
**Faculty of Engineering and Technology**
**Electrical and Computer Engineering Department**

# COMPUTER DESIGN LABORATORY
# ENCS 4110

## Experiment Nos. 4 and 5:

## ARM Addressing Modes and Subroutine / Procedure / Function Calls

Prepared by:
**Islam Zayed 1230007**

Instructor: Dr. Abualseoud Hanani
Teaching Assistant: Eng. Mohammad Abu Shelbaia

Section : 1
Date: August 2, 2025

## Abstract

Programming in ARM assembly language is effortlessly achieved by its flexible architecture including several different addressing mode techniques, subroutine calls and stack operations. Through practical experimentation and applications, we may explore the various addressing methods, particularly register direct, immediate, register indirect with offset, and PC-relative addressing. Subroutine execution employing the stack may also be practiced in programming ,providing reliability in managing control flow and temporary data storage in programs such as those involving string or array manipulation. The experiment aims to provide insight on the code efficiency and modularity obtained when appropriately implementing different addressing techniques. After obtaining results and key findings of the experiment, we determined that register indirect addressing with auto-indexing facilitates efficient data traversal, while subroutine calls promote code reusability. Reliability in the program may be ensured by utilizing proper stack operations essential for preserving register states during function calls. The principal conclusion emphasizes that expertise in ARM's addressing modes and procedure mechanisms is critical for producing efficient and maintainable low-level code.

## Table of Contents

## Table of Figures

## List of Tables

# 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." [1]

Table 1: ARM Addressing Modes

| Addressing Mode | ARM Example | Explanation |
|---|---|---|
| Register Direct | `MOV R0, R1` | Operand is in a register. |
| Immediate | `MOV R0, #15` | Operand is a constant value. |
| Direct | `LDR R0, MEM` | The operand is stored in memory. |
| Register Indirect | `LDR R0, [R1]` | The operand is a memory address stored in a register. |
| Register Indirect with Offset | `LDR R0, [R1, #4]` | The operand is a memory address in a register with an offset. |
| Register indirect pre-incrementing | `LDR R0, [R1, #4]!` | The operand is a memory address in a register, pre-incremented by an offset. |
| Register indirect post-increment | `LDR R0, [R1], #4` | The operand is a memory address in a register, post-incremented by an offset. |
| Register Indexed | `LDR R0, [R1, R2]` | The operand is a memory address, indexed by another register value. |
| Register indirect indexed with scaling | `LDR R0, [R1, R2, LSL #2]` | The operand is indexed by a register value, with scaling applied to the index. |
| PC Relative Addressing | `LDR R0, [PC, #offset]` | The operand is an address relative to the current program counter (PC), with an offset. |

### 1.1.1 Register Indirect Addressing Modes :

"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 access to the data structure like arrays.

1. Read the instruction to find the pointer register
2. Read the pointer register to find the operand address
3. Read memory at the operand address to find the operand " [1]

#### 1.1.1.1 Register Indirect Addressing with an Offset:

"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." [1]

ex.) `LDR R0, [R1, #20] ;  EA = R1 + 20`

#### 1.1.1.2 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. "[2]

ex.) `LDR R0, [R1, #4]!; EA = R1 + 20` , after **LDR**, `R1=R1 + 4`

#### 1.1.1.3 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."[1]

ex.) `LDR R0, [R1], #4;  EA = R1` , after **LDR**, `R1=R1 + 4`

#### 1.1.1.4 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. " [2]

ex.) `LDR R0, [R15, #24];  EA = R15 + 24`

### 1.2 ARM Subroutine/Procedure/Function Calls :

"ARM processors do not provide a fully automatic subroutine call/return mechanism like other processors. ARM's branch and link instruction,`BL`, automatically saves the return address in the register R14 (i.e, LR).We can use `MOV PC,LR` at the end of the subroutine to return back to the instruction after the subroutine called `BL SUBROUTINE_NAME`." [3]

### 1.3 Stack in ARM :

"The stack is a data structure, known as last in first out (LIFO).In a stack, items entered at one end and left in the reversed order.Stacks in microprocessors are implemented by using a stack pointer to point to the top of the stack in memory.As items are added to the stack,the stack pointer is moved up,and moved down as items are removed from the stack. "[1]

#### 1.3.1 Stack Types in ARM :

"ARM stacks are very flexible since the implementation is completely left to the software. A stack pointer is a register that points to the top of the stack. In the ARM processor, any one of the general purpose registers could be used as a stack pointer. Since it is left to the software to implement a stack, different implementation choices result in different types of stacks." Stacks may be classified based on the direction of growth and whether the stack pointer points to the last item or the next free location. [1]

Table 2: Types of Stacks in ARM Architecture

| Stack Type | Growth Direction | Stack Pointer Points To |
|---|---|---|
| Full Ascending Stack | Upward | Last filled location |
| Empty Ascending Stack | Upward | Next empty location |
| Full Descending Stack | Downward | Last filled location |
| Empty Descending Stack | Downward | Next empty location |

#### 1.3.2 Stack Operations in ARM :

1. PUSH: Saves registers on the stack (lowest-numbered to lowest address). Updates SP to point to the last stored value after execution.

