ECE 212 Lab - Introduction to Microprocessors
Department of Electrical and Computer Engineering
University of Alberta

Lab 3: Introduction to Subroutines

Student Name	Student
Arun Woosaree	XXXXXXX
Navras Kamal	1505463

Contents

1 Introduction **2**
1.1 Part A . . . 2
1.2 Part B . . . 2
1.3 Part C . . . 2

2 Design **3**
2.1 Part A . . . 3
2.2 Part B . . . 4
2.2.1 Sample Calculations . . . 6
2.3 Part C . . . 7

3 Testing **8**
3.1 Part A . . . 8
3.2 Part B . . . 9
3.3 Part C . . . 9

4 Questions **11**
4.1 Question 1 . . . 11
4.2 Question 2 . . . 11
4.3 Question 3 . . . 11

5 Conclusion **12**

6 Appendix **13**
6.1 Part A Assembler Code . . . 13
6.2 Part A Flowchart Diagram . . . 18
6.3 Part B Assembler Code . . . 19
6.4 Part B Flowchart Diagram . . . 23
6.5 Part C Assembler Code . . . 24
6.6 Part C Flowchart Diagram . . . 28

7 Marking Sheet **29**

8 Marking Sheet **29**

1 Introduction

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

1.1 Part A

In part A, ...

1.2 Part B

For part B, ...

1.3 Part C

FILLINTHEINTRO

The purpose of this lab is to learn and test with the Assembly language in a hands on environment in order to solidify the concepts learned in class and to improve our skill in the language. In addition, we will be learning how to handle the Netburner ColdFire boards directly, manipulating the contents of their memory and data structures. Finally, we are going to learn how to work inside the Eclipse IDE environment and how to properly use the powerful tools that come alongside it.

The code will be developed for the Netburner ColdFire Platform, which has some parameters that should be kept in mind throughout testing. There are multiple Data and Address registers, and the memory is indexed by hexadecimal codes. The data and the stored locations can each be modified directly, values can be compared and the code can branch into different sections depending on the values of the CCR (Condition Code Register) bits, which store information about the outcome from the last comparison or valid operation, such as if a value is negative or zero. These will be used to execute code conditionally.

The lab will be split into two sections, each with a different goal but with similar implementations. For one part we will be taking in an ASCII value and if the character it represents is a character included in the symbols for hexadecimal numbers then that hexadecimal value is output, otherwise it returns an error. For the second part an ASCII value is taken in, and if the character it represents is a letter in the English language (A-Z) then the ASCII code for the character in the opposite case is output. Thus, valid uppercase English letters are converted to their lowercase equivalents in ASCII and vice versa.

These experiments will introduce implementing high level programming practices of loops, if - then - else statements, using the Assembly language. More specifically, this will introduce the movement of memory and data to and from different parts of the Netburner chip, using techniques such as referencing memory addresses and copying data to local data registers. The debugger tools of the IDE will be used to closely examine this movement and to analyze all changes to the data in order to solve issues in development as well as to test the code. This is all building off the concepts explored in Lab 0.

The computing science practice of Pair Programming was also introduced, where two people develop and test code in tandem. The partners are divided into the Driver and the Navigator. In this structure the Driver is the one responsible for the physical typing of the code into the computer, and the Navigator reviews this code and clarifies the meaning of each passage in order to find bugs faster and to improve efficiency in testing. The two partners should communicate constantly and switch in order to maximize the efficiency of this working model. This will not only decrease time needed for development but it will also improve the quality of code from each partner.

2 Design

2.1 Part A

For the first part of the lab, we begin by writing our own subroutine. Some code was already supplied such that:

- Stack Entry Condition =
  1. Space allocated for the number of entries on stack (long word)
  2. Space allocated for the divisor number on stack (long word)
- Stack Exit Condtion =
  1. Number of entries on stack (long word)
  2. Divisor number on stack (long word)

