MIPS Assembly/Instruction Formats

This page describes the implementation details of the MIPS instruction formats.

Contents

R Instructions

R instructions are used when all the data values used by the instruction are located in registers.

All R-type instructions have the following format:

```
OP rd, rs, rt
```

Where "OP" is the mnemonic for the particular instruction. *rs*, and *rt* are the source registers, and *rd* is the destination register. As an example, the **add** mnemonic can be used as:

```
add $s1, $s2, $s3
```

Where the values in *$s2* and *$s3* are added together, and the result is stored in *$s1*. In the main narrative of this book, the operands will be denoted by these names.

R Format

Converting an R mnemonic into the equivalent binary machine code is performed in the following way:

opcode	rs	rt	rd	shift (shamt)	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

opcode
: The opcode is the machinecode representation of the instruction mnemonic. Several related instructions can have the same opcode. The opcode field is 6 bits long (bit 26 to bit 31).

rs, rt, rd
: The numeric representations of the source registers and the destination register. These numbers correspond to the $X representation of a register, such as $0 or $31. Each of these fields is 5 bits long. (25 to 21, 20 to 16, and 15 to 11, respectively). Interestingly, rather than rs and rt being named r1 and r2

(for source register 1 and 2), the registers were named "rs" and "rt" because t comes after s in the alphabet. This was most likely done to reduce numerical confusion.

Shift (shamt)
Used with the shift and rotate instructions, this is the amount by which the source operand *rs* is rotated/shifted. This field is 5 bits long (6 to 10).

Funct
For instructions that share an opcode, the **funct** parameter contains the necessary control codes to differentiate the different instructions. 6 bits long (0 to 5). Example: Opcode 0x00 accesses the ALU, and the funct selects which ALU function to use.

Function Codes

Because several functions can have the same opcode, R-Type instructions need a function (Func) code to identify what exactly is being done - for example, 0x00 refers to an ALU operation and 0x20 refers to ADDing specifically.

Shift Values

I Instructions

I instructions are used when the instruction must operate on an immediate value and a register value. Immediate values may be a maximum of 16 bits long. Larger numbers may not be manipulated by immediate instructions.

I instructions are called in the following way:

```
OP rt, rs, IMM
```

Where *rt* is the target register, *rs* is the source register, and *IMM* is the immediate value. The immediate value can be up to 16 bits long. For instance, the **addi** instruction can be called as:

```
addi $s1, $s2, 100
```

Where the value of $s2 plus 100 is stored in $s1.

I Format

I instructions are converted into machine code words in the following format:

opcode	rs	rt	IMM
6 bits	5 bits	5 bits	16 bits

Opcode
The 6-bit opcode of the instruction. In I instructions, all mneumonics have a one-to-one correspondence with the underlying opcodes. This is because there is no **funct** parameter to differentiate instructions with an identical opcode. 6 bits (26 to 31)

rs, rt
The source and target register operands, respectively. 5 bits each (21 to 25 and 16 to 20, respectively).[1] (http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Mips/format.html)

IMM
The 16 bit immediate value. 16 bits (0 to 15). This value is usually used as the offset value in various instructions, and depending on the instruction, may be expressed in two's complement.

J Instructions

J instructions are used when a jump needs to be performed. The J instruction has the most space for an immediate value, because addresses are large numbers.

J instructions are called in the following way:

```
OP LABEL
```

Where *OP* is the mnemonic for the particular jump instruction, and *LABEL* is the target address to jump to.

J Format

J instructions have the following machine-code format:

Opcode	Pseudo-Address

Opcode
The 6 bit opcode corresponding to the particular jump command. (26 to 31).

Address
A 26-bit shortened address of the destination. (0 to 25). The two most LSBits are removed, and the 4 MSBits are removed, and assumed to be the same as the current instruction's address.

