ECE 212 Lab - Introduction to Microprocessors
Department of Electrical and Computer Engineering
University of Alberta

Lab 2: Introduction to Addressing Modes

Student Name	Student
Arun Woosaree	XXXXXXX
Navras Kamal	1505463

Contents

1 Introduction

from the last lab The purpose of this lab is to learn and test with the Assembly language in a hands on environment in order to solidify the concepts learned in class and to improve our skill in the language. In addition, we will be learning how to handle the Netburner ColdFire boards directly, manipulating the contents of their memory and data structures. Finally, we are going to learn how to work inside the Eclipse IDE environment and how to properly use the powerful tools that come alongside it.

The code will be developed for the Netburner ColdFire Platform, which has some parameters that should be kept in mind throughout testing. There are multiple Data and Address registers, and the memory is indexed by hexadecimal codes. The data and the stored locations can each be modified directly, values can be compared and the code can branch into different sections depending on the values of the CCR (Condition Code Register) bits, which store information about the outcome from the last comparison or valid operation, such as if a value is negative or zero. These will be used to execute code conditionally.

The lab will be split into two sections, each with a different goal but with similar implementations. For one part we will be taking in an ASCII value and if the character it represents is a character included in the symbols for hexadecimal numbers then that hexadecimal value is output, otherwise it returns an error. For the second part an ASCII value is taken in, and if the character it represents is a letter in the English language (A-Z) then the ASCII code for the character in the opposite case is output. Thus, valid uppercase English letters are converted to their lowercase equivalents in ASCII and vice versa.

These experiments will introduce implementing high level programming practices of loops, if - then - else statements, using the Assembly language. More specifically, this will introduce the movement of memory and data to and from different parts of the Netburner chip, using techniques such as referencing memory addresses and copying data to local data registers. The debugger tools of the IDE will be used to closely examine this movement and to analyze all changes to the data in order to solve issues in development as well as to test the code. This is all building off the concepts explored in Lab 0.

The computing science practice of Pair Programming was also introduced, where two people develop and test code in tandem. The partners are divided into the Driver and the Navigator. In this structure the Driver is the one responsible for the physical typing of the code into the computer, and the Navigator reviews this code and clarifies the meaning of each passage in order to find bugs faster and to improve efficiency in testing. The two partners should communicate constantly and switch in order to maximize the efficiency of this working model. This will not only decrease time needed for development but it will also improve the quality of code from each partner.

2 Design

2.1 Part A

from the last lab For the first part of the lab, the address register a1 was chosen to initially point to memory location 0x2300000, which is the starting point of where the input data was stored. We used this address register to keep track of the memory location of the next long word of data to be read as an input to our program. Even though one memory location is capable of storing one ASCII character, in this lab, 4 memory locations were used to store one ASCII character, as specified in

the lab manual. Next, a2 was selected to initially point at the memory location 0x231000, which is the starting location of where our output for the converted values was. This register was used to keep track of the memory location of where the next long word of our converted data would go. The data register d2 was chosen to temporarily store data so we could do comparisons and process the input data. The *SetZeros.s* and the *DataStorage.s* files, which were provided, were used to initialize memory contents.

2.1.1 Part 1

We started with a loop branch that served as our main looping function. This loop first starts by moving data from a memory location pointed to by a1, to the data register d2 so that we could start comparing the input data to known ASCII values. In the first comparison, the input character is compared to '0x0D', which is the ASCII code for the 'Enter' key. This code is meant to signal the end of the program, so if the input was the ASCII code for 'Enter', we branched to a label that would end our program. The next step was to determine if the input character was valid. For Part A, an input character was valid if it was an ASCII character from 0-9, A-F, or a-f. In other words, the data was accepted if it was in the following ranges: 0x30-0x39, 0x41-0x46, or 0x61-0x66. If the input character was invalid, we brached to a label named 'err', which would put the error code '0xFFFFFFFF' in the memory location pointed to by a2, which keeps track of where our output data goes.

