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

The purpose of this lab is to learn and apply multiple methods of indirect register addressing using the ColdFire Assembly language in a hands on environment. This will go over the methods of Register Indirect with Offeset, Indexed Register Indirect and Postincrement Register Indirect.

Register Indirect with Offset is written in the form of (offset, %address), where the offset is a constant value. This will cause the (address + offset) memory location to be accessed instead of the basic addressed location. This operation does not modify the value stored in the address register. Indexed Register Indirect is written as (%offset, %address) where the offset is a data register. Similar to Register Indirect with Offset this does not modify the address register and points to location (offset + address), but in this case the offset is a data register, thus the offset can vary depending on the contents of the data register. This is useful to reduce redundancy or to handle loops of unknown or variable number of iterations. Finally, there is Postincrement Register Indirect, which accesses the location at (address), then afterwards will increment address by the size of the operand. Therefore, a longword operation will increase address by 4, a word by 2 and a byte by 1. This is more useful when iterating over an array of unknown or variable size, when you do not need to preserve the value of the original address, and when you are handling values of constant size.

This lab will be split into two sections. The first part will entail the demonstration of the above three methods of indirect registry accessing. This will be done with a simple example in which there is a pair of arrays of values. The nth element of the first array will need to be added with the nth element of the second array, and their sum stored in the nth element of a third array designated for outputting the values. This process will be completed a total of three times, once with each of the forms of registry accessing above.

The second part of the lab will involve an example in which the use of these indirect registering will be tested. In this case it will be demonstrated by performing a Trapezoidal Approximation of a curve. A Trapezoidal Approximation is a way of estimating the area under a curve, or the Integral of the function represented by the curve. This example will focus solely on the two dimensional case. It will take an array of points in the x axis, and a second array of points in the y axis. These values are formed into Cartesian Coordinates by pairing up the nth term of the first array with the n^{th}term of the second, creating an (x,y) coordinate pair. This represents the n^{th} point on the graph.

These can be used alongside the formula for the area under a trapezoid, which is $A = h \cdot (\frac{a+b}{2})$. In this formula it is assuming a trapezoid with parallel top and bottom of any length (a and b respectively), and a distance between them of h. It takes the average between the two parallel lengths and multiplies them by the distance between them to find the total area enclosed. In our example, this equation and the definitions of the variables will need to be modified somewhat. Instead, we will be rotating the system 90°, such that h does not represent the height, but rather the horizontal distance along the x axis between the two parallel lines. So the values of a and b will instead be two different y coordinates on the graph, and h will be the absolute value of the difference between their respective x coordinates, or Δx. By assuming that the x coordinate of b is greater than that of a it is possible to rewrite the equation in terms of two different Cartesian coordinates representing two distinct places on the graph. For our calculations it will be easier to divide at the end of the overall calculation. Thus, the new equation is $A = \frac{(x_1 - x_0) \cdot (y_0 + y_1)}{2}$. Using this formula it will be possible to gain an approximation of the area under the graph, assuming that the distance in the x axis between the points is kept small.

Thus, by using the above formula it can be seen how it is possible to use a series of cartesian coordinates to calculate an approximation of the integral of a two dimensional function. These trapezoidal calculations will need to be added together then output to a specific memory location, and in order to iterate through the lists of x and y values, indirect registering is the most effective way to handle the values. This will demonstrate an understanding of the use of Register Indirect methods.

Many of the concepts in this lab build off of ideas from the previous two, including the use and debugging of the ColdFire system from lab 0 and the application of functions such as loops, as explored in lab 1.

2 Design

2.1 Part A: Different Addressing Modes

For part A of this lab, we wrote code to add the contents of two arrays stored at different memory locations, and the result is stored at another specified memory location. This is done using three different addressing modes. In the first part, we use Register Indirect With Offset, in the second part, we use Indexed Register Indirect, and in the third part, we use Postincrement Register Indirect. All information is stored as a long word (32 bits). The operational code is provided at memory location 0x2300000, which contains the information for where to access the arrays in memory and where to store the sum. For example, the number stored at 0x2300000 contains the size of the arrays. The operational code is stored as follows:

1. 0x2300000 - Size of arrays
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

The *SetZeros.s* and the *DataStorage.s* files, which were provided, were used to initialize memory contents.

