ECE 1770

ELECTRONIC MICROPROCESSOR SYSTEMS

Lab 4: Function and Subroutine

TASK#	Grading criteria	Instructor Initial
1	Program runs successfully (5)_____ The results displayed on LCD are correct (5)_____ Submit your complete assembly codes (5)_____	
2	Program runs successfully (5)_____ The results displayed on LCD are correct (5)_____ Submit your complete assembly codes (5)_____	
3	Program runs successfully (5)_____ The results displayed on LCD are correct (5)_____ Submit your complete assembly codes (5)_____	
4	Program runs successfully (5)_____ The results displayed on LCD are correct (5)_____ Submit your complete assembly codes (5)_____	
5	Program runs successfully (5)_____ The results displayed on LCD are correct (5)_____ Submit your complete assembly codes (5)_____	
6	Program runs successfully (5)_____ The results displayed on LCD are correct (10)_____ Submit your complete assembly codes (10)_____	

In order to receive credit for this laboratory exercise, please submit this handout to your lab instructor when you are finished.

Team Members: Group Number:	TA:

ECE1770

Lab 4: Function and Subroutine

Jingtong Hu
February 10, 2019

1. Objective

The basic objective of this lab is to become familiar with writing functions and calling them in your main code. Besides, this lab also illustrates the simple implementation of a random number generator and the fundamental bit level operations.

In this lab, you will be writing several functions for this laboratory exercise. You should have only one file and you should add the new function at the end of your older functions. Also, you will be adding items to the data section. And your main code will be changing. However, you should not delete earlier data items and codes. I suggest you to insert your new main code before the older one, so that the most recent main code will start immediately after the "__main PROC" statement. But you can also insert them after the older one and test your new codes immediately by commenting the older code.

Try not to delete any code from your file. And you should upload only a main.s file to the submission entry for lab 4. Rename the main.s file 'Group_X_lab4.s'. Here the X represents the number of the group you belong to.

2. Rules for Writing Function

Before you write your first function, you should observe some standard conventions. It is a matter of taste whether you write the data section first or the program section first. You cannot mix and match. In the examples of lab 3, the program section precedes the data section since it is easier to follow the code. However, it is also common that the data section is the first and you may find someone's codes written in this way.

The program section should start with the main code (i.e., the code you want to execute). The main code should be followed by various functions. The order of these functions is not important. What is a function? A function, also known as a subroutine is a self-contained code that implements a well-defined functionality. It should be self-contained in the sense that you should be able to copy the function from one file to another and be able to assemble it without any errors as long as you provide any other function this function may call. Each function should terminate with a return by using the instruction: ***BX LR***. It is a good programming practice not to have more than one ***BX*** instruction in any function.

Once you have written your function, a user can use the function by using the instruction: ***BL label*** as many times as needed. Here the label represents the address where the first instruction of the function is located in memory (technically known as the entry point). You can place the label before the very first instruction of the function. The assembler will automatically equate the label with the address of the first instruction. If the function needs any additional information (i.e. the parameters of function), they will have to be supplied by the user prior to using the function (technically known as binding). The function you will be writing will use some of the registers for binding. Also, many functions will return some useful value to the caller. In this case, it is a good idea to return the value in one of the registers. The detail about the registers is shown in the following figure.

