**Department of Computer Engineering**

BLG 351E  
Microcomputer Laboratory Experiment Report

Experiment No : 4

Experiment Date : 06.11.2017

Group Number : Monday - 17

Group Members :

|  |  |  |
| --- | --- | --- |
| **ID** | **Name** | **Surname** |
| 150150302 | İrem Nur | Demirtaş |
| 150160706 | Merve Elif | Demirtaş |
| 150140719 | Cemal | Türkoğlu |

Laboratory Assistant : Ahmet Arış

# Introduction

The aim of this experiment using the MSP430 for implementing assembly code with stack and subroutine operations for understanding the differences between them. In this assembly code, CALL and JMP instruction was used according to purposes.

# Experiment

Code Composer Studio was used to contact with MSP430. A new assembly project was created to implement the code and debugged with step by step for understanding operations and observing registers value.

## BASICS OF A SUBROUTINE CALL

The code as shown below was implemented which takes an array of 8-bit integers and changes their signs by using 2-complement method and stored them to a memory location. This code have several call function. So, understanding the working principal PC, SP pointer, R5, R10, R6, R7 register and also content of the stack were observed.

1. ;-------------------------------------------------------------------------------
2. ;integer array
3. array       .byte   127,-128,0,55
4. lastElement
6. result      .bss    resultArray, 5   ;unitialized array
8. .text                           ; Assemble into program memory.
9. .retain                         ; Override ELF conditional linking
10. ; and retain current section.
11. .retainrefs                     ; And retain any sections that have
12. ; references to current section.
13. ;-------------------------------------------------------------------------------
14. RESET       mov.w   #\_\_STACK\_END,SP         ; Initialize stackpointer
15. StopWDT     mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer
17. ;-------------------------------------------------------------------------------
18. ; Main loop here
19. ;-------------------------------------------------------------------------------
20. Setup   mov   #array, r5 ;use r5 as a pointer
21. mov   #resultArray, r10
23. MainLoop    mov.b   @r5, r6   ;copy element of r5 to r6
24. inc     r5   ;increment for the next element
26. call    #func1   ;go to the func1
28. mov.b   r6, 0(r10)   ;store the changed value
29. inc     r10   ;increment for slot
31. cmp     #lastElement, r5   ;compare addresses for end of array
32. jlo     MainLoop   ;jump if not end to start
33. jmp     Finish
35. func1       xor.b   #0FFh, r6   ;for 2’s complement operation
36. mov.b   r6, r7   ;copy the r7
37. call    #func2   ;call func2 for increment r7
38. mov.b   r7, r6   ;copy to r6
39. ret
41. func2       inc.b   r7
42. ret
44. Finish      nop
45. ;-------------------------------------------------------------------------------
46. ; Stack Pointer definition
47. ;-------------------------------------------------------------------------------
48. .global \_\_STACK\_END
49. .sect   .stack
51. ;-------------------------------------------------------------------------------
52. ; Interrupt Vectors
53. ;-------------------------------------------------------------------------------
54. .sect   ".reset"                ; MSP430 RESET Vector
55. .short  RESET

If only R6 and R7 registers should be used, for not losing the values of R6 and R7, their values can be pushed the stack before going to the function and after returning that function their values must be popped.

## ADDER FUNCTION WITH STACK

In this part, adding operation of two number was implemented through stack. The code can be seen as below. Firstly VAR1 and VAR2 values were initialized in data segment and port1 was arranged as output for LEDs. In MainLoop, two values were pushed to the stack and was called the adder function. Adder function pops firstly the return address for not losing it and then pops two values for adding operation. The added value and also return address were pushed to the stack. After returning adder function added value was taken and cleared.

1. ;-------------------------------------------------------------------------------
2. .data
3. VAR1        .byte   10
4. VAR2        .byte   10
6. .text                           ; Assemble into program memory.
7. .retain                         ; Override ELF conditional linking
8. ; and retain current section.
9. .retainrefs                     ; And retain any sections that have
10. ; references to current section.
11. ;-------------------------------------------------------------------------------
12. RESET       mov.w   #\_\_STACK\_END,SP         ; Initialize stackpointer
13. StopWDT     mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer
15. ;-------------------------------------------------------------------------------
16. ; Main loop here
17. ;-------------------------------------------------------------------------------
19. mov.w   #0ffh,  &P1DIR
21. MainLoop
22. push.b  &VAR1       ;push VAR1 value
23. push.b  &VAR2       ;push VAR2 value
25. call    #adder
27. pop.b   R12         ;take added value
28. pop.b   R13         ;clear the stack
29. jmp     finish
31. adder
32. pop.w   R8          ;pop return address
33. pop.b   R9
34. pop.b   R10
36. add R9,R10          ;add operation
38. push.b  R10         ;push the added value
39. push.b  R8          ;push return address
41. ret
42. finish

