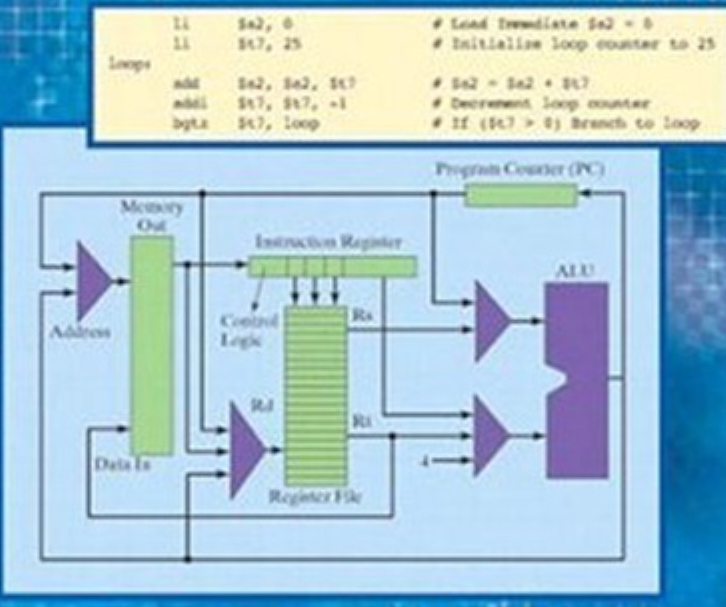
MIPS Assembly Language Programming



```
        li      $a2, 0                # Load Immediate $a2 = 0
        li      $t7, 25               # Initialize loop counter to 25
loop:
        add     $a2, $a2, $t7         # $a2 = $a2 + $t7
        addi    $t7, $t7, -1          # Decrement loop counter
        bgtz    $t7, loop             # If ($t7 > 0) Branch to loop
```

Robert L. Britton

MIPS Assembly Language Programming

Robert Britton
Computer Science Department
California State University, Chico
Chico, California

Preface

This book is targeted for use in an introductory lower-division assembly language programming or computer organization course. After students are introduced to the MIPS architecture using this book, they will be well prepared to go on to an upper-division computer organization course using a textbook such as “Computer Organization and Design” by Patterson and Hennessy. This book provides a technique that will make MIPS assembly language programming a relatively easy task as compared to writing complex Intel™ 80x86 assembly language code. Students using this book will acquire an understanding of how the functional components of computers are put together, and how a computer works at the machine language level. We assume students have experience in developing algorithms, and running programs in a high-level language.

Chapter 1 provides an introduction to the basic MIPS architecture, which is a modern Reduced Instruction Set Computer (RISC). Chapter 2 shows how to develop code targeted to run on a MIPS processor using an intermediate pseudocode notation similar to the high-level language “C”, and how easy it is to translate this notation to MIPS assembly language.

Chapter 3 is an introduction to the binary number system, and the rules for performing arithmetic, as well as detecting overflow. Chapter 4 explains the features of the PCSpim simulator for the MIPS architecture, which by the way is available for free. Within the remaining chapters, a wealth of programming exercises are provided, which every student needs to become an accomplished assembly language programmer. Instructors are provided with a set of PowerPoint slides. After students have had an opportunity to develop their pseudocode and their MIPS assembly language code for each of the exercises, they can be provided with example solutions via the PowerPoint slides.

In Chapter 5 students are presented with the classical I/O algorithms for decimal and hexadecimal representation. The utility of logical operators and shift operators are stressed. In Chapter 6, a specific argument passing protocol is defined. Most significant programming projects are a teamwork effort. Emphasis is placed on the importance that everyone involved in a teamwork project must adopt the same convention for parameter passing. In the case of nested function calls, a specific convention is defined for saving and restoring values in the temporary registers. In Chapter 7 the necessity for reentrant code is explained, as well as the rules one must follow to write such functions. Chapter 8 introduces exceptions and exception processing. In Chapter 9 a pipelined implementation of the MIPS architecture is presented, and the special programming considerations dealing with delayed loads and delayed branches are discussed. The final chapter briefly describes the expanding opportunities in the field of embedded processors for programmers who have a solid understanding of the underlying processor functionality.

Robert Britton
May 2002

Contents

Related Web Sites

www.mips.com/
http://www.ecst.csuchico.edu/~britton
http://www.cs.wisc.edu/~larus/spim.html
http://www.downcastsystems.com/mipster.asp
http://www.cs.wisc.edu/~larus/SPIM/cod-appa.pdf

CHAPTER 1

The MIPS Architecture

If at first you don't succeed,
Skydiving is definitely not for you.

1.1 Introduction

This book provides a technique that will make MIPS assembly language programming a relatively easy task as compared to writing Intel™ 80x86 assembly language code. We are assuming that you have experience in developing algorithms, and running programs in some high level language such as Pascal, C, C++, or JAVA. One of the benefits of understanding and writing assembly language code is that you will have new insights into how to write more efficient, high-level language code. You will become familiar with the task that is performed by a compiler and how computers are organized down to the basic functional component level. You may even open new opportunities for yourself in the exploding field of embedded processors.

The first thing everyone must do to apply this technique is to become familiar with the MIPS architecture. The architecture of any computer is defined by the registers that are available (visible) to the assembly language programmer, the instruction set, the memory addressing modes, and the data types.

1.2 The Datapath Diagram

It is very useful to have a picture of a datapath diagram that depicts the essential components and features of the MIPS architecture. Please note that there are many different ways that an architecture can be implemented in hardware. These days, pipelined and superscalar implementations are common in high-performance processors. An initial picture of a MIPS datapath diagram will be the straightforward simple diagram shown in Figure 1.1. This is not a completely accurate diagram for the MIPS architecture; it is just a useful starting point.

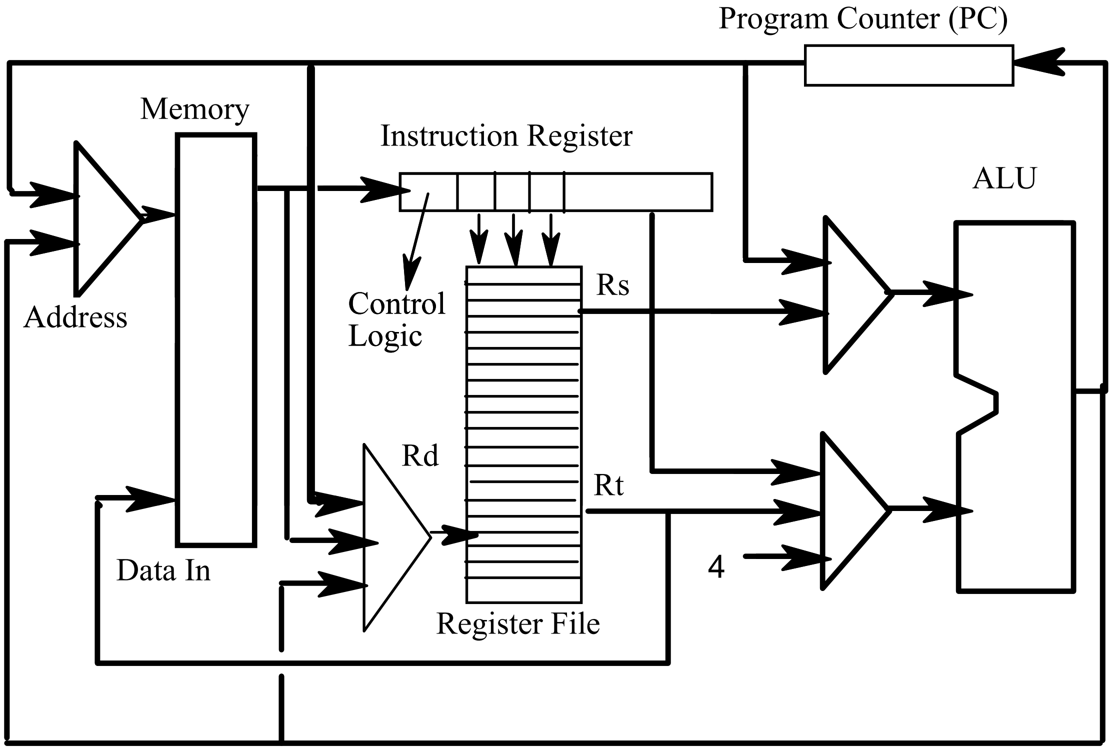


Figure 1.1 MIPS Simplified Datapath Diagram

1.3 Instruction Fetch and Execute

Computers work by fetching machine language instructions from memory, decoding and executing them. Machine language instructions and the values that are operated upon are encoded in binary. Chapter 3 introduces the binary number system. As we progress through the first two chapters, we will be expressing values as decimal values, but keep in mind that in an actual MIPS processor these values are encoded in binary. The basic functional components of the MIPS architecture shown in Figure 1.1 are:

(a) Program Counter (PC)
(b) Memory
(c) Instruction Register (IR)
(d) Register File
(e) Arithmetic and Logic Unit (ALU)
(f) Control Unit

Interconnecting all of these components, except the control unit, are busses. A bus is nothing more than a set of electrical conducting paths over which different sets of binary values are transmitted. Most of the busses in the MIPS architecture are 32-bits wide. In other words, 32 separate, tiny wires running from a source to a destination.

In this datapath diagram, we have the situation where we need to route information from more than one source to a destination, such as the ALU. One way to accomplish this is with a multiplexer. Multiplexers are sometimes called data selectors. In Figure 1.1, multiplexers are represented by the triangle-shaped symbols. Every multiplexer with two input busses must have a single control signal connected to it. This control signal comes from the control unit. The control signal is either the binary value zero or one, which is sent to the multiplexer over a single wire. In Figure 1.1, we have not shown any of the control signals, because it would make the diagram too busy. When the control signal is zero, the 32-bit value connected to input port zero (0) of the multiplexer will appear on the output of the multiplexer. When the control signal is one, the 32-bit value connected to input port one (1) of the multiplexer will appear on the output of the multiplexer. The acronym “bit” is an abbreviation of “binary digit.”

1.4 The MIPS Register File

The term “register” refers to an electronic storage component. Every register in the MIPS architecture is a component with a capacity to hold a 32-bit binary number. Anyone who has ever used an electronic hand-held calculator has experienced the fact that there is some electronic component inside the calculator that holds the result of the latest computation.

The MIPS architecture has a register file containing 32 registers. See Figure 1.2. Each register has a capacity to hold a 32-bit value. The range of values that can be represented with 32 bits is -2,147,483,648 to +2,147,483,647. When writing at the assembly language level almost every instruction requires that the programmer specify which registers in the register file are used in the execution of the instruction. A convention has been adopted that specifies which registers are appropriate to use in specific circumstances. The registers have been given names that help to remind us about this convention. Register $zero is special; it is the source of the constant value zero. Nothing can be stored in register $zero. Register number 1 has the name $at, which stands for assembler temporary. This register is reserved to implement “macro instructions” and should not be used by the assembly language programmer. Registers $k0 and $k1 are used by the kernel of the operating system and should not be changed by a user program.

1.5 The Arithmetic and Logic Unit (ALU)

The ALU, as its name implies, is a digital logic circuit designed to perform binary arithmetic operations, as well as binary logical operations such as “AND,” “OR,” and “Exclusive OR.” Which operation the ALU performs depends upon the operation code in the Instruction Register.

Number	Value	Name
0	0	$zero
1		$at
2		$v0
3		$v1
4		$a0
5		$a1
6		$a2
7		$a3
8		$t0
9		$t1
10		$t2
11		$t3
12		$t4
13		$t5
14		$t6
15		$t7
16		$s0
17		$s1
18		$s2
19		$s3
20		$s4
21		$s5
22		$s6
23		$s7
24		$t8
25		$t9
26		$k0
27		$k1
28		$gp
29		$sp
30		$fp
31		$ra

Figure 1.2 The Register File

1.6 The Program Counter (PC)

After a programmer has written a program in assembly language using a text editor, the mnemonic representation of the program is converted to machine language by a utility program called an assembler. The machine language code is stored in a file on disk. When someone wants to execute the program, another utility program, called a linking loader, loads and links together all of the necessary machine language modules into main memory. The individual instructions are stored sequentially in memory. The Program Counter (PC) is a register that is initialized by the operating system to the address of the first instruction of the program in memory. Notice in Figure 1.1 that the address in the program counter is routed to the address input of the memory via a bus. After an instruction has been fetched from memory and loaded into the instruction register (IR), the PC is incremented so that the CPU will have the address of the next sequential instruction for the next instruction fetch. The name Program Counter is misleading. A better name would be Program Pointer, but unfortunately the name has caught on, and there is no way to change this tradition.

1.7 Memory

Most modern processors are implemented with cache memory. Cache memory is located on the CPU chip. Cache memory provides fast memory access to instructions and data that were recently accessed from the main memory. How cache memory is implemented in hardware is not the subject of this text. For our purposes, the functionality of cache memory will be modeled as a large array of locations where information is stored and from which information can be fetched, one "word" at a time, or stored one "word" at a time. The term "word" refers to a 32-bit quantity. Each location in memory has a 32-bit address. In the MIPS architecture, memory addresses range from 0 to 4,294,967,295. The MIPS architecture specifies that the term "word" refers to a 32-bit value and the term "byte" refers to an 8-bit value. The MIPS architecture specifies that a word contains four bytes, and that the smallest addressable unit of information that can be referenced in memory is a byte. The address of the first byte in a word is also the address of the 32-bit word. All instructions in the MIPS architecture are 32 bits in length. Therefore, the program counter is incremented by four after each instruction is fetched.

1.8 The Instruction Register (IR)

The Instruction Register (IR) is a 32-bit register that holds a copy of the most recently fetched instruction. In the MIPS architecture three different instruction formats are defined, R - format, I - format, and J – format. (See Appendix C for details)

1.9 The Control Unit

To fetch and execute instructions, control signals must be generated in a specific sequence to accomplish the task. As you have already learned, multiplexers must have control signals as inputs. Each register has an input control line, which when activated will cause a new value to be loaded into the register. The ALU needs control signals to specify what operation it should perform. The cache memory needs control signals to specify when a read or write operation is to be performed. The register file needs a control signal to specify when a value should be written into the register file. All of these control signals come from the control unit. The control unit is implemented in hardware as a "finite state machine." How fast the computer runs is controlled by the clock rate. The clock generator is an oscillator that produces a continuous waveform as depicted in

Figure 1.3. The clock period is the reciprocal of the clock frequency. All modern computers run with clock rates in the mega-hertz (MHz) range. (Mega = 10^6)

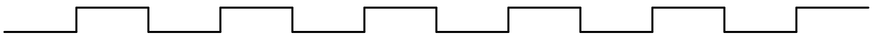


Figure 1.3 Clock Waveform

1.10 Instruction Set

Refer to Appendix C for a list of the basic integer instructions for the MIPS Architecture that we will be concerned with in this introductory level textbook. Note that unique binary codes assigned to each of the instructions. For a complete list of MIPS instructions, a good reference is the book by Kane and Heinrich "MIPS RISC Architecture." In reviewing the list of instructions in Appendix C you will find that the machine has instructions to add and subtract. The operands (source values) for these operations come from the register file and the results go to the register file. When programming in assembly language we use a mnemonic to specify which operation we want the computer to perform and we specify the register file locations using the names of the register file locations. Let us suppose that an assembly language programmer wants to add the contents of register $a1 to the contents of register $s1, and to place the results in register $v1. The assembly language instruction to accomplish this is:

```
add     $v1, $a1, $s1
```

The equivalent pseudocode statement would be: **$v1 = $a1 + $s1**

The MIPS architecture includes logical bit-wise instructions such as "AND", "OR", and "Exclusive-OR". There are instructions to implement control structures such as:

"if ... then ... else ..."

The multiply instruction multiplies two 32-bit binary values and produces a 64-bit product which is stored in two registers named High and Low. The following code segment shows how the lower 32 bits of the product of $a1 times $s1 can be moved into $v1:

```
mult    $a1, $s1
mflo    $v1
```

The following divide instruction divides the 32-bit binary value in register $a1 by the 32-bit value in register $s1. The quotient is stored in the Low register and the remainder is stored in the High register. The following code segment shows how the quotient is moved into $v0 and the remainder is moved into $v1:

```
div     $a1, $s1
mflo    $v0
mfhi    $v1
```

1.11 Addressing Modes

The MIPS architecture is referred to as a Reduced Instruction Set Computer (RISC). The designers of the MIPS architecture provide a set of basic instructions. (See Appendix C) In the case of fetching values from main memory or storing values into main memory, only one addressing mode was implemented in the hardware. The addressing mode is referred to as "**base address plus displacement.**" The MIPS architecture is a **Load/Store** architecture, which means the only instructions that access main memory are the Load and Store instructions. A load instruction accesses a value from memory and places a copy of the value found in memory in the register file. For example, the instruction:

```
        lw      $s1, 8($a0)
```

will compute the effective address of the memory location to be accessed by adding together the contents of register $a0 (the base address) and the constant value eight (the displacement). A copy of the value accessed from memory at the effective address is loaded into register $s1. The equivalent pseudocode statement would be:

```
        $s1 = Mem[$a0 + 8]
```

Notice in this example the base address is the value in register $a0, and the displacement is the constant value 8. The base address is always the content of one of the registers in the register file. The displacement is always a constant. The constant value can range from -32,768 to +32,767. In the case of the "Load Word" instruction, the effective address must be a number that is a multiple of four (4), because every word contains four bytes.

The syntax of the assembly language load instruction is somewhat confusing. If someone were to write a new MIPS assembler, the following syntax would do a better job of conveying to the user what the instruction actually does: **lw $s1, [$a0+8]**

The following is an example of a "Store Word" instruction:

```
        sw      $s1,     12($a0)
```

When the hardware executes this instruction it will compute the effective address of the destination memory location by adding together the contents of register $a0 and the constant value 12. A copy of the contents of register $s1 is stored in memory at the effective address. The equivalent pseudocode statement would be:

```
        Mem[$a0 + 12] = $s1
```

From the point of view of an assembly language programmer, memory can be thought of as a very long linear array, and the effective address is a pointer to some location in this array that the operating system has designated as the data segment. The Program Counter is a pointer into this same array, but to a different segment called the program segment. The operating system has allocated one other segment in memory called the stack segment.

1.12 Summary

When we write a program in assembly language we are creating a list of instructions that we want the processor to perform to accomplish some task (an algorithm). As soon as we have acquired a functional model of the processor and know exactly what instructions the processor can perform, then we will have mastered the first necessary component to becoming a MIPS assembly language programmer.

The continuous step by step functional operation of our simplified model for the MIPS architecture can be described as:

1. An instruction is fetched from memory at the location specified by the Program counter. The instruction is loaded into the Instruction Register. The Program Counter is incremented by four.

2. Two five bit codes Rs and Rt within the instruction specify which register file locations are accessed to obtain two 32-bit source operands.

3. The two 32-bit source operands are routed to the ALU inputs where some operation is performed depending upon the Op-Code in the instruction.

4. The result of the operation is placed back into the register file at a location specified by the 5-bit Rd code in the Instruction Register. Go to step 1.

Exercises

1.1 Explain the difference between a register and the ALU.
1.2 Explain the difference between assembly language and machine language.
1.3 Explain the difference between Cache Memory and the Register File.
1.4 Explain the difference between the Instruction Register and the Program Counter.
1.5 Explain the difference between a bus and a control line.
1.6 Identify a kitchen appliance that contains a finite state machine.
1.7 If a 500 MHz machine takes one clock cycle to fetch and execute an instruction, then what is the instruction execution rate of the machine?
1.8 How many instructions could the above machine execute in one minute?
1.9 Let's suppose we have a 40-year-old computer that has an instruction execution rate of one thousand instructions per second. How long would it take in days, hours, and minutes to execute the same number of instructions you derived for the 500 MHz machine?
1.10 What is an algorithm?

CHAPTER 2

Pseudocode

Where does satisfaction come from?
A satisfactory.

2.1 Introduction

Experienced programmers develop their algorithms using high-level programming constructs such as:

- If **(condition)** do **{this block of code}** else **do {that block of code};**
- While **(condition)** do **{this block of code};**
- For **(t0=1, t0 < s0, t0++)** do **{this block of code};**

The key to making MIPS assembly language programming easy, is to initially develop the algorithm using a high level **pseudocode** notation with which we are already familiar. Then in the final phase translate these high level pseudocode expressions into MIPS assembly language. In other words in the final phase we are performing the same function that a compiler performs, which is to translate high-level code into the equivalent assembly language code.

2.2 Develop the Algorithm in Pseudocode

When documenting an algorithm in a language such as Pascal, C, C++, or JAVA, programmers use descriptive variable names such as: speed, volume, size, count, amount, etc. After the program is compiled, these variable names correspond to memory locations, and the values stored in these memory locations correspond to the values of these variables. A compiler will attempt to develop code in such a way as to keep the variables that are referenced most often in processor registers, because access to a variable in a processor register is faster than access to random access memory (RAM). MIPS has 32 processor registers whose names were defined in Table 1.1. In the case of the MIPS architecture, all of the data manipulation instructions and the control instructions require that their operands be in the register file.

A MIPS assembly language programmer must specify within each instruction which processor registers are going to be utilized. For example, we may have a value in register $t0 corresponding to speed, a value in register $t1 corresponding to volume, a value in register $t2 corresponding to size, and a value in register $t3 corresponding to count. When using pseudocode to document an assembly language program, we must use the names of the registers we intend to use in the assembly language code. It is advisable to create a cross-reference table that defines what each processor register is being used for within the algorithm. (For example $t0: Sum, $v0: Count) We use register names in pseudocode so that the translation to assembly language code will be an easy process to perform and because we want documentation that describes how the features of the MIPS architecture were used to implement the algorithm. Unless we identify the registers being used, the pseudocode is quite limited in terms of deriving the assembly language program or having any correspondence to the assembly language code.

Pseudocode for assembly language programs will have the appearance of Pascal or C in terms of control structures and arithmetic expressions. Descriptive variable names will usually appear only in the Load Address (la) instruction where there is a reference to a symbolic memory address. In assembly language we define and allocate space for variables in the data segment of memory using assembler directives such as "**.word"** and **".spac**e". Strings are allocated space in memory using the assembler directive **".asciiz"**. See Appendix A, for a list of the most commonly used assembler directives.

Now, for an example, let us suppose we want to write an assembly language program to find the sum of the integers from one to N, where N is a value read in from the keyboard. In other words do the following: 1 + 2 + 3 + 4 + 5 + 6 + 7 + ...+ N
Below you will find a pseudocode description of the algorithm and the corresponding assembly language program, where processor register $t0 is used to accumulate the sum, and processor register $v0 is used as a loop counter that starts with the value N and counts down to 0.

