Fractional and Integer Arithmetic

using the
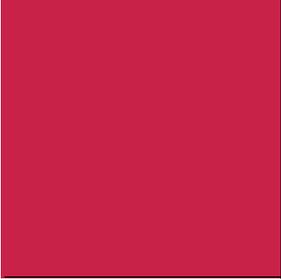
DSP56000 Family of General-Purpose
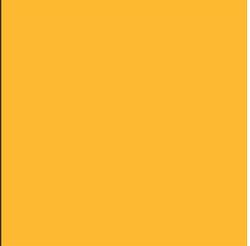
Digital Signal Processors

Table
of Contents

Table
of Contents

Table of Contents

Illustrations

Illustrations

List of Tables
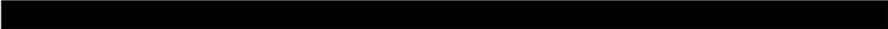
List of Tables

SECTION 1

Introduction

"Data representations are not as difficult as they may first appear. The basic difference between data representations is a shifting or scaling operation."

The DSP56000 Family of general-purpose digital signal processors (DSPs) is distinctive in that the on-chip multiplier directly supports fractional data formats and indirectly supports integer data formats. This application note discusses using the DSP56000/ DSP56001 processors to perform arithmetic operations on data represented as integers, fractions, and combinations thereof, namely mixed numbers, real numbers, or floating-point numbers. A fractional data representation was chosen for the DSP56000/ DSP56001 for the following reasons:

- The most significant product (MSP) of a multiplication has the same format as the input and can be immediately used as an input to the multiplier without a shifting operation.

- The least significant product (LSP) of a multiplication can be rounded into the MSP naturally: i.e., without having to handle a changing exponent.

- All floating-point formats use fractional mantissas

- Coefficients in digital filters are output as fractions by high-level filter-design software packages.
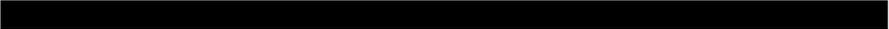
From a hardware point of view, this decision had its primary impact on the design of the multiplier (see **SECTION 4 Signed Multiplication**). Since the format of the resultant operands from addition or subtraction operations is unchanged from the format of the input operands, the choice of integer or fractional formats does not impact the design of the arithmetic logic unit (ALU).

Many data representations will be defined, including twos-complement fractional and integer numbers, real and mixed numbers, and double-precision (48-bit) numbers (see **SECTION 2 Data Representations**). **Signed Multiplication** and **Signed Division** (**SECTIONs 4 and 5**, respectively) discuss multiplication and division using these data representations.

Data representations are not as difficult as they may first appear. The basic difference between data representations is a shifting or scaling operation. Performing shifting operations with the DSP56000 Family of processors is discussed in **SECTION 2.5 Data Shifting. Mixed-and Real-number Addition And Subtraction** (Section 3) discusses operations involving more than simple ADD and SUB instructions. Division-yielding quotients and remainders that are not word (24-bit) multiples are described in **SECTION 5.6 Divide Routines With N $\leq$ 24 Bits.**   ■

SECTION 2

Data Representations

"The DSP56000 Family processors provide four distinct ways to perform data shifts . . ."

Different data representations are introduced and discussed in the following paragraphs. The following sections also present routines, where required, that convert numbers from one data representation to another.

2.1 Twos-Complement Fraction

A fraction, F, is any number whose magnitude satisfies the inequality:

$$0.0 \leq mag(F) < 1.0$$

Examples of fractions are 0.25 and -0.87. The twos-complement fractional data representation is shown in Figure 2-1. The binary word is interpreted as having a binary point after the most significant bit (MSB). The range of numbers that can be represented using N-bit twos-complement fractional data is:

$$-1.0 \leq F \leq 1\text{-}2\text{-}(N\text{-}1)$$

$$s.fffffffffffffffffffffff$$

$$\longleftarrow \text{N bits} \longrightarrow$$

where:

s = the sign bit
f = a fractional bit
. = the implied binary point
N = the number bits
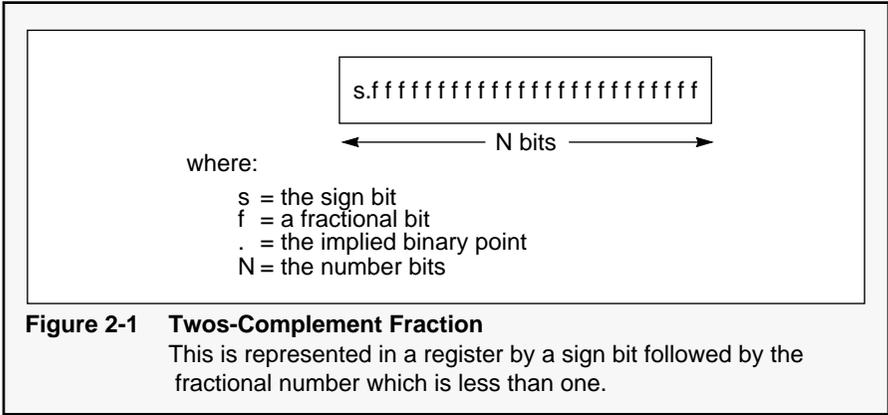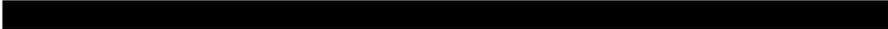
Figure 2-1   Twos-Complement Fraction
This is represented in a register by a sign bit followed by the fractional number which is less than one.

DSP56000/DSP56001 processors use a fractional data representation for all arithmetic operations. Figure 2-2 shows the bit weighting and register names for words, long words, and accumulator operands in the DSP56000/DSP56001 processors.

For words and long words, the most negative number that can be represented is -1.0 whose internal representation is $800000 and $800000000000, respectively. The "$" sign denotes a hexadecimal value. The most positive word is $7FFFFF or $1-2^{-23}$ = 0.9999998, and the most positive long word is $7FFFFFFFFFFF or $1-2^{-47}$ =0.999999999999993. These limits apply to data stored in memory and to data stored in the data ALU input pipeline registers. The accumulators, A and B, have 8-bit extension registers, A2 and B2, respectively. This extension allows word growth so that the most posi-

tive and negative numbers that can be represented in the accumulators are +255.999999999999993 and -250.0, respectively.

Word Operand (24 bits)

Weighting = -2^0 2^{-23}

Registers: X1 X0 Y1 Y0 A1 A0 B1 B0

Long Word Operand (48 bits)

Weighting = -2^0 2^{-24} 2^{-47}

Registers: X1:X0=X   Y1:Y0=Y   A1:A0=A1   B1:B0=B10

Accumulator (56 bits)

Weighting = -2^8 2^0 2^{-24} 2^{-47}

Sign Extension   Operand   Zero

Registers: A = A2:A1:A0   or   B = B2:B1:B0

Figure 2-2 DSP56000 Operands
These are fractional number representations that can be one word long, two words long or two words with eight bits of sign extension and overflow extension.

An immediate fractional number can be stored in a general-purpose register (for example, X0) by simply using the MOVE immediate instruction. For example, execution of:

MOVE #.5,X0

with result in $400000 (0.5) being stored in X0.

