ELECTRICAL AND ELECTRONICS ENGINEERING DEPARTMENT

Programming with Subroutines and Stack

Experiment 1 - Programming with Subroutines, Parameter Passing, Utilization of Stacks & the Concept of Recursion

Objectives

A subroutine is a reusable program module. A main program can call or jump to the subroutine one or more times. The stack is used in several ways when subroutines are called. Stack is an abstract data type that holds information in the memory and it is used in several ways especially when subroutines are called. Recall that until now, you have not been able to call a subroutine inside a subroutine, since you lose the address where the main program is left. With the utilization of stacks, you will be able to program with nested subroutines and moreover, you can implement recursive functions. In this lab you will learn:

- How to write subroutines and call them from the main program
- Ways to pass parameters to and from subroutines
- How to create flowcharts and develop pseudo-codes
- The function of the stack and the stack pointer
- Ways to pass parameters to and from stack
- Understanding of the hierarchy within a code
- Use of nested subroutines
- The concept of recursive programming

1 Background Information

1.1 Subroutines

A subroutine is a program module that is separate from the main or calling program. Frequently the subroutine is called by the main program many times, but the code for the subroutine only needs to be written once. When a subroutine is called, program control is transferred from the main program to the subroutine. When the subroutine finishes executing, control is returned to the main program. Using subroutines saves memory, but a greater benefit is improved program organization. In future labs, you will find it beneficial to use subroutines for various functions of your programs.

1.2 Calling Subroutines

You will recall, the program counter, or register PC (R15), always contains the address of the next instruction to be executed in a program. Calling a subroutine is similar to an unconditional branch as the program jumps to a different address other than the next address. The difference is that when the subroutine finishes executing, control goes back to the instruction after the subroutine call in the main program. A BL instruction causes the address of the next memory instruction to be pushed onto R14 (Link Register or LR) and the argument of BL to be loaded into the program counter. The argument of BL is the starting address of the subroutine. But, when you write your program, you just give it a name and the assembler figures out the rest. At the end of a subroutine, the instruction BX LR causes what was last stored in LR to be loaded into the program counter. In this way, the instruction after the BL in the main program is the next one executed. Therefore, it is important to recall that your subroutine should end with Branch and link with eXchance of the address stored in the Link Register, i.e., BX LR.

1.3 Coding with Subroutines

It is explained in Section 1.2 that a subroutine is called via BL instruction with the label of the subroutine as the operand. Thus, the starting address of the subroutine should be labeled. This can easily be done by using PROC and ENDP assembly directive. PROC indicates the starting of a procedure (subroutine) and ENDP designates the end of a procedure. The label for the line where PROC is placed actually is the label of the address of the first instruction of the subroutine. You may place your subroutines into the same source file where your main program exists. On the other hand, for more generic subroutines to be used in other projects, you may place your subroutines into separate source files so that different main programs can achieve them. In that case, you should tell the compiler that you are using subroutines from different source files in order to link them. This is done by using EXPORT and EXTERN/IMPORT assembler directives. EXPORT is used for the subroutines to make the label of the subroutine available to other source files. EXPORT directive should be followed by the name of the subroutine (e.g. EXPORT name_of_the_subroutine). EXPORT tells the compiler to remember the label to be possibly referred by other source files. EXTERN or IMPORT is used for the source files that are using labels (i.e. subroutines) that are defined in separate source files. EXTERN/IMPORT directive should be followed by the name of the subroutine that exists in one of the files added to the project (e.g. EXTERN name_of_the_subroutine). EXTERN/IMPORT tells the compiler to gather the address of the label to be referred by the current source file. An example of a subroutine coding can be:

```
;LABEL          DIRECTIVE       VALUE                   COMMENT

                               AREA     routines, CODE, READONLY
                               THUMB
                               EXPORT   Routine_name     ; make it available
                                                         ; to other sources
Routine_name   PROC
               ...             ...                       ; your routine
               BX              LR                        ; return
```

An example of a main program coding with subroutines placed in separate files can be:

```
;LABEL          DIRECTIVE       VALUE           COMMENT

               AREA            main, CODE, READONLY
               THUMB
               EXTERN          Routine_name     ; Reference external subroutine
                                                ; IMPORT can also be used
               EXPORT          __main
               ENTRY                            ; execution starts from here
__main         PROC
               ...             ...              ; your code
               BL              Routine_name     ; call to your subroutine
               ...             ...
               ENDP
```

If the subroutine and the main program are placed in the same source file, then EXPORT and EXTERN directives can be omitted.

