INTRODUCTION

In this lab, we worked on different addressing modes in assembly language using the THUMB2 instruction set. Addressing modes are used to access and manipulate data in memory locations. We used in total three addressing modes in this lab – Register Indirect with offset, Indexed Register Indirect, and Post-Increment Register Indirect. In the first part of the lab, we were instructed to add elements from two arrays and store the results in a different place using addressing modes. We used the Register Indirect with offset and defined two registers, one as base register and another one with an offset to specific memory locations. After that we used an indexed register indirect to implement dynamic memory allocation. Finally, we used the Post-increment Register Indirect to automatically go to the next memory location. For the second part of the lab, we were asked to implement the Trapezoidal rule from calculus to estimate the area under a curve whose data points will be pre stored in one of the data storage files provided to us in the eClass. The exception in this part of the lab was that we were not allowed to use the multiplication or the division instructions. So we decided to do bit shifting to do the multiplication and the division for this part and used our knowledge from part to complete part

DESIGN

Part A:

Part A of this lab focuses on writing a program that adds the contents of two arrays stored in different memory locations and places the result in a different memory location. The operational code contains information of where to access the arrays in memory and where to store the sum, which is provided at memory location 0x20001000. Each piece of information is stored as a word, where the first word at 0x20001000 contains the size of the arrays, the second word contains the address of the first array, the third word contains the address of the second array, and the fourth, fifth, and sixth words contain the location of where to store the different summed arrays. We investigated three different ways to store the arrays in this lab; one with register indirect with offset, another with indexed register indirect, and the last with post increment register indirect. For the register indirect with offset, we simply loaded the value with a numerical offset using LDR Ra, [Rb, #N], where Ra is where the value is stored, Rb is the address we are interested in starting at, and N is the offset of the address. For the indexed register indirect, we loaded the value with a register offset using LDR Ra, [Rb, Rn], where Ra is where the value is stored, Rb is the address we are interested in starting at, and Rn contains the value that we want to offset Rb by. For the post increment register indirect, we loaded the value using LDR Ra, [Rb], #N, where Ra is where the value is stored, Rb is the address we get the data from, and N is what we increment the register Rb by after the value has been loaded. We use these three different methods of loading to aid us in extracting values to be summed.

To design this program, we first make 3 separate flowcharts, one for each method of storing. For the first method of storing, we only summed the first 3 elements of the array. First we need to load the relevant values for the first summation which includes the location of the operational code, the size of the arrays, the address of the first array, the address of the second array, and the storing address for part 1, which are all stored in R0, R1, R2, R3, and R4 respectively. Then we need to load the value of the first and second array using R1 and R2 (which contains the addresses of the arrays), which are stored into R8 and R9 respectively. Then we add these values together and store this value into R10. Finally, we store the value from R10 into the address at R4 (which contains the address we want to store for Part 1). This completes our first addition. Next, we load the values of the first and second array using R1 and R2 (which contain the addresses of the arrays), but we increment each of them by 4 to make sure that they contain the location of the next array value, then we store the value into R8 and R9 respectively. Again, we add together the values and store the result into R10, then we store the value from R10 into the address at R4, but the address is incremented by 4 (to again ensure that we are storing the next value). Finally, we load the values of the first and second array using R1 and R2 (which contain the addresses of the arrays), but we increment each of them by 8 to make sure that they contain the location of the next array value (the third number in this case), then we store the value into

R8 and R9 respectively. Again, we add together the values and store the result into R10, then we store the value from R10 into the address at R4, but the address is incremented by 8 (to again ensure that we are storing to the third number location). This implementation can be shown in Figure 1.

