Faculty of Engineering and Technology
Electrical and Computer Engineering Department

Computer Design Lab
ENCS4110

Experiment No. 4
ARM Addressing Modes

Prepared by:
Abdel Rahman Shahen 1211753

Partners:
no partners

Instructor: Dr. Abdel Salam Sayyad
Teaching assistant: Eng. Hanan Awawdeh

Section: 5
Date: 11/17/2023

Abstract

Employing a theoretical framework to investigate the Register Addressing Mode, Register Indirect Addressing Mode, ARM's Autoindexing Pre-Indexed Addressing Mode, ARM's Autoindexing Post-Indexing Addressing Mode, and the Program Counter Relative (PC Relative) Addressing Mode. The primary objectives of this study are to elucidate the operational principles and advantages of each addressing mode within the ARM architecture. Key findings reveal the versatility of these modes in facilitating efficient memory access, with a particular emphasis on the dynamic autoindexing capabilities. The significance of these results lies in their direct impact on optimizing code execution and enhancing overall program performance. In conclusion, this research provides a nuanced understanding of ARM addressing modes, empowering programmers with essential insights for effective utilization in ARM architecture development.

Table of Contents

Table of Figures

1.Theory

1.1 ARM addressing modes

There are different ways to specify the address of the operands for any given operations such as load, add or branch. The different ways of determining the address of the operands are called addressing modes. In this lab, we are going to explore different addressing modes of ARM processor and learn how all instructions can fit into a single word (32 bits).

1.1.1 Literal Addressing Mode

The immediate or literal addressing mode is where a literal number appears as a parameter to an instruction.

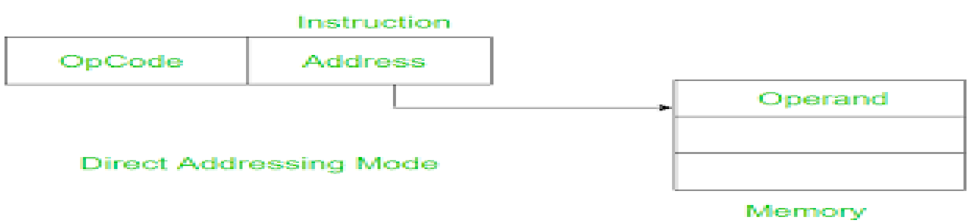

Fig. 1: Literal Addressing Mode observation

Example: MOV R0, #15

1.1.2 Register Indirect Addressing Mode

Register indirect addressing means that the location of an operand is held in a register. It is also called indexed addressing or base addressing.

Register indirect addressing mode requires three read operations to access an operand. It is very important because the content of the register containing the pointer to the operand can be modified at runtime. Therefore, the address is a variable that allows the access to the data structure like arrays.

- Read the instruction to find the pointer register
- Read the pointer register to find the operand address
- Read memory at the operand address to find the operand

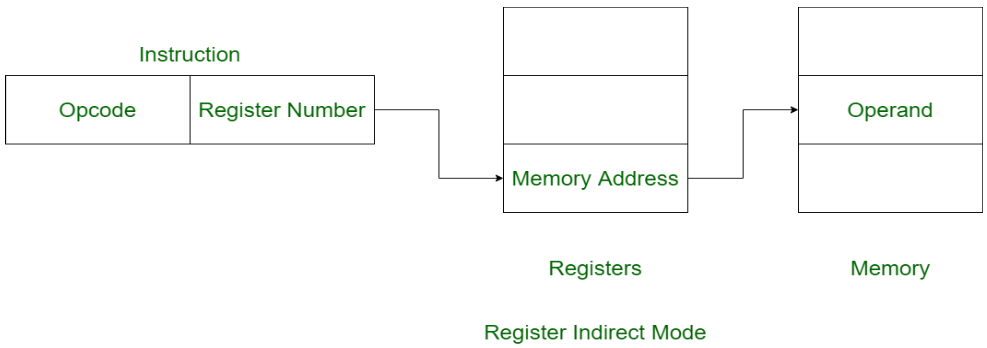

Fig. 2: Register Indirect Addressing Mode observation

Example: LDR R2, [R0]; Load R2 with the word pointed by R0

ARM supports a memory-addressing mode where the effective address of an operand is computed by adding the content of a register and a literal offset coded into load/store instruction.

