ECE 441

Microprocessors

Instructor: Dr. Jafar Saniie

Teaching Assistant: Xin Huang

Final Project Report:

**MONITOR PROJECT**

20/11/2018

By: Yuzhe Lim

Acknowledgment: I acknowledge all of the work including figures and codes are belongs to me and/or persons who are referenced.

Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Table of Contents

Abstract 2

1-) Introduction 2

2-) Monitor Program 3

2.1-) Command Interpreter 4

2.1.1-) Algorithm and Flowchart 5

2.1.2-) 68000 Assembly Code 7

2.2-) Debugger Commands 7

2.2.1-) HELP 7

2.2.2-) MDSP 8

2.2.3-) SORTW 11

2.2.4-) MM 14

2.2.5-) MS 18

2.2.6-) BF 20

2.2.7-) BMOV 23

2.2.8-) BTST 24

2.2.9-) BSC 27

2.2.10-) GO 30

2.2.11-) DF 31

2.2.12-) EXIT 33

2.2.13-) DF 34

2.2.14-) EXIT 36

2.3-)Exception Handlers 39

2.3.1-) Bus Error Exception 39

2.3.2-) Address Error Exception 40

2.3.3-) Illegal Instruction Exception 41

2.3.4-) Privilege Violation Exception 41

2.3.5-) Divide by Zero Exception 42

2.3.6-) Line A and Line F Emulators 43

2.3.7-) Check Instruction Exception 43

2.3.8-) Bus and Address Exception Handler 44

2.3.9-) General Exception Handler 45

2.4-)User Instructional Manual Exception Handlers 46

2.4.1-) Help Menu 46

2.4.2-) Invalid Message Prompt 47

3-) Discussion 48

4-) Feature Suggestions 48

5-) Conclusions 49

6-) References 49

7-) Appendix 49

Abstract

The purpose of this project is to design a monitor program that act as a interface for the user to interact with Motorola MC68000 microprocessor. The monitor program consist of fourteen basic debugger functions and custom exception handlers that is able to handle eight different system exceptions of MC68000. The structure of the program were categorized and a brief explanation of each components were given, followed by algorithm, flowcharts and the actual assembly language codes.

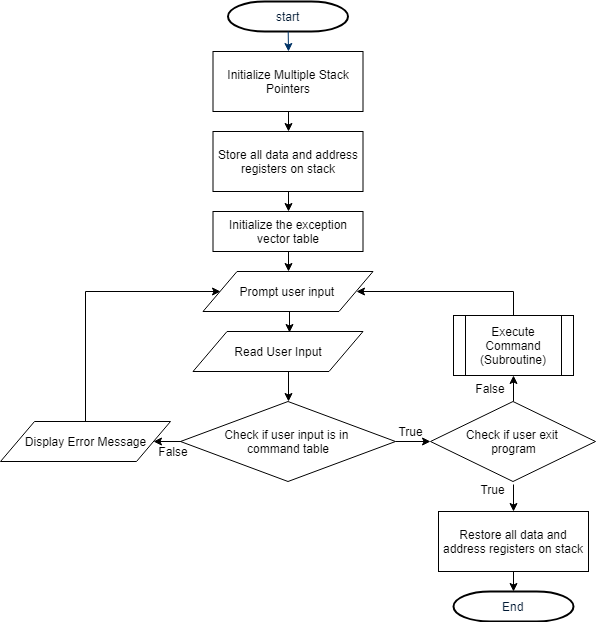
# *1-) Introduction*

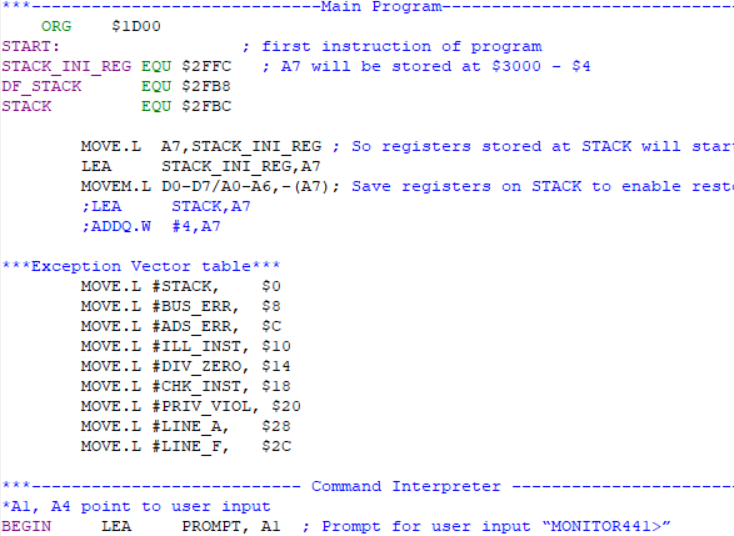
The objective of this project is to design a monitor program that acts a medium for user to be able to interact with the memory of the SANPER-1 ELU. In order to achieve the complete monitor program design, twelve essential debugger functions from the TUTOR terminal on SANPER-1 ELU such as help, memory display, sort(word), memory modify, memory set, block fill, block move, block test, block search, go function, display formatted registers, and exit function were to be implemented alongside with two additional functions. Besides, a customize exception handler for each of the eight system exceptions in MC68000 were also required to be part of the system design in order to prevent erroneous input from terminating the program. The monitor program has a constraint of 3K memory spaces starting from memory address $1000, stack size of 1K starting from memory address $3000. And the ineligibility to use Macro. All these constraints post a huge problem to the design of the program due to a high amount of optimization were needed to fulfil all the problems.

The design methodology used in this design is Test-Driven Development (TDD). The only constrain for this methodology is the programmer has to have a minimum expected outcome of each function to produce testcase. As this program is a miniature TUTOR terminal, all the outcomes and capabilities of each command are clear. So, testcases for every function are written before the actual code for the function was even written. The design allows the programmer to write minimum amount of codes as they were only design to pass all the testcases, and optimization of the code will be up next. The entire program is meant to be written in assembly language in the EASy68k Editor. The knowledge and ideas required to implement all the mentioned functions were acquired throughout the lab sessions in ECE 441.

***2-) Monitor Program***

The monitor program algorithm starts off with prompting and collecting user input through the terminal, then determine if the user input is a valid input. If the user input matches with one of the predefined function, the function will be carried out, else the program will prompt for user input indefinitely until EXIT command is entered.

*Figure 2.1. Monitor program block diagram*



***2.1-) Command Interpreter***

The algorithm for command interpreter starts off by prompting user, then read and store the input in input buffer. It then compares letter by letter of between the commands on the command table (COM\_TABL) to the string in user’s input. After looping through every command on command table, empty spaces in the respective command string is skipped before going to the next one command. The loop confirms a matching command by comparing the blank space after both the strings in command table and user input. The increment in the command table is stored in a counter during the process.

If the program reaches the end of the command table and no matching command is found, the program displays error message to user and go to the beginning of command interpreter. Else, the address of the command entered by user will be located in command address (COM\_ADDR) table by using the counter mentioned before. The program will then go to the memory location of the subroutine and execute it. Once done, program goes back to the beginning of command interpreter and repeat the whole process.

***2.1.1-) Algorithm and Flowchart***

*Prompt and read user input // User input (command) will be stored in memory with A1as pointer*

*A2 = COM\_TABL // assign A2 as pointer on the command names*

*A3 = COM\_ADDR // assign A3 pointer on the addresses of subroutines*

*A4 = A1 // assign A4 as pointer to USER\_INPUT*

*D2=0 // D2 to count increment in A2*

*While A3 >A2 // while COM\_TABL A2) hasn’t ended*

*If (A2==A4) // If A2 equals to A4*

*A2 = A2+1 // Increment A2*

*A4=A4+1 // Increment A4*

*Else //*

*A2=A2-1 // Decrement A2*

*If(A2 == ‘ ‘) // If A2 is blank space*

*A3 = A3+D2 // Increment A3 by D2*

*Jump to subroutine using COM\_ADDR address*

*Else*

*Loop through all the empty spaces in this command*

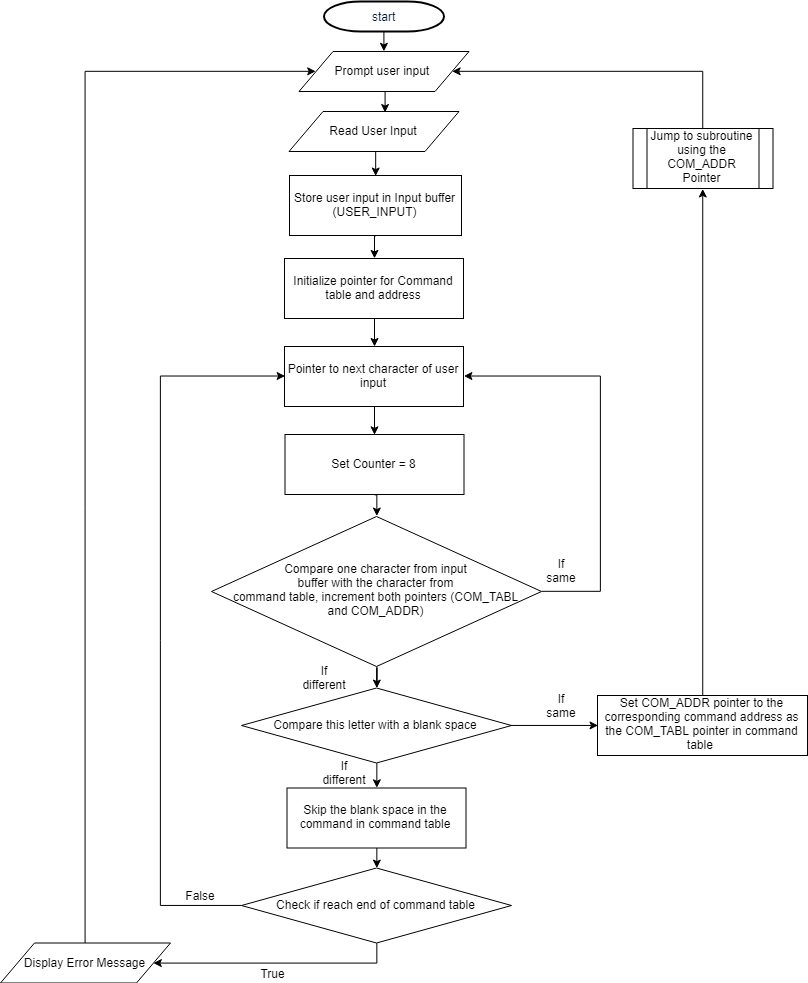
*D2 = D2+2 // length to reach next command in COM\_TABL*

*End if A2 == A3 // Reach end of COM\_TABL*

*Display Error message*

*finish // finish*

*Figure 2.2. Command Interpreter Algorithm*

***2.1-)***

*Figure 2.3. Command Interpreter Flowchart*

***2.1.2-) Command Interpreter Assembly Code***

*Figure 2.4. 68000 Assembly Code*

***2.2-) Debugger Commands***

Each debugger command is a subroutine that can be accessed from command interpreter. Every debugger command starts off by storing registers used in the process on stack then proceed to read the user’s input that comes after debugger commands (if there’s any, such as address), then it either process as it should or return error message. All subroutines restore registers used at the end of the subroutine before returning to command interpreter.

***2.2.1-) HELP***

The algorithm for HELP display the help’s table (HELP\_TABL) which includes all commands available and description of how to use them.

Command’s syntax: **HELP**

***2.2.1.1-) HELP Algorithm and Flowchart***

*Save register D0 and A1 to stack*

*A1 = HELP\_TABL // assign A2 as pointer on the help table*

*Display HELP\_TABL // Display the help table*

*Restore register D0 and A1 from stack*

*Return to command interpreter*

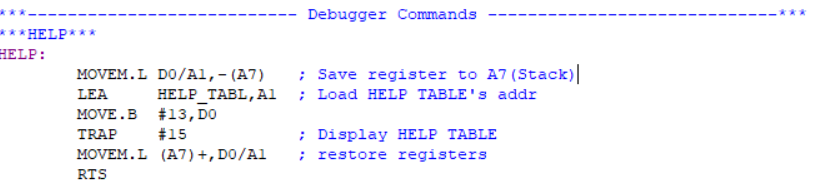
*finish*

*Figure 2.5. HELP Algorithm*

******

*Figure 2.6. HELP Flowchart*

***2.2.1.2-) HELP Assembly Code***



*Figure 2.7. HELP Assembly Code*

***2.2.2-) MDSP – Memory Display***

The algorithm for MDSP display the address alongside its memory content from <address 1> to <address 2>. If <address 2> isn’t input, <address + 16 bytes> will automatically be assigned as <address 2>.

