ECE 441
MICROCOMPUTERS AND EMBEDDED COMPUTING SYSTEMS

Instructor : Dr. Jafar Saniie
Teaching Assistant : Mr. Guojun Yang

Final Project Report:
MONITOR PROJECT
04/27/2018

By : Prashanth Chandrasekaran

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

Prashanth Chandrasekaran

Table of Contents

Prashanth Chandrasekaran

Abstract

The report outlines the design and implementation of a Monitor Program that can be used to program a MC68000 microprocessor. A comprehensive description of all the commands and their terms of usage are provided together with the codes that govern their functioning. The obstacles faced during the program development are presented followed by a list of improvements that could be made upon the source code.

1-) Introduction

The problem presented is to be able to design a user friendly interface for the MC68000, similar to the TUTOR that we have experienced in the laboratory. The goal is to implement the given set of commands and design customized exception handling routines for all the exceptions available in the MC68000. The backbone behind the development of the Monitor Program is the Command Name Table and Command Address Table which work in tandem to access the respective command subroutines. Within these subroutines, an estimation on the number of digits present in the address is made after which the specific working of the command is performed. This project emulates the Monitor Program in its entirety using the EASY68K simulator. EASY68K is a Structured Assembly Language IDE which allows programmers to edit, assemble and run MC68000 programs on a PC.

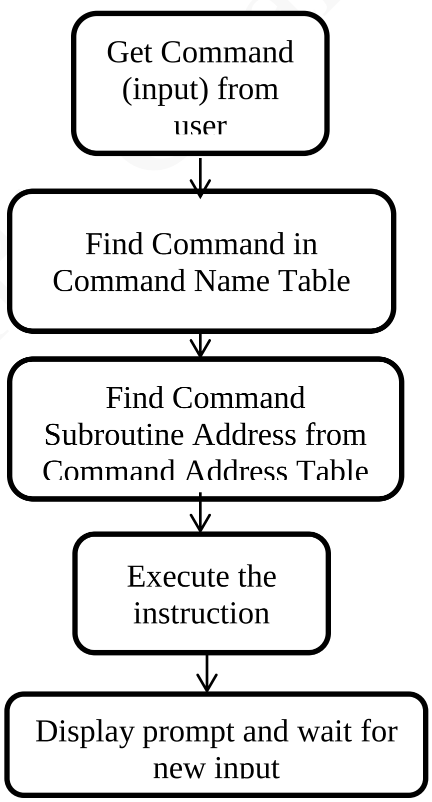


Fig.1 – Basic Flowchart of the working of the Monitor Program

The following points briefly highlight the step-by-step working of the Monitor Program:

- Display the prompt and wait for an input from the user.
- Once input has been obtained, search for the command in the COMMAND_NAME table. If found, proceed to COMMAND_ADDRESS table. If not found, display the invalid message and ask user to try again.
- Once in the COMMAND_ADDRESS table, find the subroutine address of the command entered by the user. Once found, jump to the subroutine and continue execution.
- Depending on the functionality of the command, Address and Data decoding maybe necessary. Perform all the necessary steps to complete full implementation of the command functionality.
- Once command has been executed, display the prompt once again and wait for an input from the user.

2-) Monitor Program

The Monitor Program is designed to perform all the 14 commands as listed above; the user will be able to modify registers, memory locations, sort data in the memory, search for data in the memory, fill blocks of memory as they see fit, run their own codes, etc. The program is also designed by taking into consideration the many exceptions that could occur during the users' utilization of the program and provides a means to handle said exceptions and input inaccuracies. Each one of the commands has their own set of specifics that must be followed to properly utilize their functionality. A comprehensive description of the specifics of each command can be found in the Users' Manual which provides a detailed insight into the syntax of each command and their precise usage examples.

2.1-) Command Interpreter

The Command Interpreter is the backbone of this project. It provides a means of decoding the user inputs, controls the handling of the commands and also safeguards against possible erroneous user inputs. The Command Interpreter comprises of two parts. They are as follows:

- COMMAND_NAME Table
- COMMAND_ADDRESS Table

The COMMAND_NAME table, as the name suggests, holds all the commands names. Any user input is first compared with the names stored in the COMMAND_NAME table to determine if the user inputted command is a valid one or not.

The COMMAND_ADDRESS table holds the addresses of the various subroutines that are involved in the implementation of the commands. Once the user inputted command is determined to be valid, the address of the appropriate subroutine is searched for in the COMMAND_ADDRESS table. The program then jumps to this address and continues the instruction execution. The tables are designed in such a way such that the names and labels are

Prashanth Chandrasekaran

ordered the same in both of them. This is done to facilitate a method to jump to the necessary command subroutine. The method used can be easily understood by looking at the flowchart; *n* is assumed to be the "get_ displacement" variable. When searching through the COMMAND_NAME table, this value is incremented accordingly and is used in the Address Register Indirect with Displacement addressing mode to branch to the command subroutine.

Note: *For commands that require addresses, the address format checking is performed within the command subroutine. The reasoning behind this is due to the fact that different commands have different number of letters in them and a common address format check would result in redundant complexities. Moreover, this helped in designing uniform subroutines and simplifies the understanding behind the working of the commands as they all follow a certain template. A user, if they wish, can study the source code and be able to modify any command subroutine with minimal effort.*

The COMMAND_NAME Table

```
CMDTABLE
    DC.B 'HELP  '
    DC.B 'MDSP  '
    DC.B 'SORTW '
    DC.B 'MM    '
    DC.B 'MS    '
    DC.B 'BF    '
    DC.B 'BMOV  '
    DC.B 'BTST  '
    DC.B 'BSCH  '
    DC.B 'GO    '
    DC.B 'DF    '
    DC.B 'RM    '
    DC.B 'DCON  '
    DC.B 'EXIT  '
```

The COMMAND_ADDRESS Table. The labels used points to the addresses of the subroutines

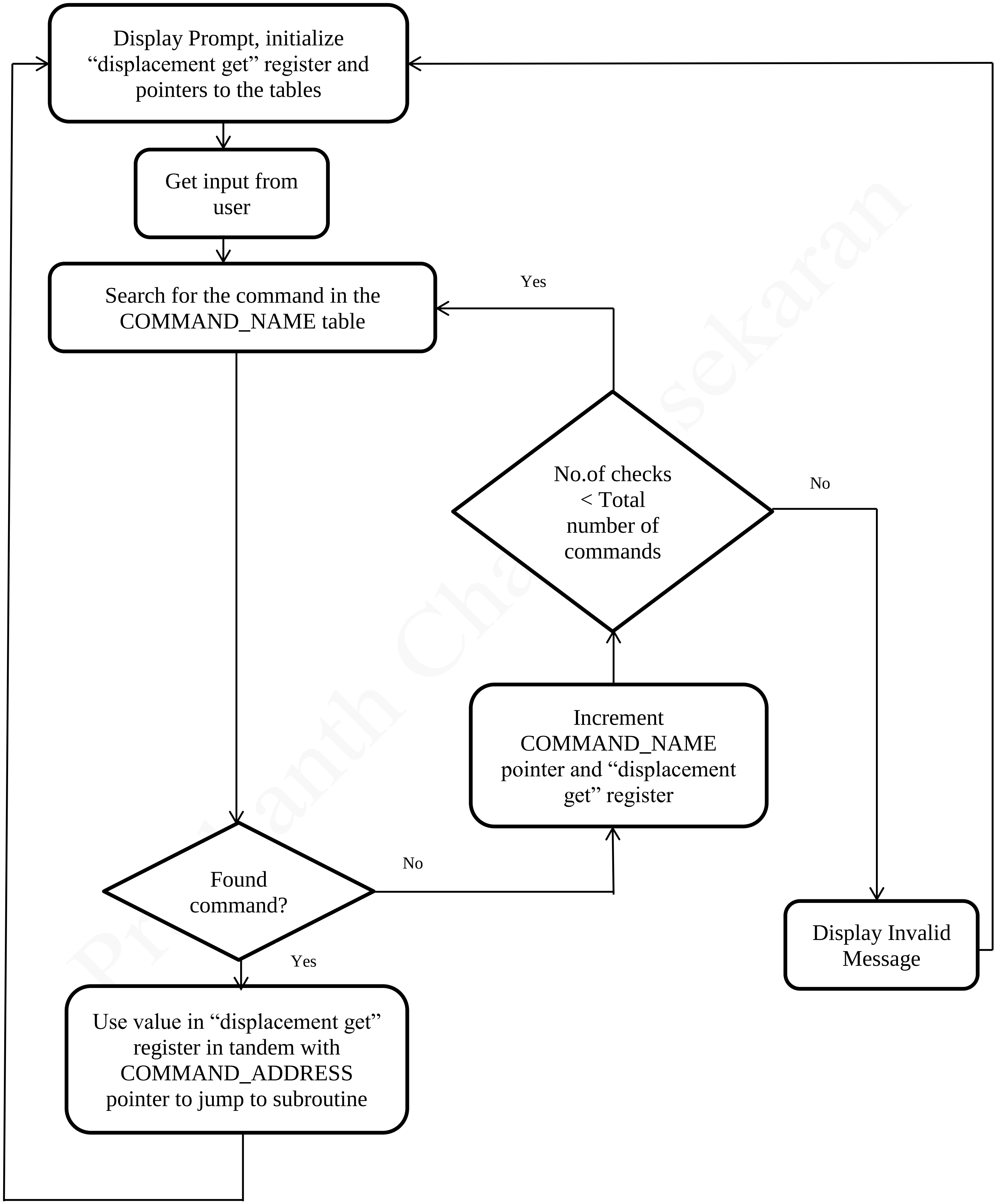
```
CMDADDRS
    DC.W HELP
    DC.W MDSP
    DC.W SORTW
    DC.W MM
    DC.W MS
    DC.W BF
    DC.W BMOV
    DC.W BTST
    DC.W BSCH
    DC.W GO
    DC.W DF
    DC.W RM
    DC.W DCON
    DC.W EXIT
```

2.1.1-) Algorithm and Flowchart

begin
Make m = 0, n=0
Get g = <user input>
Compare "command" in g with Command_Name letter-by-letter
If found
    proceed to Address acquisition
else
    do m=m+6 and n=n+1
    check if (no_of_checks (n) > no_of_commands)
    if true
        tell the user entered command is invalid
    else
        continue comparison process

Address Acquisition:
    Go to n^{th} label in COMMAND_ADDRESS table

Prashanth Chandrasekaran



Display Prompt, initialize "displacement get" register and pointers to the tables
Get input from user
Search for the command in the COMMAND_NAME table
Yes
No.of checks < Total number of commands
No
Increment COMMAND_NAME pointer and "displacement get" register
Found command?
No
Yes
Display Invalid Message
Use value in "displacement get" register in tandem with COMMAND_ADDRESS pointer to jump to subroutine

2.1.2-) 68000 Assembly Code

```
    LEA SPACE,A1            //
    JSR DISPCR              // Go to newline
    LEA PROMPT,A1           //
    JSR DISP                // Display Prompt
    LEA IP_BUFFER,A1        //
    MOVE.B #2,D0            // Set up input buffer for to receive TRAP input
    TRAP #15                //
    MOVE.L D1,D3            // Save a copy of command length in D3
    MOVEQ #0,D0             // Initialize for COMMAND_NAME search
    MOVEQ #0,D1             // Initialize for COMMAND_ADDRESS search
                            (displacement_get)
    MOVEQ #0,D2             // Initialize to count no_of_checks
    LEA A2ADDRS,A2          // Initialize Command Buffer
    LEA CMDTABLE,A4         // Initialize COMMAND_NAME Pointer
    LEA CMDADDRS,A5         // Initialize COMMAND_TABLE Pointer
RPT:    CMP.B #$20,(A1)     // Check if SPACE
    BEQ CHECK               // If yes, go to CHECK
      MOVE.B (A1)+,(A2)+    // If not, move letter into command buffer and
                              increment pointer
     BRA RPT                // Branch back to RPT
CHECK:    CMPI.B #$0C,D2    // Check if no_of_checks < no_of_commands
    BGT MAIN_INVALID        // If false, INVALID
    LEA A2ADDRS,A2          // Go to start of Command Buffer
    MOVE.W (A4,D0),D4       // Move first 2 letters of command as stored in
                              COMMAND_NAME table to D4
    CMP.W (A2),D4           // Compare this with the Command Buffer
    BEQ NEXT                // If match, go to subroutine pass
    ADDI #6,D0              // If not, go to next command in COMMAND_NAME
    ADDI #1,D1              // Increment displacement_get register
    ADDI #1,D2              // Increment no_of_checks
    BRA CHECK
NEXT:   LEA IP_BUFFER,A1    // Setup for command subroutine
    ADD.W D1,A5
    MOVE.W (A5,D1),A6       // Move address of command subroutine to A6
    JSR (A6)                // Jump to Subroutine
    JMP RERUN_UNTIL_EXIT    // Continue running until program
MAIN_INVALID:   LEA INVALID_MSG,A1
    JSR DISPCR
RERUN_UNTIL_EXIT:     MOVEQ #0,D2 //Reinitialize displacement_get
    MOVEQ #0,D4             // Clear D4
    BRA MAIN                // Branch back to program start
```

Prashanth Chandrasekaran

2.2-) Debugger Commands

This section describes in detail all the 14 commands that have been designed and implemented. The algorithm behind their design, the flowchart of their working and the assembly code used to implement it are all provided. An input buffer occupying 80 bytes is provided to store user inputs and is pointed to by **Address Register A1** *(All codes that have A1 present in them implies that the input buffer is being used. Any reference made to the input buffer implies that Address Register A1 is being discussed about)*. Before the description of each command is put forward, blocks of assembly code are provided and are briefly explained. These blocks make an appearance in almost all the debugger functions as subroutine calls and its inclusion prior to delving into the commands is done to prevent repetitive explanations of the same blocks of code.

<u>Block No.1</u>

```
CONV_2:
    MOVEM.L D0-D2/D4/A0-A6,-(SP)
    MOVE.L #16,D0
    MOVEQ #1,D1
    CLR.L D2
    CLR.L D3
    MOVE.B (A1)+,D2
    MOVE.B (A1)+,D3
    MULS.W D0,D2
    MULS.W D1,D3
    ADD.W D2,D3
    MOVEM.L (SP)+,D0-D2/D4/A0-A6
    RTS
```

This block is used to convert **two** consecutive bytes in the input buffer to its equivalent hexadecimal data.

Prashanth Chandrasekaran

Block No.2

```
CONV_3:
    MOVEM.L D0-D5/A0-A6,-(SP)
CONV_3_BEG:    MOVE.L #256,D0
    MOVEQ #16,D1
    MOVEQ #1,D2
    CLR.L D4
    CLR.L D5
    CLR.L D6
    MOVE.B (A1)+,D4
    MOVE.B (A1)+,D5
    MOVE.B (A1)+,D6
    MULS.W D0,D4
    MULS.W D1,D5
    MULS.W D2,D6
    ADD.W D4,D5
    ADD.W D5,D6
CONV_3_END:    MOVEM.L (SP)+,D0-D5/A0-A6
    RTS
```

This block is used to convert **three** consecutive bytes in the input buffer to its equivalent hexadecimal data. It is an extremely essential block as all 3 digit addresses are obtained using this block.

Block No.3

```
CONV_4:
    MOVEM.L D0-D6/A0-A6,-(SP)
CONV_4_BEG:    MOVE.L #4096,D0
    MOVE.L #256,D1
    MOVEQ #16,D2
    MOVEQ #1,D3
    CLR.L D4
    CLR.L D5
    CLR.L D6
    CLR.L D7
    MOVE.B (A1)+,D4
    MOVE.B (A1)+,D5
    MOVE.B (A1)+,D6
    MOVE.B (A1)+,D7
    MULS.W D0,D4
    MULS.W D1,D5
    MULS.W D2,D6
    MULS.W D3,D7
    ADD.W D4,D5
    ADD.W D5,D6
    ADD.W D6,D7
CONV_4_END:    MOVEM.L (SP)+,D0-D6/A0-A6
    RTS
```

This block is used to convert **four** consecutive bytes in the input buffer to its equivalent hexadecimal data. It is an extremely essential block as all 4 digit addresses are obtained using this block.

Prashanth Chandrasekaran

Block No.4

```
CONV_5:
    MOVEM.L D0-D6/A0-A6,-(SP)
CONV_5_BEG:    MOVE.L #65536,D0
    MOVE.L #4096,D1
    MOVE.L #256,D2
    MOVEQ #16,D3
    CLR.L D4
    CLR.L D5
    CLR.L D6
    CLR.L D7
    MOVE.B (A1)+,D4
    MOVE.B (A1)+,D5
    MOVE.B (A1)+,D6
    MOVE.B (A1)+,D7
    SWAP.W D0
    MULS.W D0,D4
    SWAP.W D4
    MULS.W D1,D5
    MULS.W D2,D6
    MULS.W D3,D7
    ADD.L D4,D5
    ADD.L D5,D6
    ADD.L D6,D7
    CLR.L D4
    MOVE.B (A1),D4
    MOVEQ #1,D0
    MULS.W D0,D4
    ADD.L D4,D7
CONV_5_END:    MOVEM.L (SP)+,D0-D6/A0-A6
    RTS
```

This block is used to convert **five** consecutive bytes in the input buffer to its equivalent hexadecimal data. It is an extremely essential block as all 5 digit addresses are obtained using this block.

Prashanth Chandrasekaran

Block No.5

```
GET_ADDR_ASCII:
    MOVEM.L D0/D3/D5-D7/A2-A6,-(SP)
    MOVEQ #$30,D0
    MOVEQ #$31,D1
    MOVEQ #0,D2
ASCII_CHECK_2:    CMPI.B #$39,(A1)
    BGT ASCII_RPT_31_2
ASCII_RPT_30_2:    SUB.B D0,(A1)+
    ADDQ #1,D2
    ADDQ #1,D4
    CMPI.B #$20,(A1)
    BNE ASCII_CHECK_2
    JMP ASCII_NEXT1_2
ASCII_RPT_31_2:    SUB.B D1,(A1)+
    ADDQ #1,D2
    ADDQ #1,D4
    SUBQ #1,A1
    CMP.B #$10,(A1)
    BEQ ASCII_NEXT10_2
    CMP.B #$11,(A1)
    BEQ ASCII_NEXT11_2
    CMP.B #$12,(A1)
    BEQ ASCII_NEXT12_2
    CMP.B #$13,(A1)
    BEQ ASCII_NEXT13_2
    CMP.B #$14,(A1)
    BEQ ASCII_NEXT14_2
    CMP.B #$15,(A1)
    BEQ ASCII_NEXT15_2
ASCII_NEXT10_2:    MOVE.B #10,(A1)+
    JMP ASCII_31_2
ASCII_NEXT11_2:    MOVE.B #11,(A1)+
    JMP ASCII_31_2
ASCII_NEXT12_2:    MOVE.B #12,(A1)+
    JMP ASCII_31_2
ASCII_NEXT13_2:    MOVE.B #13,(A1)+
    JMP ASCII_31_2
ASCII_NEXT14_2:    MOVE.B #14,(A1)+
    JMP ASCII_31_2
ASCII_NEXT15_2:    MOVE.B #15,(A1)+
    JMP ASCII_31_2
ASCII_31_2:  CMPI.B #$20,(A1)
    BNE ASCII_CHECK_2
ASCII_NEXT1_2: MOVEM.L (SP)+,D0/D3/D5-D7/A2-A6
    RTS
```

This block is used to convert the ASCII values of the address present in the input buffer to their corresponding hexadecimal counterparts. This block is used in combination with block 2, 3 or 4 to get the required address as has been inputted by the user.

Block No.6

```
GET_DATA:
    MOVEM.L D0/D1/D3-D7/A2-A6,-(SP)
    MOVEQ #$30,D0
    MOVEQ #$31,D1
    MOVEQ #0,D2
DATA_CHECK_2:    CMPI.B #$39,(A1)
    BGT DATA_RPT_31_2
DATA_RPT_30_2:    SUB.B D0,(A1)+
    ADDI #1,D2
    JMP DATA_RECHECK
DATA_RPT_31_2:    SUB.B D1,(A1)+
    ADDI #1,D2
    SUBQ #1,A1
    CMP.B #$10,(A1)
    BEQ DATA_NEXT10_2
    CMP.B #$11,(A1)
    BEQ DATA_NEXT11_2
    CMP.B #$12,(A1)
    BEQ DATA_NEXT12_2
    CMP.B #$13,(A1)
    BEQ DATA_NEXT13_2
    CMP.B #$14,(A1)
    BEQ DATA_NEXT14_2
    CMP.B #$15,(A1)
    BEQ DATA_NEXT15_2
DATA_NEXT10_2:    MOVE.B #10,(A1)+
    JMP DATA_RECHECK
DATA_NEXT11_2:    MOVE.B #11,(A1)+
    JMP DATA_RECHECK
DATA_NEXT12_2:    MOVE.B #12,(A1)+
    JMP DATA_RECHECK
DATA_NEXT13_2:    MOVE.B #13,(A1)+
    JMP DATA_RECHECK
DATA_NEXT14_2:    MOVE.B #14,(A1)+
    JMP DATA_RECHECK
DATA_NEXT15_2:    MOVE.B #15,(A1)+
    JMP DATA_RECHECK
DATA_RECHECK:  CMPI.B #$00,(A1)
    BNE DATA_CHECK_2
    MOVEM.L (SP)+,D0/D1/D3-D7/A2-A6
    RTS
```

Similar to the previous block, this block is used to convert the ASCII values of the **data** present in the input buffer to their corresponding hexadecimal counterparts. This block is used in combination with block 2, 3 or 4 to get the required data as has been inputted by the user.

Prashanth Chandrasekaran

Block No.7,8 and 9

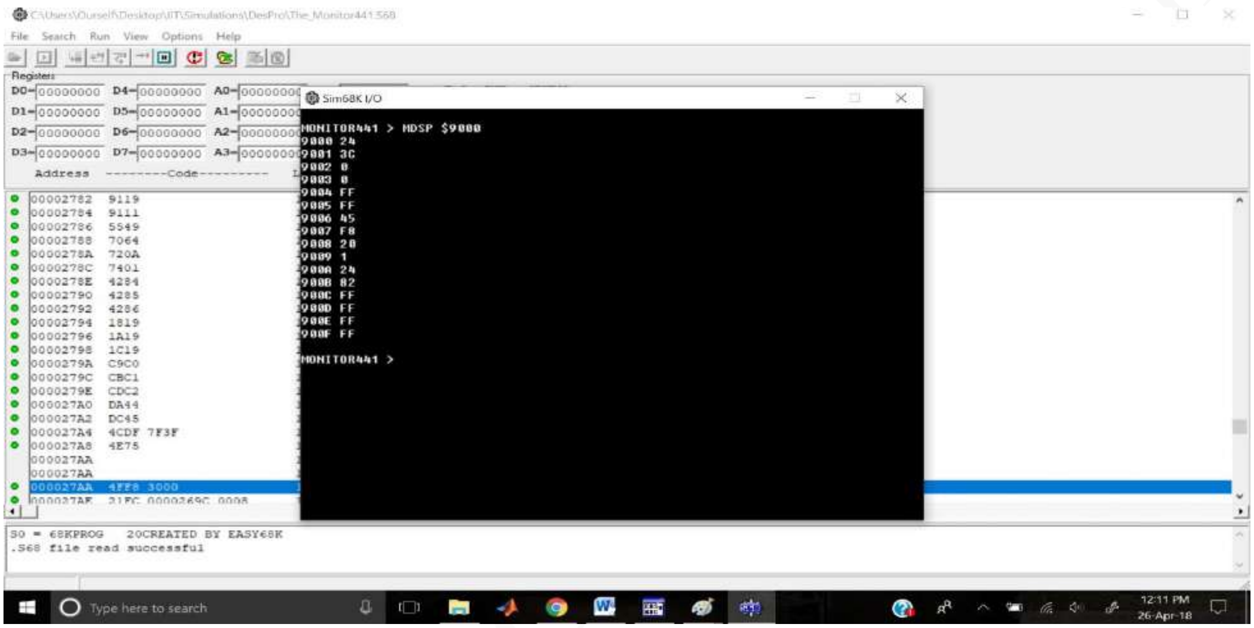
```
DISPCR:
    MOVEM.L D0,-(SP)
    MOVE.B #13,D0
    TRAP #15
    MOVEM.L (SP)+,D0
    RTS

DISP:
    MOVEM.L D0,-(SP)
    MOVE.B #13,D0
    TRAP #15
    MOVEM.L (SP)+,D0
    RTS

DISPDA:
    MOVEM.L D0/D2,-(SP)
    MOVE.B #16,D2
    MOVE.B #15,D0
    TRAP #15
    MOVEM.L (SP)+,D0/D2
    RTS
```

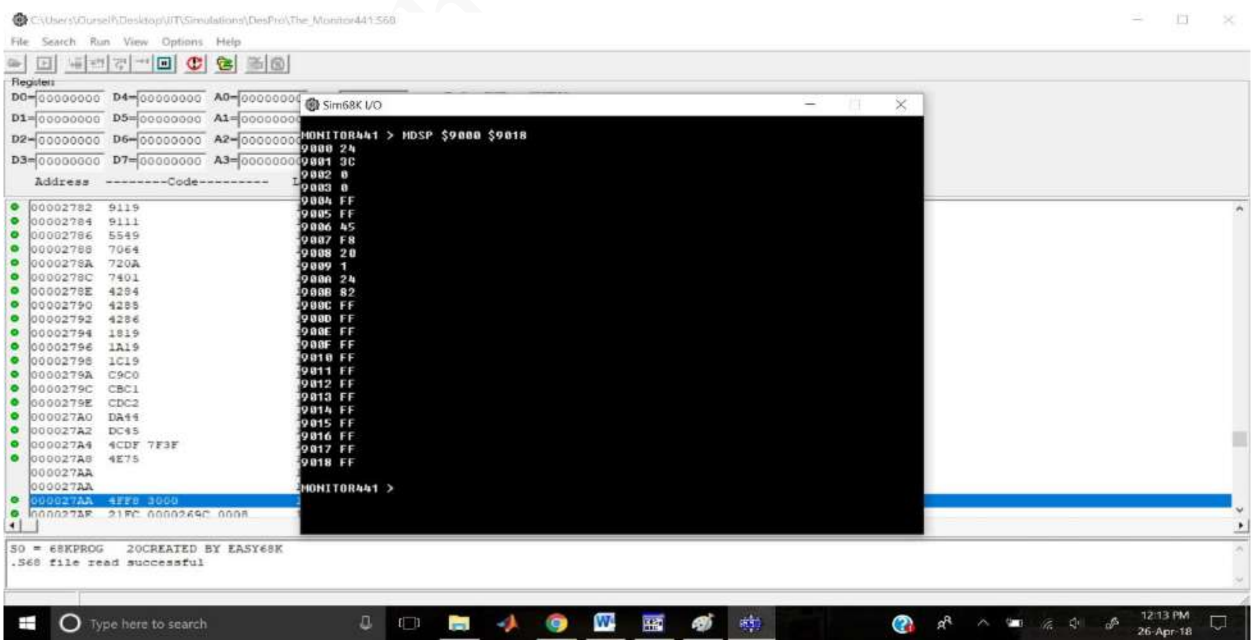
These 3 subroutines make use of TRAP #15 functions in the EASY68K simulator to either print the NULL terminated string in Address Register **A1** or Long word hexadecimal data in Data Register **D1**.

