MIPS

Reference Guide

Table of Contents

Data Registers

MIPS contains 32 registers for programmers to use:

#	Register(s)		Usage
0	$zero	→	Hard-wired to 0
1	$at	→	Reserved for assembler
2,3	$v0, $v1	→	Used to store returned values from function calls
4-7	$a0 - $a3	→	Used to store values passed as arguments to functions
8-15	$t0 - $t7	→	Temporary registers
16-23	$s0 - $s7	→	Saved temporary registers
24,25	$t8, $t9	→	Temporary registers
26, 27	$k0, $k1	→	Reserved for operating system kernel
28	$gp	→	Global pointer
29	$sp	→	Stack pointer
30	$fp	→	Frame pointer
31	$ra	→	Return address for function calls

Instruction Register Formats

The MIPS IR register supports three different register formats. They are R (register), I (immediate) and J (jump). All MIPS registers are 32-bit, so each register format is 32 bits wide. They differ in the number and types of fields they contain.

R Format

Op-Code	Rs	Rt	Rd	Rh	Function Code
000000	sssss	ttttt	ddddd	hhhhh	ffffff

I Format

Op-Code	Rs	Rt	Immediate
ffffff	sssss	ttttt	iiiiiiiiiiiiiii

J Format

Op-Code	Target
ffffff	iiiiiiiiiiiiiiiiiiiiiiiii

The R (register) format consists of five different fields. The 6-bit op-code will always be 000000. Rs, Rt and Rd are 5-bit fields that specify the locations of registers being used. Rs and Rt are sources for the operation. Rd is the destination to store the result. If Rh (shift amount) is not used, it becomes 00000. The last 5 bits are the function code. This tells the computer which type of instruction should be executed.

The I (immediate) format consists of four different fields. The 6-bit op-code determines what type of instruction should be executed. This is similar to the function code in the R-format. The Rs field is the source for the operation. The Rt is the register destination to store the result. The last 16 bits hold the value being applied in the operation.

The J (jump) format consists of only two fields. The 6-bit op-code will always be 00001f. The last 26 bits specify the location being jumped to. These type of instructions are similar to high-level language “go to” commands.

MIPS Instruction Set

ADD	Add
ADDI	Add immediate
ADDIU	Add immediate unsigned
ADDU	Add unsigned
AND	And
ANDI	And immediate
BEQ	Branch on equal
BGEZ	Branch on >= 0
BGEZAL	Branch on >= 0 and link
BGTZ	Branch on > 0
BLEZ	Branch on <= 0
BLTZ	Branch on < 0
BLTZAL	Branch on < 0 and link
BNE	Branch on != 0
DIV	Divide
DIVU	Divide unsigned
J	Jump
JAL	Jump and link
JALR	Jump and link register
JR	Jump register
LB	Load byte
LBU	Load byte unsigned
LH	Load halfword
LHU	Load halfword unsigned
LUI	Load upper immediate
LW	Load word
LWL	Load word left
LWR	Load word right
MFHI	Move from $HI

MFLO	Move from $LO
MTHI	Move to $HI
MTLO	Move to $LO
MULT	Multiply
MULTU	Multiply unsigned
NOOP	No operation
NOR	Nor
OR	Or
ORI	Or immediate
SB	Store byte
SH	Store halfword
SLL	Shift left logical
SLLV	Shift left logical variable
SLT	Set on less than
SLTI	Set on less than immediate
SLTIU	Set on less than immediate unsigned
SLTU	Set on less than unsigned
SRA	Shift right arithmetic
SRAV	Shift right arithmetic variable
SRL	Shift right logical
SRLV	Shift right logical variable
SUB	Subtract
SUBU	Subtract unsigned
SW	Store word
SWL	Store word left
SWR	Store word right
SYSCALL	System call
XOR	Xor
XORI	Xor immediate

MIPS Instruction Set (Extended)

ADD	add $d, $s, $t	*add*
	Meaning →	$d = $s + $t
additional info	Function Code →	100000

ADDI	addi $t, $s, imm	*add immediate*
	Meaning →	$t = $s + imm
additional info	Op-Code →	001000

ADDIU	addiu $t, $s, imm	*add immediate unsigned*
	Meaning →	$t = $s + imm(unsigned)
additional info	Op-Code →	001001

ADDU	addu $d, $s, $t	*add unsigned*
	Meaning →	$d = $s + $t
additional info	Function Code →	100001

