Khalifa University of Science, Technology and Research
Electronic Engineering Department

Microprocessor Systems Laboratory

ELCE333

Laboratory Report Experiment No.2

MICROCONTROLLER ASSEMBLY PROGRAM DEVELEPMENT

Lab Partners

Leena ElNeel	100037083
Muna Darweesh	100035522
Shamsah AlNabooda	100036984

Date Experiment Performed: 4/2/2015
Date Lab Report Submitted: 11/2/2015

Lab Instructors:

Mahmoud Khonji

Mohammed Ali Saif Al Zaabi

Spring 2015

Table of Contents

List of Figures and Tables

List of Figures

List of Tables

Summary

This report discusses instruction codes and particularly focuses on branching and loops. This experiment is made up of four tasks. The first requires implementing BEQ instruction, the second requires using BNE instruction. The third task deals with nested If-Else statements and the last task requires writing a code with simple loops.

1. Introduction

Branch instructions in assembly languages provide the ability to have 2 paths to single statement. Such is in the flowchart shown in Figure 1 below. The concept of the branch is setting a condition and if true, the program runs in the direction that satisfies the condition.

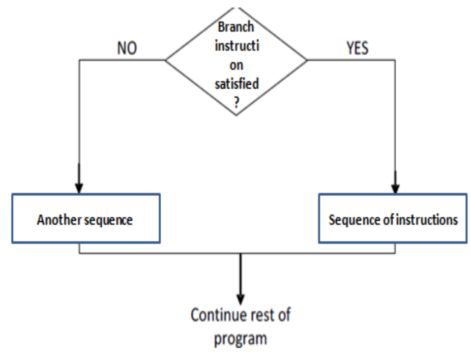


Figure 1: Flow chart of an example on simple Branch instruction

Branch instructions are also used when we encounter a stack. We initialize a pointer and keep incrementing it so that it points to the following elements in the stack which forms a loop. Figure 2 below shows an example of a loop where we use a branch instruction.

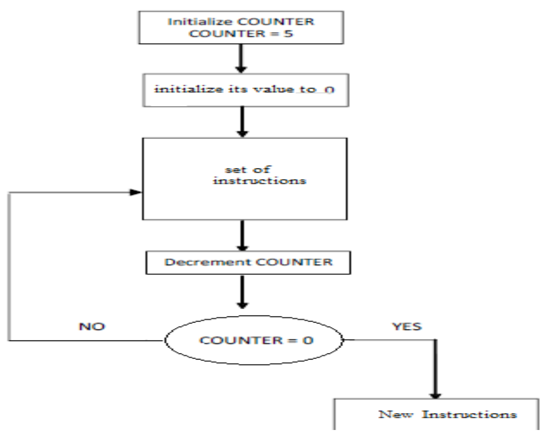


Figure 2: Example of branching code with loop

Aim:

The aims are to gain experience in the design of HCS12 assembly programs with conditional branching structures and loops.

Objectives:

- Design flow-charts containing branching and loops and Implement it using flow-charts using HCS12 assembly code Also, use Bcc instruction to implement branching and loops.

2. Design and Results

TASK 1: If-Then-Else Statement using BEQ

In the first task, we are asked to write an assembly code that checks a memory location ($1000). If the location is zero, we assign the content of $1001 to $1000. However, if memory location does not equal zero, we assign the content of $1002 to it. The code we wrote is shown below:

```
X1 EQU $1000
Y1 EQU $1001
Y2 EQU $1002
      ORG $4000
Entry:
      CLRA
      MOVB #$0,X1
      MOVB #$A1,Y1
      CMPA X1
      BEQ Eqzero
      MOVB Y1,X1
      BRA Exit
Eqzero:MOVB Y2,X1
Exit:  BRA Exit
```

To accomplish this task, we have to clear register A and compare it with X1 (memory location $1000). We then use the BEQ instruction to branch to label Eqzero where the content of Y2 is moved to X1. If the code doesn't branch, we move the content of Y1 to X1 and then exit the program. We tested the code for two conditions, 0 and 3. Table1 displays the changes after running the program, and the two figures compare the two case. Figures 3 and 4 show the attained result for test 1 and 2 respectively.

