# Lab work #4

## 1. Objective of this lab:

To explore ARM subroutine calls and implement them in Keil uVision5

- Using **BL SUB_Name**, and **MOV PC, LR** or **BX LR**
- Study and using stack

## 2. Preparation

### 2.1. ARM Subroutine/procedure/function Call

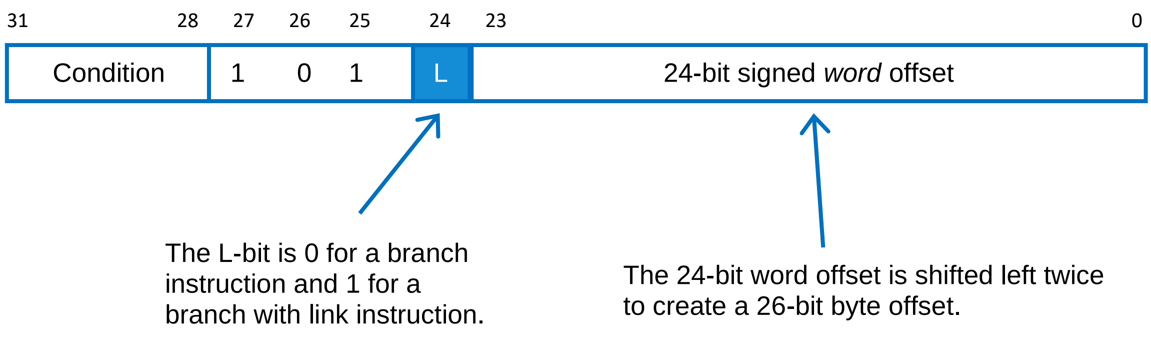
In this lab, we need to deal with function/procedure call/return in the ARM assembly language environment.

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 call **BL SUBROUTINE_NAME**. A **SUBROUTINE_NAME** is a label in the ARM program.

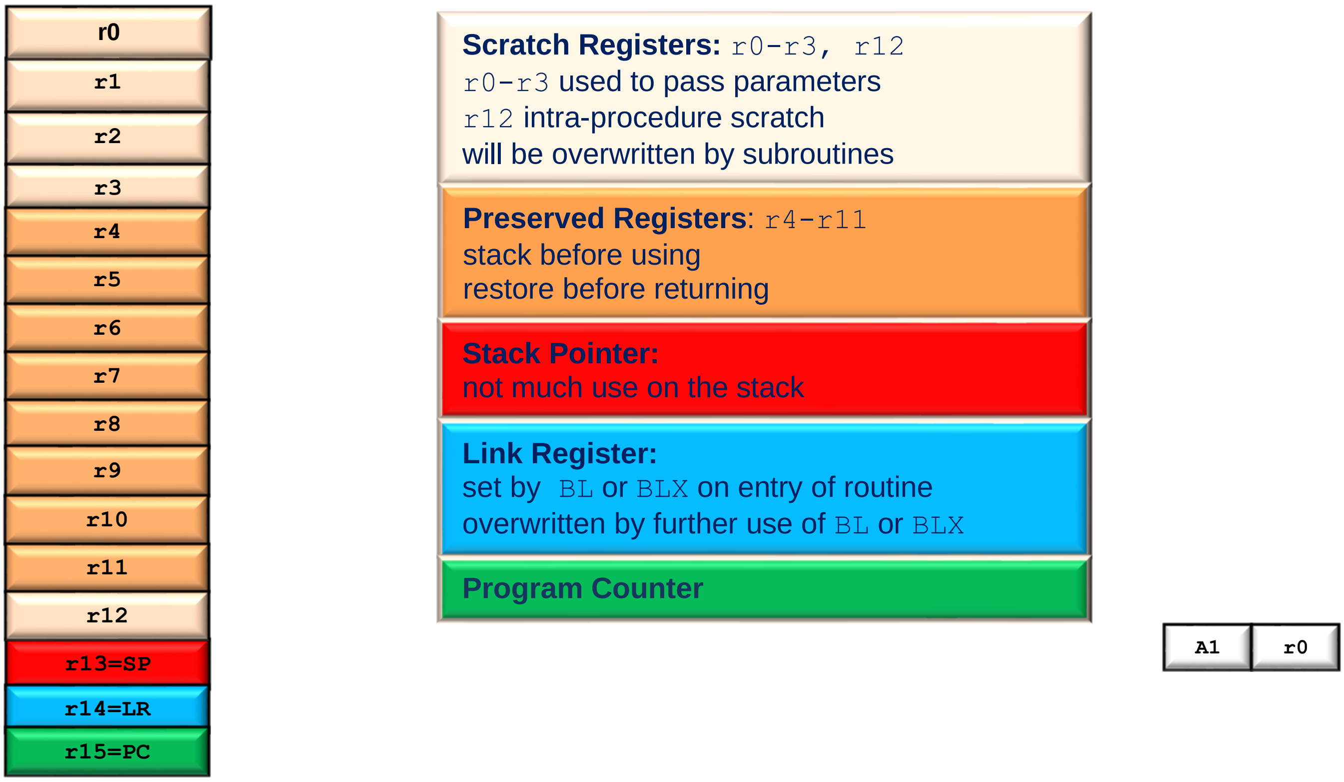
#### 2.1.1. ARM Unconditional and Conditional Subroutine Calls

```
       Mnemonic          Meaning
BL      SUB_A            ; Branch to SUB_A with link save return address in R14

CMP     R1, R2           ; branch conditionally
BLLT    SUB_B            ; if R1 < R2, then branch to SUB_B
BLLE    SUB_C            ; if R1 =< R3, then branch to SUB_C
BLGT    SUB_D            ; if R1 > R2, then branch to SUB_D
BLGE    SUB_F            ; if R1 >= R2, then branch to SUB_F

MOV     PC, LR           ; get the control of execution back after executing

BX      LR               ; Return to the calling function

Using PROC and ENDP as a pair for procedures
```