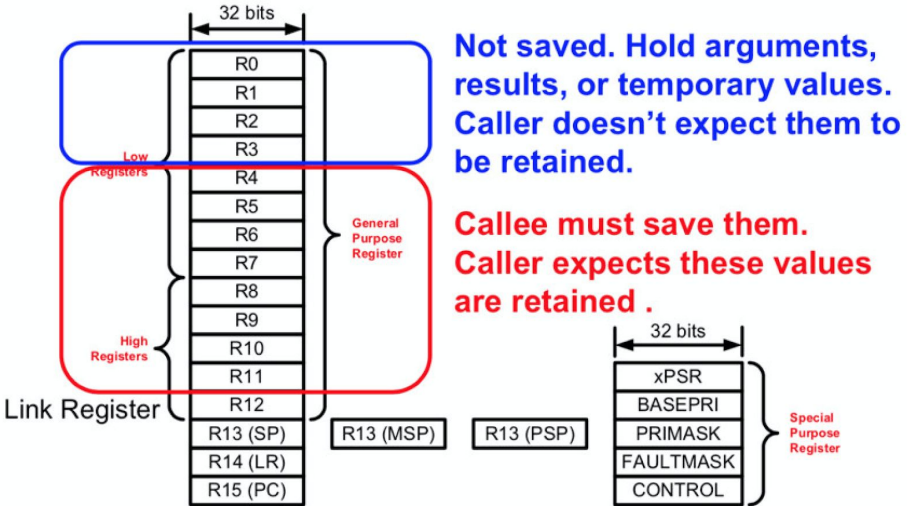

Figure 1. The detail about registers

Unless you write functions that do absolutely nothing (technically known as stubs), the function will use and modify one or more registers. If the caller wants to use the valuable data in a specific register such as R4 before and after the function call, there exists a potential problem: the value in R4 is possible to be modified by the function. If this happens, then you cannot get the correct value after the function calls. In order to avoid this problem, your function should save the values in the registers it uses by using the instruction ***PUSH {Rd}*** at the beginning of the function and then restore the values by using the instruction ***POP {Rd}*** before it returns. As the Figure 1 shows, if you want to use a valuable data after a function call, you need to save it in R4-R11 before the function call and then push and pop the register appropriately in the function. You had better not save it in R0-R3 because these registers are reserved for saving the return value of the function by default. I strongly recommend that you get into the habit of writing functions that clean up after themselves and restore the registers the way they were before the functions used them. Here are the basic rules for writing functions:

1. Decide on its functionality. Don't try to create a Swiss army knife that has multiple functionalities built in. Your function must do only one thing, and it must do it well. Your documentation for the function must clearly state the functionality
2. Decide on its name. Pick a meaningful name but keep the name to 8 characters or less. The more meaningful the name, the better it will be for when remembering its usage when you need it.
3. Decide on the registers that will be used to pass information to the function. If you do not expect to retain the value, you can save them on R1-R3. Otherwise, save them on R4-R11. The detail is shown in Figure 2.
4. Decide what registers will be used to return values back to the caller. 32-bit values can be returned using R0. 64-bit values can be returned using R0 and R1. The detail is shown in Figure 2.
5. Decide and clearly document what registers the function will use and which of these will be restored back at the end.
6. Write the function. Avoid the temptation of writing the function first and then worrying about the other items!

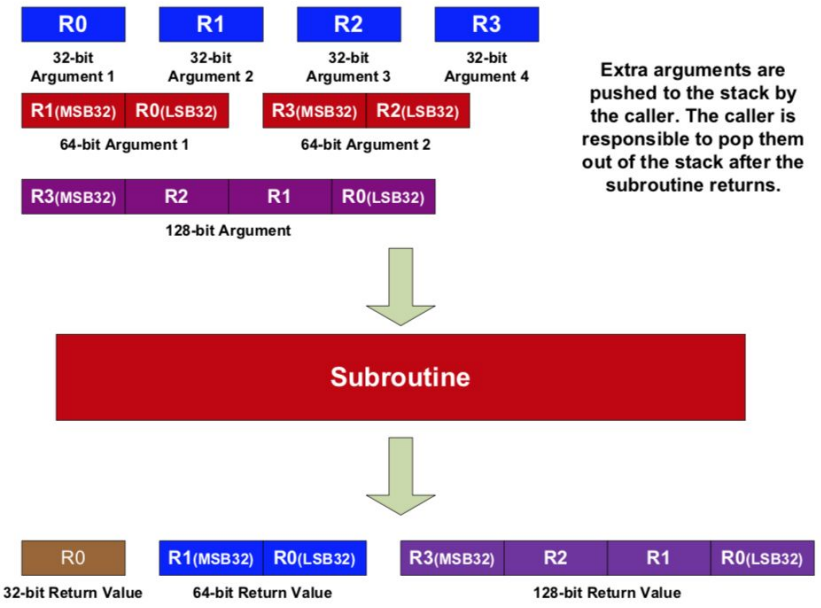

Figure 2. Passing arguments via registers

3. Calling and Writing Functions

As an example, we will start with a simple function called RAND that will return a random value every time it is called. Many random number generators (RNG) or pseudo-random number generators (PRNG) use similar techniques of shifting and adding to create a random number. In fact, most random number generators are found in common electronic casino games and video games. This RNG function remembers the last value it returned (this value can be initialized to a definite value such as zero, sometimes called a seed) and uses it to generate the next value. The

formula it uses is simple: It shifts the last value left, and adds 20 to the value. The function needs one word of storage to keep track of the random value. This storage will be allocated in the data section using the DCD directive, as follows:

```
LASTRND     DCD   0
```

