

**Khalifa University of Science, Technology and Research**

**Electronic Engineering Department**

**ELCE332 Microprocessor Systems laboratory**

**Laboratory Experiment 2**

**Microcontroller Assembly Program Development**

**Lab Partners**

Hamad Eisa Alhazami 100036985

Jasim Alhammadi 100035510

**Date Experiment Performed**: 4-2-2015

**Date Lab Report Submitted**: 11-2-2015

**Lab Instructor: Mr. Mohammed Alzaabi/ Mr. Mahmoud Khounji**

**Spring 2015**

# Summary

# 

In this report, the results of the tasks conducted in the laboratory experiment are illustrated by the means of flowcharts, tables and graphs. There were four tasks at hand; the first two were basically concerned about composing and simulating a program that depends on the value of X1. If the value of X1 is zero the program will do one action, if not, the program will conduct another action. On the same manner, task 3 involved 2 nested branches. Last but not least, in task 4, counters were created using assembly language loops.

# 

Contents

[Summary 2](#_Toc411424032)

[1. Introduction 5](#_Toc411424033)

[1.1 Aim 5](#_Toc411424034)

[1.2 Objectives 5](#_Toc411424035)

[2. Design and Results 6](#_Toc411424036)

[2.1 TASK 1: IF-THEN-ELSE STATEMENT USING BEQ 6](#_Toc411424037)

[2.2 TASK 2: IF-THEN-ELSE STATEMENT USING BNE 7](#_Toc411424038)

[2.3 TASK 3: Nested If-Then-Else Statements 8](#_Toc411424039)

[2.4 TASK 4: Simple Program with Loops 11](#_Toc411424040)

[3. Assignment questions 13](#_Toc411424041)

[4. Results and Analysis 16](#_Toc411424042)

[5. Conclusions and Recommendations 16](#_Toc411424043)

**List of tables**

[Table 1: Results of test 1 and test 2 before and after running the program using BEQ 7](#_Toc411423987)

[Table 2: Results of test 1 and test 2 before and after running the program using BNQ 8](#_Toc411423988)

[Table 3: Results before and after running the code 10](#_Toc411423989)

**List of figures**

[Figure 1: flow chart show an example of a loop 5](#_Toc411424237)

[Figure 2: Task 3 nested-if-then-else algorithm 8](#_Toc411424238)

[Figure 3: Task 4 summation by loop algorithm 11](#_Toc411424239)

[Figure 4: program flow chart 15](#_Toc411424240)

# Introduction

As shown in the figure below, the flow of instructions is initiated and only changes upon the appearance of a branch instruction. If the branch condition is satisfied, the sequence of execution is altered and the instructions that follow the branch are executed. On the other hand, the flow of instructions remains the same if the branch condition is not satisfied.

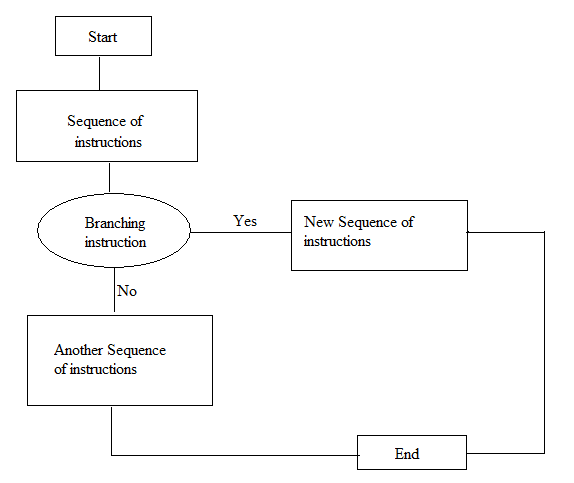


Figure 1: flow chart show an example of a loop

## 1.1 Aim

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

## 1.2 Objectives

1- Design flow-charts containing branching and loops.

2- Implement flow-charts using HCS12 assembly code.

3- Use branch instruction to implement branching and loops.

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

# Design and Results

## TASK 1: IF-THEN-ELSE STATEMENT USING BEQ

In the first task, it is required to write a program using the introduction code that checks the value of X1 by using the assembly instruction BEQ, if it is equal zero, it will assign the value of Y1 to X1. If it is not it will assign the value of Y2 to X1. The assembly code is shown below:

**; Include derivative-specific definitions**

**INCLUDE 'derivative.inc'**

**; export symbols**

**XDEF Entry**

**ORG $4000 ;Flash ROM address for Dragon12+ ]**

**X1 EQU $1000**

**Y1 EQU $1001**

**Y2 EQU $1002**

**Entry:**

**MOVB #$3, 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**

The code is executed, the results are written in table 1. As it shown in table 1, the value of X1 in the first test (X=0) has been changed to A2 (Y2 value). And in the second test, the value of X1 (X=3) has been changed to A1 (Y1 value). The values after running the code matched the predicted values of memory locations.

