ECE 1770

ELECTRONIC MICROPROCESSOR SYSTEMS

LAB3: STRUCTURED PROGRAMMING

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:	TA Observations:

ECE1770

Lab3: Structured Programming

Jingtong Hu
January 23, 2019

1. Objective

From this lab, you will start to write your first assembly program. For structured programming, the most important thing is the control flow. The sequential execution of a program can be changed by branch. And the loop is a typical kind of structure using the branch. The goal of this lab is to become familiar with elementary loops. Besides, you will know how to display the results of your program on LCD.

2. Display Results on LCD

Reminder: **Download the template project from Courseweb**, and edit the main.s for your assignments. Please do not edit any other files for this lab, which may affect your task results.

To display a digit on the LCD, there are 4 four steps. To display a number with several digits, please display each digit in different positions, and you will get the whole number displayed.

1. Initialize LCD: this is done by calling LCD initialization at the beginning of the main program. Note that it only conducted once before a loop even if you want to display multiple digits through a loop.

```
BL  LCD_Initialization
BL  LCD_Clear
```

2. Set display value and position: the display value and position are set by passing arguments to display function. Before calling display function, the display value is stored in R0, while the display position is stored in R1. Note that the value to display should be converted to its ASCII value. For instance, if the value to display is 0x01, it should be converted to 0x31. The position is in range 0-5 because there are 6 digits on the LCD.

3. Call display function: this is conducted by calling the display function after setting the display value and position in step 2.

```
BL LCD_DisplayLetter
```

4. Delay for observation: a delay is inserted after display function for you to observe the display result on LCD. Otherwise the LCD will be updated to fast and you cannot distinguish what is displayed. The delay time is set in R0 before calling delay function. The following example shows steps to delay about 1 second.

```
MOV R0, #1
BL LCD_DisplayDelay
```

3. Branching

Conditional branching in the assembly language is controlled by the state of one or more hardware flags. The state of a flag depends on the most recently executed instruction that affects the flag. Therefore, if your branching depends on the result of some instruction, then it is your responsibility to make sure that none of the instructions between the instruction you are interested in and the branching instruction affects the flags that control the branching instruction. Thus, it is a good idea to follow the instruction that sets the flag by the branching instruction.

In this lab, you might use the common important instructions that are often utilized to set/clear the flags.

Instruction	**Description**	**Affected Flags**
CMP Rn, Operand2	Rn - Operand2. The result will not be written to Rn, but set the flags.	N,Z,C,V
CMN Rn, Operand2	Rn + Operand2. The result will not be written to Rn, but set the flags.	N,Z,C,V
TST Rn, Operand2	Rn & Operand2. The result will not be written to Rn, but set the flags.	N,Z,C
TEQ Rn, Operand2	Rn ^ Operand2. The result will not be written to Rn, but set the flags.	N,Z,C

If the result of the instruction mentioned above is zero, then the Z flag will be set. And you can use the following conditional branch instructions to control the workflow of the program by checking the state of Z flag. In this lab, the following two instructions are enough. You can also use other combination of branch instruction B and various conditional codes.

BEQ label	It will cause a branch to label if the Z flag is set. Otherwise, the

	program will continue to execute the next instruction.
BNE label	It will cause a branch to label if the Z flag is not set. Otherwise, the program will continue to execute the next instruction.

Here, the label is the memory address of the instruction which the program will jump to when the condition is satisfied.

4. Counting and Employing Loops

This section first introduces loop, and then presents example code to show the loop results on LCD. After that, **tasks in 4.2-4.4 are provided for you to complete.**

4.1 Introduction

Utilizing loops in programs is important for most applications, because most systems want to repeat a computation a certain number of times. Therefore, it is imperative that most programming languages have a method for incrementing and decrementing a loop counter. After each iteration, the loop typically augments a counter and checks whether it has reached a certain value. This counter usually controls how many times the loop is executed. A counter can be stored in memory or kept in a register. If it is kept in a register, it is important for any programmer to make sure that the register is not inadvertently changed either by your code or by some third party function that you call.