Example:

LDR R0, [R1, #20]; R1 + 20; loads R0 with the word pointed at by R1+20

1.1.3 ARM's Autoindexing Pre-indexed Addressing Mode

This is used to facilitate the reading of sequential data in structures such as arrays, tables, and vectors. A pointer register is used to hold the base address. An offset can be added to achieve the effective address.

Example:
LDR R0, [R1, #4]! R1 + 4; loads R0 with the word pointed at by R1+4; then update the pointer by adding 4 to R1

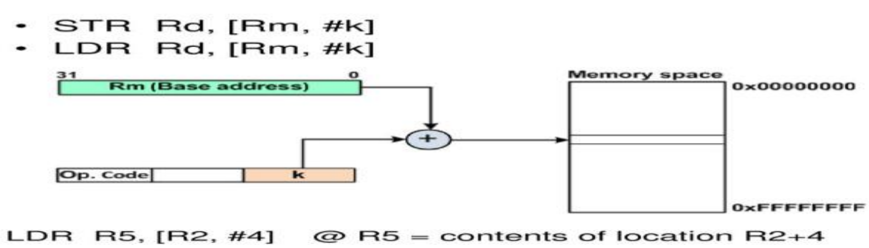

Fig. 3: Autoindexing Pre-indexed Addressing Mode observation

1.1.4 ARM's Autoindexing Post-indexing Addressing Mode

This is similar to the above, but it first accesses the operand at the location pointed by the base register, then increments the base register.
Example:
LDR R0, [R1], #4 R1; loads R0 with the word pointed at by R1; then update the pointer by adding 4 to R1

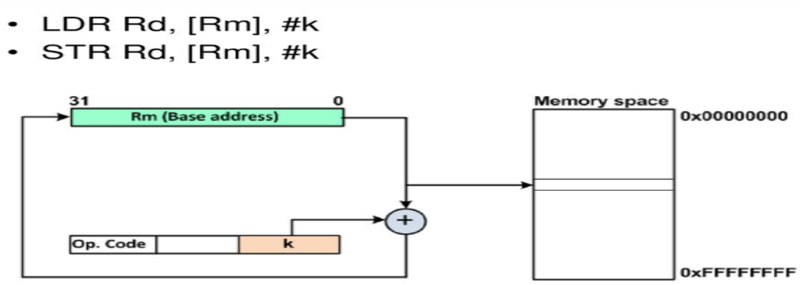

Fig. 4: Autoindexing Post-indexing Addressing Mode observation

1.1.5 Program Counter Relative (PC Relative) Addressing Mode

Register R15 is the program counter. If you use R15 as a pointer register to access operand, the resulting addressing mode is called PC relative addressing. The operand is specified with respect to the current code location.

Example:
LDR R0, [R15, #24] R15 + 24; loads R0 with the word pointed at by R15+24

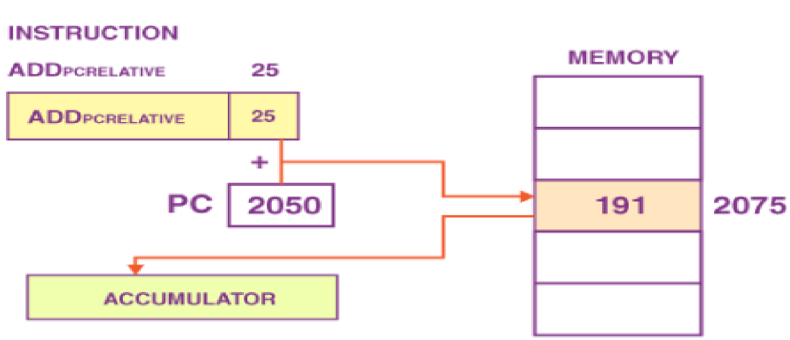

Fig. 5: ***(PC Relative) Addressing Mode***

1.2 ARM's Load and Store Encoding Format

The following picture illustrates the encoding format of the ARM's load and store instructions, which is included in the lab material for your reference. Memory access operations have a conditional execution field in bit 31, 03, 29, and 28. The load and store instructions can be conditionally executed depending on a condition specified in the instruction.

