AN IMPLEMENTATION OF THE IEEE STANDARD FOR BINARY FLOATING-POINT ARITHMETIC FOR THE MOTOROLA 6809 MICROPROCESSOR

THESIS

Presented to the Graduate Council of the North Texas State University in Partial Fulfillment of the Requirements

For the Degree of

MASTER OF SCIENCE

By

David Samuel Rosenblum, B.S.
Denton, Texas
August, 1983

This thesis describes a software implementation of the IEEE Floating-Point Standard (IEEE Task P754), which is believed to be an effective system for reliable, accurate computer arithmetic. The standard is implemented as a set of procedures written in Motorola 6809 assembly language. Source listings of the procedures are contained in appendices.
TABLE OF CONTENTS

LIST OF TABLES ........................................... v
LIST OF ILLUSTRATIONS .................................. vi

Chapter

I. INTRODUCTION .......................................... 1
II. A DESCRIPTION OF THE IEEE FLOATING-POINT
    STANDARD ............................................. 6
    Introduction
    Storage Format
    Specification of Results
III. THE DEVELOPMENT OF THE PROCEDURES ............ 16
    Introduction
    The Development Hardware
    Software Support
IV. A DESCRIPTION OF THE PROCEDURES ............... 20
    General Description
    Procedure Parameters
    The Add/Subtract Procedure
    The Multiply Procedure
    The Divide Procedure
V. THE TESTING OF THE PROCEDURES .................. 35
VI. CONCLUSIONS .......................................... 39

Appendix

A. SOURCE LISTING OF THE ADD/SUBTRACT
    PROCEDURE ........................................... 43
B. SOURCE LISTING OF THE MULTIPLY PROCEDURE .... 57
C. SOURCE LISTING OF THE DIVIDE PROCEDURE ....... 69

BIBLIOGRAPHY ............................................. 82

iv
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. Storage Format of Operand Types</td>
<td>7</td>
</tr>
<tr>
<td>II. Results of Arithmetic Operations</td>
<td>12</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>1.</td>
<td>Single-Precision IEEE Floating-Point Format</td>
</tr>
<tr>
<td>2.</td>
<td>Unpacked Format of Operands</td>
</tr>
<tr>
<td>3.</td>
<td>Flowchart of the Add/Subtract Procedure</td>
</tr>
<tr>
<td>4.</td>
<td>Flowchart of the Multiply Procedure</td>
</tr>
<tr>
<td>5.</td>
<td>Flowchart of the Divide Procedure</td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION

In the past ten years great advances in technology have occurred in the development of fast, high-density, low-cost, reliable computer hardware. With the advent of very large scale integration (VLSI) technology there will soon be available computer chips the cost and size of a common microprocessor with the computing power of an IBM 370 mainframe computer. Through the placement of over one million gates or transistors on a .25 inch by .25 inch piece of silicon, circuit miniaturization promises to make highly sophisticated computational power and mathematical processing capabilities available to practically anyone who requires such power. The applications of this technology are numerous. Complex simulations of such phenomena as atmospheric behavior would become more feasible once such processors are in production.

The heart of these "supercomputers" is the arithmetic unit. It is the actual implementation of the arithmetic capabilities of a computer which hinders the improvement of mathematical processing. This would not be true if computers had the ability to perform arithmetic with infinite precision. However, since this is not the case, great care must be taken to ensure that results remain accurate within a computer's
finite precision without committing and propagating severe roundoff errors. With the implementation of integer arithmetic there is little or no problem as long as the arithmetic is performed within the bounds specified implicitly by the computer's word size. It is the implementation of real arithmetic which makes the attainment of accurate mathematical results difficult.

The implementation of real arithmetic is usually done with some type of floating-point scheme, although there are other methods being developed, such as Hensel Code (1, 2). All floating-point systems are similar with respect to the storage of numbers: (1) a bit to indicate the sign of the number, a "mantissa" or fraction part (usually less than one) which contains the digits of the number to some implicit base, and an exponent indicating the placement of the radix point. Most floating-point systems are nothing more than a type of scientific notation.

For the most part floating-point systems have ranged from poor to mediocre. They usually employ a sloppy system of rounding, and the rather liberal tolerance by these systems of "unnormalized" operands, in which the leading digit(s) of the mantissa is (are) zero, can only serve to reduce the accuracy and useful precision of results.
Research into the distribution of the numbers in various floating-point systems has revealed several other problems inherent in many systems, such as the flushing of underflowed operands to the value zero (3). In an attempt to create a floating-point system which takes care of all of the problems associated with previous systems, a proposed standard for floating-point arithmetic has been developed by the Microprocessor Standards Committee of the Computer Society of the Institute of Electrical and Electronics Engineers (IEEE) (3).

The integrity of the arithmetic performed by the system described in the standard merits the development of software or hardware to perform such arithmetic. For most computer systems such hardware is not available; thus, if a set of software procedures were available to perform the arithmetic as outlined in the proposed standard, it would be a simple matter to interface these procedures with an application program; in addition, the application programmer would require only a familiarity with the basic storage format of numbers in the system.

The pages that follow describe such an effort as undertaken by the author. A set of procedures has been developed for the Motorola 6809 microprocessor, written in 6809 assembly language, that perform addition, subtraction, multiplication and division on input operands that are in IEEE single-precision floating-point format. The implementation of the arithmetic in these procedures adheres completely to the standard (3). A
supplementary implementation guide (4) aided the development of the procedures.

In order to understand the development and use of the procedures, one must understand the mechanics of the standard itself. A brief description of the floating-point standard and storage format is contained in Chapter II. Chapter III describes the hardware and software used to develop the procedures. Chapter IV describes the procedures and the algorithms they are based on. Chapter V describes briefly the method used to test the procedures. Finally, Chapter VI offers some conclusions about the research effort and possible further improvements. Assembly language source listings of the procedures are contained in the Appendices.
CHAPTER BIBLIOGRAPHY


CHAPTER II

A DESCRIPTION OF THE IEEE FLOATING-POINT STANDARD

Introduction

A complete, detailed description of the floating-point standard may be found in the most recent draft of the standard (1). This chapter focuses only on those portions of the standard relevant to the research effort being described.

The standard prescribes that all implementations conforming to the standard must support the single precision format (1). Besides prescribing the method of storage of the operands, the standard also specifies the method of obtaining results based on the types of the two input operands, the selection of one of four rounding modes and the choice of one of two infinity arithmetic modes. Also specified is the use of exception flags.

Storage Format

The single format consists of a sign bit (s), an eight-bit exponent (e) and a twenty-three-bit mantissa (f) stored

```
s e f
0 1 8 9 31
```

Fig. 1--Single-precision IEEE floating-point format
as shown in Figure 1. This thirty-two-bit format lends well to storage in four eight-bit bytes on a Motorola 6809 system. Table I gives a description of the five different types of operands represented in the format based on

TABLE I

STORAGE FORMAT OF OPERAND TYPES

<table>
<thead>
<tr>
<th>Operand Type</th>
<th>Value of e</th>
<th>Value of f</th>
<th>Represented Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalized Numbers</td>
<td>0 &lt; e &lt; 255</td>
<td>Any</td>
<td>(-1)^e x 2^e-127 x(1.f)</td>
</tr>
<tr>
<td>Denormalized Numbers</td>
<td>0</td>
<td>Nonzero</td>
<td>(-1)^5 x 2^-126 x(0.f)</td>
</tr>
<tr>
<td>Signed Zeros</td>
<td>0</td>
<td>0</td>
<td>(-1)^5 x 0</td>
</tr>
<tr>
<td>Signed Infinities</td>
<td>255</td>
<td>0</td>
<td>(-1)^5 x ∞</td>
</tr>
<tr>
<td>Not-a-Number (NaN)</td>
<td>255</td>
<td>Nonzero</td>
<td>. . . . .</td>
</tr>
</tbody>
</table>

(unsined) exponent and fraction values. The table is a modification of a table contained in Coonen's implementation guide (2). A sign bit of zero indicates a positive number, and a sign bit of one indicates a negative number. The stored exponent is a biased value and is treated as an unsigned number between zero and 255 inclusive. The NaN (not-a-number) is a reserved operand which has no arithmetic value, but instead serves as the result of an invalid operation; its use as an operand always causes the result to be (possibly the same) NaN.

The exponent is a biased-value power of two. Because of the greater accuracy ensured by using only normalized operands, finite operands with (biased) exponent values greater than zero are assumed to be normalized. Because
the radix used in the system is two, the leading bit of a normalized mantissa would always be one. By making this leading bit implicit, an extra bit of precision is obtained in the mantissa without requiring the added expense of additional memory. This implicit leading one is assumed to occur in the mantissa. This implicit leading one is assumed to occur to the left of the binary point, while the explicit stored mantissa occurs to the right of the binary point.

To achieve gradual underflow of operands toward zero instead of the sudden flushing of underflowed operands to zero, the standard allows for "denormalized" numbers, in which the exponent has its minimum (biased) value of zero. In this case the implicit bit to the left of the binary point is zero. Because of the difference between the hidden bits in normalized and denormalized finite operands, and because of the desire to remove large, inconsistent gaps from between successive operands, the bias on the exponent is 127 for normalized operands and 126 for denormalized operands.

Indeed, the magnitude (base two) of the largest denormalized number is $0.111111111111111111111111 \times 2^{-126}$. The next highest value in magnitude beyond this number using twenty-four bits of precision is $1.000000000000000000000000 \times 2^{-126}$, which is exactly the magnitude of the smallest normalized number.
It is a simple matter to compute the range of numbers represented in the format by finite, nonzero operands. The smallest nonzero denormalized operand has a magnitude (base ten) of $1.0 \times 2^{-149}$, which is approximately equal to $1.4 \times 10^{-45}$. The largest finite, normalized operand has a magnitude (base ten) of almost $2.0 \times 2^{127}$, which is approximately $3.4 \times 10^{38}$.

To facilitate computation of results, an unpacked format was utilized by the procedures which places the sign, exponent and mantissa into separate bytes, as shown in Figure 2. The sign bit of the IEEE format is propagated

```
s e f 1-4 guard bytes
| s | e | f | 1-4 guard bytes |
+---+---+---+---------------|
| 0 | 7 | 8 | 15 16 39 40     |
```

Fig. 2--Unpacked format of operands

throughout the sign byte of the unpacked format. The mantissa of the unpacked format always is a value less than one; that is, the unpacked mantissa is assumed to occur completely to the right of the binary point. In unpacking the mantissa of normalized operands, the leading one is made explicit, shifting it and the rest of the mantissa one bit to the right so that the leading bit is to the right of the binary point; doing this necessitates the incrementing of the exponent to reflect the shift. In unpacking the mantissas of denormalized operands, no shifting is necessary because the hidden bit is zero; however, because the bias
on the exponents of denormalized operands is one less than the bias on the exponents of normalized operands, it is necessary to increment the exponent of a denormalized operand to ensure the attainment of correct results. In either case, the incremented exponent is stored in one byte of the unpacked operand and the mantissa is stored in three bytes. Because the unpacked format is simply used to aid in performing the actual arithmetic operations, no unpacking is performed if either input operand is zero, infinity or a NaN. Depending on which operation is being performed, one to four guard bytes are maintained to account for the increased precision obtained in performing the arithmetic operations.

Specification of Results

Results of operations on the same input operands may vary only when different rounding modes or infinity arithmetic modes are selected by the user of the procedures.

The standard specifies four rounding modes to be provided by all implementations of the standard. Assuming that $Z$ is an intermediate result bracketed by two consecutive numbers $Z_1$ and $Z_2$ that are representable in the IEEE format, such that $Z_1 < Z < Z_2$, then the rounding modes are defined as follows:

(1) the Round-toward-Zero (RZ) mode selects the smaller of $Z_1$ and $Z_2$ in magnitude as the result;
(2) the Round-to-Nearest (RN) mode selects the nearer of $Z_1$
and \( Z_2 \) to \( Z \) as the result, with the one of \( Z_1 \) and \( Z_2 \) having an even-valued mantissa selected in case of a tie; (3) the Round-toward-Positive-Infinity (RP) mode selects \( Z_2 \) as the result; and (4) the Round-toward-Minus-Infinity (RM) mode selects \( Z_1 \) as the result.

The two modes of infinity arithmetic prescribed by the standard are defined as follows:

(1) the affine mode, in which \(-\infty < +\infty\); and
(2) the projective mode, in which \(-\infty = +\infty\).

The round-toward-infinity modes necessitate the use of a so-called "sticky bit" beyond the rounding precision to simulate infinite precision arithmetic. The sticky bit is maintained in the second most significant bit of the most significant guard byte of the unpacked operand and is defined to be the logical OR of all bits shifted out during normalization or mantissa alignment. Thus, it does not interfere with correct rounding-to-nearest, but instead serves to indicate the presence of an intermediate result in infinite precision located between two numbers representable in the IEEE format and allows correct results to be obtained when rounding toward infinity.

In addition to providing a numerical result, the standard specifies that all implementations must set certain exception flags when appropriate. The possible exceptions are defined as follows:
(1) **Invalid-Operation**, which occurs when a NaN is used as an input operand or certain invalid combinations of input operand types are used;

(2) **Underflow**, which occurs when after rounding and normalization a nonzero result in infinite precision would be represented as zero in the finite precision of the IEEE format;

(3) **Overflow**, which occurs when after rounding and normalization the exponent of the result exceeds the maximum exponent allowable for a finite, normalized operand;

(4) **Division-by-Zero**; and

(5) **Inexact-Result**, which occurs when the unrounded result cannot be represented exactly in the IEEE format, but instead lies between two consecutive numbers representable in the IEEE format as described above.

Table II summarizes the results delivered by operations on the sixteen possible combinations of input operand types. The table summarizes Coonen's implementation.

| TABLE II |
| RESULTS OF ARITHMETIC OPERATIONS |

<table>
<thead>
<tr>
<th>X OP Y</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+0</td>
</tr>
<tr>
<td>+0</td>
<td>a</td>
</tr>
<tr>
<td>W</td>
<td>d</td>
</tr>
<tr>
<td>+∞</td>
<td>g</td>
</tr>
<tr>
<td>NaN</td>
<td>X</td>
</tr>
</tbody>
</table>
TABLE II—Continued

<table>
<thead>
<tr>
<th>Letter</th>
<th>Operation</th>
<th>Rounding Mode</th>
<th>Infinity Mode</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Add/Sub.</td>
<td>RN,RZ,RP</td>
<td>All</td>
<td>+0</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Add/Sub.</td>
<td>RM</td>
<td>All</td>
<td>-0</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Multiply</td>
<td>All</td>
<td>All</td>
<td>+0</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Divide</td>
<td>All</td>
<td>All</td>
<td>NaN</td>
<td>Invalid-Op.</td>
</tr>
<tr>
<td>b</td>
<td>Add/Sub.</td>
<td>All</td>
<td>All</td>
<td>+0</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Mult./Div.</td>
<td>All</td>
<td>All</td>
<td>Y</td>
<td>None</td>
</tr>
<tr>
<td>c</td>
<td>Add/Sub.</td>
<td>All</td>
<td>All</td>
<td>NaN</td>
<td>Invalid-Op.</td>
</tr>
<tr>
<td></td>
<td>Multiply</td>
<td>All</td>
<td>All</td>
<td>+0</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Divide</td>
<td>All</td>
<td>All</td>
<td>X</td>
<td>None</td>
</tr>
<tr>
<td>d</td>
<td>Add/Sub.</td>
<td>All</td>
<td>All</td>
<td>+0</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Multiply</td>
<td>All</td>
<td>All</td>
<td>+∞</td>
<td>Div.-by-Zero</td>
</tr>
<tr>
<td>e</td>
<td>Add/Sub.</td>
<td>All</td>
<td>All</td>
<td>X Ø Y</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Multiply</td>
<td>All</td>
<td>All</td>
<td>X Ø Y</td>
<td>None</td>
</tr>
<tr>
<td>f</td>
<td>Add/Sub.</td>
<td>All</td>
<td>All</td>
<td>Y</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Multiply</td>
<td>All</td>
<td>All</td>
<td>±∞</td>
<td>None</td>
</tr>
<tr>
<td>g</td>
<td>Add/Sub.</td>
<td>All</td>
<td>All</td>
<td>+0</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Divide</td>
<td>All</td>
<td>All</td>
<td>X</td>
<td>None</td>
</tr>
<tr>
<td>h</td>
<td>Add/Sub.</td>
<td>All</td>
<td>All</td>
<td>NaN</td>
<td>Invalid-Op.</td>
</tr>
<tr>
<td></td>
<td>Mult./Div.</td>
<td>All</td>
<td>All</td>
<td>±∞</td>
<td>None</td>
</tr>
<tr>
<td>i</td>
<td>Add/Sub.</td>
<td>All</td>
<td>Affine, signs pos.</td>
<td>±∞</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Add/Sub.</td>
<td>All</td>
<td>Affine, signs neg.</td>
<td>-∞</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Add/Sub.</td>
<td>All</td>
<td>Affine, signs different</td>
<td>NaN</td>
<td>Invalid-Op.</td>
</tr>
<tr>
<td></td>
<td>Multiply</td>
<td>All</td>
<td>All</td>
<td>±∞</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Divide</td>
<td>All</td>
<td>All</td>
<td>NaN</td>
<td>Invalid-Op.</td>
</tr>
</tbody>
</table>

guide (2). Input operands are denoted by the capital letters X and Y, where X denotes the first operand in each of the arithmetic operations. W represents a nonzero, finite number. Any result obtained from operations on finite operands must be rounded and checked for underflow and overflow. The result of an overflow is a signed infinity, and the result of an underflow is a signed zero. The lower case
letters a through i in the first part of the table refer to a description of the result as contained in the second part of the table. When the sign of a result is indicated as plus or minus, the sign is derived according to the usual rules of the operation being performed. The circled operators indicate the formation of the result according to typical floating-point algorithms such as those described in Chapter 4 of Knuth (3). NaNs shown as results in the second part of Table II are chosen arbitrarily.
CHAPTER BIBLIOGRAPHY


CHAPTER III
THE DEVELOPMENT OF THE PROCEDURES

Introduction

The IEEE floating-point arithmetic procedures were developed at North Texas State University during the academic year 1982-83. The first part of this chapter gives a description of the hardware on which the procedures were developed. The second part describes software support systems used to develop the procedures.

