Faculty of Engineering & Technology
Electrical & Computer Engineering Department

COMPUTER DESIGN LABORATORY

ENCS4110

Report #1

Experiment #9

ARM Addressing Modes

Prepared by:

Islam Jihad - 1191375

Instructor: Dr. Abualseoud Hanani

Assistant: Eng. Raha Zabadi

Section: 1

Date: 10/Apr./2022

Abstract:

In this experiment we need to know the most important things about addressing modes and how to iterate them and pass in arrays, and to remember the functions of previous labs and reuse them.

Table of Contents

Table of figures

Theory

Review of ARM Registers Set

ARM contains 16 programmer-accessible registers and a Current Program Status Register, or CPSR, as discussed in the preceding lab. Here's a diagram of the ARM register set.

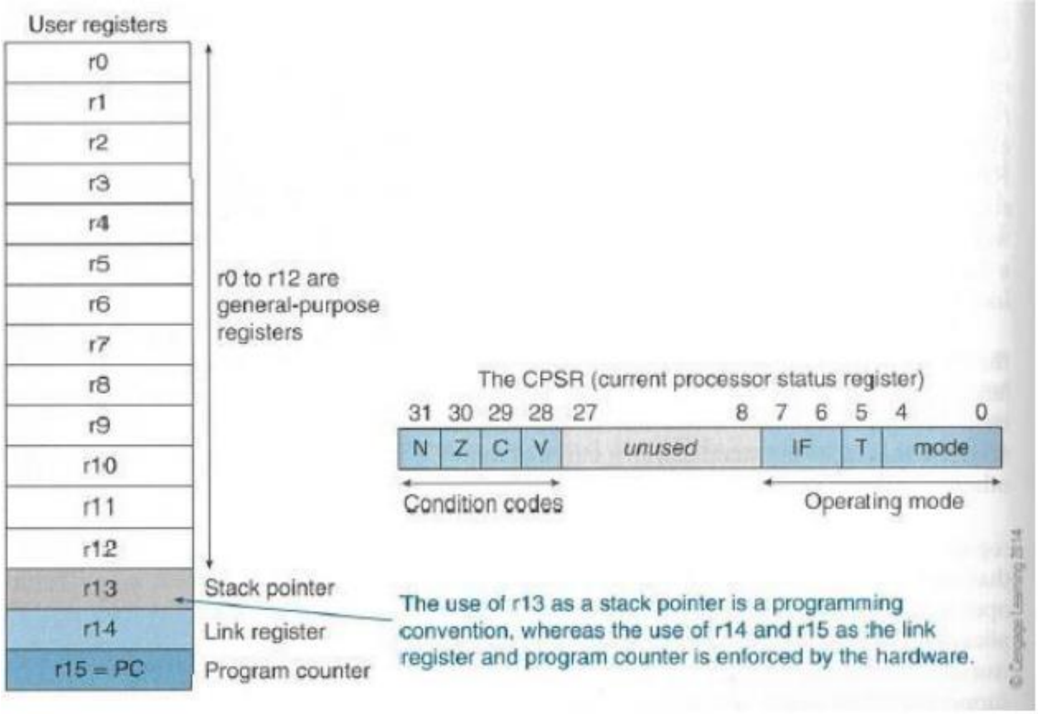


Figure 1 Review of ARM Registers Set

R0 to R12 are the general-purpose registers.

R13 is reserved for the programmer to use it as the stack pointer.

R14 is the link register which stores a subroutine return address.

R15 contains the program counter and is accessible by the programmer.

Conditonion code flags in CPSR:

N - Negative or less than flag

Z - Zero flag C - Carry or bowrrow or extendedflag

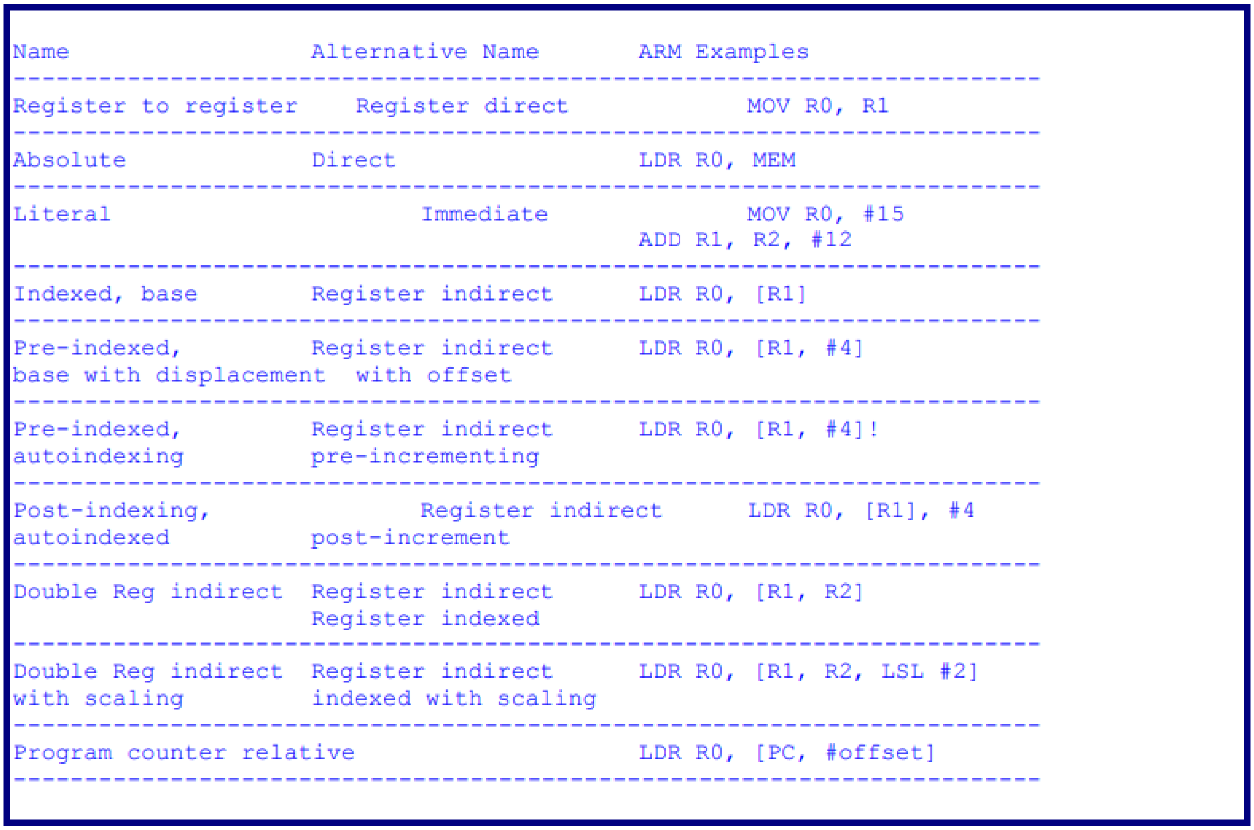
V - Overflow flag

The least-significant 8-bit of the CPSR are the control bits of the system.

The other bits are reserved.

Summary of ARM addressing Modes

For each given operation, such as load, add, or branch, there are several ways to define the address of the operands. The various methods for identifying the operands' addresses are referred to as addressing modes. In this lab, we'll look at the various addressing modes used by the ARM processor and see how all instructions may be condensed into a single word (32 bits).



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]

Figure 2 Summary of ARM addressing Modes examples

Literal Addressing Mode

An addressing mode specifies how to calculate the effective memory address of an operand by using information held in registers and/or constants contained within a machine instruction or elsewhere.

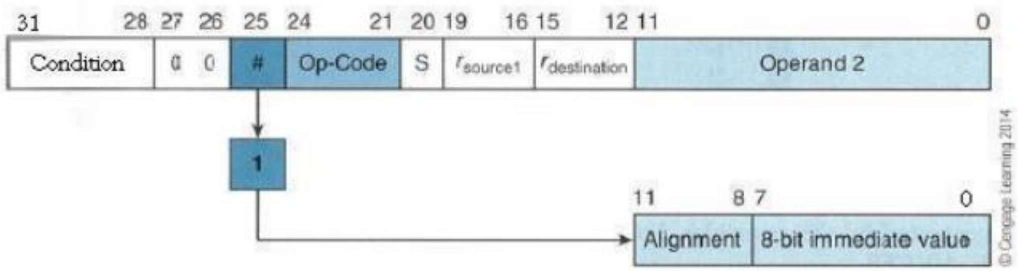
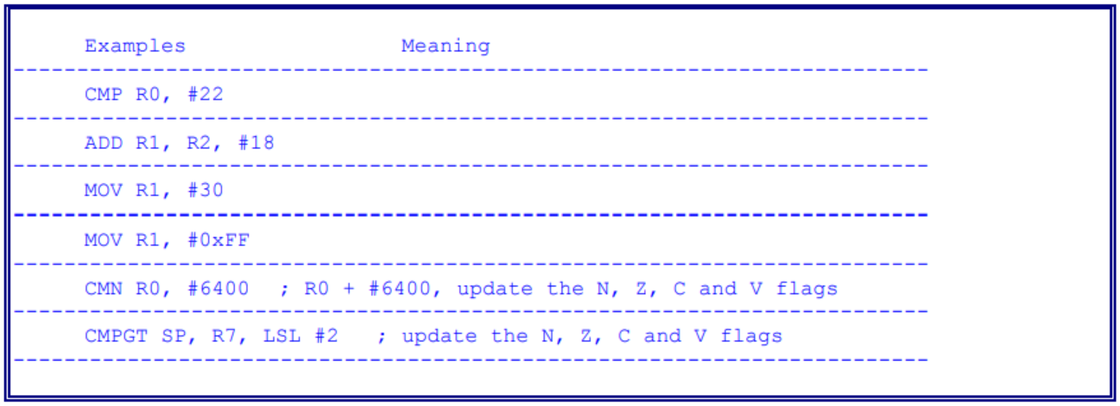


