422 Disassembler Project

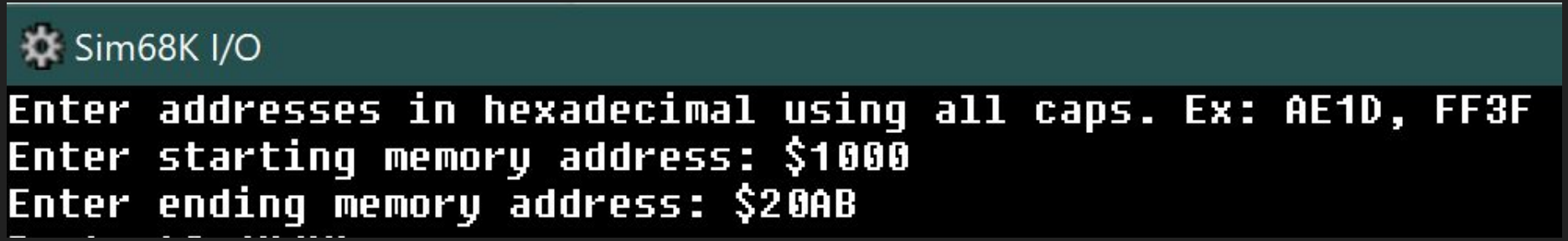
Easy68K

Paul, Mitchell, Christopher, Matthew

Overall architecture and design choices

- **Our design philosophy**
  - Modular and reusable code
  - Commenting and directions for every line of code
- **General architecture of our program**
  - **I/O, Main Loop, first digit look-up table, display buffers**
- **Program flow**

Program welcome screen + inputting memory address range



Example of how we append data to display buffers

```
MOVE.B   #'M',(A4)+ * M
MOVE.B   #'O',(A4)+ * O
MOVE.B   #'V',(A4)+ * V
MOVE.B   #'E',(A4)+ * E
MOVE.B   #'.',(A4)+ * .
MOVE.B   #'B',(A4)+ * B
MOVE.B   #9,(A4)+ *
```

Our main look-up table

```
JumpTable       CMP.B   #0,D0
                BEQ     CODE0000    *Bit manipulation/MOVEP/Immediate
                CMP.B   #1,D0
                BEQ     CODE0001    *MOVE.B
                CMP.B   #2,D0
                BEQ     CODE0010    *MOVE.L/MOVEA.L
                CMP.B   #3,D0
                BEQ     CODE0011    *MOVE.W/MOVEA.W
                CMP.B   #4,D0
                BEQ     CODE0100    *LEA, RTS, NOP (Miscellaneous), MOVEM
                CMP.B   #5,D0
                BEQ     CODE0101    *ADDQ/SUBQ/Scc/DBcc
                CMP.B   #6,D0
                BEQ     CODE0110    *BSR,BRA,Bcc
                CMP.B   #7,D0
                BEQ     CODE0111    *MOVEQ
                CMP.B   #8,D0
                BEQ     CODE1000    *OR/DIV/SBCD
                CMP.B   #9,D0
                BEQ     CODE1001    *SUB/SUBX
                CMP.B   #10,D0
                BEQ     CODE1010    *unassigned
                CMP.B   #11,D0
                BEQ     CODE1011    *CMP/EOR
                CMP.B   #12,D0
                BEQ     CODE1100    *AND/MUL/ABCD/EXG
                CMP.B   #13,D0
                BEQ     CODE1101    *ADD/ADDA/ADDX
                CMP.B   #14,D0
                BEQ     CODE1110    *Shift/Rotate
                CMP.B   #15,D0
                BEQ     CODE1111    *Special/Reserved
                BRA     INVALID     *if reached, invalid instruction
```

When first digit of opcode word is 1

```
CODE0001        JSR     SPLIT               *get EA information into D2 - D5
                MOVE.B  #'M',(A4)+ * M      *load text to buffer
                MOVE.B  #'O',(A4)+ * O
                MOVE.B  #'V',(A4)+ * V
                MOVE.B  #'E',(A4)+ * E
                MOVE.B  #'.',(A4)+ * .
                MOVE.B  #'B',(A4)+ * B
                MOVE.B  #9,(A4)+ *

                MOVE.B  #1,D7               *signify what size instruction is
                MOVE.B  D4,D0               *move bits to proper position
                MOVE.B  D5,D1               *
                JSR     MOVETable           *enter EA decoding jump table
                MOVE.B  #',',(A4)+ * ,
                MOVE.B  D3,D0               *move bits to proper position
                MOVE.B  D2,D1               *
                JSR     MOVETable           *enter EA decoding jump table
                RTS
```

