Chapter 4

The MIPS R2000 Instruction Set

by Daniel J. Ellard

4.1 A Brief History of RISC

In the beginning of the history of computer programming, there were no high-level languages. All programming was initially done in the native machine language and later the native assembly language of whatever machine was being used.

Unfortunately, assembly language is almost completely nonportable from one architecture to another, so every time a new and better architecture was developed, every program anyone wanted to run on it had to be rewritten almost from scratch. Because of this, computer architects tried hard to design systems that were backward-compatible with their previous systems, so that the new and improved models could run the same programs as the previous models. For example, the current generation of PC-clones are compatible with their 1982 ancestors, and current IBM 390-series machines will run the same software as the legendary IBM mainframes of the 1960's.

To make matters worse, programming in assembly language is time-consuming and difficult. Early software engineering studies indicated that programmers wrote about as many lines of code per year *no matter what language they used*. Therefore, a programmer who used a high-level language, in which a single line of code was equivalent to five lines of assembly language code, could be about five times more productive than a programmer working in assembly language. It's not surprising, therefore, that a great deal of energy has been devoted to developing high-level languages where a single statement might represent dozens of lines of assembly language, and will run without modification on many different computers.

By the mid-1980s, the following trends had become apparent:

- Few people were doing assembly language programming any longer if they could possibly avoid it.
- Compilers for high-level languages only used a fraction of the instructions available in the assembly languages of the more complex architectures.
- Computer architects were discovering new ways to make computers faster, using techniques that would be difficult to implement in existing architectures.

At various times, experimental computer architectures that took advantage of these trends were developed. The lessons learned from these architectures eventually evolved into the *RISC* (Reduced Instruction Set Computer) philosophy.

The exact definition of RISC is difficult to state[1], but the basic characteristic of a RISC architecture, from the point of view of an assembly language programmer, is that the instruction set is relatively small and simple compared to the instruction sets of more traditional architectures (now often referred to as *CISC*, or Complex Instruction Set Computers).

The MIPS architecture is one example of a RISC architecture, but there are many others.

4.2 MIPS Instruction Set Overview

In this and the following sections we will give details of the MIPS architecture and SPIM environment sufficient for many purposes. Readers who want even more detail should consult *SPIM S20: A MIPS R2000 Simulator* by James Larus, *Appendix A, Computer Organization and Design* by David Patterson and John Hennessy (this appendix is an expansion of the SPIM S20 document by James Larus), or *MIPS R2000 RISC Architecture* by Gerry Kane.

The MIPS architecture is a register architecture. All arithmetic and logical operations involve only registers (or constants that are stored as part of the instructions). The MIPS architecture also includes several simple instructions for loading data from memory into registers and storing data from registers in memory; for this reason, the

[1]It seems to be an axiom of Computer Science that for every known definition of RISC, there exists someone who strongly disagrees with it.

MIPS architecture is called a *load/store* architecture. In a load/store (or *load and store*) architecture, the only instructions that can access memory are the *load* and *store* instructions– all other instructions access only registers.

4.3 The MIPS Register Set

The MIPS R2000 CPU has 32 registers. 31 of these are general-purpose registers that can be used in any of the instructions. The last one, denoted register `zero`, is defined to contain the number zero at all times.

Even though any of the registers can theoretically be used for any purpose, MIPS programmers have agreed upon a set of guidelines that specify how each of the registers should be used. Programmers (and compilers) know that as long as they follow these guidelines, their code will work properly with other MIPS code.

Symbolic Name	**Number**	**Usage**
zero	0	Constant 0.
at	1	Reserved for the assembler.
v0 - v1	2 - 3	Result Registers.
a0 - a3	4 - 7	Argument Registers 1 $\cdots$ 4.
t0 - t9	8 - 15, 24 - 25	Temporary Registers 0 $\cdots$ 9.
s0 - s7	16 - 23	Saved Registers 0 $\cdots$ 7.
k0 - k1	26 - 27	Kernel Registers 0 $\cdots$ 1.
gp	28	Global Data Pointer.
sp	29	Stack Pointer.
fp	30	Frame Pointer.
ra	31	Return Address.

4.4 The MIPS Instruction Set

This section briefly describes the MIPS assembly language instruction set.

In the description of the instructions, the following notation is used:

- If an instruction description begins with an $\circ$, then the instruction is not a member of the native MIPS instruction set, but is available as a *pseudoinstruction*. The assembler translates pseudoinstructions into one or more native instructions (see section 4.7 and exercise 4.8.1 for more information).

- If the op contains a `(u)`, then this instruction can either use signed or unsigned arithmetic, depending on whether or not a `u` is appended to the name of the instruction. For example, if the op is given as `add(u)`, then this instruction can either be `add` (add signed) or `addu` (add unsigned).
- *des* must always be a register.
- *src1* must always be a register.
- *reg2* must always be a register.
- *src2* may be either a register or a 32-bit integer.
- *addr* must be an address. See section 4.4.4 for a description of valid addresses.

