MIPS Assembly Language

MIPS Registers

MIPS assembly language is a 3-address assembly language. Operands are either immediates or in registers.

There are 32 registers that we commonly use. Each is 32 bits wide. The registers are identified by a integer, numbered 0 - 31.

To reference a register as an operand, use the syntax
`$x`, where x is the number of the register you want.
examples: `$12`, `$15`

Some limitations on the use of the 32 32-bit registers. Due to conventions set by the simulator, and by the architecture, certain registers are used for special purposes. It is wise to avoid the use of those registers, until you understand how to use them properly.

- `$0`
  is 0 (use as needed)
- `$1`
  is used by the assembler (the simulator in our case) -- do **not** use it in your programs.
- `$2-7`
  are used by the simulator -- do not use them unless you know what they are for and how they are used. (They can be used as a place to pass parameters, return the result from a function, and are implied to be used for passing values in the implementation of MAL I/O instructions.)
- `$26-27`
  Used to implement the mechanism for calling special procedures that do I/O and take care of other error conditions (like overflow)
- `$29`
  is a stack pointer -- you are automatically allocated a stack (of words), and $29 is initialized to contain the address of the empty word at the top of the stack at the start of any program.

Instruction Format and Syntax

Arithmetic and Logical Operations

This is not a complete list of instructions; these are instructions you should know about, and be able to use when appropriate.

```
mnemonic number     operands      C or C++ or Java
         of operands

move       2        d, s1         d = s1;
add        3        d, s1, s2     d = s1 + s2; two's complement
addu       3        d, s1, s2     d = s1 + s2; unsigned
sub        3        d, s1, s2     d = s1 - s2; two's complement
subu       3        d, s1, s2     d = s1 - s2; unsigned
mul        3        d, s1, s2     d = s1 * s2; two's complement
div        3        d, s1, s2     d = s1 / s2;  gives quotient
divu       3        d, s1, s2     d = s1 / s2;  gives quotient
rem        3        d, s1, s2     d = s1 % s2;  gives remainder
remu       3        d, s1, s2     d = s1 % s2;  gives remainder
and        3        d, s1, s2     d = s1 & s2; bitwise AND
or         3        d, s1, s2     d = s1 | s2; bitwise OR
not        2        d, s1         d = ~s1;  bitwise complement
nand       3        d, s1, s2     d = s1 NAND s2; no C equivalent
```

```
nor         3         d, s1, s2     d = s1 NOR s2; no C equivalent
xor         3         d, s1, s2     d = s1 ^ s2; bitwise XOR
rol         3         d, s1, s2     d = rotate left of s1 by s2 places
ror         3         d, s1, s2     d = rotate right of s1 by s2 places
sll         3         d, s1, s2     d = logical left shift of s1 by s2 places
sra         3         d, s1, s2     d = arithmetic right shift of s1 by s2 places
srl         3         d, s1, s2     d = logical right shift of s1 by s2 places
```

NOTES:

1. For all 3 operand instructions, where only 2 appear in the source code, the first operand is both a source and the destination of the result.
2. cannot increase the number of operands.
3. `d` is always a register.
4. `s2` can be be a register or an **immediate** (a constant in the machine code)

Examples:

```
      move  $4, $9        # copy contents of $9 into $4

      mul   $12, $13, $14 # place 32-bit product of $13 and $14 into $12
                          # does not work correctly if result requires
                          # more than 32 bits

      add   $8, $9, $10   # two's complement sum of $9 and $10 placed in $8

      add   $20, $20, 1   # add (immediate value) 1 to the value in $20,
                          # result goes to $20
```

Note that there are other MAL arithmetic and logical instructions, but this list is sufficient for now.

On a real processor, an integer division operation gives two results: the quotient and the remainder. What happens with these two results differs among architectures.