Here is the encoding format of ARM's branch and branch-with-link instructions for your reference



### 2.1.2. Register Use in the ARM Procedure Call Standard




Register Use in the ARM Procedure Call Standard

- 16 general purpose registers
- 4 arguments (allowed to change)
- 8 variables (should be retained)
- SB=stack base; SL=stack limit
- FP=frame pointer
- IP=Intra-procedure scratch

Less commonly used as such

- SP=stack pointer
- LR=link register
- PC=program counter

| | |
|---|---|
| A1 | r0 |
| A2 | r1 |
| A3 | r2 |
| A4 | r3 |
| V1 | r4 |
| V2 | r5 |
| V3 | r6 |
| V4 | r7 |
| V5 | r8 |
| V6/SB | r8 |
| V6/SB | r9 |
| V7/SL | r10 |
| V8/FP | r11 |
| IP | r12 |
| SP | r13 |
| LR | r14 |
| PC | r15 |

### 2.1.3. An Example Using a Subroutine Call

```
; The semicolon is used to lead an inline documentation
;
; When you write your program, you could have your info at the top document block
; For Example:  Your Name, Student Number, what the program is for, and what it does etc.
;
            AREA   MYCODE, CODE, READONLY
            ENTRY

;      Procedure definitions ;;;

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

;      users main program      ;;;

            LDR    R1, N             ; Load count into R1
            MOV    R0, #0            ; Clear accumulator R0
            BL     SUMUP
            LDR    R3, SUMP          ; Load address of SUM to R3
            STR    R0, [R3]          ; Store SUM
STOP        B      STOP

            AREA   MYCODE, DATA, READWRITE
SUM         DCD    0
SUMP        DCD    SUM

N           DCD    5

            END                ; End of the program
```

### 2.1.4. Introduction to Stack

The stack is a data structure, known as last in first out (LIFO).
In a stack, items entered at one end and leave 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 (pushed), the stack pointer is moving up, and as items are removed from the stack (pulled or popped), the stack pointer is moved down.

**Stack Types**

ARM stacks are very flexible since the implementation is completely left to the software. 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 different types of stacks. Normally, there are two types of the stacks depending on which way the stack grows.

1. **Ascending Stack** - When items are pushed on to the stack, the stack pointer is increasing. That means the stack grows towards higher address.
2. **Descending Stack** - When items are pushed on to the stack, the stack pointer is decreasing. That means the stack is growing towards lower address.

