ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 351E

MICROCOMPUTER LABORATORY EXPERIMENT REPORT

EXPERIMENT NO : 4

EXPERIMENT DATE : 04.12.2024

LAB SESSION : WEDNESDAY - 12.30

GROUP NO : G8

GROUP MEMBERS:

150200009 : Vedat Akgöz

150200097 : Mustafa Can Çalışkan

150200016 : Yusuf Şahin

SPRING 2024

Contents

1 Introduction

This experiment aimed to enhance our practical understanding of function calls and stack usage in microcomputer systems. By analyzing stack operations and exploring the differences between CALL and JMP instructions, we deepened our understanding of the function call mechanism and stack memory management.

2 Materials and Methods

2.1 Part 1

In Part 1, we analyzed a simple program that processes an array of 8-bit integers using several function calls. The program stores the results in a memory location. We executed the program step by step in Debug Mode of CSS and completed the provided table for the first iteration of the main loop. The program code and table were provided as references. The completed tables are shown below:

Code	PC	R5	R10	R6	R7	SP
mov #array, r5	0XC010	0XC038	0X0204	0X00C9	0X00C9	0X0400
mov #resultArray, r10	0XC012	0XC038	0X0200	0X00C9	0X00C9	0X0400
mov.b @r5, r6	0XC014	0XC038	0X0200	0x0001	0X00C9	0X0400
inc r5	0XC016	0XC039	0X0200	0x0001	0X00C9	0X0400
call #func1	0XC028	0XC039	0X0200	0x0001	0X00C9	0X03FE
dec.b r6	0XC02A	0XC039	0X0200	0x0000	0X00C9	0X03FE
mov.b r6, r7	0XC02C	0XC039	0X0200	0x0000	0X0000	0X03FE
call #func2	0XC034	0XC039	0X0200	0x0000	0X0000	0X03FC
xor.b #0FFh, r7	0XC036	0XC039	0X0200	0x0000	0X00FF	0X03FC
ret	0XC030	0XC039	0X0200	0x0000	0X00FF	0X03FE
mov.b r7, r6	0XC032	0XC039	0X0200	0X00FF	0X00FF	0X03FE
ret	0XC01A	0XC039	0X0200	0X00FF	0X00FF	0X0400
mov.b r6, 0(r10)	0XC01E	0XC039	0X0200	0X00FF	0X00FF	0X0400
inc r10	0XC020	0XC039	0X0201	0X00FF	0X00FF	0X0400

Code	Content of the Stack
mov #array, r5	
mov #resultArray, r10	
mov.b @r5, r6	
inc r5	
call #func1	0XC01A
dec.b r6	0XC01A
mov.b r6, r7	0XC01A
call #func2	0XC01A - 0XC030
xor.b #0FFh, r7	0XC01A - 0XC030
ret	0XC01A
mov.b r7, r6	0XC01A
ret	
mov.b r6, 0(r10)	
inc r10	

2.2 Part 2

In Part 2, we implemented subroutines for basic arithmetic operations: addition, subtraction, multiplication, and division. Parameters were passed through the stack, and results were retrieved similarly. Additionally, we used the Add and Subtract subroutines to implement Multiply and Divide. Below is the assembly code for Part 2:

```
Setup
    mov #array, r5
    mov #resultArray, r10

Mainloop

    mov #20, r6
    mov #5, r7
    push r7
    push r6
    call #Divide
    pop r6
    jmp finish

Add:
```

```
    pop r5
    pop r8
    pop r9
    add r8, r9
    push r9
    push r5
    ret

Subtract:
    pop r5
    pop r8
    pop r9
    sub r9, r8
    push r8
    push r5
    ret


Multiply:
    pop r5
    pop r6
    pop r7
    push r5
    mov #0, r9
    mov #0, r8
    cmp #0,r6
    jl  Mul_R6
    jmp Mul_R7

Mul_R6:
    xor #0FFFFh, r6
    inc r6
    xor #0FFFFh, r8
    cmp #0,r7
    jl  Mul_R7
    jmp Multiply_Checked
Mul_R7
```

```
    cmp #0,r7
    jge  Multiply_Checked
    xor #0FFFFh, r7
    inc r7
    xor #0FFFFh, r8
    jmp Multiply_Checked

Multiply_Checked
    push r8
    jmp Multiply_Loop

Multiply_Loop:
    push r9
    push r6
    call #Add
    pop r9
    dec r7
    cmp #0, r7
    jne Multiply_Loop
    pop r8
    pop r5

    cmp #0, r8
    jeq Multiply_Done
    xor #0FFFFh, r9
    inc r9
    jmp Multiply_Done

Multiply_Done:
    push r9
    push r5
    ret

Divide:
    pop r5
    pop r6
    pop r7
```

```
    push r5
    mov #1, r4
    mov #0, r9
    mov #0, r8
    jmp Div_R6

Div_R6:
    cmp #0,r6
    jge  Div_R7
    xor #0FFFFh, r6
    inc r6
    xor #0FFFFh, r8
    jmp Div_R7
Div_R7
    cmp #0,r7
    jge  Divide_Checked
    xor #0FFFFh, r7
    inc r7
    xor #0FFFFh, r8
    jmp Divide_Checked

Divide_Checked
    cmp r6, r7
    jge Divide_Done
    push r8
    mov  #0, r8
    mov  r6, r9
    jmp Divide_Loop


Divide_Loop:
    push r7
    push r9
    call #Subtract
    pop r9
    inc r4
    cmp r9, r7
```

```
    jl Divide_Loop
    pop r8
    pop r5
    cmp #0, r8
    jeq Divide_Done
    xor #0FFFFh, r8
    inc r8
    jmp Divide_Done

Divide_Done
    push r4
    push r5
    ret


; Integer array
array .byte 1, 0, 127, 55
lastElement

finish
    nop
```

2.3 Part 3

Due to time constraints, we were unable to complete Part 3. The objective was to implement a recursive subroutine for calculating the dot product of two vectors. The recursive formula provided was:

$$\text{dot}(A, B, i, N) = \begin{cases} A[i] \cdot B[i] + \text{dot}(A+1, B+1, i+1, N), & \text{if } i < N \\ 0, & \text{if } i = N \end{cases}$$

Given arrays were:

```
array A: 15, 3, 7, 5
array B: 2, -1, 7, 3
```

Unfortunately, we could not proceed further with this task.

3 Results

The results of our analysis and implementation are as follows:

- Part 1: The completed table shows the progression of register values and stack content during the first iteration of the main loop.
- Part 2: Successfully implemented arithmetic subroutines for addition, subtraction, multiplication, and division. The assembly code is included in the report.
- Part 3: Due to time constraints, this part was not completed.

4 Discussion

In this experiment, we learned the practical implementation of stack-based function calls and the importance of properly managing stack memory during function execution. Part 1 provided valuable insights into how registers and the stack interact during function calls. Part 2 demonstrated the utility of subroutines and stack operations in implementing complex operations.

However, time constraints prevented us from completing Part 3. This highlights the need for better time management and preparation when approaching recursive problems in assembly programming.

5 Conclusion

This experiment successfully enhanced our understanding of stack and subroutine usage in assembly programming. Despite not completing Part 3, we gained significant practical experience in managing function calls, stack operations, and implementing arithmetic subroutines. Future efforts should focus on addressing recursive problems more efficiently.

REFERENCES