Figure 3 Literal Addressing Mode

some examples on Literal Addressing Mode:



```
     Examples                 Meaning
-------------------------------------------------------------------------
     CMP R0, #22
-------------------------------------------------------------------------
     ADD R1, R2, #18
-------------------------------------------------------------------------
     MOV R1, #30
-------------------------------------------------------------------------
     MOV R1, #0xFF
-------------------------------------------------------------------------
     CMN R0, #6400  ; R0 + #6400, update the N, Z, C and V flags
-------------------------------------------------------------------------
     CMPGT SP, R7, LSL #2   ; update the N, Z, C and V flags
-------------------------------------------------------------------------
```

Figure 4 Literal Addressing Mode examples

Register Indirect Addressing Mode

In register indirect addressing mode, the address of operand is placed in any one of the registers. The instruction specifies a register that contains the address of the operand.

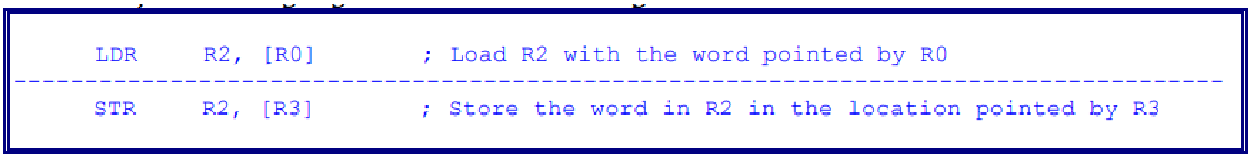
The position of an operand is maintained in a register in register indirect addressing. It's also known as base addressing or indexed addressing.

To access an operand in register indirect addressing mode, three read operations are required. It's critical because the content of the register carrying the operand's reference might be changed at runtime. As a result, the address is a variable that permits access to data structures such as arrays.

To locate the pointer register, read the instruction.

To find the operand address, read the pointer register. To determine the operand address, read memory at the operand address.

Some examples of using register indirect addressing mode:



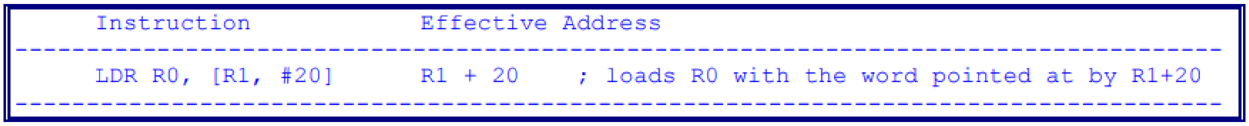
```
    LDR     R2, [R0]        ; Load R2 with the word pointed by R0
------------------------------------------------------------------------------------
    STR     R2, [R3]        ; Store the word in R2 in the location pointed by R3
```

Figure 5 Register Indirect Addressing Mode examples

Register Indirect Addressing with an Offset

Register indirect is the simplest addressing mode. The address is provided entirely by the base register. The offset is added to or subtracted from base register, and the result is the address to be accessed

The effective address of an operand is derived by adding the contents of a register and a literal offset programmed into a load/store instruction in ARM's memory-addressing mode. As an example,

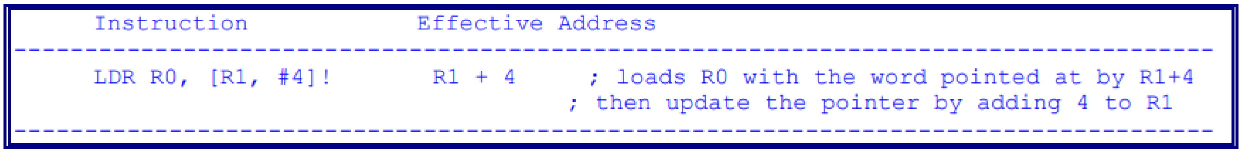


```
    Instruction             Effective Address
------------------------------------------------------------------------------------
    LDR R0, [R1, #20]       R1 + 20     ; loads R0 with the word pointed at by R1+20
------------------------------------------------------------------------------------
```

Figure 6 Register Indirect Addressing with an Offset examples

ARM's Autoindexing Pre-Indexed Addressing Mode

This is used to make sequential data in structures like arrays, tables, and vectors easier to read. The base address is stored in a pointer register. To get the effective address, an offset might be applied. As an example,

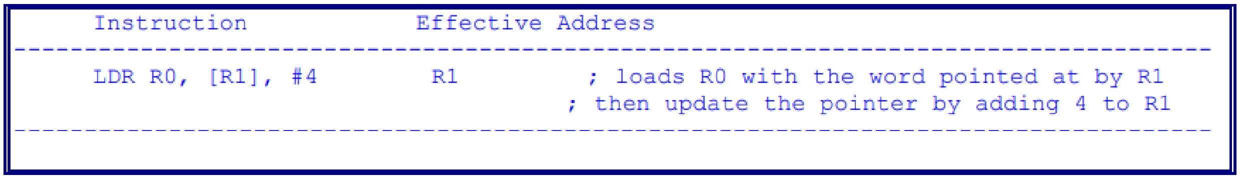


```
      Instruction               Effective Address
-----------------------------------------------------------------------------------
      LDR R0, [R1, #4]!          R1 + 4      ; loads R0 with the word pointed at by R1+4
                                            ; then update the pointer by adding 4 to R1
-----------------------------------------------------------------------------------
```

