Faculty of Engineering and Technology

Electrical and Computer Engineering Department

Computer Design Lab – ENCS4110

Experiment No. 3 Report

ARM's Flow Control Instructions

Prepared by:

Dana Ghnimat

Partners:
no partners

Instructor: Dr. Abualseoud Hanani
Assistant: Eng. Hanan Awawdeh

Section: 2

Date: 15/08/2023

Abstract

In this experiment we aim to understand ARM assembly language and its instructions using Keil uVision5 program, and we learn ARM branch instructions and their implementation, as well as we understand the importance of condition flags and how we can manipulate them to use branches, and finally we learn to investigate how to use strings in Keil uVision5.

Table of Contents

Table of figures:

Theory

ARM Register Set

There are basically two types of registers - general purpose registers and special purpose registers. General purpose registers hold data or addresses.
The letter r is placed before the registration number to identify them.
For example, the label r4 is assigned to register 4. This figure shows the active registers available in user mode, which is a protected state commonly used to run programmers. There are seven different modes the processor can operate in, which we'll run through in a moment.
The registers in this example are all 32 bits in size.
There is up to 18 active registers are available:
16 data registers and 2 processor status registers.

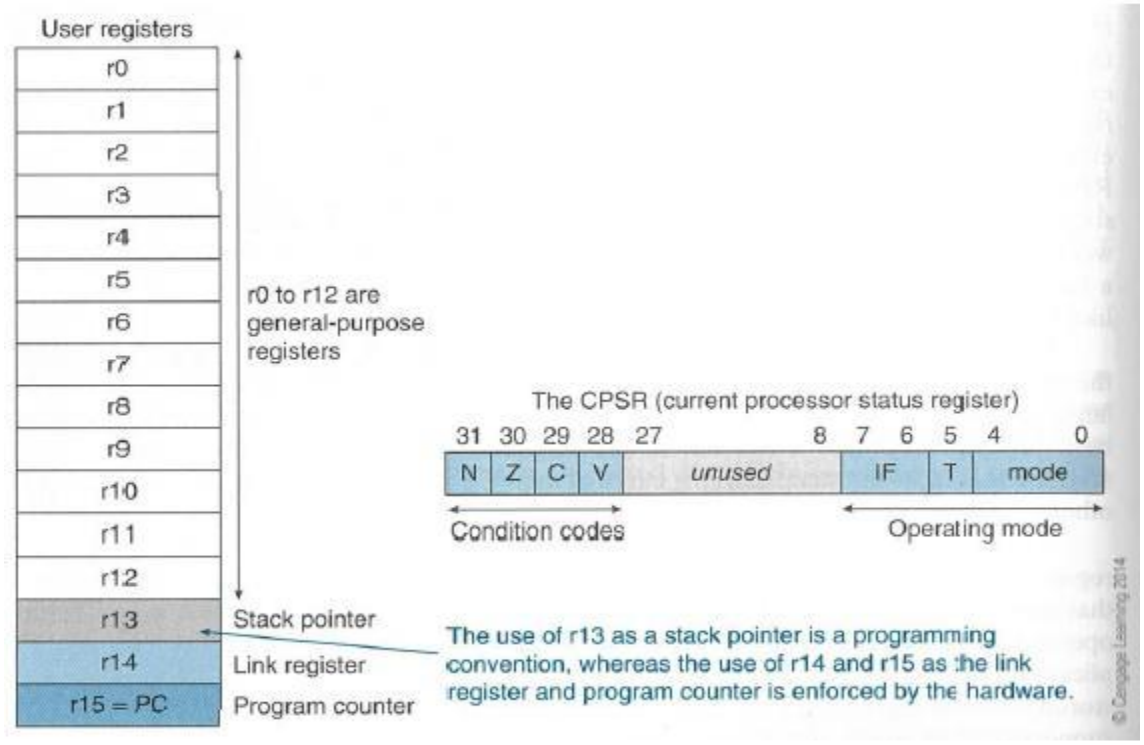
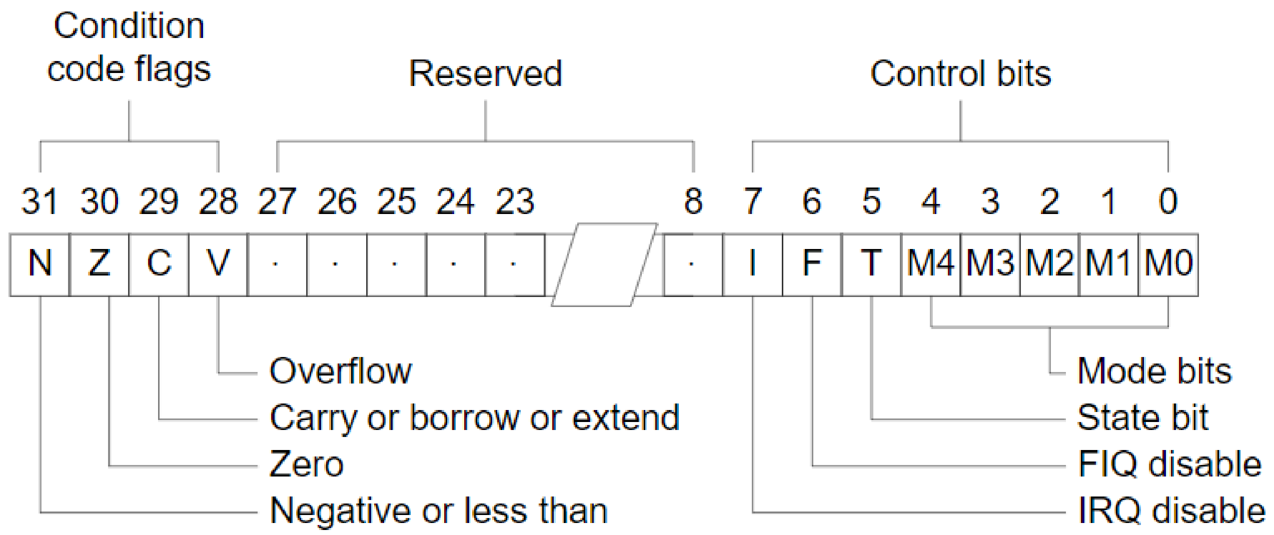

