Introduction

In this lab we were assigned to write subroutines of a program divided into smaller parts in the ARMv7-M assembly language. This lab tested our skills in understanding modular programming and helped us gain important knowledge about managing data and control flow using subroutines and stack memory. For this lab we were tasked with writing three subroutines, WelcomePrompt, Sort and Display, each subroutine to implement a specific part of a bubble sort algorithm program. In WelcomePrompt subroutine we designed and implemented a user interface tasked to prompt the user for the number of entries for sorting, the upper and lower limit which determined the range of numbers, and then asks the user to input numbers that they like to be sorted. We implemented conditional branches which would check for the relevancy of the user inputs to align with the range chosen. The WelcomePrompt subroutine checked specific conditions such as the number of entries being in the range of 3 to 10, and that the user input of the number to be sorted are greater than the lower limit and less than the upper limit. Invalid inputs will display an error message followed by a prompt to re-input the numbers and recheck the number for validity. After the user is done inputting the numbers, we implement the bubble sort algorithm in the Sort subroutine which sorts the input numbers in ascending order. The lab assignment required us to pass the number of input using stack memory and pass the numbers to be sorted using a dynamic memory address provided in register R0 to be accessed and arranged in ascending order. Lastly, after the numbers are sorted, the program moves to the next subroutine Display which displays the numbers in ascending order.

Design

Overall

The overall purpose of this lab is to write a program that performs a bubble-sort using user-provided inputs. This program contains a main file, which calls upon 3 different subroutines. The first subroutine is called WelcomePrompt and it is responsible for retrieving data from the user. The second subroutine is called Sort and it is responsible for sorting the data that the user inputted in ascending order. The third subroutine is called Display and it is responsible for displaying the sorted output to the user. Each subroutine ends with popping the program counter, to make sure that the program branches back to main.

It is worth noting that for this lab, we were provided with several pre-made subroutines. The first subroutine provided is "printf", which prints a string whose address is in register 0. The second subroutine provided is "cr", which generates a carriage return and linefeed. Essentially, cr is the same as printing a new line.The third subroutine is "value", which prints the decimal number stored in register 0 to the screen. The fourth subroutine is "getstring", which prompts the user for a number input from the keyboard and stores it into register 0. "Getstring" is also able to catch invalid inputs from the user.

Welcome Prompt:

The WelcomePrompt subroutine focuses on interacting with the user to get the list that will be sorted. The user will be asked how many entries are in the list, and what the upper and lower limit boundaries are. It will then prompt the user to enter the number that will be sorted. The number of entries is passed through the stack so that the next subroutines can access it. We do this by manually keeping track of how many items are pushed / popped onto the stack so we can calculate the correct offset from the current stack pointer. After the user inputs a valid number of entries, we store this value at the stack location by using the calculated offset. The numbers to be sorted are stored at some memory location, and this memory location is located initially in register 0. Each number occupies one word of memory, so to calculate the "flag offset", we multiply the net amount of pushes onto the stack by 4. This offset will then give us how far the original stack pointer is from our flag.

The number of entries entered by the user must be between 3 and 10 inclusive. The values entered by the user must be between the range specified by the user. The range must be valid (the lower limit must be smaller than the upper limit). If the number entered is not within the range specified by the user, or if the range is invalid, the program is to re-prompt the user with an error message. The program also flags when the last number is to be entered with the correct text prompt.

To design this subroutine, we first make a flowchart. Since this is a subroutine, we need to preserve the registers after the subroutine completes. The first thing we do is push all of the registers (including the link register) onto the stack. Then we initialize register 7 to be a counter

for the number of valid entries received, and initialize r9 to be used to offset the current memory location (both registers contain 0 to start). We store the starting address of our numbers into r10, since r0 is used so frequently in all of the other provided subroutines. We then print the welcome text to the user, and then print the prompt asking for how many numbers to be entered. We store this value into r4, and then print it to the screen. Then we check if the number of entries is within our range (3 - 10) and branch to the corresponding error message if needed. If the number of entries is valid, we store the value at the stack pointer offset by 56, since we pushed 14 items onto the stack each of size 4 bytes, so 14 * 4 = 56. Each of the error messages will print the message, then prompt the user again. Next we enter a loop that asks the user for the minimum and maximum values. The minimum value is stored in r5 and the maximum value is stored in r6. If the min is not less than the max, we branch to the corresponding error message, which then re-prompts the user.

