**ECE 212 Lab - Introduction to Microprocessors**

**Department of Electrical and Computer Engineering**

**University of Alberta**



**Lab 3: Subroutines**

| **Student Name** | **Student ID** |
| --- | --- |
| **Zhouyi Yu** | **1772739** |
| **Robert Chi** | **1756807** |

Table of Contents

[Introduction 3](#_heading=h.gjdgxs)

[Design 4](#_heading=h.30j0zll)

[Part A 4](#_heading=h.1fob9te)

[Part A Flowchart 4](#_heading=h.3znysh7)

[Part A Assembler Code 4](#_heading=h.2et92p0)

[Part B 4](#_heading=h.tyjcwt)

[Part B Flowchart 4](#_heading=h.3dy6vkm)

[Part B Assembler Code 4](#_heading=h.1t3h5sf)

[Part C 5](#_heading=h.4d34og8)

[Part C Flowchart 5](#_heading=h.2s8eyo1)

[Part C Assembler Code 5](#_heading=h.17dp8vu)

[Questions 8](#_heading=h.3rdcrjn)

[Conclusion 9](#_heading=h.26in1rg)

# Introduction

This lab deals with stack operation (push and pop), segmenting a long program/function into several smaller and simpler subroutines/sub-functions.

In part A, We created the “WelcomePrompt” subroutine which prompted the user to to enter the number of entries and the entries themselves which would be used in part B for sorting. This subroutine stored the numbers entered at a given memory location, had restrictions on the number of entries and the range of numbers as well as printed messages based on the input of the user. Finally it also flagged the last number as well as pushed the length stored in a register to a location above the stack. Creating this subroutine allowed us to gain experience working with other subroutines to take in user input and print.

In Part B, We used the data collected from part A and sorted it using an algorithm called bubble sort. This algorithm would go through each index in an array and swap them based on the size of each item in the index effectively “bubbling” the larger items to the top of the array. The stack pointer was used to grab the value stored in the end of part A which was used as the length. Also the values were taken from the memory address where the numbers were stored in part A. Creating this subroutine allowed us to gain experience working with data from another subroutine as well as taught us how to implement a basic sorting algorithm in assembly.

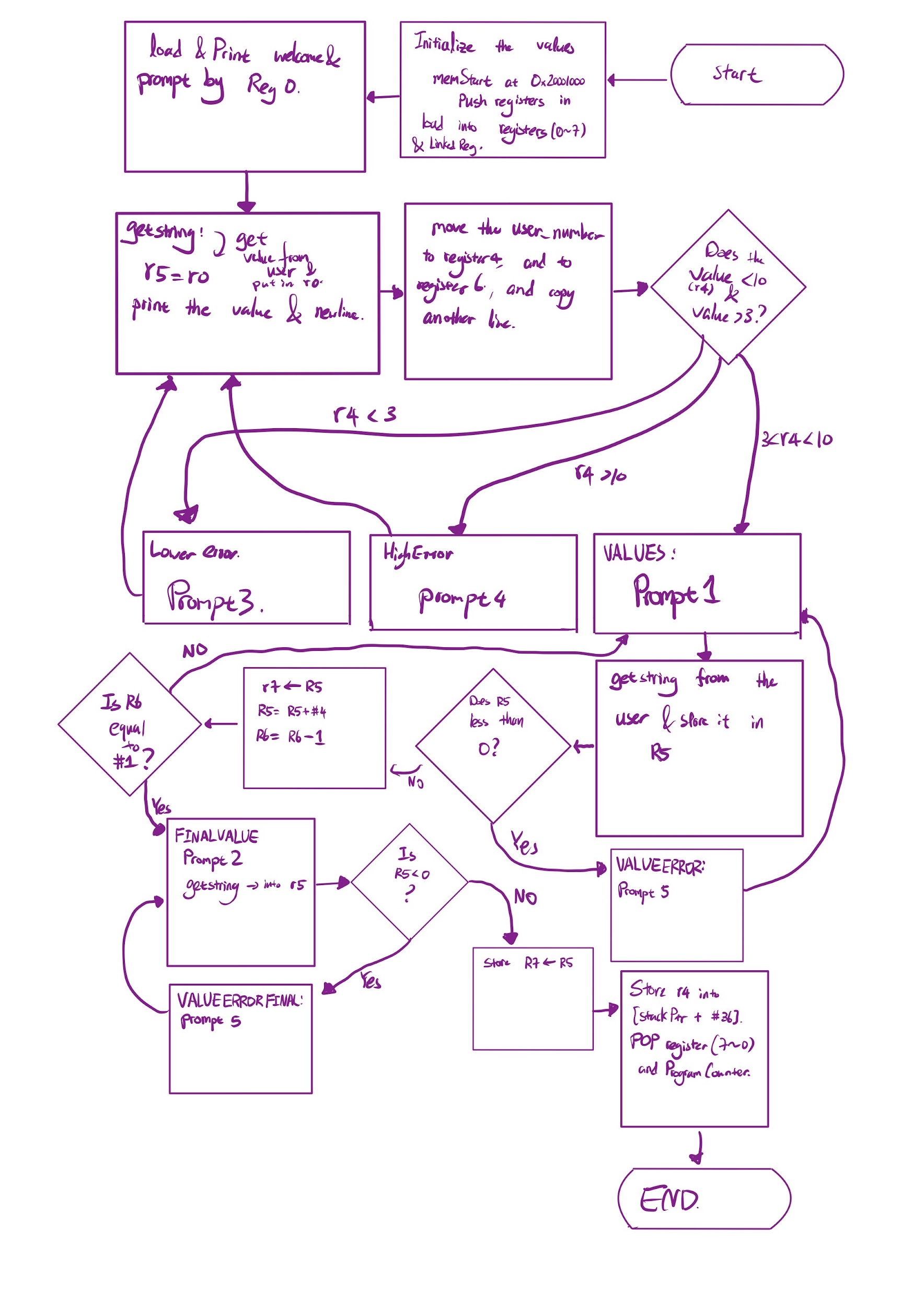
Finally, in Part C, We used the sorted data from part B and displayed it to the user. This also needed the length which was passed through part B in a similar manner to part A. We also called other subroutines in a similar manner to part A which printed out messages to the user as well as the values that were provided by the user back in Part A. All in all the subroutine just displayed the sorted data that the user originally gave along with some messages. Creating this subroutine allowed us to gain experience in completing a chain of subroutines that work with each other to create a basic program.

# Design

## **Part A**

This code is designed by first pushing the linked register and the general register 0 to 7 to the stack, and printing out the welcome prompt on the screen. After these are set, the program starts to ask input from the user and put it into r4 and r6. Checking the range r4 is between 3 and 10, we will receive the number of entries which will later be our counter, and the input process will repeat if the user gives the invalid number of entries; the screen will display the prompt that tells the user what they did wrong and go back to the input section. Then, the program will enter the VALUES section, where we grab the input from the user and store it in R5. However, we need to make sure the input number is not negative. If it is negative, then the program will enter the VALUEERROR and go back to the input values section. If it’s a valid input, then we will put the value of R5 into R7, where the memory starts. Then, we will raise the address of R5 by #4 so it can take the next input and decrease r6, which is functioned as a counter, by 1. This process will repeat until r6 reaches #1, where we will enter the FINALVALUE section that will take the last number of entry streams. In there, if the r5 is negative, then it will go to VALUEERRORFINAL section, which will print a prompt to tell the user that they need to enter a non-negative integer and go back to FINALVALUE. If the entry is valid, then we store r5 into r7. Then, we store the r4 into the address that stack pointer is pointing to plus #36. Finally we popped out all 8 general registers(0~7) and the program counter.

## **Part A Flowchart:**



## **Part A Assembler Code**

/\*Author - Wing Hoy. Last edited on Jan 17, 2022 \*/

/\*-----------------DO NOT MODIFY--------\*/

.global Welcomeprompt

.global printf

.global cr

.extern value

.extern getstring

.syntax unified

.text

Welcomeprompt:

/\*-----------------Students write their subroutine here--------------------\*/

PUSH {lr}

PUSH {r0}

PUSH {r1}

PUSH {r2}

PUSH {r3}

PUSH {r4}

PUSH {r5}

PUSH {r6}

PUSH {r7}

.equ memstart, 0x20001000

ldr r7,=memstart

//REMEMBER Stack pointer is r13 and Link pointer is r14

//DONT USE r0-r3 to store important info

ldr r0,=WelcomeString

bl printf

bl cr

LENGTH:

ldr r0,=Prompt

bl printf

bl cr

bl getstring //gets a number from the user and saves it in r0

mov r4, r0 //move the number to r4

mov r6, r4

bl value

bl cr

cmp r4,#3

blt LOWERROR

cmp r4,#10

bhi HIGHERROR

b VALUES

LOWERROR:

ldr r0,=Prompt3

bl printf

bl cr

b LENGTH

HIGHERROR:

ldr r0,=Prompt4

bl printf

bl cr

b LENGTH

VALUES: //length has been successful and stored in r4

ldr r0,=Prompt1

bl printf

bl cr

bl getstring

mov r5, r0

bl value

bl cr

cmp r5,#0

blt VALUEERROR

str r5,[r7],#4

sub r6, #1

cmp r6, #1

beq FINALVALUE

b VALUES

VALUEERROR:

ldr r0,=Prompt5

bl printf

bl cr

b VALUES

FINALVALUE:

ldr r0,=Prompt2

bl printf

bl cr

bl getstring

mov r5, r0

bl value

bl cr

cmp r5, #0

blt VALUEERRORFINAL

str r5,[r7]

b END

VALUEERRORFINAL:

ldr r0,=Prompt5

bl printf

bl cr

b FINALVALUE

END:

str r4,[sp,#36]

POP {r7}

POP {r6}

POP {r5}

POP {r4}

POP {r3}

POP {r2}

POP {r1}

POP {r0}

POP {PC}

/\*-------Code ends here ---------------------\*/

/\*-----------------DO NOT MODIFY--------\*/

.data

/\*--------------------------------------\*/

WelcomeString:

.string "Welcome to Wing's Bubble Sort Program"

Prompt:

.string "Please enter the number(3min-10max) of enteries followed by `enter'"

Prompt1:

.string "Please enter a number(positive only) followed by `enter'"

Prompt2:

.string "Please enter the last number(positive only) followed by `enter'"

Prompt3:

.string "Invalid entry, Please enter more than 2 entry"

Prompt4:

.string "Too many entries"

Prompt5:

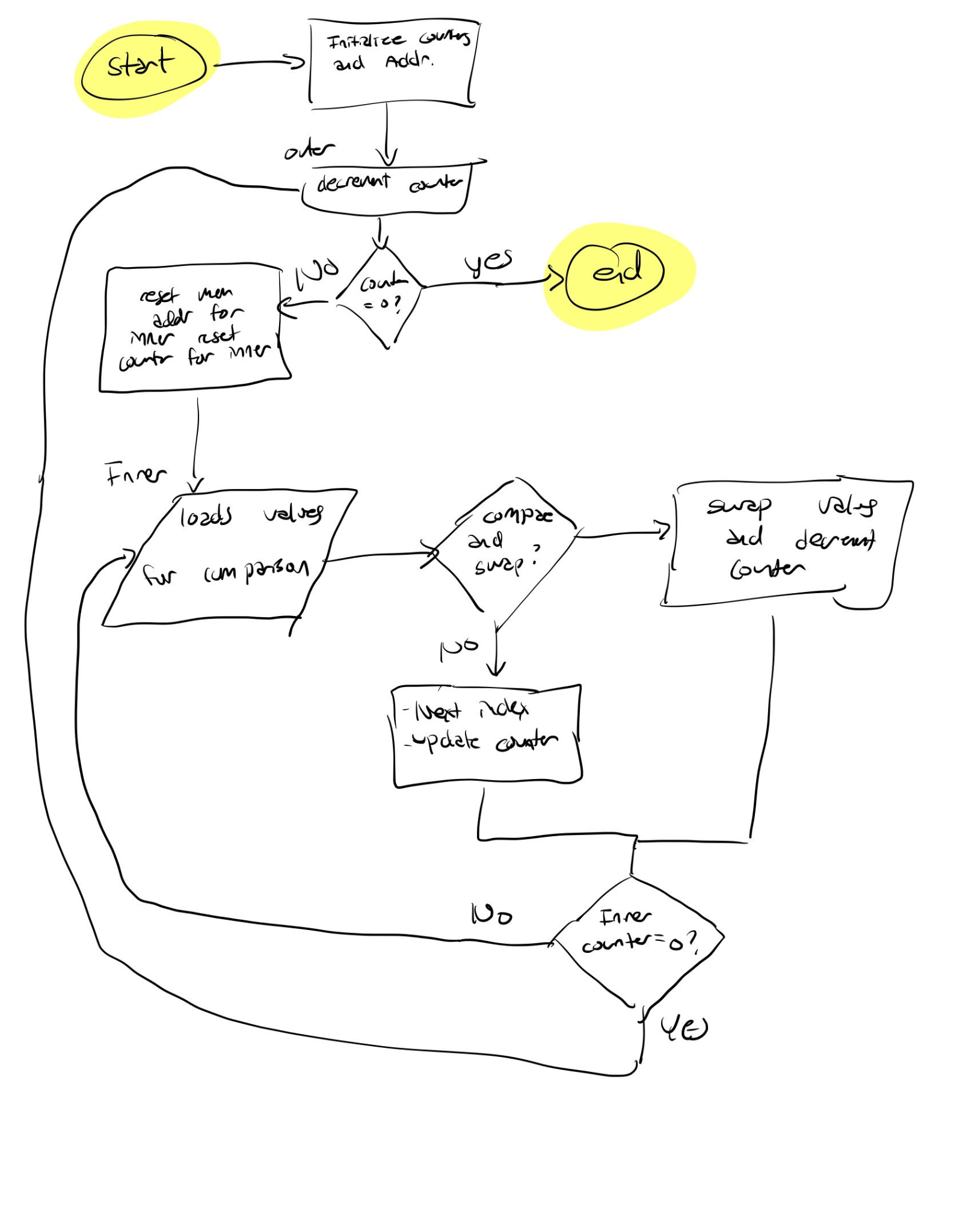
.string "Negative value entered, only postive ones accepted"

## **Part B**

This code began by loading all the data required which included getting the number of entries which would act as a counter and the starting address containing the elements to be sorted. Then two ‘’loops” were created by branching to them based on the counter not hitting zero. The outer loop would run based on the number of entries as well as the inner loops meaning this would effectively be a nested loop running n^2 times where n is the number of items that need to be sorted. the inner loop would run n times where each time the item at the current index would be compared to the item at the next index which are located 4 bytes further than the current. If the current index is larger then the two would be swapped by storing the item in the current index which is loaded in a register into the next index and vice versa for the next index. If the current was smaller than the next then the inner loop counter would just decrease by one and keep running. When the outer loop hits zero that means every index has been compared to each other which results in the larger numbers “bubbling” themselves to the top in order of size.

For an example of how the minimum and maximum values would be found we can look at the array [3,4,1,2]. The outer loop would run 4 times as well as the inner loop. The inner loop would first compare 3 and 4 and not switch since 3 is smaller than 4. Then it would compare 4 and 1 and switch them since 1 is smaller than 4 resulting in [3,1,4,2]. Then it would compare 4 and 2 resulting in [3,1,2,4] finishing our inner loop and finding our max value of 4. the counters would be adjusted accordingly and the inner loop would run again doing the comparisons resulting in [1,2,3,4]. Therefore finding the minimum value of 1. (Based on how the code was written the inner loop would run two more times since the counter for the outer loop was 4. This code could be optimized more for sure but in this case it's not necessary since our data sizes are so small and assembly is so quick. Bubble Sort would not be used in real life applications either way)

## **Part B Flowchart**



## 

## **Part B Assembler Code**

/\*Author - Wing Hoy. Last edited on Jan 17, 2022 \*/

/\*-----------------DO NOT MODIFY--------\*/

.global Display

.global printf

.global cr

.extern value

.extern getstring

.syntax unified

.text

Display:

/\*-----------------Students write their subroutine here--------------------\*/

PUSH {lr}

PUSH {r0}

PUSH {r1}

PUSH {r2}

PUSH {r3}

PUSH {r4}

PUSH {r5}

PUSH {r6}

PUSH {r7}

.equ memstart, 0x20001000

ldr r4, [sp, #36] //loads the number of entries into r4 (length)

ldr r5,=memstart

mov r6, r4 //counter

ldr r0,=Hello

bl printf

bl cr

ldr r0,=Entries

bl printf

mov r0, r4

bl value

bl cr

ldr r0,=Array

bl printf

bl cr

loop:

ldr r0, [r5]

add r5 , #4

bl value

bl cr

sub r6, #1

cmp r6, #0

beq end

b loop

end:

ldr r0,=endmsg

bl printf

bl cr

POP {r7}

POP {r6}

POP {r5}

POP {r4}

POP {r3}

POP {r2}

POP {r1}

POP {r0}

POP {PC}

/\*-------Code ends here ---------------------\*/

/\*-----------------DO NOT MODIFY--------\*/

.data

/\*--------------------------------------\*/

Hello:

.string "The numbers are sorted with bubblesort algorithm"

Entries:

.string "The number of entries was "

Array:

.string "Sorted from smallest to biggest, the numbers are:"

endmsg:

.string "program ended"

## 

## 

## **Part C**

This code is designed by first pushing the linked register and the general register 0 to 7 to the stack. After loading the number of entries into r4, start of memory into r5, and the content of r4 into r6 which will later be used as a counter, a prompt will be printed to let the user know it’s a bubble sort algorithm program. Then, after we print a value from sorted array, we print out a value, then we raise r5 by #4 so it could read the next element in the array and decrement r6 by 1. Every time we reach the end of the loop, we check if r6 is equal to 0 or not; if it is, we go to the end of the program and print the ending message and pop out every register from the stack; if it is not, then we go through the loop again until every item is popped out. This part of the program is quite simple as it is just meant to display all the work that was done in the previous two parts of the lab.

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

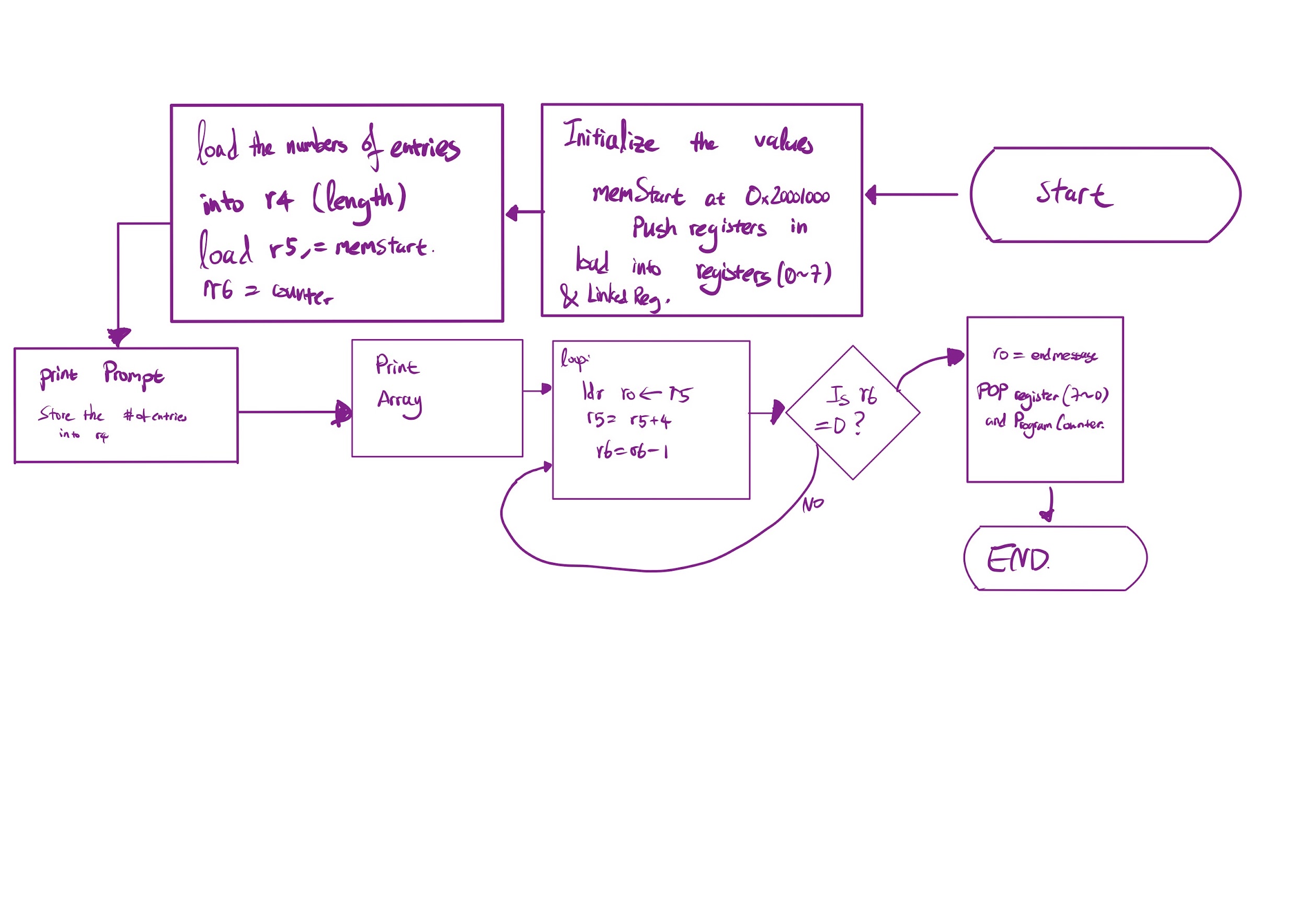
## 

## 

## 

## 

## **Part C Flowchart**



## **Part C Assembler Code**

/\*Author - Wing Hoy. Last edited on Jan 17, 2022 \*/

/\*-----------------DO NOT MODIFY--------\*/

.global Display

.global printf

.global cr

.extern value

.extern getstring

.syntax unified

.text

Display:

/\*-----------------Students write their subroutine here--------------------\*/

PUSH {lr}

PUSH {r0}

PUSH {r1}

PUSH {r2}

PUSH {r3}

PUSH {r4}

PUSH {r5}

PUSH {r6}

PUSH {r7}

.equ memstart, 0x20001000

ldr r4, [sp, #36] //loads the number of entries into r4 (length)

ldr r5,=memstart

mov r6, r4 //counter

ldr r0,=Hello

bl printf

bl cr

ldr r0,=Entries

bl printf

mov r0, r4

bl value

bl cr

ldr r0,=Array

bl printf

bl cr

loop:

ldr r0, [r5]

add r5 , #4

bl value

bl cr

sub r6, #1

cmp r6, #0

beq end

b loop

end:

ldr r0,=endmsg

bl printf

bl cr

POP {r7}

POP {r6}

POP {r5}

POP {r4}

POP {r3}

POP {r2}

POP {r1}

POP {r0}

POP {PC}

/\*-------Code ends here ---------------------\*/

/\*-----------------DO NOT MODIFY--------\*/

.data

/\*--------------------------------------\*/

Hello:

.string "The numbers are sorted with bubblesort algorithm"

Entries:

.string "The number of entries was "

Array:

.string "Sorted from smallest to biggest, the numbers are:"

endmsg:

.string "program ended"

**Questions**

1. *Is it always necessary to implement either callee or caller preservation of registers when calling a subroutine. Why?*

**Answer:** It is only necessary when the next subroutine actually uses the registers that are preserved from the current subroutine. If the subroutine never even uses the registers that are preserved then there is not really a point since the subroutine will just replace those registers with the values that it needs.

1. *Is it always necessary to clean up the stack? Why?*

**Answer:**

Yes it is always necessary to clean up the stack for many reasons. Firstly, if we don't clean up the stack that could leave unused data on the stack which could cause issues for the next subroutine that needs to use the stack because there may be data that was not meant for the new subroutine on the stack which could cause issues. Another reason is that there could be the issue of memory leaks since every time we push onto the stack we consume stack space and if we don't release this space by popping, over time we could fill the stack up with data to the point where the stack is full which will cause issues for future code that needs to access stack space. This is especially possible in cases where there are loops where stack space could be used up quickly. Overall, not cleaning up the stack can result in major issues and therefore the stack should always be cleaned up.

1. *If a proper check for the getstring function was not provided and you have access to the buffer, how would you check to see if a valid # was entered?* A detailed description is sufficient. You do not need to implement this in your code*.*

**Answer:**

You would check for a valid number by looping through the buffer and checking if the values taken in for input were numbers when translating from ASCII. Using the ASCII values that represented 0-9 we could build the numerical value that the user entered in. While looping through the buffer if the input values were not the correct ASCII values that translated to numbers then we could go to the subroutine that asked the user to give a new input. This also automatically solves the issue with negative numbers as we would prompt the user that we need a new input the second we read the ASCII for a negative sign. After getting the numerical value from the buffer checking the other facts for validity would be similar to part A where we just use compare to make sure that the input number is valid

# Conclusion

In conclusion we were able to complete the objectives of the lab which were to gain experience utilizing the STACK and subroutines by creating a program that was separated in 3 parts. Part A was to get the user input according to the parameters defined. Part B was to run the program based on the user input which in this case was Bubble sort and finally part C was to display the sorted data that the user originally gave. We found that creating these programs were largely helpful in teaching us how to utilize the stack pointer to help us store values as well as separating large programs into smaller pieces using subroutines and also utilizing other subroutines to help us complete our objectives. These skills will help in the future when creating even more complex programs that have a lot of components where transferring data using the stack and separating tasks with subroutines would prove to be helpful. Overall the lab was a success and design was done correctly to produce the required results.