Using a text editor create the following program file and experiment with the different features of the MIPS simulator. All MIPS assembly language source code files **must be saved as text only**. Chapter 4 provides a description of how to download the MIPS simulator. Note that the algorithm used here sums the integers in reverse order.

```
#       Program Name: Sum of Integers
#       Programmer:   YOUR NAME
#       Date last modified:
```

```
# Functional Description:
# A program to find the Sum of the Integers from 1 to N, where N is a value
# input from the keyboard.
#########################################################################
# Pseudocode description of algorithm:
#   main:       cout << "Please input a value for N"
#               cin >> v0
#               If ( v0 <= 0 ) stop
#               t0 = 0;
#               While (v0 > 0 ) do
#                       {
#                       t0 = t0 + v0;
#                       v0 = v0 - 1;
#                       }
#               cout << t0;
#               go to main
```

```
#############################################################################
# Cross References:
# v0: N,
# t0: Sum
#############################################################################
             .data
prompt:      .asciiz   “\n   Please Input a value for N =  ”
result:      .asciiz   “   The sum of the integers from 1 to N is ”
bye:         .asciiz   “\n  **** Adios Amigo - Have a good day ****”
             .globl    main
             .text
main:
             li          $v0, 4                  # system call code for Print String
             la          $a0, prompt             # load address of prompt into $a0
             syscall                             # print the prompt message

             li          $v0, 5                  # system call code for Read Integer
             syscall                             # reads the value of N into $v0

             blez        $v0,  end               # branch to end if  $v0  < =   0
             li          $t0, 0                  # clear register $t0 to zero
loop:
             add         $t0, $t0, $v0           # sum of integers in register $t0
             addi        $v0, $v0, -1            # summing integers in reverse order
             bnez        $v0,  loop              # branch to loop if $v0 is != zero

             li          $v0, 4                  # system call code for Print String
             la          $a0, result             # load address of message into $a0
             syscall                             # print the string

             li          $v0, 1                  # system call code for Print Integer
             move        $a0,  $t0               # move value to be printed to $a0
             syscall                             # print sum of integers
             b           main                    # branch to main

end:         li          $v0, 4                  # system call code for Print String
             la          $a0, bye                # load address of msg. into $a0
             syscall                             # print the string

             li          $v0, 10                 # terminate program run and
             syscall                             # return control to  system
```

MUST HAVE A BLANK LINE AT THE END OF THE TEXT FILE

2.3 Register Usage Convention

Within the register file different sets of registers have been given names to remind the programmer of a convention, which all MIPS assembly language programmers are expected to abide by. If all the members of a programming team do not adhere to the same convention, the entire effort will result in disaster. To simulate this situation everyone using this book is expected to adhere to the same convention. Programs should run correctly, even if the class members randomly exchange their functions.

Programs of any complexity typically involve a main program that calls functions. Any important variables in the main program that must be maintained across function calls should be assigned to registers $s0 through $s7. As programs become more complex, functions will call other functions. This is referred to as nested function calls. A function that does not call another function is referred to as a "leaf function." When writing a function, the programmer may utilize registers $t0 through $t9 with the understanding that no other code modules expect values in these registers will be maintained. If additional registers are needed within the function, the programmer may use only registers $s0 through $s7 if he/she first saves their current values on the stack and restores their values before exiting the function. Registers $s0 through $s7 are referred to as callee-saved registers. Registers $t0 through $t9 are referred to as caller-saved registers. This means that if the code module requires that the contents of certain "t" registers must be maintained upon return from a call to another function, then it is the responsibility of the calling module to save these values on the stack and restore the values upon returning from the function call. Registers $a0 through $a3 are used to pass arguments to functions, and registers $v0 and $v1 are used to return values from functions. These conventions are covered in more detail in Chapter 6.

2.4 The MIPS Instruction Set

When the MIPS architecture was defined, the designers decided that the machine would have instructions to perform the operations listed in Appendix C. At that time, the designers also decided on a binary operation code for each instruction, and the specific instruction format in binary. Given the requirement of the different instructions it was necessary to define three different formats. Some instructions are encoded in R format, some in I format and a few in J format. The list of instructions in Appendix C is not a complete list. The other instructions not in Appendix C involve floating point instructions, coprocessor instructions, cache instructions, instructions to manage virtual memory, and instructions concerned with the pipeline execution.

The designers of the MIPS assembler, the program that translates MIPS assembly language code to MIPS binary machine language code, also made some decisions to simplify the task of writing MIPS assembly language code. The MIPS assembler provides a set of macro (also called *synthetic or pseudo*) instructions. Every time a programmer specifies a macro instruction, the assembler replaces it with a set of actual MIPS instructions to accomplish the task. Appendix D provides a list of macro instructions. For example, let us suppose a programmer used the absolute value macro instruction:

```
        abs     $s0,  $t8
```

The MIPS assembler would then insert the following three instructions to accomplish the task:

```
        addu    $s0, $t0, $t8
        bgez    $t8, positive
        sub     $s0, $zero, $t8
positive:
```

Using the macro instructions simplifies the task of writing assembly language code, and programmers are encouraged to use the macro instructions. Note that when there is a need to calculate time and space parameters for a module of code the abs macro instruction would correspond to three clock cycles to execute (worse case), and three memory locations of storage space. The macro instructions have been placed in a separate appendix to assist the programmer in recognizing these two classes of instructions.

2.5 Translation of an "IF THEN ELSE" Control Structure

The "**If** (condition) **then** do {this block of code} **else** do {that block of code}" control structure is probably the most widely used by programmers. Let us suppose that a programmer initially developed an algorithm containing the following pseudocode. Can you describe what this algorithm accomplishes?

```
if ($t8 < 0) then
        {$s0 = 0 - $t8;
         $t1 = $t1 +1}
else
        {$s0 =  $t8;
         $t2 = $t2 + 1}
```

When the time comes to translate this pseudocode to MIPS assembly language, the results could appear as shown below. In MIPS assembly language, anything on a line following the number sign (#) is a comment. Notice how the comments in the code below help to make the connection back to the original pseudocode.

```
        bgez    $t8, else               # if ($t8 is > or = zero) branch to else
        sub     $s0, $zero, $t8         # $s0 gets the negative of $t8
        addi    $t1, $t1, 1             # increment $t1 by 1
        b       next                    # branch around the else code
else:
        ori     $s0, $t8, 0             # $s0 gets a copy of $t8
        addi    $t2, $t2, 1             # increment $t2 by 1
next:
```

2.6 Translation of a “WHILE” Control Structure

Another control structure is “ **While** (condition) **do** {this block of code}”.
Let us suppose that a programmer initially developed a function described by the following pseudocode where an “**if statement**” appears within the while loop. Can you describe what this function accomplishes?

```
$v0 = 1
While ($a1 < $a2) do
        {
    $t1 = mem[$a1]
    $t2 = mem[$a2]
    if ($t1 != $t2) go to break
    $a1 = $a1 + 1
    $a2 = $a2 - 1
        }
        return
break:
    $v0 = 0
    return
```

Here is a translation of the above “while” pseudocode into MIPS assembly language code.

```
        li      $v0, 1                  # Load Immediate $v0 with the value 1
loop:
        bgeu    $a1, $a2, done          # If( $a1 >= $a2) Branch to done
        lb      $t1, 0($a1)             # Load a Byte: $t1 = mem[$a1 + 0]
        lb      $t2, 0($a2)             # Load a Byte: $t2 = mem[$a2 + 0]
        bne     $t1, $t2, break         # If ($t1 != $t2) Branch to break
        addi    $a1, $a1, 1             # $a1 = $a1 + 1
        addi    $a2, $a2, -1            # $a2 = $a2 - 1
        b       loop                    # Branch to loop
break:
        li      $v0, 0                  # Load Immediate $v0 with the value 0
done:
```

2.7 Translation of a “FOR LOOP” Control Structure

Obviously a “ **for loop** ” control structure is very useful. Let us suppose that a programmer initially developed an algorithm containing the following pseudocode. In one sentence, can you describe what this algorithm accomplishes?

```
$a0 = 0;
For ( $t0 =10; $t0 > 0; $t0 = $t0 -1) do {$a0 = $a0 + $t0}
```

The following is a translation of the above “for-loop” pseudocode to MIPS assembly language code.

```
        li      $a0, 0          #  $a0 = 0
        li      $t0, 10         # Initialize loop counter to 10
loop:
        add     $a0, $a0, $t0
        addi    $t0, $t0, -1    # Decrement loop counter
        bgtz    $t0, loop       # If ($t0 > 0) Branch to loop
```

2.8 Translation of Arithmetic Expressions

Looking at the arithmetic expression below, what fundamental formula in geometry comes to mind?

$s0 = srt($a0 * $a0 + $a1 * $a1);

A translation of this pseudocode arithmetic expression to MIPS assembly language follows. In this case, we are assuming there is a library function that we can call that will return the square root of the argument, and we are assuming that the results of all computations do not exceed 32-bits. At this point, it is essential for the beginning programmer to go to Appendix C and study how the MIPS architecture accomplishes multiplication and division.

```
        mult    $a0, $a0        # Square $a0
        mflo    $t0             # t0 = Lower 32-bits of product
        mult    $a1, $a1        # Square $a1
        mflo    $a1             # t1 = Lower 32-bits of product
        add     $a0, $t0, $t1   # a0 = t0 + t1
        jal     srt             # Call the square root function
        move    $s0, $v0        # Result of sqr is returned in $v0 (Standard Convention)
```

Here is another arithmetic expression. What fundamental formula in geometry comes to mind? $s0 = (π * $t8 * $t8) / 2;

A translation of this pseudocode arithmetic expression to MIPS assembly language follows.

```
        li      $t0, 31415      # Pi scaled up by 10,000
        mult    $t8, $t8        # Radius squared
        mflo    $t1             # Move lower 32-bits of product in LOW register to $t1
        mult    $t1, $t0        # Multiply by Pi
        mflo    $s0             # Move lower 32-bits of product in LOW register to $s0
        sra     $s0, $s0, 1     # Division by two (2) is accomplished more efficiently
                                # using the Shift Right Arithmetic instruction
```

2.9 Translation of a "SWITCH" Control Structure

Below you will find a pseudocode example of a "switch" control structure where the binary value in register $s0 is shifted left either one, two, or three places depending upon what value is read in.

```
        $s0 = 32;
top:    cout  << "Input a value from 1 to 3"
        cin >> $v0
        switch ($v0)
                {case(1):  {$s0 = $s0 << 1; break;}
                  case(2):  {$s0 = $s0 << 2; break;}
                  case(3):  {$s0 = $s0 << 3; break;}
                  default:    goto top; }
        cout << $s0
```

A translation of this pseudocode into MIPS assembly language appears below:

```
                .data
                .align   2
jumptable:      .word    top, case1, case2, case3
prompt:         .asciiz  "\n\n Input a value from 1 to 3: "
                .text
top:
                li       $v0, 4               # Code to print a string
                la       $a0, prompt
                syscall
                li       $v0, 5               # Code to read an integer
                syscall
                blez     $v0, top             # Default for less than one
                li       $t3, 3
                bgt      $v0, $t3, top        # Default for greater than 3
                la       $a1, jumptable       # Load address of jumptable
                sll      $t0, $v0, 2          # Compute word offset
                add      $t1, $a1, $t0        #Form a pointer into jumptable
                lw       $t2, 0($t1)          # Load an address from jumptable
                jr       $t2                  # Jump to specific case "switch"
case1:          sll      $s0, $s0, 1          # Shift left logical one bit
                b        output
case2:          sll      $s0, $s0, 2          # Shift left logical two bits
                b        output
case3:          sll      $s0, $s0, 3          # Shift left logical three bits
output:
                li       $v0, 1               # Code to print an integer is 1
                move     $a0, $s0             # Pass argument to system in $a0
                syscall                       # Output results
```

2.10 Assembler Directives

A list of assembler directives appears in Appendix A. To allocate space in memory for a one-dimensional array of 1024 integers, the following construct is used in the C language:

int array[1024] ;

In MIPS assembly language, the corresponding construct is:

```
        .data
array:  .space  4096
```

Notice that the assembler directive ".space" requires that the amount of space to be allocated must be specified in bytes. Since there are four bytes in a word, an array of 1024 words is the same as an array of 4096 bytes.

To initialize a memory array with a set of 16 values corresponding to the powers of 2 (2^N with N going from 0 to 15) , the following construct is used in the C language:

```
int  pof2[16] ={ 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384,
                 32768 }
```

In MIPS assembly language the corresponding construct is:

```
        .data
pof2:   .word   1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768
```

The terminology used to describe the specific elements of an array is the same as that used in high-level languages: Element zero "pof2[0]" contains the value one (1). Element one "pof2[1]" contains the value 2, etc.

The following is an example of how assembly language code can be written to access element two in the array and place a copy of the value in register $s0. The Load Address (la) macro instruction is used to initialize the pointer "$a0" with the base address of the array that is labeled "pof2."

```
        la      $a0,    pof2            # a0 =&pof2
        lw      $s0,    8($a0)          # s0 = MEM[a0 + 8]
```

The Load Address (la) macro instruction initializes the pointer "$a0" with the base address of the array "pof2." After executing this code, register $s0 will contain the value 4. To access specific words within the array, the constant offset must be some multiple of four. The smallest element of information that can be accessed from memory is a byte, which is 8 bits of information. There are 4 bytes in a word. The address of a word is the same as the address of the first byte in a word. How would you write MIPS code to place the last value in the pof2 array into $t0?

2.11 Input and Output

SPIM provides a set of system services to perform input and output, which will be explained in detail in Chapter 4. At the pseudocode level, it is sufficient to indicate input and output using any construct the student feels familiar with. Looking back at the first example program in section, 2.2 we find the following constructs:

Output a string:	cout << "Please input a value for N"
Input a decimal value:	cin >> v0
Output a value in decimal:	cout << t0;

By studying the assembly language code in section 2.2 one could discover what sequence of assembly language instructions is required to accomplish the above I/O operations.

Exercises

2.1 Using Appendix A, translate each of the following pseudocode expressions into MIPS assembly language:

(a) t3 = t4 + t5 – t6;
(b) s3 = t2 / (s1 – 54321);
(c) sp = sp –16;
(d) cout << t3;
(e) cin >> t0;
(f) a0 = &array;
(g) t8 = Mem(a0);
(h) Mem(a0+ 16) = 32768;
(i) cout << "Hello World";
(j) If (t0 < 0) then t7 = 0 – t0 else t7 = t0;
(k) while (t0 != 0) { s1 = s1 + t0; t2 = t2 + 4; t0 = Mem(t2) };
(l) for (t1 = 99; t1 > 0; t1=t1 -1) v0 = v0 + t1;
(m) t0 = 2147483647 - 2147483648;
(n) s0 = -1 * s0;
(o) s1 = s1 * a0;
(p) s2 = srt(s0^2 + 56) / a3;
(q) s3 = s1 - s2 / s3;
(r) s4 = s4 * 8;
(s) s5 = π * s5;

2.2 Analyze the assembly language code that you developed for each of the above pseudocode expressions and calculate the number of clock cycles required to fetch and execute the code corresponding to each expression. Assume it takes one clock cycle to fetch and execute every instruction except multiply, which requires 32 clock cycles, and divide, which requires 38 clock cycles.

2.3 Show how the following expression can be **evaluated** in MIPS assembly language, without modifying the contents of the “s” registers:

$t0 = ($s1 - $s0 / $s2) * $s4 ;

2.4 The datapath diagram for the MIPS architecture shown in Figure 1.1 with only one memory module is referred to as a von Neumann architecture. Most implementations of the MIPS architecture use a Harvard architecture, where there are separate memory modules for instructions, and data. Draw such a datapath diagram.

2.5 Show how the following pseudocode expression can be **efficiently** implemented in MIPS assembly language:

$t0 = $s0 / 8 - 2 * $s1 + $s2;

CHAPTER 3

Number Systems

Where do you find the trees in Minnesota?
Between da twos and da fours.

3.1 Introduction

The decimal number system uses 10 different digits (symbols), (0,1, 2, 3, 4, 5, 6 ,7 ,8 ,9). The binary number system uses only two digits, 0 and 1, which are represented by two different voltage levels within a computer's electrical circuits. Any value can be represented in either number system as long as there is no limit on the number of digits we can use. In the case of the MIPS architecture, values in registers and in memory locations are limited to 32 bits. The range of values that can be stored in a register or a memory location is -2,147,483,648 to +2,147,483,647 (Assuming the use of the two's complement number system). In this chapter, we will provide a method for converting values in one number system to another. We will also discuss the process of binary addition and subtraction; and how to detect if overflow occurred when performing these operations.

3.2 Positional Notation

No doubt, the reason we use the decimal number system is that we have ten fingers. Possibly, for primitive cultures it was sufficient to communicate a quantity by holding up a corresponding number of fingers between one and ten, and associating a unique sound or symbol with these ten quantities.

The Babylonians used written symbols for numbers for thousands of years before they invented the zero symbol. The zero symbol is the essential component that makes it possible to use the positional number system to represent an unlimited range of integer quantities. When we express some quantity such as "2056" in the decimal number system we interpret this to mean: 2*1000 + 0*100 + 5*10 + 6*1

The polynomial representation of 2056 in the base ten number system is:

$$N = 2 * 10^{3} + 0 * 10^{2} + 5 * 10^{1} + 6 * 10^{0}$$

Let us assume that aliens visit us from another galaxy where they have evolved with only eight fingers. If these aliens were to communicate the quantity "2056" in the base 8 number system (octal), how would you find the equivalent value as a decimal number? The method is to evaluate the following polynomial:

$$N = 2 * 8^{3} + 0 * 8^{2} + 5 * 8^{1} + 6 * 8^{0}$$

$$N = 2 * 512 + 0 * 64 + 5 * 8 + 6 = 1070$$

Therefore, 2056 in the base eight number system is equivalent to 1070 in the base ten number system. Notice that these aliens would only use eight different symbols for their eight different digits. These symbols might be (0,1, 2, 3, 4, 5, 6, 7) or they might be some other set of symbols such as (Ω, √, ∑, ß, ∂, &, $, %); initially the aliens would have to define their digit symbols by holding up an equivalent number of fingers.

3.3 Converting Binary Numbers to Decimal Numbers

Polynomial expansion is the key to converting a number in any alien number system to the decimal number system, The binary number system may be an alien number system as far as you are concerned, but you now possess the tool to convert any binary number to the equivalent decimal value. As an exercise, convert the binary number 011010 to decimal.

$$N = 0 * 2^{5} + 1 * 2^{4} + 1 * 2^{3} + 0 * 2^{2} + 1 * 2^{1} + 0 * 2^{0}$$

Memorizing the following powers of two is an essential component of mastering this number conversion process.

2^{0}	2^{1}	2^{2}	2^{3}	2^{4}	2^{5}	2^{6}	2^{7}	2^{8}	2^{9}	2^{10}
1	2	4	8	16	32	64	128	256	512	1024

$$N = 0 * 32 + 1 * 16 + 1 * 8 + 0 * 4 + 1 * 2 + 0 * 1$$

Therefore, the binary number 011010 is equivalent to 26 in the decimal number system.

3.4 Detecting if a Binary Number is Odd or Even

Given any binary number, there is a simple way to determine if the number is odd or even. If the right most digit in a binary number is a one, then the number is odd. For example 00011100 is an even number, which is the value 28 in the decimal number system. The value 0001001 is an odd number, specifically the value 9 in decimal.

When writing MIPS assembly code the most efficient method for determining if a number is odd or even is to extract the right most digit using the logical AND instruction followed by a branch on zero instruction. This method requires only two clock cycles of computer time to accomplish the task. The use of division to determine if a number is odd or even is very inefficient because it can take as much as 38 clock cycles for the hardware to execute the division instruction. The following is a segment of MIPS assembly language code that will add one (1) to register $s1 only if the contents of register $s0 is an odd number:

```
        andi    $t8,     $s0, 1  # Extract the Least Significant Bit (LSB)
        beqz    $t8      even    # If LSB is a zero Branch to even
        addi    $s1,     $s1, 1  # Increment count in s1
even:
```

3.5 Multiplication by Constants that are a Power of Two

Another important feature of the binary number system is that multiplication by two (2) may be accomplished by shifting the number left one bit. Multiplication by four (4) can be accomplished by shifting left two bits. In general, multiplication by any number that is a power of two can be accomplished in one clock cycle using a shift left instruction. For many implementations of the MIPS architecture it takes 32 clock cycles to execute the multiply instruction, but it takes only one clock cycle to execute a shift instruction. Let us suppose that the following pseudocode describes a desired operation:

```
        $v1 = $t3 * 32
```

The most efficient way to execute this in MIPS assembly language would be to perform a shift left logical by 5 bits.

```
        sll     $v1, $t3,  5     #  $v1 = $t3 << 5
```

Notice that the constant 5 specifies the shift amount, and you should recall that:

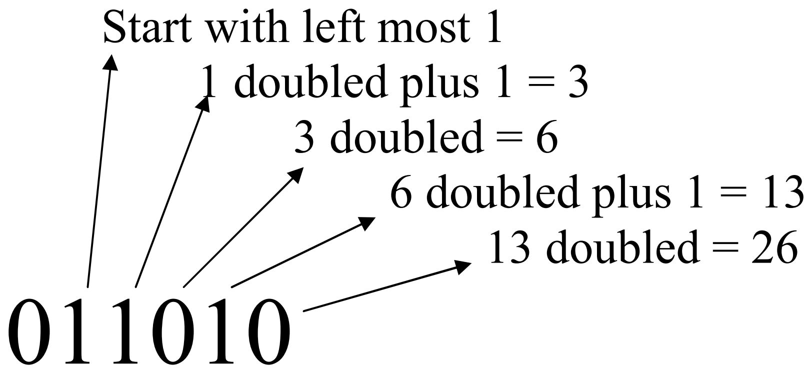
$$2^5 = 32$$

Let us suppose the original value in register $t3 is the binary number 0000000000011010, which is equivalent to 26 in the decimal number system. After shifting the binary number left 5 bits we would have 0000001101000000, which is 832 in the decimal number system (26 * 32). The analogous situation in the decimal number system is multiplication by ten. Taking any decimal number and shifting it left one digit is equivalent to multiplication by ten.

3.6 The Double and Add Method

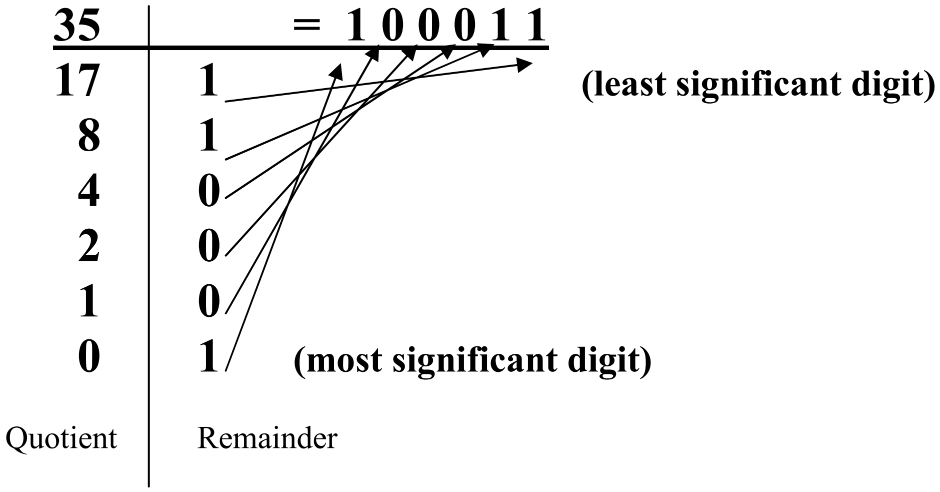
A quick and efficient method for converting binary numbers to decimal involves visually scanning the binary number from left to right, starting with the left most 1. As you visually scan to the right, double the value accumulated so far, and if the next digit to the right is a 1, add 1 to your accumulating sum.

In a previous example, we had the binary number 00011010, which is equivalent to 26 decimal. Let us use the double and add method to convert from binary to decimal.



3.7 Converting Decimal Numbers to Binary Numbers

A simple procedure to convert any decimal numbers to binary follows. Essentially this procedure is the inverse of the double and add process explained above. The process involves repeatedly dividing the decimal number by two and recording the quotient and the remainder. After each division by two, the remainder is the next digit in the binary representation of the number. Recall that any time an odd number is divided by two the remainder is one. So the remainder obtained after performing the first division by two corresponds to the least significant digit in the binary number. The remainder after performing the second division is the next more significant digit in the binary number.