User registers
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13
r14
r15 = PC
r0 to r12 are general-purpose registers
Stack pointer
Link register
Program counter
The CPSR (current processor status register)
31 30 29 28 27 8 7 6 5 4 0
N Z C V unused IF T mode
Condition codes
Operating mode
The use of r13 as a stack pointer is a programming convention, whereas the use of r14 and r15 as the link register and program counter is enforced by the hardware.
© Cengage Learning 2014

Figure 1 ARM register set

The data registers are labeled by the programmer from r0 to r15.
ARM processors contain three registers:
r13, r14 and r15, each assigned to a specific task or unique function. To distinguish them from other registers, they are usually assigned separate labels. The color register indicates which special register has been assigned.
Register 13 is used as a stack pointer (SP) and stores the top of the stack in the current processor mode.
The r14 register is called the link register (LR) and is where the kernel sets the return address each time it calls a subroutine. Register 15 is the program counter (PC) and contains the address of the next instruction to be fetched by the processor.

Current Program Status Register

CPSR is used by the ARM core to monitor and control internal functions. The vacant area has been set aside for future development as well its used in flags, status, extension, and control are the four fields of the CPSR, each of which is 8 bits wide.



N: Negative

The N flag is set by an instruction if the result is negative. In fact, N is set to the two's complement sign bit (bit 31).

Z: Zero

The Z flag is set if the result of the instruction set the flag to zero.

C: Carry (or Unsigned Overflow)

The C flag is set if the result of the unsigned operation exceeds the 32-bit result register. For instance, this bit can be used to implement 64-bit unsigned arithmetic.

V: (Signed) Overflow

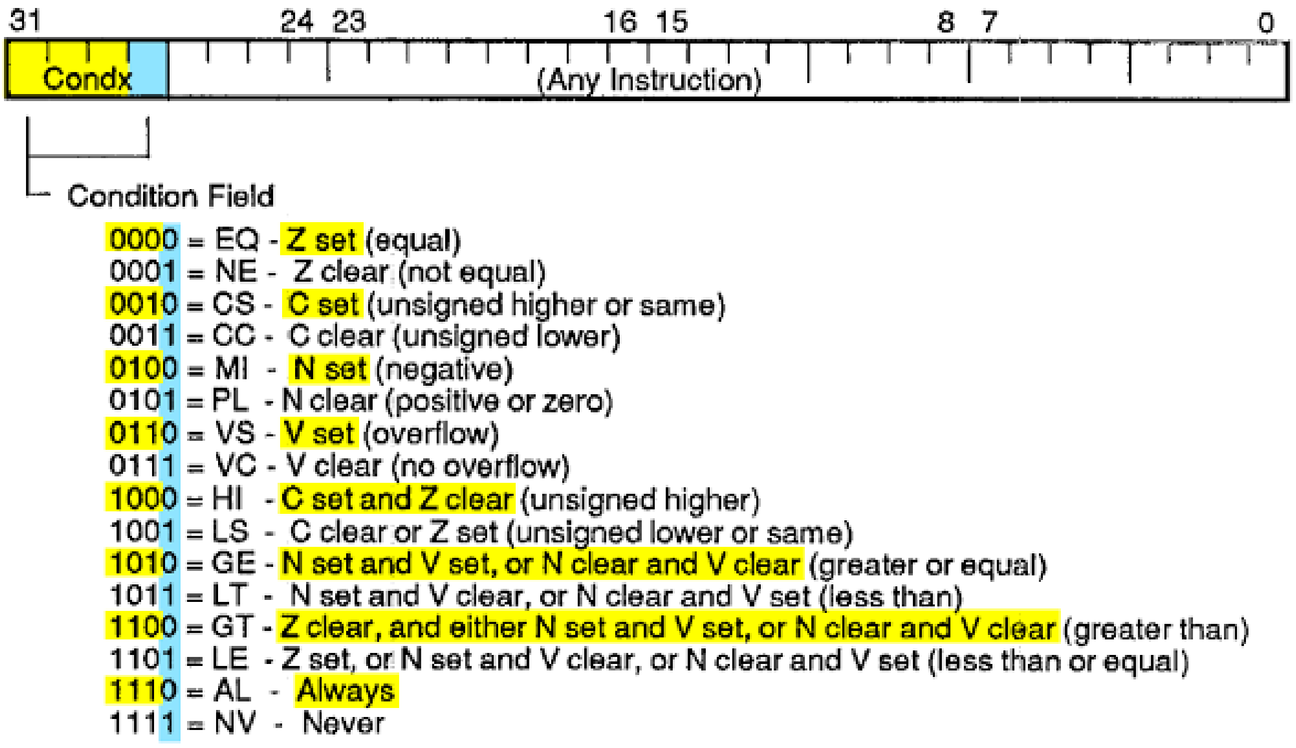
The V flag works like the C flag, but for signed operations. For example, 0x7fffffff is the largest two's complement positive integer that can be represented in 32 bits, so 0x7ffffff + 0x7ffffff triggers a signed overflow, but not an unsigned (or carry) overflow:

the result, 0xfffffffe, is true if interpreted as an unsigned quantity, but represents a negative value (-2) if interpreted as a signed quantity.

Conditional Execution

An important element of any computer is the ability of the program to modify what it does based on different conditions. Most computers provide conditional branch instructions, which move execution to another part of the program based on different conditional flags. For example, consider the code if (x == 0) { do_something }. Compiled into assembly code, this first checks the value of variable x and sets the flag Zero if x is 0. Then a conditional branch instruction jumps through the do_something code if the Zero flag is undefined. ARM processors go much further than others:

each statement becomes a conditional statement. Each statement consists of one of 16 conditions and the statement is executed only if the condition is true; otherwise, the instruction is ignored. (This is also known as predication.) The motive is to avoid unnecessary code jumps.

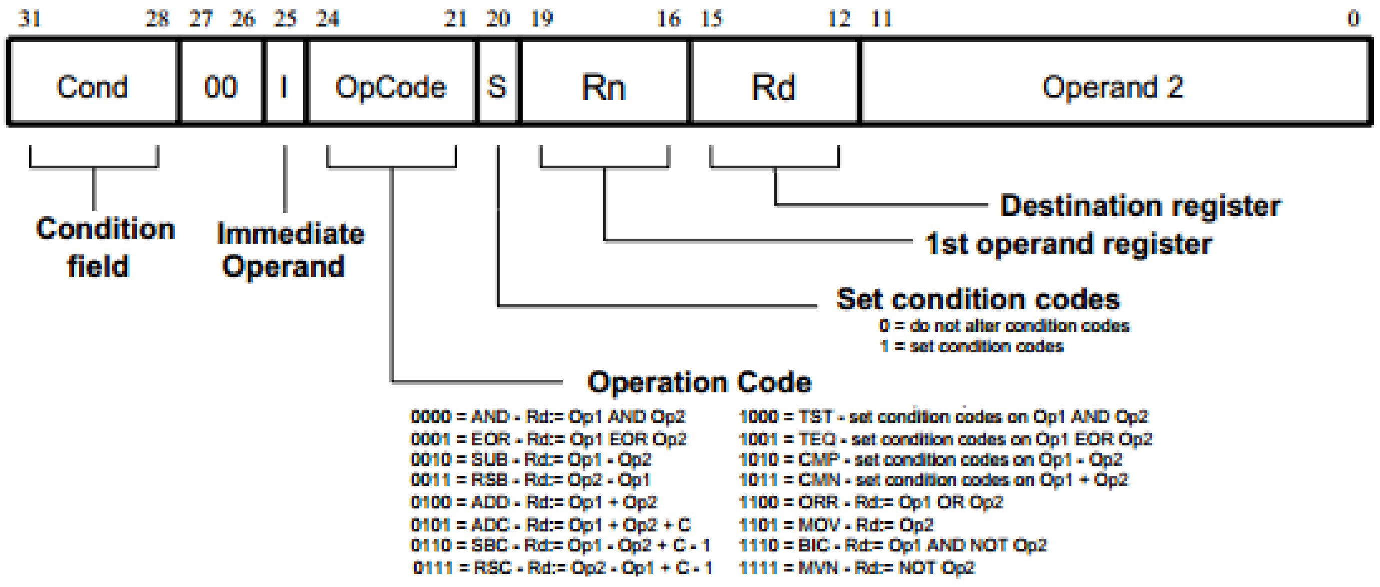


0000	= EQ	-	Z set (equal)
0001	= NE	-	Z clear (not equal)
0010	= CS	-	C set (unsigned higher or same)
0011	= CC	-	C clear (unsigned lower)
0100	= MI	-	N set (negative)
0101	= PL	-	N clear (positive or zero)
0110	= VS	-	V set (overflow)
0111	= VC	-	V clear (no overflow)
1000	= HI	-	C set and Z clear (unsigned higher)
1001	= LS	-	C clear or Z set (unsigned lower or same)
1010	= GE	-	N set and V set, or N clear and V clear (greater or equal)
1011	= LT	-	N set and V clear, or N clear and V set (less than)
1100	= GT	-	Z clear, and either N set and V set, or N clear and V clear (greater than)
1101	= LE	-	Z set, or N set and V clear, or N clear and V set (less than or equal)
1110	= AL	-	Always
1111	= NV	-	Never

Figure 2 Conditional execution

ARM Instruction Format:

Arm instruction contain 32-bit, The condition field is 4 bits wide and sits between the immediate flag, which signals that operand 2 holds an immediate value, and the condition-set flag, which we can use to update the status register during an operation (more on these later). Notice it's the opcode that determines the operation—such as addition, subtraction, or exclusive OR—that the processor will perform.



