**Department of Computer Engineering**

BLG 351E  
Microcomputer Laboratory Experiment Report

Experiment No : 4

Experiment Date : 06.11.2017

Group Number : Monday - 8

Group Members :

|  |  |  |
| --- | --- | --- |
| **ID** | **Name** | **Surname** |
| 150150114 | EGE | APAK |
| 150140066 | ECEM | GÜLDÖŞÜREN |
| 150150701 | YUNUS | GÜNGÖR |
| Click here to enter text. | Click here to enter text. | Click here to enter text. |

Laboratory Assistant : Ahmet Arış

# Introduction

In this experiment, we gained more experience with the MSP430 Education Board and its assembly language. The main aim of this experiment was to enhance information about function calls and we got familiar to the usage of the stack.

Before the experiment, we studied the documents that are provided for us in Ninova and learnt differences between CALL and JMP. As it is described in the MSP430\_Instruction\_Set document, CALL instruction gives us the opportunity of making subroutine calls to an address and all of the addressing modes can be used with this instruction.

In Program Counter register (PC), there is related addresses of each instruction, so PC holds the addresses of these instructions. If a function call occurs, then the address of the next instruction is pushed to the stack and when returning from function call, the address will be popped from stack. After popping the address from the stack, program continues form that address. JMP instruction is used when jumping unconditionally is our goal.

In the first part of the experiment, we wrote the code on CCS that is provided for us in the experiment sheet and we observed the behavior of this code.

In the second part of the experiment, we implemented the function whose name is Adder. The aim of the Adder function is to calculate the sum of two integers.

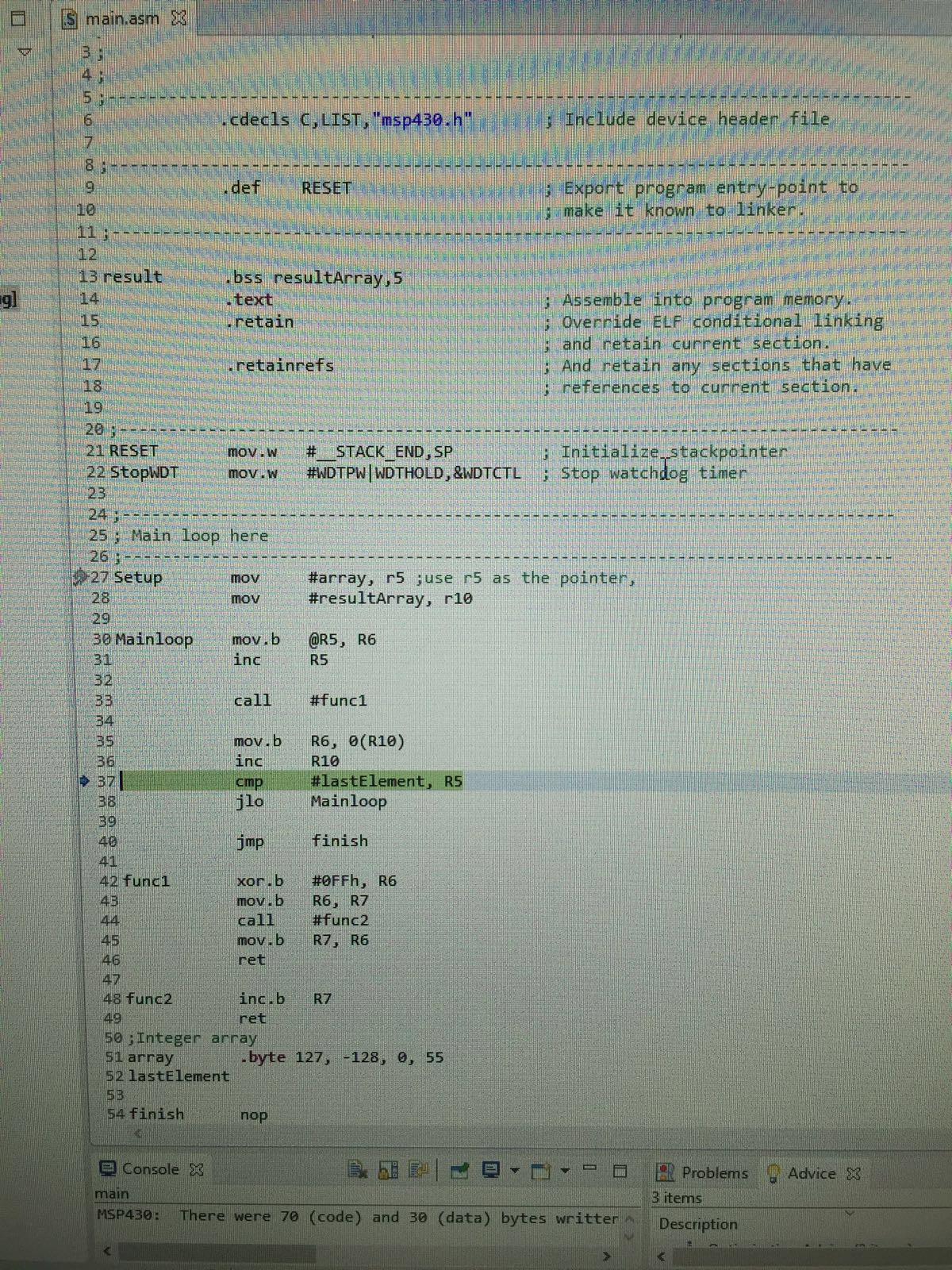
In the third part of the experiment, we implemented iterative version of the Fibonacci algorithm by using the adder implementation in the previous part.

