**Table of Contents Page**

1. **Introduction**…......................................................................................................ii
2. **Design**

Part A…......................................................................................................................1

Part B…......................................................................................................................2

Part C….......................................................................................................................3

1. **Testing**

Part A…......................................................................................................................4

Part B…......................................................................................................................4

Part C…......................................................................................................................5

1. **Questions**….............................................................................................................6
2. **Conclusion**…..........................................................................................................7
3. **Appendix**

Part A Flowchart…................................................................................................8

Part A Assembler Code…....................................................................................9

Part B Flowchart….............................................................................................12

Part B Assembler Code….................................................................................13

Part C Flowchart…...........................................................................................................15

Part C Assembler Code….................................................................................16

1. **Marking Sheet**…................................................................................................18

**Introduction**

In this design laboratory project, we gained experience dealing with stack (push and pop), dividing up a main program/function into smaller and simpler subroutines/functions as well as calling up those subroutines in times of need. Moreover, we explored the three basic parameter-passing techniques to the stack using data registers, address registers and memory locations and the concepts of ‘push’ and ‘pop’. To reach these objectives, as per the instructions we created three subroutines that would complement the pre-written skeleton for a bubble sort program (the main program).

Firstly, for Part A we implemented the subroutine ‘WelcomePrompt’ via which we welcomed the user to our program and prompted him/her to enter a number in a range which would imply the number of entries the user would want to be sorted with the bubble sort program. Based on the user input a message would be displayed on the monitor. If the user specified a number outside the prescribed limit, it will be discarded with an error message and a re-prompt. After getting a valid number, the user would be prompted for the numbers and once again, negative numbers will be discarded with an error message and a further prompt. By default, the provided subroutine would spontaneously discard invalid characters with an error message and a re-prompt unless the input is a number. The user-provided number of entries would be passed in an array form to the memory location 0x2300000 pointed by a2 for sorting later on.

In Part B, we coded for the subroutine ‘Sort’ that utilized the bubble sort algorithm to sort the numbers in the array pointed by a2 in memory. The number of entries stored on stack below the return address was utilized as a counter in the looping procedure used in the program. Thus the entry conditions were the number of entries and the base address of the array on the stack. Although there were no exit conditions, the output of the subroutine was a sorted version (in ascending order) of the array starting at the memory location 0x2300000.

Finally in Part C, we wrote the ‘Display’ subroutine that, as the name implies, displayed the array in sorted order. The entry and exit conditions of this subroutine were the same as that for ‘Sort’. ‘Display’ would display the total number of entries followed by the sorted numbers. At the end of this subroutine, we indicated the end of the program by printing “Program ended”.

**Design**

Part A

We wrote our first subroutine in the ‘Lab3a.s’ file provided beforehand. Before we coded anything, we wrote down all the strings (or messages) under the ‘.data’ section of the subroutine that we wished to be printed out at one point or another while running ‘WelcomePrompt’. For example, we labeled our welcome message with ‘Welcome\_message1’ , the user-prompting message with ‘Prompt\_message’ and the error message for negative number of entries with ‘Message\_for\_invalid’.

First and foremost, we moved the stack pointer (sp) 60 bytes below its current position using the ‘suba’ command to make space for storing and thus, preserving the information contained in all the 15 registers – d0-d7 and a0-a6. We then copied the 15 longwords from these registers to the stack using the ‘movem’ instruction. Next, based on the printed reference run of a bubble sort program at the first page of Lab 3 document, we printed out our welcome message labeled Welcome\_message1 by first storing this label (i.e., base memory address of the string) on the stack with the ‘pea’ (push effective address) command and printing it with the ‘iprintf’ subroutine using ‘jsr’ (jump to subroutine) instruction. Because after the ‘push’ from iprintf, sp still kept pointing at Welcome\_message1 on the stack, we then cleaned up the stack by adding 4 bytes to sp and moved it to its previous position. We also used the ‘cr’ subroutine here to generate a carriage return and linefeed, i.e., to move to the left-most position on the new line (next line).

Using the same procedure as above, we printed the next message prompting the user for the number of entries specifying the range restriction [2, 10]. The ‘getstring’ subroutine was utilized here to prompt the user for the number of entries. Although, by default, getstring would accept any positive and negative number (only discarding anything otherwise and re-prompting the user) and store the input in the data register d0, for our purpose we checked the user response against our restrictions. Once again, in light of the reference run of the bubble sort program in Lab 3 document, our program would display “Invalid entry” for numbers equal to or less than 1 and “Too many entries” in response to numbers greater 11. The comparisons and determinations for the correct error display would be done using the ‘cmp’ command and the control structures ‘ble’ (branch on less than or equal to 0) and ‘bge’ (branch on greater than or equal to). Discarding the invalid entry, the user would be re-prompted and this process would keep on repeating until the user provided a valid number of entries. At that point, we would make two copies – one copy in d6 to move the reserved longword-sized space on stack (allocated by the main program) using the ‘move’ command and another copy in d7 to use as a counter for the number of entries.

Now it’s time to actually get the numbers from the user he/she wants to be sorted. With this goal in mind, we print the ‘Message\_for\_numbers’ and prompt the user for positive numbers. As before, negative inputs from the user slip past getstring without raising any alarm. Thus, we check the validity of the user input by first making a copy of it in d4 and then comparing it against 0. If it is negative, our program prints out the ‘Message\_for\_negative\_number’ followed by ‘Message\_for\_numbers’ and re-prompts the user for a positive number. The subroutine continues this process and in every cycle decrements the counter, d7. When d7 reaches 1, i.e., when we have one more number to get from the user, the subroutine prints our ‘Message\_for\_last\_number’ letting the user know that his/her next entry would be the last number the subroutine accepts in the array. It should be noted that in every pass though this looping mechanism, as soon as our subroutine determines that the user input is positive, it stores it in longword format from the base address 0x2300000 pointed by the address register, a2. While this process continues WelcomePrompt would keep checking if the counter hits 0. When this happens, we jump to the label ‘Last\_Number’ where we restore the registers using, once again the ‘movem’ commands, advance sp by 60 bytes making it point to the return address. Finally, we jump back to the main program using ‘rts’ (return from the subroutine) which loads the returning address into the PC.

Part B

‘Sort’ is the subroutine where the real action of our bubble sort program takes place. Before referring to this subroutine, the main program in Lab3.s loads the number of entries to d2 from the stack and copies the base memory address of the positive number array, 0x2300000, into a2.

An important line of code we write at the beginning of ‘Sort’ is copying the longword-sized return address onto where the number of entries was originally placed in the main program after, of course, we retrieve the total number of entries into d7. This is because, following run through everything in our subroutine, we would make sp point to this new copy of the return address such that after ‘rts’ pushes this longword into PC, we go back to the main program and our stack is all cleaned up! Since we are not allowed to make any change to the main program and moreover, since our stack is not cleared in the main program after ‘Sort’ is run, we had to take this initiative to avoid the inevitable ‘Trap occurred’ error.

Next, as done in WelcomePrompt, we preserved the register values. We then composed a looping structure ‘Next\_Round’ where we first make a copy of the memory address specified by a2 into a3. This way we can refer back to the starting memory address of the array later. We decrement the ‘round’ counter d7 one unit to indicate the first round in the bubble sort algorithm. We also copy the contents of d7 into d6 to use the latter as a ‘pass’ counter. The reason behind these two operations is because we noted that in a manually ran bubble sort algorithm with n = 5 numbers – 1, 2, 3, 4, and 5 – we require 4 or (n-1) comparisons in the worst case for the actual sorting from the smallest to largest positive number. We also noted that in the (n-1)th round we needed to do (n-1) comparisons of the adjacent numbers using (n-1) passes along the array. We then copied two successive longword-sized positive numbers from the array into d3 and d4 respectively. Afterwards, we would compare to see it d3 is greater than d4. If so, we would branch to ‘Switch\_With\_Next’ where we would overwrite the contents in d4 in the previous memory location (where the longword in d3 was originally stored) using –4(a2). We would then move the contents of d4 into d3 and copy the next longword in the array to d4. If, however, the content of d3 is equal to or less than that in d4, not swapping is necessary and so, we branch to ‘Do\_Not\_Switch’ where we only copy the next two longwords from the array into d3 and d4 respectively. In both Switch\_With\_Next\_Position and Do\_Not\_Switch, we decrement d6 before repeating ‘Compare’ for comparison. When d6 (“pass” counter) hits zero it means we have made the last necessary comparison and so, we move on to the next round by literally jumping back to ‘Next\_Round’. As mentioned previously, d7 (“round” counter) is decremented here. Also, conditions are specified here such that when this counter equals 0 this would imply that we have gone over the maximum required rounds to make the necessary swaps and so, we would return to the main program. To do this, our subroutine would branch out to ‘Lab\_3c’ where the registers would be restored and implement ‘rts’.