2. POP: Restores registers from the stack. Updates the SP register to point to the location immediately above the highest loaded value after execution. [3]

# 2. Procedure and Discussion :

## 2.1.1 Experiment 4 - Example 1 : Applying Post - Indexing Mode :

### 2.1.1.1 Code :

**Code Snippet 2.1:** Example 1 - ARM Assembly Using Post-Indexing Addressing 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           ; load size of array  = 5
   LDR R2, POINTER      ; load base pointer of array  R2 = 0x00000010
   MOV R0, #0           ; initialize accumulator
LOOP
    LDR R3, [R2], #4    ; load value, then increments to next word
    ADD R0, R0, R3      ; add value from array to accumulator
    SUBS R1, R1, #1     ; decrement work counter
    BGT LOOP            ; keep looping until counter is zero
    LDR R4, SUMP        ; memory address to store sum R4=0x20000000
    STR R0, [R4]        ; store answer  = 6
    LDR R6, [R4]        ; Check the value in the SUM = 6
STOP B STOP
    END
```

### 2.1.1.2 Results :

The results of the program implemented in Code Snippet 2.1 are presented in Figure 2.1 below.

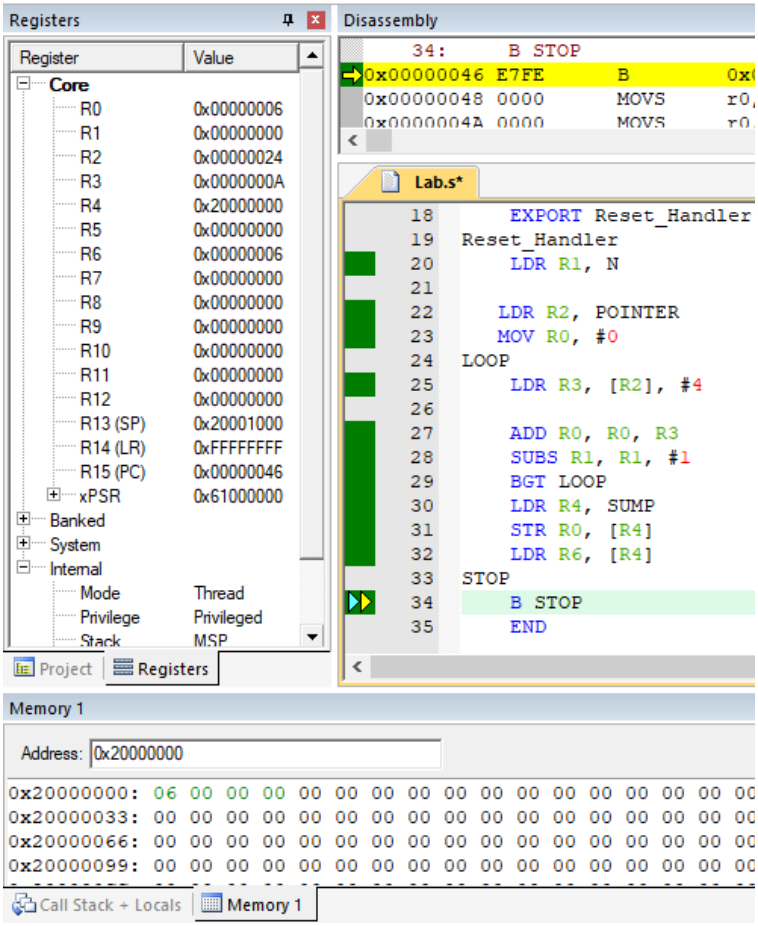


**Figure 2.1**: Register and Memory Contents After Execution of Code Snippet 2.1

### 2.1.1.3 Results Discussion :

The execution of the program begins with loading the amount of elements stored in the array NUM1 in our memory in order to indicate whether or not we had accessed all the elements in the array. A pointer to the array,called POINTER, is loaded into register 2 ,consisting of the address of our array. An accumulator is also initialized by setting the register 0 to zero. The first value of the array is then loaded onto register 3. It is important we use LDR, so as to load a whole word, and post increment to access the next 4 bytes (using post indexing) ,as the array is defined in memory as DCD, thus allocating a whole 4 Bytes for every number in the array. The newly loaded value is added to the

accumulator, and the loop condition remains true until we have accessed all the elements in the array, which is indicated by decrementing the counter until it is 0. The address of the SUM variable defined earlier in the memory is loaded onto register four by loading the value of a pointer to SUM allocated in memory earlier and calling it SUMP. The final value is stored onto this memory location, where we load the value from afterwards so that we may view it on register 6 and confirm our results, that the final value should be 6.

### 2.1.2 Experiment 4 - Example 2 : Applying Post-Indexing Mode :

#### 2.1.2.1 Code :

**Code Snippet 2.2:** Example 2 - ARM Assembly Using Post-Indexing Addressing Mode