4.4.1 Arithmetic Instructions

	Op	Operands	Description
○	`abs`	*des, src1*	*des* gets the absolute value of *src1*.
	`add(u)`	*des, src1, src2*	*des* gets *src1* + *src2*.
	`and`	*des, src1, src2*	*des* gets the bitwise and of *src1* and *src2*.
	`div(u)`	*src1, reg2*	Divide *src1* by *reg2*, leaving the quotient in register `lo` and the remainder in register `hi`.
○	`div(u)`	*des, src1, src2*	*des* gets *src1* / *src2*.
○	`mul`	*des, src1, src2*	*des* gets *src1* × *src2*.
○	`mulo`	*des, src1, src2*	*des* gets *src1* × *src2*, with overflow.
	`mult(u)`	*src1, reg2*	Multiply *src1* and *reg2*, leaving the low-order word in register `lo` and the high-order word in register `hi`.
○	`neg(u)`	*des, src1*	*des* gets the negative of *src1*.
	`nor`	*des, src1, src2*	*des* gets the bitwise logical **nor** of *src1* and *src2*.
○	`not`	*des, src1*	*des* gets the bitwise logical negation of *src1*.
	`or`	*des, src1, src2*	*des* gets the bitwise logical **or** of *src1* and *src2*.
○	`rem(u)`	*des, src1, src2*	*des* gets the remainder of dividing *src1* by *src2*.
○	`rol`	*des, src1, src2*	*des* gets the result of rotating left the contents of *src1* by *src2* bits.
○	`ror`	*des, src1, src2*	*des* gets the result of rotating right the contents of *src1* by *src2* bits.
	`sll`	*des, src1, src2*	*des* gets *src1* shifted left by *src2* bits.
	`sra`	*des, src1, src2*	Right shift arithmetic.
	`srl`	*des, src1, src2*	Right shift logical.
	`sub(u)`	*des, src1, src2*	*des* gets *src1* - *src2*.
	`xor`	*des, src1, src2*	*des* gets the bitwise exclusive **or** of *src1* and *src2*.

4.4.2 Comparison Instructions

	Op	Operands	Description
○	`seq`	*des, src1, src2*	*des* ← 1 if $src1 = src2$, 0 otherwise.
○	`sne`	*des, src1, src2*	*des* ← 1 if $src1 \neq src2$, 0 otherwise.
○	`sge(u)`	*des, src1, src2*	*des* ← 1 if $src1 \geq src2$, 0 otherwise.
○	`sgt(u)`	*des, src1, src2*	*des* ← 1 if $src1 > src2$, 0 otherwise.
○	`sle(u)`	*des, src1, src2*	*des* ← 1 if $src1 \leq src2$, 0 otherwise.
	`stl(u)`	*des, src1, src2*	*des* ← 1 if $src1 < src2$, 0 otherwise.

4.4.3 Branch and Jump Instructions

4.4.3.1 Branch

	Op	Operands	Description
	`b`	*lab*	Unconditional branch to *lab*.
	`beq`	*src1, src2, lab*	Branch to *lab* if *src1* ≡ *src2*.
	`bne`	*src1, src2, lab*	Branch to *lab* if *src1* ≠ *src2*.
○	`bge(u)`	*src1, src2, lab*	Branch to *lab* if *src1* ≥ *src2*.
○	`bgt(u)`	*src1, src2, lab*	Branch to *lab* if *src1* > *src2*.
○	`ble(u)`	*src1, src2, lab*	Branch to *lab* if *src1* ≤ *src2*.
○	`blt(u)`	*src1, src2, lab*	Branch to *lab* if *src1* < *src2*.
○	`beqz`	*src1, lab*	Branch to *lab* if *src1* ≡ 0.
○	`bnez`	*src1, lab*	Branch to *lab* if *src1* ≠ 0.
	`bgez`	*src1, lab*	Branch to *lab* if *src1* ≥ 0.
	`bgtz`	*src1, lab*	Branch to *lab* if *src1* > 0.
	`blez`	*src1, lab*	Branch to *lab* if *src1* ≤ 0.
	`bltz`	*src1, lab*	Branch to *lab* if *src1* < 0.
	`bgezal`	*src1, lab*	If *src1* ≥ 0, then put the address of the next instruction into `$ra` and branch to *lab*.
	`bgtzal`	*src1, lab*	If *src1* > 0, then put the address of the next instruction into `$ra` and branch to *lab*.
	`bltzal`	*src1, lab*	If *src1* < 0, then put the address of the next instruction into `$ra` and branch to *lab*.

4.4.3.2 Jump

Op	Operands	Description
`j`	*label*	Jump to label *lab*.
`jr`	*src1*	Jump to location *src1*.
`jal`	*label*	Jump to label *lab*, and store the address of the next instruction in `$ra`.
`jalr`	*src1*	Jump to location *src1*, and store the address of the next instruction in `$ra`.

4.4.4 Load, Store, and Data Movement

The second operand of all of the load and store instructions must be an address. The MIPS architecture supports the following addressing modes:

	Format	Meaning
○	*(reg)*	Contents of *reg*.
○	*const*	A constant address.
	const(reg)	*const* + contents of *reg*.
○	*symbol*	The address of *symbol*.
○	*symbol+const*	The address of *symbol* + *const*.
○	*symbol+const(reg)*	The address of *symbol* + *const* + contents of *reg*.

4.4.4.1 Load

The load instructions, with the exceptions of `li` and `lui`, fetch a byte, halfword, or word from memory and put it into a register. The `li` and `lui` instructions load a constant into a register.

All load addresses must be *aligned* on the size of the item being loaded. For example, all loads of halfwords must be from even addresses, and loads of words from addresses cleanly divisible by four. The `ulh` and `ulw` instructions are provided to load halfwords and words from addresses that might not be aligned properly.

	Op	Operands	Description
○	`la`	*des, addr*	Load the address of a label.
	`lb(u)`	*des, addr*	Load the byte at *addr* into *des*.
	`lh(u)`	*des, addr*	Load the halfword at *addr* into *des*.
○	`li`	*des, const*	Load the constant *const* into *des*.
	`lui`	*des, const*	Load the constant *const* into the upper halfword of *des*, and set the lower halfword of *des* to 0.
	`lw`	*des, addr*	Load the word at *addr* into *des*.
	`lwl`	*des, addr*	
	`lwr`	*des, addr*	
○	`ulh(u)`	*des, addr*	Load the halfword starting at the (possibly unaligned) address *addr* into *des*.
○	`ulw`	*des, addr*	Load the word starting at the (possibly unaligned) address *addr* into *des*.

4.4.4.2 Store

The store instructions store a byte, halfword, or word from a register into memory.

