

# *EE 455 - Programming Lab # 1*

## **Object**

In this lab, you will be introduced to 68K. You may have programmed the 68K previously in your course work (e.g., CS306) or elsewhere, in which case much of this lab will be review. If the 68K is new to you, you'll soon find that it is quite powerful, while remaining relatively simple. In either case, the environment you will be using will likely be new to you. This will not be the case when you are finished with the following exercises.

## **Introduction**

The 68K environment you will be using consists the emacs editor (or your Solaris editor of choice), an assembler and terminal emulator running on a Sun Sparc and the Motorola Educational Computer Board (EDS). The EDS is a single board computer, designed around the MC68000. It has 32K bytes (16K words) of RAM, two serial ports, a parallel Centronics style port, audio tape port (no, you don't get to use this), and a firmware monitor, called TUTOR, to aid in program development and debugging. You can find out just about everything you'd ever want to know about the EDS in the orange user's manual in your lockers.

While this lab will introduce various aspects of the 68000 instructions and addressing modes, as well as many useful TUTOR commands, you may find that you need to know more. You should be able to find almost anything you want to know in the various books you have at your disposal. These include:

The ORANGE MC68000 Educational Computer Board User's Manual, which tells you everything you want to know about the EDS and TUTOR.

The Programmer's Reference Manual from Motorola, which includes detailed information on the various instructions of the 68K. (Please note that this book covers a wide variety of processors, and that only a subset of the instructions are available on the 68000.)

The M68000 8-/16-/32- Microprocessor User's Manual from Motorola, which includes all of the hardware information about the 68K. While you'll find it very useful when you start designing hardware, its chapter 3 includes a description of data organization and addressing modes.

The course text, which has good information scattered all about.

<p>Throughout this lab, you will find various questions posed to you. These questions and others that you might think of should be answered in your reports. You should include commented listings of any programs you write. Other listings should be included as necessary to make your report clear and understandable. Assume that we will not have a copy of this hand out available when we are grading your reports.</p>
---

## **Useful Facts**

The EDS has 32K bytes of memory, which are located between addresses \$8 and \$7FFF. The space from \$8 to \$8FF is reserved for system use, leaving the range from \$900 to \$7FFF available for your use.

The EDS has two buttons you may need to use. The first is the black RESET button. When pressed, the EDS is completely reset and ready to go. You'll find that if you press this button, the following prompt will appear on the terminal.

```
TUTOR 1.3>
```

The second button is the red ABORT button. It should normally be pressed to stop a program, which appears to be hung. It will cause the processor to halt execution of the program, display the unchanged contents of the processor registers (note that RESET will modify the registers), and return the prompt.

TUTOR requires that all of your input be upper case. So set those Caps Lock keys.

You'll find that various 68K instructions and operands in the 68000 come in various sizes and types. The sizes are byte, word, and long word, which correspond to 8 bits, 16 bits, and 32 bits respectively. These are designated by the addition of .B, .W, or .L (not to be confused with mixed logic) to the instruction mnemonic or operand. Thus an instruction might look like this

```
MOVE.B D2,$1000.W
```

Indicating a byte move from register D2 to the 16-bit address \$1000. Often times, assemblers will default to an appropriate length. Unfortunately, ours does not always work right. It is strongly recommended that you always use the desired length specification to prevent confusion and assembler errors.

### **Starting with TUTOR**

Plug in your EDS power cord, and connect the communications cable from the Console port of the EDS to the serial port of the Sun Sparc on the front panel of the lab workstation. Login to the Sun and type "tutor" in a terminal window. When the EDS is powered up, you should see the TUTOR 1.3> prompt. If you don't see the TUTOR prompt, try pressing the RESET button (the black one). If that doesn't work, ask for help.

### **MD - Memory Display**

Obviously, it is important to see the contents of computer's memory. This is accomplished by using the Memory Display or MD command. Let's try it. At the prompt, type "MD 82FC". In the examples, what you should type is bold and the computer's response is normal.

```
TUTOR 1.3 > MD 82FC
0082FC 54 55 54 4F 52 20 20 31 2E 33 20 04 AE 2A 15 8C TUTOR 1.3 ..*..
TUTOR 1.3 >
```

The first item on the line is the address you specified in the command. It is followed by the contents of the sixteen bytes of memory starting at that address. You'll notice that these values,

as well as the address are all in hexadecimal. Finally, you'll find the last sixteen characters are the ASCII equivalent of those bytes. The "." can either mean an actual period (\$2E) or a non-printable character.