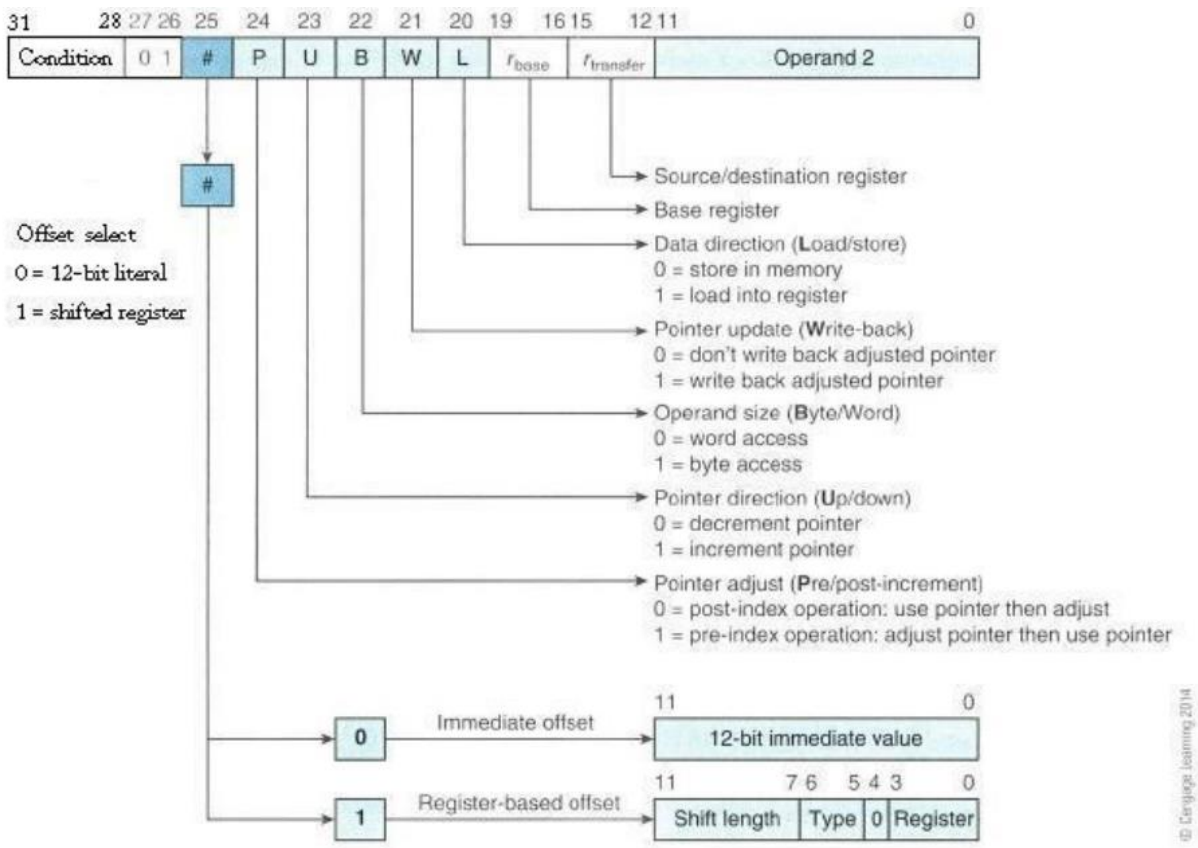

*Fig. 6: **ARM's load and store encoding format***

2. Procedure and Discussion:

2.1 An Example Program of Using Post-indexing Mode

2.1.1 Code:

Code Snippet 1: ARM Assemble Code for Using Post-indexing Mode

```
PRESERVE8
 THUMB
AREA RESET, DATA, READONLY
 EXPORT __Vectors
__Vectors
	DCD 0x20001000
	DCD Reset_Handler
	ALIGN
SUMP DCD SUM
N DCD 5
NUM1 DCD 3, -7, 2, -2, 10
POINTER DCD NUM1
	AREA MYRAM, DATA, READWRITE
SUM DCD 0
	AREA MYCODE, CODE, READONLY
	ENTRY
	EXPORT Reset_Handler
 Reset_Handler
	LDR R1, N
	LDR R2, POINTER
	MOV R0, #0
LOOP
	LDR R3, [R2], #4
	ADD R0, R0, R3
	SUBS R1, R1, #1
	BGT LOOP
	LDR R4, SUMP
	STR R0, [R4]
	LDR R6, [R4]
STOP
	B STOP
	END
```

