Introduction to Addressing Modes

Benjamin Kong — 1573684

Lora Ma ————— 1570935

ECE 212 Lab Section H11

March 7, 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

b

2.2 Part B

c

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

f

5 Conclusion

g

6 Appendix

6.1 Part A MTTY Screenshots

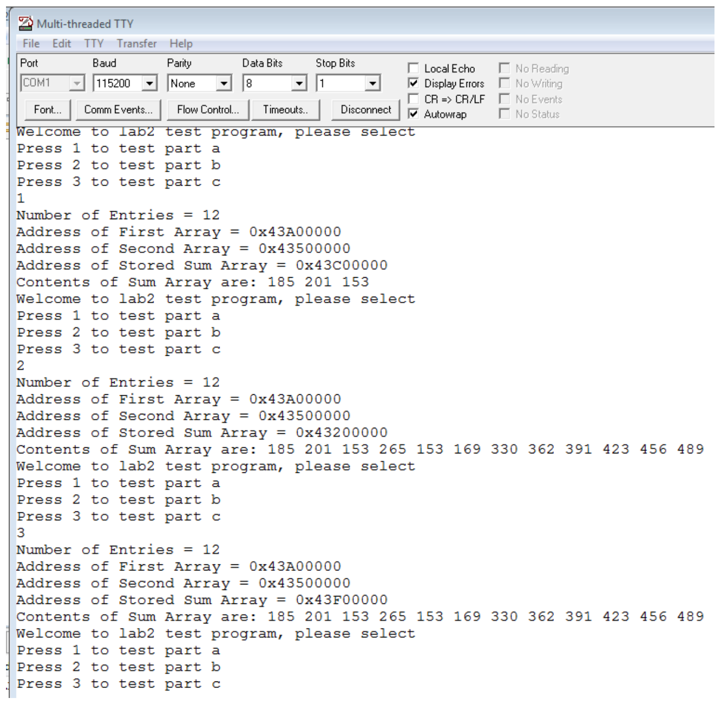

Figure 1: Screenshot of MTTY output for part A.

6.2 Part B MTTY Screenshots

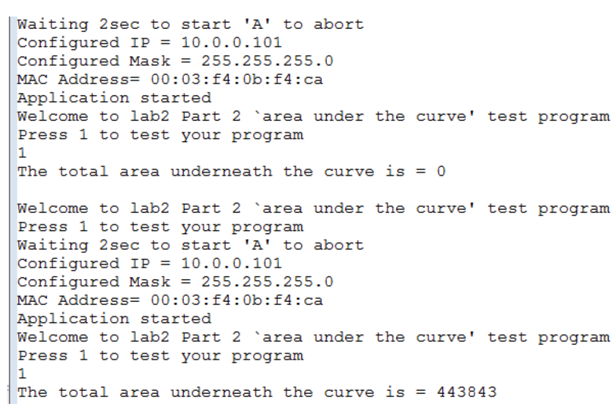

Figure 2: Screenshot of MTTY output for part B.

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*/

/*repeat above process*/
MOVE.L 8(%a2), %d1
MOVE.L 8(%a3), %d2
ADD.L %d1, %d2
MOVE.L %d2, 8(%a4)

/*Part B **********************************************************/

MOVE.L #0, %d2  /* Store 0 into d2*/

MOVEA.L #0x43000010, %a1
MOVEA.L (%a1), %a4 /*store address for result array*/

loop_partB:
CMP.L %d2, %d3 /*compare zero and d3 */
BEQ next /* exit part B*/

MOVE.L (%a2, %d2*4), %d1 /* add 4 to d2 and add to a2. Store value in d1 */
ADD.L (%a3, %d2*4), %d1 /* Add d2 with 4, a3 and d1. Store value in d1 */
MOVE.L %d1, (%a4, %d2*4) /* move the value of d1 into the value of d2+4+a4*/
ADDI.L #1, %d2 /* Add 1 to d2*/
BRA loop_partB /* loop */

```

```
/*Part C ************************************************************/

next:

MOVEA.L #0x43000014, %a1 /*intialize value of a1*/
MOVEA.L (%a1), %a4 /*initialize a4 with value at a1*/

loop_partC:
CMPI.L #0, %d3 /*compare d3 to 0*/
BEQ exit /* if equal, exit */

MOVE.L (%a2)+, %d1 /*put value of first array value into d1 and increment a3*/
ADD.L (%a3)+, %d1 /* add value in seond array to d1 and increment a3*/
MOVE.L %d1, (%a4)+ /* move value in d1 to array with results and increment the array */
SUBI.L #1, %d3 /*subtract 1 from d3*/
BRA loop_partC

exit:

/*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 */

/*End of program **************************************************/
```