Now press the return key once more. You should find that TUTOR will print the next sixteen lines of memory, starting at location \$830C. The locations you are looking at are part of the ROM that contains the TUTOR code.

Trying "MD \$82FC" should yield the same result. If you haven't caught on already, the "\$" signifies a hexadecimal number, and TUTOR (in most cases) defaults to hexadecimal. You can also try the following commands.

```
TUTOR 1.3 > MD &33532
TUTOR 1.3 > MD %1000001011111100
TUTOR 1.3 > MD @101374
```

Each of these should produce the same results, as & indicates a decimal number, % a binary number, and @ an octal number. Most likely, you'll be using hexadecimal (\$), and decimal (&) most of the time. Now try this.

```
TUTOR 1.3 > MD $82FC 5
TUTOR 1.3 > MD $82FC 10
TUTOR 1.3 > MD $82FC 11
```

You should find that the first two commands give you identical lines of sixteen bytes, while the third gave you two lines of sixteen bytes. The optional number following the address indicates how many bytes are displayed; however, it always does so in multiples of sixteen.

Now try:

```
TUTOR 1.3 > MD 9000 B;DI
009000    5983                SUBQ.L  #4,D3
009002    4280                CLR.L  D0
009004    6100013C           BSR.L  $009142
009008    6066                BRA.S  $009070
00900A    61000C08           BSR.L  $009C14
```

The DI option is the Disassemble option. You are seeing the machine and assembly language for a section of the TUTOR program. The disassembly continued until the specification for eleven (\$B) bytes was met. If you have problems with this command, you should make sure that there is no space between the ";" and the "DI." This is true for any of the options following a ";" that you will find for other commands.

## MS - Memory Set

Ok, that's well and good, but we've got to be able to change memory too. There are several commands, which permit you to modify memory. The first is the MS or Memory Set command. Try the following:

```
TUTOR 1.3 > MS 2000 12 34 45 789A 12345678
```

TUTOR 1.3 > MD 2000

The MD will permit you to see what the Memory Set did. Notice how the data can be entered as bytes, words, or long words. Now try (removing the Caps Lock for the text):

TUTOR 1.3 > MS 2000 'The Mets are Pond Scum!'

After the appropriate MD command, you should see

```
002000    54 68 65 20 4D 65 74 73  20 61 72 65 20 50 6F 6E  The Mets are Pon
002010    64 20 53 63 75 6D 21 FF  FF FF FF FF FF FF FF  d Scum!.....
```

You have now entered ASCII code and have had it converted to HEX for you! Can you mix ASCII, HEX, Decimal, etc. on the same line? Try some.

### MM - Memory Modify

Another useful command used to change the contents of memory is the MM or Memory Modify command. It has a number of options and takes the general form of:

TUTOR 1.3 > MM <address>[;<options>]

This command has a number of options. They are:

- |           |   |
|-----------|---|
| (default) | Display one byte at a time                      |
| ;W        | Display one 16-bit word at a time               |
| ;L        | Display one 32 bit long word at a time          |
| ;O        | Display one byte, accessing only odd addresses  |
| ;V        | Display one byte, accessing only even addresses |
| ;N        | Do not display the byte                         |
| ;DI       | Enter the assembler/disassembler                |

Start by clearing locations \$2000 to \$200F by using the MS command you used before. Then try the following Memory Modify command:

TUTOR 1.3 > MM 2000

You should see

```
002000 FF ?
```

With this, TUTOR is telling you that the byte \$FF is located at memory address \$2000, and it is asking you if you would like to change it. Press a CR indicating that you wish to leave the value as \$FF. You should then see and enter:

```
002001 FF ? 2C
```

This will change the contents of \$2001 to \$2C. Next, you will be prompted for \$2002, \$2003, and so on. You can continue to change bytes as you wish. When you're finished, just enter a

period (".") followed by a CR, and you will return to the TUTOR prompt. You should try each of the options until you reach the DI option.