Like the load instructions, all store addresses must be *aligned* on the size of the item being stored. For example, all stores of halfwords must be from even addresses, and loads of words from addresses cleanly divisible by four. The `swl`, `swr`, `ush` and `usw` instructions are provided to store halfwords and words to addresses which might not be aligned properly.

	Op	Operands	Description
	`sb`	*src1, addr*	Store the lower byte of register *src1* to *addr*.
	`sh`	*src1, addr*	Store the lower halfword of register *src1* to *addr*.
	`sw`	*src1, addr*	Store the word in register *src1* to *addr*.
	`swl`	*src1, addr*	Store the upper halfword in *src* to the (possibly unaligned) address *addr*.
	`swr`	*src1, addr*	Store the lower halfword in *src* to the (possibly unaligned) address *addr*.
○	`ush`	*src1, addr*	Store the lower halfword in *src* to the (possibly unaligned) address *addr*.
○	`usw`	*src1, addr*	Store the word in *src* to the (possibly unaligned) address *addr*.

4.4.4.3 Data Movement

The data movement instructions move data among registers. Special instructions are provided to move data in and out of special registers such as `hi` and `lo`.

	Op	Operands	Description
○	`move`	*des, src1*	Copy the contents of *src1* to *des*.
	`mfhi`	*des*	Copy the contents of the `hi` register to *des*.
	`mflo`	*des*	Copy the contents of the `lo` register to *des*.
	`mthi`	*src1*	Copy the contents of the *src1* to `hi`.
	`mtlo`	*src1*	Copy the contents of the *src1* to `lo`.

4.4.5 Exception Handling

Op	Operands	Description
`rfe`		Return from exception.
`syscall`		Makes a system call. See 4.6.1 for a list of the SPIM system calls.
`break`	*const*	Used by the debugger.
`nop`		An instruction which has no effect (other than taking a cycle to execute).

4.5 The SPIM Assembler

4.5.1 Segment and Linker Directives

Name	Parameters	Description
`.data`	*addr*	The following items are to be assembled into the data segment. By default, begin at the next available address in the data segment. If the optional argument *addr* is present, then begin at *addr*.
`.text`	*addr*	The following items are to be assembled into the text segment. By default, begin at the next available address in the text segment. If the optional argument *addr* is present, then begin at *addr*. In SPIM, the only items that can be assembled into the text segment are instructions and words (via the `.word` directive).
`.kdata`	*addr*	The kernel data segment. Like the data segment, but used by the Operating System.
`.ktext`	*addr*	The kernel text segment. Like the text segment, but used by the Operating System.
`.extern`	*sym size*	Declare as global the label *sym*, and declare that it is *size* bytes in length (this information can be used by the assembler).
`.globl`	*sym*	Declare as global the label *sym*.

4.5.2 Data Directives

Name	Parameters	Description
`.align`	*n*	Align the next item on the next 2^n-byte boundary. `.align 0` turns off automatic alignment.
`.ascii`	*str*	Assemble the given string in memory. Do not null-terminate.
`.asciiz`	*str*	Assemble the given string in memory. Do null-terminate.
`.byte`	*byte1* ··· *byteN*	Assemble the given bytes (8-bit integers).
`.half`	*half1* ··· *halfN*	Assemble the given halfwords (16-bit integers).
`.space`	*size*	Allocate *n* bytes of space in the current segment. In SPIM, this is only permitted in the data segment.
`.word`	*word1* ··· *wordN*	Assemble the given words (32-bit integers).

4.6 The SPIM Environment

4.6.1 SPIM syscalls

Service	Code	Arguments	Result
`print_int`	1	`$a0`	*none*
`print_float`	2	`$f12`	*none*
`print_double`	3	`$f12`	*none*
`print_string`	4	`$a0`	*none*
`read_int`	5	*none*	`$v0`
`read_float`	6	*none*	`$f0`
`read_double`	7	*none*	`$f0`
`read_string`	8	`$a0` (address), `$a1` (length)	*none*
`sbrk`	9	`$a0` (length)	`$v0`
`exit`	10	*none*	*none*

4.7 The Native MIPS Instruction Set

Many of the instructions listed here are not native MIPS instructions. Instead, they are *pseudoinstructions*– macros that the assembler knows how to translate into native

MIPS instructions. Instead of programming the “real” hardware, MIPS programmers generally use the *virtual machine* implemented by the MIPS assembler, which is much easier to program than the native machine.

For example, in most cases, the SPIM assembler will allow *src2* to be a 32-bit integer constant. Of course, since the MIPS instructions are all exactly 32 bits in length, there’s no way that a 32-bit constant can fit in a 32-bit instruction word and have any room left over to specify the operation and the operand registers! When confronted with a 32-bit constant, the assembler uses a table of rules to generate a sequence of native instructions that will do what the programmer has asked.

The assembler also performs some more intricate transformations to translate your programs into a sequence of native MIPS instructions, but these will not be discussed in this text.

By default, the SPIM environment implements the same virtual machine that the MIPS assembler uses. It also implements the bare machine, if invoked with the `-bare` option enabled.

4.8 Exercises

4.8.1

Many of the instructions available to the MIPS assembly language programmer are not really instructions at all, but are translated by the assembler into one or more instructions.

For example, the `move` instruction can be implemented using the `add` instruction. Making use of register `$0`, which always contains the constant zero, and the fact that the for any number x, $x + 0 \equiv x$, we can rewrite

```
move    des, src1
```

as

```
add     des, src1, $0
```

Similarly, since either the *exclusive or* or *inclusive or* of any number and 0 gives the number, we could also write this as either of the following:

```
or      des, src1, $0
xor     des, src1, $0
```

Show how you could implement the following instructions, using other instructions in the native MIPS instruction set:

1. `rem` *des, src1, src2*
2. `mul` *des, src1, src2*
3. `li` *des, const*
4. `lui` *des, const*

Keep in mind that the register `$at` is reserved for use by the assembler, so you can feel free to use this register for scratch space. You *must not* clobber any other registers, however.

