

mult	(multiply) multiply Rs by Rt and stores result in $Lo	mult **Rs**, **Rt**	$Lo = **Rs** * **Rt**	R	0000 00**ss** **ssst tttt** 0000 0000 00**01 1000**
multu	(multiply unsigned) multiply Rs by Rt and stores result in $Lo	multu **Rs**, **Rt**	$Lo = **Rs** * **Rt**	R	0000 00**ss** **ssst tttt** 0000 0000 00**01 1001**
noop	no operation - CPU does nothing. Most instructions with $zero as the destination register can act as a noop.	noop	This particular encoding is implemented as sll $zero, $zero, $zero	R	0000 0000 0000 0000 0000 0000 0000 0000
or	bitwise OR	or **Rd**, **Rt**, **Rs**	**Rd** = **Rs** | **Rt**	R	0000 00**ss** **ssst tttt** **dddd d**000 00**10 0101**
ori	bitwise OR immediate	ori **Rt**, **Rs**, immediate	**Rt** = **Rs** | immediate	I	0011 01**ss** **ssst tttt** iiii iiii iiii iiii
sb	(store byte) store least significant byte of Rt to memory	sb **Rt**, offset(**Rs**)	Memory[**Rs** + offset] = (0xff & **Rt**)	I	1010 00**ss** **ssst tttt** iiii iiii iiii iiii
sw	(store word) store 4 bytes at a specified address	sw **Rt**, offset(**Rs**)	Memory[**Rs** + offset] = **Rt**	I	1010 11**ss** **ssst tttt** iiii iiii iiii iiii
sll	(shift left logical) shift register value left with zeroes by specified number of bits	sll **Rd**, **Rt**, x	**Rd** = **Rt** << x	R	0000 00**ss** **ssst tttt** **dddd d**xxx xx**00 0000**
sllv	(shift left logical variable) shift register value left with zeroes by specified number of bits in source register	sllv **Rd**, **Rt**, **Rs**	**Rd** = **Rt** << **Rs**	R	0000 00**ss** **ssst tttt** **dddd d**--- ---**00 0100**

slt	(set on less than - signed) set destination register to 0x01 if source register is less than target register, else set **destination register** to 0x00.	slt **Rd**, **Rs**, **Rt**	(**Rs** < **Rt**) ? **Rd** = 1 : **Rd** = 0	R	0000 00**ss ssst tttt dddd d**000 00**10 1010**
slti	(set on less than immediate - signed) set target register to 0x01 if source register is less than immediate value, else set target register to 0x00.	slti **Rt**, **Rs**, immediate	(**Rs** < immediate) ? **Rt** = 1 : Rt = 0	R	0010 10**ss ssst tttt** iiii iiii iiii iiii
sra	(shift right arithmetic) shift a register value right with sign bit shifted in by the specified number of bits and place the value in **destination register**	sra **Rd**, **Rt**, x	**Rd** = **Rt** >> x	R	0000 00-- ---**t tttt dddd d**xxx xx**00 0011**
srl	(shift right logical) shift a register value right with zeroes in by the specified number of bits and place the value in **destination register**	srl **Rd**, **Rt**, x	**Rd** = **Rt** >> x	R	0000 00-- ---**t tttt dddd d**xxx xx**00 0010**
srlv	(shift right logical variable) shift a register value right with zeroes in by the specified number of shifts in the source register and place the value in **destination register**	srlv **Rd**, **Rt**, **Rs**	**Rd** = **Rt** >> **Rs**	R	0000 00**ss ssst tttt dddd d**000 00**00 0110**
sub	subtract two registers and store the result in the destination register	sub **Rd**, **Rs**, **Rt**	**Rd** = **Rs - Rt**	R	0000 00**ss ssst tttt dddd d**000 00**10 0010**

syscall	Generate a software interrupt and perform appropriate system call based on value in $v0	syscall 0x40404	Operation dependant on syscall number. An example syscall is socket(2, 2, 0)	R	0000 00-- ---- ---- ---- ---- **--00 1100**
xor	bitwise exclusive OR	xor **Rd**, **Rs**, **Rt**	**Rd** = **Rs** ^ **Rt**	R	0000 00**ss ssst tttt dddd d**--- ---**10 0110**
xori	bitwise exclusive OR to immediate value	xori **Rt**, **Rs**, immediate	**Rt** = **Rs** ^ immediate	I	0011 10**ss ssst tttt** iiii iiii iiii iiii

Pseudo-Instructions

The MIPS instruction set is very minimal thus there are several macros, also known as, pseudo-instructions that the assembler will translate into the corresponding instructions.

When writing MIPS assembly, some assemblers support the usage of certain pseudo-instructions and will convert them to the corresponding assembly instructions.

An example pseudo-instruction is move, the **move** instruction in MIPS is actually achieved using the **add** instruction.

So **move $s0, $s2** translates to => **add $s0, $s2, $zero**

Another common pseudo-instruction often seen in disassembly is: **la $a0, 0x7fffffff**

The **load address (la)** instruction is actually represented by two MIPS instructions:

1. lui $a0, 0x7fff - (load upper 2 bytes of address)
2. ori $a0, $a0, 0xffff - (load lower 2 bytes of address)

For more examples of pseudo-instructions, visit the link here

Further Reading

1. MIPS Instruction Reference (UIdaho)

2. Programmed Introduction to MIPS Assembly Language (Central Connecticut State University)

3. MIPT-ILAB MIPS Pseudo Instructions Git Wiki

Part 6: Jumps and Branches