0000 = AND - Rd:= Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2
0011 = RSB - Rd:= Op2 - Op1
0100 = ADD - Rd:= Op1 + Op2
0101 = ADC - Rd:= Op1 + Op2 + C
0110 = SBC - Rd:= Op1 - Op2 + C - 1
0111 = RSC - Rd:= Op2 - Op1 + C - 1
1000 = TST - set condition codes on Op1 AND Op2
1001 = TEQ - set condition codes on Op1 EOR Op2
1010 = CMP - set condition codes on Op1 - Op2
1011 = CMN - set condition codes on Op1 + Op2
1100 = ORR - Rd:= Op1 OR Op2
1101 = MOV - Rd:= Op2
1110 = BIC - Rd:= Op1 AND NOT Op2
1111 = MVN - Rd:= NOT Op2

Figure 3 ARM data-processing instruction

Procedure and Lab work

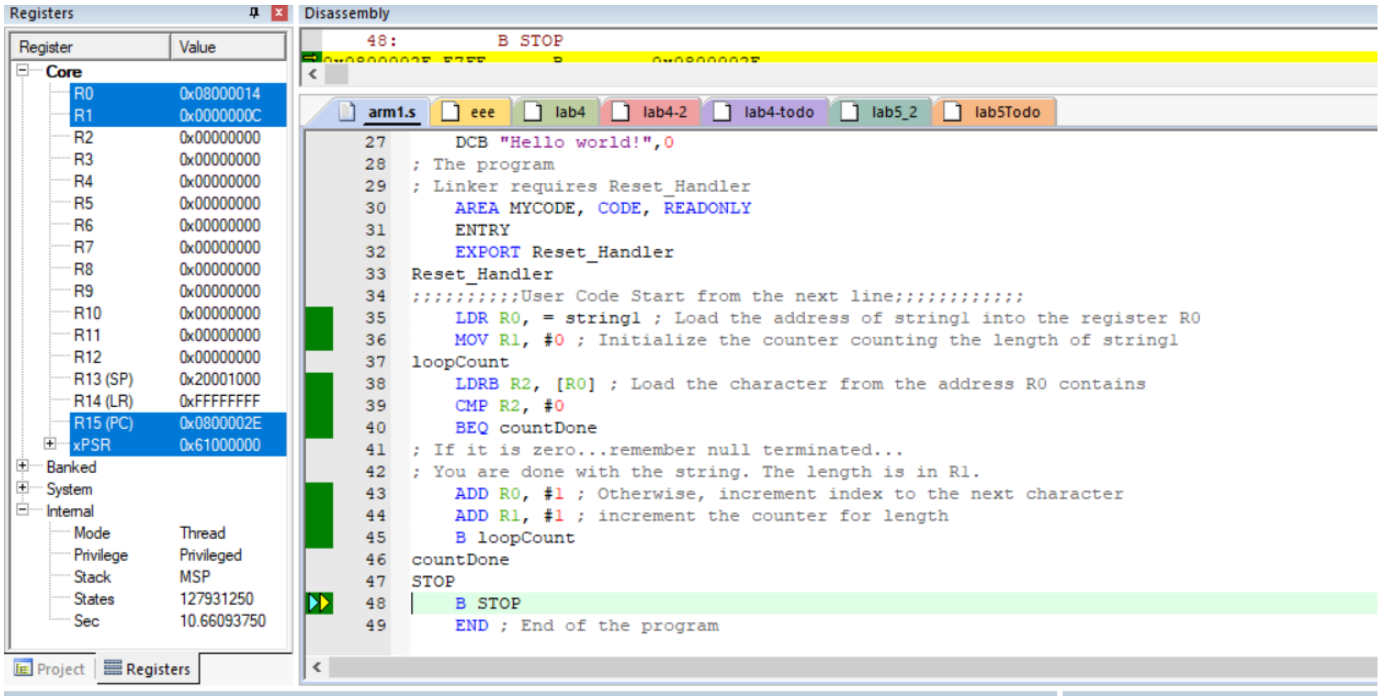
Part 1: Examples of Using Branch Instructions

Example1:

```
; This program will count the length of a string. Directives

        PRESERVE8
        THUMB
; Vector Table Mapped to Address 0 at Reset
; 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
; **************************************************************************
; Byte array/character string
; DCB type declares that memory will be reserved for consecutive bytes
; You can list comma separated byte values, or use "quoted" characters.
; The ,0 at the end null terminates the character string. You could also use "\0".
; The zero value of the null allows you to tell when the string ends.
; The DCB directive allocates one or more bytes of memory, and defines the initial
; runtime contents of the memory.
; Example Unlike C strings, ARM assembler strings are not null-terminated.
; You can construct a null-terminated C string using DCB as follows: C_string DCB "C_string",0
;**************************************************************************
string1
        DCB "Hello world!",0
; The program
; Linker requires Reset_Handler
        AREA MYCODE, CODE, READONLY
        ENTRY
        EXPORT Reset_Handler
Reset_Handler
;;;;;;;;;User Code Start from the next line;;;;;;;;;;;
        LDR R0, = string1 ; Load the address of string1 into the register R0
        MOV R1, #0 ; Initialize the counter counting the length of string1
loopCount
        LDRB R2, [R0] ; Load the character from the address R0 contains
        CMP R2, #0
        BEQ countDone
; If it is zero...remember null terminated...
; You are done with the string. The length is in R1.
        ADD R0, #1 ; Otherwise, increment index to the next character
        ADD R1, #1 ; increment the counter for length
        B loopCount
countDone
STOP
        B STOP
         END ; End of the program
```