```
        PRESERVE8
        THUMB
        AREA  RESET, DATA, READONLY
        EXPORT __Vectors
__Vectors
        DCD 0x20001000
        DCD Reset_Handler
        ALIGN
string1 DCB "Hello world!",0
    AREA    MYRAM, DATA, READWRITE
SUM  DCD 0
        AREA MYCODE, CODE, READONLY
        ENTRY
        EXPORT Reset_Handler
Reset_Handler
    LDR   R0, = string1 ;Load the address of string1 R0 = 0x8
    MOV R1, #0   ; Initialize the counter for length of string1
loopCount
    LDRB R2, [R0], #1  ; Load the char from address in R0
    CBZ R2, countDone   ; If it is zero... null terminated..
    ADD R1, #1          ; increment the counter for length
    B loopCount
countDone  B countDone
    END
```

### 2.1.2.2 Results :

The results of the program implemented in Code Snippet 2.2 are presented in Figure 2.2 below.

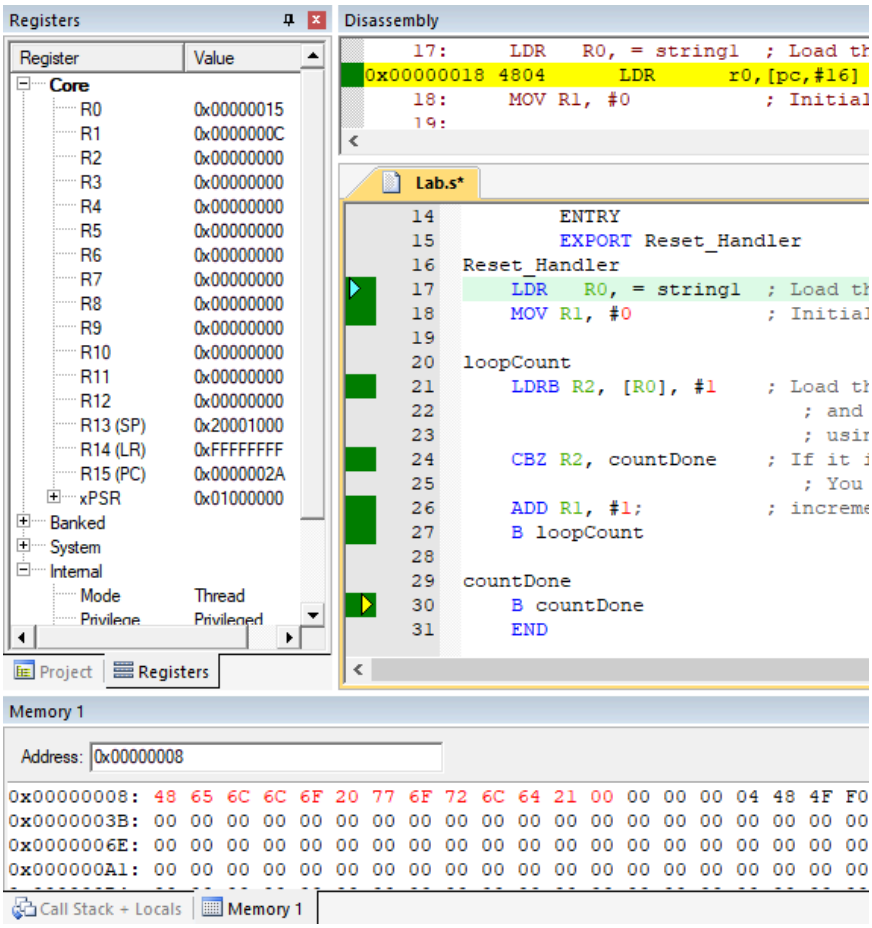


**Figure 2.2**: Register and Memory Contents After Execution of Code Snippet 2.2

### 2.1.2.3 Results Discussion :

The program initially begins by loading the address of our string into register 0. The register 1 is then initialized to 0 in order to keep count of the number of letters in our string. It is necessary that we use LDRB here to load the ASCII value of one character at a time because we had defined the string as DCB in the memory, thus allocating 1 Byte for every character in the string. The load instruction here uses post indexing afterwards, updating the address by one so that the next iteration will load the next byte. After loading the character onto the register, the instruction CBZ is used so as to branch out of the loop when the loaded ASCII value is 0, indicating we had reached the null terminator. The resulting sum of the number of characters in the string “Hello World!”

may be seen on register 1 to be 0xC, indicating 12 characters including the space and the exclamation mark. It may be also observed in the memory window that the ASCII values of the string are stored in the memory starting from address 0x8 up to address 0x15, where we see the string ends with the NULL character.

### 2.1.3 Experiment 5 - Example 1 : Using a Subroutine Call :

#### 2.1.3.1 Code :

**Code Snippet 2.3:** Example 1 - ARM Assembly Using a Subroutine Call

```
      PRESERVE8
      THUMB
    AREA  RESET, DATA, READONLY
    EXPORT __Vectors
__Vectors
    DCD 0x20001000
    DCD Reset_Handler
    ALIGN

SUMP  DCD SUM
SUMP2      DCD SUM2
N          DCD 5
    AREA MYDATA, DATA, READWRITE
SUM   DCD 0
SUM2  DCD 0
    AREA MYCODE, CODE, READONLY
    ENTRY
    EXPORT Reset_Handler
SUMUP PROC
      ADD  R0, R0, R1 ;Add number into R0
      SUBS  R1, R1, #1;Decrement loop counter R1
      BGT  SUMUP       ;Branch back if not done
      ;MOV  PC, LR
      BX  LR
      ENDP
Reset_Handler
      LDR  R1, N       ;Load count into R1  = 5
```

