Khalifa University of Science, Technology and Research
Electronic Engineering Department

ELCE333Microprocessor Systems laboratory

Laboratory Experiment 2

DEVELOPMENT & TESTING OF HCS12 PROGRAMS USING BRANCHING AND LOOPS

Lab Partners
Dana Bazazeh, 100038654
Moza Al Ali 100038604

Date Experiment Performed: 04/02/2015
Date Lab Report Submitted: 11/02/2015

Lab Instructor: Dr.Mohamed Al Zaabi/ Dr.MahmoudKhonji

SPRING 2015

Table of Contents

Table of Contents ***2***

List of Figures and Tables ***3***

Summary ***4***

1. Introduction ***5***

1.1Aims 6

1.2Objectives 6

2. Design and results ***7***

TASK - 1: IF-Then-Else Statement using BEQ **7**

TASK - 2: IF-Then-Else Statement using BNE **9**

TASK-3: Nested IF-Then-Else Statement : **10**

TASK-4: Simple Program with loops: **13**

3. Assignment Questions ***15***

4. Results and Analysis ***18***

5. Conclusions and Recommendations ***18***

List of Figures and Tables

Table 1: Memory Locations Contents when using BEQ……8
Table 2: Memory Locations Contents when using BNE……..10
Table 3: Nested If-Then-Else Statements……12

Figure 1: Flowchart of If-then-else statement using BEQ……7
Figure 2: assembly code for Task 1 Test 1……7
Figure 3: Flowchart of If-then-else statement usingBNE……..9
Figure 4: assembly code for Task 2Test 1 BNE……...9
Figure 5: Flow chart for nested if-then-else logic program BNE……...10
Figure 6: assembly code for nested if-then-else logic program BNE…….11
Figure 7: flow chart for task 4…….13
Figure 8: assembly code for task 4……..14
Figure 9: assembly code to increment accumulator b with a value of 5 for 10 times…….15
Figure 10: flow chart for assignment Q1…….……....15
Figure 11: assembly code to check and store vowls and constants…….…...16
Figure 12: flow chart diagram for assignment Q2……...…17

Summary

In this lab experiment students built an understanding of the design and test of the HCS12 assembly programs with conditional branching structures and loops. This objective was accomplished by the successful implementation of the 4 given tasks. Moreover, after the completion of these tasks, students are now able to write codes of branching conditions easily. Task 1 & 2 were about writing and simulating a code that will check whether X1 is zero or not and hence take a decision according to the result of the test.Task 3 as well included 2 nested branches. Task 4 was about implement loops; here we used assembly language to create counters.Finally and after finishing all tasks successfully, students now have a practical understanding of what they were taught in the lecture.

1. Introduction

In this lab session we were required to learn more about writing a program using the HCS12 assembly language. An assembly language program consists of a sequence of statements that tells the computerto perform the desired operations.

As we all know, software development starts with problem definition. The problem presented by the application must be fully understood before any program can be written. At the problem definition stage, the most critical thing is to get you, the programmer, and your end user to agree upon what needs to be done. To achieve this, asking questions about the task is very important.

Once the problem is known, the programmer can begin to lay out an overall plan of how to solve the problem. The plan is also called a flowchart. Informally, a flowchart is any well-defined computational procedure that takes some value, or set of values, as input, and produces some value, or set of values, as output. So it's just a sequence of computational steps that transforms the input into the output.

Hence, the flowchart describes a specific computational procedurefor achieving that input/output relationship. After you are satisfied with the algorithm or the flowchart, convert it to source code.

If the result is what you expected then you go on to test the borderline inputs. Test for the maximum and minimum values of the input. When your program passes this test, then test it for the illegal input values. If your flowchart includes several branches, then you must use enough values to exercise all the possible branches. This is to make sure that your program will operate correctly under all possible circumstances.

Aim:

To gain experience in the design, development and testing of HCS12 assembly programs with conditional branching structures and loops.

Objectives:

On completion of this experiment the students should be able to:

1- Design flow-charts containing branching and loops.