The codes for the function RAND is as follows.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Function: RAND
; Purpose: Generate a random number
; Inputs: None
; Outputs: A random value returned in registers R0
; Registers modified: register R0, R1S
; Memory usage: The most recently generated random number is
; stored in memory with label LSTRAND
; This value is used to generate the next value
; Notes: Not the best random number generator around but does a
; halfway decent job.
; Implementation Notes :
; shift the last random value left and add 20
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
RAND   PROC
        LDR     R1, =LASTRND
        LDR R0, [R1]
        LSL R0, R0, #1
        ADD R0, R0, #20
        STR R0, [R1]
        BX LR
        ENDP
```

Tasks

Task 1: Random Number Generator

In this task, you need to type the above codes of function RAND first. Then in your main code, call this function repeatedly for 6 times. Compare the random value generated by this function with the corresponding elements in array ***Comparator*** every time it is returned by the function. By doing this, you can verify that the value in the R0 changes after each call. The detailed flowchart of this task is as follows.

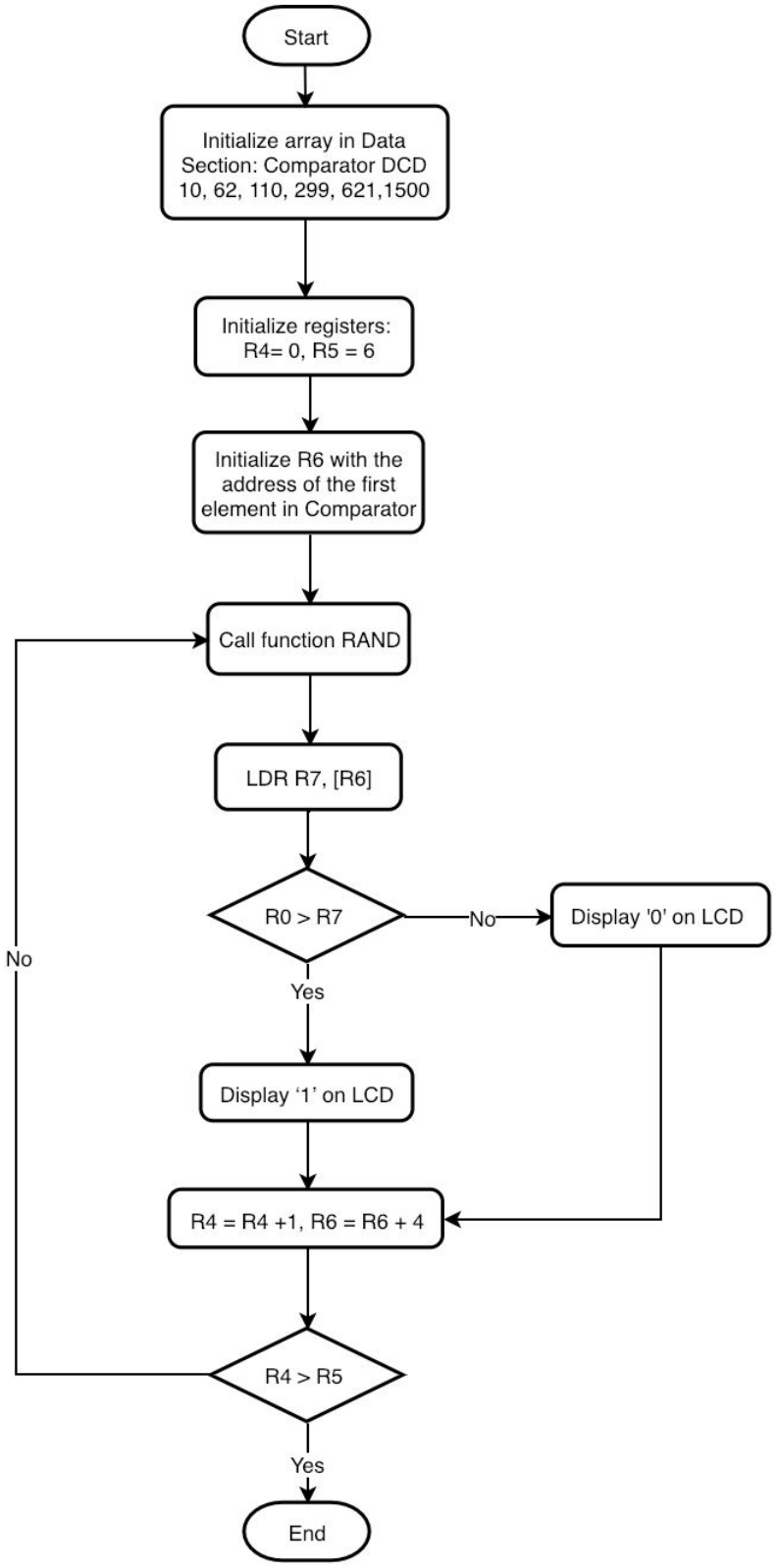

Figure 3. Flowchart of task 1

Task 2: Displaying Hexadecimal and Binary Numbers

In this task, your need to write two functions, and call them to display hexadecimal and binary numbers. The first function is to read a 2-digit hexadecimal number (**e.g. A2**) passed as the parameter and display it, and the second function is to read an 8-digit binary number (**e.g. 10100101**) passed as the parameter and display it.

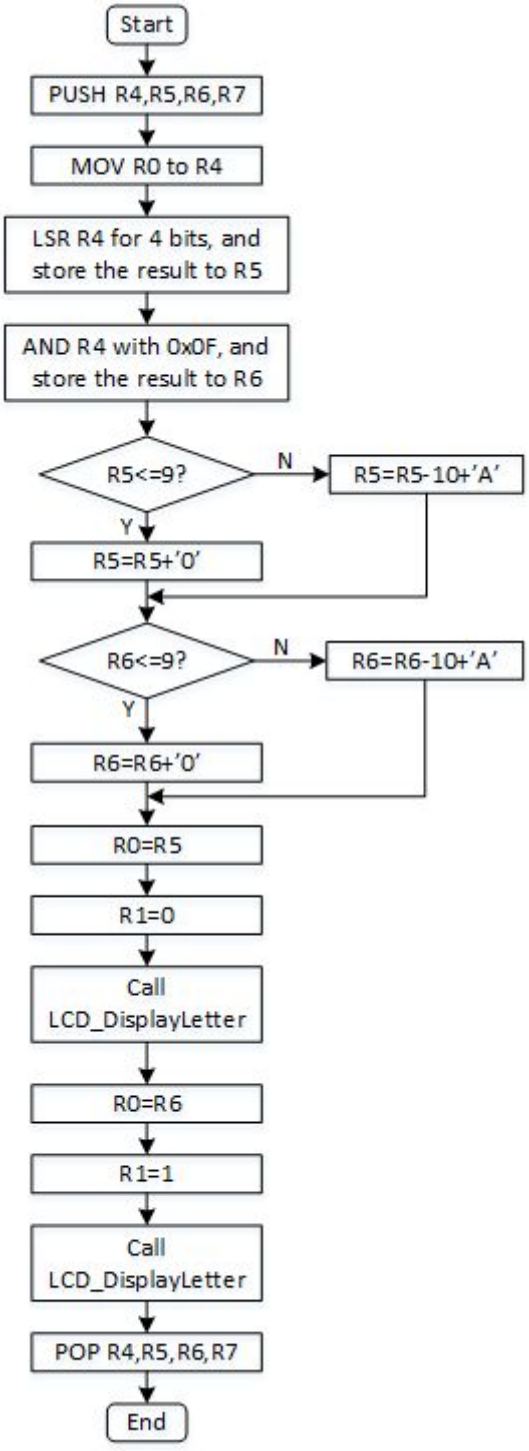

Figure 4. Flowchart of a function to display a 2-digit hexidecial number

Task 2.1

The function you write for this subtask is named **DISPLAYHEX**. To display the **hexadecimal** number **A2**, you get it from the parameter and display the two digits one by one. As shown in Figure 4, you first push R4-R7 to stack to preserve the context. Then you copy the number from R0 to R4 to further process each of the 2 digits. To get the leading digit, you need to SHIFT R4 right for 4 bits, and store the result to R5. To get the trailing digit, you need to AND R4 with 0x0F, and store the result to