1.4 Parameter Passing

A parameter is passed to the subroutine by leaving the data in a register, or memory, and allowing the subroutine to use it. A parameter is passed back to the main program by allowing the subroutine to change the data. This is the way parameters are passed in assembly language. When the parameter being passed to the subroutine is in a register, this is referred to as the call-by-value technique of passing parameters. If the data is in memory and the address is passed to the subroutine, this is called call-by-reference. It is important to document all parameter-passing details in subroutines.

Both stack usage and subroutine usage are crucial in order to write an efficient source code for your application. Generally, the memory of the microprocessor is limited and usage of stacks, as well as subroutines that are covered in the previous lab session, help reducing the source code length. In this lab, the experiments will be based on manipulating stack structure for parameter passing and using it to develop a program with higher hierarchy.

1.5 Stacks

Stack is an abstract data type that holds information in a LIFO (last-in-first-out) manner. For this very processor, the stack is actually just an area of memory whose highest address is in register R13 which is referred to as the stack pointer (SP). Make sure you understand the difference between the stack and the stack pointer. The stack is an area of memory; the stack pointer is the address of the last value pushed onto the stack.

Usually, the stack is used for storing data when subroutines are called. The stack is a last-in-first-out, i.e., LIFO structure so the last thing stored in the stack is the first thing retrieved. This resembles a pile of papers to be processed. The first paper put in the stack is the last paper to be processed. Whereas, the last paper put on the stack is the first paper to be processed. Another mechanical analogy that will demonstrate how the stack operates is the tissue dispensers used in bathrooms or public toilets. Figure 1-(a) illustrates a cross-section of a dispenser containing some tissues. As people use the restrooms, each

person takes one or more tissues from the dispenser after washing their hands. When a tissue is removed, a mechanism moves the tissue upward so that the next tissue is at the top of the dispenser. Figures 1-(b) and 1-(c) illustrate this process. Hence, the next person to wash hands can easily access the next tissue that is now placed on top. When the janitor refills the dispenser with a pile of new tissues, the mechanism goes downward so that the last tissue to be placed into the dispenser is at the top of the dispenser, as Figure 1-(d) illustrates. If how the dispenser operates is considered, it becomes obvious that the last tissue to be placed into the dispenser is the first to be used to dry hands, thus making the device a LIFO model. The stack operates exactly as this tissue dispenser, i.e, referred as a LIFO data structure.

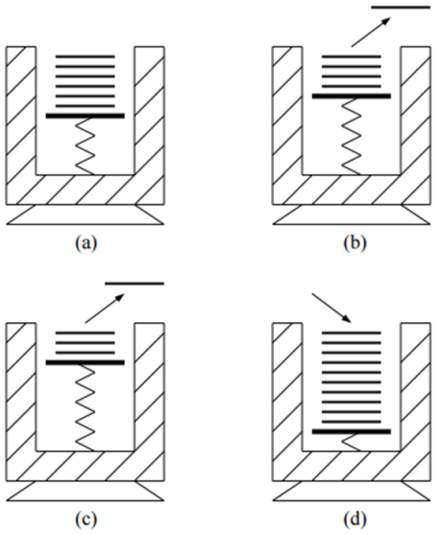

Figure 1: Tissue Dispenser Analogy

PUSH and POP instructions use to store and retrieve data to and from the stack, respectively. PUSH instruction stores 4-Byte register onto the stack and POP instruction stores the top 4-Byte data to the register. Recall that TM4C123GH6PM uses full descending stack. The push instruction first decrements the stack pointer by 4 and then stores 4 bytes of data, which is the content of the register, onto the stack. Consider what happens if register SP contains the address 0x2000.00FF and the register R0 contains the value 0x6B. Executing the instruction **PUSH R0** first decrements the stack pointer by 4, so register SP is now 0x2000.00FA and pushes the value 0x6B onto the stack at memory location 0x2000.00FA. The **POP R0** instruction first puts the contents of the top of stack into the R0 register and then increments the stack pointer by 4.