Command’s syntax: **MDSP <address 1> <address 2> or MDSP <address 1>**

***2.2.2.1-) MDSP Algorithm and Flowchart***

*Save register D0, A1,A5,A6 to stack*

*A5 = first address (HEX) // assign first converted HEX address in A5*

*If (no second address) // If user input doesn’t contain 2nd address*

*A6 = A5 + 16 bytes*

*Else*

*A6 = second address (HEX) // assign 2nd converted HEX address in A6*

*While (A6 >= A5)*

*Display value of A5 // A5 is current address*

*Display contents at A5 // Print the content in memory location A5*

*A5 = A5 +1 // go to next byte*

*End If A5 surpasses A6 // Reached the end*

*Restore register D0,A1,A5,A6 from stack*

*Return to command interpreter*

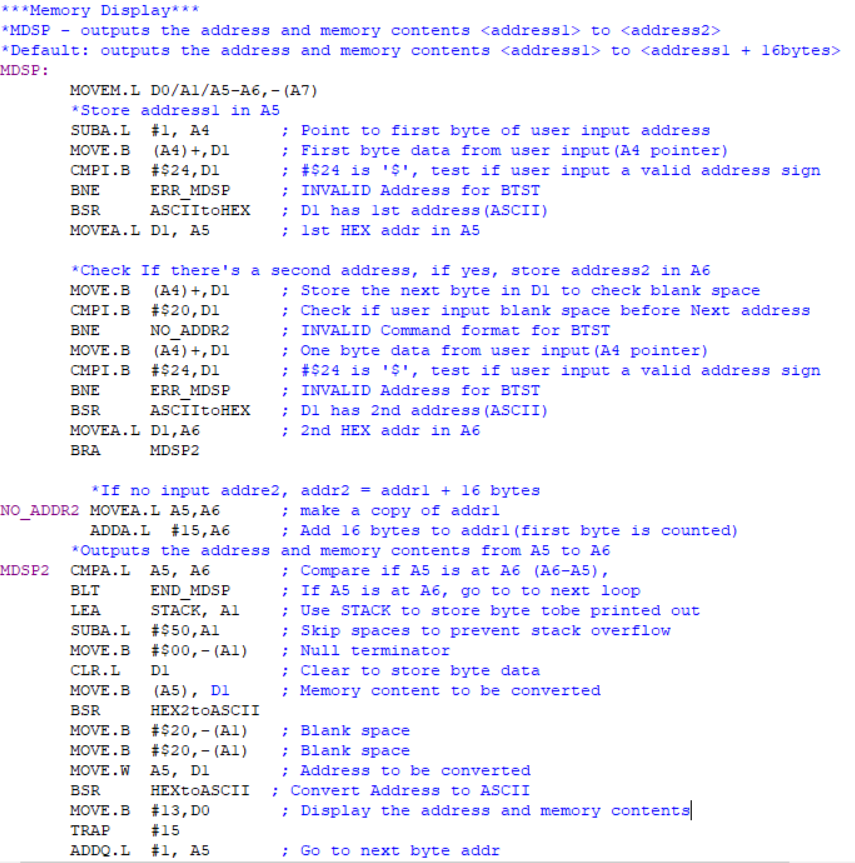
*finish*

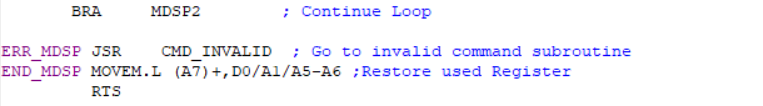
*Figure 2.8. MDSP Algorithm*



*Figure 2.9. MDSP Flowchart*

***2.2.2.2-) MDSP Assembly Code***





*Figure 2.10. MDSP Assembly Code*

***2.2.3-) SORTW - SORT***

The algorithm for SORTW rearrange the block of memory from <address1> to <address2> in word size data. The sorting method is similar to those of the famous bubble sort where the one data is being compared to the next data and depends on the sorting order, they could swap to be arranged in ascending or descending order. The process repeats as much as the amount of word size element it has within the range of addresses given. The order of the sorting can be determined by the command ‘;A’ or ‘;D’, which means ascending and descending respectively with descending as the default sorting. Address 1 and address 2 must be even for this function to be carried out.

Command’s syntax: **SORTW <address 1> <address 2> <;A or ;D)**

***2.2.3.1-) SORTW Algorithm and Flowchart***

*Save register D1,D2,A1,A2,A5,A6 to stack*

*A5 = first address (HEX) // assign first converted HEX address in A5*

*A6 = second address (HEX) // assign 2nd converted HEX address in A6*

*If (User\_Input = ;D or no User\_Input) // If user input ;D or no User Input*

*Do // Do while loop, without initial condition*

*A2 = A5 // Store 1st address in A2*

*If (A2<A2+1) //* *If next > prev word data*

*SWAP A2 with A2+1 //Swap the content of two*

*A2 = A2+1 // Increment Address*

*End if A2 > A6 // Reach end of address range to be sorted*

*Else if (User\_Input = ;A) // If user input ;A*

*Do // Do while loop, without initial condition*

*A2 = A5 // Store 1st address in A2*

*If (A2>A2+1) // If prev > next word data*

*SWAP A2 with A2+1 //Swap the content of two*

*A2 = A2+1 // Increment Address*

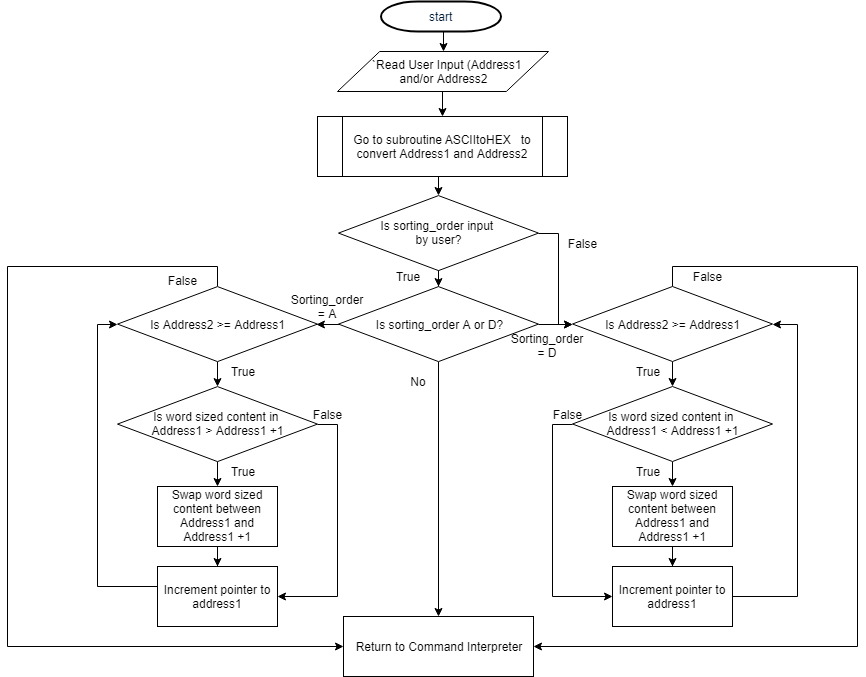
*End if A2 > A6 // Reach end of address range to be sorted*

*Restore register D1,D2,A1,A2,A5,A6 from stack*

*Return to command interpreter*

*finish*

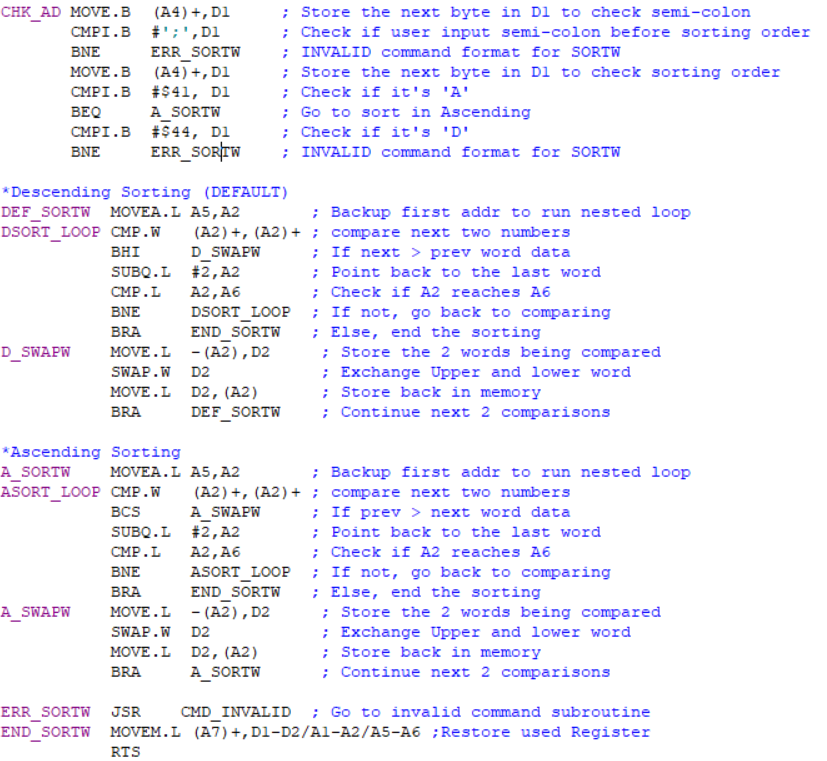
*Figure 2.11. SORTW Algorithm*



*Figure 2.12. SORTW Flowchart*

***2.2.3.2-) SORTW Assembly Code***





*Figure 2.13. SORTW Assembly Code*

***2.2.4-) MM - Memory Modify***

The algorithm for display the memory address and the content of it which allows user to modify it. The size of the content displayed and available to change depends on user input ranging from byte, word and long size, if user did not choose any of the three options, byte size operation selected automatically. The functions goes indefinitely unless user input ‘.’ After being prompted to modify the memory content.

Command’s syntax: **MM <address> ;B or ;W or ;L or null**

***2.2.4.1-) MM Algorithm and Flowchart***

*Save register D0,D1,A1,A5 to stack*

*A5 = address (HEX) // assign converted HEX address in A5*

*If(no data size is input)*

*data\_size = B*

*Else*

*Check and store user input data size in data\_size*

*While (User\_Input != ‘.’)*

*Display address of A5 and its memory content according to data\_size*

*If (User\_Input <= data­\_size) // if input data is within data\_size*

*Replace A5 memory content with User\_Input*

*A5 = A5 +1 // Only increment Address when writing is successful*

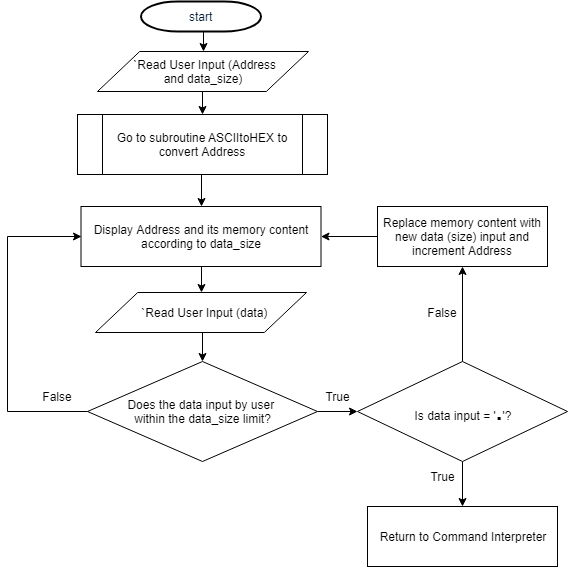
*End if (User\_input == ‘.’)*

*Restore register D0,D1,A1,A5from stack*

*Return to command interpreter*

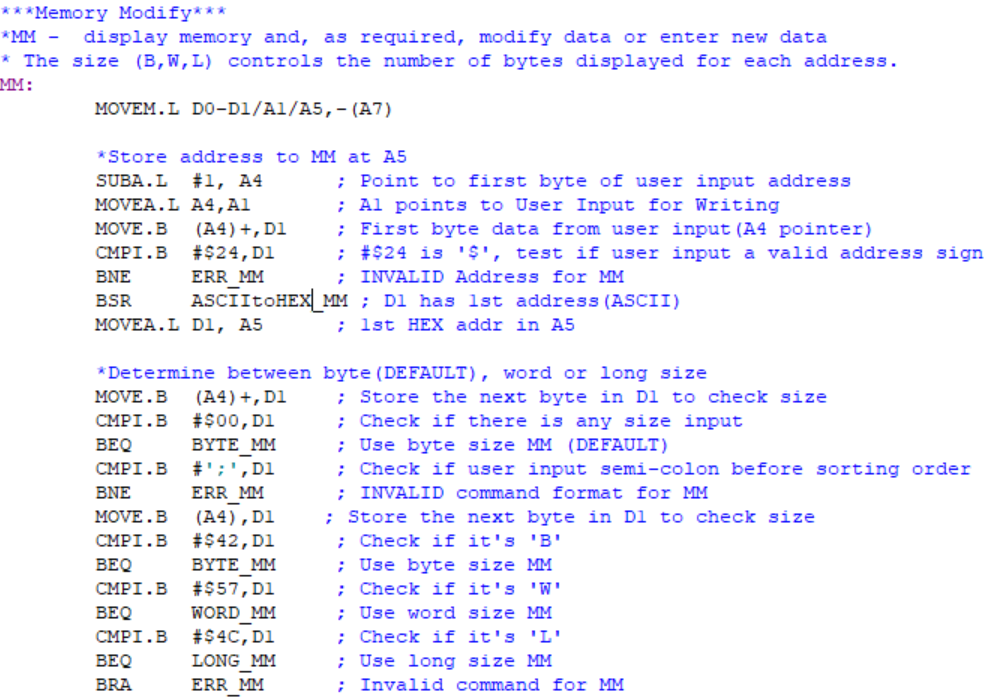
*finish*

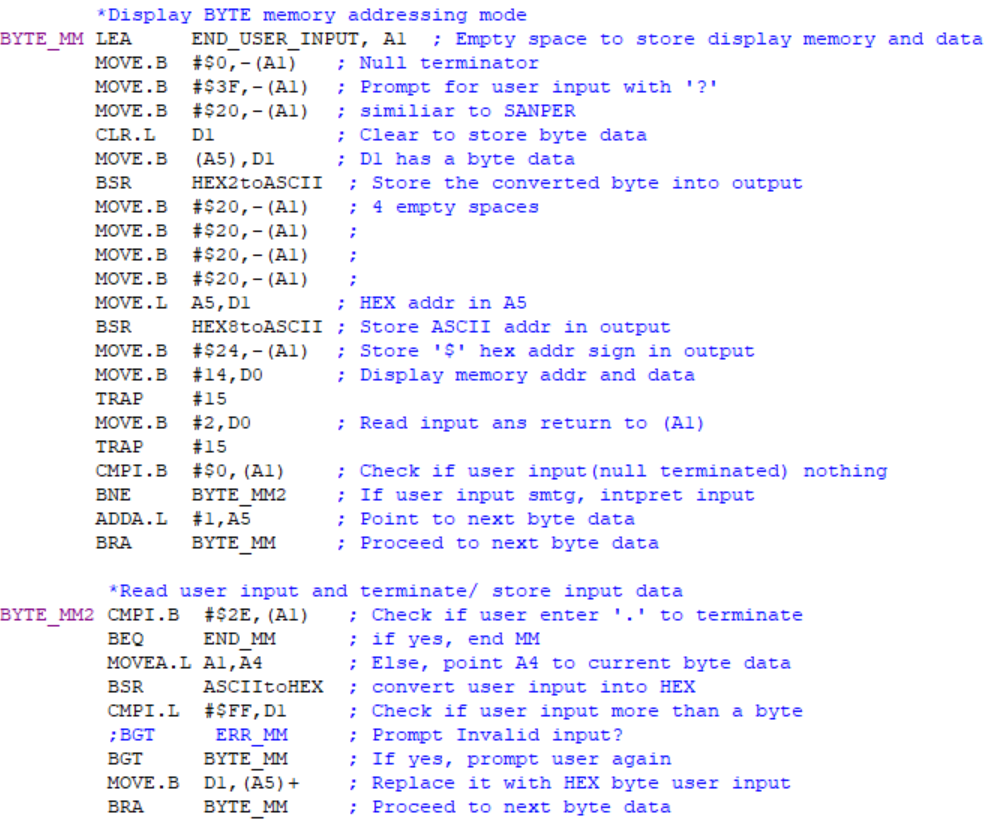
*Figure 2.14. MM Algorithm*

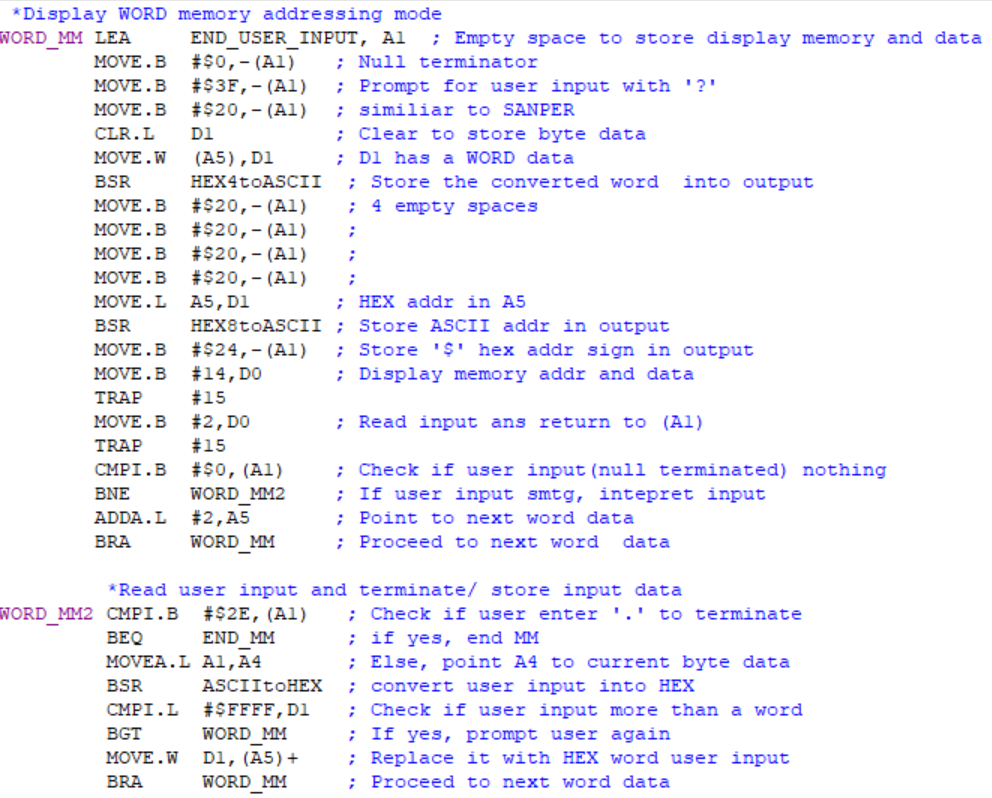
**

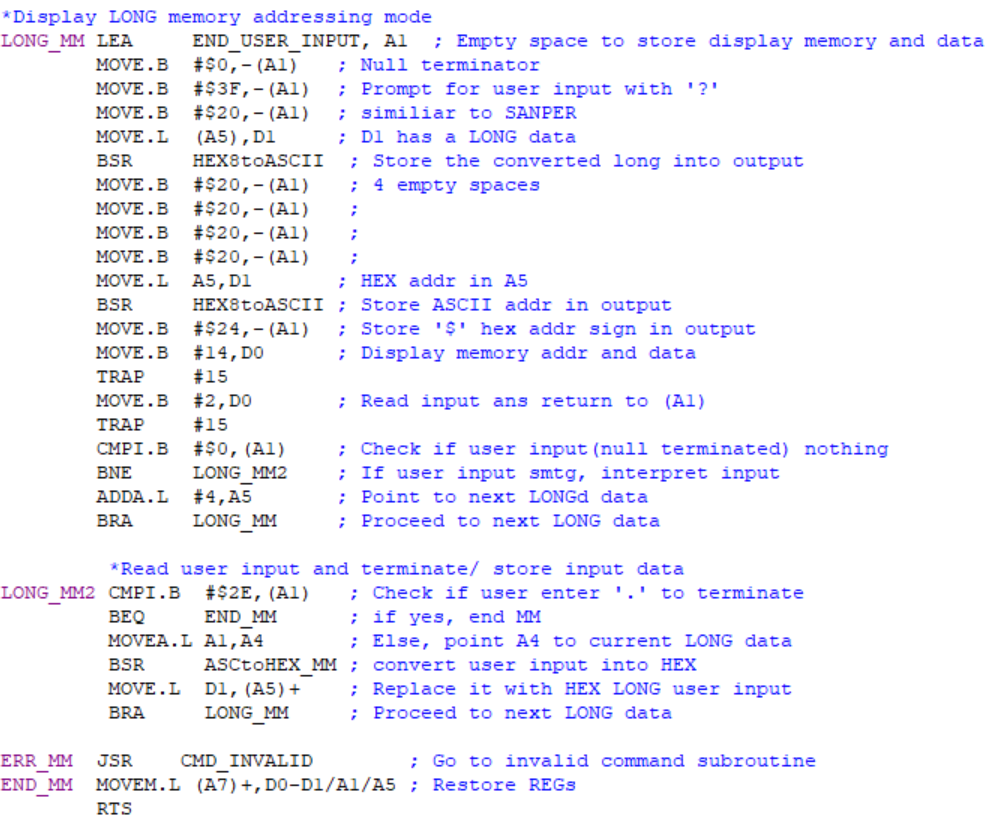
*Figure 2.15. MM Flowchart*

***2.2.4.2-) MM Assembly Code***









*Figure 2.16. MM Assembly Code*

***2.2.5-) MS - Memory Set***

The algorithm for memory writes ASCII string or hexadecimal data into address specified by user. ‘$’ sign can to be entered before data to represent a hexadecimal data whereas ASCII string can be just written like normal string.

Command’s syntax: **MS <address> $<hexadecimal value> or <ASCII string>**

***2.2.5.1-) MS Algorithm and Flowchart***

*Save register D1,A5 to stack*

*A4 = User\_Input // User input stored in memory with A4 as pointer*

*A5 = address (HEX) // assign converted HEX address in A5*

*If (A4 contains ‘$’)*

*If (Use\_Input > longword size)*

*Replace A5 memory content with last long size User\_Input*

*Else*

*Replace A5 memory content with User\_Input*

*Else*

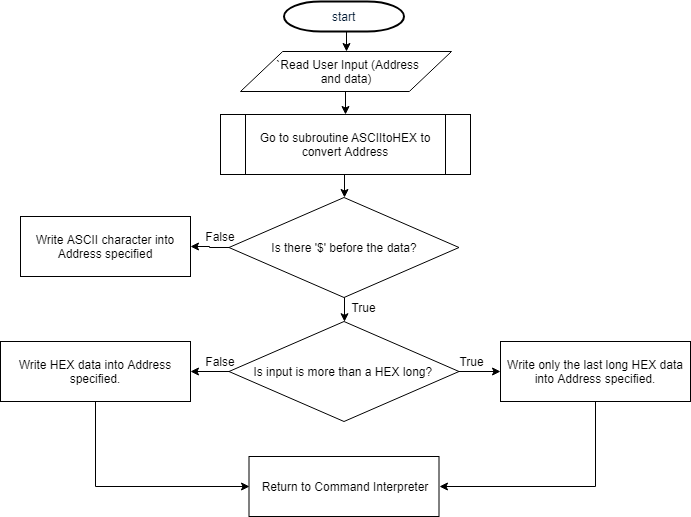
*Replace A5 memory content with User\_Input (ASCII) + Null terminator($00)*

*Restore register D1,A5 from stack*

*Return to command interpreter*

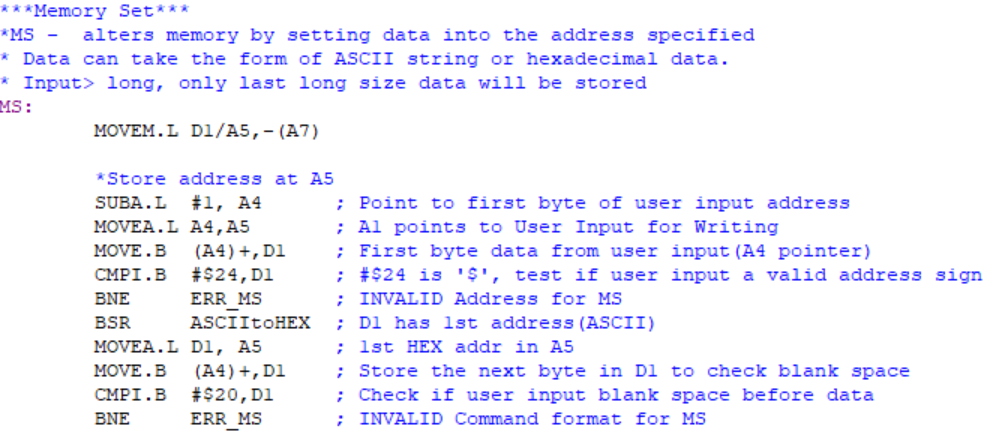
*finish*

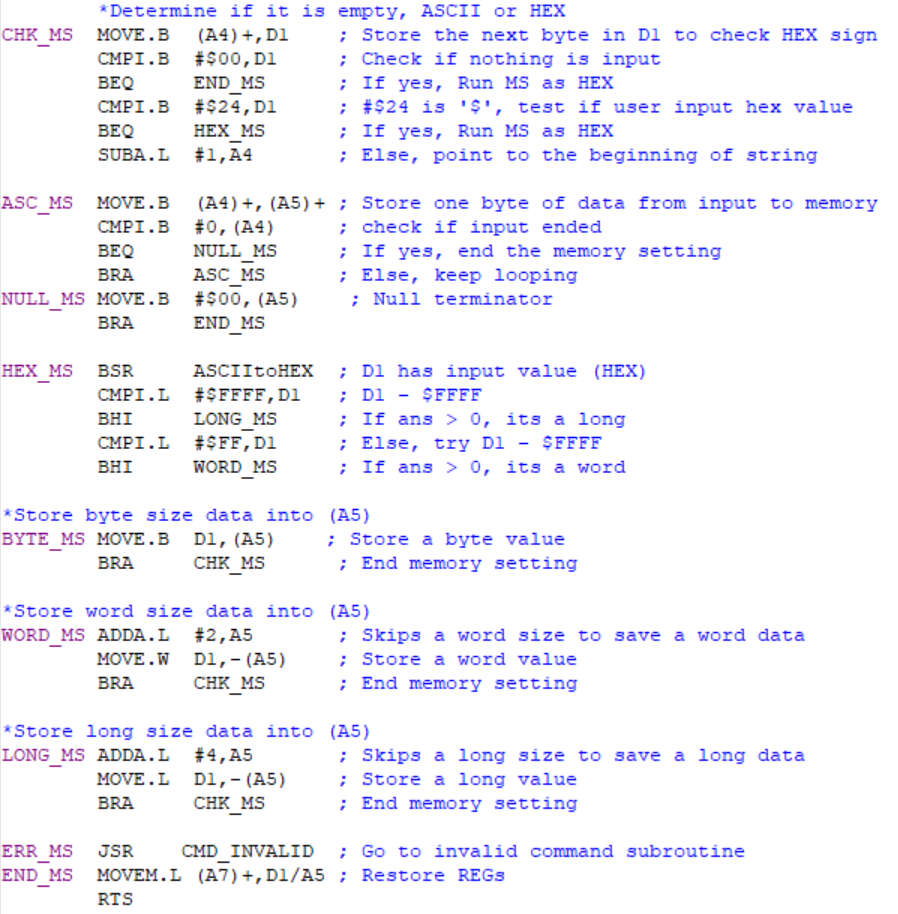
*Figure 2.17. MS Algorithm*

**

*Figure 2.18. MS Flowchart*

***2.2.5.2-) MS Assembly Code***





*Figure 2.19. MS Assembly Code*

***2.2.6-) BF- Block Fill***

The algorithm for Block Fill (BF) writes a word sized (2 bytes) data pattern from <address 1> to <address 2> specified by user. Both addresses have to be of even address, and data pattern that is less than a word size will be shifted right and leading zeroes will filled up the empty spot.

Command’s syntax: **BF <address 1> <address 2> <word sized data pattern>**

***2.2.6.1-) BF Algorithm and Flowchart***

*Save register D1,D3,D4,D6,A1,A5,A6 to stack*

*A1 = User\_Input // User input stored in memory with A1 as pointer*

*If (last bit of address 1 = 0) // Address 1 is even*

*A5 = address 1(HEX) // assign 1st converted HEX address in A5*

*If (last bit of address 2 = 0) // Address 2 is even*

*A6 = address 2(HEX) // assign 2nd converted HEX address in A6*

*Else*

*Display error message and Return to command interpreter*

*While (A6>A5) //While Address 1 hasn’t reached Address 2*

*Replace A5 memory content with word sized User\_Input*

*A5 = A5+1 // Increment A5*

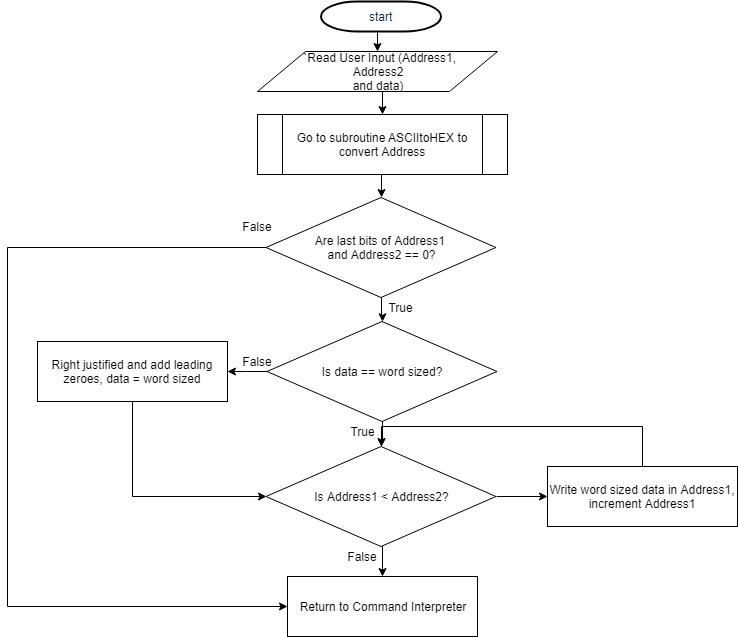
*End If (A5 == A6)*

*Restore register D1,D3,D4,D6,A1,A5,A6 from stack*

*Return to command interpreter*

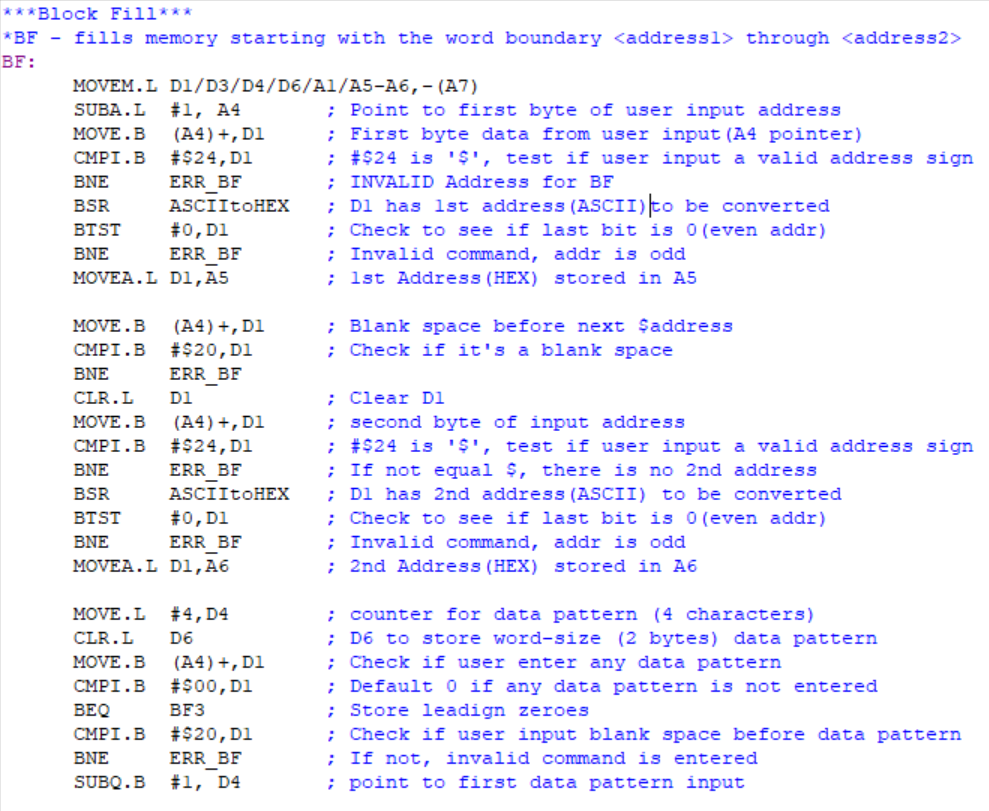
*finish*

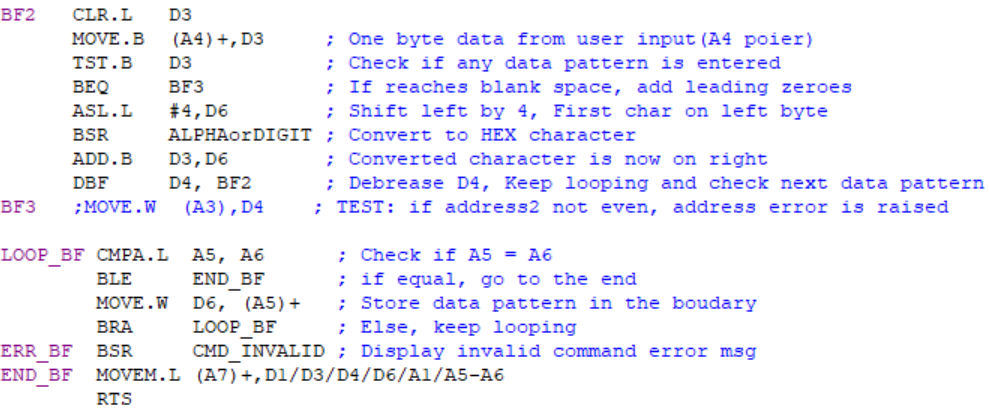
*Figure 2.20. BF Algorithm*

**

*Figure 2.21. BF Flowchart*

***2.2.6.2-) BF Assembly Code***





*Figure 2.22. BF Assembly Code*

***2.2.7-) BMOV - Block Move***

The algorithm for Block Move (BMOV) copy the block of memories (in byte size) from <address 1> through <address 2> and store it starting from <address 3>.

Command’s syntax: **BMOV <address 1> <address 2> <address 3>**

***2.2.7.1-) BMOV Algorithm and Flowchart***

*Save register D1,A2,A5,A6 to stack*

*A5 = address 1(HEX) // assign 1st converted HEX address in A5*

*A6 = address 2(HEX) // assign 2nd converted HEX address in A6*

*A2 = address 3(HEX) // assign 3rd converted HEX address in A2*

*While (A6>A5) //While Address 1 hasn’t reached Address 2*

*Replace A2 memory content with byte sized content in A5*

*A5 = A5+1 // Increment Address1 pointer*

*A2 = A2 +1 // Increment Address3 pointer*

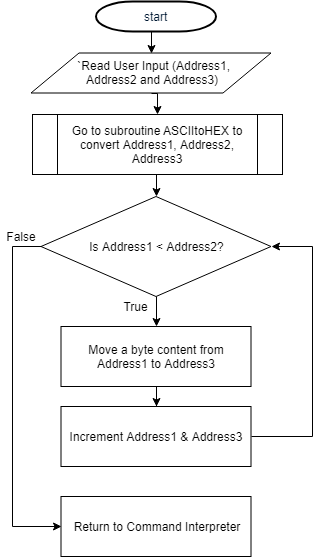
*End If (A5 == A6)*

*Restore register D1,A2,A5,A6 from stack*

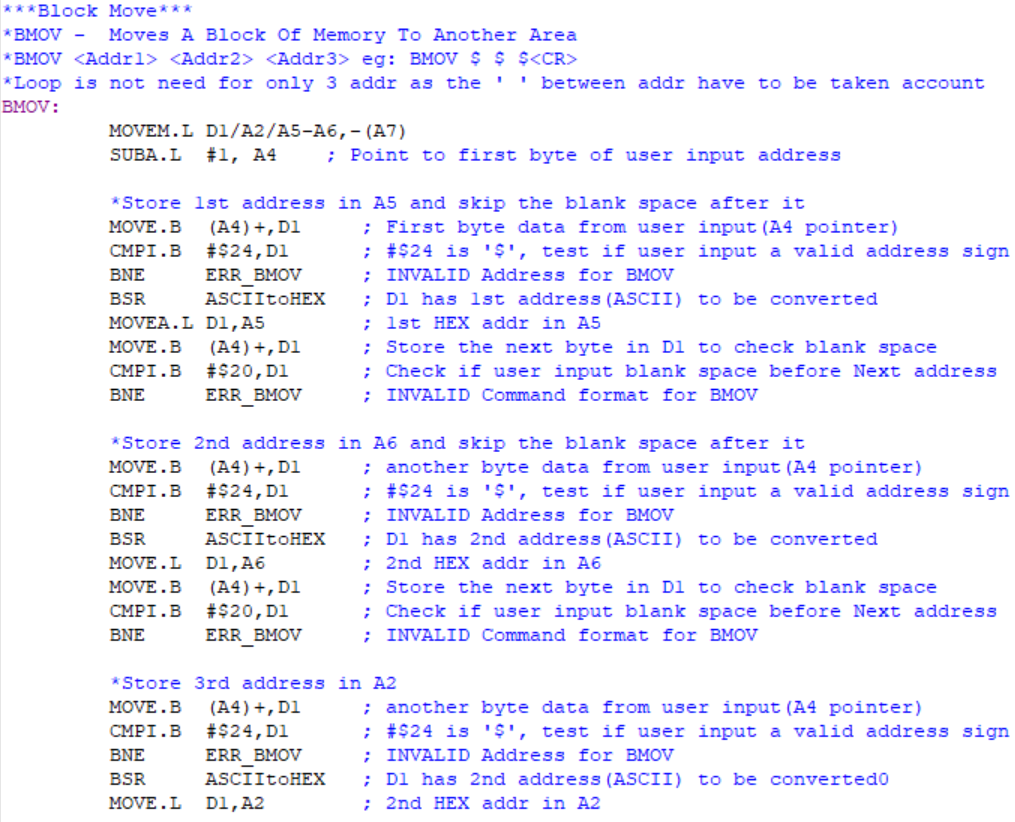
*Return to command interpreter*

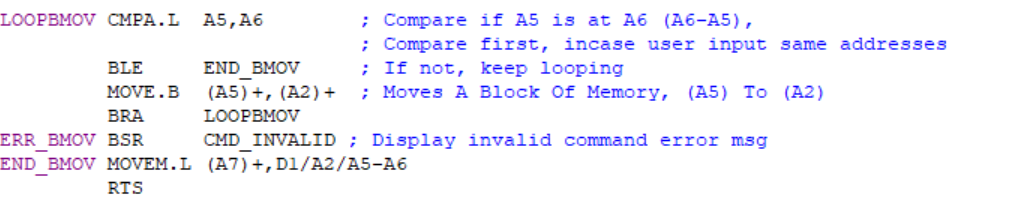
*finish*

*Figure 2.23. BMOV Algorithm*

*Figure 2.24. BMOV Flowchart*

***2.2.7.2-) Debugger Command #7 Assembly Code***





*Figure 2.25. BMOV Assembly Code*

***2.2.8-) BTST – BLOCK TEST***

The algorithm for Block Test (BTST) test the memory from addresses specified by filling up every byte from <address 1> to <address 2> with pattern $AA (1010) then check if the reading matches the writing, if any mismatch happens, error message containing the address, data stored, and the data read of the failing memory will be displayed, else a pass message will be displayed. The process is then repeated with pattern $55(0101).

Command’s syntax: **BTST <address 1> <address 2>**

***2.2.8.1-) BTST Algorithm and Flowchart***

*Save register D0,D1,A1,A2,A5,A6 to stack*

*A5, A2 = address 1(HEX) // assign 1st converted HEX address in A5,A2*

*A6 = address 2(HEX) // assign 2nd converted HEX address in A6*

*Clear D1 //to store byte data from memory*

*While (A6>A5) //While Address 1 hasn’t reached Address 2*

*Replace A5 memory content with $AA*

*Store a byte of memory from A5 in D1*

*If (D1 == $AA)*

*A5 = A5+1 // Increment Address1 pointer*

*Else*

*Display error message with A5, $AA, and D1*

*A5 = A2*

*While (A6>A5) //While Address 1 hasn’t reached Address 2*

*Replace A5 memory content with $55*

*Store a byte of memory from A5 in D1*

*If (D1 == $55)*

*A5 = A5+1 // Increment Address1 pointer*

*Else*

*Display error message with A5, $55, and D1*

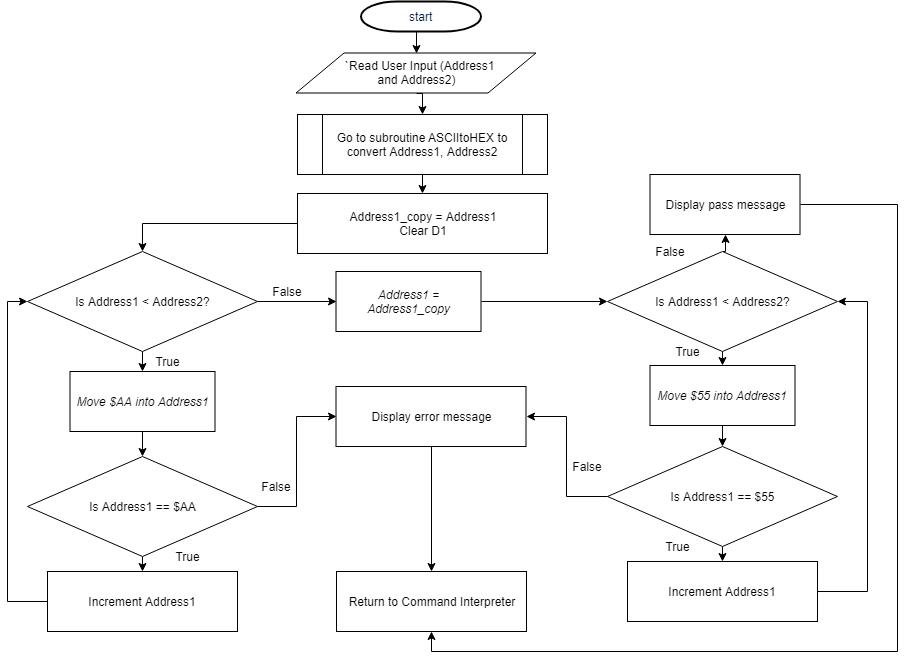
*Display pass message*

*Restore register D0,D1,A1,A2,A5,A6 from stack*

*Return to command interpreter*

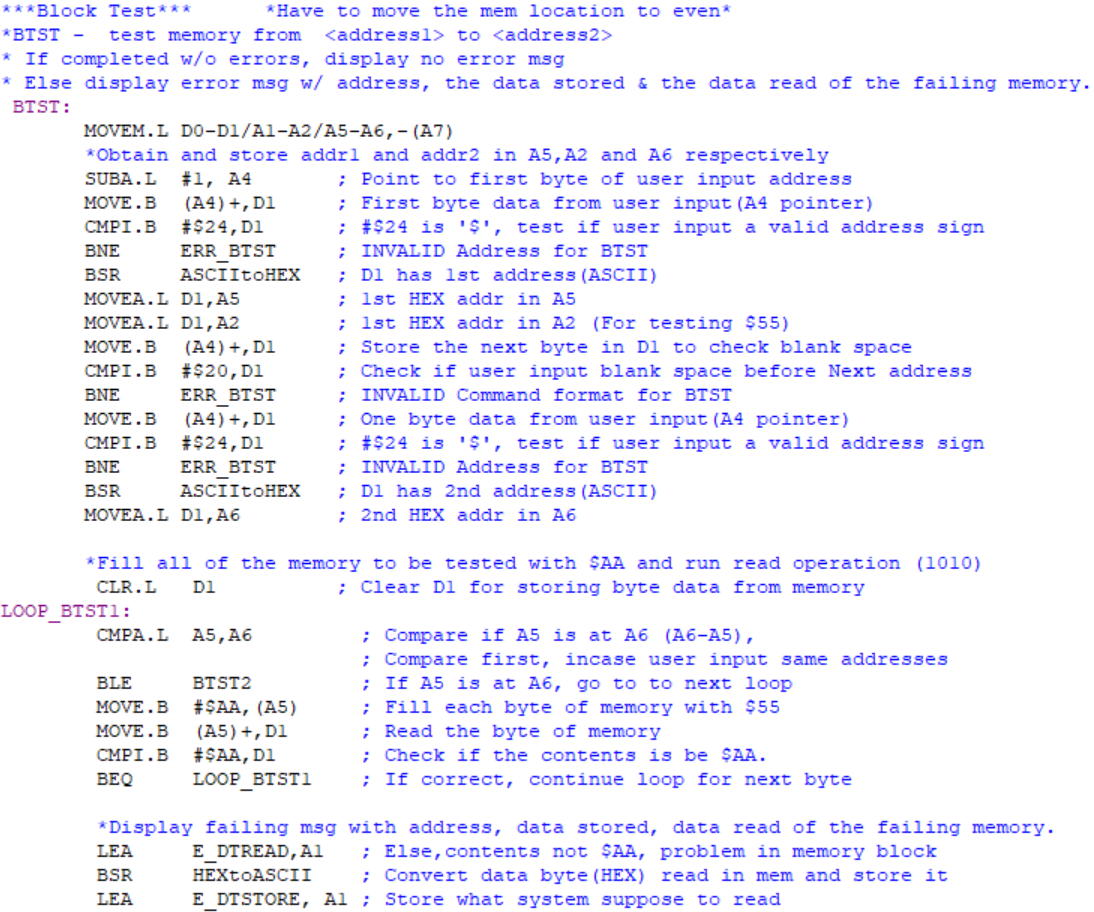
*finish*

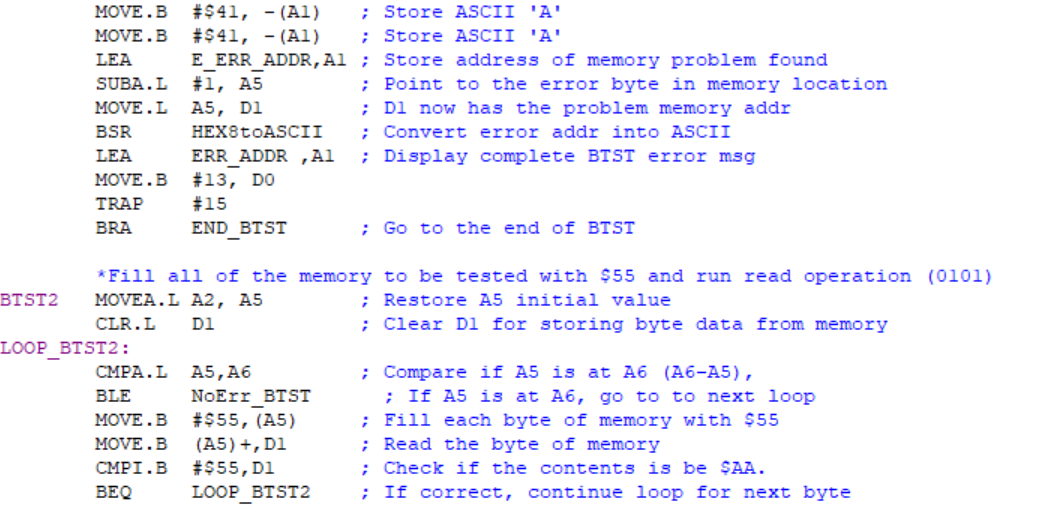
*Figure 2.26. BTST Algorithm*

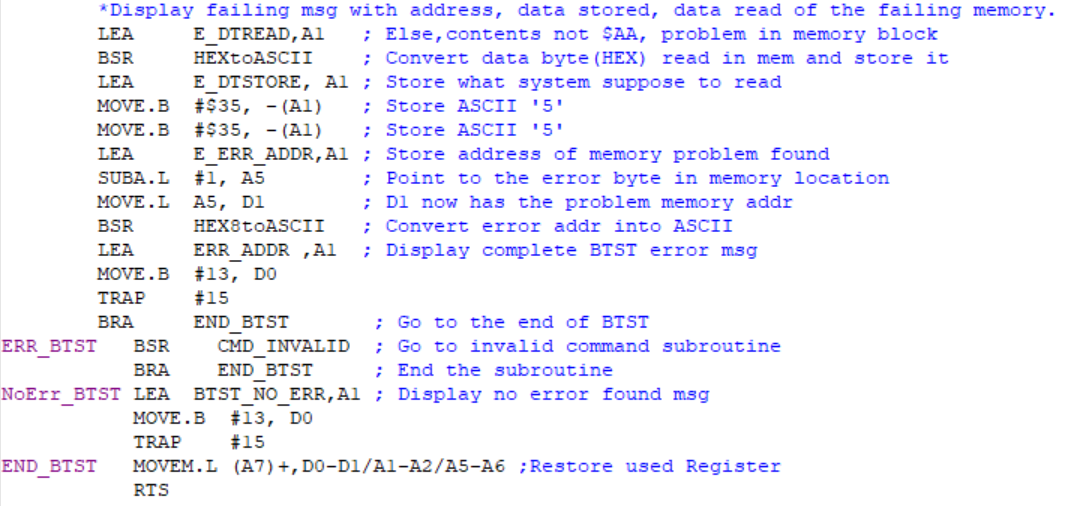
**

*Figure 2.27. BTST Flowchart*

***2.2.8.2-) BTST Assembly Code***







*Figure 2.28. BTST Assembly Code*

***2.2.9-) BSCH – BLOCK SEARCH***

The algorithm for Block Search (BSCH) search for an ASCII string from <address1> through <address2>, both inclusive. Every time the string matches with the content of memory, the memory address and memory content is displayed.

Command’s syntax: **BSCH <address 1> <address 2> <ASCII string>**

***2.2.9.1-) BSCH Algorithm and Flowchart***

*Save register D1,D2,A1 to stack*

*A5 = address 1(HEX) // assign 1st converted HEX address in A5*

*A6 = address 2(HEX) // assign 2nd converted HEX address in A6*

*A4 = User\_input ASCII string // A4 point to user input ASCII string*

*If( A1 = empty)*

*Prompt error message and return to command interpreter*

*A2 = A4 //Backup A4 location*

*Clear D2 //D2 as flag to tell if string is found at least once*

*While (A6>A5) //While Address 1 hasn’t reached Address 2*

*Compare a byte from A5 and A2*

*If (A5 == A2)*

*Compare every byte of A5 and A2 until A2 reaches null terminator*

*If (both strings are equals)*

*Display found message*

*Else*

*A2 = A4 //Backup A4 location*

*Else*

*A5 = A5+1 // Increment Address1 pointer*

*End If (A5 == A6)*

*If(D2 < =0)*

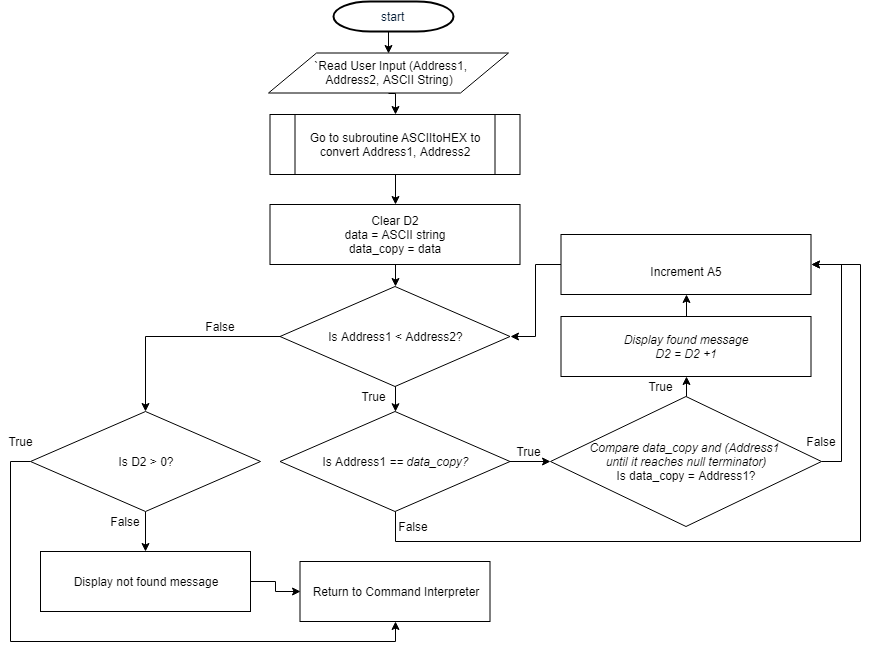
*Display Not found message2*

*Restore register D1,D2,A1 from stack*

*Return to command interpreter*

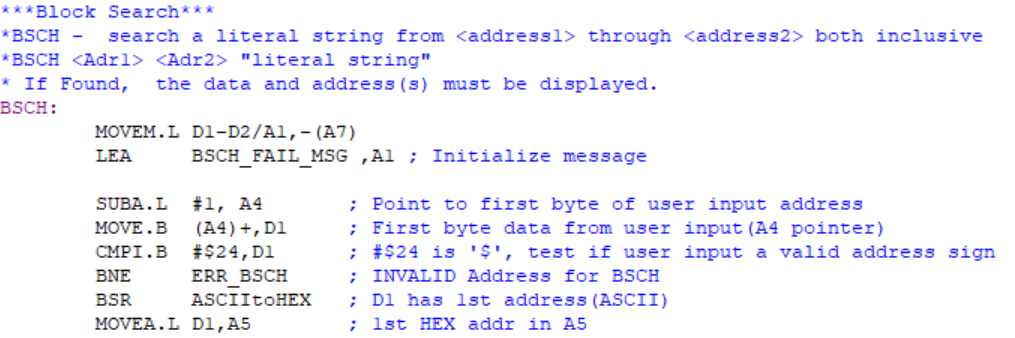
*finish*

*Figure 2.29. BSCH Algorithm*

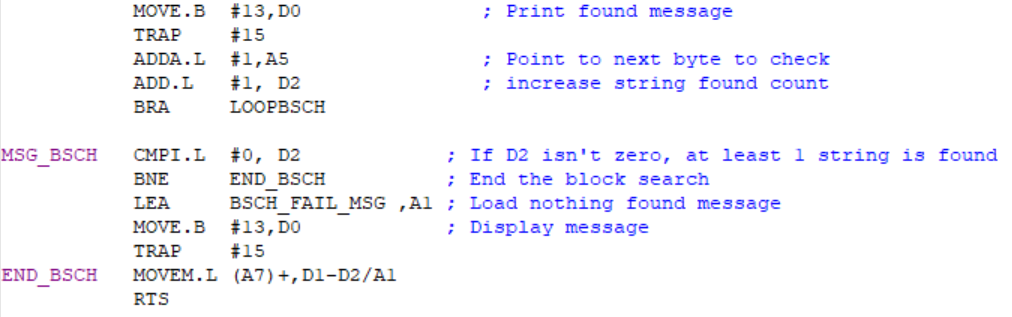
**

*Figure 2.30. BSCH Flowchart*

***2.2.9.2-) BSCH Assembly Code***







*Figure 2.31. BSCH Assembly Code*

***2.2.10-) GO***

The algorithm for GO start execution of a memory address input by user.

Command’s syntax: **GO <address>**

***2.2.10.1-) GO Algorithm and Flowchart***

*Save all data and address registers to stack*

*A6 = address 1(HEX) // assign 1st converted HEX address in A6*

*If (A6 is valid)*

*Jump to address (subroutine)*

*Else*

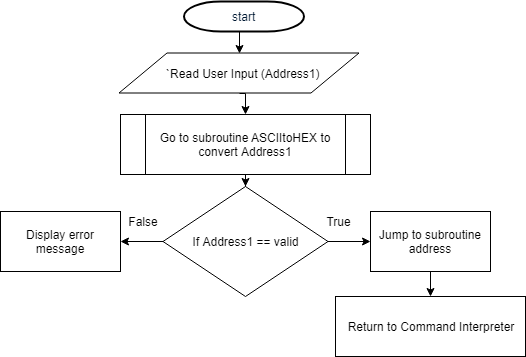
*Display error message*

*Restore all data and address registers from stack*

*Return to command interpreter*

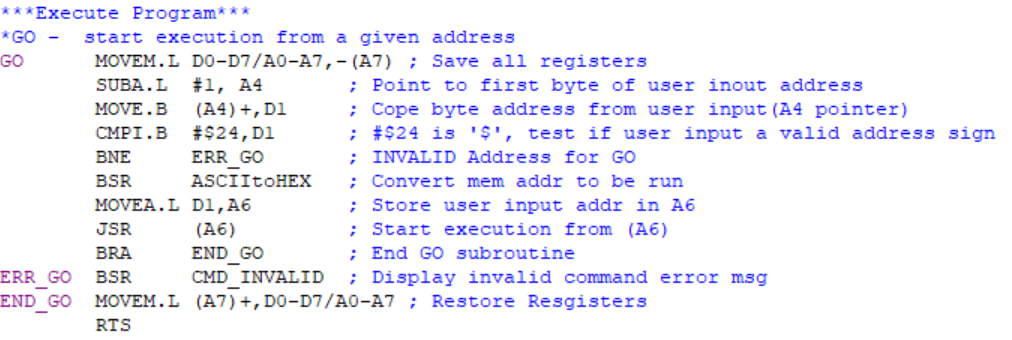
*finish*

*Figure 2.32. GO Algorithm*

**

*Figure 2.33. GO Flowchart*

***2.2.10.2-) GO Assembly Code***



*Figure 2.34. GO Assembly Code*

***2.2.11-) DF - Display Formatted Registers***

The algorithm for DF(Display Formatted Registers) retrieve current PC, SR, US, SS and Data, Address registers from stack and display them.

Command’s syntax: **DF**

***2.2.11.1-) DF Algorithm and Flowchart***

*Save all data and address registers to stack*

*Save A7 again // To be shown as SS*

*Save USP and SR to stack*

*A1 Point to the end of Display message for DF in memory*

*A2 point to end of stack*

*Move converted long size Hex registers value from A2 to A1(backward) for 16 times*

*Convert a longword A7 from A2 to A1(backward)*

*Convert a longword US from A2 to A1(backward)*

*Convert a word SR from A2 to A1(backward)*

*Convert a longword PC from A2 to A1(backward)*

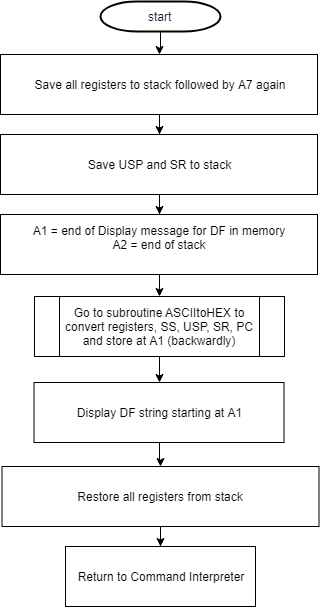
*Display string starting at A1 till null terminator*

*Restore all data and address registers from stack*

*Return to command interpreter*

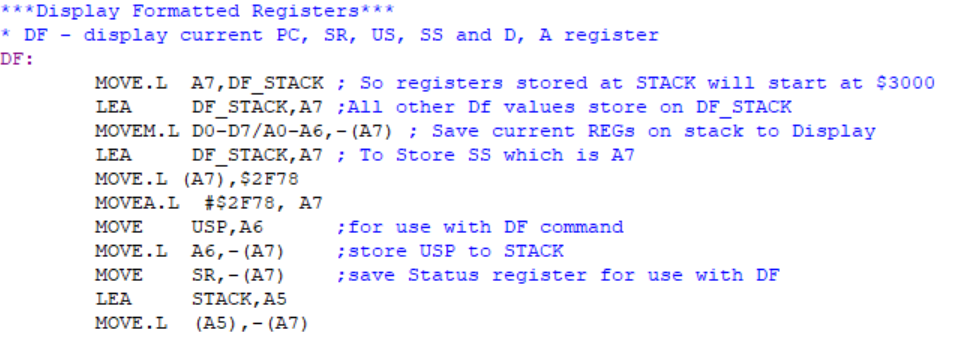
*finish*

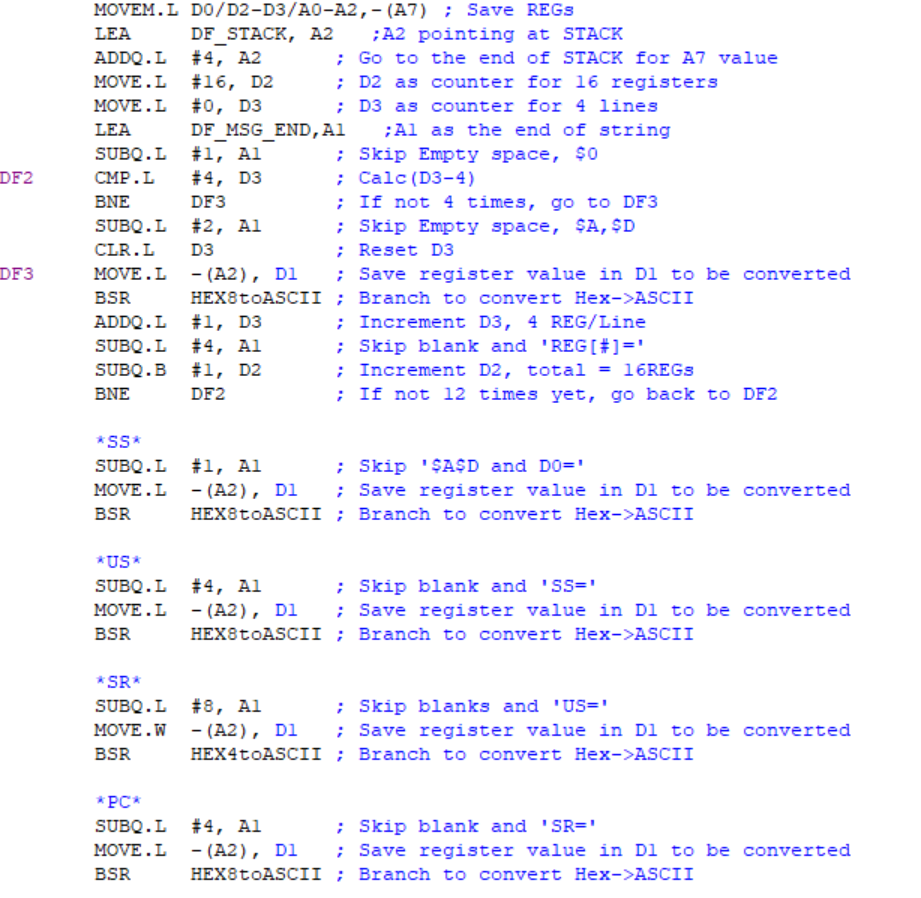
*Figure 2.35. DF Algorithm*

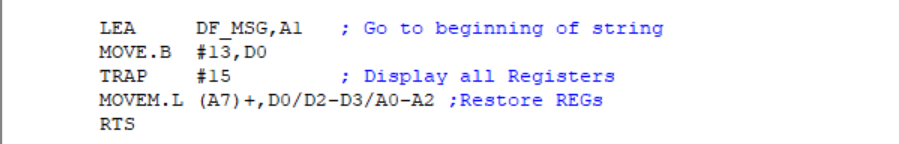
**

*Figure 2.36. DF Flowchart*

***2.2.11.2-) DF Assembly Code***







*Figure 2.37. DF Assembly Code*

***2.2.12-) EXIT***

The algorithm for EXIT terminates/exits Monitor program and restore all registers from stack.

Command’s syntax: **EXIT**

***2.2.12.1-) EXIT Algorithm and Flowchart***

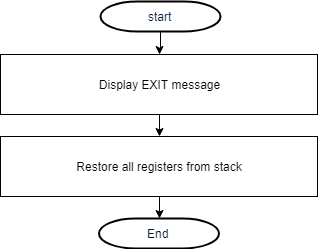
*Display EXIT message*

*Restore all data and address registers from stack*

*End program*

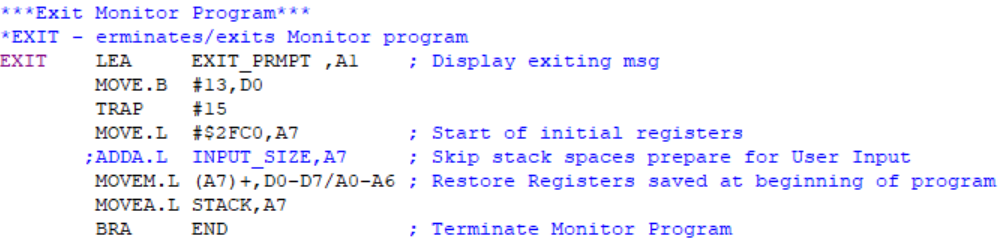
*finish*

*Figure 2.38. EXIT Algorithm*

**

*Figure 2.39. EXIT Flowchart*

***2.2.12.2-) EXIT Assembly Code***



*Figure 2.40. EXIT Assembly Code*

***2.2.13-) ADD***

The algorithm for ADD sums up two hex number and display the result (HEX) with the limit of $7FFFFFFF for each hex number.

Command’s syntax: **ADD <HEX value> <HEX value>**

***2.2.13.1-) ADD Algorithm and Flowchart***

*Save D0,D2,D4,D5,A1 to stack*

*Clear D1 // for storing data*

*D4 = first hex value*

*D5 = second hex value*

*D5 = D4 + D5 // Store the sum in D5*

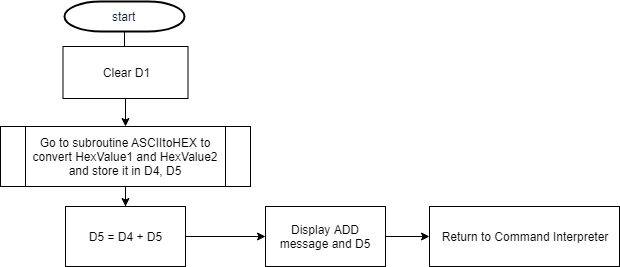
*Display ADD message followed by D5 (sum)*

*Restore D0,D2,D4,D5,A1 from stack*

*Return to command interpreter*

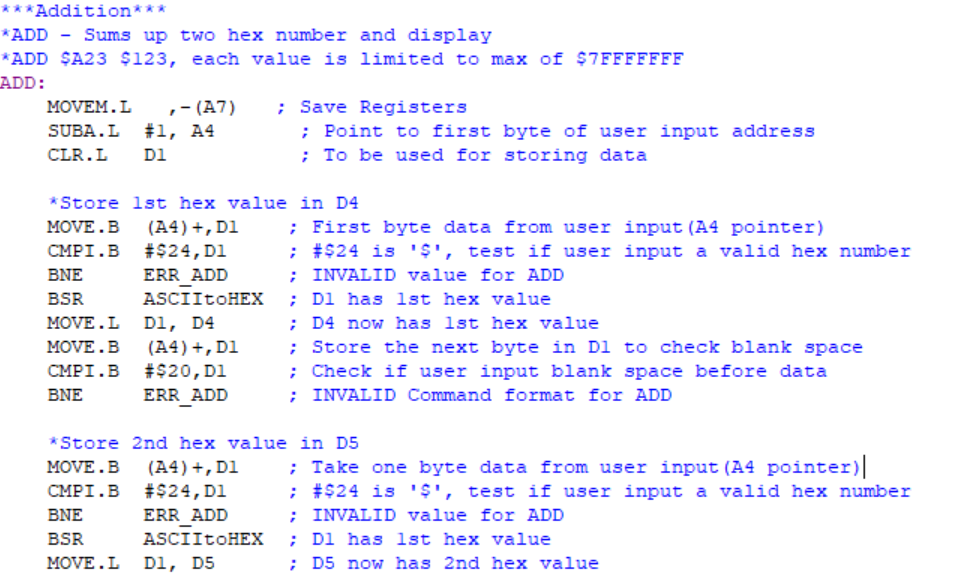
*finish*

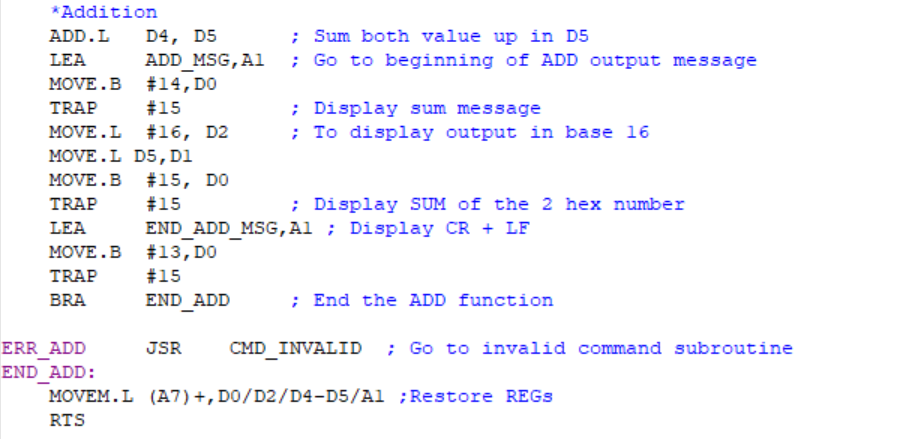
*Figure 2.41. ADD Algorithm*

**

*Figure 2.42. ADD Flowchart*

***2.2.13.2-) ADD Assembly Code***





*Figure 2.43. ADD Assembly Code*

***2.2.14-) D2H***

The algorithm for D2H converts a three digits decimal number into a hexadecimal number and display the result. The decimal number has limits below 255.

Command’s syntax: **D2H <3 digits decimal number>**

***2.2.12.1-) D2H Algorithm and Flowchart***

*Save D2 – D5, A1 to stack*

*Clear D1,D3,D4,D5 // for storing data*

*D3 = 1st decimal number*

*D4 = 2nd decimal number*

*D5 = 3rd decimal number*

*If (D3 == 0) //if 100’s decimal doesn’t exist*

*D4 = D3 // Shift 100’s to 10’s*

*Clear D3*

*If (D4 == 0) //if 10’s decimal doesn’t exist*

*D5 = D4 // Shift 10’s to 1’s*

*Clear D4*

*D3\* $64 //To find 100’s in hexadecimal*

*D4 \* $0A //To find 10’s in hexadecimal*

*D5 = D3+D4+D5 //Sums*

*If (D5 > 255) //If input is more than decimal 255*

*Display error message*

*Store 1’s hex value as ASCII in memory follow by 10’s and 100’s (backward) right behind D2H string*

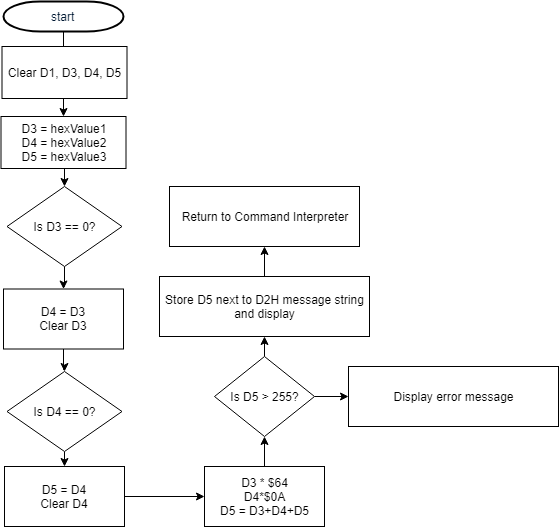
*Display D2H message till null terminator*