FR Instructions

FR instructions are similar to the R instructions described above, except they are reserved for use with floating-point numbers:

Opcode	fmt	ft	fs	fd	funct

FI Instructions

FI instructions are similar to the I instructions described above, except they are reserved for use with floating-point numbers:

Opcode	fmt	ft	Imm

Opcodes

The following table contains a listing of MIPS instructions and the corresponding opcodes. Opcode and funct numbers are all listed in hexadecimal.

Mnemonic	Meaning	Type	Opcode	Funct
add	Add	R	0x00	0x20
addi	Add Immediate	I	0x08	NA
addiu	Add Unsigned Immediate	I	0x09	NA
addu	Add Unsigned	R	0x00	0x21
and	Bitwise AND	R	0x00	0x24
andi	Bitwise AND Immediate	I	0x0C	NA
beq	Branch if Equal	I	0x04	NA
bne	Branch if Not Equal	I	0x05	NA
div	Divide	R	0x00	0x1A
divu	Unsigned Divide	R	0x00	0x1B
j	Jump to Address	J	0x02	NA
jal	Jump and Link	J	0x03	NA
jr	Jump to Address in Register	R	0x00	0x08
lbu	Load Byte Unsigned	I	0x24	NA
lhu	Load Halfword Unsigned	I	0x25	NA
lui	Load Upper Immediate	I	0x0F	NA
lw	Load Word	I	0x23	NA
mfhi	Move from HI Register	R	0x00	0x10
mflo	Move from LO Register	R	0x00	0x12
mfc0	Move from Coprocessor 0	R	0x10	NA
mult	Multiply	R	0x00	0x18
multu	Unsigned Multiply	R	0x00	0x19
nor	Bitwise NOR (NOT-OR)	R	0x00	0x27
xor	Bitwise XOR (Exclusive-OR)	R	0x00	0x26
or	Bitwise OR	R	0x00	0x25
ori	Bitwise OR Immediate	I	0x0D	NA
sb	Store Byte	I	0x28	NA
sh	Store Halfword	I	0x29	NA
slt	Set to 1 if Less Than	R	0x00	0x2A
slti	Set to 1 if Less Than Immediate	I	0x0A	NA
sltiu	Set to 1 if Less Than Unsigned Immediate	I	0x0B	NA
sltu	Set to 1 if Less Than Unsigned	R	0x00	0x2B
sll	Logical Shift Left	R	0x00	0x00
srl	Logical Shift Right (0-extended)	R	0x00	0x02
sra	Arithmetic Shift Right (sign-extended)	R	0x00	0x03
sub	Subtract	R	0x00	0x22
subu	Unsigned Subtract	R	0x00	0x23

Mnemonic	Meaning	Type	Opcode	Funct
sw	Store Word	I	0x2B	NA

Retrieved from "https://en.wikibooks.org/w/index.php?title=MIPS_Assembly/Instruction_Formats&oldid=3224433"

Next: Many Examples: **Up:** Chapter 4: Instruction Set **Previous:** An Example: the M68000

Instruction Set of MIPS Processor

- **Register file (RF):** 32 registers ($0 through $31), each for a word of 32 bits (4 bytes);
  - $0 always holds zero
  - $sp (29) is the stack pointer (SP) which always points to the top item of a stack in the memory;
  - $ra (31) always holds the return address from a subroutine

  The table below shows the conventional usage of all 32 registers.

Register Number	Mnemonic Name	Conventional Use	Register Number	Mnemonic Name	Conventional Use
$0	zero	Permanently 0	$24, $25	$t8, $t9	Temporary
$1	$at	Assembler Temporary (reserved)	$26, $27	$k0, $k1	Kernel (reserved for OS)
$2, $3	$v0, $v1	Value returned by a subroutine	$28	$gp	Global Pointer
$4–$7	$a0–$a3	Arguments to a subroutine	$29	$sp	Stack Pointer
$8–$15	$t0–$t7	Temporary (not preserved across a function call)	$30	$fp	Frame Pointer
$16–$23	$s0–$s7	Saved registers (preserved across a function call)	$31	$ra	Return Address