Table 1: Results of test 1 and test 2 before and after running the program using BEQ

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Before Program Start | | | After Program End | | |
| **$1000** | **$1001** | **$1002** | **$1000** | **$1001** | **$1002** |
| Test 1 | 00 | A1 | A2 | A2 | A1 | A2 |
| Test 2 | 03 | A1 | A2 | A1 | A1 | A2 |

## TASK 2: IF-THEN-ELSE STATEMENT USING BNE

In the second task, it is required to modify the program in task 1 and use the instruction BNE instead of using BEQ. The same result of task 1 should be shown; if X1 it is equal zero, it will assign the value of Y1 to X1. If it is not zero, it will assign the value of Y2 to X1. The assembly code is shown below:

**; Include derivative-specific definitions**

**INCLUDE 'derivative.inc'**

**; export symbols**

**XDEF Entry**

**ORG $4000 ;Flash ROM address for Dragon12+ ]**

**X1 EQU $1000**

**Y1 EQU $1001**

**Y2 EQU $1002**

**Entry:**

**MOVB #$3, X1**

**MOVB #$A1, Y1**

**MOVB #$A2, Y2**

**CLRA**

**CMPA X1 ; compare zero with X1**

**BNE Neqzero ; if X1 is zero, go to "equalzero"**

**MOVB Y2,X1 ; when X1 is not zero then X1=Y1**

**BRA Exit ; go to "exit", bypass "equal zero"**

**Neqzero MOVB Y1,X1 ; when X1=0 then make X1=Y2**

**Exit BRA Exit ; End your code here**

The code is executed and the results are written in table 2. As it shown in table 2, the value of X1 in the first test (X=0) has been changed to A2 (Y2 value). And in the second test, the value of X1 (X=3) has been changed to A1 (Y1 value). The values of memory locations after running the code matched the predicted values and task 1 values.

Table 2: Results of test 1 and test 2 before and after running the program using BNQ

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Before Program Start | | | After Program End | | |
| **$1000** | **$1001** | **$1002** | **$1000** | **$1001** | **$1002** |
| Test 1 | 00 | A1 | A2 | A2 | A1 | A2 |
| Test 2 | 03 | A1 | A2 | A1 | A1 | A2 |

## 2.3 TASK 3: Nested If-Then-Else Statements

In the third task, it is required to write a program that carries out the algorithm shown below. The algorithm starts by comparing the value X1 to certain values.



Figure 2: Task 3 nested-if-then-else algorithm

If X1 is larger than 5, it will assign the value of Y1 to X2. Otherwise it will compare X1 to -5. If X1 is larger than -5, it will assign the value of Y2 to X2. Else, it will assign the value of Y3 to X2. The code for the algorithm is shown below:

**; Include derivative-specific definitions**

**INCLUDE 'derivative.inc'**

**; export symbols**

**XDEF Entry**

**ORG $4000 ;Flash ROM address for Dragon12+ ]**

**X1 EQU $1000**

**X2 EQU $1004**

**Y1 EQU $1001**

**Y2 EQU $1002**

**Y3 EQU $1003**

**Entry:**

**MOVB #$7, X1**

**MOVB #$0, X2**

**MOVB #$A1, Y1**

**MOVB #$A2, Y2**

**MOVB #$A3, Y3**

**CLRA**

**CMPA #$5**

**BLT less**

**CMPA #$FB**

**BLT less1**

**MOVB Y3,X2**

**BRA Exit**

**less:**

**MOVB Y1,X2**

**BRA Exit**

**less1:**

**MOVB Y2,X2**

**Exit:**

**BRA Exit**

The first condition (BLT instruction) will check if the X1 value is less than 5 or not. If the condition is true, it will branch to the “less” condition (X2 = Y1). Otherwise it will continue the code execution. The second branching is to check if the value of X1 is less the -5 and it will branch if the condition true (X2=Y2). Else, it will assign the value of X2=Y3.

The code is executed and the results are written in table 1. As it shown in table 1, the value of X1 in the first test (X=0) has been changed to A2 (Y2 value). And in the second test, the value of X1 (X=3) has been changed to A1 (Y1 value). The values after running the code matched the predicted values of memory locations.

Table 3: Results before and after running the code

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Before Program Start | | | | | After Program End | | | | |
| **X1** | **X2** | **Y1** | **Y2** | **Y3** | **X1** | **X2** | **Y1** | **Y2** | **Y3** |
| Test 1 | 0 | 0 | A1 | A2 | A3 | 0 | A3 | A1 | A2 | A3 |
| Test 2 | 7 | 0 | A1 | A2 | A3 | 7 | A1 | A1 | A2 | A3 |
| Test 3 | -6 | 0 | A1 | A2 | A3 | -6 | A2 | A1 | A2 | A3 |

The code is executed and the results are written in table 3. As it shown in table 3, the value of X2 (X1=0) in the first test has been changed to A3 (Y3 value). In the second test, the value of X2 (X1=7) has been changed to A1 (Y1 value). Finally, in the third test, the value of X2 (X1= -6) is changed to A2 (Y2 value). The values after running the code matched the predicted values of memory locations.