Chapter 5

MIPS Assembly Code Examples

by Daniel J. Ellard

The following sections include the source code for several of the programs referenced by the tutorial. All of this source code is also available online.

For the convenience of the reader, the source code is listed here along with line numbers in the left margin. These line numbers do not appear in the original code, and it would be an error to include them in your own code.

5.1 add2.asm

This program is described in section 2.4.

```
## Daniel J. Ellard -- 02/21/94
## add2.asm-- A program that computes and prints the sum
##      of two numbers specified at runtime by the user.
## Registers used:
##      $t0     - used to hold the first number.
##      $t1     - used to hold the second number.
##      $t2     - used to hold the sum of the $t1 and $t2.
##      $v0     - syscall parameter and return value.
##      $a0     - syscall parameter.

main:
        ## Get first number from user, put into $t0.
        li      $v0, 5          # load syscall read_int into $v0.
        syscall                 # make the syscall.
        move    $t0, $v0        # move the number read into $t0.

        ## Get second number from user, put into $t1.
        li      $v0, 5          # load syscall read_int into $v0.
        syscall                 # make the syscall.
        move    $t1, $v0        # move the number read into $t1.

        add     $t2, $t0, $t1   # compute the sum.

        ## Print out $t2.
        move    $a0, $t2        # move the number to print into $a0.
        li      $v0, 1          # load syscall print_int into $v0.
        syscall                 # make the syscall.

        li      $v0, 10         # syscall code 10 is for exit.
        syscall                 # make the syscall.

## end of add2.asm.
```

5.2 hello.asm

This program is described in section 2.5.

```
## Daniel J. Ellard -- 02/21/94
## hello.asm-- A "Hello World" program.
## Registers used:
##       $v0     - syscall parameter and return value.
##       $a0     - syscall parameter-- the string to print.

        .text
main:
        la      $a0, hello_msg  # load the addr of hello_msg into $a0.
        li      $v0, 4          # 4 is the print_string syscall.
        syscall                 # do the syscall.

        li      $v0, 10         # 10 is the exit syscall.
        syscall                 # do the syscall.

## Data for the program:
        .data
hello_msg:      .asciiz "Hello World\n"

## end hello.asm
```

5.3 multiples.asm

This program is described in section 2.7. The algorithm used is algorithm 2.1 (shown on page 32).

```
## Daniel J. Ellard -- 02/21/94
## multiples.asm-- takes two numbers A and B, and prints out
##      all the multiples of A from A to A * B.
##      If B <= 0, then no multiples are printed.
## Registers used:
##      $t0     - used to hold A.
##      $t1     - used to hold B.
##      $t2     - used to store S, the sentinel value A * B.
##      $t3     - used to store m, the current multiple of A.

        .text
main:
        ## read A into $t0, B into $t1.
        li      $v0, 5                  # syscall 5 = read_int
        syscall
        move    $t0, $v0                # A = integer just read

        li      $v0, 5                  # syscall 5 = read_int
        syscall
        move    $t1, $v0                # B = integer just read

        blez    $t1, exit               # if B <= 0, exit.

        mul     $t2, $t0, $t1           # S = A * B.
        move    $t3, $t0                # m = A

loop:
        move    $a0, $t3                # print m.
        li      $v0, 1                  # syscall 1 = print_int
        syscall                         # make the system call.

        beq     $t2, $t3, endloop       # if m == S, we're done.
        add     $t3, $t3, $t0           # otherwise, m = m + A.

        la      $a0, space              # print a space.
        li      $v0, 4                  # syscall 4 = print_string
        syscall

```

```
        b       loop                    # iterate.
endloop:
        la      $a0, newline            # print a newline:
        li      $v0, 4                  # syscall 4 = print_string
        syscall

exit:                                   # exit the program:
        li      $v0, 10                 # syscall 10 = exit
        syscall                         # we're outta here.

## Here's where the data for this program is stored:
        .data
space:          .asciiz " "
newline:        .asciiz "\n"

## end of multiples.asm
```

5.4 palindrome.asm

This program is described in section 2.8. The algorithm used is algorithm 2.2 (shown on page 34).

```
## Daniel J. Ellard -- 02/21/94
## palindrome.asm -- read a line of text and test if it is a palindrome.
## Register usage:
##      $t1     - A.
##      $t2     - B.
##      $t3     - the character at address A.
##      $t4     - the character at address B.
##      $v0     - syscall parameter / return values.
##      $a0     - syscall parameters.
##      $a1     - syscall parameters.

        .text
main:                                   # SPIM starts by jumping to main.
                                        ## read the string S:
        la      $a0, string_space
        li      $a1, 1024
        li      $v0, 8                  # load "read_string" code into $v0.
        syscall

        la      $t1, string_space       # A = S.

        la      $t2, string_space       ## we need to move B to the end
length_loop:                            #       of the string:
        lb      $t3, ($t2)              # load the byte at addr B into $t3.
        beqz    $t3, end_length_loop    # if $t3 == 0, branch out of loop.
        addu    $t2, $t2, 1             # otherwise, increment B,
        b       length_loop             #  and repeat the loop.
end_length_loop:
        subu    $t2, $t2, 2             ## subtract 2 to move B back past
                                        #       the '\0' and '\n'.
test_loop:
        bge     $t1, $t2, is_palin      # if A >= B, it's a palindrome.

        lb      $t3, ($t1)              # load the byte at addr A into $t3,
        lb      $t4, ($t2)              # load the byte at addr B into $t4.
        bne     $t3, $t4, not_palin     # if $t3 != $t4, not a palindrome.
                                        # Otherwise,
        addu    $t1, $t1, 1             #  increment A,
```

```
        subu    $t2, $t2, 1             #  decrement B,
        b       test_loop               #  and repeat the loop.

is_palin:                               ## print the is_palin_msg, and exit.
        la      $a0, is_palin_msg
        li      $v0, 4
        syscall
        b       exit

not_palin:
        la      $a0, not_palin_msg      ## print the is_palin_msg, and exit.
        li      $v0, 4
        syscall
        b       exit

exit:                                   ## exit the program:
        li      $v0, 10                 # load "exit" into $v0.
        syscall                         # make the system call.

## Here's where the data for this program is stored:
        .data
string_space:   .space  1024            # reserve 1024 bytes for the string.
is_palin_msg:   .asciiz "The string is a palindrome.\n"
not_palin_msg:  .asciiz "The string is not a palindrome.\n"

## end of palindrome.asm
```