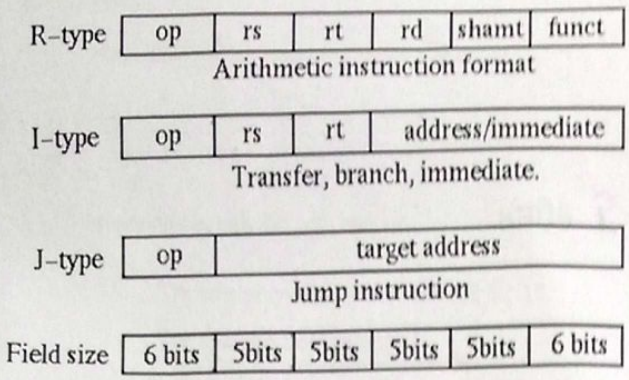
MIPS registers

- **Main memory (MM):** 2^{32} addressable bytes ($0, 1, 2, 3, \cdots, 2^{32}-1 = 4,294,967,295$, 4 Gb) or 2^{30} words ($0, 4, 8, 12, \cdots, 2^{32}-4 = 4,294,967,292$).

$2^{32}-1$	$2^{32}-2$	$2^{32}-3$	$2^{32}-4$
....	....	....	....
19	18	17	16
15	14	13	12
11	10	9	8
7	6	5	4
3	2	1	0

MIPS virtual memory

- **Instruction set**: each instruction in the instruction set describes one particular CUP operation. Each instruction is represented in both **assembly language** by the *mnimonics* and **machine language** (binary) by a word of 32 bits subdivided into several fields.



There are different types of instructions:

1. Computational Instructions

These instructions are for arithmetic or logic manipulations. In general they operate on two operands and store the result.

```
op  $rd, $rs, $rt
```

$rd ← $rs * $rt

where

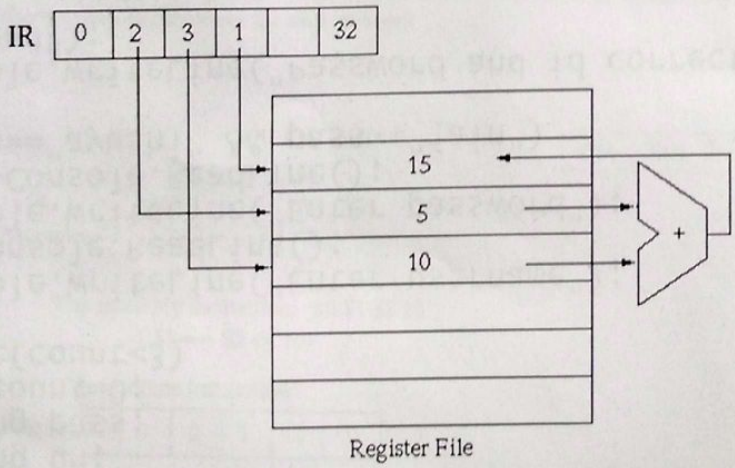
- op: opcode specifying the arithmetic/logic operation to be performed;
- $rd: destination register in which the result is to be stored;
- $rs: source register containing the 1st operand;
- $rt: source register containing the 2nd operand.

Opcode can be: add, sub, mult, div, and, or, etc.

$rd, $rs, $rt can be any of the 32 registers.

The assembly instruction: add $1 $2 $3

The machine instruction:



Note:

- The destination register is specified in the first field following the opcode field in the assembly l instruction, but the last 5-bit field in the binary machine language instruction.
- In all R-type data manipulation instructions (arithmetic, logical, shift), the operations are spe function field (6 least significant bits) in the binary instruction, with the opcode field (6 mc all equal to zero.

