Lab 4 : MIPS Function Calls

Name:

Chenjie Luo

Sign the following statement:

On my honor, as an Aggie, I have neither given nor received unauthorized aid on this academic work

Chenjie Luo

1 Objective

The objective of this lab is to make you comfortable with MIPS function calls.

2 Pre-requisite

Before starting with this lab, you should be familiar with MIPS function calls as well as MARS.

3 The Stack

As stated in the prelab, there are some situations in which a function does not need to save any data from registers into memory. When this is not the case, however, the most logical place to store the data for later use is on the stack.

The stack is a last in, first out (LIFO) data structure. This makes it ideal for function calls because a function needs only to manage the top of the stack. When another function is called, that function will push more data on the stack. When a function has completed and is ready to return control back to its caller, it will pop data off the stack, leaving the stack in the same form as it was before the function call. This means that functions need to only manage their own section of the stack and do not need to concern themselves with how other functions modify the stack.

To use the stack, simply push and pop data from it. The push operation can be done with the following instructions:

```
1       addi $sp, $sp, -4    #Decrement the stack pointer by 4 to make space
2       sw $r3, 0($sp)       #Store $r3 on the stack.
```

The above code example will push register `$r3` onto the stack. The pop operation can be done with:

```
1       lw $r3, 0($sp)       #Get the value from the stack and place it in $r3
2       addi $sp, $sp, 4     #Increment the stack pointer by 4 to release the space
```

The above code example will pop whatever value is on the stack into register `$r3`. When using the stack, you must match your push and pop operations. If you fail to do this, your function is likely to cause your program to crash.

3.1 Frame Pointer

The *Frame Pointer* (`$fp`) is a compliment to the stack pointer. Often, the stack will have a variable quantity of data pushed onto it during the lifetime of a function call. However a function may need to access data that was pushed onto the stack when it first began execution. Keeping track of how much data has been pushed onto the stack can be tedious so the frame pointer helps manage this. Typically, a frame pointer keeps track of where the stack pointer was **before** the function started and does not move throughout the lifetime of the function. This means that if a function first saves registers to the stack and then pushes a random amount of data onto the stack, it can use the frame pointer to access its register data stored on the stack. Figure 1 shows an example of the stack. Naturally, if you chose to use the frame pointer, you must save its previous value before changing it.

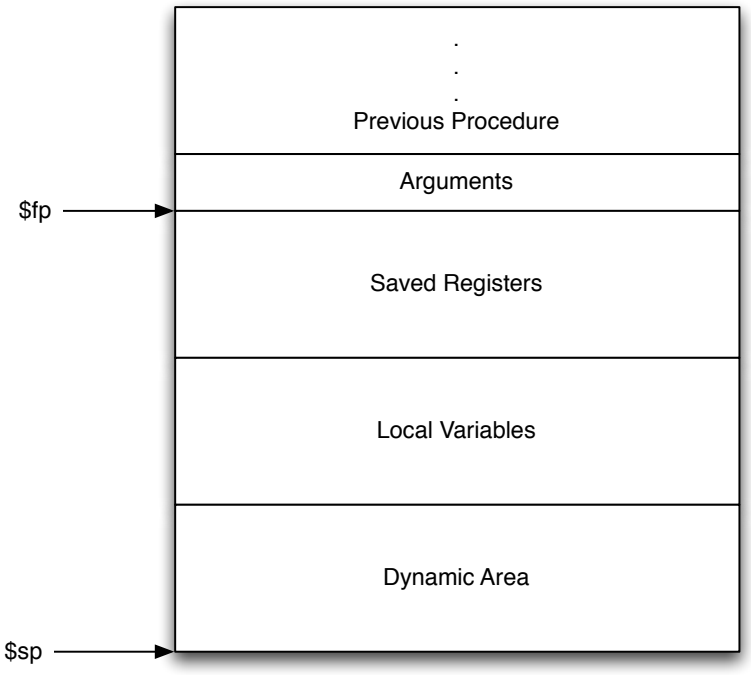

Fig. 1: MIPS Stack Frame

4 Calling functions within functions

Calling a function within a function is no more complex than calling a function under normal circumstances. The key is to utilize the stack to keep track of `$ra`. The calling function needs to save the `$ra` register before jumping and linking to the next function and must restore `$ra` before returning itself. Typically, if a function needs to call other functions, it will save the `$ra` at the start of execution and restore it at the end, making the management of `$ra` easier.