Prashanth Chandrasekaran

2.2.1-) Debugger Command # 1 – MDSP (Memory Display)

The MDSP, short for **M**emory **D**i**SP**lay, is used to display the byte size contents of the memory. The address of the memory to be viewed shall be provided by the user. If the user enters a single address, the command displays the memory starting from the address provided up until 16 bytes forward. If however the user wishes to view a larger or smaller range of memory, he/she can enter the specific addresses that fulfill their range requirements and the command shall display the data stored in this memory range.



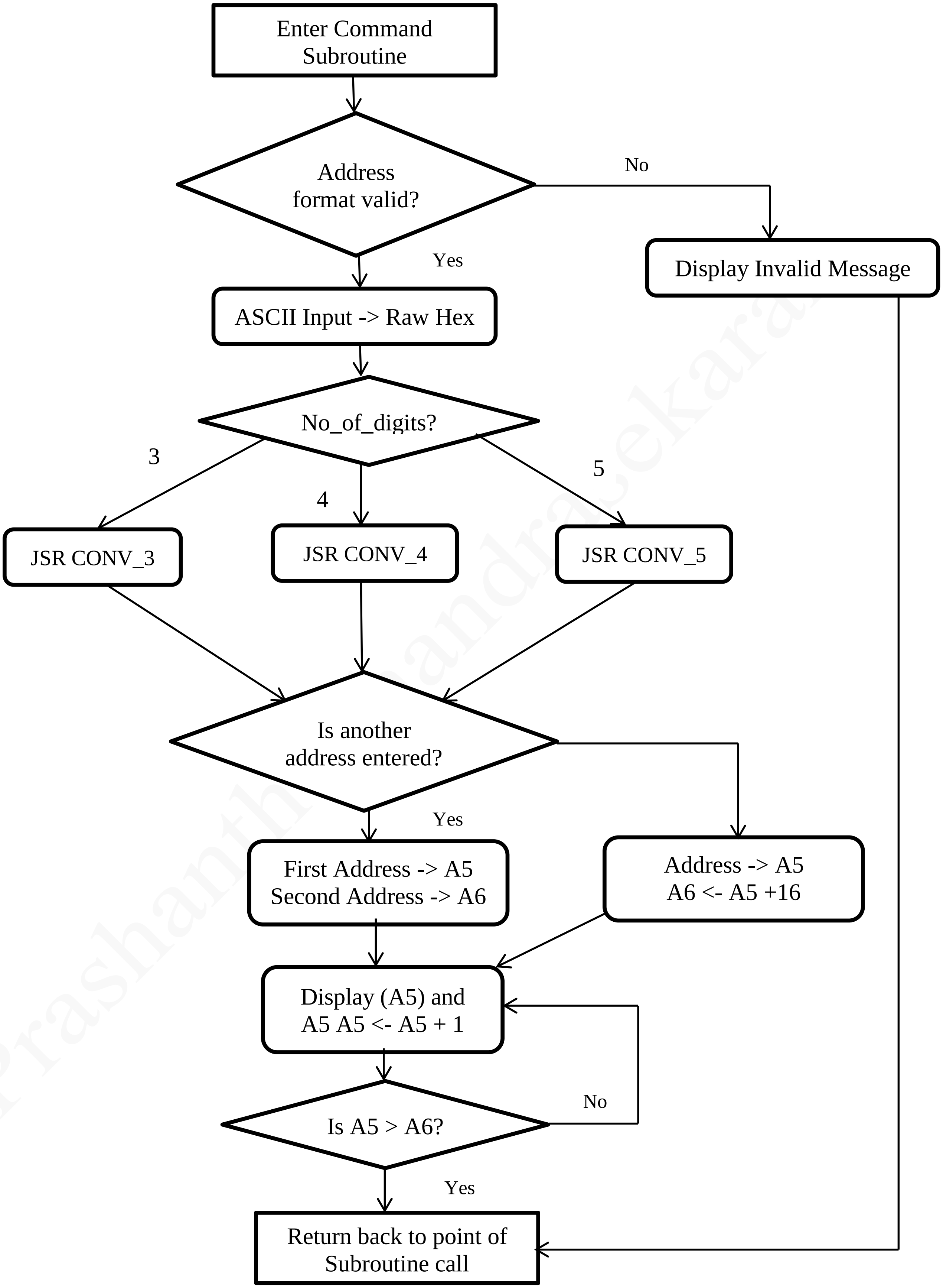


Examples usage of Memory Display command

2.2.1.1-) Algorithm and Flowchart

```
begin
        Check address format.
                If correct
                        do command
                else
                        display invalid message and wait for new input
        command:
                Convert ASCII input value of address to raw hex values
                Determine number of digits in the address
                Use appropriate conversion block to obtain the address. Move to A5
                Check no.of addresses
                        if 1
                                A5 -> A6
                                A6 -> A6 +16
                                goto display
                        else
                            Use appropriate conversion block to obtain the address. Move to A6
                                goto display
        display:
                Use apt registers for TRAP #15 function usage.
                Display contents in (A5)
                A5 -> A5+1
                Loop as long as A5 ≤ A6
end
```

Prashanth Chandrasekaran


Enter Command Subroutine
Address format valid?
No
Yes
Display Invalid Message
ASCII Input -> Raw Hex
No_of_digits?
3
4
5
JSR CONV_3
JSR CONV_4
JSR CONV_5
Is another address entered?
Yes
First Address -> A5
Second Address -> A6
Address -> A5
A6 <- A5 +16
Display (A5) and
A5 A5 <- A5 + 1
Is A5 > A6?
No
Yes
Return back to point of Subroutine call

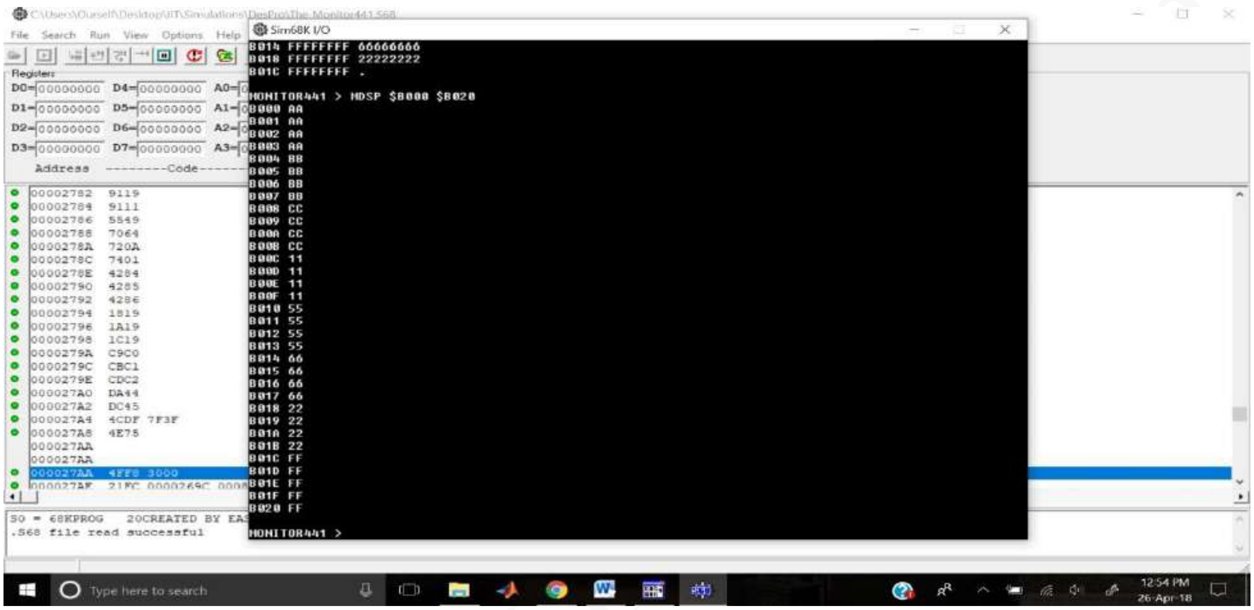
2.2.1.2-) Assembly Code

```
MDSP:
    MOVEM.L D0-D7/A0-A6,-(SP)    // Save registers on the stack
    MOVEQ #0,D4
    ADDQ #5,A1                   // Skip the 'MDSP '
    CMPI.B #$24,(A1)+            // Check if address starts with $
    BEQ MDSP_NEXT                // If yes, proceed
    JMP MDSP_INVALID             // If not, INVALID
MDSP_NEXT: JSR GET_ADDR_ASCII    // Convert ASCII input to hex
    SUB.L D2,A1                  // Go to address start
    CMPI.B #$03,D4               //
    BEQ MDSP_3                   //
    CMPI.B #$04,D4               // Branch to subroutine based on number of
    BEQ MDSP_4                   // digits in the address
    CMPI.B #$05,D4               //
    BEQ MDSP_5                   //
MDSP_3:    CMPI.B #$0C,D3        // 3 digit address. Check if 2 addresses or 1
    BGT MDSP_3RANGE              // if 2, get next addr
    JSR CONV_3                   // Convert Raw hex to actual hex
    MOVE.W D6,A5                 // A5 <- Address 1
    MOVE.L A5,A6
    ADD.L #$0F,A6                // A6 <- A5 + 16
    JMP MDSP_DIS                 // Jump to display loop
MDSP_3RANGE:    ADDQ #4,A1
    CMPI.B #$24,(A1)+            // Check if second address starts with $
    BEQ GET_NEXT_ADDR3           // If yes, proceed
    JMP MDSP_INVALID             // If not, INVALID
GET_NEXT_ADDR3:    JSR GET_ADDR_ASCII    // Convert ASCII input to hex
    SUB.L D4,A1
    SUBQ #2,A1
    JSR CONV_3                   // Convert Raw hex to actual hex
    MOVE.W D6,A5                 // A5 <- Address 1
    MOVE.L D6,D5
    ADD.L D2,A1
    ADDQ #2,A1
    JSR CONV_3                   // Convert Raw hex to actual hex
    MOVE.W D6,A6                 // A6 <- Address 2
    JMP MDSP_DIS                 // Jump to display loop
MDSP_4:    CMPI.B #$0C,D3        // 4 digit address. Check if 2 addresses or 1
    BGT MDSP_4RANGE              // if 2, get next addr
    JSR CONV_4                   // Convert Raw hex to actual hex
    MOVE.L D7,A5                 // A5 <- Address 1
    MOVE.L A5,A6
    ADD.L #$0F,A6                // A6 <- A5 + 16
    JMP MDSP_DIS                 // Jump to display loop
MDSP_4RANGE:    ADDQ #5,A1
    CMPI.B #$24,(A1)+            // Check if second address starts with $
    BEQ GET_NEXT_ADDR4           // If yes, proceed
    JMP MDSP_INVALID             // If not, INVALID
GET_NEXT_ADDR4:    JSR GET_ADDR_ASCII // Convert ASCII input to hex
    SUB.L D4,A1
    SUBQ #2,A1
    JSR CONV_4                   // Convert Raw hex to actual hex
    MOVE.L D7,A5                 // A5 <- Address 1
    MOVE.L D7,D5
```

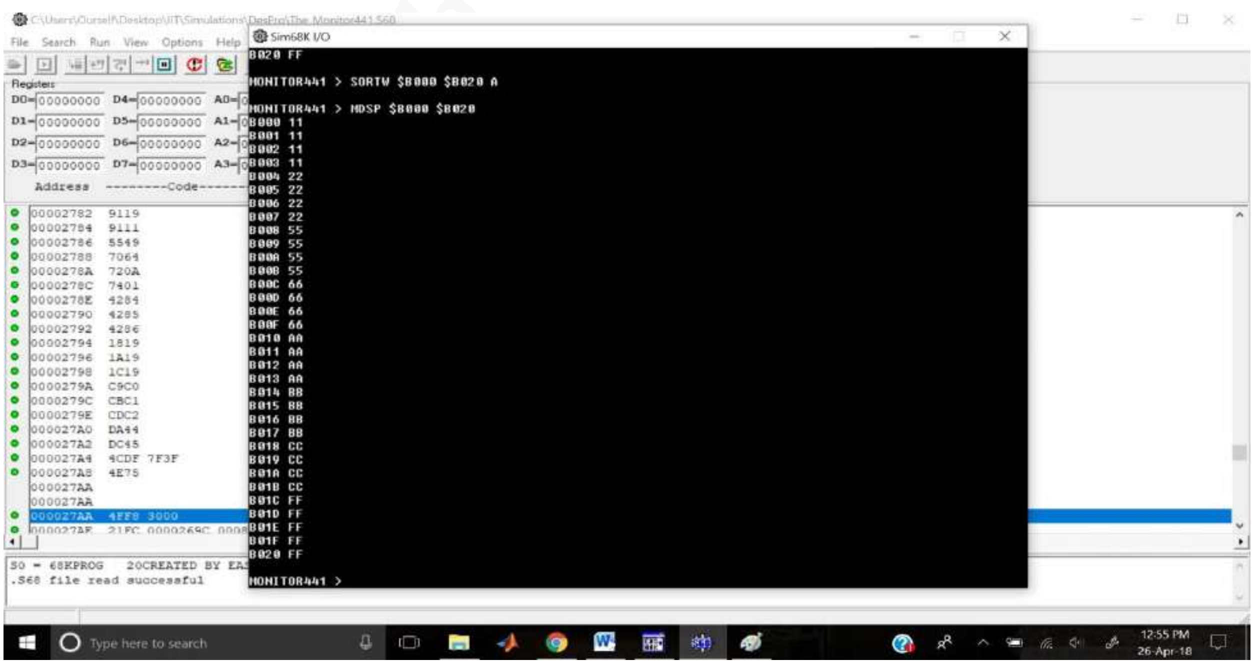
```
    ADD.L D2,A1
    ADDQ #2,A1
    JSR CONV_4                  // Convert Raw hex to actual hex
    MOVE.L D7,A6                // A6 <- Address 2
    JMP MDSP_DIS                // Jump to display loop
MDSP_5:    CMPI.B #$0C,D3       // 5 digit address. Check if 2 addresses or 1
    BGT MDSP_5RANGE             // if 2, get next addr
    JSR CONV_5                  // Convert Raw hex to actual hex
    MOVE.L D7,A5                // A5 <- Address 1
    MOVE.L A5,A6
    ADD.L #$0F,A6               // A6 <- A5 + 16
    JMP MDSP_DIS                // Jump to display loop
MDSP_5RANGE:    ADDQ #6,A1
    CMPI.B #$24,(A1)+           // Check if second address starts with $
    BEQ GET_NEXT_ADDR5          // If yes, proceed
    JMP MDSP_INVALID            // If not, INVALID
GET_NEXT_ADDR5:    JSR GET_ADDR_ASCII   // Convert ASCII input to hex
    SUB.L D4,A1
    SUBQ #2,A1
    JSR CONV_5                  // Convert Raw hex to actual hex
    MOVE.L D7,A5                // A5 <- Address 1
    MOVE.L D6,D5
    ADD.L D2,A1
    ADDQ #2,A1
    JSR CONV_5                  // Convert Raw hex to actual hex
    MOVE.L D7,A6                // A6 <- Address 2
MDSP_DIS: MOVE.L A5,D1
    JSR DISPDA                  // Display Address
    LEA SPACE,A1
    JSR DISP
    CLR.L D1
    MOVE.B (A5)+,D1             // Display Data
    JSR DISPDA
    LEA SPACE,A1
    JSR DISPCR
    CMP.L A5,A6                 // Check if A5 < A6
    BGE MDSP_DIS                // if yes, continue displaying
    JMP MDSP_END                // if not, end
MDSP_INVALID:   LEA INVALID_MSG,A1
    JSR DISPCR
MDSP_END:   MOVEM.L (SP)+,D0-D7/A0-A6 //Restore values into the registers
    RTS
```

2.2.2-) Debugger Command # 2 – SORTW (Sort Word)

The SORTW command sorts a block of memory. The addresses that govern the start and end of the block are obtained from the user and MUST be only even addresses. The input of an odd address shall result in an Invalid message display. The order of sorting, i.e, either ascending or descending is specified with the use of the letter **A** or **D** after the addresses have been entered. A detailed description of the syntax to be followed can be found in the Users' Manual.



Unsorted Data

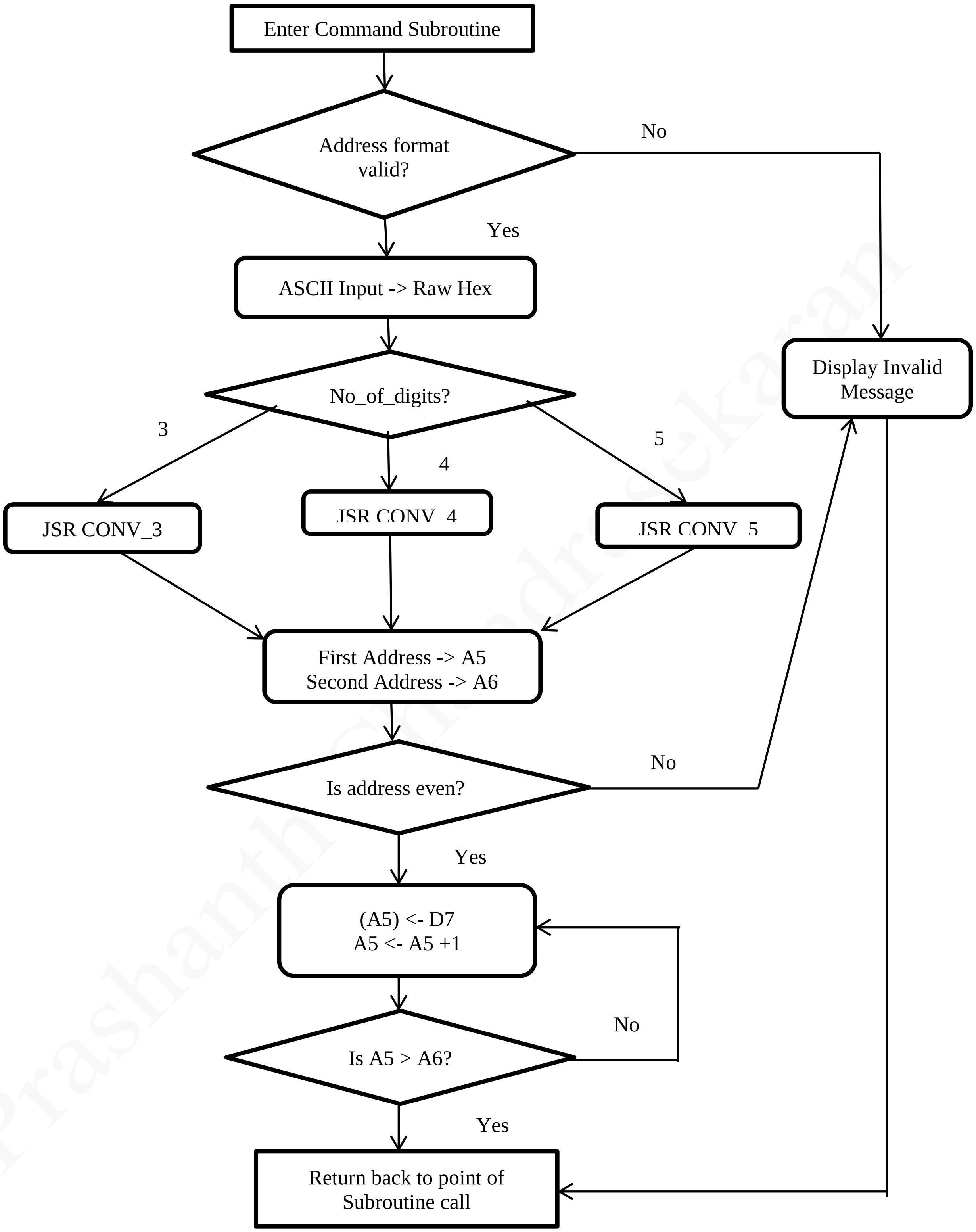


Sorted Data

2.2.2.1-) Algorithm and Flowchart

```
begin
        Check address format.
                If correct
                        do command
                else
                        display invalid message and wait for new input

        command:
                Convert ASCII input value of address to raw hex values
                Determine the order
                Determine number of digits in the address
                Use appropriate conversion block to obtain the address.
                Check if addr is even, (addr1%2==0 and addr2%2==0)
                        if true
                                put in A5 and A6
                                goto sorting_Algo
                        else
                                display invalid message and wait for new input
        sorting_Algo:
                begin bubble_Sort(list)
                        for all elements in the list
                                if(list[i] > list[i+1])  {or if(list[i] < list[i+1])}
                                        do swap(list[i],list[i+1])
                                end
                        end
                end bubble_Sort
end
```

Prashanth Chandrasekaran



Prashanth Chandrasekaran

2.2.2.2-) Assembly Code

<u>Order = Descending</u>

```
SORTW_D:
    MOVEM D0-D7/A0-A6,-(SP)          // Save registers on the stack
    MOVE.L A5,A2                     //Save a copy of starting address
D_SORT_AG:    MOVE.L A2,A5
D_CMP_CONTINUE:    CMP.W (A5)+,(A5)+//Compare consecutive memory locations
    BHI D_PERFORM_SWAP               // If less than, then SWAP
    SUBQ.L #2,A5                     //
    CMP.L A5,A6                      // else, continue comparing
    BNE D_CMP_CONTINUE               //
    JMP SORTW_D_END
D_PERFORM_SWAP:   MOVE.L -(A5),D0    // x = temp;
    SWAP.W D0                        // x = y;
    MOVE.L D0,(A5)                   // y = temp;
    BRA D_SORT_AG
SORTW_D_END:     MOVEM (SP)+,D0-D7/A0-A6 //Restore values into the registers
    RTS
```

<u>Order = Ascending</u>

```
SORTW_A:
    MOVEM D0-D7/A0-A6,-(SP)
    MOVE.L A5,A2
A_SORT_AG:    MOVE.L A2,A5
A_CMP_CONTINUE:    CMP.W (A5)+,(A5)+
    BCS A_PERFORM_SWAP               // If greater than, then SWAP
    SUBQ.L #2,A5
    CMP.L A5,A6
    BNE A_CMP_CONTINUE
    JMP SORTW_A_END
A_PERFORM_SWAP:   MOVE.L -(A5),D0
    SWAP.W D0
    MOVE.L D0,(A5)
    BRA A_SORT_AG
SORTW_A_END:     MOVEM (SP)+,D0-D7/A0-A6
    RTS
```

<u>Main</u>

```
SORTW:
    MOVEM.L D0-D7/A0-A6,-(SP)    // Save registers on the stack
    ADDQ #6,A1                   // Skip the 'SORTW '
    MOVEQ #0,D4
    CMPI.B #$24,(A1)+            // Check if first address starts with $
    BEQ SORTW_NEXT               // If yes, proceed
    JMP SORTW_INVALID            // if not, INVALID
SORTW_NEXT: JSR GET_ADDR_ASCII   // Convert ASCII to Raw Hex
    SUB.L D2,A1
    CMPI.B #$03,D4               //
    BEQ SORTW_DIGITS3            //
    CMPI.B #$04,D4               // Branch to subroutine based on number of
    BEQ SORTW_DIGITS4            // digits in the address
```

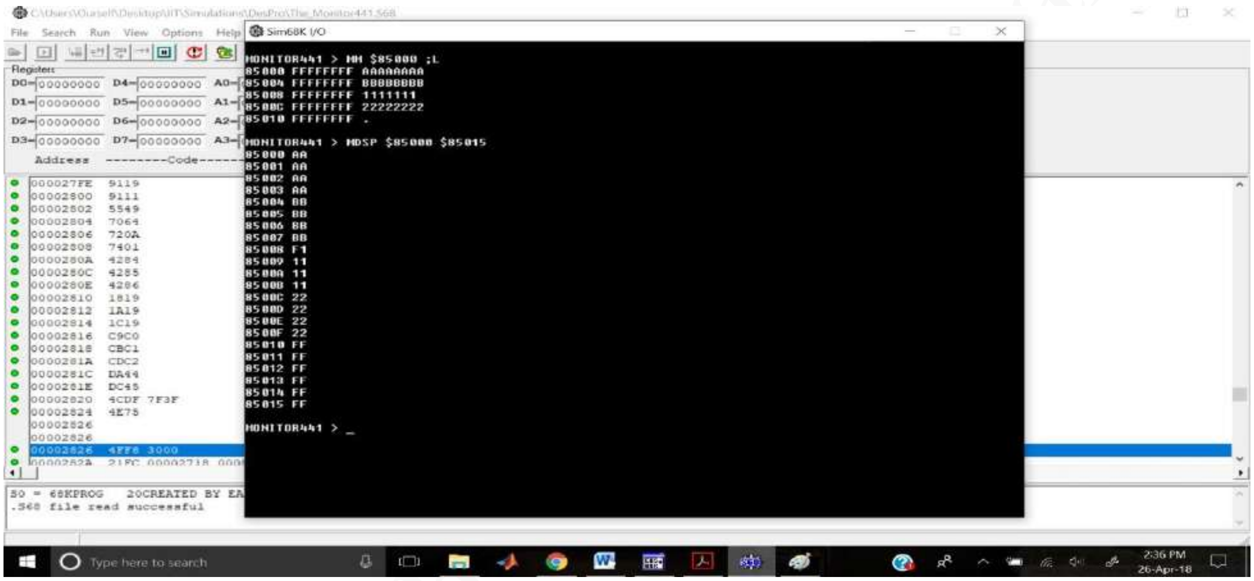

Prashanth Chandrasekaran

```
    CMPI.B #$05,D4               //
    BEQ SORTW_DIGITS5            //
SORTW_DIGITS3: ADDQ #4,A1
    CMPI.B #$24,(A1)+            // Check if second address starts with $
    BEQ SORTW_NEXT_1             // If yes, proceed
    JMP SORTW_INVALID            // if not, INVALID
SORTW_NEXT_1:    JSR GET_ADDR_ASCII   // Convert ASCII to Raw Hex
    SUB.L D4,A1
    SUBQ #2,A1
    JSR CONV_3                   // Convert Raw Hex to actual Hex
    MOVE.W D6,A5                 // A5 <- Address 1
    MOVE.L A5,D6
    MOVEQ #2,D2
    DIVU D2,D6                   //  do addr1%2
    SWAP.W D6
    CMPI.W #0,D6
    BEQ GET_NEXTADDR_3           // if 0, proceed
    JMP SORTW_INVALID            // if not, INVALID
GET_NEXTADDR_3:  ADDQ #5,A1
    JSR CONV_3                   // Convert Raw Hex to actual Hex
    MOVE.W D6,A6                 // A6 <- Address 2
    MOVE.L A6,D6
    MOVEQ #2,D2
    DIVU D2,D6                   //  do addr2%2
    SWAP.W D6
    CMPI.W #0,D6
    BEQ SORTW_SETUP              // if 0, proceed
    JMP SORTW_INVALID            // if not, INVALID
SORTW_DIGITS4:  ADDQ #5,A1
    CMPI.B #$24,(A1)+            // Check if second address starts with $
    BEQ SORTW_NEXT_2             // if yes, proceed
    JMP SORTW_INVALID            // if not, INVALID
SORTW_NEXT_2:    JSR GET_ADDR_ASCII  // Convert ASCII to Raw Hex
    SUB.L D4,A1
    SUBQ #2,A1
    JSR CONV_4                   // Convert Raw Hex to actual Hex
    MOVE.L D7,A5
    MOVE.L A5,D6
    MOVEQ #2,D2
    DIVU D2,D6                   //  do addr1%2
    SWAP.W D6
    CMPI.W #0,D6
    BEQ GET_NEXTADDR_4           // if 0, proceed
    JMP SORTW_INVALID            // if not, INVALID
GET_NEXTADDR_4:    ADDQ #6,A1
    JSR CONV_4                   // Convert Raw Hex to actual Hex
    MOVE.L D7,A6
    MOVE.L A6,D6
    MOVEQ #2,D2
    DIVU D2,D6                   //  do addr2%2
    SWAP.W D6
    CMPI.W #0,D6
    BEQ SORTW_SETUP              // if 0, proceed
    JMP SORTW_INVALID            // if not, INVALID
SORTW_DIGITS5:  ADDQ #6,A1
    CMPI.B #$24,(A1)+            // Check if second address starts with $
    BEQ SORTW_NEXT_3             // if yes, proceed
```