In assembly language, we usually use a register to maintain a value as a counter. The value will first be initialized to the number of times the loop is to be performed. After each time the loop executed, the value needs to be decreased. The structure of the loop is as follows:

1. Initialize the counter to the number of times the loop is to be performed.
2. Perform any other initializations.
3. Test if the counter is zero. If so quit the loop. (while do/ for)
4. Perform the desired task.
5. Perform any re-initializations if needed.
6. Decrement the counter.
7. Go back to 3.
8. Come here when you quit the loop. The program will execute the next instruction.

It is important to note that there are two places where the code jumps to. One to step 3 and the other to step 8. Section 4.2 will give you an example to implement a simple loop using a counter.

4.2 Example: Looping to update register periodically. This part gives an example to use instruction **BNE** to implement a loop. The number of iterations is pre-defined, and the current iteration is displayed on LCD. The example code will be illustrated, and two following-up questions will be provided.

The example code is as follows:

```
__main        PROC
        BL  LCD_Initialization
        BL  LCD_Clear

        ; r4 is the register to be increased by 1 for each loop iteration
        LDR R4, =0x00
        ; r5 is the number of iterations to loop
        LDR R5, =0x05
loop
        ; Add r4 by 1
        ADD R4, #0x01

        ; display R4 @ LCD position 0
        MOV R0, R4

        ; convert to ASCII code for display by adding 0x30
        ADD R0, #0x30
        ; set display position by setting R1
        MOV R1, #0x00
        BL LCD_DisplayLetter

        ; delay 1 second for observation
        MOV R0, #1
        ; call display function
        BL LCD_DisplayDelay

        ; compare r4 with r5
        CMP R4, R5
        ; if they are not equal, jump to loop
        ; else continue to next line of code
        BNE loop
        ; dead loop & program hangs here
stop    B               stop
```

The main function serves as the entry of the project. The LCD is first initialized by calling *LCD_Initialization*, and cleared by *LCD_Clear*. Two variables, R4 and R5 are used to control the loop. R4 stores the current iteration, and R5 stores the number of desired iterations. In each iteration, R4 is increased by 1, and compared with R5. If R4 is not equal to R5 (which implies R4 < R5), we jump back to loop by BNE, otherwise we just out of the loop by continuing execution next instruction following BNE. In each loop iteration, the iteration number is displayed on LCD by calling LCD_DisplayLetter. After that, a delay function is called for observation purpose. If it is commented out, the several iterations will be finished really fast and the result of each iteration cannot be observed.

Based on the provided code, there are two assignments:

4.2.1 Task 1: BEQ works differently as BNE. In our example, if we have to use BEQ to control the loop iterations, certain codes need to be changed to work. Please modify the code so that the loop iterations are controlled with BEQ instruction instead of BNE.

4.2.2 Task 2: Display capital letters A-Z one by one. To achieve this, you need to loop 26 iterations and display one letter for each iteration. Note that the display function *LCD_DisplayLetter* only accepts ASCII code as input, so you need to convert the letter into ASCII before displaying.

4.3 Using Memory Effectively

When you want to load the content in a memory, you need to get the memory address of the content. Then you can use the memory address to get the corresponding content in this memory. When you want to store a value in the register to the memory, you need to provide the memory address where you want to save the value first and then store the value to this memory address.

Here is a simple example to get the content in variable SUM and store a value to SUM.

```
LDR r1, =SUM    ; Get the memory address of variable SUM. Save it in r1.
LDR r2, [r1]    ; Get the content of variable SUM
ADD r2, r1, #1  ; Add one to the value in r2
STR r2, [r1]    ; Store the updated value in r2 to SUM
```

Often loops are combined with memory exploration and altering or reading consecutive memory locations (e.g. an array). In this case, you will use a register Rn to reserve the starting memory address of these consecutive memory locations. And after each time the loop is executed, you need to increase the Rn to get the next memory address.