5.5 atoi-1.asm

This program is described in section 2.9.1. The algorithm used is algorithm 2.3 (shown on page 37).

```
## Daniel J. Ellard -- 03/02/94
## atoi-1.asm -- reads a line of text, converts it to an integer, and
##      prints the integer.
## Register usage:
##      $t0     - S.
##      $t1     - the character pointed to by S.
##      $t2     - the current sum.

        .text
main:
        la      $a0, string_space       ## read the string S:
        li      $a1, 1024
        li      $v0, 8                  # load "read_string" code into $v0.
        syscall

        la      $t0, string_space       # Initialize S.
        li      $t2, 0                  # Initialize sum = 0.

sum_loop:
        lb      $t1, ($t0)              # load the byte at addr S into $t1,
        addu    $t0, $t0, 1             # and increment S.

        ## use 10 instead of '\n' due to SPIM bug!
        beq     $t1, 10, end_sum_loop   # if $t1 == \n, branch out of loop.

        mul     $t2, $t2, 10            # t2 *= 10.

        sub     $t1, $t1, '0'           # t1 -= '0'.
        add     $t2, $t2, $t1           # t2 += t1.

        b       sum_loop                #  and repeat the loop.
end_sum_loop:
        move    $a0, $t2                # print out the answer (t2).
        li      $v0, 1
        syscall

        la      $a0, newline            # and then print out a newline.
        li      $v0, 4
```

```
        syscall

exit:                                   ## exit the program:
        li      $v0, 10                 # load "exit" into $v0.
        syscall                         # make the system call.

        .data                           ## Start of data declarations:
newline:        .asciiz "\n"
string_space:   .space  1024            # reserve 1024 bytes for the string.

## end of atoi-1.asm
```

5.6 atoi-4.asm

This program is described in section 2.9.4. The algorithm used is algorithm 2.3 (shown on page 37), modified as described in section 2.9.4.

```
## Daniel J. Ellard -- 03/04/94
## atoi-4.asm -- reads a line of text, converts it to an integer,
##      and prints the integer.
##      Handles signed numbers, detects bad characters, and overflow.
## Register usage:
##      $t0     - S.
##      $t1     - the character pointed to by S.
##      $t2     - the current sum.
##      $t3     - the "sign" of the sum.
##      $t4     - holds the constant 10.
##      $t5     - used to test for overflow.
        .text
main:
        la      $a0, string_space       # read the string S:
        li      $a1, 1024
        li      $v0, 8                  # load "read_string" code into $v0.
        syscall

        la      $t0, string_space       # Initialize S.
        li      $t2, 0                  # Initialize sum = 0.

get_sign:
        li      $t3, 1                  # assume the sign is positive.
        lb      $t1, ($t0)              # grab the "sign"
        bne     $t1, '-', positive      # if not "-", do nothing.
        li      $t3, -1                 # otherwise, set t3 = -1, and
        addu    $t0, $t0, 1             #  skip over the sign.
positive:
        li      $t4, 10                 # store the constant 10 in $t4.
sum_loop:
        lb      $t1, ($t0)              # load the byte at addr S into $t1,
        addu    $t0, $t0, 1             # and increment S,

        ## use 10 instead of '\n' due to SPIM bug!
        beq     $t1, 10, end_sum_loop   # if $t1 == \n, branch out of loop.

        blt     $t1, '0', end_sum_loop  # make sure 0 <= t1
        bgt     $t1, '9', end_sum_loop  # make sure 9 >= t1
```

```

        mult    $t2, $t4                # multiply $t2 by 10.
        mfhi    $t5                     # check for overflow;
        bnez    $t5, overflow           #  if so, then report an overflow.
        mflo    $t2                     # get the result of the multiply
        blt     $t2, $0, overflow       # make sure that it isn’t negative.

        sub     $t1, $t1, ’0’           # t1 -= ’0’.
        add     $t2, $t2, $t1           # t2 += t1.
        blt     $t2, $0, overflow

        b       sum_loop                #  and repeat the loop.
end_sum_loop:
        mul     $t2, $t2, $t3           # set the sign properly.

        move    $a0, $t2                # print out the answer (t2).
        li      $v0, 1
        syscall

        la      $a0, newline            # and then print out a newline.
        li      $v0, 4
        syscall

        b       exit

overflow:                               # indicate that an overflow occurred.
        la      $a0, overflow_msg
        li      $v0, 4
        syscall
        b       exit

exit:                                   # exit the program:
        li      $v0, 10                 # load "exit" into $v0.
        syscall                         # make the system call.

        .data                           ## Start of data declarations:
newline:        .asciiz "\n"
overflow_msg:   .asciiz "Overflow!\n"
string_space:   .space  1024            # reserve 1024 bytes for the string.

## end of atoi-4.asm
```

5.7 printf.asm

Using syscalls for output can quickly become tedious, and output routines can quickly muddy up even the neatest code, since it requires several assembly instructions just to print out a number. To make matters worse, there is no syscall which prints out a single ASCII character.