Table 1- Testing results for Task1

	Before Program Start			After Program End		
	$1000	$1001	$1002	$1000	$1001	$1002
Test 1	0	A1	A2	A2	A1	A2
Test 2	3	A1	A2	A1	A1	A2

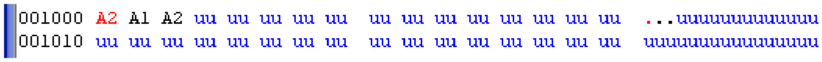


Figure 3: Memory window for test 1

\

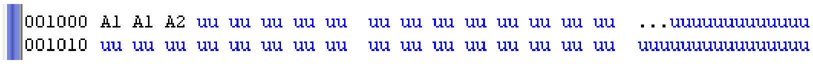


Figure 4: Memory window for test 2

Task2: If-Then-Else Statement using BNE

In task 2, we are asked to modify the code we wrote in task one but we have to use the BNE instruction which is a branch instruction but implies “not equal”. Simply, we had to reverse the code in order to keep the flowchart in the correct sequence. Here, if the content of X1 is not equal to 0, we move content of Y1 to it. On the other hand, if it is equal to 0, we move the content of Y2 to it. We have created a flowchart to help us modify the code and the flow chart is shown below. Using the flowchart, we noted down the expected theoretical results and then we jotted down the calculated results and they were the same. Table 2 in the following page shows the obtained results.

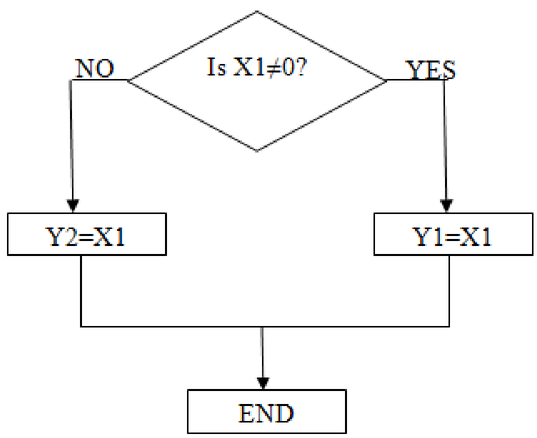


Figure 5: Flow chart of task 2

```
X1 EQU $1000
Y1 EQU $1001
Y2 EQU $1002
    ORG $4000
Entry:
    CLRA
    MOVB #$0,X1
    MOVB #$A1,Y1
    MOVB #$A2,Y2
    CMPA X1
    BNE Eqzero
    MOVB Y1,X1
    BRA Exit

Eqzero:MOVB Y2,X1
Exit:  BRA Exit
```

Table 2- Testing results for Task2

	Before Program Start			After Program End		
	$1000	$1001	$1002	$1000	$1001	$1002
Test 1	0	A1	A2	A1	A1	A2
Test 2	3	A1	A2	A2	A1	A2

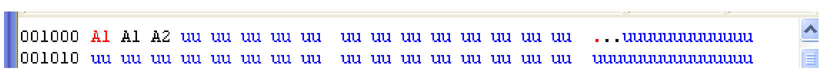


Figure 6: Memory window for test 1

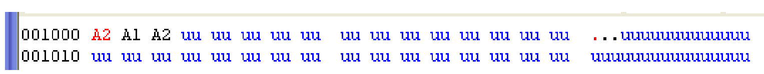


Figure 7: Memory window for test 2

Task3: Nested If-Then-Else Statement

In this task we were asked to write a program that performs the algorithm shown in the flowchart below which compare between two numbers. The code will check if X1 is greater than $5 if so, the content of Y1 will be moved to X2, otherwise, the code compare this value with -$5. If the value is greater than -$5 the contents of Y2 will be moved to X2 otherwise the contents of Y3 will be in X2.