2.1.1 Part 1: Register Indirect With Offset

We chose address register a1 to store the contents at 0x230000C, which is where the address to output the sum is located. This was done by loading the effective address into a1, and moving the contents of the memory location stored at a1 into a1 itself. In a similar fashion, address register a2 was chosen to store the contents at 0x2300004, which stores the starting address of the first array to be added. Similarly, address register a3 was chosen to store the location of where the second array to be added starts, and that memory address is stored at 0x2300008. To add the first two numbers in the arrays, the contents of the memory location pointed to by a2 were moved into data register d3, and the contents of the memory location pointed to by a3 were added to d3. The result

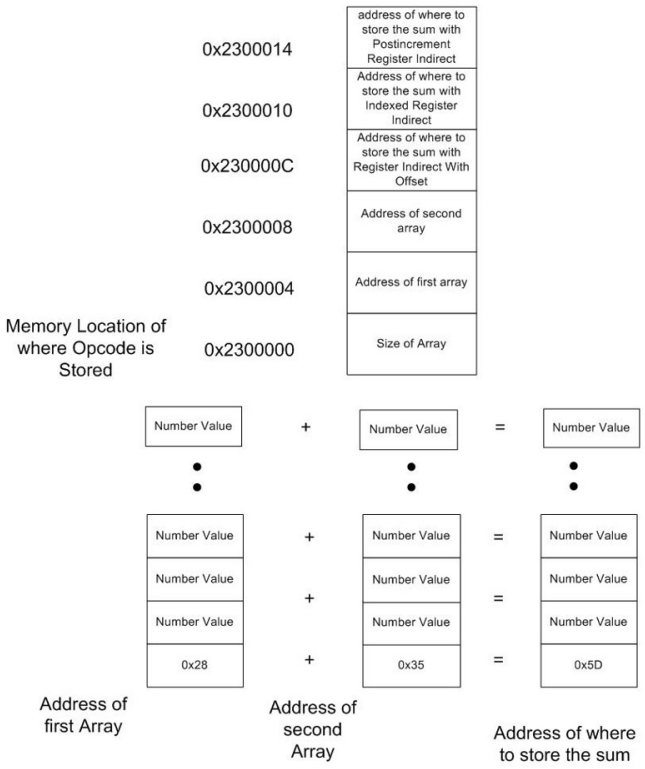


Figure 1: An illustration of how the data is organized in part A

in d3 was then stored at the memory location pointed to by a1. Next, these three lines of code were copy-pasted and modified to add offsets to the memory locations. So, to add the next two numbers in each array, the contents of the memory location (4,%a2) was moved to d3, the contents of the memory location (4,%a3) was added to d3, and the result in d3 was moved to the memory location (4,%a1). The same method was applied to add the third numbers in the arrays, but this time with an offset of 8.

2.1.2 Part 2: Indexed Register Indirect

The loading of the memory addresses in part 2 was done very similarly to part 1. Address registers a2 and a3 still contain the memory locations of where the first and second array begin. However, the contents at memory location 0x2300010 was moved to a1, which is where the result is stored for this part. Data register d1 was chosen to store the amount by which we want to offset, and d2 was chosen to be our counter variable for a loop. Additionally, both these data registers were cleared. Inside the loop, the contents of the memory location stored by a2 were moved to d3, just like in part 1. Except this time, it was offset by the number in d1. So, on the first loop iteration, it would be displaced by 0. Similarly, the contents of the memory location pointed to by a3, displaced by the value at d1, is added to d3. The result stored in d3 is then moved to the memory location a1,

also displaced by the number stored in d1. At the end of the loop, we increment the offset d1 by 4, and the counter is incrmented by 1. Then, we compare our counter to the size of the arrays, which is a value stored at 0x2300000, and if our counter is less than the size of the array, we continue the loop.

2.1.3 Part 3: Postincrement Register Indirect

Loading of the memory addresses in part 3 was also very similar to parts 1 and 2. Address registers a2 and a3 are unchanged, but the contents at memory location 0x2300014 was moved to address register a1, which is where the result is stored in this part. We chose data register d2 to be our counter variable, which was initially cleared with the value of 0. Just like for part 2, we have a loop, but this time there are no offsets. Instead, we move the contents of the memory location pointed to by a2 to d3, and postincrement a2. Similarly, the contents fo the memory location pointed to by a3 is moved to d3, and a3 is post incremented. At the end of the loop, the counter is incremented by 1, and if it is less than the size of the arrays (stored at 0x2300000), the loop continues.