2.1.2 Part 2

If the input character was valid, we had three branches to take care of each of the accepted ranges of input. For example, in the branch that took care of the input range A-F, the ASCII value of 'A' was subtracted from the input value, and the difference was added to the hex value 0xA. The converted value is then moved to the memory location pointed to by a2, which keeps track of where our output data goes. Similar steps were done to convert input characters in the other accepted ranges. After converting the input character and moving the output to the location pointed by a2, we branched to a label 'endloop', which increments the addresses stored in a1 and a2 by 4, and then branches to the loop, where the process is repeated. The flowchart diagram can be found in the Appendix.

2.1.3 Part 3

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

2.1.4 Part 4

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

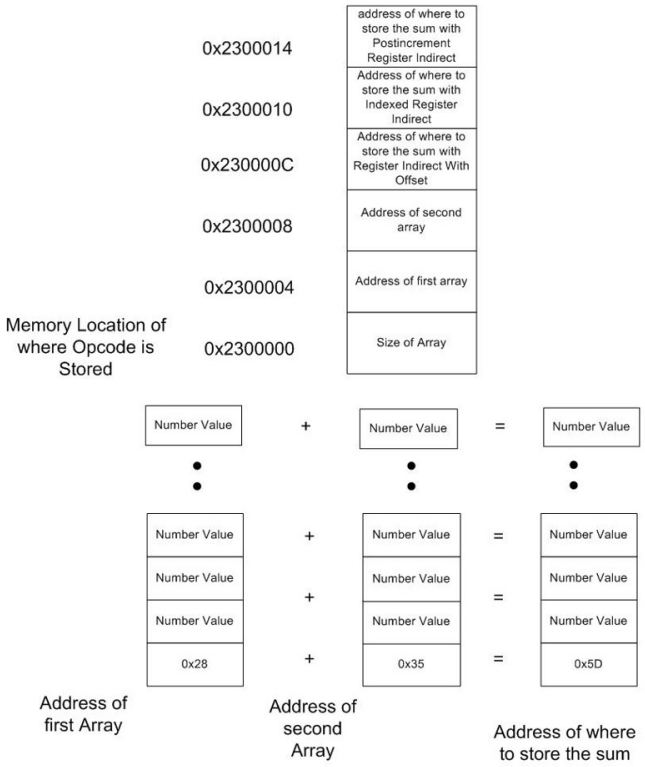


Figure 1: **need a better caption**Layout of data

2.2 Part B

from the last lab The design for Part B was very similar to Part A. The address register a1 still initially points to 0x2300000, but this time, a2 now points to 0x2320000, which is where the converted data is stored.The same data register d2 was used to temporarily store the input and process the data. Just like in Part A, the *SetZeros.s* and the *DataStorage.s* files were used to initialize memory contents as well.

Once again, we had a loop branch that initially loads input data from the location pointed to by a1 into data register d2, and the ASCII 'Enter' code still terminates the program, as before. In Part B, valid inputs are the ASCII characters A-Z and a-z (0x41-0x5A and 0x61-0x7A). Error handling was also similar to Part A, where the value 0xFFFFFFFF was stored at the memory location pointed to by a2. We had two branches to handle the two ranges of accepted characters.

For example, if an input character was in the range a-z, a difference was taken relative to the ASCII character 'a', and added to the ASCII character 'A'. The converted value is then stored at the memory location pointed to by a2, and the addresses stored in a1 and a2 are incremented by 4 before the loop starts over. The flowchart diagram can be found in the Appendix.

