# CMSC 133

## Introduction to Computer Organization, Architecture and Assembly Language

## **Laboratory Report**

| | |
|---|---|
| Names of all Authors and Student IDs | Kienth Matthew B. Tan |
| Lab No. | 1 |
| Task/Step No. | 1 |
| Lab Title | MIPS |
| Email Addresses of all Authors | kbtan@up.edu.ph |
| Date of Document Issue | 03/25/22 |
| Document Version Number | 1.0 |
| Address of Organisation Preparing this report: | Department of Computer Science<br>University of the Philippines Cebu<br>Gorordo Avenue, Lahug, Cebu City<br>Tel: (032) 232 8185 |
| Course Name | CMSC 133 Introduction to Computer Organization, Architecture and Assembly Language |
| Course Units | 3 |

**Summary**.

The Laboratory activity is all about putting into action what we learned in our previous lessons and video lectures. It aims to help us learn the basics of MIPS assembly language from basic programs while task by task progressing into more complex problems.

The First task which will be featured in this laboratory report is on how to find the summation of a specified array. The task is actually quite simple as there are comments within the file that will help us to discover on what procedures we have to use. Of course, it is not expected that there will also be prepared comments in the next task. Through this task, I learned through practice on how to use branching instructions such as beq where we only have to branch if for example *i* would be equal to *N*. The task also tackled on how we can loop a predefined array in MIPS assembly.

In approaching my method to solving this task, I first created a C program that would solve the laboratory problem. Things would then be much simpler after that as I only had to translate my C code into MIPS assembly code. Paired with my C code and the pre-placed comments in the activity, I was able to successfully create a MIPS code that will solve the problem which I solved through looping all the elements in the array.

# TABLE OF CONTENTS

# 1 The Lab Problem

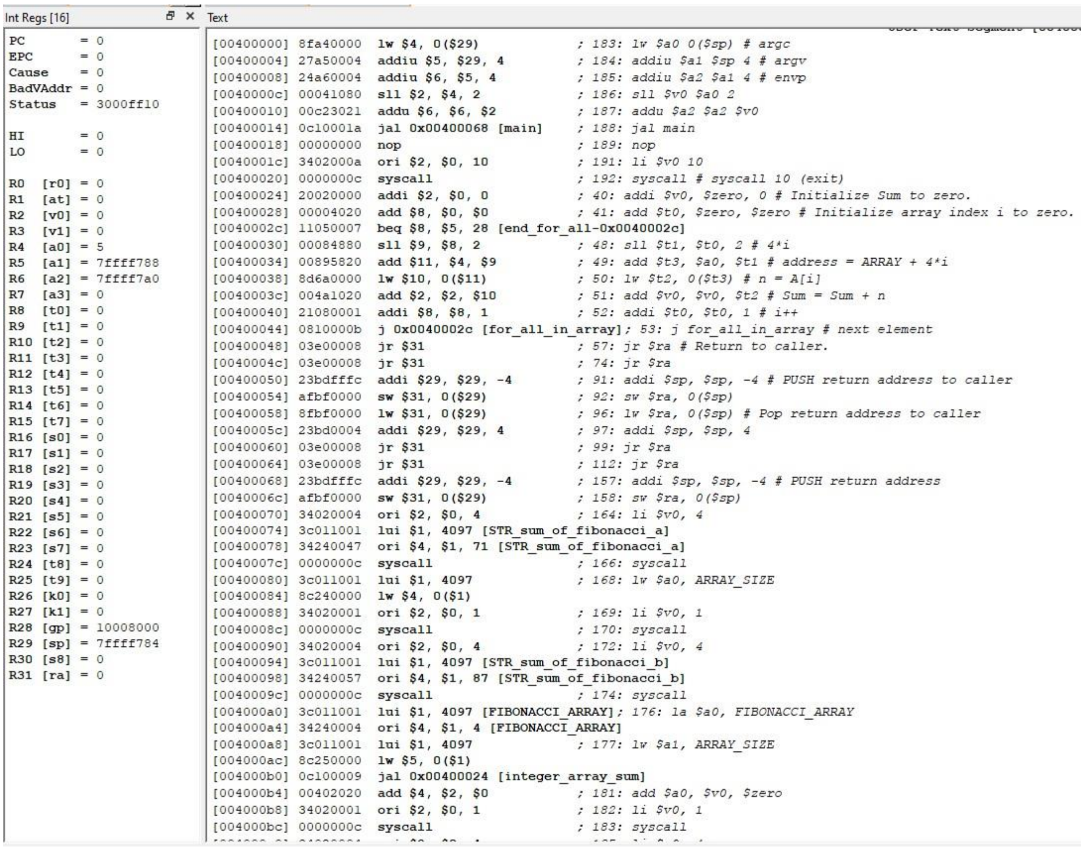
The problem to be solved in this first task of the laboratory activity is to create a subroutine wherein it has to iterate through a set of numbers, specifically the Fibonacci array and to finally add them up. This means that through the iteration or loop, we have to create an instance where it adds the previous iteration number and the current iteration number until the loop is satisfied. The sum would then be returned to the caller to be printed in the console.