4.3.1 Task 3: Define an array of six integers (each integer is 32 bit) in the DATA section using the DCD directive. Then implement a loop to display this array on LCD. For the convenience of the demo, the initial values of the array should be [1, 2, 6, 8, 9, 5].

4.3.2 Task 4: Modify the assembly codes you write in Task 3, so the values displayed on LCD are in the reverse order of the array, i.e., [5, 9, 8, 6, 2, 1].

4.3.3 Task 5: Define an array *Array1* of size six in the DATA section using DCB directive in the DATA section. For the convenience of the demo, the initial values of *Array1* should be [‘a’, ’b’, ‘c’, ‘d’, ‘e’, ‘f’]. Define another array *Array2* of size six in the DATA section using DCB directive too. The initial value of *Array2* does not matter in this task. You need to implement a loop which adds ten to each element in *Array1* and then store the addition results to *Array2*. After these steps, you can reuse the codes you implement in Task 3 to display the values in *Array2* on LCD with just a little change of the codes.

4.4 Task 6: Displaying a number

In this task, you program should be able to display any given unsigned decimal integer number (<= 6 digits) on LCD. e.g. 123456. You DO NOT need to remove the leading 0s for a number with less than 6 digits. For example, the number 123 can be displayed as 000123.

The number is stored in a variable *NUM*, which is set to *123456 initially* in your program. The number is displayed by **displaying all the digits on LCD one by one**. For example, if the value in *NUM* is a decimal number 123456, to display this value, the number 123456 is divided into six digits 1, 2, 3, 4, 5 and 6, and the six digits are converted into ASCII format, which is 0x31, 0x32, 0x33, 0x34, 0x35 and 0x36. After that, LCD_DisplayLetter function is called six times to display the digits at position 0, 1, 2, 3, 4 and 5, respectively.

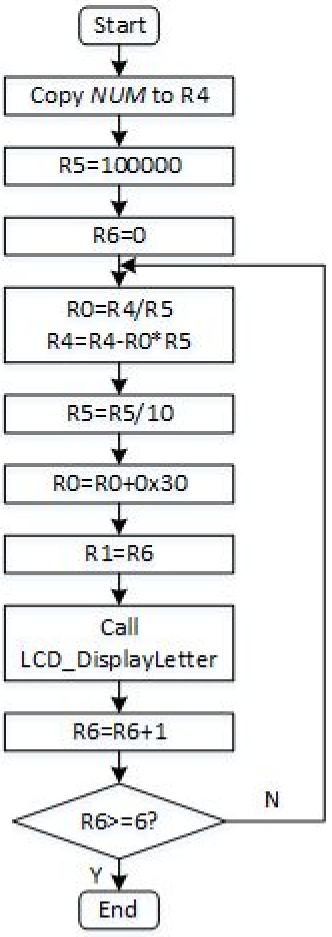

Figure 1. Flowchart of displaying a 6-digit number

The flowchart for displaying a number is shown in Figure 1. We first copy the number to display from the variable *NUM* to a register, and then use a loop to display each digit. R4 holds the number to display, and R5 holds the divisor to get each digit of the number. R6 controls the loop by counting from 0 to 5, and also sets the display position for each digit. In the loop, the MSB of R4 is stored into R0, and R4 is updated by removing the MSB from it. The divisor R5 is also updated by removing the least significant bit 0. Now R0 holds the digit to display, and R1 holds the display position. Since the LCD_DisplayLetter function reads the digit in ASCII format, the digit in R0 is converted into ASCII format by adding 0x30 to it. Also, the display position is copied to R1, and the display function is called. After displaying one digit, if R6 is greater than or equal to 6, which means we have finished displaying all the 6 digits, we jump out of the loop. Otherwise the loop continues to display the rest of the digits.

5. Task: Bubble Sort (Bonus Points)

This task is not mandatory and will not count in your grades for lab 3. You will get an extra 3 points added to your final grades. The deadline for this task is the same as lab 4. So you have about one month to finish this task.