Part C

With the hard parts over, all we are left to do is print out the sorted array. Thus, the issue of designing ‘Display’ comes into play. As if it’s our second nature, we start my preserving the register values on the stack. ‘Message\_1’ is then displayed that confirms that sorting using bubble sort algorithm is complete. ‘Message\_for\_entry\_number’ is printed next. Here we make use of the ‘value’ subroutine to print the number of entries from the top of the stack but not before it is copied there from d2. Here we do not use the contents of d2 copied on the stack in the main program before ‘Display’ is run because ‘value’ only cares about the longword sized decimal number on top of the stack. Thus, it would make our program more complicated if we would adjust sp to point the copy of d2 on the stack from the main program and then back to its current position in the subroutine. Alternatively, a simple ‘move’ of the longword in d2 on the stack before calling ‘value’ does the trick like magic! Next, we cleaned up the stack by adding 4 to sp.

‘Message\_3’ is displayed and is followed by the sorted numbers. This is done by iterating over the memory location where the array is stored using a2, copying the number on top the stack and once again utilizing ‘value’ for displaying the number on the screen. This process continues and every time d2 is decremented by 1. Codes for comparison are in place such that as soon as d2 becomes zero – all the numbers in the array are printed – and so, we print ‘Final\_message’, restore the registers and return to the main program. Since there is no more code to run, this suggests the end for our bubble sort program.

**Testing**

The testing procedures discussed below were executed with user inputs using both the files given for writing the subroutines as well as the Lab 3 Test Program.

Part A

Before we tested out our code in Lab3a.s, as per the instructions of the lab instructor, we first commented out the codes in the main program in Lab3.s that leads to the subroutines in Lab3b.s and Lab3c.s to avoid ‘Trap” errors. Next, we downloaded our code in the board. Unfortunately, despite all careful attempts to avoid the so-called “Trap occurred” message, this is what we got. Though thorough review of our code did not reveal the problem, our Professor’s (Dr. Hai Jiang) instinct about messing up with the stack pointer when returning to the main program turned out to be right. This revelation was also an eye-opener to the errors where we did not use ‘adda.l #4, %sp’ instruction right after printing the messages in our program with the ‘pea’ command. After these necessary changes were made, our program ran smoothly. For example, it started with displaying the Welcome\_message1and then prompted us to enter a number. As expected, inputting ‘-1’ and ‘11’ were rejected with error messages and we were re-prompted to enter something valid in the range [2, 10]. Moreover, when we typed in any character other than positive or negative numbers (for example, ‘a’ and ‘$’), it would be automatically rejected by getstring. Anyway, we typed in ‘5’ next and were asked to enter the 5 numbers we wanted to be sorted. We typed in ‘5’, ‘4’, ‘3’ and ‘2’ in successive order with each prompt. However, when Message\_for\_last\_number appeared on the screen, we tried inputting ‘-1’ and ‘a’ and got back Message\_for\_negative\_number and a default error message from getstring respectively. Finally, we enter ‘1’ as the last entry. After a few seconds break, the program started all over again. This implied that our codes were being implemented error-free. We also tried out our program for bigger numbers to be sorted – 200, 350, 100000 – and still got the right results. We also tried inputting two same numbers (like two ‘3’s successively or in any other order) and our program did not respond adversely to any of these.

