

# CSCE 22104

## Lab Report

---

Brent Marcus Orlina

ID: 011019116

Lab Section 001

Lab 5

## Introduction

This lab's goal was to learn how to build a recursive function in MIPS assembly. The recursive function implemented was a function giving the  $n^{th}$  fibonacci number. The  $n^{th}$  fibonacci number can be expressed as the recurrence relation

$$f_n = f_{n-1} + f_{n-2}$$

where  $f_0 = 0$  and  $f_1 = 1$ . This can be expressed into C code as

```
1 int fib(int n) {  
2     if(n > 1) return fib(n - 1) + fib(n - 2);  
3     if(n == 1) return 1;  
4     if(n == 0) return 0;  
5     return -1;  
6 }
```

Listing 1: An implementation of a fibonacci function in C.

Note that the C implementation, as shown in listing 1 returns a  $-1$  for inputs less than 0, expressing that the fibonacci function is undefined for  $n < 0$ . The program built should also prompt and read an integer from user input, allowing the user to choose the  $n^{th}$  fibonacci number to be printed.

## Approach

Since printing a prompt, reading an integer from user input, and printing the result of the fibonacci function is trivial, describing its implementation will be skipped. The implementation of the fibonacci function will be the main focus.

```
1  .text
2  ...
3  fib:
4      blt $a0, $zero, fib_ret_negative
5      beq $a0, $zero, fib_ret_zero
6      li $t0, 1
7      beq $a0, $t0, fib_ret_one
8
9      ... # handle recursive case
10
11     fib_ret_negative:
12         li $v0, -1
13         jr $ra
14
15     fib_ret_zero:
16         li $v0, 0
17         jr $ra
18
19     fib_ret_one:
20         li $v0, 1
21         jr $ra
```

Listing 2: The base cases for the fibonacci function in MIPS assembly.

Listing 2 shows the implementation for handling the base cases of the fibonacci function, as shown in listing 1. The register `$a0` has the value of  $n$ . Line 4 branches to `fib_ret_negative` if  $n < 0$  and returns a  $-1$ . Similarly, line 5 branches to `fib_ret_zero` if  $n = 0$  and a  $0$ . Both of these branches can be done using only one instruction by the use of the `$zero` register. However, to check if  $n = 1$ , the program must load a  $1$  to a register and compare `$a0` to that register. In the program, it loads a  $1$  to the `$t0` register. Line 7 branches to `fib_ret_one` if  $n = \$t0$ , or  $n = 1$ , and returns  $1$ . If all the branches fail, then it must mean  $n > 1$ , thus the program can continue and handle the recursive case.

```

1  .text
2  ...
3  fib:
4  ... # testing for base cases
5  # create stack
6  addi $sp, -8
7  sw $ra, 4($sp)
8  sw $a0, 0($sp) # we're going to store n and the result of fib(n-1) here
9
10 # fib(n-1)
11 addi $a0, $a0, -1
12 jal fib
13
14 lw $a0, 0($sp) # we load the original n passed
15 sw $v0, 0($sp) # then store the result of fib(n-1)
16
17 # fib(n-2)
18 addi $a0, $a0, -2
19 jal fib # remember that fib(n-2) is at lv0 after this instructoin
20
21 # restore stack
22 lw $t0, 0($sp) # load fib(n-1)
23 lw $ra, 4($sp)
24 addi $sp, 8
25
26 # return fib(n-1) + fib(n-2)
27 add $v0, $v0, $t0
28 jr $ra
29
30 ... # base cases

```

Listing 3: Handling the recursive case for the fibonacci function in MIPS assembly.

Listing 3 shows the recursive case for the fibonacci function. First, it creates a stack for two words, one for the return address, and another to store  $n$  and the result of calling `fib(n-1)`. The program needs to store  $n$  in the stack since it is currently in the `$a0` register, which can change after the first recursive call, since that call might also do a recursive call. Similarly, the result of `fib(n-1)` must be stored in the stack since the function will return its result in the register `$v0`, which can also change.