2.1.2 Results:

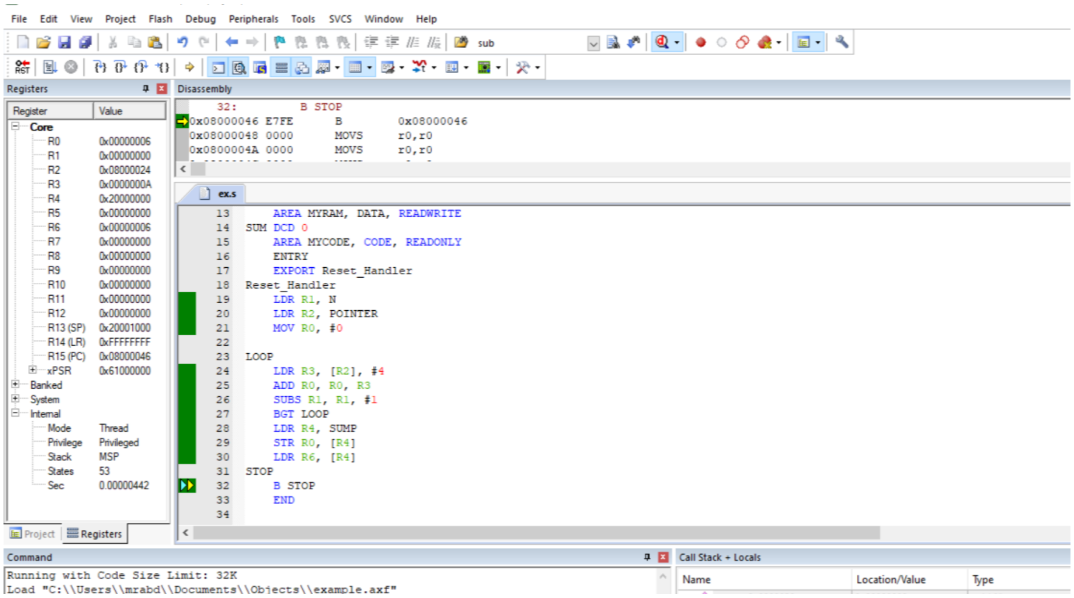

Fig. 7 example 1 Result

2.1.3 Result's Discussion:

The code calculates the sum of elements in the array NUM1 using a loop. The result is stored in memory, specifically at the location pointed to by SUMP.

2.2 An example to count the number of chars in a string

2.2.1 Code:

Code Snippet 2: ARM Assemble Code for counting the number of chars in a string

```
PRESERVE8
 THUMB
AREA RESET, DATA, READONLY
 EXPORT __Vectors
__Vectors
        DCD 0x20001000
        DCD Reset_Handler
        ALIGN
string1
        DCB "Hello world!",0
        AREA MYCODE, CODE, READONLY
        ENTRY
        EXPORT Reset_Handler
Reset_Handler
```

```
        LDR R0, = string1
        MOV R1, #0
loopCount
                LDRB R2, [R0], #1
                CBZ R2, countDone
                ADD R1, #1;
B loopCount
        countDone
                B countDone
                END
```