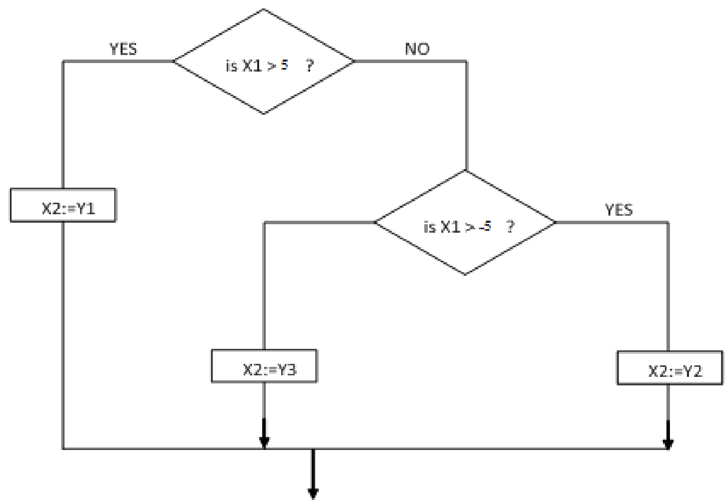


Figure 8: Flowchart of the algorithm

The code is as the following. It shows the result in the case where X1 is greater than $5, for $6.

```
X1 EQU $1000
X2 EQU $1001
Y1 EQU $1002
Y2 EQU $1003
Y3 EQU $1004
        ORG $4000
Entry:

      CLRA                      ; Clear A
      LDAA #$5                  ; Load 5 in A
      MOVB #$6,X1               ; move $6 to X1
      MOVB #$A1,X2              ; move A1 to X2
      MOVB #$A2,Y1              ; move A2 to Y1
      MOVB #$A3,Y2              ; move A3 to Y2
      MOVB #$A4,Y3              ; move A4 to Y4
      CMPA X1                   ; compare A with X1
      BLE  greater              ; if X1 is greater go to greater
      NEGA                      ; negate A
      CMPA X1                   ; compare A with X1
      BLE  greater2             ; if X1 greater than -5 go to greater2
      MOVB Y3,X2                ; if X1 is less than -5 move Y3 to X2
       BRA  Exit                ; Branch to Exit
greater:  MOVB Y1,X2
greater2 MOVB Y2,X2
Exit     BRA Exit
greater  MOVB Y1,X2
greater2 MOVB Y2,X2
```

In the is code, the compare instruction (CMPA) compare A with the content of X1, by using the BLE instruction, it checks if the contents of X1 is less than A then it branches to *greater* section.

The table below shows the results for different cases. In the first case, \$6 >\$5 so the content of Y1 at memory location \$1002 which is A2 moved to X2 at memory location \$1001. In the second case, for X1 < 5 but not less than -5, the content of Y2 at memory location \$1003 which is A3 moved to X2 at memory location \$1001. Finally, when the value of X1 is less than \$5 and -\$5, in this case -\$10 or F6, the content of Y3 at memory location \$1004 which is A4 moved to X2 at memory location \$1001.

Table 3: Cases for task 3

Case 1: X1> \$5, \$6	**Case 2: X1< \$5, \$2**	**Case 3: X1< \$5, and < −\$5, -\$10**
001000 06 A2 A2 A3 A4 uu uu uu 001010 uu uu uu uu uu uu uu uu 001020 uu uu uu uu uu uu uu uu 001030 uu uu uu uu uu uu uu uu 001040 uu uu uu uu uu uu uu uu 001050 uu uu uu uu uu uu uu uu	001000 02 A3 A2 A3 A4 uu uu uu 001010 uu uu uu uu uu uu uu uu 001020 uu uu uu uu uu uu uu uu 001030 uu uu uu uu uu uu uu uu 001040 uu uu uu uu uu uu uu uu 001050 uu uu uu uu uu uu uu uu	001000 F6 A4 A2 A3 A4 uu uu uu 001010 uu uu uu uu uu uu uu uu 001020 uu uu uu uu uu uu uu uu 001030 uu uu uu uu uu uu uu uu 001040 uu uu uu uu uu uu uu uu 001050 uu uu uu uu uu uu uu uu

Task 4: Simple program with loops

In the last task of this lab we used several assembly language instructions to write a code that will sum 5 hexadecimal values that is stored in memory. The following flow chart shows the steps of the code:

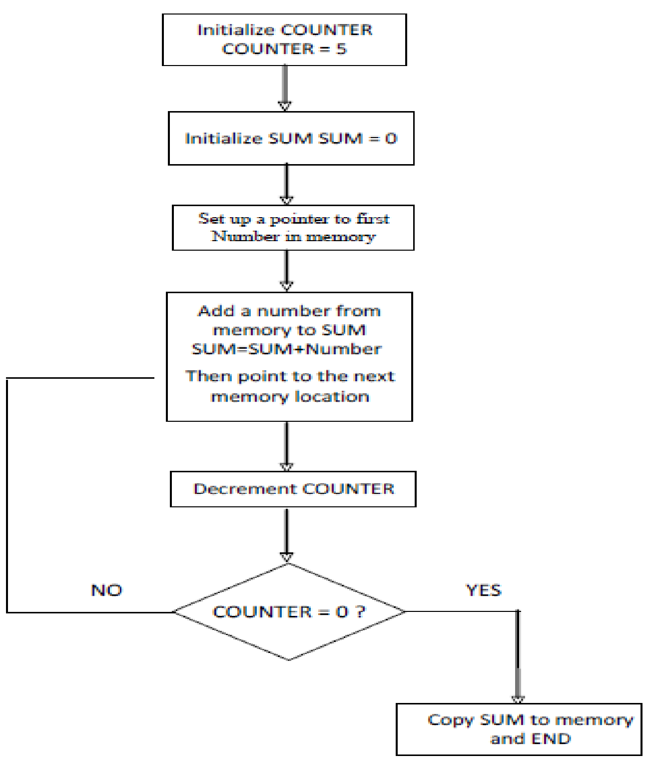


Figure 9: Flowchart task4

In order to define the 5 values in memory we used define constant directive DC.B to allocate memory locations and assigns values to them. Since the implementation of this algorithm is similar to summing cumulatively the values stored in an array a pointer was used to point to the first element and store its address in register X. Also, since the loop should run for a number of iterations and decision should be made at each one whether to exit loop or continue for iteration, accumulator A was used as a counter that is set at beginning at #$5 and been decremented and used for decision making at each iteration. Moreover, for the cumulative summation of the five values inside the loop section code register Y was set to zero initially and its final content was stored in *sum* memory location. *Sum* and *counter* were modified at each iteration of the loop so choosing a registers to represents them in the loop code is the right choice to reduce the access time and the faster availability of data.

```
counter  EQU $1002
sum      EQU $1003
NUMBERS DC.B $12,$1A,$43,$15,$28
ORG $4000 ;Flash ROM address for Dragon12+
Entry:
CLRA          ; clear acc.A
CLRB          ; clear acc. B
LDY #$0000    ; set reg. Y to zero
LDAA #$5      ; set acc. A to $5 to be used as counter
LDX #NUMBERS  ; store address of first reg. pointed to it by #NUMBER

LOOP:   LDAB 1,X+     ;load acc. B with next reg.
ABY           ; B+Y -->Y
DECA          ; decrement acc. A by one (A--)
CMPA  #$0     ; check if A=0
BEQ  end      ; if A=0, then goto end
BRA LOOP      ; goto LOOP
end:    STY sum       ; Y --> sum
BRA Exit      ; Exit code
Exit:   BRA Exit      ; End your code here
```

From calculations the sum of the set of values given should be:

$$Sum\ (\$1003) = 12_{16} + 1A_{16} + 43_{16} + 15_{16} + 28_{16} = AC_{16}$$

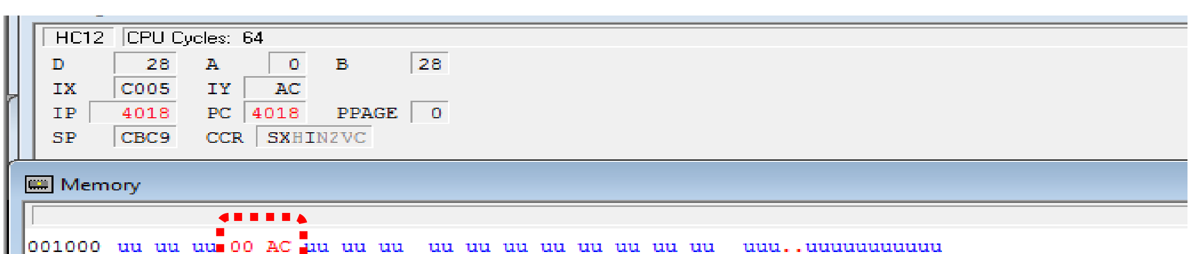


Figure 10: Simple program of loops simulation results

After simulating the code we got the expected result however it was in 16 bits because it is the size of reg. Y that we used as a temporary place to store the results of the cumulative summation process. We also got the correct number of iterations (5). Also as we can notice from figure 9 above that the code stopped and the final values appeared when the acc. A (counter) reached zero.

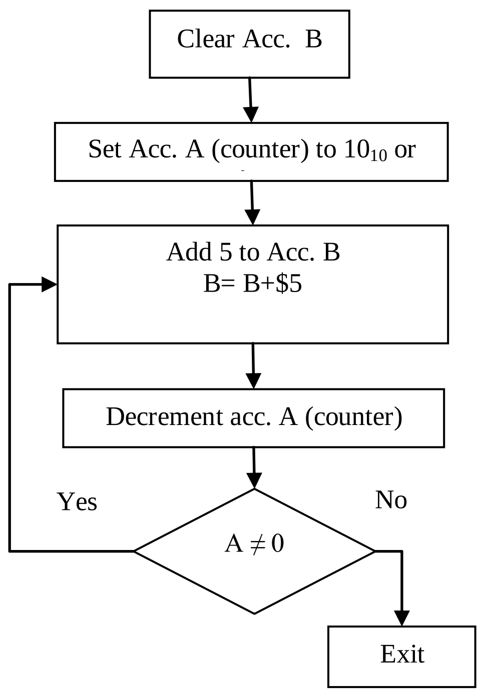
3. Assignment questions

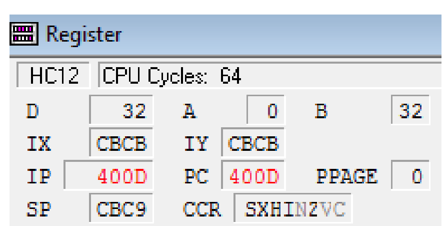
Q1) Write a program to clear the accumulator B and then add 5 to acc. B 10 times using the loop concept. Use the zero flag and BNE with DECA. Draw the program flowchart

In order to do the required operation accumulator A was used to represent counter in which it will be easier to modify than an actual memory address. First of all it should be cleared and then load it with the hexadecimal value A which is equivalent to 10 in decimal. In the loop section we add $5 to acc. B, decrement A and compare it with zero to make a decision (if not equal branch to loop section, else exit).

Counter	Acc. A	
10	5	
9	A	
8	F	
7	14	
6	19	
5	1E	
4	23	
3	28	
2	2D	
1	32	Final result