3.8 The Two's Complement Number System

Up to this point, there has been no mention of how to represent negative binary numbers. With the decimal number system, we commonly use a sign magnitude representation. We do **not** use a sign magnitude representation for binary numbers. For binary numbers we use the **Signed Two's Complement Number System**. (Sometimes referred to as the radix complement.) The major benefit of the two's complement number system is that it simplifies the design of the hardware to perform addition and subtraction. Numbers represented in the two's complement number system have a straightforward polynomial expansion. For example, an 8-bit binary number would be evaluated using the following polynomial expansion:

$$N = - d_7 * 2^7 + d_6 * 2^6 + d_5 * 2^5 + ...+ d_1 * 2^1 + d_0$$

In the two's complement number system, all numbers that have a one in the most significant digit (MSD) are negative numbers. The most significant digit has a **negative** weight associated with it. In the two's complement number system, the value plus 1 as an 8-bit number would be 00000001 and minus 1 would be 11111111. Evaluate the polynomial to verify this fact.

3.9 The Two's Complement Operation

When we take the two's complement of a number, the result will be the negative of the value we started with. One method to find the two's complement of any number is to complement all of the digits in the binary representation and then add one to this new binary value.

For example, take the value plus 26, which as an 8-bit binary number is 00011010. What does the value **minus** 26 look like in the two's complement number system? Performing the two's complement operation on 00011010 we get 11100110.

Original value 26	00011010	
Complement every Bit	11100101	
Add one	+1	
This is the value **minus** 26	11100110	in the two's complement number system.

Evaluate the polynomial to verify that this is the correct binary representation of minus 26.

3.10 A Shortcut for Finding the Two's Complement of any Number

There is a simple one-step procedure that can be used to perform the two's complement operation on any number. This is the **preferred procedure** because it is faster and less prone to error. With this procedure the original number is scanned from right to left, leaving all least significant *zeros* and the first *one* unchanged, and then complementing the remaining digits to the left. Let's apply this procedure with the following example. Suppose we start with the value **minus** 26. If we perform this shortcut two's complement operation on **minus** 26 we should get **plus** 26 as a result.

Original value -26 $\underbrace{111001}_{\text{Complemented}}10$

Resulting value +26 00011010

3.11 Sign Extension

When the MIPS processor is executing code, there are a number situations where 8-bit and 16-bit binary numbers need to be expanded into a 32-bit representation. For values represented in the two’s complement number system, this a trivial process. The process simply involves extending a copy of the sign bit into all of the additional significant digits. For example, the value 6 represented as an 8-bit binary number is: “ 00000110”, and the value 6 as a 32-bit binary number is “00000000000000000000000000000110”.

The same rule applies for negative numbers. For example the value minus 6 represented as an 8-bit binary number is: 11111010, and the value -6 as a 32-bit binary number is 11111111111111111111111111111010.

3.12 Binary Addition

With the two’s complement number system adding numbers is a simple process, even if the two operands are of different signs. The sign of the result will be generated correctly as long as overflow does not occur. (See section 3.14) Simply keep in mind that if the sum of three binary digits is two or three, a carry of a 1 is generated into the next column to the left. Notice in the example below, in the third column, we add **one** plus **one** and get **two** (10) for the result. The sum bit is a **zero** and the carry of **one** is generated into the next column. In this next column, we add the carry plus the two **ones** within the operands and get **three** (11) as a result. The sum bit is a **one** and the carry of **one** is generated into the next column.

Decimal		**Binary**
	29	**00011101**
	14	**00001110**
Sum	**43**	**00101011**

3.13 Binary Subtraction

Computers perform subtraction by adding the two’s complement of the subtrahend to the minuend. This is **also** the simplest method for humans to use when dealing with binary numbers. Let us take a simple 8-bit example where we subtract 26 from 35.

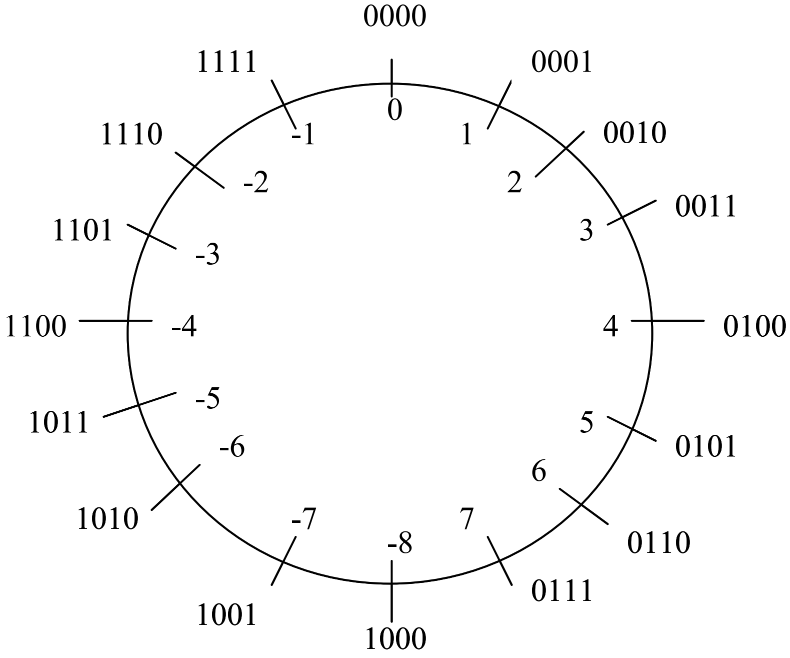
Minuend is 35	**00100011**		**00100011**
Subtrahend is 26	**-00011010**	**Take two’s complement and add**	**+11100110**
			00001001

Notice when we add the two binary numbers together, there is a carry out. We don’t care if there is carry out. Carry out does **not** indicate that overflow occurred. Converting the binary result to decimal we get the value nine, which is the correct result.

3.14 Overflow Detection

In the two's complement number system, detection of overflow is a simple proposition. When adding numbers of opposite signs, overflow is impossible. When adding numbers of the same sign, the result must have the same sign as the operands, otherwise overflow occurred. The most important thing to remember is that a carry out at the most significant stage does **not** signify that overflow has occurred in the two's complement representation. When two negative numbers are added together, there will always be a carry out at the most significant digit, but this does **not** mean that overflow occurred.

In mathematics, we refer to a number line that goes to infinity in the positive and negative domains. In the case of computers with limited precision, we do not have a number line, instead we have a number circle. When we add one to the most positive value, overflow occurs, and the result is the most negative value in the two's complement number system. Let us take a very small example. Suppose we have a computer with only 4 bits of precision. The most positive value is 7, which is (0111) in binary. The most negative value is minus 8, which is (1000) in binary. With the two's complement number system, the range of values in the negative domain is one greater than in the positive domain.



3.15 Hexadecimal Numbers

The hexadecimal number system is heavily used by assembly language programmers because it provides a compact method for communicating binary information. The hexadecimal number system is a base 16 number system. In this case, the 16 unique symbols used as digits in hexadecimal numbers are (0,1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F). A convention has been adopted to identify a hexadecimal number. The two characters “**0x**” always precede a hexadecimal number. For example, the hexadecimal number 0x1F corresponds to the decimal value 31, and corresponds to 00011111 in the binary number system.

The value 16 is equal to 2^4. Converting between binary and hexadecimal representation requires no computation, it can be done by inspection. The following table is the key to making these conversions. Converting a hexadecimal number to binary simply involves replacing every hexadecimal digit with the corresponding 4-bit code in the table below. For example, 0xA2F0 in hexadecimal corresponds to 1010001011110000 in binary. To convert a binary number to hexadecimal, start with the rightmost bits and break up the binary number into groups of 4-bits each. Then using the table below, replace every 4-bit code with the corresponding hexadecimal digit. For example, the 16-bit binary number 1111011011100111 is equivalent to 0xF6E7. Notice that hexadecimal numbers that begin with the value 8 or above are negative numbers because in the corresponding binary representation the Most Significant Digit (MSD) is a one.

Decimal	**Binary**	**Hexadecimal**
0	**0000**	**0x0**
1	**0001**	**0x1**
2	**0010**	**0x2**
3	**0011**	**0x3**
4	**0100**	**0x4**
5	**0101**	**0x5**
6	**0110**	**0x6**
7	**0111**	**0x7**
8	**1000**	**0x8**
9	**1001**	**0x9**
10	**1010**	**0xA**
11	**1011**	**0xB**
12	**1100**	**0xC**
13	**1101**	**0xD**
14	**1110**	**0xE**
15	**1111**	**0xF**

Exercises

3.1 Convert the decimal number 35 to an 8-bit binary number.

3.2 Convert the decimal number 32 to an 8-bit binary number.

3.3 Using the double and add method convert 00010101 to a decimal number.

3.4 Using the double and add method convert 00011001 to a decimal number.

3.5 Explain why the Least Significant digit of a binary number indicates if the number is odd or even.

3.6 Convert the binary number 00010101 to a hexadecimal number.
3.7 Convert the binary number 00011001 to a hexadecimal number.
3.8 Convert the hexadecimal number 0x15 to a decimal number.
3.9 Convert the hexadecimal number 0x19 to a decimal number.
3.10 Convert the decimal number -35 to an 8-bit two's complement binary number.
3.11 Convert the decimal number -32 to an 8-bit two's complement binary number.
3.12 Assuming the use of the two's complement number system find the equivalent decimal values for the following 8-bit binary numbers:

(a) 10000001
(b) 11111111
(c) 01010000
(d) 11100000
(e) 10000011

3.13 Convert the base 8 number 204 to decimal
3.14 Convert the base 7 number 204 to decimal
3.15 Convert the base 6 number 204 to decimal
3.16 Convert the base 5 number 204 to decimal
3.17 Convert the base 10 number 81 to a base 9 number.
3.18 For each row of the table below convert the given number to each of the other two bases, assuming the two's complement number system is used.

16 Bit Binary	Hexadecimal	Decimal
1111111100111100		
	0xFF88	
		-128
1111111111111010		
	0x0011	
		-25

3.19 You are given the following two numbers in two's complement representation. Perform the binary addition. Did signed overflow occur? ____ Explain how you determined whether or not overflow occurred.

01101110
00011010

3.20 You are given the following two numbers in two's complement representation. Perform the binary subtraction. Did signed overflow occur? ____ Explain how you determined whether or not overflow occurred.

11101000
-00010011

3.21 Sign extend the 2 digit hex number 0x88 to a 4 digit hex number. 0x__________.

3.22 The following subtract instruction is located at address 0x00012344. What are the two possible values for the contents of the Program Counter (PC) register after the branch instruction has executed? 0x____________ 0x____________
This branch instruction is described in Appendix C.

```
loop:       addi   $t4, $t4, -8
            sub    $t2, $t2, $t0
            bne    $t4, $t2,loop
```

3.23 You are given the following two 8-bit binary numbers in the two's complement number system. What values do they represent in decimal?

X = $\underline{10010100}_{2}$ = $\underline{\hspace{2.5cm}}_{10}$ Y = $\underline{00101100}_{2}$ = $\underline{\hspace{2.5cm}}_{10}$

Perform the following arithmetic operations on X and Y. Show your answers as 8-bit binary numbers in the two's complement number system. To subtract Y from X, find the two's complement of Y and add it to X. Indicate if overflow occurs in performing any of these operations.

X+Y	X-Y	Y-X
10010100	**10010100**	**00101100**
00101100		

Show a solution to the same arithmetic problems using the hexadecimal representations of X and Y.

3.24 The following code segment is stored in memory starting at memory location 0x00012344. What are the two possible values for the contents of the PC after the branch instruction has executed? In the comments field, add a pseudocode description for each instruction.
0x______________________ 0x______________________

```
      loop:       lw     $t0, 0($a0)     #
                  addi   $a0, $a0, 4     #
                  andi   $t1, $t0, 1     #
                  beqz   $t0, loop       #
```

CHAPTER 4

PCSpim The MIPS Simulator

My software never has bugs —
it just develops random features.

4.1 Introduction

A simulator of the MIPS R2000/R3000 is available for free down-loading at:

http://www.cs.wisc.edu/~larus/spim.html

There is a Unix version, and a Windows version called **PCSpim.** The name SPIM is just MIPS spelled backward. Jim Larus at the University of Wisconsin developed the initial version of SPIM in 1990. The major improvement of the latest version over previous versions is a feature to save the log file. After saving the log file, it can be opened using a text editor. Using the cut and paste tools, we can now print anything of interest related to the program that just ran. All of the examples presented in this book will be for the PCSpim Version. After down-loading PCSpim, the "Help File" should be reviewed.

Morgan Kaufmann Publishers have generously provided an on line version of Appendix A from the textbook "Computer Organization and Design: The Hardware/Software Interface." This is a more complete and up-to-date version of SPIM documentation than the one included with SPIM. This document provides a detailed explanation of the Unix version of SPIM. It is a suggested supplement to this textbook and is available at:

http://www.cs.wisc.edu/~larus/SPIM/cod-appa.pdf

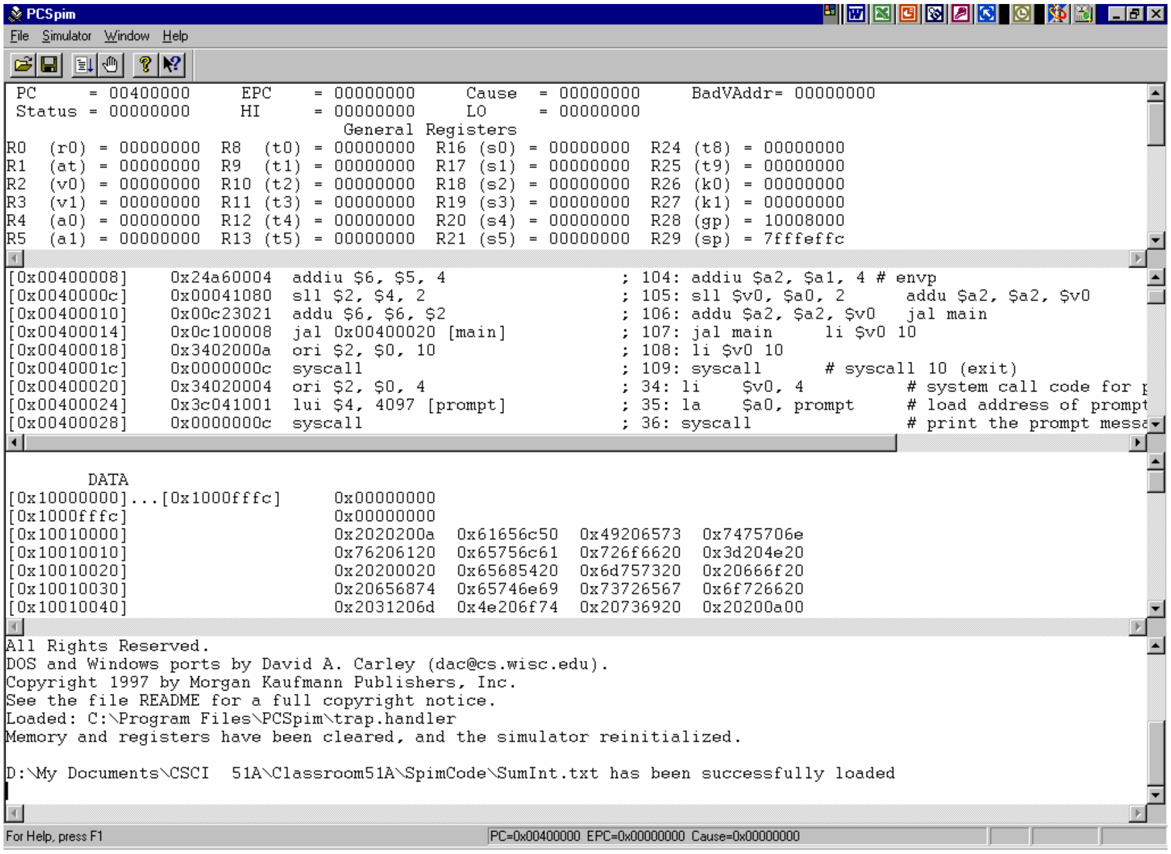
4.2 Advantages of a Simulator

There are a number of advantages in using a simulator when first learning to program in assembly language. Number one, we can learn the language without having to buy a MIPS based computer. The simulator provides debugging features. We can single step through a program and watch the contents of the registers change as each instruction executes, and we can also look at the contents of memory as the instructions execute. We can set breakpoints. A programming mistake in a simulated environment will not cause the actual machine running the simulation to crash. A programming mistake in a simulated environment will usually result in a simulated exception, where a trap handler will print a message to help us identify what instruction caused the exception.

This simulator does have some disadvantages. There is no linking loader phase. When developing an application to run on an actual MIPS processor we would assemble each code module separately. Then when the time comes to run the program, the loader would be used to relocate and load all the modules into memory and link them. Also this SPIM assembler/simulator does not provide a capability for users to define macros.

4.3 The Big Picture

Here are the steps we go through to write and run a program using PCSpim. First we use a word processor such as **Notepad** to create an assembly language source program file. Save the file as “Text Only.” Launch PCSpim. In the “**File**” pull-down menu select “**Open**” or just click the first icon on the tool bar. Select “**All Files**” for the “**Files of type**.” Figure 4.1 below shows the bottom **Messages** window announcing the program has been successfully loaded. Three other windows are also displaying useful information. Specifically the **Registers** window, which is the top window, displays the contents of the MIPS registers. The next window down displays the contents of the **Text Segment**, and the window below that displays the **Data Segment**. To open the **Console** go to the Window pull down menu and select Console.



```
 PC      = 00400000     EPC     = 00000000     Cause    = 00000000     BadVAddr= 00000000
 Status  = 00000000     HI      = 00000000     LO       = 00000000
                              General Registers
R0  (r0) = 00000000  R8  (t0) = 00000000  R16 (s0) = 00000000  R24 (t8) = 00000000
R1  (at) = 00000000  R9  (t1) = 00000000  R17 (s1) = 00000000  R25 (t9) = 00000000
R2  (v0) = 00000000  R10 (t2) = 00000000  R18 (s2) = 00000000  R26 (k0) = 00000000
R3  (v1) = 00000000  R11 (t3) = 00000000  R19 (s3) = 00000000  R27 (k1) = 00000000
R4  (a0) = 00000000  R12 (t4) = 00000000  R20 (s4) = 00000000  R28 (gp) = 10008000
R5  (a1) = 00000000  R13 (t5) = 00000000  R21 (s5) = 00000000  R29 (sp) = 7fffeffc
```

```
[0x00400008]    0x24a60004  addiu $6, $5, 4                 ; 104: addiu $a2, $a1, 4 # envp
[0x0040000c]    0x00041080  sll $2, $4, 2                   ; 105: sll $v0, $a0, 2      addu $a2, $a2, $v0
[0x00400010]    0x00c23021  addu $6, $6, $2                 ; 106: addu $a2, $a2, $v0   jal main
[0x00400014]    0x0c100008  jal 0x00400020 [main]           ; 107: jal main     li $v0 10
[0x00400018]    0x3402000a  ori $2, $0, 10                  ; 108: li $v0 10
[0x0040001c]    0x0000000c  syscall                         ; 109: syscall      # syscall 10 (exit)
[0x00400020]    0x34020004  ori $2, $0, 4                   ; 34: li    $v0, 4          # system call code for p
[0x00400024]    0x3c041001  lui $4, 4097 [prompt]           ; 35: la    $a0, prompt     # load address of prompt
[0x00400028]    0x0000000c  syscall                         ; 36: syscall               # print the prompt messa
```

```
        DATA
[0x10000000]...[0x1000fffc]     0x00000000
[0x1000fffc]                    0x00000000
[0x10010000]                    0x2020200a  0x61656c50  0x49206573  0x7475706e
[0x10010010]                    0x76206120  0x65756c61  0x726f6620  0x3d204e20
[0x10010020]                    0x20200020  0x65685420  0x6d757320  0x20666f20
[0x10010030]                    0x20656874  0x65746e69  0x73726567  0x6f726620
[0x10010040]                    0x2031206d  0x4e206f74  0x20736920  0x20200a00
```

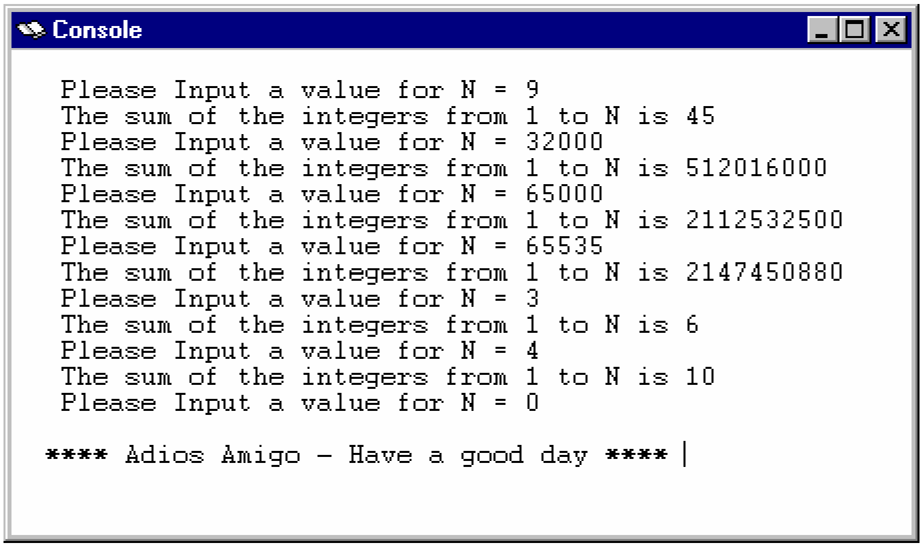
```
All Rights Reserved.
DOS and Windows ports by David A. Carley (dac@cs.wisc.edu).
Copyright 1997 by Morgan Kaufmann Publishers, Inc.
See the file README for a full copyright notice.
Loaded: C:\Program Files\PCSpim\trap.handler
Memory and registers have been cleared, and the simulator reinitialized.

D:\My Documents\CSCI  51A\Classroom51A\SpimCode\SumInt.txt has been successfully loaded
```

For Help, press F1 PC=0x00400000 EPC=0x00000000 Cause=0x00000000

Figure 4.1

In examining the contents of the **Registers** window at the top, you will find that all the registers initially containing zero except for the Program Counter (PC) and the Stack Pointer ($sp). A short sequence of instructions in the kernel of the operating system is always executed to launch a user program. That sequence of instructions starts at address 0x00400000.

To run the program, pull down the “**Simulator**” menu and select “**Go**” or just click on the third icon. The program that ran for this example “Sum of Integers” is shown in Section 2.2. For this example, we ran and responded with values every time we were prompted to “Please Input a value.” Analysis of the code will reveal the program will terminate when the value zero is typed in. When the program terminates, it would be appropriate to “**Save Log File**.” This can be done by clicking the second icon on the tool bar or selecting this option under the “**File**” pull down menu.



```
 Please Input a value for N = 9
 The sum of the integers from 1 to N is 45
 Please Input a value for N = 32000
 The sum of the integers from 1 to N is 512016000
 Please Input a value for N = 65000
 The sum of the integers from 1 to N is 2112532500
 Please Input a value for N = 65535
 The sum of the integers from 1 to N is 2147450880
 Please Input a value for N = 3
 The sum of the integers from 1 to N is 6
 Please Input a value for N = 4
 The sum of the integers from 1 to N is 10
 Please Input a value for N = 0

**** Adios Amigo - Have a good day ****
```

Figure 4.2

An input value for N greater than 65535 will result in a computed value that exceeds the range for a 32-bit representation. The hardware contains a circuit to detect when overflow occurs, and an exception is generated when overflow is detected. One of the options under “Simulation” menu is “**Break**.” If a program ever gets into an infinite loop, use this option to stop the simulation. Always “**Reload”** the program after the program has been aborted using the “**Break**” option.

