# Lab 4 (All Sections) Prelab: MIPS Function Calls

**Name:**

Kraig Orcutt

**Sign the following statement:**

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

Kraig Orcutt

## 1 Objective

The main objective of this lab is to understand function calls in MIPS. Before proceeding with this lab, you should be familiar with calling functions in MIPS (Chapter 2 of your textbook).

## 2 Function Calls

In the previous lab, you learned how to do loops and *if-then-else* constructs. This allows you to create nearly every program you could conceive. However, often we have blocks of code that need to be used in multiple points throughout the program. Instead of replicating code multiple times, the better choice is to make it into a function. Then, whenever that block of code is needed, the program needs only to jump to the function, and then jump back when done. This is often referred to as a function call, and the block of code which issued the jump is often referred to as the caller.

When you call a function, there must be a mechanism to return execution back to the caller. Without this mechanism, there would be no way for a function to know which block of code called it. MIPS provides an instruction which will save the PC before it jumps into a function. This is done through the jump-and-link instruction:

```
jal Function  #Jump and link
```

The `jal` instruction will immediately jump to the provided address like the `j` instruction, but unlike the `j` instruction, `jal` will also save the address of the instruction following the `jal` within `$ra`. Then, when the function is done doing its work, it can then jump to `$ra` register with `jr`. For example:

```
1       GetDimensions:      #Get the dimensions of a box from two corners.
2       #$a0 and $a1 are the coordinates of near corner (x, y)
3       #$a2 and $a3 are the coordinates of far corner (x, y)
4       sub $v0, $a2, $a0   #compute width
5       sub $v1, $a3, $a1   #compute height
6       jr $ra
```

## 3 Register Usage

MIPS has a convention of typical register usage in terms of function calls. You do not have to conform entirely to this convention, but it makes designing the calls much easier. Typically, the `$a` registers are used as function arguments for a function, `$t` registers are used for temporary values, and `$s` registers are used for saved values. Table 1 summarizes the typical convention.

Tab. 1: MIPS Calling Convetion

| Name | Usage | Saved? |
|---|---|---|
| `$zero` | 0 | N/A |
| `$v0-$v1` | Return Values | NO |
| `$a0-$a4` | Arguments | NO |
| `$t0-$t9` | Temporary Values | NO |
| `$s0-$s7` | Saved Values | YES |
| `$gp` | Global Pointer | YES |
| `$sp` | Stack Pointer | YES |
| `$fp` | Frame Pointer | YES |
| `$ra` | Return Address | YES |

This means that if a function does not need more than 10 temporary variables, nor calls any other functions, then it does not need to save any registers at all. It can simply use the temporary registers for its local variables and ignore the others. Conversely, if a block of code is going to make a function call, it cannot rely on the values of the temporary registers remaining the same after the call. In a pinch, the function can use the return value registers as temporary variables as well, so long as they contain the proper return value at the end.

## 4 Questions

In the following questions do not overwrite any of the saved registers listed in Table 1. Use temporary and return registers to store values.

1. Write the MIPS instructions for the following c code. Assume that the arguments are stored in `$a0` onward in the order they are presented in the function.

```
int add(int a, int b) {
  return a + b;
}
```

```
add:
add $v0, $a0, $a1
jr $v0
```

2. Write the MIPS instructions for the following c code:

```
int evaluation(int a, int b, int c, int d){
  int sum = a + b + c  + d;
  int first = a - b;
  int last = c - d;

  return sum - first + last;
}
```

```
evaluation:
add $t0, $a0, $a1
add $t1, $a2, $a3
add $t0, $t0, $t1       #total sum
sub $t1, $a0, $a1       #first
sub $t2,$a2, $a3        #last
add $t1, $t1, $t2
sub $v0, $t0, $t1
jr $v0
```

3. Consider the instruction, **jal Proc**, which is located at address 0x00401024, what is the value of `$ra` upon entering the the function `Proc`?

0x00401028

4. Consider the following high-level recursive procedure:

```
1  int f(int n, int k)
2  {
3    int b;
4
5    b=k+2;
6    if (n==0) b = 8;
7    else b = b + 4 * n + f(n-1,k+1);
8
9    return b + k;
10 }
```

Translate the high-level procedure `f` into MIPS assembly language.