Figure 7 ARM's Autoindexing Pre-Indexed Addressing Mode examples

ARM's Autoindexing Post-indexing Addressing Mode

This is identical to the previous example, except that it first reads the operand at the address indicated by the base register before incrementing the base register. As an example,

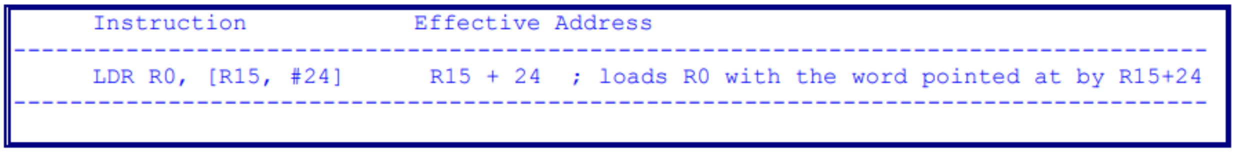


```
      Instruction               Effective Address
-----------------------------------------------------------------------------------
      LDR R0, [R1], #4           R1          ; loads R0 with the word pointed at by R1
                                            ; then update the pointer by adding 4 to R1
-----------------------------------------------------------------------------------
```

Figure 8 ARM's Autoindexing Post-indexing Addressing Mode examples

Program Counter Relative (PC Relative) Addressing Mode

The program counter is located in register R15. PC relative addressing is the addressing method that results from using R15 as a pointer register to access operand. With regard to the present code position, the operand is provided. Take a look at this illustration.

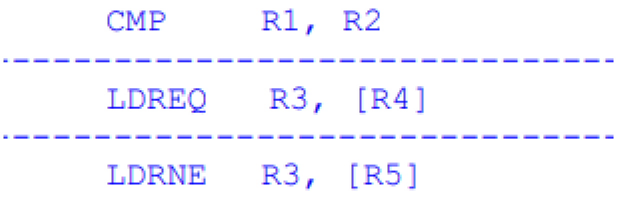


```
      Instruction               Effective Address
-----------------------------------------------------------------------------------
      LDR R0, [R15, #24]         R15 + 24   ; loads R0 with the word pointed at by R15+24
-----------------------------------------------------------------------------------
```

Figure 9 Program Counter Relative (PC Relative) Addressing Mode examples

ARM's Load and Store Encoding Format

The encoding format of the ARM's load and store instructions is shown in the accompanying diagram, which is included in the lab material for your reference. In bits 31, 03, 29, and 28, there is a conditional execution field for memory access operations. Conditionally execute load and store instructions based on a condition stated in the instruction. Take a look at the samples below:



```
      CMP       R1, R2
--------------------------------
      LDREQ     R3, [R4]
--------------------------------
      LDRNE     R3, [R5]
```