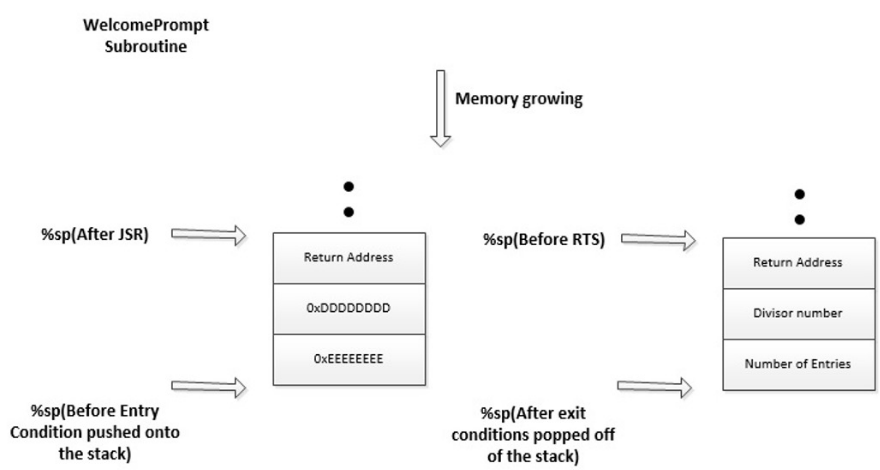


Figure 1: Visualization of the stack for Part A

We started by editing *Lab3a.s*, and chose to back up address registers *a2*-*a6* and the data registers d2-d7. Since the stack pointer was *a7*, this was accomplished by subtracting 44 from the stack pointer, then using a MOVEM.L command to push the values onto the stack. Then at the end, just before the RTS command in the subroutine, we move the values we backed up to the registers as they were before running the subroutine. Address register *a2* was chosen to hold the memory location of where the valid numbers which were entered would go, which was 0x2300000. Next, at the very end of the file, we defined several messages as strings that would be used as prompts. Labels are used to reference these strings, which are used for prompting the user in the MTTTY serial monitor to enter relevant data for each part.

With the initial setup for this subroutine completed, we moved on to actually coding the prompt messages for the program. We begin by pushing the Welcome Prompt onto the stack, and using the provided *iprintf* and *cr* subroutines to print the Welcome Message and a carriage return for the user. Once the message was displayed, we immediately clean up the stack by adding 4 to the stack pointer, since the address of the message was pushed onto the stack, and addresses take up 4 bytes of memory. For every message printed onwards, we immediately clean up the stack after the message is printed. We then branch to a label where the user is prompted to enter the number of entries. The subroutine getstr, which was provided captures input from the serial monitor, and puts it into data register *d0*. For this part, the only valid entries are the numbers 3-15, with anything else entered being rejected. This was accomplished by comparing the input to the number 15. If the number was greater than 15. we branch to a label to warn the user that an invalid entry was entered, and then the program returns to the label which prompts the user to enter the number of entries. Similarly, if the value entered was less than 3, the same thing would occur. If the entry was valid, we put the numentries value that the user entered in the spot reserved on the stack for us. This value was also copied to *d7*, which is later used as a counter for when the user enters the numbers. As a feature for the user, if the input was accepted, the value is printed to the serial monitor for the user to see. This is done in a similar method to how the messages are printed onto the stack.

In a very similar fashion, the user is then prompted to enter the divisor number. Here, the accepted values were from 2-5, with the data validation checks happening in almost exactly the same was as outlined above for entering the number of entries. The divisor also went into a spot on the stack, which was reserved for us.

Next, we enter a loop, where the user is prompted to enter numbers, until the user enters as many numbers as the user declared when prompted to enter the number of entries. The loop itself loops numentries-1 times, since there's a different prompt message for the last number. As mentioned earlier, we use *d7* as our counter, since numentries was copied to this data register. The user then goes through the same process of entering numbers, and is prompted to re-enter the number if it is not positive. This is done similarly to above, in that the number entered is compared to the number 1, and if the number entered is less than 1, we branch to a label where the user is warned about the mistake, and then branches back to continue the loop. If the number is accepted, then it is printed to the serial monitor, and then the number entered is moved to the address location pointed to by *a2*, and *a2* is post incremented. On each successful loop iteration, the number 1 is subtracted from the counter *d7*, until the counter is 1.

At this point, the user needs to enter one last number, for which there is an appropriate prompt, and the data validation is the exact same as above, with only positive values being accepted. If the number is valid, then it is printed to the serial monitor, and moved to the memory location where *a2* points.

Finally, we reach the part where the backed up registers on the stack are restored, and the subroutine hits the RTS command.

2.2 Part B

In Part B, another subroutine is written, which computes some stats on the numbers entered. Similar to Part A, some code was provided such that the stack entry and exit conditions are as follows:

- Stack Entry Condition =
  1. Space allocated for how many numbers are divisible by the divisor (long word)
  2. Number of entries (long word) on stack
  3. Divisor number (long word) on stack
- Stack Exit Condtion =
  1. How many numbers are divisible by the divisor (long word) on stack
  2. Number of entries on stack (long word)
  3. Divisor number on stack (long word)

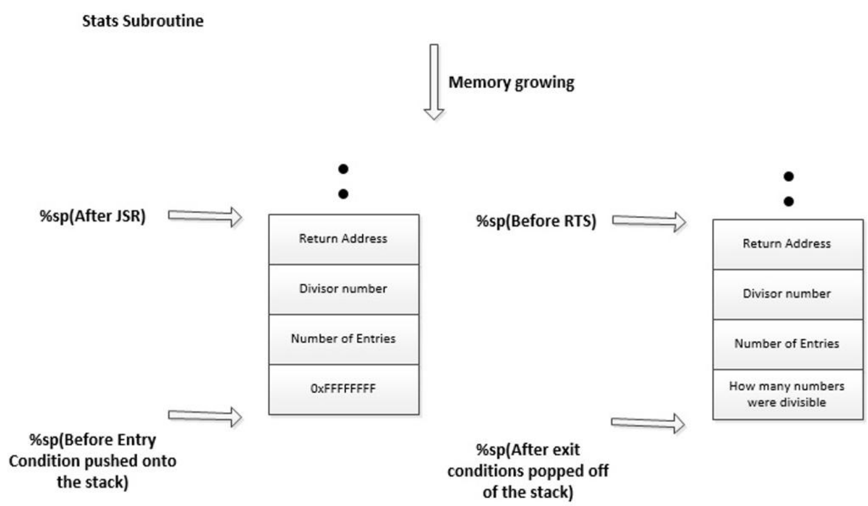


Figure 2: Visualization of the stack for Part B

We begin by backing up the registers $a0 - a6$ and $d0 - d7$ by subtracting 60 from the stack pointer, and then using the *movem* command to push those values onto the stack. Similarly, at the end of the subroutine just before rts, we restore the values by using *movem* to restore the values, then add 60 to the stack pointer before returning to the previous subroutine.