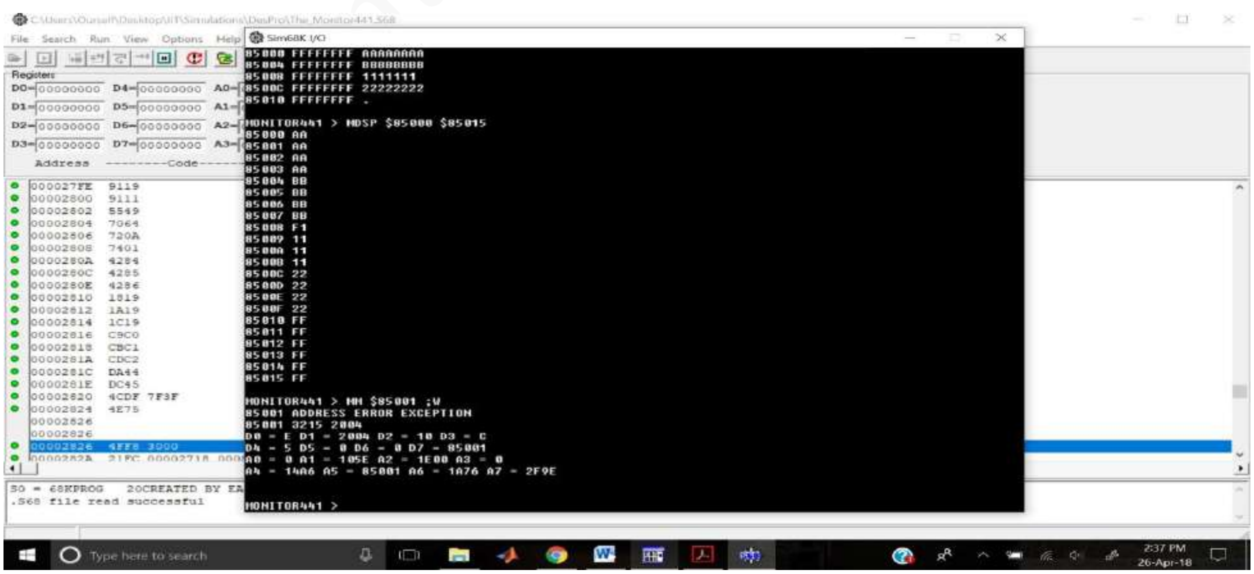
```
    JMP SORTW_INVALID           // if not, INVALID
SORTW_NEXT_3:    JSR GET_ADDR_ASCII // Convert ASCII to Raw Hex
    SUB.L D4,A1
    SUBQ #2,A1
    JSR CONV_5                  // Convert Raw Hex to actual Hex
    MOVE.L D7,A5
    MOVE.L A5,D6
    MOVEQ #2,D2
    DIVU D2,D6                  //  do addr1%2
    CMPI.W #0,D6
    BEQ GET_NEXTADDR_5          // if 0, proceed
    JMP SORTW_INVALID           // if not, INVALID
GET_NEXTADDR_5:    ADDQ #7,A1
    JSR CONV_5                  // Convert Raw Hex to actual Hex
    MOVE.L D7,A6
    MOVE.L A5,D6
    MOVEQ #2,D2
    DIVU D2,D6                  //  do addr2%2
    CMPI.W #0,D6
    BEQ SORTW_SETUP             // if 0, proceed
    JMP SORTW_INVALID           // if not, INVALID
SORTW_SETUP:  CMPI.B #$06,D4
    BEQ SORTW_DIGITS3_1
    CMPI.B #$08,D4
    BEQ SORTW_DIGITS4_1
    CMPI.B #$0A,D4
    BEQ SORTW_DIGITS5_1
SORTW_DIGITS3_1:   ADDQ #4,A1   //
    CMPI.B #$44,(A1)            // Check sorting order and go to appropriate
    BEQ GOTO_SORT_D             // sorting subroutine
    CMPI.B #$41,(A1)            //
    BEQ GOTO_SORT_A             //
SORTW_DIGITS4_1:   ADDQ #5,A1
    CMPI.B #$44,(A1)
    BEQ GOTO_SORT_D
    CMPI.B #$41,(A1)
    BEQ GOTO_SORT_A
SORTW_DIGITS5_1:   ADDQ #6,A1
    CMPI.B #$44,(A1)
    BEQ GOTO_SORT_D
    CMPI.B #$41,(A1)
    BEQ GOTO_SORT_A
GOTO_SORT_D:    JSR SORTW_D
    JMP SORTW_END
GOTO_SORT_A     JSR SORTW_A
    JMP SORTW_END
SORTW_INVALID:  LEA INVALID_MSG,A1
    JSR DISPCR
SORTW_END:  MOVEM.L (SP)+,D0-D7/A0-A6   //Restore values into the registers
    RTS
```

2.2.3-) Debugger Command # 3 – MM (Memory Modify)

The Memory Modify command displays the memory contents at the address location specified and if the user wishes, they can also modify the data. It can either display a byte of memory, a word of memory (2 bytes, 4 x 4bits) or a long word of memory (4 bytes, 8 x 4bits). To modify the contents of the memory the user must input either 2 hexadecimal digits, or 4 hexadecimal digits or 8 hexadecimal digits. Word and Long word memory modifications can be performed only from even addresses; odd address word or long word modifications will result in an Address Error Exception. The size is specified with the use of the letters **B,W** or **L**. To terminate further modifications, the user must enter a **"."**. A detailed description of the syntax to be followed can be found in the Users' Manual.



Example usage of Memory Modify command for longword data

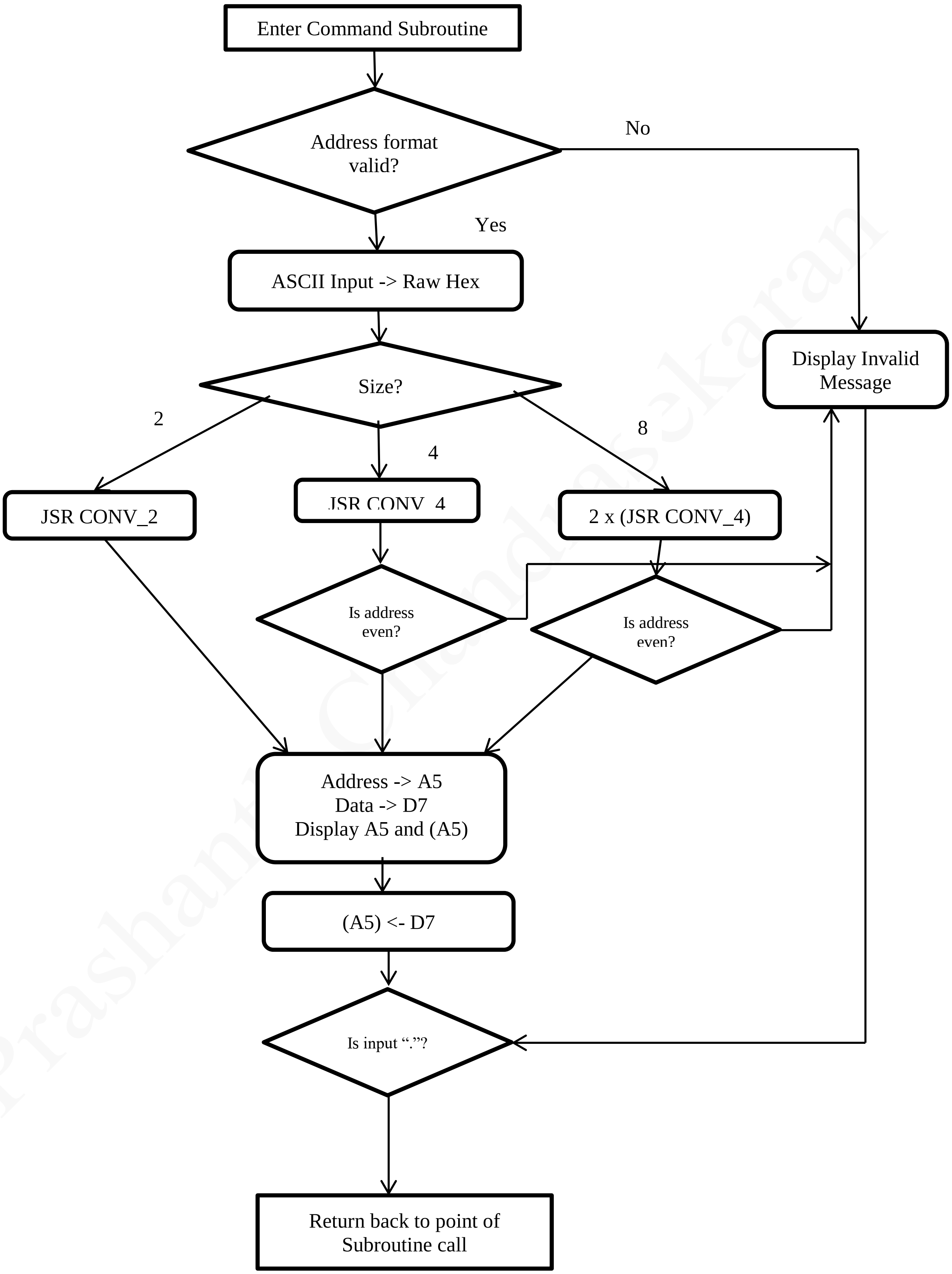


Example showing an erroneous input

2.2.3.1-) Algorithm and Flowchart

begin
Check address format.
If correct
do command
else
display invalid message and wait for new input
command:
Convert ASCII input value of address to raw hex values
Determine the size specified
Determine number of digits in the address
Use appropriate conversion block to obtain the address
Display data and wait for user input
if (user_input ==' . ') *[ASCII value of . = 0x2E]*
goto end
end

Prashanth Chandrasekaran



Enter Command Subroutine
Address format valid?
No
Yes
ASCII Input -> Raw Hex
Display Invalid Message
Size?
2
4
8
JSR CONV_2
JSR CONV_4
2 x (JSR CONV_4)
Is address even?
Is address even?
Address -> A5
Data -> D7
Display A5 and (A5)
(A5) <- D7
Is input "."?
Return back to point of Subroutine call

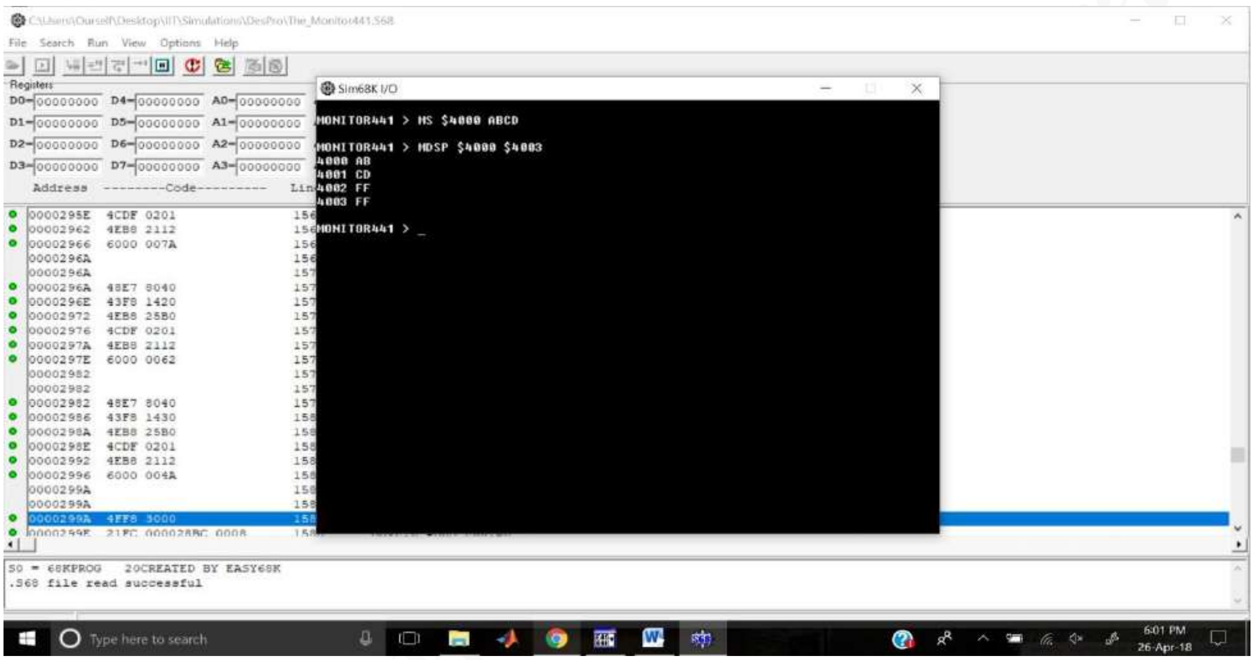
2.2.3.2-) Assembly Code

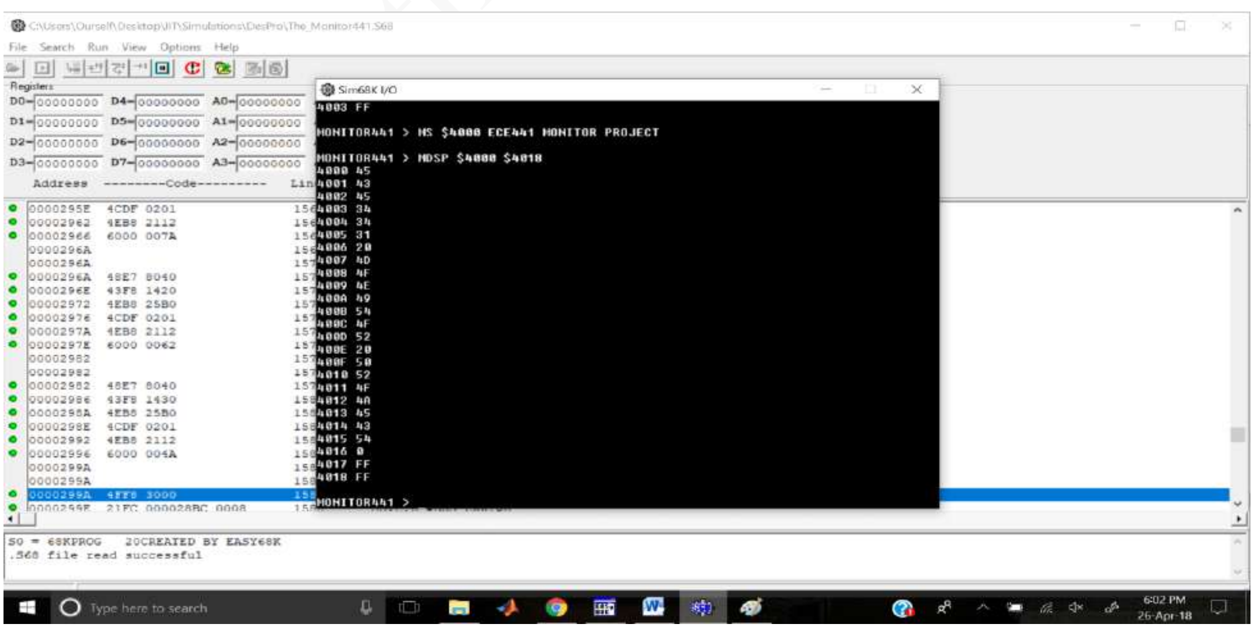
```
MM:
     MOVEM.L D0-D7/A0-A6,-(SP)  // Save registers on the stack
MM_BUFF EQU $4000
    MOVEQ #0,D4
    ADDQ #3,A1                  // Skip 'MM '
    CMPI.B #$24,(A1)+
    BEQ MM_NEXT
    JMP MM_INVALID
MM_NEXT:  JSR GET_ADDR_ASCII
    SUB.L D2,A1
    CMPI.B #$03,D4
    BEQ MM_DIGITS3
    CMPI.B #$04,D4
    BEQ MM_DIGITS4
    CMPI.B #$05,D4
    BEQ MM_DIGITS5
MM_DIGITS3: JSR CONV_3
    MOVE.W D6,A5
    JMP MM_CHECK_3
MM_DIGITS4: JSR CONV_4
    MOVE.L D7,A5
    JMP MM_CHECK_4
MM_DIGITS5: JSR CONV_5
    MOVE.L D7,A5
    JMP MM_CHECK_5
MM_CHECK_3:   ADDQ #4,A1
    CMPI.B #$3B,(A1)+           // check if ';' has been input
    BEQ MM_NEXT_1               // if yes, proceed
    JMP MM_INVALID              // if no, invalid
MM_CHECK_4:  ADDQ #5,A1
    CMPI.B #$3B,(A1)+
    BEQ MM_NEXT_1
    JMP MM_INVALID
MM_CHECK_5:  ADDQ #6,A1
    CMPI.B #$3B,(A1)+
    BEQ MM_NEXT_1
    JMP MM_INVALID
MM_NEXT_1:   CMPI.B #$42,(A1)
    BEQ MM_BYTE
    CMPI.B #$57,(A1)
    BEQ MM_WORD
    CMPI.B #$4C,(A1)
    BEQ MM_LONG
MM_BYTE:    MOVE.L A5,D1        //Byte Control
    JSR DISPDA
    LEA SPACE,A1
    JSR DISP
    CLR.L D1
    MOVE.B (A5),D1
    JSR DISPDA
    LEA SPACE,A1
    JSR DISP
    LEA MM_BUFF,A1
    MOVE.B #2,D0
```

```
    TRAP #15
    CMPI.B #$2E,(A1)
    BEQ MM_END
    JSR GET_DATA
    SUBQ #2,A1
    JSR CONV_2
    MOVE.B D3,(A5)+
    BRA MM_BYTE
MM_WORD:    MOVE.L A5,D1        //Word Control
    JSR DISPDA
    LEA SPACE,A1
    JSR DISP
    CLR.L D1
    MOVE.W (A5),D1
    JSR DISPDA
    LEA SPACE,A1
    MOVE.B #14,D0
    TRAP #15
    LEA MM_BUFF,A1
    MOVE.B #2,D0
    TRAP #15
    CMPI.B #$2E,(A1)
    BEQ MM_END
    JSR GET_DATA
    SUBQ #4,A1
    JSR CONV_4
    MOVE.W D7,(A5)+
    BRA MM_WORD
MM_LONG:    MOVE.L A5,D1        //Long Control
    JSR DISPDA
    LEA SPACE,A1
    JSR DISP
    CLR.L D1
    MOVE.L (A5),D1
    JSR DISPDA
    LEA SPACE,A1
    JSR DISP
    LEA MM_BUFF,A1
    MOVE.B #2,D0
    TRAP #15
    CMPI.B #$2E,(A1)
    BEQ MM_END
    JSR GET_DATA
    SUBQ #8,A1
    JSR CONV_4
    MOVE.W D7,(A5)+
    ADDQ #4,A1
    JSR CONV_4
    MOVE.W D7,(A5)+
    BRA MM_LONG
MM_INVALID: LEA INVALID_MSG,A1
    JSR DISPCR
MM_END: MOVEM.L (SP)+,D0-D7/A0-A6   //Restore values into the registers
    RTS
```

2.2.4-) Debugger Command # 4 – MS (Memory Set)

The **M**emory **S**et command is used to set data into the memory. It supports byte, word and long-word data set operations together with providing support for setting ASCII strings into the memory. To set the contents of the memory the user must input either 2 hexadecimal digits (byte), or 4 hexadecimal digits (word) or 8 hexadecimal digits (longword). If the user wishes to set a string of text into the memory, he/she may do so by simply typing out the text after specifying the address location. A detailed description of the syntax to be followed can be found in the Users' Manual.



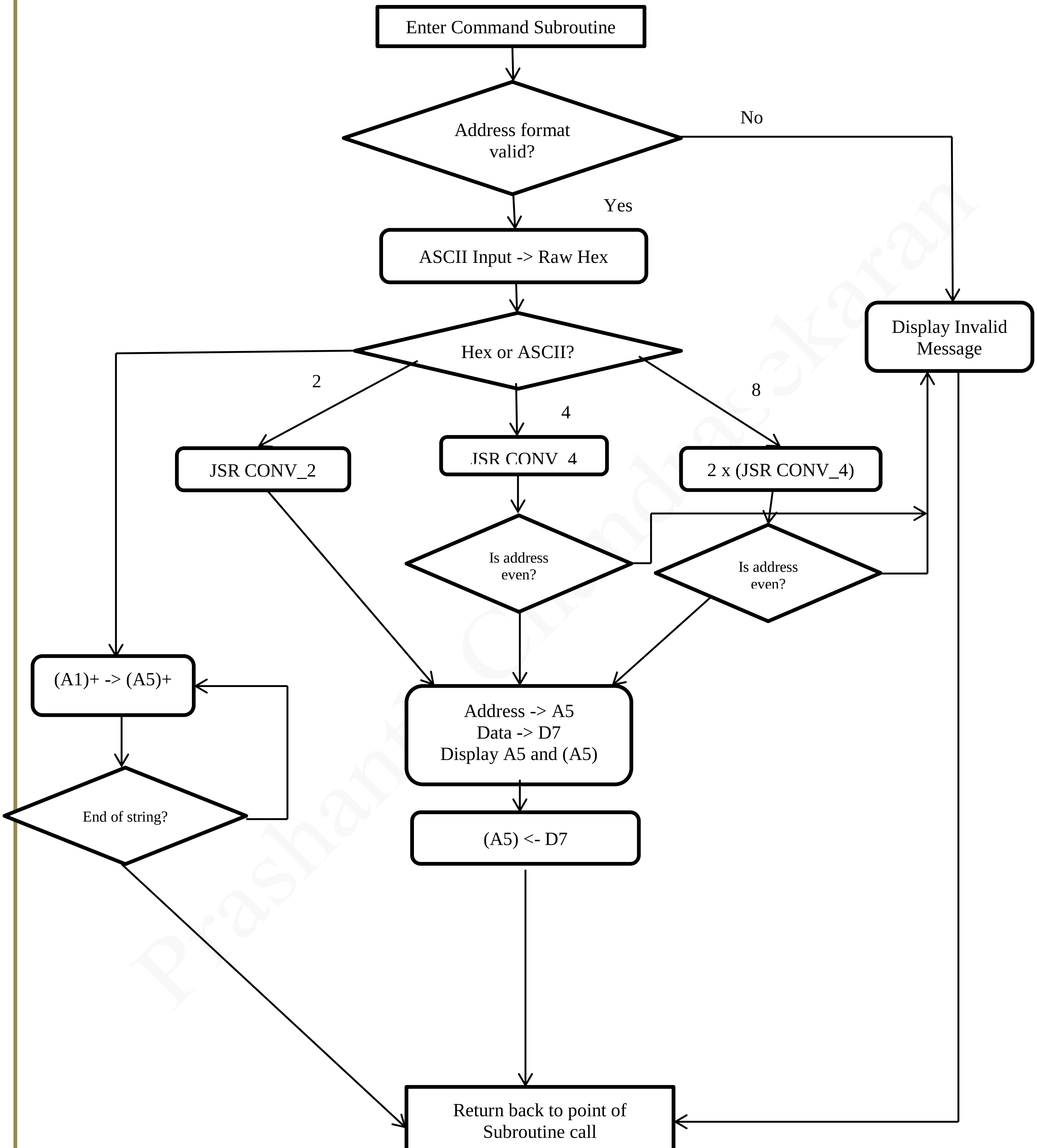


Examples usage of MS command to set word data and ASCII string text

2.2.4.1-) Algorithm and Flowchart

begin
Check address format.
If correct
do command
else
display invalid message and wait for new input
command:
Determine type of data
if string
copy from input buffer to memory location
else
Convert ASCII input value of address to raw hex values
Determine the size of the data
Determine number of digits in the address
Use appropriate conversion block to obtain the address
Convert ASCII input value of data to raw hex values
Use appropriate conversion block to obtain the data
Transfer data to memory location
end

Prashanth Chandrasekaran



Prashanth Chandrasekaran

2.2.4.2-) Assembly Code

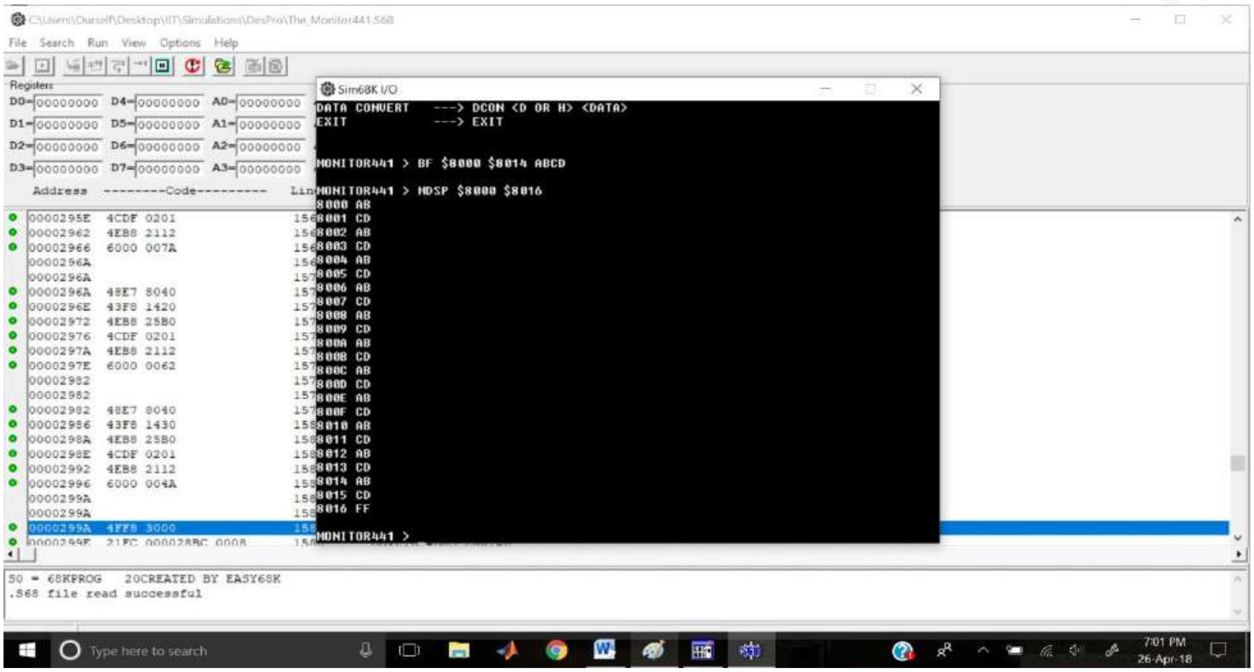
```
MS:
    MOVEM.L D0-D7/A0-A6,-(SP)
    MOVEQ #0,D4
    ADDQ #3,A1                   //Skip 'MS '
    CMPI.B #$24,(A1)+
    BEQ MS_NEXT
    JMP MS_INVALID
MS_NEXT:    JSR GET_ADDR_ASCII
    SUB.L D2,A1
    CMPI.B #$03,D4
    BEQ MS_3
    CMPI.B #$04,D4
    BEQ MS_4
    CMPI.B #$05,D4
    BEQ MS_5
MS_3:    JSR CONV_3
    MOVE.W D6,A5
    ADDQ #4,A1
    JMP GOTODATARX
MS_4:    JSR CONV_4
    MOVE.L D7,A5
    ADDQ #5,A1
    JMP GOTODATARX
MS_5:    JSR CONV_5
    MOVE.L D7,A5
    ADDQ #6,A1
GOTODATARX: CMPI.B #$12,D3      //Check if hexadecimal data or ASCII string
    BGT PUT_ASCII                // If string, go to string handler
    JSR GET_DATA                 //If not, get data
    CMPI.B #$02,D2               //
    BEQ MS_BYTE                  //
    CMPI.B #$04,D2               // Bracnh to appropriate subroutine based on
    BEQ MS_WORD                  // size of data
    CMPI.B #$08,D2               //
    BEQ MS_LONG                  //
MS_BYTE: SUBQ #2,A1
    JSR CONV_2
    MOVE.B D3,(A5)               // Move BYTE data into memory location
    JMP MS_END
MS_WORD: SUBQ #4,A1
    JSR CONV_4
    MOVE.W D7,(A5)               // Move WORD data into memory location
    JMP MS_END
MS_LONG: SUBQ #8,A1
    JSR CONV_4
    MOVE.W D7,(A5)+
    ADDQ #4,A1
    JSR CONV_4
    MOVE.W D7,(A5)               // Move 2-WORD data into memory location
    JMP MS_END
PUT_ASCII:  MOVE.B (A1)+,(A5)+  // Transfer from input buffer to memory
    CMPI.B #$00,(A1)
    BNE PUT_ASCII
    MOVE.B #$00,(A5)
```

