ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 351E

MICROCOMPUTER LABORATORY EXPERIMENT REPORT

EXPERIMENT NO	:	4
EXPERIMENT DATE	:	23.10.2019
LAB SESSION	:	WEDNESDAY - 13.30
GROUP NO	:	G10

GROUP MEMBERS:

150170062	:	Mehmet Fatih YILDIRIM
150180704	:	Cihat AKKİRAZ
150180705	:	Batuhan Faik DERİNBAY
150180707	:	Fatih ALTINPINAR

FALL 2019-2020

Contents

1 INTRODUCTION

Subroutines are very important in assembly language. Using them program size decreases and codes are getting more understandable. While using subroutines, some parameters should be enter this subroutine and return value end of the some calculation. End of the subroutine, program should be continue its execution with the instruction after subroutine call in the main program. Entering parameters to the subroutine, return values and PC address of instruction after the subroutine call should be stored somewhere. Stack is a special memory about to use during execution of the program. Coding in assembly language with MSP430G2553 microprocessor, PUSH and POP commands are using to add value to the top of the stack and remove value from top of the stack with copying it a register. So, stack is a LAST IN FIRST OUT(LIFO) structure. While calling a subroutine with CALL instruction, address value of next instruction after subroutine call in the subroutine is pushed to the stack. Thus, the program can continue to run after the subroutine operations.

2 MATERIALS AND METHODS

This experiment is conducted via using MSP430G2553 microprocessor. This microprocessor is programmed using Code Composer Studio according to desired tasks on the experiment handout. During coding below sources are used:

- MSP430 Education Board Manual [1]
- MSP430 Architecture Chapter 4 [2]
- MSP430 Instruction Set [3]
- Supplementary Chapter 6 General Purpose IO [?]

2.1 Part 1

In the first part of the experiment, a new CCS Project is created as choosing "MSP430G2553" as the target and "Empty Assembly Project" as the project template. Then, the codes in the experiment booklet(also see Figure 2.1) are run and debugged. While debugging, values in the registers and stack are examined and table(see Figure 2 is filled with these values.

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

Figure 1: Assembly Code of Part 1

For better understanding, further examination of the code, line by line if necessary, is required:

- Line 1-2: Address of array is moved to R5 and address of resultArray is move to R10.
- Line 4: The value in pointed by R5 is moved to R6. (Indirect Addressing)
- Line 5: The value of R5 is increased by 1.
- Line 6: func1 is called. Address of the next instruction(PC+2) in line 7 is pushed to stack to return main function after finishing operation in function. Address of

SP is changed because of adding new value to stack. PC value will be changed to execute func1 with this instruction.

- Line 12: R6 value is changed with X0R operation.
- Line 13: R6 value is moved to R7.
- Line 14: func2 is called. Address of the next instruction(PC+2) in line 15 is pushed to stack to return func1 function after finishing operation in func2. Address of SP is changed because of adding new value to stack. PC value will be changed to execute func2 with this instruction.
- Line 18: R7 value is increased by 1.
- Line 19: With RET instruction function is returning address(Line 15) that is pushed to stack in line 14. This value is popped from stack and appointed as new PC.
- Line 15: R7 value is moved to R6.
- Line 16: With RET instruction function is returning address(Line 7) that is pushed to stack in line 6. This value is popped from stack and appointed as new PC.
- Line 7: R6 value is moved to R10.
- Line 8: Address of last element of array is compared with R5 to see if it's at the end of the list.
- Line 9: Jump Mainloop again if value in R5 is lower than address of last element of array.
- Line 10: If R5 is bigger than lastElement or equal to lastElement finish program.

Code	PC	R5	R10	R6	R7	SP	Content of the Stack
mov #array, r5	0xC00E	0xC038	0x5FBE	0x0262	0x0262	0x0400	FFFF
mov #resultArray, r10	0xC012	0xC038	0x0200	0x0262	0x0262	0x0400	
mov.b @r5,r6	0xC014	0xC038	0x0200	0x007F	0x0262	0x0400	
inc r5	0xC016	0xC039	0x0200	0x007F	0x0262	0x0400	
call #func1	0xC028	0xC039	0x0200	0x007F	0x0262	0x03FE	C01A
xor.b #0FFh, r6	0xC02A	0xC039	0x0200	0x0080	0x0262	0x03FE	C01A
mov.b r6,r7	0xC02C	0xC039	0x0200	0x0080	0x0080	0x03FE	C01A
call #func2	0xC034	0xC039	0x0200	0x0080	0x0080	0x03FC	C030 C01A
inc.b r7	0xC036	0xC039	0x0200	0x0080	0x0081	0x03FC	C030 C01A
ret	0xC030	0xC039	0x0200	0x0080	0x0081	0x03FE	C01A
mov.b r7,r6	0xC032	0xC039	0x0200	0x0081	0x0081	0x03FE	C01A
ret	0xC01A	0xC039	0x0200	0x0081	0x0081	0x0400	
mov.b r6,0(r10)	0xC01E	0xC039	0x0200	0x0081	0x0081	0x0400	
inc r10	0xC020	0xC039	0x0201	0x0081	0x0081	0x0400	

Figure 2: Registers and stack status while Debugging

2.2 Part 2

In this section the team was asked to write four functions, namely addition, subtraction, multiplication and division, in assembly language.