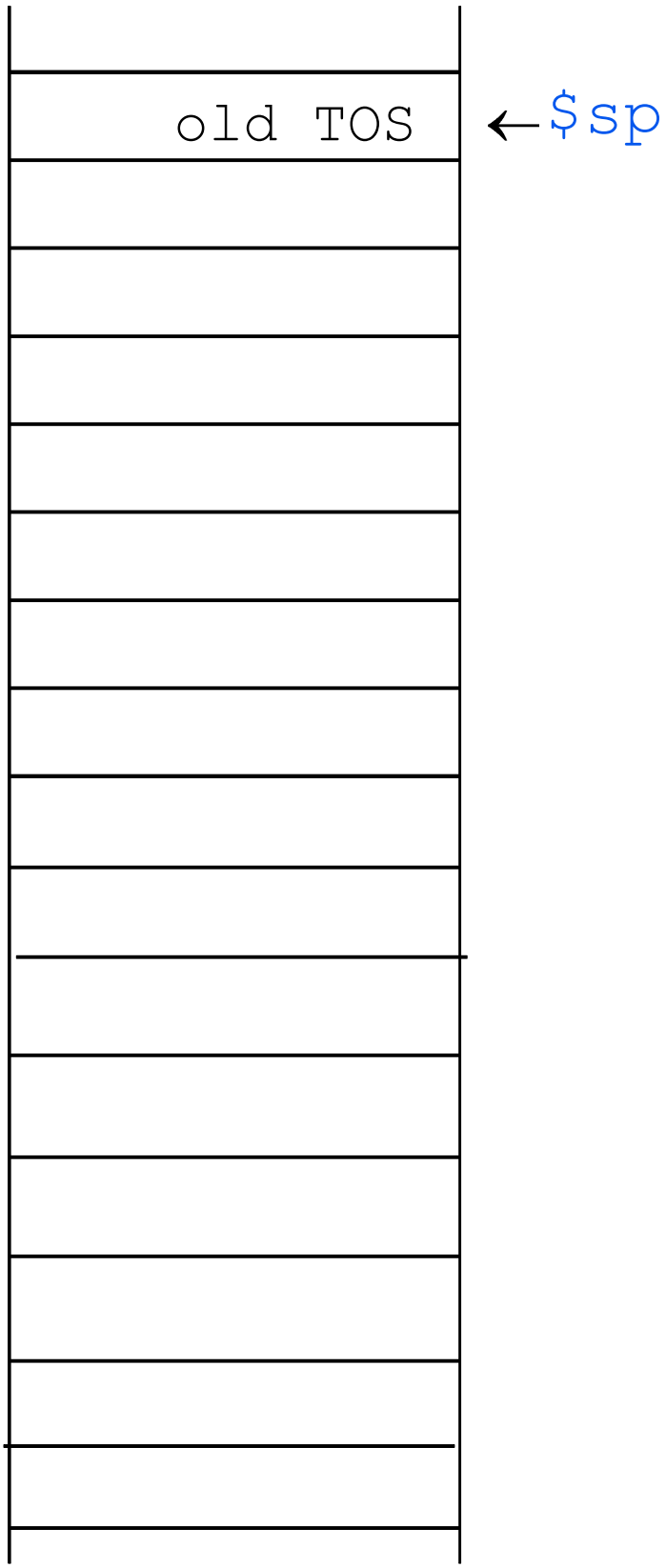
- Pay particular attention to properly saving and restoring registers across procedure calls;
- Use the MIPS preserved register convention;
- Clearly comment your code;
- Assume that the procedure starts at address `0x00400100`;
- Keep local variable `b` in `$s0`

```
f:
li $s1, 4
addi $sp, $sp, -12      # save space for 3 .word storage
sw $ra, 8($sp)          #store return address into sp0
sw $a0, 4($sp)          #store n into sp4
sw $a1, 0($sp)          #store k into sp8
beq $a0, 1,  end       # if n = 0 go to end
subi $a0, $a0, 1
addi $a1, $a1, 1
jal f

recur:
lw $t1, 4($sp)         # get n into $t1
lw $t2, 0($sp)        # get k into $t2
lw $ra, 8($sp)
addi $sp, $sp, 12
addi $s0, $t2, 2      # calculate b
mul $t1, $t1, $s1     # calculate n * 4
add $s0, $v0, $s0   # b = F(n-1, k+ 1) + b
add $s0, $s0, $t1    # b = b + 4*n
add $v0, $s0, $t2    # return b+k
jr $ra

end:
```

Step through your program by hand for the case of `f(2,4)`. Draw a picture of the stack after the completion of the last recursive call. Write the register name and data value stored at each location in the stack. Assume that when `f` is called, `$s0=0xABCD` and `$ra=0x400004`. Use the last page of this document as a guideline. You may submit a hard copy of this part of the prelab.

What is the final value of `v0`?

```
0x30
```


old TOS
←$sp