

Simple MIPS Instructions

How Simple?

MIPS has a few instructions that resemble C instructions. One way to visualize how MIPS instructions work is by visualizing the equivalent C code. In addition to understanding how MIPS instructions work, you must be aware of a few limitations and warnings to programming in the MIPS assembly language (the ones not found in C).

The Registers (Active Memory)

In MIPS, you cannot simply operate on information directly from the computer's main memory, but must first load it into special local memory called registers. Registers are generally faster than the main memory because there is less of it, and it resides directly on the MIPS CPU (Central Processing Unit). MIPS has 32 user accessible registers used in programming. One of the registers, the \$0 or \$zero register can only have a value of zero. For this reason you cannot store any data in it (although you can write to it). The rest of the registers are general purpose registers, usable for any nonspecific instructions.

Although the registers other than \$zero are for general use, to make programming for MIPS easier, certain conventions¹ have been created regarding the 31 general purpose registers. The table shown below roughly outlines how all of the registers are supposed to be used²:

Register names	Register numbers	Usage
\$zero	0	constant 0
\$at	1	reserved for assembler
\$v0 - \$v1	2 - 3	expression evaluation and results of a function
\$a0 - \$a3	4 - 7	arguments 1 through 4
\$t0 - \$t7	8 - 15	temporary (not preserved across call)
\$s0 - \$s7	16 - 23	saved temporary (preserved across call)
\$l8 - \$l9	24 - 25	temporary (not preserved across call)
\$k0 - \$k1	26 - 27	reserved for OS kernel
\$gp	28	pointer to global area
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address (used by function call)

When programming in assembly language, you should always use the "register names" instead of the "register numbers". This is done in addition to comments and good organization to clarify your code.

As you learn more about programming in MIPS's assembly language, you should also learn about the specific functions of the registers that go along with it. At this point in your understanding, you only need to know that the registers \$t0 - \$t9 and \$s0 - \$s7 are free for use. I will clarify the proper use of all the registers later.

¹ I'm covering the conventions outlined in Patterson and Hennessy's book, *Computer Organization & Design: The Hardware/Software Interface*.

² A similar table can be found in Patterson and Hennessy's book on page A-23.

From now on when dealing with MIPS, any word that has a dollar sign, '\$', before it is a register or variable in C. Anything lacking the dollar sign is just a number or numerical value. This works well because in assembly language, although you type in numbers for constants, sometimes you just type in a word that represents a numerical value. This word is called a *label* that indicates a particular location in a program's code or data space.

Simple Mathematics

The most similar of assembly instructions are those based on simple mathematical operations. These are addition and subtraction. In fact, the use of addition and subtraction for more than just addition and subtraction has to be carefully implemented when programming using MIPS.

Let's start with the register addition and subtraction. In C, you can add two variables together and make a third variable equal to the result. You can do the same in MIPS!

```
add $dest, $src1, $src2      # $dest = $src1 + $src2; /* signed */
sub $dest, $src1, $src2     # $dest = $src1 - $src2; /* signed */
```

Notice that this does not allow the use of constants, only operations involving registers. If a value isn't loaded into a register, then you can't make use of it. The \$zero register becomes handy with add and subtract. Used in \$add, it can be used to copy a value in one register to another, or negate a register's value.

```
add $dest, $zero, $src      # $dest = $src; /* signed */
sub $dest, $zero, $src     # $dest = -$src; /* signed */
```

In all MIPS instructions, you can use a register more than once, so you can make a register equal to zero as follows:

```
sub $dest, $dest, $dest    # $dest -= $dest; /* signed */
```

We often have constant values we want to add to a stored value in C. MIPS also has an instruction for this commonly occurring event in code. It's an *add immediate* instruction. It adds a 16 bit signed³ integer to the register given. This is also useful just for loading registers with 16 bit values.

```
addi $dest, $src, number   # $dest = $src + number; /* signed */
addi $dest, $zero, number  # $dest = number; /* signed */
```

There are no subtract immediate instructions because the constant number is allowed to be signed.

³ Signed means the microprocessor will allow it to have proper positive and negative values. It's not important to know how this is done at this point in the course.

There are times when you don't want to add something and have a negative number. When you add signed, you only get half of the range of possible positive values as you would get if it were unsigned. To handle the cases where you could care less about having negative numbers, MIPS gives unsigned versions of the instructions covered so far:

```
addu $dest, $src1, $src2      # $dest = $src1 + $src2; /* unsigned */
subu $dest, $src1, $src2     # $dest = $src1 - $src2; /* unsigned */
addiu $dest, $src, number;   /* unsigned */
```

To copy a 32 bit value into register, another MIPS instruction is needed. *Load upper immediate* replaces the upper 16 bits of a register with the value given. Two instructions are needed to properly load a register with a 32 bit number. For right now, just assume that *ori* works properly for loading a 32 bit value into a register.

```
lui $dest, number           # $dest = ($dest & 0xFFFF0000) | (number << 16);
lui $dest, num_upper       #
ori $dest, $dest, num_lower # $dest = num;
```

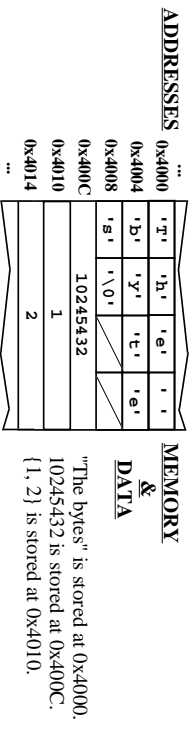
Memory Operations

What good is it if you can only use 31 variables when you program? Sometimes you need arrays of values, or need a place to store a value when you run out of the 31 available general purpose registers and need to use a single register for more than one variable. We need to access the computer's main memory!

MIPS does this by using pointers to locations in memory where you store values. When you *load* in a value from memory, you must say where it is, and when you *store* a value into memory, you must also say where to put it. For these two operations, MIPS has two instructions. One operates on bytes⁴, and the other on words⁵. You can use *load word*, *store word*, *load byte* *unsigned*, and *store byte*.

```
lw $dest, offset($src)      # $dest = *($src + offset); /* int pointer */
sw $src, offset($dest)     # *($dest + offset) = $src; /* int pointer */
lbu $dest, offset($src)    # $dest = *($src + offset); /* char pointer */
sb $src, offset($dest)     # *($dest + offset) = $src; /* char pointer */
```

The value of an address tells where our values are in bytes (8 bit increments), however MIPS can only read memory in word intervals. This means that an address (unless you use one of the special instructions for loading or storing bytes) must be a multiple of four. An address that isn't a multiple of four can lead to undesirable results.



⁴ A byte is 8 bits.
⁵ A word in MIPS is 4 bytes (32 bits).