ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 351E
MICROCOMPUTER LABORATORY EXPERIMENT REPORT

EXPERIMENT NO	:	4
EXPERIMENT DATE	:	6.12.2024
LAB SESSION	:	FRIDAY - 14.30
GROUP NO	:	G7

GROUP MEMBERS:

150210087	:	HİLAL KARTAL
150210100	:	SELİN YILMAZ
150210067	:	ALPER DAŞGIN

FALL 2024

Contents

1 INTRODUCTION [10 points]

In the first section, we executed a program that modified an integer array via nested function calls. Stepping through the program, examining stack changes, and understanding how registers are utilised for argument passing and return values were all part of the assignment. We learnt more about how the stack and function calls interact by completing a table of execution data.

Basic arithmetic operations (Add, Subtract, Multiply, and Divide) were to be implemented as subroutines for the second section. The stack was used to pass parameters, and the same process was used to obtain the results. Additionally, Add and Subtract were used in the construction of Multiply and Divide, enabling effective code reuse and modular designs.

In the third section, we had to recursively calculate the dot product of two vectors. This exercise showed that complex mathematical processes might be effectively implemented in assembly using stack-based recursion. Through the use of recursive logic and arithmetic subroutines, the dot product computation demonstrated advanced programming ideas.

All things considered, the tasks in this experiment helped us better understand subroutine design, recursive implementation, and stack-based memory management—all of which are critical abilities for programming MSP430 microcontrollers in assembly.

2 MATERIALS AND METHODS [40 points]

2.1 MATERIALS

- MSP430
- Selin's Laptop

2.2 METHODS

2.2.1 FIRST PART

In this part, a sample code with several function calls is given to us. We only ran this code and saved the results from the register. The given code is in the following.

```
 1 result     .bss   resultArray, 5
 2 
 3 Setup        mov    #array, R5
 4              mov    #resultArray, R10
 5 
 6 Mainloop     mov.b  @R5, R6
 7              inc    R5
 8              call   #func1
 9              mov.b  R6, 0(R10)
10              inc    R10
11              cmp    #lastElement, R5
12              jlo    Mainloop
13              jmp    finish
14 
15 func1        dec.b  R6
16              mov.b  R6, R7
17              call   #func2
18              mov.b  R7, R6
19              ret
20 func2        xor.b  #0FFh, R7
21              ret
22 
23 ;Integer array,
24 array        .byte  1, 0, 127, 55
25 lastElement
26 
27 finish       nop
```

Listing 1: Part 1

Line 1: This is actually above *.text* unlike other codes. It reserves 5 byte for *resultArray*.

Line 3: Moves *array* that defined on line 24 to register r5.

Line 4: Moves *resultArray* that defined on line 1 to register r10.

Line 6: Load current byte - that r5 points - to r6.

Line 7: Increment r5 by 1.

Line 8: Branch to subroutine *func1*. Saves address of next instruction to stack.

Line 15: Decrement r6.

Line 16: Move r6 to r7.

Line 17: Branch to subroutine *func2*. Save address of the next instruction to stack.

Line 20: `XOR` r7 with 0x0ff.

Line 21: Pop stack and return to popped address.

Line 18: Move r7 to r6.

Line 19: Pop stack and return to popped address.

Line 9: Move r6 to memory location pointed by r10. We loaded *resultArray* to r10 at the beginning. So we loaded r6 to the first position of *resultArray*.

Line 10: Increment r10. So it points to the second position of *resultArray*.

Line 11: Compare current address in r5 with the address of *lastElement* to check if the end of *array* is reached.

Line 12: If r5 is lower than *lastElement* then jump to line 6 *(Mainloop)*.

Line 13: If not jump to finish.

Line 27: Dead loop.

The results of these codes are given in the results part.

2.2.2 SECOND PART

The purpose of this part was to implement the functions: addition, subtraction, multiplication, and division.

We chose the values we will use as:

```
            .data           ; Data memory
Avar        .word   8
Bvar        .word   16
Cvar        .word   4
Dvar        .word   3
```

Listing 2: Data Section (Part 2)

End then we set up the registers we will use for these values as below:

```
Setup       mov.w Avar, r4  ; get varA to r4 = 8
            mov.w Bvar, r5  ; get varB to r5 = 16
            mov.w Cvar, r6  ; get varA to r4 = 4
            mov.w Dvar, r7  ; get varB to r5 = 3
```

Listing 3: Setup (Part 2)