```
      MOV  R0, #0      ;Clear accumulator R0
      BL   SUMUP
      LDR  R3, SUMP    ;Load address of SUM to R3
      STR  R0, [R3]    ;Store SUM
      LDR  R4, [R3]
      MOV   R7, #8
      LDR R5, SUMP2    ; Load address of SUM2 to R5
      STR R7, [R5]     ; Store SUM2
      LDR R6, [R5]
STOP B STOP
    END
```

### 2.1.3.2 Results :

The results of the program implemented in Code Snippet 2.3 are presented in Figure 2.3 below.

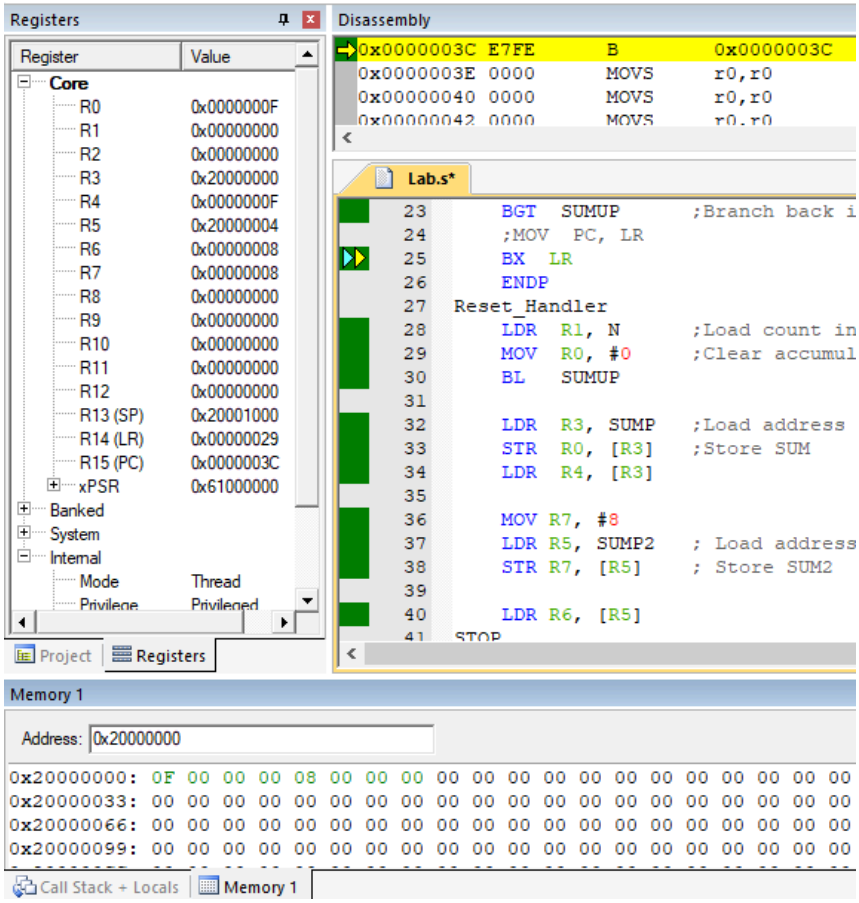


**Figure 2.3**: Register and Memory Contents After Execution of Code Snippet 2.3

#### 2.1.3.3 Results Discussion :

The ARM assembly program displayed above in code snippet 2.3 adds the first five positive integers using a subroutine SUMUP. The load count is loaded from the memory and onto a register, so that we may decrement this value after every loop iteration. The BL instruction indicates the program to branch to the subroutine defined above the reset_hander as SUMUP and save the return address. Subroutines require that they be within PROC and ENDP, as only the instructions between these directives will be executed as part of the subroutine. The subroutine iterates in a loop until the counter is 0, indicating the subroutine to branch out of the loop and perform BX LR, enabling our program to continue from where it had left off from. The memory address of the SUM variable is loaded onto a register , by loading the value of the pointer to SUM : SUMP, so that we may then store the final value onto this memory address. We may confirm our results as adding 1+2+3+4+5 should give 15, which we see is equivalent to 0xF in hexadecimal, and is indeed the value on our register and stored in SUM in the memory. The constant value 8 is also stored in memory as SUM2.

### 2.1.4 Experiment 5 - Example 2 : Using a Stack :

#### 2.1.4.1 Code :

**Code Snippet 2.4:** Example 2 - ARM Assembly Using Stack

```
      PRESERVE8
      THUMB
INITIAL_MSP EQU 0x20001000 ; Initial Main Stack Pointer Value

    AREA  RESET, DATA, READONLY
    EXPORT __Vectors
__Vectors
    DCD INITIAL_MSP
    DCD Reset_Handler
    ALIGN
SUMP  DCD SUM
SUMP2       DCD SUM2
N           DCD 5
```

```
    AREA MYDATA, DATA, READWRITE
SUM   DCD 0
SUM2  DCD 0

    AREA MYCODE, CODE, READONLY
    ENTRY
    EXPORT Reset_Handler

function1 PROC
           ;Using PROC and ENDP for procedures
      PUSH  {R5,LR}       ;Save values in the stack
      MOV R5,#8             ;Set initial value for the delay loop
delay
      SUBS R5, R5, #1
      BNE delay
      POP {R5,PC}           ;pop out the saved value from the stack,
      ;check the value in the R5 and if it is the saved value
      ENDP

Reset_Handler

      MOV  R0, #0x75
      MOV  R3, #5
      PUSH {R0, R3}         ;Notice the stack address is 0x200000FF8
      MOV  R0, #6
      MOV  R3, #7
      POP {R0, R3} ;Should be able to see R0 = #0x75 and R3 = #5
Loop
      ADD R0, R0, #1
      CMP R0, #0x80
      BNE Loop
      MOV  R5, #9 ;; prepare for function call
      BL  function1
      MOV  R3, #12