Part B

In modification to Lab3.s, we removed the comments such that the main program would then be able to call in the ‘Sort’ subroutine. Since, the sorted array was not supposed to be returned to the user until we implemented/tested ‘Display’, we had to enter the Debug mode to check whether our program was sorting the numbers starting at the memory location 0x2300000. Thus, we would step over every code in program and get a direct look if the codes were doing what we intended for them to do in the first place. However, we started small with an array of only 3 elements – 3, 2, 1 – entered this exact order. We noticed that our program finished the first round and switched 3 and 2 such that the array looked like – 2, 3, 1. However, after the second run, the program just replaced 3 with 1 and 1 with 1 in the third round. Thus, resulting array appeared to be – 2, 1, 1. Once again, consultation with our Professor revealed that we forgot to re-initialize the base memory address of the array at the start of the subsequent rounds. Thus, we adjusted our codes such that our subroutine would copy the memory address (0x2300000) pointed by a2 to a3. Then, when at the start of each round a2 has been modified from repeated use for switching and assigning integers to and from data registers d3 and d4, ‘Sort’ would reassign a2 to the base address of the array by copying the contents from a3 to a2. With this modification, we ran our program again (for the array 3, 2, 1) and noticed the resultant array in the memory to be displaying 1, 2, and 3 in the ascending order. We again received positive results with the 5-entry array - 5, 4, 3, 2, 1. Next, we tried to inputting two same numbers (like two 3’s successively or in any random order) for a 7-entry array and received the array sorted in the right order again where the two same numbers would appear consecutively in the ordered list. Based on these positive results, we concluded that our sorting algorithm was running error-free and moved on to code and test ‘Display’ next.

Part C

Based on our newfound knowledge from the roadblocks we encountered while writing and testing for the subroutines ‘WelcomePrompt’ and ‘Sort’, our coding in Lab3c.s appeared almost error-free when ran except when we had to start displaying the required numbers. Here we had to make use of the provided subroutine ‘value’ for which, afterwards, we did not clean up the stack. Thus, once again we had to taste the bitter “Trap occurred”. Anyway, after our Professor pointed this out, we reviewed the details of ‘value’ and added the necessary instruction ‘adda.l #4, %sp’ which would clean up the stack after the required numbers were displayed on the screen via ‘value’. Here, we initially missed the point that the given subroutine’s exit condition was the decimal value (longword) on the stack. However, after fixing this everything worked fine – the subroutine displayed the number of entries in the array, the sorted array and the message implying the end of the program. Thus, even though one error-free run was enough to verify the validity of our code, we repeated the test for arrays of sizes 3, 5 7 and 8 entries and got the desired results.

**Questions**

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

Yes, it is important to preserve registers since this enables us to perform operations with data within a subroutine. Otherwise, any values in registers used by the subroutine will be modified, possibly with large consequences on the operation of the caller program following the subroutine. Without the caller or callee preservation of registers mistakes can easily happen and register values will be overwritten without any feedback. This will make an algorithm seem to be working correctly but in reality it may potentially erase necessary information from the memory. Moreover, problems involving registers being overwritten are usually difficult to debug.

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

If a value is pushed on the stack but never poped , then the next value pushed on the stack will cause the location of the previous value to be lost. This results in a memory leak in the program. In addition, if we keep pushing values onto the stack as parameters to a subroutine and do not clean up after them, then the stack will grow infinitely. As a result we might eventually run out of memory for the stack. It is therefore necessary to clean up the stack after we push, since we only push a constant amount of values each time and then pop them off. We you pop values off the stack, the stack is returned to its previous state before the push and one can repeat the push values without worrying about memory boundaries or limitations.

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*.*

Checking the validity of an entered number could be done similarly the way our subroutine checks whether a number is positive. Along with the already known restrictions, we will now check for the ASCII equivalent of the entered character. Every input must be compared and verified to be within the acceptable range of ASCII characters. The valid range of ASCII characters would be from decimal 48 to decimal 57. Hence, each character has to be checked if it is a correct number in the ASCII range of 0-9. If at any point, we encounter a character that isn't in the ASCII range of 0-9, then the whole input is an invalid number. If we have reached the end of processing and all characters were valid (0-9), then the whole string is a valid string representation of a number. If the entered value is not between the valid range we should display an error message and re-prompt the user.

**Conclusion**

In this design project, we implemented three subroutines. In the first subroutine, WelcomePrompt, we received the necessary inputs from the user such that we had all the information to implement sorting of the provided positive number using a bubble sort program. This sorting was actually done using the codes in the second subroutine – Sort – we wrote. Finally, in Display, we prepared coding formulations that displayed the results and also indicated the end of the program. During this whole process, we gained a solid introductory understanding and experience writing subroutines (or functions) in the assembler language, learned about the various methods of feeding inputs to our programs and the slightly cryptic, painstaking and yet efficient nature of ‘push’ and ‘pop’ concepts. We also realized the importance of preservation of register contents for the proper implementation of our instructions. Lastly, we have come to develop an appreciation for writing programs by breaking it up into simpler programming constructs of subroutines, which are independent program bodies themselves and thus, can be applied ubiquitously with only a basic understanding behind their nature and applicability.

