Department of Electrical & Computer Engineering
ENCS2380 - Computer Organization and Microprocessor

# Project II (Assembly)

**Prepared by:** Mohamad Milhem
**ID number:** 1211053
**Instructor:** Dr. Abualseoud Hanani
**Date:** February 13, 2023

# Abstract

In this project, we aim to build an assembly program containing 3 procedures, each of which has specific work to do by modifying two strings.

This will improve our ability to read, write, and understand assembly language. Also will make us comfortable with using ARM instructions and also develop problem-solving and thinking skills.

# Contents

# 1 Theory

## 1.1 Assembly language

In computer programming, assembly language (or assembler language, or symbolic machine code), often referred to simply as Assembly and commonly abbreviated as ASM or asm, is any low-level programming language with a very strong correspondence between the instructions in the language and the architecture's machine code instructions. Assembly language usually has one statement per machine instruction (1:1), but constants, comments, assembler directives, symbolic labels of, e.g., memory locations, registers, and macros are generally also supported.

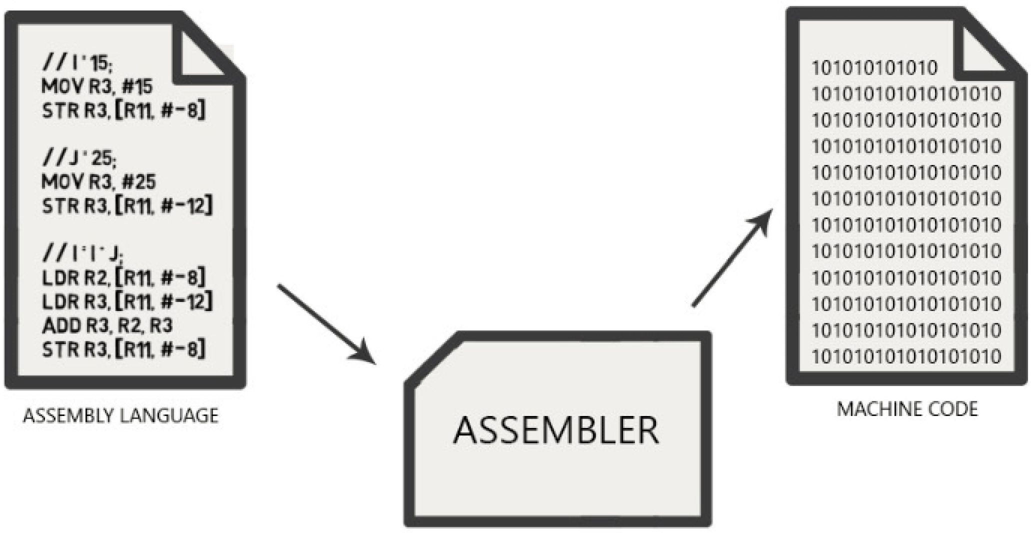


Figure 1.1: Assembly to Machine Code

# 2 Program Code

## 2.1 Procedure 1: TO SMALL

In this procedure, given a string, we want to convert all its characters to small letters. we start by loading $R_0$ with the string address that we want to read and we also load $R_1$ with the address we want to start writing the new lowercase string on.

```
TO_SMALL PROC

LOOP1
    LDRB R2, [R1]
    CMP R2, #0 ; the end of the string
    BEQ exitloop

    CMP R2, #0x20 ; skipping the space char
    BEQ next

    CMP R2, #0x5A ; ascii code for 'Z' if the char is less or equal to 'Z' it's capital, otherwise, it's small

    BLS Capital
    ; Already small
    B next
Capital
    ;CAPITAL, convert it to small
    ADD R2, #0x20
    ADD R3, #1
next

    STRB R2, [R0]  ; storing the small letter.

    ADD R0, R0, #1 ; increase the reading address pointer by 1
    ADD R1, R1, #1 ; increase the writing address pointer by 1

    B LOOP1
exitloop
    BX LR
    ENDP
```

Figure 2.1: TO SMALL procedure

The procedure starts with a loop "LOOP1", first we load the first character of the string on $R_2$, and we check the exit condition if this character was equal to 0, we exit.

Else, we continue by comparing it to the space character, if space were detected we have just to store it in the writing address and skip to the next character. If it's not a space, we compare it with 'Z' character, if the character were less or equal to 'Z' then it's a capital character. so we have to convert it to a small one by adding 32 to its value.

