Floating-Point Arithmetic with the TMS32020

APPLICATION REPORT: SPRA011

Author: Charles Crowell
Digital Signal Processor – Semiconductor Group

Digital Signal Processing Solutions
1989

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain application using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

TRADEMARKS

CONTACT INFORMATION

US TMS320 HOTLINE	(281) 274-2320
US TMS320 FAX	(281) 274-2324
US TMS320 BBS	(281) 274-2323
US TMS320 email	dsph@ti.com

Floating-Point Arithmetic with the TMS32020

Abstract

This report presents algorithm and code implementing floating-point addition, subtraction, multiplication, and division with the TMS320. The support of floating-point operations by the TI processors has made possible some applications, such as implementation of the CCITT Adaptive Differential Pulse Code Modulation (ADPCM) algorithm and image/graphics operations.

Product Support on the World Wide Web

Our World Wide Web site at www.ti.com contains the most up to date product information, revisions, and additions. Users registering with TI&ME can build custom information pages and receive new product updates automatically via email.

INTRODUCTION

The TMS32020 Digital Signal Processor is a fixed-point 16/32-bit microprocessor. However, it can also perform floating-point computations at a speed comparable to some dedicated floating-point processors.

The purpose of this application report is to analyze an implementation of floating-point addition, multiplication, and division on the TMS32020. The floating-point single-precision standard proposed by the IEEE will be examined. Using this standard, the TMS32020 performs a floating-point multiplication in 7.8 microseconds, a floating-point addition in 15.4 microseconds, and a floating-point division in 22.8 microseconds.

To illustrate floating-point formats and the tradeoffs involved in making a choice between different floating-point formats, a review of floating-point arithmetic notation and of addition, multiplication, and division algorithms is first presented.

FLOATING-POINT NOTATION

The floating-point number f may be written in floating-point format as

$$f = m \times b^{e}$$

where

$$\begin{aligned} m &= \text{mantissa} \\ b &= \text{base} \\ e &= \text{exponent} \end{aligned}$$

For example, 6,789,320 may be written as

$$0.6789320 \times 10^{7}$$

In this case,

$$\begin{aligned} m &= 0.6789320 \\ b &= 10 \\ e &= 7 \end{aligned}$$

The two floating-point numbers f_1 and f_2 may be written as

$$\begin{aligned} f_1 &= m_1 \times b^{e1} \\ f_2 &= m_2 \times b^{e2} \end{aligned}$$

Floating-point addition/subtraction, multiplication, and division for f_1 and f_2 are defined as follows:

$$f_1 \pm f_2 = (m_1 \pm m_2 \times b^{-(e_1 - e_2)}) \times b^{e1} \text{ if } e_1 \geq e_2 \quad (1)$$

or

$$= (m_1 \times b^{-(e_2 - e_1)}) \pm m_2) \times b^{e2} \text{ if } e_1 < e_2$$

$$f_1 \times f_2 = m_1 \times m_2 \times b^{(e_1 + e_2)} \quad (2)$$

$$f_1/f_2 = (m_1/m_2) \times b^{(e_1 - e_2)} \quad (3)$$

A cursory examination of these expressions reveals some of the factors involved in the implementation of floating-point arithmetic. For addition, it is necessary to shift the mantissa of the floating-point number which has the smaller exponent to the right by the difference in the magnitude of the two exponents. This is shown in the multiplication by the terms

$$b^{-(e_1 - e_2)} \text{ and } b^{-(e_2 - e_1)}$$

This right shift can result in mantissa underflow. There are also possibilities for mantissa overflow. Addition and subtraction of exponents can lead to exponent underflow and overflow. To alleviate underflow and overflow, it is necessary to decide on some scheme for roundoff. For a detailed description and analysis of underflow and overflow conditions and rounding schemes, see reference 1.

It is desirable to have all numbers normalized, i.e., the mantissas of f_1 and f_2 have the most significant digit in the leftmost position. This provides the representation with the greatest accuracy possible for a fixed mantissa length. The result of any floating-point operation must also be normalized. The factors associated with normalization, overflow, and other characteristics of floating-point implementations are best illustrated with a few examples.

Consider the addition of two binary floating-point numbers f_1 and f_2 where

$$\begin{aligned} f_1 &= 0.10100 \times 2^{011} \\ f_2 &= 0.11100 \times 2^{001} \end{aligned}$$

Both of these numbers are normalized, i.e., the first bit after the binary point is a 1. Addition requires equal exponents, so the fractions are aligned by shifting right the one with the smaller exponent and adjusting the smaller exponent. This yields

$$f_2 = 0.00111 \times 2^{011}$$

Then,

$$\begin{aligned} f_1 + f_2 &= 0.10100 \times 2^{011} + 0.00111 \times 2^{011} \\ &= 0.11011 \times 2^{011} = f_3 \end{aligned}$$

The sum may overflow the left end by one digit, thus requiring a postaddition adjustment or renormalization step. Since it is assumed that the register is only of a finite length, this renormalization will result in the loss of the lowest order bit.

Another example illustrates the overflow past the most significant bit. With an assumed register length of five, let

$$f_1 = 0.11100 \times 2^{011}$$
$$f_2 = 0.10101 \times 2^{001}$$

Then,

$$\begin{array}{rl} & 0.11100 \quad \times 2^{011} = f_1 \\ + & \underline{0.0010101 \times 2^{011} = f_2} \\ & \underline{1}.00001\underline{01} \times 2^{011} = f_3 \end{array}$$

The significance of the two digits underlined in the right part of the mantissa is suspect, since it is assumed that the corresponding bits of f_1 are zero. The left underlined digit is the overflow past the most significant bit. To finish the addition, f_3 is shifted to the right and the exponent adjusted accordingly. Thus,

$$1.0000101 \times 2^{011} = f_3$$

The shift of the fraction and the adjustment of the exponent yield

$$0.10000101 \times 2^{100} = f_3$$

The result may be rounded, giving

$$0.10001 \times 2^{100} = f_3$$

or truncated, giving

$$0.10000 \times 2^{100} = f_3$$

FLOATING-POINT ALGORITHMS

Multiplication Algorithm

The algorithm for normalized floating-point multiplication is illustrated in Figure 1. This algorithm is an implementation of Equation 2 in the section on floating-point notation. The floating-point numbers being multiplied are A and B written as

$$A = m_A \times b^{e_A} \text{ and } B = m_B \times b^{e_B}$$

The result is

$$C = m_C \times b^{e_C}$$

For the resulting m_C, there are three special cases. The m_C may be zero, in which case there is a branch to Step 10 to set $C = 0$. If $m_C \neq 0$, then the most significant bit will