We then enter the loop that asks the user for the values. This loop starts with increasing the counter (r7) by 1, then checking if this counter is the same as the number of entries. The reason why we increase the counter before is because this will check if we are at the last entry and need to prompt a slightly different message. If we are not at the last entry, we simply ask the user for the value, then store this value into r8. We then check if this entry is out of range and prompt the corresponding error message if so. If the entry is valid, then we store this value at the base address, plus whatever the corresponding offset is. Then we increase the offset by 4 for the next iteration. We then check if what we just stored was the last element, and if it is then we branch to done. If not, then we simply branch to the beginning of the loop that asks for user input.

At the beginning of the user prompt loop we check if the counter is at the same number of entries, if this is the case, we branch to the lastnumber process. This prompts the slightly different “last number” prompt, which just as before will ask the user for the input, then store this value into r8, check if it is valid, and either store it at the correct memory location or prompt the user again. When the program eventually reaches the end (done), we pop all of the register values back, including the program counter (pc). This then causes the program to return to the calling function (the main file) and continue execution. This implementation can be shown in Figure 1, and the code can be shown in Appendix [1].

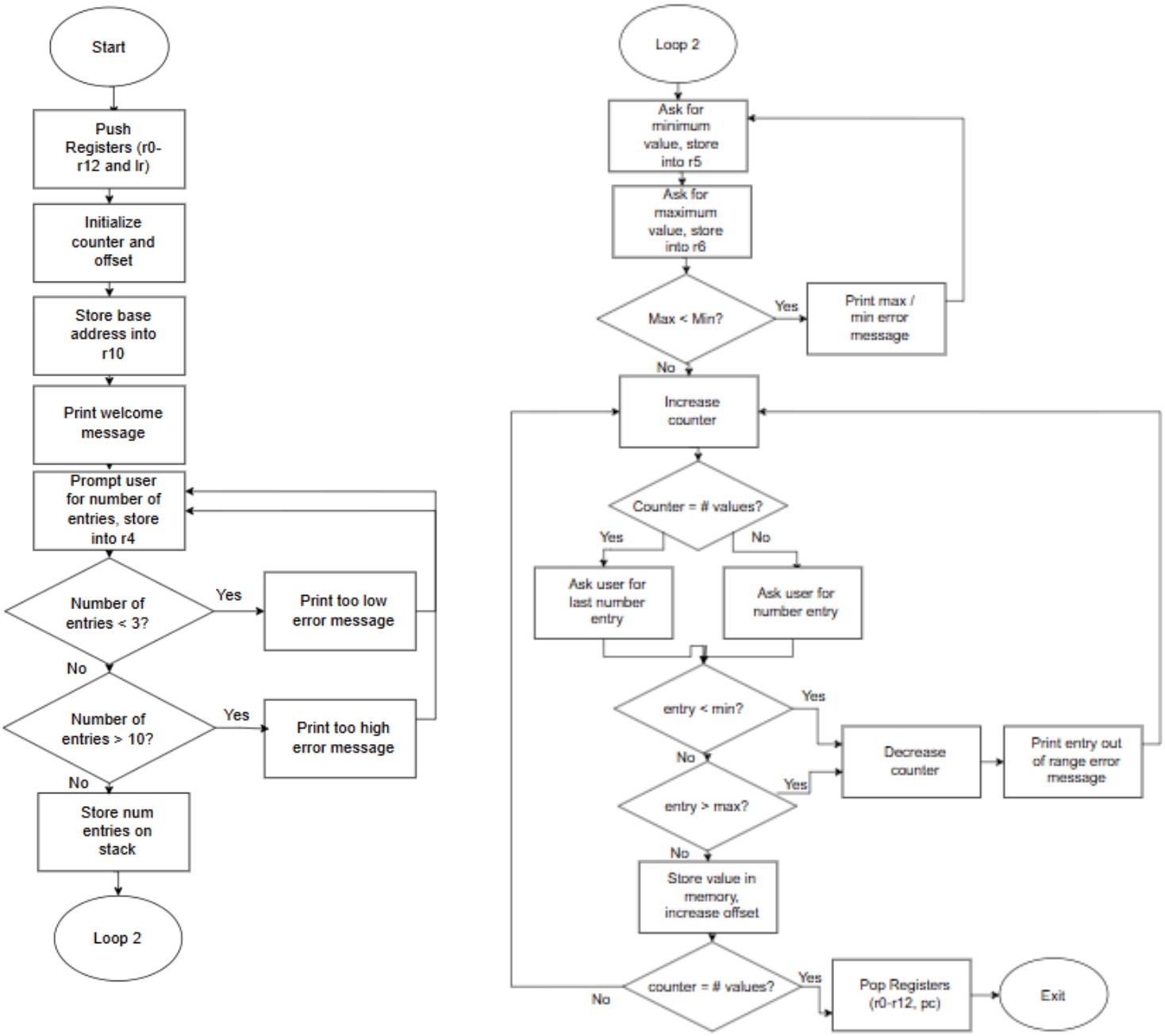

Figure 1: Flowchart of WelcomePrompt subroutine

Sort:

The sort subroutine is responsible for arranging the list of values in ascending order using the bubble sort algorithm. At the time that this subroutine is called, the user's values are in the memory location which was initially stored at r0, this register still contains the memory location since we preserved the register values in the previous subroutine. The valid number of entries is passed through the stack at a known offset. In this case, our offset is 40 since we push 10 registers onto the stack * 4 bytes per number = 40 byte offset.

A bubble sort algorithm works by comparing 2 adjacent numbers and swapping their places if the first number is bigger than the second number. Eventually, the biggest number in the list will be the last element in the list, so we repeat this process for the list but this time we exclude the last numbers(since we already know it is in its correct place). This process repeats again but now we exclude the last 2 numbers, then 3, then 4, all the way until we exclude n number, which means the list is sorted!

We can implement this bubble sort by using 2 loops. The first loop controls how many times we "rise" the biggest number to the top of the list, and the second loop controls how far down the list we go. The inner loop counter is to reset after every full iteration.

To design this subroutine, we first make a flowchart. Since this is a subroutine, we need to preserve the registers after the subroutine completes. The first thing we do is push registers 4 through 12, as well as the link register onto the stack for preservation. Next we initialize register 10 to hold our number of values, which we retrieve from the offset stack pointer. We then decrease this value by 1 to act as our loop's upper threshold, since we only need to loop n-1 times. Before entering our loop, we push the starting address of the numbers onto the stack to be accessed in the display subroutine. We start our first loop by initializing a counter of 0 stored at r5, and initialize r6 to be the base address of the array (which we can get from r0). We then enter the second loop, which compares the loop counter (r5) to the upper threshold (r10) to see if we have checked all of the pairing for this iteration. If not, then we load the current element into r7, load the next element into r8, and compare the two elements. If r7 is less than r8, then there is no need to swap and we increase the counter, increase to the next element, and branch back to the second loop. If r7 is greater than r8, then this means we need to swap the values. We do this by simply storing the register values into their corresponding memory locations. Eventually, we reach a point where the loop counter is equal to the upper threshold, which means that we have done all of the swaps that we wanted to for this iteration, so we decrease the upper threshold, and branch back to the first loop as long as the upper threshold is greater than zero. If it is equal to zero, then we simply print that we are done sorting, and pop all of the registers (again including the program counter). This implementation can be shown in Figure 2, and the code can be shown in Appendix [2].

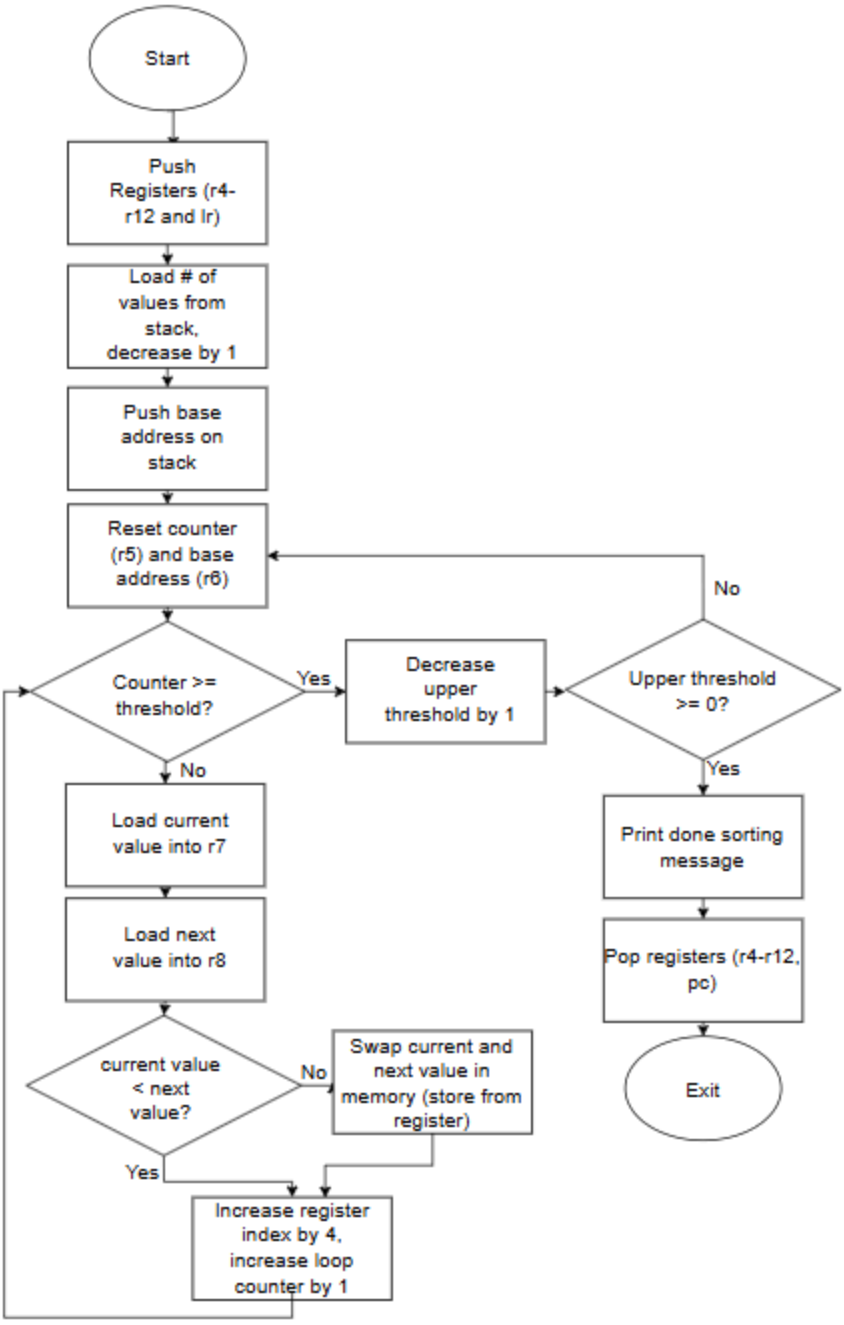

Figure 2: Flowchart of Sort subroutine

Display:

The display subroutine is responsible for displaying the number of entries, the sorted numbers from lowest to highest, and displaying a string that indicates the end of the program. The value for the number of entries and the base address for the numbers are passed through the stack. We can access these values by calculating the offset after we push our registers onto the stack. We push 10 registers onto the stack, each taking 4 bytes of space which makes 40 bytes our offset for the number of entries. One memory location after this (44) is where our base address is located. We print the values by simply using the "value" subroutine and a loop.
To design this subroutine, we make a flowchart. Since this is a subroutine, we need to preserve the registers after the subroutine completes. The first thing we do is push registers 4 through 12, as well as the link register onto the stack for preservation. Then we load the number of values into r4, and load the base address into r5 from the stack. Next we print that the numbers are sorted, then we print the number of entries using r4. Then we print the first part of the sorted prompt and enter a loop. The loop loads the current value into r0 using r5, then postincrements the index (r5) by 4 to prepare for the next value. We load this value into r0, then use the "value" subroutine to print it. After printing, we decrease the counter and check if the counter is zero, if not then we loop again. If the counter is zero, we pop the registers and our code is complete. This implementation can be shown in Figure 3, and the code can be shown in Appendix [3].

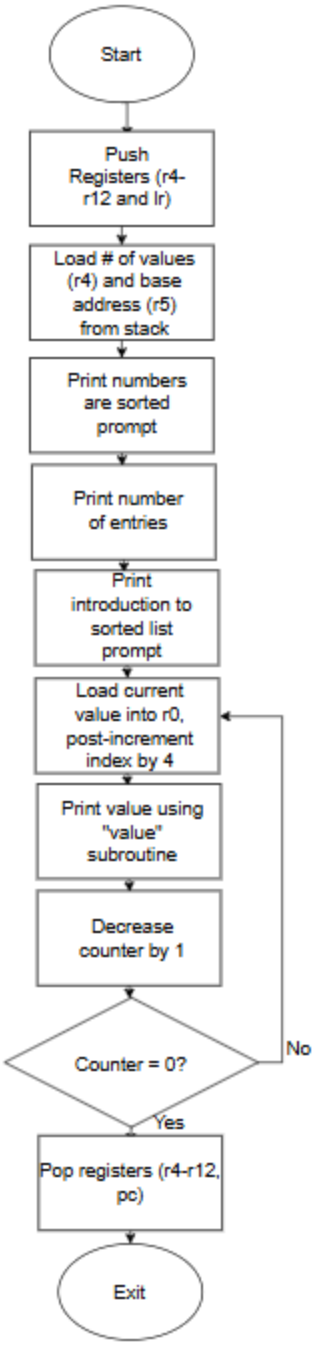

Figure 3: Flowchart of Display subroutine

Testing

We were provided with a test program in our eClass which was designed by our lab tech to test the correctness and the functionality of our program. Initially we finished coding for our program in the Keil uVision simulator and checked the program to verify our program functionality. We firstly checked out WelcomePrompt subroutine by deliberately entering valid and invalid numbers and numbers out of range to see if the program was correctly detecting the errors and that the program was correctly giving out error messages followed by a prompt to enter the numbers again. After the WelcomePrompt passed all the tests we then checked the Sort subroutine to verify if it was sorting the numbers correctly in ascending order and then we checked the Display subroutine to test for the values that they are printed in the correct order. All of these tests passed in the Keil uVision Simulator but in the lab with the real microprocessor connected, we noticed that the program was getting stuck in an infinite loop after the first subroutine that is the WelcomePrompt. This told us that the simulator was not handling the subroutines correctly and so we moved on to debug the program in the STM32 connected to the Nucleo L432KC microprocessor. Our lab TA then inspected our problem and concluded that the Keil uVision Simulator was not handling the stack memory correctly and pointed to us that our program had bugs in handling stack instructions which was causing the program to get stuck in one subroutine and not moving on to the next subroutine. After deeply analyzing the program we identified that we were improperly using the PUSH {lr} and POP {PC} instructions. In a number of cases we forgot to preserve the registers which were required by the lab assignment and forgot to POP the program counter register at the end of each subroutine.

Additionally, we also faced problems with stack management specially with handling the number of entries not being carried over to the rest of the subroutines and the handling of the memory location being dynamic in register R0. The number of entries were supposed to be stored in the register R4. The problem for the number of entries not being carried over to the rest of the subroutines originated from us incorrectly using PUSH {R4} and then pushing other values onto the stack as well for preserving the rest of the registers which caused the stack memory to grow. So when we POP {R4} in the other subroutines, some other value that we pushed onto the stack was popping out in R4 rather than the number of entries. We fixed the problem by using stack pointers and calculating the offsets by taking into account the number of values that we push and pop to point to the address containing the number of entries and retrieve that to the required subroutines. Once we fixed all the problems related to the improper use of PUSH {lr} and POP {PC} and the handling of the stack memory and the stack pointers, the subroutines functioned as expected.

After all the fixes were made, we re-ran our program to verify everything was working and verified that the program was correctly handling the invalid entries and that the input numbers were correctly passed to the remaining Sort subroutine and were sorted correctly and

then in the Display subroutine the numbers were printed in correct order with the number of entries also being correctly passed throughout the subroutines.

Questions

1. Is it always necessary to implement either callee or caller preservation of registers when calling a subroutine. Why?
   Implementing callee or caller preservation of registers is necessary to make sure that the important data in the registers are unintentionally overwritten or lost during the execution of subroutines or calling external functions that use the registers. Registers hold temporary data such as memory addresses or values to be used in the program. When a subroutine is called it might overwrite the registers with other values used to perform its operations. Using the callee and the caller prevents losing the important data from being overwritten. The callee and the caller function helps to restore the values after the necessary overwriting to the registers are done by the subroutine.

2. Is it always necessary to clean up the stack. Why?
   Cleaning up the stack is necessary because if the stack is not cleared, while calling other subroutines, the stack might grow and corrupt the program data when called later because the stack pointer would not be at the same place if the required data for the program is not pushed and popped correctly. While calling subroutines, there might be temporary values pushed or popped by the computer to complete the operations which will cause the stack pointer to misalign and incorrect data might be accessed or restored in the subroutine operation.

3. 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.
   If the getstring function did not perform the input validation of whether the input ASCII character is a valid integer, we could manually check if the input is a valid number stored in the buffer. We could take what was input in for the number and then take everything until the newline character which will mark the end of input. If a valid number was entered the computer would interpret the numbers as hexadecimal in ASCII (0x32 to 0x39 for number 1 to 9). We would make the number detection to only allow a “-” minus sign followed by a number and discard everything else. If the entered ASCII character is not within the range 0x30 to 0x39 in hex, it means that the input is not a number and will discard the input immediately. We can also put additional checks such as the input size not being a 32-bit signed integer.

Conclusion

In this lab we successfully implemented the bubble sort algorithm in ARM assembly by dividing the program into three subroutines, WelcomePrompt, Sort and Display, each subroutine having their own specific job in the program from gathering the user inputs to validating the inputs and sorting them in ascending order to displaying them is a orderly manner. We used the stack to pass the number of inputs and implemented preserving the registers in the stack and stored the numbers to be sorted in a dynamic memory location stored in R0. During the development of this program we ran into several issues such as improper stack handling and the subroutines not going to the next subroutines. All of these issues were solved by correctly using the push and pop in the stack and using stack pointers with offsets to point to the correct memory locations. The completed version of the implementation contains a functional and well structured assembly program.

References

[1] *ECE 212 Lab 3*, University of Alberta. [Online]. Available: eClass Portal, University of Alberta. [ACCESSED: 04/04/2025]

Appendix

[1] Code for Welcomeprompt subroutine

```
/* Code created by Ryan O'Handley (1883463) and Anurag Biswas Koushik
(1806274) */
/* This subroutine prompts the user for a list of numbers to be sorted */
/* If a user entered something invalid, the program catches it and displays
the corresponding error message */

/* Preliminary setup */
.global Welcomeprompt
.global printf
.global cr
.extern value
.extern getstring
.syntax unified
.text




/* Start of the subroutine */
Welcomeprompt:


push {r0-r12, lr} /* Pushing the registers to preserve them */

mov r7, #0  /* Initializing counter */
mov r9, #0  /* Initializing offset */
mov r10, r0 /* Storing base address */


/* Printing the welcome text to the user */
ldr r0, =welcometext
bl printf
bl cr


ask:
      ldr r0, =range /* Asking user for how many entries */
```

```
      bl printf
      bl cr
      bl getstring

      mov r4, r0 /* Storing how many entries in r4 */
      bl value /* Printing value to the screen */
      bl cr

      cmp r4, #3 /* Checking if not enough numbers */
      blt invalidlow

      cmp r4, #10 /* Checking if too many numbers */
      bgt invalidhigh

      str r4, [sp, #56] /* Storing amount of numbers on stack (at flag
location) */
      b ask2 /* Branching to the next prompt */

/ * Printing error message and prompting user again */
invalidlow:
      ldr r0, =toolow
      bl printf
      bl cr
      b ask

/ * Printing error message and prompting user again */
invalidhigh:
      ldr r0, =toohigh
      bl printf
      bl cr
      b ask

/ * Loop for asking the user for min and max values */
ask2:
      ldr r0, =asklowest /* Asking for minimum value */
      bl printf
      bl cr
      bl getstring

      mov r5, r0  /* Storing minimum value */
      bl value
      bl cr
```

```
	ldr r0, =askhighest /* Asking for maxiumum value */
	bl printf
	bl cr
	bl getstring

	mov r6, r0  /* Storing maximum value */
	bl value
	bl cr

	/* Checking if max and min are valid, prompting error message if not */
	cmp r6, r5
	blt lowerthanlowest
	bgt askarray


/* Error message if min is not less than max */
lowerthanlowest:
	ldr r0, =error_lowerthanlowest
	bl printf
	bl cr
	b ask2

/* Loop that asks user for input */
askarray:
	add r7, r7, #1 /* Increase the counter for entries received */
	cmp r7, r4 /* Checking if we are at the last entry */
	beq lastnumber

	ldr r0, =array /* Asking user for entry */
	bl printf
	bl cr
	bl getstring

	mov r8, r0 /* Storing entry into r8 */
	bl value
	bl cr


	/* Checking if entry is out of range and prompting corresponding error
message */
	cmp r8, r5
	blt error_range
	cmp r8, r6
	bgt error_range
```

```
store:
      str r8, [r10, r9] /* Store the value in memory */
      add r9, r9, #4 /* Increase the offset */

      cmp r7, r4 /* Checking if that was the last element */
      beq done
      b askarray


/* Prompt the user for the last number (same logic as before, just different
text prompt) */
lastnumber:
      ldr r0, =lastnumbermsg /* Ask user for last number */
      bl printf
      bl cr
      bl getstring

      mov r8, r0 /* Store number into r8 */
      bl value
      bl cr

      /* Check if the number is valid and prompt error message if so */
      cmp r8, r5
      blt error_range
      cmp r8, r6
      bgt error_range

      /* Store the value */
      b store

/* Prints the error message when the user input is out of range */
error_range:
      sub r7, r7, #1 /* Decrease the counter by 1 (since we are going to ask
again) */
      ldr r0, =errormsg_range /* Print error message */
      bl printf
      bl cr
      b askarray /* Branch to main loop */


done:
pop {r0-r12, pc} /* Restore registers */
```

```
/* End of code */

/* Different prompt strings in data section */

.data
welcometext:
      .string "Welcome to ECE 212 Bubble Sorting Program"
range:
      .string "Please enter the number (3min-10max) of entries: "
toolow:
      .string "Invalid the number entered is too low"
toohigh:
      .string "Invalid the number entered is too high"
asklowest:
      .string "Enter the minimum number"
askhighest:
      .string "Enter the maximum number"
error_lowerthanlowest:
      .string "Minimum number cannot be greater than maximum"
array:
      .string "Please enter a number"
errormsg_range:
      .string "Invalid!!! Number entered is not within the range"
lastnumbermsg:
      .string "Please enter the last number"
```

[2] Code for Sort subroutine

```
/* Code created by Ryan O'Handley (1883463) and Anurag Biswas Koushik
(1806274) */
/* This subroutine uses bubble sort to sort the elements of a list in
ascending order */

/* Preliminary setup */
.global Sort
.global printf
.global cr
.extern value
.extern getstring
.syntax unified
.text


/* Start of the subroutine */
Sort:

push {r4-r12, lr}        /* Preserving the registers */

ldr r10, [sp, #40]o  0     /* Loading the number of values from the stack
(using calculated offset) */
subs r10, r10, #1        /* Decreasing by 1 to act as our upper threshold */



/* Start of the first loop */
loop1:
    /* Reset the inner loop counter and base address */
    mov r5, #0
    mov r6, r0

/* Start of the second loop */
loop2:
    cmp r5, r10          /* Check if inner loop counter has reached threshold
*/
    bge next             /* Branch to in between steps before going back to
loop 1 */

    ldr r7, [r6]         /* Load current number into r7 */
    ldr r8, [r6, #4]     /* Load next number into r8 */
```

```
    cmp r7, r8           /* Compare current number to next number */
    ble noswap           /* Don't swap if already in order */

    str r8, [r6]         /* Swap if not in order */
    str r7, [r6, #4]

noswap:
    add r6, r6, #4       /* Increase element index by "1" (4 bytes = 1
element) */
    add r5, r5, #1       /* Increase loop counter */
    b loop2              /* Loop again */

next:
    subs r10, r10, #1    /* Decrease upper threshold (outer loop counter) */
    cmp r10, #0          /* Check if finished the list sort */
    bge loop1            /* Branch back to loop 1 if not finished */

/* Naturally branch to exit if the loop is finished */
exit:
    /* Print the sort finished message */
    ldr r0, =done
    bl printf
    bl cr
    pop {r4-r12, pc}     /* Restore saved registers and return to caller
(main) */


/* End of code */

/*  Prompt string in data section */
.data

done:
      .string "Done sorting"
```

[2] Code for Display subroutine

```
/* Code created by Ryan O'Handley (1883463) and Anurag Biswas Koushik
(1806274) */
/* This subroutine prints the number of values, as well as all of the sorted
numbers in the list */

/* Preliminary setup */
.global Display
.global printf
.global cr
.extern value
.extern getstring
.syntax unified
.text


/* Start of the subroutine */
Display:

push {r4-r12, lr} /* Preserving the registers */

ldr r4, [sp, #40] /* Loading the number of values into r4 */
ldr r5, [sp, #44] /* Loading the base address into r5 */


complete:
      /* Print that the numbers are sorted */
      ldr r0, =completemsg
      bl printf
      bl cr

      /* Print out the number of entries */
      ldr r0, =entries
      bl printf
      mov r0, r4
      bl value
      bl cr

      /* Print out the first part of the sorted prompt */
      ldr r0, =order
      bl printf
      bl cr
```

```
loop:
      /* Load the current value into r0, then increment index */
      ldr r0, [r5], #4
      bl value /* Print the value */
      bl cr

      /* Decrease the counter and branch to the loop if not finished */
      subs r4, r4, #1
      bne loop




pop {r4-r12, pc} /* Restore registers */



/* End of code */

/* Different prompt strings in data section */
.data
completemsg:
      .string "The numbers are sorted with bubblesort algorithm"
entries:
      .string "The number of entries are "
order:
      .string "Sorted from smallest to biggest, the numbers are"
```