Figure 10 ARM's Load and Store Encoding Format examples

Encoding Format of ARM's load and store instructions

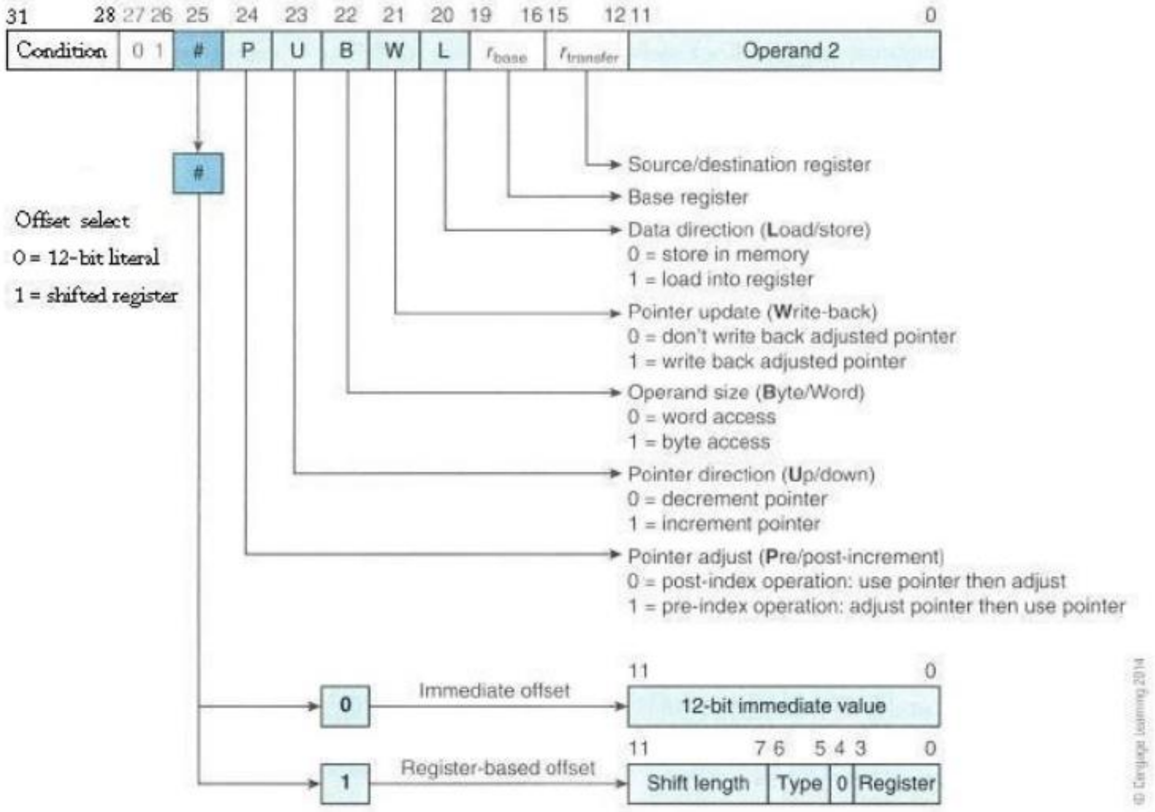
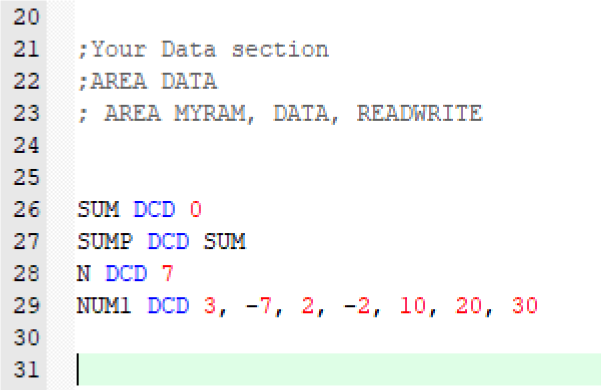


Figure 11 Encoding Format of ARM's load and store instructions

Procedure and Discussion

Program #1 add numbers greater than integer:

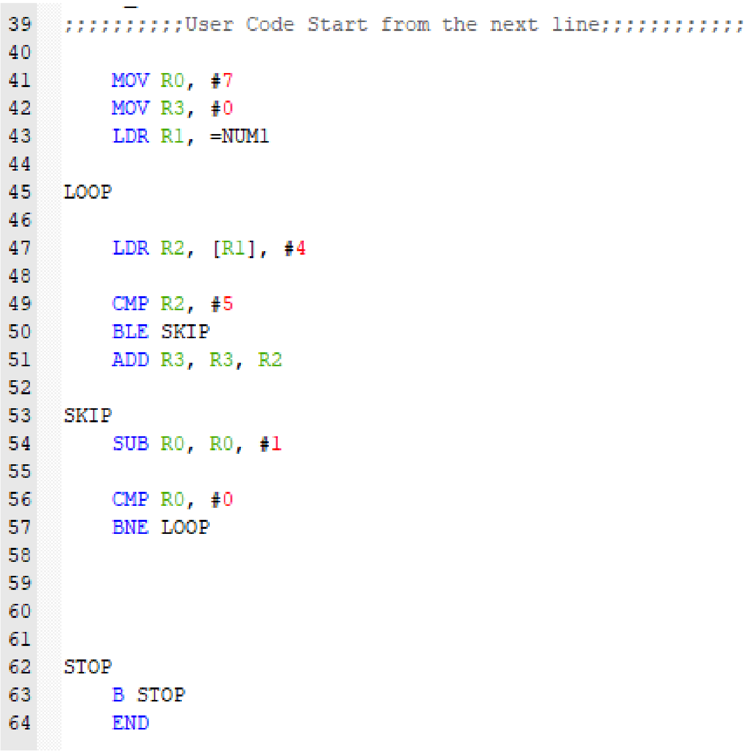
This code have to add any number greater than 5 to a summation value initialized with 0, first I created a summation value with 0, then give the N a value =7 to iterate the loop with it, then initialized an array with positive and negative numbers.



```

;Your Data section
;AREA DATA
; AREA MYRAM, DATA, READWRITE


SUM DCD 0
SUMP DCD SUM
N DCD 7
NUM1 DCD 3, -7, 2, -2, 10, 20, 30


```

Figure 12 Program #1 picture 1

Then I put a value by the number of the array length to iterate the loop by it, then I put a zero value in another register to branch the loop when it finishes. inside the loop I put the value of every element in the array and updated the address by 4 bytes after getting the index value as i++ in C language. and after getting the value from each index I compared it with number 5 and check if the value is list or equal to this number so if it's list or equal to number 5 it will branch to skip label there it will subtract 1 from the #7 which it's the array length end continue to compare if it's three each the number 0 which means the end of the array or not on the other hand if the number is greater than #5 it will skip the branch and add the number into a register I called it register #3 and after finishing the loop I put the value in the saved memory called Sum.