Depending on what the stack pointer points to we can categorize the stacks into the following two types:

1. Empty Stack - Stack pointer points to the location in which the next/first item will be stored.
   e.g. A push will store the value, and increment the stack pointer for an Ascending Stack.

2. Full Stack - Stack pointer points to the location in which the last item was stored.
   e.g. A pop will decrement the stack pointer and pull the value for an Ascending Stack

So now, we can have four possible types of stacks. They are

1. full-ascending stack,
2. full-descending stack,
3. empty-ascending stack,
4. empty-descending stack.

They can be implemented by using the register load and store instructions.

**Here are some instructions used to deal with stack:**
Push registers onto and pop registers off a full-descending stack.

```
Examples:
      PUSH {R0, R4-R7}        ; Push R0, R4, R5, R6, R7 onto the stack
      PUSH {R2, LR}           ; Push R2 and the link register onto the stack
      POP  {R0, R6, LR}       ; Pop R0, R6, and LR from the stack
      POP  {R0, R5, PC}       ; Pop R0, R5, and PC from the stack
                              ; then branch to the new PC
=========================================================================
 Reference: page 3-29 to 3-30 in "Cortex-M3 User Guide"
```

Load and store multiple registers.

```
Examples:
      STMDB       R1!,{R3-R6,R11,R12}
      LDM         R8,{R0,R2,R9}

=========================================================================
 Reference: page 3-27 to 3-28 in "Cortex-M3 User Guide"
```

### 2.1.5. Subroutine and Stack

A subroutine call can be implemented by pushing the return address on the stack and then jumping to the branch target address. When the subroutine is done, remember to pop out the saved information so that it will be able to return to the next instruction immediately after the calling point.

### 2.1.6. An Example of Using Stack

```
; The semicolon is used to lead an inline documentation
;
; When you write your program, you could have your info at the top document block
; For Example:  Your Name, Student Number, what the program is for, and what it does etc.
;
            AREA   MYCODE, CODE, READONLY
            ENTRY
            ALIGN

;      Define Procedures;;;

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 R1 and if it is the saved value
            ENDP

;      users main program      ;;;

            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 after pop
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

            AREA   MYCODE, DATA, READWRITE            S.
INITIAL_MSP EQU    0x20001000   ; Initial Main Stack Pointer Value
                         ; Allocating 1000 bytes to the stack as it grows down
Vectors     DCD    INITIAL_MSP  ; stack pointer value when stack is empty

            END                 ; End of the program
```

## 3. Lab Assignment

### 3.1. Assignment #1:

Write an ARM assembly language program **CountVowelsTwo.s** to count how many vowels and how many non-vowels are in the following string.

```
"ARM assembly language is important to learn!",0
```

You are required to implement this by using a subroutine to check if a letter is vowel or not, and count the vowels and non-vowels in the calling function.

Recommendations for writing the program:

- Put the string in the memory by using DCB.
- Use R0 to hold the address of a character in the string.
- Use R2 to be the counter for vowels.
- Use R3 to be the counter for non-vowels.
- Build the program, debug until there is no error.
- Make a screenshot to show that the build is successful with no errors.
- Run the program step by step and see how values are changing in the registers. OR just run the program and see the final results in the register R2 and R3.
- Make a screenshot to capture the results in your designated registers.

**You will hand in the following:**

1. The source code in the file **`CountVowelsTwo.s`**
2. The print out of the screen shot (print screen) to show the program has been successfully built
3. The print out of the screen shot showing the number of vowels in R2 and non-vowels in R3.

### 3.2. Assignment#2:

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

```
    For example:
The factorial of 5 is 5! = 5 x 4 x 3 x 2 x 1
The factorial of 0 is defined as 0! = 1
In general, n! = n x (n-1)!, where n is a positive integer.
If we write f(n) = n!, then f(n) = n f(n-1).
It is a recursive function.

You may try to implement it by using stack.
When you test it, you can use relatively smaller numbers such as 3, 4, 5, or 6.

For marking purpose, put the input number in the R1 and put your final result
in the register R0 or indicate it specifically in your hand-in assignment.
```

**You will hand in the following:**

1. The source code in the file **Factorial.s**
2. The print out of the screen shot (print screen) to show the program has been successfully built
3. The print out of the screen shot showing the result in R0