In the fourth part of the experiment, we implemented recursive version of the Fibonacci algorithm by using the adder implementation in the 2nd part of the experiment.

# Experiment

## Part 1 – Basıcs of a Subroutıne Call

In this part of the experiment, we wrote the code that is provided for us in the experiment on the Code Composer Studio(CCS) and investigated the code. After we fully understood what this code does, we run the given code and we filled the table step by step. The code can be observed from the Picture 2.1.1 below.



Picture 2.1.1

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Code | PC | R5 | R10 | R6 | R7 | SP | Stack Content |
| 1 | mov #array,r5 | C00E | C038 | 0200 | 007F | 1BFF | 0400 | Empty |
| 2 | mov #resultArray, r10 | C012 | C038 | 0200 | 007F | 1BFF | 0400 | Empty |
| 3 | mov.b @r5,r6 | C014 | C038 | 0200 | 007F | 1BFF | 0400 | Empty |
| 4 | inc r5 | C016 | C039 | 0200 | 007F | 1BFF | 0400 | Empty |
| 5 | call #func1 | C028 | C039 | 0200 | 007F | 1BFF | 03FE | C01A |
| 6 | xor.b #0FFh, r6 | C02A | C039 | 0200 | 0080 | 1BFF | 03FE | C01A |
| 7 | mov.b r6,r7 | C02C | C039 | 0200 | 0080 | 0080 | 03FE | C01A |
| 8 | call #func2 | C034 | C039 | 0200 | 0080 | 0080 | 03FC | C01A,C030 |
| 9 | inc.b r7 | C036 | C039 | 0200 | 0080 | 0081 | 03FC | C01A,C030 |
| 10 | ret | C030 | C039 | 0200 | 0080 | 0081 | 03FE | C01A |
| 11 | mov.b r7,r6 | C032 | C039 | 0200 | 0081 | 0081 | 03FE | C01A |
| 12 | ret | C01A | C039 | 0200 | 0081 | 0081 | 0400 | Empty |
| 13 | mov.b r6,0(r10) | C01E | C039 | 0200 | 0081 | 0081 | 0400 | Empty |
| 14 | inc r10 | C020 | C039 | 0201 | 0081 | 0081 | 0400 | Empty |