# 2 The Assembly Source Code

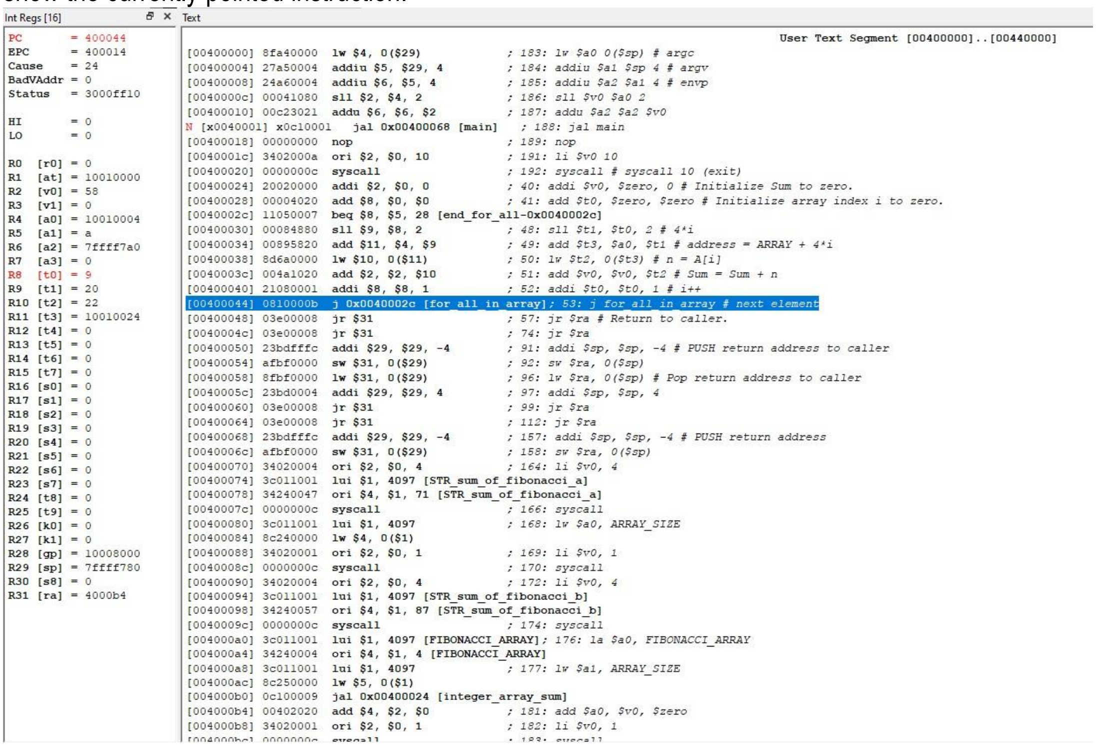
```
integer_array_sum:

DBG:    ##### DEBUGG BREAKPOINT ######

    addi    $v0, $zero, 0           # Initialize Sum to zero.
    add $t0, $zero, $zero   # Initialize array index i to zero.

for_all_in_array:

    #### Append a MIPS-instruktion before each of these comments
    # $t1 = offset, $t2 = n, $t3 = address

    beq $t0, $a1, end_for_all   # Done if i == N
    sll $t1, $t0, 2             # 4*i
    add $t3, $a0, $t1           # address = ARRAY + 4*i
    lw  $t2, 0($t3)             # n = A[i]
        add $v0, $v0, $t2       # Sum = Sum + n
        addi $t0, $t0, 1        # i++
    j   for_all_in_array        # proceeds to the next element

end_for_all:

    jr  $ra                     # Return to caller.
```