At this point, enter the following MS command to set the memory to a known state.

```
TUTOR 1.3 > MS 1000 30383000
```

Now try

```
TUTOR 1.3 > MM 1000;DI
```

You should see:

```
001000 30380000      MOVE.W $00003000,D0 ?
```

Tutor has disassembled the code at address \$1000 for you and is asking you if you want to change it. Giving a CR will leave the code intact and list the instruction at \$1004 (4 bytes later). But we want to change it. To do so, **enter a space after the question mark (very important)** and follow it with the assembly language code for the instruction.

```
001000 30380000      MOVE.W $00003000,D0 ? MOVE.L D2,D1
```

Your screen should now show:

```
001000 2200          MOVE.L D2,D1
001002 3000          MOVE.W D0,D0 ?
```

You have just entered assembly language and had the machine language code \$2200 entered into memory. You can enter programs this way; however, you will find it easier, especially for larger programs, to use the assembler on the Macintosh. If you want to enter four or five lines of code, you'll find the assembler built into TUTOR to be quite handy.

## Your First Program

Now you are ready to enter your first program. Let's do this one with the assembler built into TUTOR, as it's only a few lines long. Using the MM command with the DI option,

```
TUTOR 1.3 > MM 1000;DI
```

enter the following program.

```
MOVE.B #3,D2
ADD.B #12,D2
MOVE.B D2,$2000
MOVE.B #228,D7
TRAP #14
```

Remember that a period ends the MM command. The program will load the value 3 into D2, add the value 12 to it, and move the result to location \$2000. Try a MD command to display the

code to make sure you entered it correctly. For now, don't worry about the last two lines, we'll get to those later. It's time to run the program. This is done with the GO command, as shown below.

```
TUTOR 1.3 > GO 1000
PHYSICAL ADDRESS=000010000
```

Use the MD command to check the answer.

```
TUTOR 1.3 > MD 2000
002000 0F + Whatever else is in memory.
```

*Why is the result HEX value \$F? Weren't we adding 3 and 12? Doesn't the EDS default to HEX?* Remember, the EDS defaults to HEX most of the time. This is one occasion when it doesn't. When using the assembler, the default is decimal. If you want HEX, you have to explicitly indicate that with a \$.

Tutor does include a command, DC or Data Conversion, which will convert from HEX, Decimal, Octal, or Binary to HEX and Decimal. You should check it out in the orange manual and try it.

## Addressing Modes

There are many addressing modes in the 68000 repertoire. The following piece of code was designed to demonstrate a number of the different modes. You probably want to use the assembler on the Sun to enter this code. Pop up a new terminal window on the Sun and type "emacs" to invoke the emacs editor. When the emacs editing window opens type the following code in (or cut and paste it from this document). After you have the code entered, select "Save Buffer As" under the File menu in emacs and give the file a name with a .asm appended (that's dot asm): such as test.asm.

```
ORG          $1000
MOVE.W #$2000,A0
MOVE.W A0,A1
MOVE.W          #$3F,D1
MOVE.B #$2C,$2001
MOVE.B D1,$2004
MOVE.B $2004,$2005
MOVE.B D1,(A0)
MOVE.B D1,2(A0)
ADDQ.W #3,A1
MOVE.B 1(A0),(A1)
MOVE.B -2(A1),$2009
MOVE.B -(A1),$2006
MOVE.B (A0)+,$2007
MOVE.B (A0),$2008
MOVE.B D1,4(A0,D1.W)
MOVE.B D1,-4(A0,D1.W)
MOVE.B #228,D7
TRAP    #14
```

Assemble the program by opening a third terminal window on the Sun and typing: “asm test.asm”. (NOTE: The assembler will automatically convert the MOVE.W #2000,A0 into the correct MOVEA.W.) If you look in your directory you will see that the assembler has created several new files. The test.lst file should be opened in yet another terminal window on the Sun so that you may study the assembled listing by typing cat test.lst. **These are the only listings that are helpful in debugging as they show the actual memory contents of your program!** The only reason we left the emacs window open with the test.asm code is so that you may edit it if errors are detected during the assembly process.

*Before you download and run this program, go through it and try to compute what values will end up where and why.* (Include this in your report.) You should use your class notes and the Microprocessor User's Manual to do this.

To download the object code to the EDS you may exit emacs where you were editing test.asm and type: “load test”.