2.2.2 Results:

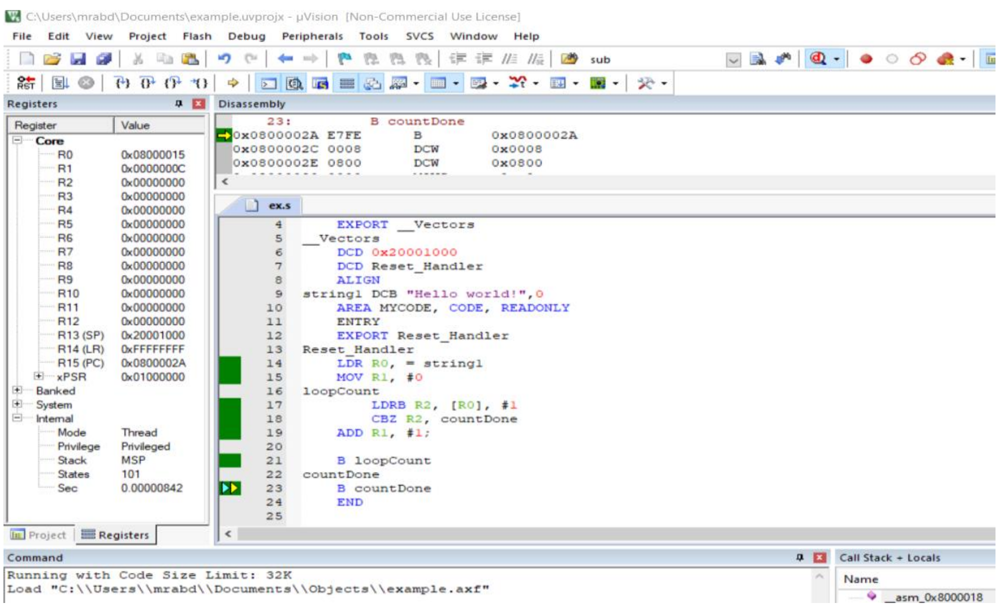

Fig. 8 example 2 Result

2.2.3 Result's Discussion:

The code initializes a string, "Hello world!", and counts the number of characters in the string. The result is stored in register R1.

2.3 Lab Work Exercises:

2.3.1 Exercise 1:

Write an ARM assembly language program AddGT.s to add up all the numbers that are great than 5 in the number array NUM1.

2.3.2 code:

Code Snippet 3 : ARM Assemble Code for to add up all the numbers that are great than 5 in the number array

```
 PRESERVE8
 THUMB
 AREA RESET, DATA, READONLY
 EXPORT __Vectors
__Vectors
 DCD 0x20001000 ; stack pointer value when stack is empty
 DCD Reset_Handler ; reset vector
 ALIGN
SUM DCD 0
SUMP DCD SUM
N DCD 7
NUM1 DCD 3, -7, 2, -2, 10, 20, 30
POINTER DCD NUM1
AREA MYCODE, CODE, READONLY
 ENTRY
 EXPORT Reset_Handler
sub_Add PROC
 ADD R4,R4,R3 ; R4+=R3
 BX LR ; branch to LR
Reset_Handler
 MOV R4,#0; R4 to save the sum


LDR R0, = NUM1 ; R0 holds the array
 MOV R1,#7 ; R1 hold the array's size
loop ;{ loop to iterate along the array
 LDR R3,[R0],#4 ; R3=R1[i]
 CMP R3,#5  ; R3>5?
 BLGT sub_Add ;if (R3 > 5 ) sub_ADD
```

```
SUBS R1,R1,#1 ; i-- , s: change zero flag
BNE loop ; (if R1==0) branch to loop
STOP
B STOP
END
```