Then, we called the functions as below:

```
;ADD call--------------
            ; push variables onto stack
            push.w  r4
            push.w  r5
            ; now parameters are ready for the fucntion call
            call    #m_add      ; add is called
            pop.w   r8          ; now result is at r6 = 24
            pop.w   r4          ; ???
;SUB call----------------
            ; push variables onto stack
```

```
11              push.w   r6
12              push.w   r7
13              ; now parameters are ready for the fucntion call
14              call     #m_sub       ; sub is called
15              pop.w    r9           ; now result is at r9 = 8
16
17 ;MUL call--------------
18              ; push variables onto stack
19              push.w   r6
20              push.w   r7
21              ; now parameters are ready for the fucntion call
22              call     #m_mul       ; sub is called
23              pop.w    r10           ; now result is at r10 = 240
24
25
26 ;DIV call--------------
27              ; push variables onto stack
28              mov.w    Avar, r4   ; get varA to r4 = 8
29              mov.w    Cvar, r6   ; get varA to r4 = 20
30              push.w   r4
31              push.w   r6
32              ; now parameters are ready for the fucntion call
33              call     #m_div     ; sub is called
34              pop.w    r11      ; now result is at r11 = 1
```
Listing 4: Function Calls (Part 2)

Line 3, 4: Push r4 (8) and r5 (16) to stack.

Line 6: Call the subroutine *m_add*. Push the address of the next instruction to stack.

Line 7, 8: After returning from the subroutine pop the values r8 (16) and r4 (8) we pushed before. And result of *m_add* is in r8 (24).

Line 11, 12: Push r6 (4) and r7 (3) to stack.

Line 14: Call subroutine *m_sub*. Push the address of the next instruction to stack.

Line 15: After returning from the subroutine pop the result is in to r9 (1).

Line 19, 20: Push r6 (4) and r7 (3) to stack.

Line 21: Call the subroutine *m_mul*.

Line 23: After returning from subroutine pop the value 12 into r10 as result of *m_mul*.

Line 28, 29: r4 = 8 and r6 = 4 again.

Line 30, 31: Push r4 (8) and r6 (4) to stack.

Line 33: Call subroutine *m_div*.Push the address of the next instruction to stack.

Line 34: After returning from the subroutine pop the result to r11 (2).

After that, we wrote the function definitions.

```
m_add        mov.w 2(SP), r5 ; get varB to r5 from stack
             mov.w 4(SP), r4 ; get varA to r4 from stack
             add.w r4, r5    ; r5 = r4 + r5
             mov.w r5, 2(sp) ; move the result to stack
             ret
;------------------------
m_sub        mov.w 2(SP), r5 ; get varB to r5 from stack
             mov.w 4(SP), r4 ; get varA to r4 from stack
             sub.w r5, r4    ; r4 = r4 - r5
             mov.w r4, 2(sp) ; move the result to stack
             ret
;------------------------
m_mul        mov.w 2(SP), r5 ; get varB to r5 from stack
             mov.w 4(SP), r4 ; get varA to r4 from stack
             mov.w #0, r6    ; result keeper

iter_m       dec      r5           ; B--
             push.w   r5            ; to get original r5 from stack after return from add call
             cmp      #0, r5    ; do r5-0 = B-0
             jl       ret_mul   ; if B<0 go to return_mul
             push.w   r4
             push.w   r6
             call     #m_add
             pop.w    r6
             pop.w    r4      ;???
             pop.w    r5      ; get r5 back
             jmp      iter_m

ret_mul      mov.w r6, 4(SP)
             pop.w r5
             ret
;------------------------
m_div        mov.w #0, r6    ; result keeper
             mov.w 2(SP), r5 ; get varB to r5 from stack
             mov.w 4(SP), r4 ; get varA to r4 from stack

iter_d  ;maybe here(cmp   #0, r4    ; do r4-0 = A-0)
             push.w   r4
             push.w   r5
             call     #m_sub
             pop.w    r4
             pop.w    r14
             cmp      #0, r4    ; do r4-0 = A-0
             jl       ret_d   ; if A<0 go to ret_div
```

```
45                inc      r6
46                jmp      iter_d
47
48 ret_d          mov.w    r6, 2(SP)
49                ret
```

Listing 5: Functions (Part 2)

Line 1, 2: Load stack's second and third element to r5 (16) and r4 (8).

Line 3: Add r4 and r5, write the result on r5. (16 + 8 = 24)

Line 4: Write the result on stack's second place.

Line 5: Pop stack and return to the popped address.

Line 7, 8: Load r5 as 4, r4 as 3.

Line 9: Subtract r4 from r5 and write the result 1 to r5.

Line 11: Pop stack and return to the popped address.

Line 13, 14, 15: Load 4 to r5, 3 to r4, and 0 to r6.

Line 17: Here *iter_m* definition starts. It is used to create a while loop structure. Decrement r5.

Line 18: Push decremented r5 to stack.

Line 19, 20: If r5 less than 0 go to *ret_mul* to end the iteration.

Line 21, 22: It is not less than 0 so we continue with pushing first r4 and then r6 to stack.