2- Implement flow-charts using HCS12 assembly code.

3- Use Bcc instruction to implement branching and loops.

4- Download, run, and test code on a Dragon Plus Trainer board.

2. Design and Results

Task 1: If-Then-Else Statement using BEQ:

For Task 1, we will edit, assemble and debug the code provided in the labscript introduction. Here, we want to check whether a given byte is 0 or not and follow the flowchart provided below in figure1 accordingly. If it is 0, we will store the contents of Y1 into X1, otherwise we will store Y1 contents into X1. X1, Y1 and Y2 are all labels that we have to define and allocate addresses to using the EQU instruction.

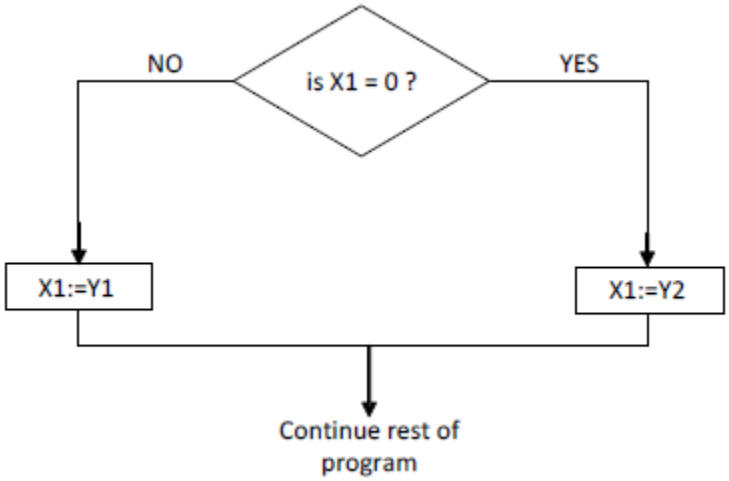

Figure 1: Flowchart of If-then-else statement using BEQ

```
  INCLUDE 'derivative.inc'
  XDEF Entry

X1 EQU $1000
Y1 EQU $1001
Y2 EQU $1002

  ORG $4000 ;Flash ROM address for Dragon12+

Entry:
  MOVB #$0, X1
  MOVB #$A1, Y1
  MOVB #$A2, Y2
  CLRA
  CMPA X1 ; compare zero with X1
  BEQ Eqzero ; if X1 is zero, go to "equalzero"
  MOVB Y1,X1 ; when X1 is not zero then X1=Y1
  BRA Exit ; go to "exit", bypass "equal zero"
Eqzero  MOVB Y2,X1 ; when X1=0 then make X1=Y2
Exit  BRA Exit ; End your code here
```

Figure 2: assembly code for Task 1 Test 1

The updated code is shown above in figure 2. We moved test bytes intoX1, Y1 and Y2. The main instruction here is the branching BEQ instruction which will check whether the CMPA X1; instruxction resulted in a zero result. If it did, we go to the Eqzero label and execute MOVB Y2,X1; otherwise we ignore the branch and move on to execute MOVB Y1,X1 before branching to exit the code. Table 1 below shows our results using 2 test cases, one where X=0 and the other X≠0.

Table 1: Memory Locations Contents when using BEQ

	Before Program Start			**After Program End**		
	X1($1000)	**Y1($1001)**	**Y2($1002)**	**X1($1000)**	**Y1($1001)**	**Y2($1002)**
Test 1	0	A1	A2	A2	A1	A2
Test 2	3	A1	A2	A1	A1	A2

Looking at the results, it is evident that when X1 is 0, Y2 contents moved to it while when nonzero, Y1 contents moved. Therefore we verified that our code is working as needed.

Task 2: If-Then-Else Statement using BNE:

In this task, we are asked to implement the same logic but using the BNE instruction instead. This will have an opposite effect as BEQ. The flow chart of the logic is as shown below in figure 3.