2.2 Part B: Trapezoidal Rule

In part B of this lab,we write a program that calculates the area underneath a curve ($y = f(x)$), using the trapezoidal rule:

$$\int_a^b f(x)dx \approx \sum_{k=1}^{N} \frac{f(x_{k-1}) + f(x_k)}{2} \Delta x_k$$

The x and y data points for the curve are stored in arrays, at different memory addresses. For convenience, the distance between each X data point is either one or 2 units. Similar to part A, we have some operational code stored at 0x2300000, which contains the information for where to access the arrays in memory and where to store the output. The operational code is as follows:

1. 0x2300000 - Number of Data Points
2. 0x2300004 - Address of where the X data points are stored
3. 0x2300008 - Address of where the Y data points are stored
4. 0x230000C - Address of temporary storage space
5. 0x2300010 - Address of where to store the final output

In our initialization, we chose d0 to store the total number of data points, by moving the contents at 0x2300000 to d0, then subtracting 1 from d0 to avoid an off-by-one indexing error. a0 was chosen to store the address of the array with X data points, so the contents at 0x2300004 were moved to a0. Similarly, the contents at 0x2300010 were moved to a2, to store the address of the array with Y data points. Data register d1 was used as a counter, and d7 stores the total area. Both of these data registers were cleared with initial value 0. Initial data points x_0 and y_0 were stored in d2 and d3, respectively. Once our initialization is done, we enter a main loop, where we decided to do our checks for exiting the loop at the beginning. The loop exits if our counter (d0) is equal to the number of points (d1), otherwise the counter is incremented. In the loop, the

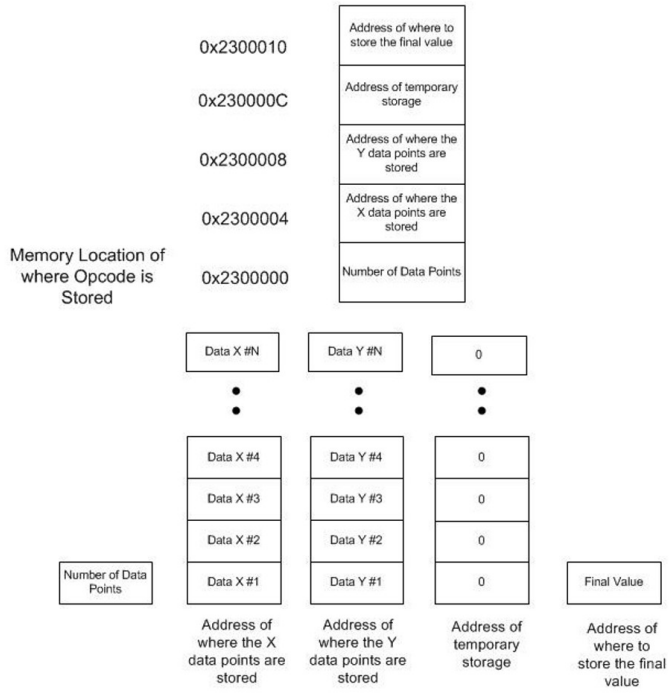


Figure 2: An illustration of how the data is organized in part B

contents of the memory address pointed to by a0 is moved to d4, and post incremented, so that d4 holds x_k. Similarly, the contents of the memory address pointed to by a1 is moved to d5, and post incremented, so that d5 holds $f(x_k)$. We get $f(x_{k-1}) + f(x_k)$ by adding d5 to d3 and storing the result in d3. Then, Δx is calculated by subtracting d4 from d2, storing the negation of that result in d2. If Δx was 2, we multiplied d3, which was $f(x_{k-1}) + f(x_k)$ by 2 and then added the result to d7. If Δx was not 2, we just added $f(x_{k-1}) + f(x_k)$ to d7. At the end of the loop, x_k and $f(x_k)$ from the current iteration are set to be x_{k-1} $f(x_{k-1})$ respectively, for the next iteration by moving the contents of d4 to d2 and also moving the contents of d5 to d3. When the loop terminates, we check if the number in d7 is odd by checking its least significant bit. If it is odd, we add one to it. Finally, the number in d7 is divided by 2 using a right bit shift operation, and the result is moved into the memory location pointed to by a2, which is where the ourput should go. We check if the number in d7 is odd before dividing it by 2, so that our program would round the number up if we get a fraction, as per the assignment specifications.