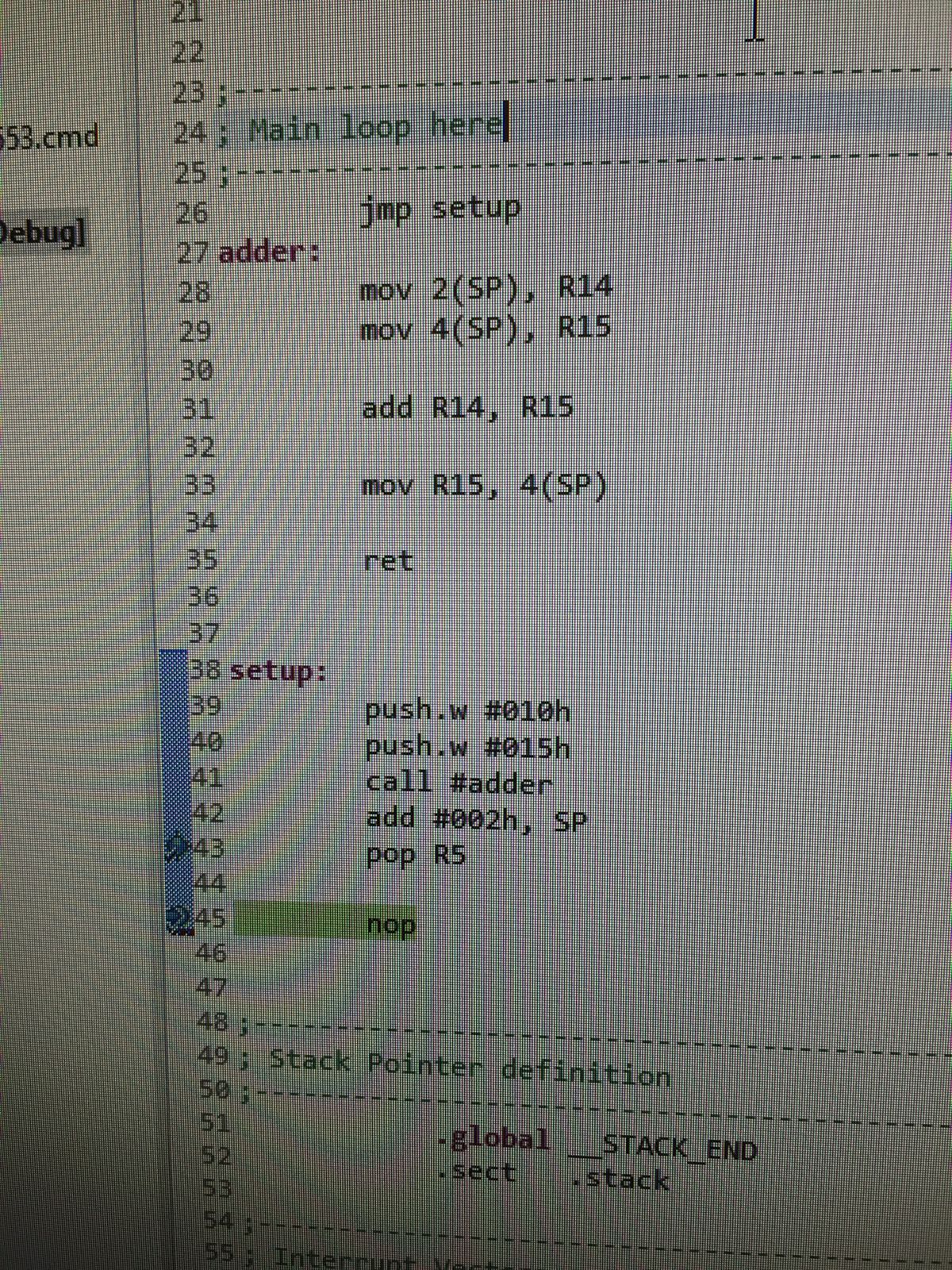
Table 2.1.1

PC will increased depending on the size of the bytes in memory since it stores the starting address of each instruction in the given row. These registers are general purpose registers. SP stands for stack pointer and it holds the address of the top element in the stack. Each CALL instruction, pushes the address of the next instruction to the stack and jumps to the subroutine. When returning form subroutine, the top value of the stack is popped from the stack and this popped value(address of an instruction) is placed into PC.

The problem of this code is the fact that it counts on R6 and R7 registers is available. But in some senerios, R6 and R7 may be used by main program in order to store the address of the arrays. We could prevent the loss of any necessary information while using nested function by using stack.