*Restore D2 – D5, A1 from stack*

*Return to command interpreter*

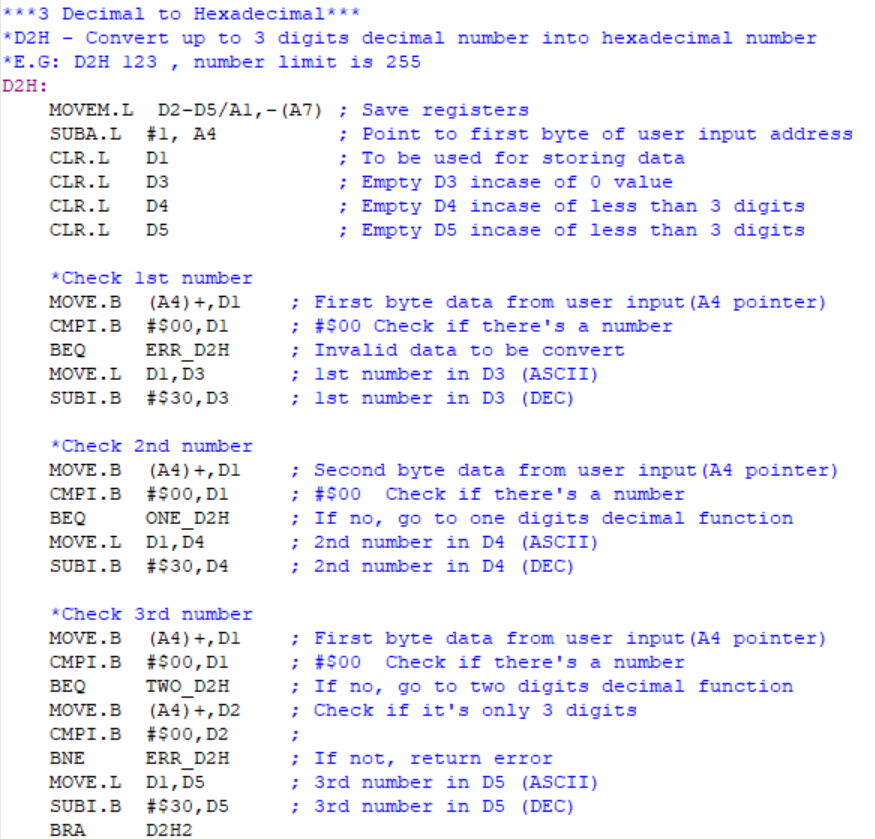
*Finish*

*Figure 2.44. D2H Algorithm*

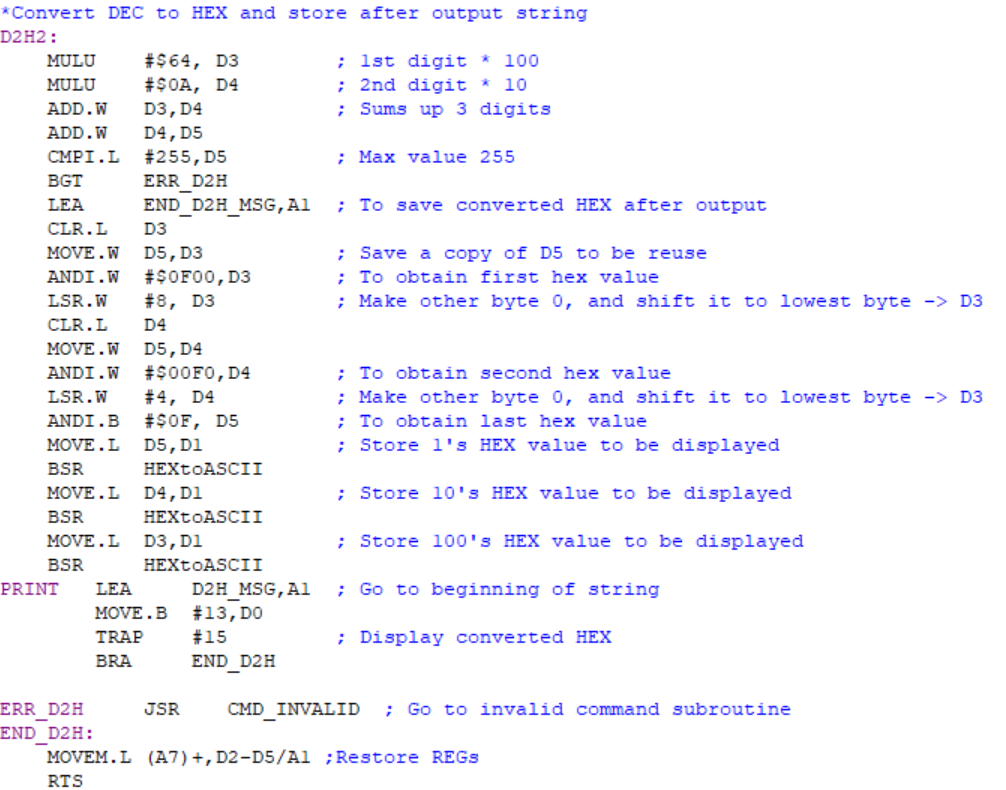
**

*Figure 2.45. D2H Flowchart*

***2.2.12.2-) D2H Assembly Code***







*Figure 2.46. D2H Assembly Code*

***2.3-) Exception Handlers***

In order to handle all exceptions using custom command, the exception vector table was modified to redirect address error, bus error, illegal instruction, division by zero, check instruction error, privilege violation, Line A and F emulator to a new exception handler subroutine. In general, all exception handlers start off by printing the respective error message followed by the DF subroutine to display the current registers exception for Bus and Address errors which required a display for SSW, BA and IR.

***2.3.1-) Bus Error Exception***

Load Bus error exception message and branch to *BUS\_ADS\_EXCPTN* to display error exception message follow by SWW, BA, IR and DF.

***2.3.1.1-) Bus Error Exception Algorithm and Flowchart***

*Save A1,D0 to stack*

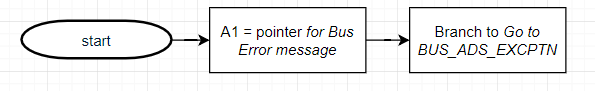
*Save A2,D1 to stack*

*A1 as pointer for Bus Error message*

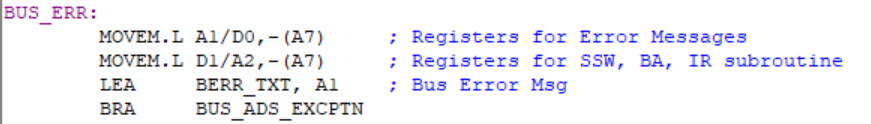
*Go to BUS\_ADS\_EXCPTN //Bus and address error exception subroutine*

*Finish*

*Figure 2.47. Bus Error Exception Algorithm*

 *Figure 2.48. Bus Error Exception Flowchart*

***2.3.1.2-)*** ***Bus Error Exception Assembly Code***



*Figure 2.49. Bus Error Exception Assembly Code*

***2.3.2-) Address Error Exception***

Load Address error exception message and branch to *BUS\_ADS\_EXCPTN* to display error exception message follow by SWW, BA, IR and DF.

***2.3.1.1-) Address Error Exception Algorithm and Flowchart***

*Save A1,D0 to stack*

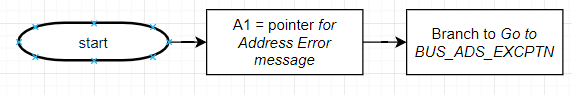
*Save A2,D1 to stack*

*A1 as pointer for Address Error message*

*Go to BUS\_ADS\_EXCPTN //Bus and address error exception subroutine*

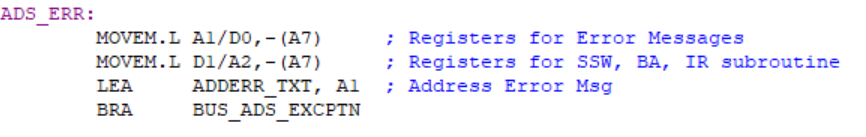
*Finish*

*Figure 2.50. Bus Error Exception Algorithm*



*Figure 2.51. Bus Error Exception Flowchart*

***2.3.1.2-) Address Error Exception Assembly Code***



*Figure 2.52. Bus Error Exception Assembly Code*

***2.3.3-) Illegal Instruction Exception***

Load Illegalerror exception message and branch to *EXCEPTION* to display error exception message follow by DF.

***2.3.3.1-) Illegal Instruction Exception Algorithm and Flowchart***

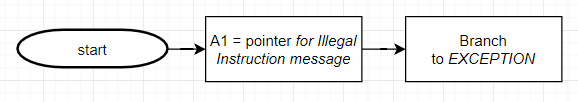
*Save A1,D0 to stack*

*A1 as pointer for Illegal Instruction message*

*Go to EXCEPTION // All error exception subroutine*

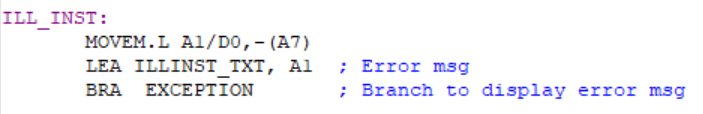
*Finish*

*Figure 2.53. Illegal Instruction Exception Algorithm*



*Figure 2.54. Illegal Instruction Exception Flowchart*

***2.3.3.2-) Illegal Instruction Exception Assembly Code***

*Figure 2.55. Illegal Instruction Exception Assembly Code*

***2.3.4-) Privilege Violation Exception***

Load Privilege Violation exception message and branch to *EXCEPTION* to display error exception message follow by DF.

***2.3.4.1-) Privilege Violation Exception Algorithm and Flowchart***

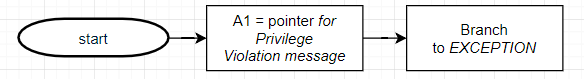
*Save A1,D0 to stack*

*A1 as pointer for Privilege Violation message*

*Go to EXCEPTION // All error exception subroutine*

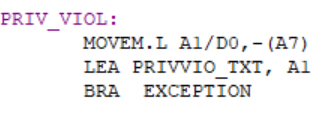
*Finish*

*Figure 2.56. Privilege Violation Exception Algorithm*



*Figure 2.57. Privilege Violation Exception Flowchart*

***2.3.4.2-) Privilege Violation Exception Assembly Code***



*Figure 2.58* *Privilege Violation Exception Assembly Code*

***2.3.5-) Divide by Zero Exception***

Load Divide by Zeroexception message and branch to *EXCEPTION* to display error exception message follow by DF.

***2.3.5.1-) Divide by Zero Exception Algorithm and Flowchart***

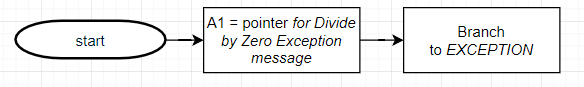
*Save A1,D0 to stack*

*A1 as pointer for Divide by Zero Exception message*

*Go to EXCEPTION // All error exception subroutine*

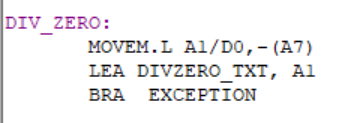
*Finish*

*Figure 2.59. Divide by Zero Exception Algorithm*



*Figure 2.60. Privilege Violation Exception Flowchart*

***2.3.5.2-) Divide by Zero Exception Assembly Code***



*Figure 2.61* *Divide by Zero Exception Assembly Code*

***2.3.6-) Line A and Line F Emulators***

Load Line A or Fexception message and branch to *EXCEPTION* to display error exception message follow by DF.

***2.3.6.1-) Line A and Line F Emulators Algorithm and Flowchart***

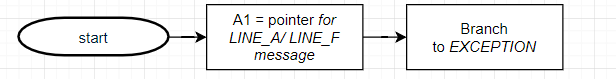
*Save A1,D0 to stack*

*A1 as pointer for LINE\_A or LIEN\_F message*

*Go to EXCEPTION // All error exception subroutine*

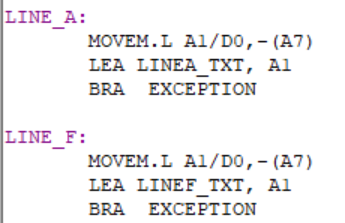
*Finish*

*Figure 2.62* *Line A and Line F Emulators Algorithm*



*Figure 2.63* *Line A and Line F Emulators Flowchart*

***2.3.6.2-) Line A and Line F Emulators Assembly Code***



*Figure 2.64* *Line A and Line F Emulators Assembly Code*

***2.3.7-) Check Instruction Exception***

Load Check Instruction exception message and branch to *EXCEPTION* to display error exception message follow by DF.

***2.3.7.1-) Check Instruction Exception Algorithm and Flowchart***

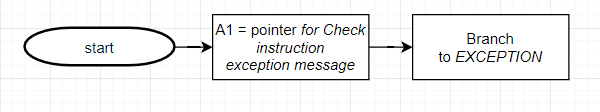
*Save A1,D0 to stack*

*A1 as pointer for Check instruction exception message*

*Go to EXCEPTION // All error exception subroutine*

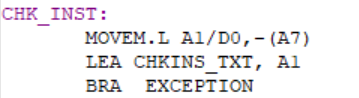
*Finish*

*Figure 2.65* *Check Instruction Exception Algorithm*



*Figure 2.66* *Check Instruction Exception Flowchart*

***2.3.7.2-) Check Instruction Exception Assembly Code***



*Figure 2.67* *Check Instruction Exception Assembly Code*

***2.3.8-) Bus and Address handler Exception***

Display exception message and branch to *EXCEPTION* to display error exception message follow by DF.

***2.3.8.1-) Bus and Address handler Exception Algorithm and Flowchart***

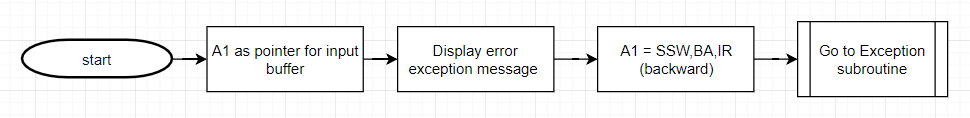
*Display Bus or Address error exception message*

*Retrieve SSW, BA, IR from stack and put them on Input buffer(A1) (backward)*

*Go to EXCEPTION // All error exception subroutine*

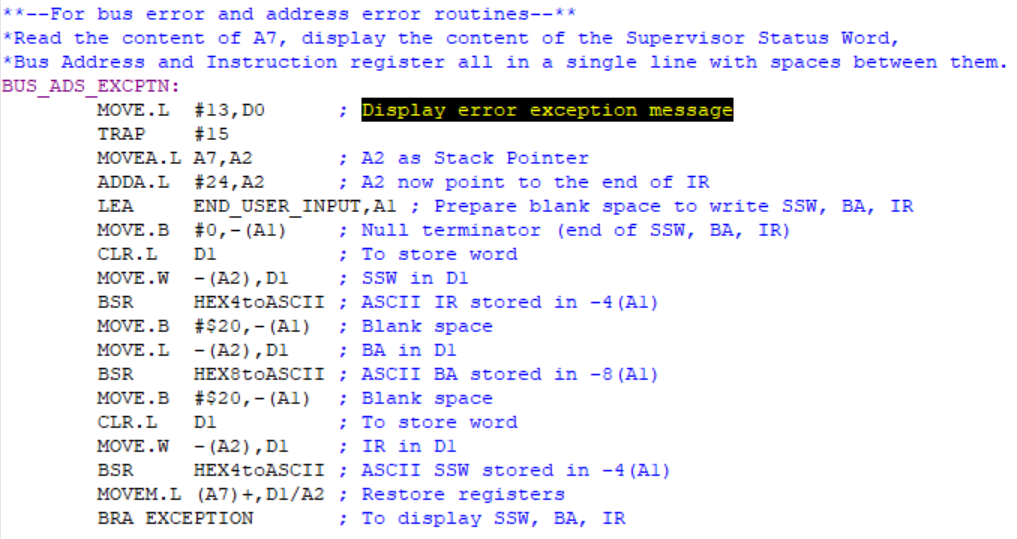
*Finish*

*Figure 2.68* *Bus and Address handler Exception Algorithm*



*Figure 2.69* *Bus and Address handler Exception Flowchart*

***2.3.8.2-) Bus and Address handler Exception Assembly Code***



*Figure 2.70 Bus and Address handler Exception Assembly Code*

***2.3.9-) Error Exception handler Exception***

Display exception message or (SSW, Ba, IR for Bus/ Address error) and branch to *DF* and branch to beginning of program.

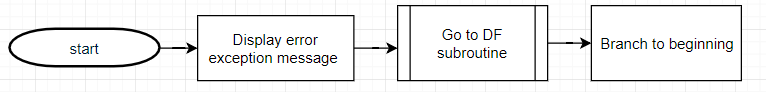
***2.3.9.1-) Error Exception handler Exception Algorithm and Flowchart***

*Display error exception message*

*Go to DF // Display DF*

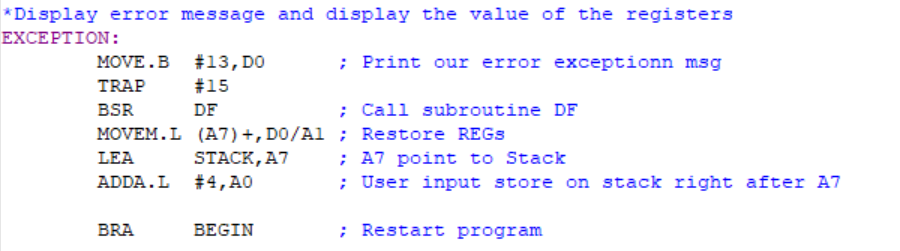
*Finish*

*Figure 2.71* *Error handler Exception Algorithm*



*Figure 2.72* *Error handler Exception Flowchart*

***2.3.8.2-) Error handler Exception Assembly Code***



*Figure 2.73 Error handler Exception Assembly Code*

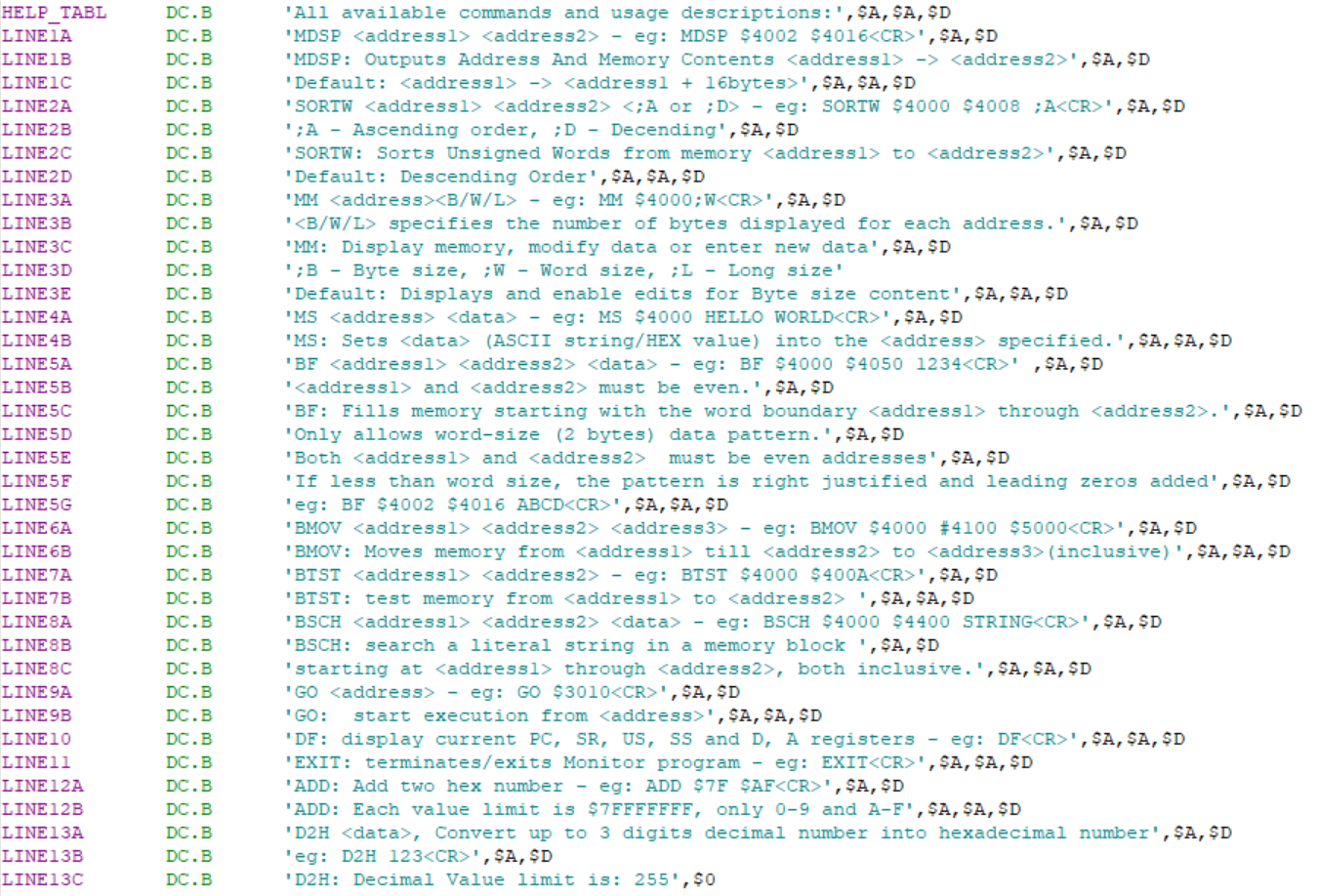
***2.4-) User Instructional Manual Exception Handlers***

The User Instructional Manual Handlers is designed to return guidance which includes ‘Invalid Command’ or Invalid Address’ message to allow easier debugging for the user after the system has checked if user’s input doesn’t match any of the predefined functions in the Monitor Program.

***2.4.1-) HELP Menu***

The HELP Menu is a string message that can be display using the HELP command in terminal, it contains all the possible command in the monitor program and a detail description and examples to ensure user can access all the commands.

***2.4.1.1-)HELP menu Assembly Code***



*Figure 2.74. HELP Menu Assembly Code*

***2.4.2-) Invalid Message Prompt***

The invalid message prompt display a more accurate of the mistake make by user for the user to narrow down the scope of typing error and fix them.

***2.4.2.1-) Invalid Message Prompt Algorithm and Flowchart***

*Save register D0, A1 to stack*

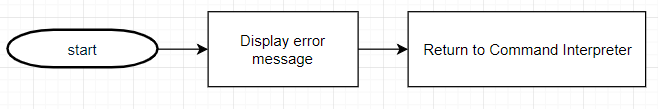
*Display Invalid Address/Command Message*

*Restore register D0,A1 from stack*

*Return to command interpreter*

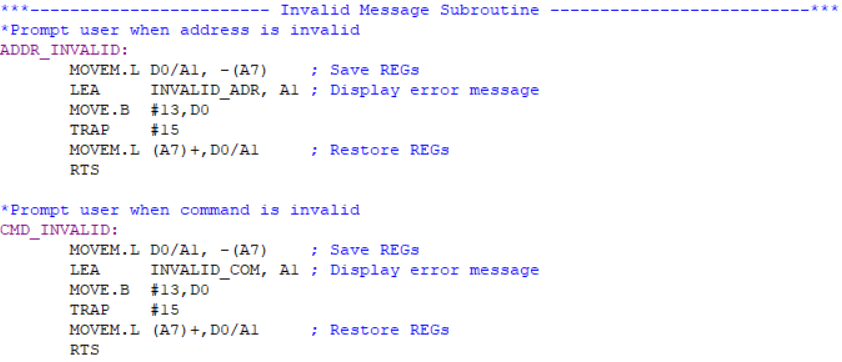
*finish*

*Figure 2.75. Invalid Message Prompt Algorithm*



*Figure 2.76* *Error handler Exception Flowchart*

***2.4.2.1-) Invalid Message Prompt Assembly Code***



*Figure 2.77. Invalid Message Prompt Assembly Code*

***3-) Discussion***

A good terminal program or just any program in general, should be solid and low to almost zero error rate. The first challenge in the design of this monitor program is the ability to produce no accountable error. For example, the command interpreter has to make check user input with the commands in command table and prompt ‘Invalid Command’ message if no match shows up and jump to the desired subroutine if and only if the command fully matches (case-sensitive). Another good instance is all the commands that takes in a parameter of address should always check if user input a ‘$’ sign before the address and before any HEX value, else a ‘Invalid Address’ message should be returned. All these checking procedures require a large amount of time to be dedicated into planning and testing every single command to make sure erroneous will not halt the program.

Another design challenge is the memory constraint. Due to the extra steps needed mentioned previously, each command will need to expand one third of its initial program size and custom error exception handlers were made to allow easier debugging on user end. All these extra line of codes makes the process of letting the program to stay within a memory size of 3k byte tough.

A good program should be structural. So, a detail planning was done before the code was written. For instance, all different commands and exception handlers were broken down into basic subtask. These subtasks are being developed independently and tested before being integrated into the main program. Being structural also improves the readability of the program, which is another aspect of a good program. All lines were commented, and a brief explanation of the subprogram was printed on top of each subtasks to allow easy understanding. The program also comes with a user/instruction manual which can be opened by running the HELP command, an example of how each command should be used was displayed to reduce the error rate as much as possible.

In a nutshell, a lot planning and restructuring of the program must be done to produce a terminal program that run smoothly, errorless, and is compacted in code-wise such as optimization. For example, repeating codes were summarize with a loop function to reduce a great amount of program size such as all the exceptions handlers share the same display function.

***4-) Feature Suggestions***

The first feature suggestion would be the generality and the flexibility of the monitor program. The program is dedicated towards the design of the SANPER-ELU and all the functions implanted centers around it. For example, the functionality of the command handler should be more flexible, when the SANPER-ELU went through an hardware upgrade, the program should be able to incorporate the changes without having a rewrite of the codes.

The commands implemented for this program should be more general as in if it being paired with other microprocessor/ machine, it should only takes a few integration process to bring the monitor program up to work. A good example of this would be that it is long and complicated process to get the Easy68k software to run on Mac OS.

Another feature suggestion is that this project should be pursued as a group project instead of individual. The reason being the amount of function that can be produced individually is limited and only consist of basic functions. A team of two would allow production of monitor program with more complex functions that rivals an actual terminal program as well as a more solid program as it is always good to have two people who constantly cross checking each other codes to climates logic mistakes and maintain program flows.

***5-) Conclusion***

In a nutshell, this project is a success as all fourteen debugger functions and eight exception handlers were tested and proven to be functioning as they were intended to be. The general flow of this program provides a platform for user to interact with MC68000 microprocessor in a user-friendly manner. All the constrains were also fulfilled although it requires a lot effort was put on the program optimization. The methodology used was a wise choice as it greatly improves the development process efficiency. This project acts as a good capstone project for student to incorporate all the knowledge and experience learn throughout the ECE classes into the development of this project.

***6-) References***

[1] Experiment 1 Lab Manual

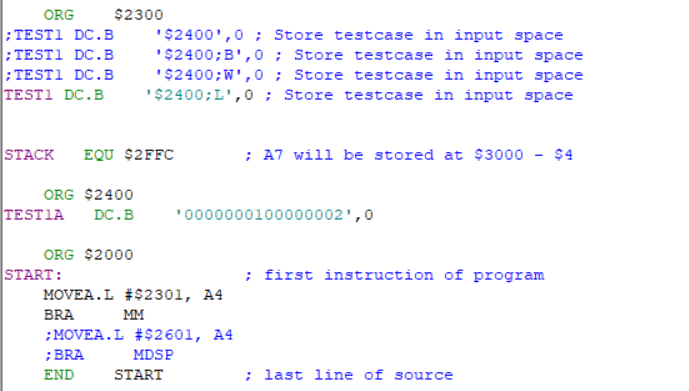
[2] Experiment 2 Lab Manual

[3] Experiment 3 Lab Manual

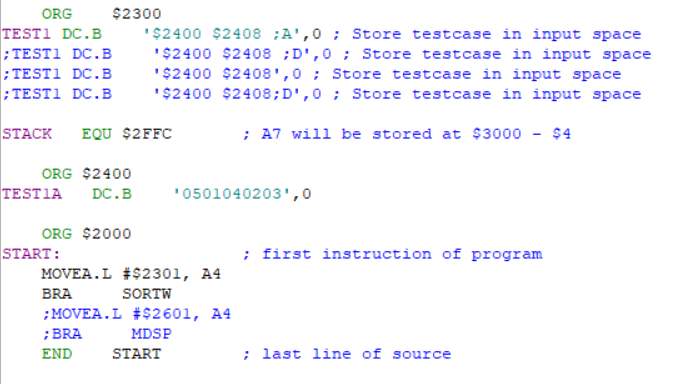
[4] MC68000 Educational Computer Board User’s Manual

[5] MOTOROLA MC68000 FAMILY Programmer’s Reference Manual

***7-) Appendix***



*Figure 2.78. Testcases for MM - Assembly Code*



*Figure 2.79. Testcases for SORTW - Assembly Code*