5.1 Introduction

Bubble Sort is the simplest sorting algorithm based on the comparison. It repeatedly steps through the array, compares adjacent pairs and swaps them if they are in the wrong order. The pass through the array is repeated until the array is sorted. And the pseudocode of this algorithm is as follows.[1]

```
procedure BubbleSort (A: array of sortable elements. The index of A is from 0.)
    n = length(A)
    j = 0
      repeat
      swapped = false
      for i = 1 to n-1-j inclusive do
              /* if this pair is out of order */
              if A[i-1] > A[i] then
            /* swap them and remember something changed */
            swap( A[i-1], A[i] )
            swapped = true
              end if
      end for
              j = j +1
      until not swapped
end procedure
```

The following simple example illustrates how bubble sort works.[2]

Initialization:

The input of this algorithm in this example is an array out of order: (5 1 4 2 8). And we want to get an array in ascending order.

First Pass:

The pass process will start with the first number in the array. And it will ends when the **last element** of the array has been compared.

(5 1 4 2 8) –> (1 5 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.

(1 5 4 2 8) –> (1 4 5 2 8), Swap since 5 > 4
(1 4 5 2 8) –> (1 4 2 5 8), Swap since 5 > 2
(1 4 2 5 8) –> (1 4 2 5 8), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:
The pass process will start with the first number in the array. And it will ends when the element, which is **second to last**, has been compared.

(1 4 2 5 8) –> (1 4 2 5 8)
(1 4 2 5 8) –> (1 2 4 5 8), Swap since 4 > 2
(1 2 4 5 8) –> (1 2 4 5 8)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

Third Pass:
The pass process will start with the first number in the array. And it will ends when the element, which is **third to last**, has been compared.

(1 2 4 5 8) –> (1 2 4 5 8)
(1 2 4 5 8) –> (1 2 4 5 8)

In this pass, there is actually no swapping happened. So the algorithm will know that the array has been in ascending order. So we get the correct output (1 2 4 5 8).

5.2 Assembly Implementation (Task 7)

The two versions of implementation [2] of the Bubble Sort in C language has been provided as follow:

Version 1:

```
int main()
{
        SIZE = 8;
   int a[SIZE] = {95, 45, 15, 78, 84, 51, 24, 12};
        int i, j, temp;
        int swap;
        do{
        swap = 0;
        for (i = 0; i < n - 1 - j; i++)
        {
                if(a[i] > a[i + 1])
                {
                        temp = a[i];
                        a[i] = a[i + 1];
                        a[i + 1] = temp;
                        swap = 1;
                }
        }
        }while (swap == 1)
        for (i = 0; i < SIZE; i++)
        {
        printf("%d\n", number[i]);
        }
        return 0;
}
```

Version 2:

```
int main()
{
        SIZE = 8;
   int a[SIZE] = {95, 45, 15, 78, 84, 51, 24, 12};
        int i, j, temp;
        int swap;
        for( j = 0; j < n-1 ; j ++)
        {
        for (i = 0; i < n - 1 - j; i++)
        {
              if(a[i] > a[i + 1])
                {
                        temp = a[i];
                        a[i] = a[i + 1];
                        a[i + 1] = temp;
                }
        }
        }
        for (i = 0; i < SIZE; i++)
        {
```

```
        printf("%d\n", number[i]);
        }
        return 0;
}
```

According to the above implementation, write your own version of bubble sort in assembly language. Version 1 is more efficient than version 2, but you can choose the version you prefer. The input array of this program (the array out of order) can be initialized in the DATA section. For the convenience of the demo, the initial value of input array should be [4,9,3,1,6,3]. However, your program should be able to handle any given array. The output (the array in ascending order) should be displayed on LED (In the C code, it uses the printf to output the array to the screen). Since the LED can only display at most six characters, if you want to run your program using other test cases , it is recommended that the length of the input array is not greater than six. Please submit the source code (main.s) to Courseweb.

6. Reference

[1]. https://en.wikipedia.org/wiki/Bubble_sort
[2]. https://www.geeksforgeeks.org/bubble-sort/