An examination of the “**Registers**” window after the program has run to completion shows registers $v0 and $t0 both contain the value ten (0x0000000a). Can you explain why? It is especially instructive to use the “**Single Step**” option to run the program and to watch the contents of these registers change as each instruction is executed. In “**Single Step**”(function Key F10) mode, the next instruction to be executed will be highlighted in the “**Text Segment**” window.

4.4 Analyzing the Text Segment

An examination of the following “Text Segment” is especially enlightening. This “Text Segment” taken from the Log File corresponds to the program in Section 2.2. The first column is the hexadecimal address of the memory location where each instruction was stored into memory. The second column shows how each assembly language instruction is encoded in machine language. The last column shows the original assembly language code.

```
   Address         Machine Language                          Original Code
[0x00400020]    0x34020004  ori $2, $0, 4                    ; 34: li  $v0, 4
[0x00400024]    0x3c041001  lui $4, 4097 [prompt]            ; 35: la  $a0, prompt
[0x00400028]    0x0000000c  syscall                          ; 36: syscall
[0x0040002c]    0x34020005  ori $2, $0, 5                    ; 38: li   $v0, 5
[0x00400030]    0x0000000c  syscall                          ; 39: syscall
[0x00400034]    0x1840000d  blez $2 52 [end-0x00400034]      ; 41: blez  $v0,  end
[0x00400038]    0x34080000  ori $8, $0, 0                    ; 42: li      $t0, 0
[0x0040003c]    0x01024020  add $8, $8, $2                   ; 44: add   $t0, $t0, $v0
[0x00400040]    0x2042ffff    addi $2, $2, -1                ; 45: addi  $v0, $v0, -1
[0x00400044]    0x14403ffe   bne $2, $0, -8 [loop-0x00400044] ; 46: bnez  $v0, end
[0x00400048]    0x34020004  ori $2, $0, 4                    ; 47: li   $v0, 4
[0x0040004c]    0x3c011001  lui $1, 4097 [result]            ; 48: la  $a0, result
[0x00400050]    0x34240022  ori $4, $1, 34 [result]
[0x00400054]    0x0000000c  syscall                          ; 49: syscall
[0x00400058]    0x34020001  ori $2, $0, 1                    ; 51: li   $v0, 1
[0x0040005c]    0x00082021  addu $4, $0, $8                  ; 52: move  $a0,  $t0
[0x00400060]    0x0000000c  syscall                          ; 53: syscall
[0x00400064]    0x04013fef   bgez $0 -68 [main-0x00400064]   ; 54: b  main
[0x00400068]    0x34020004  ori $2, $0, 4                    ; 56: li   $v0, 4
[0x0040006c]    0x3c011001  lui $1, 4097 [bye]               ; 57: la   $a0, bye
[0x00400070]    0x3424004d  ori $4, $1, 77 [bye]
[0x00400074]    0x0000000c  syscall                          ; 58: syscall
[0x00400078]    0x3402000a  ori $2, $0, 10                   ; 60: li   $v0, 10
[0x0040007c]    0x0000000c  syscall                          ; 61: syscall
```

The assembler is the program that translates assembly language instructions to machine language instructions. To appreciate what this translation process entails, every student should translate a few assembly language instructions to machine language instructions. We will now demonstrate this translation process. We will be using the information in Appendix C to verify that 0x3402000a is the correct machine language encoding of the instruction “ori $2, $0, 10.”, which is the next to last instruction in the “Text Segment” above, located at memory location [0x00400078].

In Appendix C we are shown how this instruction is encoded in binary.

ori Rt, Rs, Imm **# RF[Rt] = RF[Rs] OR Imm**

Op-Code	Rs	Rt	Imm
001101	sssss	ttttt	iiiiiiiiiiiiiiii

The macro instruction "load Immediate" (li $v0, 10) was the original assembly language instruction. Looking in Appendix D we find that the actual instruction (ori $2, $0, 10) is used to accomplish the task of loading register $v0 with the value ten. In Figure 1.2 we find that register number "2" corresponds to the register with the name $v0. So now all we have to do is fill in the binary encoding for all the specified fields of the instruction. Specifically the immediate value is ten, which is "0000000000001010," Rt is "00010," and Rs is "00000." The final step involves translating the 32-bit binary value to hexadecimal. The alternating shading is helpful for distinguishing each 4-bit field.

Op-Code	Rs	Rt	Imm
001101	00000	00010	0000000000001010

3 4 0 2 0 0 0 a

Use the information in Appendix C to verify that 0x2042ffff is the correct machine language encoding of the instruction "addi $v0, $v0, -1." This instruction is located at memory location [0x00400040]. Notice that the instructions (li) and (la) in the original code are macro instructions. Looking at the middle column we see how each macro instruction was replaced with one or more actual MIPS instructions.

4.5 Analyzing the Data Segment

Within the data segment of this example program, the following three ASCII strings were defined using the assembler directive "**.asciiz**."

```
            .data
prompt:     .asciiz    "\n   Please Input a value for N = "
result:     .asciiz    "   The sum of the integers from 1 to N is "
bye:        .asciiz    " **** Adios Amigo – Have a good day ****"
```

This directive tells the assembler to place the ASCII codes corresponding to the strings within quotes into memory sequentially one after another. Notice the "**z**" at the end of the directive indicates that a null-terminated string should be created. The ASCII null character is an 8-bit binary value zero. (See Appendix B) The Print String system utility stops printing when it finds a null character. Special characters in strings follow the C convention:

- newline \n
- tab \t
- quote \"

To begin this analysis, let’s take the following characters, “**Please**”. Using Appendix B, look up the ASCII code for each of the characters:

P –> 0x50 **l** –> 0x6c **e** –> 0x65 **a** –> 0x61 **s** –> 0x73 **e** –> 0x65

The operating system has allocated space in memory for our Data Segment to begin at address 0x10010000. Locating this ASCII code sequence is an interesting exercise, complicated by the fact that Intel processors store characters within words in reverse order, and PCSpim is running on an Intel-based platform. Below we have highlighted the location of these characters within the data segment of memory. Can you find the null character that terminates the first string?

```
                                  a  e  l  P          e  s
[0x10010000]          0x2020200a  0x61656c50  0x49206573  0x7475706e
[0x10010010]          0x76206120  0x65756c61  0x726f6620  0x3d204e20
[0x10010020]          0x20200020  0x65685420  0x6d757320  0x20666f20
[0x10010030]          0x20656874  0x65746e69  0x73726567  0x6f726620
[0x10010040]          0x2031206d  0x4e206f74  0x20736920  0x20200a00
```

4.6 System I/O

The developers of the SPIM simulator wrote primitive decimal input/output functions. Access to these functions is accomplished by generating a software exception. The MIPS instruction a programmer uses to invoke a software exception is "syscall." There are ten different services provided. In your programs, you specify what service you want to perform by loading register $v0 with a value from 1 to 10. The table below describes each system service. We will be using only those services that are shown highlighted.

Service	Code in $v0	Arguments	Results
Print an Integer	**1**	$a0 = Integer Value to be Printed	
Print Float	2		
Print Double	3		
Print String	**4**	$a0 = Address of String in Memory	
Read an Integer in (from the keyboard)	**5**		Integer Returned in $v0
Read Float	6		
Read Double	7		
Read a String in (from the keyboard)	**8**	$a0 = Address of Input Buffer in Memory $a1 = Length of Buffer (*n*)	
Sbrk	9	$a0 = amount	Address in $v0
Exit	**10**		

4.7 Deficiencies of the System I/O Services

These primitive I/O functions provided by the developers of SPIM have some undesirable characteristics:

- The decimal output function prints left justified. In most situations we would rather see the numbers printed right justified.
- The decimal input function always returns some value even if the series of keystrokes from the keyboard does not correctly represent a decimal number. You can verify this by running the following program and typing the following strings, each of which does not correctly represent a decimal integer value or the value exceeds the range that can be represented as a 32-bit binary value.

```
2.9999999
1  9
3A
4-2
-4+2
ABCD
2,147,463,647
2147463648
-2,147,463,648
-2147463649
```

```
            .data
prompt:     .asciiz   “\n   Please Input a value”
bye:        .asciiz   “\n  **** Adios Amigo - Have a good day ****”
            .globl    main
            .text
main:
            li          $v0, 4                  # system call code for Print String
            la          $a0, prompt             # load address of prompt into $a0
            syscall                             # print the prompt message

            li          $v0, 5                  # system call code for Read Integer
            syscall                             # reads the value into $v0
            beqz        $v0,  end               # branch to end if  $v0  =  0
            move        $a0, $v0
            li          $v0, 1                  # system call code for Print Integer
            syscall                             # print
            b           main                    # branch to main

end:        li          $v0, 4                  # system call code for Print String
            la          $a0, bye                # load address of msg. into $a0
            syscall                             # print the string

            li          $v0, 10                 # terminate program run and
            syscall                             # return control to  system
```

Exercises

4.1 Translate the following assembly language instructions to their corresponding machine language codes as they would be represented in hexadecimal.
(Hint – Refer to Appendix C and Appendix D.)

```
loop:   addu    $a0, $0, $t0
        ori     $v0, $0, 4
        syscall
        addi    $t0, $t0, -1
        bnez    $t0, loop
        andi    $s0, $s7, 0xffc0
        or      $a0, $t7, $s0
        sb      $a0, 4($s6)
        srl     $s7, $s7, 4
```

4.2 What is the character string corresponding to the following ASCII codes?
0x2a2a2a2a 0x69644120 0x4120736f 0x6f67696d 0x48202d20 0x20657661
(Hint – Remember that for simulations running on Intel–based platforms, the characters are stored in reverse order within each word.)

CHAPTER 5

Algorithm Development

How would you describe Al Gore playing the drums?
Algorithm.

5.1 Introduction

Everyone knows exercise is the key to developing stronger muscles and muscle tone. Some students pay a good deal of money on sport club memberships, just so they can use the exercise equipment. The remaining chapters of this book provide a wealth of exercises students can use to strengthen their "**mental muscle.**" Challenging yourself with these exercises is essential to becoming an efficient, productive assembly language programmer. Discussion and collaboration with other students can be a valuable component in strengthening your mental muscle. You should discuss the programming assignments, general strategies, or algorithms with other people, but do not collaborate in the detail development or actually writing of programs. The copying of programs (entirely or in part) is considered plagiarism. Obviously, if you only watch people workout at the sport club, you will not improve your own physical condition.

Students using this book will be presented with many challenging assembly language programming exercises. It is suggested each student should initially attempt to individually develop a pseudocode solution to each exercise. Next, the students should show each other their pseudocode, and discuss ways to refine their algorithms to make them more efficient in terms of space and time.

The final step is to translate the pseudocode to assembly language code, and to calculate performance indexes for their solutions. The two performance indexes are:

a. The number of words of assembly language code. (space)
b. The number of clock cycles required to execute the code. (time)

5.2 Instructions that Perform Logical Operations

The logical operators implemented as MIPS instructions are: AND, NOR, OR, and EXCLUSIVE-OR (XOR). These instructions are extremely useful for manipulating, extracting and inserting specifically selected bits within a 32-bit word. Programmers who really understand these instructions have a significant advantage in developing code with superior performance indexes. For those unfamiliar with these bit-wise logical operators, the following truth table defines each of the operators. In the table, x and y represent two Boolean variables, and the results of each logical operator appear in the corresponding column. Understand that these logical operations are performed upon the corresponding bits in two source registers, with each containing 32 bits, and the resulting 32 bits are stored in a destination register. The truth table describes four possible combinations of two variables, and defines the corresponding results produced by the logical operators.

Looking at the truth table for the AND operation, we find the only case where we get a one (1) for a result is when x **and** y are both one (1). For the OR operation, we get a one for a result when either x **or** y is a one (1). The operator NOR stands for NOT-OR, and we notice the row in the truth table that defines the NOR operation is simply the complement of the row describing the OR operation. For the EXCLUSIVE-OR, we get a one in the result only when the x and y input variables are different. (Not the same)

Input x	0	0	1	1
Input y	0	1	0	1
x AND y	0	0	0	1
x OR y	0	1	1	1
x NOR y	1	0	0	0
x XOR y	0	1	1	0

We will refer to the Least Significant Bit (LSB) in a word as bit 0, and we will refer to the Most Significant Bit (MSB) as bit 31. Note that if a logical operator has a 16-bit immediate operand, the hardware automatically zero extends the immediate value to form a 32-bit operand. One of the operands for a logical operator is usually referred to as a "mask." The bit pattern within a mask is used to select which fields within the other register we wish to manipulate, or extract information from. For these examples, assume the following masks are stored in registers $t8, and $t9:

$t8 = `0xffffffc0` = 111111111111111111111111111111111111111111000000$_2$

$t9 = `0x0000003f` = 000000000000000000000000000000000000000000111111$_2$

To move a copy of $t0 into $t1 with the lower 6 bits cleared to zero, the instruction to accomplish this would be:

```
          and     $t1, $t0, $t8              # $t1 = $t0 & $t8
```

In this case, anywhere there is a zero in the mask, we get zero in $t0, and anywhere there is a one in the mask, we get a copy of $t0 in $t1.

To move a copy of $t0 into $t1 with the lower 6 bits set to one, the instruction to accomplish this would be:

```
          or      $t1, $t0, $t9              # $t1 = $t0 | $t9
```

In this case, anywhere there is a one in the mask, we get a one in $t1, and anywhere there is a zero in the mask, we get a copy of $t0 in $t1.

Suppose we want to complement the lower 6 bits of register $t0, but leave all the other bits unchanged. The instruction to accomplish this would be:

```
          xor     $t0, $t0, $t9              # $t1 = $t0 ^ $t9
```

Anywhere there is a one (1) in the mask, we get a complement of the original value in $t0. Anywhere there is a zero (0) in the mask, we get a copy of the corresponding bits in $t0.

Suppose we want to know if the values in two registers $s0 and $s1 have the same sign. The instructions to accomplish this would be:

```
        xor     $t0, $s0, $s1           # The Most Significant Bit (MSB) of $t0 will be
                                        # set to a "1" if $s0 & $s1 have different signs
        bgez    $t0, same_sign          # Branch if MSD of $t0 is a zero
```

Suppose we want to swap the contents of registers $s0 and $s1. The instructions to accomplish this could be:

```
        xor     $t0, $s0, $s1           # Determine which bits are different
        move    $s0, $s1                # Move a copy of s1 into s0
        xor     $s1, $s0, $t0           # Get a copy of s0 but complement those bits
                                        # that were different
```

How would you use logical instructions to determine if the value in a register is an odd number?

5.3 Instructions that Perform Shift Operations

The MIPS architecture has instructions to shift the bits in a register either right or left. The logical shifts always shift in zeros. In the case of the shift right, there is a shift right logical (srl) instruction, and an shift right arithmetic (sra) instruction. In the case of the shift right arithmetic, a copy of the most significant bit is always maintained and shifted right to insure the sign of the number does not change. Logical shifts always shift in zeros.

Suppose $a0 contains the value –32 (11111111111111111111111111100000) and we want to divide $a0 by four (4). Shifting $a0 right arithmetic 2 bits will accomplish the task. The result is –8 (11111111111111111111111111111000). Note that 2^2 is equal to four (4). So in the case of dividing by a value that is some power of two (2), division can be accomplished in one clock cycle with the shift right arithmetic instruction. In the case of odd negative numbers, this is not a truncated division. Instead, it rounds down the next more negative integer number. For example, take the binary value equivalent to –9 and do an arithmetic shift right by one bit and the result will be –5. If truncated division of an odd negative number is required, it can still be accomplished with the following instructions.

```
        sub     $a0, $0, $a0            # Complement the number to make it positive
        sra     $a0, $a0, 1             # Shift Right by 1 is Equivalent to dividing by 2
        sub     $a0, $0, $a0            # Complement the number back to negative
```

Macro instructions are provided to rotate the bits in a register either right or left. (See Appendix D) With these instructions, any bits shifted out of one end of the register will be shifted into the other end of the register. These macro instructions require at least 3 clock cycles to execute.

5.4 Modular Program Design and Documentation

The importance of good program design and documentation can not be stressed too much. Good documentation is valuable for student programmers as well as professional programmers. Over time it is not unusual for the functional requirements of a code module to change; which will require modifications be made to previously written code. Often the programmer assigned the task of modifying a code module is not the same person who created the original version. Good documentation will save programmers hours of time in analyzing how the existing code accomplishes its functionality, and will expedite the necessary modifications to the code. Every organization will have its own documentation standards. The following pages provide a suggested minimal documentation standard. The main components of this documentation standard are:

- Functional Description
- Algorithmic Description
- Register Usage Table
- In Line Documentation

A functional description will provide the information anyone needs to know if they are searching for a function that would be useful is solving some larger programming assignment. The functional description only describes what the function does, not how it is done. The functional description must explain how arguments are passed to the function and how results are returned (if any). The following are example functional descriptions for the classical I/O functions that are described in more detail later in this chapter:

Hexout($a0: value)
A 32-bit binary value is passed to the function in register $a0 and the hexadecimal equivalent is printed out right justified.

Decout($a0: value)
A 32-bit binary value is passed to the function in register $a0 and the decimal equivalent is printed out right justified.

Decin($v0: value, $v1: status)
Reads a string of decimal digits from the keyboard and returns the 32-bit binary equivalent in register $v0. If the string does not correctly represent a decimal number error status value “1” is returned in register $v1 otherwise the status value returned is “0” for a valid decimal number.

Hexin (&string, value):
Scans a string of ASCII characters representing a hexadecimal number and returns the 32-bit binary equivalent value on the stack at Mem($sp+8). A pointer to the string is passed to the function on the stack at Mem($sp+4). Upon return the pointer in (Mem$sp+4) will be pointing to the byte that follows the last hexadecimal digit in the string.

Pseudocode explains how the function is implemented. Anyone assigned the task of modifying the code will be extremely interested in the logical structure of the existing code. The logical structure of the code is more readily understood using a high-level notation. The use of high-level pseudocode is valuable during the initial development of a code module and will be especially helpful to the maintenance programmer. The original programmer usually is not the individual who will be making modifications or improvements to the code as the years pass. Pseudocode facilitates collaboration in group projects. Pseudocode facilitates debugging.

When developing code in a high-level language the use of descriptive variable names is extremely valuable for documentation purposes. In the case of the MIPS architecture, all of the data manipulation instructions and the control instructions require their operands be in the register file. A MIPS assembly language programmer must specify within each instruction which processor registers are going to be utilized. For example, we may have a value in register $t2 corresponding to size, and a value in register $t3 corresponding to count. When using pseudocode to document an assembly language program, we must use the names of the registers we intend to use in the assembly language code. We use register names in the pseudocode so that the translation to assembly language code will be an easy process to perform and because we want documentation that describes how the MIPS architecture actually executes the algorithm. Unless we identify the registers being used, the pseudocode is quite limited in terms of deriving the corresponding assembly language program or documenting the assembly language code.

The use of a cross-reference table that defines what each processor register is being used for within the algorithm will bridge the gap between a descriptive variable name and the corresponding MIPS register. (For example: $t2 = Size, $t3 = Count). Below you will find an example header for a main program that can be used for student programming assignments. Notice the header has a “Register Usage” cross-reference table.

```
################### Example Main Program Header ###################
# Program # 1 : <descriptive name>  Programmer : < your name>
#  Due Date    : mm dd, 2001            Course: CSCI 51a
# Last Modified : mm dd hh:mm       Section:
################################################################
#  Overall Program Functional Description:
#
################################################################
#  Register Usage in Main:
#         s0 = Address of ...
#         s4 = value of ...
################################################################
#  Pseudocode Description:
#
################################################################
```

Below you will find an example header for a function. Each MIPS assembly language function should be immediately proceeded by a header such as the one shown below.

```
#################### Example Function Header  #####################
# Function Name:
# Last Modified : month day hour: minute
##################################################################
#  Functional Description:
#
##################################################################
#  Explain what parameters are passed  to the function, and how:
#         Mem(sp + 4) = Value to be printed.
#
#  Explain what values are returned by the procedure, and how:
#         Mem(sp + 4) = Binary value returned on the stack
#         Mem(sp + 8) = Status value returned on the stack
#                               0 = successful, otherwise error
#
#  Example Calling Sequence :
#
#         <show moves of parameters to registers or the stack>
#         jal  xxxxxx
#         <returns here with . . .
#################################################################
#  Register Usage in Function:
#         t0 = Address of ...
#         t4 = value of ...
#################################################################
#  Algorithmic Description in Pseudocode:
#
#################################################################
```

The use of in-line documentation can be quite helpful to identify what each block of assembly language code is accomplishing. Throughout this book there are a multitude of examples of in-line documentation. The purpose of the in-line documentation is to provide a phrase that describes what is logically being accomplished. For example:

```
                andi    $t8,      $s0, 1  # Extract the Least Significant Bit (LSB)
                beqz    $t8       even    # If LSB is a zero Branch to even
                addi    $s1,      $s1, 1  # Increment the odd count in s1
        even:
```

On the next three pages you will find a complete program consisting of a main program and a function module. Using the documentation, you should be able to follow the logic of what is being accomplished and how its being accomplished.

```
################### Example Main Program Header ###################
# Program # 1 : <descriptive name>  Programmer : < your name>
#  Due Date    : mm dd, 2001            Course: CSCI 51a
# Last Modified : mm dd hh:mm           Section:
###################################################################
#  Overall Program Functional Description:
#  This main line program is used to test the function “sum.”
#  After calling the function, results are printed.
###################################################################
#  Register Usage in Main:
#         $a0:  used to pass the  address of an array to the function
#         $a1: used to pass the length parameter  “N” to the function
###################################################################
#  Pseudocode Description:
#
###################################################################
                .data
array:          .word       -4, 5, 8, -1
msg1:           .asciiz     "\n The sum of the positive values = "
msg2:           .asciiz     "\n The sum of the negative values = "
                .globl       main
                .text
main:
                 li          $v0, 4            # system call code for print_str
                 la          $a0, msg1         # load address of msg1. into $a0
                 syscall                       # print the string

                 la          $a0, array        # Initialize address Parameter
                 li          $a1, 4            # Initialize length Parameter
                 jal         sum               # Call sum

                 move        $a0, $v0          # move value to be printed to $a0
                 li          $v0, 1            # system call code for print_int
                 syscall                       # print sum of Pos:

                 li          $v0, 4            # system call code for print_str
                 la          $a0, msg2         # load address of msg2. into $a0
                 syscall                       # print the string

                 li          $v0, 1            # system call code for print_int
                 move        $a0, $v1          # move value to be printed to $a0
                 syscall                       # print sum of neg

                 li          $v0, 10           # terminate program run and
                 syscall                       # return control to  system
```

```
#################### Example Function Header  #####################
# Function Name: Sum(&X, N, SP, SN)
# Last Modified : month day hour: minute
################################################################
# Functional Description:
# This function will find the sum of the positive and the sum of the negative
# values in an array X of length N.
#
# “X" is the address of an array passed through $a0.
# "N" is the length of the array passed through $a1.
# The function returns two values:
#          (1) Sum of the positive elements in the array passed  through $v0.
#          (2) Sum of the negative elements in the array passed through $v1.
#
################################################################
#  Example Calling Sequence :
#         la        $a0, array
#         li         $a1, 4
#         jal       sum
#         move   $a0, $v0
#
###############################################################
#  Register Usage in Function:
#         a0 = address pointer into the array
#         a1 = loop counter. (counts down to zero)
#         t0 = a value read from the array
#
###############################################################
#  Algorithmic Description in Pseudocode:
#         v0 = 0;
#         v1 = 0;
#         while( a1 > 0 )do
#                  {
#                  a1 = a1 - 1;
#                  t0 = Mem(a0);
#                  a0 = a0 + 4;
#                  If (t0 > 0) then
#                            v0 =v0 + t0;
#                  else
#                            v1 = v1 + t0;
#                  }
#         Return
#
######################################################
```

```
sum:        li      $v0, 0
            li      $v1, 0              # Initialize v0 and v1 to zero
loop:
            blez    $a1, retzz          # If (a1 <= 0) Branch to Return
            addi    $a1, $a1, -1        # Decrement loop count
            lw      $t0, 0($a0)         # Get a value from the array
            addi    $a0, $a0, 4         # Increment array pointer to next word
            bltz    $t0,  negg          # If  value is negative Branch to negg
            add     $v0, $v0, $t0       # Add to the positive sum
            b       loop                # Branch around the next two instructions
negg:
            add     $v1, $v1, $t0       # Add to the negative sum
            b       loop                # Branch to loop
retzz:      jr      $ra                 # Return
```

5.5 A Function to Print Values in Hexadecimal Representation

To display anything on the monitor, the ASCII codes for the desired characters must first be placed into an array of bytes in memory (an output buffer), and then one uses a system call to print the string (syscall 4). The specific set of instructions used to print a string appear below:

```
            li          $v0, 4          # system call code for Print a String
            la          $a0, buffer     # Load address of output buffer into $a0
            syscall
```

The syscall will send a string of characters to the display starting at the memory location symbolically referred to by "buffer." The string must contain a null character (0x00) to indicate where the string ends. It is the programmer's responsibility to place the null character at the proper location in memory to indicate where the string ends.