## Part 2

In this part of the experiment, we implemented an adder function.

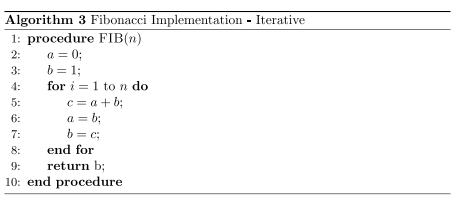


Picture 2.2.1:Adder function

This code firstly jumps to the setup label. Then it pushes the hexadecimal values #0010 and #0015 to the stack. After pushing these to values to the stack, it makes a subroutine call to adder function. Adder function stores the last pushed element (#0015) in R14 and previous pushed element(#0010) in R15. Then adds these two numbers and stores it in R15. Then the result is pushed to the stack and returned 42nd line. We added #0002 to SP so that stack pointer points to our result. Finally, we popped the top element on the stack and we finished calculating the addition of these two numbers.

## Part 3

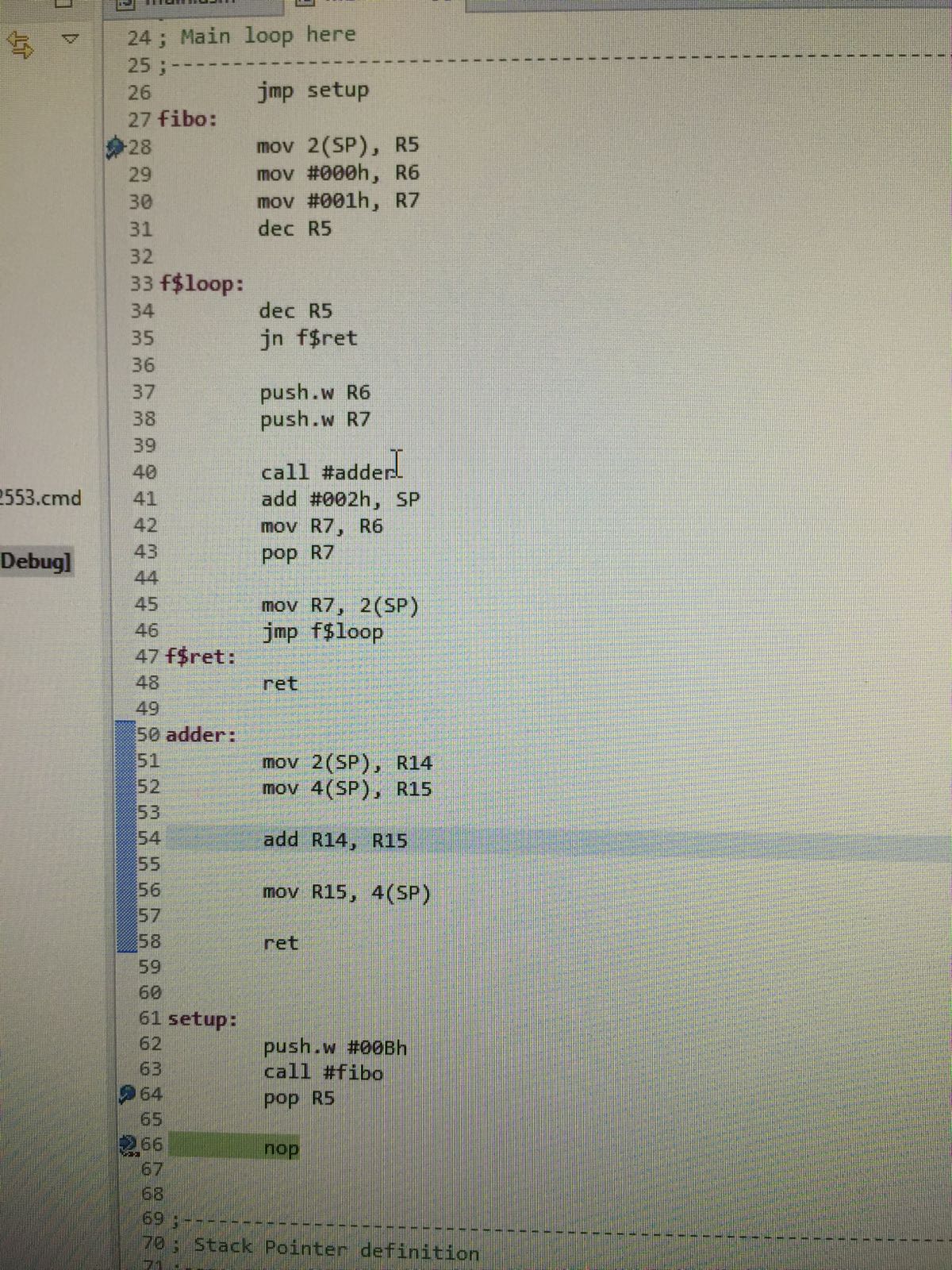
In this part of the experiment, we implemented Fibonacci algorithm and we used the adder function that we implemented in the part 2. We followed the algorithm in the Picture 2.3.1 and then implemented our code on CCS.



