Guide to MIPS

(Definitely not taken from somewhere else)

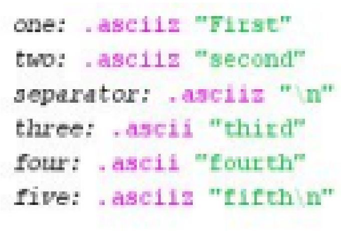
1. Data Types

In MIPS, the data are stored in the data segment (read the basics). While datas stored in the data segment are variables, they cannot be changed directly in MIPS language (more about that on registers and load/store instructions). More details about the data types in the table below:

Data Type	Value	Explanation
.ascii	string	Store string in memory **without** null terminator
.asciiz	string	Store string in memory **with** null terminator
.byte	b1, b2, ..., bn	Store n **bytes** in memory. Can be written in base 10, or hex. Each value is separated by a comma (,)
.halfword	h1, h2, ..., hn	Store n **halfword** in memory. Can be written in base 10, or hex. Each value is separated by a comma (,)
.word	w1, w2, ..., wn	Store n **word** in memory. Can be written in base 10, or hex. Each value is separated by a comma (,)
.space	n	**Reserves n bytes** of space in memory.

In MIPS, a byte is 8 bits. A halfword is 16 bits, and a word is 32 bits. Each character requires 1 byte (8 bits) of storage while integer requires 4 bytes (32 bits) of storage.

A null terminator for .asciiz is, rather than a new line (\n), is a zero (\0). If you want to use new lines, you must insert one in your string, or use another variable then print it. Examples below:



```
one: .asciiz "First"
two: .asciiz "second"
separator: .asciiz "\n"
three: .ascii "third"
four: .ascii "fourth"
five: .asciiz "fifth\n"
```

2. Register

A Register is, in simple terms, a place to hold a variable that you wanted to use. Sometimes a register is used as a function's storage for return values, so make sure the registers you want to use and the one for return values don't overlap.

Register Number	Conventional Name	Usage
$0	$zero	Constant value zero
$1	$at	Assembler Temporary
$2 - $3	$v0, $v1	Return value to functions
$4 - $7	$a0 - $a3	Argument for functions – **Not** preserved by subprogram
$8 - $15	$t0 - $t7	Temporary data register – **Not** preserved by subprogram
$16 - $23	$s0 - $s7	Saved register – Preserved by subprogram
$24 - $25	$t8 - $t9	More temporary register – **Not** preserved by subprogram
$26 - $27	$k0 - $k1	Reserved for kernel, do not use
$28	$gp	Global Area Pointer
$29	$sp	Stack Pointer
$30	$fp	Frame Pointer
$31	$ra	Return Address

You generally want to use register number 8-23 first. 16-23 is if you want your values to be preserved by subprogram. 4-7 is used for syscall arguments, and 2-3 is syscall return values.

More Registers

The registers here are used by certain instructions. More info about program counter later on J type instruction and branches.

Register Number	Conventional Name	Usage
–	pc	Program Counter
–	hi	hi and lo is used for several instruction (e.g. mult, div) for return value
–	lo	

Even more registers

This section of registers is usually used for floating points. If you don't use float or doubles, just ignore this section.

Register Number	Conventional Name	Usage
$f0 – $f3	–	Floating point return values
$f4 – $f10	–	Temporary registers – **Not** preserved by subprogram
$f12 – $f14	–	First two arguments to subprograms – **Not** preserved by subprogram
$f16 – $f18	–	More temporary registers – **Not** preserved by subprogram
$f20 – $f31	–	Saved registers – Preserved by subprogram

3. Instructions

In MIPS language, a program consists of lines of instructions. These instructions are usually something really simple like for example, adding two numbers, or branching.

Arithmetic and Logical Instructions

No need for explanations here. Just your regular everyday instructions for daily needs, such as additions, multiplications, and logical operations.