Line 23: Call *m_add* subroutine and push the address of the next instruction to stack.

Line 24: After returning from the subroutine, pop the result of the subroutine to r6.

Line 25, 26: Pop the values r4 and r5 that we pushed before going to subroutine to restore them.

Line 27: Jump to *iter_m* (line 17) to do it again until r5 becomes zero (line 19, 20).

Line 29: In here top of stack looks like: original r5 (line 18), ret (return address for mull call), r7... So we need to put the result to stack's third position.

Line 30: Pop stack.

Line 31: Pop stack and return to the popped address.

Line 33: Load 0 to r6.

Line 34, 35: Load r5 as 4, and r4 as 8.

Line 37: Here *iter_d* definition starts. It is used to create a while loop structure.

Line 38, 39: Push r4 and r5 to stack.

Line 40: Call subroutine *m_sub* and push the address of the next instruction to stack.

Line 41, 42: After returning from the subroutine, pop the result of it to r4. And to empty the stack we pop the stack to r14 again (this was the original r4 we pushed before).

Line 43, 44: If r4 is less than 0, branch to *ret_d* (line 48).

Line 45, 46: If it is not, increment r6, and jump to the *iter_d* (line 37).

Line 48: Save the result to second position of stack.

Line 49: Pop stack and return to the popped address.

2.2.3 THIRD PART

This part aims to implement a subroutine that calculates the dot product of two vectors. We defined vector above `.text` as below.

```
            .data
array_A     .word   15, 3, 7, 5
lastA
array_B     .word   2, -1, 7, 3
lastB
```

Listing 6: Data Section (Part 3)

Then we wrote the rest of it.

```
Setup       mov     #array_A, r8
            mov     #array_B, r9
            mov.w   #0, r7     ;result keeper

dotproduct  push.w  @r8
            push.w  @r9
            inc     r8
            inc     r8
            inc     r9
            inc     r9
            call    #m_mul
            pop.w   r6
            add.w   r6, r7
            cmp     #lastA, r8
            jlo     dotproduct
            jmp     finish
;-----------------------
m_mul       mov.w   2(SP), r5    ; get varB to r5 from stack
            mov.w   4(SP), r4    ; get varA to r4 from stack
            mov.w   #0, r6       ; result keeper
            cmp     #0,  r5      ;do r5-0 = B-0
            jl      iter_m_m

iter_m_p    dec     r5           ; B--
            cmp     #0, r5       ; do r5-0 = B-0
            jl      ret_mul      ; if B<0 go to return_mul
            add.w   r4, r6
```

```
28              jmp      iter_m_p
29
30 iter_m_m     inc      r5          ; B++
31              cmp      #1, r5      ; do
32              jge      ret_mul     ; if B>=1 go to return_mul
33              sub.w    r4, r6
34              jmp      iter_m_m
35
36 ret_mul      mov.w    r6, 2(SP)
37              ret
38
39 finish       nop
```

Line 1, 2, 3: Load starting address of *array_A* to r8, and *array_B* to r9. Then, load 0 to r7 as it will keep the results.

Line 5: Here *dotProduct* definition starts. r8 points to current element of *array_A*. We push that element to stack.

Line 6: Similarly, we push the current element of *array_B* to stack.

Line 7, 8, 9, 10: Array elements are 16-bit. So we increment r8 and r9 two times in order to reach the next element.

Line 11: Call subroutine *m_mull* and push the address of the next instruction.

Line 12: After returning from the subroutine, we pop the result to r6.

Line 13: Add r6 to r7 and write it on r7.

Line 14, 15, 16: Check if r8 reached to end of array_A by comparing it with *lastA*. If r8 is less than lastA, jump to *dotProduct* (line 5). If not branch to finish.

Line 18, 19, 20: Load the current element of array_B to r5, the current element of array_A to r4, and 0 to r6 as it is the result keeper.

Line 21, 22: Compare element B with 0. If it is less than 0 then jump to *iter_m_m* function as it is a multiplication with a negative number.

Line 24: Decrement r5.

Line 25, 26: Compare r5 with 0. If it is less than 0 jump to *ret_mul* which means the iteration is ended.

Line 27: Otherwise, add element A to r6 and write it on r6.

Line 28: Jump to the start of the loop *iter_m_p* (line 24).

Line 30: Increment r5.

Line 31, 32: Compare r5 with 1. If it is greater than or equal to 1, jump to *ret_mul* which means the iteration is ended.

Line 33: Subtract element A from r6 and write it on r6.

Line 34: Jump to the start of the loop *iter_m_m* (line 30).

Line 36: Save the result to the second position of stack.

Line 37: Pop stack and return to the popped address.

Line 39: Deadloop.