To help my own coding, I wrote the following `printf` function, which behaves like a simplified form of the `printf` function in the standard `C` library. It implements only a fraction of the functionality of the real `printf`, but enough to be useful. See the comments in the code for more information.

```
## Daniel J. Ellard -- 03/13/94
## printf.asm--
##      an implementation of a simple printf work-alike.

## printf--
##      A simple printf-like function.  Understands just the basic forms
##      of the %s, %d, %c, and %% formats, and can only have 3 embedded
##      formats (so that all of the parameters are passed in registers).
##      If there are more than 3 embedded formats, all but the first 3 are
##      completely ignored (not even printed).
## Register Usage:
##      $a0,$s0 - pointer to format string
##      $a1,$s1 - format argument 1 (optional)
##      $a2,$s2 - format argument 2 (optional)
##      $a3,$s3 - format argument 3 (optional)
##      $s4     - count of formats processed.
##      $s5     - char at $s4.
##      $s6     - pointer to printf buffer
##
        .text
        .globl  printf
printf:
        subu    $sp, $sp, 36            # set up the stack frame,
        sw      $ra, 32($sp)            # saving the local environment.
        sw      $fp, 28($sp)
        sw      $s0, 24($sp)
        sw      $s1, 20($sp)
        sw      $s2, 16($sp)
        sw      $s3, 12($sp)
        sw      $s4, 8($sp)
        sw      $s5, 4($sp)
```

```
        sw      $s6, 0($sp)
        addu    $fp, $sp, 36

                                        # grab the arguments:
        move    $s0, $a0                # fmt string
        move    $s1, $a1                # arg1 (optional)
        move    $s2, $a2                # arg2 (optional)
        move    $s3, $a3                # arg3 (optional)

        li      $s4, 0                  # set # of formats = 0
        la      $s6, printf_buf         # set s6 = base of printf buffer.

printf_loop:                            # process each character in the fmt:
        lb      $s5, 0($s0)             # get the next character, and then
        addu    $s0, $s0, 1             # bump up $s0 to the next character.

        beq     $s5, '%', printf_fmt    # if the fmt character, then do fmt.
        beq     $0, $s5, printf_end     # if zero, then go to end.

printf_putc:
        sb      $s5, 0($s6)             # otherwise, just put this char
        sb      $0, 1($s6)              # into the printf buffer,
        move    $a0, $s6                # and then print it with the
        li      $v0, 4                  # print_str syscall
        syscall

        b       printf_loop             # loop on.

printf_fmt:
        lb      $s5, 0($s0)             # see what the fmt character is,
        addu    $s0, $s0, 1             # and bump up the pointer.

        beq     $s4, 3, printf_loop     # if we've already processed 3 args,
                                        # then *ignore* this fmt.
        beq     $s5, 'd', printf_int    # if 'd', print as a decimal integer.
        beq     $s5, 's', printf_str    # if 's', print as a string.
        beq     $s5, 'c', printf_char   # if 'c', print as a ASCII char.
        beq     $s5, '%', printf_perc   # if '%', print a '%'
        b       printf_loop             # otherwise, just continue.

printf_shift_args:                      # shift over the fmt args,
        move    $s1, $s2                # $s1 = $s2
        move    $s2, $s3                # $s2 = $s3

```

```
        add     $s4, $s4, 1             # increment # of args processed.

        b       printf_loop             # and continue the main loop.

printf_int:                             # deal with a %d:
        move    $a0, $s1                # do a print_int syscall of $s1.
        li      $v0, 1
        syscall
        b       printf_shift_args       # branch to printf_shift_args

printf_str:                             # deal with a %s:
        move    $a0, $s1                # do a print_string syscall of $s1.
        li      $v0, 4
        syscall
        b       printf_shift_args       # branch to printf_shift_args

printf_char:                            # deal with a %c:
        sb      $s1, 0($s6)             # fill the buffer in with byte $s1,
        sb      $0, 1($s6)              # and then a null.
        move    $a0, $s6                # and then do a print_str syscall
        li      $v0, 4                  #       on the buffer.
        syscall
        b       printf_shift_args       # branch to printf_shift_args

printf_perc:                            # deal with a %%:
        li      $s5, '%'                # (this is redundant)
        sb      $s5, 0($s6)             # fill the buffer in with byte %,
        sb      $0, 1($s6)              # and then a null.
        move    $a0, $s6                # and then do a print_str syscall
        li      $v0, 4                  #       on the buffer.
        syscall
        b       printf_loop             # branch to printf_loop

printf_end:
        lw      $ra, 32($sp)            # restore the prior environment:
        lw      $fp, 28($sp)
        lw      $s0, 24($sp)
        lw      $s1, 20($sp)
        lw      $s2, 16($sp)
        lw      $s3, 12($sp)
        lw      $s4, 8($sp)
        lw      $s5, 4($sp)
        lw      $s6, 0($sp)
        addu    $sp, $sp, 36            # release the stack frame.
```

```
        jr      $ra                     # return.

        .data
printf_buf:     .space  2

## end of printf.asm
```

5.8 fib-o.asm

This program is described in section 3.1.1.3.

