4

Stack and Subroutine

4.1 Introduction

This experiments aims to enhance the practical experience about function calls and usage of the stack.

Students are recommended to check Stack operations and also the differences between **CALL** and **JMP** instructions from the documents on Ninova. Students may also bring their own computers to the laboratory on which Texas Instruments Code Composer Studio IDE is installed (For installation instructions, review CCS_ Installation.pdf).

4.2 Preliminary

The Fibonacci Sequence consists of numbers from 0 to infinity, where the elements of the series are calculated by the function of $f(n) = f(n-1)+f(n-2)$ with initial conditions of $f(1) = 1$ and $f(0) = 0$. The Fibonacci Sequence is given below:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...$$

The Fibonacci Function can be calculated either recursively or iteratively using the algorithms below:

Algorithm 2 Fibonacci Implementation - Recursive

```
1: procedure FIB(n)
2:     if (n == 0) then
3:         return 0;
4:     else if (n == 1) then
5:         return 1;
6:     else
7:         return FIB(n − 1) + FIB(n − 2);
8:     end if
9: end procedure
```

Algorithm 3 Fibonacci Implementation - Iterative

```
1: procedure FIB(n)
2:     a = 0;
3:     b = 1;
4:     for i = 1 to n do
5:         c = a + b;
6:         a = b;
7:         b = c;
8:     end for
9:     return b;
10: end procedure
```

4.3 Experiment

4.3.1 Part 1 - Basics of a Subroutine Call

Simple program which consist of several function calls is given below. The given program takes an array of 8-bit integers and changes their signs by using 2-complement method and stores them to a memory location. Please, run the following program in order to observe and understand the basics of function call mechanism in microcomputer systems.

```
1  Setup      mov    #array, r5 ;use r5 as the pointer
2             mov    #resultArray, r10
3
4  Mainloop     mov.b   @r5,r6
5             inc    r5
6
7             call     #func1
8
9             mov.b r6,0(r10)
10            inc    r10
11
12            cmp    #lastElement,r5
13            jlo    Mainloop
14
15            jmp    finish
16
17
18 func1      xor.b #0FFh, r6
19            mov.b   r6,r7
20            call  #func2
21            mov.b r7,r6
22            ret
23
24 func2      inc.b r7
25            ret
26
27
28 ;Integer array
29 array      .byte 127,-128,0,55
30 lastElement
31
32
33 finish       nop
```

Additionally, you should include the following line above *.text* section in your main.asm to define uninitialized memory location where the program stores the output.

```
1 result    .bss  resultArray,5
```

You should run the given program step by step and fill Table 4.3.1 for the first iteration of the *Mainloop* by using Debug Mode of CCS. Also, please add this table to your report with sufficient explanations.

Code	PC	R5	R10	R6	R7	SP	Content of the Stack
mov #array, r5							
mov #resultArray, r10							
mov.b @r5,r6							
inc r5							
call #func1							
xor.b #0FFh, r6							
mov.b r6,r7							
call #func2							
inc.b r7							
ret							
mov.b r7,r6							
ret							
mov.b r6,0(r10)							
inc r10							

Table 4.1. Content of Registers and Stack

As seen in the given program, there are two functions defined such as **func1** and **func2**. The first function uses **R6** to take its input and return the related output. In the same manner, the second function uses **R7**. On the other hand, main program uses **R5** and **R10** in order to store addresses of input and output arrays.

Assume that, there is a collusion in the utilization of registers between the defined function and the main program. As an example, you could assume that the main program uses **R6** and **R7** to store the address information of the arrays. In the given scenario, any call of **func1** or **func2** will disorder the expected behaviour of the main program since the values of **R6** and **R7** would be overwritten. Thus, a programmer should prevent the loss of any necessary information while using nested functions. In fact, you may have used predefined functions of which you do not know the body (which registers are used in the function) to accomplish some task, how could the loss information be avoided? Please, provide a solution to the given problem other than using different registers and include your solution to the report.

4.3.2 Experiment - Part 2

In this part, you are required to implement a function, named *Adder* that calculates the sum of two integers. For example;

$$\text{Adder}(a,b)=a+b$$

When implementing the Adder function, you are required to pass the parameters of the method **through the stack**. Please check the *Assembly_language_tutorial* from Ninova for information on how to pass parameters to a function by means of the stack and how to retrieve the result in the same manner.

4.3.3 Experiment - Part 3

Implement the iterative version of the Fibonacci algorithm described above. You are also supposed to use the adder implementation in the previous part. Observe the behavior of the registers and stack during the execution.

4.3.4 Experiment - Part 4 (Optional)

Implement the recursive version of the Fibonacci algorithm described above. You are also supposed to use the adder implementation in the previous part. Observe the behavior of the registers and stack during the execution.

4.4 Report

Briefly explain what is studied during the experiments in the report. Furthermore, your report should also contain your program code (with explanations) for Parts 2-3. If you tried to implement the optional part, please do not hesitate to include it in your report too.

You should explain the anomalies (if countered) with their possible reasons while testing your program.