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.

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

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

Assembler Directives

.data [addr]: Indicates beginning of data section. If addr is provided, then the location counter is set to addr.
.text [addr]: Indicates beginning of code section. If addr is provided, then the location counter is set to addr.
.ascii <string>: Allocates memory for string of chars. Terminated with ‘\0’, and padded with ‘\0’ to word boundary.
.space <number>: Allocates <number> of bytes of memory. <number> is increased to word boundary.
.word [value]: Allocates a word. If value is provided, then the word is set to that value.
.end: Indicates end of program. (This is ignored.)
# MIPS-32 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

## 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 Zero
- **BLTZAL**  Branch on Less than Zero and Link
- **BGEZAL**  Branch on Zero and Link

## 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

### (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**  Bitwise And
- **OR**  Bitwise OR
- **XOR**  Bitwise exclusive OR
- **NOR**  NOR

## Shift and Special Instructions
- **SLL**  Shift Left Logical
- **SLLV**  Shift Left Logical, Variable
- **SRA**  Shift Right Arithmetic
- **SRAV**  Shift Right Arithmetic, Variable
- **SRL**  Shift Right Logical
- **SRLV**  Shift Right Logical, Variable
- **BREAK**  Break
- **SYSCALL**  System Call
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 many of the core MIPS instructions. Not all instructions are listed; in particular, those involving traps, floats, or memory management are omitted.

**ADD rd, rs, rt**  
add  
Opcode: 000000  Func: 100000  
Adds rs and rt, puts result into rd. Exception on overflow.

**ADDI rt, rs, immediate**  
add, immediate  
Opcode: 001000  
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  
Opcode: 001001  
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  
Opcode: 000000  Func: 100001  
Adds rs and rt, puts result into rd. Never causes an overflow.

**AND rd, rs, rt**  
and  
Opcode: 000000  Func: 100100  
Bitwise and’s rs and rt, puts result into rd.

**ANDI rt, rs, immediate**  
and, immediate  
Opcode: 001100  
Sign-extends the 16-bit immediate to 32 bits, bitwise ands it with rs, puts result into rt.

**BEQ rs, rt, offset**  
branch equal  
Opcode: 000100  
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  
Opcode: 000001  rt: 00001  
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  
Opcode: 000001  rt: 10001  
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  
Opcode: 000111  
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  
Opcode: 000110  
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*  
Opcode: 000001  
rt: 00000  
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*  
Opcode: 000001  
rt: 10000  
If rs < 0, branches to offset [after executing the following instruction]. For most assemblers, offsets a label. Always places address of following instruction into r31. Note that rs may not itself be r31. (This is a subroutine call function.)

**BNE rs, rt, label**  
*branch not-equal*  
Opcode: 000101  
If rs ≠ rt branches to offset [after executing the following instruction]. For most assemblers, offset is a label.

**BREAK**  
*break*  
Opcode: 000000  
func: 001101  
Causes a Breakpoint exception that transfers control to the exception handler.

**DIV rs, rt**  
*divide*  
Opcode: 000000  
func: 011010  
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*  
Opcode: 000000  
func: 011011  
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*  
Opcode: 000010  
Jump to label [after executing the following instruction].

**JAL label**  
*jump and link*  
Opcode: 000011  
Jump to label [after executing the following instruction]. Places the address of the following instruction into r31. (This is a subroutine-call instruction.)

**JALR rd, rs**  
*jump and link, register*  
Opcode: 000000  
func: 001001  
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*  
Opcode: 000000  
func: 001000  
Jump to address contained in rs [after executing the following instruction].

**LA rt, addr**  
*load address into register*  
Pseudo instruction  
This is a a pseudo instruction that is translated into:  
lui $rt, addr(16..31) followed by ori $rt, $rt, addr(0..15)

**LB rt, offset(rs)**  
*load byte*  
Opcode: 100000  
Sign-extend the 16-bit offsetto 32 bits, and add it to rsto 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**  
 Opcode: 100100  
 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**  
 Opcode: 100001  
 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**  
 Opcode: 100101  
 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.

**LI rt, immediate load a 32-bit immediate into a register**  
 Pseudo instruction  
 This is a a pseudo instruction that is translated into:  
 `lui $rt, immediate(16..31) followed by ori $rt, $rt, immediate(0..15)`

**LUI rt, immediate load upper immediate**  
 Opcode: 001111  
 Put 16-bit immediate in the top half of rt and fill the bottom half with zeros.

**LW rt, offset(rs) load word**  
 Opcode: 100011  
 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**  
 Opcode: 000000  
 Func: 010000  
 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**  
 Opcode: 000000  
 Func: 010010  
 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**  
 Opcode: 000000  
 Func: 010001  
 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**  
 Opcode: 000000  
 Func: 010011  
 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**  
 Opcode: 000000  
 Func: 011000  
 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 over-flow 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**  
 Opcode: 000000  
 Func: 011001  
 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  

Pseudo instruction  

Do nothing for one cycle; good for filling a delay slot. Assemblers often use sl $0, $0, 0.