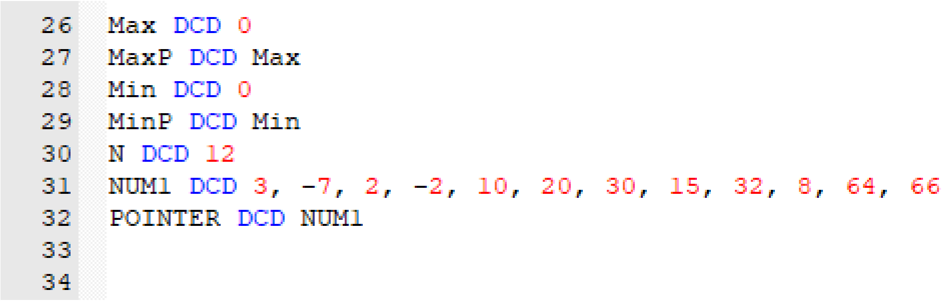


```
;;;;;;;;;;User Code Start from the next line;;;;;;;;;;;;

    MOV R0, #7
    MOV R3, #0
    LDR R1, =NUM1

LOOP

    LDR R2, [R1], #4

    CMP R2, #5
    BLE SKIP
    ADD R3, R3, R2

SKIP
    SUB R0, R0, #1

    CMP R0, #0
    BNE LOOP




STOP
    B STOP
    END
```

Figure 13 Program #1 picture 2

Program #2 Min & Max numbers

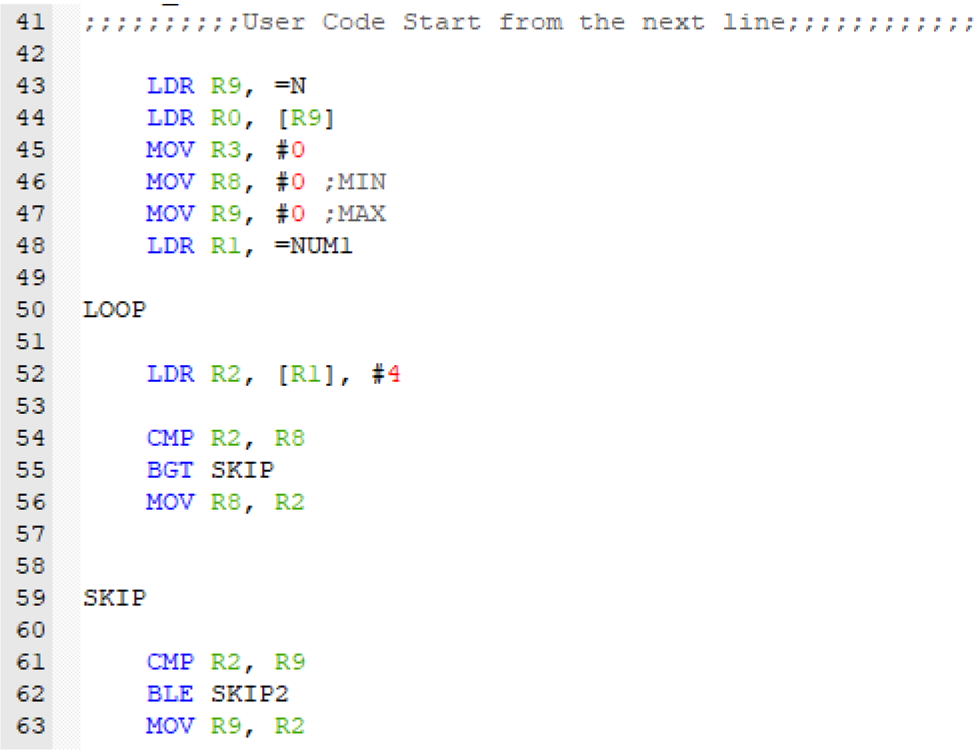
In the area I defined it a Max label and the min label and label called N have the number of elements and the label called num1 have 12 numbers as an array negative and positive numbers and old labels are defined as wards of four bytes.



```
Max DCD 0
MaxP DCD Max
Min DCD 0
MinP DCD Min
N DCD 12
NUM1 DCD 3, -7, 2, -2, 10, 20, 30, 15, 32, 8, 64, 66
POINTER DCD NUM1
```