Which give us the results as this when we build and compile the program:


Registers

Register	Value
Core	
R0	0x08000014
R1	0x0000000C
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20001000
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x0800002E
xPSR	0x61000000
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	127931250
Sec	10.66093750

```
    48:          B STOP
0x0800002E E7FE      B        0x0800002E
```

```
        DCB "Hello world!",0
; The program
; Linker requires Reset_Handler
        AREA MYCODE, CODE, READONLY
        ENTRY
        EXPORT Reset_Handler
Reset_Handler
;;;;;;;;;User Code Start from the next line;;;;;;;;;;;;
        LDR R0, = string1 ; Load the address of string1 into the register R0
        MOV R1, #0 ; Initialize the counter counting the length of string1
loopCount
        LDRB R2, [R0] ; Load the character from the address R0 contains
        CMP R2, #0
        BEQ countDone
; If it is zero...remember null terminated...
; You are done with the string. The length is in R1.
        ADD R0, #1 ; Otherwise, increment index to the next character
        ADD R1, #1 ; increment the counter for length
        B loopCount
countDone
STOP
        B STOP
        END ; End of the program
```

Figure 4 Example1

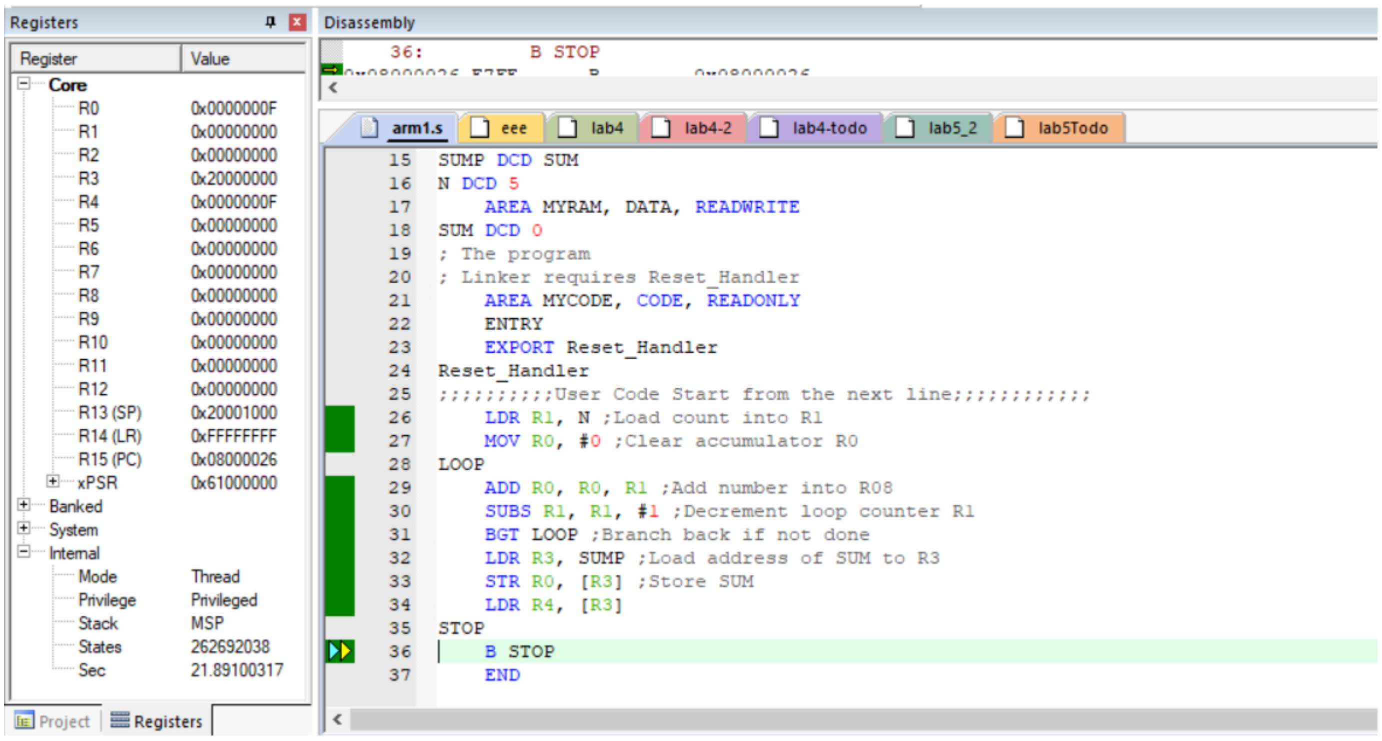
The code went as following:

The address of the string is loaded into R0, then we get R1 as a counter to count the string’s characters, we get into the loop, we load the first letter in the string into R2, and compare it to 0 which is null, since its not null or the end of the string we increment R0 to point into the next character, then we add 1 to R1 as a count as so until it reached the end of the string the branch go to countDone and exit the program, the result of the count is stored in R2 which equal to 0x0C or 12 in decimal.