STOP
    B STOP
    END
```

### 2.1.4.2 Results :

The results of the program implemented in Code Snippet 2.4 are presented in Figure 2.4 below.

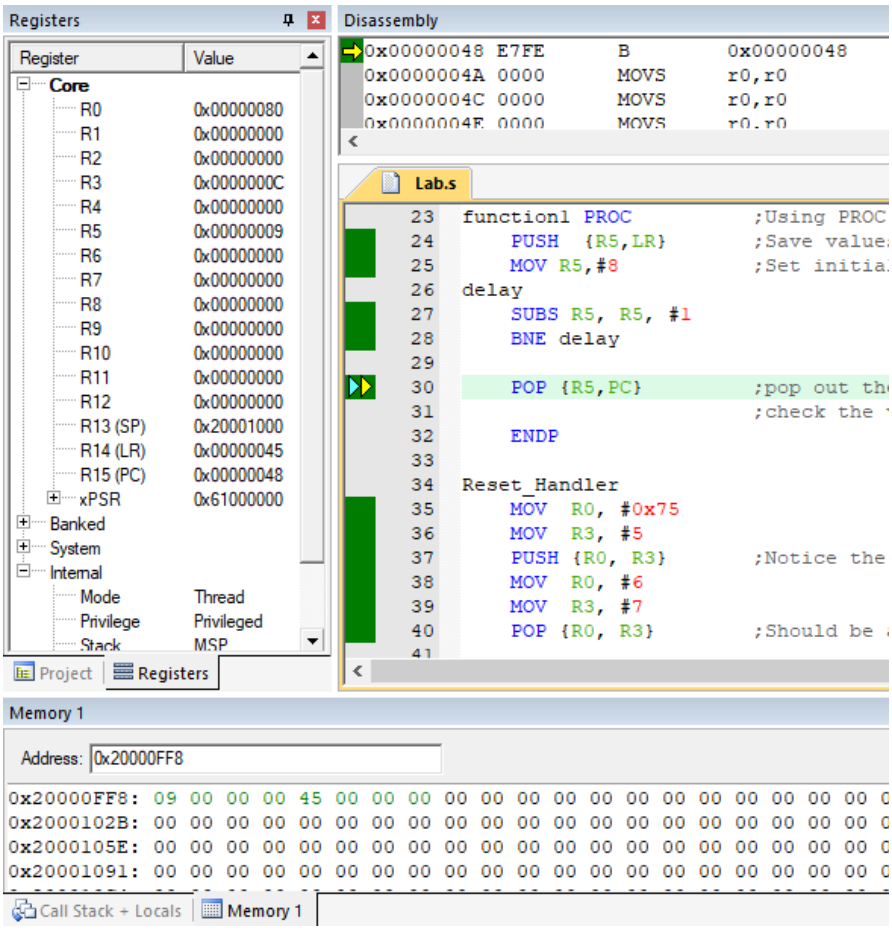


**Figure 2.4**: Register and Memory Contents After Execution of Code Snippet 2.4

### 2.1.4.3 Results Discussion :

The program begins by initializing R0 and R3 to 0x75 and 5 respectively. The values on these registers are then pushed onto the Stack, pushing the value of R5 and then the value of R3,as the PUSH instruction pushes the values of the registers beginning with the register with a greater index. The values of the registers are then changed so that when we pop the values back onto the registers it wont go unnoticed.The POP instruction pops the stack onto the registers,starting from the register with the lowest index. The program then stores the value 9 into R5 and branches to the subroutine , where the value in R5 is preserved in the stack during the delay loop which changes the value in R5. After

the loop condition fails,indicating the delay is over, the initial value of R5 is safely popped back onto R5, as well as the address of the LR safely popped into the PC. The address of the stack 0x200000FF8 ,where it may be observed in the memory window the values that had been pushed and popped to and from the stack.

## 2.2 Lab Work Exercises :

### 2.2.1.1 Experiment 4 - Lab Work Exercise 1 :

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

### 2.2.1.2 Code :

**Code Snippet 2.5:** Solution to Experiment 4, Exercise 1 in ARM Assembly

```
   PRESERVE8
    THUMB
    AREA  RESET, DATA, READONLY
    EXPORT __Vectors
__Vectors
    DCD 0x20001000
    DCD Reset_Handler
    ALIGN
N DCD 7
NUM1  DCD 3, -7, 2, -2, 10, 20, 30
POINTER DCD NUM1
SUMP    DCD SUM
    AREA MYDATA, DATA, READWRITE
SUM  DCD 0
    AREA MYCODE, CODE, READONLY
    ENTRY
    EXPORT Reset_Handler
Reset_Handler
      LDR R0,N
      LDR R1,POINTER
      LDR R5, SUMP
NEXT
      LDR R2,[R1],#4
```