Load and Store Instructions

On a load/store architecture, the only instructions that specify operands that come from or go to memory are **loads** and **stores**.

```
mnemonic number     operands      operation
         of operands

lw         2        d, addr       a word is loaded from addr and placed into d;
                                  the addr must be word aligned
lb         2        d, addr       a byte is loaded from addr and placed into
                                  the rightmost byte of d;
                                  sign extension defines the other bits of d
lbu        2        d, addr       a byte is loaded from addr and placed into
                                  the rightmost byte of d;
                                  zero extension defines the other bits of d
li         2        d, immed      the immediate value is placed into d

sw         2        d, addr       a word in d is stored to addr;
                                  the addr must be word aligned
sb         2        d, addr       a byte in the rightmost byte of d is stored to addr

la         2        d, label      the address assigned to label is placed into d
```

Notes:

1. `addr` is specified within the source code in one of 3 ways:

- `displacement(reg)`
  The immediate value (`displacement`) is added to the contents of the `reg` to form an effective address.
- `(reg)`
  The contents of the `reg` is the address.
- `label`
  The address is as assigned to `label`.

2. `d` must be a register specification

Examples:

```
la    $10, x      # place the address assigned for label x into register $10

lw    $8, x       # load the word from memory at address x into register $8

sb    $9, y       # store the contents of register $9 to memory at address y

lb    $8, ($12)   # load the byte from the address given by the contents
                  # of register $12 into the least significant byte of
                  # register $8, sign extending to define the other bits

sw    $10, 8($9)  # store the contents of register $10 to the address
                  # obtained by adding the value 8 to the contents of
                  # register $9
```

Control Instructions

Control instructions are the branches and/or jumps.

The MIPS architecture implements only a subset of these, and tasks the assembler to synthesize those *not* included in the instruction set from those included. From this list, you cannot distinguish which are synthesized from those that are included.

mnemonic	number of operands	operands	operation
b	1	label	unconditional branch to label
beq	3	r1, r2, label	branch to label if (r1) == (r2)
bne	3	r1, r2, label	branch to label if (r1) != (r2)
bgt	3	r1, r2, label	branch to label if (r1) > (r2)
bge	3	r1, r2, label	branch to label if (r1) >= (r2)
blt	3	r1, r2, label	branch to label if (r1) < (r2)
ble	3	r1, r2, label	branch to label if (r1) <= (r2)
beqz	2	r1, label	branch to label if (r1) == 0
bnez	2	r1, label	branch to label if (r1) != 0
bgtz	2	r1, label	branch to label if (r1) > 0
bgez	2	r1, label	branch to label if (r1) >= 0
bltz	2	r1, label	branch to label if (r1) < 0
blez	2	r1, label	branch to label if (r1) <= 0
j	1	label	unconditional jump to label
jal	1	label	unconditional jump to label, return address in $31

Examples:

```
beq   $8, $14, do_else  # branch to do_else if the contents of register $8
                        # is the same as the contents of register $14
```

Input and Output Instructions

Input and output are implemented by the simulator that we are using for the course. They are not actually instructions, but ways for the source code (instructions) to cause the simulator to do I/O functionality. Since the simulator was written in C, the easiest I/O functionality to mimic are simple, standard C I/O functions.

```
mnemonic number     operands        operation
         of operands

putc         1        r1            print the ASCII character in the least
                                    significant byte of r1
getc         1        r1            read a character, placing it in the least
                                    significant byte of r1
puts         1        r1/label      print the null-terminated string that begins
                                    at the address within r1 or given by label
```

Examples:

```
     .data
     str1:  .asciiz  "hello."
     .text

     puts  str1


      -------------
      |hello.
      |      ^
             more output starts here, when more is printed




     .data
     str1:  .asciiz  "hello."
     .text

     la    $12, str1    # address of first character in string
     puts  $12          # address of first character to print is in $12


      -------------
      |hello.
      |      ^
             more output starts here, when more is printed




     .data
     str1:  .asciiz  "hello.\nMy name is George."
     .text

     puts  str1

      -------------
      |hello.
      |My name is George.
      |                  ^
                         more output starts here, when more is printed




     .data
     str1:  .ascii   "Hi.\n"
     str2:  .asciiz  "I am a badger."
     .text

     puts  str1

      -------------
      |Hi.
      |I am a badger.
      |              ^
                     more output starts here, when more is printed
```

Explanation: The declarations just place the characters into memory. The characters are contiguous. The `.ascii` directive does *not* add a null terminating character to the string. The `puts` instruction knows to stop printing characters when it encounters the NULL character ('\0'). It does not find the null character at the end of the first string, so it just keeps going (through the second string) until it encounters the null character at the end of the second string.