2.1.1 Twos-Complement Integer

An integer, I, is a number that contains no decimal point and its magnitude satisfies the inequality:

$$1 < mag(I)$$

Examples of integer numbers are 1, 256, and -67. The twos complement integer data representation is shown in Figure 2-3. The binary word is interpreted as having a binary point after the least significant bit (LSB) — that is, the data is right-hand justified. The range of number that can be represented using N-bit twos-complement integers is:

$$-2^{N-1} \leq 1 \leq 2^{N-1}-1$$

The MSBs are sign-extension bits. Caution must be exercised when moving integer data on the DSP56000/DP56001 because the DSP56000/ DSP56001 will naturally tend to left-hand justify data (i.e., assuming the data is a fraction).

s i i i i i i i i i i i i i i i i i i i i i i i i .

$\longleftarrow$ N bits $\longrightarrow$

where: s = the sign bit
         i = an integer bit or sign extension bit
         . = the implied binary point
         N = the number bits

Figure 2-3   Twos-Complement Integers
These can be represented in DSP56000 registers but caution must be exercised when moving data.

The DSP56000 cross assembler provides forcing functions that can be used to facilitate handling integer data, which is best described by way of example. To store the integer value 56 ($38) in X1, the programmer would be tempted to use:

MOVE #56,X1

The value stored in X1 would be $380000 (3670016), which is clearly incorrect. The error occurred because the assembler will always pick the shortest form of instruction encoding. Data less than eight bits can be encoded in the MOVE instruction without the use of an extension word. The DSP56000/DSP56001 interprets this data as fractional and therefore stores it left-hand justified as demonstrated in this example. If the immediate long force operator, >, is used, an extension word will be used, and the data will be right-hand justified as the following example shows:

MOVE # > 56,X1

The content of X1 will be $000038. For integers of magnitude greater than 128, the short addressing is not applicable since the number will occupy more than eight bits. The value will therefore be treated as a 24-bit number by the assembler and encoded into the LSBs of the extension word. As an example, execution of:

MOVE #1234,X1

will result in X1=$0004D2 (1234).

2.2 Double-Precision Numbers

A double-precision number is a 48-bit twos-complement number, fraction or integer, that is stored as a long-word operand. The range of a double-precision twos-complement fractional number is:

$$-1 \le \text{double-precision fraction} \le 1\text{-}2^{-47}$$

The range for a double-precision integer is:

$$-140737488355328 \le \text{double-precision integer} \le 140737488355327$$

2.3 Real Numbers

A real number, R, consists of an integer part and a fractional part. The decimal point separates the two parts. Only the integer part has a sign bit, and it may assume the value zero. The real-number representation discussed in this document consists of a 24-bit integer portion and a 24-bit fractional portion (see Figure 2-4).

Figure 2-4   Real-Number Format
This is a concatenation of the twos-complement integer representation and twos-complement fraction representation.
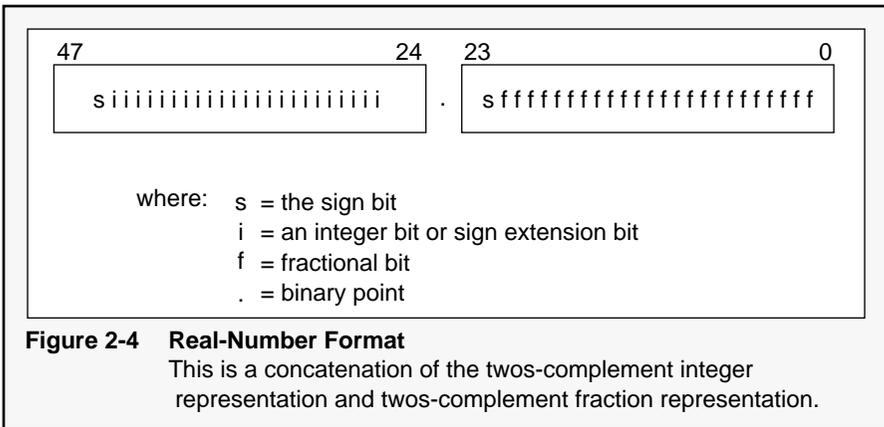
For long-word operands, like X (X1:X0), the integer part of the real number will occupy the upper 24 bits of X, X1, and the fractional part with occupy the lower 24 bits, X0. The binary point is assumed to occupy an imaginary place between bit 23 and bit 24; whereas, the sign bit occupies the leftmost bit of the integer portion. The range of a real number, R, is:

$$-8388608.0 \leq R < 8388607.9999999$$

Examples of real numbers are:

56.789, 0.345, and -789.123.

The convert to real macro, CONVR, presented in Figure 2-5, performs the conversion of a real, positive decimal number, xr, to the real-number format for storage. This macro uses the convert to integer function, CVI, built into the assembler to convert real numbers to integers by simply truncating the fractional part of the number. When the fractional portion is moved into A0, it will be signed. A left shift is subsequently performed to eliminate this sign bit.

```
;CONVR.ASM
;This macro converts a real positive decimal number,
;xr (0.0 < xr < 8388607.9999999), to the real number format.
;The signed integer part is stored in the upper part of the
;A accumulator (A1), and the unsigned fractional part is stored in A0.
CONVR macro xr                          ;macro definition
        clr a                           ;clear the accumulator
        move #(xr-@cvi(xr)),a0          ;store fractional part in A0
        asl a                           ;eliminate sign bit in fract. part
        move #@cvi(xr),a1               ;store the integer part in A1
        endm                            ;end macro definition
```

Figure 2-5   CONVR Macro Definition
This converts a real, positive decimal number to the real number format.

```
;CONVRG.ASM
;This macro converts a real decimal number,
;xr (8388607.9999999 > xr >-8388608.0), to the real number format.
;The signed integer part is stored in the upper part
;of the A accumulator (A1), and the unsigned fractional part is stored in A0.
CONVRG macro xr                 ;macro definition
        clr  a                  ;clear accumulator a
        move #(xr-@cvi(xr)),a0  ;move the fraction into A0
        asl  a   #>@cvi (xr),x1 ;shift the fraction's sign bit into A1
        move a1,x0              ;move the int. into X1, move A1 to X0
        move x1,a1              ;move the int. to A1
        sub  x0,a              ;convert the integer to one's complement
        endm
        endm
```

Figure 2-6   CONVRG Macro Definition
        This converts a real decimal number to the real-number format.

Converting a negative real number to the real-number format involves two steps. First, the absolute value of the number has to be stored into an accumulator in the real-number format; then the stored value is negated.

The convert real general macro, CONVRG, depicted in Figure 2-6, handles both positive and negative operands. The ASL instruction will eliminate the sign bit in the fractional part.

If the number is positive, the number in register A has been correctly converted. If xr is negative, the sign bit of the fractional part will propagate from A0 to A1 due to the ASL instruction. This sign bit is used to subtract out the one that was added to the integer portion when it was converted to a twos-complement number. The single case in which one should not be subtracted from the integer is when the fraction
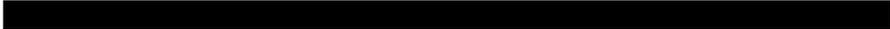
portion is zero. In this case, since CVI always returns a zero as positive, zero is subtracted from the integer, and the result is correct.