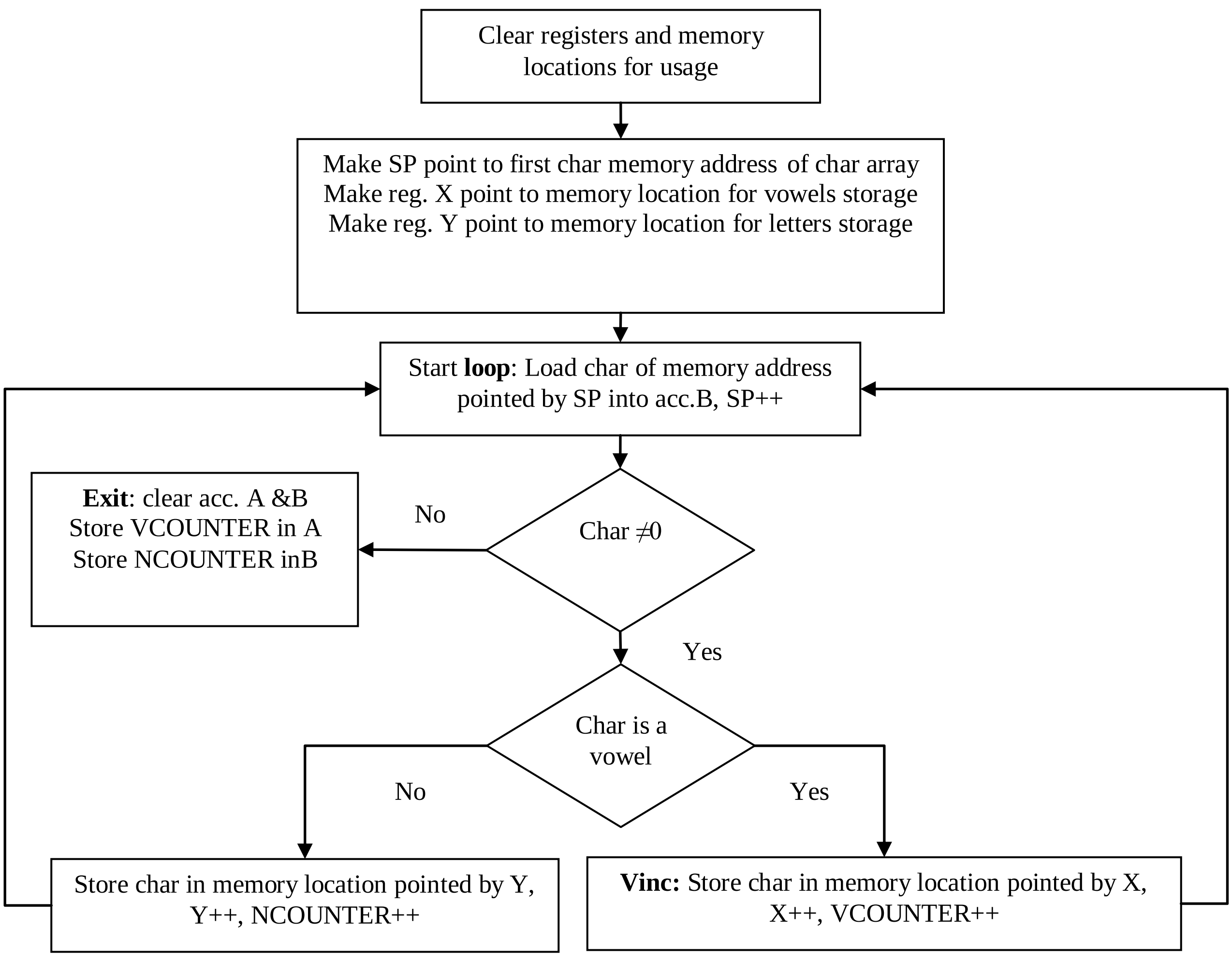
```
; Include derivative-specific definitions
        INCLUDE 'derivative.inc'
; export symbols
        XDEF Entry
sum     EQU $1003
        ORG $4000 ;Flash ROM address for Dragon12+
Entry:
        CLRA          ; clear acc.A
        CLRB          ; clear acc. B
        LDAA #$A      ; set acc. A to $A (10 in decimal) to be used as
counter
LOOP:   ADDB #$5      ; B+(M) -->B
        DECA          ; decrement acc. A by one (A--)
        CMPA  #$0     ; check if A=0
        BNE  LOOP     ; if A not equal to zero goto LOOP
        BRA Exit      ; Exit code
Exit:   BRA Exit      ; End your code here
```