3 Testing

3.1 Part A

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. Initially, for part 3, we had an error where the sum calculated in the output array was all zeros. However, we later discovered that while exiting our loop in part 2, we jumped to the end of the program, so part 3 would never run. When the bugs were ironed out, After we fixed this minor bug, we went on to the next phase of testing. Our code was tested using the provided *Lab2aTest.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, who further tested our code by modifying the DataStorage file and verifying the output.

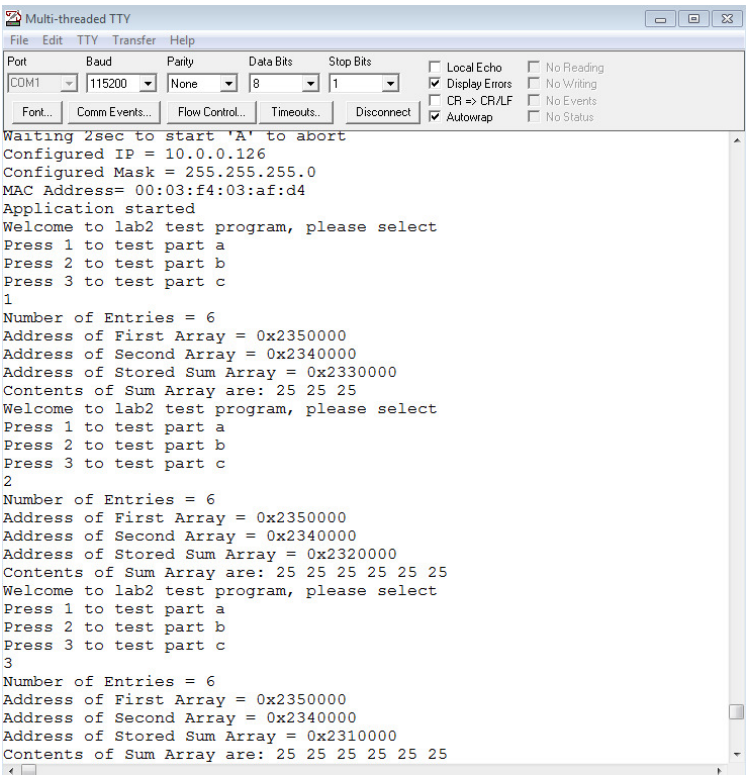


Figure 3: MTTTY output when testing our Part A solution

3.2 Part B

Although the parts were different, the procedure for testing our code for part B was 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 monitored relevant memory addresses and registers. We initially got a huge number, above 63 million, which we correctly identified as an off-by-one indexing error. This problem was rectified by subtracting one from d0, which is what we used to store the total number of points. We also included some code towards the end which would make

sure our code rounds up as per the lab specifications if we got a fraction instead of a whole number when calculating the area (as verified by a lab TA). However, this code seemed to have no effect, since our program produced the correct output anyways. Our program still worked with this code, however as it was not fully tested we left it commented out in our submission. After review we have seen that there is a more elegant way to reach the same solution using bit tests but we have left it as it was at the time that it was reviewed by the TAs in lab. In any case, we used the provided files *Lab2bTest.s*, and the *Datastorage5.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. As per the lab instructions, we used *DataStorage5.s* and *SetZeros5.s* files, although we did further test our program using *DataStorage4.s* with a lab TA and found no issues.

