# ECE 441
# Microprocessors

Instructor: Dr. Jafar Saniie

Final Project Report:

**MONITOR PROJECT**

04/27/2018

By: Chang Liu

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**

In this project, I implemented a monitor program and run it on Easy68k. This program iteratively receives user inputs and interprets the inputs to commands. For each command I wrote a subroutine, so the interpreter can search commands in a linked list and direct the program to corresponding subroutines. When a command is received, I load the starting address of user input to an address register, then start comparing the input string with strings in a linked list. If any element in the list matches, we get a function pointer to the subroutine, and use JSR command to run the subroutine.

Each subroutine first reads input parameters (if any), then finishes its task. After the task ends, a RTS command returns the program to interpreter, which immediately starts the next loop.

In the following text, detailed logic of interpreters and debugger commands will be shown. Several exception handlers are also implemented and described in this report.

## 1. Introduction

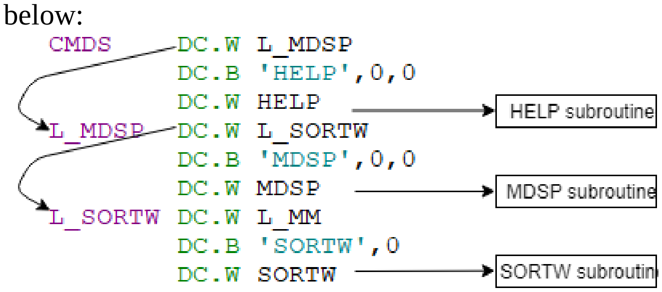
Object of this design is to allow users manipulate the simulated computer in Easy68k. To satisfy basic requirements of manipulation, I implemented 15 commands and 8 exception handlers. When the monitor program is started, a prompt “MONITOR441>” will show on screen. Users could type a command and press enter. Unless EXIT command is entered, the program will print the prompt again after processing of a command is finished.

Exception handlers are executed only when exceptions are enabled in simulator and an exception is generated. BERR and AERR prints the exception type, SSW, BA, IR, and formatted registers, while other handlers print only exception type and formatted registers.

## 2. Monitor Program

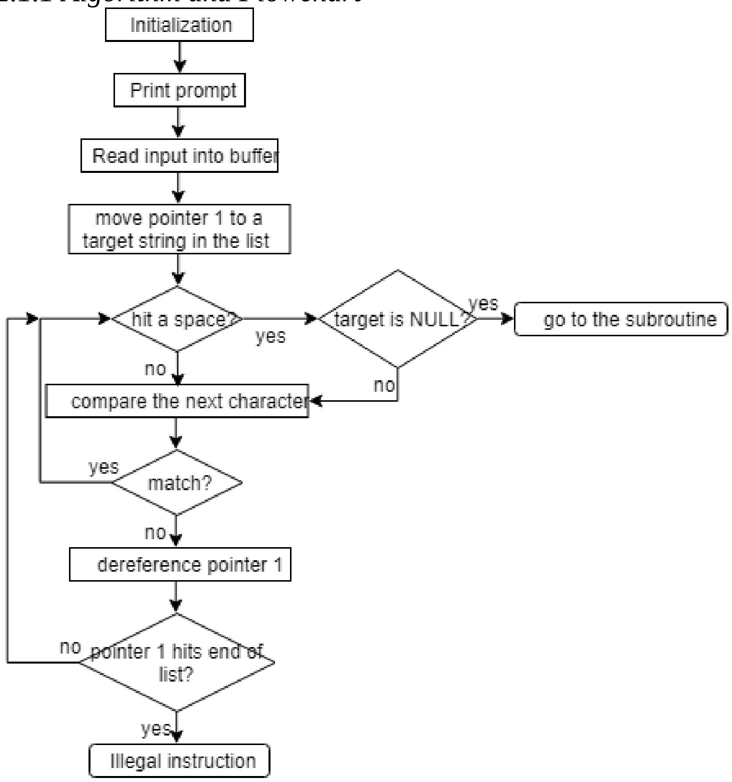
### 2.1 Command Interpreter

The interpreter first initializes some important data, including stack pointer and exception handlers. Then a prompt is printed, and A1 is assigned to the starting address of a buffer. User input is received and stored into the buffer. Instead of using two lookup tables, I use a singly linked list with structures as elements. The form of linked list is shown below:




Each element in this list is a structure containing three parts: 1. a pointer to next element 2. a string 3. a pointer to command subroutine. Now that we have the user input, we can start comparing from the first element i.e. the ‘HELP’. If all characters before a space match, we know it’s a successful match. In this case, we dereference the pointer, making it point to the next element, then subtract it by 2. For example, if we successfully match the command with ‘HELP’, we can point the address register to L_MDSP, then subtract 2 to get ‘DC.W HELP’.

*2.1.1 Algorithm and Flowchart*




*2.1.2 Command Interpreter Assembly Code*

```
      ORG   (*+1)&-2           ; Force Word alignment
CMDS        DC.W L_MDSP
            DC.B 'HELP',0,0
            DC.W HELP
L_MDSP      DC.W L_SORTW
            DC.B 'MDSP',0,0
            DC.W MDSP
L_SORTW     DC.W L_MM
            DC.B 'SORTW',0
            DC.W SORTW
L_MM        DC.W L_MS
            DC.B 'MM',0,0
            DC.W MM
L_MS        DC.W L_BF
            DC.B 'MS',0,0
            DC.W MS
L_BF        DC.W L_BMOV
            DC.B 'BF',0,0
            DC.W BF
L_BMOV      DC.W L_BTST
            DC.B 'BMOV',0,0
            DC.W BMOV
L_BTST      DC.W L_BSCH
            DC.B 'BTST',0,0
            DC.W BTST
L_BSCH      DC.W L_GO
            DC.B 'BSCH',0,0
            DC.W BSCH
L_GO        DC.W L_DF
            DC.B 'GO',0,0
            DC.W GO
L_DF        DC.W L_EXIT
```

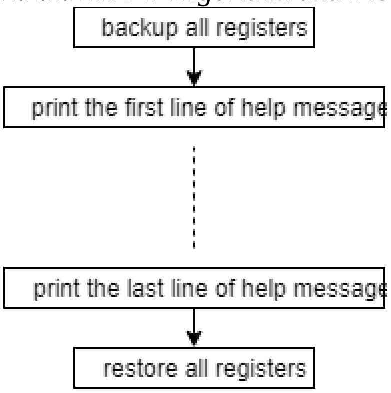
```
            DC.B 'DF',0,0
            DC.W DF
L_EXIT      DC.W L_ENCR
            DC.B 'EXIT',0,0
            DC.W EXIT
L_ENCR      DC.W L_DECR
            DC.B 'ENCR',0,0
            DC.W ENCR
L_DECR      DC.W L_RAND
            DC.B 'DECR',0,0
            DC.W DECR
L_RAND      DC.W L_END
            DC.B 'RAND',0,0
            DC.W RAND
L_END

      ORG   (*+1)&-2                  ; Force Word alignment
START:                                ; first instruction of program
            MOVE.L      #BERR,$8      ; initialize vectors
            MOVE.L      #AERR,$C
            MOVE.L      #ILLI,$10
            MOVE.L      #PRIV,$20
            MOVE.L      #DIVZ,$14
            MOVE.L      #CHKI,$18
            MOVE.L      #LINA,$28
            MOVE.L      #LINF,$2C
            MOVEA.L     #$3000,A7
            LEA         PROMPT,A1
            MOVE.B      #14,D0
            TRAP  #15
            LEA         INPUT,A1      ; input buffer
            MOVE.B      #2,D0
            TRAP  #15
            MOVEA.L     A1,A2         ; A2 is pointer of input
            LEA         CMDS,A3       ; A3 is pointer in list
            LEA         L_END,A4      ; A4 is end of list
L_TST       MOVEA.L     A3,A5
            ADDA.L      #2,A5
CMPNEXT     MOVE.B      (A2),D1       ; backup the tested character
            CMPI.B      #$20,D1       ; if the cursor hits a space,
            BNE         CMP_CNT
            CMPI.B      #0,(A5)       ; and the target instruction ends,
            BEQ         Y_MATCH       ; it's a match
CMP_CNT     CMPM.B      (A5)+,(A2)+ ; compare the two characters
            BNE         N_MATCH
            CMPI.B      #0,D1         ; if same, and target instruction ends,
            BEQ         Y_MATCH       ; it's a match
            BRA         CMPNEXT       ; otherwise, loop back
N_MATCH     MOVEA.L     A1,A2         ; cursor back to start point
            MOVEA.W     (A3),A3       ; A3 point to the head of next element
            CMP.W       #L_END,A3     ; if A3 reaches end of list,
            BEQ         N_INST        ; it's not a valid instruction
            BRA         L_TST
Y_MATCH     MOVEA.W     (A3),A3
            MOVEA.W     -2(A3),A5     ; deinference A3 and minus 2, we get address of
subroutine
            JSR         (A5)
            BRA         START         ; end the execution
```

2.2 Debugger Commands

2.2.1 HELP

This command prints help messages with regard to command syntax, and a brief description of all commands.