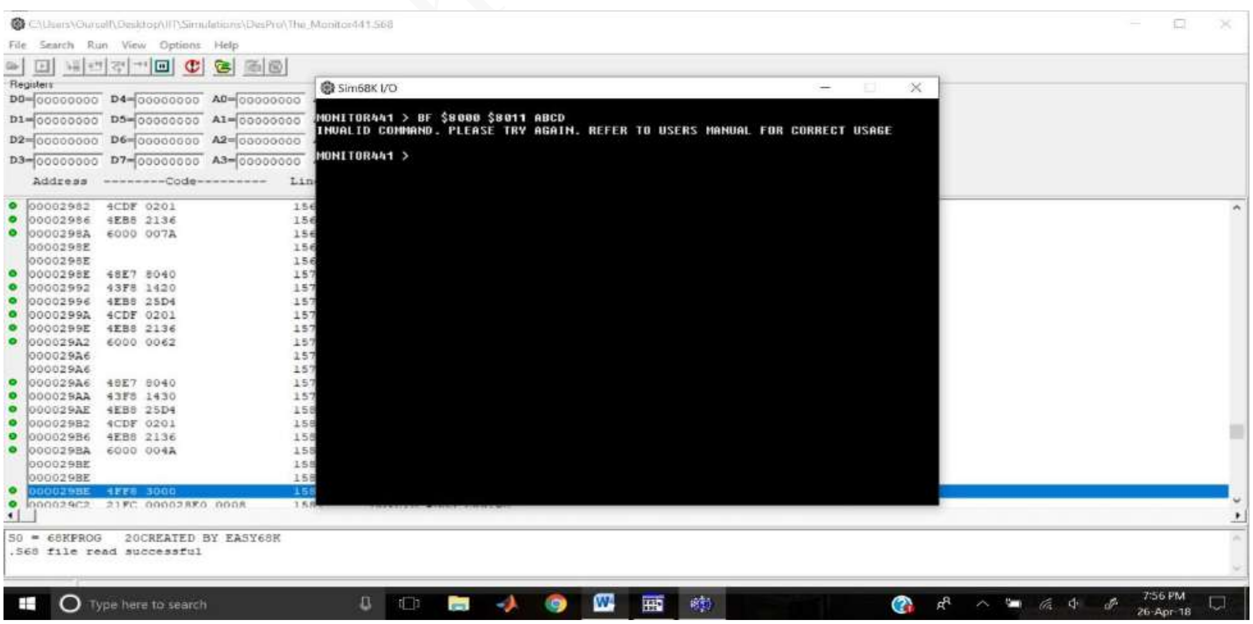
```
    JMP MS_END
MS_INVALID: LEA INVALID_MSG,A1
    JSR DISPCR
MS_END: MOVEM.L (SP)+,D0-D7/A0-A6       //Restore values into the registers
    RTS
```

Prashanth Chandrasekaran

2.2.5-) Debugger Command # 5 – BF (Block Fill)

The **B**lock **F**ill command is used to fill a block of memory with data. The user must provide the starting and ending address of the block, both of which must be even addresses. An odd address input will result in the display of the invalid command message. Only word size data is accepted, i.e, the user will only have to enter 4 hexadecimal digits of data. The data format is not right justified and hence, if a user wants to fill the block with "A", he/she must type in 000A. A detailed description of the syntax to be followed can be found in the Users' Manual.



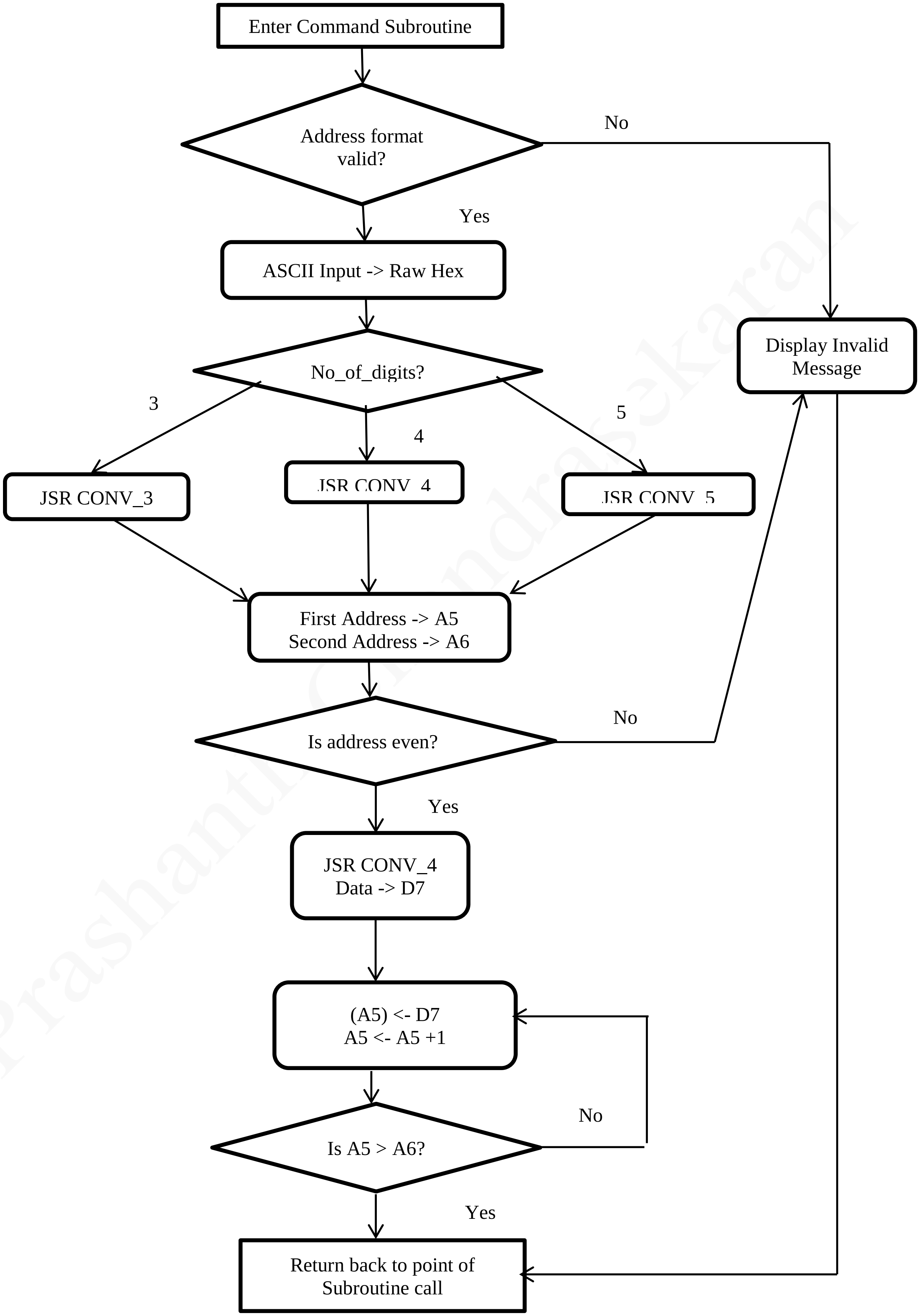


Example usage of the Block Fill command

2.2.5.1-) Algorithm and Flowchart

```
begin
    Check address format.
        If correct
            do command
        else
            display invalid message and wait for new input
    command:
        Convert ASCII input value of addresses to raw hex values
        Use appropriate conversion block to obtain the addresses
        Start addr -> A5, End addr -> A6
        if BOTH addresses are even
            Convert ASCII input value of data to raw hex values
            do fill
        else
            display invalid message and wait for new input
        fill:
            Put data in first word location [data -> Word(A5)+]
            Continue fill if A5<A6
end
```

Prashanth Chandrasekaran



Prashanth Chandrasekaran

2.2.5.2-) Assembly Code

```
BF:
    MOVEM.L D0-D7/A0-A6,-(SP)
    ADDQ #3,A1                  //Skip 'BF '
    MOVEQ #0,D4
    CMPI.B #$24,(A1)+
    BEQ BF_NEXT
    JMP BF_INVALID
BF_NEXT:    JSR GET_ADDR_ASCII
    SUB.L D2,A1
    CMPI.B #$03,D4
    BEQ BF_DIGITS3
    CMPI.B #$04,D4
    BEQ BF_DIGITS4
    CMPI.B #$05,D4
    BEQ BF_DIGITS5
BF_DIGITS3: ADDQ #4,A1
    CMPI.B #$24,(A1)+
    BEQ BFNEXT1
    JMP BF_INVALID
BFNEXT1:    JSR GET_ADDR_ASCII
    SUB.L D4,A1
    SUBQ #2,A1
    JSR CONV_3
    MOVE.W D6,A5
    MOVE.L A5,D6
    MOVEQ #2,D2
    DIVU D2,D6
    SWAP.W D6
    CMPI.B #$00,D6
    BEQ GET_NEXT_ADDR_3
    JMP BF_INVALID
GET_NEXT_ADDR_3:    ADD.L D2,A1
    ADDQ #3,A1
    JSR CONV_3
    MOVE.W D6,A6
    MOVE.L A6,D6
    MOVEQ #2,D2
    DIVU D2,D6
    SWAP.W D6
    CMPI.B #$00,D6
    BEQ BF_SKIP_3DIG
    JMP BF_INVALID
BF_DIGITS4: ADDQ #5,A1
    CMPI.B #$24,(A1)+
    BEQ BFNEXT2
    JMP BF_INVALID
BFNEXT2:    JSR GET_ADDR_ASCII
    SUB.L D4,A1
    SUBQ #2,A1
    JSR CONV_4
    MOVE.L D7,A5
    MOVE.L A5,D6
    MOVEQ #2,D2
    DIVU D2,D6
```

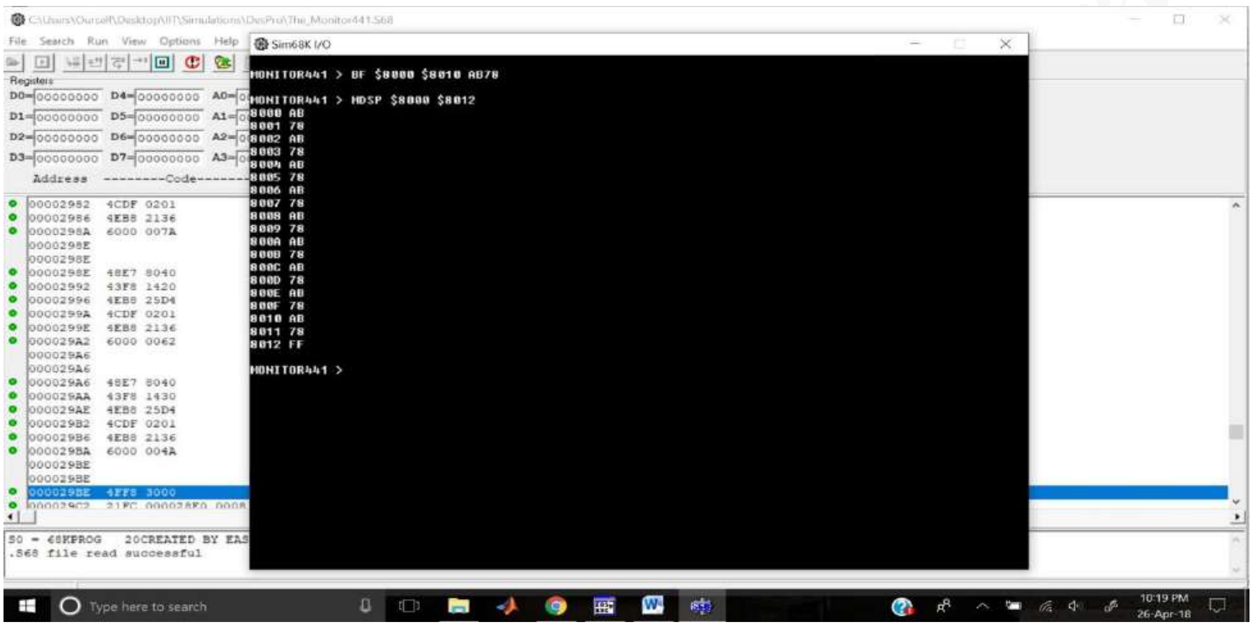
```
    SWAP.W D6
    CMPI #0,D6
    BEQ GET_NEXT_ADDR_4
    JMP BF_INVALID
GET_NEXT_ADDR_4:    ADD.L D2,A1
    ADDQ #4,A1
    JSR CONV_4
    MOVE.L D7,A6
    MOVE.L A6,D6
    MOVEQ #2,D2
    DIVU D2,D6
    SWAP.W D6
    CMPI #0,D6
    BEQ BF_SKIP_4DIG
    JMP BF_INVALID
BF_DIGITS5: ADDQ #6,A1
    CMPI.B #$24,(A1)+
    BEQ BFNEXT3
    JMP BF_INVALID
BFNEXT3:    JSR GET_ADDR_ASCII
    SUB.L D4,A1
    SUBQ #2,A1
    JSR CONV_5
    MOVE.L D7,A5
    MOVE.W A5,D6
    MOVEQ #2,D2
    DIVU D2,D6
    SWAP.W D6
    CMPI.B #$00,D6
    BEQ GET_NEXT_ADDR_5
    JMP BF_INVALID
GET_NEXT_ADDR_5:    ADD.L D2,A1
    ADDQ #5,A1
    JSR CONV_5
    MOVE.L D7,A6
    MOVE.W A6,D6
    MOVEQ #2,D2
    DIVU D2,D6
    SWAP.W D6
    CMPI.B #$00,D6
    BEQ BF_SKIP_5DIG
    JMP BF_INVALID
BF_SKIP_5DIG:    ADDQ #6,A1     // 5 digit addresses
    JSR GET_DATA                // Get Raw hex value of data
    SUBQ #4,A1
    JSR CONV_4                  // Convert Raw Hex to actual Hex
    JMP BFAG                    //Jump to Block Fill
BF_SKIP_4DIG:    ADDQ #5,A1     // 4 digit addresses
    JSR GET_DATA                // Get Raw hex value of data
    SUBQ #4,A1
    JSR CONV_4                  // Convert Raw Hex to actual Hex
    JMP BFAG                    //Jump to Block Fill
BF_SKIP_3DIG:  ADDQ #4,A1       // 4 digit addresses
    JSR GET_DATA                // Get Raw hex value of data
    SUBQ #4,A1
    JSR CONV_4                  // Convert Raw Hex to actual Hex
BFAG:    MOVE.W D7,(A5)+        // Fill block with input word data
```

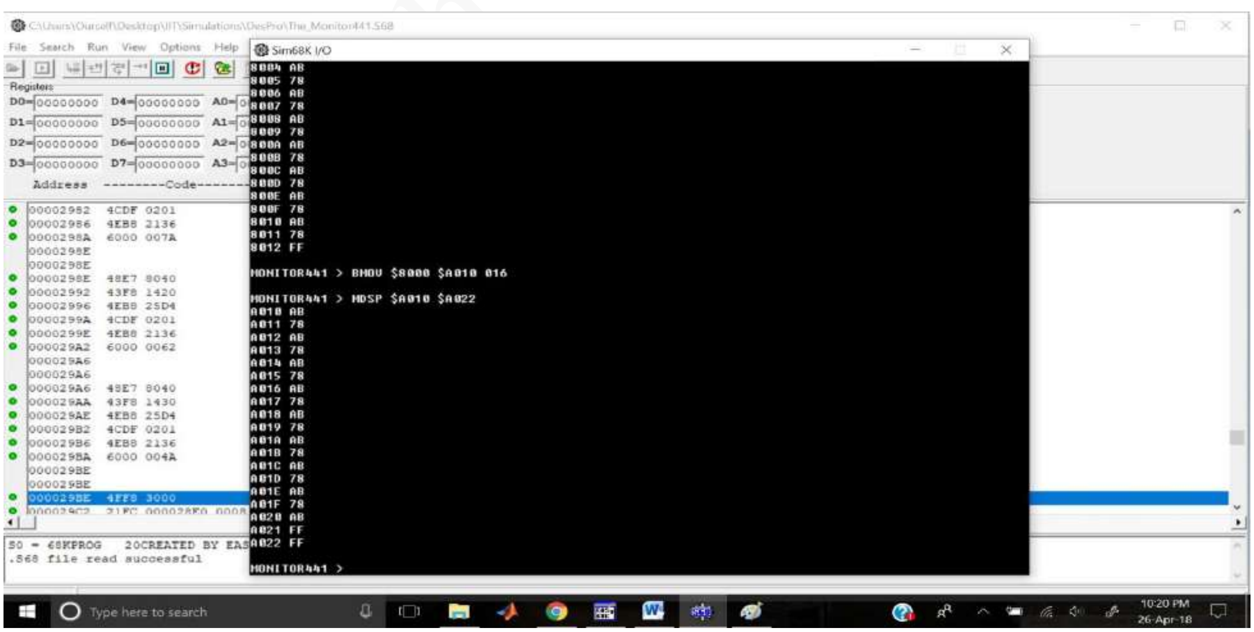
```
    CMP.L A6,A5
    BLE BFAG
    JMP BF_END
BF_INVALID: LEA INVALID_MSG,A1
    JSR DISPCR
BF_END:    MOVEM.L (SP)+,D0-D7/A0-A6
    RTS
```

Prashanth Chandrasekaran

2.2.6-) Debugger Command # 6 – BMOV (Block Move)

The **B**lock **M**ove command is used to move a block of memory from one location to another. The user must provide the starting address of the block to be moved and starting address of the memory location he/she wishes to move it to. The number of bytes must be specified after the addresses. A maximum of 999 bytes can be moved. The data format used to get the number of bytes is not right justified and hence, if a user wants to move 20 bytes, he/she must type in 020. A detailed description of the syntax to be followed can be found in the Users' Manual.



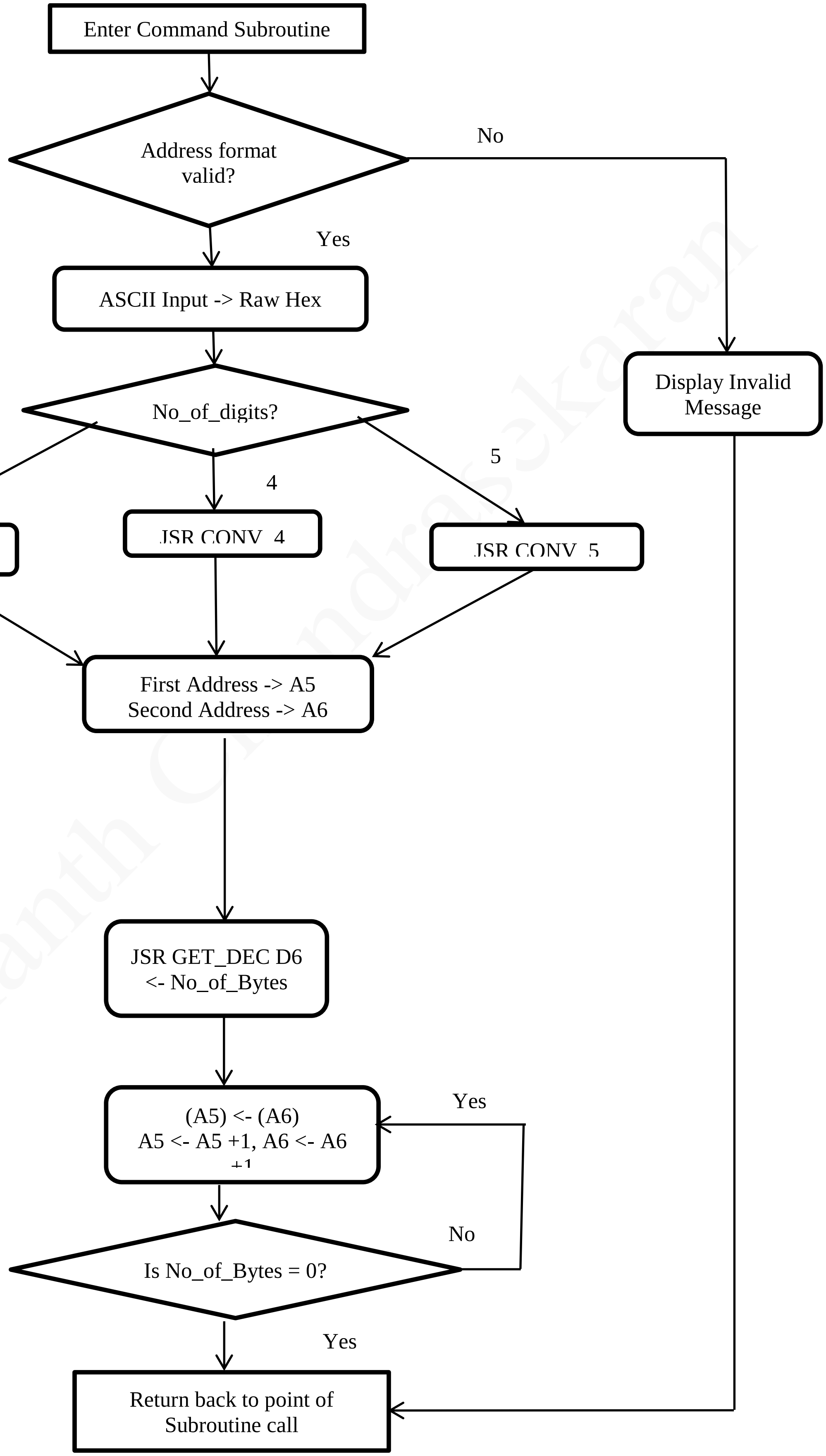


Example usage of the Block Move command

2.2.6.1-) Algorithm and Flowchart

```
begin
        Check address format.
                If correct
                        do command
                else
                        display invalid message and wait for new input
        command:
                Convert ASCII input value of addresses to raw hex values
                Use appropriate conversion block to obtain the addresses
                First block addr -> A5, Second block addr -> A6
                Convert ASCII input value of no_of_bytes to decimal
                do
                        (A5)+ == (A6)+
                        no_of_byres--
                until no_of_bytes==0
end
```

Prashanth Chandrasekaran


Enter Command Subroutine
Address format valid?
No
Yes
ASCII Input -> Raw Hex
Display Invalid Message
No_of_digits?
3
4
5
JSR CONV_3
JSR CONV_4
JSR CONV_5
First Address -> A5
Second Address -> A6
JSR GET_DEC D6 <- No_of_Bytes
(A5) <- (A6)
A5 <- A5 +1, A6 <- A6 +1
Yes
No
Is No_of_Bytes = 0?
Yes
Return back to point of Subroutine call

2.2.6.2-) Assembly Code

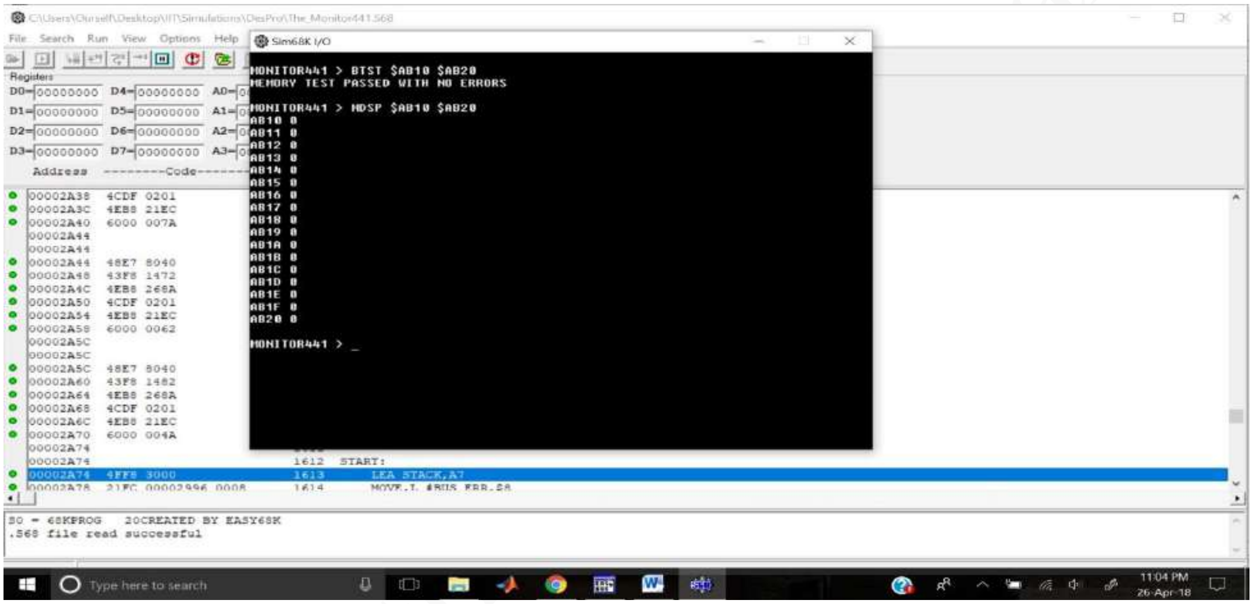
```
BMOV:
    MOVEM.L D0-D7/A0-A6,-(SP)
    MOVEQ #0,D4
    ADDQ #5,A1                      //Skip 'BMOV '
    CMPI.B #$24,(A1)+
    BEQ BMOV_NEXT
    JMP BMOV_INVALID
BMOV_NEXT:  JSR GET_ADDR_ASCII
    SUB.L D2,A1
    CMPI.B #$03,D4
    BEQ BMOV_DIGITS3
    CMPI.B #$04,D4
    BEQ BMOV_DIGITS4
    CMPI.B #$05,D4
    BEQ BMOV_DIGITS5
BMOV_DIGITS3: ADDQ #4,A1
    CMPI.B #$24,(A1)+
    BEQ BMOV_NEXT_1
    JMP BMOV_INVALID
BMOV_NEXT_1:    JSR GET_ADDR_ASCII
    SUB.L D4,A1
    SUBQ #2,A1
    JSR CONV_3
    MOVE.W D6,A5
    ADD.L D2,A1
    ADDQ #2,A1
    JSR CONV_3
    MOVE.W D6,A6
    JMP GETNOOFBYTES
BMOV_DIGITS4: ADDQ #5,A1
    CMPI.B #$24,(A1)+
    BEQ BMOV_NEXT_2
    JMP BMOV_INVALID
BMOV_NEXT_2:    JSR GET_ADDR_ASCII
    SUB.L D4,A1
    SUBQ #2,A1
    JSR CONV_4
    MOVE.L D7,A5
    ADD.L D2,A1
    ADDQ #2,A1
    JSR CONV_4
    MOVE.L D7,A6
    JMP GETNOOFBYTES
BMOV_DIGITS5:   ADDQ #6,A1
    CMP.B #$24,(A1)+
    BEQ BMOV_NEXT_3
    JMP BMOV_INVALID
BMOV_NEXT_3:    JSR GET_ADDR_ASCII
    SUB.L D4,A1
    SUBQ #2,A1
    JSR CONV_5
    MOVE.L D7,A5
    ADD.L D2,A1
    ADDQ #2,A1
```

```
    JSR CONV_5
    MOVE.L D7,A6
GETNOOFBYTES:   CMPI.B #$06,D4
    BEQ A1UPDATE3
    CMPI.B #$08,D4
    BEQ A1UPDATE4
    CMPI.B #$0A,D4
    BEQ A1UPDATE5
A1UPDATE3:    ADDQ #4,A1
    JSR GET_DEC                   // Convert Raw Hex (BCD) to Hex
    JMP BMOV_AG                   // Perform Block Move
A1UPDATE4:    ADDQ #5,A1
    JSR GET_DEC
    JMP BMOV_AG
A1UPDATE5:    ADDQ #6,A1
    JSR GET_DEC
    JMP BMOV_AG
BMOV_AG:    MOVE.B (A5)+,(A6)+
    DBEQ D6,BMOV_AG
    JMP BMOV_END
BMOV_INVALID: LEA INVALID_MSG,A1
   JSR DISPCR
BMOV_END:   MOVEM.L (SP)+,D0-D7/A0-A6
    RTS
```

Prashanth Chandrasekaran

2.2.7-) Debugger Command # 7 – BTST (Block Test)

The **B**lock **T**est command performs a destructive test of a block of memory. The user must provide the starting address and ending address of the block of memory to be tested and the command MUST be terminated with a SPACE. To test the memory, FF_{16} is written into all the memory locations within the block. The data is then read location by location from the block and compared with the value written. If the no differences between the written and read data are found, the memory test passes. The user is notified and the block is filled with zeros. If however, a difference is found, the memory test fails and the Address location of failure, the data written into the location and the data read from it are all displayed to the user. A detailed description of the syntax to be followed can be found in the Users' Manual.