The Development Hardware

The IEEE floating-point arithmetic procedures were developed on a Percom SBC/9 SS-50 Bus compatible single card microcomputer system, containing a Motorola MC6809 (one megahertz clock) microprocessor. The system contains 56K-bytes of memory, including a ROM containing the PSYMON system monitor, Version 1.20 (1). The monitor was used mainly for its memory and register examination/modification functions, which aided greatly the testing of the procedures.

Connected to an RS-232 serial I/O port in the system is an ADM-3A CRT terminal running at 9600 baud. This terminal was used for all entering, editing and testing of the procedures. Also connected to the system at another serial I/O port is a DECwriter IV printer, which was used solely for
obtaining hardcopy listings of the source programs. Finally, two single-sided double-density disk drives connected to the system were utilized for permanent storage of the source files and object code of the procedures. The disk drives were used under the control of the FLEX operating system described below.

Software Support

The procedures were developed with the aid of the 6809 FLEX Operating System (2, 3), resident on a floppy disk. The kernel of FLEX is booted into memory each time the system is powered up. In addition the the usual file manipulation commands available with FLEX, a superb editor (4) and mnemonic assembler (5) are available with the FLEX system.

It is the FLEX text editor program that was used for the majority of the development of the procedures. The FLEX editor allows for extremely easy input, moving, copying, deletion and pattern location, along with the other usual functions expected of an editor, such as displaying or re-numbering lines of text. The ease of use of the editor made the creation of the procedures much easier than was initially expected.

The FLEX Assembler is a fast macro assembler. Its macro language was not used in the development of the procedures. It is rather sophisticated for an assembler for a microcomputer system, having available a wide assortment
of features. The source listings it produces, of course, display the location counter and object code in hexadecimal of the corresponding source text, which can be entered in free format. The assembler also accepts conditional assemblies and a comprehensive set of assembler directives that control such functions as pagination of the source listing, allocation of memory and placement of headings in the source listing. Because no system (or human) is free of fault, backup disk copies were made religiously of both the operating system and the source files of the procedures themselves during the course of the development of the procedures.
CHAPTER BIBLIOGRAPHY


CHAPTER IV

A DESCRIPTION OF THE PROCEDURES

General Description

The IEEE floating-point arithmetic procedures were designed with two goals in mind: they should be fast, and they should be modular. Both goals were achieved. An average addition or subtraction operation requires approximately 0.5 milliseconds using the procedures. An average multiplication operation requires around one millisecond using the procedures. Finally, an average division operation requires less than two milliseconds using the procedures. These are admirable speeds for floating-point operations; however, these times are only rough approximations to the actual speed of the procedures, as they were determined by timing the speed of a cycle of 32,768 random operations and then dividing this empirical speed by 32,768.

Because assembler languages do not lend themselves well to the notions of structured programming, the procedures are by no means modular in the sense of a hierarchical top-down design. It was felt that an excessive number of subroutine calls, which would be required by a hierarchical design, would slow down processes that by their nature already require a great number of processor cycles. However, as an
examination of the source listings of the procedures will verify (Appendices A, B and C), the procedures are modular in the sense that they are divided into small blocks of code, each performing a single function, such as testing the X operand for a zero value, testing the Y operand for a value of infinity, or rounding the result.

All of the procedures are built around the same basic algorithm, but as Table II in Chapter II shows, each operation treats certain combinations of input operands in a way unique to that operation. Although there is an amount of redundant code among the procedures, it was felt that the procedures would be more useful coded as separate subroutines for each operation rather than as a single package. For example, a matrix multiplication program would probably require only addition, subtraction and multiplication, but not division. Taken together, the object code of the procedures occupies approximately 2.5K-bytes of memory. This small amount of memory space further justifies the coding of the procedures as separate subroutines.

As has been stated previously, the operations conform completely to the specifications in the standard (1). Two sources were consulted for a general approach to floating-point processing: Knuth's floating-point algorithms (2) and some material by Sterbenz (3). For the multiplication procedure, a special algorithm was designed by the author to take advantage of the multiplication instruction available
on the Motorola 6809 microprocessor. For the division procedure, the "comparison" algorithm described in Chapter 9 of Mano (4) was used. To ensure that the procedures conform the standard, Coonen's implementation guide (5) was consulted for the special processing required by the various combinations of input operands. To aid in the writing of the actual 6809 assembly language source code, the Motorola 6809 Programming Guide (6) was consulted.

Procedure Parameters

Each procedure has exactly the same set of input and output parameters. Upon entry to each procedure, the X index register must contain the address of the first of four contiguous bytes of memory which contain the first operand (addend, minuend, multiplicand or dividend) of the intended operation in the IEEE single-precision format. The Y index register must contain the address of the first of four contiguous bytes of memory which contain the second operand (augend, subtrahend, multiplier or divisor) of the intended operation in the IEEE single-precision format. The A accumulator must contain in bits seven (the highest-order bit) and six, respectively, a value indicating the desired rounding mode to be used, coded as follows:
(1) the value 00 is used to indicate round-to-nearest;
(2) the value 01 is used to indicate round-toward-zero;
(3) the value 10 is used to indicate round-toward-positive-infinity; and
(4) the value 11 is used to indicate round-toward-minus-infinity.

Furthermore, bit zero (the lowest-order bit) of the A accumulator must be zero if projective infinity arithmetic is to be used or one if affine infinity arithmetic is to be used.

Upon exiting each procedure, the X index register will contain the address of the first byte of the four-byte IEEE single-precision format result. The N (negative) flag of the condition code register (CCR) will be set if the result is negative; the Z (zero) flag of the CCR will be set if the result is zero; and the V (overflow) flag of the CCR will be set if an overflow or underflow occurred or if the result was a NaN (not-a-number). Bits six, five, four, three and two of the A accumulator are used as the exception flags and will indicate the occurrence of Invalid-Operation, Overflow, Underflow, Division-by-Zero and Inexact-Result exceptions, respectively, when set (equal to one). The remaining bits in the CCR and A accumulator, the contents of the Y index register and the contents of all other registers not used for parameter passing will remain unchanged from their values at entry to the procedures.
The remainder of this chapter gives a description of some of the processing unique to the algorithms of each procedure.

The Add/Subtract Procedure

Addition and subtraction are performed by the same procedure, but have different entry points within the procedure. The only difference between the two operations is that there is extra code at the subtraction entry point that negates the second (Y) operand so that the subtraction operation becomes an equivalent addition operation. In both cases, after the Y operand is loaded into storage local to the procedure, control is passed to the main body of the procedure to perform the addition. Figure 3 contains a flowchart of the Add/Subtract Procedure.

The Multiply Procedure

The availability of a multiply instruction for 8-bit unsigned integers in the 6809 instruction set simplified the actual multiplication processing. Each byte of the first three-byte unpacked input mantissa is multiplied together with each byte of the second three-byte unpacked input mantissa using the multiply instruction to yield a total of nine two-byte intermediate products that never generate a carry-out. Appropriate alignment and addition of these intermediate products produce a six-byte final product with no carry-out. This six-byte result is then normalized before
Fig. 3--Flowchart of the Add/Subtract Procedure
Fig. 3—Continued
Fig. 4--Flowchart of the Multiply Procedure
Fig. 4--Continued
rounding takes place. Figure 4 contains a flowchart of the Multiply procedure.

The Divide Procedure

The "comparison" algorithm described in Mano (4, p. 340) was chosen instead of the more common "restoring" or "non-restoring" algorithms because it was felt that these latter methods are more amenable to hardware implementation, whereas the "comparison" method requires a simpler and smaller amount of source code that is easier to comprehend and debug. First, both operands are normalized in a modification of dividend alignment to ensure that the most significant bit of each unpacked input mantissa is one. Then the dividend is extended to the right with four bytes of zeroes to form an initial partial remainder that is large enough to ensure a result accurate to the precision of the IEEE format. The divisor is then repeatedly shifted one bit to the right and is subtracted from the partial remainder only when it is smaller in comparison with the partial remainder. The shifting ceases after the divisor has been shifted through four bytes (thirty-two times), which is enough to ensure that further subtraction will have no effect upon the value of the resulting quotient. While the shifting is taking place, a corresponding bit in the quotient (initially given a value of zero) is set to one
Fig. 5 -- Flowchart of the Divide Procedure
Fig. 5—Continued
each time a subtraction takes place. Figure 5 contains a flowchart of the Divide procedure.
CHAPTER BIBLIOGRAPHY


CHAPTER V

THE TESTING OF THE PROCEDURES

The procedures were tested thoroughly, exhausting every possible combination of pairs of input operand types (e.g., the X operand a NaN and the Y operand a zero, the X operand a NaN and the Y operand infinity, etc.), and each of these pairs was tested with each possible combination of rounding mode and infinity arithmetic. In all cases the results conformed exactly to the standard. Special emphasis was placed on testing a large number of finite non-zero operands to ensure that the actual floating-point arithmetic algorithms, the hearts of the procedures, gave consistently correct results, for each possible rounding mode. Again, the results conformed completely to the standard.

To aid in the testing of finite non-zero operands, three sources were used for examples: examples worked out by hand by the author, examples given in an unpublished paper by Song (1) and examples given by Coonen in the standard as published by Computer magazine (not part of the standard) entitled "Illustrative Examples" (2). In each case the results given in the examples were verified by hand before the
procedures were tested on the examples. Without exception the procedures computed the results correctly.

As was stated previously, testing of the procedures was done using the system monitor. First the binary (object) file of a procedure was loaded into memory. The final return-from-subroutine (RTS) instruction of each procedure was changed to a software interrupt (SWI) instruction (which on this particular system causes the registers and monitor prompt to be displayed) each time the procedure was loaded using the monitor's memory examination/modification command. This was done to simplify the testing process so that a dummy subroutine call would not be needed and results would be available for immediate inspection. Once the binary file was set up, the X and Y index registers were load with arbitrarily chosen memory addresses that would be loaded with the input operands to be tested. The A accumulator was loaded next with the appropriate values to indicate the desired rounding and infinity modes to be tested. Finally, the memory addresses pointed to by the X and Y index registers were loaded with the operands to be tested. In all instances register and memory initialization were performed using the monitor's register and memory examination/modification commands. A jump command initiated the execution of the procedure being tested, and the final register values were displayed immediately after execution took place. To verify the results, the A accumulator was first checked for the setting of any
exception flags. Then the four bytes referenced by the \( X \) index register were inspected to ensure that the correct value of the result was obtained.

The consistency with which the procedures produced correct results during testing leads the author to conclude that with high probability the procedures are error-free.
CHAPTER BIBLIOGRAPHY


CHAPTER VI

CONCLUSIONS

A set of procedures has been described which performs floating-point arithmetic according to the IEEE floating-point standard (1). Their implementation as subroutines using registers as pointers to the operands suggests that they can be easily used with other software written for the 6809 microprocessor such as applications software requiring precise real arithmetic. The procedures could also serve easily as a subunit of a high-level language translator such as a BASIC or Pascal compiler. Such an implementation would provide a BASIC or Pascal programmer the same, if not better, precision in real arithmetic as most other compilers provide along with the integrity of results obtained using the IEEE system.

The relatively small scope of the described research effort necessitated that the procedures conform to only the minimum requirements of an implementation of the standard, namely that all implementations of the standard must provide single-precision arithmetic capabilities giving results that conform to the standard along with user-directed rounding and infinity arithmetic (1). Further improvements to the procedures would include the capability for double-precision
arithmetic, along with the implementation of useful functions such as a compare function, a remainder function, an absolute value function, a negate function, a move function, conversion between floating-point and integer formats and a function to compute the integer portion of a floating-point number. Beyond these basic arithmetic capabilities, the next step would be the development of procedures which perform the transcendental functions on input floating-point operands such as the sine and natural logarithm functions. However, all of the above improvements are a relatively simple matter to implement once the basic single-precision arithmetic functions are provided. For example, a compare function is nothing more than a subtraction operation that returns a boolean value instead of a numerical result, suggesting the inclusion of a third entry point in the Add/Subtract Procedure. Even the sine function when implemented as a numerical approximation is nothing more than a series of additions and multiplications with predetermined coefficients.

The advent of supercomputing power merits the development of precise, reliable floating-point software and hardware. Although relatively few theoretical conclusions have been drawn regarding the behavior of floating-point systems in general, the IEEE effort represents the most significant advance made in the improvement of the arithmetic processing capabilities of digital computers. It
is possible that in the future, totally different systems may be discovered which can perform real arithmetic with a reliability and precision unattainable in current systems; however, until such systems are developed, thoroughly tested and become widely accepted by the computing community, it is necessary to provide integrity in arithmetic processing, using current knowledge about floating-point systems, which complements naturally the sophistication in hardware of the new generation of supercomputers.
CHAPTER BIBLIOGRAPHY

APPENDIX A

SOURCE LISTING OF THE ADD/SUBTRACT PROCEDURE
THIS PROCEDURE IMPLEMENTS THE SINGLE-PRECISION FLOATING-POINT ADD AND SUBTRACT OPERATIONS OF THE IEEE FLOATING-POINT STANDARD. ON INPUT TO THE PROCEDURE, THE FOLLOWING REGISTERS MUST BE LOADED WITH APPROPRIATE DATA AS FOLLOWS:

\$ X -- ADDRESS OF FIRST BYTE OF IEEE SINGLE-PRECISION FORMAT ADDEND/MINUEND.

\$ Y -- ADDRESS OF FIRST BYTE OF IEEE SINGLE-PRECISION FORMAT AUGEND/SUBTRAHEND.

\$ A -- BYTE DESCRIBING THE Rounding MODE AND INFINITY ARITHMETIC TO BE USED, FORMED AS FOLLOWS:

\$ \$ BIT 7 -- 0 IF PROJECTIVE INFINITY MODE
( -INFINITY = +INFINITY ) (DEFAULT)
\$ 1 IF AFFINE INFINITY MODE
( -INFINITY < +INFINITY )
\$ BITS 1 & 0 -- 00 IF ROUND-TO-NEAREST (RN) MODE (DEFAULT)
\$ 01 IF ROUND-TOWARD-ZERO (RZ) MODE
\$ 10 IF ROUND-TOWARDS +INF (RP) MODE
\$ 11 IF ROUND-TOWARDS -INF (RN) MODE
\$ BITS 2 - 6 -- DON'T CARE

UPON SUCCESSFUL TERMINATION OF THE PROCEDURE, THE FOLLOWING REGISTERS WILL CONTAIN APPROPRIATE DATA AS FOLLOWS:

\$ X -- ADDRESS OF FIRST BYTE OF IEEE SINGLE-PRECISION FORMAT SUM/DIFFERENCE.