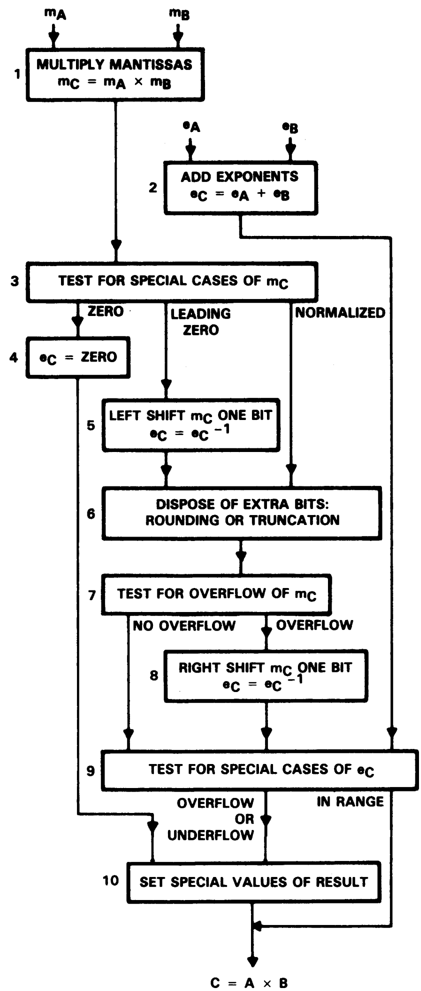

Figure 1. Floating-Point Multiplication

be in either the first or second leftmost bit. If the most significant bit is in the second leftmost bit, then a left shift of m_C is necessary (see Step 5). Otherwise, C is already in normalized form, and there is a branch to Step 6.

In Step 6, the desired rounding scheme is implemented. After this rounding, it is possible that m_C will overflow (see Step 7). In this case, it is necessary to right-shift m_C one bit (see Step 8). Special cases of e_C, are tested for in Step 9.

If there is an overflow or underflow of e_C, it is corrected in Step 10. Otherwise, the result is in range, and the calculation is complete.

Addition Algorithm

The implementation of normalized floating-point addition is more involved than for multiplication. This addition algorithm, outlined in Figure 2, is an implementation of Equation 1 in the section on floating-point notation.

In Step 1, e_A and e_B are compared to determine e_C. For this illustration of the algorithm, it is assumed that $e_A \leq e_B$. The right shift (d) required to align m_A is determined in Step 2. The procedure in Step 3 implements the right shift of m_A. In Step 4, the extra bits of m_A are discarded by using the desired rounding technique. The mantissas of A and B are then added in Step 5.

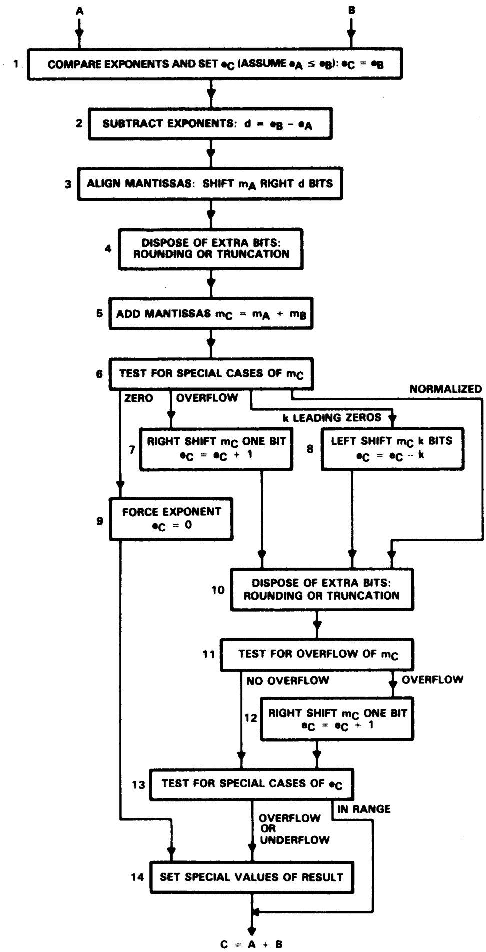

Figure 2. Floating-Point Addition

Now, the procedure becomes somewhat more involved. The m_C may be zero, in which case there is a branch to Step 9 which sets $e_C = 0$; a branch to Step 14 sets the special value of the result. The m_C may overflow, making a right shift of one necessary (see Step 7). The m_C may have k leading zeroes; therefore, a left shift of k is required. This normalization step is generally the most involved and time-consuming step to perform. The procedures in Steps 10, 11, and 12 round the m_C, test for a possible overflow due to the rounding, and adjust e_C accordingly. The special case of e_C is determined in Step 13. Finally, after Step 14, the sum C = A + B is formed.

Division Agorithm

Floating-point division is more sophisticated than multiplication and addition since fixed-point processors such as the TMS32020 are not inherently capable of performing division. For example, 1/3 = 0.3333...; only an approximation can be calculated since 1/3 must be represented in a finite number of terms. Several algorithms can be implemented to find good approximations of such numbers. The algorithm implemented in this report is shown in Figure 3.

Step 1 shows the equivalent of A/B. In Step 2, the latter term is expanded using a power series of 1/(1 + X), where ϵ (BLO/BHI) is X (ϵ simply denotes that the term is right-shifted 16 bits forming the least significant bits of a 32-bit number). The third term in the power series only affects the LSB of a 32-bit result; therefore, this term and all the following terms can be dropped, as shown in Step 3.

The equation in Step 3 can be implemented on the TMS32020 in two steps. Assuming that the result is a 32-bit number Q and that it is composed of a 16-bit QHI and a 16-bit QLO, think of the equation in Step 3 in the following manner: A/B = Q − ϵX. The first term is a fair approximation of the result Q, and the second term is a correction term to obtain a better approximation. With this in mind, it can be shown that (AHI + ϵALO)/BHI will give a 16-bit quotient and a 16-bit remainder. Due to the architecture of the TMS32020, the 16-bit quotient will be in the low word of the accumulator and the remainder will be in the high word of the accumulator after the division. Since it is desirable

A divided by B

where
$$A = AHI + \epsilon ALO$$
$$B = BHI + \epsilon BLO$$
$$\epsilon = \frac{1}{2^{WORDSIZE}} = \frac{1}{2^{16}}$$

STEP 1:
$$\frac{AHI + \epsilon ALO}{BHI + \epsilon BLO} = \frac{AHI + \epsilon ALO}{BHI}\left(\frac{1}{1 + \epsilon\left(\frac{BLO}{BHI}\right)}\right)$$

STEP 2:
$$= \frac{AHI + \epsilon ALO}{BHI}\left(1 - \epsilon\left(\frac{BLO}{BHI}\right) + \epsilon^2\left(\frac{BLO}{BHI}\right)^2 \ldots\right)$$

STEP 3:
$$\approx \frac{AHI + \epsilon ALO}{BHI} - \epsilon\left(\frac{BLO}{BHI}\right)\left(\frac{AHI + \epsilon ALO}{BHI}\right)$$

Figure 3. Division Equation

to have a floating-point result, the remainder must be divided by BHI to obtain the low word of the quotient. Now QHI and QLO have been calculated. When placing Q into the correction term (equation in Step 3), note that Q is equal to QHI + QLO. It can be shown that QLO will have no effect on the result since the correction term is multiplied by ϵ. Therefore, to calculate A divided by B, simply implement the following equation:

$$\frac{A}{B} = \frac{A}{BHI} - \epsilon\left(\frac{BLO}{BHI} \times QHI\right)$$

where the division is fixed binary (left-shifts and subtracts).

Figure 4 shows the implementation of the division algorithm that was outlined in Figure 3.

In Step 1, the dividend is right-shifted four times to prevent an overflow. Note that the result is not shifted left to compensate for this shift, because the normalization routine automatically does this. The shift causes the dividend to be limited to 27 significant bits instead of 31. In Step 2, a binary divide (left-shifts and subtracts) is implemented on the dividend by the high 16 bits of the divisor. The 32-bit result contains a quotient in the low 16 bits of the accumulator, and a remainder (R1) in the high 16 bits of the accumulator. R1 is left-shifted fifteen places in Step 3. The new R1 is divided by BHI in Step 4 to calculate the lower 16 bits of the quotient.

The quotient has now been approximated. The 32-bit result is composed of QHI and QLO, as shown in Figure 3. To obtain a better approximation, one term in the power series expansion must be added to the quotient. Therefore, the procedure in Step 5 calculates a 16-bit correction term, which is then added (or subtracted since it is the term following the "1" in the power series) to the 32-bit quotient.

Testing for an overflow of the resulting mantissa is necessary. Since the dividend was left-shifted four places, the resulting quotient will not be negative if an overflow occurred. To detect an overflow, bit 28 in the quotient must be tested. If this bit is a 1, an overflow occurred; if it is a 0, no overflow occurred. If an overflow has occurred, the exponent must be incremented. Finally, it is necessary to normalize the quotient and output the results.

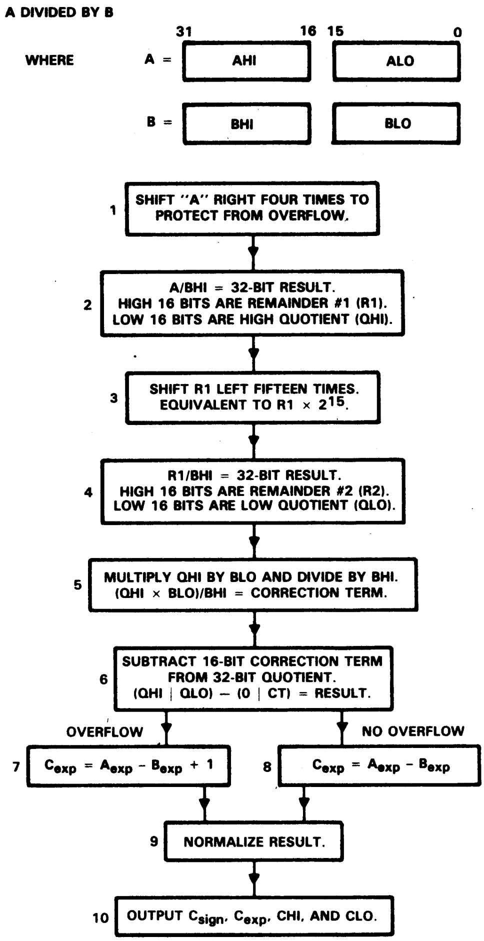

Figure 4. Floating-Point Division

IEEE FLOATING-POINT SINGLE-PRECISION FORMAT

Of interest is a set of formats known as the IEEE standard. This IEEE recommended format consists of a variety of precision formats (single, double, single-extended, and double-extended). The IEEE has also proposed several techniques for handling special cases such as overflow, underflow, $\pm\infty$, and rounding. For complete details, the reader is referred to the proposed IEEE standard.[2]

The single-precision format is a 32-bit format consisting of a 1-bit sign field s, an 8-bit biased exponent e, and a 23-bit fraction f (see Figure 5). The value of a binary floating-point number X is determined as follows:

$$X = (-1)^s \times 2^{(e-127)} \times 1.f$$

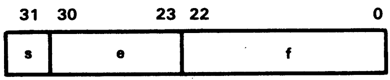

Figure 5. IEEE Floating-Point Single-Precision Format

The advantage of this format is that it is structured in such a way as to provide easy storage and straightforward input/output operations on 8-, 16- and 32-bit processors. The disadvantage with this format is that the large mantissa will generally span several words of memory.

FLOATING-POINT IMPLEMENTATION

IEEE Implementation

The IEEE single-precision format is described here as it applies to the addition, multiplication, and division algorithms. In these floating-point routines written for the TMS32020, all results are truncated to 31 bits to provide more flexibility in the user's development of a rounding scheme suitable for his application. The representations of $\pm\infty$ are ignored so that the user can decide how to handle these exceptions in a manner that is appropriate for his particular application.

I/O Considerations

The first consideration is the internal representation of the binary floating-point number. If the number is read into the TMS32020 as two 16-bit words, some processing is then necessary to put the floating-point number into a representation which is easier to process. The representation used in the TMS32020 programs in the appendices is shown in Figure 6. This internal representation may be arrived at by a simple manipulation of the IEEE bit fields. For this particular algorithm, it is assumed that the floating-point number is input to the TMS32020 as the four 16-bit fields shown in Figure 6. However, the user can easily supply his own routine to arrive at this format from two 16-bit inputs to the TMS32020 where the inputs contain the IEEE single-precision format.

The format in Figure 6 was chosen to minimize the execution time of the floating-point addition, multiplication, and division routines. The format of the result is shown in Figure 7. Notice that it is identical to the format in Figure 5 except for CLO. CLO has its 16 most significant bits valid for both the addition, multiplication, and division routines.

Normalization

Since the floating-point routines require normalization, a partial binary search algorithm is implemented in the addition and division routines in the appendices. To begin the normalization routine, note that all mantissas can be considered to be positive with the format used for the result shown in Figure 7. The binary search for the most significant bit (the leftmost 1 since the mantissa is positive) is illustrated in Figure 8.

The first move is to split the result into CHI and CLO. If CHI $\neq$ 0, the most significant bit (MSB) is the CHI; otherwise, it is the CLO. For this example, it is in CLO.