Notice this particular syscall has two parameters passed to it. The value four (4) passed to the system in register $v0 indicates the programmer wants to invoke a print string system service. The value in register $a0 is the address of the memory location where the first character of the string is located.

Since there is no system service to print values in hexadecimal representation, it would appear this should be one of the first functions we should develop. The logical instructions and the shift instruction come in handy for this algorithm. Recall that a 32-bit binary number can be represented with eight hexadecimal digits. Conceptually, then we need to iterate eight times. Within each iteration, we extract the lower 4 bits from the binary number and then shift the binary number to the right, by 4 bits. We examine the 4 bits that were extracted, and if the value is less than ten, we add the appropriate bias to create the corresponding ASCII code. If the value is ten or greater, then a different bias must be added to obtain the appropriate ASCII code for the correct hexadecimal digit in the range from A through F. Once the ASCII code has been computed, it must be placed into the output buffer, starting at the right most digit position and working to the left.

After the eight ASCII characters have been placed in the buffer, it is necessary to place three additional characters at the beginning of the buffer. Specifically, the ASCII code for a space (0x20), the ASCII code for a zero (0x30), and the ASCII code for an “x” (0x78). Finally the contents of the buffer is printed as an ASCII string, and then a return is executed.

5.6 A Function to Read Values in Hexadecimal Representation

To input characters from the keyboard one uses a system service to read a string (syscall 8) the specific set of instructions used to read a string appear below:

```
        li          $v0, 8          # system call code for Read a String
        la          $a0, buffer     # load address of  input buffer into $a0
        li          $a1, 60         # Length of buffer
        syscall
```

The read string system service will monitor the keyboard and as the keys are pressed, the corresponding ASCII codes will be placed sequentially into the input buffer in memory. When the “**Enter**” (new-line) key is depressed, the corresponding ASCII code (0x0a) is stored in the buffer followed by the null code (0x00), and control is returned to the user program.

Notice this particular syscall has three parameters passed to it: The value eight (8) passed to the system in registered $v0 indicates the programmer wants to invoke a read string system service, the value in registered $a0 specifies the address of input buffer in memory, and the value in register $a1 specifies the length of the buffer. To allocate space in memory for a 60-character buffer, the following assembler directive can be used:

```
        .data
buffer: .space  60
```

Since there is no system service to read in values in hexadecimal representation, this could be a valuable function to develop. In general, this algorithm involves reading in a string of characters from the keyboard into a buffer, and then scanning through the string of characters, converting each ASCII code to the corresponding 4-bit value. The process is essentially the inverse of the hexadecimal output function. When each new valid hexadecimal ASCII code is found, we shift our accumulator register left four bits and then insert the new 4-bit code into the accumulator. If more than 8 hexadecimal digits are found in the input string, the number is invalid, and status information should be returned to indicate an error condition. Any invalid characters in the string, such as “G,” would also be an error condition. A properly specified hexadecimal number should be preceded with the string “**0x**.”

5.7 A Function to Print Decimal Values Right Justified

Here, we will discuss the algorithm to print, right justified, the decimal equivalent of a binary number. The input to this function is a 32-bit binary number. The output will be a string of printed characters on the monitor. When we implement this code we have to determine if the input binary number is a negative value. If the number is negative, the ASCII code for a minus sign must be placed in the output buffer immediately to the left of the most significant decimal digit. The maximum number of decimal digits that will be generated is ten. The output buffer must be at least thirteen characters in length. Specifically, we need a null character at the end of the buffer, possibly ten ASCII codes for ten decimal digits, possibly a minus sign, and at least one leading space character. For a small positive value, such as nine, there will be eleven leading space characters to insure that the number appears right justified within a field of twelve characters.

The heart of this algorithm can be a "do while" control structure. Within each iteration, we divide the number by ten. From the remainder, we derive the next ASCII code for the equivalent decimal representation. The quotient becomes the number for the next iteration. The decimal digits are derived one at a time from the least significant digit working toward the most significant digit on each iteration. While the number is not equal to zero, we continue to iterate. When the quotient finally becomes zero, it is time to check if the number should be preceded by a minus sign, and then all remaining leading character positions are filled with the ASCII code for space. Finally, we use the system service to print the ASCII string in the output buffer.

5.8 A Function to Read Decimal Values and Detect Errors

As was pointed out at the end of Chapter 4, the system service to read an integer has no capability of informing the user if the input string does not properly represent a decimal number. An improved Integer Read function should return status information to the calling program so that the user can be prompted by the calling program to re-enter the value correctly when an input error is detected.

Basically, this new input function will use SYSCALL (8) to read a string of ASCII characters from the keyboard into a buffer, and then return the equivalent 32-bit binary integer value. If the input string can be correctly interpreted as a decimal integer, then a value of zero is returned in register $v1. (The status flag) If the input string cannot be correctly interpreted, then a value of "1" is returned in register $v1. In other words, $v1 will be a flag to indicate if the input value is incorrectly specified.

This algorithm consists of three phases. In the first phase, the string is scanned looking for the first digit of the number with the possibility that the Most Significant Digit (MSD) may be preceded by a minus sign. The second phase involves scanning through the following string of characters, and extracting each decimal digit by subtracting out the ASCII bias. When each new decimal digit is found, we multiply our current accumulated value by ten and add the most recently extracted decimal value to the accumulator. If overflow occurs while performing this arithmetic, then an error has occurred and appropriate status information should be returned. Detection of overflow must be accomplished by this function. An overflow exception must be avoided. Any invalid characters in the string would be an error condition. The second phase ends when a space is found or when a new line character is found. At this time, it would be appropriate to take the two's complement of the accumulated value, if the number has been preceded by a minus sign. In the final phase, we scan to the end of the buffer to verify the only remaining characters in the buffer are spaces.

Exercises

5.1 Write a MIPS assembly language program to find the sum of the first 100 words of data in the memory data segment with the label "chico." Store the resulting sum in the next memory location beyond the end of the array "chico."

5.2 Write a MIPS assembly language program to transfer a block of 100 words starting at memory location "SRC" to another area of memory beginning at memory location "DEST."

5.3 Write a MIPS function called "ABS" that accepts an integer word in register $a0 and returns its absolute value in $a0. Also, show an example code segment that calls the ABS function twice, to test the function for two different input values.

5.4 Write a function PENO (&X, N, SP, SN) that will find the sum of the positive even values and negative odd values in an array X of length N. "X" the address of an array passed through $a0. "N" is the length of the array passed through $a1. The procedure should return two values,
(1) The sum of all the positive even elements in the array passed back through $v0.
(2) The sum of all the negative odd elements in the array passed back through $v1.

5.5 Write a function SUM(N) to find the sum of the integers from 1 to N, making use the multiplication and shifting operations. The value N will be passed to the procedure in $a0 and the result will be returned in the $v0 register.

Write a MIPS assembly language main program that will call the SUM function five times each time passing a different value to the function for N, and then printing the results. The values for N are defined below:

```
N:      .word   9, 10, 32666, 32777, 654321
```

5.6 Write a function FIB(N, &array) to store the first N elements of the Fibonacci sequence into an array in memory. The value N is passed in $a0, and the address of the array is passed in register $a1. The first few numbers of the Fibonacci sequence are: 1, 1, 2, 3, 5, 8, 13, ...

5.7 Write a function that will receive 3 integer words in registers $a0, $a1, & $a2, and will return them in ordered form with the minimum value in $a0 and the maximum value in $a2.

5.8 Write the complete assembly language program, including data declarations, that corresponds to the following C code fragment. Make use of the fact that multiplication and division by powers of 2 can be performed most efficiently by shifting.

```
int main()
{       int  K,  Y ;
        int  Z[50] ;
        Y = 56
        K = 20
        Z[K] = Y  -  16  *  ( K / 4  + 210) ;
}
```

5.9 **<u>MaxMin($a0: &X, $a1: N, $v0: Min, $v1: Max)</u>**
Write a function to search through an array "X" of "N" words to find the minimum and maximum values. The address of the array will be passed to the procedure using register $a0, and the number of words in the array will be passed in register $a1. The minimum and maximum values are returned in registers $v0, and $v1.

5.10 **<u>SumMain($a0: &X, $a1: N, $v0: Sum)</u>**
Write a function to find the sum of the main diagonal elements in a two dimensional N by N array of 32-bit words. The address of the array and the size N are passed to the procedure in registers $a0 and $a1 respectively. The result is returned in $v0. The values in registers $a0 and $a1 should not be modified by this procedure. Calculate the number of clock cycles required to execute your algorithm, assuming N=4.

5.11 **<u>Det($a0: &X, $v0: D)</u>**
Write a function to find the determinant of a two by two matrix (array). The address of the array is passed to the function in registers $a0 and the result is returned in $v0. The value in register $a0 should not be modified by this function. Calculate the number of clock cycles required to execute your algorithm.

5.12 Write a function that accepts a binary value in register $a0 and returns a value in $v0 corresponding to the number of one's in $a0.

5.13 Translate the following pseudocode expression to MIPS assembly language code. Include code to insure that there is no array bounds violation when the store word (sw) instruction is executed. Note that the array “zap” is an array containing 50 words, thus the value in register $a0 must be in the range from 0 to 196. Include code to insure that the value in register $a0 is a word address. If an array bounds violation is detected or the value in register $a0 is not a word address then branch to the label “Error.”

```
        .data
zap:    .space  200
        .text

        …??? . .

        zap[$a0] = $s0;
```

5.14 Write a function to search through an array “X” of “N” words to find how many of the values are evenly divisible by four. The address of the array will be passed to the function using register $a0, and the number of words in the array will be passed in register $a1. Return the results in register $v0.

CHAPTER 6

Function Calls Using the Stack

What do you call the cabs lined up at the Dallas airport?
The yellow rows of taxis.

6.1 Introduction

One of the objectives of this book is to stress the fact that significant program development is a teamwork effort. Unless everyone in a programming team adheres to the same convention for passing arguments (parameters), the programming project will degenerate into chaos. Students using this textbook are expected to use the convention defined below. If everyone uses the same convention, then it should be possible to run a program using randomly select functions written by different students in the class.

6.2 The Stack Segment in Memory

Every program has three segments of memory assigned to it by the operating system when the program is loaded into memory by the linking loader. In general, the programmer has no control over what locations are assigned, and usually the programmer does not care. There is the "text" segment where the machine language code is stored, the "data" segment where space is allocated for global constants and variables, and the stack segment. The stack segment is provided as an area where parameters can be passed, where local variables for functions are allocated space, and where return addresses for nested function calls and recursive functions are stored. Most compiler generated code pass arguments on the stack. Given this stack area in memory, it is possible to write programs with virtually no limit on the number of parameters passed. Without a stack it would be impossible to write recursive procedures or reentrant procedures. The operating system initializes register 29 ($sp) in the register file to the base address of this stack area in memory. The stack grows toward lower addresses.

6.3 Argument Passing Convention

Silicon Graphics has defined the following parameter passing convention. The first four "in" parameters are passed to a function in $a0, $a1, $a2, and $a3. The convention states that space will be allocated on the stack for the first four parameters even though these input values are not stored on the stack by the caller. All additional "in" parameters are passed on the stack. Register $v0 is used to return a value. Very few of the functions we write in this introductory course will involve more than four "in" parameters. Yet students need to gain some experience in passing parameter on the stack. Therefore, in all the remaining examples and exercises students will be expected to pass all arguments on the stack even though this is not the convention as defined by Silicon Graphics.

When a programmer defines a function, the parameter list is declared. As an example, suppose the lead programmer has written the main program and he has assigned his subordinate, Jack, the task of writing a function called "JACK." In the process of writing this code, it becomes clear to Jack that a portion of the task he has been assigned could be accomplished by calling a library function "JILL." Let us suppose that "JILL" has five parameters where three parameters are passed **to** the function (in parameters) and two parameters returned **from** the function (out parameters). Typically, parameters can be values or addresses (pointers). Within the pseudocode description of "JILL" we would expect to find a parameter list defined such as: JILL (A, B, C, D, E)

Here is an example of how a nested function call is accomplished in assembly language: Notice we use the "unsigned" version of the "add immediate" instruction because we are dealing with an address, which is an unsigned binary number. We wouldn't want to generate an exception just because a computed address crossed over the mid-point of the memory space.

```
        addiu  $sp, $sp, -24          # Allocate Space on the Stack
        sw     $t1,  0($sp)           # First In Parameter "A" at Mem[Sp]
        sw     $t2,  4($sp)           # Second In Parameter "B" at Mem[Sp+ 4]
        sw     $t3,  8($sp)           # Third In Parameter "C" at  Mem[Sp+ 8]
        sw     $ra,  20($sp)          # Save Return address
        jal    JILL                   # Call the Function
        lw     $ra, 20($sp)           # Restore Return Address to Main Program
        lw     $t4, 12($sp)           # Get First Out Parameter "D"  at Mem[Sp+12]
        lw     $t5, 16($sp)           # Get Second Out Parameter "E"  at Mem[Sp+16]
        addiu  $sp, $sp, 24           # De-allocate Space on the Stack
```

6.4 Nested Function Calls and Leaf Functions

The scenario described above is an example of a nested function call. When the main program called JACK (jal JACK) the return address back to the main program was saved in $ra. Before JACK calls JILL, this return address must be saved on the stack, and after returning from JILL, the return address to main must be restored to register $ra. The saving and restoring of the return address is only necessary for nested function calls. The first few instructions within the function "JILL" to get the input parameters A, B, and C would be:

```
JILL:
        lw     $a0,  0($sp)           # Get First In Parameter "A" at Mem[Sp]
        lw     $a1,  4($sp)           # Get Second In Parameter "B" at Mem[Sp+4]
        lw     $a2,  8($sp)           # Get Third In Parameter "C" at  Mem[Sp+8]
```

The last few instructions in the function "JILL" to return the two out parameters D and E would be:

```
        sw     $v0, 12($sp)           # First Out Parameter "D"  at Mem[Sp+12]
        sw     $v1, 16($sp)           # Second Out Parameter "E"  at Mem[Sp+16]
        jr     $ra                    # Return to JACK
```

In the case of nested function calls, sometimes there is one more complexity. Let’s suppose in the case of the function JACK it is important the values in registers $t6 and $t7 not be lost as a result of calling JILL. The only way to insure these values will not be lost is to save them on the stack before calling JILL, and to restore them on return from JILL. Why does Jack need to save them? Because he is calling the function JILL and JILL may use the $t6 and $t7 registers to accomplish her task. Note that an efficient programmer will save only those “t” registers that need to be saved. This decision is made by understanding the algorithm being implemented. A leaf function will never have to save “t” registers. The following is an example of how Jack would insure the values in registers $t6 and $t7 are not be lost:

```
        addiu   $sp, $sp, -32           # Allocate More Space on the Stack <####
        sw      $t1,  0($sp)            # First In Parameter “A” at Mem[Sp]
        sw      $t2,  4($sp)            # Second In Parameter “B” at Mem[Sp+ 4]
        sw      $t3,  8($sp)            # Third In Parameter “C” at  Mem[Sp+ 8]
        sw      $ra,  20($sp)           # Save Return address
        sw      $t6,  24($sp)           # Save $t2 on the stack <####
        sw      $t7,  28($sp)           # Save $t3 on the stack <####
        jal     JILL                    # call the Function
        lw      $t6,  24($sp)           # Restore $t2 from the stack <####
        lw      $t7,  28($sp)           # Restore $t3 from the stack <####
        lw      $ra, 20($sp)            # Restore Return Address to Main Program
        lw      $t4, 12($sp)            # Get First Out Parameter “D”  at Mem[Sp+12]
        lw      $t5, 16($sp)            # Get Second Out Parameter “E”  at Mem[Sp+16]
        addiu   $sp, $sp, 32            # De-allocate Space on the Stack <####
```

6.5 Local Variables are Allocated Space on the Stack

As functions become more complex, a situation may arise where space in memory is required to accomplish a task. This could be a temporary data buffer, or a situation where the programmer has run out of registers and needs additional variables on the stack. For example, if Jill needs a temporary array of 16 characters the code to allocate space on the stack for this temporary array would be:

```
        addiu   $sp, $sp, -16   # Allocate Space for a temporary array
        move    $a0, $sp        # Initialize a pointer to the array
```

Before exiting the function, this buffer space must be de-allocated.

6.6 Frame Pointer

In the code immediately above you will notice that allocating space on the stack for local variables requires the address in the stack pointer be changed. In all previous examples we assumed the stack pointer would not change and we could reference the in and out parameters in memory with the same offset that was used by the caller. There is a way to establish a fixed reference point within each function that will maintain the same offset memory references to parameters as was used by the caller. As part of the register usage convention we have a register with the name $fp (register number 30), which stands for frame pointer. A stack frame is also referred to as an activation record. The activation record for any function is that segment in memory containing a function's parameters, saved registers, and local variables. The use of a frame pointer is not a necessity. Some high-level language compilers generate code that use a frame pointer and others don't. None of the exercises in this book are of sufficient complexity to warrant the use of a frame pointer.

Exercises

6.1 **MinMax (&X, N, Min, Max)**

Write a function to search through an array "X" of "N" words to find the minimum and maximum values. The parameters &X and N are passed to the function on the stack, and the minimum and maximum values are returned on the stack. (Show how MinMax is called)

6.2 **Search (&X, N, V, L)**

Write a function to sequentially search an array X of N **bytes** for the relative location L of a value V. The parameters &X, N, and V are passed to the function on the stack, and the relative location L (a number ranging from 1 to N) is returned on the stack. If the value V is not found, the value -1 is returned for L.

6.3 **Scan (&X, N, U, L, D)**

Write a function to scan an array "X" of "N" bytes counting how many bytes are ASCII codes for:

a. upper case letters - U
b. lower case letters - L
c. decimal digits - D

Return the counts on the stack. The address of the array and the number of bytes N will be passed to the function on the stack. Write a short main program to test this function.

6.4 **Hypotenuse (A, B, H)**

This is an exercise in calling nested functions and passing parameters on the stack. Write a function to find the length of the hypotenuse of a right triangle whose sides are of length A and B. Assume that a math library function "sqr(V, R)" is available, which will compute the square root of any positive value V, and return the square root result R. Write a main program to test this function.

6.5 **AVA (&X, &Y, &Z, N, status)**

Write a function to perform an absolute value vector addition. Use the stack to pass parameters. The parameters are the starting address of three different word arrays (vectors) : X, Y, Z, and an integer value N specifying the size of the vectors. If overflow ever occurs when executing this function, an error status of “1” should be returned and the function aborts any further processing. Otherwise, return the value “0” for status. The function will perform the vector addition:

$$X_i = | Y_i | + | Z_i | \text{; with } i \text{ going from 0 to } N - 1.$$

Also write a main program to test this function.

6.6 **Fibonacci (N, E)**

Write a function to return the N th element in the Fibonacci sequence. A value N is passed to the function on the stack, and the N th Fibonacci number E is returned on the stack. If N is greater than 46 overflow will occur, so return a value of 0 if N is greater than 46. Also show an example of calling this function to return the 10th element in the sequence. The first few numbers in the Fibonacci sequence are: 0, 1, 1, 2, 3, 5 . . . .

6.7 **BubSort (&X, N)**

Write a function to sort an array “X” of “N” words into ascending order using the bubble sort algorithm. The address of the array and the value N will be passed to the function on the stack. Show how the sort function is called.

6.8 **RipSort (&X, N)**

Write a function to sort an array “X” of “N” words into ascending order using the ripple sort algorithm. The address of the array and the value N will be passed to the function on the stack.

6.9 **Roots (a, b, c, Status, R1, R2)**

This is an exercise in nested function calls and passing parameters on the stack.
Write a function to calculate the roots of any quadratic equation of the form
$y = ax^2 + bx + c$ where the integer values a, b, and c are passed to the function on the stack. Status should indicate the nature of the results returned as indicated below:

0 : 2 real roots R1 & R2
1 : 1 real root in R1= -a/b
2 : 2 complex roots of the form (R1 $\underline{+}$ i R2)
3 : no roots computed (error)

Assume a math library function “sqrt” is available, that will compute the square root of a positive argument.

CHAPTER 7

Reentrant Functions

Why did the scientist install a knocker on his door?
To win the no bell prize.

7.1 Introduction

It is important that all shared operating system functions and library functions on a multi-tasking computer system be reentrant. One example of a multi-tasking system would be a time-sharing system. A time-sharing system is one where multiple users are sharing the computer's resources. These users are given the illusion that the computer is immediately responsive to every keystroke they make at their keyboards. In reality, every active user is given a slice of time to utilize the computer's resources. This is accomplished by having a real-time clock generator connected to the interrupt system. If the real-time clock produces an interrupt every one hundredth of a second, then 100 completely different tasks could be running on the machine with each task running for one hundredth of a second every second. If this is a computer that executes one million instructions per second, then every task could theoretically be executing nearly ten thousand instructions per second. It may appear that 100 different tasks are running "simultaneously," but in reality each task is running for a fraction of a second every second. When the real-time clock produces an interrupt the current address in the Program Counter (PC) is saved in a special register called EPC and then the Program Counter is loaded with the address of the interrupt processing routine. The first thing this routine does is to save the contents of any registers it will be using to handle the interrupt. Before returning back to this same user these register are restored to their original values. See Chapter 8 for more details.

Let's suppose that fifty users of a time-share system are all using the same compiler or text-editor "simultaneously." Does this mean that fifty separate copies of the text-editor must be loaded into memory, or only one copy? The answer is only one copy, if the developers of the compiler and the text-editor insured that the code was reentrant.

7.2 Rules for Writing Reentrant Code

Reentrant code is pure code. Reentrant code has no **allocated** memory *variables* in the global data segment. It is ok to have *constants* in the global data segment, but it is not ok to have memory locations that are modified by the code in the global data segment. All local *variables* for reentrant code must be dynamically allocated space on the stack. Reentrant code helps to conserve memory space, because memory is dynamically allocated space on the stack when it is needed and de-allocated when it is no longer needed.

Every user of a multi-tasking system is allocated his/her own area in memory for his/her stack. This means that if fifty different users are active, there will be fifty different stacks in different parts of memory. If these fifty different users all “simultaneously” invoke the same function, there will be fifty different areas in memory where variables are allocated space. The important concept is that there would be only one copy of the code shared by all fifty users, and when any particular user is active, the code will be operating on the data in that user’s stack.