Instruction	Syntax	Operation
add	$d, $s, $t	$d = $s + $t
addi	$d, $s, i	$d = $s + i

and	$d, $s, $t	$d = $s AND $t
andi	$d, $s, i	$t = $s AND i
div	$s, $t	$s / $t; LO = quotient, HI = remainder
mult	$s, $t	LO = $s * $t
or	$d, $s, $t	$d = $s OR $t
nor	$d, $s, $t	$d = ~($s OR $t)
ori	$d, $s, i	$t = $s OR i
sll	$d, $t, a	$d = $t << a ($t shift left a times, zeroes are shifted in)
srl	$d, $t, a	$d = $t >> a ($t shift right a times, zeroes are shifted in)
sra	$d, $t, a	$d = $t >> a ($t shift right a times, the sign bit are shifted in)
sub	$d, $s, $t	$d = $s – $t
xor	$d, $s, $t	$d = $s ˆ $t
xori	$d, $s, i	$d = $s ˆ i

Note that in addi (or similar instructions), the i is an immediate value. So rather than X = Y + Z it's something like X = Y + 5. The a in shift instructions (sll, srl, sra) are the shift amount. Also note that the div and mult instruction uses HI and LO registers.

Branch Instructions

Branch instructions is used for, well, branching. Do note though that branching instruction's jump range is smaller than jump instructions.

Instruction	Syntax	Operation
beq	$s, $t, label	Go to label if $s == $t
bgtz	$s, label	Go to label if $s > 0
blez	$s, label	Go to label if $s <= 0
bne	$s, $t, label	Go to label if $s != $t

Jump Instructions

Jump instructions is used for immediate jump to part of the program without a condition.

Instruction	Syntax	Operation
j	label	Jump to label
jal	label	Jump to label, $31 (return address) = pc
jr	$ra	Jump to address ra stored (pc = ra)

Load Instructions

Load instructions is used for loading a variable in the .data segment (or anywhere, really) into the specified register.

Instruction	Syntax	Operation
lb	$t, i ($s)	$t = MEM [$s + i]. A byte is loaded into a register from the specified address. **(load byte)**
lw	$t, i ($s)	$t = MEM [$s + i]. A word is loaded into a register from the specified address. **(load word)**
la	$t, label	loads the address of label to register $t **(load address)**
li	$t, i	loads the immediate value to register $t **(load immediate)**

Unlike in regular programming language, the i in load instructions are **an immediate** and **cannot be other registers**. You can, though, modify the address in $s register if needed (for example, adding by 4 for every iteration).

Store Instructions

Store instructions is used for storing the data from a register to the .data segment (or anywhere). It is the inverse of load instruction, and thus are pretty similar in nature.

Instruction	Syntax	Operation
sb	$t, i ($s)	MEM[$s + offset] = (0xff & $t); The least significant byte of $t is stored at the specified address.
sw	$t, i ($s)	MEM [$s + i] = $t; The contents of $t is stored at the specified address.

Data Movement Instructions

Data movement instruction handles moving data from a register to another. The move hi and move lo is used for moving contents of register HI and LO, respectively. Usually used after mult or div (of course).

Instruction	Syntax	Operation
move	$d, $s	move contents of $s to $d
mfhi	$d	The contents of register HI are moved to the specified register.
mflo	$d	The contents of register LO are moved to the specified register.

4. System call (Syscall)

Syscall is used when you want to do things with the hardware, usually something regarding I/O (input output). A syscall reads the service code it must perform from the $v0 register and sometimes uses $a or $f registers for its argument. Here are some of the service codes:

Service	Code	Arguments	Result
print integer	1	$a0 = value	(none)
print float	2	$f12 = float value	(none)
print double	3	$f12 = double value	(none)
print string	4	$a0 = address of string	(none)
read integer	5	(none)	$v0 = value read
read float	6	(none)	$f0 = value read
read double	7	(none)	$f0 = value read
read string	8	$a0 = address where string to be stored $a1 = number of characters to read + 1	(none)
memory allocation	9	$a0 = number of bytes of storage desired	$v0 = address of block
exit (end of program)	10	(none)	(none)
print character	11	$a0 = integer	(none)
read character	12	(none)	char in $v0

REMEMBER to always end your program with syscall code 10 (exit).

For more info regarding MIPS language, see MIPS Green Sheet.

Created by: Adrian Kaiser – Logical (POK Gasal 2020/2021)

Source:
- https://sweetcode.io/building-first-simple-program-mips-assembly-language/
- https://en.wikibooks.org/wiki/MIPS_Assembly/Pseudoinstructions
- https://inst.eecs.berkeley.edu/~cs61c/resources/MIPS_Green_Sheet.pdf