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

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

1.3 Part C

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, the address register a1 was chosen to initially point to memory location 0x2300000, which is the starting point of where the input data was stored. We used this address register to keep track of the memory location of the next long word of data to be read as an input to our program. Even though one memory location is capable of storing one ASCII character, in this lab, 4 memory locations were used to store one ASCII character, as specified in the lab manual. Next, a2 was selected to initially point at the memory location 0x231000, which is the starting location of where our output for the converted values was. This register was used to keep track of the memory location of where the next long word of our converted data would go. The data register d2 was chosen to temporarily store data so we could do comparisons and process the input data. The *SetZeros.s* and the *DataStorage.s* files, which were provided, were used to initialize memory contents.

We started with a loop branch that served as our main looping function. This loop first starts by moving data from a memory location pointed to by a1, to the data register d2 so that we could start comparing the input data to known ASCII values. In the first comparison, the input character is compared to '0x0D', which is the ASCII code for the 'Enter' key. This code is meant to signal the end of the program, so if the input was the ASCII code for 'Enter', we branched to a label that would end our program. The next step was to determine if the input character was valid. For Part A, an input character was valid if it was an ASCII character from 0-9, A-F, or a-f. In other words, the data was accepted if it was in the following ranges: 0x30-0x39, 0x41-0x46, or 0x61-0x66. If the input character was invalid, we brached to a label named 'err', which would put the error code '0xFFFFFFFF' in the memory location pointed to by a2, which keeps track of where our output data goes.

If the input character was valid, we had three branches to take care of each of the accepted ranges of input. For example, in the branch that took care of the input range A-F, the ASCII value of 'A' was subtracted from the input value, and the difference was added to the hex value 0xA. The converted value is then moved to the memory location pointed to by a2, which keeps track of where our output data goes. Similar steps were done to convert input characters in the other accepted ranges. After converting the input character and moving the output to the location pointed by a2, we branched to a label 'endloop', which increments the addresses stored in a1 and a2 by 4, and then branches to the loop, where the process is repeated. The flowchart diagram can be found in the Appendix.

2.2 Part B

The design for Part B was very similar to Part A. The address register a1 still initially points to 0x2300000, but this time, a2 now points to 0x2320000, which is where the converted data is stored.The same data register d2 was used to temporarily store the input and process the data. Just like in Part A, the *SetZeros.s* and the *DataStorage.s* files were used to initialize memory contents as well.

Once again, we had a loop branch that initially loads input data from the location pointed to by a1 into data register d2, and the ASCII 'Enter' code still terminates the program, as before. In Part B, valid inputs are the ASCII characters A-Z and a-z (0x41-0x5A and 0x61-0x7A). Error

handling was also similar to Part A, where the value 0xFFFFFFFF was stored at the memory location pointed to by a2. We had two branches to handle the two ranges of accepted characters.

For example, if an input character was in the range a-z, a difference was taken relative to the ASCII character 'a', and added to the ASCII character 'A'. The converted value is then stored at the memory location pointed to by a2, and the addresses stored in a1 and a2 are incremented by 4 before the loop starts over. The flowchart diagram can be found in the Appendix.

2.3 Part C

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 1: 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.

```
Testing Subroutines. Choose from the menu below
1 - Test First subroutine
2 - Test Second subroutine
3 - Test Third subroutine
4 - Test All subroutine
2
Testing 2nd subroutine.
The entered values are:
5
4
3
2
1

The Min,Max,Mean are:
1
5
3

The Numbers divisible by 2 is/are 4 2
```

Figure 2: MTTTY output when testing our Part B solution

3.3 Part C

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.

```
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 3: 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 4: 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.

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. "

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*/
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)
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

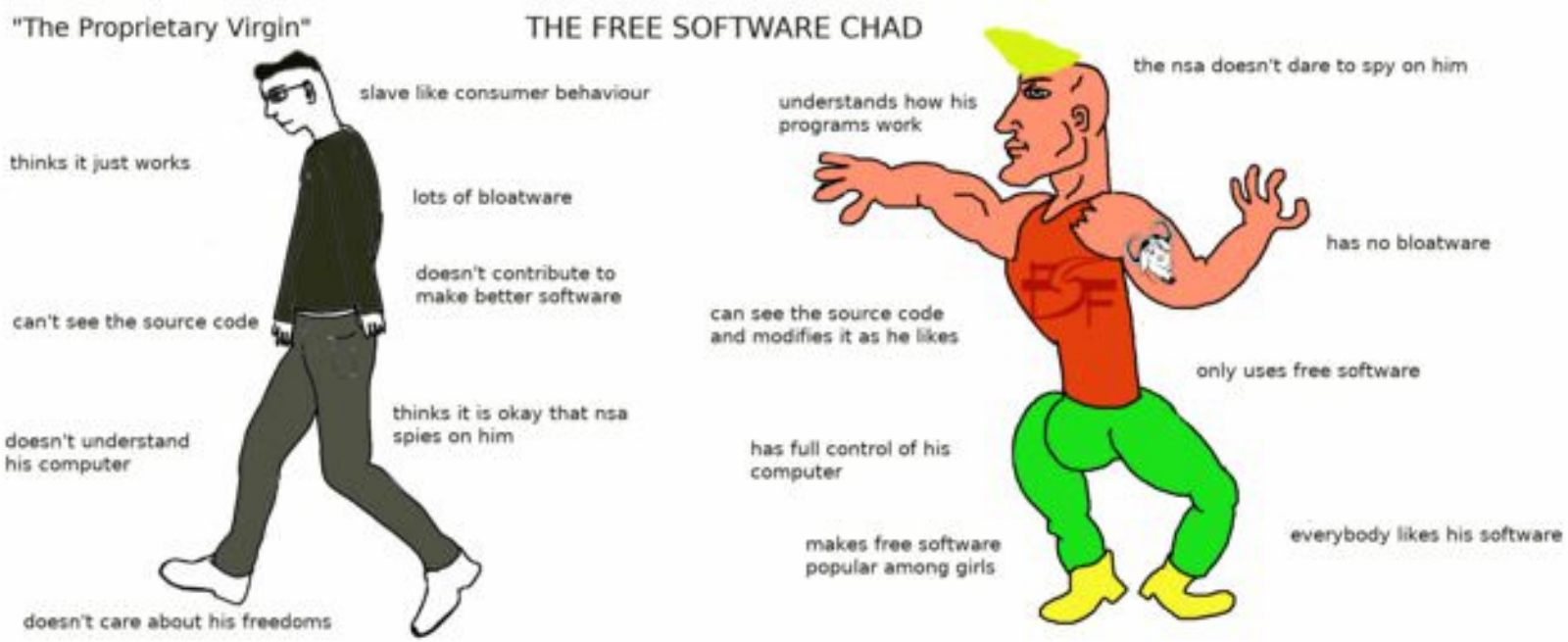
```
/* DO NOT MODIFY THIS ------------------------------------------------------*/
.text
.global Stats
.extern iprintf
.extern cr
.extern value
.extern getstring
/*-----------------------------------------------------------------------*/

/*************************************************************/
/* General Information ***************************************/
/* File Name: Lab3b.s ****************************************/
/* Names of Students: ---------------- and ------------------- **/
/* Date: ----------------                                      **/
/* General Description:                                       **/
/*                                                            **/
/*************************************************************/
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 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