**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** | : | G7 |

**GROUP MEMBERS:**

150210087 : HİLAL KARTAL

150210100 : SELİN YILMAZ

150210067 : ALPER DAŞGIN

**FALL 2024**

# Contents

# 1 INTRODUCTION [10 points]

Implementing the Russian Peasant Division (RPD) to determine the modulus of two numbers was the first job. This algorithm is perfect for microcontrollers since it can execute operations in binary systems with efficiency. Initialising variables, multiplying and dividing by powers of two iteratively, and subtracting values to find the remainder were all part of the procedure. We saw that complex arithmetic may be effectively performed by combining basic operations like shifts and subtractions. This assignment focused on conditional operations, looping constructs, and variable initialisation in Assembly.

We used a hashing approach to store and handle divided student ID information in a hash table for the second task. Three segments of the IDs (such as ABC, DEF, and GHI) were split, and a hash function was applied using the modulus operation with a divisor of 29. A designated memory space's storage places were established using the computed hash values. We employed linear probing to handle collisions, making sure that values were positioned correctly even when a hash table entry was filled. In addition to highlighting sophisticated ideas like modular arithmetic, memory addressing, and collision handling, this task also demonstrated memory allocation utilising directives like space.

These assignments gave us practical experience with advanced Assembly programming principles, such as memory management, algorithmic design, and making the most of the MSP430's capabilities. Our knowledge of the usefulness of assembly language in handling practical computer issues was significantly strengthened by this exercise.

# 2 MATERIALS AND METHODS [40 points]

## 2.1 MATERIALS

- MSP430
- Selin's Laptop

## 2.2 METHODS

### 2.2.1 FIRST PART

In this part, our purpose is to perform some steps of *Russian Peasant Division* for modulus operation. A is the dividend and B is the divisor. So, the steps look like this:

- Create variables C and D, C = B, and D = A at the beginning.
- While C $\leq$ A/2, multiply C by 2
- While B $\leq$ D:
  - Subtract C from D when D $\geq$ C
  - Divide C by 2.
- Finally, D is the remainder. C is the return.

To implement it, we first initialized some data.

```
            .data
Avar        .word   151
Bvar        .word   10
Cvar        .word   0x00
Dvar        .word   0x00
```

Listing 1: RPD

We used words in here because in the second part 32 bits are needed.

Then, in the setup part *SetupP1*, we put them in the registers we will use.

```
1 SetupP1     mov.w Avar, R12
2             mov.w Bvar, Cvar
3             mov.w Avar, Dvar
4             mov.w Bvar, R13
5             mov.w Dvar, R5
6             mov.w   Cvar, R4
```

Listing 2: SetupP1

Line 1: R12 is equal to Avar
Line 2: Cvar is equal to Bvar
Line 3: Dvar is equal to Avaar
Line 4: R13 is equal to Bvar
Line 5: R5 is equal to Dvar
Line 6: R4 is equal to Cvar

Then, according to the Russian Peasant Division's pseudocode above we implemented the program.

```
_start      mov.w   R12, R11
            rra   R11

comparison_c
```

```
5              cmp    R11, R4
6              jl     _multiply_c
7              jeq      _multiply_c
8
9  comparison_d
10             cmp    R5, R13
11             jl     _subtract_d
12             jeq    _subtract_d
13             ;D = R5 is the remainder now
14             sub.w  R5, R12
15
16             jmp    end
17
18 _multiply_c
19             rla    R4
20             jmp    comparison_c
21
22 _subtract_d
23             cmp    R4, R5
24             jl     _divide_c
25             sub.w  R4, R5
26             rra    R4
27             jmp    comparison_d
28
29 _divide_c
30             rra    R4
31             jmp    _subtract_d
32
33 end         jmp    end
```

Listing 3: Body

Line 1, 2: R12 (A) is moved to R11 and R11 is rotated right. Means, R11 = A / 2

Line 5, 6, 7: If $C \leq A/2$, branch to *_multiply_c* (line 17).

Line 10, 11, 12: If $B \leq D$, branch to *_multiply_d* (line 21).

Line 14: R5 is the remainder now, and it is subtracted from A.

Line 16: Branch to *end*.

Line 19: Multiply C by 2.

Line 20: Go to *comparison_c* (line 4) (it is a loop).

Line 23,24: If $D < C$, branch to *_divide_c* (line 29).

Line 25: Subtract C from D

Line 26: Divide C by 2.

Line 27: Branch to *comparison_d* (line 9).

Line 30: Divide C by 2.

Line 31: Branch to *_subtract_d* (loop).

Line 33: Dead loop.

#### 2.2.2 SECOND PART

In this part we are asked to implement a hash function on our student IDs with 29. So steps look like this:

- Divide the student ID into 3 part.
- Apply hash on the divided parts.
- Hash happens in two parts:
    1. First, the modulus of the split ID is found
    2. Second, according to modulus, Id is placed on the corresponding index.
        - If the placing tries to place ID on a already taken index linear probing is applied.
        - *LinearProbing*: ID is placed on the next empty spot.

To implement it, we initialized some data. Here:

- We allocate 58 bytes for hash table.
- ID is split into parts Avar, Mvar, Nvar
- B is the modulus

```
            .data
hash        .space 58
Avar    .word 150
Mvar    .word 210
Nvar    .word 187
Bvar    .word 29
Cvar    .word   0x00
Dvar    .word 0x00
```

Listing 4: RPD

Then for our hash function:

- Initialize the values
- Call modulus function
- Call linear probing function (cfstorage)
- Move the ID part to resulting index.