A good example to demonstrate the CONVRG macro is the case where xr=-1.5. After the SUB X0,A instruction has been executed, A=$00:FFFFFE:800000 (-1.5 in the real-number format). The extension register contains all zeros even though the real number is negative. That is, bit 47 is the true sign bit for the 48-bit real number. It is not immediately apparent that $00:FFFFFE:800000 represents -1.5. It becomes apparent after the absolute value of A1:A0 is taken, which is accomplished by moving the long word, A1:A0, into accumulator B (so that bit 47 is properly sign extended), and then taking the absolute value of B. This yields B=$00:000001:800000 (+1.5 in the real-number format).

In summary, real numbers must be treated as 48-bit entities — for example, do not take the absolute value of the integer portion only.

2.4 Mixed Numbers

A mixed number (MN) is a special case of a real number in that the number occupies 24 bits instead of 48 bits and satisfies the inequality:

$$-128.0 \leq MN < 128.0$$

Examples of mixed numbers are -67.875 and 89.567. As seen in the mixed-number format given

in Figure 2-7, the 24-bit word is divided into two parts. The first part (which is an 8-bit 1's complement number) contains the integer portion of the mixed number including the sign bit related to the whole 24-bit number. The second part (which is a 16-bit 2's complement number with the sign bit shifted into the first part) contains the unsigned fractional portion of the number.

s i i i i i i i . f f f f f f f f f f f f f f f f

←──────── 24 bits ────────→

where: s = the sign bit
        i = an integer bit or sign extension bit
        . = the binary point
        f = a fractional bit

Figure 2-7 Mixed Number Format
This is a variation of the real-number format that allows a real number to reside in a single register.
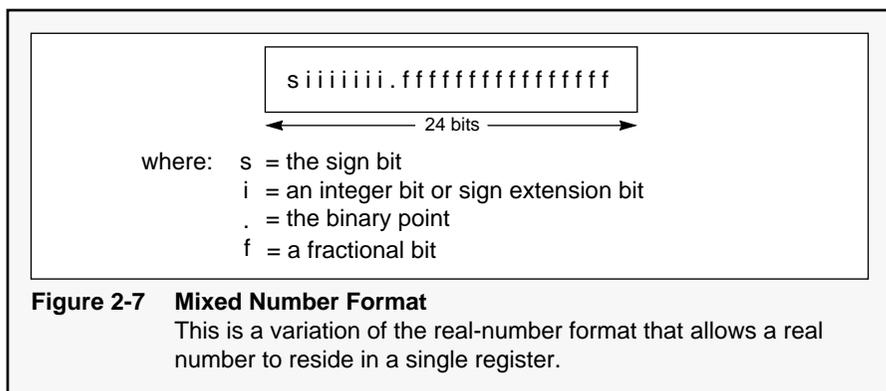
The virtue of the MN format is that MN-formatted data can be treated as 24-bit signed data by the machine — that is, the integer and fractional portions do not have to be treated separately. Two macros have been prepared to show how real numbers having a magnitude less than 128 can be stored in 24 bits and set up in the MN format.

```
;CONVMN.ASM
;This macro converts a positive decimal mixed number to the MN format.
;The mixed number is stored in A1. 0.0 ≤ xmn < 128.0
CONVMN       macro   xmn                       ;macro definition
             move    #(xmn-@cvi (xmn)),x0      ;fractional part to X0
             move    #$010000,y1               ;shift constant in y1
             mpyr    y1,x0,a  #@cvi (xmn),x1   ;shift X0;integer part in X1
             add     x1,a                      ;concatenate int. and fract.
             endm                              ;end macro definition
```

Figure 2-8   CONVMN Macro
               This converts a positive decimal mixed number to the MN format.

The first macro, convert to MN, CONVMN, converts any positive mixed number to the MN format (see Figure 2-8). The macro does not check the sign or the magnitude of the input; it is the user's responsibility to make the necessary checks. The fractional part of the number is shifted to the right by seven bits, using a shift constant, $010000, and the MPYR instruction discussed in **SECTION 2.5 Data Shifting**; therefore, the first eight bits of the 24-bit word are zero (see Figure 2-8). This instruction ensures that the fraction is stored in the upper part of the accumulator (A1 in this case) and that it is 16 bits (rounded). In parallel with the MPYR instruction, the integer is moved to X1 by using the move immediate short instruction that places the 8-bit signed integer in the upper eight bits of the 24-bit register, X1. The ADD X1,A instruction concatenates the integer and fractional parts to form the mixed number and sign extends the mixed number so that the

number can be immediately used in the data ALU. A second macro, convert to MN general, CONVM-NG, handles signed real numbers having a magnitude less than 128 (see Figure 2-9). When a negative number is detected, it is made positive, then transformed into the MN format, and finally negated again.

```
;CONVMNG.ASM
;This macro will convert a decimal number xmn,
;where -128.0 < xmn < 128.0, into the mixed number, MN, format.
;
CONVMNG   macro     xmn                    ;macro definition
    move      #(xmn - @cvi(xmn)), x0 ;obtain signed fract. part
    clr  b    #$080000,y1            ;clear b, shift constant in Y1
    mpyr      y1,x0,a#@cvi(xmn),x1   ;shift fract.;integer in X1
    eor       x1,b                   ;set sign
    jeq       _endf                  ;finished if integer = 0
    jpl       _endit                 ;jump if positive
    neg  b    #$008000,y1            ;int. positive;left shift const.
    neg  a    b1,x1                  ;fract. positive; integer in X1
    mpy       y1, x1,b               ;shift integer and store in B
    move      b0,x1                  ;obtain the shifted integer
    add       x1,a                   ;add the int. to the fract.
    neg       a                      ;negate mixed number entity
    jmp       _endf                  ;jump to the end
_enditadd b,a                        ;concatenate positive int.and fr.
_endf                                ;finished
    endm                             ;end macro definition
```

Figure 2-9   CONVMNG Macro Definition
        This converts a decimal number to the mixed-number format.

Figure 2-9 shows how the detection of the sign of the number is done using the EOR instruction. The EOR is performed with accumulator B, which is already zero. The integer will reside in the most significant byte of B1, and the N bit in the condition code register (CCR) will be set if the integer was
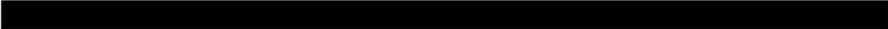
negative. The negation is now performed by using the NEG instruction. When the negative B1 is negated to turn its contents into a positive number, the integer occupies the lowest eight bits of B1. To move the 8-bit number to the upper eight bits of a 24-bit register, a left shift by 16 bits must be performed by utilizing the ideas presented in the following section on data shifting. A 16-bit left shift by the use of a shift constant will force the 8-bit number to reside in the lower 24 bits, B0, of the destination accumulator B. To concatenate the 8-bit signed integer with the 16-bit fraction, the number is moved to X1 and then added to accumulator A containing the unsigned fraction.

2.5 Data Shifting

Data shifting is used in converting one data representation into another data representation. The DSP56000 Family processors provide four distinct ways to perform data shifts:

1. 1-bit shifts/rotates
2. multi-bit shifts/rotates
3. fast multi-bit shifts
4. dynamic scaling.

These approaches to shifting are described in the following sub-sections.

2.5.1 1-Bit Shifts/Rotates

For 1-bit shifts/rotates of either 56-bit accumulator A (A=A2:A1:A0=8:24:24 bits) or 56-bit accumulator B (B=B2:B1:B0=8:24:24 bits), use the arithmetic shift right (ASR) and arithmetic shift left (ASL) instructions. If 1-bit shifts on only the most significant 24-bit word of accumulator A, A1, or accumulator B, B1 are required, use the rotate right (ROR), rotate left (ROL), logical shift right (LSR), or logical shift left (LSL) instructions.

