Lab 3 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
```

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.

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**."

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.

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.

Function Calls Using the Stack

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.

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.

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
```

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 <####
```

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.

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.

Assignment

3.1 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."

3.2 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."

3.3 **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)

3.4 **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.