Note:

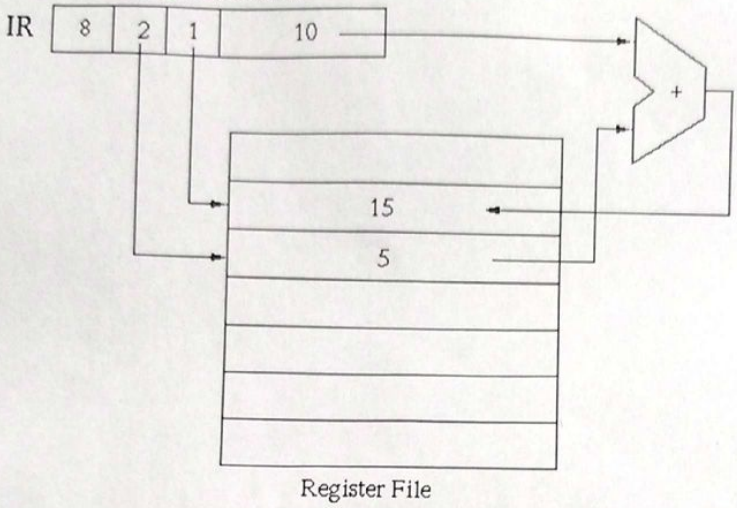
Immediate mode - replacing $rt by a constant:

```
op   $rd, $rs, constant
```

where op can only be: add, and, or.

The assembly instruction: addi $1 $2 10

The machine instruction:



2. Shift Instructions

```
op   $rd, $rs, shamt
```

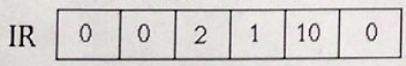
where op can be *sll* (shift left logical) *srl* (shift right logical) or *sra* (shift right arithmetic), and *shamt* specifies the number of bits to shift. The shift amount can also be specified by a variable in a register, such as the example below ($t1 holds the shift amount):

```
srlv   $rd, $rs, $t1
```

Examples:

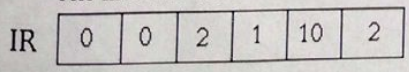
The assembly instruction: sll $1 $2 10
($1 ← $2 << 10)

The machine instruction:



The assembly instruction: srl $1 $2 10
($1 ← $2 >> 10)

The machine instruction:



3. Data Transfer Instructions

These instructions transfer data back and forth between the MM and the CPU.

- Load word from MM to register:

```
lw   $rd, offset($rs)
```

$rd ← Memory[offset + $rs]

where

- lw: the opcode for *load word*;
- $rd: destination register into which the word is to be loaded;
- $rs: source register (e.g., an index number of an array);
- offset: (e.g., the beginning address of the array in memory).
- effective MM address: offset + $rs

- Store word from register to memory:

```
sw   $rs, offset($rd)
```

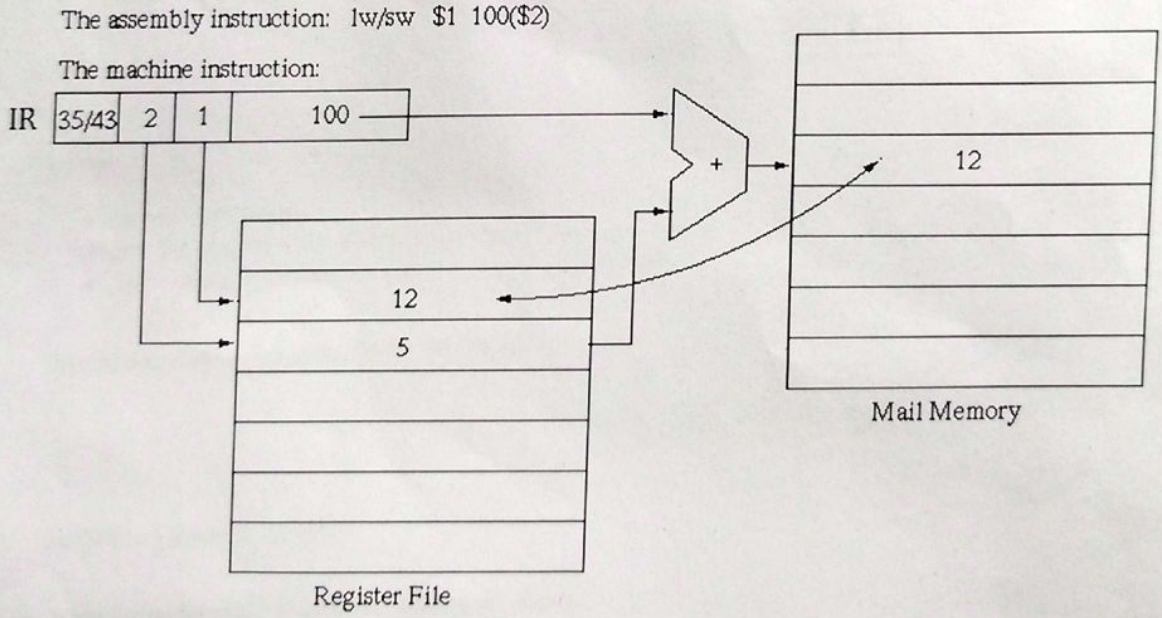
$rs → Memory[offset + $rd]