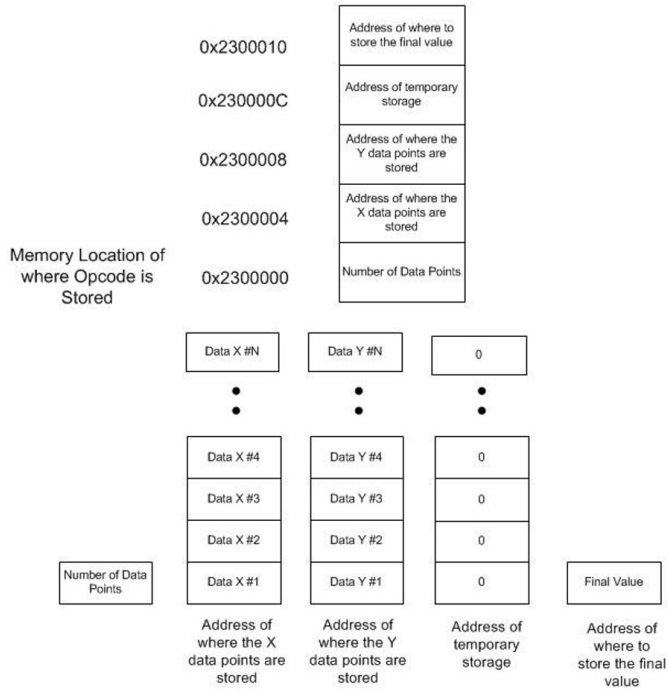


Figure 2: **need a better caption**Layout of data

3 Testing

3.1 Part A

from the last lab Initially, we visually tested our code by using the debugger in Eclipse IDE. While stepping through the code, we would check the values at relevant memory locations, and the data and address registers. When the bugs were ironed out, we went on to the next phase of testing. Our code was tested using the provided *Lab1Test.s* file. More specifically, this program was moved into the project folder, downloaded to the ColdFire microcontroller, and the MTTTY serial monitor was loaded to monitor the expected output. Our code was further tested by replacing the

'DataStorage.s' file with the other variants provided named: *DataStorage1.s*, *DataStorage2.s*, and *DataStorage3.s*. Finally, our program, which produced the correct output, was verified by a lab TA.

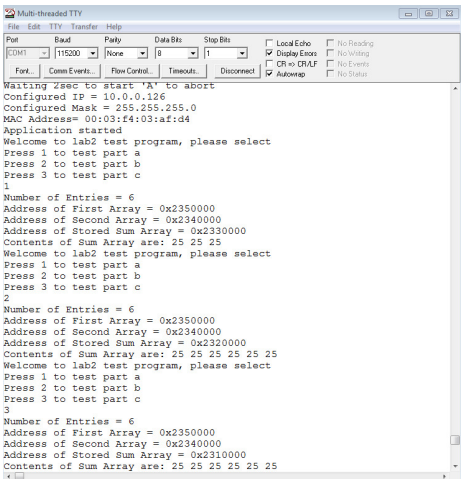


Figure 3: MTTTY output when testing our Part A solution

3.2 Part B

from the last lab The procedure for testing our code for part B was very similar to the process described above in Part A. We visually inspected our code in the Eclipse IDE, used the Eclipse debugger to step through our code, and monitor relevant memory addresses and registers. When we were confident that we had a working solution, we used the provided files *Lab1Test.s*, and the *DataStorage*.s* files to verify our solution by downloading the program to the ColdFire microcontroller, and monitoring the output in MTTTY. Finally, our solution was verified by a lab TA.

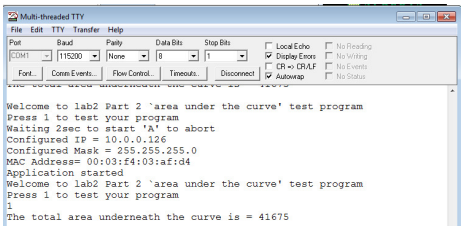


Figure 4: MTTTY output when testing our Part B solution

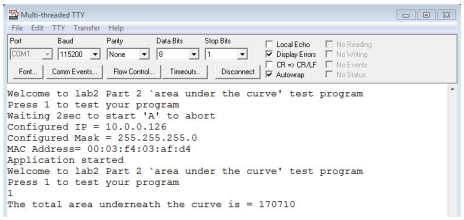


Figure 5: MTTTY output when testing our Part B solution

4 Questions

from the last lab

4.1 Question 1

"What happens when there is no exit code 0x0D provided in the initialization process? Would it cause a problem? Why or why not?"

