HW1-report

1. What is the usage of $zero? What happens if you execute `addi $zero, $zero, 5` ? (5%)

1-1: What is the usage of $zero?

The $zero register in MIPS is a special register that is hardwired to always contain the value 0.

In computer organization, there are four main principles:

1. Simplicity favors regularity;
2. Smaller is faster;
3. Good design demands compromise;
4. Make the common case fast

It's useful for a number of operations where we want to use the value 0 or compare another register to 0. Since $zero is common use case, it is the implementation of `Make the common case fast` (This way we don't have to waste another register, or any memory, holding the value.)

1-2: What happens if you execute `addi $zero, $zero, 5` ?

Essentially, any attempts to write to the $zero register will be ignored. This is because the $zero register is read-only and will always contain the value 0, regardless of any operations you perform on it.

I also try the instruction in MARS

```
.data
        output_msg:     .asciiz "The result of modified $zero is "
.text
.globl main

main:
        li      $v0, 4
        la      $a0, output_msg
        syscall

        addi    $zero, $zero, 5
        move    $a0, $zero
        li      $v0, 1
        syscall

# exit the program
        li              $v0, 10
        syscall
```

and the result will be: `The result of modified $zero is 0`

2. How to use the stack to ensure that the value of each register is correctly saved when executing a recursive function? (5%)

There are three main stages:

1. before calling the recursive function

We first save the values of the registers we want to preserve onto the stack. This is done using the `sw` (store word) instruction. The usual convention is to store the return address register `($ra)` and any argument or temporary registers (`$a0` - `$a3` , `$t0` - `$t9`) that we are using and want to preserve.

For example:

```
addi $sp, $sp, -8   # make room on the stack
sw   $ra, 4($sp)    # save $ra on the stack
sw   $a0, 0($sp)    # save $a0 on the stack
```

2. We then call the recursive function with the jal (jump and link) instruction.

3. When the recursive function is done and we want to return, we must restore the saved registers from the stack. we do this using the `lw` (load word) instruction.

For example:

```
lw   $a0, 0($sp)    # restore $a0 from the stack
lw   $ra, 4($sp)    # restore $ra from the stack
addi $sp, $sp, 8    # clean up the stack
jr   $ra           #  return to the return address
```

3. What was the most challenging part for you in this homework? (10%)

In this assignment, I found the process of accurately preserving and reestablishing the state of a function during recursion to be particularly challenging.

Initially, when working on the `fibonacci.s` script, I was confounded by the task of retaining two parameters while having only one argument at my disposal. This was due to the structure of the main function, which solely passed `n`, despite the need to restore both `n-1` and `n-2` when calculating the Fibonacci sequence.

Subsequently, I conceived a solution in which I would save `n-1` and compute the sum of `n-1` and `n-2` within the loop. This approach circumvented the need to retain an additional argument in the Fibonacci procedure, thereby resolving the issue and enhancing my understanding of recursive function implementation.

Reference

1. What is the use of a $zero register in MIPS? (https://stackoverflow.com/questions/32233570/what-is-the-use-of-a-zero-register-in-mips)
2. Zero register(Wikipedia) (https://en.wikipedia.org/wiki/Zero_register)
3. What is the use of a $zero register in MIPS? (https://stackoverflow.com/questions/32233570/what-is-the-use-of-a-zero-register-in-mips)