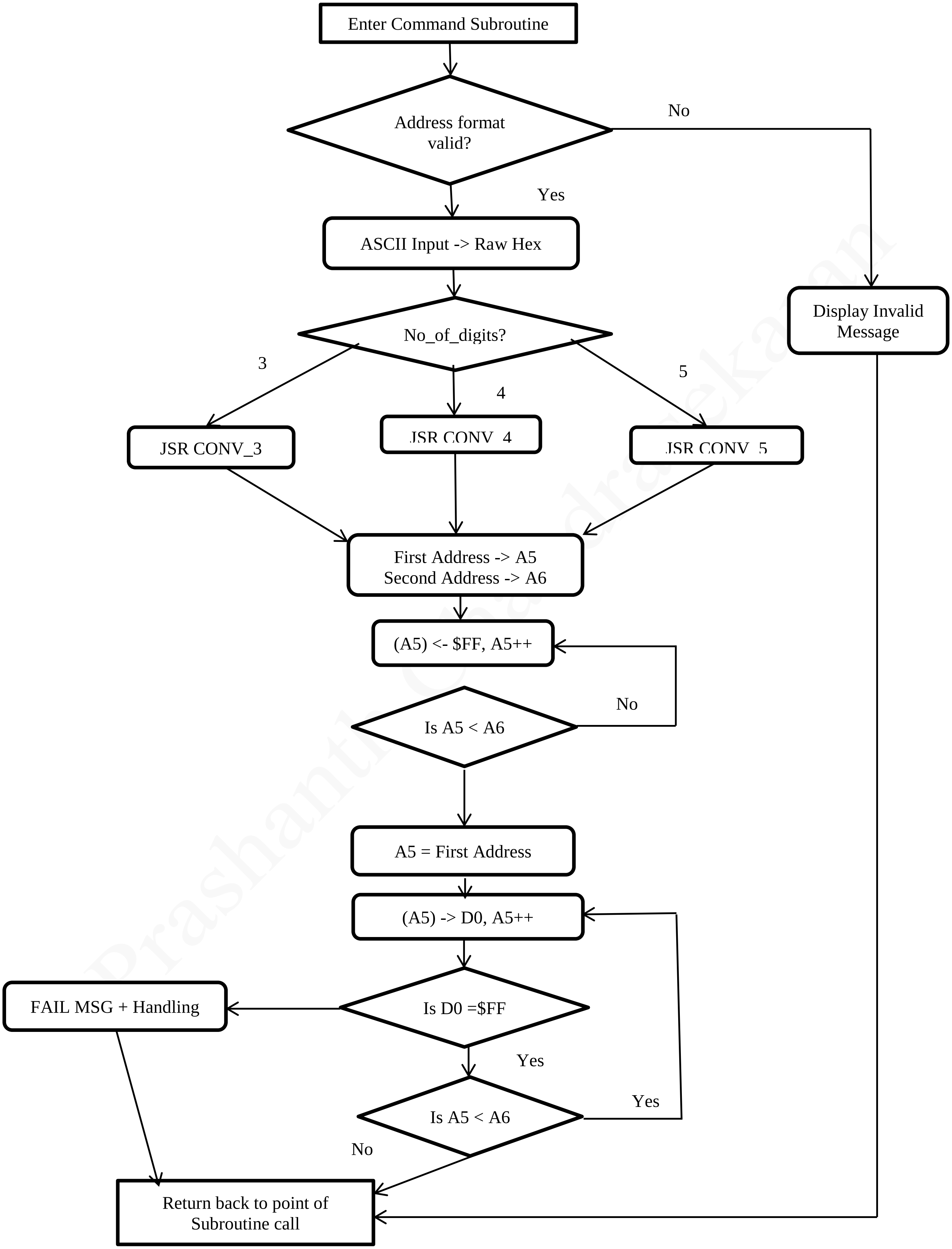


Example usage of the Block Test command

2.2.7.1-) Algorithm and Flowchart

```
begin
        Check address format.
                If correct
                        do command
                else
                        display invalid message and wait for new input
        command:
                Convert ASCII input value of addresses to raw hex values
                Use appropriate conversion block to obtain the addresses
                Start addr -> A5, End addr -> A6
                do
                        (A5)+ <-  $FF
                until (A5 ≤ A6)
                Start addr -> A5
                do
                        (A5)+ -> D0
                        Compare D0 with written value
                        if same
                                NO ERROR
                                Fill block with zeros
                        else
                                ERROR
end
```

Prashanth Chandrasekaran



Prashanth Chandrasekaran

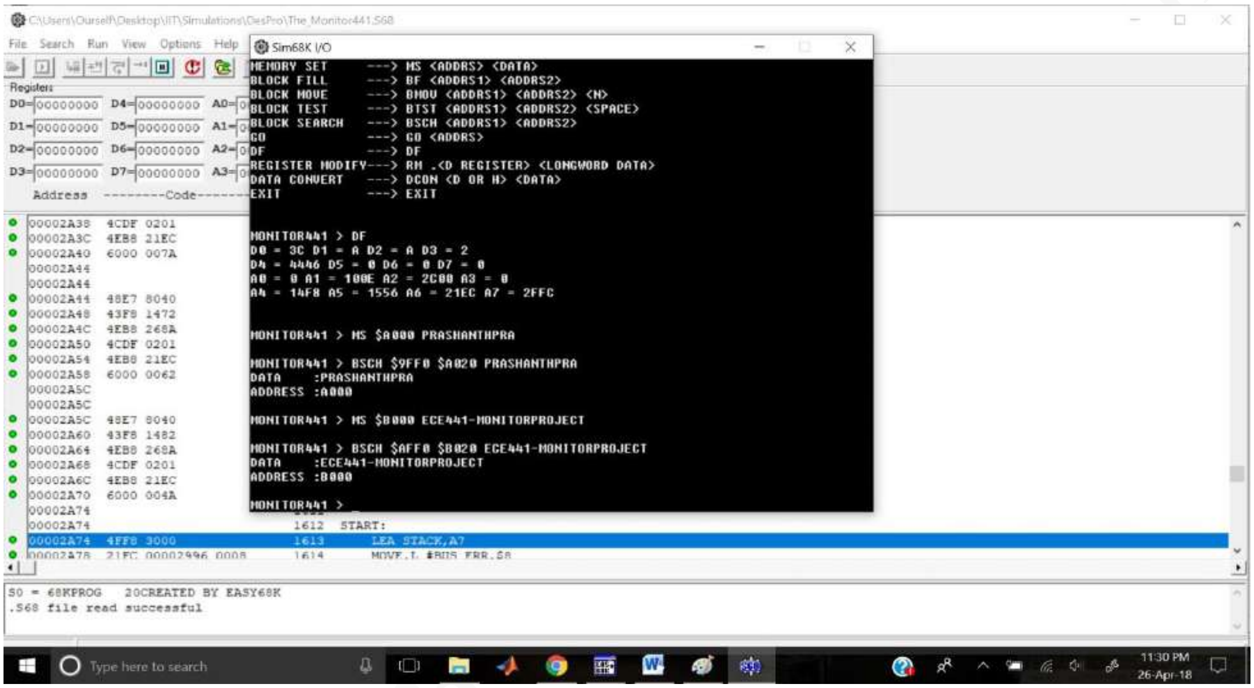
2.2.7.2-) Assembly Code

```
BTST:
    MOVEM.L D0-D7/A0-A6,-(SP)
    MOVEQ #0,D4
    ADDQ #5,A1                  //Skip 'BTST '
    CMPI.B #$24,(A1)+
    BEQ BTST_NEXT
    JMP BTST_INVALID
BTST_NEXT: JSR GET_ADDR_ASCII
    SUB.L D2,A1
    CMPI.B #$03,D4
    BEQ BTST_DIGITS3
    CMPI.B #$04,D4
    BEQ BTST_DIGITS4
    CMPI.B #$05,D4
    BEQ BTST_DIGITS5
BTST_DIGITS3: ADDQ #4,A1
    CMPI.B #$24,(A1)+
    BEQ BTST_NEXT_1
    JMP BTST_INVALID
BTST_NEXT_1:    JSR GET_ADDR_ASCII
    SUB.L D4,A1
    SUBQ #2,A1
    JSR CONV_3
    MOVE.W D6,A5
    MOVE.L D6,D5
    ADD.L D2,A1
    ADDQ #2,A1
    JSR CONV_3
    MOVE.W D6,A6
    JMP BTST_BEG
BTST_DIGITS4: ADDQ #5,A1
    CMPI.B #$24,(A1)+
    BEQ BTST_NEXT_2
    JMP BTST_INVALID
BTST_NEXT_2:    JSR GET_ADDR_ASCII
    SUB.L D4,A1
    SUBQ #2,A1
    JSR CONV_4
    MOVE.L D7,A5
    MOVE.L D7,D5
    ADD.L D2,A1
    ADDQ #2,A1
    JSR CONV_4
    MOVE.L D7,A6
    JMP BTST_BEG
BTST_DIGITS5: ADDQ #6,A1
    CMPI.B #$24,(A1)+
    BEQ BTST_NEXT_3
    JMP BTST_INVALID
BTST_NEXT_3:    JSR GET_ADDR_ASCII
    SUB.L D4,A1
    SUBQ #2,A1
    JSR CONV_5
    MOVE.L D7,A5
```

```
    MOVE.L D7,D5
    ADD.L D2,A1
    ADDQ #2,A1
    JSR CONV_5
    MOVE.L D7,A6
BTST_BEG:   MOVE.B #$FF,(A5)+           // Write in FF
    CMP.L A5,A6
    BGE BTST_BEG
    MOVE.L D5,A5
BTST_READ:    MOVE.B (A5)+,D0           // Read from the memory
    CMPI.B #$FF,D0                      // Read == Write??
    BEQ BTST_READ_CONTINUE              // If yes, proceed
    JMP BTST_ERR                        // if not, error
BTST_READ_CONTINUE: CMP.L A5,A6         // Continue test through entire block
    BGE BTST_READ
    JMP BTST_NOERR
BTST_NOERR: MOVE.L D5,A5                // Do if NO ERROR
BTSTSET:    MOVE.B #$00,(A5)+           // Fill block with zeros
    CMP.L A5,A6
    BGE BTSTSET
    LEA BTSTMSG_1,A1                    // Display Pass message
    JSR DISPCR
    JMP BTST_END
BTST_ERR:   LEA BTSTMSG_2,A1            // Display Fail Message
    JSR DISPCR
    LEA BTSTMSG_5,A1
    JSR DISP
    CLR.L D1
    SUBQ #1,A5
    MOVE.L A5,D1
    JSR DISPDA                          // Display FAIL address
    LEA SPACE,A1
    JSR DISPCR
    LEA BTSTMSG_4,A1
    JSR DISP
    CLR.L D1
    MOVE.B (A5),D1                      // Display READ data
    JSR DISPDA
    LEA SPACE,A1
    JSR DISPCR
    LEA BTSTMSG_3,A1
    JSR DISP
    CLR.L D1
    MOVE.B #$FF,D1                      // Display WRITE data
    JSR DISPDA
    JMP BTST_END
BTST_INVALID: LEA INVALID_MSG,A1
    JSR DISPCR
BTST_END:   MOVEM.L (SP)+,D0-D7/A0-A6
    RTS
```

2.2.8-) Debugger Command # 8 – BSCH (Block Search)

The **B**lock **S**earch command searches for an ASCII string of data in a block of memory. The user must provide the starting address and ending address of the block of memory to be searched in. The string is entered after the addresses have been entered. If the string is found in the block, the user is notified of its address within the block and the data stored therein. If however, the string is not found, the user is once again notified of the issue. A detailed description of the syntax to be followed can be found in the Users' Manual.



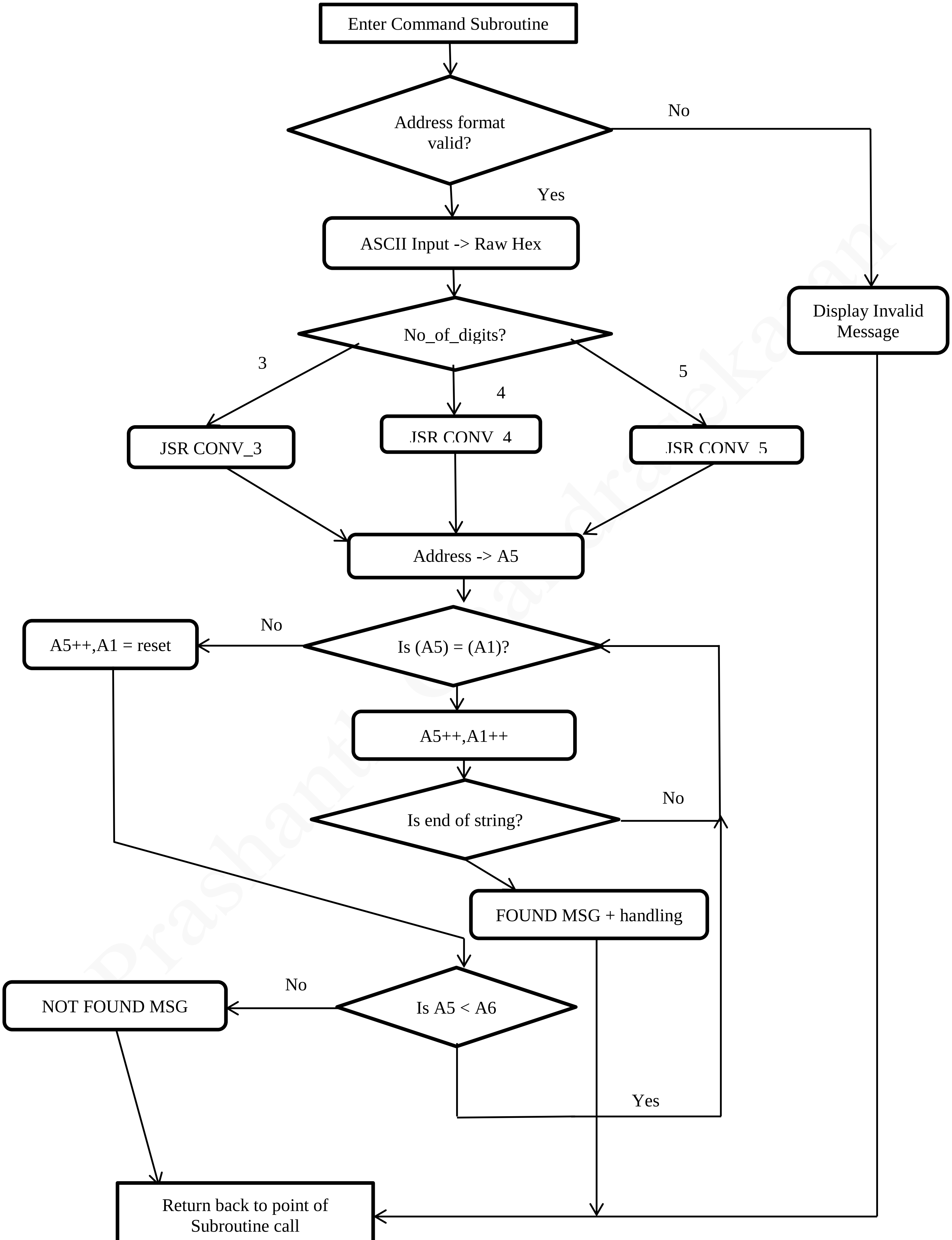
Example usage of the Block Search command

Prashanth

2.2.8.1-) Algorithm and Flowchart

```
begin
        Check address format.
                If correct
                        do command
                else
                        display invalid message and wait for new input
        command:
                Convert ASCII input value of addresses to raw hex values
                Use appropriate conversion block to obtain the addresses
                Start addr -> A5, End addr -> A6
                algo_beg:       Compare first letter of string with data in first memory location
                if same
                        do
                                Compare next letter of string with next memory location
                        until( NOT_EQUAL or end_of_string)
                else
                        A5 <- A5+1
                        if (end_of_block)
                                NOT_FOUND
                        go to algo_beg
end
```

Prashanth Chandrasekaran


Enter Command Subroutine
Address format valid?
No
Yes
ASCII Input -> Raw Hex
Display Invalid Message
No_of_digits?
3
4
5
JSR CONV_3
JSR CONV_4
JSR CONV_5
Address -> A5
No
A5++,A1 = reset
Is (A5) = (A1)?
A5++,A1++
No
Is end of string?
FOUND MSG + handling
No
NOT FOUND MSG
Is A5 < A6
Yes
Return back to point of Subroutine call

2.2.8.2-) Assembly Code

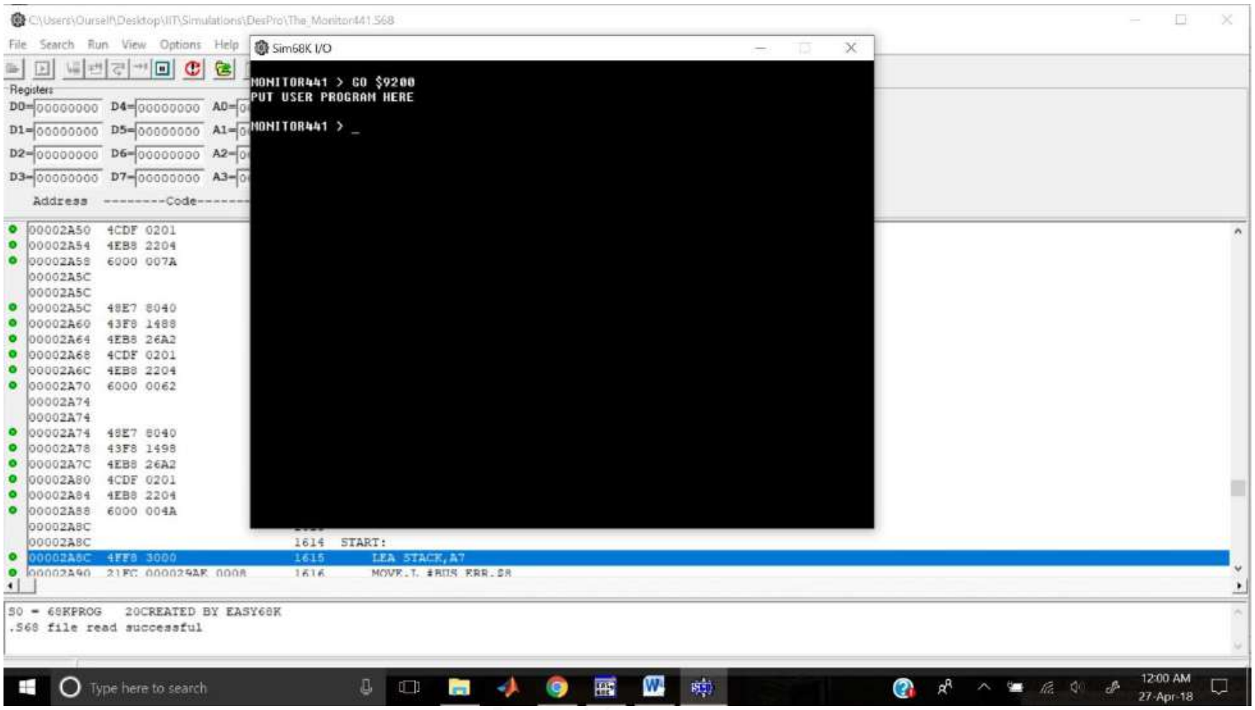
```
BSCH:
    MOVEM.L D0-D7/A0-A6,-(SP)
    MOVEQ #0,D4
    ADDQ #5,A1                      //Skip 'BSCH '
    CMPI.B #$24,(A1)+
    BEQ BSCH_NEXT
    JMP BSCH_INVALID
BSCH_NEXT:    JSR GET_ADDR_ASCII
    SUB.L D2,A1
    CMPI.B #$03,D4
    BEQ BSCH_DIGITS3
    CMPI.B #$04,D4
    BEQ BSCH_DIGITS4
    CMPI.B #$05,D4
    BEQ BSCH_DIGITS5
BSCH_DIGITS3:    ADDQ #4,A1
    CMPI.B #$24,(A1)+
    BEQ BSCH_NEXT_1
    JMP BSCH_INVALID
BSCH_NEXT_1:    JSR GET_ADDR_ASCII
    SUB.L D4,A1
    SUBQ #2,A1
    JSR CONV_3
    MOVE.W D6,A5
    MOVE.L D6,D7
    MOVE.L D6,D5
    ADD.L D2,A1
    ADDQ #2,A1
    JSR CONV_3
    MOVE.W D6,A6
    MOVE.L D6,D7
    JMP BSCH_CHECK
BSCH_DIGITS4:    ADDQ #5,A1
    CMPI.B #$24,(A1)+
    BEQ BSCH_NEXT_2
    JMP BSCH_INVALID
BSCH_NEXT_2:    JSR GET_ADDR_ASCII
    SUB.L D4,A1
    SUBQ #2,A1
    JSR CONV_4
    MOVE.L D7,A5
    MOVE.L D7,D5
    ADD.L D2,A1
    ADDQ #2,A1
    JSR CONV_4
    MOVE.L D7,A6
    JMP BSCH_CHECK
BSCH_DIGITS5:    ADDQ #6,A1
    CMPI.B #$24,(A1)+
    BEQ BSCH_NEXT_3
    JMP BSCH_INVALID
BSCH_NEXT_3:    JSR GET_ADDR_ASCII
    SUB.L D4,A1
    SUBQ #2,A1
    JSR CONV_5
```

```
    MOVE.L D7,A5
    MOVE.L D7,D5
    ADD.L D2,A1
    ADDQ #2,A1
    JSR CONV_5
    MOVE.L D7,A6
BSCH_CHECK: CMPI.B #$06,D4
    BEQ BSCH_DIGITS3_1
    CMPI.B #$08,D4
    BEQ BSCH_DIGITS4_1
    CMPI.B #$0A,D4
    BEQ BSCH_DIGITS5_1
BSCH_DIGITS3_1:    ADDQ #4,A1
    JMP BSCH_AG_BEG
BSCH_DIGITS4_1:    ADDQ #5,A1
    JMP BSCH_AG_BEG
BSCH_DIGITS5_1:    ADDQ #6,A1
BSCH_AG_BEG:   MOVEQ #0,D6              // et length of the string
GET_LENGTH:    ADDQ #1,D6
    CMPI.B #$00,(A1)+
    BNE GET_LENGTH
    CMPI.B #$FF,(A1)
    BEQ BSCH_PROCEED
    BRA GET_LENGTH
BSCH_PROCEED:    SUB.L D6,A1
    SUBQ #1,D6
BSCH_AG:    CMP.B (A1)+,(A5)+       // Compare input buffer with memory
    BEQ BSCH_CHECK_NEXT             // If first letter match, proceed
    SUBQ #1,A1                      // If not, reset input buffer & recompare
    ADDQ #1,D5
    CMP A5,A6                       // Do until entire block is searched
    BGE BSCH_AG
    JMP BSCH_NOTFOUND
BSCH_CHECK_NEXT:    SUBQ #1,D6
    CMPI.B #0,D6
    BNE BSCH_AG
    JMP BSCH_DISPLAY
BSCH_NOTFOUND:  LEA BSCHMSG_3,A1    // Display Not Found message
    JSR DISPCR
    JMP BSCH_END
BSCH_INVALID: LEA INVALID_MSG,A1
    MOVE.B #14,D0
    TRAP #15
    JMP BSCH_END
BSCH_DISPLAY:   LEA BSCHMSG_1,A1
    JSR DISP
    MOVE.L D5,A1
    JSR DISPCR                      // Display Data
    LEA BSCHMSG_2,A1
    JSR DISP
    MOVE.L D5,D1
    JSR DISPDA                      // Display Address
    LEA SPACE,A1
    JSR DISPCR
    JMP BSCH_END
BSCH_END:    MOVEM.L (SP)+,D0-D7/A0-A6
    RTS
```

Prashanth Chandrasekaran

2.2.9-) Debugger Command # 9 – GO

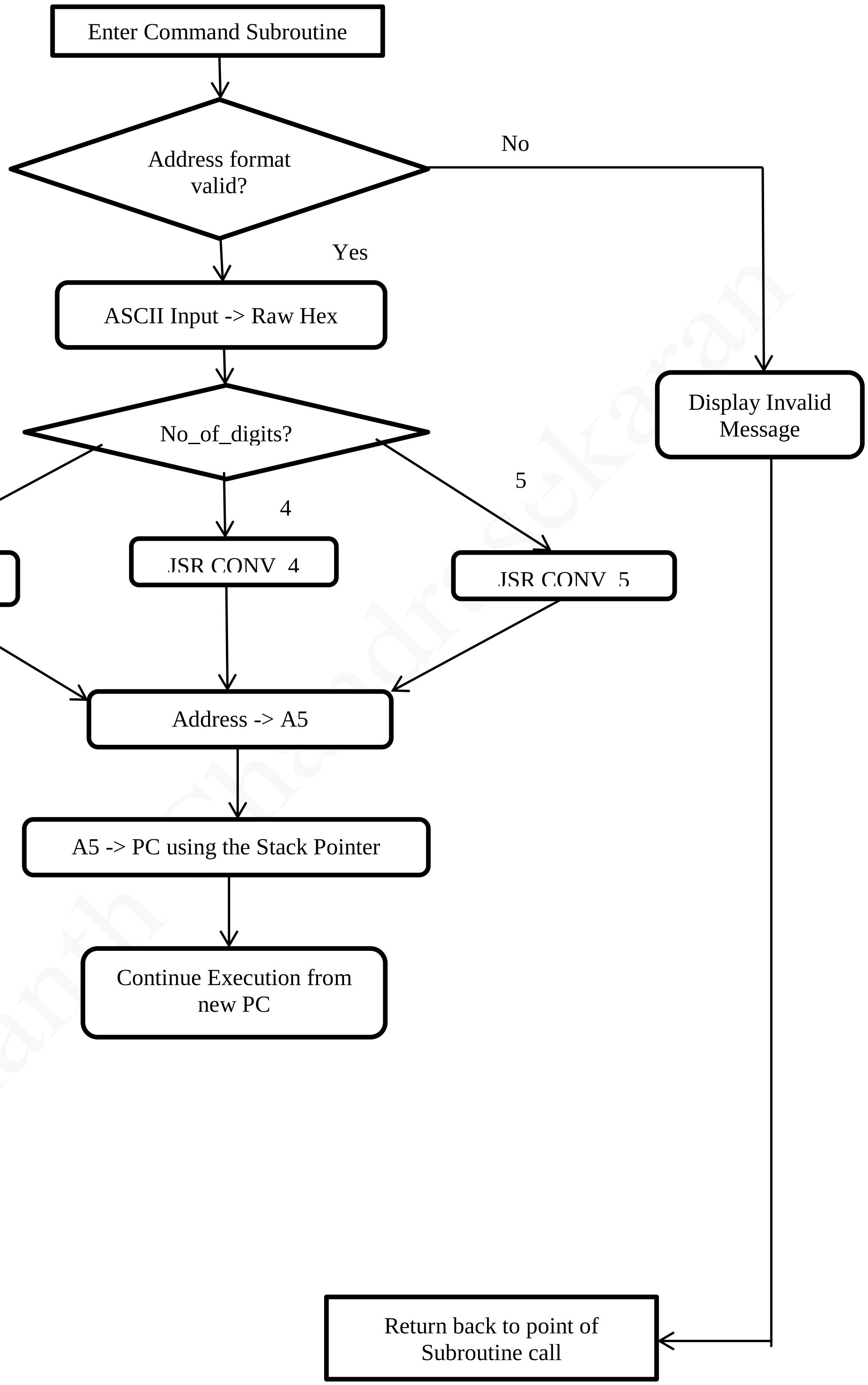
The **GO** Command is used to execute user programs. The command must be followed up by the Address of the users' program and MUST be terminated with a SPACE. A detailed description of the syntax to be followed can be found in the Users' Manual.



