**Department of Computer Engineering**

BLG 351E  
Microcomputer Laboratory Experiment Report

Experiment No : 4

Experiment Date : 10.11.2017

Group Number : Friday - 13

Group Members :

|  |  |  |
| --- | --- | --- |
| **ID** | **Name** | **Surname** |
| 150160537 | OKAN | YILDIRIM |
| 150160546 | HASAN EMRE | ARI |
|  |  |  |
|  |  |  |

Laboratory Assistant : Yusuf Hüseyin Şahin

# Introduction

In this experiment we get used to use MSP430 Education Board, MSP430G2553 microcontroller and its assembly language in terms of function calls and usage of the stack. We enhanced the practical experience. Before the experiment, we studied on 1\_MSP430\_introduction document and get familiar with Stack Pointer (SP), Routines, and Passing parameters to the routine . We did preliminary work and reminded our background information.

# Experiment

## Part 1 – Basics of a subroutıne call

We analyzed given simple program which is contain of several function calls and we got it is how working. At the first function (func1), given variable’s bits are complemented by using ***xor.b*** command. Then, it calls second function (func2), which increment the variable. Totally, program takes an array of 8- bit integers and changes their sign by using 2’s complement method. It is the first time we used **c*all*** instead of ***jmp.*** Call and **jmp** instructions are different. One of the most important difference is stack is used when call instruction is used in order to return called address.

We run the given program. We observed and understood the basic of function call mechanism. We debug the program step by and step filled the given table. Table is given below on Table 1.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Code | PC | R5 | R10 | R6 | R7 | SP | Content of the Stack |
| mov #array,r5 | 0xC00E | 0xC038 | 0x0204 | 0x00C8 | 0x00C8 | 0x0400 | 0x0000 |
| mov #resultArray,r10 | 0xC012 | 0xC038 | 0x0200 | 0x00C8 | 0x00C8 | 0x0400 | 0x0000 |
| mov.b @r5,r6 | 0xC014 | 0xC038 | 0x0200 | 0x00C8 | 0x00C8 | 0x0400 | 0x0000 |
| inc r5 | 0xC016 | 0xC039 | 0x0200 | 0x007F | 0x00C8 | 0x0400 | 0x0000 |
| call # func1 | 0xC028 | 0xC039 | 0x0200 | 0x007F | 0x00C8 | 0x03FE | 0xC01A |
| xor.b #0ffh,r6 | 0xC02A | 0xC039 | 0x0200 | 0x007F | 0x00C8 | 0x03FE | 0xC01A |
| mov.b r6,r7 | 0xC02C | 0xC039 | 0x0200 | 0x0080 | 0x0080 | 0x03FE | 0xC01A |
| call #func2 | 0xC034 | 0xC039 | 0x0200 | 0x0080 | 0x0080 | 0x03FC | 0xC030 0xC01A |
| inc.b r7 | 0xC036 | 0xC039 | 0x0200 | 0x0080 | 0x0081 | 0x03FC | 0xC030  0xC01A |
| ret | 0xC030 | 0xC039 | 0x0200 | 0x0080 | 0x0081 | 0x03FE | 0xC01A |
| mov.b r7,r6) | 0xC032 | 0xC039 | 0x0200 | 0x0081 | 0x0081 | 0x03FE | 0x0000 |
| ret | 0xC01A | 0xC039 | 0x0200 | 0x0081 | 0x0081 | 0x0400 | 0x0000 |
| mov.b r6,0(r10) | 0xC01E | 0xC039 | 0x0200 | 0x0081 | 0x0081 | 0x0400 | 0x0000 |
| inc r10 | 0xC020 | 0xC039 | 0x0201 | 0x0081 | 0x0081 | 0x0400 | 0x0000 |