\$ CCR -- CONDITION CODES SET ACCORDING TO VALUE OF RESULT AS FOLLOWS:

\$ N -- SET IF RESULT IS NEGATIVE
\$ Z -- SET IF RESULT IS ZERO
\$ V -- SET IF OVERFLOW OR UNDERFLOW OCCURRED OR RESULT IS NOT-A-NUMBER (NAN)
\$ E,F,H,I,C -- UNCHANGED FROM ENTRY VALUES

\$ A -- STICKY FLAGS SET AS FOLLOWS:

\$ BIT 6 -- INVALID OPERATION FLAG
\$ BIT 5 -- OVERFLOW FLAG
BIT 4 -- UNDERFLOW FLAG
BIT 3 -- DIVISION-BY-ZERO FLAG (NOT USED BY FADD)
BIT 2 -- INEXACT RESULT FLAG
BITS 7,1,0 -- UNCHANGED FROM ENTRY

THE VALUES IN THE REGISTERS NOT USED FOR PARAMETER PASSING
WILL BE UNCHANGED UPON SUCCESSFUL TERMINATION OF THE PROCEDURE.

0000  ORG $0000

WORKING STORAGE FOR FADD.

0000  ORBYTE RMB 1  BYTE USED IN FORMING CCR CONTENTS UPON EXIT
0001  FLAGS RMB 1  BYTE DESIGNATING INFINITY AND Rounding MODES
0002  XVAL RMB 1  X EQUALS INFINITY IF XVAL<0 (BIT 7 SET), 0 IF XVAL=0
0003  YVAL RMB 1  Y EQUALS INFINITY IF YVAL<0 (BIT 7 SET), 0 IF YVAL=0
0004  XSIGN RMB 1  SIGN BYTE OF X OPERAND IN UNPACKED FORMAT
0005  XOP RMB 4  AREA TO UNPACK X OPERAND AS INPUT
0006  XGUARD RMB 1  GUARD BYTE FOR X OPERAND
0007  YSIGN RMB 1  SIGN BYTE OF Y OPERAND IN UNPACKED FORMAT
0008  YOP RMB 4  AREA TO UNPACK Y OPERAND AS INPUT
0009  YGUARD RMB 1  GUARD BYTE FOR Y OPERAND
000A  RFACTR RMB 2  AREA FOR Rounding FACTOR

0100  ORG $0100

0100  FSUB EQU *  ENTRY POINT FOR FLOATING-POINT SUBTRACT.
0100  34 6D  PSHS  U,Y,DP,B,CC SAVE REGISTERS
0102  5F  CLR B,DP INITIALIZ E DPR TO ZERO
0103  1F  9B  TFR B,DP
0105  84 83  ANDA #583  CLEAR STICKY FLAGS
0107  97 01  STA FLG AS SAVE MOD E BIT S AND STICKY FLA GS
0109  E6  A4  LD B,Y LOAD FIRST BYTE OF Y OPERAND IN ACCB
010B  CB 80  ED RB #80  CHANGE SIGN OF Y OPERAND SO THAT ADD ROUTINE CAN BE USED
010D  D7 0B  ST B YOP SAVE FIRST BYTE OF Y OPERAND
010F  E6 21  LDB 1,Y SAVE REMAINING 3 BYTES OF Y OPERAND
0111  D7 0C  ST B YOP+1
0113  EE 22  LD V 2,Y (TWO AT ONCE)
0115  0F 0D  STU YOP+2
0117  20 11  BRA ATSYNN GO TO MAIN BODY OF ADD ROUTINE

0119  FADD EQU *  ENTRY POINT FOR FLOATING-POINT ADD.
BEGINNING OF MAIN BODY OF FLOATING-POINT ADD

ATSYNN TESTS THE Y OPERAND FOR NOT-A-NUMBER.

012A DC 08 LDD YOP SET UP FOR EXPONENT TEST
012C 1E 09 EXG A,B FIRST BYTE OF Y IN B, SECOND BYTE IN A
012E 48 LSLA GET LOW ORDER BIT OF EXPONENT IN CARRY
012F 1D SEX UNPACK SIGN INTO A
0130 59 ROLB EXPONENT NOW IN B
0131 97 0A STA YSIGN SAVE UNPACKED SIGN
0133 C1 FF CMPB #$FF CHECK FOR MAXIMUM EXPONENT
0135 23 20 BLO TESTYO NO NEED TO TEST FOR NAN IF EXPONENT NOT =$FF
0137 00 0E TST YOP+3 TEST FOR NONZERO FRACTION WHEN EXPONENT =$FF
0139 26 0F BNE YISNAN IF NONZERO BYTE Y IS NAN
013B 00 0D TST YOP+2 ELSE TEST ANOTHER BYTE
013D 26 0B BNE YISNAN Y IS NAN IF NONZERO BYTE
013F 96 0C LDA YOP+1 ELSE TEST FINAL BYTE
0141 48 LSLA SHIFT OUT LEAST SIGNIFICANT EXPONENT BIT FOR PROPER TEST
0142 26 06 BNE YISNAN IF NONZERO BYTE THEN Y IS NAN
0144 0F 03 CLR YVAL ELSE Y IS INFINITY AT THIS POINT
0146 03 03 COM YVAL MAKE YVAL NEGATIVE TO INDICATE INFINITY
0148 20 27 BRA ALOADX GO AHEAD AND PROCESS X (LEAVE Y UNMODIFIED)

014A 96 01 YISNAN LDA FLAGS GET MODE/FLAG BYTE
014C 8A 40 ORA #$40 SET INVALID OPERATION STICKY FLAG
014E 97 01 STA FLGS SAVE STICKY FLAGS
0150 C6 02 LDB #$02 SET UP BYTE TO SET Y BIT OF CCR
0152 D7 00 STB ORBYTE SAVE CCR MODIFICATION BYTE
0154 8E 0008 LDX #YOP X POINTS TO (UNMODIFIED) Y OPERAND AS RESULT
0157 7E 0449 JMP FAEIIIT WRAP UP AND EXIT

015A 5D TESTYO TSTB CHECK IF EXPONENT(Y) = 0
015B 26 10 BNE YNOTO GO ON IF NONZERO EXPONENT
015D 00 0E TST YOP+3 CHECK FOR Y OPERAND = 0
015F 26 0C BNE YNOTO GO ON IF NONZERO
0161 00 0D TST YOP+2 KEEP TESTING
0163 26 08 BNE YNOTO GO ON IF NONZERO
0165 00 0C TST YOP+1 ONCE MORE
0167 26 04 BNE YNOTO GO ON IF NONZERO
0169 0F 03 CLR YVAL CLEAR VALUE INDICATOR, INDICATING ZERO VALUE
016B 29 04 BRA ALOADX GO ON AND PROCESS X OPERAND
016D C6 7F YNOTO LDB #$7F NON-NEGATIVE VALUE Indicator MEANS Y FINITE, NONZERO
016F D7 03 STB YVAL SAVE VALUE INDICATOR
0171 ALOADX EQU $

\* ALOADX LOADS THE X OPERAND FOR PROCESSING.

0171 EE 04 LDU ,X FIRST TWO BYTES OF X OPERAND
0173 DF 05 STU XOP
0175 EE 02 LDU 2,1 LAST TWO BYTES OF X OPERAND
0177 DF 07 STU XOP+2

0179 ATSXNN EQU $

\* ATSXNN TESTS THE X OPERAND FOR NOT-A-NUMBER.

0179 DC 05 LDD XOP SET UP FOR EXPONENT TEST
017B 1E 59 EXG A,B FIRST BYTE OF X IN B, SECOND BYTE IN A
017D 48 LSLA GET LOW ORDER BIT OF EXPONENT IN CARRY
017E 1D SEI UNPACK SIGN INTO A
017F 59 ROLB EXPONENT NOW IN B
0180 97 04 STA ISIGN SAVE UNPACKED SIGN
0182 C1 FF CMPB ##FF CHECK FOR MAXIMUM EXPONENT
0184 25 23 BLO TESTXO NO NEED TO TEST FOR NAN IF EXPONENT NOT = ##FF
0186 00 08 TST XOP+3 TEST FOR NONZERO FRACTION WHEN EXPONENT = ##FF
0188 26 0F BNE XISNAN IF NONZERO BYTE X IS NAN
018A 00 07 TST XOP+2 ELSE TEST ANOTHER BYTE
018C 26 0B BNE XISNAN X IS NAN IF NONZERO BYTE
018E 96 06 LDA XOP+1 ELSE TEST FINAL BYTE
0190 48 LSLA SHIFT OUT LEAST SIGNIFICANT EXPONENT BIT FOR PROPER TEST
0191 26 06 BNE XISNAN IF NONZERO BYTE THEN X IS NAN
0193 0F 02 CLR XVAL ELSE X IS INFINITY AT THIS POINT
0195 03 02 CMI XVAL MAKE XVAL NEGATIVE TO INDICATE INFINITY
0197 20 27 BRA ATSXNF GO AHEAD AND CHECK FOR INFINITIES (LEAVE X UNMODIFIED)

0199 96 01 XISNAN LDA FLAGS SET MODE/FLAG BYTE
019B 8A 40 ORA ##40 SET INVALID OPERATION STICKY FLAG
019D 97 01 STA FLAGS SAVE STICKY FLAGS
019F C6 02 LDB ##02 SET UP BYTE TO SET V BIT OF CCR
01A1 D7 00 STB ORBYTE SAVE CCR MODIFICATION BYTE
01A3 BE 0005 LDX #XOP X POINTS TO (UNMODIFIED) X OPERAND AS RESULT
01A6 7E 0449 JMP FAEIXT WRAP UP AND EXIT

01A9 5D TESTXO TSTB CHECK IF EXPONENT(1) = 0
01A9 26 10 BNE IXOTO GO ON IF NONZERO EXPONENT
01AC 00 08 TST XOP+3 CHECK FOR X OPERAND = 0
01AE 26 0C BNE XNOTO GO ON IF NONZERO
01B0 0D 07 TST XOP+2 KEEP TESTING
01B2 26 08 BNE XNOTO GO ON IF NONZERO
01B4 0D 06 TST XOP+1 ONCE MORE
01B6 26 04 BNE XNOTO GO ON IF NONZERO
01B8 0F 02 CLR XVAL CLEAR VALUE INDICATOR, INDICATING ZERO VALUE
01BA 20 04 BRA ATSXNF GO ON AND CHECK FOR INFINITIES (LEAVE X UNMODIFIED)
01BC C6 7F XNOTO LDB #$7F NON-NEGATIVE VALUE INDICATOR MEANS X FINITE, NONZERO
01BE D7 02 STB XVAL SAVE VALUE INDICATOR
ATSXNF EQU $  

ATSXNF TESTS THE X AND Y OPERANDS FOR INFINITY VALUES AND PERFORMS THE APPROPRIATE OPERATIONS WHEN INFINITY VALUES ARE ENCOUNTERED.

01CO 0D 02 TST XVAL TEST X OPERAND VALUE INDICATOR FOR INFINITY INDICATION
01C2 2C 15 BGE XNOTNF IF X IS FINITE THEN GO ON AND TEST Y FOR INFINITY
01D4 0D 03 TST YVAL ELSE SEE IF Y OPERAND IS ALSO INFINITY
01C6 102D 00E3 LBLT AINFA D GO ADD INFINITIES IF BOTH ARE INFINITY
01CA 5F CLR8 IF Y NOT INFINITY, THEN X AS INFINITY IS RESULT, SO WRAP UP
01CB 0D 04 TST YSIGN SET UP CCR MODIFICATION BYTE
01CD 2C 02 BGE YISINF IF Y OPERAND IS POSITIVE, DO NOT SET N IN CCR
01CF CA 0B ORB #008 ELSE SET N
01D1 D7 00 XISINF STB ORBYTE SAVE CCR MODIFICATION BYTE
01D3 BE 0005 LDX #XOP POINT X TO (UNMODIFIED) X OPERAND
01D6 7E 0449 JMP FAEXIT WRAP UP AND EXIT
01D9 0D 03 XNOTNF TST YVAL TEST Y OPERAND FOR INFINITY WHEN X IS FINITE
01DB 2C 0F BGE ATSTXO IF Y IS NOT INFINITY, GO TEST X FOR ZERO VALUE
01D5 5F CLR8 OTHERWISE Y OPERAND IS RESULT, SO WRAP UP
01DE 0D 0A TST YSIGN SET UP CCR MODIFICATION BYTE
01E0 2C 02 BGE YISINF IF Y OPERAND IS POSITIVE, DO NOT SET N IN CCR
01E2 CA 0B ORB #008 ELSE SET N
01E4 D7 00 YISINF STB ORBYTE SAVE CCR MODIFICATION BYTE
01E6 8E 000B LDX #YOP POINT X TO (UNMODIFIED) Y OPERAND
01E9 7E 0449 JMP FAEXIT WRAP UP AND EXIT

ATSTXO EQU $  

ATSTXO TESTS THE X AND Y OPERANDS FOR ZERO VALUES AND PERFORMS THE APPROPRIATE OPERATIONS WHEN ZERO VALUES ARE ENCOUNTERED.

01EC 0D 02 TST XVAL TEST X OPERAND FOR ZERO VALUE
01EE 2A 15 BNE XNOTZR IF X IS NOT ZERO, GO TEST Y OPERAND FOR ZERO VALUE
01F0 0D 03 TST YVAL ELSE SEE IF Y IS ZERO ALSO
01F2 1027 00E6 LBEQ AIROAD GO ADD ZEROS IF BOTH ARE ZERO
01F6 5F CLR8 OTHERWISE Y OPERAND IS RESULT, SO WRAP UP
01F7 0D 0A TST YSIGN SET UP CCR MODIFICATION BYTE
01F9 2C 02 BGE YRLTI IF Y OPERAND IS POSITIVE, DO NOT SET N IN CCR
01FB CA 0B ORB #008 ELSE SET N
01FD 0D 00 YRLTI STB ORBYTE SAVE CCR MODIFICATION BYTE
01FF 8E 000B LDI #YOP POINT X TO (UNMODIFIED) Y OPERAND
0202 7E 0449 JMP FAEXIT WRAP UP AND EXIT
0205 0D 03 XNOTZR TST YVAL TEST Y OPERAND FOR ZERO VALUE WHEN X NOT ZERO
0207 2A 0F BNE AUNPAK IF Y NOT ZERO, GO AHEAD AND PERFORM ALIGNMENT
0209 5F CLR8 OTHERWISE Y OPERAND IS RESULT, SO WRAP UP
020A 0D 04 TST XSIGN SET UP CCR MODIFICATION BYTE
020C 2C 02 BGE XRSLTI IF X OPERAND IS POSITIVE, DO NOT SET N IN CCR
020E CA 0B ORB #008 ELSE SET N
0210 D7 00 XRSLTI STB ORBYTE SAVE CCR MODIFICATION BYTE
0212 BE 0005 LDX #XOP POINT X TO (UNMODIFIED) X OPERAND
0215 7E 0449 JMP FAEIIT WRAP UP AND EXIT

0218 AUNPAK EQU $7

* AT THIS POINT IN THE PROCEDURE, BOTH THE X AND Y OPERANDS
* ARE FINITE AND NONZERO. AUNPAK UNPACKS THE OPERANDS FOR
* PROPER COMPUTATION OF THE RESULT.

021B 0F 0F UNPKY CLR YGUILD CLEAR GUARD BYTE
021A DC 0B LDD YOP CHECK IF EXPONENT(Y) = 0
021C 59 ROLB EXPONENT BIT
021B 49 ROLA
021E 27 0B BEQ UNPKYO UNPACK FOR 0 EXPONENT
0220 4C INCA INCREMENT EXPONENT SINCE HIDDEN BIT IS SHIFTED IN
0221 97 0B STA YOP FINISH UNPACKING Y, SAVE EXPONENT
0223 96 0C LDA YOP+1 GET MOST SIGNIFICANT FRACTION BYTE
0225 8A 80 BRA #$80 SET HIGH ORDER BIT OF FRACTION SINCE EXPONENT NONZERO
0227 97 0C STA YOP+1
0229 20 09 BRA UNPKX GO ON AND PROCESS X OPERAND

022B 4C UNPKYO INCA INCREMENT EXPONENT (BIAS ON 0 EXPONENT IS 126, NOT 127)
022D 97 0B STA YOP SAVE EXPONENT
022E 08 0E LSL YOP+3 FINISH UNPACKING Y OPERAND
0230 09 0D ROL YOP+2
0232 09 0C ROL YOP+1

0234 0F 09 UNPKX CLR IGUARD CLEAR GUARD BYTE
0235 DC 05 LDD XOP CHECK IF EXPONENT(X) = 0
0236 59 ROLB EXPONENT BIT
0239 49 ROLA
023A 27 0B BEQ UNPKXO UNPACK FOR 0 EXPONENT
023C 4C INCA INCREMENT EXPONENT SINCE HIDDEN BIT WAS SHIFTED IN
023D 97 05 STA XOP FINISH UNPACKING X, SAVE EXPONENT
023F 96 06 LDA XOP+1 GET MOST SIGNIFICANT FRACTION BYTE
0241 8A 80 BRA #$80 SET HIGH ORDER BIT OF FRACTION SINCE EXPONENT NONZERO
0243 97 06 STA XOP+1
0245 20 09 BRA AALIGN GO ON AND ALIGN OPERANDS

0247 4C UNPKXO INCA INCREMENT EXPONENT (BIAS ON 0 EXPONENT IS 126, NOT 127)
0249 97 05 STA XOP SAVE EXPONENT
024A 08 08 LSL XOP+3 FINISH UNPACKING X OPERAND
024C 09 07 ROL XOP+2
024E 09 06 ROL XOP+1

0250 AALIGN EQU *

* AALIGN MAKES THE EXPONENTS OF THE OPERANDS EQUAL WITH APPRO-
* PRIATE SHIFTING OF THE FRACTIONS IN PREPARATION FOR ADDITION
* OF THE NUMBERS.
LDX $XOP
USE X AND Y TO INDEX TO FINAL FIRST AND SECOND OPERANDS
LDY $YOP
LDB XOP
PUT EXPONENT(X) IN D
CLRA
(TREAT AS 8-BIT UNSIGNED NUMBER EXTENDED TO 16 BITS)
SUBL YOP
SUBTRACT OUT Y EXPONENT
SBCA $0
EXponent(X) - EXponent(Y) NOW IN D
CMPD $0
TEST FOR NEGATIVE DIFFERENCE
BLT NEDIF
IF DIFFERENCE < 0, SWITCH OPERANDS
BGT TESTIP
ELSE IF DIFFERENCE POSITIVE, 60 AND CHECK EXPONENT VALUE
LDA $XSIGN
ELSE CHECK MAGNITUDES AND ADD (EXPONENTS ARE EQUAL)
EDRA $YSIGN
TEST SIGNS
LBEQ ADDMS8
IF SIGNS OF OPERANDS ARE EQUAL, ADD FRACTIONS
LDD $1
ELSE SWITCH FRACTIONS IF X FRACTION SMALLER THAN Y FRACTION
CMPB $1
TEST FIRST BYTE OF FRACTIONS
BLO EXSXY
IF FRACTION(X) < FRACTION(Y) THEN SWITCH OPERAND POINTERS
LDD $XOP+2
ELSE TEST REMAINING TWO BYTES OF FRACTIONS
CMPD $YOP+2
EXSXY
ASUBMS8
IF FRACTION(X) >= FRACTION(Y) GO AND SUBTRACT FRACTIONS
EXG X,Y
ELSE SWITCH OPERAND POINTERS
LBRA ASUBMS8
GO ON AND SUBTRACT MAGNITUDES
NEGIF COMA
NEGATE DIFFERENCE WHEN DIFFERENCE NEGATIVE
COMB
SWITCH OPERAND POINTERS
ADDX #1
LEXG X,Y
TEST FOR NEGATIVE DIFFERENCE
CMPD #25
IF DIFFERENCE > 25, CURRENT Y OPERAND INSIGNIFICANT
BLE ALIGNY
IF DIFFERENCE <= 25, 60 ALIGN CURRENT Y OPERAND
LDB $40
ELSE SET STICKY BIT IN GUARD BYTE FOR PROPER Rounding
STB 4,X
LBRA ATRND
GO ON AND Rounding
LSR 1,Y
SHiFT FRACTION(Y) UNTiL EXPONENTS EQual
ROR 2,Y
SHiFT INTO GUARD Byte ALSO
ROR 3,Y
INC +,Y
INCreMENT EXPONENT(Y) TO REFLECT SHiFTiNG
ADDX $-1
DECReMENT DIFFEREnCE IN D TO TEST FOR END OF ALIGNMENT
BGT ALIGNY
IF DIFFERENCE NOT ZERO, SHiFT AGAIN
ATSIGN LDA -1, X
TEST WHETHER TO ADD OR SUBTRACT MAGNITUDES
EDRA -1,Y
TEST SIGNS OF OPERANDS
BGE PROJNF
IF SIGNS OF OPERANDS ARE EQUAL, ADD FRACTIONS
BRA ASUBMS8
ELSE SUBTRACT FRACTIONS
AInfAD EQU *
AInfAD PERFORMS THE ADDITION OF TWO INFINITY OPERANDS
AND SIGNALS INVALID OPERATION IF PROJECTIVE INFINITY IS
BEING USED.
LDX -1,X
TEST WHETHER TO ADD OR SUBTRACT MAGNITUDES
EDRA -1,Y
TEST SIGNS OF OPERANDS
BGE PROJNF
IF PROJECTIVE INFINITY MODE, SIGNAL INVALID OPERATION

ABROAD ADDS TWO ZERO OPERANDS TOGETHER ACCORDING TO THE ROUNCING MODE SELECTED.

LDX #XOP  ; POINT X TO X OPERAND AS TEMPORARY RESULT
TST #SIGN ; TEST SIGNS OF OPERANDS
BNE XNEG  ; IF X OPERAND NEGATIVE THEN GO TEST SIGN OF Y OPERAND
TST #SIGN ; TEST SIGN OF Y OPERAND HERE TO SEE IF BOTH ARE POSITIVE
BEQ ASETO ; IF BOTH ARE POSITIVE, NO NEED TO CHECK ROUNCING MODE
LDB #01  ; ELSE CHECK FOR ROUNCING MODE IN USE
RORB       ; GO ON IF RN OR RP MODE (BIT 0 OF FLAGS = 0)
BCC ASETO  ; GO ON IF RP MODE (BIT 1 OF FLAGS = 0)
LDX #YOP  ; POINT X TO Y OPERAND (-0) AS FINAL RESULT (RM MODE)
BRA ASETO  ; GO SET UP RESULT

XNEG  ; TEST SIGN OF Y OPERAND
BNE ASETO ; IF BOTH ARE NEGATIVE, NO NEED TO CHECK ROUNCING MODE
LDB #01  ; ELSE CHECK FOR ROUNCING MODE IN USE
RORB       ; GO ON IF RN OR RP MODE (BIT 0 OF FLAGS = 0), MAKE X POSITIVE
BCC POSTVX ; ROUNCING MODE IS POSSIBLY RM
RORB       ; ELSE GO ON IF RM MODE (BIT 1 OF FLAGS = 1)
BCC ASETO  ; IF X OPERAND IS POSITIVE, DO NOT SET N IN CCR
LDB #04  ; X POINTS TO RESULT, SO WRAP UP
TST ,X  ; SET UP CCR MODIFICATION BYTE, SETTING Z IN CCR
BNE XSIGN ; IF X OPERAND IS POSITIVE, DO NOT SET N IN CCR
ORB #00  ; ELSE SET N

ARoad EQU $
0308 07 00 XRSLT STB ORBYTE SAVE CCR MODIFICATION BYTE
0307 7E 0449 JMP FAEXIT WRAP UP AND EXIT

0310 AADDMG EQU 1

* AADDMG PERFORMS ADDITION OF FINITE OPERANDS WHEN THEIR SIGNS ARE THE SAME.

0310 EC 03 LDD 3, X PERFORM THE ADDITION ON THE TWO LEAST SIGNIFICANT BYTES
0312 EC 23 ADDD 3, Y (THE GUARD BYTES ARE ADDED TOGETHER ALSO)
0314 ED 03 STD 3, X SAVE RESULT
0316 EC 01 LDD 1, X ADD WITH CARRY THE REMAINING SIGNIFICANT BYTES
0318 E9 22 ADCB 2, Y
031A A9 21 ABCA 1, Y
031C ED 01 STD 1, X SAVE RESULT
031E 24 41 BCC NORMAL IF THERE WAS NO CARRY OUT, GO ON AND ROUND RESULT
0320 66 01 ROR 1, X ELSE SHIFT IN CARRY
0322 66 02 ROR 2, X
0324 66 03 ROR 3, X
0326 66 04 ROR 4, X
0328 6C 84 INC , X INCREMENT EXPONENT TO REFLECT SHIFT
032A 20 35 BRA NORMAL 60 ON AND ROUND RESULT

032C ASUBMG EQU *

* ASUBMG PERFORMS ADDITION OF FINITE OPERANDS WHEN THEIR SIGNS ARE DIFFERENT.

032C EC 03 LDD 3, X PERFORM THE SUBTRACTION ON THE TWO LEAST SIGNIFICANT BYTES
032E EC 23 SUBD 3, Y (THE GUARD BYTES ARE SUBTRACTED ALSO)
0330 ED 03 STD 3, X SAVE RESULT
0332 EC 01 LDD 1, X SUBTRACT WITH CARRY THE REMAINING SIGNIFICANT BYTES
0334 E2 22 SBCB 2, Y
0336 A2 21 SBCA 1, Y
0338 ED 01 STD 1, X SAVE RESULT
033A 26 25 BNE NORMAL GO ON IF NONZERO RESULT
033C 6D 03 TST 3, X
033E 26 21 BNE NORMAL ROUND IF NONZERO RESULT
0340 6D 04 TST 4, X
0342 26 1D BNE NORMAL ROUND IF NONZERO RESULT
0344 D6 01 LDB FLAGS CHECK Rounding MODE
0346 56 RORB
0347 24 0E BCC PIRS LT IF RI OR AP MODE (BIT 0 OF FLAGS = 0) MAKE X POSITIVE
0349 56 RORB
034A 24 0B BCC PIRS LT IF RN MODE (BIT 1 OF FLAGS = 0) MAKE X POSITIVE
034C C6 0C LDB $00C SET UP CCR MODIFICATION BYTE TO SET N,Z
034E 07 00 STB ORBYTE SAVE CCR MODIFICATION BYTE
0350 C6 80 LDB $80 REPACK RESULT WITH NEGATIVE SIGN AND ZERO EXPONENT
0352 E7 04 STB , X SAVE REPACKED FIRST RESULT BYTE
0354 7E 0449 JMP FAEXIT WRAP UP AND EXIT
0356 C6 04 PIRS LT LDB $04 SET UP CCR MODIFICATION BYTE TO SET Z
ORBYTE
SAVE CCR MODIFICATION BYTE
REPACK RESULT WITH POSITIVE SIGN AND ZERO EXPONENT
SAVE REPACKED FIRST RESULT BYTE
JMP FAEXIT WRAP UP AND EXIT

NORMAL EQU +
NORMAL PERFORMS NORMALIZATION OF THE FINITE RESULT.

LOAD EXPONENT(X) INTO 0
TEST FOR MINIMUM EXPONENT
G0 ON AND ROUND WITHOUT NORMALIZATION
ELSE SEE IF HIGH ORDER BIT OF FRACTION IS 1
IF HIGH ORDER BIT SET (FRACTION (1) THEN
G0 ON AND ROUND
ELSE SHIFT FRACTION AND DECREMENT EXPONENT
(INCLUDE GUARD BYTE IN SHIFT)
G0 BACK AND TEST UNTIL FRACTION IS NORMALIZED

ATRND EQU +
ATRND DETERMINES WHETHER OR NOT TO PERFORM Rounding DEPENDING ON THE Rounding MODE SELECTED.

SET UP INITIAL Rounding FACTOR
TEST FOR MINIMUM EXPONENT TO SEE IF Rounding NECESSARY
BNE TSGARD IF NOT MINIMUM EXPONENT, GO ON
LSL RFACTR+1 ELSE SHIFT Rounding FACTOR
RL RFACTR
LDB 3,X
TEST FOR ZERO GUARD BIT WITH MINIMUM EXPONENT
RORB
BRA INXACT IF GUARD BIT 1 THEN GO ON
TSGARD TST 4,X ELSE TEST GUARD BYTE
BEQ AREPAK IF GUARD ZERO THEN NO NEED TO ROUND
LDB FLAGS SET INXACT RESULT FLAG
RORB #04
STD FLAGS SAVE STICKY FLAGS
RORB CHECK Rounding MODE IN USE
BCC RPORRN IF BIT 0 OF FLAGS = 0 THEN Rounding MODE IS RP OR RN
RORB ELSE CHECK FOR RZ OR RN MODE
BEQ AREPAK NO NEED TO ROUND IF RZ MODE
TST -1,X OTHERWISE TEST SIGN OF RESULT (RN MODE)
BEQ AREPAK IF RESULT IS POSITIVE IN RN MODE THEN NO NEED TO ROUND
RND1NF LSL RFACTR+1 ELSE CHANGE Rounding FACTOR
ROR RFACTR
BRA AROUND GO ON AND ROUND
RORB CHECK FOR RP OR RN MODE
54

```
03A6 24 06  BCC  RNMODE  IF BIT 1 OF FLAGS = 0 THEN GO PREPARE FOR AN RNDINF
03BA 6D 1F  TST  -1,X  OTHERWISE TEST SIGN OF RESULT (RP MODE)
03AA 26 13  BNE  AREPAK  IF RESULT IS NEGATIVE IN RP MODE THEN NO NEED TO RND
03AC 20 F1  BRA  RNINF  ELSE GO BACK AND CHANGE ROUNDING FACTOR

03AE E6 B4  RNMODE  LDB  ,X  DETERMINE WHICH BIT TO ROUND AT BY CHECKING EXPONENT
03B0 C1 01  CMPB  #1
03B2 27 0F  BEQ  MINEXP  GO ROUND FOR MINIMUM EXPONENT
03B4 E6 04  LDD  4,X  CHECK GUARD BYTE FOR TIE BETWEEN TWO FRACTIONS
03B6 C1 80  CMPB  #0
03B8 22 15  BHI  AROUND  IF NO TIE GO ON AND ROUND
03BA E6 03  LDD  3,X  IF GUARD BYTE INSIGNIFICANT, NO NEED TO ROUND
03BE 56  RORB  IF MOST SIGNIFICANT DISCARDED BIT = 0, NO TIE OCCURED
03BF 24 26  BCC  AREPAK  IF LEAST SIGNIFICANT BIT ALREADY 0, LEAVE ALONE
03C1 20 06  BRA  AROUND  ELSE CHECK TO SEE IF LEAST SIGNIFICANT PRESERVED BIT IS 0

03C3 6D 04  MINEXP  TST  4,X  TEST FOR ZERO GUARD BYTE
03C5 26 0B  BNE  AROUND  GO ON AND ROUND BECAUSE THERE IS NO TIE
03C7 E6 03  LDD  3,X  IF LEAST SIGNIFICANT DISCARDED BIT IS 1
03C9 56  RORB
03CA 24 18  BCC  AREPAK  IF LESS SIGNIFICANT DISCARDED BIT = 0, LEAVE ALONE
03CC 56  RORB  ELSE CHECK TO SEE IF LEAST SIGNIFICANT PRESERVED BIT IS 0
03CD 24 18  BCC  AREPAK  IF LEAST SIGNIFICANT PRESERVED BIT IS 0, LEAVE ALONE

03CF EC 03  AROUND  LDD  3,X  ADD ROUNDING FACTOR TO RESULT
03D1 D3 10  ADDD  RFACTR
03D3 ED 03  STD  3,X
03D5 EC 01  LDD  1,X  FINISH ROUNDING
03D7 C9 00  ABCB  #0
03DB 89 00  ADCA  #0
03DB ED 01  STD  1,X
03DC 24 0B  BCC  AREPAK  IF NO CARRY OUT THEN GO ON AND REPACK RESULT
03DF 66 01  ROR  1,X  ELSE SHIFT IN CARRY
03E1 66 02  ROR  2,X
03E3 66 03  ROR  3,X
03E5 EC 84  INC  ,X  INCREMENT EXPONENT TO REFLECT SHIFT

03E7 AREPAK EQU  

1  AREPAK REPACKS THE RESULT AND CHECKS FOR UNDERFLOW
   AND OVERFLOW OF THE RESULT.

03E7 6A 84  DEC  ,X  DECREMENT EXPONENT OF RESULT
03EE 27 19  BEQ  REPAKO  IF MINIMUM EXPONENT THEN REPACK AND CHECK FOR UNDERFLOW
03EB E6 84  LDD  ,X  ELSE CHECK FOR OVERFLOW
03ED C1 FF  CMPB  #0
03EF 27 3C  BEQ  REPKNF  IF MAXIMUM EXPONENT THEN REPACK AS INFINITY
03F1 68 01  LSL  1,X  ELSE REPACK NORMALIZED FINITE RESULT
03F3 66 1F  ROR  -1,X  SHIFT IN SIGN
03F5 66 84  ROR  ,X  SHIFT SIGN IN, SHIFTING EXPONENT
03F7 66 01  ROR  1,X  FINISH SHIFTING
03FF SF CLRB  SET UP CCR MODIFICATION BYTE
```
03FA 6D 84  TST ,X
03FC 2C 02  BGE XRSLT4 IF RESULT POSITIVE DO NOT SET N IN CCR
03FC CA 0B  ORB ##08 ELSE SET N
0400 87 00  XRSLT4 STB ORBYTE SAVE CCR MODIFICATION BYTE
0402 20 45  BRA FAEXIT WRAP UP AND EXIT

0404 86 1F  REPAXO ROR -1,X SHIFT IN SIGN
0406 86 84  ROR ,X  SHIFT SIGN IN, SHIFTING EXPONENT
0408 86 01  ROR 1,X SHIFT FRACTION
040A 86 02  ROR 2,X
040C 86 03  ROR 3,X
040E 5F  CLRZ SET UP CCR MODIFICATION BYTE
0410 86 84  TST ,X
0412 2C 02  BGE XRSLT5 IF RESULT POSITIVE DO NOT SET N IN CCR
0413 CA 0B  ORB ##08 ELSE SET N
0415 86 01  XRSLT5 TST 1,X TEST FOR ZERO RESULT
0417 26 12  BNE NOUFLO
0419 86 02  TST 2,X
041B 26 0E  BNE NOUFLO
041D 86 03  TST 3,X
041F 26 0A  BNE NOUFLO IF NO UNDERFLOW THEN DO NOT SET V AND Z IN CCR
0421 CA 06  ORB ##06 ELSE SET V AND Z ALSO
0423 87 00  STB ORBYTE SAVE CCR MODIFICATION BYTE
0425 96 01  LDA FLAGS SET UNDERFLOW STICKY FLAG
0427 8A 10  ORA ##10
0429 97 01  STA FLAGS SAVE STICKY FLAGS
042B 20 1C  NOUFLO BRA FAEXIT WRAP UP AND EXIT

042D 6F 01  REPKNF CLR 1,X SET UP RESULT AS INFINITY
042F 6F 02  CLR 2,X
0431 6F 03  CLR 3,X
0433 86 1F  ROR -1,X SHIFT IN SIGN
0435 86 84  ROR ,X  SHIFT SIGN IN, SHIFTING EXPONENT
0437 86 01  ROR 1,X FINISH SHIFTING
0439 C6 02  LDB ##02 SET UP CCR MODIFICATION BYTE TO SET V IN CCR
043B 86 84  TST ,X
043D 2C 02  BGE XRSLT6 IF RESULT POSITIVE THEN DO NOT SET N IN CCR
043F CA 0B  ORB ##08 ELSE SET N
0441 87 00  XRSLT6 STB ORBYTE SAVE CCR MODIFICATION BYTE
0443 96 01  LDA FLAGS SET OVERFLOW STICKY FLAG
0445 8A 20  ORA ##20
0447 97 01  STA FLAGS SAVE STICKY FLAGS

0449 FAEXIT EQU *

* FAEXIT RESTORES THE REGISTERS, LOADS THE STICKY FLAGS
* AND SETS UP A NEW CCR. X AT THIS POINT IS ALREADY POINTING
* TO THE RESULT.

0449 35 01  PULS CC RESTORE CCR (FROM TOP OF STACK)
044B 1E A8  EX6 CC,A PERFROM CCR MODIFICATION IN A
044D 8A F1  ANDA ##F1 CLEAR X,Z,Y IN CCR
044F 9A 00  ORA ORBYTE SET X,Z,Y IN CCR ACCORDING TO MODIFICATION BYTE
0451 1E 8A  EXG  A,CC  RESTORE CCR
0453 96 01  LDA  FLAGS  LOAD STICKY FLAGS/MODE BYTE INTO A
0455 35 6C  PULS  U,Y,DP,B  RESTORE OTHER REGISTERS
0457 39  RTS  RETURN FROM ROUTINE

0 ERROR(S) DETECTED
APPENDIX B

SOURCE LISTING OF THE MULTIPLY PROCEDURE
THIS PROCEDURE IMPLEMENTS THE SINGLE-PRECISION FLOATING-POINT MULTIPLY OPERATION OF THE IEEE FLOATING-POINT STANDARD.

ON INPUT TO THE PROCEDURE, THE FOLLOWING REGISTERS MUST BE LOADED WITH APPROPRIATE DATA AS FOLLOWS:

- X -- ADDRESS OF FIRST BYTE OF IEEE SINGLE-PRECISION FORMAT MULTIPLICAND.
- Y -- ADDRESS OF FIRST BYTE OF IEEE SINGLE-PRECISION FORMAT MULTIPLIER.
- A -- BYTE DESCRIBING THE ROUNDING MODE AND INFINITY ARITHMETIC TO BE USED, FORMED AS FOLLOWS:
  
  BIT 7 -- 0 IF PROJECTIVE INFINITY MODE
  (-INFINITY = +INFINITY) (DEFAULT)
  1 IF AFFINE INFINITY MODE
  (-INFINITY = +INFINITY)
  
  BITS 1 & 0 -- 00 IF ROUND-TO-NEAREST (RN) MODE (DEFAULT)
  01 IF ROUND-TOWARD-ZERO (RZ) MODE
  10 IF ROUND-TOWARD- +INF (RP) MODE
  11 IF ROUND-TOWARD- -INF (RM) MODE
  
  BITS 2 - 6 -- DON'T CARE

UPON SUCCESSFUL TERMINATION OF THE PROCEDURE, THE FOLLOWING REGISTERS WILL CONTAIN APPROPRIATE DATA AS FOLLOWS:

- X -- ADDRESS OF FIRST BYTE OF IEEE SINGLE-PRECISION FORMAT PRODUCT.
- CCR -- CONDITION CODES SET ACCORDING TO VALUE OF RESULT AS FOLLOWS:
  
  N -- SET IF RESULT IS NEGATIVE
  Z -- SET IF RESULT IS ZERO
  V -- SET IF OVERFLOW OR UNDERFLOW OCCURRED
  OR RESULT IS NOT A NUMBER (NAN)
  E,F,H,I,C -- UNCHANGED FROM ENTRY VALUES
- A -- STICKY FLAGS SET AS FOLLOWS:
  
  BIT 6 -- INVALID OPERATION FLAG
  BIT 5 -- OVERFLOW FLAG
BIT 4 -- UNDERFLOW FLAG

BIT 3 -- DIVISION-BY-ZERO FLAG (NOT USED BY FMUL)

BIT 2 -- INEXACT RESULT FLAG

BITS 7,1,0 -- UNCHANGED FROM ENTRY

THE VALUES IN THE REGISTERS NOT USED FOR PARAMETER PASSING
WILL BE UNCHANGED UPON SUCCESSFUL TERMINATION OF THE PROCEDURE.

0000 ORG $0000

WORKING STORAGE FOR FMUL.

0000 ORBYTE RMB 1 BYTE USED IN FORMING CCR CONTENTS UPON EXIT
0001 FLAG RMB 1 BYTE DESIGNATING INFINITY AND Rounding Modes
0002 XVAL RMB 1 X EQUALS INFINITY IF XVAL.CO (BIT 7 SET), 0 IF XVAL=0
0003 YVAL RMB 1 Y EQUALS INFINITY IF YVAL.CO (BIT 7 SET), 0 IF YVAL=0
0004 XOP RMB 4 AREA TO UNPACK X OPERAND AS INPUT
0005 YOP RMB 4 AREA TO UNPACK Y OPERAND AS INPUT
000C RSLTN RMB 1 SIGN BYTE FOR RESULT
000D RESULT RMB 7 AREA TO CONTAIN RESULTANT PRODUCT
0014 COUNTR RMB 1 COUNTER FOR MULTIPLY LOOP
0015 RFACTR RMB 2 AREA FOR Rounding FACTOR

0100 ORG $0100

0100 FMUL EQU * ENTRY POINT FOR FLOATING-POINT MULTIPLY.

0100 34 6D PSHS U,Y,DP,B,CC SAVE REGISTERS
0102 5F CLRB INITIALIZE DPR TO ZERO
0103 1F 98 TFR B,DP
0105 84 83 ANDA $83 CLEAR STICKY FLAGS
0107 97 01 STA FLAGS SAVE MODE BITS AND STICKY FLAGS
0109 EE 24 LDU ,Y SAVE FIRST TWO BYTES OF Y OPERAND
010B EF 08 STU YOP
010D EE 22 LDU 2,Y SAVE REMAINING TWO BYTES OF Y OPERAND
010F DF 0A STU YOP+2

0111 MTSYN R EQU $ MTSYN TESTS THE Y OPERAND FOR NOT-A-NUMBER.

0111 DC 08 LDD YOP SET UP FOR EXPONENT TEST
0113 1E 89 EXG A,B FIRST BYTE OF Y IN B, SECOND BYTE IN A
0115 4B LS LA SET LOW ORDER BIT OF EXPONENT IN CARRY
0116 1D SEI UNPACK SIGN INTO A
MLOADX EU I
LOADS THE X OPERAND FOR PROCESSING.
FIRST TWO BYTES OF X OPERAND
LAST TWO BYTES OF X OPERAND

MLOADX EQU *
MLOADX LOADS THE X OPERAND FOR PROCESSING.

MTSXNN EQU *
MTSXNN TESTS THE X OPERAND FOR NOT-A-NUMBER.

EXPONENT NOW IN B
CHECK FOR MAXIMUM EXPONENT
NO NEED TO TEST FOR NAN IF EXPONENT NOT = $FF
TEST FOR NONZERO FRACTION WHEN EXPONENT = $FF
IF NONZERO BYTE Y IS NAN
ELSE TEST ANOTHER BYTE
Y IS NAN IF NONZERO BYTE
ELSE TEST FINAL BYTE
IF NONZERO THEN Y IS NAN
ELSE CLEAR VALUE INDICATOR, INDICATING ZERO VALUE
NON-NEGATIVE VALUE INDICATOR MEANS Y FINITE, NONZERO
SAVE VALUE INDICATOR

RLRB
CMPB $FF
BNE YISGNAN
IF NONZERO BYTE Y IS NAN
TST YOP+2
ELSE TEST ANOTHER BYTE
BNE YISGNAN
Y IS NAN IF NONZERO BYTE
LDA YOP+1
ELSE TEST FINAL BYTE
RLSB
BNE YISGNAN
IF NONZERO BYTE THEN Y IS NAN
Y IS INFINITY AT THIS POINT
MAKE YVAL NEGATIVE TO INDICATE INFINITY
LDB #YOP
X POINTS TO (UNMODIFIED) Y OPERAND AS RESULT
JMP FEXIT
WRAP UP AND EXIT

TESTY0
TSTB
CHECK IF EXPONENT(Y) = 0
BNE YNOTO
GO ON IF NONZERO EXPONENT
TST YOP+3
CHECK FOR Y OPERAND = 0
BNE YNOTO
GO ON IF NONZERO
TST YOP+2
KEEP TESTING
BNE YNOTO
GO ON IF NONZERO
TST YOP+1
ONCE MORE
BNE YNOTO
IF NONZERO THEN GO ON
CLR YVAL
ELSE CLEAR VALUE INDICATOR, INDICATING ZERO VALUE
BRA MLOADX
GO ON AND PROCESS X OPERAND
YNOTO
LDB #Y7F
NON-NEGATIVE VALUE INDICATOR MEANS Y FINITE, NONZERO
STB YVAL
SAVE VALUE INDICATOR

LDU ,X
FIRST TWO BYTES OF X OPERAND
STU XOP
STU 2,X
LAST TWO BYTES OF X OPERAND
STU XOP+2

LDU YOP
SET UP FOR EXPONENT TEST
SEX A,B
FIRST BYTE OF X IN B, SECOND BYTE IN A
LSLA SET LOW ORDER BIT OF EXPONENT IN CARRY
ROLB EXPONENT NOW IN B
0165 C1 FF CMPB #4FF CHECK FOR MAXIMUM EXPONENT
0167 25 23 LDL TESTXO NO NEED TO TEST FOR NAN IF EXPONENT NOT = #FF
0169 0D 07 TST XOP+3 TEST FOR NONZERO FRACTION WHEN EXPONENT = #FF
016B 26 0F BNE XISNAN IF NONZERO BYTE X IS NAN
016D 0D 06 TST XOP+2 ELSE TEST ANOTHER BYTE
016F 26 0B BNE XISNAN X IS NAN IF NONZERO BYTE
0171 96 05 LDA XOP+1 ELSE TEST FINAL BYTE
0173 49 LSLA
0174 26 06 BNE XISNAN IF NONZERO BYTE THEN X IS NAN
0176 0F 02 CLR XVAL ELSE X IS INFINITY AT THIS POINT
0178 03 02 CCM XVAL MAKE XVAL NEGATIVE TO INDICATE INFINITY
017A 20 27 BRA MTSXNF GO AHEAD AND CHECK FOR INFINITIES (LEAVE X UNMODIFIED)
017C 96 01 XISNAN LDA FLAGS SET MODE/FLAG BYTE
017E 8A 40 ORA ##40 SET INVALID OPERATION STICKY FLAG
0180 97 01 STA FLAGS SAVE STICKY FLAGS
0182 C6 02 LDB #02 SET UP BYTE TO SET V BIT OF CCR
0184 D7 00 STB ORBYTE SAVE CCR MODIFICATION BYTE
0186 BE 0004 LDX #XO P X POINTS TO (UNMODIFIED) X OPERAND AS RESULT
0189 7E 03E6 JMP FMEXIT WRAP UP AND EXIT
018C 5D TESTXO TSTB CHECK IF EXPONENT(X) = 0
018D 26 10 BNE XNOTO GO ON IF NONZERO EXPONENT
018F 0D 07 TST XOP+3 CHECK FOR X OPERAND = 0
0191 26 0C BNE XNOTO GO ON IF NONZERO
0193 0D 06 TST XOP+2 KEEP TESTING
0195 26 08 BNE XNOTO GO ON IF NONZERO
0197 0D 05 TST XOP+1 ONCE MORE
0199 26 04 BNE XNOTO IF NONZERO THEN GO ON
019B 0F 02 CLR XVAL ELSE CLEAR VALUE INDICATOR, INDICATING ZERO VALUE
019D 20 04 BRA MTSXNF GO ON AND CHECK FOR INFINITIES (LEAVE X UNMODIFIED)
019F C6 7F XNOTO LDB #7F NON-NEGATIVE VALUE INDICATOR MEANS X FINITE, NONZERO
01A1 B7 02 STB XVAL SAVE VALUE INDICATOR

01A3 MTSXNF EDU %
% MTSXNF TESTS THE X AND Y OPERANDS FOR INFINITY VALUES
% AND PERFORMS THE APPROPRIATE OPERATIONS WHEN INFINITY
% VALUES ARE ENCOUNTERED.
01A3 86 04 LDB XOP CREATE SIGN OF RESULT
01A5 1D 04 SEI
01A6 97 0C STA RSLTNS
01A8 26 0B LDB YOP
01A9 1D 04 SEI
01A9 98 0C EDRA RSLTNS
01AB 97 0C STA RSLTNS SAVE RESULT SIGN BYTE
01AF 08 04 LSL XOP PUT RESULT SIGN INTO UNPACKED X OPERAND
01B1 97 0C ASR RSLTNS
01B3 06 04 ROR XOP
01B5 08 06 LSL YOP PUT RESULT SIGN INTO UNPACKED Y OPERAND
01B7 07 0C ASR RSLTNS
01B9 06 08 ROR YOP
XVAL
TEST X OPERAND VALUE INDICATOR FOR INFINITY INDICATION
01BB 0D 02 TST XVAL
01BD 2C 23 BGE XNOTNF IF X IS FINITE THEN GO ON AND TEST Y FOR INFINITY
01BF 5F CLR B
01C0 0D 03 TST YVAL SEE IF Y OPERAND IS ZERO
01C2 26 10 BNE XNOTNF IF Y OPERAND IS NONZERO THEN X OPERAND IS RESULT
01C4 96 01 LDA FLAGS ELSE SET MODE/FLAG BYTE
01C6 DA 40 ORA ##40 SET INVALID OPERATION STICKY FLAG
01C8 97 01 STA FLAGS SAVE STICKY FLAGS
01CA 02 00 ORB ##02 SET UP BYTE TO SET V BIT OF CCR
01CC 06 05 LDB XOP+1 MAKE X OPERAND LOOK LIKE A NOT-A-NUMBER
01CE 7F ORB ##7F
01D0 D7 05 STB XOP+1 SAVE MODIFIED BYTE OF X OPERAND
01D2 20 06 BRA XISINF WRAP UP
01D4 0D 04 YNTZRO TST XOP SET UP CCR MODIFICATION BYTE
01D6 2C 02 BGE XISINF IF X OPERAND IS POSITIVE, DO NOT SET N IN CCR
01D8 CA 08 ORB ##08 ELSE SET N
01DA D7 00 XISINF STB ORBYTE SAVE CCR MODIFICATION BYTE
01DC 0E 0004 LDX #XOP POINT X TO X OPERAND AS RESULT
01DF 7E 03E6 JMP FMEXIT WRAP UP AND EXIT
01E2 0D 03 XNOTNF TST YVAL TEST Y OPERAND FOR INFINITY WHEN X IS FINITE
01E4 2C 23 BGE MTSTKO IF Y IS NOT INFINITY, GO TEST X FOR ZERO VALUE
01E6 5F CLR B
01E7 0D 02 TST YVAL TEST X OPERAND FOR ZERO VALUE
01E9 26 10 BNE XNTZRO IF X IS NONZERO THEN Y OPERAND IS RESULT
01EB 96 01 LDA FLAGS ELSE SET MODE/FLAG BYTE
01ED 0A 40 ORA ##40 SET INVALID OPERATION STICKY FLAG
01EF 97 01 STA FLAGS SAVE STICKY FLAGS
01F1 CA 02 ORB ##02 SET UP BYTE TO SET V BIT OF CCR
01F3 06 09 LDB YOP+1 MAKE Y OPERAND LOOK LIKE A NOT-A-NUMBER
01F5 7F ORB ##7F
01F7 D7 09 STB YOP+1 SAVE MODIFIED BYTE OF Y OPERAND
01F9 20 06 BRA YISINF WRAP UP
01FB 0D 08 XNTZRO TST YOP SET UP CCR MODIFICATION BYTE
01FD 2C 02 BGE YISINF IF Y OPERAND IS POSITIVE, DO NOT SET N IN CCR
01FF CA 08 ORB ##08 ELSE SET N
0201 D7 00 YISINF STB ORBYTE SAVE CCR MODIFICATION BYTE
0203 0E 0008 LDX #YOP POINT X TO Y OPERAND AS RESULT
0206 7E 03E6 JMP FMEXIT WRAP UP AND EXIT

0209 MTSTKO EQU 1

MTSTKO TESTS THE X AND Y OPERANDS FOR ZERO VALUES AND PERFORMS THE APPROPRIATE OPERATIONS WHEN ZERO VALUES ARE ENCLOSED.
LDX #XOP
JMP FMEXIT
POINT X TO (UNMODIFIED) X OPERAND

XNOTZR
TST YVAL
TEST Y OPERAND FOR ZERO VALUE WHEN X NOT ZERO

BNE MUNPAK
IF Y NOT ZERO, GO AHEAD AND PERFORM MULTIPLICATION

LDB ##04
OTHERWISE Y OPERAND IS RESULT, SO WRAP UP

TST YOP
SET UP CCR MODIFICATION BYTE, SETTING Z IN CCR

BGE YRSBIT
IF Y OPERAND IS POSITIVE, DO NOT SET N IN CCR

ORB ##08
ELSE SET N

STB ORBYTE
SAVE CCR MODIFICATION BYTE

LDB TST
BGE LOX

JMP
WRAP UP AND EXIT

XNOTZR
TST
NE
LDB TST
BGE YRSBIT
1$04
YOP
YRSLT4
STB
LOX
JMP

YVAL
LUNPAK
1$04
YOP
YRSLT4
4*08
ORBYTE
HOP
FNEXIT

TEST Y OPERAND FOR ZERO VALUE WHEN X NOT ZERO

IF Y NOT ZERO, GO AHEAD AND PERFORM MULTIPLICATION

OTHERWISE Y OPERAND IS RESULT, SO WRAP UP

SET UP CCR MODIFICATION BYTE, SETTING Z IN CCR

IF Y OPERAND IS POSITIVE, DO NOT SET N IN CCR

ELSE SET N

SAVE CCR MODIFICATION BYTE

POINT X TO (UNMODIFIED) V OPERAND
WRAP UP AND EXIT

0231 MUNPAK EQU ?

; AT THIS POINT IN THE PROCEDURE, BOTH THE X OPERAND AND
; THE Y OPERAND ARE FINITE AND NONZERO. MUNPAK UNPACKS
; THE OPERANDS FOR MULTIPLICATION.

UNPKY LDD YOP
CHECK IF EXPONENT(Y) = 0

ROLA
SHIFT IN LOW ORDER EXPONENT BIT

ROLA

BEQ UNPKY0
GO UNPACK FOR 0 EXPONENT

INCA
INCREMENT EXPONENT SINCE HIDDEN BIT IS SHIFTED IN

STA YOP
FINISH UNPACKING Y, SAVE EXPONENT

LDA YOP+1
GET MOST SIGNIFICANT FRACTION BYTE

DRA ##00
SET HIGH ORDER BIT OF FRACTION SINCE EXPONENT NONZERO

STA YOP+1

UNPKX
YOP
YOP+3
UNPKX
YOP+2
UNPKX
YOP+1

UNPKY0 INCA
INCREMENT EXPONENT (BIAS ON 0 EXPONENT IS 125, NOT 127)

STA YOP
SAVE EXPONENT

LSL YOP+3
FINISH UNPACKING Y OPERAND

ROL YOP+2

ROL YOP+1

UNPKX LDD XOP
CHECK IF EXPONENT(X) = 0

ROLB
SHIFT IN LOW ORDER BIT OF EXPONENT

ROLA

BEQ UNPKX0
UNPACK FOR 0 EXPONENT

INCA
INCREMENT EXPONENT SINCE HIDDEN BIT WAS SHIFTED IN

STA XOP
FINISH UNPACKING X, SAVE EXPONENT

LDA XOP+1
GET MOST SIGNIFICANT FRACTION BYTE

DRA ##80
SET HIGH ORDER BIT OF FRACTION SINCE EXPONENT NONZERO

STA XOP+1

BRA MLTPLY
GO ON AND MULTIPLY

UNPKX0 INCA
INCREMENT EXPONENT (BIAS ON 0 EXPONENT IS 126)

STA XOP
SAVE EXPONENT

LSL XOP+3
FINISH UNPACKING X OPERAND

ROL XOP+2

ROL XOP+1
MLTPY PERFORMS MULTIPLICATION OF THE FRACTIONAL PARTS OF THE OPERANDS.

LDB @XOP+4 X WILL POINT TO THE CURRENT BYTE OF THE X OPERAND
LDB @RESULT+5 U WILL POINT TO WHERE INTERMEDIATE PRODUCTS ARE ADDED

LOAD BYTE OF X FOR MULTIPLICATION
LOAD BYTE OF Y FOR MULTIPLICATION
FORM INTERMEDIATE PRODUCT IN D
ADD INTERMEDIATE PRODUCT INTO RESULT
FORM INTERMEDIATE PRODUCT IN D
ADD INTERMEDIATE PRODUCT INTO RESULT
FORM INTERMEDIATE PRODUCT IN D
ADD INTERMEDIATE PRODUCT INTO RESULT
LOAD BYTE OF X FOR MULTIPLICATION
LOAD BYTE OF Y FOR MULTIPLICATION AND CHANGE POINTER
FORM INTERMEDIATE PRODUCT IN D
ADD INTERMEDIATE PRODUCT INTO RESULT
LOAD BYTE OF X FOR MULTIPLICATION
LOAD BYTE OF Y FOR MULTIPLICATION AND CHANGE POINTER
FORM INTERMEDIATE PRODUCT IN D
ADD INTERMEDIATE PRODUCT INTO RESULT
LOAD BYTE OF X FOR MULTIPLICATION
LOAD BYTE OF Y FOR MULTIPLICATION AND CHANGE POINTER
FORM INTERMEDIATE PRODUCT IN D
ADD INTERMEDIATE PRODUCT INTO RESULT
LOAD EXPONENT(X) INTO D
(TREAT AS 8-BIT UNSIGNED INTEGER EXTENDED TO 16 BITS)
ADD EXPONENT(Y) TO EXPONENT(X)
SUBTRACT EXTRA BIAS OF 127
IF RESULT EXPONENT < 1 GO ON AND TAKE CARE OF UNDERFLOW
ELSE SEE IF HIGH ORDER BIT OF RESULT FRACTION IS 1

NORMAL PERFORMS NORMALIZATION OF THE FINITE RESULT AND CREATES THE EXPONENT OF THE RESULT.
0280 2D 34  BLT  MTRND  IF HIGH ORDER BIT SET (FRACTION < 0) THEN GO ON AND ROUND
0282 1083 0001  CMPD  #1  ELSE TEST FOR MINIMUM EXPONENT
0286 27 2E  BNE  MTRND  IF MINIMUM EXPONENT THEN GO ON AND ROUND
0288 83 0001  SUBD  #1  ELSE SHIFT FRACTION AND DECREMENT EXPONENT
028A 08 13  LSL  RESULT+6  (INCLUDE ALL RESULT BYTES IN SHIFT)
028C 09 12  ROL  RESULT+5
028E 09 11  ROL  RESULT+4
0290 09 10  ROL  RESULT+3
0292 09 0F  ROL  RESULT+2
0294 09 0E  ROL  RESULT+1
0296 20  E5  BRA  NRMLIZ  GO BACK AND TEST UNTIL FRACTION IS NORMALIZED

0298 04 0E  HRMLO  LSR  RESULT+1  SHIFT FRACTION AND INCREMENT EXPONENT
029A 06 0F  ROR  RESULT+2  (INCLUDE ALL RESULT BYTES IN SHIFT)
029C 06 10  ROR  RESULT+3
029E 06 11  ROR  RESULT+4
02A0 06 12  ROR  RESULT+5
02A2 06 13  ROR  RESULT+6
02A4 24 0A  BCC  INEXP  IF 0 WAS SHIFTED OUT THEN GO ON
02A6 1E 03  EXB  D, U  ELSE TEMPORARILY SAVE EXPONENT
02A8 06 11  LDB  RESULT+4  SET STICKY BIT
02AA CA 40  ORB  ##40
02AC 07 11  STB  RESULT+4
02AD 1E 30  EXB  U, D  RESTORE EXPONENT
02A0 C3 0001  INEXP  ADDD  #1  INCREMENT EXPONENT
02A2 2F  E3  BLE  NRML  KEEP SHIFTING UNTIL EXPONENT = 1

02E6  MTRND  EBU  $

* MTRND DETERMINES WHETHER OR NOT TO PERFORM ROUNDING DEFENDING
* ON THE ROUNDING MODE SELECTED.

02E6 8E 0080  LDI  ##80  SET UP INITIAL ROUNDING FACTOR
02E8 9F 15  STI  RFACTR
02EA 1F 03  TFR  D, U  SAVE EXPONENT TEMPORARILY IN U
02ED 1083 0001  CMPD  #1
02F1 26 09  BNE  TSGARD  IF NOT MINIMUM EXPONENT, GO ON
02F3 08 16  LSL  RFACTR+1  SHIFT ROUNDING FACTOR
02F5 09 15  ROL  RFACTR
02F7 06 10  LDB  RESULT+3  TEST FOR ZERO GUARD BIT WITH MINIMUM EXPONENT
02F9 56  RORB
02FA 25 0C  BCS  INXACT  IF GUARD BIT 1 THEN GO ON
02FC 0D 11  TSGARD  TST  RESULT+4  ELSE TEST GUARD BYTES
02FE 26 08  BNE  INXACT  IF NONZERO GUARD BYTE THEN GO ON
0300 0D 12  TST  RESULT+5  ELSE KEEP TESTING
0302 26 04  BNE  INXACT
0304 0D 13  TST  RESULT+6  LAST GUARD BYTE
0306 27 6A  BNE  MREPAC  IF ZERO GUARD BYTES THEN NO NEED TO ROUND
0308 06 01  INXACT  LDB  FLAGS  SET INEXACT RESULT FLAG
030A CA 04  ORB  ##04
030C 07 01  STB  FLAGS  SAVE STICKY FLAGS
030E 56  RORB  CHECK ROUNDING MODE IN USE
0310 24 0D  BCC  RRORRN  IF BIT 0 OF FLAGS = 0 THEN ROUNDING MODE IS RP OR RN
0311 56
0312 24 SE
0314 00 OC
0316 27 5A
0318 08 16
031C 20 3C
031E 56
031F 24 06
0321 GD OC
0323 26 4D
0325 20 Fl
0327 1F 30
0329 1083 0001
032D 27 17
032F 06 11
0331 C1 80
0333 22 25
0335 25 38
0337 0D 12
0339 26 IF
033B 0D 13
033D 26 18
033F 06 10
0341 56
0342 24 2E
0344 20 14
0346 0D 11
0348 26 10
034A 0D 12
034C 26 0C
034E 0D 13
0350 26 08
0352 06 10
0354 56
0355 24 18
0357 56
0358 24 18
035A 0D 10
035C D3 15
036B 06 0E
036D 06 0E
0370 33 41

0312 56
RORB
0314 00
BCC MAEPACK
0316 27
BEQ MAEPACK
0318 08
RNDINF LSL RFACTR+1
031A 09
ROL RFACTR
031C 20
BRA MAROUND GO ON AND ROUND
031E 56
RPOAR RORB
0321 00
BCC RMND OF IF RZ OR RM MODE
0323 26
BNE MAEPACK IF RESULT IS POSITIVE IN RM MODE THEN NO NEED TO ROUND
0325 20
BRA MAROUND GO ON AND ROUND
0327 1F
RNDINF TFR, U,D
0329 1083
CMP D #1
032D 27
BEQ MINEXP GO ROUND FOR MINIMUM EXPONENT
032F 06
LD ST RESULT+4 CHECK GUARD BYTE FOR TIE BETWEEN TWO FRACTIONS
0331 C1
CMP D F80
0333 22
BMI MAROUND IF NO TIE GO ON AND ROUND
0335 25
BLO MAEPACK IF GUARD BYTE INSIFFICANT, NO NEED TO ROUND
0337 0D
TST RESULT+5 ELSE MAKE SURE TIE EXISTS BY CHECKING REST OF GUARD
0339 26
BNE MAROUND
033B 0D
TST RESULT+6
033D 26
BNE MAROUND
033F 06
LD ST RESULT+3 ELSE MAKE LEAST SIGNIFICANT PRESERVED BIT 0 WHEN TIE
0341 56
RORB TEST LEAST SIGNIFICANT PRESERVED BIT
0342 24
BCC MAEPACK IF LEAST SIGNIFICANT BIT ALREADY 0, LEAVE ALONE
0344 20
BRA MAROUND ELSE GO ON AND ROUND
0346 0D
MINEXP TST RESULT+4 TEST FOR ZERO GUARD BYTES
0348 26
BNE MAROUND GO ON AND ROUND BECAUSE THERE IS NO TIE
034A 0D
TST RESULT+5 ELSE FINISH TESTING REMAINING GUARD BYTES
034C 26
BNE MAROUND
034E 0D
TST RESULT+6
0350 26
BNE MAROUND IF NONZERO GUARD THEN GO ON
0352 06
LD ST RESULT+3 ELSE TEST MOST SIGNIFICANT DISCARDED BIT FOR 1
0354 56
RORB
0355 24
BCC MAEPACK IF MOST SIGNIFICANT DISCARDED BIT = 0, NO TIE OCCURED
0357 56
RORB ELSE CHECK TO SEE IF LEAST SIGNIFICANT PRESERVED BIT IS 0
0358 24
BCC MAEPACK IF LEAST SIGNIFICANT PRESERVED BIT = 0, LEAVE ALONE
035A 0D
MAROUND LED RESULT+3 ADD Rounding Factor to Result
035C D3
ADDO RFACTR
036B 06
LDD RESULT+3 FINISH ROUNDING
036D 06
ADDC #0
036E 89
ADCA #0
0366 DD
ADD RESULT+1
0368 24
BCC MAEPACK IF NO CARRY OUT THEN GO ON AND REPACK RESULT
036A 06
RDR RESULT+1 ELSE SHIFT IN CARRY
036C 06
RDR RESULT+2
036E 06
RDR RESULT+3
0370 33
LEAU ,1,0 INCREMENT EXPONENT TO REFLECT SHIFT
MREPAK resects the result and checks for underflow and overflow of the result.

