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

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.

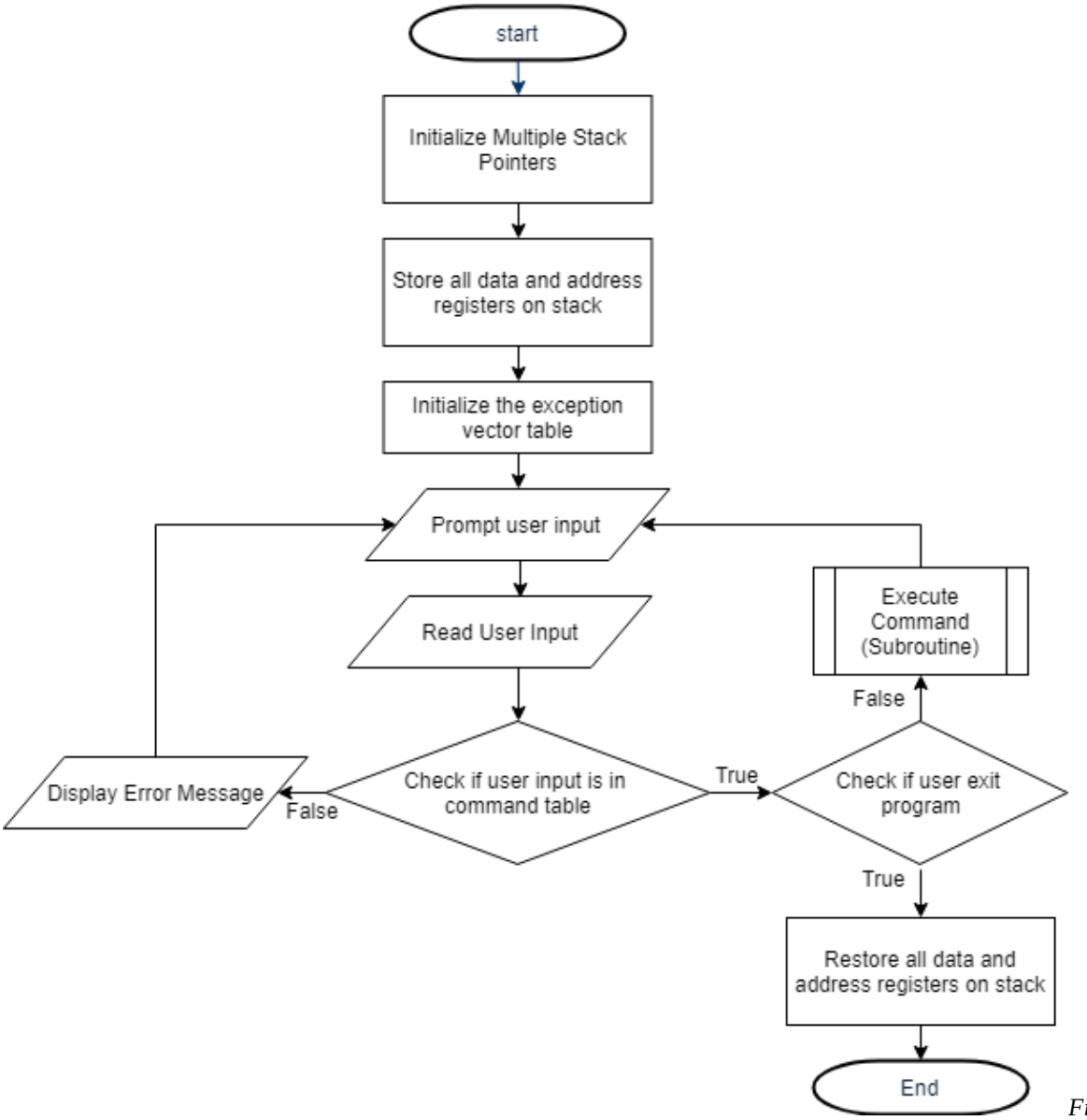

Figure 2.1. Monitor program block diagram

```
***-----------------------------Main Program-----------------------------
    ORG    $1D00
START:                  ; first instruction of program
STACK_INI_REG EQU $2FFC   ; A7 will be stored at $3000 - $4
DF_STACK      EQU $2FB8
STACK         EQU $2FBC

        MOVE.L  A7,STACK_INI_REG ; So registers stored at STACK will star
        LEA     STACK_INI_REG,A7
        MOVEM.L D0-D7/A0-A6,-(A7); Save registers on STACK to enable rest
        ;LEA     STACK,A7
        ;ADDQ.W  #4,A7

***Exception Vector table***
        MOVE.L #STACK,    $0
        MOVE.L #BUS_ERR,  $8
        MOVE.L #ADS_ERR,  $C
        MOVE.L #ILL_INST, $10
        MOVE.L #DIV_ZERO, $14
        MOVE.L #CHK_INST, $18
        MOVE.L #PRIV_VIOL, $20
        MOVE.L #LINE_A,   $28
        MOVE.L #LINE_F,   $2C
```

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-)

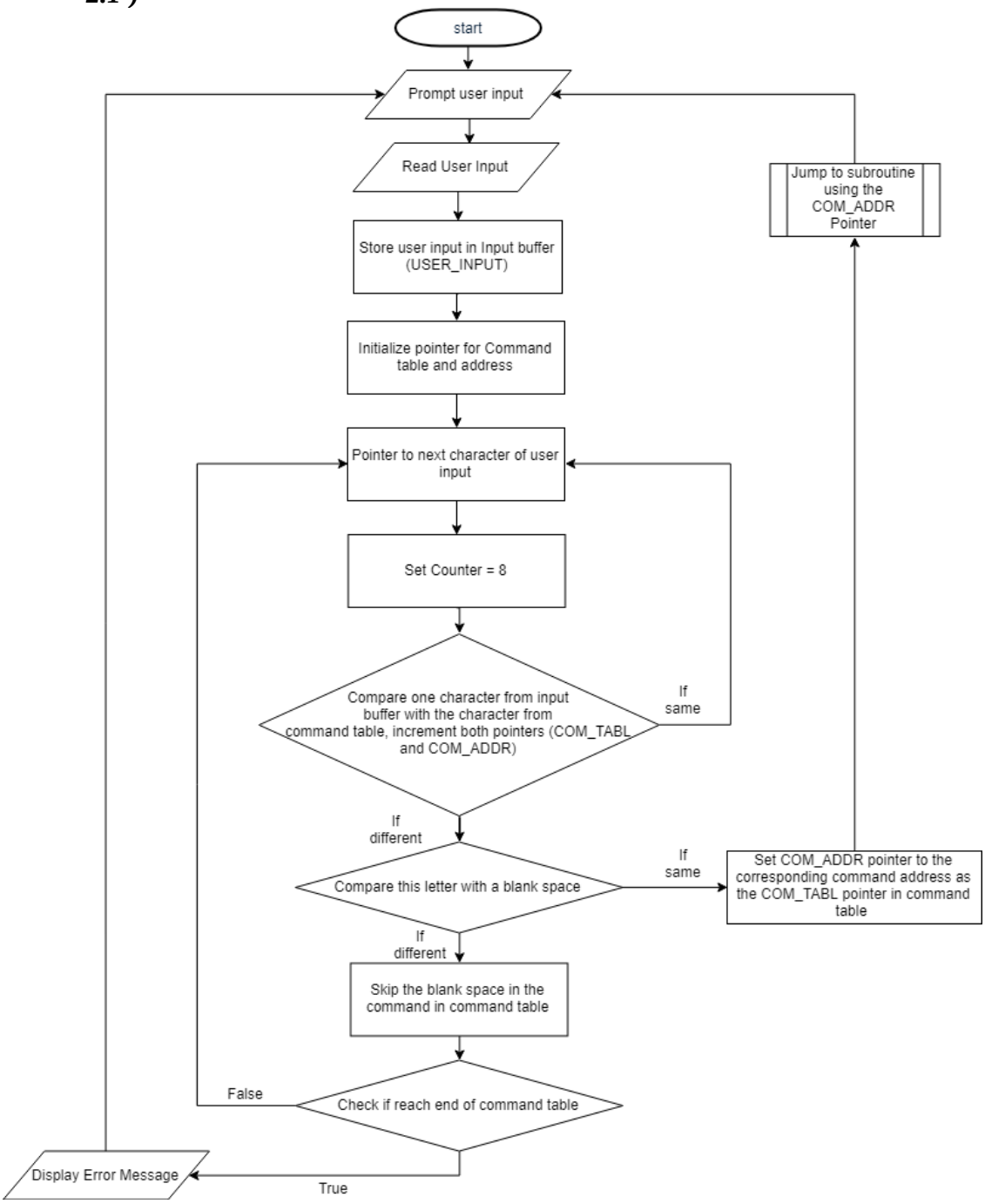

Figure 2.3. Command Interpreter Flowchart

2.1.2-) Command Interpreter Assembly Code

```
***---------------------------- Command Interpreter -----------------------------***
*A1, A4 point to user input
BEGIN     LEA     PROMPT, A1   ; Prompt for user input "MONITOR441>"
          MOVE.L  #14,D0       ; Display Prompt
          TRAP    #15
          LEA     USER_INPUT, A1    ; When user input, store to reserve user input space
          MOVE.L  #2,D0        ; Read input ans return to (A1)
          TRAP    #15

          LEA     COM_TABL,A2  ; Load Command Table, pointer on the command names.
          LEA     COM_ADDR,A3  ; Load Command ADDR, pointer on the addr of the subroutines
          CLR.L   D2           ; Command Counter
LOOKUP    MOVEA.L A1,A4        ; Point to user input
          MOVE.L  #9, D4
CMP_Char  SUBI.B  #1, D4
          CMPM.B  (A2)+, (A4)+ ; Compare byte (input) vs (CMD table)
          BEQ     CMP_Char     ; If same, keep comparing
          CMPI.B  #$20, -(A2)  ; Check if reach blank space
          BEQ     Run_CMD      ; If 0, means all strings are same, select command
NEXTCMD   ADDA.L  #1, A2       ; Loop thru the rest of blank spaces
          SUBI.B  #1, D4
          BNE     NEXTCMD      ;
          ADDQ.L  #2,D2        ; Else, point to next cmd address
          ;ADDQ.L  #8,A4        ; Go to next command
          CMPA.L  A2,A3        ; Check if reach end of COM_TABL
          BGE     LOOKUP       ; Else, keep looking up command
          BSR     CMD_INVALID  ; Prompt Invalid command
          BRA     BEGIN        ; Start prompt again

Run_CMD   ADDA.L  D2,A3        ; point to the cmd adr in COM_ADDR
          MOVEA.L #0,A5        ; clear A5, used for subroutine call
          MOVEA.W (A3),A5      ; move that command's address to register
          JSR     (A5)         ; jump to that command's subroutine (below)
          BRA     BEGIN        ; Prompt for new command
```

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

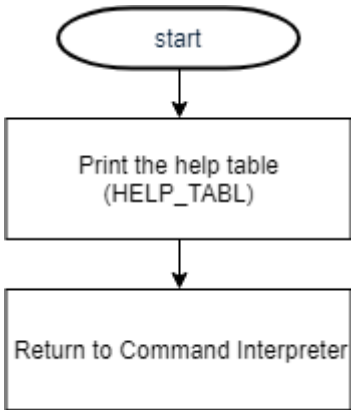

Figure 2.6. HELP Flowchart

2.2.1.2-) HELP Assembly Code

```
***--------------------------- Debugger Commands -----------------------------***
***HELP***
HELP:
        MOVEM.L D0/A1,-(A7)    ; Save register to A7(Stack)
        LEA     HELP_TABL,A1   ; Load HELP TABLE's addr
        MOVE.B  #13,D0
        TRAP    #15            ; Display HELP TABLE
        MOVEM.L (A7)+,D0/A1    ; restore registers
        RTS
```