In the end, we store the small letter on the writing address and increase the reading and the writing address pointers by 1 to move to the next character.

### 2.1.1 Procedure 2: Count Common

In this procedure, given two strings, we want to count how many characters are common between them, the approach used here is using outer and two inner loops, the outer loop will iterate from 'a' to 'z' and in each iteration of the outer loop, the first inner loop will iterate over the first string and put a flag on if the character appears in the first string.

Similarly, in each iteration of the outer loop (for each character from 'a' to 'z'), the second inner loop will iterate over the second string and put a flag on if the character appears in the second string.

If the two flags were on in the same iteration (the same character appears in the first and second string), this means this character is a common character, so we increase the counter by 1.

**Note: this procedure is long so we will break it into parts.**

```
Count_Common PROC
    LDR R2, = 0x61 ; load R2 = 'a'

LOOP2                    ;; Looping through the letters from 'a' to 'z' small
    LDR R0, = String1
    LDR R1, = String2
    MOV R6, #0
LOOPSMALL_1              ;; looping through the first string and increase R6 is small or capital match is found
                         ;; (There is a char in string1 that is equal to char stored in R2).
    LDRB R4, [R0]

    CMP R4, #0
    BEQ exit1

    SUBS R5, R2, R4
    BEQ SAME1    ; SAME CHAR IN R2
    CMP R5, #32  ; SAME CHAR IN R2 But Capital
    BEQ SAME1

    B next1
SAME1
    ADD R6, #0x1
    B exit1
next1



    ADD R0, #0x1
    B LOOPSMALL_1
exit1
```

Figure 2.2: Count Common procedure part 1

As shown in the previous figure, we start the method by loading the value of ’a’ to R2 which will iterate over all the characters from ’a’ to ’z’.

’LOOP2’ is the outer loop that will iterate over all the characters, we also can see the first inner loop which will iterate over all the characters of the first string and add one as a flag to R6 and exit when a match is found.

```
LOOPSMALL_2      ;; looping through the second string and increase R6 is small or capital match is found
                 ;; (There is a char in string2 that is equal to char stored in R2).
    LDRB R4, [R1]

    CMP R4, #0
    BEQ exit2

    SUBS R5, R2, R4
    BEQ SAME2
    CMP R5, #32
    BEQ SAME2

    B next2
SAME2
    ADD R6, #0x1
    B exit2
next2


    ADD R1, #0x1
    B LOOPSMALL_2
exit2

```

Figure 2.3: Count Common procedure part 2

As shown in the previous figure, "LOOPSMALL 2" is very similar to the first inner loop we talked about in part 1, it has the same job but this time over the second string. if a match is found $R_6$ is increased again and the loop will terminate.

```
    CMP R6, #0x2
    BEQ COMMON_CHAR ;; if R6 is equal to 2 then the char in R2 appeared in string1 and in string2 so it's a common char.
    B SKIP
COMMON_CHAR
    ADD R3, #0x1
SKIP

    ADD R2, #0x1
    CMP R2, #0x7B
    BNE LOOP2


    BX LR
    ENDP
```

Figure 2.4: Count Common procedure part 3

In this part, the only work remaining is to check if $R_6$ is equal to 2 which means that the character we are considering now appeared in both strings so it's a common character. so we add 1 to the counter (R3) and continue to check the next character by increasing R2 by 1.

## 2.2 Procedure 3: Encrypt

In this procedure, we are given a string and we want to encrypt it by inverting the binary representation of each character in it.

```
Encrypt PROC
    LDR R3, = 0xFFFFFFFF
LOOP3
    LDRB R2, [R1]
    CMP R2, #0
    BEQ exit3

    EOR R2, R2, R3

    STRB R2,[R0]

    ADD R0, #0x1
    ADD R1, #0x1

    B LOOP3

exit3
    BX LR
    ENDP


    END
```

Figure 2.5: Encrypt procedure part 3

The work here is simple, we will iterate over the string and load the characters one by one, on R2, we will XOR them with R3 which all of its bits equal to 1, this action will invert all of the bits in R2 and then we store the new value of the character in the writing memory address.