After calling `fib(n-1)`, as shown in lines 11 and 12,  $n$  is loaded back into register `$a0` since there is no guarantee that the `$a0` register has remained the same. Then, the result of `fib(n-1)`, currently in the `$v0` register, is stored into the stack, where  $n$  was, since `$v0` will change after calling `fib(n-2)`. The program can store the result into memory where  $n$  previously was since  $n$  has already been loaded into the `$a0` register, and it will no longer be used after the second recursive call.

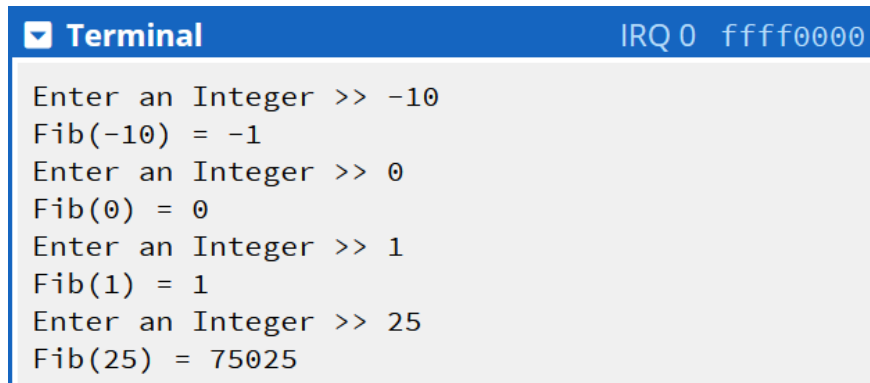
The recursive call `fib(n-2)` is called and its result is in the `$v0` register. All of the needed calls have been called and it is time for the function to return its result. The stack is restored by loading the result of `fib(n-1)` onto the temporary register `$t0` and loading the return address back to the `$ra` register. Finally, the stack pointer is bumped back up to its original position.

The result of `fib(n-1)`, now stored in `$t0`, and `fib(n-2)`, still stored in `$v0`, is added together to `$v0` and the function finally returns. Thus, the fibonacci function is complete.

## Experimentation

The fibonacci function was tested by first testing the base cases of the function,  $n < 0$  and  $n = 0, 1$ , and verifying that the function had the correct outputs. Then, the recursive case was tested and verifying that the function's output was correct through the website [oeis.org](http://oeis.org), which lists the  $n^{th}$  fibonacci number up to 40. The  $n$  chosen for testing was 25.

## Results & Discussion



```
Enter an Integer >> -10
Fib(-10) = -1
Enter an Integer >> 0
Fib(0) = 0
Enter an Integer >> 1
Fib(1) = 1
Enter an Integer >> 25
Fib(25) = 75025
```

Figure 1: The output of the program implemented in MIPS assemble when ran 4 times, with the cases of  $n$  equal to  $-10$ ,  $0$ ,  $1$ , and  $25$  being tested.

The fibonacci function works as expected. Figure 1 shows the program being ran four times. The first case tests the behavior of the function when given a negative number, which is undefined. For the fibonacci function implemented, the behavior chosen was to output a  $-1$ , signifying that it is an invalid input. The next two cases,  $n = 0$  and  $1$ , tests teh base cases of the function. It correctly outputs  $0$  and  $1$  for each input respectively. The last case calculates the the  $25^{th}$  fibonacci number, which correctly outputs  $75025$ .

## Conclusions

The fibonacci function implemented in MIPS assembly works correctly. The knowledge learned from this lab was learning how to implement a recursive function, specifically recognizing the base cases of a recursive function and handling the creation of a stack to support recursion. It was also learned how to optimize the stack size by recognizing which variables are unused after a certain point, allowing the reuse of the variable's spot in the stack.