We begin by reading the numentries and divisor off of the stack, which were put there by the subroutine in Part A. The addresses 0x2300000 and 0x2310000 were loaded into address registers *a2* and *a3* respectively. *a2* kept track of the location were the numbers were entered, and *a3* kept track of where the min, mean, max, and divisible numbers would be output.

To begin calculating the minimum number, we copy the numentries to another data register to use as a counter. The first number is read by indirectly addressing *a2* with a post increment, and moved to a data register which will hold the final minimum value. Inside a loop, which loops numentries-1 times, (since the first number was already read), the next number is read by indirectly addressing *a2* with a post increment, and moved into a temporary data register, where it is then compared to the data register which is supposed to hold the minimum value. If the temporary value is less than the minimum value, we move the temporary value to the data register which holds the minumum value. If the temporary value is greater than the current minimum, then the loop continues. At the end of every loop iteration, the loop counter is decremented by 1, and compared to the number 1, since it loops numentries-1 times. Once this loop is complete, the minumum value is stored in data register *d5*, so we move it to the memory location pointed at by *a3* and post increment it.

The maximum number is calculated in a very similar manner compared to how the min was calculated. First, *a2* is reset to 0x2300000, and the loop counter is reset to numentries, since we post incremented *a2* when finding the min, and we decremented the counter when going through the numbers again to find the min. Then, we go through a loop that is almost the same as for calculating the minumum number. except that the current maximum number is held in one data register, and a temporary value is read and compared to the maximum number. If the temporary number is greater than the current maximum, then the temporary number becomes the maximum, and by the end of the loop, the maximum value is stored in data register *d6*, so we move it to the memory location pointed at by *a3* and post increment it.

To find the mean, address register *a2* was reset to 0x2300000, and the loop counter was reset to numentries again. A data register was used to store the cumulative sum of the numbers, which was accomplished by first clearing the data register, then looping over the list of numbers entered (pointed at by *a2*), and adding each number to the sum. This was done by indirectly addressing *a2* with a post increment to copy the value to a data register, then adding that value to our cumulative sum. At the end of the loop, we have the cumulative sum, so to find the mean, we simply divide the cumulative sum by the numentries by using *divs.l*. When the division was complete, the mean was stored in data register *d4*, so so we move it to the memory location pointed at by *a3* and post increment it.

Finally, to find the divisible numbers, we once more reset *a2* to 0x2300000, and the loop counter to numentries. We go through one last loop, where a number is read by indirectly addressing *a2*, moving it to a data register, and then using *divs.w* to divide the number by the divisor, since *divs.w* provides the remainder stored in the 16 most significant bits. If the number is divisible, it should not have a remainder, so we check the remainder by bit shifting the division result 16 bits to the right, and check if the remainder is 0. If so, a counter which holds the number of divisible numbers is incremented by 1, and the divisible number is moved to the location pointed at by *a3* and post incremented. The loop then continues until all the numbers have been processed. At the end of the loop, the number of divisible entries is stored in data register *d7*, so we move it to the spot reserved on the stack for us.

We then reach the end of the subroutine, where we restore the backed up register values on the stack, add 60 to the stack pointer, and return to the previous subroutine.

2.2.1 Sample Calculations

Numbers entered: 1,2,3,4,5
Numentries: 5
Divisor: 2

Min: 1

Max: 5
Mean: $\frac{1+2+3+4+5}{5} = 3$
Numdivisible: 2
Divisible numbers: 2,4

2.3 Part C

In Part C, the last subroutine is written, which prints the stats on the numbers entered. Similar to the previous parts, some code was provided such that the stack entry and exit conditions are as follows:

- Stack Entry Condition =
    1. How many numbers were divisible by the divisor
    2. Number of entries (long word)
    3. Divisor number (long word)
- Stack Exit Condtion =
    1. How many numbers are divisible by the divisor
    2. Number of entries (long word)
    3. Divisor number (long word)

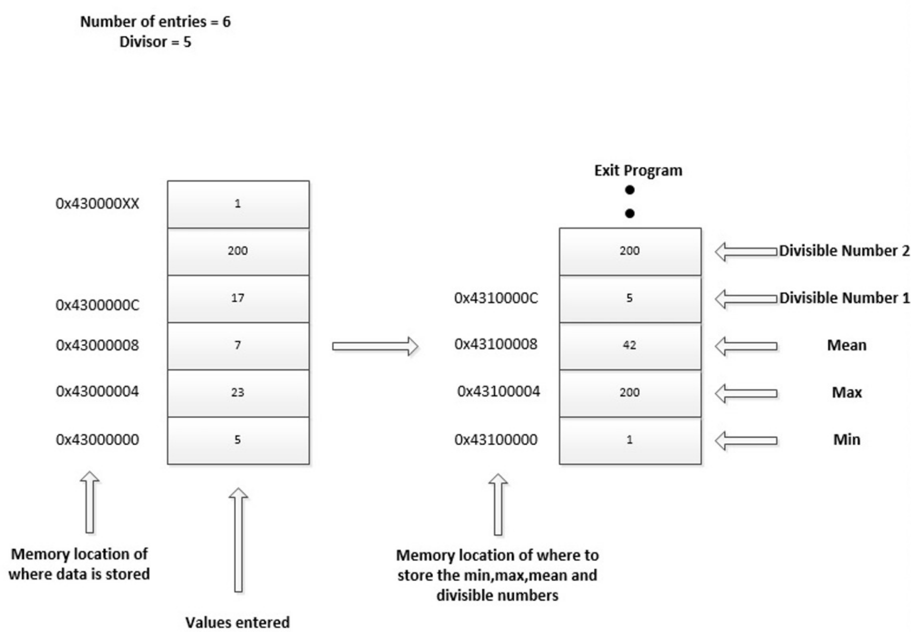


Figure 3: Visualization of the stack for Part A

Just like in part B, we begin by backing up the registers $a0 - a6$ and $d0 - d7$ by subtracting 60 from the stack pointer, and then using the *movem* command to push those values onto the stack. Similarly, at the end of the subroutine just before rts, we restore the values by using *movem* to restore the values, then add 60 to the stack pointer before returning to the previous subroutine. We begin by loading 0x2300000 into *a2*, 0x2310000 into *a3*, and the divisor, numentries, and numdivisible are read from the stack and moved into data registers.

At the very end of the program, we define strings just like in Part A, which are printed for the user to read. We begin by first printing the number of entries. A message is printed by pushing the address of a string defined earlier onto the stack, and using *iprintf*. Next, the actual number of entries is pushed onto the stack and printed to the serial monitor using the subroutine *value*. The stack is also cleaned up by adding to the stack pointer. The user is then prompted that the numbers will be printed, which is done in a very similar manner, using *iprintf*. Then, a loop iterates over the numbers, which are indirectly addressed with *a2* with post increment, and printed with the subroutine *value*. The data register holding the numentries is decremented at the end of each iteration of the loop, until it reaches 0, which is when all the numbers have been printed. The min, max, mean, and number of divisible numbers are all printed in a similar manner to how the numentries was printed, with a message printed first using *iprintf*, and then the value being printed with the subroutine *value*. Whenever something is printed, the stack is immediately cleaned up afterwards by adding 4 to the stack pointer whenever a value is printed.

3 Testing

3.1 Part A

Initially, we visually tested our code by using the debugger in Eclipse IDE. While stepping through the code, we would check the values at relevant memory locations, and the data and address registers. When the bugs were ironed out, we went on to the next phase of testing. Our code was tested using the provided *Lab1Test.s* file. More specifically, this program was moved into the project folder, downloaded to the ColdFire microcontroller, and the MTTTY serial monitor was loaded to monitor the expected output. Our code was further tested by replacing the 'DataStorage.s' file with the other variants provided named: *DataStorage1.s*, *DataStorage2.s*, and *DataStorage3.s*. Finally, our program, which produced the correct output, was verified by a lab TA.

```
Testing Subroutines. Choose from the menu below
1 - Test First subroutine
2 - Test Second subroutine
3 - Test Third subroutine
4 - Test All subroutine
1
Testing 1st subroutine.
Welcome to our amazing statistics program
Please enter the number (3min-15max) of entries followed by enter
5
Please enter the divisor (2min-5max) followed by enter
2
Please enter a number(positive only)
5
Please enter a number(positive only)
Invalid entry, please enter proper value.
Please enter a number(positive only)
4
Please enter a number(positive only)
3
Please enter a number(positive only)
2
Please enter the last number (positive only)
1
The Number of entries is 5
The Divisor number is 2
```

Figure 4: MTTTY output when testing our Part A solution

3.2 Part B

The procedure for testing our code for part B was very similar to the process described above in Part A. We visually inspected our code in the Eclipse IDE, used the Eclipse debugger to step through our code, and monitor relevant memory addresses and registers. When we were confident that we had a working solution, we used the provided files *Lab1Test.s*, and the *DataStorage*.s* files to verify our solution by downloading the program to the ColdFire microcontroller, and monitoring the output in MTTTY. Finally, our solution was verified by a lab TA.

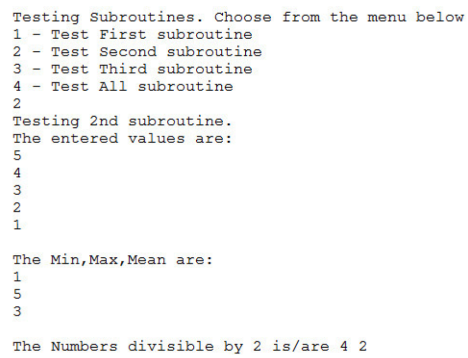


Figure 5: MTTTY output when testing our Part B solution

3.3 Part C

The procedure for testing our code for part C was very similar to the process described above in Part A. We visually inspected our code in the Eclipse IDE, used the Eclipse debugger to step through our code, and monitor relevant memory addresses and registers. When we were confident that we had a working solution, we used the provided files *Lab1Test.s*, and the *DataStorage*.s* files to verify our solution by downloading the program to the ColdFire microcontroller, and monitoring the output in MTTTY. Finally, our solution was verified by a lab TA.

```
The Numbers divisible by 3 is/are 24 45 63 18
Testing Subroutines. Choose from the menu below
1 - Test First subroutine
2 - Test Second subroutine
3 - Test Third subroutine
4 - Test All subroutine
3
Testing 3nd subroutine.

The number of entries was: 10

The entered number(s) were:
32
24
17
14
45
63
10
19
18
5

Min number: 5

Max number: 63

Mean number: 24

There are 4 number(s) divisible by 3
24
45
63
18
```

Figure 6: MTTTY output when testing our Part B solution

```
582
Please enter a number(positive only)
54
Please enter a number(positive only)
80211
Please enter the last number (positive only)
69

The number of entries was: 10

The entered number(s) were:
89
51
56
1000
150
16
582
54
80211
69

Min number: 16

Max number: 80211

Mean number: 8227

There are 6 number(s) divisible by 2
56
1000
150
16
582
54
The stack at beginning is set at SP = 0x200027C8
```

Figure 7: MTTTY output when testing our Part B solution

4 Questions

4.1 Question 1

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

A: Yes. This would cause a problem, because there would be no way to exit the program, so the program would keep reading data and moving the converted values to memory locations, until the program attempts to read or write to a memory location that is restricted or non-existant. This would then cause the program to crash.

4.2 Question 2

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

A: Assuming the data range would be fixed and hardcoded into the Assembly code it would be possible to write the max range (ie. 10) into an unused data register such as %d3. Then, instead of checking for the enter code on each iteration where there is an invalid value, we could check the value stored in %d3 before checking the validity of the value stored in the current memory address. If %d3 is zero then jump to the **end** label, breaking the loop. Another way to do it would be again to assume that the number of iterations is fixed and the size of data being checked is fixed as being long-words would be to check the value of the memory address stored in (%a1) after each iteration before returning to the beginning of the loop and after the memory addresses have been incremented. If at that point the memory address is [(initial memory address 0x2300000) + (0x4 * N)], where N is the number of desired iterations (this value would be hardcoded, this is just a general case), then jump to the **end** label, breaking the loop. This way would be more memory efficient as it does not require an additonal data register and the modification of a counter value. Thus it isn't always necessary, though it is usually good practice regardless

4.3 Question 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.

"Hello Students, In question no. 3, you are asked to answer the following: You have to describe how you are going to check that the entered number is valid or not. For example, one entry is say 409 and another is 4h9. Here, 4h9 is a wrong number. Now please explain how are you going to check that? Thanks."

A: Assuming the data range would be fixed and hardcoded into the Assembly code it would be possible to write the max range (ie. 10) into an unused data register such as %d3. Then, instead of checking for the enter code on each iteration where there is an invalid value, we could check the value stored in %d3 before checking the validity of the value stored in the current memory address. If %d3 is zero then jump to the **end** label, breaking the loop. Another way to do it would be again to assume that the number of iterations is fixed and the size of data being checked is fixed as being long-words would be to check the value of the memory address stored in (%a1) after each iteration before returning to the beginning of the loop and after the memory addresses have been incremented. If at that point the memory address is [(initial memory address 0x2300000) + (0x4 * N)], where N is the number of desired iterations (this value would be hardcoded, this is just a general case), then jump to the **end** label, breaking the loop. This way would be more memory efficient as it does not require an additonal data register and the modification of a counter value.

5 Conclusion

This lab demonstrated how to perform operations and modify data while moving it around using the Assembly language for the ColdFire architechture. In addition, the lab improved our understanding of the debugger software, a very powerful tool in the development of this kind of code. The main issue we found was related to the hardware itself, as there was some instances where the code did not execute properly and the board itself needed to be reset. The other issue we faced was mostly around getting used to the software and the workflow in the Eclipse IDE and the debugger. Once we understood the ways to use the tools we found our workflow sped up considerably, as we were able to check step by step and find bugs at the source. The last issue we had was with the syntax of the code, but that was solved quickly by reading over documentation and with the help of the TAs. Overall the lab went smoothly and has indeed succeeded at the goals of improving our familiarity and skill with the Netburner ColdFire system, Assembly code and Pair Programming practices.