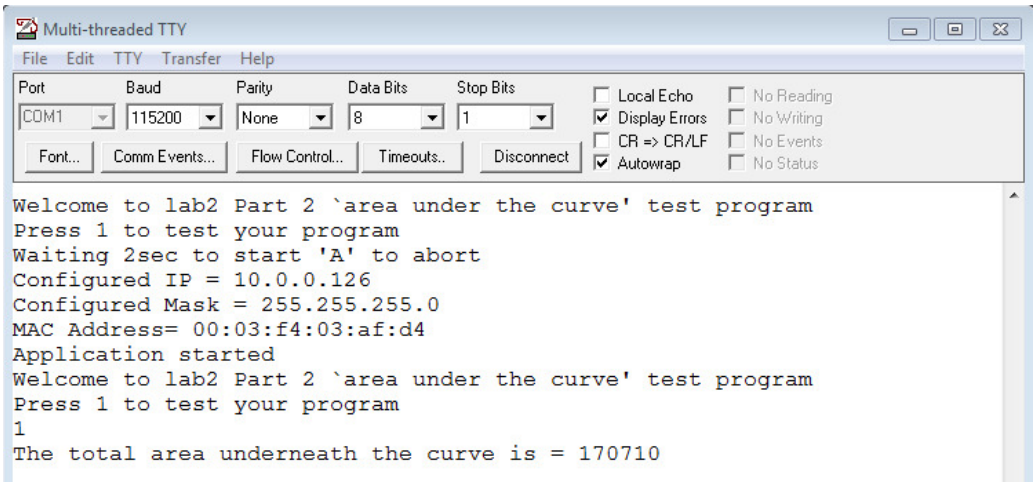


Figure 4: MTTTY output when testing our Part B solution with *DataStorage5.s*

4 Questions

1. *What are the advantages of using the different addressing modes covered in this lab?*
   (a) Register Indirect With Offset
      i. Great if you know the position of data relative to some memory location
      ii. Doesn't modify the address register's value
      iii. Can access any specific memory address you want relative to the value stored in the address register
   (b) Indexed Register Indirect
      i. Doesn't modify the address register's value
      ii. You can offset in terms of a variable, so you have greater control over the offset
      iii. Works well in loops in order to access multiple sequential memory locations while being able to easily return to the original address as the value in the address register is never changed

(c) Postincrement Register Indirect

i. You don't have to keep track of the offset, it automatically increments the address register by the size of the data you're working with

ii. Very useful for dealing with arrays, as in part B

iii. Works nicely in loops to access multiple sequential memory locations and be pointing to the next memory location afterwards

2. *If the differece between the X data points are not restricted to be either one or two units, how would you modify your program to calculate the area? You do not need to do this in your code.*
A: Instead of hard coding two scenarios where Δx is either 1 or 2, we could just divide $f(x_{k-1}) + f(x_k)$ by 2 with a bit shift operation to the right by one at each iteration rather than at the end, and then use the *muls.l* instruction to multiply the result by Δx to match the equation for the trapezoidal rule.

3. *From the data points, what is the function (y=f(x))? What is the percent error between the theoretical calculated area and the one obtain in your program?*

From the data points, we can tell the function is $y = x^2$.
The value obtained in our program was **170710**.
The actual value should be:

$$\int_0^{80} x^2 dx = \frac{x^3}{3}\Big|_0^{80} = \frac{80^3}{3} = \frac{512000}{3} = \mathbf{170666.666...}$$

Therefore, the percent error is:

$$\frac{|\frac{512000}{3} - 170710|}{\frac{512000}{3}} \times 100 = \frac{13}{512}\% = \mathbf{0.025390625\%}$$

5 Conclusion

The first part of this lab demonstrated how to use multiple forms of accessing registers indirectly and their individual benefits. The forms of indirect access explored in this lab were Register Indirect with Offset, Indexed Register Indirect and Postincrement Register Indirect. The second part of this lab was a practical demonstration which could be solved using the aforementioned methods. In this example a graph is stored as (x,y) pairs, with each half of the pair being an element in one of two different arrays. The issues we experienced were mostly user error, such as accidentally skipping over portions of the code, however in testing we ended up having to reflash the board relatively frequently, which slowed the pace of testing somewhat. However, once we got the feel for the testing and with some assistance from the TAs we were able to resolve our issues. Overall the lab went smoothly, and has indeed succeeded at it's goal of furthering our knowledge of the various methods of referencing memory locations.