2.3.3 Results:

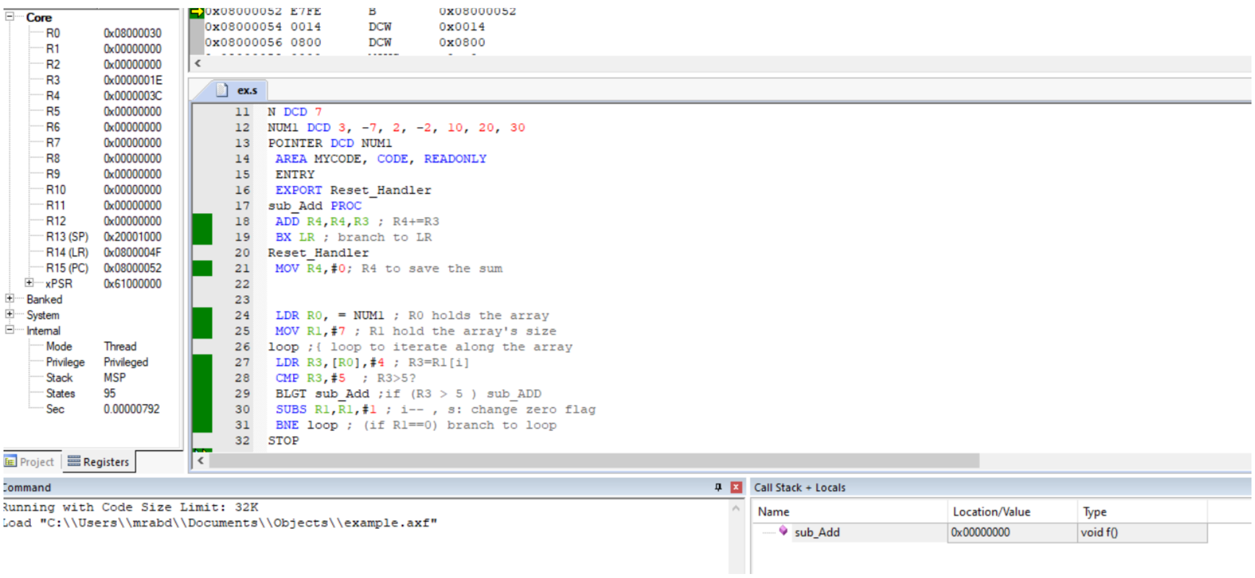

Fig. 9 lap work 1 Result

2.3.4 Result's Discussion:

The code initializes a sum variable, iterates through an array, and accumulates elements greater than 5 using a subroutine. The sum is stored in memory, providing a specialized arithmetic operation for qualifying array elements. Verification involves inspecting the memory location specified by SUMP or the final sum. The code structure indicates a controlled arithmetic operation tailored to specific conditions within the array.

As you can see the result in R4 = to 16*3 + 12 = 60

And if you add 10 + 20 + 30 which equals 60 you will see that the code works properly.

2.4.1 Exercise 2:

Write an ARM assembly language program Min-Max.s to find the maximum value and the minimum value in the number array NUM1.

2.4.2 code:

Code Snippet 4: ARM Assemble Code for to find min and max in an array

```
	PRESERVE8
 THUMB
 AREA RESET, DATA, READONLY
 EXPORT __Vectors
__Vectors
 DCD 0x20001000 ; stack pointer value when stack is empty
 DCD Reset_Handler ; reset vector
 ALIGN
SUM DCD 0
SUMP DCD SUM
N DCD 7
NUM1 DCD 3, -7, 2, -2, 10, 20, 30
POINTER DCD NUM1
 AREA MYCODE, CODE, READONLY
 ENTRY
 EXPORT Reset_Handler
Reset_Handler
 MOV R4,#0; R4 to save the min
 MOV R5,#0; R5 to save the max
 LDR R0, = NUM1 ; R0 holds the array
 MOV R1,#7 ; R1 hold the array's size
 LDR R3,[R0],#4 ; R3=arr[0]
 MOV R4,R3; R4 = arr[0]
 MOV R5,R3; R5 = arr[0]
 SUBS R1,R1,#1
loop ;{ loop to iterate along the array
 LDR R3,[R0],#4 ; R3=arr[i]
 CMP R3,R4;
 MOVLT R4,R3; if (arr[i] < R4 ) R4 = arr[i];
 CMP R3,R5;
 MOVGT R5,R3; if (arr[i] > R5) R5 = arr[i]
 SUBS R1,R1,#1 ; i-- , s: change zero flag
```