*2.2.1.1 HELP Algorithm and Flowchart*




*2.2.1.2 HELP Assembly Code*

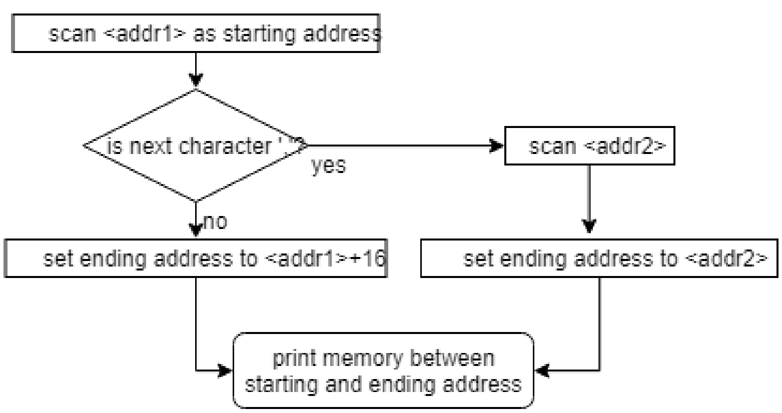
```
M_HELP      DC.B 'Monitor help message: ',$D,$A,$A,0
M_HP1       DC.B 'MDSP <addr1>,[<addr2>] --Display a memory block',$D,$A,0
M_HP2       DC.B 'SORTW <addr1>,<addr2>;(A/D) --Sort a block of memory',$D,$A,0
M_HP3       DC.B 'MM <addr1>;(B/W/L) --Display memory and enter new data',$D,$A,0
M_HP4       DC.B 'MS <addr1>;(number/text in single quotes) --Set a block of
      memory',$D,$A,0
M_HP5       DC.B 'BF <addr1>,<addr2>;(number) --Fill the memory block with a
      number',$D,$A,0
M_HP6       DC.B 'BMOV <addr1>,<addr2>;<addr3> --Move block between addr1 and addr2 to
      location addr3',$D,$A,0
M_HP7       DC.B 'BTST <addr1>,<addr2> --Test a block of memory',$D,$A,0
M_HP8       DC.B 'BSCH <addr1>,<addr2>;(text in single quotes) --Search for the text
      within memory block',$D,$A,0
M_HP9       DC.B 'GO <addr1> --Start executing at address addr1',$D,$A,0
M_HP10      DC.B 'DF --Display formatted registers',$D,$A,0
M_HP11      DC.B 'EXIT --Exit the monitor program',$D,$A,0
M_HP12      DC.B 'ENCR <addr1>,<addr2>;<key>,<IV> --Encrypt the memory block with key and
      IV',$D,$A,0
M_HP13      DC.B 'DECR <addr1>,<addr2>;<key>,<IV> --Decrypt the memory block with key and
      IV',$D,$A,0
M_HP14      DC.B 'RAND --Generate a random number between 0 and 99',$D,$A,0

HELP        MOVEM.L     D0-D7/A0-A6,-(SP)
            LEA         M_HELP,A1
            MOVE.B      #14,D0
            TRAP  #15
            LEA         M_HP1,A1
            MOVE.B      #14,D0
            TRAP  #15
            LEA         M_HP2,A1
            MOVE.B      #14,D0
            TRAP  #15
            LEA         M_HP3,A1
            MOVE.B      #14,D0
            TRAP  #15
            LEA         M_HP4,A1
            MOVE.B      #14,D0
            TRAP  #15
            LEA         M_HP5,A1
            MOVE.B      #14,D0
            TRAP  #15
            LEA         M_HP6,A1
```

```
            MOVE.B      #14,D0
            TRAP  #15
            LEA         M_HP7,A1
            MOVE.B      #14,D0
            TRAP  #15
            LEA         M_HP8,A1
            MOVE.B      #14,D0
            TRAP  #15
            LEA         M_HP9,A1
            MOVE.B      #14,D0
            TRAP  #15
            LEA         M_HP10,A1
            MOVE.B      #14,D0
            TRAP  #15
            LEA         M_HP11,A1
            MOVE.B      #14,D0
            TRAP  #15
            LEA         M_HP12,A1
            MOVE.B      #14,D0
            TRAP  #15
            LEA         M_HP13,A1
            MOVE.B      #14,D0
            TRAP  #15
            LEA         M_HP14,A1
            MOVE.B      #14,D0
            TRAP  #15
            MOVEM.L     (SP)+,D0-D7/A0-A6
            RTS
```

### 2.2.2 MDSP

This command receives one or two parameters. If only one parameter (<addr1>) is detected, 16 bytes of memory is printed, starting from <addr1>. If two parameters (<addr1>, <addr2>) are detected, printed memory ranges from <addr1> to <addr2>, but not include <addr2>. I wrote a subroutine P_MM that receives two addresses and prints memory between them, so MDSP subroutine only has to sets the two addresses and call P_MM.

#### *2.2.2.1 MDSP Algorithm and Flowchart*




#### *2.2.2.2 MDSP Assembly Code*

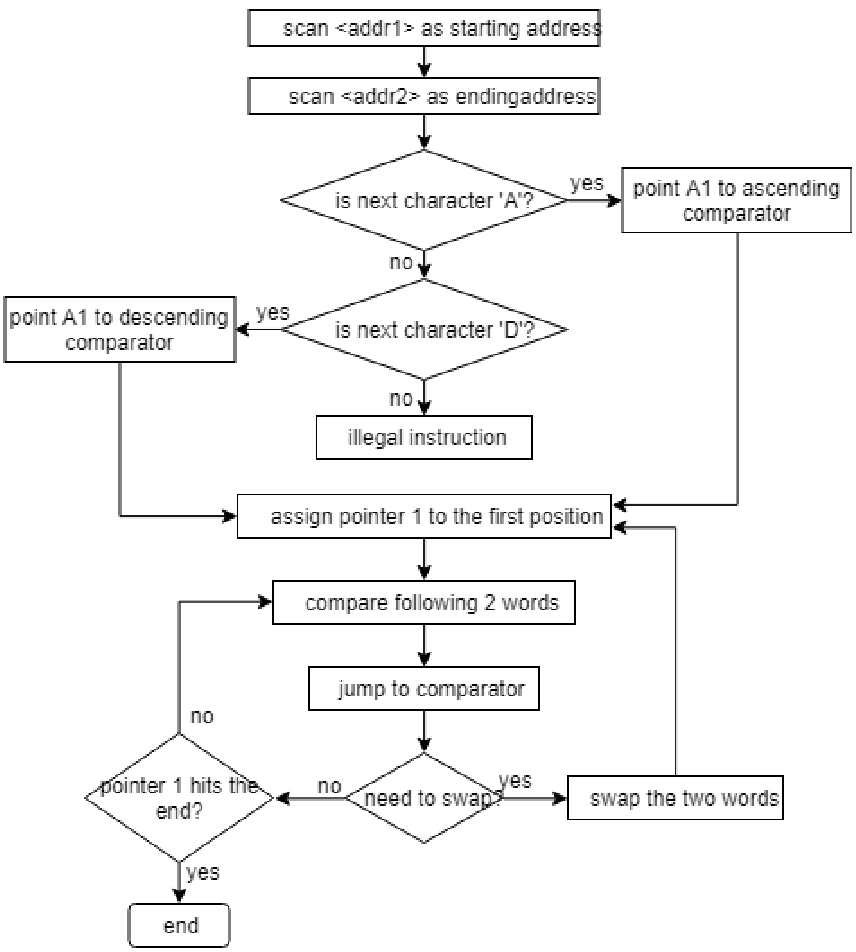
```
MDSP        MOVEM.L     D0-D7/A0-A6,-(SP)
            BSR         SC_NXT
            CMPI.B      #$2C,(A2)   ; if ',' is found,
            BEQ         MD_2N       ; 2 parameters are received
            MOVEA.L     D7,A5       ; A5 points to beginning of block
            MOVEA.L     A5,A6
            ADDA.L      #16,A6            ; A6 points to ending of block, i.e. A5+16
            CLR.L       D4
            BSR         P_MM        ; print content between A5 and A6
```

```
             BRA          MD_END
MD_2N        MOVEA.L      D7,A5        ; A5 has the beginning address of memory block
             BSR          SC_NXT
             MOVEA.L      D7,A6        ; A6 has the ending address of memory block
             CLR.L        D4
             BSR          P_MM
MD_END       LEA          M_CRLF,A1    ; print an empty line to finish
             MOVE.B       #13,D0
             TRAP  #15
             MOVEM.L      (SP)+,D0-D7/A0-A6
             RTS
```