Picture 2.3.1: Iterative Fibonacci Algorithm

Fibonacci(0) is always 0, and Fibonacci(1) is always 1. When we want to calculate other Fibonacci numbers, we need to add previous two Fibonacci numbers. For example if we want to find out Fibonacci(2), we need to ad Fibonacci(0)+Fibonacci(1). This algorithm above, simply calculates all Fibonacci numbers Fibonacci(0)…..Fibonacci(n); by adding Fibonacci(n-1) and Fibonacci(n-2) in each for loop iteratively. And returns the Fibonacci(n).

The code that we implemented in the experiment can be observed from the Picture 2.3.2 below.



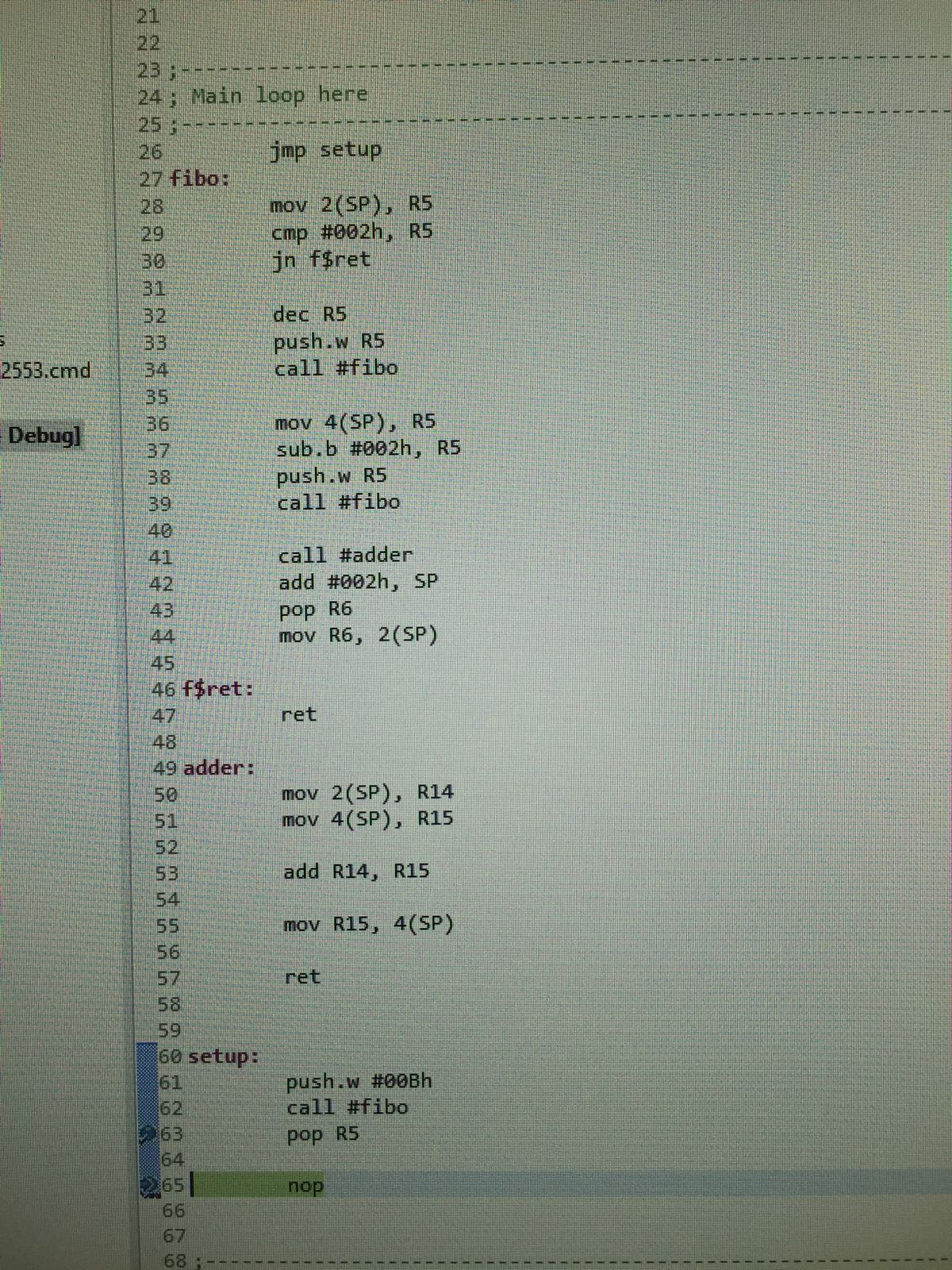
Picture 2.3.2: Iterative Fibonacci Algorithm