A: Yes. This would cause a problem, because there would be no way to exit the program, so the program would keep reading data and moving the converted values to memory locations, until the program attempts to read or write to a memory location that is restricted or non-existant. This would then cause the program to crash.

4.2 Question 2

from the last lab *"How can our code be modified to provide a variable address range? For example, what if I only wanted to convert the first 10 data entires? "*

A: Assuming the data range would be fixed and hardcoded into the Assembly code it would be possible to write the max range (ie. 10) into an unused data register such as %d3. Then, instead of checking for the enter code on each iteration where there is an invalid value, we could check the value stored in %d3 before checking the validity of the value stored in the current memory address. If %d3 is zero then jump to the **end** label, breaking the loop. Another way to do it would be again to assume that the number of iterations is fixed and the size of data being checked is fixed as being long-words would be to check the value of the memory address stored in (%a1) after each iteration before returning to the beginning of the loop and after the memory addresses have been incremented. If at that point the memory address is [(initial memory address 0x2300000) + (0x4 * N)], where N is the number of desired iterations (this value would be hardcoded, this is just a general case), then jump to the **end** label, breaking the loop. This way would be more memory efficient as it does not require an additonal data register and the modification of a counter value.

5 Conclusion

from the last lab This lab demonstrated how to perform operations and modify data while moving it around using the Assembly language for the ColdFire architechture. In addition, the lab improved

our understanding of the debugger software, a very powerful tool in the development of this kind of code. The main issue we found was related to the hardware itself, as there was some instances where the code did not execute properly and the board itself needed to be reset. The other issue we faced was mostly around getting used to the software and the workflow in the Eclipse IDE and the debugger. Once we understood the ways to use the tools we found our workflow sped up considerably, as we were able to check step by step and find bugs at the source. The last issue we had was with the syntax of the code, but that was solved quickly by reading over documentation and with the help of the TAs. Overall the lab went smoothly and has indeed succeeded at the goals of improving our familiarity and skill with the Netburner ColdFire system, Assembly code and Pair Programming practices.