### 2.2.3 SORTW

Because there are two conditions in this program: ascending and descending, so I use A1 as a function pointer that points to any one of two comparators. A comparator receives condition code and determines whether to swap. In this way, repetition of code is avoided. This subroutine uses bubble sort, which has two loops. Neighboring words are compared, then the control is immediately passed to comparator. Comparator uses BLT or BHI to determine whether returning control to swap program (DO_SWAP) or not (NO_SWAP).

#### *2.2.3.1 SORTW Algorithm and Flowchart*




#### *2.2.3.2 SORTW Assembly Code*

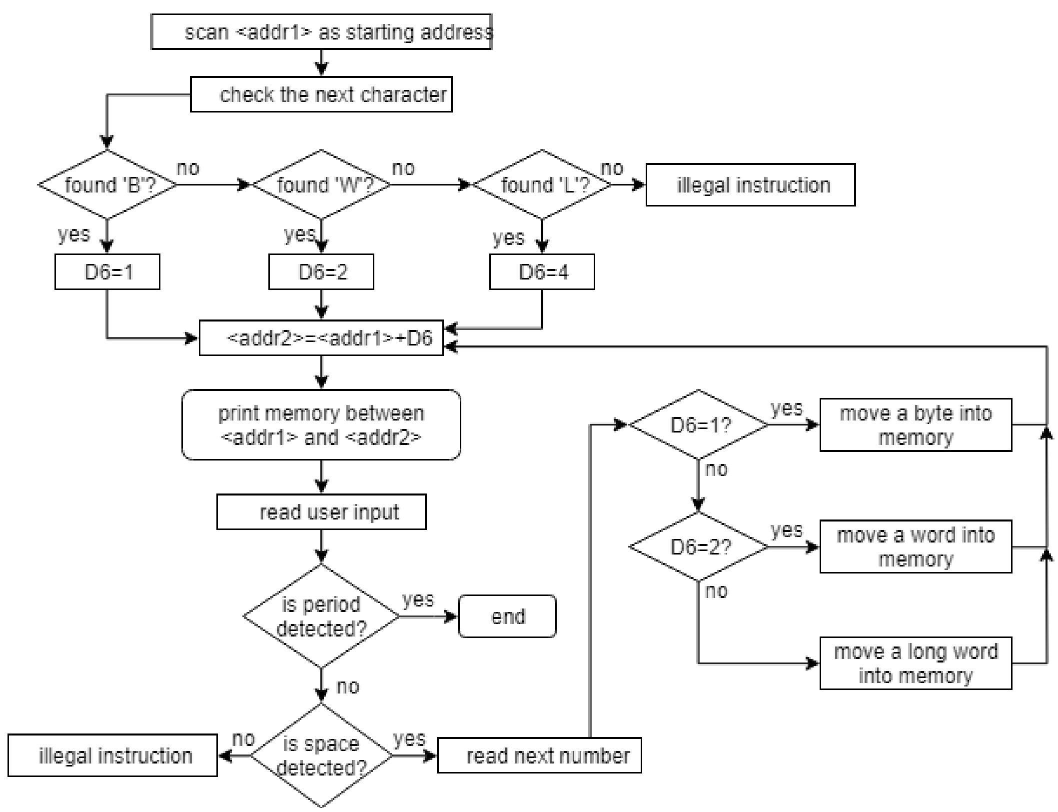
```
SORTW        MOVEM.L      D0-D7/A0-A6,-(SP)
             BSR          SC_NXT
             MOVEA.L      D7,A5
             MOVEA.L      A5,A4
             BSR          SC_NXT
             MOVEA.L      D7,A6
             SUBQ.L       #2,A6
             CMPI.B       #$41,1(A2)  ; check if the input is 'A'
```

```
              BEQ          ASCE
              CMPI.B       #$44,1(A2)  ; check if the input is 'D'
              BEQ          DESC
              BRA          N_INST            ; if neither matches, illegal instruction
ASCE          LEA          CMPER_A,A1  ; load ascending comparator to A1
              BRA          SRT_LO1
DESC          LEA          CMPER_D,A1  ; load descending comparator to A1
SRT_LO1       MOVEA.L      A4,A5       ; A5 points to the first position
SRT_LO2       CMP.W        (A5)+,(A5)+
              JMP          (A1)        ; A1 is a function pointer to comparator
NO_SWAP       SUBQ.L       #2,A5       ; no need to swap, move to the next position
              CMP.L        A5,A6       ; check for the end of list
              BNE          SRT_LO2     ; go to the second loop
              BRA          SRT_END
DO_SWAP       MOVE.L       -(A5),D2    ; swap the two words
              SWAP.W       D2
              MOVE.L       D2,(A5)
              BRA          SRT_LO1     ; go to the first loop
SRT_END       MOVEM.L      (SP)+,D0-D7/A0-A6
              RTS
CMPER_A       BLT          DO_SWAP           ; comparator for ascending
              BRA          NO_SWAP
CMPER_D       BHI          DO_SWAP           ; comparator for descending
              BRA          NO_SWAP
```

2.2.4 MM

This command is similar to the MM command in TUTOR. It receives an address and a character (B, W or L), which indicates the length of each modification. Then length information is stored in a data register, functioning in two ways. First, the data register acts as an addend when moving address register to the next location. Second, when moving new value into memory, the data register is a signal to direct control to different subroutines.

