Lab4 – Mastering RISC-V Calling Conventions and Stack Management

Introduction

For this lab you will be provided with a simple recursive program that does multiplication via repeated addition. We will also see system calls in action to print out results. This program is provided via our class git repo. There are 3 parts to this lab. Part 1 is about understanding how a recursive program works, Part 2 involves making a simple modification to the code from Part 1, and finally, in Part 3 you will be developing a simple recursive program on your own.

PART 1: Understanding the Provided Program

Walk through the provided code with the debugger and make sure you understand what is going on. The hard coded values multiply 5x3 and prints out: "The result of 5 * 3 = 15".

The implemented algorithm for performing recursive multiplication is:

```
#####################################
#ALGORITHM
#  multiply(a,b):
#    if b == 0:
#      return 0
#    else:
#      return a + multiply(a, b-1)
#####################################
```

For the input values of 5 and 3, you will be making a total of 4 calls to multiply:

```
multiply(5,3):
      = 5 + multiply(5,2)
      = 5 + (5 + multiply(5,1))
      = 5 + (5 + (5 + multiply(5,0))
      = 5 + (5 + (5 + (0)))
      = 15
```

Deliverables for Part 1:

1. Walk through the sample code until you understand it. Modify the input arguments and make sure the program still operates correctly. (nothing to hand in).
2. Why was it necessary to save 8 bytes (2 words) on the stack for each call to `multiply()`?
3. In order to follow the RISC-V conventions we had to use register a0 as both the first parameter to the multiply function as well as to pick up the results from calling the function. Explain how the line of code `sw a0, 4(sp)` helps to achieve this.
4. Walk through the program in the debugger with the sample inputs of multiply(5,3). Show all values on the stack each time `jal multiply` is executed. I am looking for

how the stack grows from its base case, through each recursive call, and then how it shrinks back to its initial base case. Example:

fact(3,2)	fact(3,1)	fact(3,0)	return 1	return 2	return 6
????	????	????	????	????	????
	????	????	????	????	
		????	????		

PART 2: Optimizing the Multiply Program

Recall the algorithm for Part 1:

```
#####################################
#ALGORITHM
#  multiply(a,b):
#    if b == 0:
#      return 0
#    else:
#      return a + multiply(a, b-1)
#####################################
```

Given the way the program is written, it will have much different execution characteristics if we call it using multiply(100,2) versus multiply(2,100) even though 100*2 = 2* 100 = 200. In order to optimize this program please modify it as follows:

```
#####################################
#ALGORITHM
#  multiply(a,b):
#    if b > a:
#      return multiply(b, a)
#    else if b == 0:
#      return 0
#    else:
#      return a + multiply(a, b-1)
#####################################
```

Deliverables for Part 2:

Please hand in the source code showing your modifications for part 2 – after you have tested it.

PART 3: Writing your own simple recursive program

For the final part of this lab you will be writing a recursive program to calculate the factorial of a number. You can use the code from part 2 as your template. You may also use the `mul` instruction, which is available in our simulator. For example, `mul t0,t1,t2` results in `t0=t1*t2`. This will for the most part follow the same approach as parts 1 and 2. The basic algorithm to calculate the factorial of a number is as follows:

```
#####################################
#ALGORITHM
#  factorial(a):
#    #base case
#    if a == 0:
#       return 1
#    #recursive case
#    else:
#      return a * factorial(a-1)
#####################################
```

Deliverables for Part 3:

After you write and test your program, please test it with `factorial(4)` at first, and then play with some other values. You should produce output for `factorial(4)` as "The result of 4 != 24"

Then hand in the following:

1. Your commented source code
2. Step through the debugger for `factorial(4)` and show the stack state each time the `factorial()` function is called.
3. EXTRA CREDIT (+5). Although the above is correct a better implementation for the `factorial()` function returns one for the base case when `a==0 || a==1`. Implement this enhancement for extra credit.
4. MORE EXTRA CREDIT: As I stated above you are allowed to use the `mul` instruction for this lab. For extra credit, integrate the multiply function you created in Part 2 instead of using the RISC-V `mul` instruction. This is like recursion on recursion. Be careful with your register assignments and what you need to save/restore from the stack if you choose to take this on.

On blackboard, please submit all deliverables as individual files. Provide one combined .doc or .pdf file for all of the questions I have asked you to answer. Provide files named lab4-part2.s and lab4-part3.s as your solutions to the these parts of the lab.