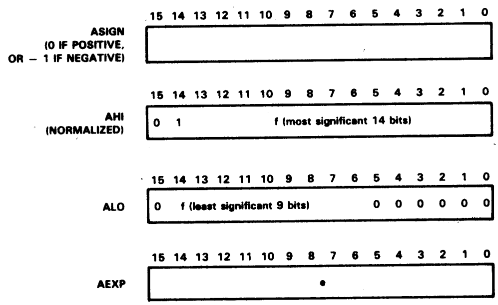

Figure 6. Floating-Point Representation

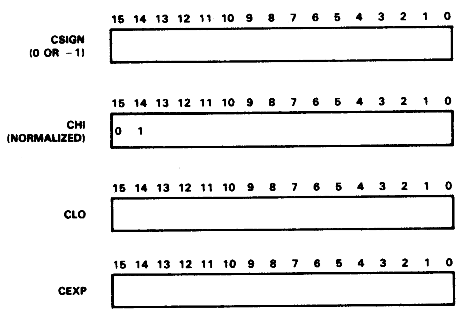

Figure 7. Result Representation

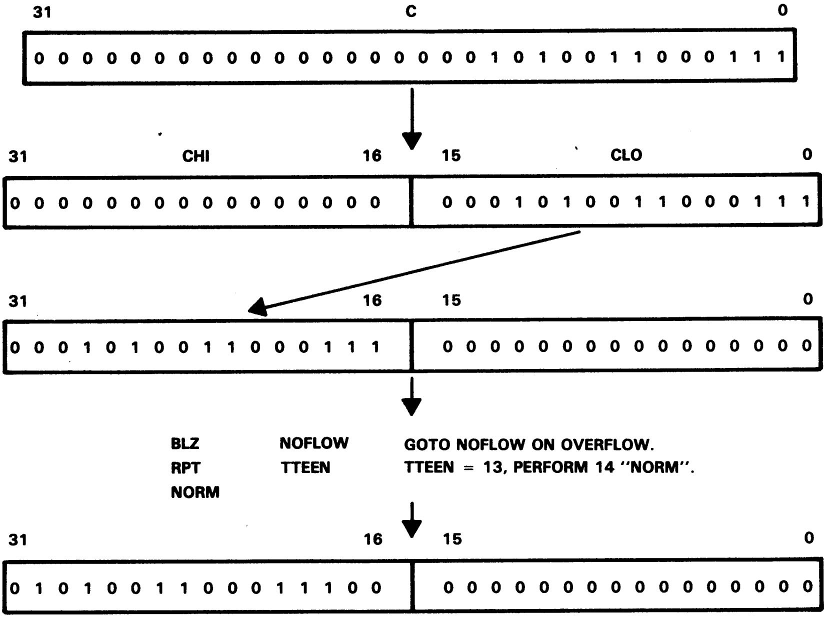

Figure 8. Partial Binary Search

The next step is to form a 32-bit result with CLO in the most significant word position. It is now possible for the MSB to be in the highest bit location since CLO has been left-shifted 16 times. If this is the case, an overflow has occurred, and the result must be right-shifted once. The normalization routine tests this by branching to NOFLOW if the result is negative. If the number is not negative, the normalization can continue.

The NORM instruction is used in the repeat mode to complete the normalization. Note that this whole normalization routine can be replaced by the following two instructions: RPTK 29 and NORM. The RPTK instruction causes the NORM instruction to be repeated 30 times, thus normalizing a 32-bit number. This method is not implemented here due to the timing. These two instructions always take 31 cycles to normalize a 32-bit number. The normalization routine here takes only 22 cycles (worst case) for normalizing a 32-bit number. Therefore, if program space is more important than timing efficiency, it is best to replace the normalization routine with these two instructions.

Added Precision

As illustrated in Figure 7, the 16 most significant bits of CLO are valid, i.e., C is valid for 31 places beyond the binary point. Oftentimes the user is not as concerned with the IEEE standard as in being certain that he has enough accuracy for his particular application. Since the TMS32020 uses 16-bit words, the routines in the appendices implicitly maintain a 30-bit mantissa. They also implicitly use a 16-bit exponent. If the user desires this added accuracy and dynamic range, then it is readily implementable with no additional cost in execution time. The normalization for the addition, as mentioned previously, operates over the entire 32-bit accumulator. For the strict IEEE format, the user will only want to normalize over the 25 most significant bits of the accumulator. The structure of the normalization routine makes this modification simple.

The routines in the appendices make no provision for the representation of $\pm\infty$ and exponent underflow and overflow. The user of the routines should consider the degree of significance of these results and the way they should be handled for his particular application. Since these routines are written to operate at maximum speed, truncation of results is used. If the user desires to implement a rounding scheme, then he will also need to check for the possibility of overflow due to the rounding scheme. This step is shown in the multiplication, addition, and division flowcharts (see Figures 1, 2, and 3).

SUMMARY

The TMS32020 may be used to perform floating-point operations with great accuracy, wide dynamic range, and high-speed execution. The design engineer has the responsibility of deciding what type of floating-point format is best for his application. To aid in understanding floating-point operations, several examples have been given that illustrate the manipulations necessary to implement floating-point addition, multiplication, and division algorithms. Flowcharts for these algorithms are also included. The appendices contain the TMS32020 code for the IEEE floating-point single-precision format used in addition, multiplication, and division. The addition and multiplication routines may also be used without modification to implement a format with up to a 30-bit mantissa and a 16-bit exponent without any increase in execution time.

ACKNOWLEDGEMENTS

Major portions of this application report were taken from "Floating-Point Arithmetic with the TMS32010," an application report written by Ray Simar, Jr. The author would also like to thank Gwyn Guidy for her assistance with the floating-point division algorithm.

REFERENCES

1. D.J. Kuck, *The Structure of Computers and Computations*, **Vol 1**, John Wiley & Sons (1978).
2. J. Coonen et al, "A Proposed Standard for Binary Floating-Point Arithmetic," *ACM Signum Newsletter*, 4-12 (October 1979).
3. Donald E. Knuth, *Seminumerical Algorithms*, **Vol 2**, 2nd Edition, Addison-Wesley (1981).