7.3 Reentrant I/O Functions

System services to perform I/O should be reentrant. In Chapter Five we discussed the algorithms to perform I/O in decimal and hexadecimal representation. To make these functions reentrant, allocation of space for character buffers must be removed from the global data segment and code must be inserted into the functions to dynamically allocate space on the stack for character buffers. Lets suppose you want to allocate space on the stack for an input buffer of 32 characters, initialize a pointer in $a0 to point to the first character in this buffer, and then read in a string of characters from the keyboard into this buffer. This can be accomplished with the following instructions.

```
        addiu   $sp, $sp, -32           # Allocate space on top of stack
        move    $a0, $sp                # Initialize $a0 as a pointer to the buffer
        li      $a1, 32                 # Specify length of the buffer
        li      $v0, 8                  # System call code for Read String
        syscall
```

7.4 Personal Computers

Most personal computers now have multi-tasking operating systems. For example, a text editor, a spell check program, and voice recognition program could all be running at the same time. The voice recognition program would be analyzing the speech waveforms producing text as input to the word processor, and the interactive spell check monitor would be warning the user of spelling errors it detects. If all of these applications were developed by the same organization, it would have developed a library of functions to perform certain tasks. It would be important all of the code in this library be reentrant code, because multiple applications conceivably could be executing the same library function “simultaneously.”

7.5 Recursive Functions

Writing a recursive function is similar to writing reentrant code, but with one additional complexity. In the case of reentrant code, an interrupt is the event that would create a situation where two users are sharing the same code. When an interrupt occurs, the interrupt handler, which is a system program, will save the values of all the processor registers on the stack when the exception occurred.

In the case of writing a recursive function, it is the responsibility of the programmer to save on the stack the contents of all registers relevant to the current invocation of the function before a recursive call is executed. Upon returning from a recursive function call the values saved on the stack must be restored to the relevant registers.

Exercises

7.1 **Reverse**
Write a reentrant function that will read in a string of characters (60 characters maximum) and will print out the string in reverse. For example the string "July is Hot" will be printed as "toh si yluJ."

7.2 **Palindrome (b)**
Write a reentrant function that will determine if a string is a palindrome. The function should read in a string (16 characters max) placing them in a buffer on the stack. This procedure should call a "Search" function to determine the exact number of actual characters in the string. A Boolean value of true or false (1 or 0) will be returned on the stack to indicate if the string is a palindrome.

7.3 **Factorial**
Write an iterative function to compute N factorial, and then write a recursive function to compute N factorial. Compare the time required to execute the two different versions.

7.4 **Fibonacci (N, E)**
Write a recursive function to return the N^{th} element in the Fibonacci sequence.
Use the stack to pass information to and from the function. A value of 0 should be returned, if overflow occurs when this function executes.
The first few numbers in the sequence are: 0, 1, 1, 2, 3, 5, 8, . . .

7.5 **Determinant (&M, N, R)**
Write a recursive function to find the determinant of a N x N matrix (array). The address of the array M and the size N are passed to the function on the stack, and the result R is returned on the stack:

7.6 Scan (&X, Num)

Write an efficient MIPS assembly language function **Scan (&X, Num)** that will scan through a string of characters with the objective of locating where all the lower case vowels appear in the string, as well as counting how many total lower case vowels appeared in a string. Vowels are the letters a, e, i, o, u.
The address of the string **"X"** is passed to the function on the stack, and the number of vowels found **"NUM"** is returned on the stack. A null character terminates the string. Within this function, you must include code to call any student's "PrintDecimal" function to print, right Justified, the relative position within the string where each vowel was found. Notice this will be a **nested function call**. Here is an example string:

The quick brown fox.

For the above example string the output of your program would be

```
A Vowel was Found at Relative Position :          3
A Vowel was Found at Relative Position :          6
A Vowel was Found at Relative Position :          7
A Vowel was Found at Relative Position :         13
A Vowel was Found at Relative Position :         18
```

Analyze your Scan function. Is it a reentrant function? _____(yes/no)
Explain why.

CHAPTER 8

Exception Processing

How much do pirates pay for ear rings?
A Buccaneer.

8.1 Introduction

Under normal circumstances, anytime the mouse button is pushed or the mouse is moved, or a key on the keyboard is pressed, the computer responds. The natural question to ask is, "How does one program a computer to provide this kind of interactive response from the computer?” The answer is interrupts.

The key to building a computer system that provides superior processing throughput, and also provides interactive response is to include within the hardware design some method for interrupting the currently running program when certain external events occur. The method implemented by the MIPS designers was to include additional hardware referred to as coprocessor 0 that contains a number of specialized registers for exception handling as well as support for memory mapping. This is where the “Translation Lookaside Buffer” (TLB) is located. This coprocessor is designed to send an interrupt signal to the CPU control unit when an exception occurs. Coprocessor 1 is the floating–point unit.

8.2 The Trap Handler

Whenever an exception occurs and the processor has reached the state where the next instruction would be fetched, the CPU controller goes to a special state. In this special state, the **Cause** register is loaded with a number to identify the source of the interrupt. Mode information in the status register is changed and all interrupts are disabled. Also, the address at which the program can be correctly restarted is saved in a register called the Exception Program Counter (**EPC**) and the program counter is loaded with the address in memory where the first instruction of the interrupt response routine is located. The interrupt response routine is simply some MIPS assembly language code that was written by a “systems programmer.”

In the case of the PCSpim simulator this exception processing entry point is memory location 0x8000080. This segment of memory is referred to as kernel segment 0 (kseg0). Students are encouraged to analyze PCSpim's trap handler routine, which is always displayed in the Text Segment. This file "trap.handler" is also available in the PCSpim folder that you downloaded over the internet. A copy of the existing trap handler appears in Appendix E. To gain a more in depth understanding of trap handlers, students could experiment by creating a new version of the trap handler.

The MIPS architecture defines 17 exceptions: 8 external interrupts (6 hardware interrupts and 2 software interrupts) and 9 program exception conditions, which are also referred to as *traps*. One example of a trap is an arithmetic overflow. For PCSpim we only have traps. The part of the machine that contains the Status, Cause, and EPC registers is referred to as coprocessor 0. This coprocessor contains another register called Bad Virtual Address, which is loaded when a Address Error is detected. For example, attempting to read or write to an address where no physical memory exists, or attempting to write to the text segment, which is protected as read only. The instructions used to access the coprocessor registers are shown in the example below:

```
mfc0   $k0,  $13 # CPU register $k0 is loaded with contents of coprocessor register 13
mtc0   $0,   $13  # CPU register $0 is stored in coprocessor register 13
```

The second instruction, **Move to Coprocessor (mtc0)**, can be confusing because the destination register is specified in the right hand field, which is different from all other MIPS instructions. Coprocessor register 8 is the Bad Virtual Address register, 12 is the Status register, 13 is the Cause register, and 14 is the EPC register.

The first task of the interrupt response routine is to execute some code to save the **state** of the machine as it existed when the interrupt occurred. After the interrupt has been processed the machine is placed back in this same **state** and the instruction "Return From Exception" (rfe) is executed. The rfe instruction loads the program Counter (PC) with the contents of the EPC register and restores the Current and Previous Mode in the Status register. Registers $k0 and $k1 are reserved for the operating system. It is possible that the interrupt handler can be written to use only these registers and very little else. Analyzing PCSpim's interrupt handler you will find that the only registers it saves are $v0 and $a0. This interrupt handler is not reentrant because it saves these two registers in specific memory locations. To make the code reentrant these registers would have to be saved on a stack allocated to the operating system.

Real time systems and embedded processors provide much more sophisticated priority interrupt systems, where a lower priority interrupt handler routine can be interrupted by a higher priority interrupt. The MIPS architecture provides interrupt, mask bits within the Status register, which makes it possible to write a priority interrupt handler.

Exercises

8.1 Write your most efficient assembly language code translation for the following function and main line calling program. Note, all communication with the procedure must use a stack frame. Make use of the fact that multiplication and division by powers of 2 can be performed most efficiently by shifting.

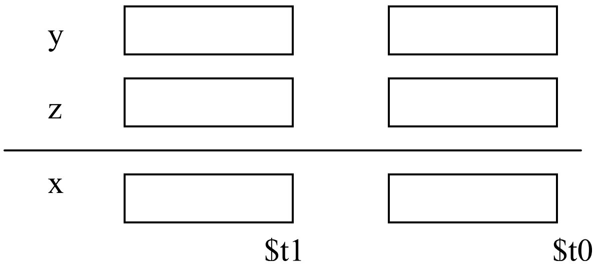
```
void  chico (int *X, int Y,  int Z )
{*X =  Y/ 4  -  Z * 10 + *X * 8 ;}
int  main()
{int J, K, L , M ;
cin >> J, K, L;
chico (& J,  K, L);
M = J  - ( K + L);
cout <<  M;
return 0
}
```

8.2 Write a function **MUL32** (m, n, p, f) that will find the 32-bit product "p" of two arguments m and n. If the two's complement representation of the product cannot be represented with 32 bits, then the error flag "f" should be set to 1 otherwise the error flag is set to 0. Pass all arguments on the stack.

8.3 Write a function **Adduovf** (x, y, s) that will find the 32-bit sum "s" of two unsigned arguments "x" and "y". An exception should be generated if the unsigned representation of the sum results in overflow. Perform all communication on the stack.

8.4 Write a function **Add64** that will perform a 64-bit addition: x = y + z, where the values for x, y, and z are stored as **two** 32-bit words each:
Add64 (x1: $t1, x0: $t0,   y1: $t3,   y0: $t2,   z1: $t5,   z0: $t4)
All six parameters are passed on the stack. If the 64-bit sum results in overflow an exception should be generated.

After writing your code, calculate the performance indexes:
Space:________________(Words of code)
Time: ________________(Maximum number of clock cycles to execute)



8.5 When the MIPS **mult** instruction is executed, a 64-bit product is produced. Many programs doing numerical calculations are designed to process only 32-bit results. For applications written to perform 32-bit precision arithmetic, it is extremely important to detect when the product of two numbers can not be represented with 32 bits, so that some alternative procedures may be invoked.

Write a reentrant library function anyone could use, called **MUL32 (m, n, p).** All function parameters will be passed on the stack. This function should provide the following features:

1. If the product of the two input arguments **m** and **n** cannot be represented with 32 bits (assuming the two's complement number system), then an exception should be generated.

2. This function should also provide the following optimization features:
   (a) Check if **m** or **n** is zero, and return zero without taking 32 clock cycles to execute the **mult** instruction.
   (b) Check if **m** or **n** is plus one (1) or minus one (-1), and return the correct result without taking 32 clock cycles to execute the **mult** instruction.
   (c) Check if **m** or **n** is plus two (2) or minus two (-2), and return the correct result without taking 32 clock cycles to execute the **mult** instruction. If the product cannot be represented with 32-bits, then an exception should be generated.

Provide inline comments and write a paragraph in English to describe all of the conditions that your code tests for to detect overflow.

(HINT—The **XOR** instruction can be useful when writing this function.)

With this function you will demonstrate how to write MIPS assembly language code involving nested function calls. Write a function to perform a vector product. **vectorprod (&X, &Y, &Z, N, status).** Vectorprod will call the **MUL32** function. Use the stack to pass arguments. The in parameters are the starting address of three different word arrays (vectors) : X, Y, Z, and an integer value N specifying the size of the vectors. Status is an out parameter to indicate if overflow ever occurred while executing this function. The procedure will perform the vector product:

$X_i = Y_i * Z_i$; with i going from 0 to N - 1

Write a MIPS assembly language **main** program that could be used to test the **vectorprod** function.

CHAPTER 9

A Pipelined Implementation

What's another word for thesaurus?

9.1 Introduction

In all the previous chapters of this book we developed assembly language code for a simplified model of the MIPS architecture. A technique call pipelining is used in all actual implementations of the MIPS architecture to build a processor that will run almost five times faster than the simplified model introduced in Chapter 1. Essentially this speed up is accomplished by utilizing a concept analogous to Henry Ford's method of building cars on an assembly line. Using this concept the CPU chip is designed to process each instruction by passing information through a series of functional units. The following explanation describes a five stage pipeline implementation. Within the processor there are five different stages of hardware to process an instruction. With a five stage pipelined implementation, we will have five different instructions at different stages of execution moving through the pipeline. It takes five clock cycles for any instruction to work its way through the pipeline, but since there are five instructions being executed simultaneously, the average execution rate is one instruction per clock cycle. The function performed in each of these five stages is:

1. Fetch the instruction from cache memory and load it into the Instruction register (IR). Increment the Program Counter (PC) by four.

2. Fetch values Rs and Rt from the Register File. If it is a branch instruction and the branch condition is met, then load the PC with the branch target address.

3. Perform an arithmetic or logic function in the ALU. This is the stage where an addition is performed to calculate the effective address for a load or store instruction.

4. If the instruction is a load, read from the data cache. If the instruction is a store, write to the data to cache. Otherwise, pass the data in the Result register on to the Write Back register.

5. Store the value in the Write Back register to the Register File. Notice that when the first instruction into the pipeline is storing results back to the Register File the forth instruction into the pipeline is simultaneously reading from the register file.

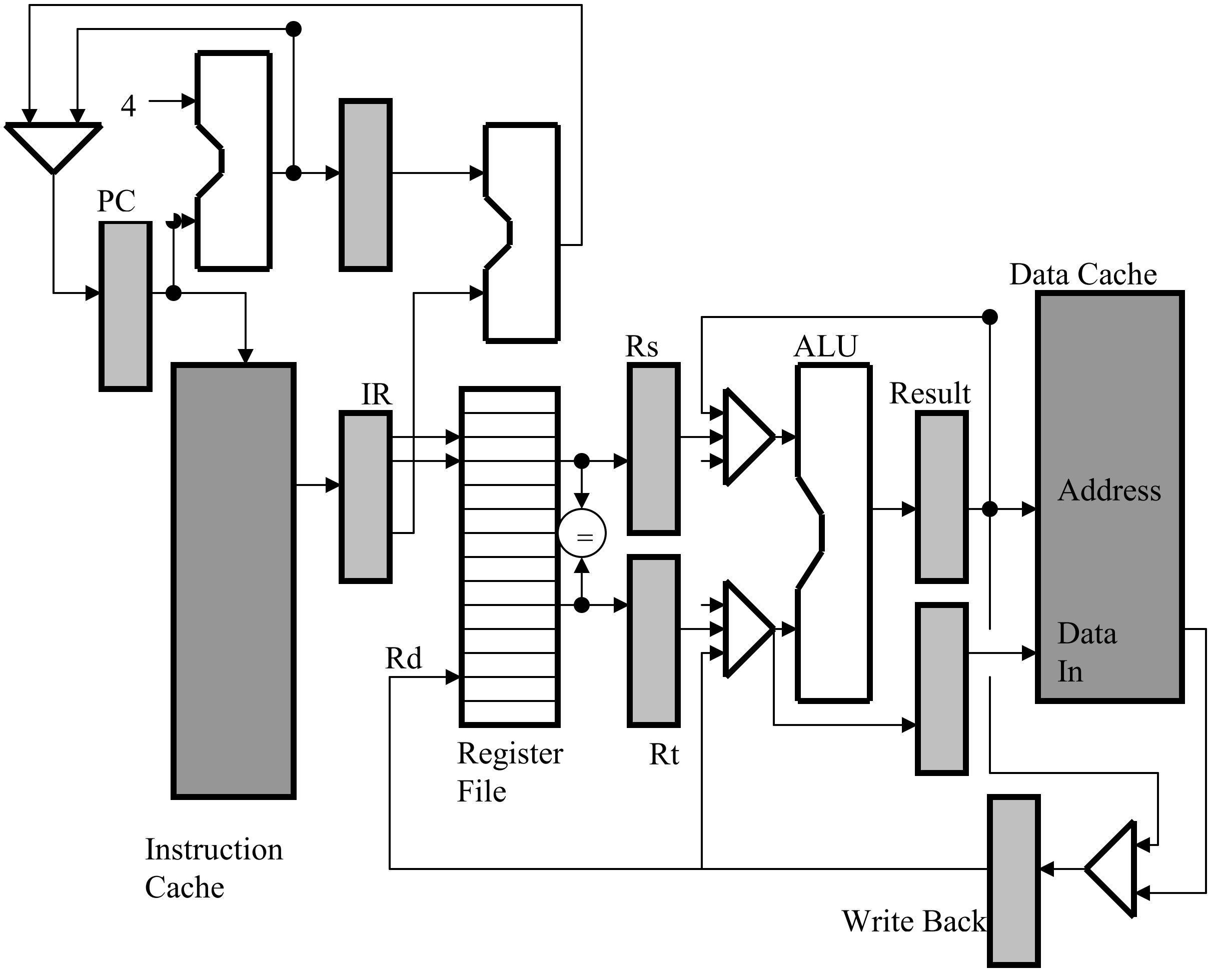


Figure 9.1 Pipelined Datapath Diagram

9.2 A Pipelined Datapath

Figure 9.1 shows a five stage datapath diagram with all the major components identified. Understand that during each clock cycle all five pipeline stages are performing their function and passing their results on to the next pipeline stage, via a buffer register. The Instruction Register (IR) is the buffer register that holds the results of phase one. Registers Rs and Rt hold the results of phase two. The Result register holds the result of the ALU operation. The Write Back register holds the data read from memory in the case of a load instruction, and for all other instructions it is a buffer register holding the value that was in the Result register at the end of the previous clock cycle.

A pipelined processor creates new challenges for the assembly language programmer. Specifically these challenges are referred to as data hazards and control hazards. The term data hazard refers to the following situation. Suppose we have three sequential instructions x, y, and z that come into the pipeline. Suppose that x is the first instruction into the pipeline followed by y and z. If the results computed by instruction x are need by y or z then we have a data hazard. The hardware solution to this problem is to include forwarding paths in the machine's datapath so that even thought the results have not yet been written back to the register file, the needed information is forwarded from the Result register or the Write Back register to the input of the ALU. Some of these forwarding paths are shown in figure 9.1. There is one type of data hazard that cannot be solved with forwarding hardware. This is a situation where a load instruction is immediately followed by an instruction that would use the value fetched from memory. The only solution to this problem is to rearrange the assembly language code so that the instruction following the load is not an instruction that uses the value being fetched from memory. In recognition of this situation we refer to load instructions on pipelined processors as being "delayed loads." If the existing instructions in the algorithm cannot be rearranged to solve the data hazard then a "no operation" (nop) instruction is placed in memory immediately following the load instruction.

Essentially there is no way to eliminate the control hazards so the solution is to recognize that a branch or jump instruction will take effect after the instruction following the branch or jump is in the pipeline. This means that a programmer must recognize this fact and organize the code to take this into account. Typically programmers will find some instruction in their code that the branch is not dependant on that proceeded the branch and place it in the delay slot following the branch. If the existing instructions in the algorithm cannot be rearranged to solve the control hazard then a "no operation" (nop) instruction is placed in memory immediately following the branch instruction.

9.3 PCSpim Option to Simulate a Pipelined Implementation

The latest version of PCSpim provides an option to run the simulator as if it were a pipelined implementation. To invoke this option go to settings under the simulation pull down menu and click the "Delayed Branches" and "Delayed Load" boxes. Using this option you can gain experience in writing assembly language code that will run on a pipelined implementation of the MIPS architecture.

Compilers are used to translate high-level code to assembly language code. The final phase of this translation is called code generation. Whoever writes the program to do code generation has to be aware of these special requirements associated with a pipelined implementation.

Exercises

9.1 Taking into consideration delayed branches and delayed loads, write a MIPS function to search through an array "X" of "N" words to find how many of the values in the array are evenly divisible by four. The address of the array will be passed to the function using register $a0, and the number of words in the array will be passed in register $a1. Return results in register $v0.

9.2 Taking into consideration delayed branches and delayed loads, write a MIPS function to sort an array “X” of “N” words into ascending order using the bubble sort algorithm. The address of the array and the value N will be passed to the function on the stack. Show how the sort function is called.

9.3 Taking into consideration delayed branches and delayed loads, write a MIPS function **Adduovf** (x, y, s) that will find the 32-bit sum “s” of two unsigned arguments “x” and “y”. An exception should be generated if the unsigned representation of the sum results in overflow. Perform all communication on the stack.

9.4 Taking into consideration delayed branches and delayed loads, write a MIPS, function to return the N^{th} element in the Fibonacci sequence. A value N is passed to the function on the stack, and the N^{th} Fibonacci number E is returned on the stack. If N is greater than 46 overflow will occur, so return a value of 0 if N is greater than 46. Also show an example of calling this function to return the 10th element in the sequence. The first few numbers in the Fibonacci sequence are: 0, 1, 1, 2, 3, 5 . . . .

CHAPTER 10

Embedded Processors

What did the hotdog say when he crossed the finish line?
I'm the wiener!

10.1 Introduction

Embedded processors are ubiquitous. They are found in toys, games, cameras, appliances, cars, VCRs, televisions, printers, fax machines, modems, disk drives, instrumentation, climate control systems, cellular telephones, routers, and aerospace applications, just to name a few. It is estimated in the year 2000, over $15 billion worth of embedded processors were shipped. If you go to the following web-site you can read in more detail exactly how the MIPS processor is being used in embedded systems:

www.mips.com/

10.2 Code Development for Embedded Processors

An embedded system usually lacks secondary storage (e.g. a hard disk). Typically all of the code is stored in Read Only Memory (ROM). Usually, most of the code written for embedded processors is first written in a high-level languages such as C. Programmers who can visualize how the high-level code will be translated into assembly language code will most likely develop the "best" code. Then programmers who have an intimate understanding of the assembly language for the target processor, will analyze the code generated by the compiler looking for ways to make further optimizations. In other words, they look for ways to speed up the execution, or to reduce the amount of code that has to be stored in ROM. Typically, for real-time applications, the code must be fine-tuned, to meet the system's performance requirements. Any programmer with the skills to accomplish this kind of optimization will be highly sought after. The kernel of the operating system deals with responding to interrupts and scheduling tasks. This code as well as the I/O drivers will typically be the first code to be scrutinized.

The Motorola 68000 and its derivatives currently have the largest share of the embedded market. While the MIPS processor is classified as a RISC processor, the Motorola 68000 is classified as a Complex Instruction Set Computer (CISC). With a solid understanding of the MIPS processor and experience in developing assembly language code for the MIPS processor, it is a relatively easy task to make the transition to assembly language for other processors.

10.3 Memory Mapped I/O

MIPS processors communicate with the outside word using memory mapped Input Output (I/O). There are registers within the I/O devices. Unique address decode logic is associated with each of these registers. When the MIPS processor reads or writes to one of these address the processor is actually reading from or writing to a register in one of the I/O devices. Using this straight forward technique an embedded processor can acquire information about the state of whatever is being controlled and can send out signals to change the state.

10.4 References

To write code for embedded systems you will need to know much more than has been provided by this introductory book. In this book we have not discussed how to initialize and control the cache.

Currently there are two very good books available specifically targeted for MIPS embedded systems programming. "See MIPS Run" by Dominic Sweetman (ISBN 1-55860-410-3) and "The MIPS Programmers Handbook" by Farquhar and Bunce (ISBN 1-55860-297-6). These books describe how to initialization and manage cache memory, and how to implement a priority interrupt system. These books also provide a detailed explanation of the coprocessor 1, which implements the floating point instructions.