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, rt, sa  

Shift left logical  

Shift contents of rt left by the amount indicated in sa, insertion zeroes into the emptied low order bits. Put the result into rd.

SLLV rd, rs, rt  

Shift left logical, variable  

Shift contents of rt left by the amount indicated in the bottom five bits of rs (0..4), inserting zeroes into the low order bits. Put result into rd.

SLT rd, rs, rt  

Set on less-than  

If rs < rt with a signed comparison, put 1 into rd. Otherwise put 0 into rd.

SLTI rt, rs, immediate  

Set on less-than, immediate  

Sign-extend the 16-bit immediate to a 32-bit value. If rs is less than 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 is less than 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 rt < rs with an unsigned comparison, put 1 into rd. Otherwise put 0 into rd.

SRA rd, rt, sa  

Shift right arithmetic  

Shift contents of rt right by the amount indicated by sa, sign-extending the high order bits. Put result into rd.

SRAV rd, rs, rt  

Shift right arithmetic, variable  

Shift contents of rt right by the amount indicated in the bottom five bits of rs (0..4), sign-extending the high order bits. Put result into rd.

SRL rd, rt, sa  

Shift right logical  

Shift contents of rt right by the amount indicated in sa, zero-filling the high order bits. Put result into rd.
SRLV rd, rs, rt  
*shift right logical, variable*

Shift contents of rt right by the amount indicated in the bottom five bits of rs(0..4), 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.

SYSCALL  
*system call*

Causes a System Call exception. For ECS 50 the response is based on the value in $v0. 1 = print_int, 10 = exit.

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 ors it with rs, putting result into rt.

**MIPS Example: Initializing an Array**

This shows an assembly language program which initializes the integer array *arr* such that each of its ten elements is equal to the index of that element. It also prints the value after it is inserted in the array.

```assembly
# Written by: Matt Bishop and adapted by Sean Davis
# Registers used:
#    $0 -- to get a 0 (standard usage)
#    $a0 -- to choose system service
#    $t0 -- index of arr
#    $t1 -- offset of current element from base of arr
#    $t2 -- temporary (usually holds result of comparison)
#
.data 0x40  # set start address of data section
arr:   .space 40  # allocate 10 words
       .text 0  # set start address of instructions
init:  addu $t0, $0, $0  # initialize index of array
loop:  sll $t1, $t0, 2  # go to offset of next element
       sw $t0, arr($t1)  # store integer into element
       add $a0, $t0, $0  # copy from $t0 to $a0
       addi $v0, $0, 1  # set $v0 to print_integer code for syscall
       syscall          # print the integer in $a0
       addiu $t0, $t0, 1  # add one to current array index
       slti $t2, $t0, 10  # see if the index is 10 yet
       bne $t2, $0, loop  # nope -- go back for another
       nop    # for the delay slot
       addiu $v0, $0, 10  # set $v0 to exit code for syscall
       syscall          # exit
.end
```
### MIPS32 Encoding of the Opcode Field

<table>
<thead>
<tr>
<th>opcode</th>
<th>bits 31..29</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td>SPECIAL δ</td>
</tr>
<tr>
<td>1 001</td>
<td>REGIMM δ</td>
</tr>
<tr>
<td>2 010</td>
<td>J</td>
</tr>
<tr>
<td>3 011</td>
<td>JAL</td>
</tr>
<tr>
<td>4 100</td>
<td>BEQ</td>
</tr>
<tr>
<td>5 101</td>
<td>BNE</td>
</tr>
<tr>
<td>6 110</td>
<td>BLEZ</td>
</tr>
<tr>
<td>7 111</td>
<td>BGTZ</td>
</tr>
</tbody>
</table>

### MIPS32 SPECIAL Opcode Encoding of Function Field

<table>
<thead>
<tr>
<th>function</th>
<th>bits 5..3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td>SLL¹</td>
</tr>
<tr>
<td>1 001</td>
<td>MOVZ</td>
</tr>
<tr>
<td>2 010</td>
<td>JMVI</td>
</tr>
<tr>
<td>3 011</td>
<td>DIV</td>
</tr>
<tr>
<td>4 100</td>
<td>ADD</td>
</tr>
<tr>
<td>5 101</td>
<td>MFHI</td>
</tr>
<tr>
<td>6 110</td>
<td>TGE</td>
</tr>
<tr>
<td>7 111</td>
<td>β</td>
</tr>
</tbody>
</table>

### MIPS32 REGIMM Encoding of rt Field

<table>
<thead>
<tr>
<th>rt</th>
<th>bits 18..16</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td>BLTZ</td>
</tr>
<tr>
<td>1 01</td>
<td>TGEI</td>
</tr>
<tr>
<td>2 10</td>
<td>BLTZAL</td>
</tr>
<tr>
<td>3 11</td>
<td>*</td>
</tr>
</tbody>
</table>