```
      CBZ R0,DONE ;COMPARE AND BRANCH ON ZERO
      SUB R0,#1
      CMP R2,#5
      BLT LOOP
      ADD R3,R3,R2
LOOP
      B NEXT
DONE
      STR R3,[R5]
STOP
    B STOP
    END
```

### 2.2.1.3 Results :

The results of the program implemented in Code Snippet 2.5 are presented in Figure 2.5 below.

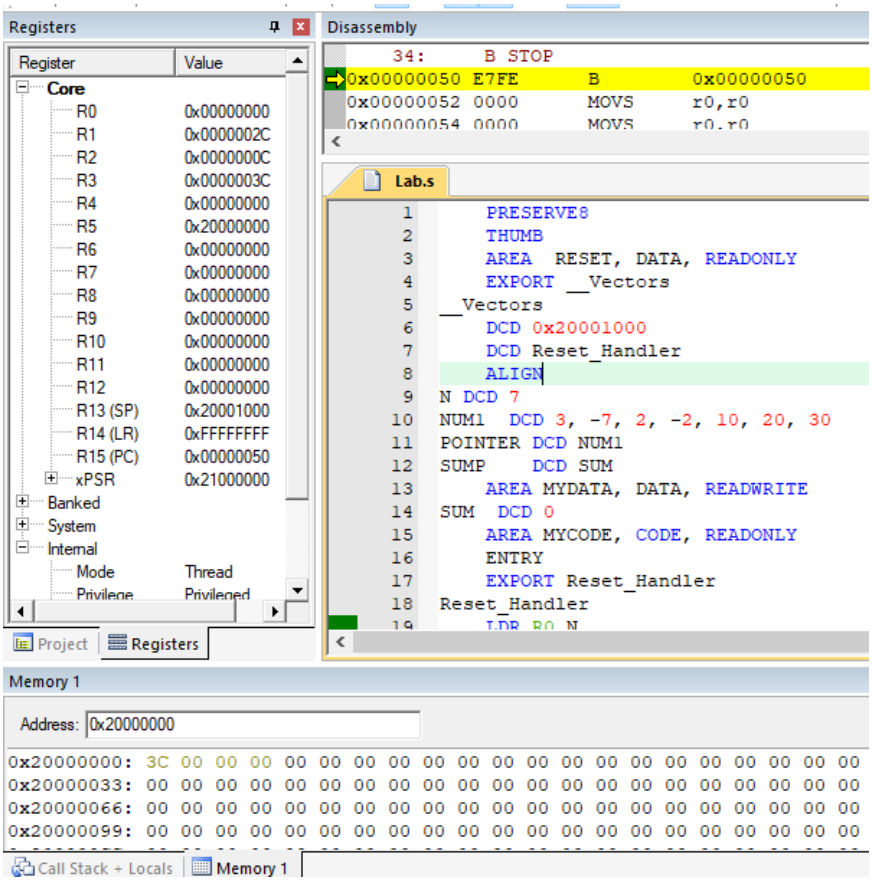


**Figure 2.5**: Register and Memory Contents After Execution of Code Snippet 2.5

#### 2.2.1.4 Results Discussion :

The code displayed above in code snippet 2.5 begins with loading onto registers the number of elements we have in the array, a pointer to the array, and a pointer to the final sum. The first number in the array is loaded onto the register, as here we loaded the word located in the memory address POINTER1 points to , as this pointer's value is the memory address of the array NUM1. After loading the element of the array onto the register POINTER1 is incremented by 4 using post indexing, so that in the next iteration the following element in the array may be loaded, considering that every element in the array is allocated a whole word in memory since we had defined them using the directive DCD. The process is repeated,each time decrementing the counter by 1, until the counter is found to be zero (which we checked using the CBZ instruction). The program then branches out of the loop, and stores the result in R3 into the memory address allocated for the SUM final value, which we had obtained to be 0x3C = 60, ensuring that we had indeed obtained desired results.

#### 2.2.2.1 Experiment 5 - Lab Work Exercise 2 :

Write an ARM assembly language program that will have a user defined function/procedure factorial to calculate the factorial for a given number.

#### 2.2.2.2 Code :

**Code Snippet 2.6:** Solution to Experiment 5, Exercise 2 in ARM Assembly

```
        PRESERVE8
        THUMB
        AREA RESET, DATA, READONLY
        EXPORT __Vectors
__Vectors
        DCD     0x20001000          ; Initial Stack Pointer
        DCD     Reset_Handler       ; Reset Handler
        ALIGN
N           DCD     5               ; Number to compute factorial

        AREA MYDATA, DATA, READWRITE
```

```
RESULT      DCD     0               ; Memory location to store result

        AREA MYCODE, CODE, READONLY
        ENTRY
        EXPORT Reset_Handler
        EXPORT Factorial


Factorial PROC
        PUSH    {LR}            ; Save return address
        MOV     R0, #1
Loop
        CMP     R1, #1          ; If R1 <= 1, exit loop
        BLT     Done
        MUL     R0, R0, R1      ; R0 = R0 * R1
        SUB     R1, R1, #1
        B       Loop
Done
        POP     {PC}            ; Return
        ENDP