You should get the Tutor prompt after your code is loaded. Ok, the program should now be down on the EDS, and you can run it with the GO command and check the output with the MD command. *Do you get what you expected? If not, why? Do you understand the addressing modes used? Can you explain how the source and destination were computed for each MOVE instruction?* (Please include with each line of code an appropriate description of how the source and destination are determined.)

## More Programs

Let's try a simple program, which uses the autoincrement addressing mode to copy one block of memory to another. The code you'll need follows. You can use either the MM command or the assembler to generate your code. If you use the MM command, you'll have to omit the START label and replace the START after the BNE instruction with the address \$1000.

```
          ORG          $1000
START MOVE.B (A1)+, (A2)+
        SUBQ.W #1, D0
        BNE          START
        MOVE.B #228, D7
        TRAP        #14
```

*Ok, what's this do? How's it work?* Well, the address registers A1 and A2 are used to pass the starting locations of the two blocks of memory to the program. Data register D0 contains the number of bytes to be moved. Let's get ready to run this program. Now set the address registers. You can do this by entering:

```
TUTOR 1.3 > .A1 1000
TUTOR 1.3 > .A2 2000
TUTOR 1.3 > .D0 0C
```

The last line will put the value 0C into data register D0. If you want to see the data registers or address registers, simply enter .Dn or .An, where n indicates the register number. This also works for the Program Counter (.PC), the Status Register (.SR), and both the User and

Supervisor Stack Pointers (.US and .SS). You can also use the DF or Display Formatted Registers command to see them. You could do this:

```
TUTOR 1.3 > DF
PC=00001004 SR=2700=.S7.... US=FFFFFFF7 SS=00000786
D0=0000000C D1=FFFFFFF D2=FFFFFFF03 D3=FFFFFFF
D4=FFFFFFF D5=FDFFFFFF D6=FFFFFFF D7=FFFFFFF
A0=FFFFFFF A1=00001000 A2=00002000 A3=FFFFFFF
A4=FFFFFFF A5=FFFFFFF A6=FFFFFFF A7=00000786
-----001004      307C2000                MOVE.W #8192,A0
```

If you haven't already, download the program to the EDS. *With the values you entered for A1, A2, and D0, what do you expect to be moved where?* Run the program with

```
TUTOR 1.3 > GO 1000
```

Now examine the contents of memory around \$2000. You might use the DI option. *Does this look familiar? Why? What's going on here?*

Write a program using the autoincrement addressing mode to add a series of N bytes of memory. You can specify the starting location of the block of data and the number of bytes as you did in the copy program. Save the answer at some memory location with the MOVE command and check to see that it is correct. You'll also have to define a series of memory locations to add up. *Can you make the program so that it will give a correct 16 bit result for the sum of a series of 8 bit data values?* You might check the Programmer's Reference Manual for the variations of the ADD instruction. You should also add the instructions

```
MOVE.B #228,D7
TRAP      #14
```

to the end of this and the other programs you write in the lab. (See the end for why.)

If you have troubles getting your program to work, you might find it helpful to single step your program. This can be done by first setting the Program Counter to the starting address with a command like

```
TUTOR 1.3 > .PC 1000
```

If you then issue the T or Trace command, your program will be single stepped. At each point along the way, TUTOR will display the status of the processor registers, as well as the next instruction to be executed. You can examine or change any of the registers or memory before executing the next step. You should consult your orange book for more information on the trace command, as well as breakpoints.

You should also try writing a program that will set a block of memory (defined by a start address in register A1 and an end address in register A2) to some known value (defined in register D0). *How does your program work? Can you think of other ways of performing the same function?*

Now that you've got that running, another useful command is the BF or Block Fill command. Try it.

```
TUTOR 1.3 > BF 3000 300E 1919
```

Use the MD command to check out the results of this command at location \$3000 and above. As you entered the copy program earlier, you might also check out the BM or Block Move command in the orange Tutor manual.