APPENDIX A


```
             ********************************************************
             *                                                      *
             *    THIS IS A FLOATING-POINT ADDITION ROUTINE WHICH   *
             *    IMPLEMENTS THE IEEE PROPOSED FLOATING-POINT       *
             *    FORMAT ON THE TMS32020.                           *
             *                                                      *
             ********************************************************
             *
             *    INITIAL FORMAT (ALL 16 BIT WORDS)
             *    ------------------
             *    |   ALL 0 OR 1   |       ASIGN (0 OR -1)
             *    ------------------
             *
             *    ------------------
             *    |0|.   15 BITS   |        AHI (NORMALIZED)
             *    ------------------
             *
             *    ------------------
             *    |0|  9 BITS |--0-|        ALO
             *    ------------------
             *
             *    ------------------
             *    |                |       AEXP (-127 TO 128)
             *    ------------------
             *
             *    TO CORRESPOND WITH IEEE FORMAT,
             *    INPUT 0.1F * 2 ** (E + 1)
             *    INSTEAD OF 1.F * 2 **E, AND SUBTRACT 127 FROM E.
             *
             *    THE FINAL FORMAT IS THE SAME AS THE INITIAL FORMAT
             *    EXCEPT THAT FOR CLO WE HAVE:
             *
             *    ------------------
             *    |   16 BITS      |       CLO
             *    ------------------
             *
             *    ALL 16 BITS OF CLO ARE VALID. ANYTHING PAST THESE HAS
             *    BEEN TRUNCATED.
             *
             ********************************************************
             *                                                      *
             *    WORST CASE (EXCLUDING INITIALIZATION AND I/O):    *
             *    15.4 MICROSECONDS.                                *
             *    THIS TIMING INCLUDES THE NORMALIZATION.           *
             *    WORDS OF PROGRAM MEMORY: 217                      *
             *                                                      *
             ********************************************************
             *
                        IDT     ' FLTADD'
0000                    AORG
          0000   ASIGN  EQU     0
          0001   AEXP   EQU     1
          0002   AHI    EQU     2
          0003   ALO    EQU     3
          0004   BSIGN  EQU     4
          0005   BEXP   EQU     5
          0006   BHI    EQU     6
```

```
          0007  BLO    EQU    7
          0008  CSIGN  EQU    8
          0009  CEXP   EQU    9
          000A  CHI    EQU    10
          000B  CLO    EQU    11
          000C  D      EQU    12
          000D  ONE    EQU    13
          000E  TEMP   EQU    14
          000F  THREE  EQU    15
          0010  SIXT   EQU    16
          0011  RESID  EQU    17
          0012  TTEEN  EQU    18
                *
                *      INITIALIZATION
                *
     0000 C804         LDPK   4              BEGIN ON PAGE 4.
     0001 CE07         SSXM                  SET SIGN EXTENSION.
     0002 5589         LARP   1
     0003 D100         LRLK   AR1,>200
     0004 0200
     0005 CB07         RPTK   7
     0006 80A0         IN     *+,PA0
     0007 5588         LARP   0
     0008 C000         LARK   AR0,0          CLEAR EXPONENT REGISTER.
     0009 CA01         LACK   1
     000A 600D         SACL   ONE            ONE = 1
     000B CA10         LACK   16
     000C 6010         SACL   SIXT
     000D CA03         LACK   3
     000E 600F         SACL   THREE
     000F CA0D         LACK   13
     0010 6012         SACL   TTEEN
                *
                *      BEGIN FLOATING POINT ADD
                *
     0011 2001  UP     LAC    AEXP           FIND LARGEST NUMBER.
     0012 1005         SUB    BEXP
     0013 F680         BZ     AEQB           IF EXP ARE THE SAME, JUMP TO AEQB.
     0014 0043
     0015 F380         BLZ    ALTB           IF A IS LESS THAN B, JUMP TO ALTB.
     0016 004D
                *
     0017 CE23  AGTB   NEG
     0018 0010         ADD    SIXT           D = (16-D)
     0019 F380         BLZ    A1             JUNP IF EXP DIFFERENCE IS > 16
     001A 0028
                *
                *    EXPONENT DIFFERENCE < 16
                *
     001B 600C         SACL   D
     001C 3C0C         LT     D
     001D 4206         LACT   BHI            BHI IS SHIFTED RIGHT "D" TIMES.
     001E 6806         SACH   BHI
     001F 6011         SACL   RESID          RESIDUAL BITS MUST BE MAINTAINED.
     0020 4207         LACT   BLO            BLO IS SHIFTED RIGHT "D" TIMES.
     0021 CE18         SFL                   MSB (THE 0) IS SHIFTED AWAY.
     0022 6807         SACH   BLO
```

```
0111 0023 2007          LAC     BLO
0112 0024 4D11          OR      RESID       GET BITS THAT WERE SHIFTED FROM BHI.
0113 0025 6007          SACL    BLO
0114 0026 FF80          B       A2
     0027 0031
0115               *
0116               *    EXPONENT DIFFERENCE >16
0117               *
0118 0028 0010     A1   ADD     SIXT
0119 0029 F380          BLZ     A3            JUMP IF EXPONENT DIFF > 32
     002A 0039
0120 002B 600C          SACL    D
0121 002C 3C0C          LT      D
0122 002D 4206          LACT    BHI
0123 002E 6807          SACH    BLO
0124 002F CA00          ZAC
0125 0030 6006          SACL    BHI
0126 0031 2000     A2   LAC     ASIGN         A IS LARGER THAN B.
0127 0032 6008          SACL    CSIGN         THEREFORE, CSIGN = ASIGN.
0128 0033 2001          LAC     AEXP          ALIGN THE B MANTISSA.
0129 0034 6009          SACL    CEXP
0130 0035 2103          LAC     ALO,1         GET RID OF EXTRA BIT.
0131 0036 6003          SACL    ALO
0132 0037 FF80          B       CHKSGN        DO BOTH NUMBERS HAVE THE SAME SIGN?
     0038 0078
0133               *
0134               *    A >> B ,  RESULT = A
0135               *
0136 0039 2002     A3   LAC     AHI
0137 003A 600A          SACL    CHI
0138 003B 2103          LAC     ALO,1
0139 003C 600B          SACL    CLO           A IS LARGER THAN B
0140 003D 2000          LAC     ASIGN         THEREFORE CSIGN = ASIGN
0141 003E 6008          SACL    CSIGN
0142 003F 2001          LAC     AEXP
0143 0040 6009          SACL    CEXP
0144 0041 FF80          B       AROUND
     0042 00D6
0145               *
0146 0043 2000     AEQB LAC     ASIGN       IF SIGNS ARE THE SAME, CSIGN = ASIGN
0147 0044 6008          SACL    CSIGN
0148 0045 2103          LAC     ALO,1         ALIGN MANTISSAS.
0149 0046 6003          SACL    ALO