# 3 Execution Example

First, we should declare our variables, we do so by adding them into two DATA areas, one for reading-only data, and the other one for reading-writing data.

```
    AREA Strings, DATA, READONLY ; Declaring the two sentences.
String1 DCB "HELLO WORLD",0
String2 DCB "Bye World",0
```

Figure 3.1: READONLY Data Area

```
    AREA Texts, DATA, READWRITE
TEX1 SPACE 20
TEX2 SPACE 20
Count1 SPACE 1
Count2 SPACE 1
COMMON SPACE 1
ENCRYPT1 SPACE 20
ENCRYPT2 SPACE 20
```

Figure 3.2: READWRITE Data Area

To do the first task, we should initialize a pointer to the reading address and to the writing address and a counter to count the number of characters converted to small.

```
    LDR R0, = TEX1 ; using the procedure TO_SMALL to convert the String1 to small letters and store it in TEX1
    LDR R1, = String1
    MOV R3, #0
    BL TO_SMALL
    LDR R0, = Count1
    STRB R3, [R0]
```

Figure 3.3: initializing to call the TO SMALL procedure with the first String

R0 is the writing address pointer, R1 is the reading address pointer, R3 is the counter and after calling the procedure we store R3 in the memory address equal to Count1.

```
    LDR R0, = TEX2 ; using the procedure TO_SMALL to convert the String2 to small letters and store it in TEX2
    LDR R1, = String2
    MOV R3, #0
    BL TO_SMALL
    LDR R0, = Count2
    STRB R3, [R0]
```

Figure 3.4: initializing to call the TO SMALL procedure with the second String

Same as before, but for the second string.

For the following input, the output will be like this.

```
    AREA Strings, DATA, READONLY ; Declaring the two sentences.
String1 DCB "HELLO WORLD",0
String2 DCB "Bye World",0
```

Figure 3.5: Example input TO SMALL

```
Memory 1
Address: 0x20000000
0x20000000: hello world.........bye world..............................................
0x20000050: ...........................................................................
0x200000A0: ...........................................................................
0x200000F0: ...........................................................................
```

Figure 3.6: Example Output TO SMALL

```
0x20000028:   0A   02
```

Figure 3.7: Example Output Counter1 and Counter 2, respectively

Moving on Counting the common characters.

```
;
;; Calling the procedure Count_Common to count the common letters between two strings.
;; Storing the address of the first String in R0
;; Storing the address of the second String in R1
;; Storing the count value in R3
;

LDR R0, = String1
LDR R1, = String2
MOV R3, #0
BL Count_Common
LDR R0, = COMMON
STRB R3,[R0]
```

Figure 3.8: initializing to call the Count Common procedure

R0 is a pointer to the first string, R1 is a pointer to the second string, and R3 is the common characters counter.

```
    AREA Strings, DATA, READONLY ; Declaring the two sentences.
String1 DCB "HELLO WORLD",0
String2 DCB "Bye World",0
```

Figure 3.9: Example input Counting Common

```
0x2000002A:   06
```

Figure 3.10: Example Output Procedure Counting Common

Moving to encryption.

**NOTE: in the code attached with this report the encryption is for the two strings as ordered but to have a meaningful run we will encrypt and decrypt the same string so we will have the original one back when the process is finished.**

```
    ;; Encrypt the first string (address in R1), Destination address in R0
    LDR R0, = ENCRYPT1
    LDR R1, = String1
    BL Encrypt

    ;; Encrypt the first string (address in R1), Destination address in R0
    LDR R0, = ENCRYPT2
    LDR R1, = ENCRYPT1
    BL Encrypt

STOP
    B STOP
```

Figure 3.11: initializing to call encrypt procedure

As we have mentioned, we will encrypt and decrypt the encrypted string so we will get the same string as the original one and run a meaningful example. The code attached to this report will have two calls for the encrypt procedure with the two strings as ordered in the project.

```
0x2000003F: HELLO WORLD
```

Figure 3.12: Example output Procedure encrypt

# 4 Conclusion

In this project, we have implemented an assembly program that contains 3 procedures that modify two strings. Each procedure has its own way of working. we put a lot of ARM assembly instructions that we have learned in the class into an application.

As all of the above code was written and applied in Keil uvision 5 software, we developed some knowledge in using it and working with ARM assembly with it.

Finally, I would mention that solving these problems in the assembly language requires much more thinking process than solving them in a high-level language like C or Java, this also can develop problem-solving skills and simple thinking in students.

# List of Figures

# List of Tables