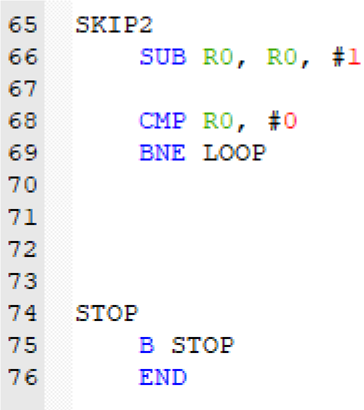
Figure 14 Program #2 picture 1

From line 43 to 48 I loaded the labels of array and number of elements and put them in registers and define the three registers of #0 one to know the minimum and one to know the maximum and one to compare the loop if it reached the end. Inside the loop I loaded the first element from the array and put the value in register 2 then added four bytes to address so I can move to the next element now after loading the first element I compared it with the minimum value which is stored before as #0 so if the number is greater then the previous stored value then skip to check the number with the Max value but if the number is less or equal to the previous saved value then move the value to the minimum register Then when skip to the Max register converter it will compare the value with the previous saved value in register 9 as a maximum value and compare if it's greater then the previous value so if it's greater it will replace the value and put the biggest value in the maximum register and if not it will skip two label SKIP2 Here it will subtract one from the 12 elements and compare the new value with #0 so if it reached zero it will break from the loop and if not it will continue iterative the loop.



```
;;;;;;;;;;User Code Start from the next line;;;;;;;;;;;;

    LDR R9, =N
    LDR R0, [R9]
    MOV R3, #0
    MOV R8, #0 ;MIN
    MOV R9, #0 ;MAX
    LDR R1, =NUM1

LOOP

    LDR R2, [R1], #4

    CMP R2, R8
    BGT SKIP
    MOV R8, R2


SKIP

    CMP R2, R9
    BLE SKIP2
    MOV R9, R2
```

Figure 15 Program #2 picture 2

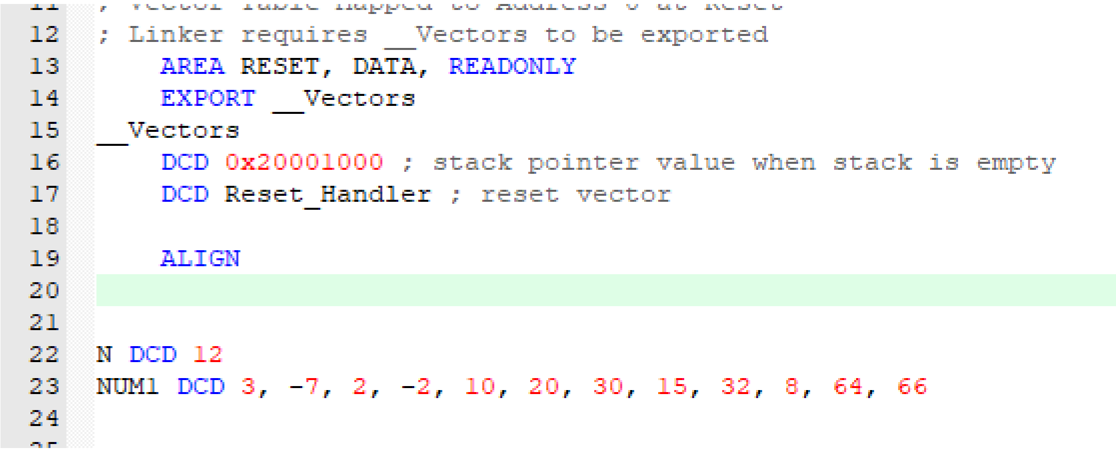


```
SKIP2
    SUB R0, R0, #1

    CMP R0, #0
    BNE LOOP




STOP
    B STOP
    END
```

Figure 16 Program #2 picture 3

Program #3 even and would arrays

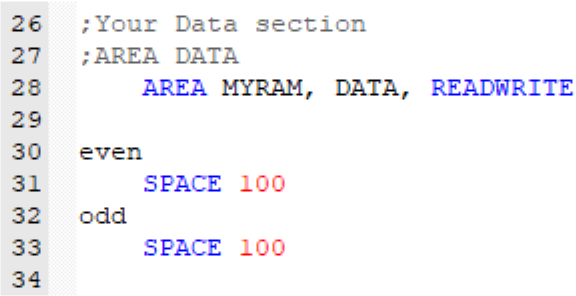
I defined the array of numbers and a label called N in the read only area so I can read them freely because there is no need to change the values



```
; Linker requires __Vectors to be exported
    AREA RESET, DATA, READONLY
    EXPORT __Vectors
__Vectors
    DCD 0x20001000 ; stack pointer value when stack is empty
    DCD Reset_Handler ; reset vector

    ALIGN


N DCD 12
NUM1 DCD 3, -7, 2, -2, 10, 20, 30, 15, 32, 8, 64, 66

