Introduction

This is a group assignment on *calling* and *defining* simple functions in MIPS assembly. Parameter passing, stack frames, and return values are all explored.

Assignment Goals

Learning Outcomes After completing this group assignment, each student is expected to be able to

- **Trace** MIPS assembly code across function call and return.
- Use `$a*` and `$v*` registers according to MIPS calling conventions.
- **Push**, **pop**, and access values in the *activation record* of a function.

Procedure

Get out paper for a *single* turn-in at the end of class. Copy enough of each question so that the paper could stand alone as a study guide.
Assign (Least-recently Held) Roles: *Manager*, *Recorder*, *Reflector*, *Speaker*. Everyone should help the whole team contribute and manage time.
Answer these questions:

1. (a) Assume `int sum(int a, int b, int c, int d)` is translated to MIPS assembly. The following block of code, in `main`, calls `sum`.
   What are the (decimal) values of the four (4) parameters when control is transfered to `sum`?

```
li   $a0, 101
li   $a1, 0x19
sub  $a2, $a0, $a1
li   $a3, 0x111
jal  sum

mov  $a0, $v0
li   $v0, 1
syscall
```

   (b) Assuming `sum` returns the sum of its parameters, what is the output of the above snippet? Be *exact* with the formatting.

   (c) Implement `sum` in Java as simply as you can. What other functions does your Java function *call*?

   (d) Given what you know about MIPS calling conventions, implement `sum` in MIPS, again *as simply as you can*.

2. Playing with the stack:

   (a) Write the two lines we have used to push **`$ra`** on the stack.

   (b) Compare what happens in memory if the two lines in the push above were *reversed*.

   (c) What does that tell you about where the **`$sp`** points (relative to the topmost item on the calling stack) in the way we are using it?

   (d) Write the two lines we use to pop the top of the calling stack into **`$ra`**.

3. Consider the following code

```
1    .data
2  prefix:
3    .asciiz "["
4  suffix:
5    .asciiz "]"
6  eoln:
7    .asciiz "\n"
8
9    .text
10   .globl main
11 main:
12   li    $t0, 0x10
13
14   # while $t0 >= 0
15 test:
16   bltz  $t0, afterWhile
17
18   # println [ i ]
19   li    $v0, 4
20   la    $a0, prefix
21   syscall
22
23   li    $v0, 1
24   move  $a0, $t0
25   syscall
26
27   li    $v0, 4
28   la    $a0, suffix
29   syscall
30
31   li    $v0, 4
32   la    $a0, eoln
33   syscall
34
35 decrement:
36   subi  $t0, $t0, 1
37   j     test
38
39 afterWhile:
40   li    $v0, 10
41   syscall
```

   (a) What is the meaning of the **j** instruction in line 37? How is it different from the **bltz** in line 16?

   (b) What is the *output* of the code?

   (c) What is **$t0** being used for here? What are the largest and smallest values it will hold while this is running?

   (d) Dr. Ladd split `suffix` and `eoln` into two strings. Give pros and cons to this decision.

4. Consider the operation in the previous question. Imagine rewriting it as a *recursive* function, `printEm`. With the same **.data** segment, `main` is rewritten to the following:

```
9    .text
10   .globl main
11 main:
12   li    $a0, 0x10
13   jal   printEm
14
15   li    $v0, 10
16   syscall
```

   (a) Write the Java function signature of the `printEm` function. How do you know the number/order of parameters.

   (b) Implement `printEm`, in Java, so the rewritten `main` produces the same output as question 2.

   (c) Translate the recursive `printEm` into MIPS. You probably want to start with the prologue/epilogue, then the base case, then setting up the recursive call. Feel free to keep the code as simple as possible (but no simpler).