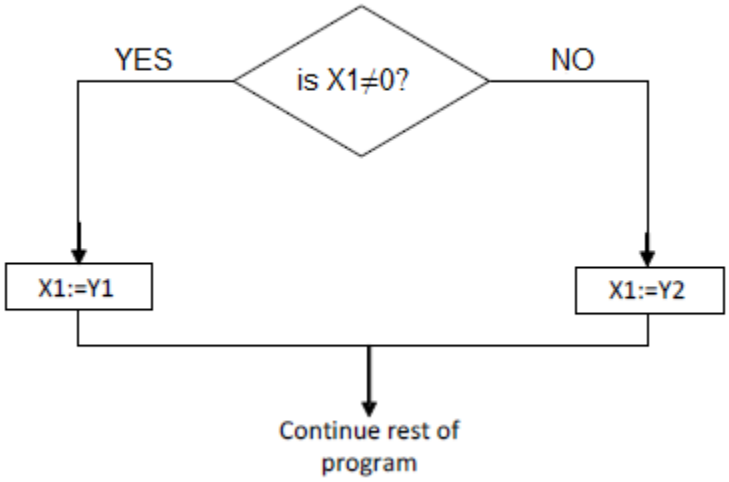

Figure 3: Flowchart of If-then-else statement using BNE

```
 INCLUDE 'derivative.inc'
 XDEF Entry

X1 EQU $1000
Y1 EQU $1001
Y2 EQU $1002

 ORG $4000 ;Flash ROM address for Dragon12+

Entry:
 MOVB #$0, X1
 MOVB #$A1, Y1
 MOVB #$A2, Y2
 CLRA
 CMPA X1 ; compare zero with X1
 BNE Noteqzero ; if X1 is zero, go to "equalzero"
 MOVB Y2,X1 ;when X1 is  zero then X1=Y2
 BRA Exit ; go to "exit", bypass "equal zero"
Noteqzero  MOVB Y1,X1 ; when X1 is not 0 then make X1=Y1
Exit  BRA Exit ; End your code here
```

Figure 4: assembly code for Task 2Test 1

In figure 4 above we can see that the only amendment done is that when we used the BNE instruction, the logic flipped and therefore we exchanged the instructions such that the branching will take us to execute the instruction MOVB Y1,X1. Again, after making and stepping through the program, we fill in the table to note down the result of the 2 test cases just as in task 1. Table 2 below shows the results found.

Table 2: Memory Locations Contents when using BNE.

	Before Program Start			After Program End		
	X1($1000)	**Y1($1001)**	**Y2($1002)**	**X1($1000)**	**Y1($1001)**	**Y2($1002)**
Test 1	0	A1	A2	A2	A1	A2
Test 2	3	A1	A2	A1	A1	A2

Looking at the results, it is clear that they are identical to our expected results and the one achieved in Task 1. Therefore we verified that our code is working as needed using the BNE instruction.

Task 3: Nested If-Then-Else Statements

For this task, we are required to write an assembly code program that will implement the if-then-else flow chart shown below in figure 5.