5 Questions

1. Enter the following code into a file named *lab4-1.s* and run it with MARS. Use it to answer the questions below:

```
1         .data
2         msg1: .asciiz "Enter the first number\n"
3         msg2: .asciiz "Enter the second number\n"
4         msg:  .asciiz "The product is "
5         .text
6         .globl main
7         .globl my_mul
8
9         main:
10        addi $sp, $sp, -8   #make room for $ra and $fp on the stack
11        sw $ra, 4($sp)      #push $ra
12        sw $fp, 0($sp)      #push $fp
13
14        la $a0, msg1        #load address of msg1 into $a0
15        li $v0, 4
16        syscall             #print msg1
17        li $v0, 5
18        syscall             #read_int
19        add, $t0, $v0, $0   #put in $t0
20        la $a0, msg2        #load address of msg2 into $a0
21        li $v0, 4
22        syscall             #print msg2
23        li $v0, 5
24        syscall             #read_int
25        add $a1, $v0, $0    #put in $a1
26        add $a0, $t0, $0    #put first number in $a0
27        add $fp, $sp, $0    #set fp to top of stack prior
28        #to function call
29        jal my_mul          #do mul, result is in $v0
30        add $t0, $v0, $0    #save the result in $t0
31        la $a0, msg
32        li $v0, 4
33        syscall             #print msg
34        add $a0, $t0, $0    #put computation result in $a0
35        li $v0, 1
36        syscall             #print result number
37
38        lw $fp, 0($sp)      #restore (pop) $fp
39        lw $ra, 4($sp)      #restore (pop) $ra
40        addi $sp, $sp, 8    #adjust $sp
41        jr $ra              #return
42
43        my_mul:                 #multiply $a0 with $a1
44        #does not handle negative $a1!
45        #Note: This is an inefficient way to multipy!
```

```
46          addi $sp, $sp, -4    #make room for $s0 on the stack
47          sw $s0, 0($sp)       #push $s0
48
49          add $s0, $a1, $0     #set $s0 equal to $a1
50          add $v0, $0, $0      #set $v0 to 0
51          mult_loop:
52          beq $s0, $0, mult_eol
53
54          add $v0, $v0, $a0
55          addi $s0, $s0, -1
56          j mult_loop
57
58          mult_eol:
59          lw $s0, 0($sp)       #pop $s0
60          jr $ra
```

(a) The above code example uses a subroutine to compute a multiplication. It contains an error; what is it? You may wish to step through the program in MARS to find it.

After stepping through the program, I found the code ends at line 41, which is jr $ra. The problem is when we step to "jr $ra", the address stored in $ra has been released. As a result, the program doesn't know which line to return. And the solution is deleting this line because it is not necessary. Besides, because $ra is never used, it is also unnecessary to preserve $ra. We should use syscall to end main function instead. And I used "addi $v0, $0, 10 syscall"

(b) After fixing the code, demonstrate its operation to the TA. ✔

(c) What is the hexadecimal value of `$fp` before the my_mult function call? What is its value during the my_mult function call? What is the `$fp` after main returns?

Before the my_mul function call, the hexadecimal value of $fp is 0. During the function call, value of $fp is 0x7fffeff4. After main returns, It turns 0 again.

2. (a) Write a MIPS program with the following specifications:

- Use the my_mul function in question 1 to create a function which computes factorials: $n! = n \cdot (n-1) \cdot ... \cdot 3 \cdot 2 \cdot 1$.
- Each function must save all variables it modifies.
- Save your file as "*lab4-2.s*", and make sure you comment the code.
- Demonstrate your progress to the TA; ✔

(b) Your function uses a sub-routine to multiply two numbers. Does the order of its arguments affect the speed at which it runs? If so, explain why.

Yes, the reason lies in if we use the larger Multiplicand is used to do subtraction, it will take longer time to operate the iteration.

3. Create a program with the same functionality as that of Question 2 but with the following additional specifications:

- Contains a function to read an integer from the user into `$v0`. All other registers must be the same when it completes. You may use the *syscall*.
- Contains a function to print a message addressed by `$a0`. All registers must be the same when it completes. You may use the *syscall*.
- Contains a function to present the user with a prompt to enter a number. The prompt (message) is contained in `$a0` and the result should be stored in `$v0`. All other registers must be the same when it completes. You may use the previous two functions.
- Save your file as "*lab4-3.s*", and make sure you comment the code.
- Demonstrate your progress to the TA; ✔.

6 Deliverables

Submit the following:

- A completed copy of this lab.
- All source code files created in this lab (with comments).