3 RESULTS [15 points]

3.1 FIRST PART

- When we look at the memory address 0x200 it has 00FF at the end of the first iteration.
- Our table result for the first part:

Code	PC	R5	R10	R6	R7	SP	Content of the Stack
`mov #array, r5`	C00E	C038	0204	00C9	00C9	0400	FFFF
`mov #resultArray, r10`	C012	C038	0200	00C9	00C9	0400	FFFF
`mov.b @r5, r6`	C014	C038	0200	0001	00C9	0400	FFFF
`inc r5`	C016	C039	0200	0001	00C9	0400	FFFF
`call #func1`	C028	C039	0200	0001	00C9	03FE	C01A
`dec.b r6`	C02A	C039	0200	0000	00C9	03FE	C01A
`mov.b r6, r7`	C02C	C039	0200	0000	0000	03FE	C01A
`call #func2`	C034	C039	0200	0000	0000	03FC	C030 C01A
`xor.b #0FFh, r7`	C036	C039	0200	0000	00FF	03FC	C030 C01A
`ret`	C030	C039	0200	0000	00FF	03FE	C01A
`mov.b r7, r6`	C032	C039	0200	00FF	00FF	03FE	C01A
`ret`	C01A	C039	0200	00FF	00FF	0400	0000
`mov.b r6, 0(r10)`	C01E	C039	0200	00FF	00FF	0400	0000
`inc r10`	C020	C039	0201	00FF	00FF	0400	FFFF

Table 1: Part 1: Table Result

3.2 SECOND PART

Our results for the second part were

- For $Add(a, b) = a + b$: a = 8, b = 16 and Result:24
- For $Subtract(a, b) = a - b$: a = 4, b = 3 and Result:1
- For $Multiply(a, b) = a * b$: a = 4, b = 3 and Result:12

- For $Divide(a,b) = a/b(Integerdivision)$: a = 8, b = 4 and Result:2

3.3 THIRD PART

Result for the third part is $(15 * 2) + (3 * -1) + (7 * 7) + (5 * 3) = 91$

- Our result for third part: 91
- We were able to get the desired result.

4 DISCUSSION [25 points]

4.1 PART 1

We began with a program that used two functions (func1 and func2) to parse an integer array. Important lessons gained include:

Function Call Mechanism: Nestled calls are made possible by the call instruction's preservation of the return address on the stack. The way the software keeps track of return addresses while a function is being executed was made clear by looking at stack content.

```
call #func1
call #func2
ret
ret
...
```

Parameter Passing: Data was sent between routines using registers (such as R6, R7), exhibiting effective data handling without requiring a lot of stack actions.

```
call #func1
call #func2
...
```

Stack Dynamics: We were able to see how the stack expands and contracts with each function call and return by completing the stack table.

4.2 PART 2

We observed the following after implementing the arithmetic subroutines (Add, Subtract, Multiply, and Divide):

With stack-based parameter passing, modularity and reusability were ensured by pushing parameters onto the stack prior to executing subroutines and popping them once results were returned.

```
push.w  R4
push.w  R6
pop.w   R11
pop.w   R9
...
```

Building Complex Operations: The use of Add and Subtract as Multiply and Divide building blocks demonstrated how simple operations may be coupled to carry out intricate tasks.

```
m_add mov.w 2(SP), r5 ;
    mov.w 4(SP), r4 ;
    add.w r4, r5    ;

    mov.w r5, 2(sp) ;
    ret
;-----------------------
m_sub mov.w 2(SP), r5 ;
    mov.w 4(SP), r4 ;
    sub.w r5, r4    ;

    mov.w r4, 2(sp) ;
    ret
;-----------------------
m_mul mov.w 2(SP), r5 ;
    mov.w 4(SP), r4 ;

    mov.w #0, r6
        ...
```

Efficient Memory Use: We followed standard practices for assembly programming by minimising register usage by utilising the stack.

4.3 PART 3

In order to perform the dot product of two vectors recursively, complex concepts like:

In assembly, recursive logic involves storing the return address and intermediate values on the stack for every recursive call. Results spread back through the call chain after the function ended when the basic case (i = N) was reached.

```
call #func1
ret
call #func2
ret
...
```

Effective parameter handling made it possible for the recursive function to run smoothly on several data points by passing addresses and indices across the stack.

Subroutine Reusability: The advantages of modular assembly programming were demonstrated by the reuse of the arithmetic operations carried out in part 2.

5 CONCLUSION [10 points]

- In this experiment we mainly learned about how to use the stack for parameters for functions and how to handle calling a function in a function.
- We mostly struggled with the return from the functions as we forgetted to empty the stack until the return address, however that was easily fixed by running over each line with the debugger.

REFERENCES

[1] Microcomputer Lab. Micro_experiment_4. *Lab Booklet*, 2024.