*2.2.4.1 MM Algorithm and Flowchart*



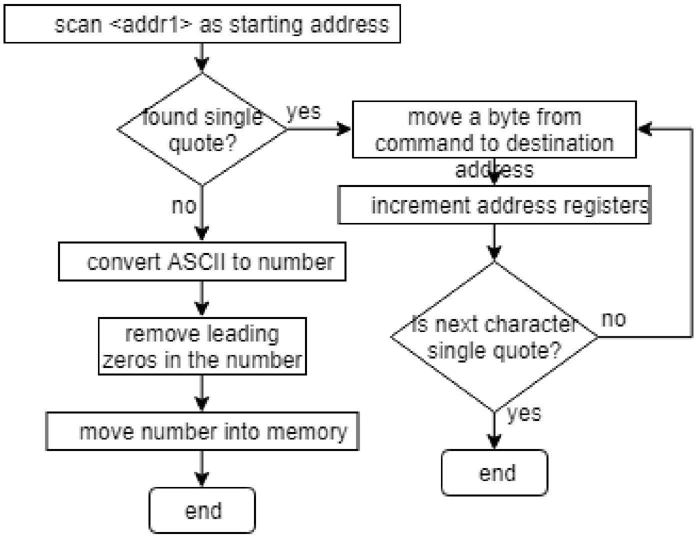
*2.2.4.2 MM Assembly Code*

```
MM          MOVEM.L     D0-D7/A0-A6,-(SP)
            BSR         SC_NXT
            MOVEA.L     D7,A3
            CMPI.B      #$3B,(A2)+  ; check ';'
            BNE         N_INST
            CMPI.B      #$42,(A2)   ; check 'B'
            BEQ         MM_LOB
            CMPI.B      #$57,(A2)   ; check 'W'
            BEQ         MM_LOW
            CMPI.B      #$4C,(A2)   ; check 'L'
            BEQ         MM_LOL
            BRA         N_INST
MM_CT1      MOVEA.L     A3,A5
            MOVEA.L     A5,A6
            ADDA.L      D6,A6       ; move A6 to next position
            MOVE.L      #1,D4       ; print without space
            BSR         P_MM
            LEA         TXT_BUF,A1  ; receive input
            MOVE.B      #2,D0
            TRAP  #15
            MOVEA.L     A1,A2
            CMP.B       #$2E,(A2)   ; check '.'
            BEQ         MM_END
            CMP.B       #$20,(A2)   ; check ' '
            BNE         N_INST
            BSR         SC_NXT
            CMPI.B      #1,D6       ; use D6 to determine length
            BEQ         MM_MOB
            CMPI.B      #2,D6
            BEQ         MM_MOW
            CMPI.B      #4,D6
            BEQ         MM_MOL
MM_CT2      BRA         MM_CT1
MM_END      LEA         M_CRLF,A1   ; print an empty line to finish
            MOVE.B      #13,D0
            TRAP  #15
            MOVEM.L     (SP)+,D0-D7/A0-A6
            RTS
MM_LOB      MOVEQ.L     #1,D6       ; load value to D6
            BRA         MM_CT1
MM_LOW      MOVEQ.L     #2,D6
            BRA         MM_CT1
MM_LOL      MOVEQ.L     #4,D6
            BRA         MM_CT1
MM_MOB      MOVE.B      D7,(A3)+
            BRA         MM_CT2
MM_MOW      MOVE.W      D7,(A3)+
            BRA         MM_CT2
MM_MOL      MOVE.L      D7,(A3)+
            BRA         MM_CT2
```

2.2.5 MS

The MS command determines input type in the follow way: if the input is wrapped by ‘ ’, it is treated as a string. Otherwise, it is treated as a number. The command receives an address and the content to be set into memory. If the input is string, there is no limit in length. String is moved into memory byte by byte. If the input is number, the maximum length is 8 hexadecimal digits. The ASCII represented number is converted into hexadecimal number, then stored into memory.

*2.2.5.1 MS Algorithm and Flowchart*



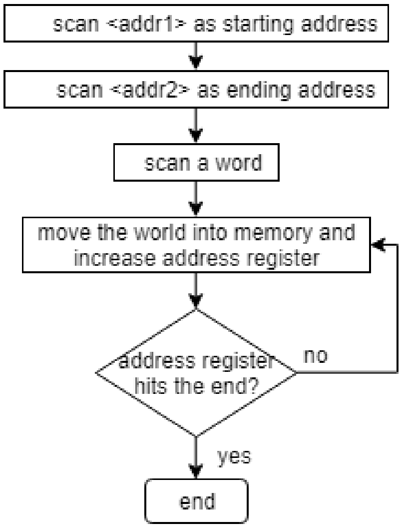

*2.2.5.2 MS Assembly Code*

```
MS          MOVEM.L     D0-D7/A0-A6,-(SP)
            BSR         SC_NXT
            MOVEA.L     D7,A3       ; A3 has the starting address
            CMPI.B      #$2C,(A2)+
            BNE         N_INST
            CMPI.B      #$27,(A2)
            BNE         MS_HEX            ; if the first char is not ''', it's hex
            ADDQ.L      #1,A2       ; otherwise, it's ascii
MS_LO1      MOVE.B      (A2)+,(A3)+ ; move ascii into destination memory location
            CMPI.B      #$27,(A2)
            BNE         MS_LO1
            BRA         MS_END
MS_HEX      MOVEA.L     A2,A5
MS_LO2      CMPI.B      #0,(A2)           ; if '' is found,
            BEQ         MS_E1       ; end the loop
            ADDQ.L      #1,A2
            BRA         MS_LO2            ; otherwise, loop
MS_E1       MOVEA.L     A2,A6
            BSR         ATOI
            MOVE.L      D7,D6       ; D7 has the hex number, D6 is a copy
MS_LO3      ROL.L #8,D6       ; test the highest byte
            CMPI.B      #0,D6       ; if it's zero
            BNE         MS_E2
            LSL.L #8,D7       ; remove the leading zero byte in D7
            BRA         MS_LO3
MS_E2       MOVE.L      D7,(A3)
MS_END      LEA         M_CRLF,A1   ; print an empty line to finish
            MOVE.B      #13,D0
            TRAP  #15
            MOVEM.L     (SP)+,D0-D7/A0-A6
            RTS
```

2.2.6 BF

This command receives three parameters. The first two are starting and ending addresses. The third parameter is a word. A loop moves the word to all even locations in the memory block.

*2.2.6.1 BF Algorithm and Flowchart*



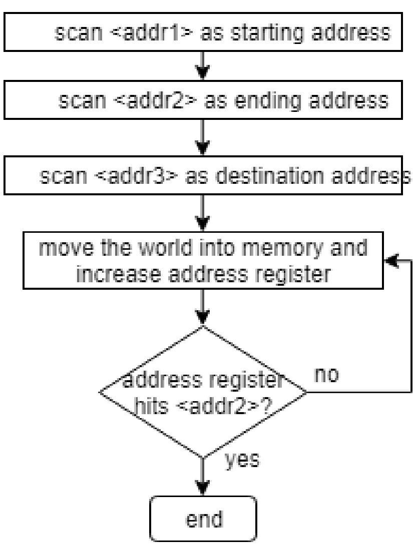

*2.2.6.2 BF Assembly Code*

```
BF          MOVEM.L     D0-D7/A0-A6,-(SP)
            BSR         SC_NXT
            MOVEA.L     D7,A5        ; A5 and A6 has starting and ending address
            BSR         SC_NXT
            MOVEA.L     D7,A6
            BSR         SC_NXT
BF_LO       MOVE.W      D7,(A5)+     ; move the word into memory
            CMP.L A5,A6
            BNE         BF_LO
            LEA         M_CRLF,A1
            MOVE.B      #13,D0
            TRAP  #15
            MOVEM.L     (SP)+,D0-D7/A0-A6
            RTS
```

2.2.7 BMOV

This command receives three addresses. The first two are starting and ending address of source. The third address is starting address of destination. A loop moves every byte from source to destination.

