C and Assembly Language Linkages

When a program calls a function, the calling sequence (or linkage) is that sequence of instructions which set up (and make) the function call, and which clean up afterwards. When you call an assembly language subroutine from a C program, your assembly language subroutine must assume the caller follows the C calling conventions described in the compiler's manual. In our case, when a C program calls a function, it allocates on the stack 36 bytes (9 words) for the registers to be preserved across the function call, but the called function does the saving and restoring. So, if the assembly language program calls a C library function, it too must allocate 36 bytes before the call, and deallocate the 36 bytes after the function returns.

This is an assembly language program solving the Tower of Hanoi; it is written in MIPS 3000 assembler (and was assembled and run using the *cc*(1) and *gcc*(1) programs on the DECStations). The program is in ~farrens/academic/50/samples/hanoi.s.

```
#
# This procedure is the Tower of Hanoi as done for a RISC machine
# using the DEC assembler (either cc or gcc); it is called from a
# C program.  Here are the specs:
#
# Name: hanoi
# Written by:  Matt Bishop, bishop@cs.ucdavis.edu
#              minor modifications to a program done for COSC 37
#              (Dartmouth College) in April 1992
# Action:  This prints messages saying how to move n disks from one
#          tower to another without ever placing a larger disk on a
#          smaller; it's the classic Tower of Hanoi puzzle.
# Assumptions:  * This is called either by a C program compiled by the
#                 ULTRIX or GNU C compilers, or by something which uses
#                 the same calling sequence.
#               * There are 3 towers.
#               * The number of disks to move is non-negative.
# Input Parameters: All are passed through the registers, following
#                   standard R3000/4000 conventions:
#                   $4 -- number of disks to be moved
#                   $5 -- string to be printed to represent first tower
#                   $6 -- string to be printed to represent second tower
#                   $7 -- string to be printed to represent third tower
# Output Parameters: $2 -- number of moves made
# Registers used:   $0 -- to get a 0 (standard usage)
#                   $8 -- temporary (usually holds $4 - 1)
#                  $31 -- return address (standard usage)
# Stack Usage:   return address           0($sp)
#                number of disks - 1      4($sp)
#                number of moves          8($sp)
#                number of disks (arg 1) 12($sp)
#                to pointer (arg 2)      16($sp)
#                from pointer (arg 3)    20($sp)
#                to pointer (arg 4)      24($sp)
# Side Effects: prints messages to stdout
#
# first, the printing format
#
        .data
```

```
        .align  0
fmt:    .ascii  "move disk %d from %s to %s\X0A\X00":1
        .text
 #
 #      now, the program
 #
 #      prologue
        .align  2               # word alignment
        .globl  hanoi           # this is called from another file
        .ent    hanoi 2         # procedure begins
        .set    noreorder       # don't want the following rearranged!
hanoi:
        addiu   $sp, $sp, -28   # make room on the stack
        sw      $31, 0($sp)     # save the return address
        add     $8, $0, $4      # move the argument into a temporary
        sw      $5, 16($sp)     # save the from, to, other
        sw      $6, 20($sp)     #   ... pointers (passed in
        sw      $7, 24($sp)     #   ... registers 5-7)
        #
        #       first, see if we have to move any disks
        #       if not, just return 0
        #
        bne     $8, 0, nzero    # do we need to move anything (n == 0)
        nop                     # for the delay slot
        add     $2, $0, $0      # nope -- store 0 in return register
        beq     $0, $0, epilog  # branch to epilogue
        nop                     # for the delay slot
        #
        #       now move n-1 disks from "from" to "other"
        #       we'll move them from "other" to "to" later
        #
nzero:  addu    $4, $8, -1      # move disks above disk n
        lw      $5, 16($sp)     #   ... from "from"
        lw      $6, 24($sp)     #   ... to "other"
        lw      $7, 20($sp)     #   ... using "to" as intermediate
        sw      $4, 4($sp)      # and the disk number n-1
        jal     hanoi           # recurse
        sw      $8, 12($sp)     # save the current value of n
                                # note this is done BEFORE the jal!
        lw      $8, 12($sp)     # restore this procedure's idea of n
                                # note this is done AFTER the jal!
        #
        #       now we print the move
        #       and track it in the move counter
        #
        addiu   $3, $2, 1       # one more move made
        la      $4, fmt         # get the format string address
                                #   ... really translates to
                                #       lui $8, 0
                                #       addiu $8, $8, fmt
        add     $5, $0, $8      # get the current disk number
        lw      $6, 16($sp)     # get the peg which it is on
        lw      $7, 20($sp)     # get the peg to move it to
        sw      $3, 8($sp)      # save the current number of moves
        #
        # this requires a bit of explanation; UNIX library routines
        # assume the CALLER has made room on the stack for the 9
        # registers that are saved across calls ... so we do this
        # here (36 = 9 * 4 and each register is 4 bytes)
```

```
        #
        addiu   $sp, $sp, -36    # make room on the stack
        jal     printf           # print the message
        nop                      # no-op to handle delay
        addiu   $sp, $sp, 36     # make room on the stack
        lw      $3, 8($sp)       # restore the current number of moves
        #
        #       now move n-1 disks from "other" to "to"
        #
        lw      $4, 4($sp)       # move disks n-1 and above
        lw      $5, 24($sp)      # ... from the "other" peg
        lw      $6, 20($sp)      # ... to the "to" peg
        lw      $7, 16($sp)      # ... using "from" as intermediate
        jal     hanoi            # recurse
        nop                      # need this to clear the pipeline
        lw      $3, 8($sp)       # get the current number of moves
        nop                      # again, needed to clear pipeline
        addu    $3, $3, $2       # add the number of moves that call made
                                 # to the number made so far in this call
        add     $2, $0, $3       # store it in the return register
        #
        #       clean up and return
        #
epilog: lw      $31, 0($sp)      # restore return address
        addu    $sp, 28          # pop the stack
        j       $31              # return (actually, jr ...)
        nop                      # for the next cycle ...
        .end    hanoi            # end of procedure
```