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

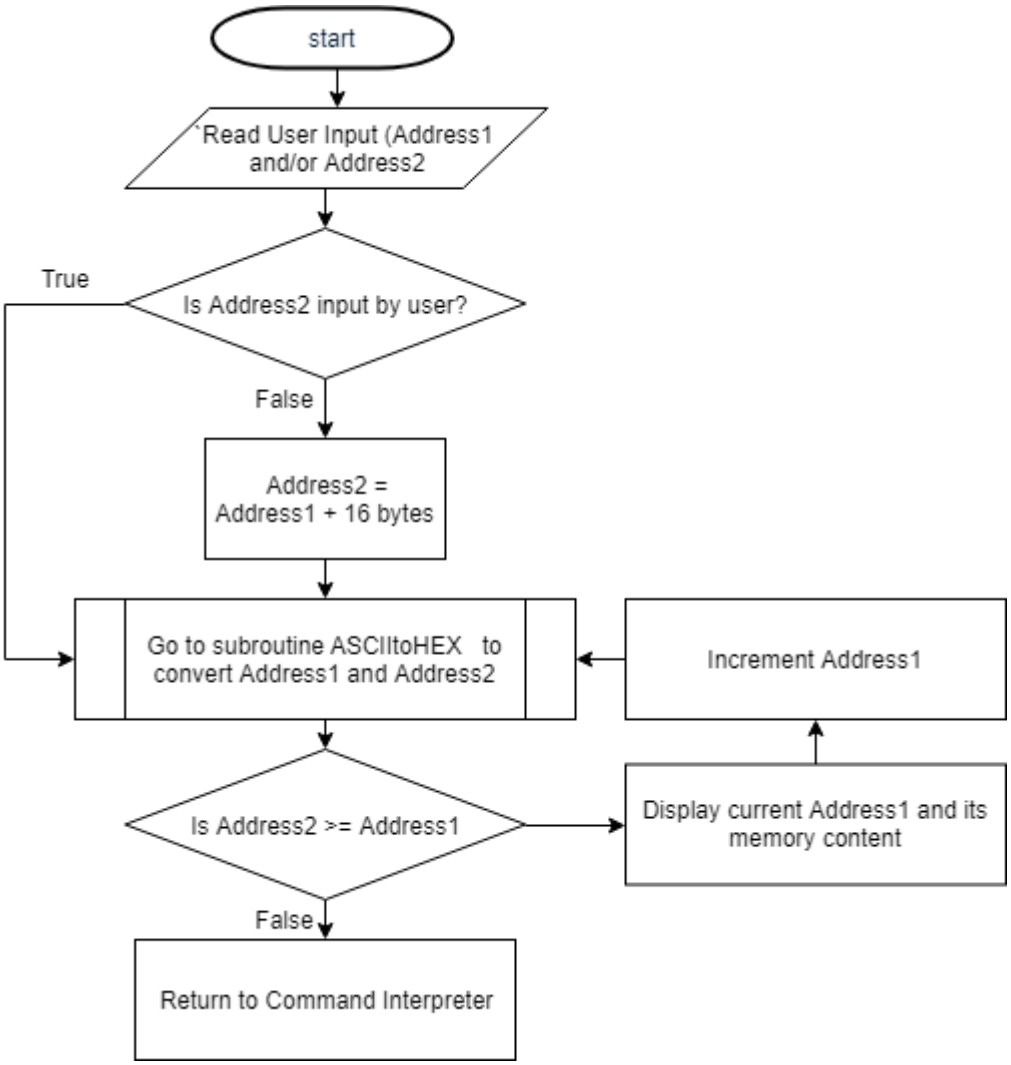

Figure 2.9. MDSP Flowchart

2.2.2.2-) MDSP Assembly Code

```
***Memory Display***
*MDSP - outputs the address and memory contents <address1> to <address2>
*Default: outputs the address and memory contents <address1> to <address1 + 16bytes>
MDSP:
       MOVEM.L D0/A1/A5-A6,-(A7)
       *Store address1 in A5
       SUBA.L  #1, A4        ; Point to first byte of user input address
       MOVE.B  (A4)+,D1      ; First byte data from user input(A4 pointer)
       CMPI.B  #$24,D1       ; #$24 is '$', test if user input a valid address sign
       BNE     ERR_MDSP      ; INVALID Address for BTST
       BSR     ASCIItoHEX    ; D1 has 1st address(ASCII)
       MOVEA.L D1, A5        ; 1st HEX addr in A5

       *Check If there's a second address, if yes, store address2 in A6
       MOVE.B  (A4)+,D1      ; Store the next byte in D1 to check blank space
       CMPI.B  #$20,D1       ; Check if user input blank space before Next address
       BNE     NO_ADDR2      ; INVALID Command format for BTST
       MOVE.B  (A4)+,D1      ; One byte data from user input(A4 pointer)
       CMPI.B  #$24,D1       ; #$24 is '$', test if user input a valid address sign
       BNE     ERR_MDSP      ; INVALID Address for BTST
       BSR     ASCIItoHEX    ; D1 has 2nd address(ASCII)
       MOVEA.L D1,A6         ; 2nd HEX addr in A6
       BRA     MDSP2

         *If no input addre2, addr2 = addr1 + 16 bytes
NO_ADDR2 MOVEA.L A5,A6       ; make a copy of addr1
         ADDA.L  #15,A6      ; Add 16 bytes to addr1(first byte is counted)
       *Outputs the address and memory contents from A5 to A6
MDSP2  CMPA.L  A5, A6        ; Compare if A5 is at A6 (A6-A5),
       BLT     END_MDSP      ; If A5 is at A6, go to to next loop
       LEA     STACK, A1     ; Use STACK to store byte tobe printed out
       SUBA.L  #$50,A1       ; Skip spaces to prevent stack overflow
       MOVE.B  #$00,-(A1)    ; Null terminator
       CLR.L   D1            ; Clear to store byte data
       MOVE.B  (A5), D1      ; Memory content to be converted
       BSR     HEX2toASCII
       MOVE.B  #$20,-(A1)    ; Blank space
       MOVE.B  #$20,-(A1)    ; Blank space
       MOVE.W  A5, D1        ; Address to be converted
       BSR     HEXtoASCII  ; Convert Address to ASCII
       MOVE.B  #13,D0        ; Display the address and memory contents
       TRAP    #15
       ADDQ.L  #1, A5        ; Go to next byte addr
       BRA     MDSP2         ; Continue Loop

ERR_MDSP JSR    CMD_INVALID  ; Go to invalid command subroutine
END_MDSP MOVEM.L (A7)+,D0/A1/A5-A6 ;Restore used Register
         RTS
```

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

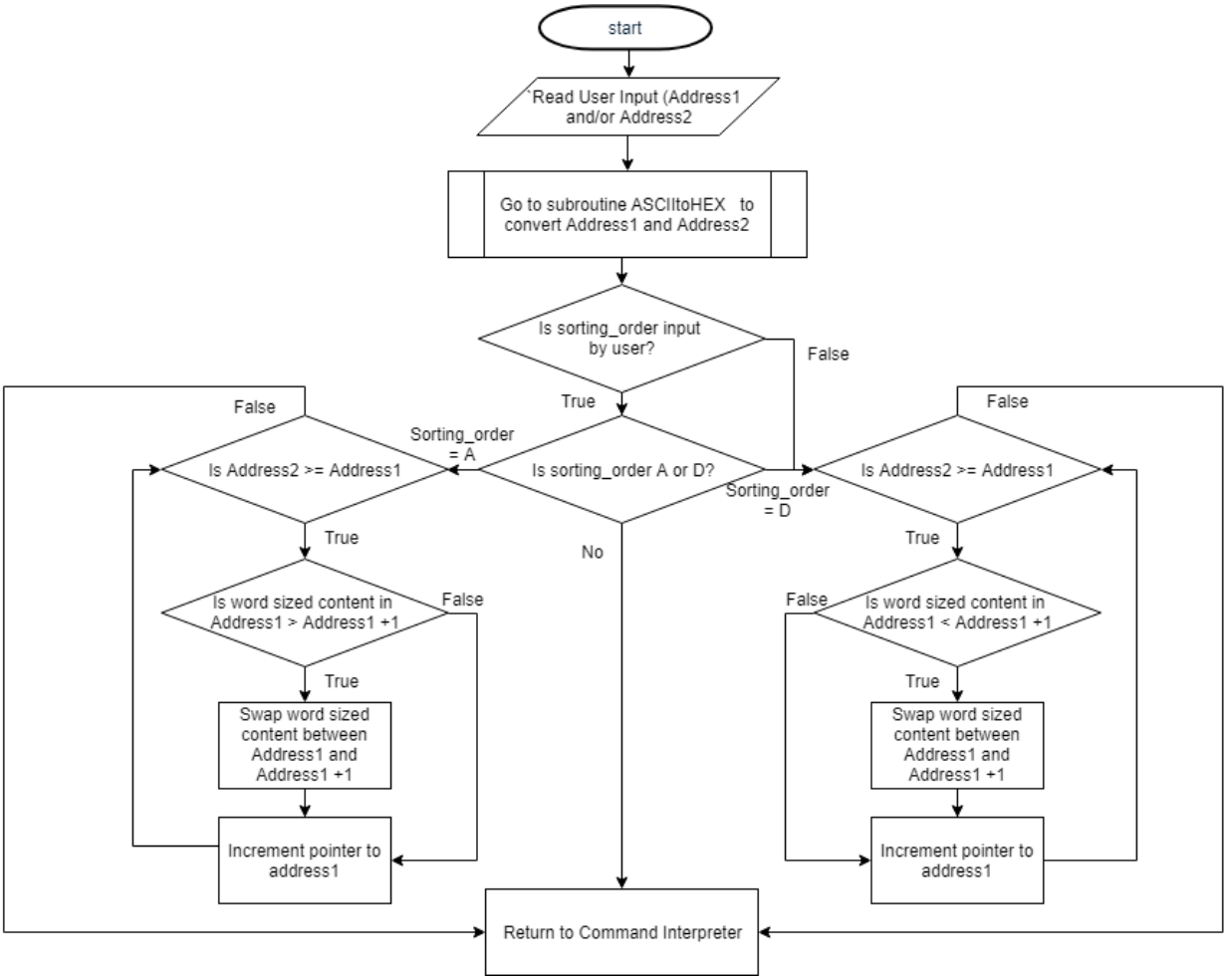

Figure 2.12. SORTW Flowchart

2.2.3.2-) SORTW Assembly Code

```
***Sort in Word***
*SORTW -  sorts <address1> to <address2> in word size data
*(A or D) specifies whether the list is sorted in Ascending or Descending order
SORTW:
       MOVEM.L D1-D2/A1-A2/A5-A6,-(A7)

       *Store addr1 in A5
       SUBA.L  #1, A4        ; Point to first byte of user input address
       MOVE.B  (A4)+,D1      ; First byte data from user input(A4 pointer)
       CMPI.B  #$24,D1       ; #$24 is '$', test if user input a valid address sign
       BNE     ERR_SORTW     ; INVALID Address for BTST
       BSR     ASCIItoHEX    ; D1 has 1st address(ASCII)
       MOVEA.L D1, A5        ; 1st HEX addr in A5

       *Store addr2 in A6
       MOVE.B  (A4)+,D1      ; Store the next byte in D1 to check blank space
       CMPI.B  #$20,D1       ; Check if user input blank space before Next address
       BNE     ERR_SORTW     ; INVALID Command format for BTST
       MOVE.B  (A4)+,D1      ; One byte data from user input(A4 pointer)
       CMPI.B  #$24,D1       ; #$24 is '$', test if user input a valid address sign
       BNE     ERR_SORTW     ; INVALID Address for SORTW
       BSR     ASCIItoHEX_MM ; D1 has 2nd address(ASCII)
       MOVEA.L D1,A6         ; 2nd HEX addr in A6

       MOVE.B  (A4)+,D1      ; Store the next byte in D1 to check blank space
       CMPI.B  #$20,D1       ; Check if user input blank space before sorting order
       BEQ     CHK_AD        ; Check if A or D is input
       CMPI.B  #';',D1       ; Check if user input blank space before semicolon
       BEQ     ERR_SORTW     ; If yes, invalid command format
       CMPI.B  #$00,D1       ; Check if there is any size input
       BEQ     DEF_SORTW     ; use default: descending (D1=0)
CHK_AD MOVE.B  (A4)+,D1      ; Store the next byte in D1 to check semi-colon
       CMPI.B  #';',D1       ; Check if user input semi-colon before sorting order
       BNE     ERR_SORTW     ; INVALID command format for SORTW
       MOVE.B  (A4)+,D1      ; Store the next byte in D1 to check sorting order
       CMPI.B  #$41, D1      ; Check if it's 'A'
       BEQ     A_SORTW       ; Go to sort in Ascending
       CMPI.B  #$44, D1      ; Check if it's 'D'
       BNE     ERR_SORTW     ; INVALID command format for SORTW

*Descending Sorting (DEFAULT)
DEF_SORTW  MOVEA.L A5,A2       ; Backup first addr to run nested loop
DSORT_LOOP CMP.W   (A2)+,(A2)+ ; compare next two numbers
           BHI     D_SWAPW     ; If next > prev word data
           SUBQ.L  #2,A2       ; Point back to the last word
           CMP.L   A2,A6       ; Check if A2 reaches A6
           BNE     DSORT_LOOP  ; If not, go back to comparing
           BRA     END_SORTW   ; Else, end the sorting
D_SWAPW    MOVE.L  -(A2),D2     ; Store the 2 words being compared
           SWAP.W  D2           ; Exchange Upper and lower word
           MOVE.L  D2,(A2)      ; Store back in memory
           BRA     DEF_SORTW    ; Continue next 2 comparisons

*Ascending Sorting
A_SORTW    MOVEA.L A5,A2       ; Backup first addr to run nested loop
ASORT_LOOP CMP.W   (A2)+,(A2)+ ; compare next two numbers
           BCS     A_SWAPW     ; If prev > next word data
           SUBQ.L  #2,A2       ; Point back to the last word
           CMP.L   A2,A6       ; Check if A2 reaches A6
           BNE     ASORT_LOOP  ; If not, go back to comparing
           BRA     END_SORTW   ; Else, end the sorting
A_SWAPW    MOVE.L  -(A2),D2     ; Store the 2 words being compared
           SWAP.W  D2           ; Exchange Upper and lower word
           MOVE.L  D2,(A2)      ; Store back in memory
           BRA     A_SORTW      ; Continue next 2 comparisons

ERR_SORTW  JSR    CMD_INVALID  ; Go to invalid command subroutine
END_SORTW  MOVEM.L (A7)+,D1-D2/A1-A2/A5-A6 ;Restore used Register
           RTS
```

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

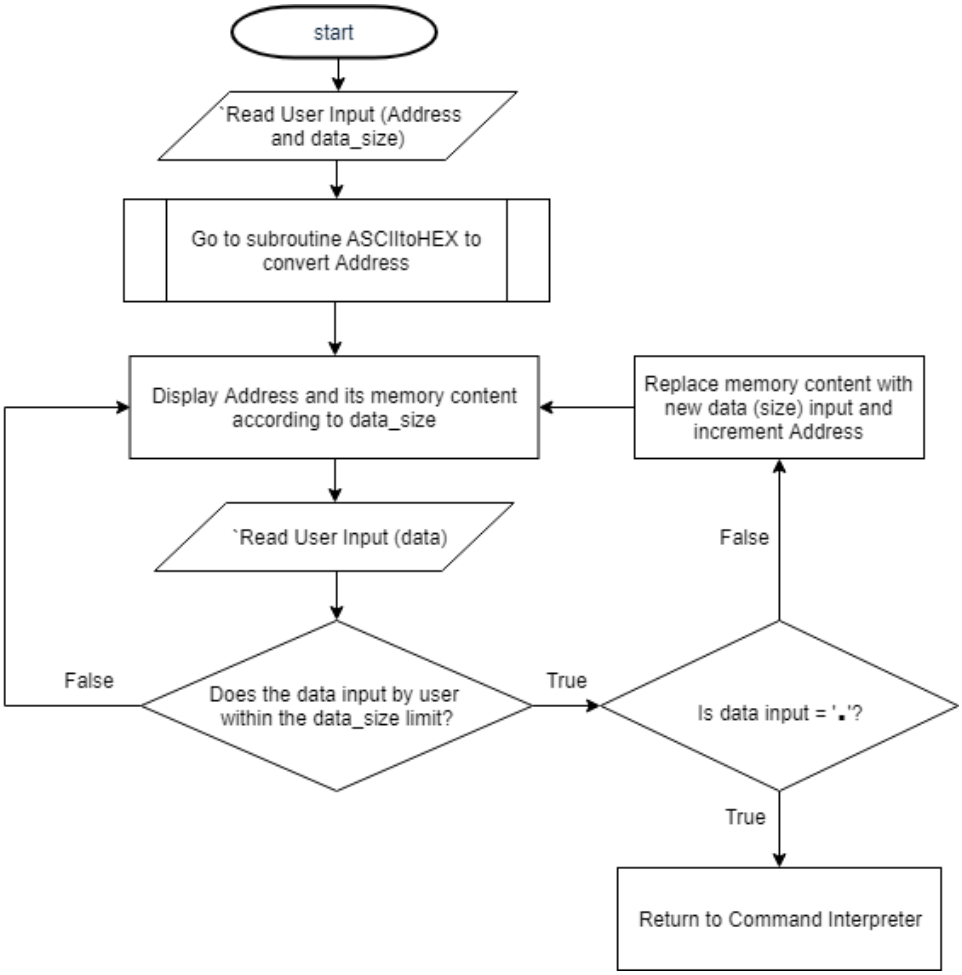

Figure 2.15. MM Flowchart

2.2.4.2-) MM Assembly Code

```
***Memory Modify***
*MM -  display memory and, as required, modify data or enter new data
* The size (B,W,L) controls the number of bytes displayed for each address.
MM:
        MOVEM.L D0-D1/A1/A5,-(A7)

        *Store address to MM at A5
        SUBA.L  #1, A4      ; Point to first byte of user input address
        MOVEA.L A4,A1       ; A1 points to User Input for Writing
        MOVE.B  (A4)+,D1    ; First byte data from user input(A4 pointer)
        CMPI.B  #$24,D1     ; #$24 is '$', test if user input a valid address sign
        BNE     ERR_MM      ; INVALID Address for MM
        BSR     ASCIItoHEX_MM ; D1 has 1st address(ASCII)
        MOVEA.L D1, A5      ; 1st HEX addr in A5

        *Determine between byte(DEFAULT), word or long size
        MOVE.B  (A4)+,D1    ; Store the next byte in D1 to check size
        CMPI.B  #$00,D1     ; Check if there is any size input
        BEQ     BYTE_MM     ; Use byte size MM (DEFAULT)
        CMPI.B  #';',D1     ; Check if user input semi-colon before sorting order
        BNE     ERR_MM      ; INVALID command format for MM
        MOVE.B  (A4),D1    ; Store the next byte in D1 to check size
        CMPI.B  #$42,D1     ; Check if it's 'B'
        BEQ     BYTE_MM     ; Use byte size MM
        CMPI.B  #$57,D1     ; Check if it's 'W'
        BEQ     WORD_MM     ; Use word size MM
        CMPI.B  #$4C,D1     ; Check if it's 'L'
        BEQ     LONG_MM     ; Use long size MM
        BRA     ERR_MM      ; Invalid command for MM
       *Display BYTE memory addressing mode
BYTE_MM LEA     END_USER_INPUT, A1  ; Empty space to store display memory and data
        MOVE.B  #$0,-(A1)   ; Null terminator
        MOVE.B  #$3F,-(A1)  ; Prompt for user input with '?'
        MOVE.B  #$20,-(A1)  ; similiar to SANPER
        CLR.L   D1          ; Clear to store byte data
        MOVE.B  (A5),D1     ; D1 has a byte data
        BSR     HEX2toASCII  ; Store the converted byte into output
        MOVE.B  #$20,-(A1)   ; 4 empty spaces
        MOVE.B  #$20,-(A1)   ;
        MOVE.B  #$20,-(A1)   ;
        MOVE.B  #$20,-(A1)   ;
        MOVE.L  A5,D1       ; HEX addr in A5
        BSR     HEX8toASCII ; Store ASCII addr in output
        MOVE.B  #$24,-(A1)  ; Store '$' hex addr sign in output
        MOVE.B  #14,D0      ; Display memory addr and data
        TRAP    #15
        MOVE.B  #2,D0       ; Read input ans return to (A1)
        TRAP    #15
        CMPI.B  #$0,(A1)    ; Check if user input(null terminated) nothing
        BNE     BYTE_MM2    ; If user input smtg, intpret input
        ADDA.L  #1,A5       ; Point to next byte data
        BRA     BYTE_MM     ; Proceed to next byte data

        *Read user input and terminate/ store input data
BYTE_MM2 CMPI.B  #$2E,(A1)   ; Check if user enter '.' to terminate
        BEQ     END_MM      ; if yes, end MM
        MOVEA.L A1,A4       ; Else, point A4 to current byte data
        BSR     ASCIItoHEX  ; convert user input into HEX
        CMPI.L  #$FF,D1     ; Check if user input more than a byte
        ;BGT     ERR_MM     ; Prompt Invalid input?
        BGT     BYTE_MM     ; If yes, prompt user again
        MOVE.B  D1,(A5)+    ; Replace it with HEX byte user input
        BRA     BYTE_MM     ; Proceed to next byte data
```

```
 *Display WORD memory addressing mode
WORD_MM LEA     END_USER_INPUT, A1  ; Empty space to store display memory and data
        MOVE.B  #$0,-(A1)    ; Null terminator
        MOVE.B  #$3F,-(A1)   ; Prompt for user input with '?'
        MOVE.B  #$20,-(A1)   ; similiar to SANPER
        CLR.L   D1           ; Clear to store byte data
        MOVE.W  (A5),D1      ; D1 has a WORD data
        BSR     HEX4toASCII  ; Store the converted word  into output
        MOVE.B  #$20,-(A1)   ; 4 empty spaces
        MOVE.B  #$20,-(A1)   ;
        MOVE.B  #$20,-(A1)   ;
        MOVE.B  #$20,-(A1)   ;
        MOVE.L  A5,D1        ; HEX addr in A5
        BSR     HEX8toASCII ; Store ASCII addr in output
        MOVE.B  #$24,-(A1)   ; Store '$' hex addr sign in output
        MOVE.B  #14,D0       ; Display memory addr and data
        TRAP    #15
        MOVE.B  #2,D0        ; Read input ans return to (A1)
        TRAP    #15
        CMPI.B  #$0,(A1)     ; Check if user input(null terminated) nothing
        BNE     WORD_MM2     ; If user input smtg, intepret input
        ADDA.L  #2,A5        ; Point to next word data
        BRA     WORD_MM      ; Proceed to next word  data

         *Read user input and terminate/ store input data
WORD_MM2 CMPI.B  #$2E,(A1)    ; Check if user enter '.' to terminate
         BEQ     END_MM       ; if yes, end MM
         MOVEA.L A1,A4        ; Else, point A4 to current byte data
         BSR     ASCIItoHEX   ; convert user input into HEX
         CMPI.L  #$FFFF,D1    ; Check if user input more than a word
         BGT     WORD_MM      ; If yes, prompt user again
         MOVE.W  D1,(A5)+     ; Replace it with HEX word user input
         BRA     WORD_MM      ; Proceed to next word data
*Display LONG memory addressing mode
LONG_MM LEA     END_USER_INPUT, A1  ; Empty space to store display memory and data
        MOVE.B  #$0,-(A1)    ; Null terminator
        MOVE.B  #$3F,-(A1)   ; Prompt for user input with '?'
        MOVE.B  #$20,-(A1)   ; similiar to SANPER
        MOVE.L  (A5),D1      ; D1 has a LONG data
        BSR     HEX8toASCII  ; Store the converted long into output
        MOVE.B  #$20,-(A1)   ; 4 empty spaces
        MOVE.B  #$20,-(A1)   ;
        MOVE.B  #$20,-(A1)   ;
        MOVE.B  #$20,-(A1)   ;
        MOVE.L  A5,D1        ; HEX addr in A5
        BSR     HEX8toASCII ; Store ASCII addr in output
        MOVE.B  #$24,-(A1)   ; Store '$' hex addr sign in output
        MOVE.B  #14,D0       ; Display memory addr and data
        TRAP    #15
        MOVE.B  #2,D0        ; Read input ans return to (A1)
        TRAP    #15
        CMPI.B  #$0,(A1)     ; Check if user input(null terminated) nothing
        BNE     LONG_MM2     ; If user input smtg, interpret input
        ADDA.L  #4,A5        ; Point to next LONGd data
        BRA     LONG_MM      ; Proceed to next LONG data

         *Read user input and terminate/ store input data
LONG_MM2 CMPI.B  #$2E,(A1)    ; Check if user enter '.' to terminate
         BEQ     END_MM       ; if yes, end MM
         MOVEA.L A1,A4        ; Else, point A4 to current LONG data
         BSR     ASCtoHEX_MM ; convert user input into HEX
         MOVE.L  D1,(A5)+     ; Replace it with HEX LONG user input
         BRA     LONG_MM      ; Proceed to next LONG data

ERR_MM  JSR    CMD_INVALID        ; Go to invalid command subroutine
END_MM  MOVEM.L (A7)+,D0-D1/A1/A5 ; Restore REGs
        RTS
```

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

```
***Memory Set***
*MS -  alters memory by setting data into the address specified
* Data can take the form of ASCII string or hexadecimal data.
* Input> long, only last long size data will be stored
MS:
        MOVEM.L D1/A5,-(A7)

        *Store address at A5
        SUBA.L  #1, A4      ; Point to first byte of user input address
        MOVEA.L A4,A5       ; A1 points to User Input for Writing
        MOVE.B  (A4)+,D1    ; First byte data from user input(A4 pointer)
        CMPI.B  #$24,D1     ; #$24 is '$', test if user input a valid address sign
        BNE     ERR_MS      ; INVALID Address for MS
        BSR     ASCIItoHEX  ; D1 has 1st address(ASCII)
        MOVEA.L D1, A5      ; 1st HEX addr in A5
        MOVE.B  (A4)+,D1    ; Store the next byte in D1 to check blank space
        CMPI.B  #$20,D1     ; Check if user input blank space before data
        BNE     ERR_MS      ; INVALID Command format for MS
```

```
        *Determine if it is empty, ASCII or HEX
CHK_MS  MOVE.B   (A4)+,D1     ; Store the next byte in D1 to check HEX sign
        CMPI.B   #$00,D1      ; Check if nothing is input
        BEQ      END_MS       ; If yes, Run MS as HEX
        CMPI.B   #$24,D1      ; #$24 is '$', test if user input hex value
        BEQ      HEX_MS       ; If yes, Run MS as HEX
        SUBA.L   #1,A4        ; Else, point to the beginning of string

ASC_MS  MOVE.B   (A4)+,(A5)+ ; Store one byte of data from input to memory
        CMPI.B   #0,(A4)      ; check if input ended
        BEQ      NULL_MS      ; If yes, end the memory setting
        BRA      ASC_MS       ; Else, keep looping
NULL_MS MOVE.B   #$00,(A5)     ; Null terminator
        BRA      END_MS

HEX_MS  BSR      ASCIItoHEX   ; D1 has input value (HEX)
        CMPI.L   #$FFFF,D1    ; D1 - $FFFF
        BHI      LONG_MS      ; If ans > 0, its a long
        CMPI.L   #$FF,D1      ; Else, try D1 - $FFFF
        BHI      WORD_MS      ; If ans > 0, its a word

*Store byte size data into (A5)
BYTE_MS MOVE.B   D1,(A5)     ; Store a byte value
        BRA      CHK_MS       ; End memory setting

*Store word size data into (A5)
WORD_MS ADDA.L   #2,A5        ; Skips a word size to save a word data
        MOVE.W   D1,-(A5)     ; Store a word value
        BRA      CHK_MS       ; End memory setting

*Store long size data into (A5)
LONG_MS ADDA.L   #4,A5        ; Skips a long size to save a long data
        MOVE.L   D1,-(A5)     ; Store a long value
        BRA      CHK_MS       ; End memory setting

ERR_MS  JSR    CMD_INVALID   ; Go to invalid command subroutine
END_MS  MOVEM.L (A7)+,D1/A5 ; Restore REGs
        RTS
```