*2.2.7.1 BMOV Algorithm and Flowchart*



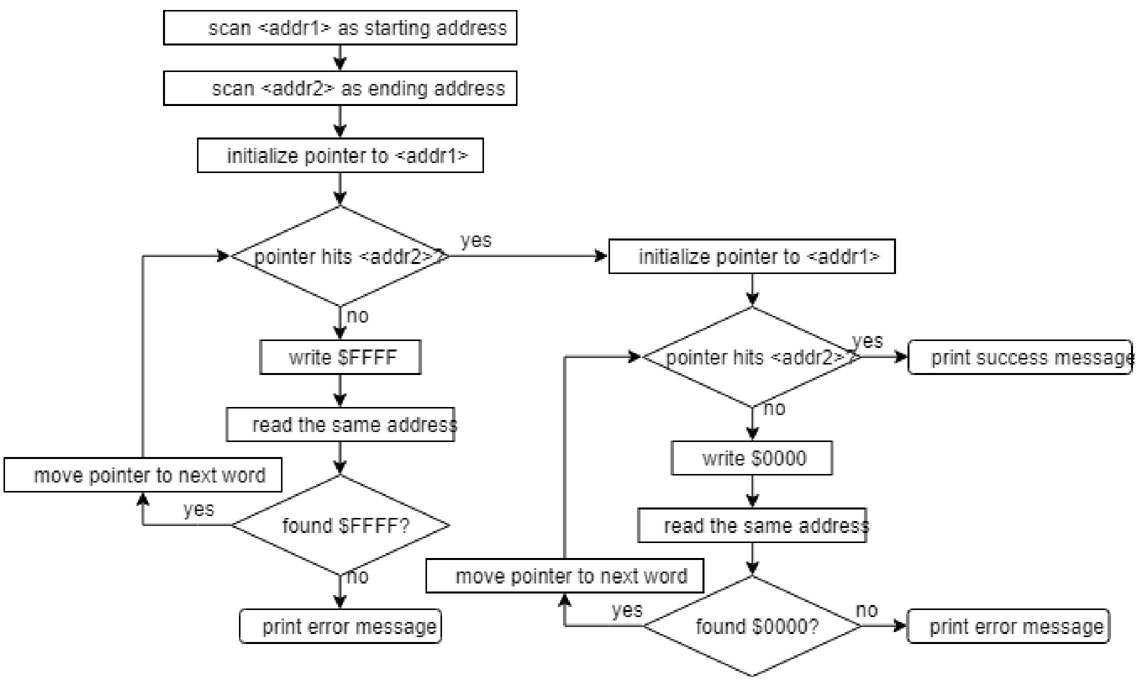
*2.2.7.2 BMOV Assembly Code*

```
BMOV          MOVEM.L      D0-D7/A0-A6,-(SP)
              BSR          SC_NXT
              MOVEA.L      D7,A5        ; A5 and A6 has starting and ending address
              BSR          SC_NXT
              MOVEA.L      D7,A6
              BSR          SC_NXT
              MOVEA.L      D7,A4
BM_LO         MOVE.B       (A5)+,(A4)+ ; move a byte
              CMP.L        A5,A6
              BNE          BM_LO
              LEA          M_CRLF,A1
              MOVE.B       #13,D0
              TRAP  #15
              MOVEM.L      (SP)+,D0-D7/A0-A6
              RTS
```

2.2.8 BTST

To test whether the memory is working correctly, I use two loops to write and read the memory. The first loop writes $FFFF to each word and check, while the second loop writes $0000. If inequality at any location is detected, the program will print location, wrote value and read value.

*2.2.8.1 BTST Algorithm and Flowchart*




*2.2.8.2 BTST Assembly Code*

```
BTST          MOVEM.L      D0-D7/A0-A6,-(SP)
              BSR          SC_NXT
              MOVEA.L      D7,A5
              BSR          SC_NXT
              MOVEA.L      D7,A6
              MOVEA.L      A5,A3        ; A3 has a copy of starting address
              LEA          M_BT_F,A4
BT_LO1        CMP.L A5,A6
              BEQ          BT_E1
              MOVE.W       #$FFFF,(A5)
              CMPI.W       #$FFFF,(A5) ; write and read $FFFF
              BEQ          BT_C1
              BRA          BT_ERR
BT_C1         ADDA.L       #2,A5
              BRA          BT_LO1
```

```
BT_E1       MOVEA.L     A3,A5
            LEA         M_BT_0,A4
BT_LO2      CMP.L A5,A6
            BEQ         BT_E2
            MOVE.W      #0,(A5)
            CMPI.W      #0,(A5)          ; write and read $0000
            BEQ         BT_C2
            BRA         BT_ERR
BT_C2       ADDA.L      #2,A5
            BRA         BT_LO2
BT_E2       LEA         M_BT_Y,A1   ; print success message
            MOVE.B      #13,D0
            TRAP  #15
BT_END      LEA         M_CRLF,A1
            MOVE.B      #13,D0
            TRAP  #15
            MOVEM.L     (SP)+,D0-D7/A0-A6
            RTS
BT_ERR      LEA         M_BT_FA,A1  ; print error message
            MOVE.B      #14,D0
            TRAP  #15
            MOVE.L      A5,D1
            MOVE.L      #16,D2
            MOVE.B      #15,D0
            TRAP  #15
            LEA         M_BT_W,A1   ; 'wrote:'
            MOVE.B      #14,D0
            TRAP  #15
            MOVEA.L     A4,A1
            MOVE.B      #14,D0
            TRAP  #15
            LEA         M_BT_R,A1   ; 'read:'
            MOVE.B      #14,D0
            TRAP  #15
            MOVEA.L     A5,A6
            ADDA.L      #2,A6
            MOVE.L      #1,D4
            BSR         P_MM
            BRA         BT_END
```

2.2.9 BSCH

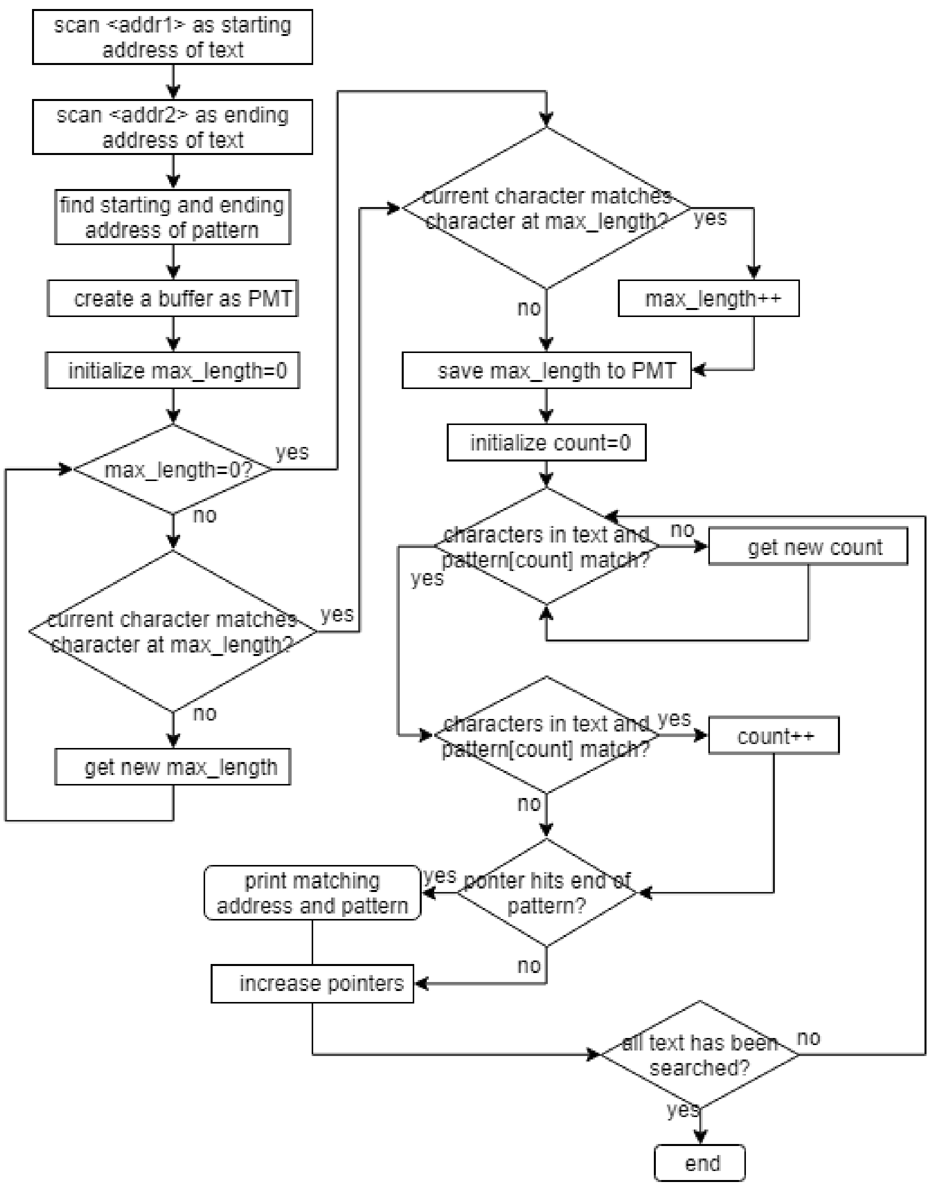
I translated a java code to assembly code, which use KMP algorithm. Advantage of this algorithm is to avoid repetitive comparison. If mismatching happens while two parts within the substring are the same, we can save computation by skipping the same part, instead of start over. The java code I referenced [1] is shown below:

```
int[] computeLspTable(String pattern) {
    int[] lsp = new int[pattern.length()];
    lsp[0] = 0;  // Base case
    for (int i = 1; i < pattern.length(); i++) {
        // Start by assuming we're extending the previous LSP
        int j = lsp[i - 1];
        while (j > 0 && pattern.charAt(i) != pattern.charAt(j))
            j = lsp[j - 1];
        if (pattern.charAt(i) == pattern.charAt(j))
            j++;
        lsp[i] = j;
    }
    return lsp;
}
int search(String pattern, String text) {
    int[] lsp = computeLspTable(pattern);
```

```
        int j = 0;  // Number of chars matched in pattern
        for (int i = 0; i < text.length(); i++) {
            while (j > 0 && text.charAt(i) != pattern.charAt(j)) {
                // Fall back in the pattern
                j = lsp[j - 1];  // Strictly decreasing
            }
            if (text.charAt(i) == pattern.charAt(j)) {
                // Next char matched, increment position
                j++;
                if (j == pattern.length())
                    return i - (j - 1);
            }
        }
        return -1;  // Not found
    }
```