where

- sw: the opcode for *save word*;
- $rs: source register whose content is to be stored;
- $rd: destination register, e.g., an index number of an array;
- offset: e.g., the beginning address of the array in memory.
- MM address: offset + $rd

The assembly instruction: lw/sw $1 100($2)

The machine instruction:



4. Program Control

Usually the program is executed in the straight line fashion, i.e., the next instruction to be executed is the one that follows the previous one currently being executed. But some time it is needed to conditionally or unconditionally jump to some other part of the program (e.g., functions, loops, etc.) by the program control instructions.

- Branch to a labeled statement if two variables are equal:

```
beq   $rd, $rs, L1
```

or not equal:

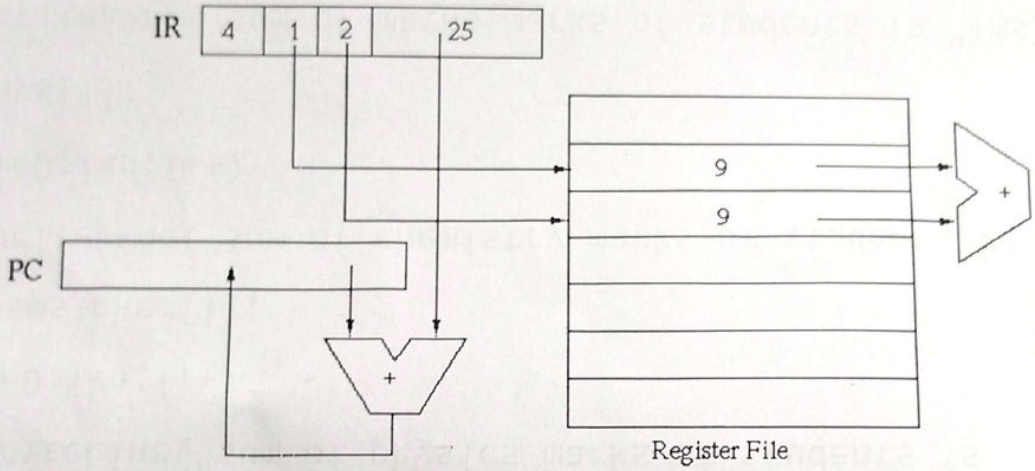
```
bne  $rd, $rs, L1
```

where

- beq (bne): branch if equal (not equal);
- $rd, $rs: two registers holding the variables;
- L1: label for the target statement.

The assembly instruction: beq $1 $2 Lable

The machine instruction:



- Set a register to 1 if first variable is smaller than the second, set the register to 0 otherwise.

```
slt  $rd, $rs, $rt
```

where

- slt: set if less than
- $rs, $rt: registers containing two variables to be compared;
- $rd: register to be set to 1 if $rs < $rt, or 0 otherwise.

Immediate mode -- replacing $rt by a constant:

```
slti  $rd, $rs, constant
```

set $rd to 1 if $rs < constant.