Reset_Handler
        LDR     R1,=N
        LDR     R1, [R1]        ; Load value into R1
        BL      Factorial       ; Call Factorial
        LDR     R3, =RESULT
        STR     R0, [R3]        ; Store result
STOP
        B       STOP            ; Infinite loop
        END
```

### 2.2.2.3 Results :

The results of the program implemented in Code Snippet 2.6 are presented in Figure 2.6 below.

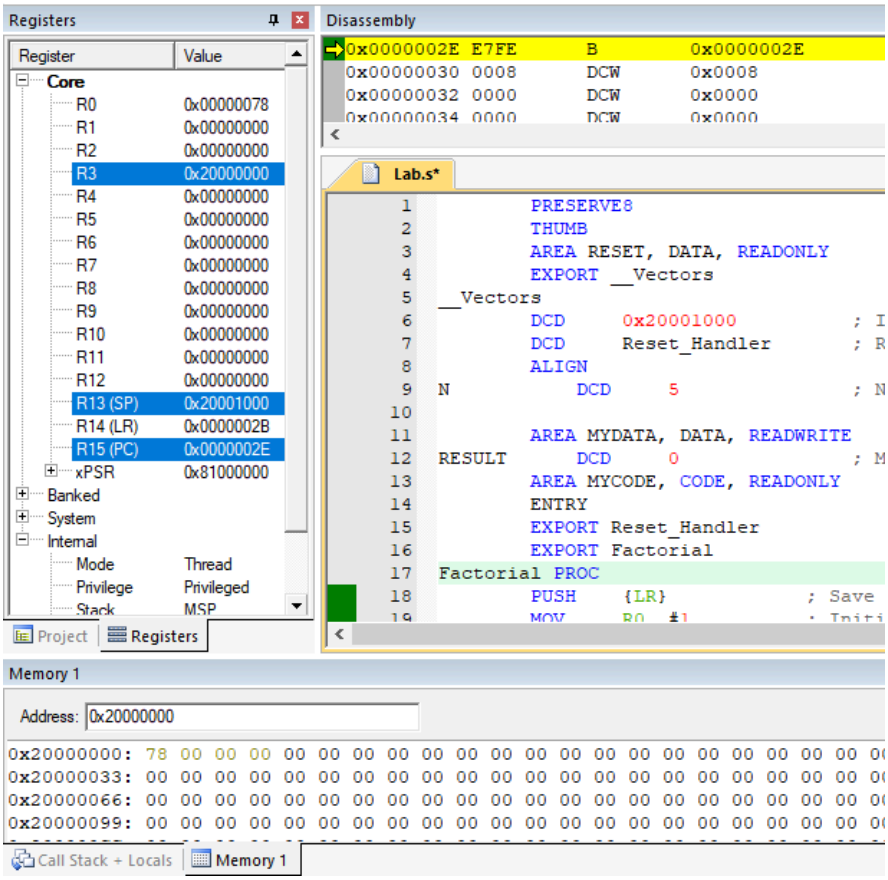


**Figure 2.6**: Register and Memory Contents After Execution of Code Snippet 2.6

### 2.2.1.4 Results Discussion :

The value we need to calculate the factorial is stored in N, so we load its address to then load the value. The program then branches to the factorial function which saves the return address from LR in the stack before it computes the factorial in the loop.The loop keeps iterating, multiplying the current value to the number on the register before post decrementing, as long as the value is not less than 1. After the factorial is computed, the program branches out of the loop and pops the return address onto the PC so that the program may continue from where it had left off from before the subroutine. We then obtained the address of the result variable so that we may store the result 120 = 0x78.

## 2.3 Lab Task (To-Do) :

### 2.3.1 To-Do :

Find the mod - 8 for each integer in arr and store them in reverse in rev- arr. You should have two procedures.

1. Mod - 8
2. Reverse

   The reset handler should do the following as well

   R0 → address (arr)

   R1 →N

   R2 → address(rev - arr)

   BL reverse

### 2.3.2 Code :

**Code Snippet 2.7:** ARM Assembly Language Code implementing the ToDo Task

```
        PRESERVE8
        THUMB
         AREA  RESET, DATA, READONLY
         EXPORT __Vectors

__Vectors
        DCD 0x20001000          ; Initial stack pointer
        DCD Reset_Handler       ; Reset handler address
        ALIGN

arr     DCD 11, 15, 17, 19, 20 ; Input array
N       DCB 5                   ; Number of elements

        AREA MYDATA, DATA, READWRITE
rev_arr SPACE 5                 ; Output array (byte-sized)

        AREA MYCODE, CODE, READONLY
        ENTRY
        EXPORT Reset_Handler
```

```
; Reverse array with MOD_8 operation on each element
REVERSE PROC
        PUSH {LR}               ; Save return address

ARR_LOOP
        CBZ R1, STORE_LOOP      ; If R1 == 0, done reading
        LDR R5, [R0], #4        ; Load next element, post-increment
        BL MOD_8                ; Modify value (MOD_8 or MOD_4)
        PUSH {R5}               ; Push modified value to stack
        SUB R1, R1, #1          ; Decrement loop counter
        B ARR_LOOP

STORE_LOOP
        CBZ R2, EXIT            ; If R2 == 0, done storing
        POP {R5}                ; Pop from stack (reversed order)
        STRB R5, [R3], #1       ; Store byte to rev_arr
        SUB R2, R2, #1          ; Decrement store counter
        B STORE_LOOP

EXIT
        POP {PC}                ; Return from REVERSE
        ENDP

; Apply MOD 8 (AND with 7)
MOD_8 PROC
        PUSH {LR}
        AND R5, R5, #7          ; R5 = R5 % 8
        POP {PC}
        ENDP

; Program entry point
Reset_Handler
        LDR R0, =arr            ; R0 = address of arr
        LDR R1, =N
        LDRB R1, [R1]           ; R1 = N (number of elements)
        MOV R2, R1              ; R2 = copy of N for store loop
        LDR R3, =rev_arr        ; R3 = destination buffer
        BL REVERSE              ; Call reverse function
```

