MIPS Architecture

Registers

The MIPS processor has 32 general-purpose registers, plus one for the program counter (called PC) and two for the results of the multiplication and division operations, called HI and LO, for the high 32 bits and the low 32 bits of the answer. The following chart summarizes the registers' usage.

number	*name*	*use*
$0		always holds the value 0
$1	$at	reserved by the assembler
$2 ... $3	$v0 ... $v1	expression evaluation and function results
$4 ... $7	$a0 ... $a3	*first 4 function parameters
$8 ... $15	$t0 ... $t7	*temporaries
$16 ... $23	$s0 ... $s7	saved values
$24 ... $25	$t8 ... $t9	*temporaries
$26 ... $27	$k0 ... $k1	reserved for use by operating system
$28	$gp	global pointer
$29	$sp	stack pointer
$30	$s8	saved value
$31	$ra	return address

A * in the use column means the values in those registers are **not** preserved across procedure calls.

Opcode Formats

The MIPS processor uses 3 different types of instructions.

I-Type (Immediate) Instructions

bits 31 ... 26	opcode
bits 25 ... 21	source register
bits 20 ... 16	target (destination) register
bits 15 ... 0	immediate operand

J-Type (Jump) Instructions

bits 31 ... 26	opcode
bits 25 ... 0	target (destination) offset

R-Type (Register) Instructions

bits 31 ... 26	opcode
bits 25 ... 21	source register
bits 20 ... 16	target (source) register
bits 15 ... 11	destination register
bits 10 ... 6	shift amount
bits 5 ... 0	function information

Complete MIPS R3000 Instruction Set

	Load/Store Instructions
LB	Load Byte
LBU	Load Byte Unsigned
LH	Load Halfword
LHU	Load Halfword Unsigned
LW	Load Word
LWL	Load Word Left
LWR	Load Word Right
SB	Store Byte
SH	Store Halfword
SW	Store Word
SWL	Store Word Left
SWR	Store Word Right

	Arithmetic Instructions (ALU Immediate)
ADDI	Add Immediate
ADDIU	Add Immediate Unsigned
SLTI	Set on Less Than Immediate
SLTIU	Set on Less than Immediate Unsigned
ANDI	AND Immediate
ORI	OR Immediate
XORI	Exclusive OR Immediate
LUI	Load Upper Immediate

	Arithmetic Instructions (3-operand, Register Type)
ADD	Add
ADDU	Add Unsigned
SUB	Subtract
SUBU	Subtract Unsigned
SLT	Set on Less Than
SLTU	Set on Less Than Unsigned
AND	AND
OR	OR
XOR	Exclusive OR
NOR	NOR

	Multiply/Divide Instructions
MULT	Multiply
MULTU	Multiply Unsigned
DIV	Divide
DIVU	Divide Unsigned
MFHI	Move From HI
MTHI	Move To HI
MFLO	Move From LO
MTLO	Move to LO

	Jump & Branch Instructions
J	Jump
JAL	Jump and Link
JR	Jump to Register
JALR	Jump and Link Register
BEQ	Branch on Equal
BNE	Branch on Not Equal
BLEZ	Branch on Less than or Equal to Zero
BGTZ	Branch on Greater than Zero
BLTZ	Branch on Less than Zero
BGEZ	Branch on $\geq$ Zero
BLTZAL	Branch on Less than Zero and Link
BGEZAL	Branch on $\geq$ Zero and Link

	Coprocessor Instructions
LWCz	Load Word from Coprocessor
SWCz	Store Word to Coprocessor
MTCz	Move to Coprocessor
MFCz	Move from Coprocessor
CTCz	Move Control to Coprocessor
CFCz	Move Control from Coprocessor
COPz	Coprocessor Operation
BCzT	Branch on Coprocessor z True
BCzF	Branch on Coprocessor z False

	Shift Instructions
SLL	Shift Left Logical
SRL	Shift Right Logical
SRA	Shift Right Arithmetic
SLLV	Shift Left Logical Variable
SRLV	Shift Right Logical Variable
SRAV	Shift Right Arithmetic Variable
	Special Instructions
BREAK	Break
SYSCALL	System Call

	System Control Coprocessor (CP0) Instructions
MTC0	Move to CP0
MFC0	Move From CP0
TLBR	Read indexed TLB Entry
TLBWI	Write indexed TLB entry
TLBWR	Write Random TLB entry
TLBP	Probe TLB for matching entry
RFE	Restore from Except

Rules on Delays and Interlocks

- There is one delay slot after any branch or jump instruction, i.e., the following instruction is executed even if the branch is taken. That following instruction must not be itself a jump or branch.
- There is one delay slot after a “load” no matter what size is being loaded. That is, the instruction after a “load” must *not* use the register being loaded.
- Multiplication will place its results in the LO and HI registers after an undefined number of following instructions have executed. There’s a hardware interlock to stall further multiplications, divisions, or move from LO or HI to execute until the operation is finished.
- Division is like multiplication but most likely slower.

MIPS Opcodes and Formats

These are synopses of the MIPS instructions. Not all instruction s are listed; in particular, those involving traps or memory management are omitted.

ADD rd, rs, rt *add*
Adds `rs` and `rt`, puts result into `rd`. Exception on overflow.

ADDI rt, rs, immediate *add, immediate*
Sign-extends the 16-bit `immediate` to 32 bits, adds it to `rs`, puts result into `rt`. Exception on overflow.

ADDIU rt, rs, immediate *add, unsigned immediate*
Sign-extends the 16-bit `immediate` to 32 bits, adds it to `rs`, puts result into `rt`. Never causes an overflow.

ADDU rd, rs, rt *add, unsigned*
Adds `rs` and `rt`, puts result into `rd`. Never causes an overflow.

AND rd, rs, rt *and*
Bitwise and's `rs` and `rt`, puts result into `rd`.

ANDI rt, rs, immediate *and, immediate*
Sign-extends the 16-bit `immediate` to 32 bits, bitwise ands it with `rs`, puts result into `rt`.

BEQ rs, rt, offset *branch equal*
If `rs` = `rt`, branches to `offset` [after executing the following instruction]. For most assemblers, `offset` is a `label`.

BGEZ rs, offset *branch greater-equal-zero*
If `rs` ≥ 0, branches to `offset` [after executing the following instruction]. For most assemblers, `offset` is a `label`.

BGEZAL rs, offset *branch greater-equal-zero, and link*
If `rs` ≥ 0, branches to `offset` [after executing the following instruction]. For most assemblers, `offset` is a `label`. Always places address of following instruction into `r31`. Note that `rs` may not itself be `r31`. (This is a subroutine-call instruction.)

BGTZ rs, offset *branch greater-than-zero*
If `rs` > 0, branches to `offset` [after executing the following instruction]. For most assemblers, `offset` is a `label`.

BLEZ rs, offset *branch less-equal-zero*
If `rs` ≤ 0, branches to `offset` [after executing the following instruction]. For most assemblers, `offset` is a `label`.

BLTZ rs, offset *branch less-than-zero*
If `rs` < 0, branches to `offset` [after executing the following instruction]. For most assemblers, `offset` is a `label`.

BLTZAL, rs offset *branch less-than-zero, and link*
If `rs` < 0, branches to `offset` [after executing the following instruction].

For most assemblers, `offset` is a `label`. Always places address of following instruction into `r31`. Note that `rs` may not itself be `r31`. (This is a subroutine-call instruction.)

BNE rs, rt, label *branch not-equal*

If `rs` ≠ `rt`, branches to `offset` [after executing the following instruction]. For most assemblers, `offset` is a `label`.

DIV rs, rt *divide*

Divides `rs` by `rt`, treating both as (signed) 2's complement numbers. Quotient goes into special register LO and remainder into special register HI. Get them via the MFHI and MFLO instructions. No overflow exception occurs, and the result is undefined if `rt` contains 0.
Note that divides take an undefined amount of time; other instructions will execute in parallel. MFHI and MFLO will interlock until the division is complete.

DIVU rs, rt *divide, unsigned*

Divides `rs` by `rt`, treating both as unsigned numbers. Quotient goes into special register LO and remainder into special register HI. Get them via the MFHI and MFLO instructions. No overflow exception occurs, and the result is undefined if `rt` contains 0.
Note that divides take an undefined amount of time; other instructions will execute in parallel. MFHI and MFLO will interlock until the division is complete. This instruction never causes an exception.

J label *jump*

Jump to `label` [after executing the following instruction.]

JAL label *jump and link*

Jump to `label` [after executing the following instruction]. Places address of following instruction into `r31`. (This is a subroutine-call instruction.)

JALR rd, rs *jump and link, register*

Jump to address contained in `rs` [after executing the following instruction]. Places address of following instruction into `rd`. Note that `rs` and `rd` may not be the same register. If `rd` is omitted in the assembly language, it is register 31. (This is a subroutine-call instruction.)

JR rs *jump, register*

Jump to address contained in `rs` [after executing the following instruction].

LB rt, offset(rs) *load byte*

Sign-extend the 16-bit `offset` to 32 bits, and add it to `rs` to get an effective address. Load the byte from this address into `rt` and sign-extend it to fill the entire register.

LBU rt, offset(rs) *load byte, unsigned*

Sign-extend the 16-bit `offset` to 32 bits, and add it to `rs` to get an effective address. Load the byte from this address into `rt` and zero-extend it to fill the entire register.

LH rt, offset(rs) *load halfword*

Sign-extend the 16-bit `offset` to 32 bits, and add it to `rs` to get an effective address. Load the halfword (16 bits) from this address into `rt` and sign-extend it to fill the entire register. Exception if odd address.

LHU rt, offset(rs) *load halfword, unsigned*

Sign-extend the 16-bit `offset` to 32 bits, and add it to `rs` to get an effective address. Load the halfword (16 bits) from this address into `rt` and zero-extend it to fill the entire register. Exception if odd address.

LUI rt, immediate *load upper immediate*

Put 16-bit `imediate` in the top half of `rt` and fill the bottom half with zeros.

LW rt, offset(rs) *load word*

Sign-extend the 16-bit `offset` to 32 bits, and add it to `rs` to get an effective address. Load the word (32 bits) from this address into `rt`. Exception if address is not word-aligned.

MFHI rd *move from HI*

Move contents of special register `HI` into `rd`. Neither of the two instructions following this may modify the `HI` register! Note that multiplication and division put results into `HI`. `MFHI` stalls until that operation is complete.

MFLO rd *move from LO*

Move contents of special register `LO` into `rd`. Neither of the two instructions following this may modify the `LO` register! Note that multiplication and division put results into `LO`. `MFLO` stalls until that operation is complete.

MTHI rs *move to HI*

Move contents of `rs` into special register `HI`. May cause contents of `LO` to become undefined; no need to get specific here; just be sure to do `MTLO` too.

MTLO rs *move to LO*

Move contents of `rs` into special register `LO`. May cause contents of `HI` to become undefined; no need to get specific here; just be sure to do `MTHI` too.

MULT rs, rt *multiply*

Multiplies `rs` by `rt`, treating both as (signed) 2's complement numbers. Low word of result goes into special register LO and high word into special register HI. Get them via the MFHI and MFLO instructions. No overflow exception occurs.

Note that multiplies take an undefined amount of time; other instructions will execute in parallel. MFHI and MFLO will interlock until the multiplication is complete.

MULTU rs, rt *multiply, unsigned*

Multiplies `rs` by `rt`, treating both as unsigned numbers. Low word of result goes into special register LO and high word into special register HI.

Get them via the MFHI and MFLO instructions. No overflow exception occurs.
Note that multiplies take an undefined amount of time; other instructions will execute in parallel. MFHI and MFLO will interlock until the multiplication is complete. This instruction never causes an exception.

NOP *no-op*
Do nothing for one cycle; good for filling a delay slot.

NOR rd, rs, rt *nor*
Performs bitwise logical nor of `rs` and `rt`, putting result into `rd`.

OR rd, rs, rt *or*
Performs bitwise logical or of `rs` and `rt`, putting result into `rd`.

ORI rt, rs, immediate *or, immediate*
Zero-extends 16-bit `immediate` to 32 bits, and bitwise ors it with `rt`, putting result into `rd`.

SB rt, offset(rs) *store byte*
Sign-extend the 16-bit `offset` to 32 bits, and add it to `rs` to get an effective address. Store least significant byte from `rt` into this address.

SH rt, offset(rs) *store halfword*
Sign-extend the 16-bit `offset` to 32 bits, and add it to `rs` to get an effective address. Store least significant byte from `rt` into this address. Exception if odd address.

SLL rd, rs, immediate *shift left logical*
Shift contents of `rs` left by the amount indicated in `immediate` (0..31), inserting zeros into the low order bits. Put result into `rd`.

SLLV rd, rt, rs *shift left logical, variable*
Shift contents of `rt` left by the amount indicated in the bottom five bits of `rs` (0..31), inserting zeros into the low order bits. Put result into `rd`.

SLT rt, rs, rt *set on less-than*
If `rs` < `rt` with a signed comparison, put 1 into `rt`. Otherwise put 0 into `rt`.

SLTI rt, rs, immediate *set on less-than, immediate*
Sign-extend the 16-bit `immediate` to a 32-bit value. If `rs` < this value with a signed comparison, put 1 into `rt`. Otherwise put 0 into `rt`.

SLTIU rt, rs immediate *set on less-than, immediate unsigned*
Sign-extend the 16-bit `immediate` to a 32-bit value. If `rs` < this value with an unsigned comparison, put 1 into `rt`. Otherwise put 0 into `rt`.

SLTU rd, rs, rt *set on less-than, unsigned*
If `rs` < `rt` with an unsigned comparison, put 1 into `rd`. Otherwise put 0 into `rd`.

SRA rd, rt, immediate *shift right arithmetic*
Shift contents of `rs` right by the amount indicated in `immediate` (0..31), sign-extending the high order bits. Put result into `rd`.

SRAV rd, rt, rs *shift right arithmetic, variable*

Shift contents of `rt` right by the amount indicated in the bottom five bits of `rs` (0..31), sign-extending the high order bits. Put result into `rd`.

SRL rd, rt, immediate *shift right logical*

Shift contents of `rt` right by the amount indicated in `immediate` (0..31), zero-filling the high order bits. Put result into `rd`.

SRLV rd, rt, rs *shift right logical, variable*

Shift contents of `rt` right by the amount indicated in the bottom five bits of `rs` (0..31), zero-filling the high order bits. Put result into `rd`.

SUB rd, rs, rt *subtract*

Put `rs` – `rt` into `rd`. Exception if overflow.

SUBU rd, rs, rt *subtract unsigned*

Put `rs` – `rt` into `rd`. Never causes exception.

SW rt, offset(rs) *store word*

Sign-extend the 16-bit `offset` to 32 bits, and add it to `rs` to get an effective address. Store `rt` into this address. Exception if address is not word-aligned.

XOR rd, rs, rt *exclusive or*

Performs bitwise exclusive xor of `rs` and `rt`, putting result into `rd`.

XORI rt, rs, immediate *xor immediate*

Zero-extends 16-bit immediate to 32 bits, and bitwise exclusive xors it with `rs`, putting result into `rt`.

MIPS Example: Initializing an Array

This shows an assembly language program which initializes the (externally-defined) integer array *arr* such that each of its ten elements is equal to the index of that element. The program is in ~farrens/academic/50/samples/init1.s.

```
 #
 # This procedure initializes the (externally defined) array arr
 # so that the value stored in the i'th element is i.
 #
 # Name: init
 # Written by: Matt Bishop, bishop@cs.ucdavis.edu, February 27, 1995
 # Action: for the array arr, defined as int arr[10], set the i'th
 #          element to i.
 # Assumptions: This is called either by a C program compiled by the
 #              ULTRIX or GNU C compilers, or by something which uses
 #              the same calling sequence.
 #              The array is declared as int arr[10]
 # Input Parameters: none
 # Output Parameters: none
 # Registers used: $0 -- to get a 0 (standard usage)
 #                 $4 -- index of arr
 #                 $8 -- temporary (usually holds result of comparison)
 #                 $14 -- offset of current element from base of arr
 #                 $31 -- return address (standard usage)
 # Stack Usage:none
 # Side Effects: none
 #
         .extern arr 40          # declaration of arr
         .text                   # now the instructions
         .align  2               # word alignment
         .globl  init            # make init visible to all
         .ent    init 2          # it's also an entry point
         .set    noreorder       # do NOT reorder instructions
 #
 # it begins
 #
init:    addu    $4, $0, $0      # initialize index of array
 #
 # the loop
 #
top:     sll     $14, $4, 2      # go to offset of next element
         sw      $4, arr($14)    # load integer into element
         addiu   $4, $4, 1       # get next index
         slti    $8, $4, 10      # see if it's 10 yet
         bne     $8, $0, top     # nope -- go back for another
         nop                     # for the delay slot
 #
 # bye!
 #
         j       $31             # return
         nop                     # for the delay slot
         .end    init            # (syntactic) end of procedure
```