CSC258H Week 11 Lab: Function Calls

1 Introduction

Last week, we built branches and loops in MIPS assembly using labels and branches. This week, we will write functions. To do so, we must manipulate the stack to set up and tear down frames, and we will have to agree on a convention for passing arguments and return values between the calling site and the function being called.

The last assignment requires that you be able to set up functions, so it's important you be comfortable with the material from lab today. **You may need additional time to finish this lab, so starting early is recommended.** Ask the TA a question whenever you see something you don't understand.

This week's lab assumes that you completed all of the material from last week. If you do not get the program in Section 4 working last week, finish that lab before this week's session.

2 Function Call Interface

The function call conventions have two sides: the caller and callee responsibilities.

Caller: Before calling a function, the caller must save any temporary registers (t registers) on the stack and then prepare the arguments for the callee. The first four arguments are stored in $a0$ to $a3$. Any other arguments are stored on the stack. After the callee returns, the caller restores temporary registers and fetches the return value from $v0$.

Callee: The callee must set up and tear down its on stack frame. The callee saves the caller's fp (frame pointer) and its own ra (return address) on the stack, saves any callee save registers (s registers) that it will use, and allocates space for any local variables that need to be stored on the stack at some point during the function's execution. At the end of the function call, the caller restores the fp and ra registers, clears the allocated stack space, and places the return value in $v0$.

Function calls are a *convention* – an agreement between tools and programmers to follow a specific procedure. **Your code in this week's lab may run if you do not completely follow our convention, but it is incorrect if you do not fully implement it.** Functions can be implemented using several different conventions, and you should implement the one specified in Section A.6 of Jim Larus's MIPS reference (which is slightly different from the one gcc implements that we saw in class and the one in the textbook):

`http://pages.cs.wisc.edu/~larus/HP_AppA.pdf`

Before lab, generate an ordered list of operations/tasks for both the caller and callee that you must perform to implement Larus's function call convention.

3 Simple Function Call

Open your `average` program from last week, and recall that the program asks the user for N integers and then prints the average of those integers. The code should be roughly divided into three parts: a section that prompts for an integer length and checks it for legality, a loop that prompts for N integers and keeps a running sum, and a short section that computes the average and prints it.

Identify the code that prompts for an integer. Write a function that takes a single argument – the address of the prompt to print – and that returns an integer provided by the user. Verify that this integer is greater than zero before returning it to the user. If the integer is zero or negative, prompt the user for input again until they provide a positive integer. *Pay very careful attention to your function call convention.* Make sure that it saves the argument, frame pointer, and return address on entrance to the function and that the stack frame is torn down appropriately after each call.

Now, alter your program so that it calls your integer prompting function when it prompts for a length and when it asks for an integer to add to the sum. After these changes, your program should work exactly as before *except that it only accepts positive integers.* Single step through your program and pay very careful attention to the stack. Make sure that the ra and fp registers are stored where you expect, and verify that after each function call, the stack is returned to its original state. When you are ready, call your TA over to demonstrate your code. **They will verify aspects of the convention.**

4 Recursive Calls

Next, write a program that uses recursion to compute the Pell numbers. Recall that P_0 is 0, P_1 is 1, and $P_N = 2 * P_{N-1} + P_{N-2}$. Your program should prompt the user for an integer N and use recursion (calling a function named `pell`) to compute P_N. It should print that Pell number before exiting.

If the function call convention you're using is not complete, a recursive function will likely reveal the flaw ... (Note: You may be able, however, to get a convention to work that does not match Larus's conventions.) If you run into problems, set a breakpoint at the beginning of the function call in the caller and a second breakpoint at the beginning of the function return in the callee. Step through the code, checking the stack at each step. The call and stack setup should be *exactly* reversed during the stack teardown and return. If the two are identical, then double-check that your code saves any registers that it uses; if not, any following recursive calls will overwrite them.

5 Summary of TODOs

Below is a short summary of the steps to be completed for this lab:

1. Complete last week's `average` program if you have not already.
2. Create a list of tasks that must be completed for both the caller and callee to implement Larus's function call convention.
3. Implement a function in last week's `average` program.
4. Implement a new program that uses a recursive call to `pell` to compute the requested Pell number.

Evaluation (3 marks in total): 1 mark for showing your TA your list of tasks, 1 mark for having your TA observe as you single step through a simple function call that implement's Larus's convention, and 1 mark for implementing Larus's convention to implement a recursive call. Be prepared to explain what various lines of assembly code do and to defend how they fully implement the specified conventions.

Next week's lab is the last lab of the course. We'll be exploring exceptions and exception handling.