```
STOP    B STOP                  ; Infinite loop after finish
        END             ; Infinite loop to end program
```

### 2.3.3 Results :

The results of the program implemented in Code Snippet 2.7 are presented in Figure 2.7 below.

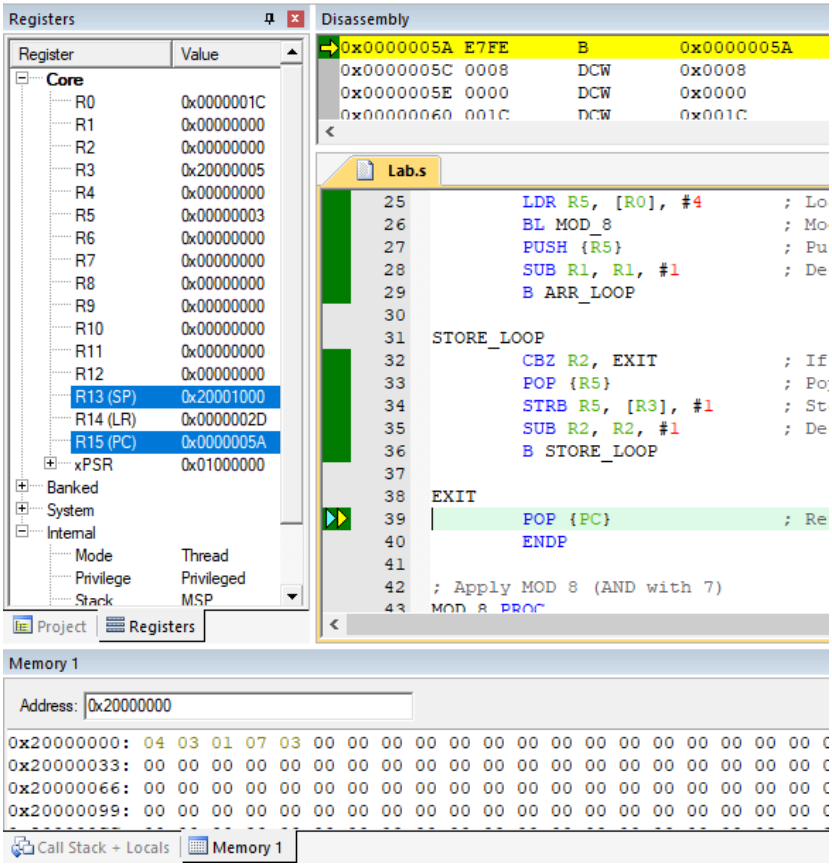


**Figure 2.7**: Register and Memory Contents After Execution of Code Snippet 2.7

### 2.3.4 Results Discussion :

The ARM assembly program begins with loading the address of the array into the register R0. The value of the element count is obtained by first loading the address of the variable N,and then loading the value , 5, located at that address. The register 2 is then initialized to this value as well , as knowing the amount of elements is of equal importance in the reverse array. The address of the reverse array is loaded into the

register 3. The program then calls the reverse function by the BL instruction, which branches to our function while saving the return address into the LR for later. The program continues execution in the user defined REVERSE function, initially saving the value of the LR before entering a loop where every element is loaded into a register and branches off to another function, MOD - 8 before pushing the mod - 8 result into the Stack. Loading every element requires post indexing by 4, so that the next iteration will have access to the next word. After the mod - 8 of every element in the loop had been computed and pushed into the stack, we pop the values in reversed order and store them onto the memory address of the reverse array. The store function requires that only a byte is stored,as the reverse array is defined using SPACE 5 in memory, and therefore allocating 5 bytes in the memory for the reverse array. We see that we had indeed obtained the desired results as the mod - 8 of our array in order would be 3,7,1,3,4,which may be observed in Fig. 2.7 to be stored in reverse in the memory.

## Conclusion :

After completing the experiment regarding ARM's various addressing modes, subroutine implementation, and stack operations, I am confident in implementing such architecture to produce more solid, reliable low level code. Data handling efficiency was achieved when applying various addressing techniques, such as register indirect with offsets and auto-indexing, appropriately in the program. I practiced implementing different tasks and exercises including summing array values, calculating factorials, and manipulating strings using subroutines and a stack for managing return addresses and register values. The subroutines provided had demonstrated function calls with the BL instruction, along with the function returns facilitated by the LR and PC registers. Further supporting modular design, I investigated how stack operations in ARM's software based stack management may enable me to safely preserve register values and return addresses during subroutine calls. The experiment had enriched my comprehension in regards to how high-level programming logic may efficiently be implemented with low - level instructions using registers, procedural calls, and a stack proficiently. All in all, I developed greater intuition for ARM architecture and its flexibility and support for modular programming, particularly in subjects where performance, memory optimization, and power efficiency are crucial for writing reliable, low-level code in real-world embedded systems.

## References :

[1]:Manual for Computer Design Lab, 2025, Birzeit University.

[2]:https://developer.arm.com/documentation/den0013/d/ARM-Thumb-Unified-Assembly-Language-Instructions, [Accessed on July 30 , 2025]

[3]: https://mabushelbaia.com/encs4110/, [Accessed on July 31 ,2025]

## Appendices: