Introduction to Addressing Modes

Benjamin Kong — 1573684
Lora Ma ———— 1570935

ECE 212 Lab Section H11

March 8, 2020

Contents

1 Introduction

As we learned in class, assembly language stores data in memory based on addresses. In this lab, we will investigate several different ways to address memory that is stored in memory. We also experiment with the differences between reading and writing to memory using these different addressing modes.

In part A of the lab, we wrote a program that adds adjacent contents of two arrays stored at different memory locations using three different methods to access memory:

- Register Indirect With Offset,
- Indexed Register Indirect, and
- Postincrement Register.

The resulting array from adding the contents with each of the different addressing mode types are stored in three different locations before being output afterwards to the MTTY console. Note that for the first type of addressing mode (Register Indirect With Offset), we only perform the addition for the first 3 adjacent values to demonstrate that we understand this type of addressing.

In part B of the lab, we created a function that calculated the area underneath a curve given the data points using the trapezoidal rule. Using the data points stored in memory (x and y data points), it is mathematically trivial to calculate the area formed by the data points. Note that the distance between each x data point is either one, two, or four units.

2 Design

2.1 Part A

For part A, we designed three different methods in completing the simple task of taking two arrays with the same number of elements to form a new array. This new array contains the adjacent values from the two input arrays added together. For example, if we have an array A = [1, 2, 3] and an array B = [4, 5, 6], we would add the adjacent values and get a new array R = [1 + 4, 2 + 5, 3 + 6] = [5, 7, 9].

In part 1, we used the Register Indirect With Offset method. In this method, we took the first element in each array given and add them and store the results in the results array. Then, we shift the address of the given arrays to the next element by incrementing their addresses by 4. Then, we add the numbers at those addresses together, increment the address of the results array by 4 to the next empty address, and store the results in that address. Then, we shift the address of the give arrays by incrementing them by 8. Then, we add the numbers at those addresses together, increment the address of the results array by 8 to the next empty address, and store the results in that address.

In part 2, we set an initial offset of 0. Then, we check if the offset is equal to the size of the given arrays. If it is not equal, we set take the values of each array at the address of the first element plus the longword offset. Then we add the numbers and store it in the results array at the address of its first element plus the longword offset. Afterwards, we add one to the offset and the repeat the process until the offset is equal to the number of elements in the original array.

In part 3, we check if the size of the array is equal to zero. If it's not equal to zero, we take the current value pointed by the address for each array. These values are then added and then the array addresses are incremented by 1 long word. This value is stored in the results array pointed by its address and then the address is incremented by one long word. Afterwords, we subtract one from the array size and repeat the process.

2.2 Part B

In part B, we used the trapezoidal rule that allowed us to estimate the area under a curve.

$$A = \frac{\delta x}{2}(y_1 - y_2)$$

3 Testing

3.1 Part A

If properly implemented, part A should correctly add adjacent elements in two different arrays into a new resulting array, regardless of which of the three methods used. These methods were

- Register Indirect With Offset,
- Indexed Register Indirect, and
- Postincrement Register.

For example, given two arrays $A = [A_0, A_1, ...]$ and $B = [B_0, B_1, ...]$ to add, our resultant array would be $C = [A_0 + B_0, A_1 + B_1, ...]$. In order to test that our program functioned correctly, we ran our code against a test case (one of the DataStorage files) and compared our results to the expected results which were given to us by our TAs.

Upon testing our program, it was evident that our program worked since our output matched what was expected and all methods used to add the arrays resulted in the same resultant array. Our screenshot of the MTTY terminal output is included in the appendix.

3.2 Part B

If properly implemented, part B should correctly calculate the area under a curve given a set of data points with x and y coordinates. We use the trapezoidal rule to accomplish this: we sum up the area of the trapezoid between each point from the first point to the last point and achieve the area under the graph.

In order to determine if our program actually does this correctly, we ran our code against several test cases provided on eClass given in the form of DataStorage files and compared our output to the expected output provided to us.

Upon testing our program, it was evident that our program worked since the areas that we found for each test case corresponded to the expected output. Our screenshot of the MTTY terminal output is included in the appendix.

4 Questions

What are the advantages of using the different addressing modes covered in this lab?

Let "Register Indirect with Offset" be mode 1, "Indexed Register Indirect" be mode 2, and "Postincrement Register" be mode 3. An obvious advantage mode 2 and mode 3 have over mode 1 is that since we are essentially able to loop until the array is finished, we can input an array of any size we wish rather

than hard-coding each iteration of the loop. It is also significantly less tedious to implement mode 2 and mode 3 since significantly less code is needed when using a loop-like structure.

Upon comparing mode 2 and mode 3, the difference is less pronounced. Mode 2 is slightly more complicated than mode 3 since mode 2 requires the coder to keep track of the address offset using another variable, while mode 3 automatically increments the address to the next location in memory each time the loop executes, removing the need for the coder to keep track of the current offset.

If the difference between the X data points are not restricted to be either one, two, or four units, how would you modify your program to calculate the area?

From the data points, what is the function $y = f(x)$? What is the percent error between the theoretical calculated area and the one obtained in the program? Calculate all 3 functions.

From DataStorage4, it is evident that the function is $f(x) = x^2$, and the data points given indicate that the range we are interested in is $[0, 50]$. Therefore, the actual area under the graph is given by

$$A_1 = \int_0^{50} x^2 \, \mathrm{d}x = \frac{125000}{3} \approx 41666.666. \tag{1}$$

Our program output the value of 41675. Therefore, calculating error, we have

$$\epsilon_1 = \frac{41675 - 41666.666}{41666.666} \cdot 100\% = 0.02\%. \tag{2}$$

From DataStorage5, it is evident that the function is again $f(x) = x^2$, and the data points given indicate that the range is $[0, 80]$. Therefore, the actual area under the graph is given by

$$A_2 = \int_0^{80} x^2 \, \mathrm{d}x = \frac{512000}{3} \approx 170666.666. \tag{3}$$

Our program output the value of 170710. Therefore, calculating error, we have

$$\epsilon_2 = \frac{170710 - 170666.666}{170666.666} \cdot 100\% \approx 0.02539\%. \tag{4}$$

From DataStorage6, it is evident that the function is again $f(x) = x^2$, and the data points given indicate that the range is $[0, 110]$. Therefore, the actual area under the graph is given by

$$A_3 = \int_0^{110} x^2 \, \mathrm{d}x = \frac{1331000}{3} \approx 443666.666. \tag{5}$$

Our program output the value of 443843. Therefore, calculating error, we have

$$\epsilon_3 = \frac{443843 - 443666.666}{443666.666} \cdot 100\% \approx 0.03974\%. \tag{6}$$

5 Conclusion

g

6 Appendix

6.1 Part A MTTY Screenshots

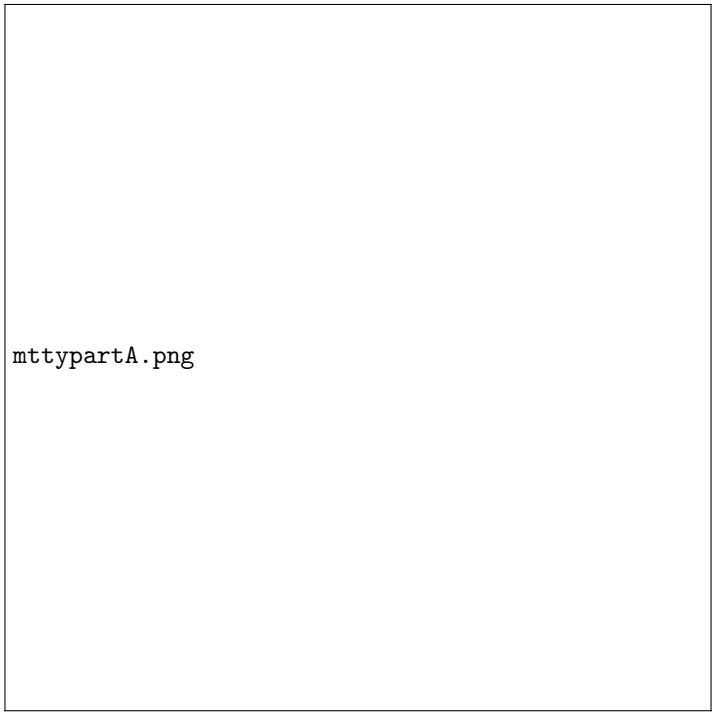

Figure 1: Screenshot of MTTY output for part A.

6.2 Part B MTTY Screenshots

Figure 2: Screenshot of MTTY output for part B.