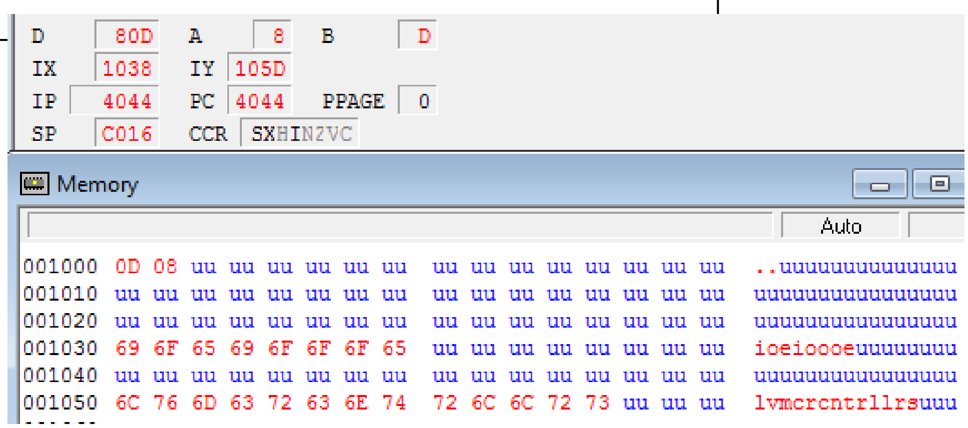


Q2) Write an HCS12 assembly program that reads the following string of 8 bit small letter characters and located in memory starting at location $1010.String: "ilovemicrocontrollers". The end of the sequence is indicated by a null. The program should count the number of vowel letters and store them in register A and the number of constant letters and store them in register B. the program should also copy the vowels letters to memory starting at location $1030, and the constant letters to memory starting at location $1050. Draw the program flowchart.

In this code the indexed addressing mode was used for the purpose of calculating the effective address to be used for reading data that have been stored in memory via the directive constant memory allocation method. In addition to that, it was also used to store a number of characters in a sequence memory location starting at a given address. Moreover, branch instructions were used for comparing the letters detected to identify vowels from non-vowels letters.



```
; Include derivative-specific definitions
          INCLUDE 'derivative.inc'
          XDEF Entry
LETTERS     DC.B 'ilovemicrocontrollers',$00 ; store sequence of letters in an array
NCOUNT:     SET $1000 ; counter for normal letters
VCOUNT:     SET $1001 ; counter for vowel  letters
          ORG $4000
Entry:    ; clearing registers and memory addresses for usage
          CLRA
          CLRB
          LDS #$0000
          LDX #$0000  ;
          LDY #$0000
          CLR NCOUNT
          CLR VCOUNT
          LDS #LETTERS  ; SP point to the memory address of first char
          LDX #$1030    ; used to store vowels detected at this address
          LDY #$1050    ; used to store vowels detected at this address
LOOP: LDAB 1,SP+    ; load acc.A with next memory address of the letters array
          CMPB #$000    ; check when letters array ends
          BEQ Exit      ; if reached end of array go to exit
         ; comparison to detect vowels starts here
          CMPB #'a'
          BEQ Vinc
          CMPB #'e'
          BEQ Vinc
          CMPB #'i'
          BEQ Vinc
          CMPB #'o'
          BEQ Vinc
          CMPB #'u'
          BEQ Vinc
         ; at this stage the letter in acc. B is normal
          INC NCOUNT  ; increment normal letters counter
          STAB Y      ; store the letter at address pointed to it by Y
          INY         ; increment to point to next memory address
          BRA LOOP
Vinc:     INC VCOUNT  ; increment vowels counter
          STAB X      ; store the vowels at address pointed to it by X
          INX         ; increment to point to next memory address
          BRA LOOP
Exit:     CLRA ; clearing acc. A & B to restore the counters value in them
          CLRB
          LDAA VCOUNT
          LDAB NCOUNT
```



4. Conclusion

In this experiment introduces the branches and loops in assembly language. In the first task, the If-Then-Else statement using BEQ was investigated. This BEQ command is a branch instruction that modifies the condition code register which means branch-if equal-to-zero. It branches if the value of Z flag in condition code register is 1. The instruction BNE was used in the second task that means branch-if not equal-to-zero. This instruction branches if the value of Z flag in condition code register is 0. In task three, present the idea of nested loops with branching conditions. The instruction BLE branches the value of Z flag in condition code register is 1. The last task, it is investigated the usage of the loop in adding numbers that are stored memory locations using ABY which adds the register and with memory contents and storing the sum into specific memory location after the counter is zero.