The company Algorithmics Ltd., is probably the most experienced MIPS technology support group anywhere. It was founded in 1988 by a group of MIPS veterans. Their web-site is: http://www.algor.co.uk/

APPENDIX A

Quick Reference

Integer Instruction Set

Name	Syntax		Space/Time
Add:	add	Rd, Rs, Rt	1/1
Add Immediate:	addi	Rt, Rs, Imm	1/1
Add Immediate Unsigned:	addiu	Rt, Rs, Imm	1/1
Add Unsigned:	addu	Rd, Rs, Rt	1/1
And:	and	Rd, Rs, Rt	1/1
And Immediate:	andi	Rt, Rs, Imm	1/1
Branch if Equal:	beq	Rs, Rt, Label	1/1
Branch if Greater Than or Equal to Zero:	bgez	Rs, Label	1/1
Branch if Greater Than or Equal to Zero and Link:	bgezal	Rs, Label	1/1
Branch if Greater Than Zero:	bgtz	Rs, Label	1/1
Branch if Less Than or Equal to Zero:	blez	Rs, Label	1/1
Branch if Less Than Zero and Link:	bltzal	Rs, Label	1/1
Branch if Less Than Zero:	bltz	Rs, Label	1/1
Branch if Not Equal:	bne	Rs, Rt, Label	1/1
Divide:	div	Rs, Rt	1/38
Divide Unsigned:	divu	Rs, Rt	1/38
Jump:	j	Label	1/1
Jump and Link:	jal	Label	1/1
Jump and Link Register:	jalr	Rd, Rs	1/1
Jump Register:	jr	Rs	1/1
Load Byte:	lb	Rt, offset(Rs)	1/1
Load Byte Unsigned:	lbu	Rt, offset(Rs)	1/1
Load Halfword:	lh	Rt, offset(Rs)	1/1
Load Halfword Unsigned:	lhu	Rt, offset(Rs)	1/1
Load Upper Immediate:	lui	Rt, Imm	1/1
Load Word:	lw	Rt, offset(Rs)	1/1
Load Word Left:	lwl	Rt, offset(Rs)	1/1
Load Word Right:	lwr	Rt, offset(Rs)	1/1
Move From High:	mfhi	Rd	1/1
Move From Low:	mflo	Rd	1/1
Move to High:	mthi	Rs	1/1
Move to Low:	mtlo	Rs	1/1
Multiply:	mult	Rs, Rt	1/32
Multiply Unsigned:	multu	Rs, Rt	1/32
NOR:	nor	Rd, Rs, Rt	1/1
OR:	or	Rd, Rs, Rt	1/1
OR Immediate:	ori	Rt, Rs, Imm	1/1

Store Byte:	sb	Rt, offset(Rs)	1/1
Store Halfword:	sh	Rt, offset(Rs)	1/1
Shift Left Logical:	sll	Rd, Rt, sa	1/1
Shift Left Logical Variable:	sllv	Rd, Rt, Rs	1/1
Set on Less Than:	slt	Rd, Rt, Rs	1/1
Set on Less Than Immediate:	slti	Rt, Rs, Imm	1/1
Set on Less Than Immediate Unsigned:	sltiu	Rt, Rs, Imm	1/1
Set on Less Than Unsigned:	sltu	Rd, Rt, Rs	1/1
Shift Right Arithmetic:	sra	Rd, Rt, sa	1/1
Shift Right Arithmetic Variable:	srav	Rd, Rt, Rs	1/1
Shift Right Logical:	srl	Rd, Rt, sa	1/1
Shift Right Logical Variable:	srlv	Rd, Rt, Rs	1/1
Subtract:	sub	Rd, Rs, Rt	1/1
Subtract Unsigned:	subu	Rd, Rs, Rt	1/1
Store Word:	sw	Rt, offset(Rs)	1/1
Store Word Left:	swl	Rt, offset(Rs)	1/1
Store Right:	swr	Rt, offset(Rs)	1/1
System Call:	syscall		1/1
Exclusive OR:	xor	Rd, Rs, Rt	1/1
Exclusive OR Immediate:	xori	Rt, Rs, Imm	1/1

Macro instructions

Name	Syntax		Space/Time
Absolute Value:	abs	Rd, Rs	3/3
Branch if Equal to Zero:	beqz	Rs, Label	1/1
Branch if Greater Than or Equal :	bge	Rs, Rt, Label	2/2
Branch if Greater Than or Equal Unsigned:	bgeu	Rs, Rt, Label	2/2
Branch if Greater Than:	bgt	Rs, Rt, Label	2/2
Branch if Greater Than Unsigned:	bgtu	Rs, Rt, Label	2/2
Branch if Less Than or Equal:	ble	Rs, Rt, Label	2/2
Branch if Less Than or Equal Unsigned:	bleu	Rs, Rt, Label	2/2
Branch if Less Than:	blt	Rs, Rt, Label	2/2
Branch if Less Than Unsigned:	bltu	Rs, Rt, Label	2/2
Branch if Not Equal to Zero:	bnez	Rs, Label	1/1
Branch Unconditional:	b	Label	1/1
Divide:	div	Rd, Rs, Rt	4/41
Divide Unsigned:	divu	Rd, Rs, Rt	4/41
Load Address:	la	Rd, Label	2/2
Load Immediate:	li	Rd, value	2/2
Move:	move	Rd, Rs	1/1
Multiply:	mul	Rd, Rs, Rt	2/33
Multiply (with overflow exception):	mulo	Rd, Rs, Rt	7/37
Multiply Unsigned (with overflow exception):	mulou	Rd, Rs, Rt	5/35
Negate:	neg	Rd, Rs	1/1
Negate Unsigned:	negu	Rd, Rs	1/1
Nop:	nop		1/1

Not:	not	Rd, Rs	1/1
Remainder Unsigned:	remu	Rd, Rs, Rt	4/40
Rotate Left Variable:	rol	Rd, Rs, Rt	4/4
Rotate Right Variable:	ror	Rd, Rs, Rt	4/4
Remainder:	rem	Rd, Rs, Rt	4/40
Rotate Left Constant:	rol	Rd, Rs, sa	3/3
Rotate Right Constant:	ror	Rd, Rs, sa	3/3
Set if Equal:	seq	Rd, Rs, Rt	4/4
Set if Greater Than or Equal:	sge	Rd, Rs, Rt	4/4
Set if Greater Than or Equal Unsigned:	sgeu	Rd, Rs, Rt	4/4
Set if Greater Than:	sgt	Rd, Rs, Rt	1/1
Set if Greater Than Unsigned:	sgtu	Rd, Rs, Rt	1/1
Set if Less Than or Equal:	sle	Rd, Rs, Rt	4/4
Set if Less Than or Equal Unsigned:	sleu	Rd, Rs, Rt	4/4
Set if Not Equal:	sne	Rd, Rs, Rt	4/4
Unaligned Load Halfword Unsigned:	ulh	Rd, n(Rs)	4/4
Unaligned Load Halfword:	ulhu	Rd, n(Rs)	4/4
Unaligned Load Word:	ulw	Rd, n(Rs)	2/2
Unaligned Store Halfword:	ush	Rd, n(Rs)	3/3
Unaligned Store Word:	usw	Rd, n(Rs)	2/2

System I/0 Services

Service	Code in $v0	Arguments	Results
Print an Integer	**1**	$a0 = Integer Value to be Printed	
Print Float	2		
Print Double	3		
Print a String	**4**	$a0 = Address of String in Memory	
Read an Integer	**5**		Integer Returned in $v0
Read Float	6		
Read Double	7		
Read a String	**8**	$a0 = Address of Input Buffer in Memory $a1 = Length of Buffer (n)	
Sbrk	9	$a0 = amount	Address in $v0
Exit	**10**		

The system call Read Integer reads an entire line of input from the keyboard up to and including the newline. Characters following the last digit in the decimal number are ignored. Read String has the same semantics as the Unix library routine fgets. It reads up to n – 1 characters into a buffer and terminates the string with a null byte. If fewer than n – 1 characters are on the current line, Read String reads up to and including the newline and again null-terminates the string. Print String will display on the terminal the string of characters found in memory starting with the location pointed to by the address stored in $a0. Printing will stop when a null character is located in the string. Sbrk returns a pointer to a block of memory containing n additional bytes. Exit terminates the user program execution and returns control to the operating system.

Assembler Directives

Morgan Kaufmann Publishers has generously provided an on-line version of Appendix A from “Computer Organization and Design: The Hardware/Software Interface” (as a Adobe PDF file). This is a more complete and up-to-date version of SPIM documentation than the one included with SPIM. Every student should down-load this file, which is available at:
http://www.cs.wisc.edu/~larus/SPIM/cod-appa.pdf
An exhaustive list of assembler directives may be found starting on page A-51 of this document.

The following is a quick reference to the most commonly used assembler directives, which was extracted from the above source.

.align n Align the next datum on a 2^n byte boundary. For example, .align 2 aligns the next value on a word boundary. .align 0 turns off automatic alignment of .half, .word, .float, and .double directives until the next .data or .kdata directive.
.ascii str Store the string *str* in memory, but do not null-terminate it.
.asciiz str Store the string *str* in memory and null-terminate it.
.byte b1,..., bn Store the *n* values in successive bytes of memory.
.data <addr> Subsequent items are stored in the data segment. If the optional argument *addr* is present, subsequent items are stored starting at address *add*r.
.globl sym Declare that label *sym* is global and can be referenced from other files.
.space n Allocate *n* bytes of space in the current segment (which must be the data segment in SPIM).
.text <addr> Subsequent items are put in the user text segment. In SPIM, these items may only be instructions or words (see the .word directive below). If the optional argument *addr* is present, subsequent items are stored starting at address *add*r.
.word w1,..., wn Store the *n* 32-bit quantities in successive memory words.

Strings are enclosed in double quotes (”). Special characters in strings follow the C convention:

- newline \n
- tab \t
- quote \"

The ASCII code “back space” is not supported by the SPIM simulator.

Numbers are base 10 by default. If they are preceded by *0x,* they are interpreted as hexadecimal. Hence, 256 and 0x100 denote the same value.

APPENDIX B

ASCII Codes

dec	hex	Char	dec	hex	Char	dec	hex	Char	dec	hex	Char
0	00	null	32	20	sp	64	40	@	96	60	'
1	01	soh	33	21	!	65	41	A	97	61	a
2	02	stx	34	22	"	66	42	B	98	62	b
3	03	etx	35	23	#	67	43	C	99	63	c
4	04	eot	36	24	$	68	44	D	100	64	d
5	05	enq	37	25	%	69	45	E	101	65	e
6	06	ack	38	26	&	70	46	F	102	66	f
7	07	bel	39	27	'	71	47	G	103	67	g
8	08	bs	40	28	(	72	48	H	104	68	h
9	09	ht	41	29	)	73	49	I	105	69	i
10	0a	nl	42	2a	*	74	4a	J	106	6a	j
11	0b	vt	43	2b	+	75	4b	K	107	6b	k
12	0c	np	44	2c	,	76	4c	L	108	6c	l
13	0d	cr	45	2d	-	77	4d	M	109	6d	m
14	0e	so	46	2e	.	78	4e	N	110	6e	n
15	0f	si	47	2f	/	79	4f	O	111	6f	o
16	10	dle	48	30	0	80	50	P	112	70	p
17	11	dc1	49	31	1	81	51	Q	113	71	q
18	12	dc2	50	32	2	82	52	R	114	72	r
19	13	dc3	51	33	3	83	53	S	115	73	s
20	14	dc4	52	34	4	84	54	T	116	74	t
21	15	nak	53	35	5	85	55	U	117	75	u
22	16	syn	54	36	6	86	56	V	118	76	v
23	17	etb	55	37	7	87	57	W	119	77	w
24	18	can	56	38	8	88	58	X	120	78	x
25	19	em	57	39	9	89	59	Y	121	79	y
26	1a	sub	58	3a	:	90	5a	Z	122	7a	z
27	1b	esc	59	3b	;	91	5b	[	123	7b	{
28	1c	fs	60	3c	<	92	5c	\	124	7c	|
29	1d	gs	61	3d	=	93	5d	]	125	7d	}
30	1e	rs	62	3e	>	94	5e	^	126	7e	~
31	1f	us	63	3f	?	95	5f	_	127	7f	del

APPENDIX C

Integer Instruction Set

Add:

add Rd, Rs, Rt **# RF[Rd] = RF[Rs] + RF[Rt]**

Op-Code	Rs	Rt	Rd		Function Code
000000	sssss	ttttt	ddddd	00000	100000

Add contents of Reg.File[Rs] to Reg.File[Rt] and store result in Reg.File[Rd].
If overflow occurs in the two's complement number system, an exception is generated.

Add Immediate:

addi Rt, Rs, Imm **# RF[Rt] = RF[Rs] + Imm**

Op-Code	Rs	Rt	Imm		
001000	sssss	ttttt	iiiii	iiiii	iiiiii

Add contents of Reg.File[Rs] to sign extended Imm value, store result in Reg.File [Rt].
If overflow occurs in the two's complement number system, an exception is generated.

Add Immediate Unsigned:

addiu Rt, Rs, Imm **# RF[Rt] = RF[Rs] + Imm**

Op-Code	Rs	Rt	Imm		
001001	sssss	ttttt	iiiii	iiiii	iiiiii

Add contents of Reg.File[Rs] to sign extended Imm value, store result in Reg.File[Rt].
No overflow exception is generated.

Add Unsigned:

addu Rd, Rs, Rt **# RF[Rd] = RF[Rs] + RF[Rt]**

Op-Code	Rs	Rt	Rd		Function Code
000000	sssss	ttttt	ddddd	00000	100001

Add contents of Reg.File[Rs] to Reg.File[Rt] and store result in Reg.File [Rd].
No overflow exception is generated.

And:

and Rd, Rs, Rt **# RF[Rd] = RF[Rs] AND RF[Rt]**

Op-Code	Rs	Rt	Rd		Function Code
000000	sssss	ttttt	ddddd	00000	100100

Bitwise logically AND contents of Register File[Rs] with Reg.File[Rt] and store result in Reg.File[Rd].

And Immediate:

andi Rt, Rs, Imm **# RF[Rt] = RF[Rs] AND Imm**

Op-Code	Rs	Rt	Imm
001100	sssss	ttttt	iiiiiiiiiiiiiiii

Bitwise logically AND contents of Reg.File[Rs] wih zero-extended Imm value and store result in Reg.File[Rt].

Branch Instructions

The immediate field contains a signed 16-bit value specifying the number of words away from the current program counter address to the location symbolically specified by the label. Since MIPS uses byte addressing, this word offset value in the immediate field is shifted left by two bits and added to the current contents of the program counter when a branch is taken. The SPIM assembler generates the offset from the address of the branch instruction. Whereas the assembler for an actual MIPS processor will generate the offset from the address of the instruction following the branch instruction since the program counter will have already been incremented by the time the branch instruction is executed.

Branch if Equal:

Beq Rs, Rt, Label **# If (RF[Rs] == RF[Rt])then PC = PC + Imm<< 2**

Op-Code	Rs	Rt	Imm
000100	sssss	ttttt	iiiiiiiiiiiiiiii

If Reg.File[Rs] is equal to Reg.File[Rt] then branch to label.

Branch if Greater Than or Equal to Zero:

bgez Rs, Label **# If (RF[Rs] >= RF[0]) then PC = PC + Imm<< 2**

Op-Code	Rs	code	Imm
000001	sssss	00001	iiiiiiiiiiiiiiii

If Reg.File[Rs] is greater than or equal to zero, then branch to label.

Branch if Greater Than or Equal to Zero and Link:

bgezal Rs, Label **# If(RF[Rs] >= RF[0])then**
{RF[$ra] = PC;
PC = PC + Imm<< 2 }

Op-Code	Rs	code	Imm
000001	sssss	10001	iiiiiiiiiiiiiiii

If Reg.File[Rs] is greater than or equal to zero, then save the return address in Reg.File[$rs] and branch to label. (Used to make conditional function calls)

Branch if Greater Than Zero:

bgtz Rs, Label **# If (RF[Rs] > RF[0]) then PC = PC + Imm<< 2**

Op-Code	Rs	Rt	Imm
000111	sssss	00000	iiiiiiiiiiiiiiii

If Reg.File[Rs] is greater than zero, then branch to label.

Branch if Less Than or Equal to Zero:

blez Rs, Label **# If (RF[Rs] <= RF[0]) then PC = PC + Imm<< 2**

Op-Code	Rs	Rt	Imm
000110	sssss	00000	iiiiiiiiiiiiiiii

If Reg.File[Rs] is less than or equal to zero, then branch to label.

Branch if Less Than Zero and Link:

bltzal Rs, Label **# If RF[Rs] < RF[0] then**
{RF[$ra] = PC;
PC = PC + Imm<< 2 }

```
Op-Code  Rs    code   Imm
000001sssss10000iiiiiiiiiiiiiiii
```

If Reg.File[Rs] is less than zero then save the return address in Reg.File[$rs] and branch to label.

Branch if Less Than Zero:

bltz Rs, Label **# If RF[Rs] < RF[0] then PC = PC + Imm<< 2**

```
Op-Code  Rs    code   Imm
000001sssss00000iiiiiiiiiiiiiiii
```

If Reg.File[Rs] is less than zero then branch to label.

Branch if Not Equal:

bne Rs, Rt, Label **# If RF[Rs] != RF[Rt] then PC = PC + Imm<< 2**

```
Op-Code  Rs    Rt     Imm
000101ssssstttttiiiiiiiiiiiiiiii
```

If Reg.File[Rs] is not equal to Reg.File[Rt] then branch to label.

Divide:

div Rs, Rt **# Low = Quotient (RF[Rs] / RF[Rt])**
# High = Remainder (RF[Rs] / RF[Rt])

```
Op-Code  Rs    Rt                Function Code
000000sssssttttt0000000000011010
```

Divide the contents of Reg.File[Rs] by Reg.File[Rt]. Store the quotient in the LOW register, and store the remainder in the HIGH register. The sign of the quotient will be negative if the operands are of opposite signs. The sign of the remainder will be the same as the sign of the numerator, Reg.File[Rs]. No overflow exception occurs under any circumstances. It is the programmer's responsibility to test if the divisor is zero before executing this instruction, because the results are undefined when the divisor is zero. For some implementations of the MIPS architecture, it takes 38 clock cycles to execute the divide instruction.

Divide Unsigned:

divu Rs, Rt **# Low = Quotient (RF[Rs] / RF[Rt])**
# High = Remainder (RF[Rs] / RF[Rt])

```
Op-Code  Rs     Rt                   *Function Code
000000 sssss ttttt 00000 00000 011011
```

Divide the contents of Reg.File[Rs] by Reg.File[Rt], treating both operands as unsigned values. Store the quotient in the LOW register, and store the remainder in the HIGH register. The quotient and remainder will always be positive values. No overflow exception occurs under any circumstances. It is the programmer's responsibility to test if the divisor is zero before executing this instruction, because the results are undefined when the divisor is zero. For some implementations of the MIPS architecture, it takes 38 clock cycles to execute the divide instruction.

Jump:

j Label **# PC = PC(31:28) | Imm<< 2**

```
Op-Code           Imm
000010 iiiiiiiiiiiiiiiiiiiiiiiiii
```

Load the PC with an address formed by concatenating the first 4-bits of the current PC with the value in the 26-bit immediate field shifted left 2-bits.

Jump and Link: (Use this instructions to make function calls.

jal Label **# RF[$ra] = PC; PC = PC(31:28) | Imm<< 2**

```
Op-Code           Imm
000010 iiiiiiiiiiiiiiiiiiiiiiiiii
```

Save the current value of the Program Counter (PC) in Reg.File[$ra], and load the PC with an address formed by concatenating the first 4-bits of the current PC with the value in the 26-bit immediate field shifted left 2-bits.

Jump and Link Register: (Use this instructions to make function calls.

jalr Rd, Rs **# RF[Rd] = PC; PC = RF[Rs]**

Op-Code	Rs		Rd		*Function Code
000000	sssss	00000	ddddd	00000	001001

Save the current value of the Program Counter (PC) in Reg.File[Rd] and load the PC with the address that is in Reg.File[Rs]. A programmer must insure a valid address has been loaded into Reg.File[Rs] before executing this instruction.

Jump Register: (Use this instructions to return from a function call.)

jr Rs **# PC = RF[Rs]**

Op-Code	Rs				*Function Code
000000	sssss	00000	00000	00000	001000

Load the PC with an the address that is in Reg.File[Rs].

Load Byte:

lb Rt, offset(Rs) **# RF[Rt] = Mem[RF[Rs] + Offset]**

Op-Code	Rs	Rt	Offset
100000	sssss	ttttt	iiiiiiiiiiiiiiii

The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. An 8-bit byte is read from memory at the effective address, sign extended and loaded into Reg.File[Rt].

Load Byte Unsigned:

lbu Rt, offset(Rs) **# RF[Rt] = Mem[RF[Rs] + Offset]**

Op-Code	Rs	Rt	Offset
100100	sssss	ttttt	iiiiiiiiiiiiiiii

The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. An 8-bit byte is read from memory at the effective address, zero extended and loaded into Reg.File[Rt].

Load Halfword:

lh Rt, offset(Rs) **# RF[Rt] = Mem[RF[Rs] + Offset]**

Op-Code	Rs	Rt	Offset
100001	sssss	ttttt	iiiiiiiiiiiiiiii

The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. A 16-bit half word is read from memory at the effective address, sign extended and loaded into Reg.File[Rt]. If the effective address is an odd number, an address error exception occurs.

Load Halfword Unsigned:

lhu Rt, offset(Rs) **# RF[Rt] = Mem[RF[Rs] + Offset]**

Op-Code	Rs	Rt	Offset
100101	sssss	ttttt	iiiiiiiiiiiiiiii

The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. A 16-bit half word is read from memory at the effective address, zero extended and loaded into Reg.File[Rt]. If the effective address is an odd number, an address error exception occurs.

Load Upper Immediate: (This instruction in conjunction with an OR immediate instruction is used to implement the Load Address pseudo instruction - la Label)

lui Rt, Imm **# RF[Rt] = Imm<<16 | 0x0000**

Op-Code		Rt	Imm
001111	00000	ttttt	iiiiiiiiiiiiiiii

The 16-bit immediate value is shifted left 16-bits concatenated with 16 zeros and loaded into Reg.File[Rt].

Load Word:

lw Rt, offset(Rs) **# RF[Rt] = Mem[RF[Rs] + Offset]**

Op-Code	Rs	Rt	Offset
100011	sssss	ttttt	iiiiiiiiiiiiiiii

The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. A 32-bit word is read from memory at the effective address and loaded into Reg.File[Rt]. If the least two significant bits of the effective address are not zero, an address error exception occurs. There are four bytes in a word, so word addresses must be binary numbers that are a multiple of four, otherwise an address error exception occurs.

Load Word Left:

lwl Rt, offset(Rs) **# RF[Rt] = Mem[RF[Rs] + Offset]**

Op-Code	Rs	Rt	Offset
100010	sssss	ttttt	iiiiiiiiiiiiiiii

The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective byte address. From one to four bytes will be loaded left justified into Reg.File[Rt] beginning with the effective byte address then it proceeds toward a lower order byte in memory, until it reaches the lowest order byte of the word in memory. This instruction can be used in combination with the LWR instruction to load a register with four consecutive bytes from memory, when the bytes cross a boundary between two words.

Load Word Right:

lwr Rt, offset(Rs) **# RF[Rt] = Mem[RF[Rs] + Offset]**

Op-Code	Rs	Rt	Offset
100110	sssss	ttttt	iiiiiiiiiiiiiiii

The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective byte address. From one to four bytes will be loaded right justified into Reg.File[Rt] beginning with the effective byte address then it proceeds toward a higher order byte in memory, until it reaches the high order byte of the word in memory. This instruction can be used in combination with the LWL instruction to load a register with four consecutive bytes from memory, when the bytes cross a boundary between two words.