6 Appendix

6.1 Part A Assembler Code

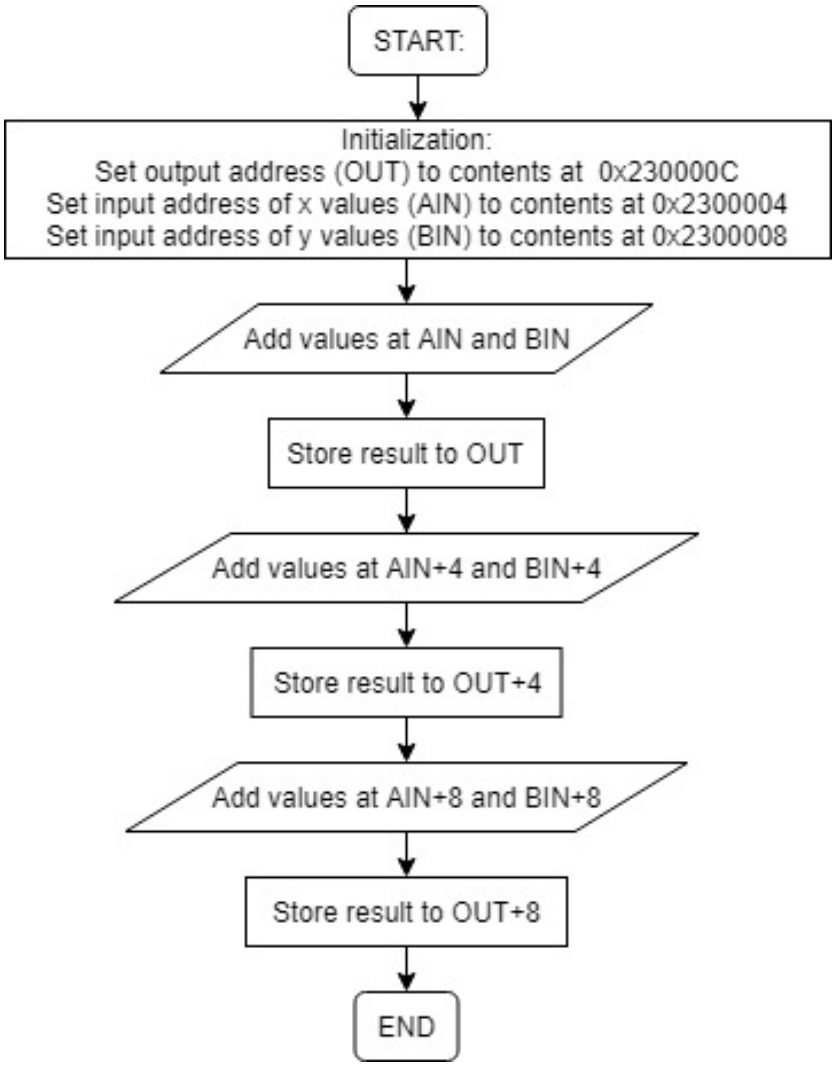
```
/* DO NOT MODIFY THIS ------------------------------------------------------*/
.text

.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  /* Set a1 to the location where the result is to be held */
move.l (%a2), %a2  /* Set a2 to the location where the first array begins */
move.l (%a3), %a3  /* Set a3 to the locaiton where the second array begins */
```

```
move.l (%a2), %d3 /* Move the first element in the first array to
                     data register d3 */
add.l (%a3), %d3  /* Add the first element in the second array to
                     data register d3 */
move.l %d3, (%a1) /* Move the sum to the first location of the result array */

move.l (0x4,%a2), %d3 /* Move the second element in the first array to d3 */
add.l (0x4,%a3), %d3  /* Add second element in second array to d3 */
move.l %d3, (0x4,%a1) /* Move sum to second location of result array */

move.l (0x8,%a2), %d3 /* Move the third element in the first array to d3 */
add.l (0x8,%a3), %d3  /* Move the third element in the second array to d3 */
move.l %d3, (0x8,%a1) /* Move sum to third location of result array */

/* Continue to the next part */



/*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  /* Set a1 to the location of the result array */
move.l (%a2), %a2  /* Set a2 to the locaiton of the first array  */
move.l (%a3), %a3  /* Set a3 to the locaiton of the second array */

clr.l %d1 /* Reset offset amount to 0     */
clr.l %d2 /* Reset counter to 0           */

loopb:    /* Beginning of loop for part b */

/* At the end of each loop, increments the offset value by 4*n where n is the
number of iterations (0 <= n < [number of elements in array]) */

move.l (%d1,%a2), %d3  /* Move the nth element of the first array to d3 */
add.l (%d1,%a3), %d3   /* Add the nth element of the second array to d3 */
move.l %d3, (%d1,%a1)  /* Move sum to the nth element of the output array */

add.l #4, %d1         /* Increment offset amount by 4 */
add.l #1, %d2         /* Increment counter value by 1 */
cmp.l 0x2300000, %d2 /* Compare counter to # of elements in the array */
blt loopb             /* If it is less than # of elements, repeat the loop */
```