Here is an example of how to call this subroutine from a C program. This program is in ~farrens/academic/50/samples/callhanoi.c.

```
/*
 * TOWER OF HANOI -- see any good puzzle book for this puzzle
 * Used in ECS50 to call the MIPS "Towers of Hanoi" subroutine
 *
 * Written 2/28/94 by Matt Bishop
 */
#include <stdio.h>
#include <stdlib.h>

int hanoi(int, char *, char *, char *);/* solve Towers of Hanoi */

void main(int argc, char *argv[])
{
        long ndisks;            /* number of disks to move */
        long nmoves;            /* how many moves did it take */

        /*
         * error checking
         */
        if (argc != 2){
                fprintf(stderr, "Usage: %s number_of_disks\n", argv[0]);
                exit(EXIT_FAILURE);
        }

        /*
         * get the number of disks to move
         */
        if (sscanf(argv[1], "%d", &ndisks) != 1 || ndisks <= 0){
```

```
                fprintf(stderr, "Argument must be a positive integer\n");
                exit(EXIT_FAILURE);
        }

        /*
         * move them
         */
        nmoves = hanoi(ndisks, "disk #1", "disk #2", "disk #3");
        printf("Moving %d disks required %d moves\n", ndisks, nmoves);

        /*
         * say bye with flair!
         */
        exit(EXIT_SUCCESS);
}
```

Using *gcc* or *cc*

Gcc is the GNU C compiler; it can also assemble programs. Put the assembly language program in a file with a name ending in ".s", and give it to *gcc*. So, to compile the above programs together, simply say:

```
gcc callhanoi.c hanoi.s
```

and the resulting executable will be named a.out. If you want to call it "hanoi", use the –o option:

```
gcc -o hanoi callhanoi.c hanoi.s
```

If you simply want to generate an object file (that is, the output of the assembler without any linking), give the –c option:

```
gcc -c hanoi.s
```

The object file will be called "hanoi.o". If you want to *disassemble* it, to see what the actual machine language instructions were, use the *dis* program:

```
dis hanoi.o
```

You will notice that some assembly language instructions translate into more than one MIPS instruction. If you didn't turn off instruction reordering, some instructions may be rearranged. Both these events are quite normal.

If you want the compiler to translate the C program into an equivalent assembly language program, say

```
gcc -S callhanoi.c
```

The assembly will be in the file "callhanoi.s". This is useful for figuring out how the C compiler calls functions.

You can get all the options for *gcc* from its manual page. Type man gcc for this.

The compiler *cc*(1) is DEC's version of the C compiler. The options above all apply to it, and it uses the same calling sequence as *gcc*.