2.5.2 Multi-Bit Shifts/Rotates

The most straightforward approach for multi-bit shifts/rotates of the accumulator is to use ASR, ASL, LSR, LSL, ROR, or ROL instructions with the repeat instruction, REP, or the hardware DO loop instruction, with the loop consisting of a single instruction as the examples in Figure 2-10 show. The repeat instruction is not interruptible; whereas, the DO instruction is interruptible.

```
       REP #n
       ASL A
  or
       DO  #n, END1
       ASR A
       END1
```

(where n is the number of positions to be shifted/rotated)

Figure 2-10 Multi-Bit Shifts Using REPeat, DO

The DSP56000 macro cross assembler supports macros with the MACRO DEFINITION and MACLIB directives (see Reference 1). Two accumulator shift macros, one for left shifts, SHLAC, and the other for right shifts, SHRAC, can be defined as shown in Figure 2-11.

```
;Macros for performing multi-bit shifts right and left.
;For the two given macros
;       Let   acc  = accumulator A or B
;             n    = the number of bits to be shifted
;
SHRAC    macro  acc,n              ;macro definition for shifting the
         rep    #n                 ;accumulator right n bits.
         asr    acc                ;shift right
         endm                      ;end macro definition
;
SHLAC    macro  acc,n              ;macro definition for shifting the
         rep    #n                 ;accumulator left n bits
         asl    acc                ;shift left
         endm                      ;end macro definition
```

Figure 2-11 Multi-Bit Shift Macros use the ASR or ASL instruction shifts one bit per instruction cycle.

2.5.3 Fast Multi-Bit Shifts

The fastest way to do multi-bit shifting is to multiply the operand by a shift constant. In the case of a right shift, the constant KR is a fraction given by $KR=2^{-n}$. The example in Table 2-1 shows how to shift the content of X0 right by four bits. The shifted result resides in the upper part of accumulator A, A1. The code executed to implement the 4-bit right shift shown in Table 2-1 is:

MOVE #KR,X1   MPY X0,X1,A

Table 2-1 Fast 4-Bit Right Shift		
Register	**Hexadecimal Value**	**Comments**
X0	060000	Value To Be Shifted
X1	080000	Shift Constant, KR
A1	006000	Shifted Result

Similarly, the example in Table 2-2 shows how to shift the content of X0 left by four bits. In the case of a left shift, the constant KL is an integer given by $KL=2^{n-1}$. KL is 2^{n-1}, not 2^n, because the DSP56000 multiplier is fractional, thereby automatically implementing a 1-bit left shift. The result for left shifts resides in A0, the least significant word of A. The code executed to implement the 4-bit left shift is:

```
MOVE   #>KL,X1
MPY    X0,X1,A
```

Table 2-2 Fast 4-Bit Left Shift		
Register	**Hexadecimal Value**	**Comments**
X0	060000	Value To Be Shifted
X1	000008	Shift Constant, KL
A0	600000	Shifted Result
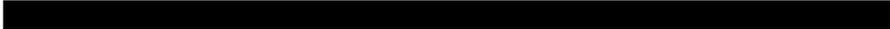
Generating the constants for the shifts is made easy by using the POW and CVI functions built into the DPS56000 macro cross assembler. The raise to the power function, POW, returns a real number for any base raised to a real number. For example,

K=@pow(2,-4) returns 0.0625, and

K=@pow(2,+4) returns 16.0

However, because DPS56000 is a fractional machine, the assembler will limit real numbers unless precautions are taken. In the previous example, the object code for 16.0 will be limited to $7FFFFF or +0.999998 decimal by the assembler, which is incorrect. To obtain the integer form of the real number, the assembler provides a convert to integer function, CVI. The CVI function converts real numbers to integers by truncating the fractional portions. For example,

@CVI (@pow(2,+4))

returns 16 (not 16.0), which will be assembled as $000010 in object code.

The previous instruction sequences can be put in two distinct macros for programming ease. The two macros, MSHR for multi-bit shifts right and MSHL for multi-bit shifts left, are listed in Figure 2-12.

```
;Macro definitions for generating right and left shift constants,
;KR and KL, and performing the right and left shifts.
;
;         Let     s   = the source register
;                 m   = the multiplier register
;                 n   = the number of bits to be shifted
;                 acc = the destination accumulator
;
;         where   s,m can be one of X0,X1,Y0,Y1
;         and     acc can be A or B
;
;
MSHR      macro   s,m,n,acc                 ;four input variables.
          move    #@pow(2,-n),m             ;load the multiplier register
          mpy     s,m,acc                   ;shift right n bits
          endm                              ;end macro definition
;
MSHL      macro   s,m,n,acc                 ;four input variables
          move    #>@cvi(@pow(2,n-1)),m     ;load the multiplier register
          mpy     s,m,acc                   ;shift left n bits
          endm                              ;end macro definition
```

Figure 2-12 Constant Generation and Multi-Bit Shifts
          This uses the ultiply instruction to perfor multi-bit sht in one
          instruction cycle.

The immediate long-data move (note the greater
than sign, >, in the move instruction) must be used
in the MSHL example to prevent the data from be-
ing treated as a fraction and shifted accordingly. In
the following example, for:

MOVE #2,X1

X1 will be $020000 because the immediate short data
(i.e., data which can be represented as eight bits) is
treated as an 8-bit fraction occupying the two most sig-
nificant bytes of the destination register; whereas, for:

MOVE #>2,X1
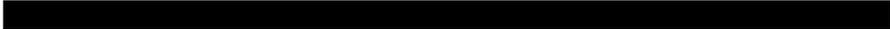
X1 will be $000002 because the data is treated as an integer occupying the least significant bytes of the destination register with sign extension.

The immediate long-data move is a two-word instruction that executes in two instruction cycles; whereas, the immediate short-data move is a one-word instruction that executes in one instruction cycle. The immediate short-data move can be used for multi-bit right shifts of less than or equal to eight bits. For right shifts of more than eight bits, the immediate long-data move must be used with the appropriate 24-bit fraction, 2**(-n), utilizing the POW directive.

2.5.4 No-Overhead, Dynamic Scaling (1-Bit Shifts)

For no-overhead 1-bit shifts of either accumulator A or B, the scaling mode is used. In this mode the shift occurs automatically when transferring data from either of the 56-bit data ALU accumulators, A or B (not A2, A1, A0, A10 or B2, B1, B0, B10), to the XD or YD buses. This shift function is activated by appropriately setting the scaling-mode bits in the status register. This mode is primarily intended for adding a scaling operation to existing code without modifying the code (simply setting the scaling-mode bits). For more details on the scaling mode, consult the **DSP56000 Digital Signal Processor User's Manual** (see Reference 2) and the **DATA ALU** subsection of ADI1290 (see Reference 3). ∎

SECTION 3

Mixed- and Real-Number Addition and Subtraction

The arithmetic operations of addition and subtraction performed on mixed and real numbers are discussed in the following paragraphs.

3.1 Mixed Numbers

Mixed numbers can be represented in a 24-bit word using the MN format discussed in **SECTION 3.2 Real Numbers**. To better understand addition and subtraction of mixed numbers, consider the examples in the following paragraphs.

3.1.1 Addition

Two examples of mixed-number addition are considered.

Example One—The simplest case is the addition of two positive numbers as shown in Table 3-1. The instruction executed is:

ADD X1,A

Table 3-1 Positive Mixed Numbers with Sum Less than 128		
Register	**Hexadecimal Value**	**Mixed-Number Value**
X1	43C000	67.75
A (before)	00 : 178000 : 000000	23.50
A (after)	00 : 5B4000 : 000000	91.25

Example Two—In this example, the result of the addition will be greater than 128, which is the limit for 24-bit MN-formatted mixed numbers. However, the status register will signify the use of the extension part of the accumulator; thus, the exact representation of the mixed number having a magnitude greater than 128 can be contained in the accumulator. It cannot be stored as a 24-bit word, however, since it requires more than 24 bits to represent it.

Consider the example shown in Table 3-2. The value of the status register, SR, is $0320, signifying the E bit, bit 5, has been set. The hexadecimal value of B1 represents the decimal number 131.0 because the status register indicates the extension bits of accumulator B are in use; thus, bit 47 of B is not a sign bit but part of the mixed number.

Table 3-2 Positive Mixed Numbers with Sum Greater than 128		
Register	**Hexadecimal Value**	**Mixed-Number Value**
Y1	464000	70.25
B (before)	00 : 3CC000 : 00	60.75
B (after)	00 : 830000 : 00	131.00

3.1.2 Subtraction

Mixed-number subtraction is as straightforward as the mixed-number addition.

Example Three— Consider the case shown in Table 3-3. The instruction executed is:

        SUB X0,B

The status register value remains the same.

Table 3-3 Mixed-Number Subtraction		
Register	**Hexadecimal Value**	**Mixed-Number Value**
X0	178000	23.50
B (before)	00 : 43C000 : 000000	67.75
B (after)	00 : 2C4000 : 000000	44.25

Example Four— Consider the case depicted in Table 3-4 where the result is negative. The N bit, bit 3, and the borrow (carry) bit, bit 0, in the status register are set, which indicates that the result is negative and that a borrow has occurred. The magnitude of the negative mixed number can be easily found by executing the ABS B instruction.

Table 3-4 Mixed-Number Subtraction with Negative Result		
Register	**Hexadecimal Value**	**Mixed-Number Value**
X1	464000	70.25
B (before)	00 : 3CC000 : 000000	60.75
B (after)	FF : F68000 : 000000	-9.50

3.2 Real Numbers

Consider real numbers having the format discussed in **SECTION 2.3 Real Numbers** in which the signed integer occupies the most significant 24-bit word, and the unsigned fraction occupies the least significant 24-bit word of a 48-bit-long word.

3.2.1 Addition

The numbers to be added should be moved into any of the acceptable source registers (X, Y, A, B) and destination registers (A, B) for the 48-bit addition (see Reference 2). If the sum of the fractional parts

is greater than unity, a carry is propagated into the integer part. If, after adding the two real numbers, the integer result cannot be represented in 24 bits, then the extension part of the accumulator will be used. Bit 5 in the status register will indicate whether the extension bits are in use.

Example One—Consider the case shown in Table 3-5. The instruction executed is:

ADD X,A

Although the real-number source may have been saved in A10, bit 47 and A2 must represent proper sign extension if the C bit in the status register is to be set correctly, which is necessary when doing multiple-precision arithmetic. The fraction parts in X0 and A0 are unsigned.

Table 3-5 Real-Number Addition		
Register	Hexadecimal Value	Real-Number Value
X	000237 : C00000	567.750
A (before)	00 : 0003DB : A00000	987.625
A (after)	00 : 000613 : 600000	1555.375

Example Two— Consider the case shown in Table 3-6. If the first bit of the result is interpreted as a sign bit, the decimal value of A1 is not 8389160 but is -8388056. The reason that the hexadecimal value in A represents the correct result (i.e., +8389160) is

because the extension bit, bit 5 in the status register, is set. This fact indicates that the extension bits of accumulator A are in use; therefore, the sign bit is not the left-most bit (bit 47) of A1 but is the left-most bit (bit 55) of the extension register, A2.

Table 3-6   Real -Number Addition Using the Extension Bit

Register	Hexadecimal Value	Real-Number Value
X	000237 : C00000	567.750
A (before)	00 : 7FFFF0 : A00000	8388592.625
A (after)	00 : 800228 : 600000	8389160.375

3.2.2 Subtraction

The subtraction of real numbers is similar to the addition of real numbers.

Example Three— The case shown in Table 3-7 generates a positive result. The instruction executed is:

SUB X,A
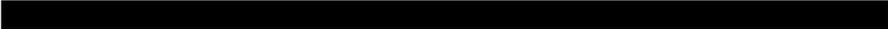
Table 3-7	Real-Number Subtraction with a Positive Result	
Register	**Hexadecimal Value**	**Real-Number Value**
X	000138 : C00000	312.750
A (before)	00 : 00037A : 400000	890.250
A (after)	00 : 000241 : 800000	577.500

Example Four— The case depicted in Table 3-8 generates a negative result. The N bit, bit 3, and the C bit, bit 0, in the status register are set, indicating the result is negative and a borrow has occurred. The magnitude of the negative real number can be easily found by executing the ABS A instruction.

Table 3-8	Real-Number Subtraction with a Negative Result	
Register	**Hexadecimal Value**	**Real-Number Value**
X	00037A : 400000	890.250
A (before)	00 : 000138 : C00000	312.750
A (after)	FF : FFFDBE : 800000	-577.500

∎

SECTION 4

Signed Multiplication

"Integer or fractional multiplication can be accomplished on any signed hardware multiplier by appropriate shifting."

Consider the multiplication of two signed-integer numbers (see Figure 4-1). The product of the signed multiplier is 2N-1 bits long. To keep the product properly normalized and to further process the product, it is advantageous to format the product as multiple operand words. Therefore, there is an extra bit because two sign bits exist before multiplication and only one exists after the multiplication. Integer multipliers use the extra bit as a sign-extension bit.

Multiplication of signed fractions is shown in Figure 4-2. As is the case for signed-integer multiplication, the result of the multiplication is a 2N-1 bit word, including the sign bit. In this case, the extra bit is appended to the LSP as a zero in the LSB position. This bit is called the zero-fill bit.
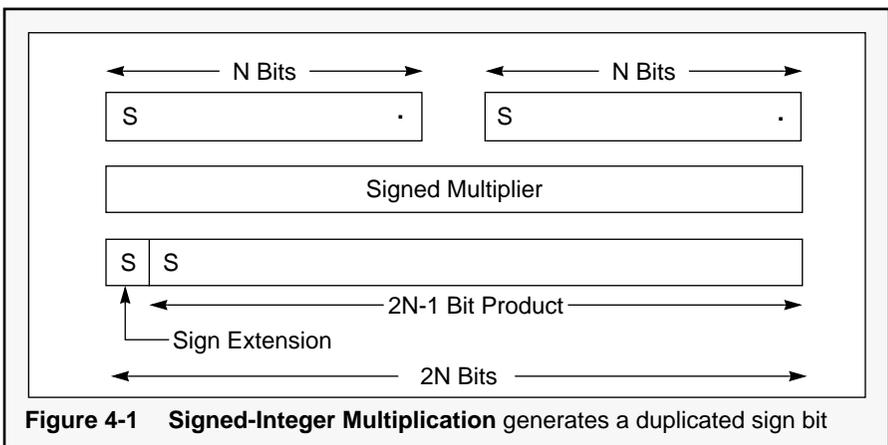
Figure 4-1   Signed-Integer Multiplication generates a duplicated sign bit

---
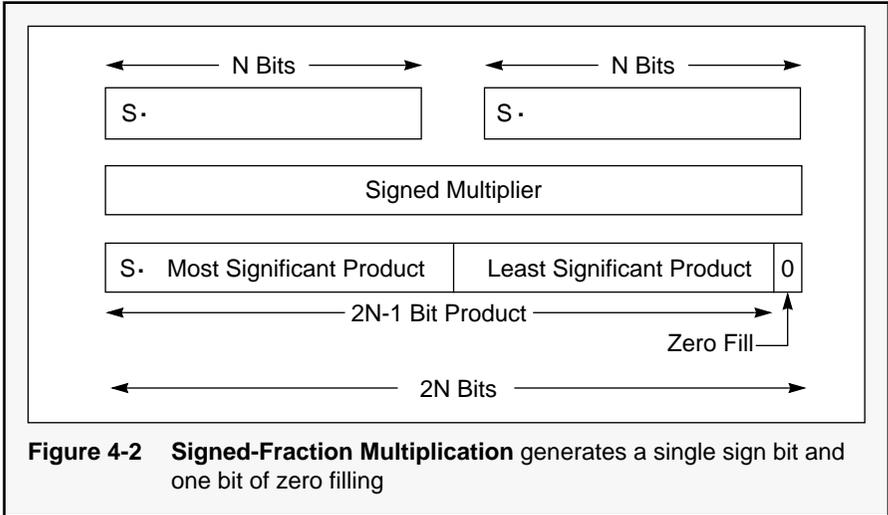
Figure 4-2   Signed-Fraction Multiplication generates a single sign bit and one bit of zero filling

In summary, signed-integer and signed-fractional multipliers differ only in the way in which they treat the extra bit. In the integer case, the bit is used for sign extension; whereas, in the fractional case, it is used as zero fill. Integer or fractional multiplication can be accomplished on any signed hardware multiplier by appropriate shifting. The following paragraphs discuss performing fractional, integer, mixed-number, and real-number multiplications using the DSP56000 Family of processor.

4.1 Multiplication of a Signed Fraction with a Signed Fraction

Let the values of the 24-bit general-purpose registers X1 and X0 be as shown in Table 4-1. After executing MPY X0,X1,A on the DSP56000/ DSP56001, the content of accumulator A is as shown in Table 4-1. The last bit of the accumulator is zero, and the first bit carries the sign of the product. When accumulator A is rounded to 24 bits using the instructions RND or MPYR X0,X1,A, the value in A is $00:009D99:000000 (see Table 4-1). The lower 24 bits, A0, are zeros, and the eight sign-extension bits, A2, of the 56-bit accumulator are zeros, indicating a positive number.

Table 4-1 Signed-Fraction Multiplication		
Register	**Hexadecimal Value**	**Real-Number Value**
X0	0647D9	+0.049067616462708
X1	0C8BD3	+0.098017096519470
A	00 : 009D98 : B815B6	+0.004809465298806
A (RND)	00 : 009D99 : 000000	+0.004809498786926

4.2 Multiplication of a Signed Integer with a Signed Integer

Consider the case represented in Table 4-2 in which two signed integers in X0 and X1 are multiplied, and the result is stored in accumulator A. It can be seen from Table 4-2 that if the contents of X0, X1, and A are interpreted as fractions, the result is correct. However, if the contents of X0, X1, and A are interpreted as integers, then a shift is required immediately after the multiplication to obtain the correct results. This shift moves the LSB out of the accumulator and adds a sign-extension bit in the MSB position. Therefore, the instruction sequence to perform integer multiplication on DSP56000/ DSP56001 processors is a multiplication followed by a right shift, namely,

MPY X0,X1,A

ASR A

Table 4-2 Signed-Integer Multiplication			
Register	**Hexadecimal Value**	**Integer Value**	**Fractional Value**
X0	000002	2	2.3841858E-07
X1	000138	312	3.7193298E-05
A	00 : 000000 : 0004E0	1248	8.8675733E-11

Table 4-3 Signed-Integer and Signed-Fraction Multiplication

Register	Hexadecimal Value	Integer Value	Fractional Value
X0	400000		0.5000000
X1	00007F	127	
A10	00003F : 800000		
A1	00003F	63	
A0	800000		0.5 (unsigned)

4.3 Multiplication of a Signed Integer with a Signed Fraction

Multiplication of an integer with a fractional number is a unique case since the result will be a real number — i.e., it will consist of an integer and a fractional part. When the contents of X0 and X1 are as shown in Table 4-3, execution of the instruction MPY X0,X1,A will result in A1=$00003F and A0=$800000.

The integer part will be stored in the upper 24 bits, A1, of the 48-bit result, and the fractional part will reside in the lower 24 bits, A0, of the result. A0 is being interpreted as an unsigned fraction. When performing multiple-precision arithmetic on real numbers, it is necessary to convert real numbers into a signed-integer operand and a signed-fraction

operating (see **SECTION 4.4 Double-Precision Multiplication**). To format A0 as a twos-complement positive fraction, two shift operations must be performed, LSLA followed by ASRA. The execution of LSLA shifts the MSP, A1, one bit left and inserts a zero in the LSB position of A1. The execution of ASRA shifts the full 56-bit accumulator A one bit right, thereby restoring A1 and forming a positive twos-complement fraction in A0. If the product of the multiplication is negative, then introducing the sign bit in the fractional part involves three steps. First, the absolute value of the number must be obtained. Second, the shift LSLA followed by ASRA should be performed to generate a signed twos-complement fraction. Finally, the negative values of both parts, integer and fractional, must be obtained separately. The convert to signed integer and signed fraction routine, CONVSISF, given in Figure 4-3, implements these three steps.

```
;CONVSISF.ASM
;This routine will convert a negative 56-bit number in the real number format
;(with a signed integer in A2:A1 and an unsigned fraction in A0)
;to a signed integer in A1 and a signed fraction in B1
;
                abs  a         ;obtain the absolute value of the result
                lsl  a         ;shift left to introduce sign bit
                asr  a         ;introduce positive sign in fractional part
                move a0,b      ;move positive fraction to B1
                neg  b #0,a0   ;negate fraction, clear A0
                neg  a         ;negate integer
```

Figure 4-3   CONVSISF Routine converts a negative 56-bit number to a signed integer and signed fraction

4.4 Double-Precision Multiplication

In double-precision multiplication, two 48-bit numbers are multiplied together to generate a 96-bit signed product. The concept of double-precision multiplication is depicted in Figure 4-4. When two 48-bit numbers P and Q (where P1 and P0 are the most significant and least significant 24-bit words, respectively, of P and, similarly, Q1 and Q0 for Q) are multiplied, four single-precision products, P0Q0, P1Q0, P0Q1, and P1Q1, are generated. These products must be added with the proper weighting to yield the correct result, R3:R2:R1:R0.

```
                    P1 : P0  x
                    Q1 : Q0
                  ──────────────
                     Q0P0
                   P1Q0
                    Q1P0         +
                 P1Q1
                  ──────────────
                  R3:R2:R1:R0
```

Figure 4-4 **Double-Precision Multiplication** can be performed by a set of partial multiplications and additions
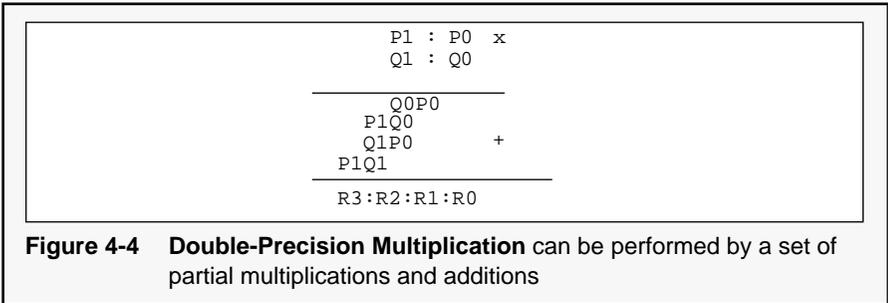
4.5 Double-Precision Multiplication of Fractions

The flowchart for the 48-bit general fraction multiplication routine, MULT48FG, is given in Figure 4-5. To compensate for the fact that signed multiplications are performed, a trick is used. The trick is to force bits 23 of P0 (P0(#23)) and Q0 (Q0(#23)) to

zero before performing the Q0P0, P1Q0, and Q1P0 multiplications and then to adjust the result if P0(#23) and/or Q0(#23) were set. The least significant word of the adjusted intermediate product, LS(IP), is concatenated with the least significant 48 bits of the result, R1:R0. IP is then shifted right by 24 bits to weight the MSB of IP correctly before performing the P1Q1 multiplication and accumulation. When A2 is moved in A1, A1 is sign extended. After the P1Q1 multiplication, A contains the sign-extended result, R3:R2. This routine executes in 27 cycles if both P0(#23) and Q0(#23) are set and in 26 cycles if both are zero. Listed in Figure 4-6, MULT48FG performs double-precision signed multiplication of fractions. Consider the multiplication of two 48-bit fractions stored in X and Y as shown in Table 4-4. The result is stored in accumulators A and B. The upper 48 bits of the 96-bit result are stored in A10, and the lower 48 bits are stored in B10 (see Table 4-4). A2 contains the sign extension of A10. The fractional result in decimal form is obtained after concatenating the two results, A0:B10, as indicated.

Table 4-4 Double-Precision Fractional Multiplication

Register	Hexadecimal Value	Fractional Value
X	345678 : FFFFFF	0.408888936042779
Y	006789 : 7FFFFF	0.003159701824181
A10	002A55 : CE41FA	
B10	9683FB : 000002	
A10:B10		0.001291967117102

4-8

MOTOROLA

zero before performing the Q0P0, P1Q0, and Q1P0 multiplications and then to adjust the result if P0(#23) and/or Q0(#23) were set. The least significant word of the adjusted intermediate product, LS(IP), is concatenated with the least significant 48 bits of the result, R1:R0. IP is then shifted right by 24 bits to weight the MSB of IP correctly before performing the P1Q1 multiplication and accumulation. When A2 is moved in A1, A1 is sign extended. After the P1Q1 multiplication, A contains the sign-extended result, R3:R2. This routine executes in 27 cycles if both P0(#23) and Q0(#23) are set and in 26 cycles if both are zero. Listed in Figure 4-6, MULT48FG performs double-precision signed multiplication of fractions. Consider the multiplication of two 48-bit fractions stored in X and Y as shown in Table 4-4. The result is stored in accumulators A and B. The upper 48 bits of the 96-bit result are stored in A10, and the lower 48 bits are stored in B10 (see Table 4-4). A2 contains the sign extension of A10. The fractional result in decimal form is obtained after concatenating the two results, A0:B10, as indicated.

Table 4-4 Double-Precision Fractional Multiplication

| Register	Hexadecimal Value	Fractional Value
X	345678 : FFFFFF	0.408888936042779
Y	006789 : 7FFFFF	0.003159701824181
A10	002A55 : CE41FA	
B10	9683FB : 000002	
A10:B10		0.001291967117102

Figure 4-5   MULT48FG Flowchart produces a double precision multiply of two fractions on the DSP56000

```
;MULT48FG.ASM
;This routine will execute the multiplication of two 48-bit FRACTIONAL numbers
;that are already stored in memory as follows.
;        x:$Paddr P1        y:$Paddr P0
;        x:$Qaddr Q1        y:$Qaddr Q0
;The initial 48-bit numbers are:
;       P  = P1:P0        (24:24 bits)
;       Q  = Q1:Q0        (24:24 bits)
;P0 with bit #23 forced to zero is P0'
;Q0 with bit #23 forced to zero is Q0'
;The result, R, is a 96 bit number that is stored in the two
;accumulators A and B as follows:
;       R  = R3:R2:R1:R0
;          = A10:B10 (48:48 bits)
;          = A:B10 (sign extended)
     move   #paddr,r4                ;initialize pointer for P
     move   #qaddr,r5                ;initialize pointer for Q
     move   #$7fffff,x1              ;load x1 with mask value
     move   y:(r4),a                 ;load A with P0
     and    x1,a        y:(r5),b     ;create P0';Q0 into B
     and    x1,b        a1,x0        ;create Q0';P0' into x0
     clr    a           b1,y0        ;clear A, Q0' into y0
     mpy    x0,y0,b     x:(r4),x1    ;mpy P0' with Q0', P1 into x1
     move   b1,a0                    ;most significant word MS (P0'Q0')to a0
     mac    x1,y0,a     x:(r5),y1    ;P1 * Q0' + a into a, Q1 into Y1
     mac    x0,y1,a                  ;P0'* Q1 + a into a
     jset   #23,y:(r4),one           ;P0(#23)= 1?
     jset   #23,y:(r5),two           ;Q0(#23)= 1?
     jmp    thr                      ;both P0(#23) and Q0(#23) = 0
one  add    y,a         #$000800,y0  ;adjust for P0(#23) = 1; load y0
     jclr   #23,y:(r5),thr           ;both P0(#23) and Q0(#23) = 1?
     mac    y0,y0,a                  ;generate cross term ($400000) and adj.
two  add    x,a                      ;adjust for Q0(#23) = 1 product
thr  move   a0,b1                    ;concatenate R1:R0 in B10
     move   a1,x0                    ;shift accumulator A 24 bits right
     move   a2,a                     ;and sign extend
     move   x0,a0                    ;interm. product (IP) weighted properly
     mac    x1,y1,a                  ;R3:R2 sign extended in A
     end                             ;end of routine
```

Figure 4-6   MULT48FG Routine implements a double precision multiply of two fractions on the DSP56000

4.6 Double-Precision Multiplication of Integers

Double-precision integer multiplication is the same as double-precision fractional multiplication except that an ASR instruction needs to be introduced in

the routine. The ASR eliminates the zero-fill bit and adds a sign-extension bit, thereby converting the fractional multiplier into an integer multiplier as discussed in **SECTION 4 Signed Multiplication**. The shift right is done in two stages since the result is 96 bits. The lower 48 bits are shifted first, which results in a zero in bit 47. The upper 48 bits are subsequently shifted right with bit 0 going to the carry bit. If the carry is set, a one is loaded into bit 47 of the lower 48 bits of the result. The double-precision multiplication is performed by the 48-bit general integer multiplication routine, MULT48IG, listed in Figure 4-7. An example is given in Table 4-5. The result of the multiplication is stored in the two accumulators. The 96-bit result can be obtained by concatenating A10 with B10 (see Table 4-5).