6 Appendix

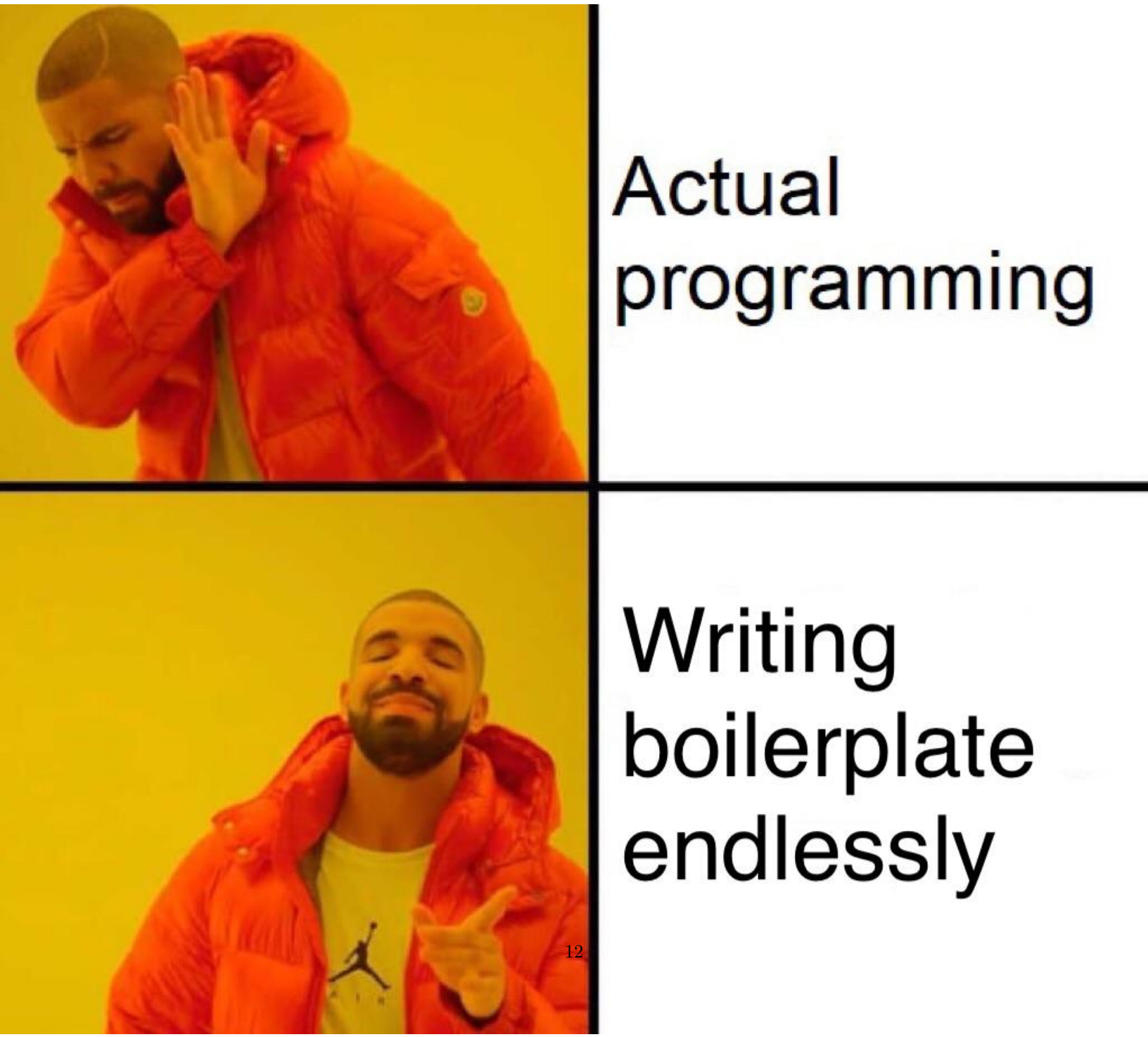
6.1 Part A Assembler Code

```
/* DO NOT MODIFY THIS ------------------------------------------------------*/
.text

.global AssemblyProgram

AssemblyProgram:
lea       -40(%a7),%a7 /*Backing up data and address registers */
movem.l %d2-%d7/%a2-%a5,(%a7)
/*------------------------------------------------------------------------*/

/************************************************************/
/* General Information ******************************************/
/* File Name: Lab2a.s ***************************************/
/* Names of Students: Arun Woosaree and Navras Kamal          **/
/* Date: February 12, 2018                                    **/
/* General Description:                                       **/
/*                                                            **/
/************************************************************/

/*Write your program here*************************************/

/*
1.  0x2300000 - Size of array
2.  0x2300004 - address of first array
3.  0x2300008 - address of second array
4.  0x230000C - address of where to store the sum with Register Indirect With Offset
5.  0x2300010 - address of where to store the sum with Indexed Register Indirect
6.  0x2300014 - address of where to store the sum with Postincrement Register Indirect
*/

/*Part A ***************************************************/
/* adding with register indirect with offset*/

lea 0x230000C, %a1 /* location of the array where result will be stored */
lea 0x2300004, %a2 /* location of the array where address of first array */
lea 0x2300008, %a3 /* location of the array where address of second array */
move.l (%a1), %a1
move.l (%a2), %a2
move.l (%a3), %a3

move.l (%a2), %d3
add.l (%a3), %d3
```

```
move.l %d3, (%a1)

move.l (0x4,%a2), %d3
add.l (0x4,%a3), %d3
move.l %d3, (0x4,%a1)

move.l (0x8,%a2), %d3
add.l (0x8,%a3), %d3
move.l %d3, (0x8,%a1)



/*Part B ******************************************************/
/* indexed register indirect */


lea 0x2300010, %a1 /* where result will be stored */
lea 0x2300004, %a2 /* address of first array */
lea 0x2300008, %a3 /* address of second array */
move.l (%a1), %a1
move.l (%a2), %a2
move.l (%a3), %a3

clr.l %d1 /* offset */
clr.l %d2 /* counter starts at 0  */

loopb:

move.l (%d1,%a2), %d3
add.l (%d1,%a3), %d3
move.l %d3, (%d1,%a1)

add.l #4, %d1 /* increment offset */
add.l #1, %d2 /* increment counter */
cmp.l 0x2300000, %d2
blt loopb



/*Part C ******************************************************/
/* post increment register indirect*/


lea 0x2300014, %a1 /* where result will be stored */
lea 0x2300004, %a2 /* address of first array */
```

```
lea 0x2300008, %a3 /* address of second array */
move.l (%a1), %a1
move.l (%a2), %a2
move.l (%a3), %a3

clr.l %d2 /* counter starts at 0  */

loopc:

move.l (%a2)+, %d3
add.l (%a3)+, %d3
move.l %d3, (%a1)+

add.l #1, %d2 /* increment counter */
cmp.l 0x2300000, %d2
bge end
bra loopc




/*End of program ************************************************/
end:
/* DO NOT MODIFY THIS ------------------------------------------------------*/
movem.l (%a7),%d2-%d7/%a2-%a5 /*Restore data and address registers */
lea       40(%a7),%a7
rts
/*------------------------------------------------------------------------*/
```