In order to implement the given task, team created following pieces of codes given in Figures 2.2.1 - 2.2.4.

Under each figure, explanation of the code can be found.

```
1  Add_func        push    r4
2                  push    r5
3                  mov             6(sp), r4
4                  mov             8(sp), r5
5                  add             r5, r4
6                  mov             r4, 10(sp)
7                  pop             r5
8                  pop             r4
9                  ret
10
```

Figure 3: Addition Function

- Line 1-2: Pushes registers to be used to stack in order to keep their value.

- Line 3-4: Retrieves caller parameters from stack
- Line 5-6: Does the addition and pushes to result location of the stack
- Line 7-8: Loads used registers with previous values for later use in caller function
- Line 9: Returns back to next instruction after the caller function

```
1  Sub_func        push    r4
2                  push    r5
3                  mov             6(sp), r4
4                  mov             8(sp), r5
5                  sub             r5, r4
6                  mov             r4, 10(sp)
7                  pop             r5
8                  pop             r4
9                  ret
10
```

Figure 4: Subtraction Function

- Line 1-2: Pushes registers to be used to stack in order to keep their value.
- Line 3-4: Retrieves caller parameters from stack
- Line 5-6: Does the subtraction and pushes to result location of the stack
- Line 7-8: Loads used registers with previous values for later use in caller function
- Line 9: Returns back to next instruction after the caller function

```
 1  ; R6 = i
 2  Mul_func        push    r4
 3                  push    r5
 4                  push    r6
 5                  mov             #0, r6
 6                  mov             8(sp), r4
 7                  mov             10(sp), r5
 8  for_mul         cmp             r5, #0
 9                  jz              mul_return
10                  add             r4, r6
11                  dec             r5
12                  jmp             for_mul
13  mul_return      mov             r6, 12(sp)
14                  pop             r6
15                  pop             r5
16                  pop             r4
17                  ret
18
```

Figure 5: Multiplication Function

- Line 2-4: Pushes registers to be used to stack in order to keep their value.
- Line 5-7: Retrieves caller parameters from stack and initializes temporary registers, namely R6 in this case
- Line 8-12: Does the multiplication until calling parameter ”b” is 0. Decrements ”b” every step so it is utilized as an index. The parameter ”a” is added to itself ”b” times and stored in R6 in the loop.
- Line 13-16: Stores the return value R6 and loads used registers with previous values for later use in caller function
- Line 17: Returns back to next instruction after the caller function

```
1  Div_func        push    r4
2                  push    r5
3                  push    r6
4                  mov             #0, r6
5                  mov             8(sp), r4
6                  mov             10(sp), r5
7  for_div         cmp             r5, r4
8                  jl              div_return
9                  sub             r5, r4
10                 inc             r6
11                 jmp             for_div
12 div_return      mov             r6, 12(sp)
13                 pop             r6
14                 pop             r5
15                 pop             r4
16                 ret
17
```

Figure 6: Division Function

- Line 1-3: Pushes registers to be used to stack in order to keep their value.
- Line 4-6: Retrieves caller parameters from stack and initializes temporary registers, namely R6 in this case.
- Line 7-11: Does the division until calling parameter "a" is less than "b". Subtracts "b" from "a" every step and increments R6 every step so R6 is utilized as the result. The parameter "b" is subtracted from "a" R6 times in the loop so R6 holds the integer division value.
- Line 12-15: Stores the return value R6 and loads used registers with previous values for later use in caller function
- Line 16: Returns back to next instruction after the caller function

2.3 Part 3

In this part of the experiment call of a subroutine from another subroutine needed to be implemented. Since add subroutine from part 2 has to be used in this part, both functions should use stack as intended for not overriding each other's data thus producing a wrong result.

