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: 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

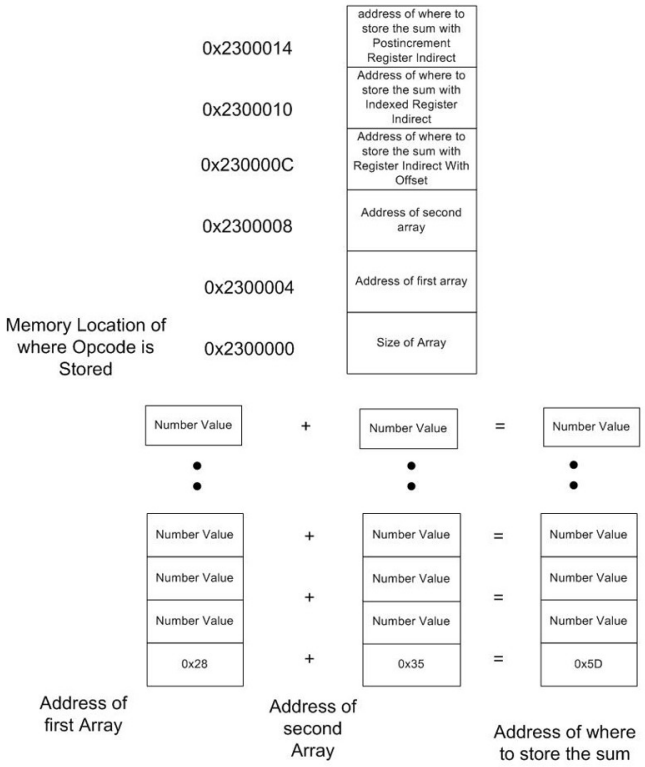


Figure 1: An illustration of how the data is organized in part A

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 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 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.

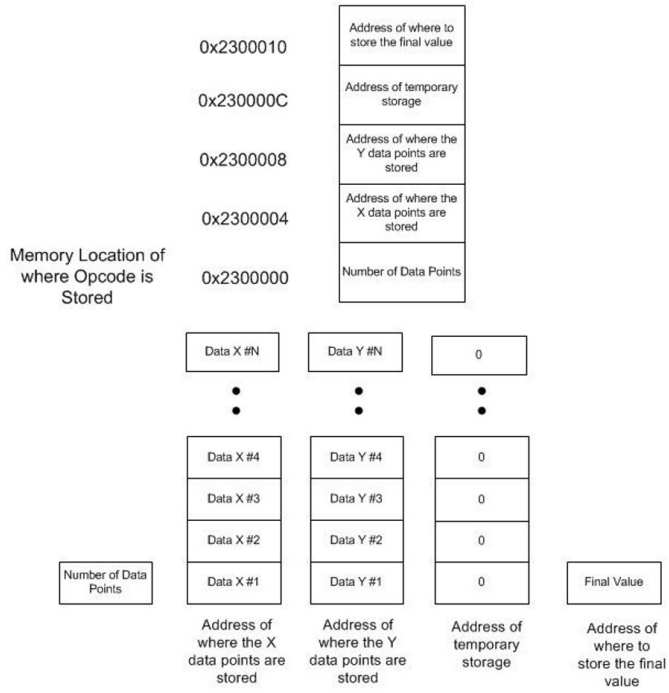


Figure 2: An illustration of how the data is organized in part B

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. 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.

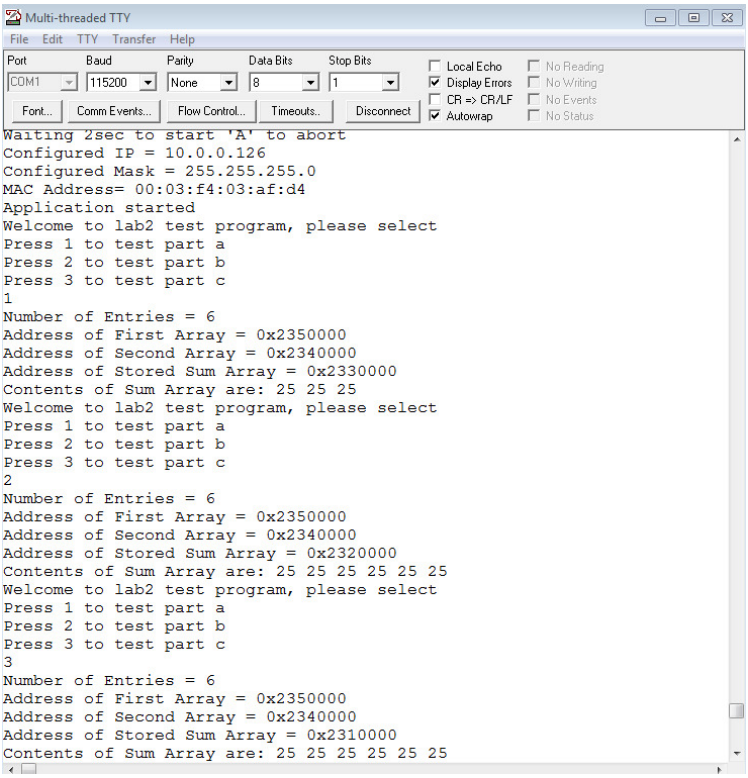


Figure 3: MTTTY output when testing our Part A solution

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, so we left it in since it does not change our output and served as a check. 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.

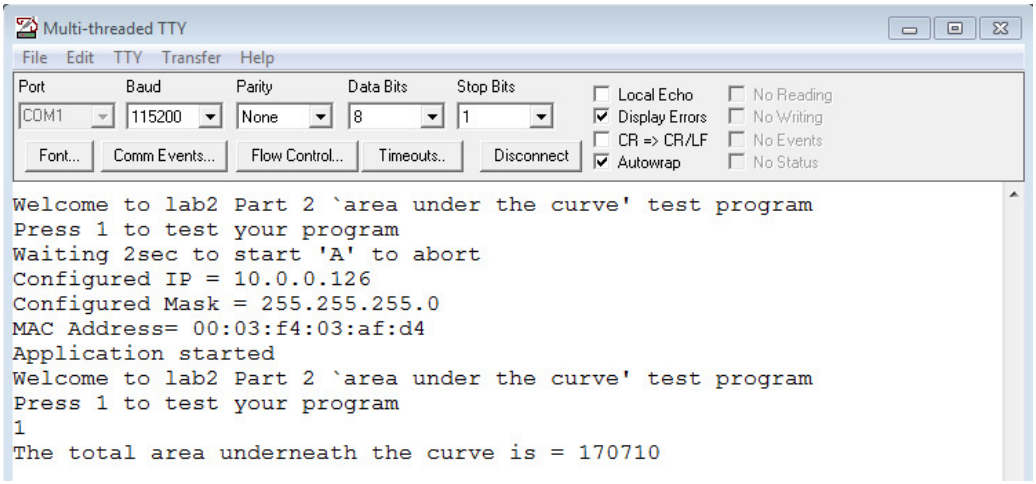


Figure 4: MTTTY output when testing our Part B solution with *DataStorage5.s*

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 only plan on accessing one thing
      ii. Doesn't modify the address register's value
      iii. Can access any 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

   (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
      iii. Works nicely in loops

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, 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

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

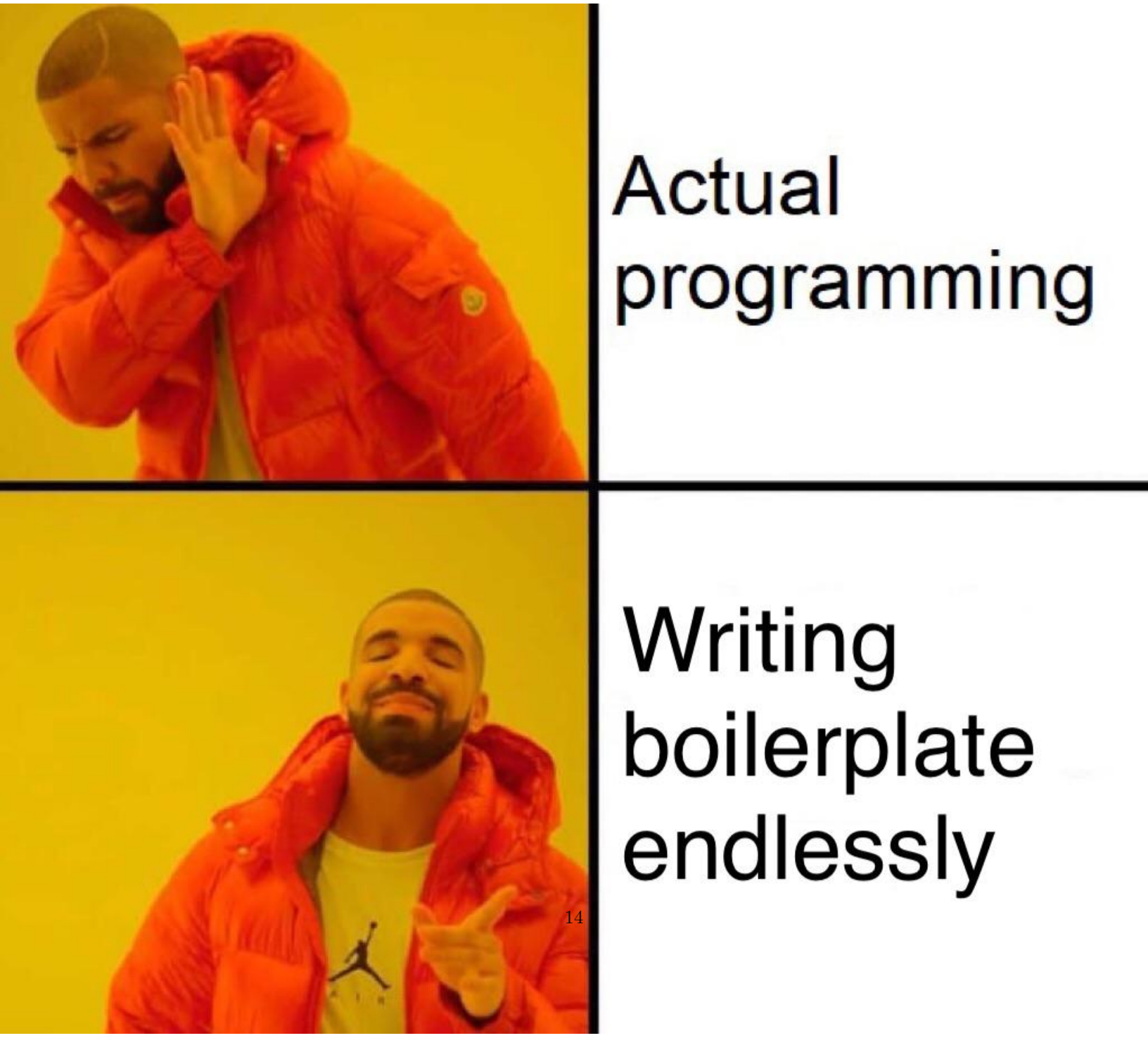
```
/* DO NOT MODIFY THIS ------------------------------------------------------*/
.text

.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 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 */
```

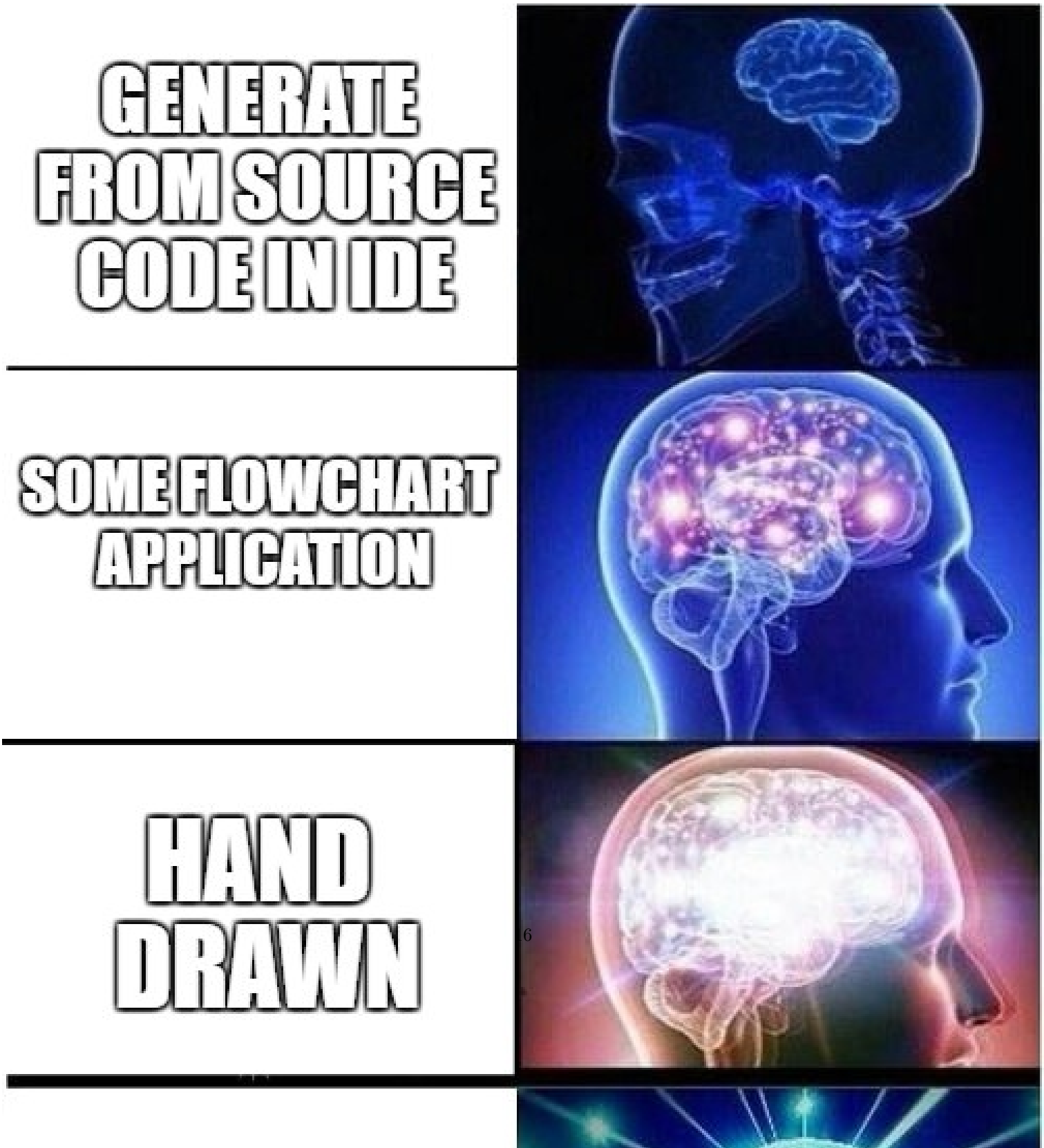
```
lea 0x2300008, %a3 /* address of second array */
move.l (%a1), %a1
move.l (%a2), %a2
move.l (%a3), %a3

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 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 /*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