EEAP 282
PROGRAMMING ASSIGNMENT #4

UNIMPLEMENTED INSTRUCTIONS

OVERVIEW:
This Programming Assignment will take the CRC computation program you wrote for Programming Assignment #2 and #3 and implement it as a 68000 assembly language instruction.

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           ;exception routine
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

```
%1010xxxxDDDDSSSS and 16-bit extension word
```

where %1010 is the op code, xxx are don't cares (i.e. we don't use them), %SSSS, designates the source, and %DDDD designates the destination. All of these are specified in binary. The destination is a data register where the CRC result is to be put. A data register destination will be specified in the form %xnnn where n is a three bit binary number indicating the destination register. In this notation %x000 indicates D0, %x001 indicates D1, etc. up to %x111 which indicates D7. A memory location destination is specified by %1xxx. The mask for the CRC calculation will be provided in an extension word. (Remember that there is more than one choice for the mask, i.e. CCITT or Xmodem among others.). The source is an address register and is indicated in a manner similar to a data register destination. %x000 indicates the source is A0, %x001 indicates the source is A1, etc. up to %x111 indicating the source is A7.

This instruction will compute the CRC. The source address register An has the starting address of the data block to be checked. The destination data register Dn contains the number of bytes in the data block. Register Dn will be overwritten by your routine to place thecalculated CRC into it.

Your program may look like:

```
        ....
        LEA     DATA,A5              ; A5 <- Address of Data Set
        MOVE.W  #BYTES,D3            ; D2 <- number of bytes
        DC.W    %1010000000110101    ; CRC  A5,D3  <--your
                                     ; instruction word
        DC.W    M16                  ; EXTENSION WORD <--your
                                     ; mask extension word
        ....                         ; D3 will have XMODEM CRC
        <another instruction>


M16     EQU     $A001                ; (XMODEM MASK)
```

Include an instruction after your last CRC instruction to show that your program properly returns from the exception.

IMPORTANT NOTE:
You have to set an option in the debugger to enable processing of exceptions. 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. This has been what is happening when you have a TRAP #0 in your programs. 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 but without the message to the journal window.

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 the debugger command:

```
Breakpoint Instruction <address>
```

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

HINT: Remember to set both the User and System Stack Pointer or your program will produce strange results

References:

1. Leo J. Scanlon, The 68000: Principles and Programming
2. Ford & Topp, pages 664-669.

Report format:

Your lab report should use the following format:

1. signed title page with short lab abstract
2. assembler listing of your program - as before not all the data listings need to be included in the program listing.
3. Your program should be capable of handling either a data register or memory address destination and you should include sample debugger screens showing how you tested your CRC instruction in both modes.

Answer the following questions:

1. Explain why you have inserted a DC.W statement in the middle of your code.
2. Explain how you set up the exception vector table to cause the 68000 to jump to your CRC routine.
3. Provide drawings of what is on the system stack immediately before the unimplemented instruction exception, when you execute the first instruction in your exception service routine, and immediately after you return from the exception.