In order to obtain a subroutine that calculates permutation the team decided on the following piece of code given in Figure 7.

```
 1  Perm_func       push    r4
 2                  push    r5
 3                  push    r6
 4                  mov             #1, r6
 5                  mov             8(sp), r4
 6                  mov             10(sp), r5
 7  for_perm        cmp             #0, r5
 8                  jz              perm_return
 9                  push    r6
10                  push    r6
11                  push    r4
12                  call    Mul_func
13                  pop             r4
14                  pop             r6
15                  pop             r6
16                  dec             r4
17                  dec             r5
18                  jmp             for_perm
19  perm_return     mov             r6, 12(sp)
20                  pop             r6
21          pop             r5
22                  pop             r4
23                  ret
```

Figure 7: Permutation Function

- Line 1-3: Pushing registers that is going to be used during execution of the subroutine so they don't lose their value. So functionality of the caller function or main routine won't be disrupted.

- Line 4: $R6$ will hold the end result. It is also the register that one of the numbers that will be multiplied.

- Line 5-6: Loading function parameters relative to the stack pointer.
- Line 7-8: When $P(n, r)$ (n = R4, r = R5) needs to be calculated, the result can be obtained by multiplying all the of the numbers starting from n to $n - r + 1$. In this multiplication there will be r many numbers, thus implementation of a for loop is required which will work r times.
- Line 9-15: Multiplication function is called with the parameters $R6$ which is the current result of the multiplication and $R4$ which is a number from $n(n-1)...(n-r+1)$. Result is loaded to R6.
- Line 16-17: Decrementing $R4$ in order to obtain next value from $n(n-1)...(n-r+1)$ that needs to be multiplied and $R5$ so for loop is called r many times.
- Line 18: Jump to for loop
- Line 19: Writing the result to the stack relative to the stack pointer by 12 bytes. Since there are 3 registers that are pushed to be saved, a return address, 2 function parameters which is in total $(3 + 1 + 2)x2 = 12$.
- Line 20-22: Loading original values of the registers
- Line 23: Returning to the address where this function called in the first place.

In order to obtain a recursive function that calculates factorial the team decided on the following piece of code given in Figure 8.

```
 1  Fact_func       push    r4
 2                  push    r5
 3                  mov             6(sp), r4
 4                  cmp             #2, r4
 5                  jl              fact_rtrn
 6                  mov             r4, r5
 7                  dec             r5
 8                  push    r5
 9                  push    r5
10                  call    Fact_func
11                  pop             r5
12                  pop             r5
13                  push    r4
14                  push    r5
15                  push    r4
16                  call    Mul_func
17                  pop             r4
18                  pop             r4
19                  pop             r4
20                  mov             r4, 4(sp)
21                  pop             r5
22                  pop             r4
23                  ret
24  fact_rtrn       mov             #1, 4(sp)
25                  pop             r5
26                  pop             r4
27                  ret
```

Figure 8: Factorial Function

- Line 1-2: Saving the values of register that will used during this operation
- Line 3: Loading the parameter of the function from stack.
- Line 4-5 and Line 24-27: Exit condition of the recursive function. When $n < 2$, $fact(n)$ will return 1 since $fact(1) = fact(0) = 1$
- Line 6-7: Since $n - 1$ is required to calculate $fact(n - 1)$ value of $R4$ is copied to $R5$ then decremented.

- Line 8-12: Called $fact(n-1)$ and result will be written on the $R5$ which was holding the value of $n-1$ now holds result of the $fact(n-1)$
- Line 13-19: A function call is made in order to obtain the value of $fact(n)$ which is equal to $nx(n-1)$ the value of n was stored in $R4$ and result of $fact(n-1)$ is stored in $R5$ thus these registers are pushed to stack.
- Line 20: Result of the $fact(n)$ is written to stack.
- Line 21-23: Popping original values of the used registers then returning.

2.4 Part 4

We could use 16 bits in a way that they would represent some necessary information about the number, for example; its sign, etc. This way, we could represent fractional part of a number as well as its integer part. Some format like IEEE 754 half-precision binary floating-point format (binary16) can be used in this manner. According to Wikipedia, "The IEEE 754 standard specifies a binary16 as having the following format:

Sign bit: 1 bit Exponent width: 5 bits Significand precision: 11 bits (10 explicitly stored)

The format is laid out as follows:

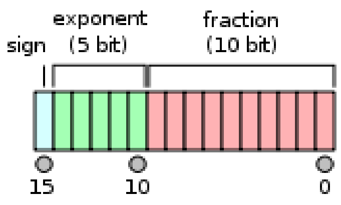

Figure 9: binary16 format

The format is assumed to have an implicit lead bit with value 1 unless the exponent field is stored with all zeros. Thus only 10 bits of the significand appear in the memory format but the total precision is 11 bits. In IEEE 754 parlance, there are 10 bits of significand, but there are 11 bits of significand precision ($log10(211)$≈3.311 decimal digits, or 4 digits ± slightly less than 5 units in the last place). ..."[4]

3 RESULTS

In the first part of the experiment, the team had try to understand to the given code as using debug mode and observing register and stack values step by step. The team observed what happens on the stack and on the some special registers while calling CALL, RET, JMP and other instructions. The results are recorded in the table.

