UNIMPLEMENTED INSTRUCTIONS

OVERVIEW:
This Programming Assignment will take the sine computation program you wrote for Programming Assignment #4 and implement it as a 68000 assembly language instruction.
Reference: Leo J. Scanlon, The 68000: Principles and Programming

The design specifications for the MC68000 included several instructions that were not implemented in the final commercial microprocessor. These instructions included string manipulation, floating-point arithmetic and others. Motorola reserved approximately 20% of the total microcode memory (the memory INSIDE the 68000 which contains the 68000's internal programs for instruction decoding and execution) for custom or special purpose versions of the 68000.

If you carefully study your Programmer's Reference Card you will see that NO instructions begin with %1010 or %1111. These are known as the "unimplemented" op codes and would have belonged to the instructions (such as string manipulation and floating point arithmetic) which were not put into the 68000's final microcode.

Rather than just ignore any instruction beginning with %1010 or %1111, the Motorola designers were clever and provided a mechanism for the programmer to use these codes as his/her own instructions in the 68000 instruction set. Specifically, Motorola provided a unique vector number in the exception map (the vector table) for each of these unimplemented op codes.

To use either of these op codes, simple insert a word value in your program that has a most significant hex digit of $A (%1010) or $F (%1111). You can do this very easily by using the define constant directive, such as DC $Axxx or DC $Fxxx, where x can be any hex digit. When the 68000 fetches this constant and attempts to decode it, the 68000 will recognize this constant as an unimplemented instruction (since it begins with $A or $F) and will trap to the unimplemented instruction routine via address $28 (for %1010) or $2C (for %1111). This means that the address of your unimplemented instruction routine must be placed in address $28 or $2C by YOU - the programmer - not Motorola.

Example:
Emulate a set of floating point instructions using the op code 10102. Four such routines will be implemented:

FPADD - floating point add
FPSUB - floating point subtract
FPMUL - floating point multiply
FPDIV - floating point divide

which are defined by appropriate labels to each routine.

These instructions are data register to data register only with the following machine code format.

Note that the instruction is only a single word long since only data registers are allowed as source or destination registers.

Example program:

```
* This exception routine is executed if the 68000 encounters a "1010"
instruction. It decodes the operation field of the instruction (bits 3
and
4) and, using this number as an index, jumps to a floating point add,
subtract, multiply or divide routine elsewhere in memory. Registers A1
and
D1 are affected.

* Initialize the 1010 vector.
        ORG     $28
        DC.L    FLTP            ;1010 vector now points to FLTP

        ORG     $1000
FLTP    MOVEA.L 2(SP),A1        ;get PC address of NEXT instruction
        MOVE    -2(A1),D1       ;fetch 1010 instruction and put into D1
        MOVE    D1,-(SP)        ;save instruction in stack
        ANDI    #$0018          ;mask out all BUT operation field
        LSR     #1,D1           ;calculate index (op field ¥ 4)
        LEA     OPADDR, A1      ;put jump table addresses into A1
        MOVEA.L 0(A1,D1.W),A1   ;change to jump table address
        JMP     (A1)            ;jump to appropriate routine

* The floating point routines can be put anywhere in memory.

OPADDR  DC.L    FPADD, FPSUB, FPMUL, FPDIV

        ORG     $____   address of floating point add routine
FPADD...
        ORG     $____   address of floating point subtract routine
FPSUB...
        ORG     $____   address of floating point multiply routine
FPMUL...
        ORG     $____   address of floating point divide routine
FPDIV...
```

Your programming assignment:

Your program should decode the following instruction word format

`%1010xxxADDDDSSSS`

where %1010 is the op code, xxx are don't cares (i.e. we don't use them),%A identifies the operation to be performed, %DDDD designates the source, and %DDDD designates the destination. All of these are specified in binary. A=0 indicates that a sine operation should be performed, A=1 indicates that a cosine operation is to be performed. The destination can be either a memory location or a data register. A Data register destination would be specified in the form %x000 indicates destination to be put in D0, %x001

indicates destination to be put in D1, etc. up to %x111 indicating destination to be put in D7. A memory location destination is specified by %1xxx. The address of the memory location will then follow in an extension word. The source can only be a data register and is indicated in a manner similar to a data register destination, i.e., %x000 indicates source in D0, %x001 indicates source in D1, etc. up to %x111 indicating source in D7.

This instruction will compute the sine or cosine of a binary angle given in the format of Programming Assignment #4 and #5. The source effective address must be a data register and the destination effective address must be a data register or a memory location.

General comments: Your program must pass parameters to the subroutine through the stack, values returned must be passed on the stack, i.e.

```
push arguements
jsr subroutine
pop results
```

Also, include an instruction after your last subroutine call to show that your program properly returns to the main program. The way that the program can be internally organized is up to you but you might want to examine the use of jump tables, see Section 10.1 of your textbook.

IMPORTANT NOTE:
You have to set an option in the debugger. At the command line type:

```
Debugger Options General Exceptions Normal
```

or

```
Debugger Options General Exceptions Report
```

The default is

```
Debugger Options General Exceptions Stop
```

This command deals with what the debugger does on an exception. As one would guess it defaults to Stop which will simply display a message in the journal window and NOT process your exception. I would recommend the Report option, it flashes a message in the journal window and processes the exception just as you would expect. The Normal option processes the exception the same way the Report option does without the message to the journal window. This info is in the debugger manual on page 4-2.

A problem with working with exceptions and interrups is that the TRAP #0 which you were using to stop your programs no longer works. This is because you now have the exceptions properly activated and the debugger will no longer halt when an exception occurs. You can single step the program but cannot stop the program when running it using a Program Run. The best solution is to use the debugger to set a breakpoint. This can be done by

```
Breakpoint Instruction <address>
```

where address is the address of the instruction at which to wish to stop.

HINT: Remember about User and System Stack Pointer

This lab must be checked out and will receive a grade of 0 or 1 which will be assigned at the time of the checkout. The formal lab writeup will consist of the following:

1. title page in the normal format
2. detailed listing of the capabilities of your code, i.e. can you do both source and destination for any data register, can you decode a destination memory location, etc.
3. a detailed picture of the stack after it has been called. This should be AFTER any movem's or other instructions which put stuff on the stack have occurred.
4. a commented program listing and will be turned in at the time of the checkout. Formal checkout times will be announced later.