R6. To convert each of the two digits to ASCII for displaying, you need a branch because each digit ranges from 0-9 and A-F, and the two ranges need different conversion. For the range 0-9, the digit is added to ASCII '0'. For range A-F, the digit is first minused by decimal number 10 and then added to '0'. After getting the two digits, you call LCD_DisplayLetter two times to display each digit. Lastly, you POP R4-R7 from the stack to restore the context.

Task 2.2

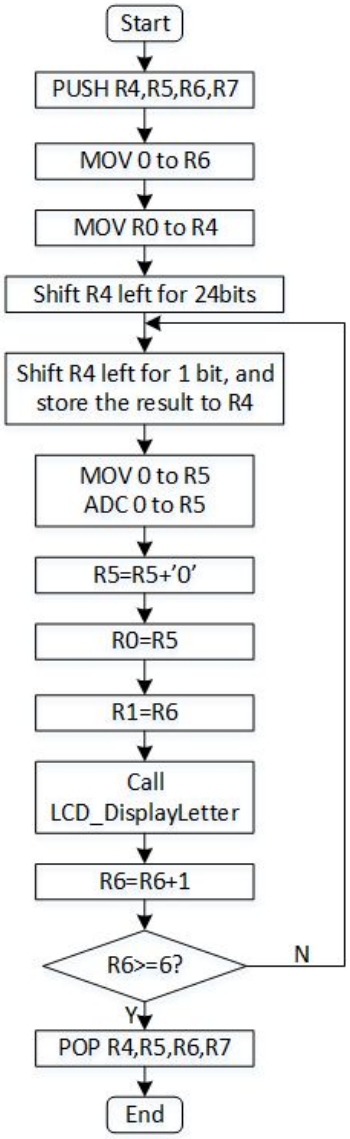

Figure 5. Flowchart of a function to display a 8-bit binary number

The function you write for this subtask is named **DISPLAYBIN**. To display the **binary** number **10100101**, you read it from the parameter and display all the 8 digits one by one. As shown in Figure 5, you get the 8 digits one by one and display them accordingly. Similar to displaying hexadecimal number, you first preserve the context. After that you copy the binary from R0 to R4, and shift 8 times to get each digit. For each digit, you shift R4 left for 1 bit to move the leading digit to Carry flag. To read the Carry, you need the ADC instruction to add 0 to Carry, and store the result to R5. Then you convert R5 to ASCII before calling LCD_DisplayLetter.

Task 3: Bit operation

Task 3.1

In this task, you need to write a function **BITSET** to read the parameter passed to it (e.g. **0x01**), set the bit 4 **ORR** it with 0x10, and return the result (**e.g. 0x11**) to the caller by R0. The caller then displays the result by calling the function **DISPLAYBIN** you designed in task 2.2.

Task 3.2

In this subtask, based on task 3.2, you write a function **BITCLEAR** to **clear** the bit 4 by **AND** the parameter with 0xEF. Similar to task 3.1, the caller first calls **BITCLEAR** and then calls DISPLAYBIN to display the result.

Task 3.3

In this subtask, based on task 3.2, you write a function **BITTOGGLE** to **toggle** the bit 4 by **EOR** the parameter with 0x10. Similar to task 3.1, the caller first calls **BITTOGGLE** and then calls DISPLAYBIN to display the result.

Task 3.4

In this subtask, your function **BITTEST** should be able to test if a specific bit (**e.g. bit 2**) in a hexadecimal number (**e.g. 0x04**) is 1. If the bit is 1, your function displays a **1** on LCD, otherwise your function displays a **0**.