# 3 Screenshots

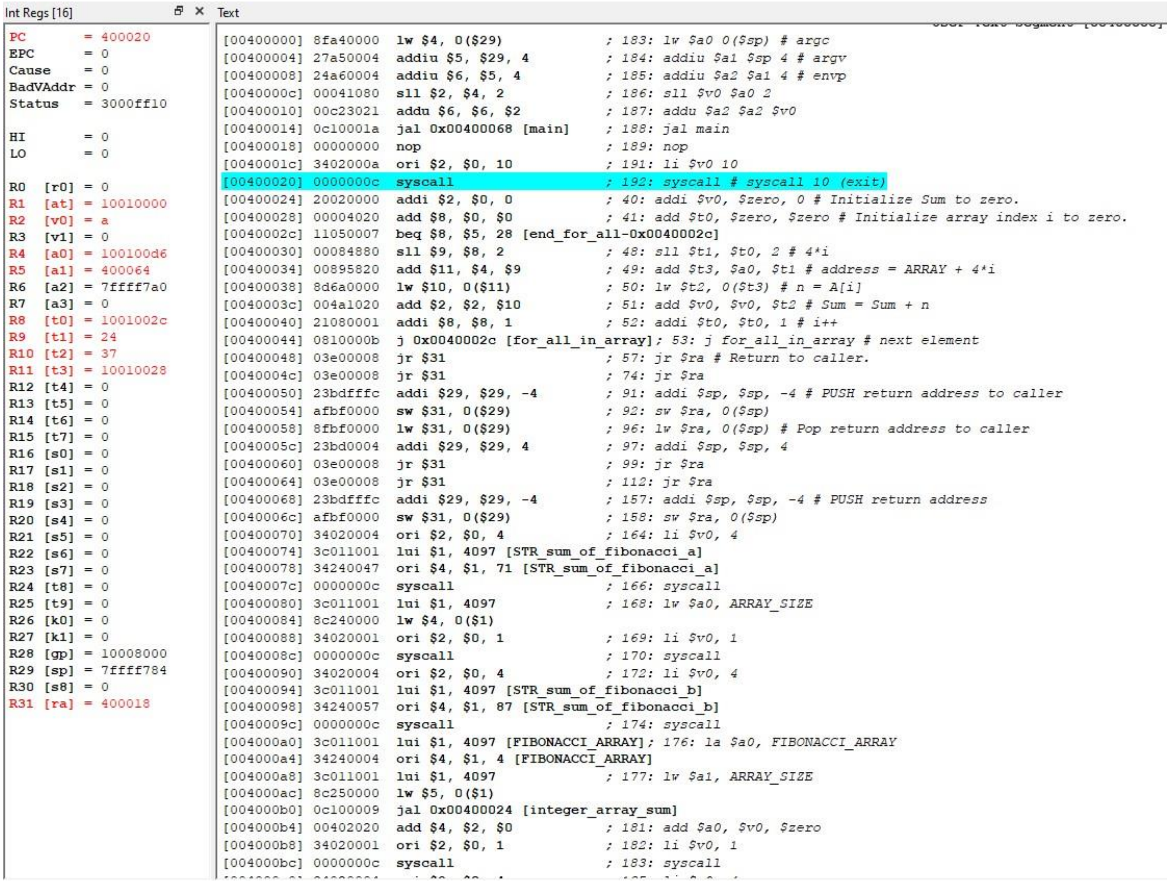
A. **Start of Execution** – Show the current state of the registers, code/program/text segment and data and stack segments in the memory after the program is loaded. Show the initial values of the registers. Show that your program and data has been loaded in the code and the data segments respectively.




B. **Middle of Execution** – Show the current state of the registers and data and stack segments mid-execution. Also show the currently pointed instruction.



C. **End of Execution** – Show the current state of registers and data and stack segments after the execution of the last instruction of your program. Also show the currently pointed instruction after the execution of the last instruction.




## 4 Learning & Insights

Through this laboratory activity, I learned how to check what values goes into the registers, how to load values from the registers, and also in how to manipulate the values within it. I also learned how to translate higher-level programming languages like C code into MIPS assembly as I created a program in C before solving it using MIPS. Both are quite similar in solving the task but MIPS is much more integral in trying to see how the values change step-by-step through the registers.

Through checking out the main function, I also learned that to exit the program we had to store 10 into $v0 which probably means that it is a code for exiting the program when the syscal detects it. This is useful to know in the future as my programs probably won't work if I don't do so.

In looping the array, since $a0 points to the address of the first integer in the array, we had to shift left by two bits to access the next element in the array as each integer in the array is composed of 4 bytes or 1 word. This is important to know since it actually composes of two lines to access the next array which is shifting it by 4 bytes through sll or 4*i where i is the current index and then adding it to $a0. This means that progressively, to access the second element of the array the offset would be 4 then the third would be 8, the fourth element would be 12, etc. Overall, this means that to access the next element of the array, we need to always shift left by 4 bytes since an integer in MIPS is 1 word or 4 bytes.

# 5 Comparison to a High-Level Implementation

Here is an implementation of the lab problem in the language C.

```
int sum(int arr[], int n)
{
    int sum = 0; // initialize sum

    for (int i = 0; i < n; i++)
    sum += arr[i];

    return sum;
}

int main()
{
    int arr[] = {1,1,2,3,5,8,13,21,34,55};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("The sum of the first %d Fibonnaci numbers is %d", n, sum(arr, n));
    return 0;
}
```

Similar to our assembly source code, the main acts as the caller and the function sum acts as the callee. We can also see that the iteration is much more understandable and much shorter as we don’t have to add instructions such as sll or shift-left-logical and add to access the current and next element of the array. The loops are also slightly different as in C, all the necessary procedures to loop is within a single line while in MIPS we had to use several lines of code to initialize the value, increment the *i,* and whether to branch when a condition is met or not .

The way of printing the output is quite different, while in C we can simply write printf and the output we want to print, the way we had to print in MIPS is to use instructions such as load immediate, load address, and syscall to pass control to the system and do the printing for you.

# 6 Conclusion

Overall, this laboratory activity helped polish what I learned from the lecture videos provided, it enabled me to understand MIPS assembly by hand. Similar to other programming languages, I realized that it is important to understand MIPS Assembly as it will assist me in future projects especially those that includes the specific use of addresses, memories and registers. This particular task helped in discovering how the program moves data around, how it transforms that data, and in how we can make programming decisions based on what we saw. I am looking forward to what I will learn and would then be capable to do after solving the other future tasks in this laboratory activity.