MREPAK resects the result and checks for underflow and overflow of the result.

\[ \begin{align*}
0372 & \text{1E } 30 & & \text{DEC}M & \text{U} & \text{D} & \text{DECREMENT EXPONENT OF RESULT FOR REPACKING} \\
0374 & \text{63 } 0001 & & \text{SUB} & \text{1L} & \text{I} \\
0377 & \text{27 } 1E & & \text{BEQ} & \text{MREPAK} & \text{IF MINIMUM EXPONENT THEN REPACK AND CHECK FOR UNDERFLOW} \\
0379 & \text{1083 } 00FF & & \text{CMP} & \text{##FF} & \text{ELSE CHECK FOR OVERFLOW} \\
037D & \text{2C } 46 & & \text{BGE} & \text{REPKNF} & \text{IF OVERFLOW THEN REPACK AS INFINITY} \\
037F & \text{D7 } 0D & & \text{STD} & \text{RESULT} & \text{ELSE SAVE RESULT EXPONENT FOR REPACKING} \\
0381 & \text{08 } 0E & & \text{LSL} & \text{RESULT+1} & \text{REPACK NORMALIZED FINITE RESULT} \\
0383 & \text{06 } 0C & & \text{ROR} & \text{RSLSN} & \text{SHIFT IN SIGN} \\
0385 & \text{06 } 0D & & \text{ROR} & \text{RESULT} & \text{SHIFT SIGN IN, SHIFTING EXPONENT} \\
0387 & \text{06 } 0E & & \text{ROR} & \text{RESULT+1} & \text{FINISH SHIFTING} \\
0389 & \text{5F } 00 & & \text{CLRB} & \text{SET UP CCR MODIFICATION BYTE} \\
038A & \text{6D } 00 & & \text{TST} & \text{RESULT} & \text{SAVE EXPONENT FOR REPACKING} \\
038C & \text{2C } 02 & & \text{BGE} & \text{XRSLT2} & \text{IF RESULT POSITIVE DO NOT SET N IN CCR} \\
038E & \text{CA } 08 & & \text{OR} & \text{##08} & \text{ELSE SET N} \\
0390 & \text{B7 } 00 & & \text{XRSLT2} & \text{STB} & \text{ORBYTE} & \text{SAVE CCR MODIFICATION BYTE} \\
0392 & \text{BE } 0000 & & \text{LDX} & \#\text{RESULT} & \text{POINT X TO RESULT} \\
0395 & \text{20 } 4F & & \text{BRA} & \text{FMEXIT} & \text{WRAP UP AND EXIT} \\
0397 & \text{D7 } 0D & & \text{REPAKO} & \text{STB} & \text{RESULT} & \text{SAVE EXPONENT FOR REPACKING} \\
0399 & \text{06 } 0C & & \text{ROR} & \text{RSLSN} & \text{SHIFT IN SIGN} \\
039B & \text{06 } 0D & & \text{ROR} & \text{RESULT} & \text{SHIFT SIGN IN, SHIFTING EXPONENT} \\
039D & \text{06 } 0E & & \text{ROR} & \text{RESULT+1} & \text{SHIFT FRACTION} \\
039F & \text{06 } 0F & & \text{ROR} & \text{RESULT+2} & \\
03A1 & \text{06 } 10 & & \text{ROR} & \text{RESULT+3} & \\
03A3 & \text{5F } 00 & & \text{CLRB} & \text{SET UP CCR MODIFICATION BYTE} \\
03A4 & \text{6D } 00 & & \text{TST} & \text{RESULT} & \\
03A6 & \text{2C } 02 & & \text{BGE} & \text{XRSLT3} & \text{IF RESULT POSITIVE DO NOT SET N IN CCR} \\
03A8 & \text{CA } 08 & & \text{OR} & \text{##08} & \text{ELSE SET N} \\
03AA & \text{0D } 0E & & \text{XRSLT3} & \text{TST} & \text{RESULT+1} & \text{TEST FOR ZERO RESULT} \\
03AC & \text{26 } 12 & & \text{BNE} & \text{NOUFLO} & \\
03AE & \text{0D } 0F & & \text{TST} & \text{RESULT+2} & \\
03B0 & \text{26 } 0E & & \text{BNE} & \text{NOUFLO} & \\
03B2 & \text{0D } 10 & & \text{TST} & \text{RESULT+3} & \\
03B4 & \text{26 } 0A & & \text{BNE} & \text{NOUFLO} & \text{IF NO UNDERFLOW THEN DO NOT SET V AND Z IN CCR} \\
03B6 & \text{CA } 06 & & \text{ORB} & \text{##06} & \text{ELSE SET V AND Z ALSO} \\
03B8 & \text{D7 } 00 & & \text{STB} & \text{ORBYTE} & \text{SAVE CCR MODIFICATION BYTE} \\
03BA & \text{96 } 01 & & \text{LDA} & \text{FLAGS} & \text{SET UNDERFLOW STICKY FLAG} \\
03BC & \text{0A } 10 & & \text{ORA} & \text{##10} & \\
03BE & \text{97 } 01 & & \text{STA} & \text{FLAGS} & \text{SAVE STICKY FLAGS} \\
03C0 & \text{BE } 0000 & & \text{NOUFLO} & \text{LDX} & \#\text{RESULT} & \text{POINT X TO RESULT} \\
03C3 & \text{20 } 21 & & \text{BRA} & \text{FMEXIT} & \text{WRAP UP AND EXIT} \\
03C5 & \text{4F } 00 & & \text{REPKNF} & \text{CLRA} & \text{SET UP RESULT AS INFINITY} \\
03C6 & \text{5F } 00 & & \text{CLRB} & \\
03C7 & \text{DD } 0D & & \text{STD} & \text{RESULT} & \\
03C9 & \text{DD } 0F & & \text{STD} & \text{RESULT+2} & \\
03CB & \text{03 } 0D & & \text{COM} & \text{RESULT} & \text{CREATE MAXIMUM EXPONENT IN RESULT} 
\end{align*} \]
03CD 06 0C  ROR  RSLTSN  SHIFT IN SIGN
03CF 06 0D  ROR  RESULT  SHIFT SIGN IN, SHIFTING EXPONENT
03D1 06 0E  RDR  RESULT+1  FINISH SHIFTING
03D3 C6 02  LDB  #02  SET UP CCR MODIFICATION BYTE TO SET V IN CCR
03D5 00 0D  TST  RESULT
03D7 2C 02  BGE  XRLT4  IF RESULT POSITIVE THEN DO NOT SET N IN CCR
03D9 CA 08  ORB  #08  ELSE SET N
03DB 07 00  XRLT4  STB  ORBYTE  SAVE CCR MODIFICATION BYTE
03DD 96 01  LDA  FLAGS  SET OVERFLOW STICKY FLAG
03DF 8A 20  ORA  #00  SAVE CCR MODIFICATION BYTE
03E1 97 01  STA  FLAGS  SAVE STICKY FLAGS
03E3 8E 0000  LD X  @RESULT  POINT X TO RESULT

03E6  FMEXIT  EQU  *
*
*
FMEXIT RESTORES THE REGISTERS, LOADS THE STICKY FLAGS
AND SETS UP A NEW CCR. X AT THIS POINT IS ALREADY POINTING
TO THE RESULT.

03E6 35 01  PULS  CC  RESTORE CCR (FROM TOP OF STACK)
03E8 1E 40  EXG  CC,A  PERFORM CCR MODIFICATION IN A
03EA 84 51  ANDA  #0F  CLEAR N, Z, V IN CCR
03EC 9A 00  ORA  ORBYTE  SET N, Z, V IN CCR ACCORDING TO MODIFICATION BYTE
03EE 1E 8A  EAG  A,CC  RESTORE CCR
03F0 96 01  LDA  FLAGS  LOAD STICKY FLAGS/MODE BYTE INTO A
03F2 35 6C  PULS  U,Y,DP,B  RESTORE OTHER REGISTERS
03F4 39  RTS  RETURN FROM ROUTINE

*  END

0 ERROR(S) DETECTED
APPENDIX C

SOURCE LISTING OF THE DIVIDE PROCEDURE
DAVID S. ROSENBLUM
NORTH TEXAS STATE UNIVERSITY
SPRING 1983

This procedure implements the single-precision floating-point division operation of the IEEE floating-point standard.

On input to the procedure, the following registers must be loaded with appropriate data as follows:

X -- Address of first byte of IEEE single-precision format dividend.

Y -- Address of first byte of IEEE single-precision format divisor.

A -- Byte describing the rounding mode and infinity arithmetic to be used, formed as follows:

   Bit 7 -- 0 if projective infinity mode
           ( -infinity = +infinity ) (default)
           1 if affine infinity mode
           ( -infinity = +infinity )
   Bits 1 & 0 -- 00 if round-to-nearest (RN) mode (default)
                01 if round-toward-zero (RZ) mode
                10 if round-toward- +inf (RP) mode
                11 if round-toward- -inf (RM) mode
   Bits 2 - 6 -- Don't care

Upon successful termination of the procedure, the following registers will contain appropriate data as follows:

X -- Address of first byte of IEEE single-precision format quotient.

CCR -- Condition codes set according to value of result as follows:

   N -- Set if result is negative
   Z -- Set if result is zero
   V -- Set if overflow or underflow occurred
   or result is not a number (NaN)
   E,F,H,I,C -- Unchanged from entry values

A -- Sticky flags set as follows:

   Bit 6 -- Invalid operation flag
   Bit 5 -- Overflow flag
BIT 4 -- UNDERFLOW FLAG
BIT 3 -- DIVISION-BY-ZERO FLAG
BIT 2 -- INEXACT RESULT FLAG
BITS 7,1,0 -- UNCHANGED FROM ENTRY

THE VALUES IN THE REGISTERS NOT USED FOR PARAMETER PASSING
WILL BE UNCHANGED UPON SUCCESSFUL TERMINATION OF THE PROCEDURE.

0000 ORG $0000

* WORKING STORAGE FOR FDIV.

0000 ORBYTE RMB 1   BYTE USED IN FORMING CCR CONTENTS UPON EXIT
0001 FLAGS RMB 1    BYTE DESIGNATING INFINITY AND ROUNCING MODES
0002 XVAL RMB 1     X EQUALS INFINITY IF XVAL<0 (BIT 7 SET), 0 IF XVAL=0
0003 YVAL RMB 1     Y EQUALS INFINITY IF YVAL<0 (BIT 7 SET), 0 IF YVAL=0
0004 RSLTSN RMB 1   SIGN BYTE FOR RESULT
0005 RESULT RMB 5   AREA TO FORM QUOTIENT
0006 XOP RMB 8      AREA TO UNPACK X OPERAND
0012 YOP RMB 8      AREA TO UNPACK Y OPERAND
001A RFACTR RMB 2   AREA FOR ROUNCING FACTORS
001C EXPONT RMB 2   AREA FOR RESULT EXPONENC STORAGE
001E DMASK RMB 1    AREA FOR MASK TO FORM QUOTIENT
001F OUTCTR RMB 1   AREA FOR OUTER LOOP COUNTER FOR DIVISION
0020 INCTR RMB 1    AREA FOR INNER LOOP COUNTER FOR DIVISION

0100 ORG $0100

0100 FDIV EQU $1000  ENTRY POINT FOR FLOATING-POINT DIVISION.

0100 34 6D      PSHS U,Y,DP,B,CC SAVE REGISTERS
0102 5F        CLR B,DP INITIALIZE DPR TO ZERO
0103 1F 9B      TFR B,DP
0105 54 83      ANDA ##63 CLEAR STICKY FLAGS
0107 97 01      STA FLAGS SAVE MODE BITS AND STICKY FLAGS
0109 EE A4      LDU ,Y SAVE FIRST TWO BYTES OF Y OPERAND
010B DF 12      STU YOP
010D EE 22      LDU 2,Y SAVE REMAINING TWO BYTES OF Y OPERAND
010F DF 14      STU YOP+2

0111 DTSYNN EQU $1

% DTSYNN TESTS THE Y OPERAND FOR NOT-A-NUMBER.

0111 BC 12      LDD YOP SET UP FOR EXPONENT TEST
DLADX EQU #

DLADX LOADS THE X OPERAND FOR PROCESSING.

DLUX, X

FIRST TWO BYTES OF X OPERAND

STU XOP

LAST TWO BYTES OF X OPERAND

STU XOP+2

DTSXNN EQU #

DTSXNN TESTS THE X OPERAND FOR NOT-A-NUMBER.

LDD XOP

SET UP FOR EXPONENT TEST

EXG A,B

FIRST BYTE OF Y IN B, SECOND BYTE IN A
$01A3 DTSXNF EQU 1

; DTSXNF TESTS THE X AND Y OPERANDS FOR INFINITY VALUES
; AND PERFORMS THE APPROPRIATE OPERATIONS WHEN INFINITY
; VALUES ARE ENCOUNTERED.

$01A3 26 0A  LDB XOP   CREATE SIGN OF RESULT
$01A3 1D   SEX
$01A6 97 04  STA RSLTSN
$01A6 97 12  LDB YOP
$0162 48  LSLA     GET LOW ORDER BIT OF EXPONENT IN CARRY
$0163 1D   SEX      UNPACK SIGN INTO A
$0164 59   ROLB     EXponent NOW IN B
$0165 C1 FF  CMPB ##FF  CHECK FOR MAXIMUM EXPONENT
$0166 25 23  ELD TESTX0 NO NEED TO TEST FOR NAN IF EXPONENT NOT = ##FF
$0169 00 0B  TST XOP+3  TEST FOR NONZERO FRACTION WHEN EXPONENT = ##FF
$016B 26 0F  BNE XISNAN IF NONZERO BYTE X IS NAN
$016D 00 0C  TST XOP+2  ELSE TEST ANOTHER BYTE
$016F 26 0B  BNE XISNAN X IS NAN IF NONZERO BYTE
$0171 96 08  LDA XOP+1 ELSE TEST FINAL BYTE
$0173 48   LSLA
$0174 26 06  BNE XISNAN IF NONZERO BYTE THEN X IS NAN
$0176 00 02  CLR XVAL ELSE X IS INFINITY AT THIS POINT
$0178 03 02  COM XVAL MAKE XVAL NEGATIVE TO INDICATE INFINITY
$017A 20 27  BRA DTSXNF GO AHEAD AND CHECK FOR INFINITIES (LEAVE X UNMODIFIED)

$017C 96 01  XISNAN LDA FLAGS GET MODE/FLAG BYTE
$017E 8A 40  ORA ##40  SET INVALID OPERATION STICKY FLAG
$0180 97 01  STA FLAGS SAVE STICKY FLAGS
$0182 C6 02  LDB ##02  SET UP BYTE TO SET V BIT OF CCR
$0184 87 00  STS ORBYTE SAVE CCR MODIFICATION BYTE
$0188 6E 000A LDX #XOP X POINTS TO (UNMODIFIED) X OPERAND AS RESULT
$0189 7E 0435 JMP FEXIT WRAP UP AND EXIT

$018C 6D TESTX0 TSTB CHECK IF EXPONENT(X) = 0
$01DB 26 10  BNE XNOTO GO ON IF NONZERO EXPONENT
$018F 00 00  TST XOP+3 CHECK FOR Y OPERAND = 0
$0191 26 0C  BNE XNOTO GO ON IF NONZERO
$0193 00 0C  TST XOP+2 KEEP TESTING
$0195 26 08  BNE XNOTO GO ON IF NONZERO
$0197 00 08  TST XOP+1 ONCE MORE
$0199 26 04  BNE XNOTO IF NONZERO THEN GO ON
$019B 0F 02  CLR XVAL ELSE CLEAR VALUE INDICATOR, INDICATING ZERO VALUE
$019D 20 04  BRA DTSXNF GO ON AND CHECK FOR INFINITIES (LEAVE X UNMODIFIED)
$019F C6 7F XNOTO LDB ##7F NON-NEGATIVE VALUE INDICATOR MEANS X FINITE, NONZERO
$01A1 87 02  STB XVAL SAVE VALUE INDICATOR

; DTSXNF TESTS THE X AND Y OPERANDS FOR INFINITY VALUES
; AND PERFORMS THE APPROPRIATE OPERATIONS WHEN INFINITY
; VALUES ARE ENCOUNTERED.
01B5 08 12 LSL YOP PUT RESULT SIGN INTO Y OPERAND
01B7 07 04 ASR RESLTSN
01B9 06 12 ROR YOP
01BA 00 02 TST XVAL TEST X OPERAND VALUE INDICATOR FOR INFINITY INDICATION
01BB 2C 23 BGE XNOTE IF X IS FINITE THEN GO ON AND TEST Y FOR INFINITY
01BF 5F CLR Flags ELSE PREPARE B FOR CCR MODIFICATION BYTE
01C0 0D 03 TST YVAL SEE IF Y OPERAND IS INFINITY
01C2 2C 10 BGE YFINTE IF Y OPERAND IS FINITE THEN X OPERAND IS RESULT
01C4 96 01 LDA FLAGS ELSE SET MODE/FLAG BYTE
01C6 BA 40 ORA #40 SET INVALID OPERATION STICKY FLAG
01C8 97 01 STA FLAGS SAVE STICKY FLAGS
01CA CA BA ORB #02B SET UP BYTE TO SET V BIT OF CCR
01CC 96 0B LDA XOP+1 MAKE X OPERAND LOOK LIKE A NOT-A-NUMBER
01CE BA 7F ORA #7F
01D0 97 08 STA XOP+1 SAVE MODIFIED BYTE OF X OPERAND
01D2 20 06 BRA XISINF WRAP UP
01DA 00 0A YFINTE TST XOP SET UP CCR MODIFICATION BYTE
01DB 2C 02 BGE XISINF IF X OPERAND IS POSITIVE, DO NOT SET N IN CCR
01DB CA 08 ORB #0B ELSE SET N
01DA 00 00 XISINF STB ORBYTE SAVE CCR MODIFICATION BYTE
01DB BE 000A LDX #XOP POINT X TO Y OPERAND AS RESULT
01DF 7E 0435 JMP FDEIIT WRAP UP AND EXIT

01E2 0D 03 XNOTINF TST YVAL TEST Y OPERAND FOR INFINITY WHEN X IS FINITE
01E4 2C 1A BGE DTSTXO IF Y IS NOT INFINITY, GO TEST X FOR ZERO VALUE
01E6 4F CLR Flags ELSE SET UP RESULT AS ZERO
01E7 5F CLR Flags
01E8 0D 14 STD YOP+2 SAVE LEAST SIGNIFICANT RESULT BYTES
01EA 07 04 ASR RESLTSN INSERT CORRECT SIGN INTO ZERO RESULT
01EC 46 RORA
01ED 56 RORR
01EE 0D 12 STD YOP SAVE MOST SIGNIFICANT RESULT BYTES
01F0 06 04 LDB #0B SET UP BYTE TO SET Z IN CCR
01F2 0D 12 TST YOP SEE IF RESULT IS NEGATIVE
01F4 2C 02 BGE YISINF IF RESULT IS POSITIVE THEN GO ON
01F6 0A 08 ORB #0B ELSE SET N IN CCR MODIFICATION BYTE
01FB 87 00 YISINF STB ORBYTE SAVE CCR MODIFICATION BYTE
01FA 8E 0012 LDY #YOP POINT X TO Y OPERAND AS RESULT
01FD 7E 0435 JMP FDEEXIT WRAP UP AND EXIT

0200 DTSTXO EQU 1

DTSTXO TESTS THE X AND Y OPERANDS FOR ZERO VALUES
AND PERFORMS THE APPROPRIATE OPERATIONS WHEN ZERO VALUES ARE ENCOUNTERED.