This is a (somewhat) optimized version of a program which computes Fibonacci numbers. The optimization involves not building a stack frame unless absolutely necessary. I wouldn't recommend that you make a habit of optimizing your code in this manner, but it can be a useful technique.

```
## Daniel J. Ellard -- 02/27/94
## fib-o.asm-- A program to compute Fibonacci numbers.
##      An optimized version of fib-t.asm.
## main--
## Registers used:
##      $v0     - syscall parameter and return value.
##      $a0     - syscall parameter-- the string to print.
        .text
main:
        subu    $sp, $sp, 32            # Set up main's stack frame:
        sw      $ra, 28($sp)
        sw      $fp, 24($sp)
        addu    $fp, $sp, 32

        ## Get n from the user, put into $a0.
        li      $v0, 5                  # load syscall read_int into $v0.
        syscall                         # make the syscall.
        move    $a0, $v0                # move the number read into $a0.
        jal     fib                     # call fib.

        move    $a0, $v0
        li      $v0, 1                  # load syscall print_int into $v0.
        syscall                         # make the syscall.

        la      $a0, newline
        li      $v0, 4
        syscall                         # make the syscall.

        li      $v0, 10                 # 10 is the exit syscall.
        syscall                         # do the syscall.

## fib-- (hacked-up caller-save method)
## Registers used:
##      $a0     - initially n.
```

```
##      $t0     - parameter n.
##      $t1     - fib (n - 1).
##      $t2     - fib (n - 2).
        .text
fib:
        bgt     $a0, 1, fib_recurse     # if n < 2, then just return a 1,
        li      $v0, 1                  # don't build a stack frame.
        jr      $ra
                                        # otherwise, set things up to handle
fib_recurse:                            # the recursive case:
        subu    $sp, $sp, 32            # frame size = 32, just because...
        sw      $ra, 28($sp)            # preserve the Return Address.
        sw      $fp, 24($sp)            # preserve the Frame Pointer.
        addu    $fp, $sp, 32            # move Frame Pointer to new base.

        move    $t0, $a0                # get n from caller.

                                        # compute fib (n - 1):
        sw      $t0, 20($sp)            # preserve n.
        sub     $a0, $t0, 1             # compute fib (n - 1)
        jal     fib
        move    $t1, $v0                # t1 = fib (n - 1)
        lw      $t0, 20($sp)            # restore n.

                                        # compute fib (n - 2):
        sw      $t1, 16($sp)            # preserve $t1.
        sub     $a0, $t0, 2             # compute fib (n - 2)
        jal     fib
        move    $t2, $v0                # t2 = fib (n - 2)
        lw      $t1, 16($sp)            # restore $t1.

        add     $v0, $t1, $t2           # $v0 = fib (n - 1) + fib (n - 2)
        lw      $ra, 28($sp)            # restore Return Address.
        lw      $fp, 24($sp)            # restore Frame Pointer.
        addu    $sp, $sp, 32            # restore Stack Pointer.
        jr      $ra                     # return.

## data for fib-o.asm:
        .data
newline:        .asciiz "\n"

## end of fib-o.asm
```

5.9 treesort.asm

This program is outlined in section 3.2. The treesort algorithm is given in algorithm 3.1 (shown on page 51).

```
## Daniel J. Ellard -- 03/05/94
## tree-sort.asm -- some binary tree routines, in MIPS assembly.
##
##      The tree nodes are 3-word structures.  The first word is the
##      integer value of the node, and the second and third are the
##      left and right pointers.
##      &&&     NOTE-- the functions in this file assume this
##      &&&     representation!

## main --
##      1. Initialize the tree by creating a root node, using the
##              sentinel value as the value.
##      2. Loop, reading numbers from the user.  If the number is equal
##              to the sentinel value, break out of the loop; otherwise
##              insert the number into the tree (using tree_insert).
##      3. Print out the contents of the tree (skipping the root node),
##              by calling tree_print on the left and right
##              children of the root node.
## Register usage:
##      $s0     - the root of the tree.
##      $s1     - each number read in from the user.
##      $s2     - the sentinel value (right now, this is 0).
        .text
main:
        li      $s2, 0          # $s2 = the sentinel value.

                ## Step 1: create the root node.
                ## root = tree_node_create ($s2, 0, 0);
        move    $a0, $s2                # val   = $s2
        li      $a1, 0                  # left  = NULL
        li      $a2, 0                  # right = NULL
        jal     tree_node_create        # call tree_node_create
        move    $s0, $v0                # and put the result into $s0.


                ## Step 2: read numbers and add them to the tree, until
                ## we see the sentinel value.
                ## register $s1 holds the number read.
```

```
input_loop:
        li      $v0, 5                  # syscall 5 == read_int.
        syscall
        move    $s1, $v0                # $s1 = read_int

        beq     $s1, $s2, end_input     # if we read the sentinel, break.

                                        # tree_insert (number, root);
        move    $a0, $s1                # number= $s1
        move    $a1, $s0                # root  = $s0
        jal     tree_insert             # call tree_insert.

        b       input_loop              # repeat input loop.
end_input:

                ## Step 3: print out the left and right subtrees.
        lw      $a0, 4($s0)             # print the root's left child.
        jal     tree_print

        lw      $a0, 8($s0)             # print the root's right child.
        jal     tree_print

        b       exit                    # exit.
## end of main.

## tree_node_create (val, left, right): make a new node with the given
##      val and left and right descendants.
## Register usage:
##      $s0     - val
##      $s1     - left
##      $s2     - right
tree_node_create:
                                        # set up the stack frame:
        subu    $sp, $sp, 32
        sw      $ra, 28($sp)
        sw      $fp, 24($sp)
        sw      $s0, 20($sp)
        sw      $s1, 16($sp)
        sw      $s2, 12($sp)
        sw      $s3, 8($sp)
        addu    $fp, $sp, 32
                                        # grab the parameters:
        move    $s0, $a0                # $s0 = val
        move    $s1, $a1                # $s1 = left
```

```
        move    $s2, $a2                # $s2 = right

        li      $a0, 12                 # need 12 bytes for the new node.
        li      $v0, 9                  # sbrk is syscall 9.
        syscall
        move    $s3, $v0

        beqz    $s3, out_of_memory      # are we out of memory?

        sw      $s0, 0($s3)             # node->number  = number
        sw      $s1, 4($s3)             # node->left    = left
        sw      $s2, 8($s3)             # node->right   = right

        move    $v0, $s3                # put return value into v0.
                                        # release the stack frame:
        lw      $ra, 28($sp)            # restore the Return Address.
        lw      $fp, 24($sp)            # restore the Frame Pointer.
        lw      $s0, 20($sp)            # restore $s0.
        lw      $s1, 16($sp)            # restore $s1.
        lw      $s2, 12($sp)            # restore $s2.
        lw      $s3, 8($sp)             # restore $s3.
        addu    $sp, $sp, 32            # restore the Stack Pointer.
        jr      $ra                     # return.
## end of tree_node_create.

## tree_insert (val, root): make a new node with the given val.
## Register usage:
##      $s0     - val
##      $s1     - root
##      $s2     - new_node
##      $s3     - root->val (root_val)
##      $s4     - scratch pointer (ptr).
tree_insert:
                                # set up the stack frame:
        subu    $sp, $sp, 32
        sw      $ra, 28($sp)
        sw      $fp, 24($sp)
        sw      $s0, 20($sp)
        sw      $s1, 16($sp)
        sw      $s2, 12($sp)
        sw      $s3, 8($sp)
        sw      $s3, 4($sp)
        addu    $fp, $sp, 32