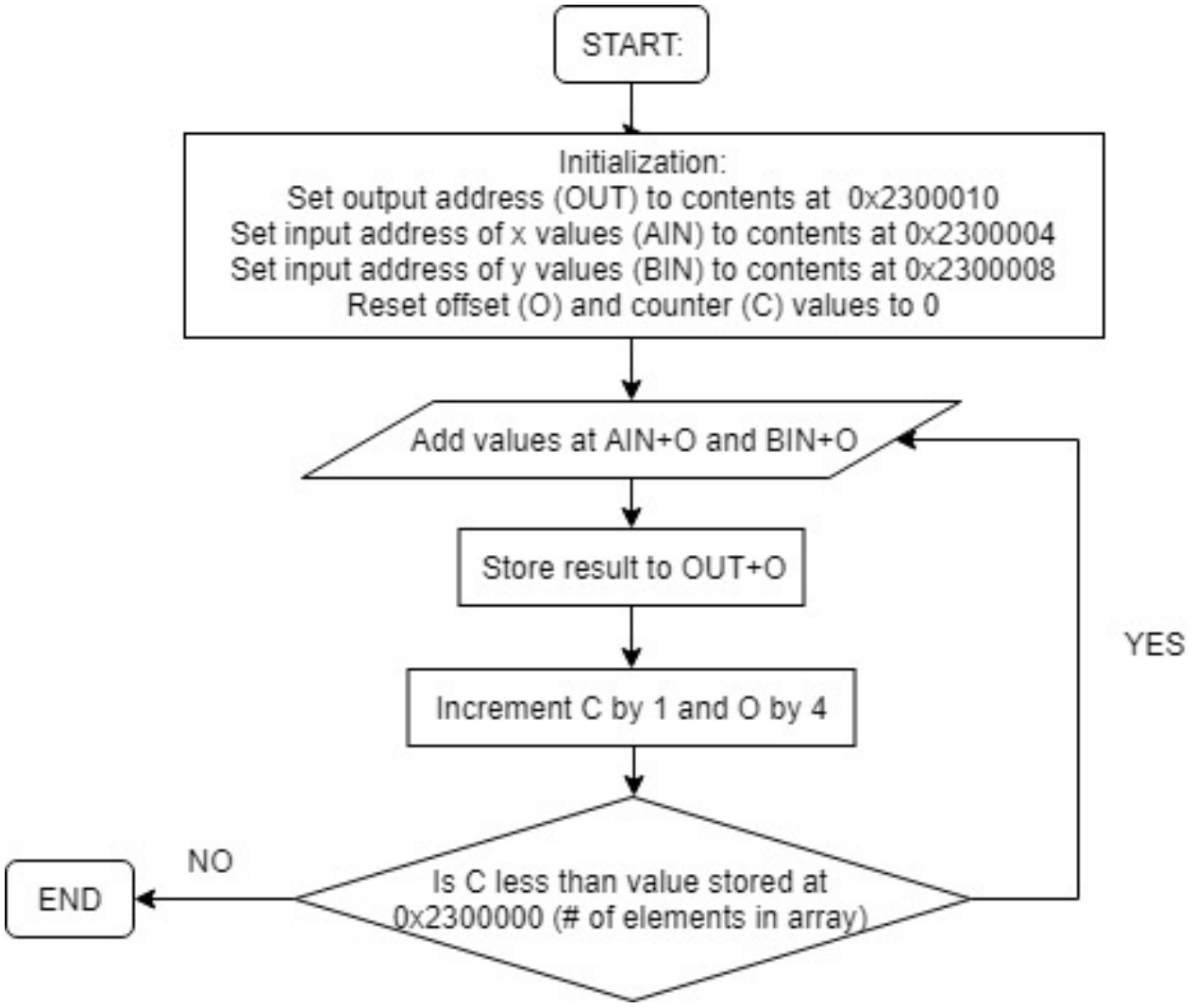
```
                    /* Otherwise continue to the next part */



/*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  /* Set address to location of result array */
move.l (%a2), %a2  /* Set address to location of first array  */
move.l (%a3), %a3  /* Set address to location of second array */

clr.l %d2 /* Reset counter to 0          */

loopc:    /* Beginning of loop for part c */

move.l (%a2)+, %d3 /* Move the value at the location pointed at by a2 to d3,
                      then increment the address in a2 by the size of
                      a long word (4) */
add.l (%a3)+, %d3  /* Add the value at the location pointed at by a3, then
                      increment the address in a3 by the size of a long word*/
move.l %d3, (%a1)+ /* Move the sum to the location pointed at by a1, then
                      increment the address in a1 by the size of a long word*/

add.l #1, %d2         /* Increment counter by 1 */
cmp.l 0x2300000, %d2 /* Compare the counter to the number of elements */
bge end               /* If the counter is greater than or equal to the number
                         of elements, quit the loop and continue
                         to the next part */
bra loopc             /* Otherwise repeat the loop */




/*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 Diagrams

6.2.1 Part 1: Register Indirect With Offset



6.2.2 Part 2: Indexed Register Indirect



6.2.3 Part 3: Postincrement Register Indirect

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 /* Stores total number of points to d0 */
sub.l #1, %d0         /* Subtracts one from this value, as one point will be
                         used in initialization */
move.l 0x2300004, %a0 /* Stores the location of the first x element in a0 */
move.l 0x2300008, %a1 /* Stores the location of the first y element in a1 */
move.l 0x2300010, %a2 /* Stores output location ot a2 */

clr.l %d1 /* Resets counter variable to 0 */
clr.l %d7 /* Resets total area to 0       */

/* The area of the trapezoid will be calculated with the formula
((x1-x0)*(y0+y1))/2 */

move.l (%a0)+, %d2 /* Move the first x0 to d2, then increment a0 by 4 */
move.l (%a1)+, %d3 /* Move the first y0 to d3, then increment a1 by 4 */

mainloop:      /* The main loop of this function       */
cmp.l %d0, %d1 /* Check if all points have been used   */