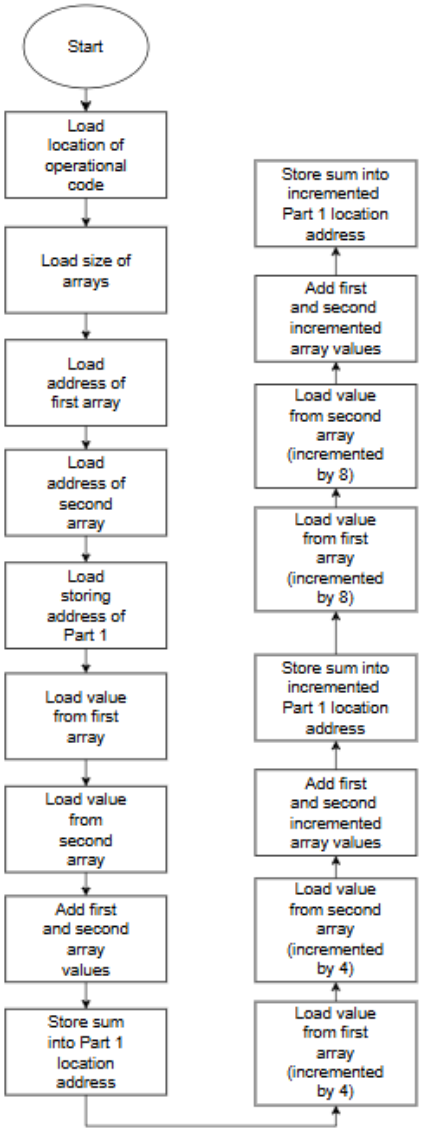

Figure 1: Register index with offset flowchart

For the second method of storing, we summed all of the elements in the array. To do this, we created a counter that would count how many summations we performed. This counter is used at the beginning of the program to see if we have reached the number of items in the array. We start the program by initializing the counter and register offset to 0, then we begin a loop. This loop

starts by comparing the counter to the number of data points, and if the counter exceeds the number of data points, we exit the program. We load the value of the first and second array into R8 and R9 by loading from the address of the arrays plus whatever the offset value is (initially the offset is 0). Then we sum these 2 values and store the result into R10. We then store the value from R10 into the address for Part 2 to be stored plus whatever the offset is. Finally, we increase the counter by 1, and increase the offset by 4 (since the data is stored in 4 byte intervals). This implementation can be shown in Figure 2.

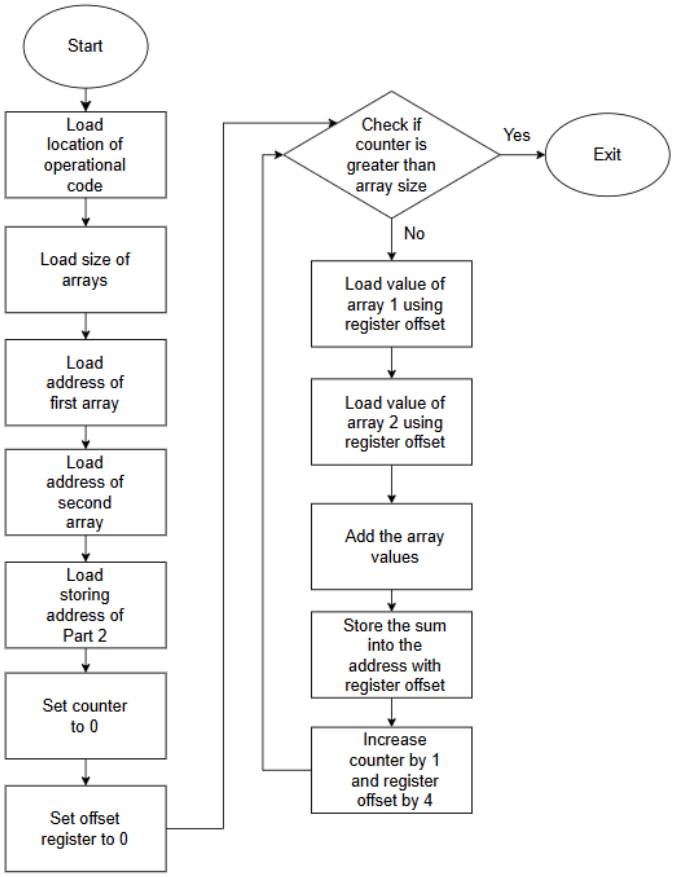

Figure 2: Indexed register indirect flowchart

For the third method of storing, we summed all of the elements in the array. To do this, we created a counter that would count how many summations we performed. This counter is used at the beginning of the program to see if we have reached the number of items in the array. We start the program by initializing the counter and register offset to 0, then we begin a loop. This loop starts by comparing the counter to the number of data points, and if the counter exceeds the number of data points, we exit the program. We load the value of the first and second array into R8 and R9 by loading from the address of the arrays (which are R2 and R3), then we increment each of the register values by 4 by using the post increment syntax. Then we sum these 2 values

and store the result into R10. We then store the value from R10 into the address for Part 3 (which is R6) to be stored, and post increment the R6 register. Finally, we increase the counter by 1. This implementation can be shown in Figure 3.

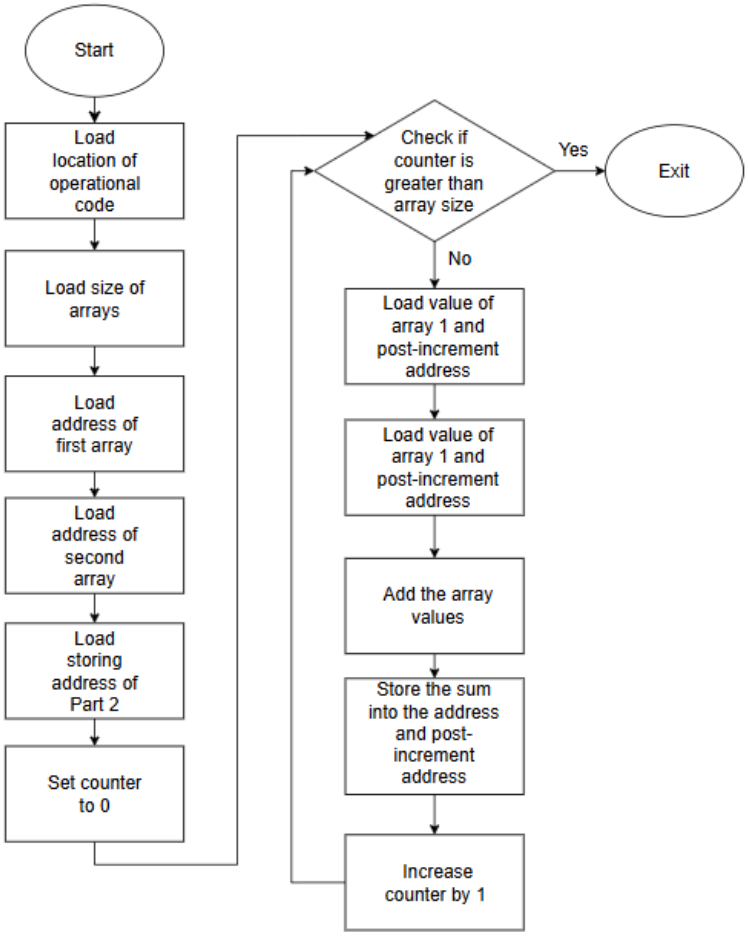

Figure 3: Post-increment register indirect flowchart

Part A summary:

Each different part of this program followed the same procedure, where we load the relevant data values, add the values together, and store them into their respective locations. The only difference is that Part 1 used a register indirect with offset (where we simply increase the location address), Part 2 used an indexed register indirect (where we increase the location address using another register), and Part 3 used a post increment register indirect (where we extract the data from a memory location in a register, and increase the register after the data has been extracted in the same operation). The code implementation of this can be found in Appendix [1].

Part B:

Part B of this lab focuses on writing a program that calculates the area underneath a curve by using the Trapezoidal rule. The Trapezoidal rule works by calculating the area of an infinitesimally small trapezoid, and summing up all of the trapezoids under a function. The data points for the curve are both stored as arrays. The operational code contains information of how many data points there are, where the x data points are stored, where the y data points are stored, a temporary storage space address, and the address of where to store the final area value, which is provided at memory location 0x20001000. Each piece of information is stored as a word, where the first word at 0x20001000 contains the size of the array, the next word contains the address where the x points are stored, and so on. It is worth noting that for this program, it is assumed that the distance between each x data point is either 1 2 or 4, and also that the multiplication and division instructions cannot be used. To combat this problem, we instead use the lsl and lsr operation, since it is equivalent to multiplying or dividing by a power of 2. Whenever we use this shift, it truncates the decimal portion of the result, which could lead to rounding errors. To account for this, we check whenever there is a division of an odd number by 2, and store how many times this happens. At the very end, we simply add on the rounding error to make up for the loss. It is also required to round up the final sum. To do this, we checked if there would be a decimal after adding the error, and if there was then we would round up by simply adding 1 (since we add the truncated error). It is worth noting that since we perform our operations on the difference between 2 points, we only have to do the number of points minus 1 iterations.

To design this program, we make a flowchart to help ensure a structured plan. The first thing we do is load the location of the operational code into register R0. Then we load the number of data points, address of x values, address of y values, and address to store the sum into registers R1 through R4 respectively. Next, we initialize the iteration counter, sum of areas counter, and round error counter to be 0, which is stored in R5, R9, and R10 respectively. Immediately we subtract 1 from the number of data points to do n-1 operations, since our calculations work for pairs. We then start a loop. This loop starts with checking if there are any data points left, if not, then we branch to the “done” loop, if so, then we load the current x value into R6, increment the address of the x values (which is stored in R2), then load the next x value into R7. We then calculate the difference between these x values, and store the value back into R7. A similar approach for the y values; where we load the current y value into R6, then increment the address of the y values (which is stored in R3), then load the next y value into R8. Then we calculate the sum of these y values and store this value back into R8. To clarify, the reason we do this is because the area of a trapezoid is simply the sum of the heights (y values) times the difference between the base (x values) divided by 2. Next, we compare the difference delta x to determine which form of equation we want to use.

If delta x is 1, we need to divide by 2 in the formula but it could be the case that this will not leave us with an integer result. To combat this, we check if the sum is an even or odd

number. If the sum is even, we can perform the division without trouble, but if the sum is odd, then we have to account for the rounding error. We first get the least significant bit of the sum and store it into R11, since it will tell us if the sum is even or odd (0 for even, 1 for odd). Then we divide the sum by 2 (which truncates the potential decimal) and store this value into R8 (this is the area of the trapezoid). We then increase the error counter by the least significant since it will count the amount of truncated divisions we performed. Next, we add the area to the sum of areas stored in R9, and decrease the iteration counter which is stored in R5.

If delta x is 2, we simply add the sum to the sum of areas stored in R9, since for this case, the sum is the area of the trapezoid. We then decrease the iteration counter by 1 which is stored in R5.

If delta x is 4, we simply multiply the sum by 2 and store it back into R8 to get the area, then we add this area to the sum of areas stored in R9, and decrease the iteration counter by 1 which is stored in R5.

After each check of delta x, we return to the beginning of the main loop which eventually will bring us to the done loop when we are out of data points. Since we want our final answer to be rounded up, we can utilize the truncation of division to our advantage. If our error counter is an even number, then this means that we will be adding a whole number to our end sum, so there is no need to round up. If our error counter is an odd number, then this means we will be adding a decimal number to our end sum (specifically a decimal ending in 0.5). The reason for this is because our rounding error for each iteration is a consistent 0.5 since an odd number divided by 2 will always end in 0.5. So if we have an even number of errors, then it will be the same computation as an even number divided by 2 (which is a whole number), likewise, if we have an odd number of errors, it will be the same computation as an odd number divided by 2 (which is a decimal ending in 0.5). Using this theory, we get the least significant bit of the error counter to check if it is even or odd and store this value into R11. Then we divide the error counter by 2 to get how much needs to be added to the sum, and we store this value back into R10. Then we update the sum in R9 by adding the error. We then add the least significant but the end sum, since an even number of errors means we do not have to round up, and an odd number or errors means we do have to round up. Finally, we store this result into the address located at R4. This implementation can be shown in Figure 4.

Part B summary: This program followed a similar procedure as part A; we load the relevant data values, perform calculation to get the change in x and sum of y values for 2 points, and then check to use different variations of the trapezoid area formula depending on the value of delta x. If delta x is 1, we have to start a running “error counter”, which counts the amount of times that there is a rounding error when we divide by 2. We still calculate the area and add it to the sum. If delta x is 2 or delta x is 4, we simply calculate the area and add it to the sum. Once we have calculated all of the data values, we finally need to add the error back and determine whether or not we round our final value. To add the error back, we simply divided the error counter by 2 and added it to the sum since each error iteration caused a 0.5 difference between the actual area and

the calculated area. To determine if we needed to round up or not, we again checked our error counter. If the error counter is even , then there is no need to round up since the final sum doesn't end in 0.5, but if our error counter is odd, then we need to round up since our final sum does end in 0.5. We added the least significant bit of our error counter to the end sum since it accounts for both of these scenarios. The code implementation of this can be seen in Appendix [2].

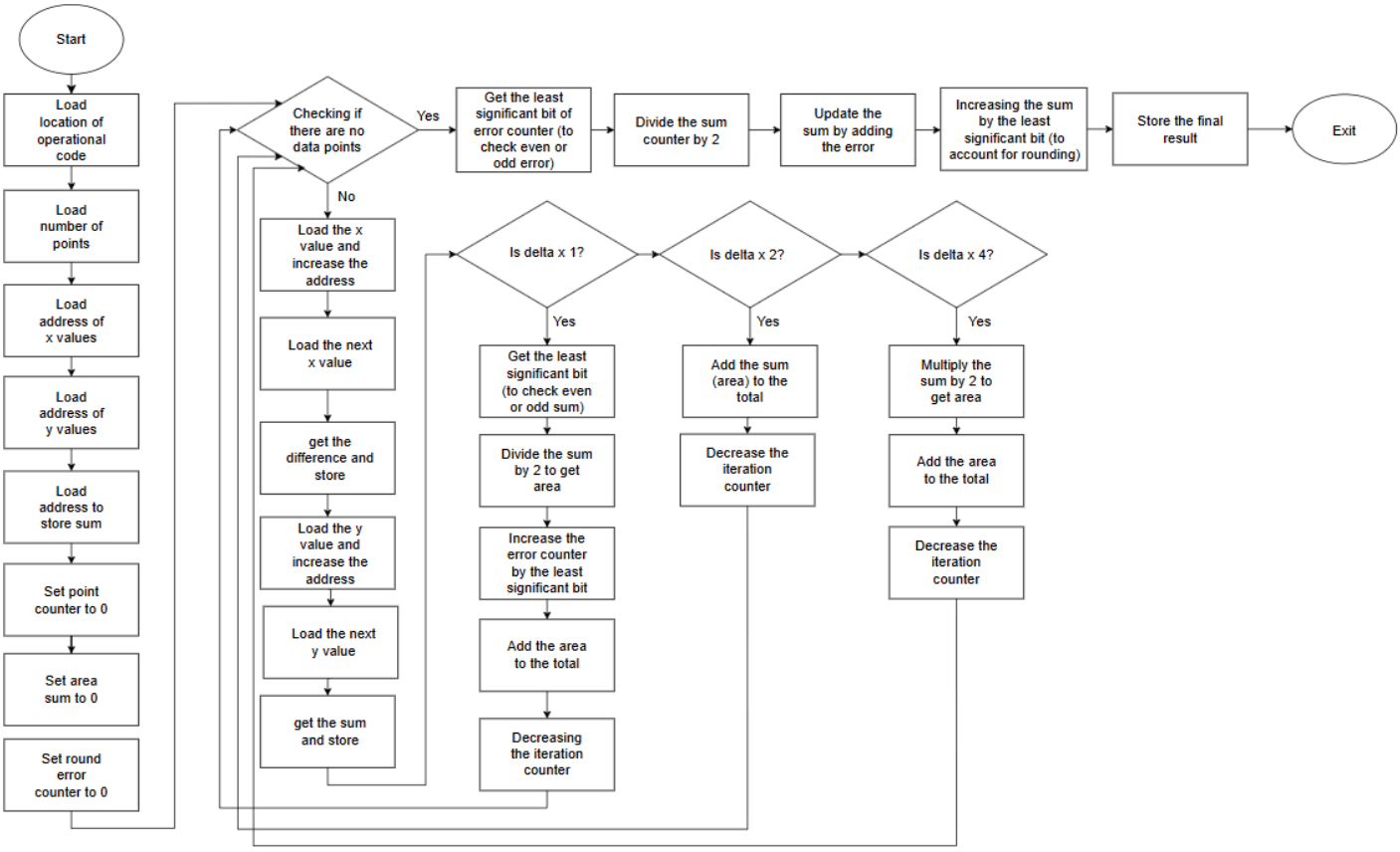

Figure 4: Trapezoid rule flowchart

TESTING

The correctness of the assembly programs was tested using structured testing with tests cases provided to us in the eClass with expected outputs. Our testing methods included verifying that the program was executing correctly outputting the expected results. The testing also included checking memory access accordingly, mathematical calculations and the process of writing the data correctly into memory to ensure the program meets all the requirements of the lab.

For the first part, the program was tested using the datastorage files from the eClass which writes values to specific locations in the memory while the program starts, giving us a predetermined test case with answers to verify the correctness of the program. Part A was split into three different parts testing the correctness of our implementation of the three different addressing modes – Register Indirect with offset, Indexed Register Indirect and Post-Increment Register with Indirect and upon testing they returned the expected output. Firstly, the two arrays were initialized in the memory with the known values and we checked the memory to ensure that the correct values were loaded and then checked the registers to ensure that the registers were loaded with the elements with the correct addressing mode. Then we observed if the registers were performing the correct arithmetic for the correct elements from the memory. After ensuring that the program was computing the correct math for the correct elements, we then moved on to confirming whether the program was storing the results in the correct memory address and with the correct addressing mode. All these things were verified as expected then we finally checked if the program was correctly following the loop and there were no terminations before the full task was done and there were no infinite loops.

For part B, we again used one of the datastorage files with the predetermined data and the known correct result. We checked if the program was loading the correct values to the defined registers with the correct address mode just like part A. Multiplication and division instructions were not allowed, so we validated if out program was doing the bit shifting correctly, and correctly outputting the same values we would have gotten if we multiplied or divided. After ensuring that the values were correct, we moved on to validate if the program was rounding up the values correctly as it was required by the lab manual that it rounds up the result if it was a fraction.

While testing, there were a few things that the program did not get right at first which required further attention. One of the instances was when the program was not correctly rounding up the value as they were not correctly accounted for in the sum. Initially we didn't think that this might cause some inaccuracies in the results but in the process of thorough testing we found out the problem and fixed it by rounding up the value after the sum so that the correct bit shift takes place.

Overall, after all the testing and providing all the fixes that the code needed, both the programs in Part A and Part B functioned as expected providing the correct results for all the test cases. The structured testing process ensured that the memory access, the arithmetic, and all the ways of indexing functioned correctly.

QUESTIONS

1. What are the advantages of using the different addressing modes covered in this lab? Indicate any restrictions or limitations.

   In this lab we used three different types of addressing modes - Register Indirect with offset, Indexed Register Indirect and Post-Increment Register Indirect. Each of these addressing modes have their unique advantage, restrictions and limitations.
   Register Indirect with offset allows us to efficiently access memory locations by implementing a register with the base address and then offsetting the base register. It is used for structured data access. Register Indirect with offset has limitations such as the offset is always fixed.

2. Do the different Addressing Modes affect the CCR bits in the ASPR register? If yes, which bits are affected?

   Yes, different addressing modes do affect the CCR bits in the ASPR register depending on the operations performed. The affected ASPR bits are Negative (N), Carry (C), Zero (Z) and the Overflow (V). N is set if the output of an arithmetic operation is negative, Z bit is set if the output of an operation is 0, C bits can be affected as memory address calculations can cause overflow due to offsets. V bits can also be affected in case a signed overflow happens while working or loading data.

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? Do this for all 3 different test cases.

   The function in the datasets is .
   In DataStorage4.s, the x values range from 0 to 50
   In DataStorage5.s, the x values range from 0 to 80
   In DataStorage6.s, the x values range from 0 to 110
   All the DataStorage files contain 50 x values and 50 y values
   ∴

   Theoretical areas:
   For DataStorage4.s

$$\int_0^{50} x^2\,dx = 41166.\overline{6}$$

   For DataStorage5.s

$$\int_0^{80} x^2\,dx = 170666.\overline{6}$$

For DataStorage6.s

$$\int_0^{110} x^2\,dx = 443666.\overline{6}$$

Calculated area:
For DataStorage4.s = 41675
For DataStorage5.s = 170710
For DataStorage6.s = 443843

Percentage Error:

$$\%\ Error = \frac{Theoretical\ Area - Calculated\ Area}{Theoretical\ Area} \times 100$$

For DataStorage4.s = 1.235%
For DataStorage5.s = 0.025%
For DataStorage6.s = 0.034%

CONCLUSION

In this lab we implemented and analyzed three addressing modes in assembly language programming- Register Indirect with offset, Indexed Register Indirect and Post-Increment Register Indirect. We also implemented the Trapezoidal rule from calculus in our assembly program to approximate the area under a curve, using bit shifting only, without using any sort of multiplication and division instructions. We ensured both of our programs work correctly through various testing and comparing our program's output to the correct output given to us by out lab instructors. We also verified our results with the theoretical values of the area with minimal errors which again confirmed the validity of our program.

APPENDIX

[1] Code for Part A

```
/* Code created by Ryan O'Handley (1883463) and Anurag Biswas Koushik (1806274) */
/* This program adds the elements of 2 arrays and stores the value in another array location*/
/* 3 different methods were used to load and increment the array values and addresses*/



/* Preliminary setup*/
.global TestAsmCall
.global printf
.global cr
.syntax unified

.text
TestAsmCall:
PUSH {lr}
/*--------------------------------------*/


ldr r0, =0x20001000 /* Loading the address we get the data from */
ldr r1, [r0] /* Loading the size of the arrays */
ldr r2, [r0, #4] /* loading the address of the first array */
ldr r3, [r0, #8] /* loading the address of the second array */
ldr r4, [r0, #12] /* loading the address where to store Part 1*/
ldr r5, [r0, #16] /* loading the address where to store Part 2*/
ldr r6, [r0, #20] /* loading the address where to store Part 3*/

/* Part 1 */
ldr r8, [r2] /* Loading the first value of the first array */
ldr r9, [r3] /* Loading the first value of the second array */
add r10, r8, r9 /* Adding the values and storing into R10 */
str r10, [r4] /* Storing the sum into the Part 1 sum address */

ldr r8, [r2, #4] /* Loading the second value of the first array */
ldr r9, [r3, #4] /* Loading the second value of the second array */
add r10, r8, r9 /* Adding the values and storing into R10 */
str r10, [r4, #4] /* Storing the sum into the incremented Part 1 sum address */
```

```
ldr r8, [r2, #8] /* Loading the third value of the first array */
ldr r9, [r3, #8] /* Loading the third value of the second array */
add r10, r8, r9 /* Adding the values and storing into R10 */
str r10, [r4, #8] /* Storing the sum into the incremented Part 1 sum address */


/*part 2*/
mov r7, #0 /* Setting the counter to 0 */
mov r4, #0 /* Setting the register offset to 0 */

/* Creating the main loop */
loop1:
/* Checking if the counter has exceeded the number of points */
cmp r7, r1
bge done1 /* Exiting to Part 3 if the counter exceeds the number of points */


ldr r8, [r2, r4] /* Loading the value of the first array with a register offset */
ldr r9, [r3, r4] /* Loading the value of the second array with a register offset */
add r10, r8, r9 /* Adding the values and storing into R10 */
str r10, [r5, r4] /* Storing the sum into the incremented register offset address */

add r4, #4 /* Incrementing the register offset by 4 (bytes) */
add r7, #1 /* Incrementing the counter by 1 */
b loop1 /* Restarting the loop */


/* Break out to part 3 */
done1:

/* part 3 */

mov r7, #0 /* Resetting the counter */


/* Creating the main loop */
loop2:

/* Checking if the counter has exceeded the number of points */
cmp r7, r1
```

```
bge done2 /* Exiting if the counter exceeds the number of points */


ldr r8, [r2], #4 /* Loading the value of the first array, then incrementing the address register */
ldr r9, [r3], #4 /* Loading the value of the second array, then incrementing the address register */
add r10, r8, r9 /* Adding the values and storing into R10 */
str r10, [r6], #4 /* Storing the sum into the Part 3 address, then incrementing the address register
*/
add r7, #1 /* Incrementing the counter by 1 */
b loop2 /* Restarting the loop */

/* Break out of Part 3 */
done2:


/* Exiting the program */

POP {PC}

.data
```

[2] Code for Part B

```
/* Code created by Ryan O'Handley (1883463) and Anurag Biswas Koushik (1806274) */
/* This program calculates the area under a function by using the trapezoid rule approximation */
ldr r0, =0x20001000 //number of data points//
ldr r1, [r0] //n//
ldr r2, [r0, #4] //x points//
ldr r3, [r0, #8] //y points//
ldr r4, [r0, #16] //store address//
mov r5, #0 //initialize the counter//
mov r9, #0 //initialize the sum of areas//

mov r10, #0 //initialize the counter for round error//

sub r1, r1, #1 //convert to n-1 since we process the last point automatically//

loop:
cmp r1, #0 //checking if there are no data points//
beq done

ldr r6, [r2] //x1 value//
add r2, r2, #4 //x2 address//
ldr r7, [r2] //x2 value//
sub r7, r7, r6 //getting the difference and storing//

ldr r6, [r3] //y1 value//
add r3, r3, #4 //y2 address//
ldr r8, [r3] //y2 value//
add r8, r8, r6 //getting the sum and storing//



//comparing the delta x to determine which form of equation to use//
cmp r7, #1
beq divide2
cmp r7, #2
beq sum
cmp r7, #4
beq multiply2



divide2:
and r11, r8, #1 //getting the LSB (to check if sum is even or odd)
```

```
lsr r8, r8, #1 //divide the sum by 2//
add r10, r10, r11 //add to the counter//
add r9, r9, r8 //add to the total//
sub r1, r1, #1 //decrease the counter//
b loop

sum:
add r9, r9, r8 //add to the total//
sub r1, r1, #1 //decrease the counter//
b loop

multiply2:
lsl r8, r8, #1 //multiply the sum by 2//
add r9, r9, r8 //add to the total//
sub r1, r1, #1 //decrease the counter//
b loop

done:
and r11, r10, #1 //getting the LSB (to check if error is even or odd)
lsr r10, r10, #1 //divide the counter by 2//
add r9, r9, r10 //update the counter by adding the error
add r9, r9, r11 //rounding up the counter//
str r9, [r4] //storing the final result//
```

REFERENCES

[1] "ECE 212 Lab 2: Addressing Modes," *ECE 212 Laboratory Manual*, University of Alberta. [Online]. Available: eClass Portal, University of Alberta. [ACCESSED: 14/03/2025]

[2] *Test Program Use*, University of Alberta. [Online]. Available: eClass Portal, University of Alberta. [ACCESSED: 14/03/2025].