ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 351E

MICROCOMPUTER LABORATORY EXPERIMENT REPORT

EXPERIMENT NO	:	3
EXPERIMENT DATE	:	29.11.2024
LAB SESSION	:	FRIDAY - 14.30
GROUP NO	:	G9

GROUP MEMBERS:

150220723 : Abdulsamet Ekinci

150210906 : Emil Huseynov

150220902 : Nahid Aliyev

FALL 2024

Contents

1 INTRODUCTION [10 points]

This experiment focused on advanced assembly programming using the MSP430 microcontroller. Specifically, we implemented the Russian Peasant Division (RPD) algorithm for modulus calculation and a hashing algorithm with linear probing for collision resolution. The purpose of the experiment was to explore complex low-level operations such as binary multiplication, subtraction, and memory management through assembly. Additionally, we learned how to handle collisions in hash tables using linear probing and implemented efficient memory allocation for storing hashed values.

2 MATERIALS AND METHODS [40 points]

This section describes the hardware, software, and methods used in the experiment.

2.1 Hardware

- **MSP430G2553 Microcontroller:** A low-power microcontroller with 16-bit RISC architecture.
- **LEDs:** Connected to the output pins for displaying the results of the calculations.
- **Buttons:** Configured as inputs for triggering actions based on conditions.

2.2 Software

- **Code Composer Studio (CCS):** Used to write, debug, and flash assembly programs onto the MSP430.

2.3 Part 1: Russian Peasant Division (RPD) for Modulus Calculation

Objective: Implement the Russian Peasant Division algorithm to calculate the modulus of two numbers.

Procedure:

- **Step 1: Initialization** - Load the dividend and divisor into registers R4 and R5. Initialize ‘C‘ and ‘D‘ to the divisor and dividend values, respectively.

```
mov.w   #150d, R4     ; A = 150 (dividend)
mov.w   #10d, R5      ; B = 10 (divisor)
mov.w   R5, R7        ; C = B
```

```
mov.w   R4, R8          ; D = A
```

- **Step 2: Multiply C by 2** - Continuously double the value of 'C' until it exceeds half of 'A'.

```
First_Loop:
    cmp.w  R7, R9       ; Compare A with C
    jlo    Second_Loop  ; Exit if C > A/2
    add.w  R7, R7       ; C = C * 2
    jmp    First_Loop   ; Repeat loop
```

- **Step 3: Subtract C from D** - If 'D' is greater than or equal to 'C', subtract 'C' from 'D' and halve 'C' until 'D' is smaller than 'C'.

```
Second_Loop:
    cmp.w  R5, R8       ; Compare D with B
    jl     End          ; Exit if D < B
    cmp.w  R7, R8       ; Compare D with C
    jlo    Halve_C      ; If D < C, halve C
    sub.w  R7, R8       ; D = D - C
    jmp    Second_Loop
Halve_C:
    rra.w  R7           ; C = C / 2
    jmp    Second_Loop
End:
    mov.b  R8, &P1OUT   ; Display remainder D on Port 1 LEDs
    jmp    End          ; Stay here (halt program)
```

Challenges and Observations:

- The condition checking and iteration control for the doubling of 'C' and halving of 'C' were a bit tricky. Ensuring that the conditions for the loops were set up correctly and efficiently, especially when modifying 'C', required careful attention to avoid infinite loops or incorrect results.
- Debugging the logic to handle when 'D' becomes smaller than 'C' was essential, as any misstep would cause incorrect results in the modulus operation.

2.4 Part 2: Hashing with Linear Probing

Objective: Implement a hashing algorithm using linear probing to resolve collisions and store student ID values.

Procedure:

- **Step 1: Initialize the hash table** - Allocate space for the hash table and initialize it with '-1' values to signify empty slots.

```
.data
hash_table .space 58     ; Allocate space for the hash table
split_ids  .word 123, 7, 789

.text
mov.w   #hash_table, R5
mov.w   #29, R6          ; Size of the hash table (29 slots)
mov.w   #-1, R7          ; Value indicating empty slots
InitializeHashTable:
    mov.w   R7, 0(R5)    ; Initialize each slot to -1
    add.w   #2, R5
    dec.w   R6
    jnz     InitializeHashTable
```

- **Step 2: Hashing the student ID parts** - The student ID is split into parts, and the hash function is applied (modulo 29) to each part to determine the appropriate index in the table.

```
mov.w   #split_ids, R4
mov.w   #hash_table, R5
mov.w   #3, R6

HashLoop:
    mov.w   @R4+, R7
    clr.w   R8
    mov.w   R7, R9

HashFunction:
    cmp.w   #29, R9      ; Apply modulo 29 operation
    jl      HashDone
    sub.w   #29, R9
    jmp     HashFunction

HashDone:
    mov.w   R9, R8
```

- **Step 3: Handle collisions with linear probing** - If the calculated index is occupied, the algorithm moves to the next available slot using linear probing.

```
mov.w   R5, R10
rla.b   R8
add.w   R8, R10

CollisionCheck:
    cmp.w   #-1, 0(R10)   ; Check if the slot is empty
    jeq     PlaceInTable
    add.w   #2, R10
    jmp     CollisionCheck

PlaceInTable:
    mov.w   R7, 0(R10)    ; Store value in the table
    dec.w   R6
    jnz     HashLoop
```

Challenges and Observations:

- The condition handling within the collision check required special attention. Implementing linear probing meant ensuring that the program properly checks for empty slots and does not overwrite existing data.
- Properly handling the modulo operation for hashing was a bit tricky, as we needed to ensure that the value stays within bounds (0-28).
- Debugging the hash function and collision logic was challenging, especially when trying to ensure that collisions were resolved correctly without overwriting data or accessing invalid memory locations.

3 RESULTS AND DISCUSSION [30 points]

Both algorithms, the Russian Peasant Division and the hashing with linear probing, were successfully implemented and tested on the MSP430 microcontroller. The results of the modulus calculation were displayed on the LEDs connected to Port 1, and the hash table was properly populated with the hashed student ID values.

However, we faced some challenges during implementation:

- The condition checking in loops and ensuring correct iteration was particularly difficult in both parts of the experiment.

- Debugging the program to handle edge cases in the modulus operation (e.g., when ‘D ¡ C‘ or handling ‘A‘ and ‘B‘ values correctly) was a non-trivial task.
- The collision resolution in the hash table using linear probing required careful checks to avoid overwriting data in case of a collision.

In future experiments, we will focus on improving our understanding of condition handling and loop control structures in assembly programming, as these are fundamental for efficient low-level programming.

4 CONCLUSION [20 points]

This experiment helped us understand the intricacies of low-level programming and the importance of proper condition handling and loop control. We successfully implemented the Russian Peasant Division for modulus calculation and a hashing algorithm with linear probing for collision resolution. Despite some difficulties with conditions and debugging, we managed to overcome the challenges and complete the experiment successfully.