```

```
                                # grab the parameters:
        move    $s0, $a0                # $s0 = val
        move    $s1, $a1                # $s1 = root

                                # make a new node:
                                # new_node = tree_node_create (val, 0, 0);
        move    $a0, $s0                # val   = $s0
        li      $a1, 0                  # left  = 0
        li      $a2, 0                  # right = 0
        jal     tree_node_create        # call tree_node_create
        move    $s2, $v0                # save the result.

        ## search for the correct place to put the node.
        ## analogous to the following C code:
        ##      for (;;) {
        ##              root_val = root->val;
        ##              if (val <= root_val) {
        ##                      ptr = root->left;
        ##                      if (ptr != NULL) {
        ##                              root = ptr;
        ##                              continue;
        ##                      }
        ##                      else {
        ##                              root->left = new_node;
        ##                              break;
        ##                      }
        ##              }
        ##              else {
        ##                      /* the right side is symmetric. */
        ##              }
        ##      }
        ##
        ##      Commented with equivalent C code (you will lose many
        ##      style points if you ever write C like this...).
search_loop:
        lw      $s3, 0($s1)             # root_val = root->val;
        ble     $s0, $s3, go_left       # if (val <= s3) goto go_left;
        b       go_right                # goto go_right;

go_left:
        lw      $s4, 4($s1)             # ptr = root->left;
        beqz    $s4, add_left           # if (ptr == 0) goto add_left;
        move    $s1, $s4                # root = ptr;
        b       search_loop             # goto search_loop;
```

```

add_left:
        sw      $s2, 4($s1)             # root->left = new_node;
        b       end_search_loop         # goto end_search_loop;

go_right:
        lw      $s4, 8($s1)             # ptr = root->right;
        beqz    $s4, add_right          # if (ptr == 0) goto add_right;
        move    $s1, $s4                # root = ptr;
        b       search_loop             # goto search_loop;

add_right:
        sw      $s2, 8($s1)             # root->right = new_node;
        b       end_search_loop         # goto end_search_loop;

end_search_loop:

                                        # release the stack frame:
        lw      $ra, 28($sp)            # restore the Return Address.
        lw      $fp, 24($sp)            # restore the Frame Pointer.
        lw      $s0, 20($sp)            # restore $s0.
        lw      $s1, 16($sp)            # restore $s1.
        lw      $s2, 12($sp)            # restore $s2.
        lw      $s3, 8($sp)             # restore $s3.
        lw      $s4, 4($sp)             # restore $s4.
        addu    $sp, $sp, 32            # restore the Stack Pointer.
        jr      $ra                     # return.
## end of node_create.

## tree_walk (tree):
##      Do an inorder traversal of the tree, printing out each value.
##      Equivalent C code:
##      void            tree_print (tree_t *tree)
##      {
##              if (tree != NULL) {
##                      tree_print (tree->left);
##                      printf ("%d\n", tree->val);
##                      tree_print (tree->right);
##              }
##      }
## Register usage:
##      s0      - the tree.
tree_print:
                                        # set up the stack frame:
```

```
        subu    $sp, $sp, 32
        sw      $ra, 28($sp)
        sw      $fp, 24($sp)
        sw      $s0, 20($sp)
        addu    $fp, $sp, 32
                                        # grab the parameter:
        move    $s0, $a0                # $s0 = tree

        beqz    $s0, tree_print_end     # if tree == NULL, then return.

        lw      $a0, 4($s0)             # recurse left.
        jal     tree_print

                                        # print the value of the node:
        lw      $a0, 0($s0)             # print the value, and
        li      $v0, 1
        syscall
        la      $a0, newline            # also print a newline.
        li      $v0, 4
        syscall

        lw      $a0, 8($s0)             # recurse right.
        jal     tree_print

tree_print_end:                         # clean up and return:
        lw      $ra, 28($sp)            # restore the Return Address.
        lw      $fp, 24($sp)            # restore the Frame Pointer.
        lw      $s0, 20($sp)            # restore $s0.
        addu    $sp, $sp, 32            # restore the Stack Pointer.
        jr      $ra                     # return.
## end of tree_print.


## out_of_memory --
##      The routine to call when sbrk fails.  Jumps to exit.
out_of_memory:
        la      $a0, out_of_mem_msg
        li      $v0, 4
        syscall
        j       exit
## end of out_of_memory.

## exit --
##      The routine to call to exit the program.
```

```
exit:
        li      $v0, 10                 # 10 is the exit syscall.
        syscall
        ## end of program!
## end of exit.

## Here's where the data for this program is stored:
        .data
newline:        .asciiz "\n"
out_of_mem_msg: .asciiz "Out of memory!\n"

## end of tree-sort.asm
```