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

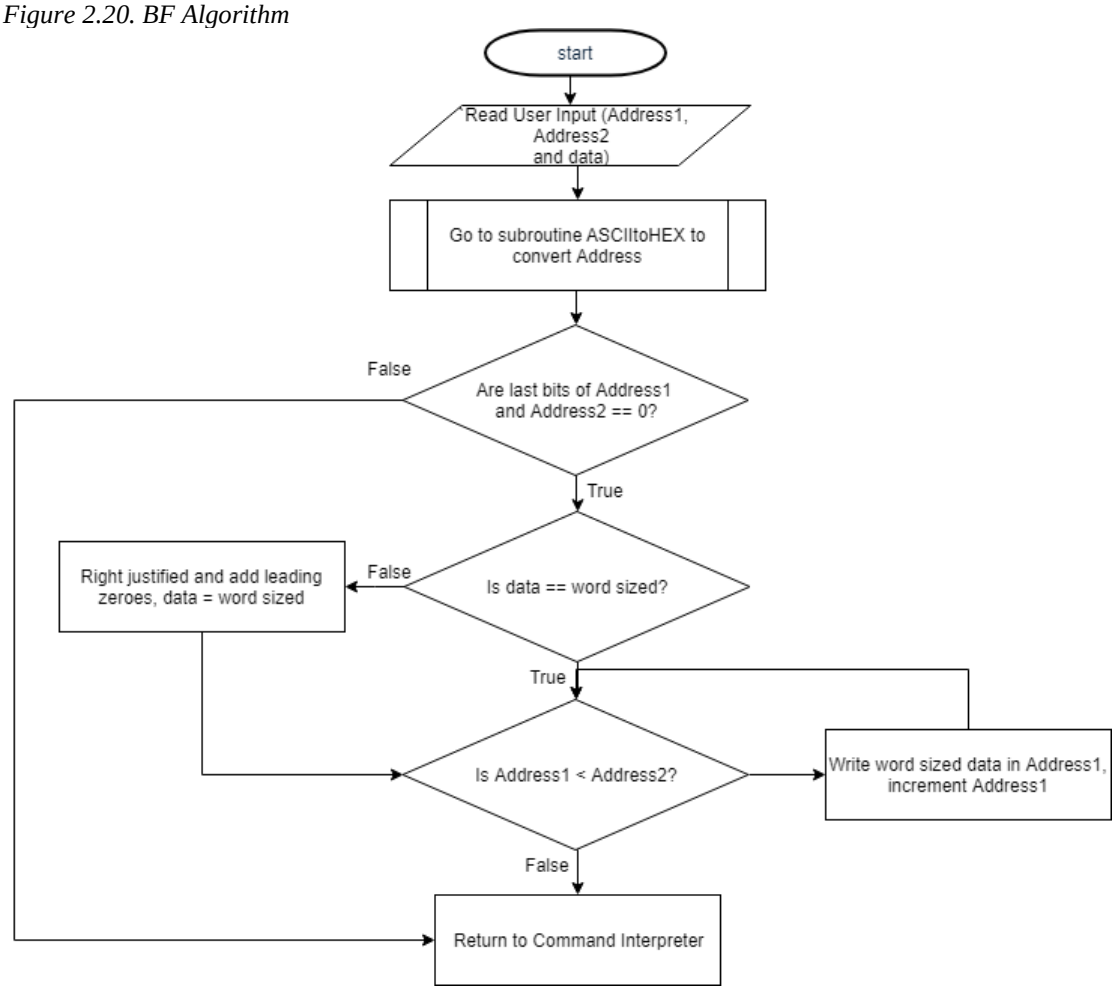

Figure 2.21. BF Flowchart

2.2.6.2-) BF Assembly Code

```
***Block Fill***
*BF - fills memory starting with the word boundary <address1> through <address2>
BF:
      MOVEM.L D1/D3/D4/D6/A1/A5-A6,-(A7)
      SUBA.L  #1, A4          ; Point to first byte of user input address
      MOVE.B  (A4)+,D1        ; First byte data from user input(A4 pointer)
      CMPI.B  #$24,D1         ; #$24 is '$', test if user input a valid address sign
      BNE     ERR_BF          ; INVALID Address for BF
      BSR     ASCIItoHEX      ; D1 has 1st address(ASCII)to be converted
      BTST    #0,D1           ; Check to see if last bit is 0(even addr)
      BNE     ERR_BF          ; Invalid command, addr is odd
      MOVEA.L D1,A5           ; 1st Address(HEX) stored in A5

      MOVE.B  (A4)+,D1        ; Blank space before next $address
      CMPI.B  #$20,D1         ; Check if it's a blank space
      BNE     ERR_BF
      CLR.L   D1              ; Clear D1
      MOVE.B  (A4)+,D1        ; second byte of input address
      CMPI.B  #$24,D1         ; #$24 is '$', test if user input a valid address sign
      BNE     ERR_BF          ; If not equal $, there is no 2nd address
      BSR     ASCIItoHEX      ; D1 has 2nd address(ASCII) to be converted
      BTST    #0,D1           ; Check to see if last bit is 0(even addr)
      BNE     ERR_BF          ; Invalid command, addr is odd
      MOVEA.L D1,A6           ; 2nd Address(HEX) stored in A6

      MOVE.L  #4,D4           ; counter for data pattern (4 characters)
      CLR.L   D6              ; D6 to store word-size (2 bytes) data pattern
      MOVE.B  (A4)+,D1        ; Check if user enter any data pattern
      CMPI.B  #$00,D1         ; Default 0 if any data pattern is not entered
      BEQ     BF3             ; Store leadign zeroes
      CMPI.B  #$20,D1         ; Check if user input blank space before data pattern
      BNE     ERR_BF          ; If not, invalid command is entered
      SUBQ.B  #1, D4          ; point to first data pattern input
BF2   CLR.L   D3
      MOVE.B  (A4)+,D3        ; One byte data from user input(A4 poier)
      TST.B   D3              ; Check if any data pattern is entered
      BEQ     BF3             ; If reaches blank space, add leading zeroes
      ASL.L   #4,D6           ; Shift left by 4, First char on left byte
      BSR     ALPHAorDIGIT    ; Convert to HEX character
      ADD.B   D3,D6           ; Converted character is now on right
      DBF     D4, BF2         ; Debrease D4, Keep looping and check next data pattern
BF3   ;MOVE.W  (A3),D4     ; TEST: if address2 not even, address error is raised

LOOP_BF CMPA.L  A5, A6        ; Check if A5 = A6
        BLE     END_BF        ; if equal, go to the end
        MOVE.W  D6, (A5)+     ; Store data pattern in the boudary
        BRA     LOOP_BF       ; Else, keep looping
ERR_BF  BSR     CMD_INVALID ; Display invalid command error msg
END_BF  MOVEM.L (A7)+,D1/D3/D4/D6/A1/A5-A6
        RTS
```

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

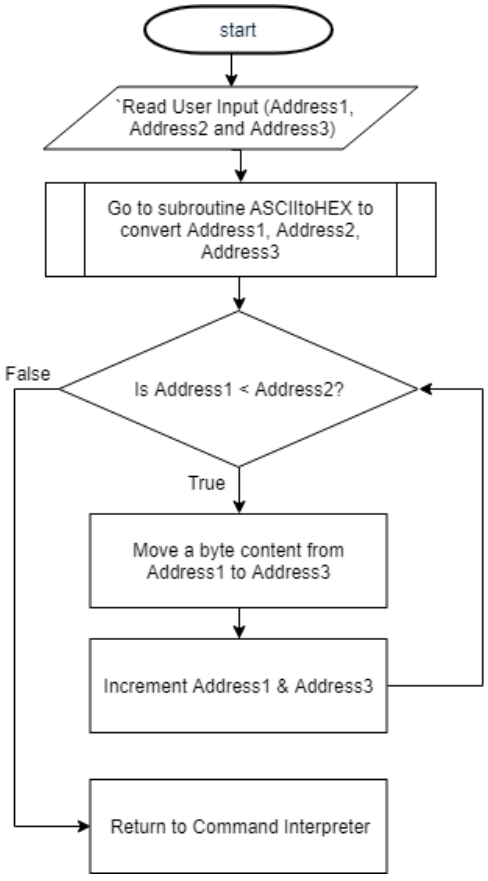

Figure 2.24. BMOV Flowchart

2.2.7.2-) Debugger Command #7 Assembly Code

```
***Block Move***
*BMOV -  Moves A Block Of Memory To Another Area
*BMOV <Addr1> <Addr2> <Addr3> eg: BMOV $ $ $<CR>
*Loop is not need for only 3 addr as the ' ' between addr have to be taken account
BMOV:
        MOVEM.L D1/A2/A5-A6,-(A7)
        SUBA.L  #1, A4    ; Point to first byte of user input address

        *Store 1st address in A5 and skip the blank space after it
        MOVE.B  (A4)+,D1     ; First byte data from user input(A4 pointer)
        CMPI.B  #$24,D1      ; #$24 is '$', test if user input a valid address sign
        BNE     ERR_BMOV     ; INVALID Address for BMOV
        BSR     ASCIItoHEX   ; D1 has 1st address(ASCII) to be converted
        MOVEA.L D1,A5        ; 1st HEX addr in A5
        MOVE.B  (A4)+,D1     ; Store the next byte in D1 to check blank space
        CMPI.B  #$20,D1      ; Check if user input blank space before Next address
        BNE     ERR_BMOV     ; INVALID Command format for BMOV

        *Store 2nd address in A6 and skip the blank space after it
        MOVE.B  (A4)+,D1     ; another byte data from user input(A4 pointer)
        CMPI.B  #$24,D1      ; #$24 is '$', test if user input a valid address sign
        BNE     ERR_BMOV     ; INVALID Address for BMOV
        BSR     ASCIItoHEX   ; D1 has 2nd address(ASCII) to be converted
        MOVE.L  D1,A6        ; 2nd HEX addr in A6
        MOVE.B  (A4)+,D1     ; Store the next byte in D1 to check blank space
        CMPI.B  #$20,D1      ; Check if user input blank space before Next address
        BNE     ERR_BMOV     ; INVALID Command format for BMOV

        *Store 3rd address in A2
        MOVE.B  (A4)+,D1     ; another byte data from user input(A4 pointer)
        CMPI.B  #$24,D1      ; #$24 is '$', test if user input a valid address sign
        BNE     ERR_BMOV     ; INVALID Address for BMOV
        BSR     ASCIItoHEX   ; D1 has 2nd address(ASCII) to be converted0
        MOVE.L  D1,A2        ; 2nd HEX addr in A2

LOOPBMOV CMPA.L  A5,A6       ; Compare if A5 is at A6 (A6-A5),
                             ; Compare first, incase user input same addresses
        BLE     END_BMOV     ; If not, keep looping
        MOVE.B  (A5)+,(A2)+  ; Moves A Block Of Memory, (A5) To (A2)
        BRA     LOOPBMOV
ERR_BMOV BSR     CMD_INVALID ; Display invalid command error msg
END_BMOV MOVEM.L (A7)+,D1/A2/A5-A6
        RTS
```

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

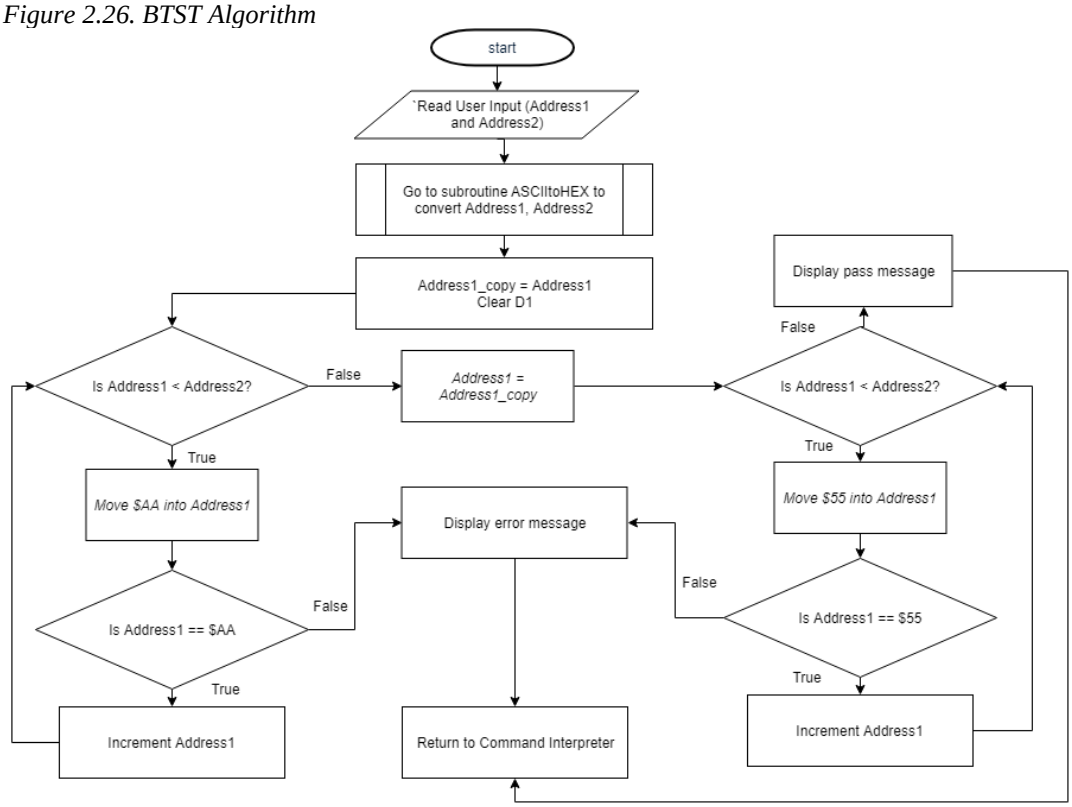

Figure 2.27. BTST Flowchart

2.2.8.2-) BTST Assembly Code

```
***Block Test***      *Have to move the mem location to even*
*BTST -  test memory from  <address1> to <address2>
* If completed w/o errors, display no error msg
* Else display error msg w/ address, the data stored & the data read of the failing memory.
 BTST:
       MOVEM.L D0-D1/A1-A2/A5-A6,-(A7)
       *Obtain and store addr1 and addr2 in A5,A2 and A6 respectively
       SUBA.L  #1, A4        ; Point to first byte of user input address
       MOVE.B  (A4)+,D1      ; First byte data from user input(A4 pointer)
       CMPI.B  #$24,D1       ; #$24 is '$', test if user input a valid address sign
       BNE     ERR_BTST      ; INVALID Address for BTST
       BSR     ASCIItoHEX    ; D1 has 1st address(ASCII)
       MOVEA.L D1,A5         ; 1st HEX addr in A5
       MOVEA.L D1,A2         ; 1st HEX addr in A2 (For testing $55)
       MOVE.B  (A4)+,D1      ; Store the next byte in D1 to check blank space
       CMPI.B  #$20,D1       ; Check if user input blank space before Next address
       BNE     ERR_BTST      ; INVALID Command format for BTST
       MOVE.B  (A4)+,D1      ; One byte data from user input(A4 pointer)
       CMPI.B  #$24,D1       ; #$24 is '$', test if user input a valid address sign
       BNE     ERR_BTST      ; INVALID Address for BTST
       BSR     ASCIItoHEX    ; D1 has 2nd address(ASCII)
       MOVEA.L D1,A6         ; 2nd HEX addr in A6

       *Fill all of the memory to be tested with $AA and run read operation (1010)
        CLR.L   D1           ; Clear D1 for storing byte data from memory
LOOP_BTST1:
        CMPA.L  A5,A6          ; Compare if A5 is at A6 (A6-A5),
                               ; Compare first, incase user input same addresses
        BLE     BTST2          ; If A5 is at A6, go to to next loop
        MOVE.B  #$AA,(A5)      ; Fill each byte of memory with $55
        MOVE.B  (A5)+,D1       ; Read the byte of memory
        CMPI.B  #$AA,D1        ; Check if the contents is be $AA.
        BEQ     LOOP_BTST1     ; If correct, continue loop for next byte

        *Display failing msg with address, data stored, data read of the failing memory.
        LEA     E_DTREAD,A1    ; Else,contents not $AA, problem in memory block
        BSR     HEXtoASCII     ; Convert data byte(HEX) read in mem and store it
        LEA     E_DTSTORE, A1 ; Store what system suppose to read
        MOVE.B  #$41, -(A1)    ; Store ASCII 'A'
        MOVE.B  #$41, -(A1)    ; Store ASCII 'A'
        LEA     E_ERR_ADDR,A1 ; Store address of memory problem found
        SUBA.L  #1, A5         ; Point to the error byte in memory location
        MOVE.L  A5, D1         ; D1 now has the problem memory addr
        BSR     HEX8toASCII    ; Convert error addr into ASCII
        LEA     ERR_ADDR ,A1   ; Display complete BTST error msg
        MOVE.B  #13, D0
        TRAP    #15
        BRA     END_BTST       ; Go to the end of BTST

        *Fill all of the memory to be tested with $55 and run read operation (0101)
BTST2   MOVEA.L A2, A5         ; Restore A5 initial value
        CLR.L   D1             ; Clear D1 for storing byte data from memory
LOOP_BTST2:
        CMPA.L  A5,A6          ; Compare if A5 is at A6 (A6-A5),
        BLE     NoErr_BTST       ; If A5 is at A6, go to to next loop
        MOVE.B  #$55,(A5)      ; Fill each byte of memory with $55
        MOVE.B  (A5)+,D1       ; Read the byte of memory
        CMPI.B  #$55,D1        ; Check if the contents is be $AA.
        BEQ     LOOP_BTST2     ; If correct, continue loop for next byte
```

```
        *Display failing msg with address, data stored, data read of the failing memory.
        LEA     E_DTREAD,A1   ; Else,contents not $AA, problem in memory block
        BSR     HEXtoASCII    ; Convert data byte(HEX) read in mem and store it
        LEA     E_DTSTORE, A1 ; Store what system suppose to read
        MOVE.B  #$35, -(A1)   ; Store ASCII '5'
        MOVE.B  #$35, -(A1)   ; Store ASCII '5'
        LEA     E_ERR_ADDR,A1 ; Store address of memory problem found
        SUBA.L  #1, A5        ; Point to the error byte in memory location
        MOVE.L  A5, D1        ; D1 now has the problem memory addr
        BSR     HEX8toASCII   ; Convert error addr into ASCII
        LEA     ERR_ADDR ,A1  ; Display complete BTST error msg
        MOVE.B  #13, D0
        TRAP    #15
        BRA     END_BTST      ; Go to the end of BTST
ERR_BTST   BSR    CMD_INVALID  ; Go to invalid command subroutine
           BRA    END_BTST     ; End the subroutine
NoErr_BTST LEA  BTST_NO_ERR,A1 ; Display no error found msg
           MOVE.B  #13, D0
           TRAP    #15
END_BTST   MOVEM.L (A7)+,D0-D1/A1-A2/A5-A6 ;Restore used Register
           RTS
```

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

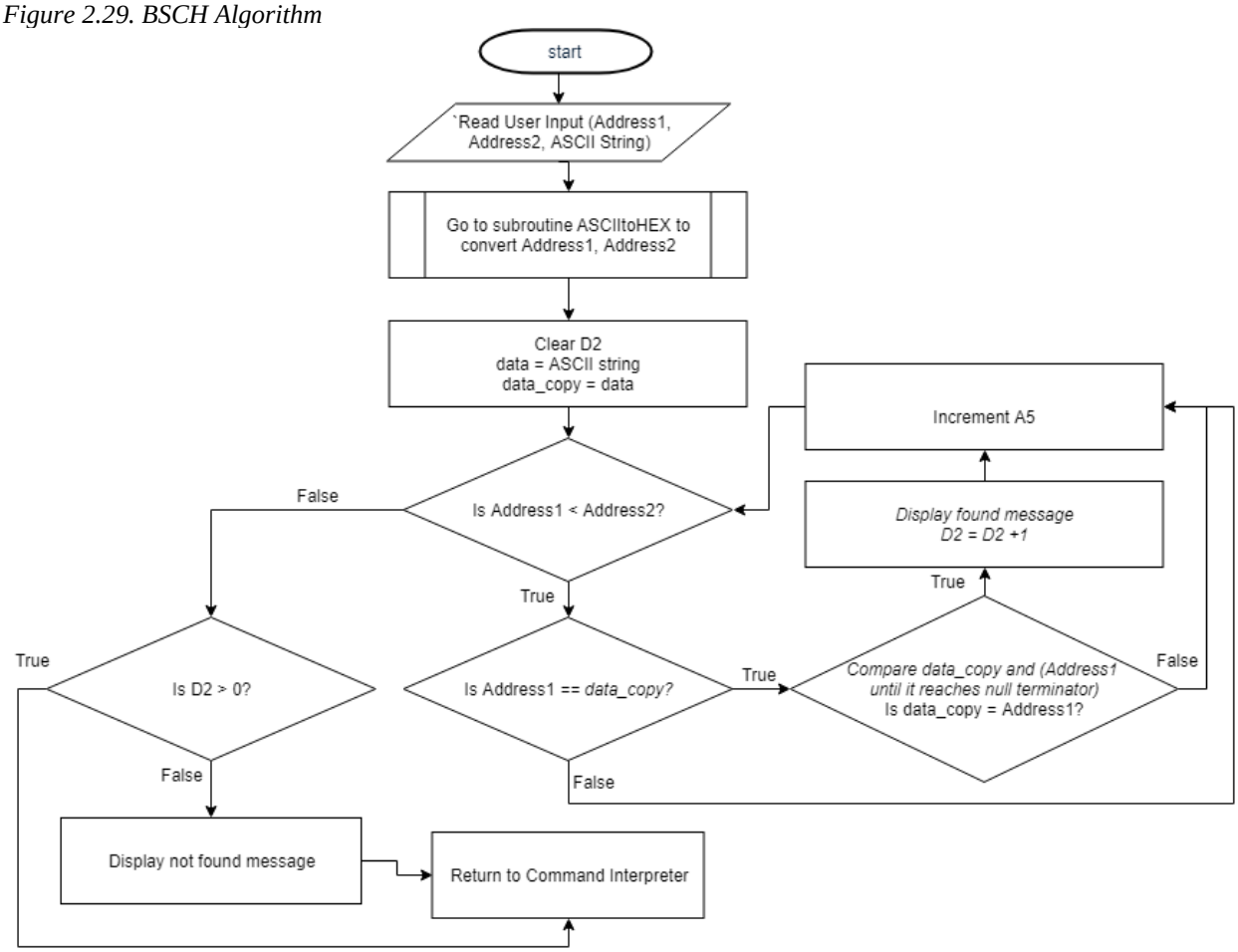

Figure 2.30. BSCH Flowchart

2.2.9.2-) BSCH Assembly Code

```
***Block Search***
*BSCH -  search a literal string from <address1> through <address2> both inclusive
*BSCH <Adr1> <Adr2> "literal string"
* If Found,  the data and address(s) must be displayed.
BSCH:
        MOVEM.L D1-D2/A1,-(A7)
        LEA     BSCH_FAIL_MSG ,A1 ; Initialize message

        SUBA.L  #1, A4          ; Point to first byte of user input address
        MOVE.B  (A4)+,D1        ; First byte data from user input(A4 pointer)
        CMPI.B  #$24,D1         ; #$24 is '$', test if user input a valid address sign
        BNE     ERR_BSCH        ; INVALID Address for BSCH
        BSR     ASCIItoHEX      ; D1 has 1st address(ASCII)
        MOVEA.L D1,A5           ; 1st HEX addr in A5
```

```
        MOVE.B   (A4)+,D1      ; Store the next byte in D1 to check blank space
        CMPI.B   #$20,D1       ; Check if user input blank space before Next address
        BNE      ERR_BSCH      ; INVALID Command format for BMOV
        MOVE.B   (A4)+,D1      ; One byte data from user input(A4 pointer)
        CMPI.B   #$24,D1       ; #$24 is '$', test if user input a valid address sign
        BNE      ERR_BSCH      ; INVALID Address for BSCH
        BSR      ASCIItoHEX    ; D1 has 2nd address(ASCII)
        MOVEA.L D1,A6          ; 2nd HEX addr in A6
        MOVE.B   (A4)+,D1      ; Store the next byte in D1 to check blank space
        CMPI.B   #$20,D1       ; Check if user input blank space before input string
        BNE      ERR_BSCH      ; INVALID Command format for BMOV
        MOVE.B   (A4),D1       ; Store the next byte in D1 to check blank space
        CMPI.B   #$00,D1       ; Check if user input empty string ''
        BEQ      ERR_BSCH      ; INVALID literal string
        CLR.L    D2            ; flag for checking if anything is found

LOOPBSCH CMPA.L  A5,A6         ; Compare if A5 is at A6 (A6-A5),
                               ; Compare first, incase user input same addresses
        BEQ      MSG_BSCH      ; Quit loop of A5 is at A6
        MOVEA.L A4, A2         ; A2 point to user input
        CMP.B    (A5)+,(A2)+   ; Compare one byte of character
        BNE      LOOPBSCH      ; Keep looping if doesnt equal
        MOVE.L   A5,A3         ; Save my A5 in A3, if found A5 will point to the String

BSCH2   CMPI.B   #0,(A2)       ; Check if string ended
        BEQ      BSCH_FOUND    ; If ended, same string found
        CMP.B    (A3)+,(A2)+   ; Compare one byte of character
        BNE      LOOPBSCH      ; IF not equal, check next literal string
        BRA      BSCH2         ; else, compare the next character

ERR_BSCH    BSR      CMD_INVALID  ; Display invalid command error msg
            BRA      END_BSCH

BSCH_FOUND SUBA.L   #1,A5      ; Point to the first character of string found
           MOVE.L   A5,D1
           LEA      END_BSCH_FOUND,A1    ; Point to the end end of found addr msg
           BSR      HEX8toASCII          ; Change to the address found
           LEA      BSCH_FOUND_MSG,A1
           MOVE.B   #13,D0                ; Print found message
           TRAP     #15
           ADDA.L   #1,A5                 ; Point to next byte to check
           ADD.L    #1, D2                ; increase string found count
           BRA      LOOPBSCH

MSG_BSCH   CMPI.L   #0, D2            ; If D2 isn't zero, at least 1 string is found
           BNE      END_BSCH          ; End the block search
           LEA      BSCH_FAIL_MSG ,A1 ; Load nothing found message
           MOVE.B   #13,D0            ; Display message
           TRAP     #15
END_BSCH   MOVEM.L (A7)+,D1-D2/A1
           RTS
```

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

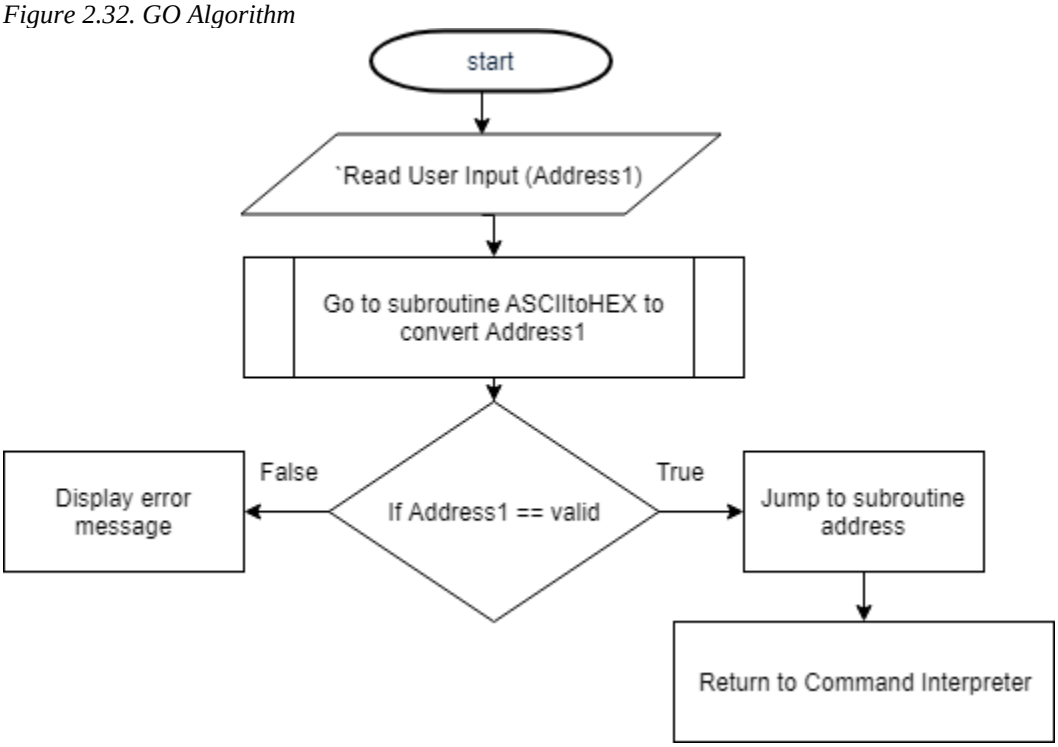

Figure 2.33. GO Flowchart

2.2.10.2-) GO Assembly Code

```
***Execute Program***
*GO -  start execution from a given address
GO      MOVEM.L D0-D7/A0-A7,-(A7) ; Save all registers
        SUBA.L  #1, A4        ; Point to first byte of user inout address
        MOVE.B  (A4)+,D1      ; Cope byte address from user input(A4 pointer)
        CMPI.B  #$24,D1       ; #$24 is '$', test if user input a valid address sign
        BNE     ERR_GO        ; INVALID Address for GO
        BSR     ASCIItoHEX    ; Convert mem addr to be run
        MOVEA.L D1,A6         ; Store user input addr in A6
        JSR     (A6)          ; Start execution from (A6)
        BRA     END_GO        ; End GO subroutine
ERR_GO  BSR     CMD_INVALID   ; Display invalid command error msg
END_GO  MOVEM.L (A7)+,D0-D7/A0-A7 ; Restore Resgisters
        RTS
```

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

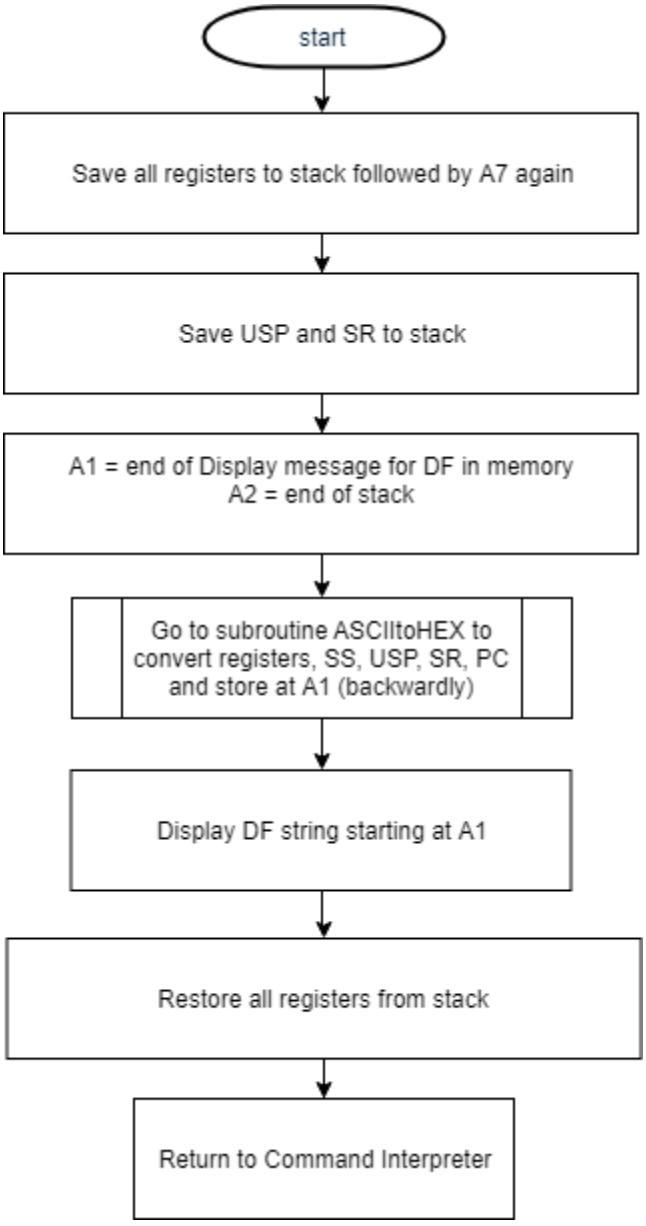

Figure 2.36. DF Flowchart

2.2.11.2-) DF Assembly Code

```
***Display Formatted Registers***
* DF - display current PC, SR, US, SS and D, A register
DF:
        MOVE.L  A7,DF_STACK ; So registers stored at STACK will start at $3000
        LEA     DF_STACK,A7 ;All other Df values store on DF_STACK
        MOVEM.L D0-D7/A0-A6,-(A7) ; Save current REGs on stack to Display
        LEA     DF_STACK,A7 ; To Store SS which is A7
        MOVE.L (A7),$2F78
        MOVEA.L  #$2F78, A7
        MOVE    USP,A6      ;for use with DF command
        MOVE.L  A6,-(A7)    ;store USP to STACK
        MOVE    SR,-(A7)    ;save Status register for use with DF
        LEA     STACK,A5
        MOVE.L  (A5),-(A7)
```

```
        MOVEM.L D0/D2-D3/A0-A2,-(A7) ; Save REGs
        LEA     DF_STACK, A2   ;A2 pointing at STACK
        ADDQ.L  #4, A2       ; Go to the end of STACK for A7 value
        MOVE.L  #16, D2      ; D2 as counter for 16 registers
        MOVE.L  #0, D3       ; D3 as counter for 4 lines
        LEA     DF_MSG_END,A1   ;A1 as the end of string
        SUBQ.L  #1, A1       ; Skip Empty space, $0
DF2     CMP.L   #4, D3       ; Calc(D3-4)
        BNE     DF3          ; If not 4 times, go to DF3
        SUBQ.L  #2, A1       ; Skip Empty space, $A,$D
        CLR.L   D3           ; Reset D3
DF3     MOVE.L  -(A2), D1    ; Save register value in D1 to be converted
        BSR     HEX8toASCII ; Branch to convert Hex->ASCII
        ADDQ.L  #1, D3       ; Increment D3, 4 REG/Line
        SUBQ.L  #4, A1       ; Skip blank and 'REG[#]='
        SUBQ.B  #1, D2       ; Increment D2, total = 16REGs
        BNE     DF2          ; If not 12 times yet, go back to DF2

        *SS*
        SUBQ.L  #1, A1       ; Skip '$A$D and D0='
        MOVE.L  -(A2), D1    ; Save register value in D1 to be converted
        BSR     HEX8toASCII ; Branch to convert Hex->ASCII

        *US*
        SUBQ.L  #4, A1       ; Skip blank and 'SS='
        MOVE.L  -(A2), D1    ; Save register value in D1 to be converted
        BSR     HEX8toASCII ; Branch to convert Hex->ASCII

        *SR*
        SUBQ.L  #8, A1       ; Skip blanks and 'US='
        MOVE.W  -(A2), D1    ; Save register value in D1 to be converted
        BSR     HEX4toASCII ; Branch to convert Hex->ASCII

        *PC*
        SUBQ.L  #4, A1       ; Skip blank and 'SR='
        MOVE.L  -(A2), D1    ; Save register value in D1 to be converted
        BSR     HEX8toASCII ; Branch to convert Hex->ASCII

        LEA     DF_MSG,A1    ; Go to beginning of string
        MOVE.B  #13,D0
        TRAP    #15          ; Display all Registers
        MOVEM.L (A7)+,D0/D2-D3/A0-A2 ;Restore REGs
        RTS
```

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

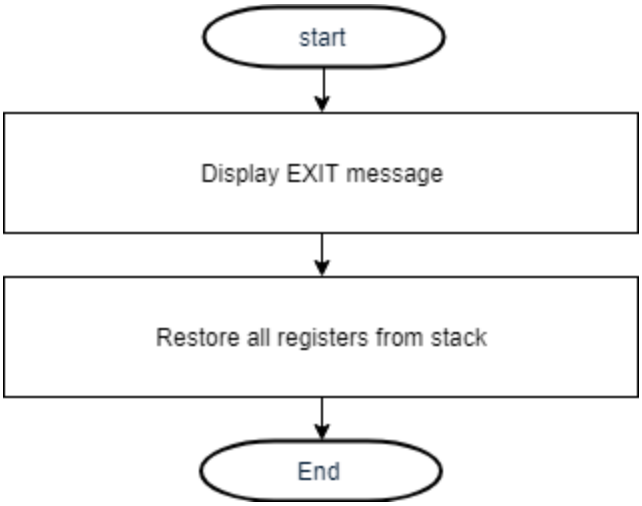

Figure 2.39. EXIT Flowchart

2.2.12.2-) EXIT Assembly Code

```
***Exit Monitor Program***
*EXIT - erminates/exits Monitor program
EXIT    LEA     EXIT_PRMPT ,A1     ; Display exiting msg
        MOVE.B  #13,D0
        TRAP    #15
        MOVE.L  #$2FC0,A7          ; Start of initial registers
       ;ADDA.L  INPUT_SIZE,A7      ; Skip stack spaces prepare for User Input
        MOVEM.L (A7)+,D0-D7/A0-A6 ; Restore Registers saved at beginning of program
        MOVEA.L STACK,A7
        BRA     END                ; Terminate Monitor Program
```

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

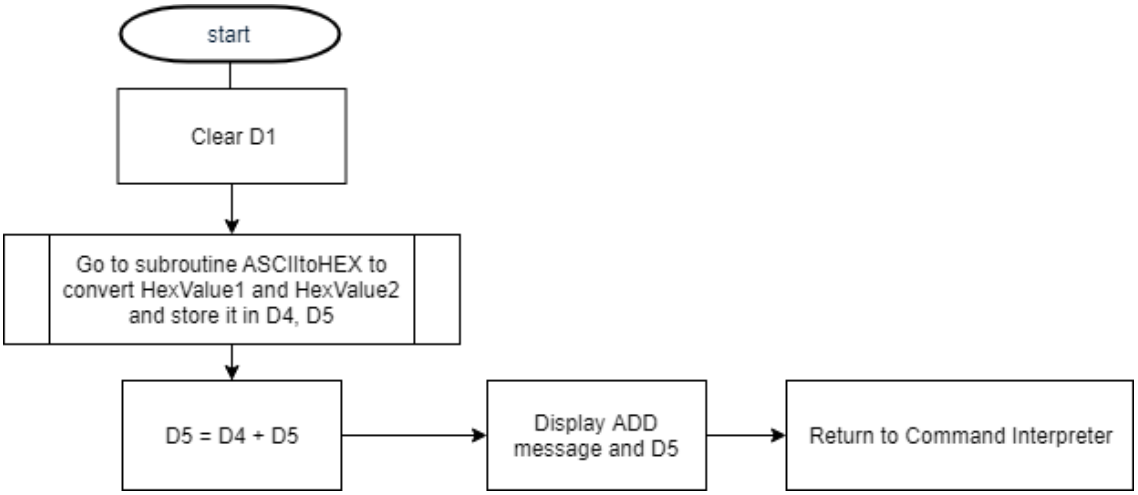

Figure 2.42. ADD Flowchart

2.2.13.2-) ADD Assembly Code

```
***Addition***
*ADD - Sums up two hex number and display
*ADD $A23 $123, each value is limited to max of $7FFFFFFF
ADD:
    MOVEM.L   ,-(A7)    ; Save Registers
    SUBA.L  #1, A4       ; Point to first byte of user input address
    CLR.L   D1           ; To be used for storing data

    *Store 1st hex value in D4
    MOVE.B  (A4)+,D1    ; First byte data from user input(A4 pointer)
    CMPI.B  #$24,D1     ; #$24 is '$', test if user input a valid hex number
    BNE     ERR_ADD     ; INVALID value for ADD
    BSR     ASCIItoHEX  ; D1 has 1st hex value
    MOVE.L  D1, D4      ; D4 now has 1st hex value
    MOVE.B  (A4)+,D1    ; Store the next byte in D1 to check blank space
    CMPI.B  #$20,D1     ; Check if user input blank space before data
    BNE     ERR_ADD     ; INVALID Command format for ADD

    *Store 2nd hex value in D5
    MOVE.B  (A4)+,D1    ; Take one byte data from user input(A4 pointer)
    CMPI.B  #$24,D1     ; #$24 is '$', test if user input a valid hex number
    BNE     ERR_ADD     ; INVALID value for ADD
    BSR     ASCIItoHEX  ; D1 has 1st hex value
    MOVE.L  D1, D5      ; D5 now has 2nd hex value
```

```
    *Addition
    ADD.L   D4, D5       ; Sum both value up in D5
    LEA     ADD_MSG,A1   ; Go to beginning of ADD output message
    MOVE.B  #14,D0
    TRAP    #15          ; Display sum message
    MOVE.L  #16, D2      ; To display output in base 16
    MOVE.L D5,D1
    MOVE.B  #15, D0
    TRAP    #15          ; Display SUM of the 2 hex number
    LEA     END_ADD_MSG,A1 ; Display CR + LF
    MOVE.B  #13,D0
    TRAP    #15
    BRA     END_ADD      ; End the ADD function

ERR_ADD     JSR    CMD_INVALID  ; Go to invalid command subroutine
END_ADD:
    MOVEM.L (A7)+,D0/D2/D4-D5/A1 ;Restore REGs
    RTS
```

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

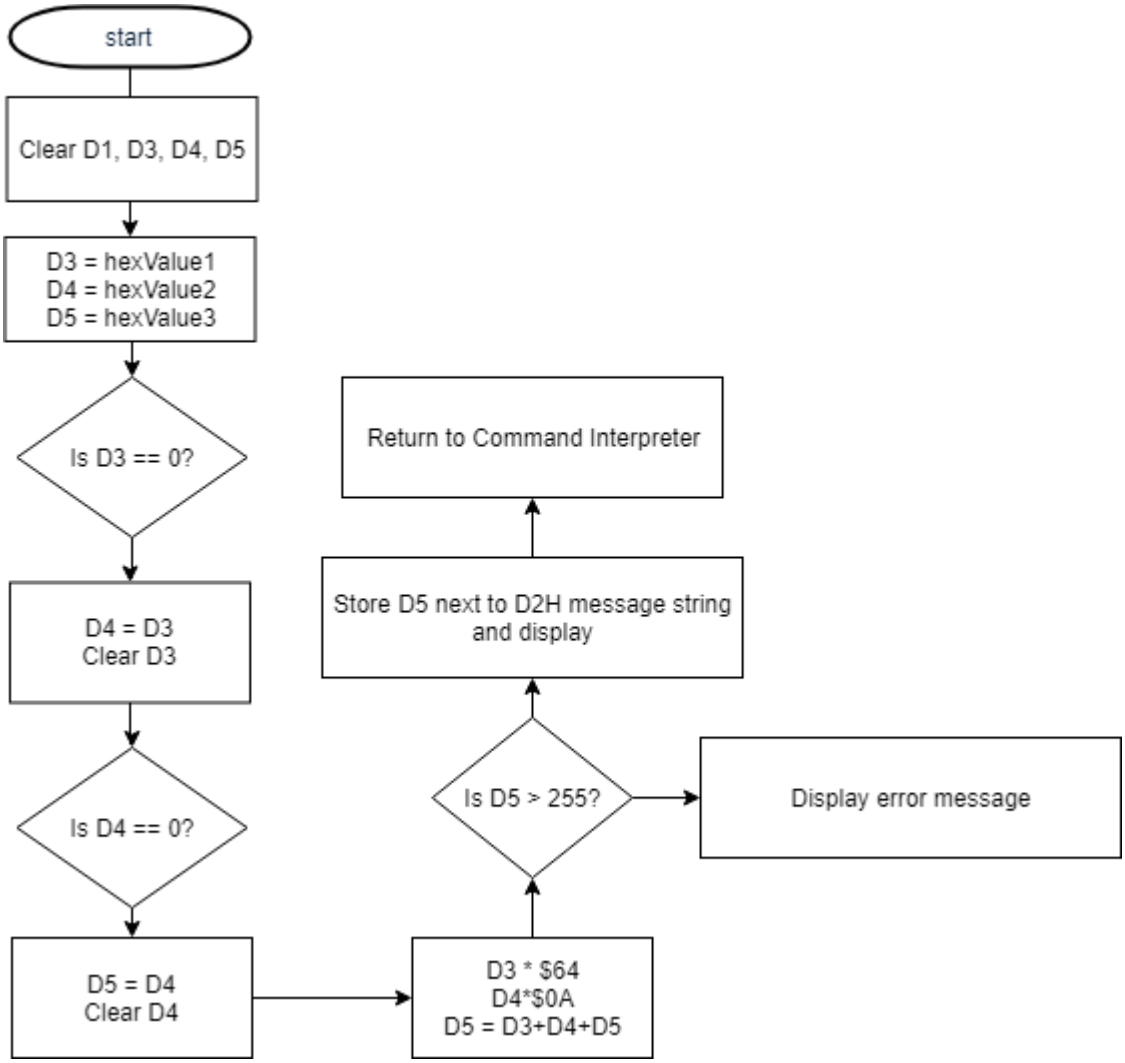

Figure 2.45. D2H Flowchart

2.2.12.2-) D2H Assembly Code

```
***3 Decimal to Hexadecimal***
*D2H - Convert up to 3 digits decimal number into hexadecimal number
*E.G: D2H 123 , number limit is 255
D2H:
    MOVEM.L  D2-D5/A1,-(A7) ; Save registers
    SUBA.L   #1, A4         ; Point to first byte of user input address
    CLR.L    D1             ; To be used for storing data
    CLR.L    D3             ; Empty D3 incase of 0 value
    CLR.L    D4             ; Empty D4 incase of less than 3 digits
    CLR.L    D5             ; Empty D5 incase of less than 3 digits

    *Check 1st number
    MOVE.B   (A4)+,D1    ; First byte data from user input(A4 pointer)
    CMPI.B   #$00,D1     ; #$00 Check if there's a number
    BEQ      ERR_D2H     ; Invalid data to be convert
    MOVE.L   D1,D3       ; 1st number in D3 (ASCII)
    SUBI.B   #$30,D3     ; 1st number in D3 (DEC)

    *Check 2nd number
    MOVE.B   (A4)+,D1    ; Second byte data from user input(A4 pointer)
    CMPI.B   #$00,D1     ; #$00  Check if there's a number
    BEQ      ONE_D2H     ; If no, go to one digits decimal function
    MOVE.L   D1,D4       ; 2nd number in D4 (ASCII)
    SUBI.B   #$30,D4     ; 2nd number in D4 (DEC)

    *Check 3rd number
    MOVE.B   (A4)+,D1    ; First byte data from user input(A4 pointer)
    CMPI.B   #$00,D1     ; #$00  Check if there's a number
    BEQ      TWO_D2H     ; If no, go to two digits decimal function
    MOVE.B   (A4)+,D2    ; Check if it's only 3 digits
    CMPI.B   #$00,D2     ;
    BNE      ERR_D2H     ; If not, return error
    MOVE.L   D1,D5       ; 3rd number in D5 (ASCII)
    SUBI.B   #$30,D5     ; 3rd number in D5 (DEC)
    BRA      D2H2
ONE_D2H MOVE.L   D3,D5   ; Place the only digit at 1's
        CLR.L D3         ; Empty D3 because only 1 digit
        BRA      D2H2    ; Go to Dec to Hex conversion

TWO_D2H MOVE.L   D4,D5   ; Place the 2nd digit at 1's
        MOVE.L   D3,D4   ; Place the 1st digit at 10's
        CLR.L D3         ; Empty D3 because only 1 digit
        BRA      D2H2    ; Go to Dec to Hex conversion
```

```
*Convert DEC to HEX and store after output string
D2H2:
    MULU    #$64, D3        ; 1st digit * 100
    MULU    #$0A, D4        ; 2nd digit * 10
    ADD.W   D3,D4           ; Sums up 3 digits
    ADD.W   D4,D5
    CMPI.L  #255,D5         ; Max value 255
    BGT     ERR_D2H
    LEA     END_D2H_MSG,A1  ; To save converted HEX after output
    CLR.L   D3
    MOVE.W  D5,D3           ; Save a copy of D5 to be reuse
    ANDI.W  #$0F00,D3       ; To obtain first hex value
    LSR.W   #8, D3          ; Make other byte 0, and shift it to lowest byte -> D3
    CLR.L   D4
    MOVE.W  D5,D4
    ANDI.W  #$00F0,D4       ; To obtain second hex value
    LSR.W   #4, D4          ; Make other byte 0, and shift it to lowest byte -> D3
    ANDI.B  #$0F, D5        ; To obtain last hex value
    MOVE.L  D5,D1           ; Store 1's HEX value to be displayed
    BSR     HEXtoASCII
    MOVE.L  D4,D1           ; Store 10's HEX value to be displayed
    BSR     HEXtoASCII
    MOVE.L  D3,D1           ; Store 100's HEX value to be displayed
    BSR     HEXtoASCII
PRINT   LEA     D2H_MSG,A1  ; Go to beginning of string
        MOVE.B  #13,D0
        TRAP    #15         ; Display converted HEX
        BRA     END_D2H

ERR_D2H     JSR    CMD_INVALID  ; Go to invalid command subroutine
END_D2H:
    MOVEM.L (A7)+,D2-D5/A1 ;Restore REGs
    RTS
```

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

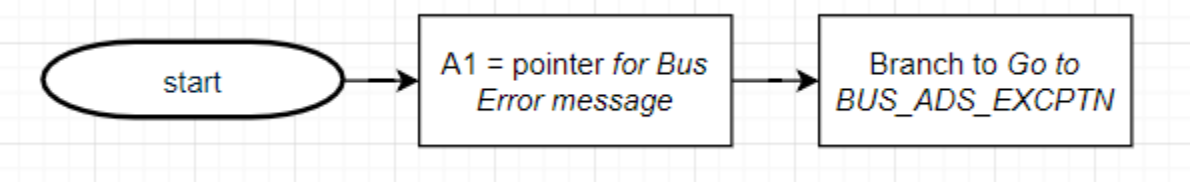

Figure 2.48. Bus Error Exception Flowchart

2.3.1.2-) Bus Error Exception Assembly Code

```
BUS_ERR:
        MOVEM.L A1/D0,-(A7)      ; Registers for Error Messages
        MOVEM.L D1/A2,-(A7)      ; Registers for SSW, BA, IR subroutine
        LEA     BERR_TXT, A1     ; Bus Error Msg
        BRA     BUS_ADS_EXCPTN
```

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

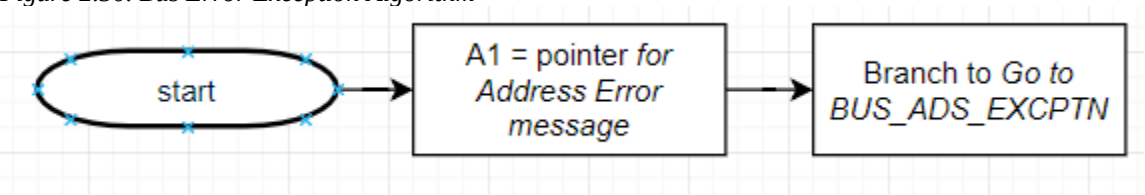

Figure 2.51. Bus Error Exception Flowchart

2.3.1.2-) Address Error Exception Assembly Code

```
ADS_ERR:
        MOVEM.L A1/D0,-(A7)     ; Registers for Error Messages
        MOVEM.L D1/A2,-(A7)     ; Registers for SSW, BA, IR subroutine
        LEA     ADDERR_TXT, A1  ; Address Error Msg
        BRA     BUS_ADS_EXCPTN
```

Figure 2.52. Bus Error Exception Assembly Code

2.3.3-) Illegal Instruction Exception

Load Illegal error 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

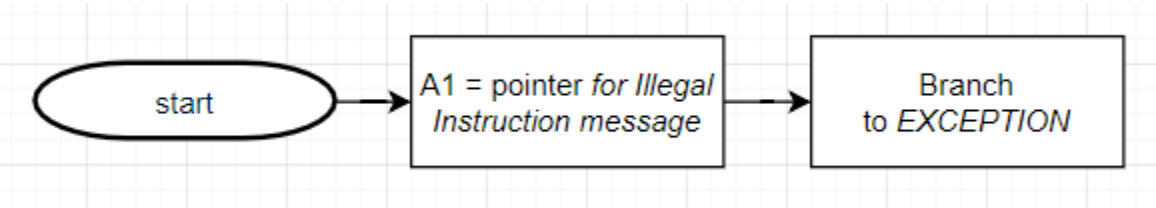

Figure 2.54. Illegal Instruction Exception Flowchart

2.3.3.2-) Illegal Instruction Exception Assembly Code

```
ILL_INST:
        MOVEM.L A1/D0,-(A7)
        LEA ILLINST_TXT, A1   ; Error msg
        BRA  EXCEPTION        ; Branch to display error msg
```

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

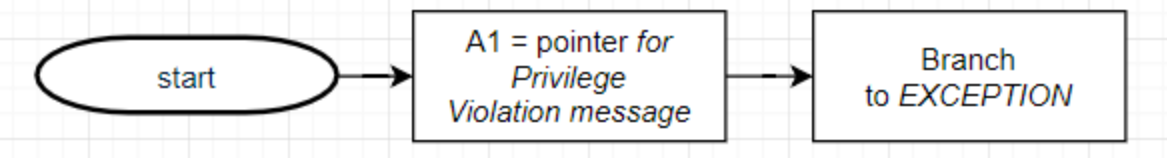

Figure 2.57. Privilege Violation Exception Flowchart

2.3.4.2-) Privilege Violation Exception Assembly Code

```
PRIV_VIOL:
        MOVEM.L A1/D0,-(A7)
        LEA PRIVVIO_TXT, A1
        BRA  EXCEPTION
```

Figure 2.58 Privilege Violation Exception Assembly Code

2.3.5-) Divide by Zero Exception

Load Divide by Zero exception 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

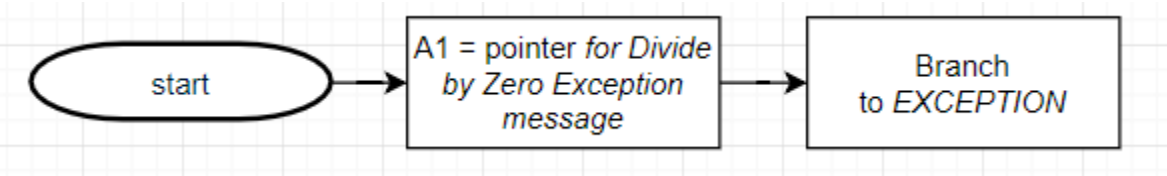

Figure 2.60. Privilege Violation Exception Flowchart

2.3.5.2-) Divide by Zero Exception Assembly Code

```
DIV_ZERO:
        MOVEM.L A1/D0,-(A7)
        LEA DIVZERO_TXT, A1
        BRA  EXCEPTION
```

Figure 2.61 Divide by Zero Exception Assembly Code

2.3.6-) Line A and Line F Emulators

Load Line A or F exception 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

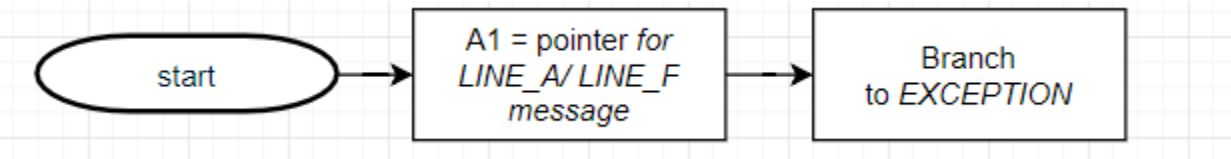

Figure 2.63 Line A and Line F Emulators Flowchart

2.3.6.2-) Line A and Line F Emulators Assembly Code

```
LINE_A:
        MOVEM.L A1/D0,-(A7)
        LEA LINEA_TXT, A1
        BRA  EXCEPTION

LINE_F:
        MOVEM.L A1/D0,-(A7)
        LEA LINEF_TXT, A1
        BRA  EXCEPTION
```

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

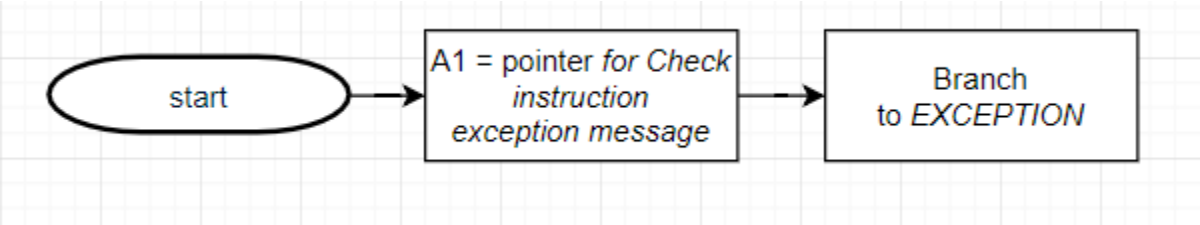

Figure 2.66 Check Instruction Exception Flowchart

2.3.7.2-) Check Instruction Exception Assembly Code

```
CHK_INST:
        MOVEM.L A1/D0,-(A7)
        LEA CHKINS_TXT, A1
        BRA  EXCEPTION
```

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

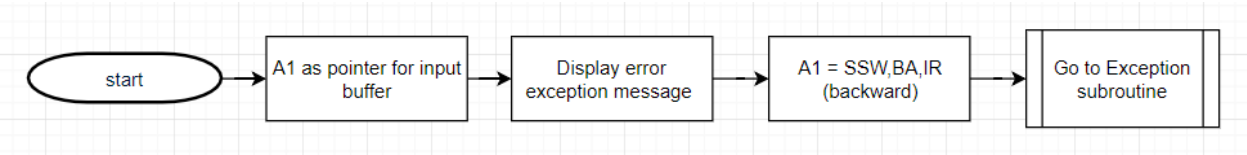

Figure 2.69 Bus and Address handler Exception Flowchart

2.3.8.2-) Bus and Address handler Exception Assembly Code

```
**--For bus error and address error routines--**
*Read the content of A7, display the content of the Supervisor Status Word,
*Bus Address and Instruction register all in a single line with spaces between them.
BUS_ADS_EXCPTN:
        MOVE.L  #13,D0      ; Display error exception message
        TRAP    #15
        MOVEA.L A7,A2       ; A2 as Stack Pointer
        ADDA.L  #24,A2      ; A2 now point to the end of IR
        LEA     END_USER_INPUT,A1 ; Prepare blank space to write SSW, BA, IR
        MOVE.B  #0,-(A1)    ; Null terminator (end of SSW, BA, IR)
        CLR.L   D1          ; To store word
        MOVE.W  -(A2),D1    ; SSW in D1
        BSR     HEX4toASCII ; ASCII IR stored in -4(A1)
        MOVE.B  #$20,-(A1)  ; Blank space
        MOVE.L  -(A2),D1    ; BA in D1
        BSR     HEX8toASCII ; ASCII BA stored in -8(A1)
        MOVE.B  #$20,-(A1)  ; Blank space
        CLR.L   D1          ; To store word
        MOVE.W  -(A2),D1    ; IR in D1
        BSR     HEX4toASCII ; ASCII SSW stored in -4(A1)
        MOVEM.L (A7)+,D1/A2 ; Restore registers
        BRA EXCEPTION       ; To display SSW, BA, IR
```

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

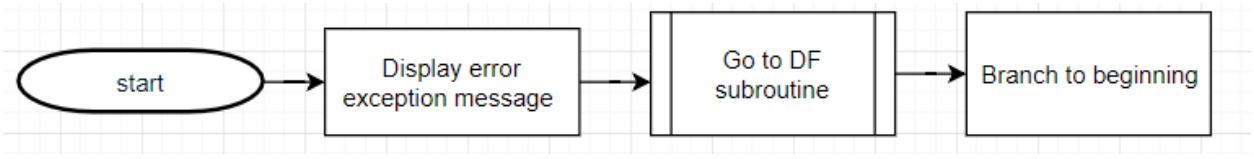

Figure 2.72 Error handler Exception Flowchart

2.3.8.2-) Error handler Exception Assembly Code

```
*Display error message and display the value of the registers
EXCEPTION:
        MOVE.B  #13,D0      ; Print our error exceptionn msg
        TRAP    #15
        BSR     DF          ; Call subroutine DF
        MOVEM.L (A7)+,D0/A1 ; Restore REGs
        LEA     STACK,A7    ; A7 point to Stack
        ADDA.L  #4,A0       ; User input store on stack right after A7

        BRA     BEGIN       ; Restart program
```

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

```
HELP_TABL   DC.B    'All available commands and usage descriptions:',$A,$A,$D
LINE1A      DC.B    'MDSP <address1> <address2> - eg: MDSP $4002 $4016<CR>',$A,$D
LINE1B      DC.B    'MDSP: Outputs Address And Memory Contents <address1> -> <address2>',$A,$D
LINE1C      DC.B    'Default: <address1> -> <address1 + 16bytes>',$A,$A,$D
LINE2A      DC.B    'SORTW <address1> <address2> <;A or ;D> - eg: SORTW $4000 $4008 ;A<CR>',$A,$D
LINE2B      DC.B    ';A - Ascending order, ;D - Decending',$A,$D
LINE2C      DC.B    'SORTW: Sorts Unsigned Words from memory <address1> to <address2>',$A,$D
LINE2D      DC.B    'Default: Descending Order',$A,$A,$D
LINE3A      DC.B    'MM <address><B/W/L> - eg: MM $4000;W<CR>',$A,$D
LINE3B      DC.B    '<B/W/L> specifies the number of bytes displayed for each address.',$A,$D
LINE3C      DC.B    'MM: Display memory, modify data or enter new data',$A,$D
LINE3D      DC.B    ';B - Byte size, ;W - Word size, ;L - Long size'
LINE3E      DC.B    'Default: Displays and enable edits for Byte size content',$A,$A,$D
LINE4A      DC.B    'MS <address> <data> - eg: MS $4000 HELLO WORLD<CR>',$A,$D
LINE4B      DC.B    'MS: Sets <data> (ASCII string/HEX value) into the <address> specified.',$A,$A,$D
LINE5A      DC.B    'BF <address1> <address2> <data> - eg: BF $4000 $4050 1234<CR>' ,$A,$D
LINE5B      DC.B    '<address1> and <address2> must be even.',$A,$D
LINE5C      DC.B    'BF: Fills memory starting with the word boundary <address1> through <address2>.',$A,$D
LINE5D      DC.B    'Only allows word-size (2 bytes) data pattern.',$A,$D
LINE5E      DC.B    'Both <address1> and <address2>  must be even addresses',$A,$D
LINE5F      DC.B    'If less than word size, the pattern is right justified and leading zeros added',$A,$D
LINE5G      DC.B    'eg: BF $4002 $4016 ABCD<CR>',$A,$A,$D
LINE6A      DC.B    'BMOV <address1> <address2> <address3> - eg: BMOV $4000 #4100 $5000<CR>',$A,$D
LINE6B      DC.B    'BMOV: Moves memory from <address1> till <address2> to <address3>(inclusive)',$A,$A,$D
LINE7A      DC.B    'BTST <address1> <address2> - eg: BTST $4000 $400A<CR>',$A,$D
LINE7B      DC.B    'BTST: test memory from <address1> to <address2> ',$A,$A,$D
LINE8A      DC.B    'BSCH <address1> <address2> <data> - eg: BSCH $4000 $4400 STRING<CR>',$A,$D
LINE8B      DC.B    'BSCH: search a literal string in a memory block ',$A,$D
LINE8C      DC.B    'starting at <address1> through <address2>, both inclusive.',$A,$A,$D
LINE9A      DC.B    'GO <address> - eg: GO $3010<CR>',$A,$D
LINE9B      DC.B    'GO:  start execution from <address>',$A,$A,$D
LINE10      DC.B    'DF: display current PC, SR, US, SS and D, A registers - eg: DF<CR>',$A,$A,$D
LINE11      DC.B    'EXIT: terminates/exits Monitor program - eg: EXIT<CR>',$A,$A,$D
LINE12A     DC.B    'ADD: Add two hex number - eg: ADD $7F $AF<CR>',$A,$D
LINE12B     DC.B    'ADD: Each value limit is $7FFFFFFF, only 0-9 and A-F',$A,$A,$D
LINE13A     DC.B    'D2H <data>, Convert up to 3 digits decimal number into hexadecimal number',$A,$D
LINE13B     DC.B    'eg: D2H 123<CR>',$A,$D
LINE13C     DC.B    'D2H: Decimal Value limit is: 255',$0
```

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

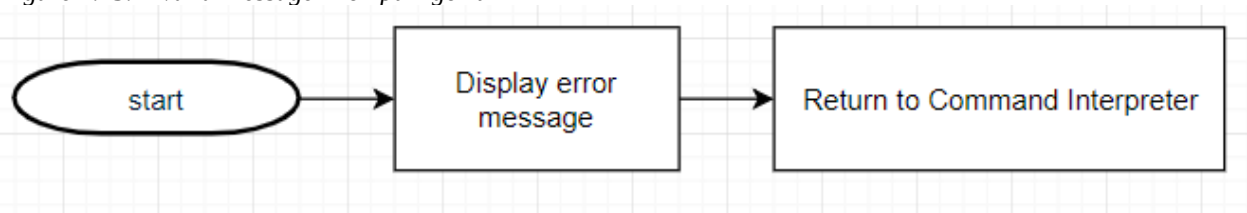

Figure 2.76 Error handler Exception Flowchart

2.4.2.1-) Invalid Message Prompt Assembly Code

```
***------------------------ Invalid Message Subroutine --------------------------***
*Prompt user when address is invalid
ADDR_INVALID:
       MOVEM.L D0/A1, -(A7)     ; Save REGs
       LEA     INVALID_ADR, A1 ; Display error message
       MOVE.B  #13,D0
       TRAP    #15
       MOVEM.L (A7)+,D0/A1      ; Restore REGs
       RTS

*Prompt user when command is invalid
CMD_INVALID:
       MOVEM.L D0/A1, -(A7)     ; Save REGs
       LEA     INVALID_COM, A1 ; Display error message
       MOVE.B  #13,D0
       TRAP    #15
       MOVEM.L (A7)+,D0/A1      ; Restore REGs
       RTS
```

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

```
    ORG    $2300
;TEST1 DC.B    '$2400',0 ; Store testcase in input space
;TEST1 DC.B    '$2400;B',0 ; Store testcase in input space
;TEST1 DC.B    '$2400;W',0 ; Store testcase in input space
TEST1 DC.B    '$2400;L',0 ; Store testcase in input space


STACK   EQU $2FFC       ; A7 will be stored at $3000 - $4

    ORG $2400
TEST1A   DC.B    '0000000100000002',0

    ORG $2000
START:                  ; first instruction of program
    MOVEA.L #$2301, A4
    BRA     MM
    ;MOVEA.L #$2601, A4
    ;BRA     MDSP
    END    START        ; last line of source
```

Figure 2.78. Testcases for MM - Assembly Code

```
    ORG     $2300
TEST1 DC.B     '$2400 $2408 ;A',0 ; Store testcase in input space
;TEST1 DC.B     '$2400 $2408 ;D',0 ; Store testcase in input space
;TEST1 DC.B     '$2400 $2408',0 ; Store testcase in input space
;TEST1 DC.B     '$2400 $2408;D',0 ; Store testcase in input space

STACK   EQU $2FFC       ; A7 will be stored at $3000 - $4

    ORG $2400
TEST1A   DC.B     '0501040203',0

    ORG $2000
START:                  ; first instruction of program
    MOVEA.L #$2301, A4
    BRA     SORTW
    ;MOVEA.L #$2601, A4
    ;BRA     MDSP
    END     START       ; last line of source
```

Figure 2.79. Testcases for SORTW - Assembly Code