Move From High:

mfhi Rd **# RF[Rd] = HIGH**

Op-Code			Rd		Function Code
000000	00000	00000	ddddd	00000	010000

Load Reg.File[Rd] with a copy of the value currently in special register HIGH.

Move From Low:

mflo Rd **# RF[Rd] = LOW**

Op-Code			Rd		Function Code
000000	00000	00000	ddddd	00000	010010

Load Reg.File[Rd] with a copy of the value currently in special register LOW.

Move to High:

mthi Rs **# HIGH = RF[Rs]**

Op-Code	Rs				Function Code
000000	sssss	00000	00000	00000	010001

Load special register HIGH with a copy of the value currently in Reg.File[Rs].

Move to Low:

mtlo Rs **# LOW = RF[Rs]**

Op-Code	Rs				Function Code
000000	sssss	00000	00000	00000	010011

Load special register LOW with a copy of the value currently in Reg.File[Rs].

Multiply:

mult Rs, Rt **# High |Low = RF[Rs] * RF[Rt]**

Op-Code	Rs	Rt			Function Code
000000	sssss	ttttt	00000	00000	011000

Multiply the contents of Reg.File[Rs] by Reg.File[Rt] and store the lower 32-bits of the product in the LOW register, and store the upper 32-bits of the product in the HIGH register. The two operands are treated as two's complement numbers, the 64-bit product is negative if the signs of the two operands are different. No overflow exception occurs under any circumstances. For some implementations of the MIPS architecture it takes 32 clock cycles to execute the multiply instruction.

Multiply Unsigned:

multu Rs, Rt **# High |Low = RF[Rs] * RF[Rt]**

Op-Code	Rs	Rt			Function Code
000000	sssss	ttttt	00000	00000	011001

Multiply the contents of Reg.File[Rs] by Reg.File[Rt] and store the lower 32-bits of the product in the LOW register, and store the upper 32-bits of the product in the HIGH register. The two operands are treated as unsigned positive values. No overflow exception occurs under any circumstances. For some implementations of the MIPS architecture it takes 32 clock cycles to execute the multiply instruction.

NOR:

nor Rd, Rs, Rt **# RF[Rd] = RF[Rs] NOR RF[Rt]**

Op-Code	Rs	Rt	Rd		Function Code
000000	sssss	ttttt	ddddd	00000	100111

Bit wise logically NOR contents of Register File[Rs] with Reg.File[Rt] and store result in Reg.File[Rd].

OR:

or Rd, Rs, Rt **# RF[Rd] = RF[Rs] OR RF[Rt]**

Op-Code	Rs	Rt	Rd		Function Code
000000	sssss	ttttt	ddddd	00000	100101

Bit wise logically OR contents of Register File[Rs] with Reg.File[Rt] and store result in Reg.File[Rd].

OR Immediate:

ori Rt, Rs, Imm **# RF[Rt] = RF[Rs] OR Imm**

Op-Code	Rs	Rt	Imm
001101	sssss	ttttt	iiiiiiiiiiiiiiii

Bit wise logically OR contents of Reg.File[Rs] wih zero extended Imm value and store result in Reg.File[Rt].

Store Byte:

sb Rt, offset(Rs) **# Mem[RF[Rs] + Offset] = RF[Rt]**

Op-Code	Rs	Rt	Offset
101000	sssss	ttttt	iiiiiiiiiiiiiiii

The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. The least significant 8-bit byte in Reg.File[Rt] are stored in memory at the effective address.

Store Halfword:

sh Rt, offset(Rs) **# Mem[RF[Rs] + Offset] = RF[Rt]**

Op-Code	Rs	Rt	Offset
101001	sssss	ttttt	iiiiiiiiiiiiiiii

The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. The least significant 16-bits in Reg.File[Rt] are stored in memory at the effective address. If the effective address is an odd number, then an address error exception occurs.

Shift Left Logical:

sll Rd, Rt, sa **# RF[Rd] = RF[Rt] << sa**

Op-Code		Rt	Rd	sa	Function Code
000000	00000	ttttt	ddddd	00000	000000

The contents of Reg.File[Rt] are shifted left sa-bits & the result is stored in Reg.File[Rd].

Shift Left Logical Variable:

sllv Rd, Rt, Rs **# RF[Rd] = RF[Rt] << RF[Rs] amount**

Op-Code	Rs	Rt	Rd		Function Code
000000	sssss	ttttt	ddddd	00000	000100

The contents of Reg.File[Rt] are shifted left by the number of bits specified by the low order 5-bits of Reg.File[Rs], and the result is stored in Reg.File[Rd].

Set on Less Than: (Used in branch macro instructions)

slt Rd, Rs, Rt **# if (RF[Rs] < RF[Rt]) then RF[Rd] =1 else RF[Rd] = 0**

Op-Code	Rs	Rt	Rd		Function Code
000000	sssss	ttttt	ddddd	00000	101010

If the contents of Reg.File[Rs] are less than the contents of Reg.File[Rt], then Reg.File[Rd] is set to one, otherwise Reg.File[Rd] is set to zero; assuming the two's complement number system representation.

Set on Less Than Immediate: (Used in branch macro instructions)

slti Rt, Rs, Imm **# if (RF[Rs] < Imm) then RF[Rt] =1 else RF[Rt] = 0**

Op-Code	Rs	Rt		Imm	
001010	sssss	ttttt	iiiii	iiiii	iiiiii

If the contents of Reg.File[Rs] are less than the sign-extended immediate value then Reg.File[Rt] is set to one, otherwise Reg.File[Rt] is set to zero; assuming the two's complement number system representation.

Set on Less Than Immediate Unsigned: (Used in branch macro instructions)

sltiu Rt, Rs, Imm **# if (RF[Rs] < Imm) then RF[Rt] =1 else RF[Rt] = 0**

Op-Code	Rs	Rt	Imm
001011	sssss	ttttt	iiiiiiiiiiiiiiii

If the contents of Reg.File[Rs] are less than the sign-extended immediate value, then Reg.File[Rt] is set to one, otherwise Reg.File[Rt] is set to zero; assuming an unsigned number representation (only positive values).

Set on Less Than Unsigned: (Used in branch macroinstructions)

sltu Rd, Rs, Rt **# if (RF[Rs] < RF[Rt]) then RF[Rd] =1 else RF[Rd] = 0**

Op-Code	Rs	Rt	Rd		Function Code
000000	sssss	ttttt	ddddd	00000	101010

If the contents of Reg.File[Rs] are less than the contents of Reg.File[Rt], then Reg.File[Rd] is set to one, otherwise Reg.File[Rd] is set to zero; assuming an unsigned number representation (only positive values).

Shift Right Arithmetic:

sra Rd, Rt, sa **# RF[Rd] = RF[Rt] >> sa**

Op-Code		Rt	Rd	sa	Function Code
000000	00000	ttttt	ddddd	00000	000011

The contents of Reg.File[Rt] are shifted right sa-bits, sign-extending the high order bits, and the result is stored in Reg.File[Rd].

Shift Right Arithmetic Variable:

srav Rd, Rt, Rs **# RF[Rd] = RF[Rt] >> RF[Rs] amount**

Op-Code	Rs	Rt	Rd		Function Code
000000	sssss	ttttt	ddddd	00000	000111

The contents of Reg.File[Rt] are shifted right, sign-extending the high order bits, by the number of bits specified by the low order 5-bits of Reg.File[Rs], and the result is stored in Reg.File[Rd].

Shift Right Logical:

srl Rd, Rt, sa **# RF[Rd] = RF[Rt] >> sa**

Op-Code		Rt	Rd	sa	Function Code
000000	00000	ttttt	ddddd	00000	000010

The contents of Reg.File[Rt] are shifted right sa-bits, inserting zeros into the high order bits, the result is stored in Reg.File[Rd].

Shift Right Logical Variable:

srlv Rd, Rt, Rs **# RF[Rd] = RF[Rt] >> RF[Rs] amount**

Op-Code	Rs	Rt	Rd		Function Code
000000	sssss	ttttt	ddddd	00000	000110

The contents of Reg.File[Rt] are shifted right, inserting zeros into the high order bits, by the number of bits specified by the low order 5-bits of Reg.File[Rs], and the result is stored in Reg.File[Rd].

Subtract:

sub Rd, Rs, Rt **# RF[Rd] = RF[Rs] - RF[Rt]**

Op-Code	Rs	Rt	Rd		Function Code
000000	sssss	ttttt	ddddd	00000	100010

Subtract contents of Reg.File[Rt] from Reg.File[Rs] and store result in Reg.File[Rd]. If overflow occurs in the two's complement number system, an exception is generated.

Subtract Unsigned:

subu Rd, Rs, Rt **# RF[Rd] = RF[Rs] - RF[Rt]**

Op-Code	Rs	Rt	Rd		Function Code
000000	sssss	ttttt	ddddd	00000	100011

Subtract contents of Reg.File[Rt] from Reg.File[Rs] and store result in Reg.File[Rd]. No overflow exception is generated.

Store Word:

sw Rt, offset(Rs) **# Mem[RF[Rs] + Offset] = RF[Rt]**

Op-Code	Rs	Rt	Offset
101011	sssss	ttttt	iiiiiiiiiiiiiiii

The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. The contents of Reg.File[Rt] are stored in memory at the effective address. If the least two significant bits of the effective address are not zero, an address error exception occurs. There are four bytes in a word, so word addresses must be binary numbers that are a multiple of four, otherwise an address error exception occurs.

Store Word Left:

swl Rt, offset(Rs) **# Mem[RF[Rs] + Offset] = RF[Rt]**

Op-Code	Rs	Rt	Offset
101010	sssss	ttttt	iiiiiiiiiiiiiiii

The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. From one to four bytes will be stored left justified into memory beginning with the most significant byte in Reg.File[Rt], then it proceeds toward a lower order byte in memory, until it reaches the lowest order byte of the word in memory. This instruction can be used in combination with the SWR instruction, to store the contents of a register into four consecutive bytes of memory, when the bytes cross a boundary between two words.

Store Word Right:

swr Rt, offset(Rs) **# Mem[RF[Rs] + Offset] = RF[Rt]**

Op-Code	Rs	Rt	Offset
101110	sssss	ttttt	iiiiiiiiiiiiiiii

The 16-bit offset is sign extended and added to Reg.File[Rs] to form an effective address. From one to four bytes will be stored right justified into memory beginning with the least significant byte in Reg.File[Rt], then it proceeds toward a higher order byte in memory, until it reaches the highest order byte of the word in memory. This instruction can be used in combination with the SWL instruction, to store the contents of a register into four consecutive bytes of memory, when the bytes cross a boundary between two words.

System Call: (Used to call system services to perform I/O)

syscall

```
   Op-Code                                      Function Code
00000000000000000000000000001100
```

A user program exception is generated.

Exclusive OR:

xor Rd, Rs, Rt

# RF[Rd] = RF[Rs] XOR RF[Rt]

```
   Op-Code      Rs      Rt      Rd               Function Code
000000ssssstttttddddd00000100110
```

Bit wise logically Exclusive-OR contents of Register File[Rs] with Reg.File[Rt] and store result in Reg.File[Rd].

Exclusive OR Immediate:

xori Rt, Rs, Imm

# RF[Rt] = RF[Rs] XOR Imm

```
   Op-Code      Rs      Rt           Imm
001110ssssstttttiiiiiiiiiiiiiiii
```

Bit wise logically Exclusive-OR contents of Reg.File[Rs] with zero extended Imm value and store result in Reg.File[Rt]

APPENDIX D

Macro Instructions

Name	Actual Code	Space/Time
Absolute Value:		
abs Rd, Rs	addu Rd, $0, Rs bgez Rs, 1 sub Rd, $0, Rs	**3/3**
Branch if Equal to Zero:		
beqz Rs, Label	beq Rs, $0, Label	**1/1**
Branch if Greater than or Equal :		
bge Rs, Rt, Label	slt $at, Rs, Rt beq $at, $0, Label	**2/2**
If Reg.File[Rs] > = Reg.File[Rt] branch to Label Used to compare values represented in the two's complement number system.		
Branch if Greater than or Equal Unsigned		
bgeu Rs, Rt, Label	sltu $at, Rs, Rt beq $at, $0, Label	**2/2**
If Reg.File[Rs] > = Reg.File[Rt] branch to Label Used to compare addresses (unsigned values).		
Branch if Greater Than:		
bgt Rs, Rt, Label	slt $at, Rt, Rs bne $at, $0, Label	**2/2**
If Reg.File[Rs] > Reg.File[Rt] branch to Label Used to compare values represented in the two's complement number system.		
Branch if Greater Than Unsigned:		
bgtu Rs, Rt, Label	sltu $at, Rt, Rs bne $at, $0, Label	**2/2**
If Reg.File[Rs] > Reg.File[Rt] branch to Label Used to compare addresses (unsigned values).		
Branch if Less Than or Equal:		
ble Rs, Rt, Label	slt $at, Rt, Rs beq $at, $0, Label	**2/2**
If Reg.File[Rs] < = Reg.File[Rt] branch to Label Used to compare values represented in the two's complement number system.		

Branch if Less Than or Equal Unsigned:

bleu Rs, Rt, Label		sltu $at, Rt, Rs	**2/2**
		beq $at, $0, Label	

If Reg.File[Rs] < = Reg.File[Rt] branch to Label
Used to compare addresses (unsigned values).

Branch if Less Than:

blt Rs, Rt, Label		slt $at, Rs, Rt	**2/2**
		bne $at, $0, Label	

If Reg.File[Rs] < Reg.File[Rt] branch to Label
Used to compare values represented in the two's complement number system

Branch if Less Than Unsigned:

bltu Rs, Rt, Label		sltu $at, Rs, Rt	**2/2**
		bne $at, $0, Label	

If Reg.File[Rs] < Reg.File[Rt] branch to Label
Used to compare addresses (unsigned values).

Branch if Not Equal to Zero:

bnez Rs, Label		bne Rs, $0, Label	**1/1**

Branch Unconditional

b Label		bgez $0, Label	**1/1**

Divide:

div Rd, Rs, Rt		bne Rt, $0, ok	**4/41**
		break $0	
	ok:	div Rs, Rt	
		mflo Rd	

Divide Unsigned:

divu Rd, Rs, Rt		bne Rt, $0, ok	**4/41**
		break $0	
	ok:	divu Rs, Rt	
		mflo Rd	

Load Address:

la Rd, Label		lui $at, Upper 16-bits of Label	**2/2**
		ori Rd, $at, Lower 16-bits of Label	

Used to initialize pointers.

Load Immediate:

li Rd, value		lui $at, Upper 16-bits of value	**2/2**
		ori Rd, $at, Lower 16-bits of value	

Initialize registers with negative constants and values greater than 32767.

Load Immediate:

li Rd, value	ori Rt, $0, value	**1/1**

Initialize registers with positive constants less than 32768.

Move:

move Rd, Rs	addu Rd, $0, Rs	**1/1**

mul Rd, Rs, Rt	mult Rs, Rt	**2/33**
	mflo Rd	

Multiply (with overflow exception):

mulo Rd, Rs, Rt		mult Rs, Rt	**7/37**
		mfhi $at	
		mflo Rd	
		sra Rd, Rd, 31	
		beq $at, Rd, ok	
		break $0	
	ok:	mflo Rd	

Multiply Unsigned (with overflow exception):

mulou Rd, Rs, Rt		multu Rs, Rt	**5/35**
		mfhi $at	
		beq $at, $0, ok	
	ok:	break $0	
		mflo Rd	

Negate:

neg Rd, Rs	sub Rd, $0, Rs	**1/1**

Two's complement negation. An exception is generated when there is an attempt to negate the most negative value: 2,147,483,648.

Negate Unsigned:

negu Rd, Rs	subu Rd, $0, Rs	**1/1**

Nop:

nop	or $0, $0, $0	**1/1**

Used to solve problems with hazards in the pipeline.

Not:

not Rd, Rs	nor Rd, Rs, $0	**1/1**

A bit-wise Boolean complement.

Remainder:

rem Rd, Rs, Rt	bne Rt, $0, 8	**4/40**
	break $0	
	div Rs, Rt	
	mfhi Rd	

Remainder Unsigned:

```
remu  Rd, Rs, Rt           bne Rt, $0, ok              4/40
                           break $0
                      ok:  divu Rs, Rt
                           mfhi Rd
```

Rotate Left Variable:

```
rol  Rd, Rs, Rt            subu $at, $0, Rt            4/4
                           srlv $at, Rs, $at
                           sllv Rd, Rs, Rt
                           or Rd, Rd, $at
```

The lower 5-bits in Rt specifys the shift amount.

Rotate Right Variable:

```
ror  Rd, Rs, Rt            subu $at, $0, Rt            4/4
                           sllv $at, Rs, $at
                           srlv Rd, Rs, Rt
                           or Rd, Rd, $at
```

Rotate Left Constant:

```
rol      Rd, Rs, sa        srl $at, Rs, 32-sa          3/3
                           sll Rd, Rs, sa
                           or Rd, Rd, $at
```

Rotate Right Constant:

```
ror  Rd, Rs, sa            sll $at, Rs, 32-sa          3/3
                           srl Rd, Rs, sa
                           or Rd, Rd, $at
```

Set if Equal:

```
seq  Rd, Rs, Rt            beq Rt, Rs, yes             4/4
                           ori Rd, $0, 0
                           beq $0, $0, skip
                     yes:  ori Rd, $0, 1
                     skip:
```

Set if Greater Than or Equal:

```
sge  Rd, Rs, Rt            bne Rt, Rs, yes             4/4
                           ori Rd, $0, 1
                           beq $0, $0, skip
                     yes:  slt Rd, Rt, Rs
                     skip:
```

Set if Greater Than or Equal Unsigned:

```
sgeu      Rd, Rs, Rt       bne Rt, Rs, yes             4/4
                           ori Rd, $0, 1
                           beq $0, $0, skip
                     yes:  sltu Rd, Rt, Rs
                     skip:
```

Set if Greater Than:			
sgt Rd, Rs, Rt		slt Rd, Rt, Rs	**1/1**
Set if Greater Than Unsigned:			
sgtu Rd, Rs, Rt		sltu Rd, Rt, Rs	**1/1**
Set if Less Than or Equal:			
sle Rd, Rs, Rt		bne Rt, Rs, yes	**4/4**
		ori Rd, $0, 1	
		beq $0, $0, skip	
	yes:	slt Rd, Rs, Rt	
	skip:		
Set if Less Than or Equal Unsigned:			
sleu Rd, Rs, Rt		bne Rt, Rs, yes	**4/4**
		ori Rd, $0, 1	
		beq $0, $0, skip	
	yes:	sltu Rd, Rs, Rt	
	skip:		
Set if Not Equal:			
sne Rd, Rs, Rt		beq Rt, Rs, yes	**4/4**
		ori Rd, $0, 1	
		beq $0, $0, skip	
	yes:	ori Rd, $0, 0	
	skip:		
Unaligned Load Halfword Unsigned:			
ulh Rd, 3(Rs)		lb Rd, 4(Rs)	**4/4**
		lbu $at, 3(Rs)	
		sll Rd, Rd, 8	
		or Rd, Rd, $at	
Unaligned Load Halfword:			
ulhu Rd, 3(Rs)		lbu Rd, 4(Rs)	**4/4**
		lbu $at, 3(Rs)	
		sll Rd, Rd, 8	
		or Rd, Rd, $at	
Unaligned Load Word:			
ulw Rd, 3(Rs		lwl Rd, 6(Rs)	**2/2**
		lwr Rd, 3(Rs)	
Unaligned Store Halfword:			
ush Rd, 3(Rs)		sb Rd, 3(Rs)	**3/3**
		srl $at, Rd, 8	
		sb $at, 4(Rs)	
Unaligned Store Word:			
usw Rd, 3(Rs)		swl Rd, 6(Rs)	**2/2**
		swr Rd, 3(Rs)	

APPENDIX E

A Trap Handler

```
# SPIM S20 MIPS simulator.
# The default trap handler for spim.
#
# Copyright (C) 1990-1995 James Larus, larus@cs.wisc.edu.
# ALL RIGHTS RESERVED.
#
# SPIM is distributed under the following conditions:
#
# You may make copies of SPIM for your own use and modify those copies.
#
# All copies of SPIM must retain my name and copyright notice.
#
# You may not sell SPIM or distributed SPIM in conjunction with a commercial
# product or service without the expressed written consent of James Larus.
#
# Define the exception handling code.  This must go first!
                .kdata
__m1_:          .asciiz "  Exception "
__m2_:          .asciiz " occurred and ignored\n"
__e0_:          .asciiz "  [Interrupt] "
__e1_:          .asciiz  ""
__e2_:          .asciiz  ""
__e3_:          .asciiz  ""
__e4_:          .asciiz  "  [Unaligned address in inst/data fetch] "
__e5_:          .asciiz  "  [Unaligned address in store] "
__e6_:          .asciiz  "  [Bad address in text read] "
__e7_:          .asciiz  "  [Bad address in data/stack read] "
__e8_:          .asciiz  "  [Error in syscall] "
__e9_:          .asciiz  "  [Breakpoint] "
__e10_:         .asciiz  "  [Reserved instruction] "
__e11_:         .asciiz  ""
__e12_:         .asciiz  "  [Arithmetic overflow] "
__e13_:         .asciiz  "  [Inexact floating point result] "
__e14_:         .asciiz  "  [Invalid floating point result] "
__e15_:         .asciiz  "  [Divide by 0] "
__e16_:         .asciiz  "  [Floating point overflow] "
__e17_:         .asciiz  "  [Floating point underflow] "
__excp:         .word __e0_,__e1_,__e2_,__e3_,__e4_,__e5_,__e6_,__e7_,__e8_,__e9_
                .word __e10_,__e11_,__e12_,__e13_,__e14_,__e15_,__e16_,__e17_
s1:             .word 0
s2:             .word 0
```

```
        .ktext 0x80000080
        .set noat
        # Because we are running in the kernel, we can use $k0/$k1 without
        # saving their old values.
        move    $k1, $at                # Save $at
        .set at
        sw      $v0, s1                 # Not re-entrent and we can't trust $sp
        sw      $a0, s2
        mfc0    $k0, $13                # Cause
        sgt     $v0, $k0, 0x44          # ignore interrupt exceptions
        bgtz    $v0, ret
        addu    $0, $0, 0
        li      v0, 4                   # syscall 4 (print_str)
        la      $a0, __m1_
        syscall
        li      $v0, 1                  # syscall 1 (print_int)
        srl     $a0, $k0, 2             # shift Cause reg
        syscall
        li      $v0, 4                  # syscall 4 (print_str)
        lw      $a0, __excp($k0)
        syscall
        bne     $k0, 0x18, ok_pc        # Bad PC requires special checks
        mfc0    $a0, $14                # EPC
        and     $a0, $a0, 0x3           # Is EPC word-aligned?
        beq      $a0, 0, ok_pc
        li      $v0, 10                 # Exit on really bad PC (out of text)
        syscall
ok_pc:
        li      $v0,4                   # syscall 4 (print_str)
        la      $a0, __m2_
        syscall
        mtc0     $0, $13                # Clear Cause register
ret:
        lw      $v0, s1
        lw       $a0, s2
        mfc0    $k0, $14                # EPC
        .set noat
        move     $at, $k1               # Restore $at
        .set at
        rfe                             # Return from exception handler
        addiu    $k0, $k0, 4            # Return to next instruction
        jr $k0
```

```
# Standard startup code.  Invoke the routine main with no arguments.

        .text
        .globl __start
__start:
        lw      $a0, 0($sp)             # argc
        addiu   $a1, $sp, 4             # argv
        addiu   $a2, $a1, 4             # envp
        sll     $v0, $a0, 2
        addu    $a2, $a2, $v0
        jal      main
        li      $v0 10
        syscall                         # syscall 10 (exit)
```