In the second part of the experiment, the team did many times pushed a value to the stack and pulled a value from the stack. While calling subroutines with parameters, parameters and 2 more of PC value are pushed to the stack. At the end of the subroutine results are pulled to see result of subroutine operation, before pushed 2 more of PC is pulled to resume the main program with instruction after calling the subroutine, and before pushed parameters are pulled to avoid unnecessary space in the stack.

Subroutines are written in the second part are tested using main function in Figure 10. This main function will be explained line by line for only while testing addition function. Other functions are used in main function exactly same way with addition function.

- Line 2: The starting address of the allocated yArray in memory is saved to r7 to save and display the results.
- Line 3: In order to decrement SP value by 2, 0 is pushed to stack.
- Line 4-5: Values that will be added are pushed to stack.
- Line 6: Add function is called and address of instruction(PC+2) in line 7 is pushed to stack to return main function after finishing operation in function.
- Line 7-8-9: R4 value(a), R5 value(b) and R6 value(result of function) is pulled from stack.

After executing whole main function, results are moved to yArray. The team observed the results using memory window in Code Composer. The results of all functions matched the mathematical knowledge of the team.

```
    ; R4 = a, R5 = b, R6 = result
                        mov             #yArray, r7
main            push    #0
                        push    #2
                        push    #5
                        call    #Add_func
                        pop             r4
                        pop             r5
                        pop             r6
                        mov     r6, 0(r7)
                        add             #2, r7
                        push    #0
                        push    #2
                        push    #5
                        call    #Sub_func
                        pop             r4
                        pop             r5
                        pop             r6
                        mov     r6, 0(r7)
                        add             #2, r7
                        push    #0
                        push    #5
                        push    #2
                        call    #Mul_func
                        pop             r4
                        pop             r5
                        pop             r6
                        mov     r6, 0(r7)
                        add             #2, r7
                        push    #0
                        push    #2
                        push    #5
                        call    #Div_func
                        pop             r4
                        pop             r5
                        pop             r6
                        mov     r6, 0(r7)
                        add             #2, r7
                        push    #0
                        push    #2
                        push    #5
```

Figure 10: Calling and testing functions written in Part 2

In the third part of the experiment, two subroutines are implemented to do permutation and factorial operation as using some implemented subroutines in the second part. Like previous part, the team have cope with many push and pull operations through stack. As a result, with the implementation that detailed in section 2 the operations are calculated successfully.

```
; R4 = a, R5 = b, R6 = result

                        mov             #yArray, r7
main            push    #0
                        push    #2
                        push    #5
            call        #Perm_func
                        pop             r4
                        pop             r5
                        pop             r6
                        mov     r6, 0(r7)
                        add             #2, r7
                        push    #0
                        push    #5
                        call    #Fact_func
                        pop             r5
                        pop             r6
                        mov     r6, 0(r7)
                        add             #2, r7
end                     nop
                        jmp             end
```

Figure 11: Calling and testing functions written in Part 3

In order to test the functions in the third part, the main function is written in the same way as the main function in the second part. It is explained enough. The written main function can be seen in Figure 11

4 DISCUSSION

At the end of the experiment, it was seen that stack is useful during nested function calls and recursive functions. Also, this temporary storage is used to pass parameters to function and retrieve return value from the function. It is obvious that stack is so important while developing applications with functions.
Please refer to section 2 ”MATERIALS AND METHODS” for exclusively detailed information, tables, images, analysis, interpretation and results, covering all the required material under other sections.

5 CONCLUSION

With this experiment, team have done many push and pop operations as already mentioned. The team implemented subroutines to perform some mathematical operations. While calling these subroutines and getting the result of the result the stack is used. Also, it was seen that usage of stack during nested function calls and recursive functions. While using instructions like CALL, RET, JMP what happens on the stack and in the program execution have understood. The difference between CALL and JMP operations are observed.

REFERENCES

[1] Texas Instruments. Msp430 education board document. 2009.

[2] Texas Instruments. Msp430 architecture. 2009.

[3] Texas Instruments. Msp430 architecture. December 2004 - Revised July 2013.

[4] Ieee binary16 format.

[5] Overleaf documentation https://tr.overleaf.com/learn.

[6] Detailed info on writing reports https://projects.ncsu.edu/labwrite/res/res-studntintro-labparts.html.