ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 351E

MICROCOMPUTER LABORATORY EXPERIMENT REPORT

EXPERIMENT NO : 3

EXPERIMENT DATE : 27.11.2024

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

In this experiment, we delve deeper into the intricacies of assembly programming on the MSP430 board, focusing on more complex algorithms and their implementation. Part 2 introduces the modulus operation using steps inspired by the Russian Peasant Division (RPD), an efficient method particularly suited for binary systems. This approach allows us to perform division-related tasks by leveraging binary manipulations, making it highly relevant for low-level programming. In Part 3, we explore hashing mechanisms by writing assembly code to hash student IDs directly into memory. Using the `.space` directive, we allocate memory for this operation, demonstrating practical memory management in assembly. These exercises aim to solidify our understanding of MSP430 syntax and functionality, bridging theoretical concepts with hands-on applications.

2 MATERIALS AND METHODS

2.1 Part 2

In Part 2, we implemented the modulus operation using steps inspired by the Russian Peasant Division (RPD) algorithm, which is highly efficient for binary systems. The process involves initializing variables C and D with the divisor B and dividend A, respectively. We double C until it surpasses $A/2$ and iteratively subtract C from D whenever $D \geq C$, halving C after each step. This iterative process calculates the modulus and identifies powers of 2 to represent A in terms of C. Below is the commented assembly implementation:

```
.data
used_powers .space 20     ; Reserve 20 bytes for tracking powers used
index       .word 0       ; Index variable
sum_powers  .word 0       ; Variable to sum powers

.text
.global main
main:
    mov.w   #151, R4      ; Load dividend A = 151 into R4
    mov.w   #8, R5        ; Load divisor B = 8 into R5
    mov.w   R5, R6        ; Initialize C = B in R6
    mov.w   R4, R7        ; Initialize D = A in R7
    mov.w   #0, R8        ; Clear exponent tracker R8
```

```
    mov.w   #1, R12       ; Initialize power accumulator
    mov.w   #0, R10       ; Clear sum of powers

loop1:
    cmp.w   R6, R7        ; Compare C with A
    jlo     loop1_end     ; Exit loop if C > A
    inc.w   R8            ; Increment exponent
    add.w   R6, R6        ; Double C
    jmp     loop1         ; Repeat loop

loop1_end:
    rra.w   R6            ; Halve C to set correct upper limit
    dec.w   R8            ; Adjust exponent for the loop

loop2:
    cmp.w   R5, R7        ; Compare D with B
    jl      loop2_end     ; Exit if D < B

    cmp.w   R6, R7        ; Check if D >= C
    jlo     loop1_end     ; If not, skip subtraction
    sub.w   R6, R7        ; D = D - C
    mov.w   R8, R14       ; Copy exponent to R14

sume_loop:
    dec.w   R14           ; Decrement exponent
    rla.w   R12           ; Left shift power accumulator
    cmp.w   #0, R14       ; Check if exponent is zero
    jne     sume_loop     ; Repeat until zero
    cmp.w   #1, R12       ; Check if current power is 1
    jeq     enough        ; If so, exit

    add.w   R12, R10      ; Add power to sum
    mov.w   #1, R12       ; Reset power accumulator
enough:
    nop                   ; No operation (placeholder)

skip_subtract:
```

```
    cmp.w   #0, R8        ; Check if exponent is zero
    jeq     skip_shift    ; Skip if true

skip_shift:
    jmp     loop2         ; Continue loop
loop2_end:
    nop                   ; No operation (end of program)
```

This code demonstrates efficient use of binary operations for calculating the modulus and tracking the decomposition of the dividend into powers of the divisor. However, unfortunately we had lost our most of the time to deal with storing redundant data given in the experiment file and this brought us time problems for the Part 3.

2.2 Part 3

As we said, due to redundant and unnecessary work in the Part 2, led us to fail to complete Part 3, however; we do not want to leave it as it is and therefore, here is our code:

```
.data
hash_table .space 58       ; Allocate 58 bytes for the hash table
split_ids  .space 6        ; Allocate 6 bytes for storing split ID values (150, 200, (

.text
.global main
main:
    ; Load split ID values into memory (manually or programmatically)
    mov.w   #123, R4       ; First part of ID (ABC)
    mov.w   #456, R5       ; Second part of ID (DEF)
    mov.w   #789, R6       ; Third part of ID (GHI)

    ; Load the base address of the hash table
    mov.w   #hash_table, R10

    ; Process first ID part (ABC)
    call    #hash_id       ; Hash ABC
    call    #store_hash    ; Store ABC in the table

    ; Process second ID part (DEF)
```

```
    mov.w   R5, R4          ; Load DEF into R4
    call    #hash_id        ; Hash DEF
    call    #store_hash     ; Store DEF in the table

    ; Process third ID part (GHI)
    mov.w   R6, R4          ; Load GHI into R4
    call    #hash_id        ; Hash GHI
    call    #store_hash     ; Store GHI in the table

    ; End program
    nop

hash_id:
    mov.w   #29, R7         ; Load divisor (29)
    div.w   R7, R4          ; R4 = R4 / 29, remainder in R5
    mov.w   R5, R8          ; Move remainder (hash index) to R8
    ret

store_hash:
    add.w   R8, R10         ; Calculate address: hash_table + index
    cmp.b   #0, 0(R10)      ; Check if the location is empty
    jeq     store_value     ; If empty, store value
    inc.w   R8              ; Increment index for linear probing
    jmp     store_hash      ; Repeat check

store_value:
    mov.w   R4, 0(R10)      ; Store ID part at calculated address
    ret
```

3 RESULTS

In this experiment, we worked on advanced assembly programming using the MSP430 microcontroller. In Part 2, we successfully implemented the Russian Peasant Division (RPD) algorithm to compute modulus operations using binary arithmetic. However, managing redundant data, as described in the experiment file, took more time than anticipated and impacted our ability to complete Part 3 during the lab session.

For Part 3, we wrote assembly code to hash student IDs into a memory-allocated hash table. Although we could not test it in the lab, the code incorporates a hash function and resolves collisions with linear probing. Despite these challenges, the experiment provided a deeper understanding of assembly programming and binary systems.

4 DISCUSSION

The biggest challenge we faced was time management. In Part 2, extra time was spent handling redundant data as required by the experiment file, which delayed progress and affected the completion of Part 3. While we managed to write the hashing code after the lab session, we could not test it thoroughly.

This experiment highlighted the importance of efficient planning and focusing on core tasks to avoid time-related issues. Simplifying the instructions or extending lab hours could improve the workflow, allowing for a more balanced approach to all tasks.

5 CONCLUSION

This experiment enhanced our understanding of assembly programming and the MSP430 microcontroller. Part 2 successfully demonstrated modulus computation using the RPD algorithm, and Part 3 allowed us to apply hashing and memory management concepts.

Despite time constraints, we gained valuable insights into low-level programming and the importance of effective time management. This experience will help us better approach similar challenges in the future, making it a valuable learning opportunity overall.

REFERENCES