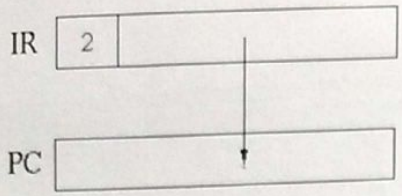
- Jump unconditionally to a certain labeled statement:

```
j  label
```

6/12/2017

The assmebly instruction: j Label

The machine instruction:



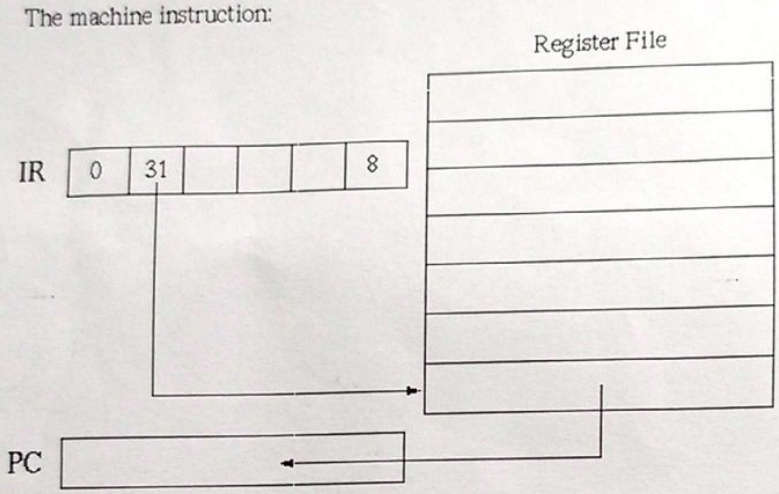
- Jump unconditionally to a certain statement whose address is given in a register:

```
jr   $rd
```

- Jump unconditionally to a subroutine (or function, procedure, etc.) whose address is given as a symbol SubroutineAddress:

The assmebly instruction: jr $31

The machine instruction:




```
jal  SubroutineAddress
```

Operations related to function calls:

- Branch to a subroutine by jal (jump and link)
  1. update PC ($PC \leftarrow PC + 4$) to point to the next instruction;
  2. save content of PC into register $ra ($31) as the return address;
  3. load starting address of subroutine (symbolized as SubroutineAddress) to PC.
- Return to calling routine:

  To return from subroutine to the calling routine, the last statement of the subroutine must be:

```
jr   $ra
```

- Recursion: For recursive subroutine calls, previous content of $ra ($31) needs to be stored in a *stack* in the memory.

MIPS R3000 Instruction Set

MIPS Machine Language

MIPS R3000 Instruction Set Summary

MIPS Operands

Name	Example	Comments
32 registers	$0, $1, $2,..., $31	Fast location for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $0 always equal 0. Register $1 is reserved for the assembler to handle pseudo instructions and large constants
2^{30} memory words	Memory[0], Memory[4],..., Memory[4293967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls

MIPS Assembler Instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $1,$2,$3	$1 = $2 + $3	3 operands; exception possible
	subtract	sub $1,$2,$3	$1 = $2 - $3	3 operands; exception possible
	add immediate	addi $1,$2,100	$1 = $2 + 100	+ constant; exception possible
	add unsigned	addu $1,$2,$3	$1 = $2 + $3	3 operands; exception possible
	subtract unsigned	subi $1,$2,$3	$1 = $2 - $3	3 operands; exception possible
	add immediate unsigned	addi $1,$2,100	$1 = $2 + 100	+ constant; exception possible
	Move from coprocessor register	mfc0 $1,$epc	$1 = $epc	Used to get of Exception PC
Logical	and	and $1,$2,$3	$1 = $2 & $3	3 register operands; Logical AND
	or	or $1,$2,$3	$1 = $2 | $3	3 register operands; Logical OR
	and immediate	and $1,$2,100	$1 = $2 & 100	Logical AND register, constant
	or immediate	or $1,$2,100	$1 = $2 | 100	Logical OR register, constant
	shift left logical	sll $1,$2,10	$1 = $2 << 10	Shift left by constant
	shift right logical	srl $1,$2,10	$1 = $2 >> 10	Shift right by constant
Data transfer	load word	lw $1, (100)$2	$1 = Memory[$2+100]	Data from memory to register
	store word	sw $1, (100)$2	Memory[$2+100] = $1	Data from memory to register
	load upper immediate	lui $1,100	$1 = 100 * 2^{16}	Load constant in upper 16bits

