# Experiment 6

# Subroutines in Assembly Language

A program is made up of instructions which implement the solution to a problem. In a given program, it is often needed to perform a particular sub-task many times on different data values. So, we split the program into smaller units which solve a particular part of the problem. These task specific sub-units are termed as functions, procedures or subroutines. In assembly language, we use the word subroutine for all subprograms to distinguish between functions used in other programming languages and those used in assembly languages.

The block of instructions that constitute a subroutine can be included at every point in the main program when that task is needed. However, this would result in unnecessary waste of memory space. Rather, only one copy of the instructions that constitute the subroutine is placed in memory and can be accessed repeatedly.

## Components of Subroutines

An assembly language routine has:

1. Entry point: The location of the first instruction in the routine.
2. Parameters: The list of registers or memory locations that contain the parameters for the routine.
3. Return values: The list of registers or memory locations to save the results.
4. Working storage: The registers or memory location required by the routine to perform its task.

## Calling Subroutines

There are two ideas behind a subroutine:

1. You should be able to call the subroutine from anywhere.
2. Once the subroutine is complete, it should return back to the place that called the subroutine.

There are special instructions for transferring control to subroutines and restoring control to the main program. The instruction that transfers the control is usually termed as call, jump or branch to subroutine. The calling program is called Caller and the subroutine called is known as Callee. The instruction that transfer control back to the caller is known as Return.

Calling a subroutine requires a deviation from the default sequential execution of instructions. When a program branches to a subroutine, the processor begins execution of the instructions that make up the subroutine and branch to the subroutine by modifying the Program Counter (PC). In ARM, the Branch and Link instruction (BL) is used to Branch to subroutine.

**Example 6.1**

```
BL   my_subroutine     ; my_subroutine points to the
                       ; first line of a subroutine
```

This instruction saves the current address of program counter (PC) in the link register (LR) before placing the starting address of the subroutine in the program counter.

When the subroutine has completed its task, the processor must be able to branch back (return) to the instruction immediately following the branch instruction that invoked the subroutine. To return from the subroutine we move the value stored in the link register to program counter which returns the control to the next instruction from which subroutine was called. Thus, to return from subroutine we should use the following instruction:

**Example 6.2**

```
BX    LR             ; return back after the subroutine call
MOV   PC, LR         ; serves the same purpose
```

## ARM Application Procedure Call Standard

The ARM Application Procedure Call Standard (AAPCS) for the ARM architecture defines how subroutines can be independently written, separately complied, and assembled to work together. Some parts of the specifications are listed below:

- The first four registers R0  R3 are used to pass argument values into a subroutine and to return a result value from a function.
- The registers R4  R8, R10, and R11 are normally used to hold the values of a routine's local variables.
- ARM and THUMB, C and C++ compilers always use a full descending stack and it must be eight-byte aligned.

Although your program may work without conforming to the above specification, it is a good practice to have these specifications in mind. In particularly, when calling between C, C++,

and assembly language is needed, assembly language modules must conform to the appropriate AAPCS standard.

## Parameter Passing

When calling a subroutine, a calling program needs a mechanism to provide to the subroutine the input parameters, the operands that will be used in computation in the subroutine or their addresses. Later, the subroutine needs a mechanism to return output parameters, the results of the subroutine computation. This exchange of information between a calling program and a subroutine is referred to as parameter passing. Parameter passing may be accomplished in three different ways:

1. Place the parameters in the registers.
2. Place the parameters in a block of memory.
3. Transfer the parameters and results on the hardware stack.

The registers often provide a fast, convenient way of passing parameters and returning results. Number of available registers limit the usefulness of this method. This method can result in unforeseen side effects and it also lacks generality.

In this lab manual, we will discuss parameter passing through registers.

### Passing Parameters in Registers

This is the simplest method to pass the parameters via the registers. Memory addresses, counters and other data can be passed to the subroutines through registers. For example, a subroutine takes two strings of equal length as its input parameters. The length of the strings can be passed through the register R0 and the starting address of both the strings can be passed via the registers R1 and R2 respectively.

**Example 6.3**

```
MOV  R0, #StrLength    ; R0 contains the lenght of strings
LDR  R1, =String1      ; R1 has the starting address of first string
LDR  R2, =String2      ; R2 has the starting address of second string
BL   my_subroutine     ; Jump to subroutine
```

Passing the parameter through the registers is easy and the subroutine assume that parameters are already placed in the registers. Results of a subroutine can also returned in the registers or the memory location to store the results can be passed as parameters via the registers. This method is limited by the number of available registers.

## Types of Parameters

We can specify the parameters in various different ways that are described below:

### pass-by-value

In this method actual values are placed in a parameter list. Pass by value means you are making a copy in memory of the actual parameter's value that is passed in, a copy of the contents of the actual parameter. Changing the value in the subroutine will not change the actual value of the parameter.