### 32 Bit Multiply

The 68000 has built-in 16 bit multiply and divide routines. These exist in both signed and unsigned varieties (MULU/DIVU and MULS/DIVS). Can you write a 32 bit unsigned multiply routine? Here's a hint. Think of the 32 bit data, X and Y, as being made of two 16 bit values,  $\langle x_1, x_0 \rangle$  and  $\langle y_1, y_0 \rangle$ . They can be written as:

$$X = (2^{16} x_1 + x_0) \text{ and } Y = (2^{16} y_1 + y_0)$$

Then the product is

$$X Y = 2^{32} x_1 y_1 + 2^{16} (y_1 x_0 + y_0 x_1) + x_0 y_0$$

You should be able to do this with four MULUs and four ADDs. Try out your program. Multiplying \$11111111 and \$22222222 should yield the 64 bit result \$02468ACF0ECA8642. *Is that what you got? Try \$44444444 and \$88888888. What did you get? Is that right? (If you think a minute, you don't have to work it all out. It's VERY simple!) Try \$66666666 and \$33333333 and report your answer.*

### If You Have Time, Try this (or at least read it)

The EDS has a Parallel Interface/Timer (PI/T) chip. In addition to other things, it can generate interrupts at a predefined interval. The following program illustrates this.

```
TCR EQU $10021 ;Location of the Timer Control Register
TIVR EQU $10023 ;Timer Interrupt Vector Register
CPRH EQU $10027 ;Counter Preload Register High
CPRM EQU $10029 ;Counter Preload Register Middle
CPRL EQU $1002B ;Counter Preload Register Low
CNTRH EQU $1002F ;Counter Register High
CNTRM EQU $10031 ;Counter Register Middle
CNTRL EQU $10033 ;Counter Register Low
TSR EQU $10035 ;Timer Status Register
; The program follows
ORG $1000
TIMER MOVE.B #$A0,TCR.L ;In TCR, enable interrupts, normal clock
;control, counter preload register used,
```

```

                                ;and timer off
MOVE.W $$2100,SR               ;Accept level 2 interrupts and above
MOVE.B $$40,TIVR.L             ;Put interrupt vector ($64) into TIVR
MOVE.L $$3000,$100            ;Put address of interrupt routine at
                                ;vector address ($100 = 4 * $40)
MOVE.B $$01,CPRH.L             ;Load $01E848 into 24 bit counter preload
MOVE.B $$E8,CPRM.L             ;register. This will give an interrupt every
MOVE.B $$48,CPRL.L             ; 8  $\mu$ S * $1E848 = 1 Sec
MOVE.L #0,$2000                ;clear clock counter memory
ORI.B #1,TCR.L                 ;Start the counter
TIMER1 BRA    TIMER1           ;Just sit here waiting for interrupts.
;
;The Interrupt service routine follows.
ORG    $3000                    ;Yep, we stuck it at $3000
ISR    ANDI.B $$FE,TCR.L         ;Turn off interrupts for a while
ADDQ.L #1,$2000                 ;Increment the clock counter by 1
MOVE.W $$2010,A6                ;Put a starting string address in A6
MOVE.B #13,(A6)+                ;Put a CR at A6 and increment it by one
MOVE.L $2000.W,D0                ;Put the address of the word to be converted
                                ;to an ASCII string into D0
MOVE.B #236,D7                  ;call HEX2DEC to convert the clock count
TRAP   #14                       ;to ASCII
MOVE.B #'s',(a6)+               ;Stick an "s" on the end of the string.
MOVE.W $$2010,A5                ;Put the start address of the string in A5
MOVE.B #243,D7                  ;Call OUTPUT to send string in (A5) to (A6)-1
TRAP   #14                       ;to port 1 (the console)
ORI.B #1,TCR.L                 ;turn the interrupts back on.
RTE                                ;Return from Exception (i.e., interrupt)

```

This program will increment and display the long data word in location \$2000 once every second. *Can you explain what this program is doing? Can you make it display every 2 seconds? Every half second? Can you change the display format?*

This program uses the TRAP #14 commands to convert this value to an ASCII string and to display it on the console terminal. Once again, that TRAP #14 appears. A "TRAP" is a type of exception. Typically exceptions mean something is wrong (*i.e.*, divide by zero or an illegal instruction) or something needs to be serviced, as with interrupts. In the EDS, TRAP #14 permits the user to call a series of system calls. The call to be used is specified in register D7 (you may have noticed the MOVE.B #xxx,D7 before each TRAP #14). #228 is used to return the user to the TUTOR prompt. That's why you used it so often before. #243 sends a string to the console port, while #236 converts a long word to an ASCII representation of a decimal string. After the system call is executed, you will usually be returned to your program; however, you might have the registers destroyed. Check out chapter 5 of the orange book for a full description of the TRAP #14 instructions.