The main purpose you will be using the stack for is saving data when a subroutine is called. For example, assume your main program uses registers R0-R4. You call a subroutine that is going to calculate some value and pass it back to the main program. After you call the subroutine, you can just push all the data onto the stack before executing any instructions in the subroutine. The subroutine can then use all the registers for its internal use and store the data that the main program needs in one of the memory locations. At the end of the subroutine, you can pop the stored data from the stack and then return control to the main program. The following code is an illustrative example for that case:

```
;LABEL   DIRECTIVE       VALUE    COMMENT
;main program
        ...
        BL              my_subr
        ...
;subroutine part
my_subr PUSH            {R0}
        PUSH            {R1}
        PUSH            {R2}
        PUSH            {R3}
        ...                      ; subroutine instructions
        POP             {R3}
        POP             {R2}
        POP             {R1}
        POP             {R0}
        BX              LR
```

Calling a subroutine inside a subroutine, i.e. nested subroutines, can be an example to the aforementioned case as well. In that case, in order not to lose the return address information, LR should be preserved upon calling a subroutine inside a subroutine. Thus, the following code structure is a good practice if your subroutines call another subroutine:

```
;LABEL   DIRECTIVE       VALUE    COMMENT
;subroutine part
sub1    PUSH            {LR}
        ...                      ; subroutine instructions
        BL              sub2     ; another subroutine call
        ...                      ; subroutine instructions
        POP             {LR}
        BX              LR
```

Another case you will encounter may be calling a subroutine that is not written by you. In that case, to make sure that the registers you are using are not being modified, you preserve them by PUSH and POP instruction pairs before and after the call of the subroutine. Note that, some utility subroutines modify R5 register. Thus, you should take precaution if your main program makes use of R5 register. The following code is an illustrative example for that case:

```
;LABEL   DIRECTIVE       VALUE    COMMENT
;main program
        ...
        PUSH            {R0}
        PUSH            {R1}
        PUSH            {R2}
        PUSH            {R3}
        BL              my_subr
        ...                      ; some instructions to process
        ...                      ; registers that my_subr modifies
        POP             {R3}
        POP             {R2}
        POP             {R1}
        POP             {R0}
        ...                      ; rest of the main program
```

The important thing to know when using the stack is that pushes and pops have to be used in pairs because the stack is a LIFO data structure. If you push several registers, you have to pop them off the stack in the opposite order.

1.6 Concept of Recursion

In computer science, the term recursion refers to the method where the solution to a problem depends on solutions to smaller instances of the same problem, that is, in application a function is expected to call itself within the program text. If you are unfamiliar with the concept, please do some research for broader information and coded examples.

1.7 Pseudo-code Generation

A pseudo-code uses an expressive, clear, and concise method to describe an algorithm. It may have short English phrases, arrows (←) to indicate storing of data into variables, or it may look somewhat like a high level programming language without the details of the syntax. Since the pseudo-code focuses on the algorithm to solve the problem, instead of the syntax, it abstracts out the problem solving stage from the code writing stage with appropriate instructions and syntax. Below is an example of a problem statement and the pseudo-code developed for it.

Problem: Write an assembly language source code to be executed on an MCU with two registers A and B as follows: The program should read three unsigned numbers at memory addresses `0x0000`, `0x0001`, and `0x0002`, sort them from largest to smallest, and store them to addresses `0x0003`, `0x0004`, and `0x0005`. i.e. at the end of the execution, memory addresses `0x0003` and `0x0005` should contain the largest and smallest numbers respectively.

Pseudo-code: Note that this is just one algorithm to solve this problem. A and B stand for register A and register B for convenience, but they do not have to be in a pseudo-code.

```
A ← 1st Number (0000)
B ← 2nd Number (0001)
If A < B
       A → Smallest (0005)
       B → Largest  (0003)
Else
       B → Smallest
       A → Largest
A ← Largest     (0003)
B ← 3rd Number  (0002)
If A < B
       B → Largest
       A → Middle (0004)
       Done.
Else
       B → Middle
       A ← Smallest
       If A < B
             Done.
       Else
             A → Middle
             B → Smallest
       Done.
```

1.8 Flowcharts

A flowchart illustrates the steps in a process. It contains different symbols to indicate various tasks that the program has to do. There are many dedicated symbols for different types of tasks, but not all have to be utilized for the algorithm to be clear. In general the tasks in the regular flow are indicated by rectangular boxes, and a decision point to change the program flow (If-Then-Else, Repeat-Until) is indicated by diamonds. For example the algorithm described in the previous section can be depicted in a flowchart as below.