0150 0047 2107          LAC     BLO,1
0151 0048 6007          SACL    BLO
0152 0049 2001          LAC     AEXP          SET C EXPONENT =  A EXPONENT.
0153 004A 6009          SACL    CEXP
0154 004B FF80          B       CHKSGN        DO BOTH NUMBERS HAVE THE SAME SIGN?
     004C 0078
0155               *
0156 004D 0010     ALTB ADD     SIXT          D = (16-D)
0157 004E F380          BLZ     B1            JUMP IF EXP DIFF > 16
     004F 005D
0158 0050 600C          SACL    D
0159 0051 3C0C          LT      D
0160 0052 4202          LACT    AHI           AHI GETS SHIFTED "D" TIMES.
0161 0053 6802          SACH    AHI
```

```
0054 6011          SACL     RESID        MAINTAIN EXTRA BITS.
0055 4203          LACT     ALO          ALO GETS SHIFTED "D" TIMES.
0056 CE18          SFL                   MSB (THE 0) IS SHIFTED AWAY.
0057 6803          SACH     ALO
0058 2003          LAC      ALO
0059 4D11          OR       RESID        GET RESIDUAL BITS.
005A 6003          SACL     ALO
005B FF80          B        B2
005C 0066
              *
              *   EXPONENENT DIFFERENCE > 16
              *
005D 0010     B1   ADD      SIXT
005E F380          BLZ      B3           JUMP IF EXP DIFF > 32
005F 006E
0060 600C          SACL     D
0061 3C0C          LT       D
0062 4202          LACT     AHI
0063 6803          SACH     ALO
0064 CA00          ZAC
0065 6002          SACL     AHI
0066 2004     B2   LAC      BSIGN        B IS THE BIGGEST NUMBER.
0067 6008          SACL     CSIGN        THEREFORE, LET THE SIGN OF C=BSIGN.
0068 2005          LAC      BEXP         SET C EXPONENT = B EXPONENT.
0069 6009          SACL     CEXP
006A 2107          LAC      BLO,1        GET RID OF EXTRA BIT.
006B 6007          SACL     BLO
006C FF80          B        CHKSGN       DO BOTH NUMBERS HAVE THE SAME SIGN?
006D 0078
              *
              *        B >> A ,     RESULT = B
              *
006E 2006     B3   LAC      BHI
006F 600A          SACL     CHI
0070 2107          LAC      BLO,1
0071 600B          SACL     CLO          B IS THE BIGGEST NUMBER
0072 2004          LAC      BSIGN        THEREFORE, LET THE SIGN OF C=BSIGN
0073 6008          SACL     CSIGN
0074 2005          LAC      BEXP         SET C EXPONENT = B EXPONENT
0075 6009          SACL     CEXP
0076 FF80          B        AROUND
0077 00D6
              *
0078 2000     CHKSGN LAC    ASIGN        CHECK THE SIGNS.
0079 1004          SUB      BSIGN
007A F680          BZ       ADNOW        IF THEY ARE THE SAME, JUST ADD.
007B 00A9
007C F380          BLZ      AISNEG
007D 008C
007E 4002     BISNEG ZALH   AHI          DO (|A| - |B|),
007F 4903          ADDS     ALO          SINCE B < 0  AND  A > 0.
0080 4507          SUBS     BLO
0081 4406          SUBH     BHI
0082 F680          BZ       CZERO
0083 009A
0084 F380          BLZ      CNEG
0085 00A1
```

```
0086 680A          SACH   CHI
0087 600B          SACL   CLO
0088 CA00          ZAC
0089 6008          SACL   CSIGN
008A FF80          B      NORMAL     GO AND NORMALIZE RESULT.
008B 00B3
008C 4006  AISNEG  ZALH   BHI        DO (|B| - |A|),
008D 4907          ADDS   BLO        SINCE A < 0  AND  B > 0.
008E 4503          SUBS   ALO
008F 4402          SUBH   AHI
0090 F680          BZ     CZERO
0091 009A
0092 F380          BLZ    CNEG
0093 00A1
0094 680A          SACH   CHI
0095 600B          SACL   CLO
0096 CA00          ZAC
0097 6008          SACL   CSIGN
0098 FF80          B      NORMAL     GO AND NORMALIZE RESULTS.
0099 00B3
           *
009A CA00  CZERO   ZAC               HERE, ONLY IF RESULT = 0.
009B 6009          SACL   CEXP
009C 6008          SACL   CSIGN
009D 600A          SACL   CHI
009E 600B          SACL   CLO
009F FF80          B      AROUND     OUTPUT A ZERO.
00A0 00D6
           *
00A1 CE1B  CNEG    ABS               HERE, IF RESULT IS NEGATIVE.
00A2 680A          SACH   CHI
00A3 600B          SACL   CLO
00A4 D001          LALK   >FFFF
00A5 FFFF
00A6 6008          SACL   CSIGN
00A7 FF80          B      NORMAL     GO NORMALIZE RESULT.
00A8 00B3
           *
00A9 4002  ADNOW   ZALH   AHI        IF SIGNS ARE THE SAME, JUST ADD.
00AA 4903          ADDS   ALO
00AB 4907          ADDS   BLO
00AC 4806          ADDH   BHI
00AD 680A          SACH   CHI
00AE 600B          SACL   CLO
00AF F080          BV     OVFLOW     DID AN OVERFLOW OCCUR?
00B0 00C4
00B1 F680          BZ     CZERO      IS RESULT = 0 ?
00B2 009A
           *
           *       NORMALIZE
           *
00B3 200A  NORMAL  LAC    CHI        DOES CHI HAVE THE MSB?
00B4 F680          BZ     LO1
00B5 00BC
00B6 400A          ZALH   CHI        IF YES, NORMALIZE RESULT.
00B7 490B          ADDS   CLO
00B8 4B12          RPT    TTEEN      WILL PERFORM 14 "NORMS"
```

```
00B9 CEA2          NORM
00BA FF80          B       OUTPUT      GO OUTPUT RESULTS.
00BB 00D0
00BC 400B  LO1     ZALH    CLO         HERE IF CLO HAS MSB.
00BD C010          LARK    AR0,16      OFFSET EXPONENT BY 16.
00BE F380          BLZ     NOFLOW      DID BIT SEARCH CAUSE OVERFLOW?
00BF 00CD
00C0 4B12          RPT     TTEEN       IF NOT, NORMALIZE RESULT.
00C1 CEA2          NORM
00C2 FF80          B       OUTPUT      GO OUTPUT RESULT.
00C3 00D0
           *
           *
           *       FINISHED WITH NORMALIZATION
           *
           *       HERE ONLY IF OVERFLOW OCCURRED DURING ADDITION
           *
           *
00C4 CE06  OVFLOW  RSXM                RESET SIGN EXTENSION TO SHIFT RIGHT
00C5 CE19          SFR                 SHIFT RIGHT.
00C6 680A          SACH    CHI         STORE NORMALIZED MANTISSA.
00C7 600B          SACL    CLO
00C8 2009          LAC     CEXP        DECREMENT EXPONENT.
00C9 000D          ADD     ONE
00CA 6009          SACL    CEXP
00CB FF80          B       AROUND      GO OUTPUT RESULTS.
00CC 00D6
           *
           *       OVERLOW OCCURRED DURING BIT SEARCH
           *
00CD 5590  NOFLOW  MAR     *-          DECREMENT EXPONENT.
00CE CE06          RSXM                RSXM FOR LOGICAL RIGHT SHIFT.
00CF CE19          SFR                 PERFORM RIGHT SHIFT.
           *
           *
           *       TAKE CARE OF EXPONENT & NORMALIZED MANTISSA,
           *       THEN OUTPUT RESULTS.
           *
           *
00D0 700E  OUTPUT  SAR     AR0,TEMP    HERE AFTER NORMALIZATION.
00D1 680A          SACH    CHI         SAVE NORMALIZED MANTISSA.
00D2 600B          SACL    CLO
00D3 2009          LAC     CEXP        ADJUST EXPONENT.
00D4 100E          SUB     .TEMP
00D5 6009          SACL    CEXP
           *
00D6 5589  AROUND  LARP    1           RESET POINTER.
00D7 4B0F          RPT     THREE
00D8 E0A0          OUT     *+,PA0
00D9 CE1F          IDLE                WAIT FOR INTERRUPT.
NO ERRORS, NO WARNINGS
```