Example2:

```
;;; Directives
        PRESERVE8
        THUMB
; Vector Table Mapped to Address 0 at Reset
; 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
;Your Data section
;AREA DATA
; AREA MYRAM, DATA, READWRITE
SUMP DCD SUM
N DCD 5
        AREA MYRAM, DATA, READWRITE
SUM DCD 0
; The program
; Linker requires Reset_Handler
        AREA MYCODE, CODE, READONLY
        ENTRY
        EXPORT Reset_Handler
Reset_Handler
;;;;;;;;;User Code Start from the next line;;;;;;;;;;;
        LDR R1, N ;Load count into R1
        MOV R0, #0 ;Clear accumulator R0
LOOP
        ADD R0, R0, R1 ;Add number into R08
        SUBS R1, R1, #1 ;Decrement loop counter R1
        BGT LOOP ;Branch back if not done
        LDR R3, SUMP ;Load address of SUM to R3
        STR R0, [R3] ;Store SUM
        LDR R4, [R3]
STOP
        B STOP
        END
```

Exciting the code above will give us the result as:



Register	Value
Core	
R0	0x0000000F
R1	0x00000000
R2	0x00000000
R3	0x20000000
R4	0x0000000F
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20001000
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x08000026
xPSR	0x61000000
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	262692038
Sec	21.89100317

```
    36:             B STOP
```

```
SUMP DCD SUM
N DCD 5
    AREA MYRAM, DATA, READWRITE
SUM DCD 0
; The program
; Linker requires Reset_Handler
    AREA MYCODE, CODE, READONLY
    ENTRY
    EXPORT Reset_Handler
Reset_Handler
;;;;;;;;;;User Code Start from the next line;;;;;;;;;;;
    LDR R1, N ;Load count into R1
    MOV R0, #0 ;Clear accumulator R0
LOOP
    ADD R0, R0, R1 ;Add number into R08
    SUBS R1, R1, #1 ;Decrement loop counter R1
    BGT LOOP ;Branch back if not done
    LDR R3, SUMP ;Load address of SUM to R3
    STR R0, [R3] ;Store SUM
    LDR R4, [R3]
STOP
    B STOP
    END
```

Figure 5 Example2

Noting that: the counter load into R1, and initiate R0 with 0, then we enter the loop, we add R1 into R0, which will be R0 = 0x5 for first loop, then R1 decrement by 1, R1 is greater than 0 we stay in the loop, we load the address SUMP into R3 then we store the value of R0 into R3, and load the value by getting it load to R4 again, the loop goes on until R1 equal to zero which will give us the sum up to 0x0F or 15 in decimal, the answer is right and we see that in the value of R4.

Lab work:

Write an ARM assembly language program CountVowelsOne.s to count how many vowels and how many non-vowels are in the following string. "ARM assembly language is important to learn!",0