The computeLspTable(String pattern) method returns a longest suffix-prefix table, also called partial match table (PMT). The computeLspTable(String pattern, String text) method searches matching locations using PMT. To translate the reference code into assembly code, I focused on the control flow such as if and while, and expressed them using branch commands.

*2.2.9.1 BSCH Algorithm and Flowchart*



*2.2.9.2 BSCH Assembly Code*

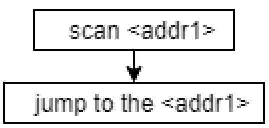
```
BSCH  MOVEM.L     D0-D7/A0-A6,-(SP)
            BSR         SC_NXT
            MOVEA.L     D7,A5
            BSR         SC_NXT
            MOVEA.L     D7,A6       ; A5 and A6 define start and end of text
            CMPI.B      #$3B,(A2)+
            BNE         N_INST            ; check ';'
            CMPI.B      #$27,(A2)+
            BNE         N_INST            ; check '''
            MOVEA.L     A2,A3
            ADDA.L      #1,A3
            CMPI.B      #$27,(A3)+
            BEQ         N_INST
BS_LO1      CMPI.B      #$27,(A3)+
            BNE         BS_LO1            ; A2 and A3 define start and end of pattern
            SUBA.L      #1,A3
            MOVE.B      #0,(A3)           ; prepare pattern to output
            MOVE.L      A3,D3
            SUB.L A2,D3       ; D3 is the length of pattern
* calculate partial match table
            LEA         PMT,A0            ; A0 is the start of PMT
            LEA         PMT,A1
            CLR.L D0                ; D0 is maximum length of last substring
            MOVEA.L     A2,A4       ; A4 is the cursor in pattern
            ADDA.L      #1,A4
            MOVE.B      #0,(A1)+
BS_LO2      CMPI.L      #0,D0       ; if maximum length=0 or contents are the same
            BEQ         BS_CNT1
            MOVE.B      (A4),D1
            CMP.B (A2,D0),D1
            BEQ         BS_CNT1           ; do not loop
            MOVE.B      -1(A0,D0),D0      ; otherwise, get new maximum length
            BRA         BS_LO2            ; loop
BS_CNT1     MOVE.B      (A4)+,D1
            CMP.B (A2,D0),D1  ; if contents are the same, increase maximum length
            BNE         BS_CNT2
            ADDQ.B      #1,D0
BS_CNT2     MOVE.B      D0,(A1)+
            CMPA.L      A4,A3
            BNE         BS_LO2
* find matching positions using PMT
            CLR.L D0                ; D0 is a counter in pattern
BS_LO3      CMPI.L      #0,D0       ; if counter=0 or contents are the same
            BEQ         BS_CNT3
            MOVE.B      (A5),D1
            CMP.B (A2,D0),D1
            BEQ         BS_CNT3           ; do not loop
            MOVE.B      -1(A0,D0),D0      ; otherwise, get the new counter
            BRA         BS_LO3
BS_CNT3     MOVE.B      (A5),D1
            CMP.B (A2,D0),D1
            BNE         BS_CNT4
            ADDQ.B      #1,D0       ; if contents are the same, increase counter
BS_CNT4     CMP.B D0,D3       ; check if the whole pattern is compared
            BNE         BS_CNT5
            MOVEM.L     D0,-(SP)
            MOVE.L      A5,D1       ; print address and pattern
            SUB.L D3,D1
            MOVE.B      #16,D2
```

```
              MOVE.B      #15,D0
              TRAP  #15
              LEA         M_SPQ,A1
              MOVE.B      #14,D0
              TRAP  #15
              MOVEA.L     A2,A1
              MOVE.B      #14,D0
              TRAP  #15
              LEA         M_QCRLF,A1
              MOVE.B      #14,D0
              TRAP  #15
              MOVEM.L     (SP)+,D0
              MOVE.B      -1(A0,D0),D0
BS_CNT5       ADDA.L      #1,A5
              CMPA.L      A5,A6
              BNE         BS_LO3
BS_END        LEA         M_CRLF,A1
              MOVE.B      #13,D0
              TRAP  #15
              MOVEM.L     (SP)+,D0-D7/A0-A6
              RTS
```

## 2.2.10 GO

The GO command reads an address uses JMP to run the program.

### *2.2.10.1 GO Algorithm and Flowchart*



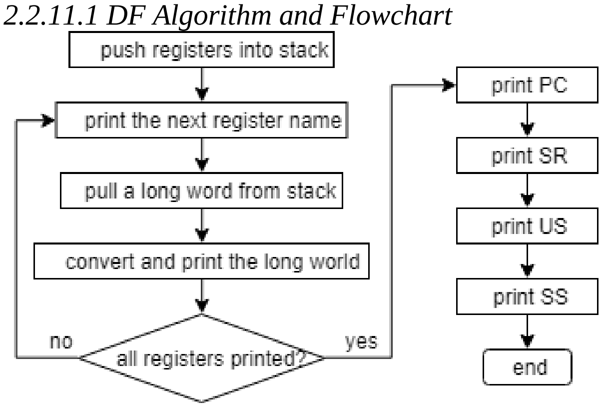

### *2.2.10.2 GO Assembly Code*

```
GO            MOVEM.L     D0-D7/A0-A6,-(SP)
              BSR         SC_NXT              ; read the address
              MOVEA.L     D7,A5
              JMP         (A5)                ; jump to program
              LEA         M_CRLF,A1
              MOVE.B      #13,D0
              TRAP  #15
              MOVEM.L     (SP)+,D0-D7/A0-A6
              RTS
```

## 2.2.11 DF

This command first stores all registers in stack using MOVEM, then extract registers one by one, printing the value on screen. After D and A registers are printed, PC, SR, US and SS are also fetched and printed.