beq endloop    /* If they have been, end the loop      */
add.l #1, %d1  /* Otherwise increment the counter by 1 */

move.l (%a0)+, %d4 /* Store the next x1 into d4, then increment a0 by 4 */
```

```
move.l (%a1)+, %d5 /* Store the next y1 into d5, then increment a1 by 4 */

add.l %d5, %d3  /* y0 + y1, overwriting the value in d3 */
sub.l %d4, %d2  /* x0 - x1, overwriting the value in d2 */
neg.l %d2       /* Changes the result of the above operation to x1 - x0,
                   stored in d2 */
cmp.l #2, %d2   /* The value of (x1-x0) can only be 1 or 2.  Compare
                   the value in d2 to 2 */
bne skp         /* If (x1-x0) is not 2 it is 1, so skip the below operation */
asl.l #1, %d3   /* If (x1-x0) is 2 multiply the value in d3 by 2 using an
                   arithmetic shift left by 1 */

skp:            /* Jump here to avoid executing the multiplication, or just
                   continue here naturally */
add.l %d3, %d7  /* Add the value stored in d3 to the total sum stored in d7 */
move.l %d4, %d2 /* Set current x1 to be used as the next x0, stored in d2 */
move.l %d5, %d3 /* Set current y1 to be used as the next y0, stored in d3 */

bra mainloop    /* Restart the main loop */

endloop: /* Go here once the above loop is complete */

/* The code below was a test to make sure the result rounded up.  It is not
tested and not refined.  At the request of the TAs it is here in the code but
commented out and will not run, as such it is not neccessarily part of our
final answer for this part of the lab */

/*
move.l %d7, %d6 /* Copy the final answer to d6 */
and.l #0b1, %d6 /* Reduce it to its least significant bit */
cmpl #0, %d6    /* Compare that bit to 0 */
bne skppy       /* If it is zero the number is even, so skip the next line */
add.l #1, %d7   /* If the value is odd then add 1 to the value in d7, this will
                   make it the next highest nearest number, which when divided
                   by one will be equivalent to rounding up the division
                   of the odd value */
skppy:          /* Code will continue here */
*/

asr.l #1, %d7     /* Divide the overall sum by 2, using an arethmetic shift
                     right by 1, completing the sum of trapezoidal areas */
move.l %d7, (%a2) /* Store the final answer at the appropriate location */
```

```
/*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