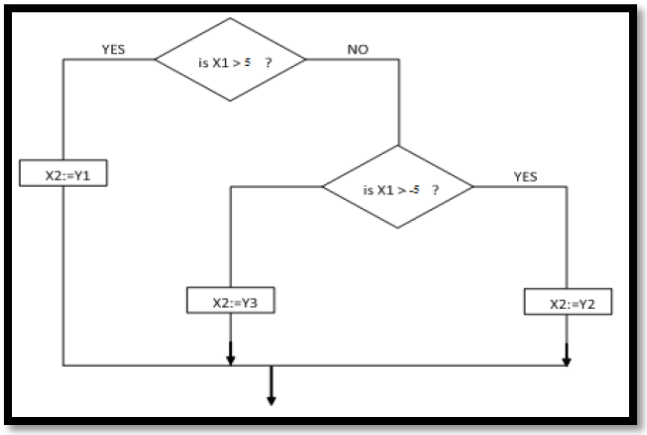

Figure 5: Flow chart for nested if-then-else logic program

As we can see in the flowchart above, we need to compare the contents of X1 with the hexadecimal 5 and check whether X1 is bigger or smaller. According to the comparison result, we will go forward with the program, where if X1 is bigger we will move the contents of Y1 to X2 and end program. On the other hand, if it is not bigger, meaning it is equal or less than 5, we will move into another nested if-then-else logic comparison. Here we will check whether X1 is bigger than -5 where if true will move contents of Y2 into X2. However, if X1 is less than or equal -5, we will move contents of Y3 into X2. The final assembly code is shown below in figure 6.

```
INCLUDE 'derivative.inc'
  XDEF Entry

X1 EQU $1000
X2 EQU $1001
Y1 EQU $1002
Y2 EQU $1003
Y3 EQU $1004

  ORG $4000 ;Flash ROM address for Dragon12+

Entry:
  MOVB #$FA, X1
  MOVB #$A4, X2
  MOVB #$A1, Y1
  MOVB #$A2, Y2
  MOVB #$A3, Y3
  CLRA
  LDAA #$5
  CMPA X1 ; compare zero with X1
  BLT YES
  LDAA #$5
  NEGA
  CMPA X1
  BLT YES2
  MOVB X2,Y3 ;
  BRA Exit ;

YES MOVB Y1,X2 ; when X1 is not zero then X1=Y1
    BRA Exit ; go to "exit", bypass "equal zero"

YES2 MOVB X2,Y2 ; when X1 is not zero then X1=Y1
     BRA Exit ;
```


Figure 6: assembly code for nested if-then-else logic program

In the code shown above, we give each of the labels X1,X2,Y1,Y2 and Y3 addresses from $1000-$1004 accordingly and move the test bytes into the contents of each of these address. Our first test is using X1=10 (X1>5). Moreover, we will be using accumulator A to hold the constant value 5 that is needed for the comparison. TheCMPA X1 instruction will do the operation A-(X1) and this will either be positive or negative. We will consider the case that it is negative which means X1>5. In this case, we need to branch to the YES label and execute MOVB Y1,X2 and exit. However, if X1≤5, we will continue with the execution to move to the next if-then-else instruction where again we will use the BLT instruction in the same way. We will present 3 different test cases as shown below in Table 2.

The results of the simulation are shown in the table below.

Table 3:Nested If-Then-Else Statements

	Before Program Start					**After Program End**				
	X1	**X2**	**Y1**	**Y2**	**Y3**	**X1**	**X2**	**Y1**	**Y2**	**Y3**
Test 1 X1 >5	10	A4	A1	A2	A3	10	A1	A1	A2	A3
Test 2 -5≤X1≤5	5	A4	A1	A2	A3	5	A2	A1	A2	A3
Test 3 X1<-5	FA	A4	A1	A2	A3	FA	A3	A1	A2	A3

From Table 1 above, we found that the content moved to X1 matched our expected value and that our code is working properly. In addition, we used FA in hexadecimal to represent -6 decimal in 2s complement. We chose the value of 5 in the second test to verify that the equality works properly on the boundary condition.

Task 4: Simple Program with loops:

For task 4, we need to use branching instructions to implement a loop that will add numbers together and decrement a pre-defined counter while doing so. This loop program can be written using branching and decrement instructions. The main idea behind the program is to add 5 numbers and store the sum in a memory address. A wider understanding of the logic can be attained by looking at the flowchart of figure 7 below.