### *2.2.11.1 DF Algorithm and Flowchart*




### *2.2.11.2 DF Assembly Code*

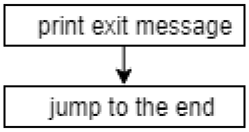
```
DF            MOVEM.L     D0-D7/A0-A6,-(SP)
              LEA         M_REGS,A0
DF_LO         MOVE.W      #5,D1
```

```
        MOVEA.L     A0,A1       ; print register name
        MOVE.B      #1,D0
        TRAP  #15
        MOVE.L      (SP)+,D1    ; extract register value
        BSR         ITOA
        MOVEA.L     A5,A1
        MOVE.W      #8,D1
        MOVE.B      #1,D0
        TRAP  #15               ; print register value
        ADDA.L      #5,A0
        CMPI.B      #0,(A0)
        BNE         DF_LO
        LEA         M_PC,A1           ; print 'PC='
        MOVE.B      #14,D0
        TRAP  #15
        MOVE.L      #DF,D1            ; print PC value
        BSR         ITOA
        MOVEA.L     A5,A1
        MOVE.W      #8,D1
        MOVE.B      #1,D0
        TRAP  #15
        LEA         M_SR,A1           ; print 'SR='
        MOVE.B      #14,D0
        TRAP  #15
        MOVE.W      SR,D1       ; print SR value
        BSR         ITOA
        MOVEA.L     A5,A1
        ADDA.L      #4,A1
        MOVE.W      #4,D1
        MOVE.B      #1,D0
        TRAP  #15
        LEA         M_US,A1           ; print 'US='
        MOVE.B      #14,D0
        TRAP  #15
        MOVE.L      USP,A1            ; print US value
        MOVE.L      A1,D1
        BSR         ITOA
        MOVEA.L     A5,A1
        MOVE.W      #8,D1
        MOVE.B      #1,D0
        TRAP  #15
        LEA         M_SS,A1           ; print 'SS='
        MOVE.B      #14,D0
        TRAP  #15
        MOVE.L      SP,D1
        BSR         ITOA
        MOVEA.L     A5,A1
        MOVE.W      #8,D1
        MOVE.B      #1,D0
        TRAP  #15

        LEA         M_CRLF,A1
        MOVE.B      #13,D0
        TRAP  #15
        RTS
```

### 2.2.12 EXIT

This command first prints a message, then branch to the end of program. This is the only way to jump out of the loop in interpreter.

#### *2.2.12.1 EXIT Algorithm and Flowchart*

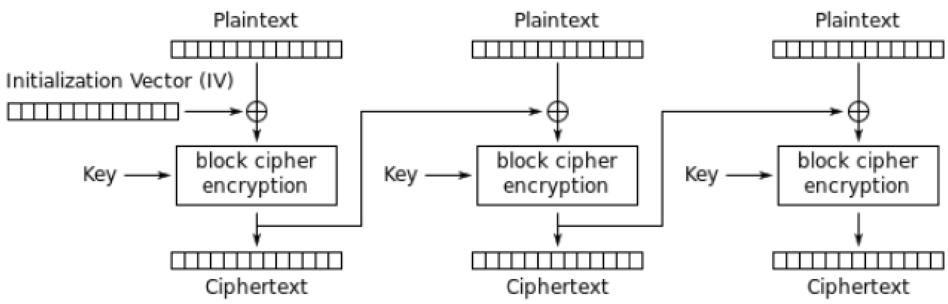



*2.2.12.2 EXIT Assembly Code*

```
EXIT  LEA         M_END,A1
            MOVE.B      #13,D0
            TRAP  #15
            BRA     END_L
```

2.2.13 ENCR

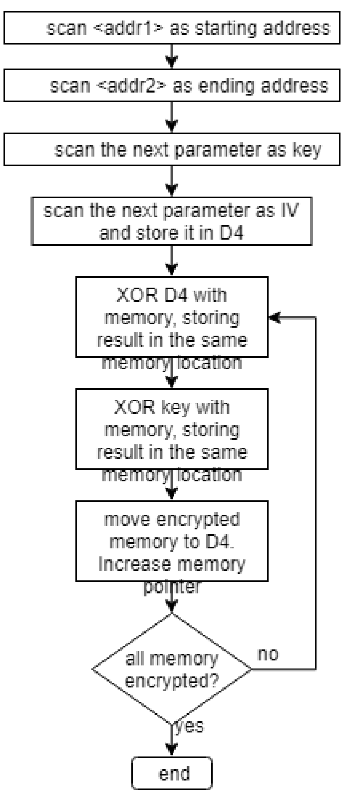
This command uses DES with Cipher Block Chaining (CBC) mode to encrypt a memory block. It receives 2 addresses and 2 numbers. The first number is key, and the second is initialization vector (IV). The process of encryption [2] is shown below:




Cipher Block Chaining (CBC) mode encryption

The block size of my program is a word, so starting and ending addresses should be even. Also, key and IV should not be longer than a word, otherwise only the lower word is valid.

*2.2.13.1 ENCR Algorithm and Flowchart*



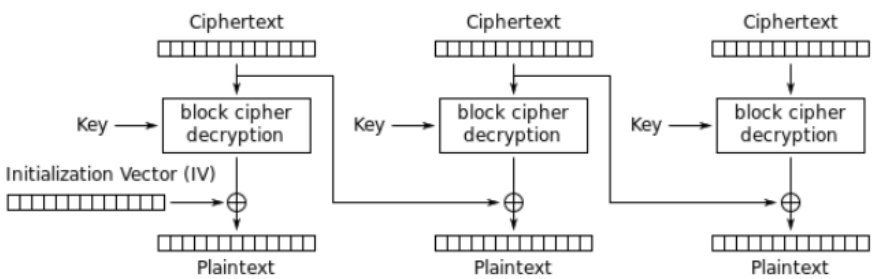

*2.2.13.2 ENCR Assembly Code*

```
* Encryption and decryption using Cipher-Block Chaining (CBC) mode
ENCR  MOVEM.L     D0-D7/A0-A6,-(SP)
            BSR         SC_NXT
```

```
            MOVEA.L     D7,A5       ; A5 and A6 are starting and ending address
            BSR         SC_NXT
            MOVEA.L     D7,A6
            BSR         SC_NXT
            MOVE.W      D7,D3       ; D3 is key
            BSR         SC_NXT
            MOVE.W      D7,D4       ; D4 is IV
ENC_LO      EOR.W D4,(A5)           ; XOR plaintext with the last ciphertext or IV
            EOR.W D3,(A5)           ; XOR block with key
            MOVE.W      (A5)+,D4
            CMPA.L      A5,A6
            BNE         ENC_LO
            LEA         M_CRLF,A1
            MOVE.B      #13,D0
            TRAP  #15
            MOVEM.L     (SP)+,D0-D7/A0-A6
            RTS
```

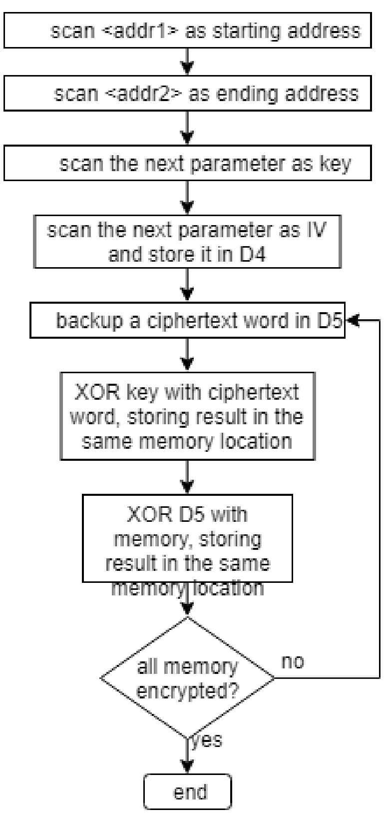
2.2.14 DECR

This program uses key and IV to decrypt a memory. If the key and IV are the same as encryption, the decrypted text will be identical with plaintext before encryption. The process [2] is shown below:




Cipher Block Chaining (CBC) mode decryption

*2.2.14.1 DECR Algorithm and Flowchart*



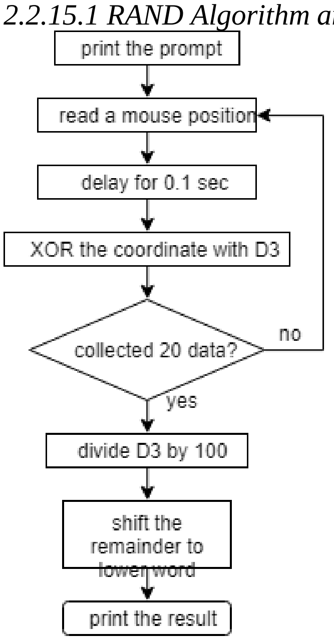
*2.2.14.2 DECR Assembly Code*

```
DECR  MOVEM.L     D0-D7/A0-A6,-(SP)
            BSR         SC_NXT
            MOVEA.L     D7,A5       ; A5 and A6 are starting and ending address
            BSR         SC_NXT
            MOVEA.L     D7,A6
            BSR         SC_NXT
            MOVE.W      D7,D3       ; D3 is key
            BSR         SC_NXT
            MOVE.W      D7,D4       ; D4 is IV
DEC_LO      MOVE.W      (A5),D5           ; D5 is a backup of ciphertext
            EOR.W D3,(A5)           ; XOR ciphertext with key
            EOR.W D4,(A5)+    ; XOR block with the last ciphertext or IV
            MOVE.W      D5,D4
            CMPA.L      A5,A6
            BNE         DEC_LO
            LEA         M_CRLF,A1
            MOVE.B      #13,D0
            TRAP  #15
            MOVEM.L     (SP)+,D0-D7/A0-A6
            RTS
```

2.2.15 RAND

This command generates a random number between 0 and 99. This is achieved by collecting mouse location for a few seconds, summing all coordinate data and divide the sum by 100. Remainder of the division is my result.