6 Appendix

6.1 Part A Assembler Code

```
/* DO NOT MODIFY THIS --------------------------------------------------*/
.text
.global WelcomePrompt
.extern iprintf
.extern cr
.extern value
.extern getstring
/*----------------------------------------------------------------------*/

/****************************************************************************/
/* General Information ********************************************/
/* File Name: Lab3a.s **********************************************/
/* Names of Students: Arun Woosaree and Navras Kamal          **/
/* Date: March 5, 2018                                         **/
/* General Description:                                        **/
/* Takes in values with prompts for the statistics calculator  **/
/****************************************************************************/
WelcomePrompt:
/*Write your program here****************************************/

/* getstr stores value in d0*/
/* iprintf pops last thing on stack and prints it */

/* allocate 44 bytes because we are using up to 11 registers*/
/* 11*44 = 44 bytes*/
suba.l #44, %sp

/* back up register contents  onto the stack*/
movem.l %a2-%a6/%d2-%d7, (%sp)

lea 0x2300000, %a2

/*-------------------------------------*/
pea WelcomeMessage
/*print the welcome message*/
jsr iprintf
jsr cr
/*clean up stack*/
addq.l #4 , %sp
bra numentries
/*-------------------------------
```

```
________________________________________
/*--------------------------------------*/
failnumentries:
pea INVALIDENTRY
jsr iprintf
jsr cr
/*clean up stack*/
addq.l #4 , %sp
bra numentries

numentries:
/*prompt number of entries*/
pea Plztellusnumentries
jsr iprintf
jsr cr
/*clean up stack*/
addq.l #4, %sp
jsr getstring


/*sanitize input*/
cmp.l #15, %d0
bgt failnumentries
cmp.l #3, %d0
blt failnumentries

/*success, replace value on stack*/
move.l %d0, 52(%sp)

/*counter used for loop*/
move.l %d0, %d7

/*print what user entered*/
move.l %d0, -(%sp)
jsr value
jsr cr
addq.l #4, %sp
bra divisor
/*-----------------------------------------------------
------------------------------------------------------------
-------------------------------------------------------------

/*-----------------------------------------------------*/
faildivisor:
pea INVALIDENTRY
jsr iprintf
jsr cr
```

```
/*clean up stack*/
addq.l #4 , %sp

divisor:
/*divisor*/
pea Plztellusdivisor
jsr iprintf
jsr cr
/*cleanup stack*/
addq.l #4, %sp
/*get the string*/
jsr getstring
/*sanitize input*/

cmp.l #5, %d0
bgt faildivisor
cmp.l #2, %d0
blt faildivisor

/*success, replace value on stack*/
move.l %d0, 48(%sp)

/*print what user entered*/
move.l %d0, -(%sp)
jsr value
jsr cr
addq.l #4, %sp

bra loopvals
/*------------------------------------------------------------
--------------------------------------------------------------
--------------------------------------------------------------

/*-------------------------------------------------------------*/
failloopvals:
pea INVALIDENTRY
jsr iprintf
jsr cr
/*clean up stack*/
addq.l #4 , %sp

loopvals:
/*get first n-1 numbers*/
pea Plzenternumber
jsr iprintf
jsr cr
/*clean up stack*/
```

```
addq.l #4, %sp

/*get the string and push onto stack*/
jsr getstring
/*sanitize input*/
/*positive only*/
cmp.l #1, %d0
blt failloopvals
/*success,  move to memory location*/
move.l %d0, (%a2)+
sub.l #1, %d7


/*print what user entered*/
move.l %d0, -(%sp)
jsr value
jsr cr
addq.l #4, %sp

cmp.l #1, %d7
beq lastnum
bra loopvals
/*--------------------------------------------------------------------------
------------------------------------------------------------------------------------
------------------------------------------------------------------------------------
/*-----------------------------------------------------------------------------*/
faillastnum:
pea INVALIDENTRY
jsr iprintf
jsr cr
/*clean up stack*/
addq.l #4 , %sp

lastnum:
/*get the last number*/
pea Plzenterlastnum
jsr iprintf
jsr cr
/*clean up stack*/
addq.l #4, %sp

/*get the string and push onto stack*/
jsr getstring
/*sanitize input*/

cmp.l #1, %d0
```

```
blt faillastnum

/*success, move to memory location*/
move.l %d0, (%a2)+

/*print what user entered*/
move.l %d0, -(%sp)
jsr value
jsr cr
addq.l #4, %sp

/* restore values */
movem.l (%sp), %a2-%a6/%d2-%d7
add.l #44, %sp

/* return to original program */
rts

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

/*All Strings placed here ************************************************/
.data

WelcomeMessage:
.string "Welcome to our amazing statistics program"

Plztellusnumentries:
.string "Please enter the number (3min-15max) of entries followed by enter"

Plztellusdivisor:
.string "Please enter the divisor (2min-5max) followed by enter"

Plzenternumber:
.string "Please enter a number(positive only)"

Plzenterlastnum:
.string "Please enter the last number (positive only)"

INVALIDENTRY:
.string "Invalid entry, please enter proper value."
/*End of Strings ***********************************************/
```

