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 textbook 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. Programmers are expected to use the convention defined in Section 6.3. If everyone uses the same convention, then it should be possible to run a main program using functions written by many different individuals.

6.2 THREE DIFFERENT MEMORY SEGMENTS

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. 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 functions or reentrant functions. Reentrant functions are defined in Chapter 7. 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

In addition to a register usage convention, there is a parameter-passing convention that will accommodate any number of parameters, by using the stack. This convention states that 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 function exercises presented in this textbook 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.

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, three of which are passed *to* the function (in parameters) and two of which are 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 such as JILL (A, B, C, D, E) defined.

Here is an example of how a **nested function call** is accomplished in MIPS assembly language:

```
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    # Deallocate Space on the Stack
```

Notice that 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 crosses over the midpoint of the memory space.

6.4 NESTED FUNCTION CALLS AND LEAF FUNCTIONS

The scenario described in the previous section 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 are as follows:

```
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 following are the last few instructions in the function JILL to return the two out parameters D and E:

```
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
```

With nested function calls, sometimes there is one more complexity. Let's suppose that in the case of the function JACK it is important that the values in registers $t6 and $t7 not be lost as a result of calling JILL. The only way to ensure that 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. The decision as to what registers need to be saved is dependent on understanding the algorithm being implemented. A leaf function never needs to save t registers. The following is an example of how Jack would ensure that the values in registers $t6 and $t7 would 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 $t6 on the stack <####
sw      $t7,  28($sp)   # Save $t7 on the stack <####
jal     JILL            # call the Function
lw      $t6,  24($sp)   # Restore $t6 from the stack <####
lw      $t7,  28($sp)   # Restore $t7 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    # Deallocate Space on the Stack <####
```

6.5 ALLOCATING SPACE ON THE STACK FOR LOCAL VARIABLES

As functions become more complex, a situation may arise where additional 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 local variables allocated space on the stack. For example, if Jill needs a temporary array of 16 characters, the code to allocate space on the stack is the first instruction shown next. The second instruction initializes register `$a0` as a pointer to the beginning of this new buffer area on the stack. The two instructions are as follows:

```
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 deallocated. For this example, deallocation is accomplished by executing the following instruction:

```
addiu   $sp, $sp, 16    # Deallocate space
```

6.6 FRAME POINTER

In the preceding code example, you will notice that allocating space on the stack for additional local variables requires the address in the stack pointer to be changed. In all previous examples we assumed the stack pointer would not change and we could reference each element on the stack 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**. When used, the frame pointer points to the top most word of the activation record. The use of a frame pointer is not a necessity. Some high-level language compilers generate code that use a frame pointer and others do not. None of the exercises in this text are sufficiently complex to warrant the use of a frame pointer.

6.7 DYNAMIC MEMORY ALLOCATION

Programs request services from the operating system by executing the `syscall` instruction, which causes a software exception. One of the functions of the operating system is to manage memory. During run time, programs can make requests to the operating system to be dynamically allocated additional space in memory by performing a `syscall (9)`. The **heap** is the portion of memory managed by the operating system from which the user is allocated dynamic memory space.

The following program provides a demonstration of dynamic memory allocation with PCSpim:

```
#####################################################################
# Functional Description: Testing Dynamic Storage Allocation
#####################################################################
      .data                 #
      .byte   -1:8          # Eight bytes sacrificed to get around bug.
msg:  .asciiz "**Hello*World**"
      .text
main:                       # DYNAMICALY ALLOCATE SPACE
      li      $a0, 16       # $a0 = string length including null.
      li      $v0, 9        # Upon return $v0 will point to dynamically
      syscall               # allocated string for 16 characters.
      move    $a0, $v0      # Save Ptr to dynamically allocated string
      la      $a1, msg      # $a1 = source pointer
loop:
      lb      $t9,0($a1)    # Get a character.
      sb      $t9,0($v0)    # Store in dynamically allocated space.
      addi    $a1, $a1, 1   # Increment pointers.
      addi    $v0, $v0, 1
      bnez    $t9, loop     # Branch to loop if not at end of string.
      li      $v0, 4        # Print contents of dynamically allocated
      syscall               # string.
      li      $v0, 10
      syscall
```

The number of additional bytes of memory space requested is passed to the operating system in register `$a0`. A pointer to this newly allocated space is passed back to the user in register `$v0`. The size of the dynamic string is specified in bytes, but the number must be a multiple of four.

6.8 AN EXAMPLE DYNAMIC STRING ALLOCATION PROGRAM

The objective of the next main program and associated functions is to create two dynamic strings and then to append these two dynamic strings to create a new larger dynamic string. For this example, the first dynamic string will contain the characters `This is a test` and the second dynamic string will contain the characters `of our string routines`. Here is the code:

```
#####################################################################
# Functional Description: A main program
# to test dynamic string allocation functions
# For syscall 9 - we must specify size of dynamic string
# in bytes, but the number must be a multiple of 4
#####################################################################
        .data                   # Data declaration section
title:  .asciiz  "\n Dynamic Allocation of Memory Exercise\n\n"
str1:   .asciiz  "This is a test"
str2:   .asciiz  "of our string routines."
        .text
main:                           # Start of code section
        la      $a0, title
        jal     print
        la      $a0, str1
        jal     create
        move    $s1, $v0        # Get pointer to dynamic str1

        la      $a0, str2
        jal     create
        move    $s2, $v0        # Get pointer to dynamic str2
        move    $a0, $s1
        move    $a1, $s2
        jal     append
        move    $a0, $v0        # Returned pointer to new string
        jal     print
        li      $v0, 10
        syscall
#####################################################################
# Functional Description: Create ($v0, $a0)
# Involves a nested call to length,
# $v0:points to a dynamically allocated string
# $a0:points to the source string that fills the dynamically
  allocated string
#####################################################################
create:
        addi    $sp, $sp,-8     # Allocate space
        sw      $ra, 0($sp)
        sw      $a0, 4($sp)     # Save source pointer
```