```

Figure 17 Program #3 picture 1

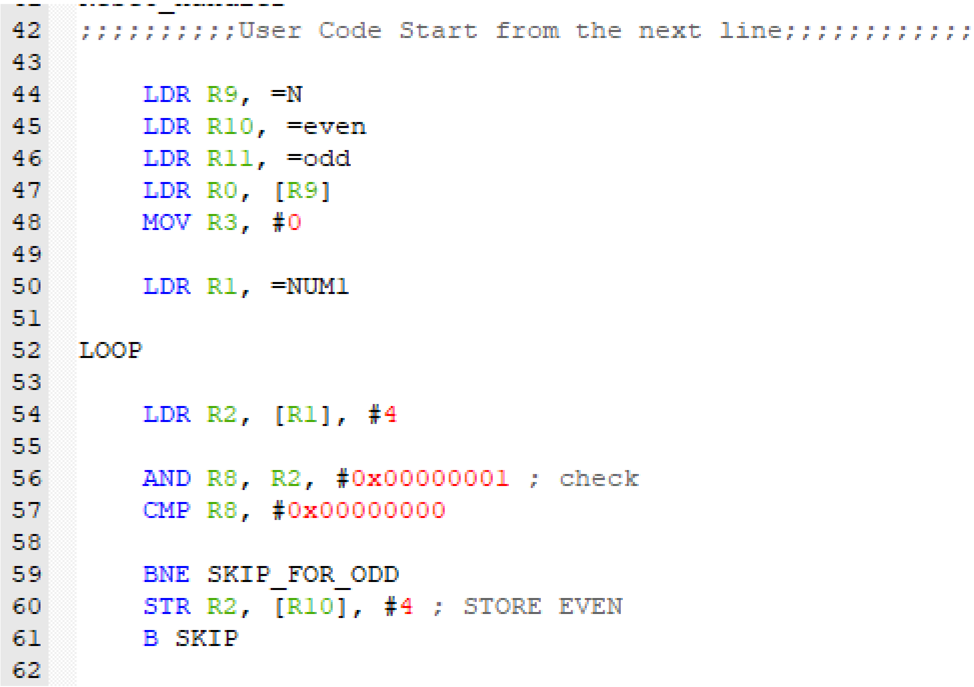
And define the two arrays in the read write area so I can change the values later I defined it two arrays even and odd each of them is made of 100 bites as empty elements, I had to do this because if I did not, the program will overwrite the even over the odd or the opposite because the 2 arrays are saved in the same addresses



```
26  ;Your Data section
27  ;AREA DATA
28      AREA MYRAM, DATA, READWRITE
29
30  even
31      SPACE 100
32  odd
33      SPACE 100
34
```

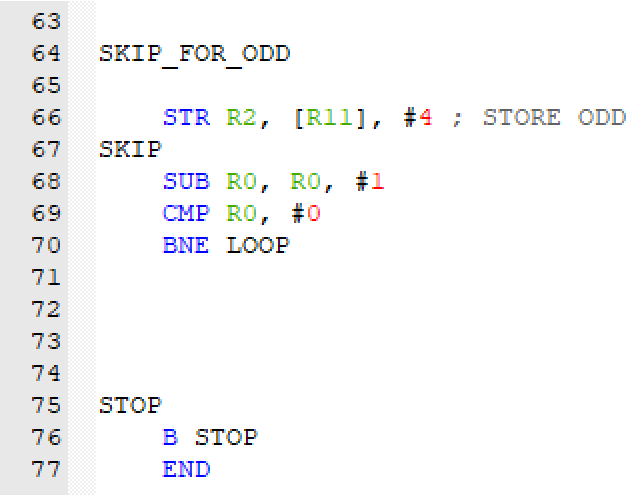
Figure 18 Program #3 picture 2

From line 44 -50 all I did is to define registers for every label the even the odd the array and the number of elements in the array and the register with initialized number zero to compare the loop if it end or not, Inside the loop I grabbed the first value from the array then added four bytes to the address so I can move to the next element in the array then I made an AND operation with the value 0x00000001 in hexadecimal this trick will check if the number ends with one or zero byte so if the answer was one this means the number is odd and if the answer was zero this means the number is even and based on this trick I can spirit the even numbers of the odd numbers in different arrays. There is still the code branch to the even or told based on the answer above and store the value in the even or odd array. at the end of the code, it's subtract 1 from the register that I have the number of elements in the array and compare if it reaches the zero or not so it breaks the loop or iterative it.



```
;;;;;;;;;;User Code Start from the next line;;;;;;;;;;;;

    LDR R9, =N
    LDR R10, =even
    LDR R11, =odd
    LDR R0, [R9]
    MOV R3, #0

    LDR R1, =NUM1

LOOP

    LDR R2, [R1], #4

    AND R8, R2, #0x00000001 ; check
    CMP R8, #0x00000000

    BNE SKIP_FOR_ODD
    STR R2, [R10], #4 ; STORE EVEN
    B SKIP

```

Figure 19 Program #3 picture 3



```

SKIP_FOR_ODD

    STR R2, [R11], #4 ; STORE ODD
SKIP
    SUB R0, R0, #1
    CMP R0, #0
    BNE LOOP




STOP
    B STOP
    END
```

Figure 20 Program #3 picture 4

Conclusion

We can conclude That we can use some functions that make it easier to iterative some programs such as ARM's Autoindexing Post-Indexing Addressing Mode and have much less code and less pressure on CPU as we limited the number of operations specially in loops.

we can conclude that we have to focus on the operations when we are using the signed or unsigned numbers in signed numbers, we can use bigger or less to compare, but in unsigned numbers we use high and low, we must focus on this because the update of flags will change in a way that we cannot follow it

References

The lab manual