APPENDIX B

```
NO$IDT      32020 FAMILY MACRO ASSEMBLER   PC0.7 84.348    15:24:53  03-27-85
             *** PRERELEASE ***
                                                                PAGE 0001

                   ***********************************************************
                   *                                                         *
                   *    THIS IS A FLOATING-POINT MULTIPLICATION ROUTINE WHICH *
                   *    IMPLEMENTS THE IEEE PROPOSED FLOATING-POINT FORMAT    *
                   *    ON THE TMS32020.                                      *
                   *                                                         *
                   ***********************************************************
                   *
                   *    INITIAL FORMAT (ALL 16-BIT WORDS)
                   *    ------------------
                   *    |   ALL 0 OR 1   |       ASIGN (0 OR -1)
                   *    ------------------
                   *
                   *    ------------------
                   *    |0|.   15 BITS   |       AHI (NORMALIZED)
                   *    ------------------
                   *
                   *    ------------------
                   *    |0|  9 BITS |--0-|       ALO
                   *    ------------------
                   *
                   *    ------------------
                   *    |                |       AEXP (-127 TO 128)
                   *    ------------------
                   *
                   *    TO CORRESPOND WITH IEEE FORMAT,
                   *    INPUT 0.1F * 2 ** (E + 1)
                   *    INSTEAD OF 1.F * 2 **E, AND SUBTRACT 127 FROM E.
                   *
                   *    THE FINAL FORMAT IS THE SAME AS THE INITIAL FORMAT
                   *    EXCEPT THAT FOR CLO WE HAVE:
                   *
                   *    ------------------
                   *    |   16 BITS      |       CLO
                   *    ------------------
                   *
                   *    ALL 16 BITS OF CLO ARE VALID. ANYTHING PAST THESE HAS
                   *    BEEN TRUNCATED.
                   *
                   ***********************************************************
                   *                                                         *
                   *    WORST CASE (EXCLUDING INITIALIZATION AND I/O):       *
                   *    7.8 MICROSECONDS.                                    *
                   *    THIS TIMING INCLUDES THE NORMALIZATION.              *
                   *    WORDS OF PROGRAM MEMORY: 60                          *
                   *                                                         *
                   ***********************************************************
                   *
     0000                        AORG
           0000    ASIGN   EQU      0
           0001    AEXP    EQU      1
           0002    AHI     EQU      2
           0003    ALO     EQU      3
           0004    BSIGN   EQU      4
           0005    BEXP    EQU      5
           0006    BHI     EQU      6
```

```
     0007 BLO     EQU     7
     0008 CSIGN   EQU     8
     0009 CEXP    EQU     9
     000A CHI     EQU     10
     000B CLO     EQU     11
     000C THI     EQU     12
     000D NEGONE  EQU     13
     000E TLO     EQU     14
     000F TEMP    EQU     15
          *
          *
          *       INITIALIZATION
          *
          *
          *
0000 C804         LDPK    4            BEGIN ON PAGE 4.
0001 CE07         SSXM                 SET SIGN EXTENSION.
0002 5589         LARP    1
0003 D100         LRLK    AR1,>200
0004 0200
0005 CB07         RPTK    7            READ NUMBERS INTO BLOCK B0.
0006 80A0         IN      *+,PA0
0007 C000         LARK    AR0,0        CLEAR EXPONENT REGISTER.
0008 5588         LARP    0
0009 D001         LALK    >FFFF
000A FFFF
000B 600D         SACL    NEGONE       NEGONE = -1
          *
          *
          *       BEGIN FLOATING-POINT MULTIPLICATION.
          *
          *
000C 2001 UP      LAC     AEXP         ADD EXPONENTS.
000D 0005         ADD     BEXP
000E 6009         SACL    CEXP
          *
000F 3C03         LT      ALO          FIRST PRODUCT (ALO * BHI)
0010 3806         MPY     BHI
0011 CE14         PAC
0012 680C         SACH    THI
0013 600E         SACL    TLO
          *
0014 3C02         LT      AHI         SECOND PRODUCT (AHI * BLO)
0015 3807         MPY     BLO
          *
0016 CE15         APAC                HAS EFFECT OF (AHI * BLO + ALO * BHI) * 2 ** -15.
0017 CE15         APAC
          *
0018 480C         ADDH    THI
0019 490E         ADDS    TLO
001A 680C         SACH    THI
          *
001B 3806         MPY     BHI         (AHI * BHI)
001C CE14         PAC
001D 490C         ADDS    THI
          *
```

```
001E 690A          SACH   CHI,1        GET RID OF EXTRA SIGN BITS.
001F 610B          SACL   CLO,1
              *
0020 F580          BNZ    OK           IS RESULT ZERO?
0021 0026
0022 CA00          ZAC
0023 6009          SACL   CEXP
0024 FF80          B      SETSIN
0025 002F
              *
0026 400A     OK   ZALH   CHI          NORMALIZE AND WRAP UP.
0027 490B          ADDS   CLO
0028 CEA2          NORM
0029 680A          SACH   CHI
002A 600B          SACL   CLO
002B 700F          SAR    AR0,TEMP
002C 2009          LAC    CEXP
002D 100F          SUB    TEMP
002E 6009          SACL   CEXP
              *
002F 4100     SETSIN ZALS ASIGN        WHAT IS SIGN OF RESULT?
0030 4C04          XOR    BSIGN
0031 F580          BNZ    NEG
0032 0037
0033 CA00          ZAC
0034 6008          SACL   CSIGN
0035 FF80          B      OUTPUT
0036 0039
0037 200D     NEG  LAC    NEGONE
0038 6008          SACL   CSIGN
0039 5589     OUTPUT LARP 1            OUTPUT RESULTS.
003A CB03          RPTK   3
003B E0A0          OUT    *+,PA0
003C CE1F          IDLE
NO ERRORS, NO WARNINGS
```