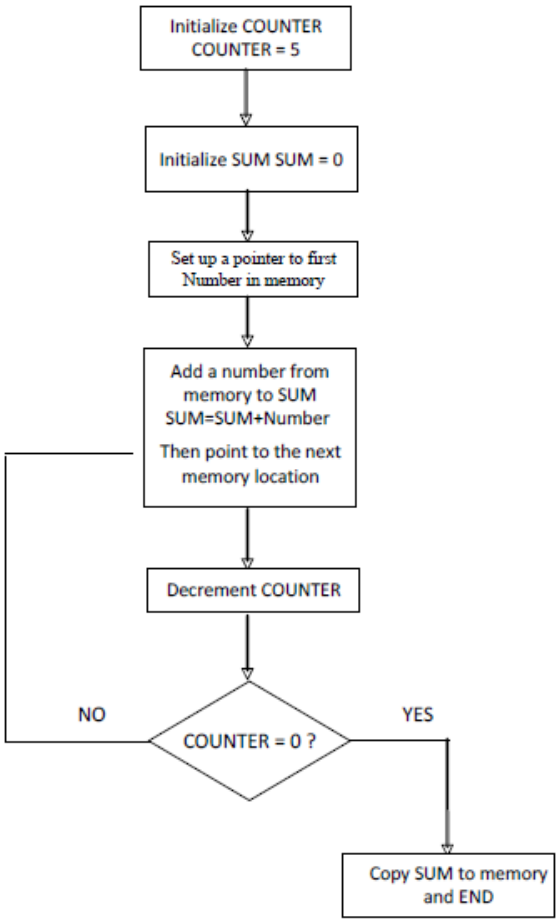

Figure 7: flow chart for task 4

:

```
 INCLUDE 'derivative.inc'
 XDEF Entry

SUM EQU $1000
NUMBERS DC.B $12,$1A,$43,$15,$28

        ORG $4000
Entry:
        CLRA
        CLRB
        LDAA #$5
        LDY #$0
         LDX #NUMBERS

RL:
        LDAB 1, X+
         ABY
        DECA
        CMPA #$0
        BEQ Exit
        BRA RL

Exit    STY SUM
        BRA Exit
```

Figure 8: assembly code for task 4.

As can be seen in figure 8 above, we first initialize the SUM variable to memory location $1000. Then we define an array NUMBERS containing the 5 numbers, using the DC.B directive. Moreover, we have assigned accumulator A as the counter and loaded it with the value 5. The index register X will be used to point to the memory address containing the number to be added, and will therefore be incremented by 1 using the LDAB instruction and that is after moving the contents of the address in register X to accumulator B(postfix). Next, we use ABY to add the contents of registers B and Y and store result in Y. We will then decrement the counter using DECA and check whether the counter reached 0 using the BEQ to move to label exit where we save the value of register Y into SUM. If the counter did not yet reach 0, it will repeat loop RL.

After debugging and executing the program, we found that the final result stored in SUM was AC which is the correct summation of the numbers provided; therefore our code must be working properly.

3. Assignment Questions

1.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.

```
INCLUDE 'derivative.inc'
  XDEF Entry

FIVE EQU $1000
  ORG $4000

Entry:
  LDAA #$5
  STAA FIVE
  CLRA
  CLRB
  LDAA #10

RR:
    ADDB FIVE
    DECA
    CMPA #$0
    BNE RR

Exit  BRA Exit
```

Figure 9: assembly code to increment accumulator b with a value of 5 for 10 times