At the beginning, program jumps to “setup”. We pushed the hexadecimal value “#00Bh”, which is the value that we want to calculate its Fibonacci number. And then we called the subroutine “fibo”. When we used “call” instruction, the memory address of the next instruction(64th line) is pushed to the stack and when the program returns from this subroutine call, it will continue from the 64th line. In the “fibo” subroutine, we stored the value “#00Bh” in the R5 register by using the notation 2(SP). We used R6 and R7 registers to store hexadecimal values “#000h” and “#001h” as Fibonacci(0) and Fibonacci(1) numbers. And we decremented value of R5. In the routine which is labeled as f$loop, we decremented the value of R5 again, and we checked if the value in the R5 register is negative or not. If if is zero or positive, we pushed the values of R6 and R7 to the stack and we called the subroutine adder. In this subroutine, we stored the value of R7 into R14 register and the value of R6 into R15 register by using the stack pointer. After we added the values of R15 and R14, we stored the result in the R15 register and we pushed the result to the stack. After program is done with addition operation, it jumps to wherever it left in order to do addition, which means the program jumps to 41st line. We added “#002h” to the stack pointer so now stack pointer shows the result of the addition. We stored the value of R7 into R6 and we stored the value that stack pointer shows(the result of the addition), in R7 register by popping this value from the stack. And then we pushed the value of R7 to stack and we continued to do this steps until the value of R5 becomes negative. When R5 becomes negative, the program goes back to where it left in the first subroutine call and finishes the program.

## Part 4

In this part of the experiment, we implemented recursive Fibonacci algorithm in the CCS. The source code of this experiment can be observed from the Picture 2.4.1 which is placed below.

Picture 2.4.1: Recursive Fibonacci Algorithm



As a beginning, the code jumps to the portion of code which is labeled as “setup”. We pushed the hexadecimal number “#00Bh” which is the number that we want to obtain its Fibonacci number. Then we called the subroutine which is labeled as “fibo”, since we used “call” instruction, so the memory address of the next instruction (the address of the 63rd line) is pushed to the stack. We stored the original number that we want to calculate its Fibonacci number in R5 register with the help of the 2(SP) notation. Then we used “cmp” to compare the value of R5 and the hexadecimal value “#002h”. If the result of this comparison is negative, then we jumped to “f$ret” and returned to the position that program has left in the first place (63rd line). Otherwise, we decremented the value of R5 and we pushed this value into stack and we made recursive call to the fibo subroutine.

# Conclusion

Comment on any difficulties you have faced, what you have learned etc.