APPENDIX C

```
NO$IDT       32020 FAMILY MACRO ASSEMBLER   PC0.7 84.348    15:25:17  03-27-85
              *** PRERELEASE ***
                                                                  PAGE 0001

                 *********************************************************
                 *                                                       *
                 *    THIS IS A FLOATING-POINT DIVISION ROUTINE WHICH    *
                 *    IMPLEMENTS THE IEEE PROPOSED FLOATING-POINT FORMAT *
                 *    ON THE TMS32020.                                   *
                 *                                                       *
                 *********************************************************
                 *
                 *    INITIAL FORMAT (ALL 16-BIT WORDS)
                 *    ------------------
                 *    |   ALL 0 OR 1    |       ASIGN (0 OR -1)
                 *    ------------------
                 *
                 *    ------------------
                 *    |0|.    15 BITS   |       AHI (NORMALIZED)
                 *    ------------------
                 *
                 *    ------------------
                 *    |0|  9 BITS |--0-|        ALO
                 *    ------------------
                 *
                 *    ------------------
                 *    |                 |       AEXP (-127 TO 128)
                 *    ------------------
                 *
                 *    TO CORRESPOND WITH IEEE FORMAT,
                 *    INPUT 0.1F * 2 ** (E + 1)
                 *    INSTEAD OF 1.F * 2 **E, AND SUBTRACT 127 FROM E.
                 *
                 *    THE FINAL FORMAT IS THE SAME AS THE INITIAL FORMAT
                 *    EXCEPT THAT FOR CLO WE HAVE:
                 *
                 *    ------------------
                 *    |   16 BITS       |       CLO
                 *    ------------------
                 *
                 *    ALL 16 BITS OF CLO ARE VALID. ANYTHING PAST THESE HAS
                 *    BEEN TRUNCATED.
                 *
                 *********************************************************
                 *                                                       *
                 *    WORST CASE (EXCLUDING INITIALIZATION AND I/O):     *
                 *    22.8 MICROSECONDS.                                 *
                 *    THIS TIMING INCLUDES THE NORMALIZATION.            *
                 *    WORDS OF PROGRAM MEMORY: 92                        *
                 *                                                       *
                 *********************************************************
                 *
     0000                     AORG    0
            0000 ASIGN  EQU     0
            0001 AEXP   EQU     1
            0002 AHI    EQU     2
            0003 ALO    EQU     3
            0004 BSIGN  EQU     4
            0005 BEXP   EQU     5
            0006 BHI    EQU     6
```

```
      0007 BLO    EQU    7
      0008 CSIGN  EQU    8
      0009 CEXP   EQU    9
      000A CHI    EQU    10
      000B CLO    EQU    11
      000C NEGONE EQU    12
      000D TEMP   EQU    13
      000E FOUR   EQU    14
      000F QM     EQU    15
      0010 QL     EQU    16
      0011 R1     EQU    17
      0012 R2     EQU    18
      0013 CL     EQU    19
      0014 M1000  EQU    20
      0015 ONE    EQU    21
      0016 THREE  EQU    22
      0017 FITEEN EQU    23
      0018 THIRTY EQU    24
      0019 TTEEN  EQU    25
                  *
                  *
                  *      INITIALIZATION
                  *
                  *
                  *
0000 C804                LDPK   4            BEGIN ON PAGE 4.
0001 CE07                SSXM                SET SIGN EXTENSION.
0002 5589                LARP   1
0003 D100                LRLK   AR1,>200
0004 0200
0005 CB07                RPTK   7            READ NUMBERS INTO BLOCK B0.
0006 80A0                IN     *+,PA0
0007 5588                LARP   0
0008 C000                LARK   AR0,0        CLEAR EXPONENT REGISTER.
0009 D001                LALK   >FFFF
000A FFFF
000B 600C                SACL   NEGONE       NEGONE = -1
000C D001                LALK   >1000
000D 1000
000E 6014                SACL   M1000        M1000 = >1000
000F CA04                LACK   4
0010 600E                SACL   FOUR         FOUR = 4
0011 CA01                LACK   1
0012 6015                SACL   ONE          ONE = 1
0013 CA03                LACK   3
0014 6016                SACL   THREE        THREE = 3
0015 CA0F                LACK   15
0016 6017                SACL   FITEEN       FITEEN = 15
0017 CA1E                LACK   30
0018 6018                SACL   THIRTY       THIRTY = 30
0019 CA0D                LACK   13
001A 6019                SACL   TTEEN        TTEEN = 13
001B CA00                ZAC
001C 6009                SACL   CEXP         CLEAR CEXP
                  *
                  *
```

```
                *      FINISHED WITH INITIALIZATION
                *
                *
001D 2000              LAC    ASIGN      CSIGN = ASIGN, IF ASIGN = BSIGN.
001E 6008              SACL   CSIGN
001F 1004              SUB    BSIGN
0020 F680              BZ     OK         ELSE, CSIGN = -1.
0021 0023
0022 200C              LAC    NEGONE
                *      SACL   CSIGN
0023 4002       OK     ZALH   AHI        SHIFT DIVIDEND TO PROTECT FROM OVERFLOW.
0024 4903              ADDS   ALO
0025 4B16              RPT    THREE
0026 CE19              SFR
                *
0027 4B17              RPT    FITEEN     QM = AHI!ALO / BHI, R1 = REMAINDER.
0028 4706              SUBC   BHI
0029 6811              SACH   R1         HIGH ACCUMULATOR RETAINS REMAINDER.
002A 600F              SACL   QM
002B 2F11              LAC    R1,15      (R1 * 2**15) / BHI   GIVES QL, AND R2.
002C 4B17              RPT    FITEEN     COMPUTES (R1 * 2**15) / BHI.
002D 4706              SUBC   BHI
002E 6812              SACH   R2         HIGH ACCUMULATOR RETAINS REMAINDER.
002F 6010              SACL   QL
                *
0030 3C0F              LT     QM         CORRECTION TERM = (QM * BLO) / BHI.
0031 3807              MPY    BLO        COMPUTES (QM * BLO).
0032 CE14              PAC
0033 4B17              RPT    FITEEN     COMPUTES (QM * BLO) / BHI.
0034 4706              SUBC   BHI
0035 6013              SACL   CL
0036 400F              ZALH   QM         QM!QL - 0!CL = CHI!CLO
0037 4910              ADDS   QL
0038 1013              SUB    CL
0039 600B              SACL   CLO
003A 680A              SACH   CHI
003B 200A              LAC    CHI        DID AN OVERFLOW OCCUR?
003C 4E14              AND    M1000
003D F680              BZ     NOOVF      IF NOT, GOTO NOOVF.
003E 0041
003F 2015              LAC    ONE        ELSE, INCREMENT CEXP.
0040 6009              SACL   CEXP
0041 2001       NOOVF  LAC    AEXP       COMPUTE RESULTING EXPONENT.
0042 1005              SUB    BEXP
0043 0009              ADD    CEXP
0044 6009              SACL   CEXP
                *
                *
                *      NORMALIZE
                *
0045 200A       NORMAL LAC    CHI        DOES CHI HAVE THE MSB?
0046 F680              BZ     L01
0047 004E
0048 400A              ZALH   CHI        IF YES, NORMALIZE RESULT.
0049 490B              ADDS   CLO
004A 4B19              RPT    TTEEN      WILL PERFORM 14 "NORMS".
```

```
004B CEA2         NORM
004C FF80         B       OUTPUT      GO OUTPUT RESULTS.
004D 0057
004E 400B  L01    ZALH    CLO         HERE, IF CLO HAS MSB.
004F F380         BLZ     NOFLOW      DID BIT SEARCH CAUSE OVERFLOW?
0050 0055
0051 4B19         RPT     TTEEN       IF NOT, NORMALIZE RESULT.
0052 CEA2         NORM
0053 FF80         B       OUTPUT      GO OUTPUT RESULT.
0054 0057
           *
           *
           *      FINISHED WITH NORMALIZATION
           *
           *      OVERFLOW OCCURRED DURING BIT SEARCH
           *
0055 CE06  NOFLOW RSXM                RSXM FOR LOGICAL RIGHT SHIFT.
0056 CE19         SFR                 PERFORM RIGHT SHIFT.
           *
           *
           *      TAKE CARE OF EXPONENT & NORMALIZED MANTISSA,
           *      THEN OUTPUT RESULTS.
           *
           *
0057 680A  OUTPUT SACH    CHI         SAVE NORMALIZED MANTISSA.
0058 600B         SACL    CLO
0059 5589         LARP    1           RESET POINTER.
005A 4B16         RPT     THREE       OUTPUT RESULTS, CSIGN, CEXP, CHI, AND CLO.
005B E0A0         OUT     *+,PA0
005C CE1F         IDLE                WAIT FOR INTERRUPT.
NO ERRORS, NO WARNINGS
```