### 2.2.3 Hash

```
hash1 mov   #hash, r10  ; load the start of hash

    mov.w Avar, R12
    mov.w Bvar, Cvar
    mov.w Avar, Dvar

    mov.w Bvar, R13

    mov.w Dvar, R5
    mov.w   Cvar, R4

    call      #modulus
    mov.w     Avar, 0(R10)


hash2 mov &hash, r10

    mov.w Mvar, R12

    mov.w Bvar, Cvar
    mov.w Avar, Dvar

    mov.w Bvar, R13

    mov.w Dvar, R5
    mov.w   Cvar, R4

    call  #modulus
    call  #cf_storage
    add.w r5, r5
    add.w r5, r10
    mov Mvar, 0(R10)

    mov &hash, r10


hash3 mov.w Nvar, R12
    mov.w Bvar, Cvar
    mov.w Avar, Dvar

    mov.w Bvar, R13

    mov.w Dvar, R5
```

```
44     mov.w   Cvar, R4
45
46     call      #modulus
47     call      #cf_storage
48     add.w   r5, r5
49     add.w     r5, r10
50     mov.w     Nvar, 0(R10)
51     jmp     end
```

Listing 5: Body

Line 1: Loads the start of hash to r10.

Line 2-10: Readies the parameters for the modulus function

Line 12: Call modulus

Line 13: Moves the ID value into hash array according to index

Line 16: Start of the second hash

- Only difference here is to call to linear prob function at line 29
- R5 is the return of linear probing it is it is aligned an added to R10.

Line 37: Start of the second hash

- Only difference here is to call to linear prob function at line 37
- R5 is the return of linear probing it is it is aligned an added to R10.

#### 2.2.4 Linear Probing

```
cf_storage  mov.w   #58, r1

      mov      R5, R12
      rla   R12
      rla   R12
      add      R12, R10
      mov      0(R10), R6

      cmp #0, r6
      jeq turn_back

          add.w   r5, r5
          cmp   r5, r10
          jl    cf_storage
            jmp   cf_storage
```

```

turn_back ret
```

Listing 6: RPD

This part did not work.

### 2.2.5 Modulus

```
1 modulus
2       mov.w   R12, R11
3       rra   R11
4
5 comparison_c
6       cmp   R11, R4
7       jl    _multiply_c
8
9 comparison_d
10      cmp   R5, R13
11      jl    _subtract_d
12      jeq   _subtract_d
13      sub.w R5, R12
14
15      jmp   end_modulus
16
17 _multiply_c
18      rla   R4
19      jmp   comparison_c
20
21 _subtract_d
22      cmp   R4, R5
23      jl    _divide_c
24      sub.w R4, R5
25      rra   R4
26      jmp   comparison_d
27
28 _divide_c
29      rra   R4
30      jmp   _subtract_d
31
32 end_modulus   ret
```

Listing 7: RPD

Modulus function is taken from the first part. Only Line 15 is changed to return from the function call.

# 3 RESULTS [15 points]

## 3.1 FIRST PART

We assigned 151 to $A$ (R11) and 8 to $B$ (R12). The output was observed as 8 in $C$ (R4) and 7 in $D$ (R5). To verify the system's correctness with other values, we assigned 150 to $A$ and 10 to $B$. The output was observed as 5 in $C$ and 0 in $D$.

## 3.2 SECOND PART

We assigned 150 to $Avar$, 021 to $Nvar$ and 187 to $Mvar$. At the first hash the result could be shown correctly at the memory but as for the second and third hashes we could not see the desired results as our linear probing function did not work.

# 4 DISCUSSION [25 points]

In the first section, we implemented an assembly program to calculate the modulus of a given dividend and divisor. We learned the following information from this task:

- **How to Compare and Branch:**
  We deepened our understanding of the `cmp` instruction. For example, the operation:

  ```
  cmp     R4, R5
  ```

  compares $R5$ with $R4$. When followed by a command like:

  ```
  jl to_somewhere
  ```

  it jumps to the specified location if $R5 < R4$. This helped us to comprehend loop structures in assembly programming and use them.

- **How to Divide and Multiplicate by 2:**
  We practiced implementing division and multiplication in assembly. Division was achieved using the arithmetic right rotate (`rra`) instruction, while multiplication was performed using the arithmetic left rotate (`rla`) instruction.

- **How to Subtract Numbers:**
  We learned the correct placement of the destination and source registers during subtraction. For example:

  ```
  sub R4, R5
  ```

  subtracts $R4$ from $R5$, with the result stored in $R5$.

# 5 CONCLUSION [10 points]

Through this exercise, we learnt how to use more advanced algorithms on the MSP430 microcontroller and developed a better comprehension of its fundamental operations. Using binary arithmetic, which involves repeated multiplication, division, and subtraction, we learnt how to quickly compute the modulus operation using the Russian Peasant Division technique. We used linear probing to handle collisions and a hashing approach to store divided student ID data in a memory hash table. The significance of effective memory management and addressing was highlighted by this challenge. At first, we had trouble comprehending the hashing method, particularly when it came to properly allocating memory with the.space directive and addressing values. We fixed these problems by closely examining the memory browser and placement operations. The beginning of

.SPACE: In the data section, a block of memory was allocated using this instruction. For instance, 58 bytes are set aside for our hash table in.space 58.
MOV.W: With the help of this instruction, we were able to precisely place and retrieve hash values by interacting with certain memory regions.
All things considered, this exercise improved our ability to solve Assembly programming problems, including memory management, algorithm implementation, and debugging strategies for practical uses.

## REFERENCES