Part A Assembler Code

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

.text

.global WelcomePrompt

.extern iprintf

.extern cr

.extern value

.extern getstring

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

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* General Information \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* File Name: Lab3a.s \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Names of Students: Ishtiak Ahmed (1269389) and Mohammad Sirajee\*/

/\* (1255986) \*\*/

/\* Date: March 12, 2013 \*\*/

/\* General Description: Implementation of a subroutine called \*\*/

/\* ‘WelcomePrompt’ that prompts the user to enter the number of \*\*/

/\* entries (for to be bubble sorted) followed by the numbers for \*\*/

/\* sorting from the keyboard \*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

WelcomePrompt:

/\* Write your program here \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Preservation of register contents on top of the stack and \*/

/\* pointer assignment for the memory location 0x2300000 \*/

suba.l #60, %sp

movem.l %a0-%a6/%d0-%d7, (%sp)

movea.l #0x2300000, %a2

/\* Print the first message/string of the program \*/

pea Welcome\_message1

jsr iprintf

adda.l #4, %sp /\* Clean up the stack \*/

jsr cr

/\* Print the second message/string of the program and prompt

the user for a number \*/

/\* Check to the see if the number is in the range [2,10] \*/

Check: pea Prompt\_message

jsr iprintf

adda.l #4, %sp /\* Clean up the stack \*/

jsr cr

jsr getstring

move.l %d0, %d6

jsr cr

cmp.l #1, %d6

ble Invalid

cmp.l #11, %d6

bge Too\_Many

/\* Instructions to deal with negative, excessive and valid number \*/

/\* of entries \*/

Valid: move.l %d6, 64(%sp)

move.l %d6, %d7 /\* Use d7 as a counter\*/

bra Get\_Numbers

Invalid: pea Message\_for\_invalid

jsr iprintf

adda.l #4, %sp

jsr cr

bra Check

Too\_Many: pea Message\_for\_too\_many

jsr iprintf

adda.l #4, %sp

jsr cr

bra Check

/\* Print a message/string prompting the user for a number\*/

Get\_Numbers: pea Message\_for\_numbers

jsr iprintf

adda.l #4, %sp

jsr cr

/\* Print a message/string indicating the last entry \*/

Get\_Number: jsr getstring

move.l %d0, %d4

jsr cr

cmp.l #0, %d4

blt Negative

/\* Instructions to deal with positive and negative user entries\*/

Positive: move.l %d4, (%a2)+

sub.l #1, %d7

cmp.l #0, %d7

beq Lab\_3b

cmp.l #1, %d7 /\* Check if the counter is at the 2nd last entry \*/

bne Get\_Numbers

bra Last\_Number

Negative: pea Message\_for\_negative\_number

jsr iprintf

adda.l #4, %sp

jsr cr

bra Get\_Numbers

Last\_Number: pea Message\_for\_last\_number

jsr iprintf

adda.l #4, %sp

jsr cr

bra Get\_Number

/\* Restore the registers and go back to the main program\*/

Lab\_3b: movem.l %a0-%a6/%d0-%d7, (%sp)

adda.l #60, %sp

rts

/\*End of Subroutine \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

.data

/\*All Strings placed here \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*WelcomeLines:\*/

Welcome\_message1: .string "Welcome to the Bubble Sort Program "

Prompt\_message: .string "Please enter the number (2min-10max) of entries followed by 'enter'"

Message\_for\_invalid: .string "Invalid entry. Please enter more than 1 entry"

Message\_for\_too\_many: .string "Too many entries"

Message\_for\_numbers: .string "Please enter a number (positive only) followed by 'enter'"

Message\_for\_negative\_number: .string "Negative value entered, only positive ones accepted"

Message\_for\_last\_number: .string "Please enter the lst number (positive only) followed by 'enter'"

/\*End of Strings \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Part B Assembler Code

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

.text

.global Sort

.extern iprintf

.extern cr

.extern value

.extern getstring

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

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* General Information \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* File Name: Lab3b.s \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Names of Students: Ishtiak Ahmed (1269389) and Mohammad Sirajee\*/