*2.2.15.1 RAND Algorithm and Flowchart*




*2.2.15.2 RAND Assembly Code*

```
* generate a random number between 0 and 99. Random data is collected form mouse.
RAND        MOVEM.L     D0-D7/A0-A6,-(SP)
            LEA         M_RAND,A1
            MOVE.B      #13,D0
            TRAP  #15
            LEA         MUS_BUF,A1
            CLR.L D2                ; D2 is counter
            CLR.L D3                ; D3 is a random number
RAND_LP     MOVE.B      #$0,D1            ; read a mouse position
            MOVE.B      #61,D0
            TRAP  #15
            EOR.L D1,D3
            MOVE.L      #10,D1            ; delay for 0.1 sec
```

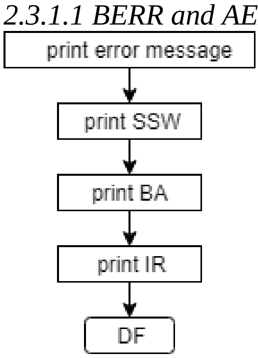
```
            MOVE.B      #23,D0
            TRAP  #15
            ADDQ.B      #1,D2
            CMP.B       #20,D2            ; loop for 20 iterations
            BNE         RAND_LP
            LSL.L       #8,D3
            LSL.L       #8,D3
            LSR.L       #8,D3
            LSR.L       #8,D3
            DIVU.W      #100,D3           ; mod the number by 100
            LSR.L       #8,D3
            LSR.L       #8,D3
            MOVE.L      D3,D1       ; print the random number
            MOVE.L      #10,D2
            MOVE.B      #15,D0
            TRAP  #15
            LEA         M_CRLF,A1
            MOVE.B      #13,D0
            TRAP  #15
            MOVEM.L     (SP)+,D0-D7/A0-A6
            RTS
```

### 2.3 Exception Handlers

Exception handlers are assigned in the beginning of program. BERR and AERR print SSW, BA, IR and formatted registers, while other handlers print only formatted registers.

#### 2.3.1 BERR, AERR

##### *2.3.1.1 BERR and AERR Flowchart*



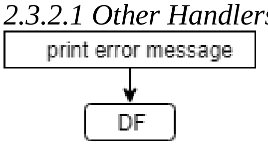

##### *2.3.1.2 BERR and AERR Assembly Code*

```
BERR        MOVEM.L     A1/A5/D0/D1,-(SP)
            LEA         M_BERR,A1
            MOVE.B      #13,D0
            TRAP  #15
            BRA         SBI               ; print SSW, BA, IR
AERR        MOVEM.L     A1/A5/D0/D1,-(SP)
            LEA         M_AERR,A1
            MOVE.B      #13,D0
            TRAP  #15
            BRA         SBI               ; print SSW, BA, IR
SBI         LEA         M_SSW,A1
            MOVE.B      #14,D0
            TRAP  #15
            MOVE.W      16(SP),D1   ; print SSW
            BSR         ITOA
            MOVEA.L     A5,A1
            ADDA.L      #4,A1
            MOVE.W      #4,D1
            MOVE.B      #1,D0
            TRAP  #15
            LEA         M_BA,A1
```

```
            MOVE.B      #14,D0
            TRAP  #15
            MOVE.L      18(SP),D1   ; print BA
            BSR         ITOA
            MOVEA.L     A5,A1
            MOVE.W      #8,D1
            MOVE.B      #1,D0
            TRAP  #15
            LEA         M_IR,A1
            MOVE.B      #14,D0
            TRAP  #15
            MOVE.W      22(SP),D1   ; print IR
            BSR         ITOA
            MOVEA.L     A5,A1
            ADDA.L      #4,A1
            MOVE.W      #4,D1
            MOVE.B      #1,D0
            TRAP  #15
            MOVEM.L     (SP)+,A1/A5/D0/D1
            BSR         DF                  ; print registers
            BRA         START
```

### 2.3.2 Other Handlers

#### *2.3.2.1 Other Handlers Flowchart*




#### *2.3.2.2 Other Handlers Assembly Code*

```
ILLI        MOVEM.L     A1/D0,-(SP)
            LEA         M_ILLI,A1
            MOVE.B      #13,D0
            TRAP  #15
            MOVEM.L     (SP)+,A1/D0
            BSR         DF
            BRA         START
PRIV        MOVEM.L     A1/D0,-(SP)
            LEA         M_PRIV,A1
            MOVE.B      #13,D0
            TRAP  #15
            MOVEM.L     (SP)+,A1/D0
            BSR         DF
            BRA         START
DIVZ        MOVEM.L     A1/D0,-(SP)
            LEA         M_DIVZ,A1
            MOVE.B      #13,D0
            TRAP  #15
            MOVEM.L     (SP)+,A1/D0
            BSR         DF
            BRA         START
CHKI        MOVEM.L     A1/D0,-(SP)
            LEA         M_CHKI,A1
            MOVE.B      #13,D0
            TRAP  #15
            MOVEM.L     (SP)+,A1/D0
            BSR         DF
            BRA         START
LINA        MOVEM.L     A1/D0,-(SP)
            LEA         M_LINA,A1
            MOVE.B      #13,D0
            TRAP  #15
```

```
            MOVEM.L     (SP)+,A1/D0
            BSR         DF
            BRA         START
LINF        MOVEM.L     A1/D0,-(SP)
            LEA         M_LINF,A1
            MOVE.B      #13,D0
            TRAP  #15
            MOVEM.L     (SP)+,A1/D0
            BSR         DF
            BRA         START
```

**3. Discussion**

The major challenge of this monitor design is to make the code shorter and more readable. I implemented several helper functions to finish some common tasks, like converting ascii to number, converting number to ascii, printing a memory block, and scanning the next parameter. These subroutines are used in almost every command, so it largely simplifies the code. Some commands requiring users to provide operating mode, such as SORTW, can be simplified by using function pointer. In this case, all codes except comparator can be implemented only once. Two comparators are pointed by an address register, which is called in program to compare two words.

**4. Feature Suggestions**

To improve this program, the command interpreter can be modified. Now the interpreter has few error checking capabilities, which might cause illegal instruction to be executed. To make the program more robust, input address length should be checked, and text should be checked to make sure it's wrapped by single quotes. Also, codes can be added to ignore leading spaces, which makes the monitor easier to use.

**5. Conclusion**

In this project, I wrote assembly code to receive user input, execute commands and handle exceptions. Knowledges applied in this project include linked list, sorting algorithm, searching substring algorithm and exception vectors. While interpreting user input, all kinds of illegal input should be considered and detected. This requires me to test my program with different inputs. Debugging the code is an important in this project, because some subroutines are too long to be inspected manually.

**6. References**


[1] https://www.nayuki.io/page/knuth-morris-pratt-string-matching

[2] https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#Cipher_Block_Chaining_(CBC)

[3] MOTOROLA M68000 FAMILY Programmer's Reference Manual