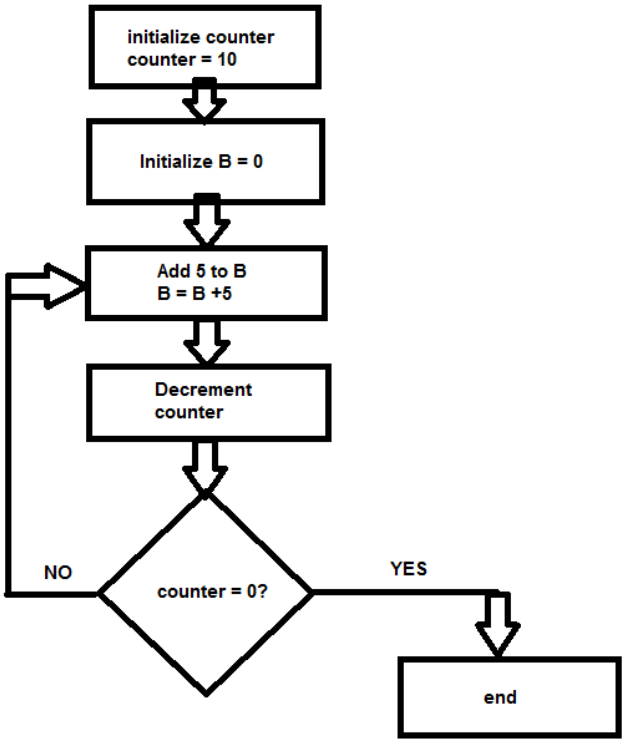

Figure 10: flow chart for assignment Q1

We set a memory location FIVE to hold the value of 5 and then use accumulator a as a counter. It is important to note that accumulator a should be loaded with the decimal value of 10 rather than hexadecimal since these are not the same. The loop RR will use ADDB to add the contents of location FIVE which has the value 5 to B and store the answer in B. As long as the counter is not yet 0, the BNE instruction will branch to repeat the loop RR.

2. Write an HCS12 assembly program that reads the following string of 8-bit small letter characters located in memory starting at location $1010.

String : "ilovemicrocontrollers"

```
   INCLUDE 'derivative.inc'
XDEF  Entry
countVowels EQU $1000
countConstants EQU $1002
Vowels EQU $1030
Constants EQU $1050
STRING   DC.B 'ilovemicrocontrollers',$00

              ORG $4000
 Entry:
     CLRA
     CLRB
     CLR countVowels
     CLR countConstants
     LDX #STRING
     LDY #Vowels

Repeat  LDAA X
     CMPA #$00  ;Check if end of string
     BEQ done   ;If we checked the whole string then end the program
     CMPA #$61  ;check if a
     BEQ v
     CMPA #$65  ;check if e
     BEQ v
     CMPA #$69  ; check if i
     BEQ v
     CMPA #$6F   ;check if o
     BEQ v
     CMPA #$75   ;check if u
     BEQ v
      ; if it is a constant continue
    STAA Constants
    INC Constants
    INX
    INC countConstants
    BRA Repeat

v:  ;if it is a vowel branch here
    STAA Y
    INC countVowels
    INY
    INX
    BRA Repeat

done:
  LDAA countVowels
  LDAB countConstants
Exit   BRA Exit
```

Figure 11: assembly code to check and store vowls and constants

Here, we will initialize 4 memory locations, 2 to hold the counters for vowels and constants and the other 2 to hold the address where we will store the letters. We define the string and use index register X to point to the next letter as we did in task 4 and Y to hold memory location where we will store vowels. The first loop REPEAT will have the code to check whether the letter is a vowel or if we reached the end of string using the ascii equivalents of the vowels and 00 for null. If a vowel is found we will branch to V where we store the letter in Y and then increment the vowel counter as well as X and Y. Otherwise we will continue and do the same but for constant letters. At the end we will store the contents of the counters into accumulator A and B respectively.

The flowchart for this program is shown below:

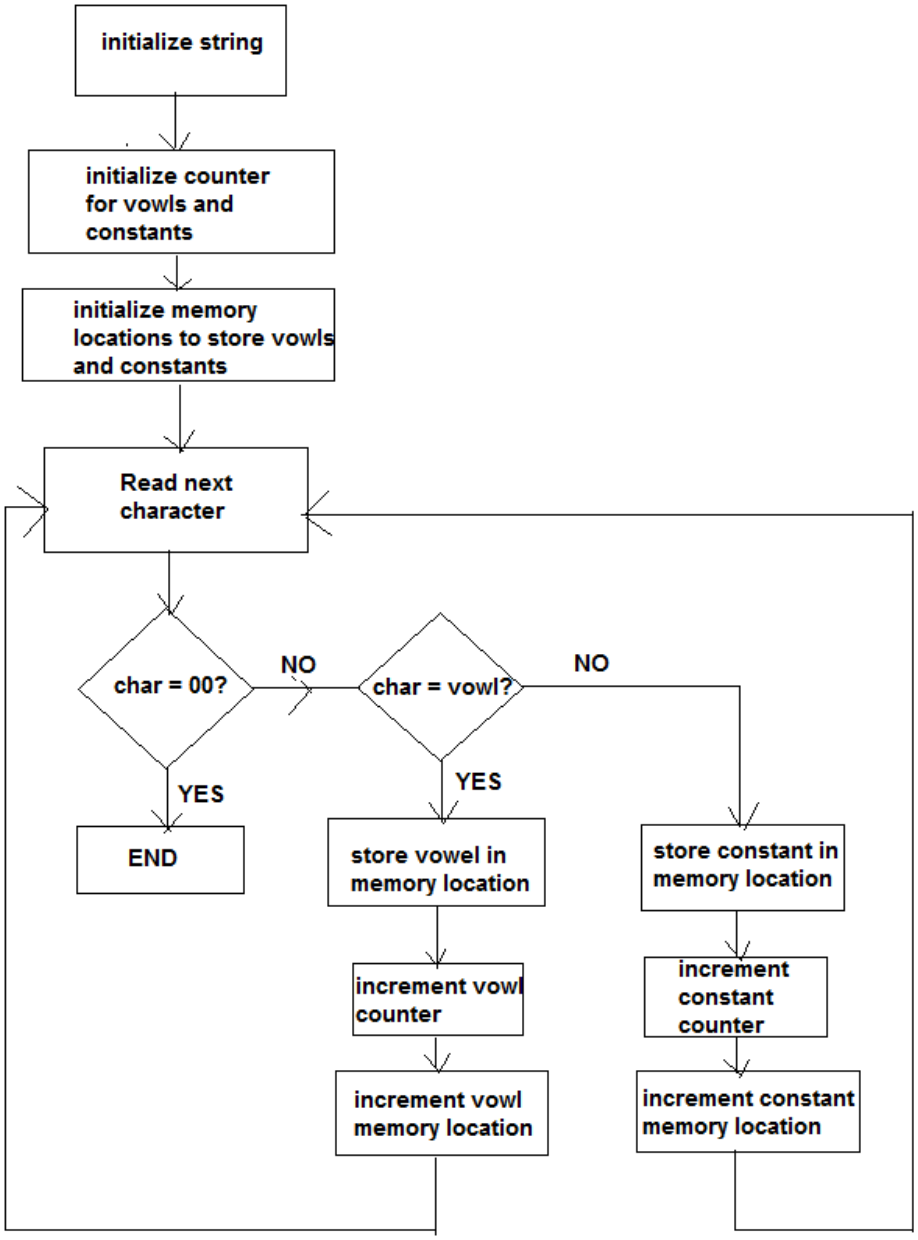

Figure 9: flow chart diagramfor assignment Q2.

4. Results and Analysis

The main topic explained in the 4 tasks was the use of the branch commands. Briefly, task 1 was about the use BEQ, the branch command will be executed if the value saved in the register equals zero. Whereas in the second task, the command BNQ was used, this command is totally the opposite of the command used in the first task; here the branch will be executed if the value in the register is not equal zero. On the other hand, the third task used two impeded loops the first one tested if the number is larger than 10, while other loop tested if the number is less than -10. Finally, the forth task discussed four the branch loop, this was used to do a sum of five numbers from the memory location and was done using a counter in register A, register X was used as a pointer to the next number, therefore the result was saved in B.

5. Conclusions and Recommendations

To summarize this report, students now have a good general understanding about the procedure of writing a program using the Code Worrier. Moreover, students believe that objectives have been also reached at the end of this laboratory experiment. Different tasks were provided with different skills required. In the first task, students were asked to test a code by using 2 values of X, i.e. X=0 and X≠0. By this test, the importance of BEQ branch was observed by students. Apart from that, the second task represented another branch condition BNE which is the inverse of BEQ so the results were opposite to the first task. Moving to the third task, a new program was introduced, operation that includes more than one branch condition such as BLT condition. It helped in creating nested If-Else-statements. Finally, in task 4 students created a program using loops and counters.

To sum up, this lab session concentrated on teaching students the branching and looping mechanisms using the assembly language.