Example usage of the GO command

2.2.9.1-) Algorithm and Flowchart

begin
Check address format.
If correct
do command
else
display invalid message and wait for new input
command:
Convert ASCII input value of addresses to raw hex values
Use appropriate conversion block to obtain the addresses
Update program counter with input address
Continue execution
end


Enter Command Subroutine
Address format valid?
No
Yes
ASCII Input -> Raw Hex
Display Invalid Message
No_of_digits?
3
4
5
JSR CONV_3
JSR CONV_4
JSR CONV_5
Address -> A5
A5 -> PC using the Stack Pointer
Continue Execution from new PC
Return back to point of Subroutine call

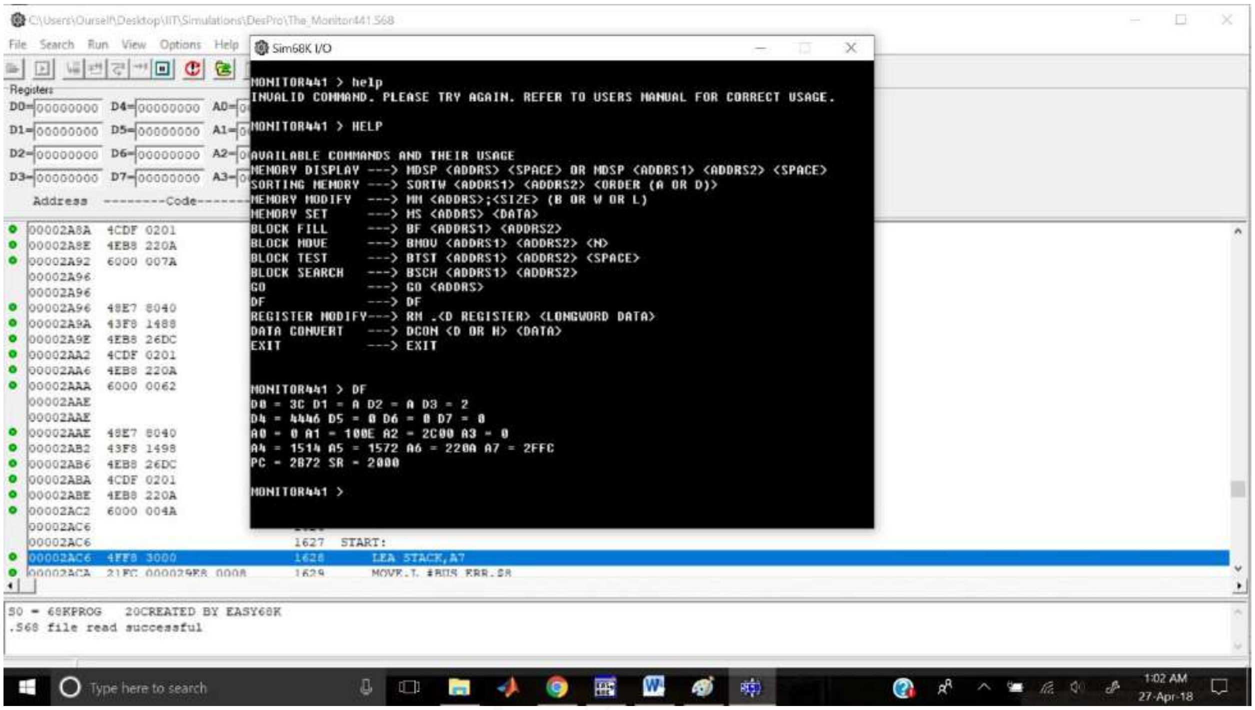
2.2.9.2-) Assembly Code

```
GO:
    MOVEM.L D0-D7/A0-A6,-(SP)
    MOVEQ #0,D4
    ADDQ #3,A1
    CMPI.B #$24,(A1)+
    BEQ GO_NEXT
    JMP GO_INVALID
GO_NEXT:    JSR GET_ADDR_ASCII
    SUB.L D2,A1
    CMPI.B #$03,D4
    BEQ GO_3
    CMPI.B #$04,D4
    BEQ GO_4
GO_3:    JSR CONV_3
    MOVE.W D6,A5
    BSR GO_TO_ADDR
GO_4:   JSR CONV_4
    MOVE.L D7,A5
    BSR GO_TO_ADDR
GO_TO_ADDR: MOVE.W A5,(2,SP)
    RTS
GO_INVALID: LEA INVALID_MSG,A1
    JSR DISPCR
GO_END: MOVEM.L (SP)+,D0-D7/A0-A6
    RTS
```

Prashanth Chandrasekaran

2.2.10-) Debugger Command # 10 – DF (Display Formatted Registers)

The **DF** Command displays all the registers along with their corresponding values. The subroutine behind this instruction is also used in exception processing to aide the user in finding out where an error might have occured.



Example usage of the Display Formatted Registers command

2.2.10.1-) Algorithm and Flowchart

begin

i = 0

do

print data register D[i]

i <- i+1

while(i<8)

do

print address register A[i]

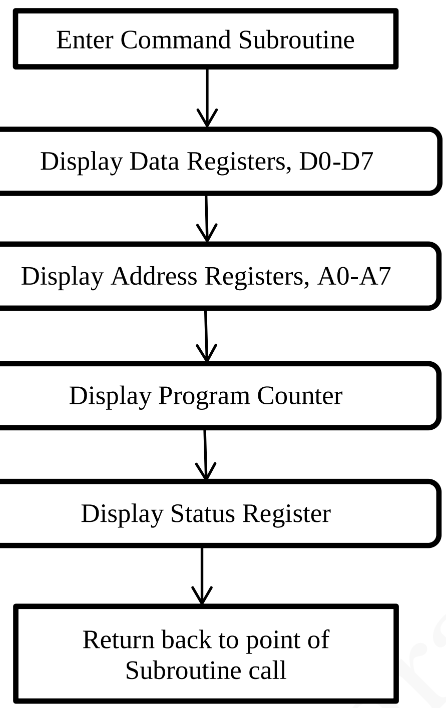
i <- i+1

while(i<8)

print PC

print SR

end



2.2.10.2-) Assembly Code

```
DF:
    MOVEM.L D0-D7/A0-A7,-(SP)
    LEA DATAREG_0,A1
    JSR DISP
    MOVE.L (SP),D1
    JSR DISPDA
    LEA SPACE,A1
    JSR DISP
    LEA DATAREG_1,A1
    JSR DISP
    ADD.L #$4,SP
    MOVE.L (SP),D1
    JSR DISPDA
    LEA SPACE,A1
    JSR DISP
    LEA DATAREG_2,A1
    JSR DISP
    ADD.L #$4,SP
    MOVE.L (SP),D1
    JSR DISPDA
    LEA SPACE,A1
    JSR DISP
    LEA DATAREG_3,A1
    JSR DISP
    ADD.L #$4,SP
    MOVE.L (SP),D1
    JSR DISPDA
```

Prashanth Chandrasekaran

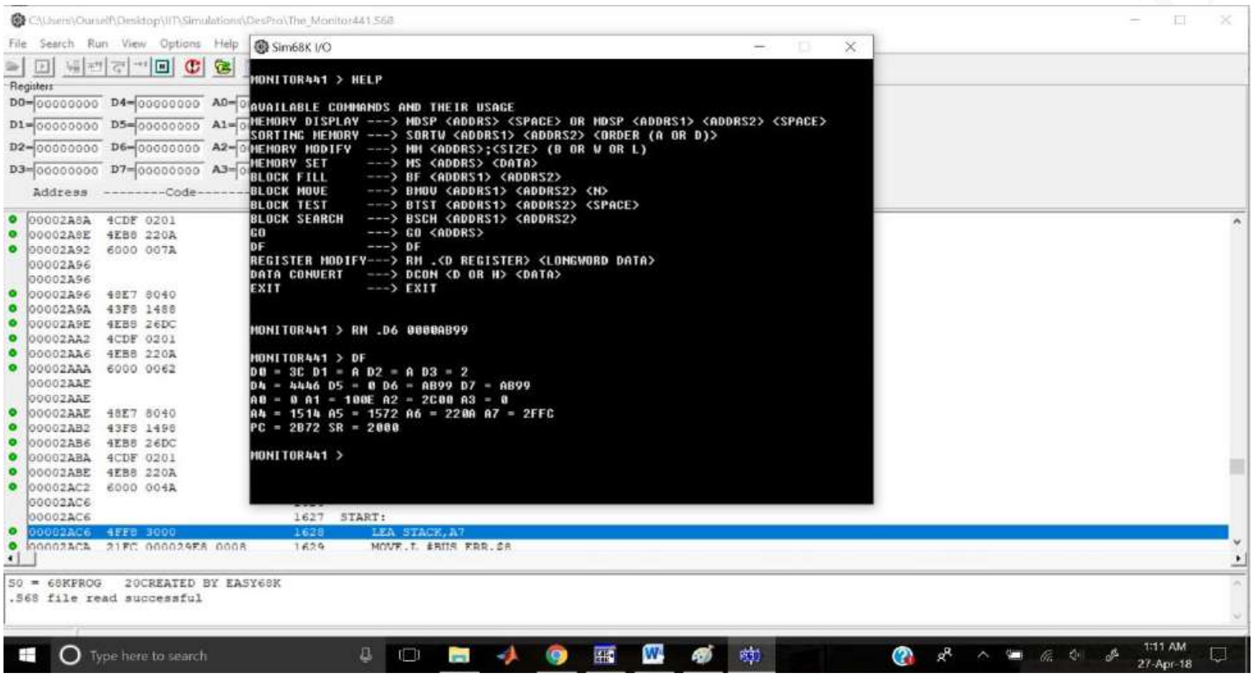
```
LEA SPACE,A1
JSR DISPCR
LEA DATAREG_4,A1
JSR DISP
ADD.L #$4,SP
MOVE.L (SP),D1
JSR DISPDA
LEA SPACE,A1
JSR DISP
LEA DATAREG_5,A1
JSR DISP
ADD.L #$4,SP
MOVE.L (SP),D1
JSR DISPDA
LEA SPACE,A1
JSR DISP
LEA DATAREG_6,A1
JSR DISP
ADD.L #$4,SP
MOVE.L (SP),D1
JSR DISPDA
LEA SPACE,A1
JSR DISP
LEA DATAREG_7,A1
JSR DISP
ADD.L #$4,SP
MOVE.L (SP),D1
JSR DISPDA
LEA SPACE,A1
JSR DISPCR
LEA ADDRREG_0,A1
JSR DISP
ADD.L #$4,SP
MOVE.L (SP),D1
JSR DISPDA
LEA SPACE,A1
JSR DISP
LEA ADDRREG_1,A1
JSR DISP
ADD.L #$4,SP
MOVE.L (SP),D1
JSR DISPDA
LEA SPACE,A1
JSR DISP
LEA ADDRREG_2,A1
JSR DISP
ADD.L #$4,SP
MOVE.L (SP),D1
JSR DISPDA
LEA SPACE,A1
JSR DISP
LEA ADDRREG_3,A1
JSR DISP
ADD.L #$4,SP
MOVE.L (SP),D1
JSR DISPDA
LEA SPACE,A1
```

```
JSR DISPCR
LEA ADDRREG_4,A1
JSR DISP
ADD.L #$4,SP
MOVE.L (SP),D1
JSR DISPDA
LEA SPACE,A1
JSR DISP
LEA ADDRREG_5,A1
JSR DISP
ADD.L #$4,SP
MOVE.L (SP),D1
JSR DISPDA
LEA SPACE,A1
JSR DISP
LEA ADDRREG_6,A1
JSR DISP
ADD.L #$4,SP
MOVE.L (SP),D1
JSR DISPDA
LEA SPACE,A1
JSR DISP
LEA ADDRREG_7,A1
JSR DISP
ADD.L #$4,SP
MOVE.L (SP),D1
JSR DISPDA
LEA SPACE,A1
JSR DISPCR
LEA PROGCOUNT,A1
JSR DISP
ADD.L #$4,SP
MOVE.L (SP),D1
JSR DISPDA
SUB.L #$40,SP
LEA SPACE,A1
JSR DISP
LEA STATUSREG,A1
JSR DISP
MOVE.W SR,D1
JSR DISPDA
LEA SPACE,A1
JSR DISPCR
MOVEM.L (SP)+,D0-D7/A0-A7
RTS
```

Prashanth Chandrasekaran

2.2.11-) Debugger Command # 11 – RM (Register Modify)

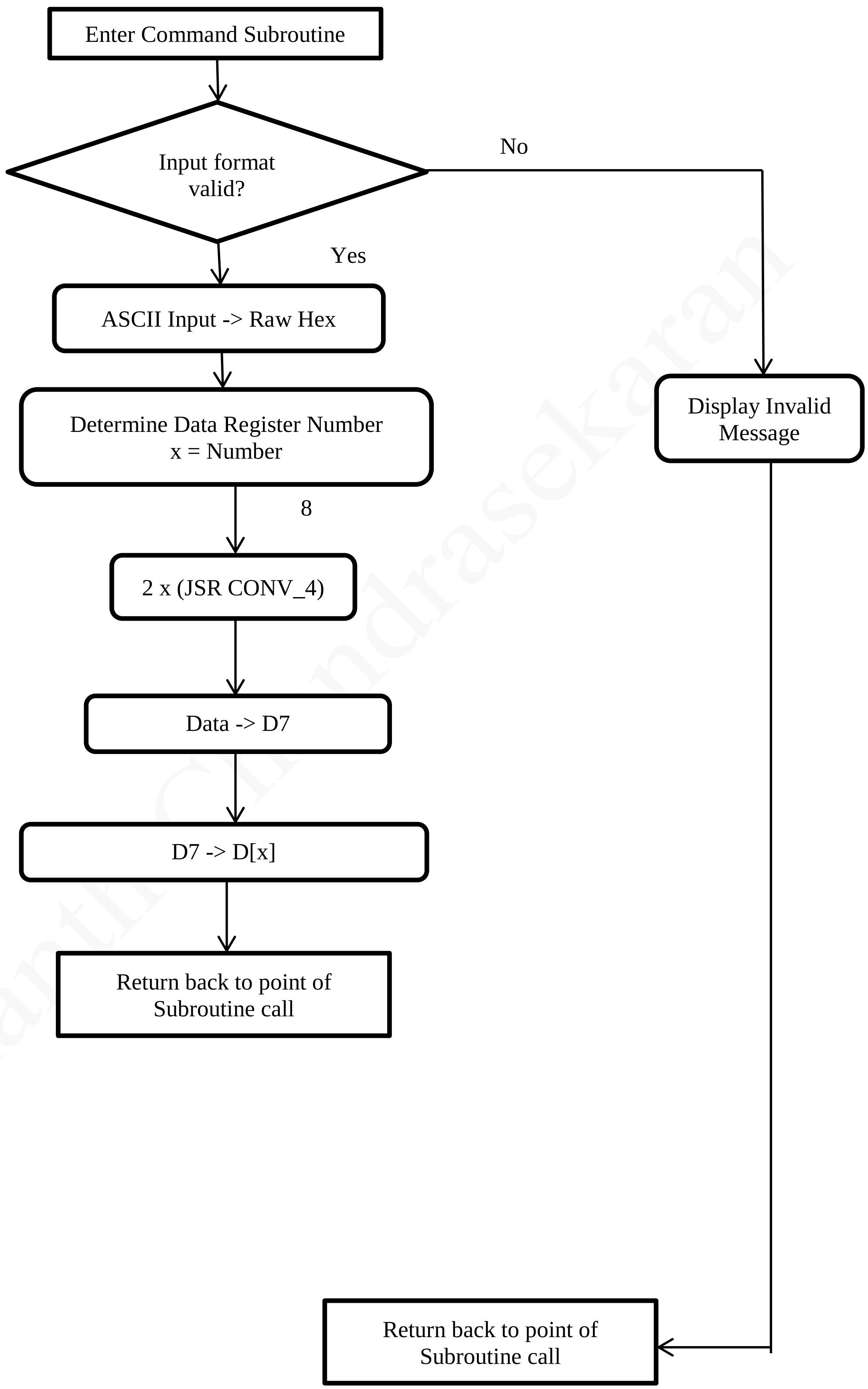
The **R**egister **M**odify command is used to modify the contents held in the data registers. The user must enter the data register to be modified and the longword data to be put into it. The data format is not right justified. Hence, if the user wishes to set the data to 10_{16}, the user must enter 00000010 for the instruction to take effect. . A detailed description of the syntax to be followed can be found in the Users' Manual.



Example usage of the Register Modify command

2.2.11.1-) Algorithm and Flowchart

begin
Check input format.
        If correct
                do command
        else
                display invalid message and wait for new input
    command:
    Determine the data register number
    Convert ASCII input value of data to raw hex values
    Use appropriate conversion block to obtain the hex data
    Put the data into the specified data register
end



Prashanth Chandrasekaran

2.2.11.2-) Assembly Code

```
RM:
    ADDQ #3,A1
    CMPI.B #$2E,(A1)+
    BEQ RM_NEXT
    JMP RM_INVALID
RM_NEXT:    CMPI.B #$44,(A1)
    BEQ DRM
    JMP RM_INVALID
DRM:  ADDQ #1,A1
    CMPI.B #$30,(A1)
    BEQ D0RM
    CMPI.B #$31,(A1)
    BEQ D1RM
    CMPI.B #$32,(A1)
    BEQ D2RM
    CMPI.B #$33,(A1)
    BEQ D3RM
    CMPI.B #$34,(A1)
    BEQ D4RM
    CMPI.B #$35,(A1)
    BEQ D5RM
    CMPI.B #$36,(A1)
    BEQ D6RM
    CMPI.B #$37,(A1)
    BEQ D7RM
    CMPI.B #$38,(A1)
    BGE RM_INVALID
D0RM:   ADDQ #2,A1
    JSR GET_DATA
    SUBQ #8,A1
    JSR CONV_4
    MOVE.W D7,D6
    SWAP.W D6
    ADDQ #4,A1
    JSR CONV_4
    CLR.W D6
    MOVE.W D7,D6
    MOVE.L D6,D0
    JMP RM_END
D1RM:   ADDQ #2,A1
    JSR GET_DATA
    SUBQ #8,A1
    JSR CONV_4
    MOVE.W D7,D6
    SWAP.W D6
    ADDQ #4,A1
    JSR CONV_4
    CLR.W D6
    MOVE.W D7,D6
    MOVE.L D6,D1
    JMP RM_END
D2RM:   ADDQ #2,A1
    JSR GET_DATA
    SUBQ #8,A1
```

Prashanth Chandrasekaran

```
    JSR CONV_4
    MOVE.W D7,D6
    SWAP.W D6
    ADDQ #4,A1
    JSR CONV_4
    CLR.W D6
    MOVE.W D7,D6
    MOVE.L D6,D2
    JMP RM_END
D3RM:   ADDQ #2,A1
    JSR GET_DATA
    SUBQ #8,A1
    JSR CONV_4
    MOVE.W D7,D6
    SWAP.W D6
    ADDQ #4,A1
    JSR CONV_4
    CLR.W D6
    MOVE.W D7,D6
    MOVE.L D6,D3
    JMP RM_END
D4RM:   ADDQ #2,A1
    JSR GET_DATA
    SUBQ #8,A1
    JSR CONV_4
    MOVE.W D7,D6
    SWAP.W D6
    ADDQ #4,A1
    JSR CONV_4
    CLR.W D6
    MOVE.W D7,D6
    MOVE.L D6,D4
    JMP RM_END
D5RM:   ADDQ #2,A1
    JSR GET_DATA
    SUBQ #8,A1
    JSR CONV_4
    MOVE.W D7,D6
    SWAP.W D6
    ADDQ #4,A1
    JSR CONV_4
    CLR.W D6
    MOVE.W D7,D6
    MOVE.L D6,D5
    JMP RM_END
D6RM:   ADDQ #2,A1
    JSR GET_DATA
    SUBQ #8,A1
    JSR CONV_4
    MOVE.W D7,D6
    SWAP.W D6
    ADDQ #4,A1
    JSR CONV_4
    CLR.W D6
    MOVE.W D7,D6
    JMP RM_END
D7RM:   ADDQ #2,A1
```

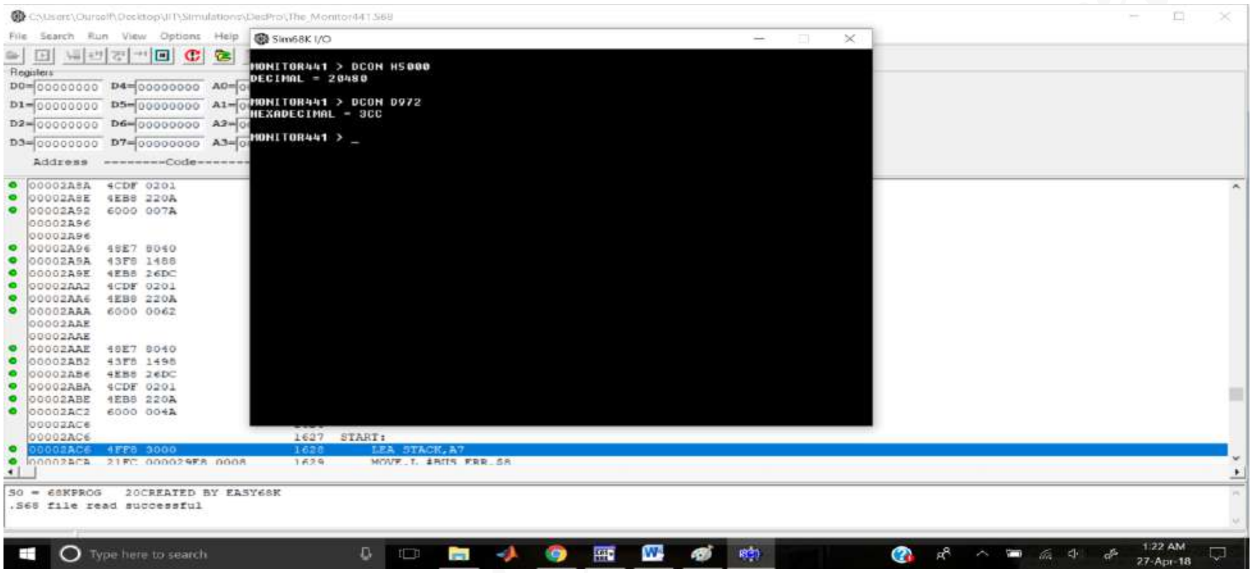
Prashanth Chandrasekaran

```
    JSR GET_DATA
    SUBQ #8,A1
    JSR CONV_4
    MOVE.W D7,D6
    SWAP.W D6
    ADDQ #4,A1
    JSR CONV_4
    CLR.W D6
    MOVE.W D7,D6
    MOVE.L D6,D7
    JMP RM_END
RM_INVALID: LEA INVALID_MSG,A1
    JSR DISPCR
RM_END: RTS
```

Prashanth Chandrasekaran

2.2.12-) Debugger Command # 12 – DCON (Data Conversion)

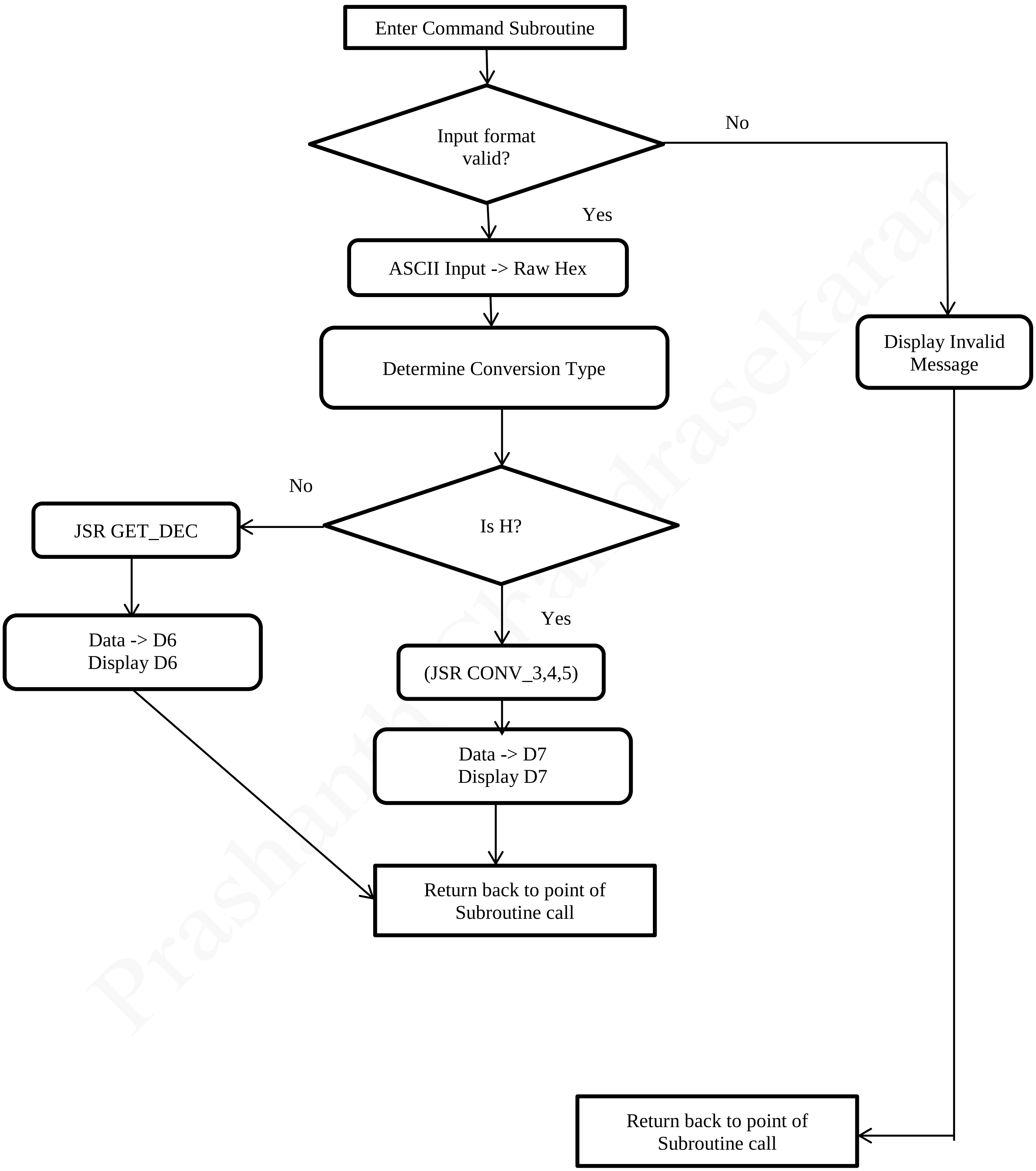
The **D**ata **CON**version command allows for conversion of hexadecimal data its decimal equivalent and the also for the conversion of decimal data to its hexadecimal equivalent. If **H** precedes the data, a Hex-to-Decimal conversion shall be performed. If **D** precedes the data, a Decimal-to-Hex conversion shall be performed. The data format is not right justified, so the user must be careful while providing inputs to this command. The command MUST also be terminated with a SPACE for it to function properly A detailed description of the syntax to be followed can be found in the Users' Manual.



Example usage of the Data Conversion command

2.2.12.1-) Algorithm and Flowchart

begin
Check input format.
        If correct
            do command
        else
            display invalid message and wait for new input
    command:
        Determine conversion type
        if (hex-to-dec)
            Convert ASCII input value of data to raw hex values
            Use appropriate conversion block to obtain the hex data
            Display to user
        else if (dec-to-hex)
            Convert ASCII input value of data to raw decimal values
            Use appropriate conversion block to obtain the decimal data
            Display to user
end

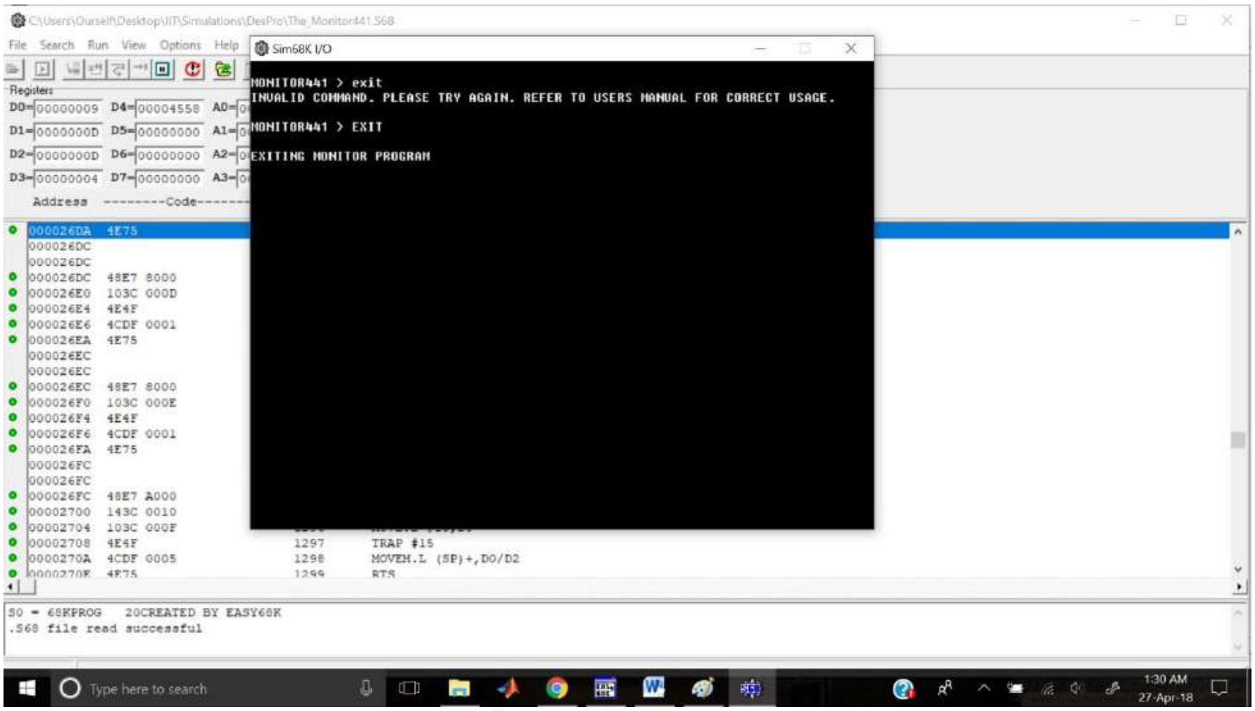

Enter Command Subroutine
Input format valid?
No
Yes
ASCII Input -> Raw Hex
Display Invalid Message
Determine Conversion Type
No
Is H?
JSR GET_DEC
Data -> D6
Display D6
Yes
(JSR CONV_3,4,5)
Data -> D7
Display D7
Return back to point of Subroutine call
Return back to point of Subroutine call

2.2.12.2-) Assembly Code

```
DCON:
    MOVEM.L D0-D7/A0-A6,-(SP)
    MOVEQ #0,D4
    ADDQ #5,A1
    CMPI.B #$44,(A1)
    BEQ DECTOHEX
    CMPI.B #$48,(A1)
    BEQ HEXTODEC
    JMP DCON_INVALID
DECTOHEX:   ADDQ #1,A1
    JSR GET_DEC
    LEA DCONMSG_2,A1
    JSR DISP
    MOVE.L D6,D1
    JSR DISPDA
    LEA SPACE,A1
    JSR DISPCR
    JMP DCON_END
HEXTODEC:   ADDQ #1,A1
    JSR GET_DATA
    CMPI.B #$03,D2
    BEQ HEX2
    CMPI.B #$04,D2
    BEQ HEX3
    CMPI.B #$05,D2
    BEQ HEX4
    CMPI.B #$06,D2
    BEQ HEX5
HEX2:   SUB.L D2,A1
    JSR CONV_2
    MOVE.L D3,D7
    JMP DCON_DISPLAYD
HEX3:   SUB.L D2,A1
    JSR CONV_3
    MOVE.L D6,D7
    JMP DCON_DISPLAYD
HEX4:   SUB.L D2,A1
    JSR CONV_4
    JMP DCON_DISPLAYD
HEX5:   SUB.L D2,A1
    JSR CONV_5
DCON_DISPLAYD:   LEA DCONMSG_1,A1
    JSR DISP
    MOVE.L D7,D1
    MOVE.B #10,D2
    MOVE.B #15,D0
    TRAP #15
    LEA SPACE,A1
    JSR DISPCR
    JMP DCON_END
DCON_INVALID:   LEA INVALID_MSG,A1
    JSR DISPCR
DCON_END:   MOVEM.L (SP)+,D0-D7/A0-A6
    RTS
```

2.2.13-) Debugger Command # 14 – EXIT

The **EXIT** command stops the Monitor Program execution. Once this instruction is executed, the user will not be able to execute any further commands. The Monitor Program can only be run again by closing out of the simulator and restarting the entire simulation.



Example usage of the EXIT command

2.2.13.1-) Assembly Code

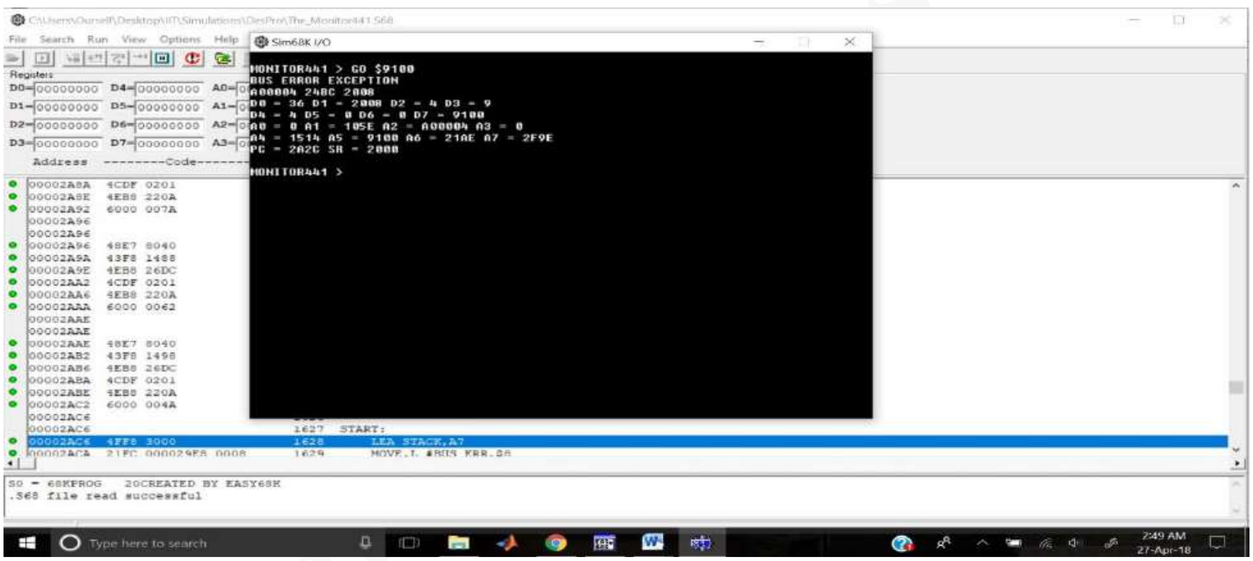
```
EXIT:
    LEA SPACE,A1
    JSR DISPCR
    LEA EXITMSG,A1
    JSR DISP
    MOVE.B #9,D0
    TRAP #15
    RTS
```

2.3-) Exception Handlers

The Monitor Program handles all the exceptions that can arise from its usage with customized Interrupt Service Routines. The Interrupt Service Routines provide the user with an error message and also displays the values of the registers at the time that the exception occurred. Address and Bus error service routines provide additional information to aide the user in correcting their code and avoid exceptions from further occurring. The addresses of the customized service routines are initialized at the program start.

2.3.1-) Bus Error Exception

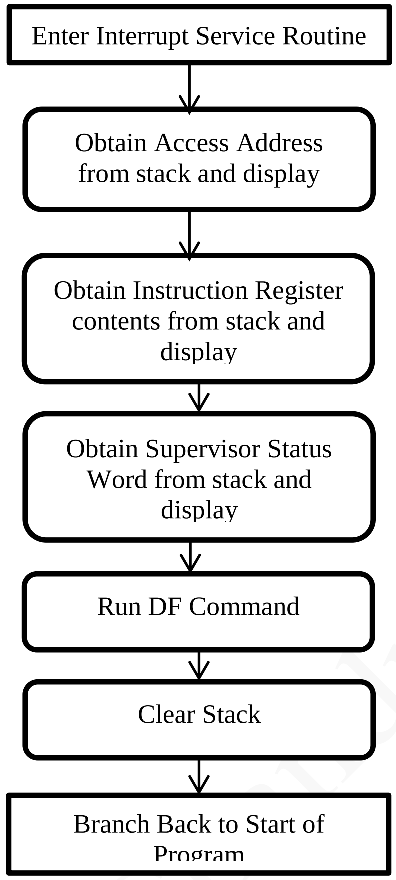
A Bus Error Exception will occur if the user tries to perform a read or write access in parts of memory that are either deemed invalid or if such an address doesn’t exist on the system. The ISR to handle a Bus Error Exception displays an error message, the Supervisor Status Word (SSW), the contents in the Instruction Register (IR) and the Access Address.



Example output of a Bus Error Exception

2.3.1.1-) Algorithm and Flowchart

begin
Display Error Message
Obtain Supervisor Status Word from Stack
Display to user
Obtain Instruction Register from Stack
Display to user
Obtain Accces Address from Stack
Display to user
do DF command
end



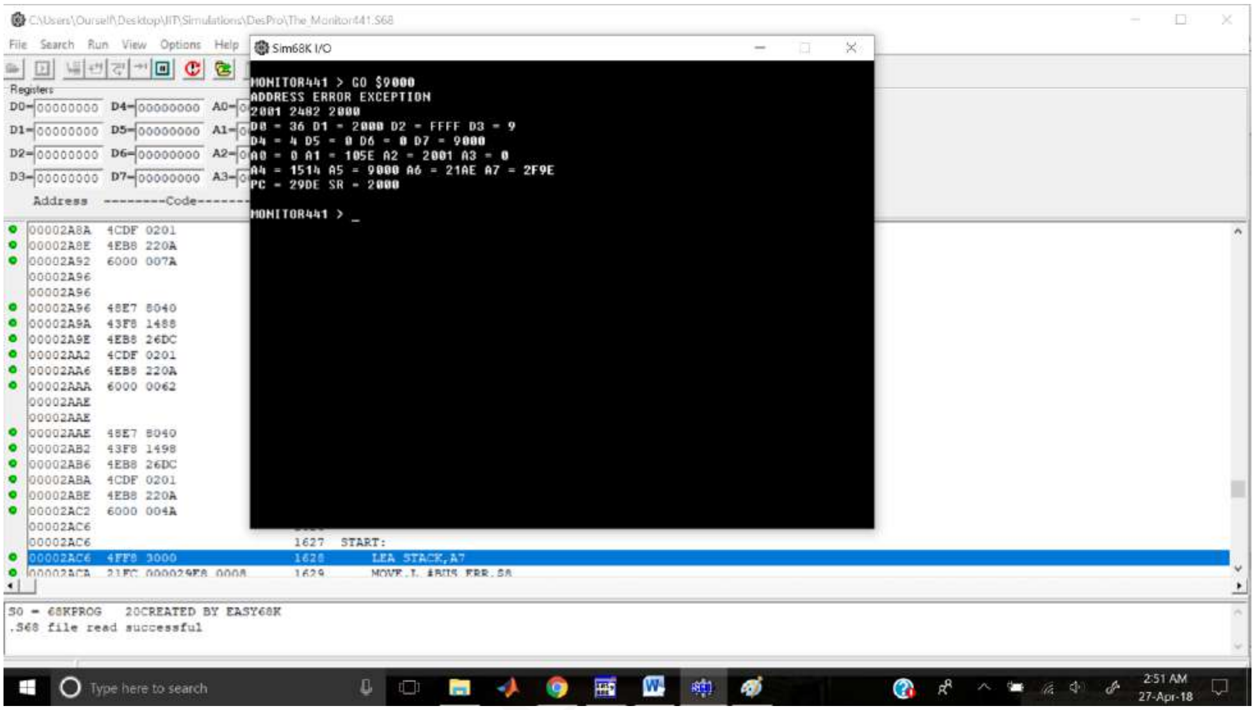
2.3.1.2-) Assembly Code

```
BUS_ERR:
    MOVEM.L D0-D2/A1,-(SP)
    LEA BERR_MSG,A1
    JSR DISPCR
    MOVE.L (18,SP),D1                // Access Address
    JSR DISPDA
    LEA SPACE,A1
    JSR DISP
    CLR.L D1
    MOVE.W (22,SP),D1                // Instruction Register
    JSR DISPDA
    LEA SPACE,A1
    JSR DISP
    CLR.L D1
    MOVE.W (24,SP),D1                // Supervisor Status Word
    JSR DISPDA
    LEA SPACE,A1
    JSR DISPCR
    JSR DF
    ADD.L #$1E,SP
    BRA MAIN
```

Prashanth Chandrasekaran

2.3.2-) Address Error Exception

An Address Error Exception will occur if the user tries to perform a word or longword memory access from an odd location. The ISR to handle an Address Error Exception displays an error message, the Supervisor Status Word (SSW), the contents in the Instruction Register (IR) and the Access Address.



Example output of an Address Error Exception

2.3.2.1-) Algorithm and Flowchart

begin

Display Error Message

Obtain Supervisor Status Word from Stack

Display to user

Obtain Instruction Register from Stack

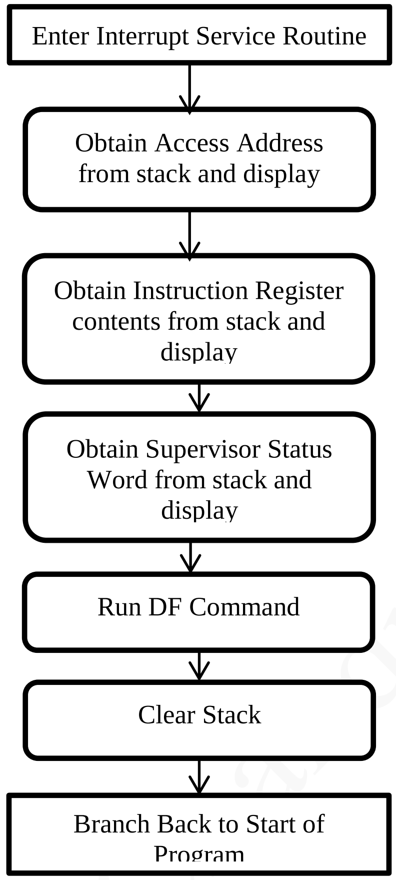
Display to user

Obtain Accces Address from Stack

Display to user

do DF command

end

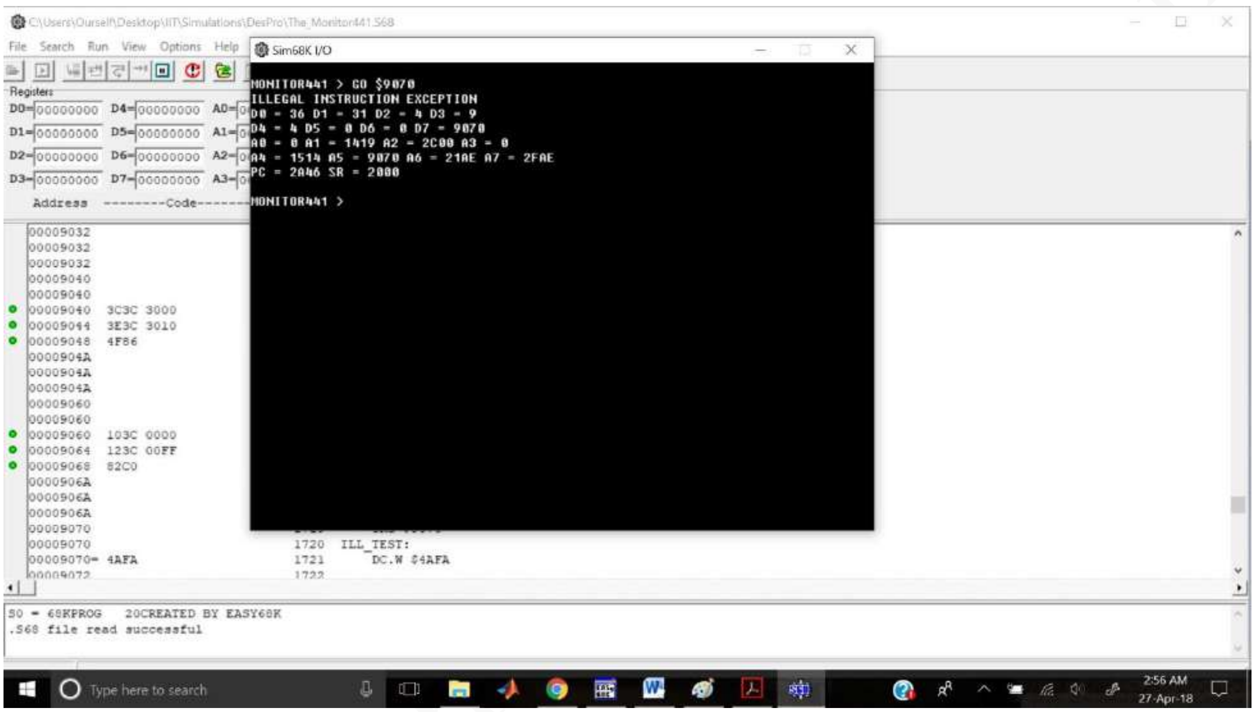


2.3.2.2-) Assembly Code

```
BUS_ERR:
    MOVEM.L D0-D2/A1,-(SP)
    LEA BERR_MSG,A1
    JSR DISPCR
    MOVE.L (18,SP),D1
    JSR DISPDA
    LEA SPACE,A1
    JSR DISP
    CLR.L D1
    MOVE.W (22,SP),D1
    JSR DISPDA
    LEA SPACE,A1
    JSR DISP
    CLR.L D1
    MOVE.W (24,SP),D1
    JSR DISPDA
    LEA SPACE,A1
    JSR DISPCR
    JSR DF
    ADD.L #$1E,SP
    BRA MAIN
```

2.3.3-) Illegal Instruction Exception

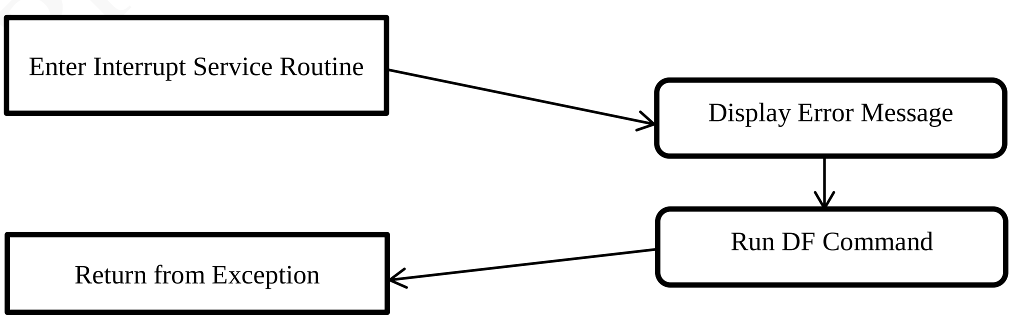
Illegal instruction is the term used to refer to any of the word bit patterns that do not match the bit pattern of the first word of a legal MC68000 instruction. If such an instruction is fetched, an illegal instruction exception occurs. In the MC68000, the fetching of $4AFA, $4AFB AND $4AFC results in an illegal instruction exception. The Service Routine displays an error message and the values of the registers at the time of exception.



Example output of an Illegal Instruction Exception

2.3.3.1-) Algorithm and Flowchart

begin
  Display Error Message
  do DF command
end

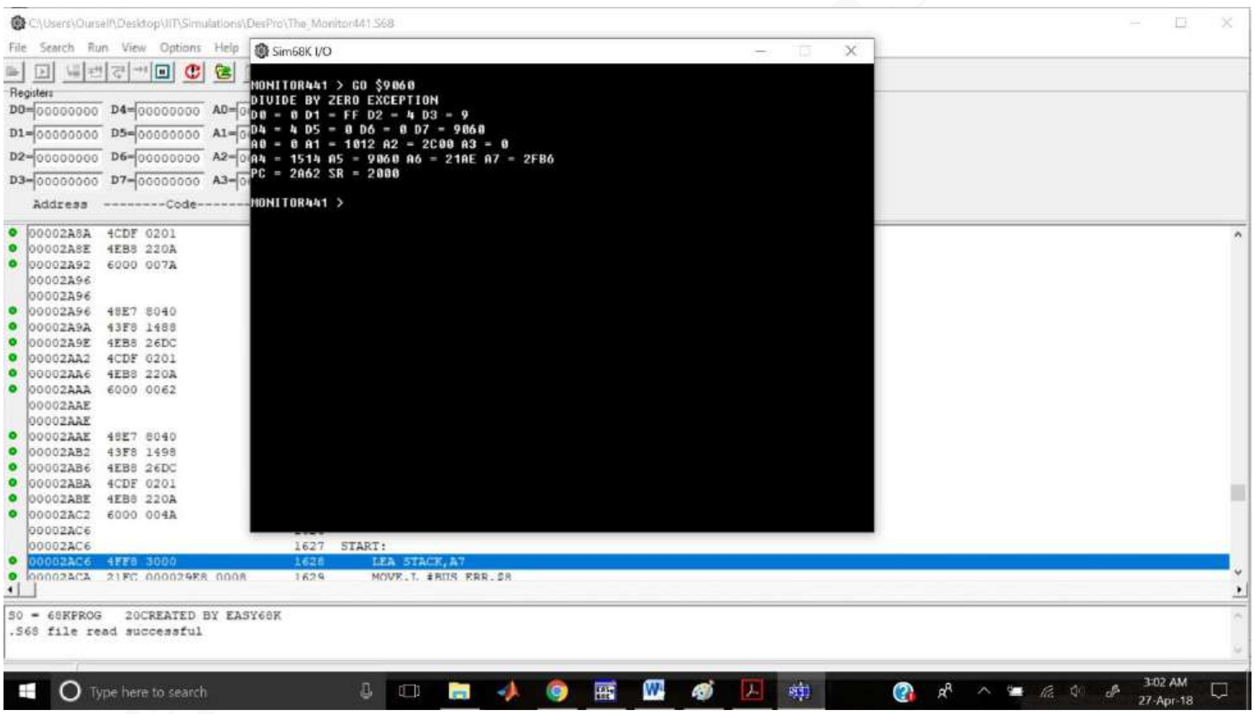


2.3.3.2-) Assembly Code

```
IL_INST:
    MOVEM.L D0/A1,-(SP)
    LEA ILLERR_MSG,A1
    JSR DISPCR
    JSR DF
    MOVEM.L (SP)+,D0/A1
    BRA MAIN
```

2.3.4-) Divide-by- Zero Exception

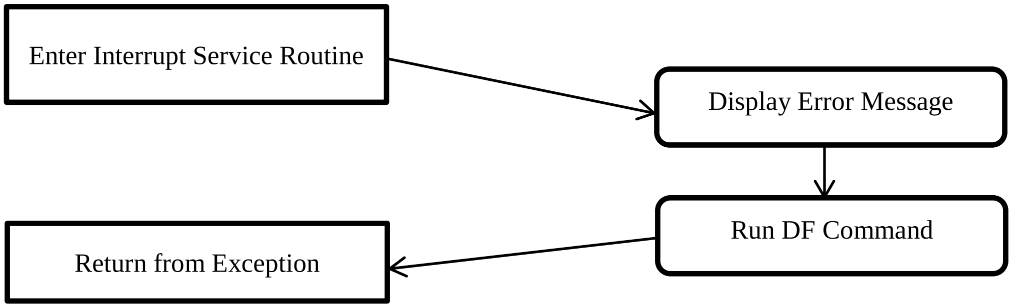
A Divide-by-Zero exception occurs if the user tries to perform a division operation with the divisor as a zero. The Service Routine displays an error message and the values of the registers at the time of exception.