0200 0D 02 TST YVAL TEST X OPERAND FOR ZERO VALUE
0202 26 26 BNE XNOTE IF X IS NOT ZERO, GO TEST Y OPERAND FOR ZERO VALUE
0204 0D 03 TST YVAL ELSE TEST Y OPERAND FOR ZERO VALUE
020A 26 12 BNE ASLTO IF Y OPERAND IS NOT ZERO THEN X OPERAND IS RESULT
0208 96 01 LDA FLAGS ELSE SET FLAG/MODE BYTE
0208 BA 40 ORA #40 SET INVALID OPERATION STICKY FLAG
020C 97 01 STA FLAGS SAVE STICKY FLAGS
I AT THIS POINT IN THE PROCEDURE, BOTH THE X OPERAND AND THE V OPERAND ARE FINITE AND NONZERO. DUNPAK UNPACKS THE OPERANDS FOR DIVISION.

UNPKY LOD YOP CHECK IF EXPONENT(Y) = 0
024E  BE UNPKYO GO UNPACK FOR 0 EXPONENT
024F 99 ROLB SHIFT IN LOW ORDER EXPONENT BIT
0250 BA ROLA
0251 4C INCA INCREMENT EXPONENT SINCE HIDDEN BIT IS SHIFTED IN
0252 97 STA YOP FINISH UNPACKING Y, SAVE EXPONENT
0253 96 13 LDA YOP+1 GET MOST SIGNIFICANT FRACTION BYTE
0254 8A 80 ORA ##80 SET HIGH ORDER BIT OF FRACTION SINCE EXPONENT NONZERO
0255 97 13 STA YOP+1
0256 9C 09 BRA UNPKX GO ON AND PROCESS X OPERAND

UNPKYO INCA INCREMENT EXPONENT (BIAS ON 0 EXPONENT IS 126, NOT 127)
025E 74 STA YOP SAVE EXPONENT
0260 08 15 LSL YOP+3 FINISH UNPACKING Y OPERAND
0262 09 14 ROL YOP+2
0264 09 13 ROL YOP+1
CHECK IF EXPONENT(X) = 0
SHIFT IN LOW ORDER BIT OF EXPONENT

INCA  UNPKXO UNPACK FOR 0 EXPONENT
INCREMENT EXPONENT SINCE HIDDEN BIT WAS SHIFTED IN
FINISH UNPACKING X, SAVE EXPONENT
SET MOST SIGNIFICANT FRACTION BYTE
SET HIGH ORDER BIT OF FRACTION SINCE EXPONENT NONZERO
FINISH UNPACKING X OPERAND
INCREMENT EXPONENT SINCE HIDDEN BIT WAS SHIFTED IN
X OPERAND FINISH UNPACKING, SAVE EXPONENT
NORMALIZE X OPERAND IF X OPERAND NORMALIZED THEN NORMALIZE Y OPERAND ELSE SHIFT FRACTION OF X OPERAND DECREMENT EXPONENT TO REFLECT SHIFT
NORMALIZE Y OPERAND IF Y OPERAND NORMALIZED THEN SET UP FOR DIVIDE ELSE SHIFT FRACTION OF Y OPERAND INCREMENT EXPONENT TO REFLECT SHIFT
GO TEST AGAIN
INCREMENT EXPONENT ONCE MORE SINCE QUOTIENT IS FRACTION
SAVE EXPONENT CLEAR OUT OPERAND AREAS CLEAR X OPERAND GUARD BYTES CLEAR Y OPERAND GUARD BYTES CLEAR RESULT AREA SET UP OUTER LOOP COUNTER
STA OUTCTR  SAVE OUTER COUNTER
LDA @B  SET UP INNER LOOP COUNTER
STA INCTR  SAVE INNER COUNTER
LDA @#B0  SET UP MASK FOR QUOTIENT
STA GMASK  SAVE QUOTIENT MASK
LDX @XOP+1 X WILL POINT TO PARTIAL REMAINDER
LDY @YOP+1 Y WILL POINT TO DIVISOR
LDD #0008  U WILL POINT TO QUOTIENT
LOOP LDD ,X  SET UP FOR OPERAND COMPARISON
CMPD ,Y  COMPARE HIGH ORDER BYTES OF OPERANDS
BLO NEIXT IF DIVISOR TOO LARGE THEN GO ON AND SHIFT DIVISOR
BHI SBTRCT ELSE IF PARTIAL REMAINDER IS GREATER SUBTRACT DIVISOR
LDD 2,X ELSE TEST LOW ORDER BYTES OF OPERANDS
CMPD 2,Y
BLO NEIXT IF DIVISOR TOO LARGE THEN GO ON AND SHIFT DIVISOR
BHI SBTRCT ELSE IF PARTIAL REMAINDER IS GREATER SUBTRACT DIVISOR
LDD ,U ELSE CREATE FINAL QUOTIENT BIT (OPERANDS EQUAL) IN RESULT
STB ,U SAVE RESULT BYTE
BRA NORMAL GO ON AND NORMALIZE RESULT
SBTRCT LDD 2,X SUBTRACT DIVISOR FROM PARTIAL REMAINDER
SUBD 2,Y SUBTRACT LOW ORDER BYTES
STD 2,X SAVE LOW ORDER BYTES
LDD ,X SUBTRACT HIGH ORDER BYTES
SBCB 1,Y
SBCA ,Y
STD ,X SAVE HIGH ORDER BYTES
LDB ,U CREATE QUOTIENT BIT IN RESULT
DBR GMASK
STB ,U SAVE RESULT BYTE
SBTRCT NEXT LSR GMASK SHIFT QUOTIENT MASK FOR NEXT ITERATION OF LOOP
LSR ,Y SHIFT DIVISOR
LDR 1,Y
LDR 2,Y
LDR 3,Y
DEC INCTR TEST FOR END OF INNER LOOP
BNE LOOP IF INNER LOOP NOT FINISHED THEN GO BACK
LDA @#B0 ELSE REINITIALIZE QUOTIENT MASK
STA GMASK REINITIALIZE INNER LOOP COUNTER
LAAX 1,X UPDATE PARTIAL REMAINDER POINTER
LEAY 1,Y UPDATE DIVISOR POINTER
LEAU 1,U UPDATE RESULT POINTER
DEC OUTCTR TEST FOR END OF INNER LOOP
BNE LOOP IF DIVISION NOT FINISHED THEN GO BACK

NORMAL EQU $
NORMALIZES THE FINITE RESULT.

LD EXPONT LOAD EXPONENT FOR TESTING
IF RESULT EXPONENT < 1 GO ON AND TAKE CARE OF UNDERFLOW
0323 0D 06  NAMIZ TST RESULT+1 ELSE SEE IF HIGH ORDER BIT OF RESULT FRACTION IS 1  
0325 26 26  BGT DTRND IF HIGH ORDER BIT SET (FRACTION < 0) THEN GO ON AND ROUND  
0327 B3 0001  SUBD $1 ELSE SHIFT FRACTION AND DECREMENT EXPONENT  
032A 08 09  LSL RESULT+4  
032C 09 08  ROL RESULT+3  
032E 09 07  ROL RESULT+2  
0330 09 06  ROL RESULT+1  
0332 20 19  BRA DTRND GO ON AND ROUND  
0334 04 06  NRMLO LSR RESULT+1 SHIFT FRACTION AND INCREMENT EXPONENT  
0336 06 07  ROR RESULT+2  
0338 06 08  ROR RESULT+3  
033A 06 09  ROR RESULT+4  
033C 06 0A  BCC INEXP IF 0 WAS SHIFTED OUT THEN GO ON  
033E 0E 03  EXB D,U ELSE TEMPORARILY SAVE EXPONENT  
0340 06 09  LDB RESULT+4 SET STICKY BIT IN FRACTION  
0342 CA 40  ORR $$40  
0344 07 09  STB RESULT+4  
0346 0E 30  EXB U,D RESTORE EXPONENT  
0348 C3 0001  INEXP ADDP $1 INCREMENT EXPONENT  
034A 2F 07  BLE NAMLO KEEP SHIFTING UNTIL EXPONENT = 1

0340 DTRND EQU  

$ DTRND DETERMINES WHETHER OR NOT TO PERFORM ROUNDING DEPENDING $ ON THE Rounding MODE SELECTED.

0340 8E 0080  LDI $$80 SET UP INITIAL ROUNDING FACTOR 
0350 9F 1A  STX RFACTR  
0352 1F 03  TFR D,U SAVE EXPONENT TEMPORARILY IN U  
0354 18B 0001  CMPP $1  
0356 26 09  BNE TSGARD IF NOT MINIMUM EXPONENT, GO ON  
0358 0B 18  LSL RFACTR+1 SHIFT ROUNDING FACTOR  
035A 09 1A  ROL RFACTR  
035E D6 08  LDB RESULT+3 TEST FOR ZERO GUARD BIT WITH MINIMUM EXPONENT  
0360 56  RORB  
0361 25 04  BCS INACT IF GUARD BIT 1 THEN GO ON  
0363 0D 09  TSGARD TST RESULT+4 ELSE TEST GUARD BYTE  
0365 27 5A  BEQ DREPAK IF ZERO GUARD BYTE THEN NO NEED TO ROUND  
0367 D6 01  INACT LDB FLAGS SET INEXACT RESULT FLAG  
0369 CA 04  ORR $604  
036B D7 01  STB FLAGS SAVE STICKY FLAGS  
036D 56  RORB CHECK ROUNDING MODE IN USE  
036F 24 0D  BCC RPRAN IF BIT 0 OF FLAGS = 0 THEN Rounding MODE IS RP OR RN  
0371 56  RORB ELSE CHECK FOR RZ OR RN MODE  
0373 24 4E  BCC DREPAK NO NEED TO ROUND IF RZ MODE  
0375 0D 04  TST RSLTH OTHERWISE TEST SIGN OF RESULT (RN MODE)  
0377 27 4A  BEQ DREPAK IF RESULT IS POSITIVE IN RN MODE THEN NO NEED TO ROUND  
0379 0B 1B  RMNINF LSL RFACTR+1 ELSE CHANGE ROUNDING FACTOR  
037B 09 1A  ROL RFACTR  
037D 20 2C  BRA ROUND GO ON AND ROUND
CHECK FOR RP OR RN MODE
IF BIT 1 OF FLAGS = 0 THEN GO PREPARE FOR RN Rounding
OTHERWISE TEST SIGN OF RESULT (RP MODE)
IF RESULT IS NEGATIVE IN RP MODE THEN NO NEED TO ROUND
ELSE GO BACK AND CHANGE ROUNDING FACTOR
DETERMINE WHICH BIT TO ROUND AT BY CHECKING EXPONENT
ROUND FOR MINIMUM EXPONENT
CHECK GUARD BYTE FOR TIE BETWEEN TWO FRACTIONS
TEST LEAST SIGNIFICANT PRESERVED BIT IS 0
ELSE LEAVE ALONE
ADD ROUNDING FACTOR TO RESULT
FINISH ROUNDING
IF NO CARRY OUT THEN GO ON AND REPACK RESULT
ELSE SHIFT IN CARRY
INCREMENT EXPONENT TO REFLECT SHIFT
DREPAK REPACKS THE RESULT AND CHECKS FOR UNDERFLOW
AND OVERFLOW OF THE RESULT.
RESULT+1 FINISH SHIFTING
SET UP CCR MODIFICATION BYTE
RESULT IF RESULT POSITIVE DO NOT SET N IN CCR
ELSE SET N
SET UP CCR MODIFICATION BYTE
SAVE CCR MODIFICATION BYTE
POINT X TO RESULT
WRAP UP AND EXIT
SAVE EXPONENT FOR REPACKING
SHIFT IN SIGN
SHIFT SIGN IN, SHIFTING EXPONENT
SHIFT FRACTION
SET UP CCR MODIFICATION BYTE
SET UP CCR MODIFICATION BYTE
IF RESULT POSITIVE DO NOT SET N IN CCR
ELSE SET N
SAVE CCR MODIFICATION BYTE
SAVE EXPONENT FOR REPACKING
SHIFT IN SIGN
SHIFT SIGN IN, SHIFTING EXPONENT
SHIFT FRACTION
SET UP CCR MODIFICATION BYTE
SET UP CCR MODIFICATION BYTE
IF RESULT POSITIVE DO NOT SET N IN CCR
ELSE SET N
TEST FOR ZERO RESULT
IF NO UNDERFLOW THEN DO NOT SET V AND Z IN CCR
ELSE SET V AND Z ALSO
SAVE CCR MODIFICATION BYTE
SET UNDERFLOW STICKY FLAG
SAVE STICKY FLAGS
POINT X TO RESULT
WRAP UP AND EXIT
SET UNDERFLOW STICKY FLAG
SAVE STICKY FLAGS
POINT X TO RESULT
SET OVERFLOW STICKY FLAG
NOT SET N IN CCR
SAVE STICKY FLAGS
POINT X TO RESULT
SET UP RESULT AS INFINITY
CREATE MAXIMUM EXPONENT IN RESULT
SHIFT IN SIGN
SHIFT SIGN IN, SHIFTING EXPONENT
FINISH SHIFTING
SET UP CCR MODIFICATION BYTE TO SET V IN CCR
NOT SET N IN CCR
SAVE STICKY FLAGS
POINT X TO RESULT
SET OVERFLOW STICKY FLAG
NOT SET N IN CCR
SAVE STICKY FLAGS
POINT X TO RESULT
FOEXIT EQU 1
FEXIT RESTORES THE REGISTERS, LOADS THE STICKY FLAGS AND SETS UP A NEW CCR. X AT THIS POINT IS ALREADY POINTING TO THE RESULT.

0435 35 01 PULS CC RESTORE CCR (FROM TOP OF STACK)
0437 1E A8 EXG CC,A PERFORM CCR MODIFICATION IN A
0439 84 F1 ANDA $F1 CLEAR N,Z,V IN CCR
043B 9A 00 ORA ORBYTE SET N,Z,V IN CCR ACCORDING TO MODIFICATION BYTE
043D 1E 0A EXG A,CC RESTORE CCR
043F 96 01 LDA FLAGS LOAD STICKY FLAGS/MODE BYTE INTO A
0441 35 6C PULS U,Y,DP,B RESTORE OTHER REGISTERS
0443 39 RTS RETURN FROM ROUTINE

END

0 ERROR(S) DETECTED
BIBLIOGRAPHY

Books


Articles


Fraley, Bob and Walther, Steve, "Proposal to Eliminate Denormalized Numbers," ACM SIGNUM Newsletter, Special Issue (October, 1979), 22-23.


Unpublished Materials


Other Materials