### pass-by-reference

In this method the address of parameters are placed in the parameter list. Changing the value of parameter in the subroutine will also change the actual value. You must be more careful while using this method as some parameters may change their value even if you don't intend to do so.

### pass-by-name

Instead of passing either the value or the reference of the parameter a string containing the name of the parameter is passed. This method is flexible but time consuming as a lookup table must be consulted every time to access the value of the variable.

## ARM Conditional Branch Instructions

Computer programs normally execute instructions sequentially.i.e., the next instruction to be executed is the one placed at next memory location after the current one. However, to handle structures like conditional statements and loops learned in high-level languages, we need to alter this straight line control flow. Branch instructions allow this sequence to be varied. Conditional branch instructions allow the CPU to follow a course of action depending upon the result of computations. When a conditional branch occurs, a condition is checked. If the condition is true, then the jump occurs. A jump means updating the PC with the instruction to execute, which in turn causes that instruction to be fetched and run. If the condition is false, then the instruction at next memory address is executed which can be PC + 4 or PC + 2 depending upon the instruction encoding.

ARM supports different branch instructions for conditional executions. Depending on the condition these instructions transfer the control from one part of the program to other. Unlike Branch-and-Link (BL) instruction they do not save contents of Program counter (PC) register to the Link Register (LR).

| Branch | Interpretetion | Normal Uses |
|---|---|---|
| B | | Always take this branch |
| BAL | Always | Always take this branch |
| BEQ | Equal | Comparison equal or result zero |
| BNE | Not Equal | Comparison not equal or non-zero result |
| BPL | Plus | Result positive or zero |
| BMI | Minus | Result minus or negative |
| BCC | Carry Clear | Arithmetic operation didn't give carry out |
| BLO | Lower | Unsigned comparison gave lower |
| BCS | Carry Set | Arithmetic operation gave carry out |
| BHS | Higher or Same | Unsigned comparison gave higher or same |
| BVC | Overflow Clear | Signed integer operation; no overflow occured |
| BVS | Overflow Set | Signed integer operation; overflow occured |
| BGT | Greater Than | Signed integer comparison gave greater than |
| BGE | Greater or Equal | Signed integer comparison gave greater than or equal |
| BLT | Less Than | Signed integer comparison gave less than |
| BLE | Less or Equal | Signed integer comparison gave less than or equal |
| BHI | Higher | Unsigned comparison gave higher |
| BLS | Lower or Same | Unsigned comparison gave lower or same |

## Examples

Now, we consider some examples of subroutines. Parameters are passed through the registers and results are stored in memory through the registers.

**Example 6.4**

```
        PRESERVE8
        THUMB

        AREA    MyData, DATA, READWRITE
SUMP    DCD 0           ; global variable initialized to zero

        AREA    |.text|, CODE, READONLY, ALIGN=2
        ENTRY
        EXPORT __main

__main
        LDR R1, N               ; Load count into R1
        MOV R0, #0              ; Clear accumulator R0
        BL  SUMUP               ; Call the subroutine SUMUP

        LDR R3, =SUMP           ; Load address of SUM to R3
        STR R0, [R3]            ; Store accumulated SUM
        B   STOP
```

```
SUMUP  PROC                    ; Assembler directive to indicate
                               ; start of subroutine
       ADD  R0, R0, R1         ; Add number into R0
       SUBS R1, R1, #1         ; Decrement loop counter R1
       BGT  SUMUP              ; Branch back if not done
       BX   LR                 ; Restore the address of PC
       ENDP

N      DCD 5                   ; Local variable initialized to 5
       ALIGN
STOP
     END
```

**Example 6.5**

```
       PRESERVE8
       THUMB

       AREA  myDATA, DATA, READWRITE
Result SPACE 8                   ; reserves eight bytes for variable result

       AREA  Program, CODE, READONLY
       ENTRY
       EXPORT   __main

__main
       LDR  R0, =Value1        ; Load starting address of first 64-bit no.
       LDR  R1, =Value2        ; Load starting address of second 64-bit no.
       BL   SUM64

       LDR  R0, =Result
       STR  R6, [R0]           ; Store higher 32-bits to Result
       STR  R7, [R0, #4]       ; Store lower 32-bits to Result
       B    STOP

SUM64  PROC
       LDR  R2, [R0]           ; Load the value of higher 32-bits
       LDR  R3, [R0, #4]       ; Load the value of lower 32-bits
       LDR  R4, [R1]
       LDR  R5, [R1, #4]
       ADDS R7, R3, R5         ; Add with flags update
       ADC  R6, R2, R4         ; Add with carry
       BX   LR
       ENDP

STOP
```

```
    Value1  DCD  0x12A2E640, 0xF2100123
    Value2  DCD  0x001019BF, 0x40023F51

            END
```