Conditional branch	branch on equal	beq $1,$2,100	if ($1 == $2) go to PC+4+100	Equal test; PC relative branch
	branch on not equal	bne $1,$2,100	if ($1 != $2) go to PC+4+100	Not equal test; PC relative
	set on less than	slt $1,$2,$3	if ($2 < $3) $1 = 1; else $1 = 0	Compare less than; 2`s complement
	set less than immediate	slti $1,$2,100	if ($2 < 100) $1 = 1; else $1 = 0	Compare < constant; 2`s complement
	set less than unsigned	sltu $1,$2,$3	if ($2 < $3) $1 = 1; else $1 = 0	Compare less than; natural number
	set less than immediate unsigned	sltiu $1,$2,100	if ($2 < 100) $1 = 1; else $1 = 0	Compare constant; natural number
Unconditional jump	jump	j 10000	goto 10000	Jump to target address
	jump register	j $31	goto $31	For switch, procedure return
	jump and link	jal 10000	$31 = PC + 4;go to 10000	For procedure call

MIPS Floating-Point Operands

No FPO

Name	Example	Comments
32 floating-point registers	$f0, $f1, $f2,..., $f31	MIPS floating point register are used in pairs for double precision numbers. Odd numbered registers cannot be used for arithemetic or branch, just for data transfer of the right "half" of double precision register pairs.
2^{30} memory words	Memory[0], Memory[4],..., Memory[4293967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls

MIPS Floating-Point Instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	add.s $f2,$f4,$f6	$f2 = $f4 + $f6	Floating-Point add (single precision)
	FP subtract single	sub.s $f2,$f4,$f6	$f2 = $f4 - $f6	Floating-Point sub (single precision)
	FP multiply single	mul.s $f2,$f4,$f6	$f2 = $f4 * $f6	Floating-Point multiply (single precision)
	FP divide single	div.s $f2,$f4,$f6	$f2 = $f4 / $f6	Floating-Point divide (single precision)
	FP add double	add.d $f2,$f4,$f6	$f2 = $f4 + $f6	Floating-Point add (double precision)
	FP.dubtract double	.dub.d $f2,$f4,$f6	$f2 = $f4 - $f6	Floating-Point sub (double precision)
	FP multiply double	mul.d $f2,$f4,$f6	$f2 = $f4 * $f6	Floating-Point multiply (double precision)
	FP divide double	div.d $f2,$f4,$f6	$f2 = $f4 / $f6	Floating-Point divide (double precision)
Data	load word coprocessor 1	lwc1	$f1 =	32-bit data to FP register

transfer		$f1,100($2)	Memory[$2+100]	
	store word coprocessor 1	swc1 $f1,100($2)	Memory[$2+100] = $f1	32-bit data to memory
Arithmetic	branch on FP true	bc1t 100	if (cond == 1) go to PC+4+100	PC relative branch if FP condition
	branch on FP false	bc1f 100	if (cond == 0) go to PC+4+100	PC relative branch if not condition
	FP compare single (eq,ne,lt,le,gt,ge)	c.lt.s $f2,$f4	if ($f2 < $f4) cond=1; else cond=0	Floating-point compare less than single precision
	FP compare double (eq,ne,lt,le,gt,ge)	c.lt.d $f2,$f4	if ($f2 < $f4) cond=1; else cond=0	Floating-point compare less than double precision