45. ;-------------------------------------------------------------------------------
46. ; Stack Pointer definition
47. ;-------------------------------------------------------------------------------
48. .global \_\_STACK\_END
49. .sect   .stack
51. ;-------------------------------------------------------------------------------
52. ; Interrupt Vectors
53. ;-------------------------------------------------------------------------------
54. .sect   ".reset"                ; MSP430 RESET Vector
55. .short  RESET

In this part, instead of push and pop operation for return address, index could be used for clear and safe operation.

## FIBONACCI WITH ITERATION

In this part, Fibonacci was implemented with iteratively. The pseudo code of Fibonacci as shown in below.

**procedure** FIB(*n*)  
a = 0;  
b = 1;  
**for** i = 1 to *n* **do**  
 c = a + b;  
 a = b;  
 b = c;  
**end** **for**  
**return** b;  
**end procedure**

Firstly,in the data segment, N value was initialized as 8. Port1 was arranged as output for turning LEDs to observe the added values. In the mainloop, the number pushed to the stack, and FIB function was called. In FIB function, a and b values were initialized as in pseudo code. After that, return address popped and N values was taken again with pop operation. For the for loop i declared as 1, and checked with N in every iteration. In the next step, a and b values were popped for adder function and after that function added value was taken. a = b and b = c operations, as seen in pseudo code, was implemented. Therefore, this added value was written to the port1 for observing in the LEDs. After returning the FIB function, the stack was cleared with pop operation.

1. ;-------------------------------------------------------------------------------
2. .data
3. NUM         .byte   8
5. .text                           ; Assemble into program memory.
6. .retain                         ; Override ELF conditional linking
7. ; and retain current section.
8. .retainrefs                     ; And retain any sections that have
9. ; references to current section.
10. ;-------------------------------------------------------------------------------
11. RESET       mov.w   #\_\_STACK\_END,SP         ; Initialize stackpointer
12. StopWDT     mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer
14. ;-------------------------------------------------------------------------------
15. ; Main loop here
16. ;-------------------------------------------------------------------------------
18. mov.w   #0ffh,  &P1DIR  ;direction port1 as output
20. MainLoop
21. push.b  &NUM   ;push number value
22. call    #FIB
23. pop.b   R12   ;clear the stack
24. jmp     finish
26. adder
27. pop.w   R8          ;pop return address
28. pop.b   R9
29. pop.b   R10
31. add R9,R10          ;add operation
33. push.b  R10         ;push the added value
34. push.b  R8          ;push return address
36. ret
38. FIB         clr.b   R15         ;a = 0
39. clr.b   R4
40. inc.b   R4          ;b = 1
41. pop.w   R5          ;pop return address
42. pop.b   R6          ;N
43. clr.b   R7          ;i = 0
45. LOOP        inc.b   R7          ;i = 1
46. cmp     R6,R7       ;if i less than n
47. jge     exit
49. push.b  R15   ;push a
50. push.b  R4   ;push b
51. call    #adder   ;c = a + b
52. pop.b   R11   ;pop added value (c)
53. mov.b   R4,R15   ;a = b
54. mov.b   R11,R4   ;b = c
55. mov.b   R4,&PIOUT   ;turn the LEDs
56. jmp     LOOP
58. exit        push.b  R4
59. push.b  R5
60. ret
62. finish

65. ;-------------------------------------------------------------------------------
66. ; Stack Pointer definition
67. ;-------------------------------------------------------------------------------
68. .global \_\_STACK\_END
69. .sect   .stack
71. ;-------------------------------------------------------------------------------
72. ; Interrupt Vectors
73. ;-------------------------------------------------------------------------------
74. .sect   ".reset"                ; MSP430 RESET Vector
75. .short  RESET

# Conclusion

In conclusion, learned the stack and subroutine operations. When a function called, return address added to the stack for continuing after returning from the function. So, for using a value from stack, return address must be considered and not lost. In addition, for call by value, stack can be used with pushed values.