6.2 Part A Flowchart Diagram



6.3 Part B Assembler Code

```
/* DO NOT MODIFY THIS ------------------------------------------------------*/
.text

.global AssemblyProgram

AssemblyProgram:
lea       -40(%a7),%a7 /*Backing up data and address registers */
movem.l %d2-%d7/%a2-%a5,(%a7)
/*------------------------------------------------------------------------*/

/*************************************************************/
/* General Information ***************************************/
/* File Name: Lab2.s *****************************************/
/* Names of Students: Arun Woosaree and Navras Kamal          **/
/* Date: February 12, 2018                                    **/
/* General Description:                                       **/
/*                                                            **/
/*************************************************************/

/*Write your program here*************************************/


move.l 0x2300000, %d0 /*total # of points*/
sub.l #1, %d0
move.l 0x2300004, %a0 /*address of x points*/
move.l 0x2300008, %a1 /*address of y points*/
move.l 0x2300010, %a2 /*output final answer here*/
clr.l %d1 /*reset counter*/
clr.l %d7 /*reset total area*/

move.l (%a0)+, %d2 /*store initial x0*/
move.l (%a1)+, %d3 /*store initial y0*/

mainloop:
cmp.l %d0, %d1 /*check if all points have been used*/
beq endloop /*if they have been, end*/
add.l #1, %d1 /*otherwise increment counter variable*/

move.l (%a0)+, %d4 /*store x1*/
move.l (%a1)+, %d5 /*store y1*/

add.l %d5, %d3 /*a+b*/

sub.l %d4, %d2 /*h*/
```

```
neg.l %d2
cmp.l #2, %d2 /*is h 2??*/
bne skp
asl.l #1, %d3 /*multiply by 2 if it is*/

skp:
add.l %d3, %d7 /*add result to final answer*/
move.l %d4, %d2 /*set x0 to current x1*/
move.l %d5, %d3 /*set y0 to current y1*/

bra mainloop /*restart mainloop*/

endloop:
/*
move.l %d7, %d6
and.l #0b1, %d6
cmpl #0, %d6
bne skppy
add.l #1, %d7
*/
skppy:
asr.l #1, %d7
move.l %d7, (%a2)
/*End of program ************************************************/

/* DO NOT MODIFY THIS ------------------------------------------------*/
movem.l (%a7),%d2-%d7/%a2-%a5 /*Restore data and address registers */
lea       40(%a7),%a7
rts
/*-----------------------------------------------------------------------*/
```

6.4 Part B Flowchart Diagram

7 Marking Sheet