To specify, the function first uses instruction **TST** to test if bit 2 of R0 (which is the parameter passed by the caller) is 1. If it is, your function calls LCD_DisplayLetter and passes two parameters R0 and R1 to represent the letter and position to display.

Task 4: Sum of Odd Numbers

In this task, you need to write a function to calculate the sum of 1+3+5+...+N, where N is a parameter passed to it, and return the result to the caller by R0. To specify, the callers passes **N=11** to your function by R0. You need to write a loop to

calculate the sum. You can use R4 to iterate through the sequence 1, 2, 3, 4, 5, …, N. **Note that even numbers are included in the iterations, but not added to the sum.** In each iteration, you first determine whether R4 is odd. If yes, you branch to a label to add R4 to the sum. Otherwise you branch to another label to skip this iteration and continue next iteration.

Task 5: Multiplication

In this task, you need to write a function **MULTIPLY**, which takes two 8-bit unsigned integers as inputs (**e.g. 0x07 and 0x08**). The two parameters are passed to your function by the caller, and should not be pre-defined in your function. Your function first read the two integers from R0 and R1, and calculates the product of the two numbers by the **32-bit instruction MUL**, and stores the lower 8 bits of the result to a global 8-bit integer variable ***RESULT***. For simplicity, the carry flag can be ignored. Besides, the function displays ***RESULT*** by calling **DISPLAYHEX** designed in task 2.1.

Task 6: Recursive Function

Task 6.1

In this task, you need to write a recursive function SUM(n) to calculate the sum of consecutive numbers: (1 +2 + 3 + …… + n). By using this function, you need to calculate (1+2+3+ …… + 10) in your main codes and display the result in the binary form on LCD. You can call the function you implement in ***task 2.2*** to help you display your results.

Task 6.2

In this task, you need to write a recursive function SUMODD(n) to calculate the sum of consecutive odd numbers: (1 + 3 + 5 + …… + n). n is a odd number here. By using this function, you need to calculate (1+3 +5 + …… + 11) in your main codes and display the result in the binary form on LCD. You can call the function you implement in ***task 2.2*** to help you display your results.