BNE loop ; (if R1==0) branch to loop
STOP
 B STOP
 END

2.4.3 Results:

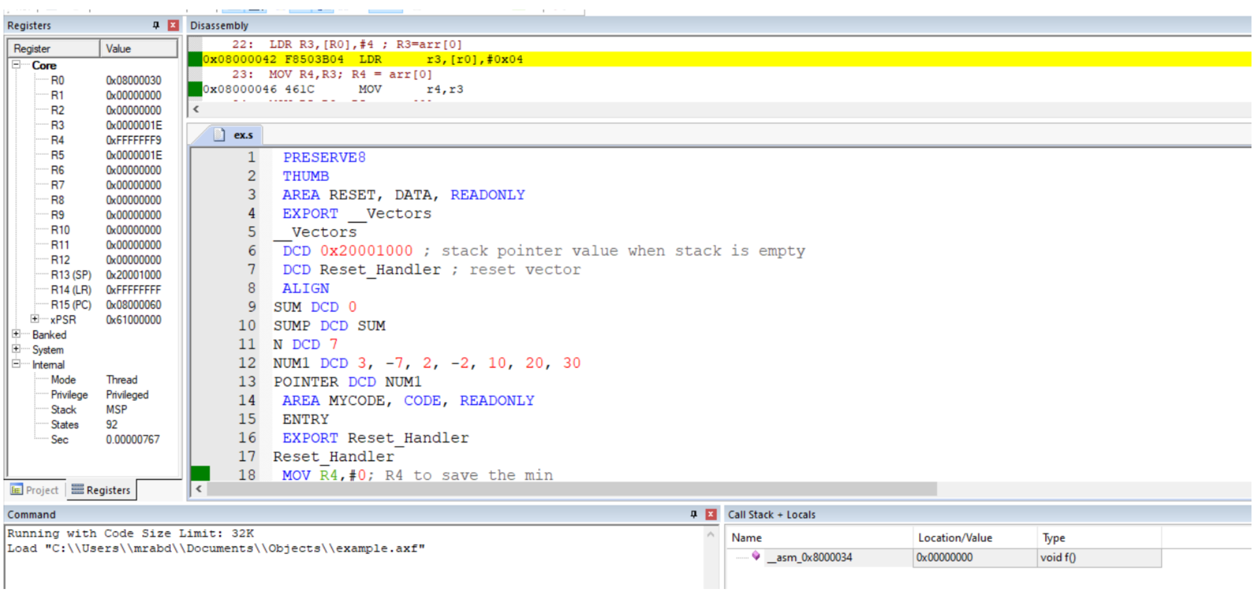

Fig. 10 lap work 2 Result

2.4.4 Result's Discussion:

The ARM assembly code finds the minimum and maximum values in the array NUM1. It initializes variables (R4 for min ,R5 for max) and iterates through the array, updating min and max values accordingly. The final min and max are stored in memory, providing an efficient algorithm for array extremum identification. Verification involves inspecting the memory location specified by Sump for the min and max values. The code structure showcases a systematic approach to identifying extremum values in an array.

As you can see the minimum value is -7 and it's saved in R4

And the maximum value is 30 and it's saved in R5

2.5.1 Exercise to Do:

Write an ARM assembly program to sort an array of integers. The sorted array should be stored in memory.

2.5.2 code :

```
PRESERVE8
 THUMB
 AREA RESET, DATA, READONLY
 EXPORT __Vectors
__Vectors
 DCD 0x20001000 ; stack pointer value when stack is empty
 DCD Reset_Handler ; reset vector
 ALIGN
size DCD 6
 AREA RESET,DATA,READWRITE
array DCD 3,5,4,12,7,9
 AREA MYCODE, CODE, READONLY
ENTRY
 EXPORT Reset_Handler
Reset_Handler
 LDR R0, =array
 LDR R1, =size
 LDR R1, [R1]
loop1
 MOV R2, #0
 MOV R3, R1
loop2
 LDR R4, [R0]
 LDR R5, [R0, #4]
 CMP R4, R5
 BLE noSwap
 ; Swap elements
 STR R5, [R0]
 STR R4, [R0, #4]
 MOV R2, #1
noSwap
 ADD R0, R0, #4
 SUBS R3, R3, #1
```

```
 BNE loop2
 SUBS R1, R1, #1
 TST R2, #1
 BNE loop1
STOP
 B STOP
 END
```