/\* (1255986) \*\*/

/\* Date: March 12, 2013 \*\*/

/\* General Description: Implementation of a subroutine called \*\*/

/\* 'Sort' that performs the bubble sort algorithm\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Sort:

/\*Write your program here\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

move.l 4(%sp), %d7 /\* Number of entries is implied by %d7; \*/

move.l (%sp), 4(%sp) /\* Move the longword in (%sp) to the next memory location\*/

/\* Preservation of register contents on top of the stack and \*/

/\* pointer assignment for the memory location 0x2300000 \*/

suba.l #60, %sp

movem.l %a0-%a6/%d0-%d7, (%sp)

movea.l %a2, %a3 /\* Copy the contents of %2 to %a3 \*/

Next\_Round: move.l %a3, %a2

sub.l #1, %d7 /\* "Round" counter \*/

beq Lab\_3c

move.l %d7, %d6 /\* "Pass" counter \*/

move.l (%a2)+, %d3

move.l (%a2), %d4

/\* Compare two successive numbers in the array \*/

Compare: cmp.l %d3, %d4

bge Do\_Not\_Switch

/\*bra Switch\_With\_Next\_Position\*/

/\* Sort the numbers in the array in ascending order \*/

Switch\_With\_Next\_Position: move.l %d4, -4(%a2)

move.l %d3, (%a2)+

move.l (%a2), %d4

sub.l #1, %d6

cmp.l #0, %d6

beq Next\_Round

bra Compare

Do\_Not\_Switch: move.l (%a2)+, %d3

move.l (%a2), %d4

sub.l #1, %d6

cmp.l #0, %d6

beq Next\_Round

bra Compare

/\* Restore the registers and go back to the main program\*/

Lab\_3c: movem.l (%sp), %a0-%a6/%d0-%d7

adda.l #64, %sp

rts

/\*End of Subroutine \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

.data

/\*All Strings placed here \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*End of Strings \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Part C Assembler Code

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

.text

.global Display

.extern iprintf

.extern cr

.extern value

.extern getstring

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

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* General Information \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* File Name: Lab3c.s \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Names of Students: Ishtiak Ahmed (1269389) and Mohammad Sirajee\*/

/\* (1255986) \*\*/

/\* Date: March 12, 2013 \*\*/

/\* General Description: Implementation of a subroutine 'Display' \*\*/

/\* that displays the numbers from lowest to highest \*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Display:

/\*Write your program here\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Preservation of register contents on top of the stack and \*/

/\* pointer assignment for the memory location 0x2300000 \*/

suba.l #60, %sp

movem.l %a0-%a6/%d0-%d7, (%sp)

movea.l 64(%sp), %a2 /\* Copy the starting address of the sorted array to %a2 \*/

move.l 68(%sp), %d2 /\* Copy the number of entries from the stackt to %d2 \*/

/\* Display Message1 \*/

pea Message\_1

jsr iprintf

adda.l #4, %sp /\* Clean up the stack \*/

jsr cr

/\* Display the message of the number of entries\*/

pea Message\_for\_entry\_number

jsr iprintf

adda.l #4, %sp /\* Clean up the stack \*/

/\* Display the number of entries \*/

move.l %d2, -(%sp)

jsr value

adda.l #4, %sp

jsr cr

/\* Display Message3 \*/

pea Message\_3

jsr iprintf

adda.l #4, %sp

jsr cr

/\* Display the sorted numbers in ascending order \*/

Print\_Sorted\_Numbers: move.l (%a2)+, -(%sp)

jsr value

adda.l #4, %sp

jsr cr

sub.l #1, %d2

bne Print\_Sorted\_Numbers

/\* Display the last message \*/

pea Final\_message

jsr iprintf

adda.l #4, %sp

jsr cr

jsr cr

/\* Restore the registers and go back to the main program\*/

Back\_To\_Main\_Program: movem.l %a0-%a6/%d0-%d7, (%sp)

adda.l #60, %sp

move.l (%sp), 8(%sp)

add.l #8, %sp

rts

/\*End of Subroutine \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

.data

/\*All Strings placed here \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Message\_1: .string "The numbers are sorted with bubblesort algorithm"

Message\_for\_entry\_number: .string "The number of entries was "

Message\_3: .string "Sorted from smallest to biggest, the number are: "

Final\_message: .string "Program ended"

/\*End of Strings \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/