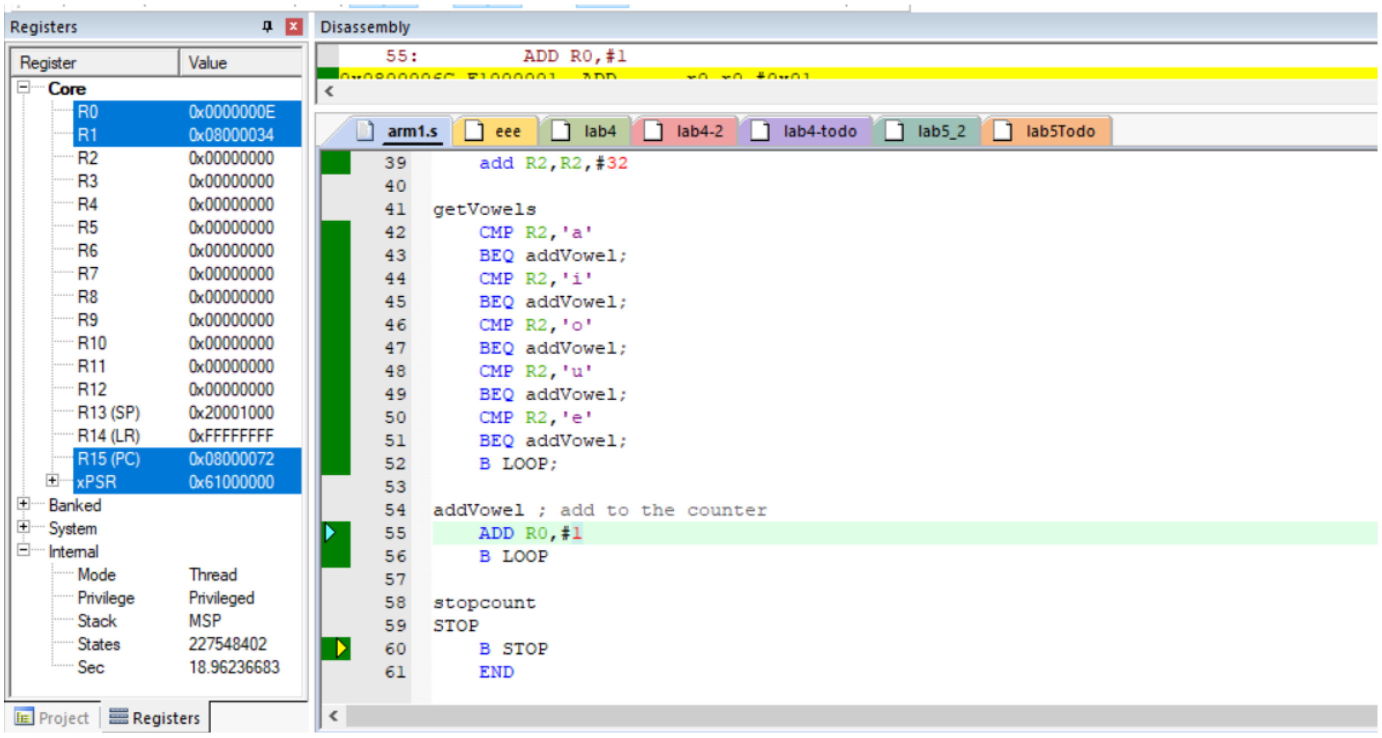
```
;;; Directives
        PRESERVE8
        THUMB
; Vector Table Mapped to Address 0 at Reset
; Linker requires __Vectors to be exported
        AREA RESET, DATA, READONLY
        EXPORT __Vectors
__Vectors
        DCD 0x20001000 ; stack pointer value when stack is empty
        DCD Reset_Handler ; reset vector
        ALIGN
;Your Data section
;AREA DATA
; AREA MYRAM, DATA, READWRITE
string1
        DCB "ARM assembly language is important to learn!",0
        AREA MYRAM, DATA, READWRITE
SUM DCD 0
; The program
; Linker requires Reset_Handler
        AREA MYCODE, CODE, READONLY
        ENTRY
        EXPORT Reset_Handler
Reset_Handler
;;;;;;;;;;User Code Start from the next line;;;;;;;;;;;;
        LDR R1, =string1 ;Load string into R1
        MOV R0, #0 ;Clear accumulator R0
LOOP
        LDRB R2,[R1]
        CMP R2,#0
        BEQ stopcount
        ADD R1,#1
        CMP R2,' ' ; compare it to jump character '' we don't want to count it since it not a vowel
        BEQ LOOP
        CMP R2,#90 ; character is uppercase
        BLS returnSmall
        B getVowels
returnSmall
        add R2,R2,#32
```

```
getVowels
        CMP R2,'a'
        BEQ addVowel;
        CMP R2,'i'
        BEQ addVowel;
        CMP R2,'o'
        BEQ addVowel;
        CMP R2,'u'
        BEQ addVowel;
        CMP R2,'e'
        BEQ addVowel;
        B LOOP;

addVowel ; add to the counter
        ADD R0,#1
        B LOOP

stopcount
STOP
        B STOP
        END
```

And it will give us the answer in R0 which is 14 vowel character stored in R0.



Register	Value
Core	
R0	0x0000000E
R1	0x08000034
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20001000
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x08000072
xPSR	0x61000000
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	227548402
Sec	18.96236683

```
    55:          ADD R0,#1
0x0800006C F1000001  ADD       r0,r0,#0x01
```

```
        add R2,R2,#32

getVowels
        CMP R2,'a'
        BEQ addVowel;
        CMP R2,'i'
        BEQ addVowel;
        CMP R2,'o'
        BEQ addVowel;
        CMP R2,'u'
        BEQ addVowel;
        CMP R2,'e'
        BEQ addVowel;
        B LOOP;

addVowel ; add to the counter
        ADD R0,#1
        B LOOP

stopcount
STOP
        B STOP
        END
```