2.5.3 Results:

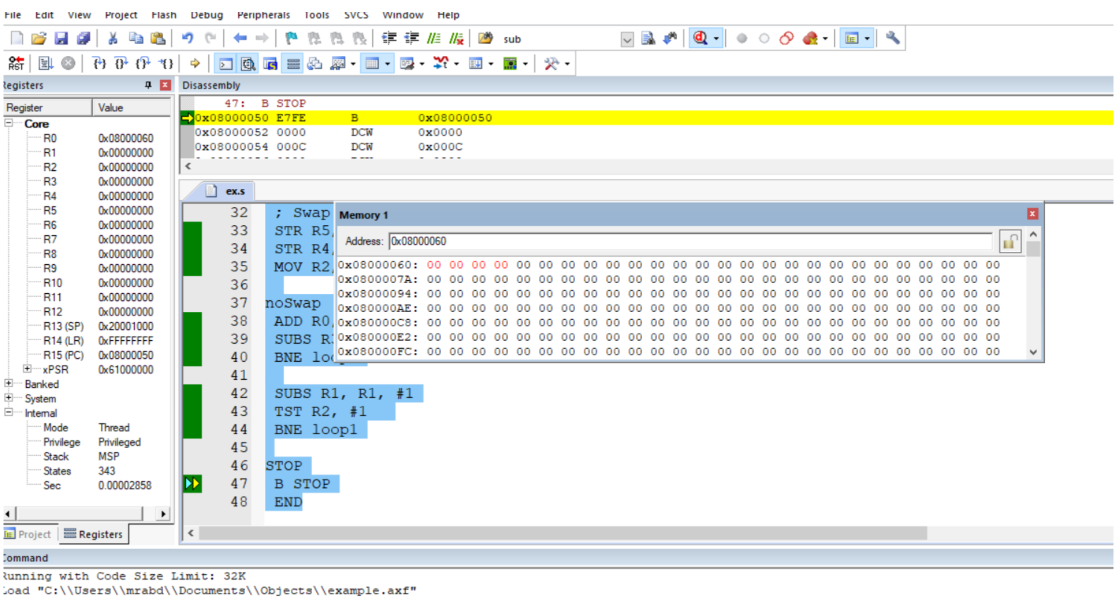

Fig. 11 to do result

2.5.4 Result's Discussion:

The provided ARM assembly code employs the Bubble Sort algorithm to arrange an array in ascending order. Utilizing nested loops, it compares adjacent elements and swaps them if necessary. The process iterates until the entire array is sorted, with the result visible in the memory.

But the problem that I not able to see a reason why the memory array elements are not changing

This problem caused to the array not the get sorted.

Conclusion:

In the exploration of addressing modes, we delved into the mechanisms by which processors access operands during instruction execution. Direct addressing involved specifying the memory address directly in the instruction, offering simplicity but limiting flexibility. Register addressing utilized processor registers for operand storage, enhancing speed. Indirect addressing allowed referencing memory locations indirectly through pointers, enabling dynamic data manipulation. Additionally, immediate and indexed addressing modes offered distinct advantages. This experiment illuminated the nuanced choices in addressing modes, showcasing their impact on program efficiency, adaptability, and the intricate balance between simplicity and complexity in designing effective instruction sets for diverse computing tasks.

Name	Alternative Name	ARM Examples
Register to register	Register direct	MOV R0, R1
Absolute	Direct	LDR R0, MEM
Literal	Immediate	MOV R0, #15 ADD R1, R2, #12
Indexed, base	Register indirect	LDR R0, [R1]
Pre-indexed, base with displacement	Register indirect with offset	LDR R0, [R1, #4]
Pre-indexed, autoindexing	Register indirect pre-incrementing	LDR R0, [R1, #4]!
Post-indexing, autoindexed	Register indirect post-increment	LDR R0, [R1], #4
Double Reg indirect	Register indirect Register indexed	LDR R0, [R1, R2]
Double Reg indirect with scaling	Register indirect indexed with scaling	LDR R0, [R1, R2, LSL #2]
Program counter relative		LDR R0, [PC, #offset]

Fig. 12 a summary for the addressing modes

References:

[1]: Manual for Computer Design Lab, 2023, Birzeit University.