```
        jal     length
        addi    $a0, $v0,1      # $a0 = length N + 1
        addi    $a0, $a0,4
        srl     $a0, $a0,2
        sll     $a0, $a0,2
        li      $v0, 9
        syscall                 # $v0 now points to string with
                                # space for N+1 characters
        move    $t0, $v0        # t0 gets copy of destination pointer
        lw      $t1, 4($sp)     # Get source pointer
createloop:
        lb      $t9, 0($t1)     # Transfer character
        sb      $t9, 0($t0)
        addi    $t1, $t1, 1     # Increment temp pointers
        addi    $t0, $t0, 1
        bnez    $t9, createloop
        lw      $ra, 0($sp)
        lw      $a0, 4($sp)     # Save source pointer
        addi    $sp, $sp,8      # Deallocate space
        jr      $ra
#########################################################################
# Functional Description:Length($v0, $a0) returns a value in $v0
# corresponding to the length of the string pointed to by $a0.
# Length does not enclude null at end of string
#########################################################################
length:
        li      $v0, -1
        move    $t1, $a0
lengthloop:
        lb      $t0, 0($t1)
        addi    $t1, $t1, 1
        addi    $v0, $v0, 1
        bnez    $t0, lengthloop
        jr      $ra
#########################################################################
# Functional Description: Append($v0, $a0, $a1)
# Involves a nested call to length. Must save $a0, $a1, & $ra
# $a0:points to first source string
# $a1:points to second source string
# $v0:points to a dynamically allocated string containing above two
# strings concatinated
#########################################################################
append:
        addi    $sp, $sp,-16    # Allocate space
        sw      $ra, 0($sp)
        sw      $a0, 4($sp)     # source1 pointer
        sw      $a1, 8($sp)     # source2 pointer
```

```
        jal     length
        addi    $t3, $v0, 1     # $t3 = length first string + 1
        sw      $t3, 12($sp)    # Save on stack
        lw      $a0, 8($sp)
        jal     length          # Get length of second string
        lw      $t3, 12($sp)    # Get length of first string + 1
        add     $a0, $v0,$t3    # a0 is length of both strings + 1
        addi    $a0, $a0, 4
        srl     $a0, $a0, 2
        sll     $a0, $a0, 2
        li      $v0, 9
        syscall                 # $v0 now points to a string with
                                # space for both strings
        move    $t3, $v0        # $t3 is temp pointer to destination
        lw      $t1, 4($sp)     # Get source 1 pointer
xferfirst:
        lb      $t9,0($t1)      # Transfer character
        sb      $t9,0($t3)
        addi    $t1, $t1, 1     # Increment temp pointers
        addi    $t3, $t3, 1
        bnez    $t9, xferfirst
        addi    $t3, $t3, -1    # To write over null char
        lw      $t2, 8($sp)     # Pointer to second string
xfersecond:
        lb      $t9,0($t2)      # Transfer character
        sb      $t9,0($t3)
        addi    $t2, $t2, 1     # Increment temp pointers
        addi    $t3, $t3, 1
        bnez    $t9, xfersecond
        lw      $ra, 0($sp)
        addi    $sp, $sp, 16 # Deallocate space
        jr      $ra
#####################################################################
# Functional Description: Print($a0)Prints string pointed to by $a0
#####################################################################
print:
        li      $v0, 4
        syscall
        jr      $ra
```

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. uppercase letters (U)
b. lowercase 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 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

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

Also, write a main program to test this function.

6.5 Fibonacci (N, E)
Write an iterative function to return the Nth element in the Fibonacci sequence. A value N is passed to the function on the stack, and the Nth 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, ...

6.6 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.

6.7 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.

6.8 Write your most efficient assembly language code translation for the following function and main line calling program.

```
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
}
```

Note that all communication with the function 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.

6.9 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.

6.10 The purpose of this exercise is to use pointers in assembly language to manage dynamic structures. Most modern computer languages use pointers of some kind to provide a mechanism for dealing with dynamic structures—structures whose sizes change during execution. We use `syscall 9` to acquire memory space from the heap at run time. A dynamic string is a dynamically allocated varying length string. Such strings end with a null character(0). Such strings are accessed via a pointer to the first character of the string. The shortest string is defined to be of length zero, because it contains no characters other than a single null character. In this case, the pointer will point at the null character. For this exercise pointers are memory addresses passed to the functions in registers. Construct the following functions in MIPS assembly language:

Length ($v0, $a0)—returns the length in `$v0` of the referenced string, pointed to by the pointer in `$a0`. Length is similar to the `strlen` function in C/C++.

Create ($v0, $a0)—returns a pointer in `$v0` to the ASCII string that it dynamically allocates and loads with the string pointed to by the pointer in `$a0`.

Append ($v0, $a0, $a1)—returns a pointer in `$v0` to the dynamic string that results from concatenating the two ASCII strings pointed to by `$a0`, and `$a1`.

Print ($a0)—prints the string pointed to by pointer in `$a0`.

Write a main program to test your functions by performing these actions:

1. Create a string, str1, containing "This is a test".
2. Create a string, str2, containing "of our string routines."
3. Create a string, str3, by appending str1 to str2.
4. Print a title, followed by str1, str2, and str3. Properly label your output.
5. Create an empty string; call it str4.
6. Perform a loop 5 times that
   a. Inputs a string from the keyboard, call it str5
   b. Appends str5 to str4
   c. Prints str4