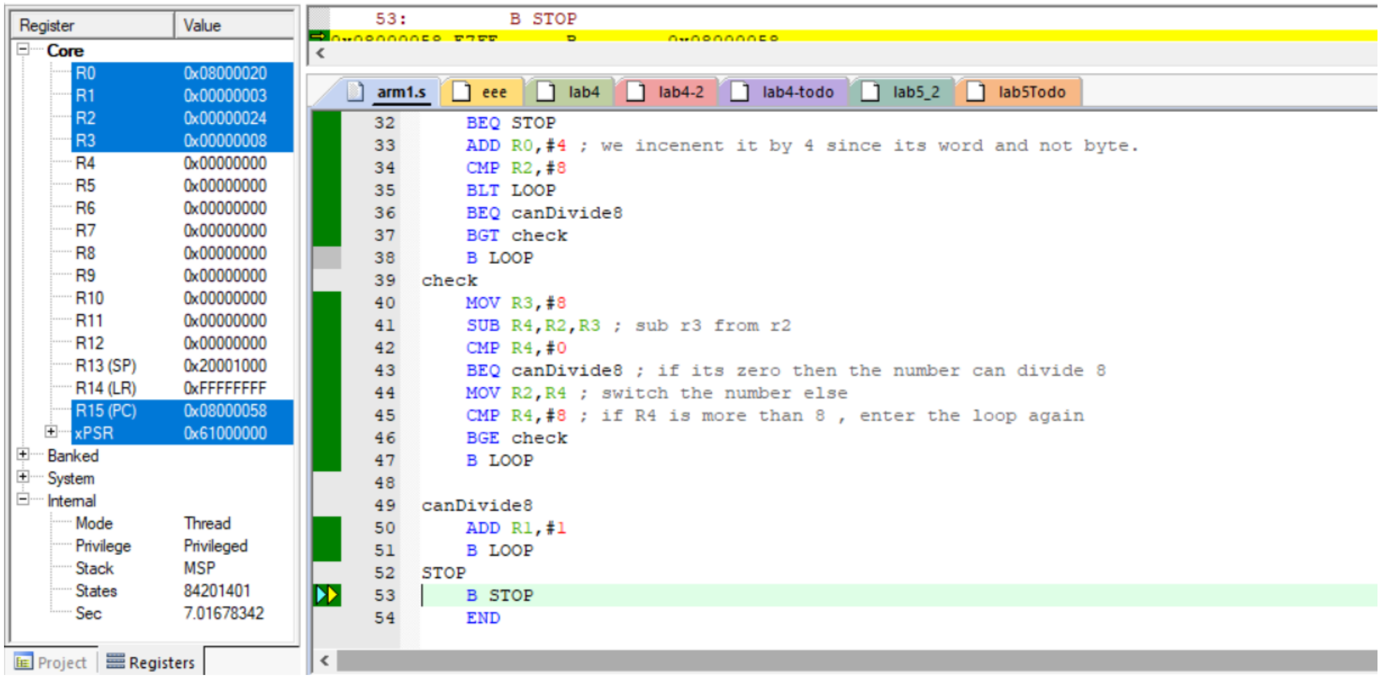
Figure 6 Lab work

To do:

We got asked in class to write a code that will check if the number can mod 8.

```
;;; Directives
        PRESERVE8
        THUMB
; Vector Table Mapped to Address 0 at Reset
; 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
;Your Data section
;AREA DATA
; AREA MYRAM, DATA, READWRITE
SUMP DCD SUM
N DCD 5,16,20,8,64,'$'
        AREA MYRAM, DATA, READWRITE
SUM DCD 0
; The program
; Linker requires Reset_Handler
        AREA MYCODE, CODE, READONLY
        ENTRY
        EXPORT Reset_Handler
Reset_Handler
;;;;;;;;;;User Code Start from the next line;;;;;;;;;;;;
        LDR R0, =N ;Load count into R1
        MOV R1,#0
LOOP
        LDR R2,[R0]
        CMP R2,'$' ;which is the end of the array
        BEQ STOP
        ADD R0,#4 ; we incenent it by 4 since its word and not byte.
        CMP R2,#8
        BLT LOOP
        BEQ canDivide8
        BGT check
        B LOOP
check
        MOV R3,#8
        SUB R4,R2,R3 ; sub r3 from r2
        CMP R4,#0
        BEQ canDivide8 ; if its zero then the number can divide 8
        MOV R2,R4 ; switch the number else
        CMP R4,#8 ; if R4 is more than 8 , enter the loop again
        BGE check
        B LOOP

canDivide8
        ADD R1,#1
        B LOOP
STOP
        B STOP
        END
```



Register	Value
Core	
R0	0x08000020
R1	0x00000003
R2	0x00000024
R3	0x00000008
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20001000
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x08000058
xPSR	0x61000000
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	84201401
Sec	7.01678342

```
    53:          B STOP
0x08000058 E7FE      B        0x08000058
```

```
    BEQ STOP
    ADD R0,#4 ; we incenent it by 4 since its word and not byte.
    CMP R2,#8
    BLT LOOP
    BEQ canDivide8
    BGT check
    B LOOP
check
    MOV R3,#8
    SUB R4,R2,R3 ; sub r3 from r2
    CMP R4,#0
    BEQ canDivide8 ; if its zero then the number can divide 8
    MOV R2,R4 ; switch the number else
    CMP R4,#8 ; if R4 is more than 8 , enter the loop again
    BGE check
    B LOOP

canDivide8
    ADD R1,#1
    B LOOP
STOP
    B STOP
    END
```

Figure 7 To do

So as we see there’s three numbers in the array that can mod 8 which are 16,8 and 64 and we see in the results stored in R1 that 0x03 is the amount of numbers can mod 8, this code is looping to divide by using subtraction and checking again.

Conclusion

In this experiment we learnt how to use strings in assembly language as well as we learnt how to store and load into registers using LDR and LDRB and STR, and how we can use conditions and branches in our code to make it easier to understand and simpler and use less instructions using loops.

References

https://roboticelectronics.in/arm-registers/

https://developer.arm.com/documentation/ddi0210/c/Programmer-s-Model/The-program-status-registers

https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/condition-codes-1-condition-flags-and-codes

https://www.righto.com/2016/01/conditional-instructions-in-arm1.html

https://www.allaboutcircuits.com/technical-articles/how-to-write-assembly-basic-assembly-instructions-ARM-instruction-set/