Input validation

Current input validation is a two step process:

- Validate input characters
- Validate input address range
  - Low order addresses only and within a specified range defined in variables
- Keep MEMStart and MEMEnd variables as the input range for comparison

Input validation

In decoding the user ASCII, input is divided into D2 - D5 temporarily

While the decoded input is still in registers

- Check if in 0 - F range
- Branch away if not

```
CMP.W   #15, D2
BGT     INV_INP
CMP.W   #15, D3
BGT     INV_INP
CMP.W   #15, D4
BGT     INV_INP
CMP.W   #15, D5
BGT     INV_INP
```

Invalid decoding (invalid instruction)

Decoding an instruction as invalid happens through the various checks to reach a valid instruction

INVALID subroutine gets called whenever something cannot be a valid instruction any longer

Successful Decoding (CODE1101 ADD/ADDA)

JSR SPLIT
Bits 012 are in D5
Bits 345 are in D4
Bits 678 are in D3
Bits 9,10,11 are in D2

Move the correct Data Register to D1, and then compare to see what it matches with

In this case its bits 8, 7, 6

We then compare a number to the data register to the Data register to see if it matches

```
CODE1101        JSR     SPLIT       ;Bits 012 are in D5, 345 are in D4, 678 are in D3, and
                MOVE.W  D3,D1       ;move bits 8, 7, 6 into D1

                MOVE.B  #'A',(A4)+  ;
                MOVE.B  #'D',(A4)+  ;
                MOVE.B  #'D',(A4)+  ;
                JSR     ADD_OPMOD_TABLE
                RTS

ADD_OPMOD_TABLE MOVE.W  D3,D1       ;move bits 8, 7, 6 into D1

                CMP.W   #$0,D1      ;if its 0 (000) than its a Byte    <ea> + Dn -> Dn
                BEQ     B_EADN      ;
                CMP.W   #$1,D1      ;if its 1 (001) than its a Word    <ea> + Dn -> Dn
                BEQ     W_EADN      ;
                CMP.W   #$2,D1      ;if its 2 (010) than its a Long    <ea> + Dn -> Dn
                BEQ     L_EADN      ;

                CMP.W   #$4,D1      ;if its 4 (100) than its a Bite    Dn + <ea> -> <ea>
                BEQ     B_DNEA      ;
                CMP.W   #$5,D1      ;if its 5 (101) than its a Word    Dn + <ea> -> <ea>
                BEQ     W_DNEA      ;
                CMP.W   #$6,D1      ;if its 6 (110) than its a Long    Dn + <ea> -> <ea>
                BEQ     L_DNEA      ;

                CMP.W   #$3,D1      ;if its 3 (011) than its an ADDA WORD
                BRA     ADDA_W
                CMP.W   #$7,D1      ;if its 7 (111) than its an ADDA Long
                BRA     ADDA_L
                RTS
```

Successful Decoding (continued)

L_DNEA prints .L and a tab

DNEA prints D and moves the register number into D1

EncodeChar turns the ASCII Data Register number into Hex and adds it to the good buffer to be printed

ADDEA adds the effective address and then checks to see if its indirect, or if we need to pre or post decrement it.

CheckReg checks to see if we are using immediate data or an address and prints that out if needed.

If any of these do not work out, then we branch to “INVALID” which exits the subroutine and signals an invalid instruction.

```
L_DNEA          MOVE.B  #'.',(A4)+  ;
                MOVE.B  #'L',(A4)+  ;
                MOVE.B  #9,(A4)+  ;tab
                MOVE.W  #2,D7
                JSR     DNEA
                RTS
```

```
DNEA            MOVE.B  #'D',(A4)+
                MOVE.W  D2,D1
                JSR     EncodeChar
                MOVE.B  #',',(A4)+
                JSR     ADDEA
                RTS
```

```
ADDEA           MOVE.W  D4,D0
                MOVE.W  D5,D1
                CMP.B   #2,D0
                BEQ     AnIndirect
                CMP.B   #3,D0
                BEQ     AnIncrement
                CMP.B   #4,D0
                BEQ     AnDecrement
                CMP.B   #7,D0
                BEQ     CheckReg
                BRA     INVALID
```