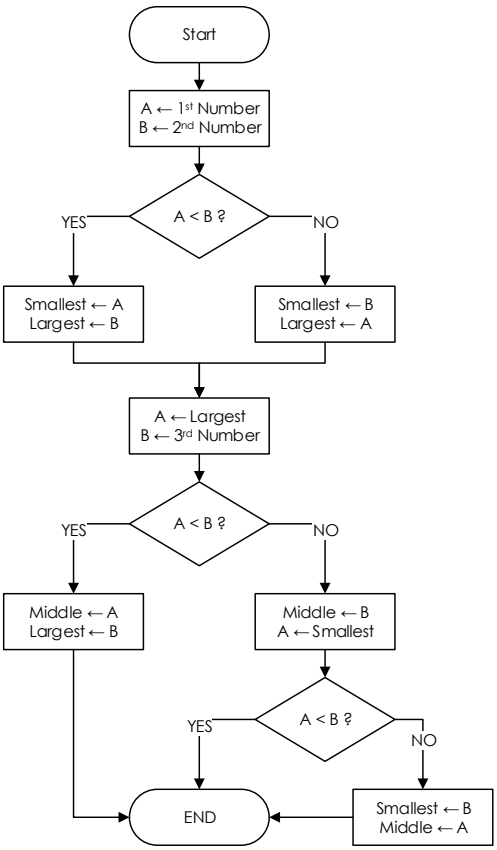

Figure 2: Example Flowchart for the Pseudo-code Given in 1.7

2 Preliminary Work

1. (18%) Write a subroutine, **CONVRT**, that converts an m-digit decimal number represented by n **bits** ($n < 32$) in register R4 into such a format that the ASCII codes of the digits of its decimal equivalent would listed in the memory starting from the location address of which is stored in register R5. When printed using OutStr, the printed number is to contain no leading 0s, that is, exactly m digits should be printed for an m-digit decimal number. Before writing the subroutine, the corresponding pseudo-code or flow chart is to be generated.

   Some exemplar printings *(righthand side)* for the corresponding register contents *(lefthand side)* are provided below:

   ```
   R4: 0x7FFFFFFF --- 2147483647 (max. value possible)
   R4: 0x0000000A --- 10
   R4: 0x00000000 --- 0
   ```

2. (7%) Write a program that, in an infinite loop, waits for a user prompt (any key to be pressed) and prints the decimal equivalent of the number stored in 4 bytes starting from the memory location **NUM**. Note that you may define **NUM** by using proper assembly directives. In this part, you are expected to use the subroutine you are written in Part-1. Explain which arguments should be passed and how.

3. (35%) Write a program for decimal number guessing using binary search method. The number is to be an integer in the range $(0, 2^n)$, i.e. $0 < number < 2^n$, where $n < 32$ and n is determined by a user-input. Then, the guessing phase is to be handled through a simple interface where the processor outputs its current guess in decimal base and calculate the next according to the user inputs, **D** standing for down, **U** standing for up, or **C** standing for correct. To fulfill the requirements given above, include the subroutine **CONVRT** from the Part-1 in your main program as well as a new subroutine **UPBND** that updates the search boundaries after each guess. Prior to writing the code itself, draw a flowchart of the main algorithm leaving the subroutine parts as black boxes.

4. (40%) Write a recursive program that computes first N elements of the mFibonacci Sequence - which is a modified version of Fibonacci Sequence. The number N is to be an integer in the range (0,16) and is determined by a user-input.Then, the computation phase starts and the program computes the first N terms of a modified Fibonacci Sequence. After the computation finishes the sequence of numbers is going to be printed.

   Fibonacci Sequence: The Fibonacci sequence is one of the most well-known series of numbers in mathematics that has found applications in advanced mathematics, computer science, and statistics etc.Each number in the sequence is the addition of the last two numbers, starting with 0, and 1. So, the sequence is: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, and so on. The mathematical expression describing it is;

   $$F_n = F_{n-1} + F_{n-2} \text{ where n}\geq 2.$$

   mFibonacci Sequence: Each number in the modified Fibonacci Sequence is computed by using the mathematical expression given below;

   $$F_n = 2 * F_{n-1} + F_{n-2} \text{ where n}\geq 2,\ F_0 = 0 \text{ and } F_1 = 1.$$

   So, the sequence is: 0, 1, 2, 5, 12, 29, 70, 169, 408, and so on.

   For this item, please explain how you solve the problem and draw a flowchart of your algorithm.

   Please remember that the input is to be provided via *Termite* in decimal base and your program will calculate all the terms one by one by calling the recursive function you typed and return the output via *Termite* in decimal base.