6.3 Part A Assembler Code

```
/*Part A ********************************************************/
MOVEA.L #0x43000000, %a1
MOVE.L (%a1), %d3 /* d1 is the size of our array*/
MOVEA.L #0x43000004, %a1
MOVEA.L (%a1), %a2  /* address of first array */
MOVEA.L #0x43000008, %a1
MOVEA.L (%a1), %a3 /* address of second array */
MOVEA.L #0x4300000C, %a1
MOVEA.L (%a1), %a4 /* where to store adjacent sums */

MOVE.L (%a2), %d1 /* make a copy of first array value */
MOVE.L (%a3), %d2 /* make a copy of second array value */
ADD.L %d1, %d2 /* add first array value and second array value and put result into d2*/
MOVE.L %d2, (%a4) /* move added value into address at a4 */

MOVE.L 4(%a2), %d1 /*increment first array index and move new value into d1*/
MOVE.L 4(%a3), %d2 /*increment second array index and move new value into d2*/
ADD.L %d1, %d2 /*add values together*/
MOVE.L %d2, 4(%a4) /*put added value into incremented array a4*/

```

```
21  /*repeat above process*/
22  MOVE.L 8(%a2), %d1
23  MOVE.L 8(%a3), %d2
24  ADD.L %d1, %d2
25  MOVE.L %d2, 8(%a4)
26
27  /*Part B ********************************************************/
28
29  MOVE.L #0, %d2  /* Store 0 into d2*/
30
31  MOVEA.L #0x43000010, %a1
32  MOVEA.L (%a1), %a4 /*store address for result array*/
33
34  loop_partB:
35  CMP.L %d2, %d3 /*compare zero and d3 */
36  BEQ next /* exit part B*/
37
38  MOVE.L (%a2, %d2*4), %d1 /* add 4 to d2 and add to a2. Store value in d1 */
39  ADD.L (%a3, %d2*4), %d1 /* Add d2 with 4, a3 and d1. Store value in d1 */
40  MOVE.L %d1, (%a4, %d2*4) /* move the value of d1 into the value of d2+4+a4*/
41  ADDI.L #1, %d2 /* Add 1 to d2*/
42  BRA loop_partB /* loop */
43
44  /*Part C ********************************************************/
45
46  next:
47
48  MOVEA.L #0x43000014, %a1 /*intialize value of a1*/
49  MOVEA.L (%a1), %a4 /*initialize a4 with value at a1*/
50
51  loop_partC:
52  CMPI.L #0, %d3 /*compare d3 to 0*/
53  BEQ exit /* if equal, exit */
54
55  MOVE.L (%a2)+, %d1 /*put value of first array value into d1 and increment a3*/
56  ADD.L (%a3)+, %d1 /* add value in seond array to d1 and increment a3*/
57  MOVE.L %d1, (%a4)+ /* move value in d1 to array with results and increment the array */
58  SUBI.L #1, %d3 /*subtract 1 from d3*/
59  BRA loop_partC
60
61  exit:
62
63  /*End of program ************************************************/
```

6.4 Part B Assembler Code

```
/*Write your program here*****************************************/
MOVEA.L #0x43000000, %a1
MOVE.L (%a1), %d3 /* load value for data points at address a1 into d3*/

MOVEA.L #0x43000004, %a1
MOVEA.L (%a1), %a2 /*load array for x points in a2*/

MOVEA.L #0x43000008, %a1
MOVEA.L (%a1), %a3 /*load array for y points in a3*/

MOVEA.L #0x43000010, %a1
MOVEA.L (%a1), %a4 /*load results array in a4*/

CLR.L %d2 /*clear d2*/

loopVals:
CMPI.L #1, %d3 /* compare 1 to data point */
BEQ exit /* if equal, go to exit */

SUBI.L #1, %d3 /*reduce counter by 1*/

/* Load x vals, calculate delta x */
MOVE.L 4(%a2), %d0
SUB.L (%a2)+, %d0

/* Load y vals, calculate sum of y */
MOVE.L 4(%a3), %d1 /* increment y array and put new val into d1 */
ADD.L (%a3)+, %d1 /* post increment a3 after adding current a3 val to d1*/

loop_X:
CMPI.L #1, %d0 /*compare value 1 with d0*/
BEQ area /*if equal, calculate area*/
LSR.L #1, %d0 /*logical shift right by 1 in d0*/
LSL.L #1, %d1 /*logical shift left by 1 in d1*/
BRA loop_X

area:
ADD.L %d1, %d2 /*add d1 and d2*/
BRA loopVals

exit:
LSR.L #1, %d2 /* divide by 2 */
MOVE.L %d2, (%a4) /* store in results */
```

```
44
45  /*End of program ************************************************/
```