ECE 441
Microprocessors

Instructor: Dr. Jafar Saniie
Teaching Assistant: Xinrui Yu

Final Project Report:
MONITOR PROJECT
11/30/2018

By: Michelle Yang

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 Monitor program is a way of allowing users to interface with the MC68000 microprocessor without writing any assembly. It allows the user to access and manipulate memory and register contents through a set of instructions entered at the terminal, including necessary error handling should the user enter a bad input. The program is implemented as 4 major pieces, each as a collection of subroutines: setup, command interpreter, individual command handlers, and individual exception handlers. Several helper subroutines were written to maximize reusability, reduce complexity, and increase readability of the total code.

The commands that are supported by the monitor program include displaying the help manual, memory display, memory sort, memory modify, memory set, block fill, block move (duplicate), block search, execute, display registers, echo, register modify, and exit. The exceptions handled are bus errors, address errors, illegal instruction errors, divide by zero errors, check errors, privilege violation errors, line A errors, and line F errors.

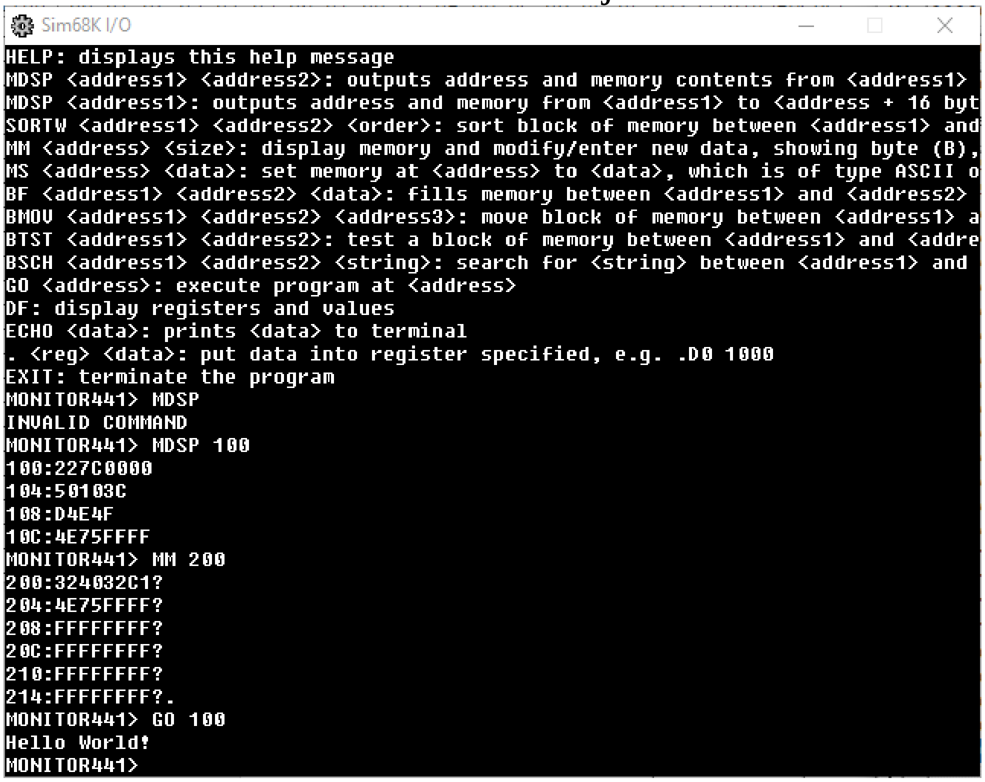
1-) Introduction

The Monitor program is a way of allowing users to interface with the MC68000 microprocessor without writing any assembly. It allows the user to access and manipulate memory and register contents through a set of instructions entered at the terminal, including necessary error handling should the user enter a bad input.

The entire program is written in MC68000 assembly, with each component written as a separate subroutine. The monitor program is implemented in 4 distinct pieces:

- Startup (1 subroutine)
- Command interpreter (1 subroutine)
- Individual command handlers (13 subroutines)
- Individual exception handlers (8 subroutines)

The program is implemented for the Easy68K simulator, which has some differences in syntax. However, the program can easily be converted to run on an actual MC68000 processor, such as one found in the SANPER-1 unit.



```
Sim68K I/O
HELP: displays this help message
MDSP <address1> <address2>: outputs address and memory contents from <address1>
MDSP <address1>: outputs address and memory from <address1> to <address + 16 byt
SORTW <address1> <address2> <order>: sort block of memory between <address1> and
MM <address> <size>: display memory and modify/enter new data, showing byte (B),
MS <address> <data>: set memory at <address> to <data>, which is of type ASCII o
BF <address1> <address2> <data>: fills memory between <address1> and <address2>
BMOV <address1> <address2> <address3>: move block of memory between <address1> a
BTST <address1> <address2>: test a block of memory between <address1> and <addre
BSCH <address1> <address2> <string>: search for <string> between <address1> and
GO <address>: execute program at <address>
DF: display registers and values
ECHO <data>: prints <data> to terminal
. <reg> <data>: put data into register specified, e.g. .D0 1000
EXIT: terminate the program
MONITOR441> MDSP
INVALID COMMAND
MONITOR441> MDSP 100
100:227C0000
104:50103C
108:D4E4F
10C:4E75FFFF
MONITOR441> MM 200
200:324032C1?
204:4E75FFFF?
208:FFFFFFFF?
20C:FFFFFFFF?
210:FFFFFFFF?
214:FFFFFFFF?.
MONITOR441> GO 100
Hello World!
MONITOR441>
```

Figure 1: Demo of monitor program

2-) Monitor Program

The design objectives are the following:
- Support basic debugger functions entered by human user from terminal
- Display an error message if the inputted command is invalid
- Notify the user of any exceptions by handling and printing exception state

The design constraints are the following:
- Entire code (including any hard coded data) must be less than 3KB in size
- Stack size should take up less than 1 KB
- No macros allowed
- Bad inputs should not crash the program but should print minimum number of error statements

The flow of the program is as follows:
1. The user starts the program.
2. The program does setup work, including saving existing state of registers and setting the exception handler location in the interrupt vector table.
3. The program sits and waits for the user to enter a command.
4. Once a command is entered, the command interpreter goes through a list of supported commands and checks if the input matches any of them. If not, the user is notified of the invalid input.
5. If yes, the parser calls the appropriate subroutine to handle the command, passing as an argument the full string that the user inputted at the terminal.
6. The subroutine performs the expected operation and passes control back to the terminal to wait for the next input.
7. If an exception occurs during processing, the appropriate exception handler is called. Once it completes, control is passed back to the terminal for the next input.

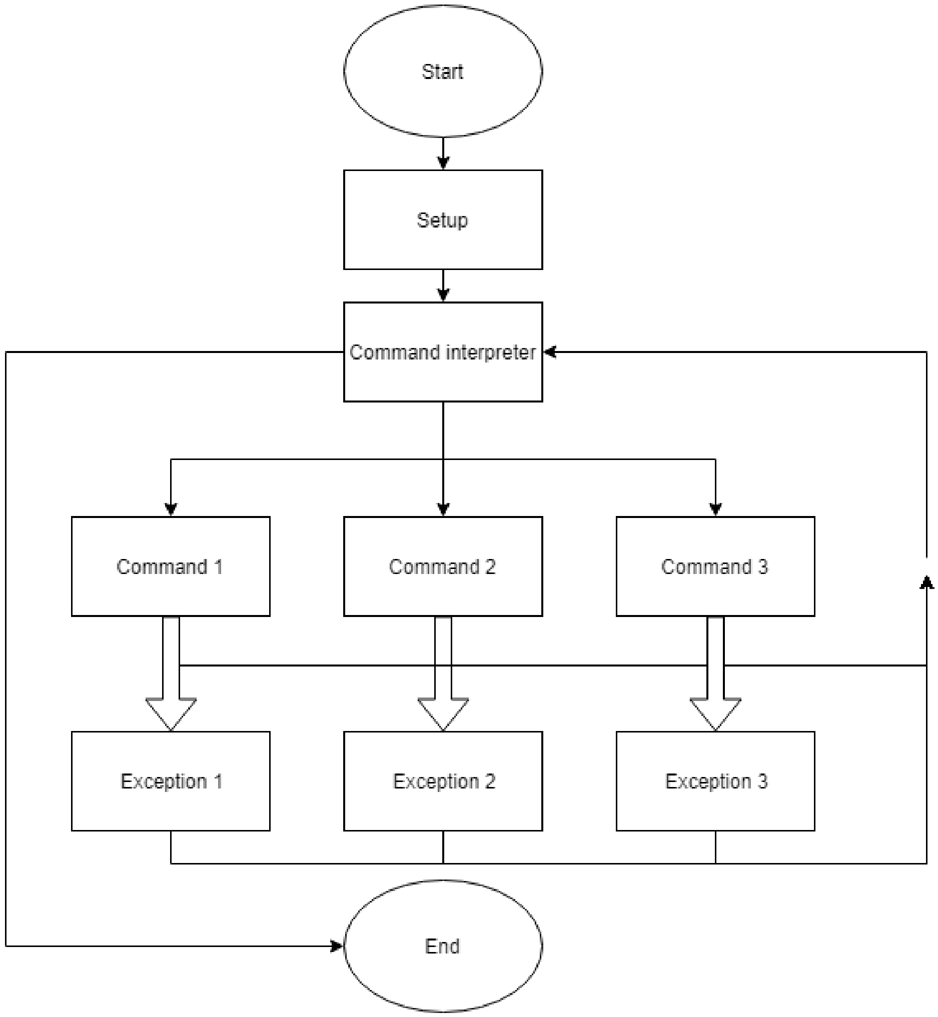


Figure 2: High level design of the monitor program

2.1-) Command Interpreter

The command interpreter, called PARSE, determines which instruction is entered and passes control to the appropriate subroutine. It works by going through a table of commands and checking character-by-character whether the first word of the input is in the table.

2.1.1-) Algorithm and Flowchart

Each command is given a fixed offset, therefore the list of commands is a table. The table of commands is traversed in order by adding the offset and compared against the inputted command to see if there are any matches. This is described in the flowchart in figure 3.

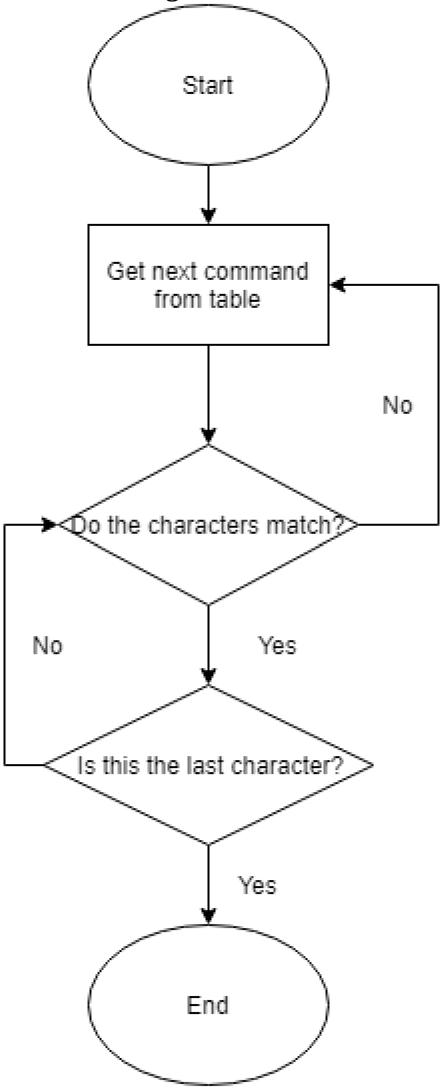


Figure 3: Command interpreter flowchart

2.1.2-) Command Interpreter Assembly Code

```
*------------------------------------------
* Parse commands
* D7 signifies EXIT command received
*------------------------------------------
PARSE   MOVEM.L A1/A2/A3/D0,-(SP)     ;save registers

        CLR D7

        MOVEA.L #INPUT,A1       ;get front of string

        MOVEA.L #COMP_TBL,A2    ;get item in command string
table
        MOVEA.L #COMP_TBL,A3

        MOVE.L  #4,D0           ;set D0 to number of chars to
check

PHELP   CMP.B   (A1)+,(A3)+     ;is command HELP?
        DBNE    D0,PHELP        ;check next character
        BNE     NHELP           ;if did not match, check next
string
        BSR     HELP
        BRA     EXITPARSE

NHELP   MOVEA.L #INPUT,A1
        BSR     NCHAR
        MOVE.L  #4,D0

PMDSP   CMP.B   (A1)+,(A3)+     ;is command MDSP?
        DBNE    D0,PMDSP        ;check next character
        BNE     NMDSP
        BSR     MDSP
        BRA     EXITPARSE       ;if all chars matched, exit

NMDSP   MOVEA.L #INPUT,A1
        BSR     NCHAR
        MOVE.L  #5,D0

PSORTW  CMP.B   (A1)+,(A3)+     ;is command
SORTW?
        DBNE    D0,PSORTW       ;check next
character
        BNE     NSORTW
        BSR     SORTW
        BRA     EXITPARSE       ;if all chars
matched, exit

NSORTW  MOVEA.L #INPUT,A1
        BSR     NCHAR
        MOVE.L  #2,D0

PMM     CMP.B   (A1)+,(A3)+     ;is command
MM?
        DBNE    D0,PMM          ;check next
character
        BNE     NMM
        BSR     MM
        BRA     EXITPARSE       ;if all chars
matched, exit

NMM     MOVEA.L #INPUT,A1
        BSR     NCHAR
        MOVE.L  #2,D0

PMS     CMP.B   (A1)+,(A3)+     ;is command
MS?
        DBNE    D0,PMS          ;check next
character
        BNE     NMS
        BSR     MS
        BRA     EXITPARSE       ;if all chars
matched, exit
```

```
NMS      MOVEA.L #INPUT,A1
         BSR     NCHAR
         MOVE.L  #2,D0

PBF      CMP.B   (A1)+,(A3)+      ;is command
BF?
         DBNE    D0,PBF           ;check next
character
         BNE     NBF
         BSR     BF
         BRA     EXITPARSE        ;if all chars
matched, exit

NBF      MOVEA.L #INPUT,A1
         BSR     NCHAR
         MOVE.L  #4,D0

PBMOV    CMP.B   (A1)+,(A3)+      ;is command
MOV?
         DBNE    D0,PBMOV         ;check next
character
         BNE     NBMOV
         BSR     BMOV
         BRA     EXITPARSE        ;if all chars
matched, exit

NBMOV    MOVEA.L #INPUT,A1
         BSR     NCHAR
         MOVE.L  #4,D0

PBTST    CMP.B   (A1)+,(A3)+      ;is command
BTST?
         DBNE    D0,PBTST         ;check next
character
         BNE     NBTST
         BSR     BTST
         BRA     EXITPARSE        ;if all chars
matched, exit

NBTST    MOVEA.L #INPUT,A1
         BSR     NCHAR
         MOVE.L  #4,D0

PBSCH    CMP.B   (A1)+,(A3)+      ;is command
BSCH?
         DBNE    D0,PBSCH         ;check next
character
         BNE     NBSCH
         BSR     BSCH
         BRA     EXITPARSE        ;if all chars
matched, exit

NBSCH    MOVEA.L #INPUT,A1
         BSR     NCHAR
         MOVE.L  #2,D0
PGO      CMP.B   (A1)+,(A3)+      ;is command
GO?
         DBNE    D0,PGO           ;check next
character
         BNE     NGO
         BSR     GO
         BRA     EXITPARSE        ;if all chars
matched, exit

NGO      MOVEA.L #INPUT,A1
         BSR     NCHAR
         MOVE.L  #2,D0

PDF      CMP.B   (A1)+,(A3)+      ;is command
DF?
         DBNE    D0,PDF           ;check next
character
         BNE     NDF
         BSR     DF
         BRA     EXITPARSE        ;if all chars
matched, exit

NDF      MOVEA.L #INPUT,A1
         BSR     NCHAR
         MOVE.L  #4,D0

PECHO    CMP.B   (A1)+,(A3)+      ;is command
ECHO?
         DBNE    D0,PECHO         ;check next
character
         BNE     NECHO
         BSR     ECHO
         BRA     EXITPARSE        ;if all chars
matched, exit

NECHO    MOVEA.L #INPUT,A1
         BSR     NCHAR
         MOVE.L  #1,D0

PMOD     CMP.B   (A1)+,(A3)+      ;is command
ECHO?
         DBNE    D0,PMOD          ;check next
character
         BNE     NMOD
         BSR     REGMOD
         BRA     EXITPARSE        ;if all chars
matched, exit

NMOD   MOVEA.L #INPUT,A1
         BSR     NCHAR
         MOVE.L  #4,D0

PEXIT    CMP.B   (A1)+,(A3)+      ;is command
EXIT?
         DBNE    D0,PEXIT         ;check next
character
         BNE     NEXIT
```

```
        MOVE.L  #1,D7
        BRA     EXITPARSE       ;if all chars matched, exit

NEXIT   MOVEA.L #INVALID,A1     ;if got here, failed
        MOVE.B  #13,D0
        TRAP    #15
EXITPARSE
        MOVEM.L (SP)+,A1/A2/A3/D0 ;restore registers
        RTS
```

Listing 1: 68000 Assembly Code for command interpreter

2.2-) Debugger Commands

Each debugger command is implemented with a separate subroutine. The command parser will determine whether the command is supported by checking the first word (sequence of characters before the first space) is a supported command and passing control to the appropriate subroutine.

12 commands are supported:
- HELP: help manual, which includes syntax
- MDSP: memory display, which shows everything stored in a given address range
- SORTW: sort, which sorts a block of memory in a given address range
- MM: memory modify, which shows the data at a given address and allows the user to modify the location
- MS: memory set, which is a single instruction to modify the data stored at an address
- BF: fill memory in an address range with a single word-sized value
- BMOV: copy a block of memory to a different location
- BTST: a destructive memory test
- BSCH: search for an ASCII string in an address range
- GO: set the program counter (PC) to a given address, effectively starting execution at the entered address
- DF: displays registers and their contents
- ECHO: echos input back to terminal
- . D1: allows modification of registers
- EXIT: terminate the program

2.2.1-) HELP

The HELP command does not take any arguments. It displays all available commands, what they do, and their arguments.

2.2.1.1-) Debugger Command #1 Algorithm and Flowchart

The help strings are hardcoded into memory using the DC.B instruction. The HELP command goes through each string and outputs it to the terminal. Note that since the strings are of very different lengths, a table is not used. Instead, the address of each string is explicitly moved to a register, and TRAP #15 is called to output it to the terminal.

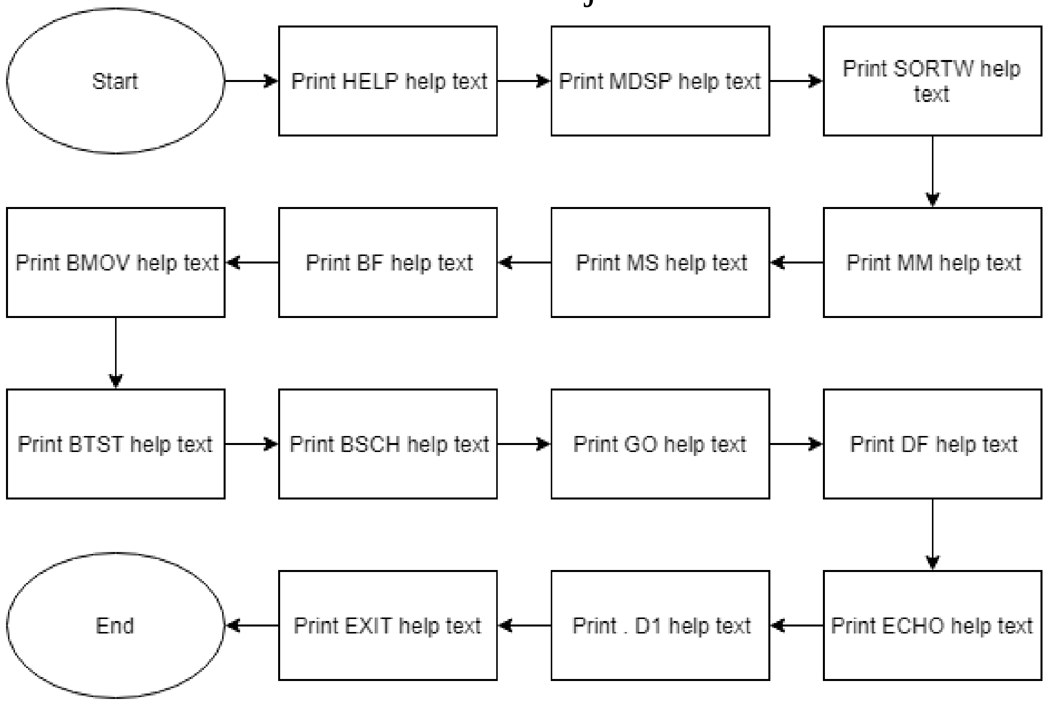


Figure 4: HELP command flowchart

2.2.1.2-) HELP Command Assembly Code

```
*-----------------------------------------
* HELP
*-----------------------------------------
HELP     MOVEM.L D0/A1,-(SP)     ;save
registers

         MOVEA.L #HELP1,A1
         MOVE.B  #13,D0
         TRAP    #15

         MOVEA.L #HELP2,A1
         MOVE.B  #13,D0
         TRAP    #15

         MOVEA.L #HELP2A,A1
         MOVE.B  #13,D0
         TRAP    #15

         MOVEA.L #HELP3,A1
         MOVE.B  #13,D0
         TRAP    #15

         MOVEA.L #HELP4,A1
         MOVE.B  #13,D0
         TRAP    #15

         MOVEA.L #HELP5,A1
         MOVE.B  #13,D0
         TRAP    #15

         MOVEA.L #HELP6,A1
         MOVE.B  #13,D0
         TRAP    #15

         MOVEA.L #HELP7,A1
         MOVE.B  #13,D0
         TRAP    #15

         MOVEA.L #HELP8,A1
         MOVE.B  #13,D0
         TRAP    #15

         MOVEA.L #HELP9,A1
         MOVE.B  #13,D0
         TRAP    #15

         MOVEA.L #HELP10,A1
         MOVE.B  #13,D0
         TRAP    #15

         MOVEA.L #HELP11,A1
         MOVE.B  #13,D0
         TRAP    #15

         MOVEA.L #HELP12,A1
         MOVE.B  #13,D0
         TRAP    #15

         MOVEA.L #HELP13,A1
         MOVE.B  #13,D0
         TRAP    #15
```

```
        MOVEA.L #HELP14,A1
        MOVE.B  #13,D0
        TRAP    #15
        RTS
```

```
                MOVEM.L (SP)+,D0/A1     ;restore
registers
```

Listing 2. HELP Command Assembly Code

2.2.2-) MDSP

The MDSP command, or memory display, has two variants:

1. If two arguments are given, display in long size the memory between the two addresses
2. If one argument is given, display in long size the next 16 bytes after the given address (4 longs)

The addresses are given in hexadecimal, and the memory is displayed in hex as well.

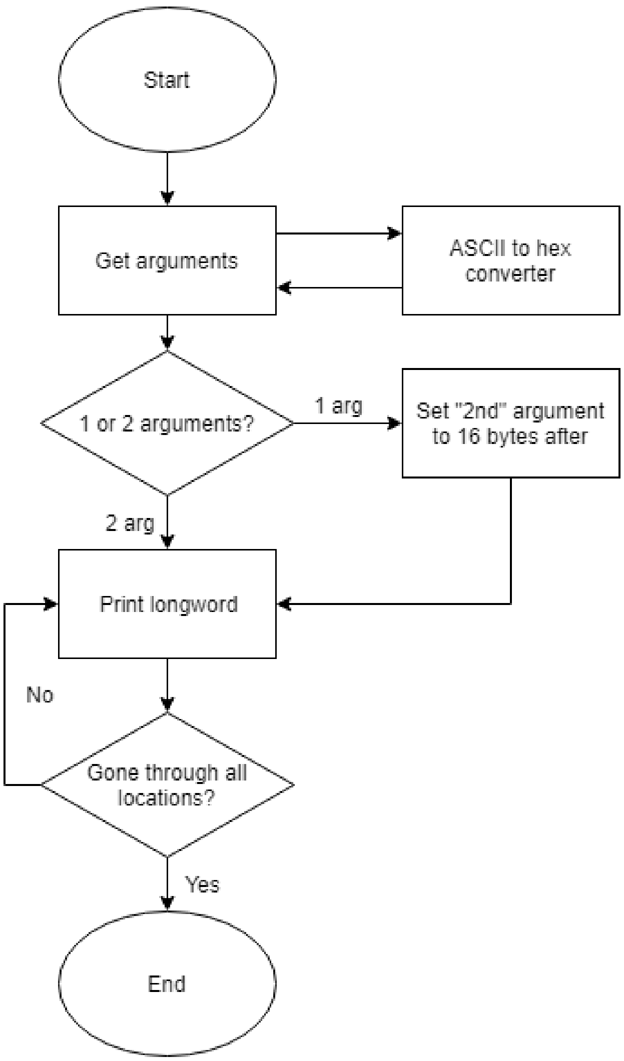


Figure 5: MDSP Command Flowchart

2.2.2.1-) MDSP Command Algorithm and Flowchart

The ASCII helper subroutine, which reads a hex encoded ASCII string and converts it to the correct hex value, was written for the MDSP command and was used over and over for other commands as well. The MDSP command was implemented as shown in figure 5. Of note: trap 15 of the Easy68K trap #15 function was also extensively used to allow automatic conversion to hex encoded ASCII, used for displaying to terminal.

2.2.2.2-) MDSP Command Assembly Code

```
*---------------------------------------------------
* MDSP
*---------------------------------------------------
MDSP    MOVEM.L D0/D1/A2/A3,-(SP)

        BSR     ASCII           ;get first argument
        MOVEA.L D1,A2
        BSR     ASCII           ;get second argument
        MOVEA.L D1,A3

        CMPA.L  #0,A3           ;how many arguments?
        BNE     MEMLOOP
        MOVEA.L A2,A3
        ADDA.L  #16,A3          ;if one argument, 16
bytes

MEMLOOP CMPA.L  A2,A3           ;continue while still
in range
        BLE     MDSPEXIT

        MOVE.L  A2,D1           ;print address
        MOVE.B  #16,D2
        MOVE.B  #15,D0
        TRAP    #15

        MOVE.L  #$3A,D1         ;print colon
```

```
        MOVE.B  #6,D0
        TRAP    #15

        MOVE.L  (A2)+,D1        ;print value
        MOVE.B  #16,D2
        MOVE.B  #15,D0
        TRAP    #15

        BSR     CRLF

        BRA     MEMLOOP

MDSPEXIT MOVEM.L (SP)+,D0/D1/A2/A3
        RTS
```

Listing 3. MDSP Command Assembly Code

2.2.3-) SORTW

The SORTW, or memory sort, command, takes 3 arguments: the start address, the end address, and the order. The addresses are inputted as hex from the terminal, and the order is inputted as either the letters “A” or “D”, signifying ascending or descending, respectively. Ascending is taken to mean higher value for higher addresses.

2.2.3.1-) SORTW Command Algorithm and Flowchart

An implementation of bubble sort was used for this command. The algorithm can be described in pseudocode as follows:

```
i = start, j = start                        // use i and j as indexes
while (i < end)                             // continue until everything is sorted
    j = i
    while (j < end)                         // bubble largest (or smallest)
        if (ascending && mem[j] < mem[j+1])     // ascending
            swap mem[j] and mem[j+1]        // swap if out of order
        if (descending && mem[j+1] < mem[j])    // descending
            swap mem[j] and mem[j+1]        // swap if out of order
        j++;
    i++
```

Listing 4. SORTW Command Pseudocode

2.2.3.2-) SORTW Command Assembly Code

```
*---------------------------------------------
* SORTW
*---------------------------------------------
SORTW   MOVEM.L D0/D1/D2/A2/A3/A4,-(SP)

        BSR     ASCII           ;first arg
        MOVEA.L D1,A2
        BSR     ASCII           ;second arg
        MOVEA.L D1,A3
        MOVE.B  (A1)+,D2        ;third arg

        MOVEA.L A2,A4
SORTLOOP MOVEA.L A4,A2
BUBBLE  CMP.B   #$41,D2         ;is it "A"?
        BGT     DSC
ASC     CMP.W   (A2)+,(A2)+
        BLS     CMPNXT          ;ascending
        BRA     SWAP
DSC     CMP.W   (A2)+,(A2)+
        BHI     CMPNXT          ;descending

SWAP    MOVE.L  -(A2),D0
        SWAP.W  D0
        MOVE.L  D0,(A2)
        BRA     SORTLOOP
```

```
CMPNXT  SUBA.L   #2,A2
        CMP.L    A2,A3
        BGT      BUBBLE
```

Listing 5. SORTW Command Assembly Code

```
        MOVEM.L (SP)+,D0/D1/D2/A2/A3/A4
        RTS
```

2.2.4-) MM

The MM, or memory modify command, takes two arguments: an address and data size, expressed as “B” for byte, “W” for word, or “L” for longword. It displays the data stored at the specified address, with the option to change the data at that location. “Enter” can be pressed to advance to the next address without modifying the data at the address, and “.” can be pressed to exit the MM command.

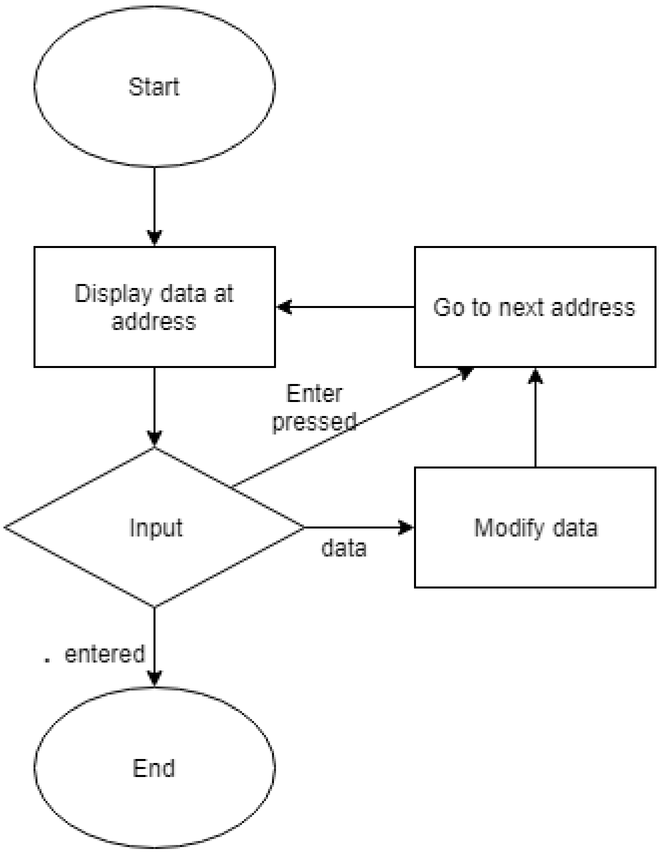


Figure 6: MM Command Flowchart

2.2.4.1-) MM Command Algorithm and Flowchart

The ASCII subroutine is used to get the address, and the data size is fetched separately. It behaves as shown in the flowchart in figure 6:

2.2.4.2-) MM Command Assembly Code

```
*------------------------------------------
* MM
*------------------------------------------
MM      MOVEM.L D0/D1/D2/D3/A2,-(SP)

        BSR     ASCII           ;get address 
argument
        MOVEA.L D1,A2
        MOVE.B  (A1)+,D3

MMNXT   MOVE.L  A2,D1           ;print 
address
        MOVE.B  #16,D2
        MOVE.B  #15,D0
        TRAP    #15

        MOVE.L  #$3A,D1         ;print colon
        MOVE.B  #6,D0
        TRAP    #15

        CLR.L   D1
        CMP.B   #$42,D3         ;a byte?
        BNE     MMWORD
        MOVE.B  (A2)+,D1
        BRA     MMTRAP

MMWORD  CMP.B   #$57,D3         ;a word?
        BNE     MMLONG
        MOVE.W  (A2)+,D1
        BRA     MMTRAP

MMLONG  MOVE.L  (A2)+,D1        ;must be long

MMTRAP  MOVE.B  #16,D2
        MOVE.B  #15,D0
        TRAP    #15

        MOVE.L  #$3F,D1         ;print ?
        MOVE.B  #6,D0
        TRAP    #15

        MOVEA.L #INPUT,A1
        MOVE.B  #2,D0
        TRAP    #15             ;read value

        CMP.B   #$2E,(A1)       ;quit on . 
entered
        BEQ     EXMM
```

```
        TST.B   D1
        BEQ     MMNXT

        BSR     ASCII           ;convert
value to hex

        CMP.B   #$42,D3         ;a byte?
        BNE     MMWORD2
        MOVE.B  D1,-(A2)
        ADDA.L  #1,A2
        BRA     MMNXT

MMWORD2 CMP.B   #$57,D3         ;a word?
        BNE     MMLONG2
        MOVE.W  D1,-(A2)
        ADDA.L  #2,A2
        BRA     MMNXT

MMLONG2 MOVE.L  D1,-(A2)        ;must be long
        ADDA.L  #4,A2

        BRA     MMNXT

EXMM    MOVEM.L (SP)+,D0/D1/D2/D3/A2
        RTS
```

Listing 6. MM Command Assembly code

2.2.5-) MS

The MS, or memory set, command, takes 2 arguments: address and data. It stores the given word-size data value to the given address location.

2.2.5.1-) MS Command Algorithm and Flowchart

The implementation can be expressed in pseudocode as follows:

Get address
Get data
Write data to memory

Listing 7. MS Command pseudocode

2.2.5.2-) MS Command Assembly Code

```
*------------------------------------------------------------
* MS
*------------------------------------------------------------
MS      MOVEM.L D1/A2,-(SP)

        BSR     ASCII           ;read address argument
        MOVEA.L D1,A2
        BSR     ASCII           ;read data argument

        MOVE.W  D1,(A2)         ;write data to memory

        MOVEM.L (SP)+,D1/A2
        RTS
```

Listing 8. MS Command Assembly code

2.2.6-) BF

The BF, or block fill, command takes 3 arguments: start address, end address, and data. It fills the block of memory between the two addresses with the given word-sized data value.

2.2.6.1-) BF Command Algorithm and Flowchart

The ASCII subroutine is used to get the three arguments. The block filling algorithm follows the flowchart in figure 7:

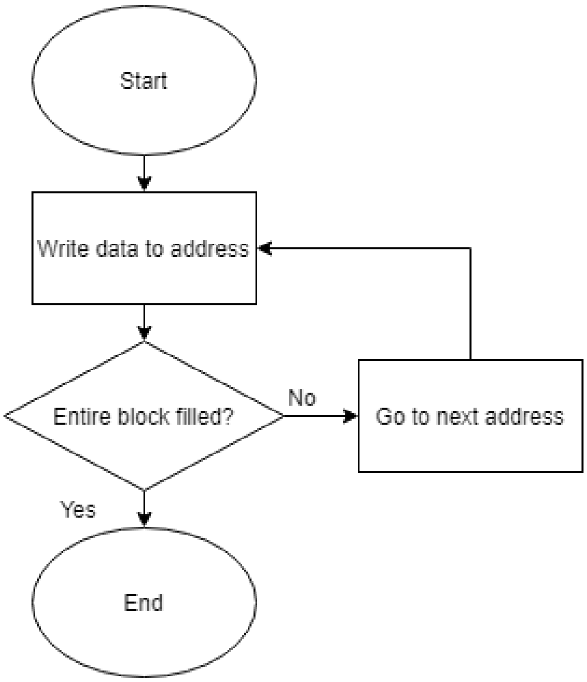


Figure 7: BF Command Flowchart

2.2.6.2-) BF Command Assembly Code

```
*-------------------------------------------------------
* BF
*-------------------------------------------------------
BF      MOVEM.L D1/A2/A3,-(SP)

        BSR     ASCII           ;read start address
        MOVEA.L D1,A2
        BSR     ASCII           ;read end address
        MOVEA.L D1,A3
        BSR     ASCII           ;read data

BFLOOP  CMP.L   A2,A3
        BLT     EXBF
        MOVE.W  D1,(A2)+        ;write to memory
        BRA     BFLOOP

EXBF    MOVEM.L (SP)+,D1/A2/A3
        RTS
```

Listing 9: BF Command Assembly code

2.2.7-) BMOV

The BMOV, or block move, command takes 3 arguments: original block start address, original block end address, and new block start address. It copies a block of memory between the given addresses and makes a copy to the block starting at the new address.

2.2.7.1-) BMOV Command Algorithm and Flowchart

The ASCII subroutine is used to get the three arguments. The block filling algorithm follows the flowchart in figure 8:

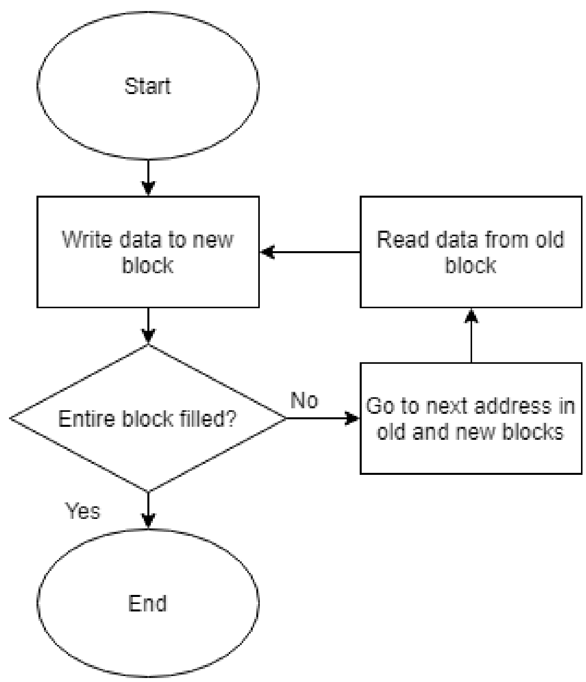


Figure 8: BMOV Command Flowchart

2.2.7.2-) BMOV Command Assembly Code

```
*-----------------------------------------------------
* BMOV
*-----------------------------------------------------
BMOV    MOVEM.L D1/A2/A3/A4,-(SP)

        BSR     ASCII           ;original start
        MOVEA.L D1,A2
        BSR     ASCII           ;original end
        MOVEA.L D1,A3
        BSR     ASCII           ;new start
        MOVEA.L D1,A4

BMOVLOOP CMP.L  A2,A3
        BLT     EXBMOV
        MOVE.W  (A2)+,(A4)+     ;copy memory
        BRA     BMOVLOOP

EXBMOV  MOVEM.L (SP)+,D1/A2/A3/A4
        RTS
```

Listing 10: BMOV Command Assembly code

2.2.8-) BTST

The BTST, or block test, command takes 2 arguments: start address and end address. It tests that reading and writing of all bits in the block of memory works as desired. One usage is to test a new memory chip.

2.2.8.1-) BTST Command Algorithm and Flowchart

The test works as described in figure 9:

1. Fill block with $55 in each location
2. Read each location and check if value stored is $55
3. Fill block with $AA in each location
4. Read each location and check if value stored is $AA

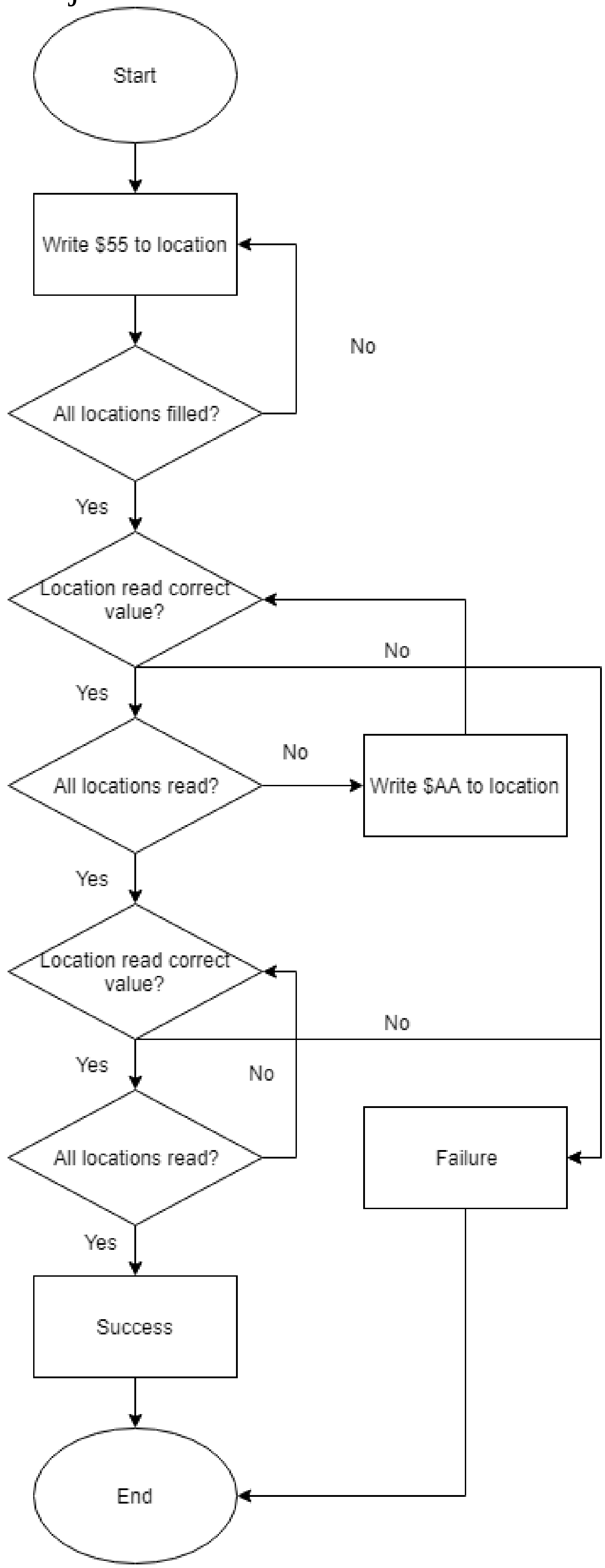


Figure 9: BTST Command Flowchart

2.2.8.2-) BTST Command Assembly Code

```
*---------------------------------------------
* BTST
*---------------------------------------------
BTST    MOVEM.L D0/D1/D2/A2/A3/A4,-(SP)

        BSR     ASCII           ;get start
address
        MOVEA.L D1,A2
        MOVEA.L D1,A4
```

```
        BSR     ASCII           ;get end
address
        MOVEA.L D1,A3

BTSTLOOP1 CMP.L A2,A3
        BLT     ENDLOOP1
        MOVE.B  #$55,(A2)+      ;fill memory
        BRA     BTSTLOOP1
ENDLOOP1 MOVEA.L A4,A2
        MOVE.B  #$55,D2

BTSTLOOP2 CMP.L A2,A3
        BLT     ENDLOOP2
        CMP.B   #$55,(A2)       ;check read
value
        BNE     BAD
        MOVE.B  #$AA,(A2)+      ;fill memory
        BRA     BTSTLOOP2
ENDLOOP2 MOVEA.L A4,A2
        MOVE.B  #$AA,D2

BTSTLOOP3 CMP.L A2,A3
        BLT     GOOD
        CMP.B   #$AA,(A2)+      ;check read
value
        BNE     BAD
        BRA     BTSTLOOP3

BAD     SUBA.L  #1,A2           ;go to broken
address

        MOVE.L  A2,D1           ;print
address
        MOVE.B  #16,D2
        MOVE.B  #15,D0
        TRAP    #15

        MOVE.L  #$3A,D1         ;print colon
        MOVE.B  #6,D0
        TRAP    #15

        MOVE.L  D2,D1           ;print data
stored
        MOVE.B  #16,D2
        MOVE.B  #15,D0
        TRAP    #15

        MOVE.L  #$2C,D1         ;print comma
        MOVE.B  #6,D0
        TRAP    #15

        MOVE.L  (A2),D1         ;print data
read
        MOVE.B  #16,D2
        MOVE.B  #15,D0
        TRAP    #15

        BRA     EXBTST

GOOD    MOVEA.L #SUCCESS,A1     ;print
success
        MOVE.B  #13,D0
        TRAP    #15

EXBTST  MOVEM.L (SP)+,D0/D1/D2/A2/A3/A4
        MOVEA.L #FAILURE,A1     ;print
failure
        MOVE.B  #13,D0
        TRAP    #15
        RTS
```

Listing 11: BTST Command Assembly code

2.2.9-) BSCH

The BSCH, or block search, command takes three arguments: start address, end address, and search string. The command searches for the location of a given string in the block specified.

2.2.9.1-) BSCH Command Algorithm and Flowchart

BSCH attempts to do a linear search of the whole block. It tries to match character by character between the block and the query, starting at a different location of the search block on each try. This is shown in the flowchart in figure 10.

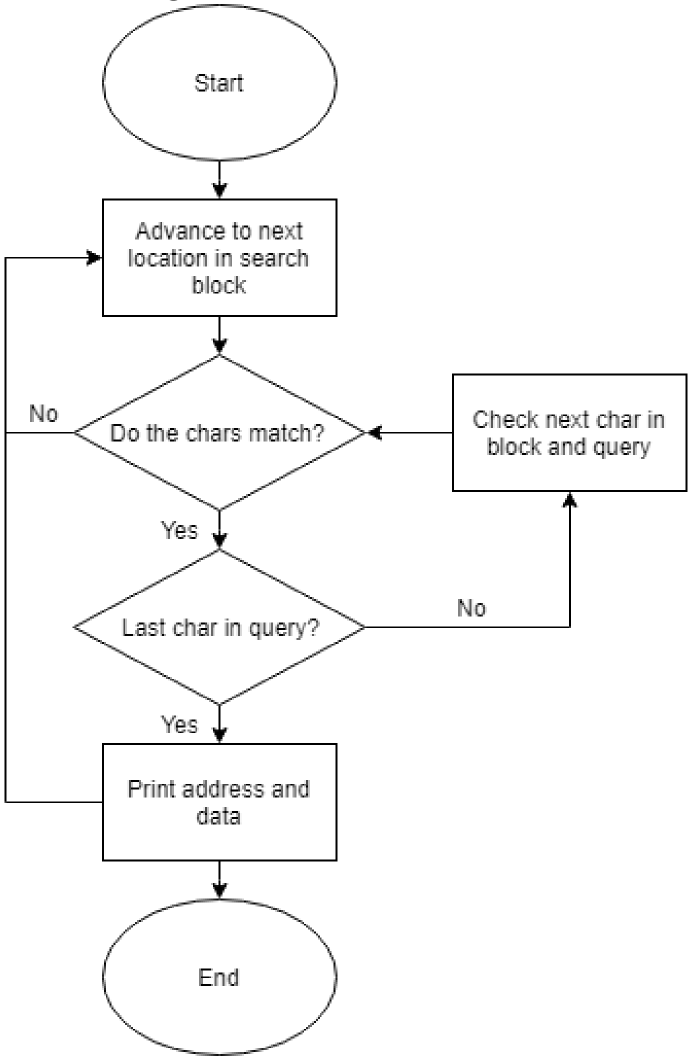


Figure 10: BSCH Command Flowchart

2.2.9.2-) BSCH Command Assembly Code

```
*-----------------------------------------------
* BSCH
*-----------------------------------------------
BSCH    MOVEM.L D0/D1/D2/A1/A2/A3/A4/A5,-(SP)

        BSR     ASCII           ;get start
address
        MOVEA.L D1,A2
        MOVEA.L A2,A5
        BSR     ASCII           ;get end address
        MOVEA.L D1,A3
        MOVEA.L A1,A4           ;save start of
search string

        CLR.L   D0
LENSCH  CMP.B   #$00,(A1)+      ;get length of
search string
        BEQ     SAVELEN
        ADDI.L  #1,D0
        BRA     LENSCH

SAVELEN MOVE.L  D0,D3

SCHLOOP MOVEA.L A4,A1           ;restore to start
of search
        MOVEA.L A5,A2           ;check with next
char
        MOVE.L  D3,D0           ;restore search
length
        CMP.L   A2,A3
        BLT     ENDSCH

        ADDA.L  #1,A5           ;next starting
point

SCHFIND CMP.B   (A1)+,(A2)+     ;does string match?
        DBNE    D0,SCHFIND      ;check next character
        BNE     SCHLOOP

        SUBA.L  #1,A5           ;go back to starting address
        MOVE.L  A5,D1           ;print address
        MOVE.B  #16,D2
        MOVE.B  #15,D0
        TRAP    #15

        MOVE.L  #$3A,D1         ;print colon
        MOVE.B  #6,D0
        TRAP    #15

        MOVE.L  A5,A1           ;print data
        MOVE.B  #13,D0
        TRAP    #15

        ADDA.L  #1,A5           ;restore to next starting address
```

```
        BRA     SCHLOOP

ENDSCH  MOVEM.L (SP)+,D0/D1/D2/A1/A2/A3/A4/A5
        RTS
```

Listing 12: BSCH Command Assembly code

2.2.10-) GO

The GO, or execute, command takes one argument: the address of the program to be executed. In this implementation, the target program must be written as a subroutine, returning control to the main program with RTS.

2.2.10.1-) GO Command Algorithm and Flowchart

The target program address is read using the ASCII subroutine, which reads an ASCII-encoded hex value and converts it to hex. The JSR instruction does not support address register direct, so address register indirect is used with displacement of zero.

2.2.10.2-) GO Command Assembly Code

```
*------------------------------------------------------------
* GO
*------------------------------------------------------------
GO      MOVEM.L D1/A1,-(SP)

        BSR     ASCII           ;get address
        MOVEA.L D1,A1           ;use arbitrary address
        JSR     0(A1)

        MOVEM.L (SP)+,D1/A1
        RTS
```

Listing 13: GO Command Assembly code

2.2.11-) DF

The DF, or display formatted registers, command does not take any arguments. It displays the value stored in the program counter, status register, user stack pointer, system stack pointer, data registers, and address registers without the modifications within the monitor program itself.

2.2.11.1-) DF Command Algorithm and Flowchart

As the DF command needs to display all register values prior to the execution of the monitor program, all register values must be saved as part of startup. The values are saved in a table in memory as part of the SAVE subroutine (see appendix A). The DF command only needs to go through the table and print out each value saved.

2.2.11.2-) DF Command Assembly Code

```
*--------------------------------
* DF
*--------------------------------
DF      MOVEM.L D0/D1/D2/A0/A1,-(SP)
```

```
        MOVEA.L #REGVAL,A0

        MOVEA.L #REGPC,A1
```

```
        MOVE.B  #14,D0          ;print PC
        TRAP    #15
        MOVE.L  (A0)+,D1        ;print value
        MOVE.B  #16,D2
        MOVE.B  #15,D0
        TRAP    #15
        BSR     CRLF

        MOVEA.L #REGSR,A1
        MOVE.B  #14,D0          ;print SR
        TRAP    #15
        MOVE.L  (A0)+,D1        ;print value
        MOVE.B  #16,D2
        MOVE.B  #15,D0
        TRAP    #15
        BSR     CRLF

        MOVEA.L #REGUS,A1
        MOVE.B  #14,D0          ;print US
        TRAP    #15
        MOVE.L  (A0)+,D1        ;print value
        MOVE.B  #16,D2
        MOVE.B  #15,D0
        TRAP    #15
        BSR     CRLF

        MOVEA.L #REGSS,A1
        MOVE.B  #14,D0          ;print SS
        TRAP    #15
        MOVE.L  (A0)+,D1        ;print value
        MOVE.B  #16,D2
        MOVE.B  #15,D0
        TRAP    #15
        BSR     CRLF

        MOVEA.L #REGD0,A1
        MOVE.B  #14,D0          ;print D0
        TRAP    #15
        MOVE.L  (A0)+,D1        ;print value
        MOVE.B  #16,D2
        MOVE.B  #15,D0
        TRAP    #15
        BSR     CRLF

        MOVEA.L #REGD1,A1
        MOVE.B  #14,D0          ;print D1
        TRAP    #15
        MOVE.L  (A0)+,D1        ;print value
        MOVE.B  #16,D2
        MOVE.B  #15,D0
        TRAP    #15
        BSR     CRLF

        MOVEA.L #REGD2,A1
        MOVE.B  #14,D0          ;print D2
        TRAP    #15
        MOVE.L  (A0)+,D1        ;print value
        MOVE.B  #16,D2
        MOVE.B  #15,D0
        TRAP    #15
        BSR     CRLF

        MOVEA.L #REGD3,A1
        ADDA.L  #4,A1
        MOVE.B  #14,D0          ;print D3
        TRAP    #15
        MOVE.L  (A0)+,D1        ;print value
        MOVE.B  #16,D2
        MOVE.B  #15,D0
        TRAP    #15
        BSR     CRLF

        MOVEA.L #REGD4,A1
        MOVE.B  #14,D0          ;print D4
        TRAP    #15
        MOVE.L  (A0)+,D1        ;print value
        MOVE.B  #16,D2
        MOVE.B  #15,D0
        TRAP    #15
        BSR     CRLF

        MOVEA.L #REGD5,A1
        MOVE.B  #14,D0          ;print D5
        TRAP    #15
        MOVE.L  (A0)+,D1        ;print value
        MOVE.B  #16,D2
        MOVE.B  #15,D0
        TRAP    #15
        BSR     CRLF

        MOVEA.L #REGD6,A1
        MOVE.B  #14,D0          ;print D6
        TRAP    #15
        MOVE.L  (A0)+,D1        ;print value
        MOVE.B  #16,D2
        MOVE.B  #15,D0
        TRAP    #15
        BSR     CRLF

        MOVEA.L #REGD7,A1
        MOVE.B  #14,D0          ;print D7
        TRAP    #15
        MOVE.L  (A0)+,D1        ;print value
        MOVE.B  #16,D2
        MOVE.B  #15,D0
        TRAP    #15
        BSR     CRLF

        MOVEA.L #REGA0,A1
        MOVE.B  #14,D0          ;print A0
        TRAP    #15
        MOVE.L  (A0)+,D1        ;print value
        MOVE.B  #16,D2
        MOVE.B  #15,D0
```

```
        TRAP     #15
        BSR      CRLF

        MOVEA.L #REGA1,A1
        MOVE.B   #14,D0           ;print A1
        TRAP     #15
        MOVE.L   (A0)+,D1         ;print value
        MOVE.B   #16,D2
        MOVE.B   #15,D0
        TRAP     #15
        BSR      CRLF

        MOVEA.L #REGA2,A1
        MOVE.B   #14,D0           ;print A2
        TRAP     #15
        MOVE.L   (A0)+,D1         ;print value
        MOVE.B   #16,D2
        MOVE.B   #15,D0
        TRAP     #15
        BSR      CRLF

        MOVEA.L #REGA3,A1
        MOVE.B   #14,D0           ;print A3
        TRAP     #15
        MOVE.L   (A0)+,D1         ;print value
        MOVE.B   #16,D2
        MOVE.B   #15,D0
        TRAP     #15
        BSR      CRLF

        MOVEA.L #REGA4,A1
        MOVE.B   #14,D0           ;print A4
        TRAP     #15
        MOVE.L   (A0)+,D1         ;print value
        MOVE.B   #16,D2
        MOVE.B   #15,D0
        TRAP     #15
        BSR      CRLF

        MOVEA.L #REGA5,A1
        MOVE.B   #14,D0           ;print A5
        TRAP     #15
        MOVE.L   (A0)+,D1         ;print value
        MOVE.B   #16,D2
        MOVE.B   #15,D0
        TRAP     #15
        BSR      CRLF

        MOVEA.L #REGA6,A1
        MOVE.B   #14,D0           ;print A6
        TRAP     #15
        MOVE.L   (A0)+,D1         ;print value
        MOVE.B   #16,D2
        MOVE.B   #15,D0
        TRAP     #15
        BSR      CRLF

        MOVEA.L #REGA7,A1
        MOVE.B   #14,D0           ;print A7
        TRAP     #15
        MOVE.L   (A0)+,D1         ;print value
        MOVE.B   #16,D2
        MOVE.B   #15,D0
        TRAP     #15
        BSR      CRLF

        MOVEM.L (SP)+,D0/D1/D2/A0/A1
        RTS
```

Listing 14: DF Command Assembly code

2.2.12-) ECHO

The ECHO command behaves like the command with the same name in a standard Linux bash terminal: it prints out the value of the input received at terminal. This instruction is intended to be used as part of a larger “script,” or collection of commands, to display information to the user.

2.2.12.1-) ECHO Command Algorithm and Flowchart

The command is implemented by moving the string pointer to after the command (done in the interpreter) and passing the pointer to trap 13.

2.2.12.2-) ECHO Command Assembly Code

```
*------------------------------------------------------------
* ECHO
*------------------------------------------------------------
ECHO    MOVEM.L D0,-(SP)
        MOVE.B   #13,D0           ;echo input
```

```
        TRAP #15
        MOVEM.L (SP)+,D0
        RTS
```

Listing 15: ECHO Command Assembly code

2.2.13-) . D1 (Register Modify)

The register modify commands are a collection of 16 “commands”. Each is entered as “. ” followed by data or address register, and given an argument of the value to be stored in the given register.

2.2.13.1-) Register Modify Command Algorithm and Flowchart

The modifications to register values should be on the register values outside of the monitor program execution. Any changes are written to the memory table of register values that DF reads off of. The external register values are updated on program termination.

A table of register “names” stored in hex (for example, \$D1, \$A7) is in memory. To detect which register is being modified, a comparison is done against each value in the table, as indicated in figure 11. A more memory efficient method of register detection would be to first check if the register is a data or address register by reading and comparing the first character to the ASCII values of D and A. The next single character would then be the register number. The drawbacks of the latter approach would be additional code complexity from number detection.

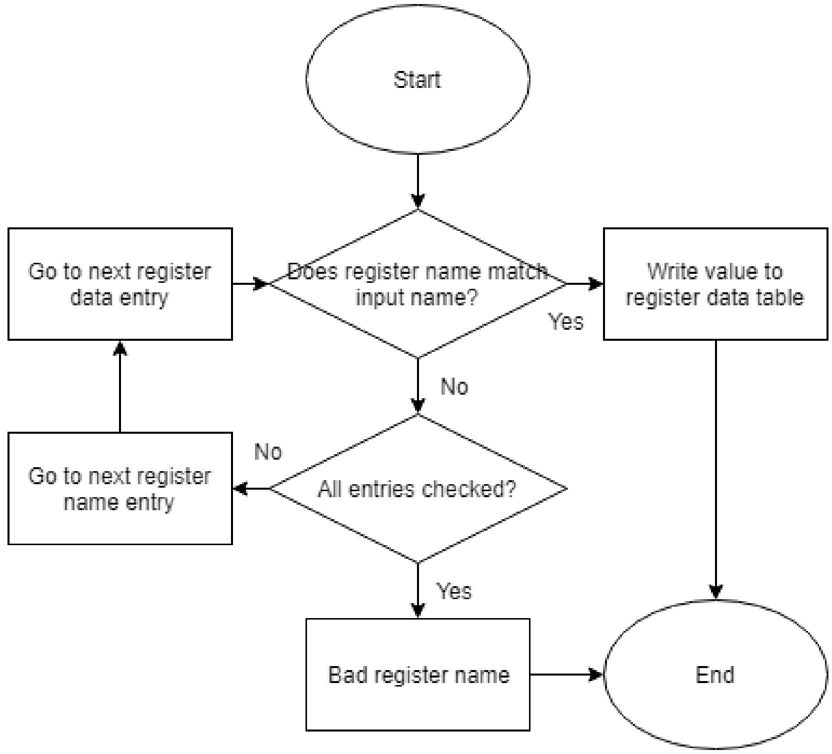


Figure 11: Register Modify Command Flowchart

2.2.13.2-) Register Modify Command Assembly Code

```
*---------------------------------------------
* REGMOD
*---------------------------------------------
REGMOD  MOVEM.L D0/D1/D2/A1/A2,-(SP)

        CLR     D0
        BSR     ASCII           ;get register
        MOVE.L  D1,D2
        BSR     ASCII           ;get data

        MOVEA.L #REGVAL,A1
        ADDA.L  #16,A1
        MOVEA.L #REGNAME,A2

MODLOOP CMP.B   (A2)+,D2
        BEQ     MODFOUND
        ADDA.L  #4,A1
        ADDI.B  #1,D0
        CMP.B   #16,D0
        BGT     NOTFOUND
        BRA     MODLOOP

NOTFOUND MOVEA.L #BADREG,A1
        MOVE.B  #13,D0
        TRAP    #15
        BRA EXREGMOD

MODFOUND MOVE.L D1,(A1)
EXREGMOD MOVEM.L (SP)+,D0/D1/D2/A1/A2
        RTS
```

Listing 16: Register Modify Command Assembly code

2.2.14-) EXIT

The EXIT command does not take any arguments. It terminates the monitor program and returns control to the processor.

2.2.14.1-) EXIT Command Algorithm and Flowchart

The EXIT command does not have its own subroutine. Instead, it is detected by the parser, which sets a flag. On every iteration, the main routine checks this flag and exits the program on flag set.

2.2.14.2-) EXIT Command Assembly Code

```
PEXIT   CMP.B   (A1)+,(A3)+     ;is command EXIT?
        DBNE    D0,PEXIT        ;check next character
        BNE     NEXIT
        MOVE.L  #1,D7           ;set exit flag
        BRA     EXITPARSE       ;if all chars matched, exit
```
Listing 17: Parse subroutine snippet: Register Modify Command Assembly code

```
        TST.B   D7
        BNE     EXITMAIN
```
Listing 18: Main routine snippet: Register Modify Command Assembly code

2.3-) Exception Handlers

Exception handlers are registered with the MC68000 on program start by modifying the interrupt vector table. The following types of exceptions are handled:
- Bus error
- Address error
- Illegal instruction error
- Divide by zero error
- Privilege violation error
- Line A error
- Line F error

2.3.1-) Bus Error Exception

A bus error exception occurs when an instruction attempts to access memory at a memory location that does not exist. To register a bus error exception handler, the handler address is written into memory location $008, the interrupt vector table entry that corresponds to bus errors.

2.3.1.1-) Bus Error Exception Algorithm and Flowchart

When a bus error exception occurs, the address of the instruction that caused the error, the supervisor status word (SSW), the bus address (BA), and the instruction register (IR), data registers, and address registers are all printed. The data and address registers are printed by utilizing the DF subroutine written for the DF command. To get the SSW, BA, and IR, the supervisor stack pointer (SSP) must be manipulated to get the required data.

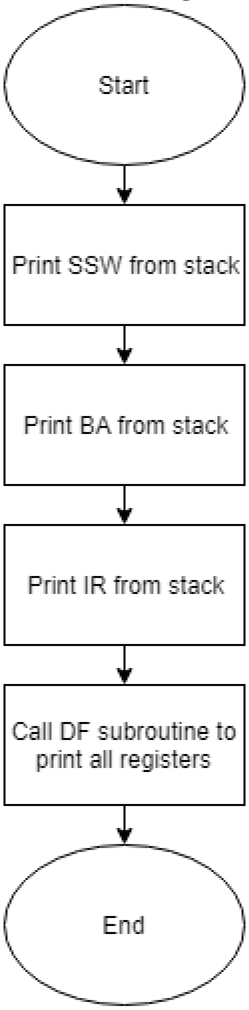


Figure 12: Bus Error Exception Flowchart

2.3.1.2-) Bus Error Exception Assembly Code

```
*-------------------------
* Bus error exception
*-------------------------
BUS     MOVEM.L D0/A1,-(SP)

        MOVEA.L #BUS_MSG,A1
        MOVE.B  #13,D0
        TRAP    #15

        CLR     D1
        SUBA.W  #$2,A7
;move SSP to point to SSW
        MOVE.L  (A7)+,D1
;print SSW
        MOVE.B  #16,D2
        MOVE.B  #15,D0
        TRAP    #15

        BSR     SPACE

        CLR     D1
        MOVE.L  (A7)+,D1        ;print BA
        MOVE.B  #16,D2
        MOVE.B  #15,D0
        TRAP    #15

        BSR     SPACE

        CLR     D1
        MOVE.W  (A7)+,D1        ;print IR
        MOVE.B  #16,D2
        MOVE.B  #15,D0
        TRAP    #15

        BSR     CRLF
        BSR     DF

        MOVEM.L (SP)+,D0/A1
        RTE
```

Listing 19: Bus Error Exception Assembly Code

2.3.2-) Address Error Exception

An address error occurs when an instruction attempts to access a word or a longword using an odd address. To register an address error exception handler, the handler address is written into memory location $00C, the interrupt vector table entry that corresponds to address errors.

2.3.1.1-) Address Error Exception Algorithm and Flowchart

A similar algorithm is implemented as for bus error exceptions.

2.3.1.2-) Address Error Exception Assembly Code

```
*---------------------------------------------
* Address error exception
*---------------------------------------------
ADDR    MOVEM.L D0/D1/D2/A1,-(SP)

        MOVEA.L #ADDR_MSG,A1
        MOVE.B  #13,D0
        TRAP    #15

        CLR     D1
        SUBA.W  #$2,A7          ;move SSP to
point to SSW
        MOVE.L  (A7)+,D1        ;print SSW
        MOVE.B  #16,D2
        MOVE.B  #15,D0
        TRAP    #15

        BSR     SPACE

        CLR     D1
        MOVE.L  (A7)+,D1        ;print BA
        MOVE.B  #16,D2
        MOVE.B  #15,D0
        TRAP    #15

        BSR     SPACE

        CLR     D1
        MOVE.W  (A7)+,D1        ;print IR
        MOVE.B  #16,D2
        MOVE.B  #15,D0
        TRAP    #15

        BSR     CRLF
```

```
        BSR     DF                                              RTE

        MOVEM.L (SP)+,D0/D1/D2/A1
```
Listing 20: Address Error Exception Assembly Code

2.3.3-) Illegal Instruction Exception

An illegal instruction exception occurs when a given bit pattern in memory that is expected to be an instruction does not correspond to any valid instruction. To register an illegal instruction exception handler, the handler address is written into memory location $010, the interrupt vector table entry that corresponds to illegal instructions.

2.3.3.1-) Illegal Instruction Exception Algorithm and Flowchart

There is no need to print the SSW, BA, and IR for non memory access related errors. Therefore, the algorithm is a simplified version of the one used for bus error, where DF is called to print the register values.

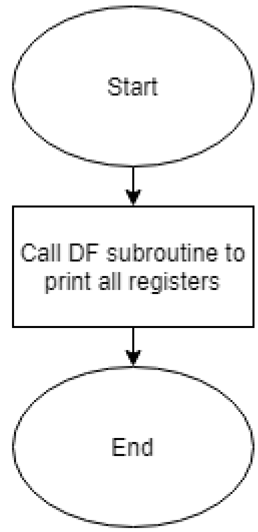


Figure 13: Illegal Instruction Exception Flowchart

2.3.3.2-) Illegal Instruction Exception Assembly Code

```
*------------------------------------------------------------
* Illegal instruction exception
*------------------------------------------------------------
ILLI    MOVEM.L D0/A1,-(SP)
        MOVEA.L #ILLI_MSG,A1
        MOVE.B  #13,D0
        TRAP    #15
        BSR     DF
        MOVEM.L (SP)+,D0/A1
        RTE
```
Listing 21: Illegal Instruction Exception Assembly Code

2.3.4-) Privilege Violation Exception

A privilege violation exception occurs when a supervisor instruction is executed while in user mode (for example, writing to the status register). To register a privilege violation exception handler, the handler address is written into memory location $020, the interrupt vector table entry that corresponds to privilege violations.

2.3.4.1-) Privilege Violation Exception Algorithm and Flowchart

A similar algorithm is implemented as for the illegal instruction exception.

2.3.4.2-) Privilege Violation Exception Assembly Code

```
*------------------------------------------------------------
* Privilege violation exception
*------------------------------------------------------------
PRIV    MOVEM.L D0/A1,-(SP)
        MOVEA.L #PRIV_MSG,A1
        MOVE.B  #13,D0
        TRAP    #15
        BSR     DF
```

```
        MOVEM.L (SP)+,D0/A1
        RTE
```

Listing 22: Privilege Violation Exception Assembly Code

2.3.5-) Divide by Zero Exception

A divide by zero exception occurs when a division instruction is called with registers holding zero. To register a divide by zero exception handler, the handler address is written into memory location $014, the interrupt vector table entry that corresponds to divide by zero.

2.3.5.1-) Divide by Zero Exception Algorithm and Flowchart

A similar algorithm is implemented as for the illegal instruction exception.

2.3.5.2-) Divide by Zero Exception Assembly Code

```
*------------------------------------------------------------
* Divide by zero exception
*------------------------------------------------------------
DIV0    MOVEM.L D0/A1,-(SP)
        MOVEA.L #DIV0_MSG,A1
        MOVE.B  #13,D0
        TRAP    #15
        BSR     DF
        MOVEM.L (SP)+,D0/A1
        RTE
```

Listing 23: Divide By Zero Exception Assembly Code

2.3.6-) CHK Instruction Exception

A CHK instruction exception occurs when the CHK condition evaluates to false. The instruction is generally used to guarantee certain state before execution of a program. To register a CHK exception handler, the handler address is written into memory location $018, the interrupt vector table entry that corresponds to CHK errors.

2.3.6.1-) CHK Instruction Exception Algorithm and Flowchart

A similar algorithm is implemented as for the illegal instruction exception.

2.3.6.2-) CHK Instruction Exception Assembly Code

```
*------------------------------------------------------------
* Check instruction exception
*------------------------------------------------------------
CHK     MOVEM.L D0/A1,-(SP)
        MOVEA.L #CHK_MSG,A1
        MOVE.B  #13,D0
        TRAP    #15
        BSR     DF
        MOVEM.L (SP)+,D0/A1
        RTE
```

Listing 24: CHK Instruction Exception Assembly Code

2.3.7-) Line A and Line F Emulators

Line A and line F emulator exceptions occurs when instructions with opcodes beginning with hex A or F respectively, which run on coprocessors, do not correspond to valid instructions on the target processor. The F-line is generally used for floating point instructions. To register Line A and Line F exception handlers, the handler addresses are written into memory locations $028 and $02C respectively, the corresponding interrupt vector table entries.

2.3.7.1-) Line A and Line F Emulators Algorithm and Flowchart

A similar algorithm is implemented as for the illegal instruction exception.

2.3.7.2-) Line A and Line F Emulators Assembly Code

```
*---------------------------------------------
* Line A emulator exception
*---------------------------------------------
LNEA    MOVEM.L D0/A1,-(SP)
        MOVEA.L #LNEA_MSG,A1
        MOVE.B  #13,D0
        TRAP    #15
        BSR     DF
        MOVEM.L (SP)+,D0/A1
        RTE
```

Listing 25: Line A Emulator Exception Assembly Code

```
*---------------------------------------------
* Line F emulator exception
*---------------------------------------------
LNEF    MOVEM.L D0/A1,-(SP)
        MOVEA.L #LNEF_MSG,A1
        MOVE.B  #13,D0
        TRAP    #15
        BSR     DF
        MOVEM.L (SP)+,D0/A1
        RTE
```

Listing 26: Line E Emulator Exception Assembly Code

2.4-) User Instructional Manual Exception Handlers

See Section 2.2.1 on the HELP command.

3-) Discussion

One of the problems often encountered during this project was difficulty in debugging assembly code. Assembly, being a low-level language, is lacking in terms of development tools offered to high-level languages, such as C, C++, and Java. Debugging was very time consuming and relied on foresight and intuition.

In addition, the 3KB size restriction on the size of the full program requires judicious code reuse. When writing assembly, it can often be easier to write specific implementations for each use case instead of writing a generic subroutine. The use of a generic parser (by traversing the command table) meant that the register modify instructions required an extra space inside the instruction, taking away from the aesthetics.

4-) Feature Suggestions

One feature that should be added is the ability to print hex values right aligned and padded with zeros. Currently, using the trap 15 offered by Easy68K to convert from a hex value to hex encoded ASCII prints values of less than a given size (for example word) with only as many digits as necessary. This makes the values printed for the DF command and the exception handlers unaligned and ugly.

Better error checking should also be done to provide more helpful feedback to the user when commands are mistyped. There are three parts to this. First, the current implementation does not support lower case versions of the same commands. Second, when the command is typed without required arguments, the output given to the user is "invalid command" instead of "not enough arguments". Finally, no checking is done on hex inputs. The user may enter an invalid hex number (for example, KKK) and the program would not have been able to know that there was a mistake. A user may enter a number larger than can be accepted for a given size (for example, FFFFFFFF for a byte sized input). If an argument was missing, the ASCII subroutine would have returned zero, which means that the monitor program would not be able to tell if there was a missing input or the number entered was actually zero.

The final improvement that could be done would be more advanced commands, for example the ones supported by Linux bash terminal such as ls, cd, etc. This would make the monitor program more similar to a full fledged operating system and would be too complex for the scope of this project.

5-) Conclusion

The monitor program was successfully built with the functionality to handle most of the commands supported by the TUTOR software. It provides a way to let users manipulate processor state without directly writing assembly and modifying memory. If error checking was implemented correctly, this also makes the processor more secure and less likely to fail.

6-) References

Text I/O. (n.d.). Retrieved November 26, 2018, from http://www.easy68k.com/QuickStart/TrapTasks.htm

Thomas L. Harman, & Barbara Lawson. (1985). *The Motorola MC68000 Microprocessor Family: Assembly Language, Interface Design, and System Design*. Englewood Cliffs, N.J. 07632: Prentice-Hall, Inc.

7-) Appendix

7.1-) Appendix A: Hard coded values

```
        ORG     $1000

PROMPT  DC.B    'MONITOR441> ',0 ;prompt
INVALID DC.B    'INVALID COMMAND',0

INPUT   DS.B    80              ;buffer for
commands

COMP_TBL                        ;table of
commands
        DC.B    'HELP',0
        DS.B    2               ;padding to 6
bytes
        DC.B    'MDSP ',0
        DS.B    1
        DC.B    'SORTW ',0
        DC.B    'MM ',0
        DS.B    3
        DC.B    'MS ',0
        DS.B    3
        DC.B    'BF ',0
        DS.B    3
        DC.B    'BMOV ',0
        DS.B    1
        DC.B    'BTST ',0
        DS.B    1
        DC.B    'BSCH ',0
        DS.B    1
        DC.B    'GO ',0
        DS.B    3
        DC.B    'DF',0
        DS.B    4
        DC.B    'ECHO ',0
        DS.B    1
        DC.B    '. ',0
        DS.B    4
        DC.B    'EXIT',0
        DS.B    2

REGPC   DC.B    'PC=',0
REGSR   DC.B    'SR=',0
REGUS   DC.B    'US=',0
```

```
REGSS   DC.B    'SS=',0
REGD0   DC.B    'D0=',0
REGD1   DC.B    'D1=',0
REGD2   DC.B    'D2=',0
REGD3   DC.B    'D3=',0
REGD4   DC.B    'D4=',0
REGD5   DC.B    'D5=',0
REGD6   DC.B    'D6=',0
REGD7   DC.B    'D7=',0
REGA0   DC.B    'A0=',0
REGA1   DC.B    'A1=',0
REGA2   DC.B    'A2=',0
REGA3   DC.B    'A3=',0
REGA4   DC.B    'A4=',0
REGA5   DC.B    'A5=',0
REGA6   DC.B    'A6=',0
REGA7   DC.B    'A7=',0

REGVAL  DS.L    1
        DS.L    1
        DS.L    1
        DS.L    1
        DS.L    1
        DS.L    1
        DS.L    1
        DS.L    1
        DS.L    1
        DS.L    1
        DS.L    1
        DS.L    1
        DS.L    1
        DS.L    1
        DS.L    1
        DS.L    1
        DS.L    1
        DS.L    1
        DS.L    1
        DS.L    1
        DS.L    1

REGNAME DC.B    $D0
        DC.B    $D1
        DC.B    $D2
        DC.B    $D3
        DC.B    $D4
        DC.B    $D5
        DC.B    $D6
        DC.B    $D7
        DC.B    $A0
        DC.B    $A1
        DC.B    $A2
        DC.B    $A3
        DC.B    $A4
        DC.B    $A5
        DC.B    $A6
        DC.B    $A7

HELP1   DC.B    'HELP: displays this help message',0
HELP2   DC.B    'MDSP <address1> <address2>: outputs address and memory contents from <address1> 
to <address2>',0
HELP2A  DC.B    'MDSP <address1>: outputs address and memory from <address1> to <address + 16 
bytes',0
HELP3   DC.B    'SORTW <address1> <address2> <order>: sort block of memory between <address1> and 
<address2>, in ascending (A) or descending (D) order',0
HELP4   DC.B    'MM <address> <size>: display memory and modify/enter new data, showing byte (B), 
word (W), or longword (L) bytes',0
HELP5   DC.B    'MS <address> <data>: set memory at <address> to <data>, which is of type ASCII or 
hex',0
HELP6   DC.B    'BF <address1> <address2> <data>: fills memory between <address1> and <address2> 
with <data>, which is word size',0
HELP7   DC.B    'BMOV <address1> <address2> <address3>: move block of memory between <address1> 
and <address2> to location starting at <address3>',0
HELP8   DC.B    'BTST <address1> <address2>: test a block of memory between <address1> and 
<address2>',0
HELP9   DC.B    'BSCH <address1> <address2> <string>: search for <string> between <address1> and 
<address2>',0
HELP10  DC.B    'GO <address>: execute program at <address>',0
HELP11  DC.B    'DF: display registers and values',0
HELP12  DC.B    'ECHO <data>: prints <data> to terminal',0
HELP13  DC.B    '. <reg> <data>: put data into register specified, e.g. .D0 1000',0
HELP14  DC.B    'EXIT: terminate the program',0

SUCCESS DC.B    'BTST successful',0
FAILURE DC.B    'BTST failed',0
```

```
BADREG  DC.B    'Register not found',0

BUS_MSG  DC.B   'Bus error occurred',0
ADDR_MSG DC.B   'Address error occurred',0
ILLI_MSG DC.B   'Illegal instruction error occurred',0
DIV0_MSG DC.B   'Divide by zero error occurred',0
CHK_MSG  DC.B   'Check error occurred',0
PRIV_MSG DC.B   'Privilege violation error occurred',0
LNEA_MSG DC.B   'Line 1010 error occurred',0
LNEF_MSG DC.B   'Line 1111 error occurred',0
```

7.2-) Appendix B: Setup subroutines

```
*---------------------------------------------
* Start here: set up and call to main
*---------------------------------------------
START   BSR     SAVE            ;save
registers in memory
        BSR     SETUP           ;set up
exceptions
        BSR     MAIN            ;call main

        MOVE.B  #9,D0           ;exit program
        TRAP    #15
*---------------------------------------------
* Setup exception handler
*---------------------------------------------
SETUP   MOVE.L  #BUS,$008
        MOVE.L  #ADDR,$00C
        MOVE.L  #ILLI,$010
        MOVE.L  #DIV0,$014
        MOVE.L  #CHK,$018
        MOVE.L  #PRIV,$020
        MOVE.L  #LNEA,$028
        MOVE.L  #LNEF,$02C
        RTS

*---------------------------------------------
* SAVE: save registers
*---------------------------------------------
SAVE    MOVEM.L D0/A0/A1,-(SP)
        MOVEM.L D0/A0/A1,-(SP)
        MOVEA.L #REGVAL,A1

        MOVE.L  0(PC),(A1)+
        MOVE.W  #0,(A1)+
        MOVE.W  SR,(A1)+
        MOVE.L  A7,(A1)+        ;SSP
        MOVE.L  SP,(A1)+        ;USP

        MOVEA.L #REGVAL,A1
        ADDA.L  #16,A1

        MOVE.L  D0,(A1)+
        MOVE.L  D1,(A1)+
        MOVE.L  D2,(A1)+
        MOVE.L  D3,(A1)+
        MOVE.L  D4,(A1)+
        MOVE.L  D5,(A1)+
        MOVE.L  D6,(A1)+
        MOVE.L  D7,(A1)+
        MOVE.L  A0,(A1)+

        MOVEM.L (SP)+,D0/A0/A1
        MOVEA.L #REGVAL,A0
        ADDA.L  #52,A0
        MOVE.L  A1,(A0)+


        MOVE.L  A2,(A0)+
        MOVE.L  A3,(A0)+
        MOVE.L  A4,(A0)+
        MOVE.L  A5,(A0)+
        MOVE.L  A6,(A0)+
        MOVE.L  A7,(A0)+

        MOVEM.L (SP)+,D0/A0/A1
        RTS
```

7.3-) Appendix C: ASCII subroutine

```
*------------------------------------------------------------
* DIGIT: convert single ascii digit to hex
* Input: single ascii character in register D0
* Output: hex value in register D0
*------------------------------------------------------------
DIGIT   CMP.B   #$40,D0
        BGT     HIGHER
        SUBI.B  #$30,D0
        BRA     EXITDIGIT
```

```
HIGHER  SUBI.B  #$37,D0
EXITDIGIT RTS

*-----------------------------------------------------------
* ASCII: convert ascii to hex
* Input: pointer to start of string in A1
* Output: hex value in register D1
*-----------------------------------------------------------
ASCII   MOVEM.L D0,-(SP)        ;save registers
        CLR.L   D0              ;clear for digit manipulation
        CLR.L   D1              ;clear for sum

CHAR    MOVE.B  (A1)+,D0        ;move digit
        TST.B   D0              ;continue until end of string
        BEQ     EXITASCII
        CMP.B   #$20,D0         ;continue until empty space
        BEQ     EXITASCII

        BSR     DIGIT

        MULS.W  #$10,D1         ;multiply by 10
        ADD.L   D0,D1           ;add digit (Horner)

        BRA     CHAR

EXITASCII MOVEM.L (SP)+,D0      ;restore registers
        RTS
```

7.3-) Appendix D: print help subroutines

```
*---------------------------------------------
* SPACE: print space
*---------------------------------------------
SPACE   MOVEM.L D0/D1,-(SP)

        MOVE.L  #$20,D1         ;print space
        MOVE.B  #6,D0
        TRAP    #15

        MOVEM.L (SP)+,D0/D1
        RTS
```

```
*---------------------------------------------
* CRLF: print carriage return followed by
line feed
*---------------------------------------------
CRLF    MOVEM.L D0/D1,-(SP)

        MOVE.B  #$D,D1          ;print cr
        MOVE.B  #6,D0
        TRAP    #15
        MOVE.B  #$A,D1          ;print lf
        MOVE.B  #6,D0
        TRAP    #15

        MOVEM.L (SP)+,D0/D1
        RTS
```