AND	and $d, $s, $t	*and*
	Meaning →	$d = $s and $t
additional info	Function Code→	100100

ANDI	andi $t, $s, imm	*and immediate*
	Meaning →	$t = $s and imm
additional info	Op-Code→	001100

BEQ	beq $s, $t, offset	*branch on equal*
	Meaning →	if $s == $t branch to offset
additional info	Op-Code→	000100

BGEZ	bgez $s, offset	*branch >= zero*
	Meaning →	if $s >= 0 branch to offset
additional info	Op-Code→	000001
	Rt →	00001

BGEZAL	bgezal $s, offset	*branch >= zero and link*
	Meaning →	if $s >= 0 branch to offset save return address in $ra
additional info	Op-Code→	000001
	Rt →	10001

BGTZ	bgtz $s, offset	*branch > zero*
	Meaning →	if $s > 0 branch to offset
additional info	Op-Code→	000111
	Rt →	00000

BLEZ	blez $s, offset	*branch <= zero*
	Meaning →	if $s <= 0 branch to offset
additional info	Op-Code→	000110
	Rt →	00000

BLTZ	bltz $s, offset	*branch < zero*
	Meaning →	if $s < 0 branch to offset
additional info	Op-Code→	000001
	Rt →	00000

BLTZAL	bltzal $s, offset	*branch < zero and link*
	Meaning →	if $s < 0 branch to offset save return address in $ra
additional info	Op-Code→	000001
	Rt →	10000

BNE	bne $s, $t, offset	*branch on not equal*
	Meaning →	if $s != $t branch to offset
additional info	Op-Code→	000101

DIV	div $s, $t	*divide*
	Meaning →	$LO = $s / $t $HI = $s % $t
additional info	Function Code →	011010

DIVU	divu $s, $t	*divide unsigned*
	Meaning →	$LO = $s / $t $HI = $s % $t
additional info	Function Code →	011011

J	j target	*jump*
	Meaning →	Jump to target location
additional info	Op-Code →	000010

JAL	jal target	*jump and link*
	Meaning →	Jump to target location save return address in $ra
additional info	Op-Code →	000011

JALR	jal $d, $s	*jump and link register*
	Meaning →	Jump to location specified by $s Save return address in $d
additional info	Function Code →	001001

JR	jr $s	*jump register*
	Meaning →	Jump to target location contained in register $s
additional info	Function Code→	001000

LB	lb $t, offset($s)	*load byte*
	Meaning →	$t = [$s + offset]
additional info	Op-Code →	100000

LBU	lbu $t, offset($s)	*load byte unsigned*
	Meaning →	$t = [$s + offset]
additional info	Op-Code →	100100

LH	lh $t, offset($s)	*load halfword*
	Meaning →	$t = halfword [$s + offset]
additional info	Op-Code →	100001

LHU	lhu $t, offset($s)	*load halfword unsigned*
	Meaning →	$t = halfword [$s + offset]
additional info	Op-Code →	100101

LUI	lb $t, imm	*load upper immediate*
	Meaning →	$t = imm after imm is shifted left 16 bits
additional info	Op-Code →	001111

LW	lw $t, offset($s)	*load word*
	Meaning →	$t = [$s + offset]
additional info	Op-Code →	100011

LWL	lwl $t, offset($s)	*load word left*
	Meaning →	$t = [$s + offset]
additional info	Op-Code →	100010

LWR	lwr $t, offset($s)	*load word right*
	Meaning →	$t = [$s + offset]
additional info	Op-Code →	100110

MFHI	mfhi $d	*move from HI*
	Meaning →	$d = $HI
additional info	Function Code →	010000

MFLO	mflo $d	*move from LO*
	Meaning →	$d = $LO
additional info	Function Code →	010010

MTHI	mfhi $s	*move to HI*
	Meaning →	$HI = $s
additional info	Function Code →	010001

MTLO	mtlo $s	*move to LO*
	Meaning →	$LO = $s
additional info	Function Code →	010011

MULT	mult $s, $t	*multiply*
	Meaning →	$LO = $s * $t
additional info	Function Code →	011000

MULTU	multu $s, $t	*multiply unsigned*
	Meaning →	$LO = $s * $t
additional info	Function Code →	011001

NOOP	noop	*no operation*
	Meaning →	no operation

NOR	nor $d, $s, $t	*nor*
	Meaning →	$d = $s nor $t
additional info	Function Code →	100111

OR	or $d, $s, $t	*or*
	Meaning →	$d = $s or $t
additional info	Function Code →	100101

ORI	ori $t, $s, imm	*or immediate*
	Meaning →	$t = $s or imm
additional info	Op-Code →	001101

SB	sb $t, offset($s)	*store byte*
	Meaning →	[$s + offset] = least significant bit of $t
additional info	Op-Code →	101000

SH	sh $t, offset($s)	*store halfword*
	Meaning →	[$s + offset] = half word $t
additional info	Op-Code →	101001

SLL	sll $d, $t, h	*shift left logical*
	Meaning →	$d = $t shifted left h times
additional info	Function Code →	000000

SLLV	sllv $d, $t, $s	*shift left logical variable*
	Meaning →	$d = $t shifted left # times in $s
additional info	Function Code →	000100

SLT	slt $d, $s, $t	*set on less than*
	Meaning →	if $s < $t then $d = 1 else $d = 0
additional info	Function Code →	101010

SLTI	slti $t, $s, imm	*set on less than immediate*
	Meaning →	if $s < imm then $t = 1 else $t = 0
additional info	Op-Code →	001010

SLTIU	sltiu $t, $s, imm	*SLT immediate unsigned*
	Meaning →	if $s < imm then $t = 1 else $t = 0
additional info	Op-Code →	001011

SLTU	sltu $d, $s, $t	*set on less than unsigned*
	Meaning →	if $s < $t then $d = 1 else $d = 0
additional info	Function Code →	101011

SRA	sra $d, $t, h	*shift right arithmetic*
	Meaning →	$d = $t shifted right h times
additional info	Function Code →	000011

SRAV	srav $d, $t, $s	*shift right arith. variable*
	Meaning →	$d = $t shifted right # times in $s
additional info	Function Code →	000111

SRL	srl $d, $t, h	*shift right logical*
	Meaning →	$d = $t shifted right h times
additional info	Function Code →	000010

SRLV	srlv $d, $t, $s	*shift right logical variable*
	Meaning →	$d = $t shifted right # times in $s
additional info	Function Code →	000110

SUB	sub $d, $s, $t	*subtract*
	Meaning →	$d = $s - $t
additional info	Function Code →	100010

SUBU	subu $d, $s, $t	*subtract unsigned*
	Meaning →	$d = $s - $t
additional info	Function Code →	100011

SW	sw $t, offset($s)	*store word*
	Meaning →	[$s + offset] = $t
additional info	Op-Code →	101011

SWL	swl $t, offset($s)	*store word left*
	Meaning →	[$s + offset] = $t
additional info	Op-Code →	101010

SWR	swr $t, offset($s)	*store word right*
	Meaning →	[$s + offset] = $t
additional info	Op-Code →	101110

SYSCALL	syscall	*system call*
	Meaning →	Sends an interrupt
additional info	Function Code →	001100

XOR	xor $d, $s, $t	*exclusive or*
	Meaning →	$d = $s xor $t
additional info	Function Code →	100110

XORI	xori $t, $s, imm	*exclusive or immediate*
	Meaning →	$t = $s xor imm
additional info	Op-Code →	001110

SPIM Programming

Every program written in SPIM needs a data and text segment.

```
#.data signifies the beginning of the data segment
.data

#.text starts the “text” portion of the program
.text
```

Within the data segment you can initialize your variables. All variables are initialized in the form:

Name: .Type Content

The name is user defined. It can be any name the programmer wishes to call the variable by. The variable types are the following:

.ascii	→	ASCII string
.asciiz	→	ASCII string followed by a null terminator
.byte	→	Byte
.double	→	Double
.float	→	Float
.word	→	Word

SPIM can be downloaded for free at
http://www.cs.wisc.edu/~larus/spim.html

Program Examples

```
#This program prints to screen the string “Hello World!”

#.data signifies the beginning of the data segment
.data

#If hello is called within the main program it will lead to the string.
#.asciiz means that the string is in ASCII format followed by
#a NULL terminator
hello: .asciiz "Hello World!"

.globl main

#.text starts the “text” portion of the program
.text

#Start main program
main:

	#Setting register $v0 equal to 4 tells the processor that
	#a string in register $a0 is going to be printed to screen
	li $v0, 4
	#Setting content of $a0 to string hello
	la $a0, hello
	#Calling system to perform output
	syscall
```

```
#This program inputs a number and then displays the number

#Data portion of program
.data

.globl main

#Text portion of program
.text

#Start main program
main:

	#Setting register $v0 to 5 tells the processor that
	#an integer is going to be entered from the keyboard
	li $v0, 5
	#calling system to perform input
	syscall

	#The integer that was entered will now be in
	#register $v0.
	#Moving this value into register $t0.
	move $t0, $v0

	#Setting register $v0 to 1 tells the processor that the
	#contents of register $a0 are going to be printed to the monitor
	li $v0, 1
	#Moving content of register $t0 into register $a0
	move $a0, $t0
	#calling system to perform output
	syscall
```

```
#This program asks the user for two integers and then displays the sum

#Data portion of the program
.data

#Creating ASCII strings for input prompt and output
Msg: .asciiz "Enter in an integer: "
Msg2: .asciiz "The sum is: "
#Creating ASCII string for a carriage return
return: .asciiz "\n"

.globl main

#Text portion of the program
.text
#Starting main program
main:

       #Print to screen string “Enter in an integer: “
       li $v0, 4
       la $a0, Msg
       syscall

       #Input an integer from keyboard into register $v0
       li $v0, 5
       syscall
       #Move content of register $v0 into register $t0
       move $t0, $v0

       #Print to screen string “Enter in an integer: “
       li $v0, 4
       la $a0, Msg
       syscall

       #Input an integer from keyboard into register $v0
       li $v0, 5
       syscall
       #move content of register $v0 into register $t1
       move $t1, $v0

       #Print to screen string “\n” car carriage return.
       li $v0, 4
       la $a0, return
       syscall

       #Print to screen string “The sum is: “
       li $v0, 4
       la $a0, Msg2
       syscall

       #Adding registers $t0 and $t1 and store sum in $t2
       add $t2,$t0,$t1
       #Move content of register $t2 (the sum) into register $a0
       move $a0, $t2
       #Print to screen content of $a0
       li $v0, 1
       syscall
```

```
#This program asks the user for two numbers and displays their product

#Data portion of the program
.data

#Creating ASCII string for input prompt
msg1:  .asciiz "Please enter a number: "
#Creating ASCII string for output
msg2:  .asciiz "The product is: "

.globl main

#Text portion of the program
.text

#Starting main program
main:

       #Printing to screen string “Please enter a number: “
       li $v0, 4
       la $a0, msg1
       syscall

       #Input an integer from keyboard into register $v0
       li $v0, 5
       syscall
       #Move content of register $v0 into register $t0
       move $t0, $v0

       #Printing to screen string “Please enter a number: “
       li $v0, 4
       la $a0, msg1
       syscall

       #Input an integer from keyboard into register $v0
       li $v0, 5
       syscall
       #Move content of register $v0 into register $t1
       move $t1, $v0

       #Multiplying $t0 by $t1. Product will be stored in register $LO
       mult $t0, $t1

       #Moving content of $LO (the product) into register $t2
       mflo $t2

       #Printing to screen string “The product is: “
       li $v0, 4
       la $a0, msg2
       syscall

       #Moving content of $t2 (the product) into register $a0
       move $a0, $t2

       #Printing to screen content of $a0
       li $v0, 1
       syscall
```

```
#This program asks the user for an integer and then determines if it
#is even or odd

#Data portion of the program
.data

#Creating ASCII string for input prompt
question: .asciiz "Please enter an integer: "
#Creating ASCII string for output if the number is even
even: .asciiz "That number is even"
#Creating ASCII string for output if the number is odd
odd: .asciiz "That number is odd"

.globl main

#Text portion of the program
.text

#Starting main program
main:

	#Print to screen the string “Please enter an integer: “
	li $v0, 4
	la $a0, question
	syscall

	#Input an integer from keyboard and store it in register $v0
	li $v0, 5
	syscall
	#Move content of $v0 into register $t0
	move $t0, $v0

	#Load register $t1 with immediate value of 2
	li $t1, 2
	#Divide $t0 by $t1.
	#$t0 % $t1 will be stored in $HI. $t0 * $t1 will be stored in $LO
	div $t0, $t1

	#Move content of $HI into register $t2
	mfhi $t2
	#If register $t2 is 0 (NUM % 2 = 0) then branch to AAA
	beq $t2, $zero, AAA

	#Print to screen string “That number is odd” if haven’t branched
	li $v0, 4
	la $a0, odd
	syscall
	#Jump to BBB (to skip message for even number)
	j BBB
#AAA start
AAA:
	#Print to screen string “That number is even”
	li $v0, 4
	la $a0, even
	syscall
#BBB start
BBB:
```