Example output of a Divide-by-Zero Exception

2.3.4.1-) Algorithm and Flowchart

begin
    Display Error Message
    do DF command
end

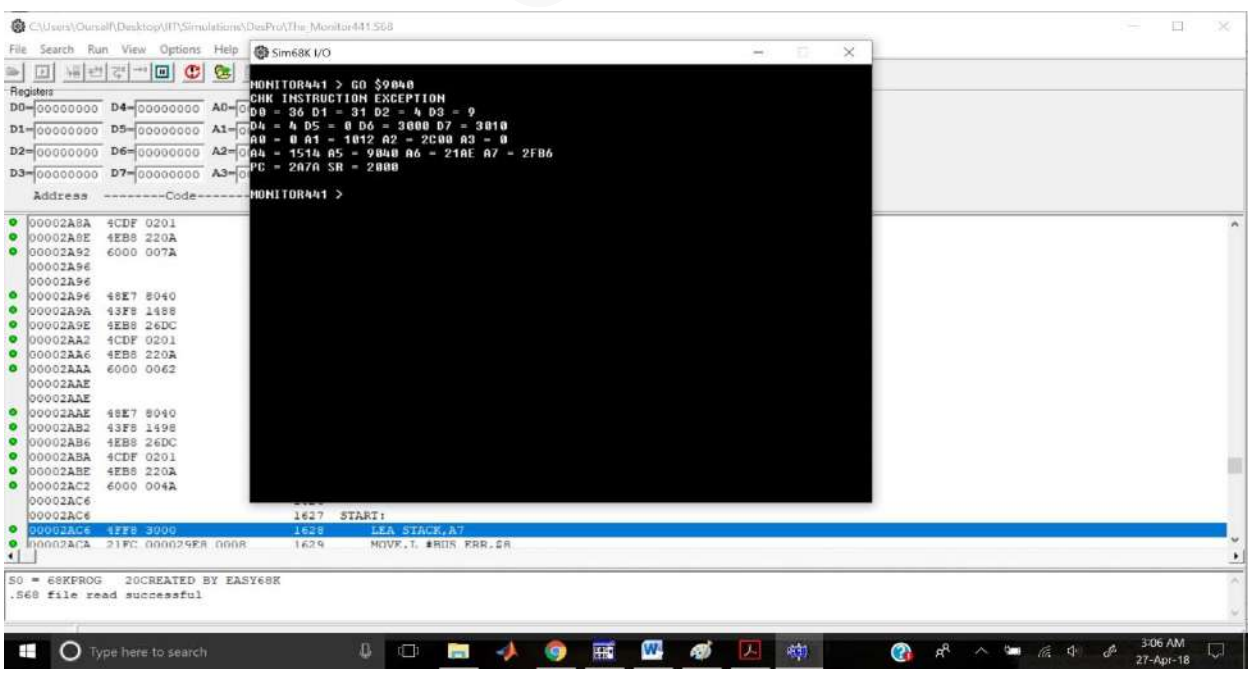


2.3.4.2-) Assembly Code

```
DIV_Z:
    MOVEM.L D0/A1,-(SP)
    LEA DIVZERR_MSG,A1
    JSR DISPCR
    MOVEM.L (SP)+,D0/A1
    JSR DF
    BRA MAIN
```

2.3.5-) CHK Instruction Exception

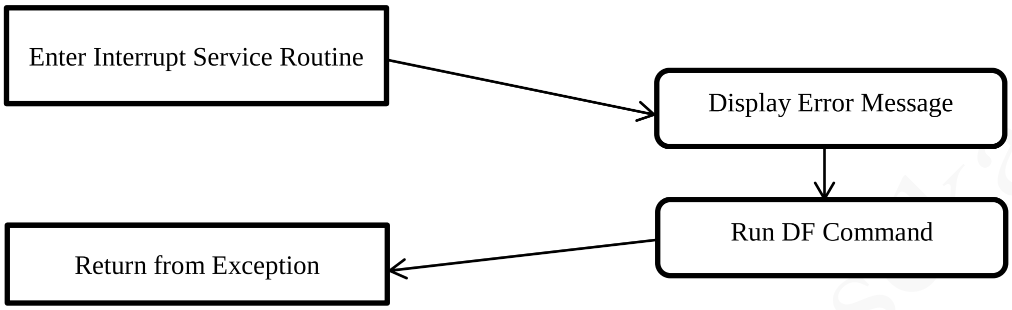
The CHK Instruction compares the value in the data register specified in the instruction to zero and to an upper bound value. If the register value is less than zero or greater than the upper bound, a CHK instruction exception occurs. The Service Routine displays an error message and the values of the registers at the time of exception.



Example output of a CHK Instruction Exception

2.3.5.1-) Algorithm and Flowchart

begin
    Display Error Message
    do DF command
end

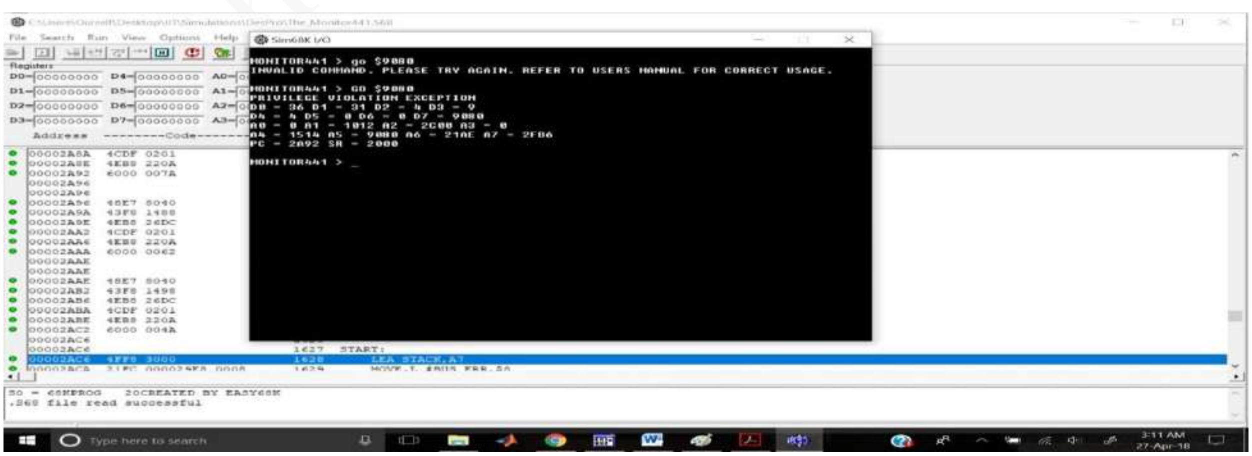


2.3.5.2-) Assembly Code

```
CHK_INST:
    MOVEM.L D0/A1,-(SP)
    LEA CHKERR_MSG,A1
    JSR DISPCR
    MOVEM.L (SP)+,D0/A1
    JSR DF
    BRA MAIN
```

2.3.6-) Privilege Violation Exception

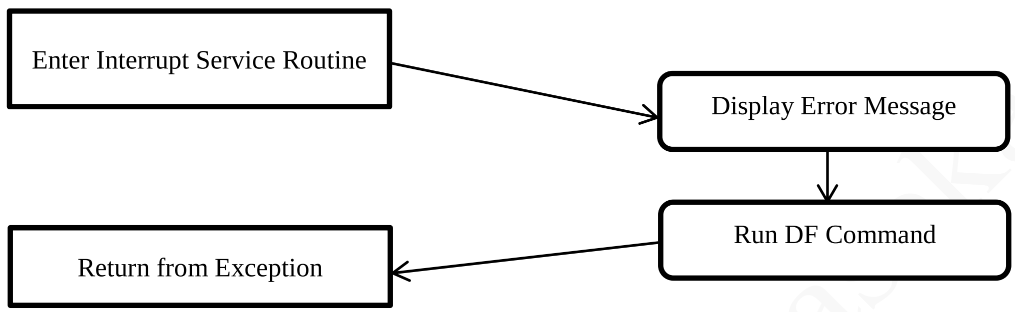
To provide system security, various instructions are privileged. An attempt to execute one of the privileged instructions while in the user mode causes an exception. The Service Routine displays an error message and the values of the registers at the time of exception.



Example output of a Privilege Violation Exception

2.3.6.1-) Algorithm and Flowchart

begin
    Display Error Message
    do DF command
end



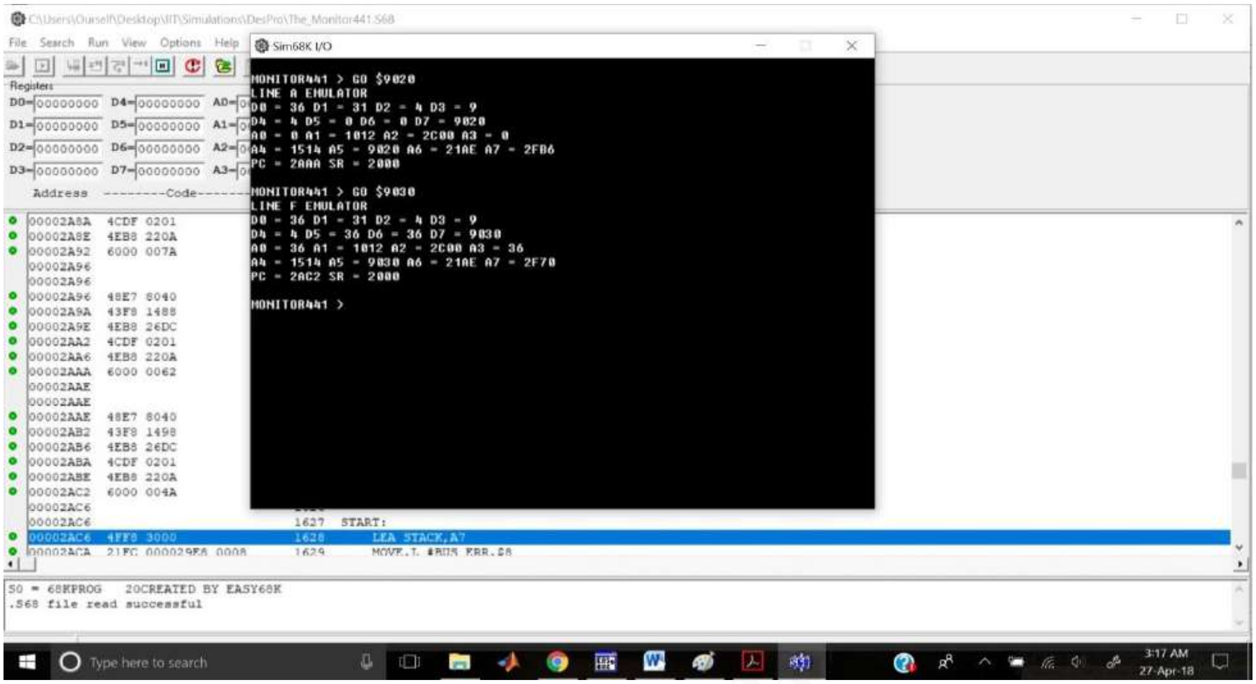
2.3.6.2-) Assembly Code

```
PRI_VIO:
    MOVEM.L D0/A1,-(SP)
    LEA PRIVERR_MSG,A1
    JSR DISPCR
    MOVEM.L (SP)+,D0/A1
    JSR DF
    BRA MAIN
```

2.3.7-) Line A and Line F Emulator Exceptions

Word patterns with bits 15–12 equaling 1010 (LINE A) or 1111 (LINE F) are distinguished as unimplemented instructions, and separate exception vectors are assigned to these patterns to permit efficient emulation. The Service Routine displays an error message and the values of the registers at the time of exception.

Prashanth Chandrasekaran

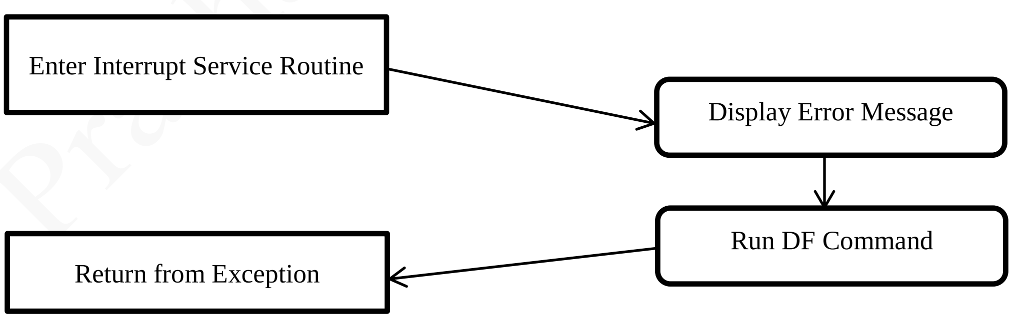


Example output of Line A and Line F Emulator Exceptions

2.3.7.1-) Algorithm and Flowchart

Both these exceptions follow the same algorithm as listed below

begin
    Display Error Message
    do DF command
end



2.3.7.2-) Assembly Code

```
LINE_A:
    MOVEM.L D0/A1,-(SP)
    LEA LINEAERR_MSG,A1
    JSR DISPCR
    MOVEM.L (SP)+,D0/A1
    JSR DF
    BRA MAIN

LINE_F:
    MOVEM.L D0/A1,-(SP)
    LEA LINEFERR_MSG,A1
    JSR DISPCR
    MOVEM.L (SP)+,D0/A1
    JSR DF
    BRA MAIN
```

Prashanth Chandrasekaran

2.4-) User Instruction Manual

This section covers all the commands together with their syntax for proper usage and examples of their usage.

HELP
DISPLAYS THE HELP MESSAGE. DISPLAYS THE AVAILABLE COMMANDS AND THEIR BRIEF USAGE DESCRIPTIONS.

MDSP <ADDRESS 1> <SPACE> OR
MDSP <ADDRESS 1> <ADDRESS 2> <SPACE>

DISPLAYS THE ADDRESSES AND MEMORY CONTENTS STORED AT THE ADDRESSES. COMMAND MUST BE TERMINATED WITH A SPACE.
IF THE USER ENTERS A SINGLE ADDRESS, THE COMMAND DISPLAYS MEMORY CONTENTS UP UNITL 16 LOCATIONS AHEAD.
IF THE USER ENTERS TWO ADDRESSES, THE COMMAND DISPLAYS MEMORY CONTENTS WITHIN THE RANGE SPECIFIED.

EXAMPLE:
MDSP $900 <space>
MDSP $1000 $102A <space>

SORTW <ADDRESS 1(START)> <ADDRESS 2(END)> <A/D>

SORTS WORD THAT FALL WITHIN THE RANGE OF LOCATIONS SPECIFIED. BOTH ADDRESSES MUST BE EVEN TO AVOID ADDRESSING ERRORS.
THE LETTER **A** OR **D** DETERMINES THE ORDER OF SORTING

EXAMPLE:
SORTW $5000 $5040 A

MM <ADDRESS> ;<BYTE (B), WORD(W) OR LONG(L)>

DISPLAYS THE ADDRESS AND MEMORY CONTENTS OF THE LOCATION SPECIFIED IN EITHER BYTES, WORD OR LONGWORDS AS SPECIFED BY THE USER. IT IS TERMINATED USING ".".
WORD AND LONGWORD ACCESSES CAN ONLY BE PERFORMED FROM **EVEN** ADDRESSES

EXAMPLE:
MM $4503 ;B
MM $6000 ;L

Prashanth Chandrasekaran

MS <ADDRESS> <BYTE, WORD, LONGWORD OR ASCII STRING DATA>

SETS THE MEMORY LOCATION WITH THE DATA SPECIFIED BY THE USER. SUPPORTS BYTE, WORD, LONGWORD AND ASCII STRING INPUTS.

EXAMPLE:
MS $750 AA
MS $9000 1234ABCD
MS $B100 ECE-441 MONITOR PROJECT

BF <ADDRESS 1(START)> <ADDRESS 2(END)> <WORD SIZE DATA>

FILL THE BLOCK OF MEMORY WITHIN THE ADDRESSES SPECIFIED BY THE USER. BOTH MUST BE **EVEN** ADDRESSES.
WORD SIZE DATA IS TO BE ENTERED AS 4 HEXADECIMAL DIGITS. NO RIGHT JUSTIFICATION IS PROVIDED SO CARE MUST BE TAKEN WHILE PROVIDING ARGUMENTS TO THIS COMMAND.

EXAMPLE:
BF $5000 $5040 ABCD
BF $800 $810 1234
BF $9010 $9020 0007

BMOV <ADDRESS 1(CURRENT LOC)> <ADDRESS 2(NEW LOC)> <NO_OF_BYTES>

MOVES A BLOCK OF MEMORY FROM ONE LOCATION TO ANOTHER.
SIZE OF THE BLOCK IS DETERMINED BY THE NO_OF_BYTES ARGUMENT.
CAN TRANSFER UPTO A MAXIMUM OF 999 BYTES. NO RIGHT JUSTIFICATION IS PROVIDED SO CARE MUST BE TAKEN WHILE PROVIDING NO_OF_BYTES TO THIS COMMAND.

EXAMPLE:
BMOV $7000 $8000 200
BMOV $800 $810 064
BMOV $9010 $9020 007

BTST <ADDRESS 1(START)> <ADDRESS 2(END)> <SPACE>

PERFORMS A DESTRUCTIVE TEST ON A BLOCK OF MEMORY WHOSE RANGE IS SPECIFIED BY THE USER.
IF PASSED, IT WRITES **ZEROS** INTO ALL LOCATIONS WITHIN THE BLOCK

Prashanth Chandrasekaran

EXAMPLE:
BTST $7000 $7600 <space>
BTST $A00 $A20 <space>

BSCH <ADDRESS 1(START)> <ADDRESS 2(END)> <ASCII STRING>

SEARCHES FOR A LITERAL STRING INPUT BY THE USER WITHIN THE BLOCK OF MEMORY AS SPECIFIED BY THE USER INPUTS.

EXAMPLE:
BSCH $B0F0 $B120 ECE-441 MONITOR PROJECT

GO <ADDRESS> <SPACE>

USED TO RUN USER PROGRAMS. THE STARTING ADDRESS OF THE USER PROGRAM IS MUST BE GIVEN AS AN INPUT BY THE USER.
COMMAND MUST BE TERMINATED WITH A SPACE.

EXAMPLE:
GO $9200 <space>

RM .<DATA REGISTER (D0-D7)> <LONGWORD DATA>

USED TO MODIFY THE VALUES IN THE DATA REGISTER.
ONLY ACCEPTS LONG WORD DATA. INPUT DATA FORMAT IS NOT ROGHT JUSTIFIED SO CARE MUST BE TAKEN WHILE PROVIDING ARGUMENTS TO THIS COMMAND.

EXAMPLE:
RM .D3 ABCD1234
RM .D7 00000007

DCON H<HEX-DATA> OR DCON D<DEC-DATA>

USED TO PERFORM EITHER HEXADECIMAL TO DECIMAL OR DECIMAL TO HEXADECIMAL DATA CONVERSION.
SUPPORTS UPTO 5-DIGIT HEXADECIMAL DATA AND 3 DIGIT DECIMAL DATA.
NO RIGHT JUSTIFICATION IS PROVIDED SO CARE MUST BE TAKEN WHILE PROVIDING ARGUMENTS TO THIS COMMAND.

EXAMPLE:
DCON H1000
DCON D256

Prashanth Chandrasekaran

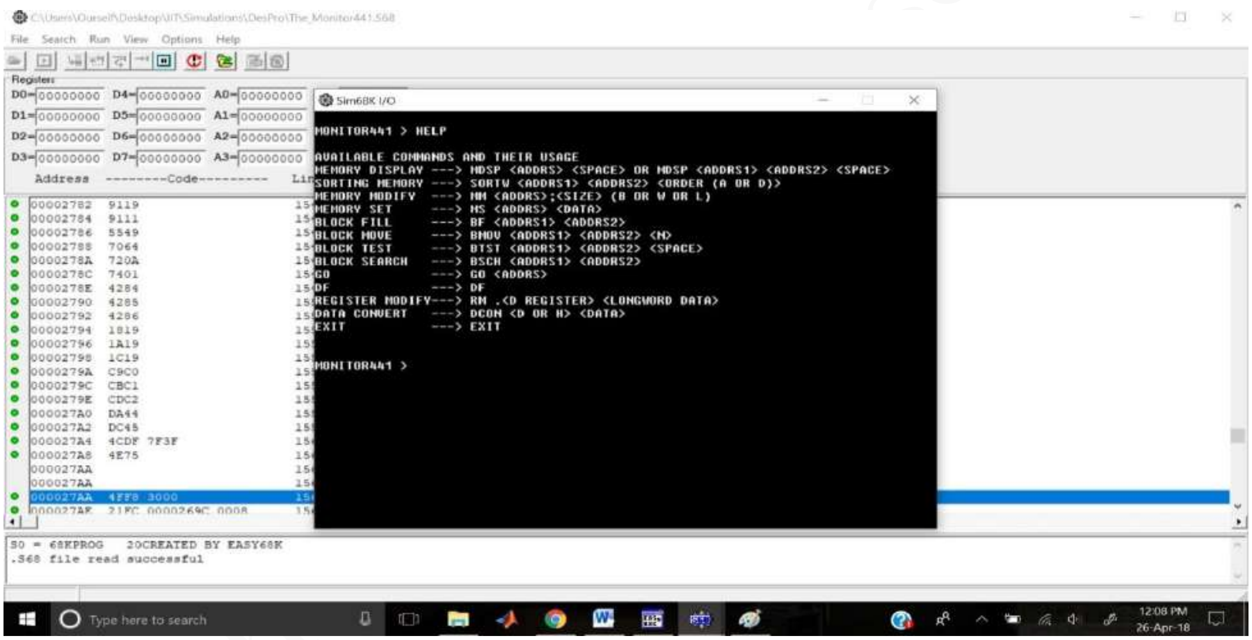
EXIT

USED TO EXIT FROM THE MONITOR PROGRAM. EQUIVALENT TO QUITTING TUTOR ON A PC.
TO RUN THE MONITOR PROGRAM AGAIN, RESTART THE ENTIRE SIMULATOR.

EXAMPLE:
EXIT

2.4.1-) HELP Command

The HELP command displays all the available commands and their brief usage descriptions. The commands are displayed along with the syntax to be followed for their usage. A more detailed description regarding the syntax along with correct usage examples are provided in the Users' Manual.



Examples usage of Help command

2.4.1.1-) Algorithm and Flowchart

begin

push registers into stack

Display Message for Command 1
Display Message for Command 2
Display Message for Command 3
Display Message for Command 4
Display Message for Command 5
Display Message for Command 6

Display Message for Command 7
Display Message for Command 8
Display Message for Command 9
Display Message for Command 10
Display Message for Command 11
Display Message for Command 12
Display Message for Command 13
Display Message for Command 14
pop registers from stack
return back to main program and wait for user input

2.4.1.2-) Assembly Code

```
HELP:
    MOVEM.L D0/A1,-(SP)
    LEA SPACE,A1
    JSR DISPCR
    LEA HELPMSG_1,A1
    JSR DISPCR
    LEA HELPMSG_2,A1
    JSR DISPCR
    LEA HELPMSG_3,A1
    JSR DISPCR
    LEA HELPMSG_4,A1
    JSR DISPCR
    LEA HELPMSG_5,A1
    JSR DISPCR
    LEA HELPMSG_6,A1
    JSR DISPCR
    LEA HELPMSG_7,A1
    JSR DISPCR
    LEA HELPMSG_8,A1
    JSR DISPCR
    LEA HELPMSG_9,A1
    JSR DISPCR
    LEA HELPMSG_10,A1
    JSR DISPCR
    LEA HELPMSG_11,A1
    JSR DISPCR
    LEA HELPMSG_12,A1
    JSR DISPCR
    LEA HELPMSG_13,A1
    JSR DISPCR
    LEA HELPMSG_14,A1
    JSR DISPCR
    LEA SPACE,A1
    JSR DISPCR
    MOVEM.L (SP)+,D0/A1
    RTS
```


Prashanth Chandrasekaran

3-) Discussion

The design and development of the program code to obtain a fully functioning Monitor Program, albeit with a few bugs, was not performed without encountering a fair share of challenges. The first hurdle was the design of the command interpreter. Clever use of the registers was necessary in order to facilitate the use of address-register indirect with displacement addressing mode. Next, the implementation of each command required extensive testing to ensure that each command performed its functionality as desired with properly formatted outputs an error messages before it could be imported into the main program. Another challenge was to try to maximize the address space upon which the Monitor Program could operate. Each command has a block of code in its subroutine that is dedicated to maximizing the address space utilization. The final challenge was to get all the commands and Interrupt Service Routines to gel perfectly and not result in an unnecessary halts or errors.

4-)Feature Suggestions

- Perform right justification of the input data wherever necessary. For example, in the Block Fill command, if the user wants to fill the block of memory with 7_{16}, they must type out the command as

  BMOV $3000 $3050 0007

- At present, commands that use addresses and data inputs aren't "no_of_digits" compatible. For example, if a user wants to use MDSP from location $FF0_{16}$ to 1020_{16}, they must type out the command as

  MDSP $0FF0 $1020 <space>

Making these commands "no_of_digits" compatible would make the users' experience of the Monitor Program easier.

- At present, the commands can operate on addresses in the range $00000 - $FFFFF. An added feature would be to increase the memory space in which the Monitor Program can operate.

- The Register Modify command can be expanded to include Address Registers as well.

- The Data Conversion command can be expanded upon to perform a wide variety of base conversions. Increasing the range of data that can be used and introducing Octal and Binary forms in this command would essentially transform this command into a programmer's calculator.

- Certain commands require a <SPACE> for its termination, else its functionality is compromised. At present, this bug is unavoidable due to the ASCII-to-Hex decoding logic employed in the design and development of the program code. Further work can be directed towards the removal of this bug.

Prashanth Chandrasekaran

- Add in the ability to set breakpoints. This will make the users' work much easier when they wish to develop their own code as it will aid in easy and efficient debugging.

- To take it a step further, a Checksum type feature can be included in the Monitor Program as a means of error detection.

5-) Conclusions

The Monitor Program designed performs all the required commands in an almost perfect manner. There are still a few bugs and glitches that need ironing out; but with a few changes and additions, the program will run as smooth as the TUTOR software that it wishes to emulate.

6-) References

[1] *MC68000 Educational Computer Board Users' Manual*, 2nd ed. Motorola Inc, 1982.

[2] *MOTOROLA M68000 FAMILY Programmer's Reference Manual*. MOTOROLA, 1992.

[3] *M68000 8-/16-/32-bit microprocessors user's manual*, 9th ed. Phoenix, Ariz.: Motorola, 1994.

[4] 2018. [Online]. Available: http://research.cs.tamu.edu/prism/lectures/mbsd/mbsd_l9.pdf. [Accessed: 27- Apr- 2018].

[5] "EASy68K", *Easy68k.com*, 2018. [Online]. Available: http://www.easy68k.com/. [Accessed: 27- Apr- 2018].

[6] E. Balagurusamy, *Object Oriented Programming with C++*, 6th ed. Tata McGraw-Hill Education Pvt. Ltd, 2013.

Prashanth Chandrasekaran