6.2 Part A Flowchart Diagram

6.3 Part B Assembler Code

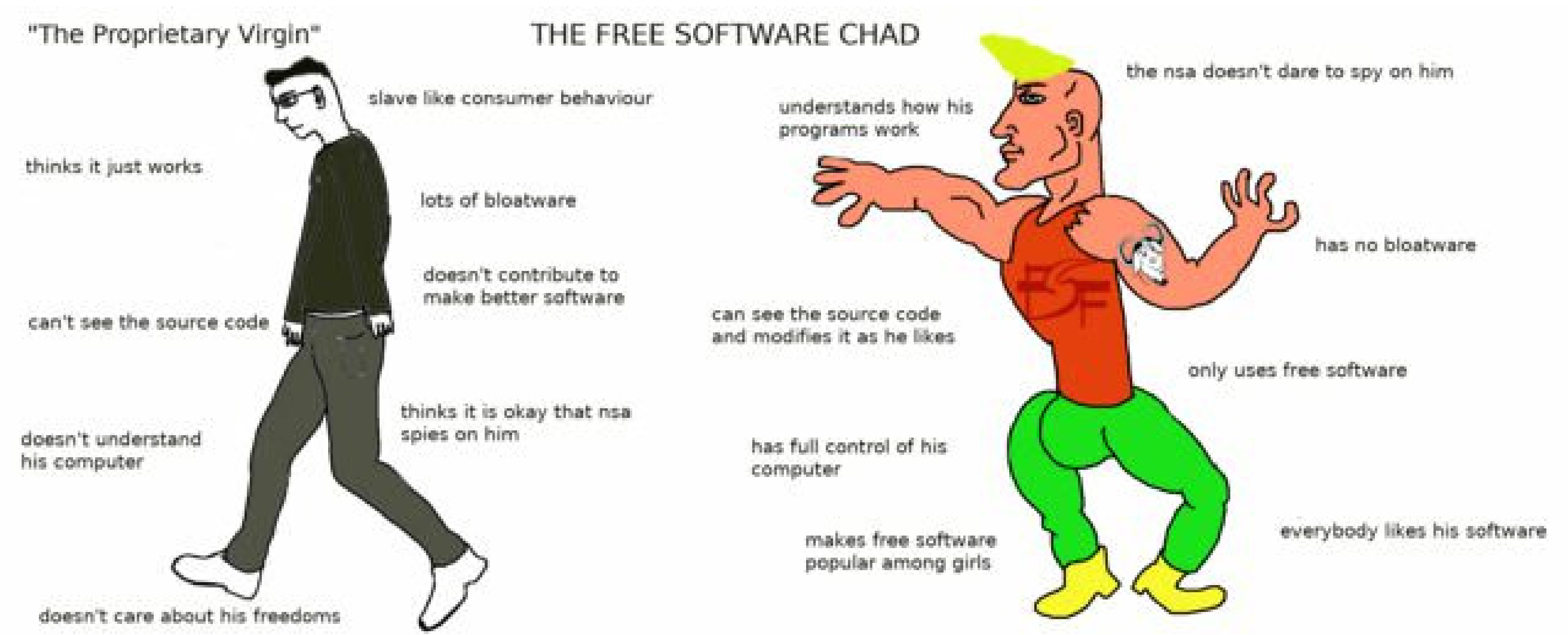
```
/* DO NOT MODIFY THIS ------------------------------------------------*/
.text
.global Stats
.extern iprintf
.extern cr
.extern value
.extern getstring
/*-----------------------------------------------------------------------*/

/*****************************************************************/
/* General Information ******************************************/
/* File Name: Lab3b.s ***********************************/
/* Names of Students: Arun Woosaree and Navras Kamal **/
/* Date: March 5, 2018                                         **/
/* General Description:                                        **/
/* Computes stats on the numbers entered (min mean max)        **/
/*****************************************************************/
Stats:
/*Write your program here*****************************************/

/* backup values */
suba.l #60, %sp
 movem.l %a0-%a6/%d0-%d7, (%sp)

/*divisor is at 4sp
jsr value
jsr cr
*/
/*numentries is at 8sp*/
/*
move.l 68(%sp), -(%sp)
jsr value
jsr cr
addq.l #4, %sp
*/
move.l 64(%sp), %d0 /* divisor */
move.l 68(%sp) ,%d1 /* numentries */

lea 0x2300000, %a2 /* numbers are here */
lea 0x2310000, %a3 /*divisible numbers stored here*/

move.l %d1, %d7 /*counter*/
move.l (%a2)+, %d5 /*first number*/
findmin:
```

```
        loopmin:
        move.l (%a2)+, %d6
        cmp.l %d5, %d6
                bge nochangemin
                move.l %d6, %d5 /* new min */
        nochangemin:
        subq.l #1, %d7
        cmp.l #1, %d7
        bne loopmin
/* min is stored in d5*/
move.l %d5, (%a3)+


/*
move.l %d5, -(%sp)
jsr value
jsr cr
addq.l #4, %sp
*/

lea 0x2300000, %a2 /* numbers are here */

move.l %d1, %d7 /*counter*/
move.l (%a2)+, %d6 /*first number*/
findmax:
        loopmax:
        move.l (%a2)+, %d4
        cmp.l %d6, %d4
                ble nochangemax
                move.l %d4, %d6 /* new max */
        nochangemax:
        subq.l #1, %d7
        cmp.l #1, %d7
        bne loopmax
/*max is stored in d6*/
move.l %d6, (%a3)+
/*
move.l %d6, -(%sp)
jsr value
jsr cr
addq.l #4, %sp
*/


findmean:
        clr.l %d4
```

```
        lea 0x2300000, %a2 /* numbers are here */
        move.l %d1, %d7 /*counter*/
        loopaddvalues:
        move.l (%a2)+, %d3
        add.l %d3, %d4
        subq.l #1, %d7
        cmp.l #0, %d7
        bne loopaddvalues
        /* sum in d4*/

        /*divide d4 by numentries*/
        divs.l %d1, %d4

        /* mean should be in d4*/
        move.l %d4, (%a3)+
/*move.l %d4, -(%sp)
jsr value
jsr cr
addq.l #4, %sp
*/


finddivisible:
        lea 0x2300000, %a2 /* numbers are here */

        /* at this point, d6, d5 , d4 store values and therefore should not be touched*
        /* this is the last time using the numentries, so will modify d1 directly */
        /* ok so d7, d3, and d2 are left free to use*/
        clr.l %d7
        loopdivisible:
        move.l (%a2)+, %d3
        /* copy d3*/
        move.l %d3, %d4

        /* check if it's divisible, divisor is d0*/
        divs.w %d0, %d3
        /* get remainder*/
        lsr.l #8, %d3
        lsr.l #8, %d3
        /*if 0, divisible*/
        bne notdivisible
        /*it is divisible, move copy*/
        move.l %d4 , (%a3)+
        /*increment counter for divisible numbers*/
        addq.l #1, %d7
```

```
        notdivisible:
        subq.l #1, %d1
        cmp.l #0, %d1
        bne loopdivisible

        /* d7 holds number of divisible numbers*/

/*move.l %d7, -(%sp)
jsr value
jsr cr
addq.l #4, %sp
*/

move.l %d7, 72(%sp)

movem.l (%sp), %a0-%a6/%d0-%d7
adda.l #60 , %sp
rts
/*End of Subroutine ***************************************************/
.data
/*All Strings placed here ***************************************************/



/*End of Strings ***************************************************/
```