## 2.4 TASK 4: Simple Program with Loops

In the last task, it is required to write a code that implements a loop that results in the summation of five numbers. Several steps are required in order to write the code as shown in the algorithm on the right. The counter, sum, and pointer should be initialized first to the registers. Then, the five numbers should be summed using the loop. Finally, the summation should be stored in the required memory location. The code for the program is shown below:

Figure 3: Task 4 summation by loop algorithm

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**; Include derivative-specific definitions**

**INCLUDE 'derivative.inc'**

**; export symbols**

**XDEF Entry**

**ORG $4000 ;Flash ROM address for Dragon12+ ]**

**NUMBERS DC.B $12, $1A, $43, $15, $28**

**SUM1 EQU $1000**

**Entry:**

**LDAA #$5**

**LDY #$0000**

**LDX #NUMBERS**

**loop:**

**LDAB 1, X+**

**ABY**

**DECA**

**CMPA #$0**

**BNE loop**

**STY SUM1**

**JMP EXIT**

**EXIT:**

**JMP EXIT**

As it shown in the code above, the sum memory location is initialized first. Then, the counter, summation, and the numbers pointer are initialized to the registers A, Y, and X consecutively. For each iteration in the loop, it adds the number pointed by the pointer register (reg. X) to the sum (reg. Y). Then, it decrements the counter and compares it with the value zero to either branch or continue executing the code. Once the counter reaches zero, the compiler continues executing the code, and saves the value in reg. Y in the memory location $1000.

# 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-specific definitions**

**INCLUDE 'derivative.inc'**

**; export symbols**

**XDEF Entry**

**ORG $4000 ;Flash ROM address for Dragon12+ ]**

**SUM1 EQU $1000**

**Entry:**

**CLRB**

**LDAA #$A**

**loop: ADDB #$5**

**DECA**

**CMPA #$0**

**BNE loop**

**STAB SUM1**

**JMP EXIT**

**EXIT:**

**JMP EXIT**

**2. Write an HCS12 assembly program that reads a string of 8-bit characters located in memory starting at location $1010. The end of the sequence is indicated by a null. The program should count the number of upper-case letters and the number of lower-case letters and store them in registers A and B respectively. The program should also copy the upper-case letters to memory stating at location $1100, and the lower-case letters to memory starting at location $1200. Test your program using a set of at least ten characters.**

**; Include derivative-specific definitions**

**INCLUDE 'derivative.inc'**

**; export symbols**

**XDEF Entry**

**ORG $4000 ;Flash ROM address for Dragon12+ ]**

**STRINGS DC.B 'ilovemicrocontollers',$00**

**VowelC EQU $1070**

**ConstC EQU $1090**

**VowelS EQU $1030**

**ConstS EQU $1050**

**Entry:**

**CLR $1070 ;Vowel counter**

**CLR $1090 ;Const counter**

**CLRA**

**CLRB**

**LDX #STRINGS**

**LDY #VowelS**

**loop LDAA X ;(X)-->A**

**CMPA #$00 ;Compare null**

**BEQ exit**

**CMPA #$69 ; Compare 'i'**

**BEQ vowel**

**CMPA #$65 ;Compare 'e'**

**BEQ vowel**

**CMPA #$61 ;Compare 'a'**

**BEQ vowel**

**CMPA #$6F ;Compare 'o'**

**BEQ vowel**

**CMPA #$75 ;Compare 'u'**

**BEQ vowel**

**constant:**

**STAA ConstS ;(A)--> Starting at $1050**

**INX**

**INC ConstC**

**BRA loop**

**vowel:**

**STAA Y**

**INC VowelC**

**INY**

**INX**

**BRA loop**

**exit:**

**LDAA VowelC**

**LDAB ConstC**

**Exit BRA Exit**

# 

Figure 4: program flow chart

# Results and Analysis

In the first task, the instruction CMPA and BEQ are used to compare and branch to the location if the condition is true or continue to the next instruction if the condition if false. In the second task, the instruction BNQ was used instead of BEQ. BNQ instruction is exactly the opposite of BEQ, it branches if the condition is false and vice versa. The same algorithm logic was used for the code but the values of Y1 and Y2 registers were swapped. Both tasks 1 and 2 showed the same results. In the third task, the instruction BLE (branch less than) branches if the condition is true and continues if condition is false. BLE can be changed with the instruction BLT to perform the same operation but the code will differ in branching the conditions. In the final task, the instruction LDAB was used to load an array of numbers to the register. The instruction DECA was used to decrement the value of acc. A. the loop was iterated five times as it was observed in CodeWarrior while running the code. Then, it breaks the loop and stores the sum by the instruction STY. The summation was compared to the calculator summation and they were equal.

# Conclusions and Recommendations

After completing all tasks in the laboratory session, it is now easy to construct flow-charts with branching and loops. In addition to that, implementing them using HCS12 assembly code and using branch instructions is clearer now.