Experiment 3

Sevim Eftal Akşehirli, 150190028
Hacer Yeter Akıncı, 150200006
Büşra Özdemir, 150200036
Aslı Yel, 150200054

***Abstract*—This experiment aims to strengthen understanding of MSP430 board and assembly coding through three parts. The first involves creating code for the modulus operation (mod(A,B)), saving the outcome in a register. Next, assembly code is crafted to generate an array of 50 numbers divisible by 3 or 4, using memory allocation directives. The third part focuses on reversing the initial array to create a new list, showcasing the reverse order of values stored in the original array.**

***Index Terms*—keywords like LEds, Inputs, Modulus, Arrays, Memory, Registers etc.**

I. Introduction

This experiment is all about learning assembly coding and how to work with the MSP430 board. Understanding modulus operations is essential for calculations and lays the foundation for complex programming. Using the MSP430 board, we'll practice manipulating registers and allocating memory—essential skills in making embedded systems. This experiment introduces us to low-level programming through MSP430 and assembly language. It provides essential insights into how computers efficiently function and interact with hardware. Despite the existence of various other technologies and languages, this learning is crucial for navigating embedded system programming.

II. Materials and Methods

Listing 1: Assembly Code for Part1

```
setup    mov.w  #122d , R8   ;A
       mov.w   #10d , R9 ;B

modulus    sub    R9, R8
        cmp    R9, R8
        jge    modulus
        jl     end

end       jmp    end
```

PART1:

Setup: This section initializes R8 with 122 and R9 with 10.

Modulus: This section is a loop that calculates the modulus operation (R8 % R9) until R8 is less than R9.

End: It is an infinite loop that stops the process.

Listing 2: Assembly Code for Part2

```
setup    mov.w #1d, R8 ;iteration
       mov.w #arr , R12 ;array
       mov.b #0d, R10 ;count

mainloop1 mov.w R8, R6
       mov.w #3d, R7
       jmp    modulo

mainloop2 mov.w R8, R6
       mov.w #4d, R7
       jmp    modulo

check    cmp    #0d, R6
       jeq    load
       cmp    #4, R7
       jeq    iterate
       jmp    mainloop2

load     mov.w R8, 0(R12)
       inc    R10
       cmp    #50d, R10
       jeq    end
       add    #2d, R12
       jmp    iterate

iterate   inc    R8
       jmp    mainloop1

modulo    sub    R7, R6
       cmp    R7, R6
       jge    modulo
       jl     check

end       jmp    end

     .data
arr    .space  100
```

PART2:

Setup: R8 represents the iteration and it is initialized with a value of 1. R12 keeps the first memory address of the array named arr. R10 is initialized to 0 and acts as a counter.

Mainloop1 and Mainloop2: These two labels load R7 with 3 and 4 to check the R8 is divisible 3 or 4 respectively.

Modulo: This section is a loop that calculates the modulus operation (R6 % R7) until R6 is less than R7 and when it is smaller, it goes to the check section.

Check: This section checks the R6 value first. If it is equal to 0 then it goes to the load section. Otherwise, it checks the R7 value with 4 and if it equals, it goes to iterate otherwise goes to mainloop2.

Iterate: This part increments R8 for the next iteration

Load: If the number is divisible by 3 or 4, it is stored in memory starting from the first address of the array. When the array size reaches 50, it goes to the end loop.

End: It is an infinite loop that stops the process.

Listing 3: Assembly Code for Part3

```
setup     mov.w #1d, R8  ;iteration
          mov.w #arr, R12  ;array
          mov.b #0d, R10  ;count
          mov.w #0d, R13  ; reverse count
          mov.w #arr2, R14  ;array2

mainloop1 mov.w R8, R6
          mov.w #3d, R7
          jmp   modulo

mainloop2 mov.w R8, R6
          mov.w #4d, R7
          jmp   modulo

check     cmp   #0d, R6
          jeq   load
          cmp   #4d, R7
          jeq   iterate
          jmp   mainloop2

load      mov.w R8, 0(R12)
          inc   R10
          cmp   #50d, R10
          jeq   reverse
          add   #2d, R12
          jmp   iterate

iterate   inc   R8
          jmp   mainloop1

modulo    sub   R7, R6
          cmp   R7, R6
          jge   modulo
          jl    check

reverse   mov.w 0(R12), R11
          mov.w R11, 0(R14)
          inc   R13
          cmp   #50d, R13
          jeq   end
          sub   #2d, R12
          add   #2d, R14
          jmp   reverse

end       jmp   end

        .data
arr     .space  100

        .data
arr2    .space 100
```

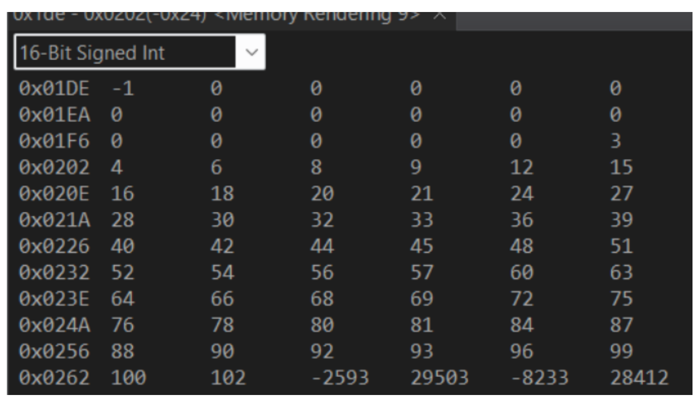
PART3:

As in Part 2, an array of size 50 called arr is created, consisting of numbers divisible by 3 and 4. The array we created in the reverse section is stored in a new array named arr2, in the reverse form of the arr array.

III. Results

For the first part, we wrote a code that calculates the modulus operation. For example, when we used the numbers 122 and 10, the code calculated the modulus and found 2. This answer was saved in register R8. We have obtained the result just as expected.

For the second part, we created another code to calculate which numbers, starting from 1, can be divisible by 3 or 4. These numbers (up to 50) were kept in an array we named 'arr'. The code did this by repeatedly checking if a number could be divisible by 3 or 4. We looked at the memory, and everything was there exactly as in the photo and as expected from us.



16-Bit Signed Int

0x01DE	-1	0	0	0	0	0
0x01EA	0	0	0	0	0	0
0x01F6	0	0	0	0	0	3
0x0202	4	6	8	9	12	15
0x020E	16	18	20	21	24	27
0x021A	28	30	32	33	36	39
0x0226	40	42	44	45	48	51
0x0232	52	54	56	57	60	63
0x023E	64	66	68	69	72	75
0x024A	76	78	80	81	84	87
0x0256	88	90	92	93	96	99
0x0262	100	102	-2593	29503	-8233	28412

Fig. 1: Result of Part-2

For part 3, after filling in 'arr' like part 2, we made a new list called 'arr2' with the same numbers but in reverse order. The code successfully flipped the order, and when we checked the memory, 'arr2' had the numbers in reverse. This means the code worked just as expected.

IV. Discussion and Summary

The first part consisted of a short modulus operation calculation. We wrote easily. The part that challenged us the most in this experiment was the second part. We needed to create an array containing 50 positive numbers divisible by 3 and 4. We wanted to find the addresses of the array we created and check its accuracy in memory, but we couldn't figure out how to do it for a while. We finally did it and the results we got were exactly as we should have gotten. In the third part, we saved the reverse version of the array we created in the second part into a new array. This wasn't a very difficult process and we got it done.