6.4 Part B Flowchart Diagram



6.5 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: Arun and Navras                          **/
/* Date: March 16 2018                                         **/
/* General Description:                                        **/
/*                                                             **/
/*****************************************************************/
Display:
/*Write your program here****************************************/
/* backup values */
suba.l #60, %sp
movem.l %a0-%a6/%d0-%d7, (%sp)


lea 0x2300000, %a2 /* numbers are here */
lea 0x2310000, %a3 /*divisible numbers stored here*/

move.l 72(%sp) , %d7 /* divisor*/
move.l 76(%sp), %d6/* numentries*/
move.l 80(%sp), %d5/*numdivisible*/

jsr cr
/* tell user numentries*/
pea numentries
jsr iprintf
move.l %d6, -(%sp)
jsr value
jsr cr
jsr cr
adda.l #8, %sp

/* print numbers */
pea numbers
jsr iprintf
```

```
adda.l #4, %sp
jsr cr

loopnumbers:
move.l (%a2)+, -(%sp)
jsr value
jsr cr
adda.l #4, %sp
subq.l #1, %d6
bne loopnumbers

jsr cr
/*min*/
pea min
jsr iprintf
move.l (%a3)+, -(%sp)
jsr value
jsr cr
jsr cr
adda.l #8, %sp

/*max*/
pea max
jsr iprintf
move.l (%a3)+, -(%sp)
jsr value
jsr cr
jsr cr
adda.l #8, %sp

/*mean*/
pea mean
jsr iprintf
move.l (%a3)+, -(%sp)
jsr value
jsr cr
jsr cr
adda.l #8, %sp

/*print num divisible*/
pea numdivisible
jsr iprintf
move.l %d5, -(%sp)
jsr value
pea numdivisible2
jsr iprintf
```

```
move.l %d7, -(%sp)
jsr value
jsr cr
adda.l #16, %sp

loopdivisible:
move.l (%a3)+, -(%sp)
jsr value
jsr cr
adda.l #4, %sp

subq.l #1, %d5
bne loopdivisible

/*restore values*/
movem.l (%sp), %a0-%a6/%d0-%d7
adda.l #60 , %sp
rts

/*End of Subroutine ************************************************/
.data
/*All Strings placed here ************************************************/
numentries:
.string "The number of entries was: "

numbers:
.string "The entered number(s) were: "

min:
.string "Min number: "

max:
.string "Max number: "

mean:
.string "Mean number: "

numdivisible:
.string "There are "
numdivisible2:
.string " number(s) divisible by "

endprogram:
.string "End of program"
```

```
/*End of Strings **************************************************/
```

6.6 Part C Flowchart Diagram

8 Marking Sheet