Labelling Invalid Instructions

- When unreadable instructions are found a pointer is marked as true

```
INVALID         MOVE.B  #$01,INVALID_PTR
                RTS
```

- In the OPDec decoding loop this pointer is checked to decide whether the decoded instruction should be printed or data

```
*this subroutine is for decoding OPcode word
OPDec   JSR     DIGIT1              *isolate first digit of opcode word at D0
        LEA     GOOD_BUFFER,A4      *load good buffer to A4
        LEA     BAD_BUFFER,A5       *load bad buffer to A5
        MOVE.W  D1,OPCode           *store opcode for printing later
        JSR     JumpTable           *enter jump table after
        MOVE.B  #$00,(A4)           *signifies end of good buffer
        LEA     GOOD_BUFFER,A1      *prepare to print GOOD_BUFFER
        CMP.B   #$01,INVALID_PTR    *check to see if input was invalid
        BEQ     PDATA               *branch to printing data
        MOVE.B  #13,D0              *prepare to print GOOD_BUFFER
        TRAP    #15                 *print GOOD_BUFFER
        RTS
```

Printing Invalid Instructions as Data

Printing Data

```
PDATA    MOVE.B   #$00,INVALID_PTR
         JSR      DATA
         JSR      HEXOP
         MOVE.B   #$00,(A5)
         LEA      BAD_BUFFER ,A1
         MOVE.B   #13,D0
         TRAP     #15
```

Add “DATA” to buffer

```
DATA     MOVE.B   #'D',(A5)+ * D
         MOVE.B   #'A',(A5)+ * A
         MOVE.B   #'T',(A5)+ * T
         MOVE.B   #'A',(A5)+ * A
         RTS
```

Read data at hex to buffer

```
HEXOP    MOVE.W   OPCode,D6
         MOVE.B   #9,(A5)+          * tab
         MOVE.B   #'$',(A5)+          * $
         JSR      BAddWord
         RTS
```

Example: Invalid Instructions

Instructions In Memory:

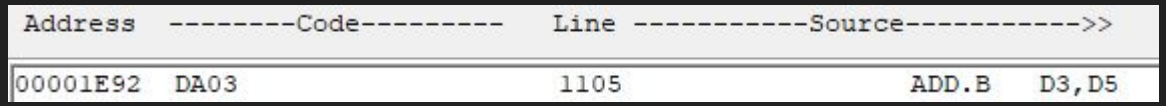
```
Address  --------Code---------   Line -----------Source----------

000010D4                           92           ;now is the time
000010D4  B47C 000F                93           CMP.W   #15, D2
000010D8  6E00 004E                94           BGT     INV_INP
000010DC  B67C 000F                95           CMP.W   #15, D3
000010E0  6E00 0046                96           BGT     INV_INP
000010E4  B87C 000F                97           CMP.W   #15, D4
000010E8  6E00 003E                98           BGT     INV_INP
000010EC  BA7C 000F                99           CMP.W   #15, D5
000010F0  6E00 0036               100           BGT     INV_INP
```

Console Output:

```
Enter addresses in hexadecimal using all caps. Ex: AE1D, FF3F
Enter starting memory address: $10D4
Enter ending memory address: $10F0
10D4     DATA     $B47C
10D6     DATA     $000F
10D8     BGT     $004E
10DC     DATA     $B67C
10DE     DATA     $000F
10E0     BGT     $0046
10E4     DATA     $B87C
10E6     DATA     $000F
10E8     BGT     $003E
10EC     DATA     $BA7C
10EE     DATA     $000F
Restart? (Y/N): _
```

Example: Invalid Instructions

Instructions In Memory:



Address	--------Code---------	Line	-----------Source----------->>	
00001E92	DA03	1105	ADD.B	D3,D5

Console Output:

```
Enter addresses in hexadecimal using all caps. Ex: AE1D, FF3F
Enter starting memory address: $1E90
Enter ending memory address: $1E92
1E90     ADD.B     D4,D5
Restart? (Y/N):
```