***Table 1 Content of Registers and Stack***

In this program, four registers are used and there is a possibility to collision of registers between the defined function and the main program. As programmer we should avoid data loss so that we develop a solution to protect data by using stack instead of using different registers. Our solution is given below:

result .bss resultArray,5 ; above .text section

;Mainloop

Setup mov #array, r5 ;use r5 as the pointer

mov #resultArray,r6

push r5

push r6

Mainloop mov.b @r5,r6

push r6

call #funcl

pop.b r5

pop r6

mov.b r5,0(r6)

pop r5

inc r5

inc r6

push r5

push r6

cmp #lastElement,r5

jlo Mainloop

jmp finish

func1 pop r5 ; get return address

pop.b r6 ; get r6

xor.b #OFFh, r6 ; exchange bits, first step of 2’s complement

push r5 ; strore the return adress

push.b r6 ;store the edited r6 on the stack for passing parameters

call #func2

pop.b r6 ; get the parameters from func2

pop r5 ; ; get the parameters from func2

push.b r6 ;store the edited r6 on the stack for passing parameters

push r5 ; top of the stack must be return address

ret

func2 pop r5 ; get return address

pop.b r6 ; get r6

inc.b r6 ; increment r6, second step of 2’s complement

push.b r6 ;store the edited r6 on the stack for passing parameters

push r5 ; top of the stack must be return address

ret

;Integer array

array .byte 127,-128,0,55

lastElement

Semantic of the program is the same for each two method. However, we used stack and take advantage of stack in this method.

The stack pointer (SP/R1) is used by the CPU to store the return addresses of subroutine calls and interrupts. When we call the subroutine, current address is pushed on the top of the stack. In subroutine, ret command is return the address which is in the top of the stack. We always used the value on the top of the stack, in other words, we did not use such as 2(SP), 0(SP) expressions. We also used the method of passing the parameters of the method through the stack. At the each function firstly we get the return address by using pop command and we used push command to add the this return address to the top of the stack before the return. In this way, we used much push and pop command, yet we used only a two register.

## Part 2 adder

In this part, we implemented Adder function that calculates the sum of two integers. We used the method of passing the parameters of the method through the stack. The parameters necessary to execute the routine are placed ın the stack using the PUSH instruction. Returning from the routine, the stack can again be used to pass the parameters, using POP instruction. Our code is given below:

mov.b #002h,r5

mov.b #005h,r6

push r5

push r6

call #adder

jmp finish

adder pop r7

pop r5

pop r6

add.b r5,r6

push r6

push r7

ret

finish nop

The above code performs the addition of the given arguments. Firstly, before calling the function, we assume that the values are in the stack. To do this, we first put the values into a register and put the registers into the stack. Next, we called the adder function. We assigned the return address of the Adder function to variable R7. The first value in the stack is assigned to R5 register, and the second value is assigned to R6 register. We have summed the values in these two registers. At the end of the sum, the total R6 is assigned to the register. After, we push the value of in R6 register to the stack. Finally, the value inside of R7 register that return address of adder function pushed to stack. End of the program, total value saved in the stack.

## Part 3 ıterative fıbonaccı

In this part, we implemented iterative Fibonacci algorithm described as the preliminary section. We used the method of passing the parameters of the method through the stack again and we use the Adder implementation above. Our program is given below:

;Iterative fibonacci

mov.b #000h,r5

mov.b #001h,r6

push r5

push r6

fibonacci pop r6

pop r5

push r6

push r5

push r6

call #adder

jmp fibonacci

adder pop r7

pop r6

pop r5

add.b r5,r6

push r6

push r7

ret

finish nop

The above code performs the addition of the given arguments. First, before calling the function, we assume that the values are in the stack. To do this, we first put the values into a register and put the registers into the stack. We assigned values to the registers to access the values in the stack. We set the values in stack to be first value, second value and first value again. Next we called the adder function. The Adder function added the two values at the top of the stack and stores the result back into the stack. As a result, two values at the top of the stack are the total value and the first value. When the Adder function returns, the value at the top of the stack is the value if it evaluates to the number of Fibonacci.

# Conclusion

We learn to how to use routine, call command and stack. We had a difficulty in debugging the program step by step and writing the result on the table because there was a problem on the our computer. It shows the value on the register inconsistently and the results are totally different from the other computers. We wasted too much time on filling table due to the that problem. We changed the computer at last time. However we implement other program fast. It was challenging but joyful also.