Table 4-5 Double-Precision Integer Multiplication

Register	Hexadecimal Value	Integer Value
X	000006 : 123456	101856342

NOTE: The A10:B10 concatenated result is 4.52943438572962E + 19.

4.7 Multiplication of a Real Number with a Real Number

When two real numbers are multiplied together, four 24-bit multiplications must be performed: one integer with an integer, one fraction with a fraction, and two fraction with an integer. Both the integer and the fractional parts must be in the signed twos-complement format. The result will be 96 bits long; the most significant 48 bits will be the integer part, and the least significant 48 bits will be the fractional portion.

To perform a real-number multiplication using the real multiply routine, REALMULT, the multiplicand, P, is stored in register X and the multiplier, Q, is stored in register Y (see Figure 4-8). The signed-integer portion of the real-number result, Ri, is stored in A10, and the unsigned fractional part, Rf, is stored in B10. The data ALU programmer's model for REALMULT is shown in Figure 4-9 An example is given in Table 4-6.

Register	Hexadecimal Value	Integer Value	Fractional Value
	Table 4-6 Real-Number Multiplication		
X1	00007B	123	
X0	600000		0.75
Y1	FFFFB1	-79	
Y0	B00000		-0.625
A10	FF : FFFFFF : FFD982		
B10	68 : 000000 : 000000		

NOTE: The A10:B10 concatenated result is -9853.59375.

```
;MULT48IG.ASM
;This routine will execute the multiplication of two 48-bit INTEGER numbers
;that are already stored in memory as follows.
;        x:$Paddr P1        y:$Paddr P0
;        x:$Qaddr Q1        y:$Qaddr Q0
;The initial 48-bit numbers are:
;        P = P1:P0    (24:24 bits)
;        Q = Q1:Q0    (24:24 bits)
;P0 with bit #23 forced to zero is P0'
;Q0 with bit #23 forced to zero is Q0'
;The result, R, is a 96 bit number that is stored in the two
;accumulators A and B as follows:
;        R = R3:R2:R1:R0
;          = A10:B10 (48:48 bits)
;          = A:B10 (sign extended)
    move   #paddr,r4              ;initialize pointer for P
    move   #qaddr,r5              ;initialize pointer for Q
    move   #$7fffff,x1            ;load x1 with mask value
    move   y:(r4),a               ;load A with P0
    and    x1,a       y:(r5),b    ;create P0';Q0 into B
    and    x1,b       a1,x0       ;create Q0'; P0' into x0
    clr    a          b1,y0       ;clear A, Q0' into y0
    mpy    x0,y0,b    x:(r4),x1   ;mpy P0' with Q0', P1 into x1
    move   b1,a0                  ;most significant word MS (P0'Q0')to a0
    mac    x1,y0,a    x:(r5),y1   ;P1 * Q0' + a into a, Q1 into Y1
    mac    x0,y1,a                ;P0'* Q1 + a into a
    jset   #23,y:(r4),one         ;P0(#23)= 1?
    jset   #23,y:(r5),two         ;Q0(#23)= 1?
    jmp    thr                    ;both P0(#23) and Q0(#23) = 0
one add    y,a        #$000800,y0 ;adjust for P0(#23) = 1; load y0
    jclr   #23,y:(r5),thr         ;both P0(#23) and Q0(#23) = 1?
    mac    y0,y0,a                ;generate cross term ($400000) and adj.
two add    x,a                    ;adjust for Q0(#23) = 1 product
thr move   a0,b1                  ;concatenate R1:R0 in B10
    move   #0,b2                  ;clear the extension register B2
    asr    b                      ;start adjusting the product to integer
    move   a1,x0                  ;shift accumulator A 24 bits right
    move   a2,a                   ;and sign extend
    move   x0,a0                  ;interm. product (IP) weighted properly
    mac    x1,y1,a                ;R3:R2 sign extended in A
    asr    a                      ;finish adjusting the product to integ.
    jcc    end                    ;finished if A(#0) is 0
    move   #$800000,x0            ;if A(#0) is 1
    or     x0,b                   ;set B10 (#47)
    end                           ;end of routine
```

Figure 4-7   MULT48IG Routine which multiplies two integer numbers on the DSP56000

```
;REALMULT.ASM
;
;This routine multiplies two signed real numbers P and Q. It assumes the signed
;integer part of P(Pi), is in X1 and the signed fractional part (Pf), is in X0,
;the signed integer part of Q(Qi), is in Y1 and the signed fractional part of Q(Qf)
;is in Y0. The signed integer part of the result, Ri, is stored in A10
;and the unsigned fractional part, Rf, is stored in B10.
;
;       mpy  x0,y0,b          ;Pf * Qf
        asl  b                ;remove the sign bit from the product
        move b2,a             ;shift PfQf product 24 bits right
        move b1,a0            ;and preload accumulator A
        mac  x1,y0,a          ;mult. Pi with Qf and accumulate in A
        mac  x0,y1,a          ;mult. Pf with Qi and accumulate in A
        move a0,b1            ;concatenate MS (RF) with LS (RF)
        move a1,a0            ;adjust weighting by shifting interm.
        move a2,a1            ;product right 24 bits in prep. for
        mac  x1,y1,a          ;final product Pi * Qi + A into A
        asr  a               ;eliminate the zero fill bit
```

Figure 4-8   REALMULT Routine which multiplies two signed real numbers on the DSP56000

Signed P Integer = P_i		Signed P Fraction = P_f
X1		X0
Signed Q Integer = Q_i		Signed Q Fraction = Q_f
Y1		Y0

Unsigned Fractional Part of R, R_f

A10

Signed Integer Part of R, R_i

B10
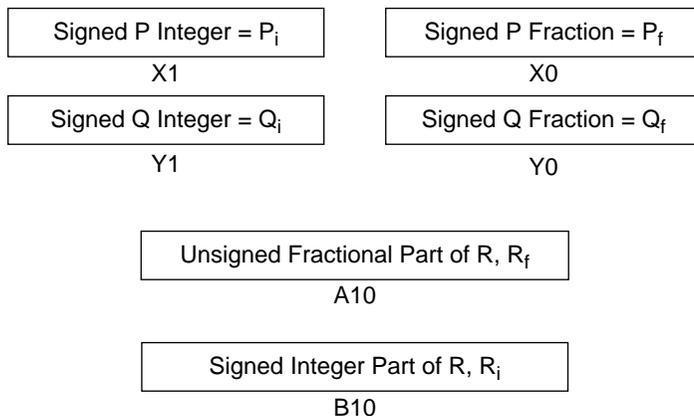
Figure 4-9   REALMULT Data ALU Programmer's Model
Accumulators A10 and B10 hold a single 96 bit mixed-number result.

This routine is similar to MULT48FG or MULT48IG in that the interim products must be properly weighted to yield the correct result. Unlike the MULT48FG and MULT48IG routines, however, there are no adjustment terms to consider because the fractions, Pf and Qf, are assumed to be signed. The CONVSISF macro in Figure 4-3 will perform the conversion of a real number into a signed integer and signed fraction.

The 96-bit result from the REALMULT routine should be treated as an entity. If the positive value of a negative result is required, then the absolute value of the whole 96 bits should be obtained before the integer part and fractional part can be separated.

Table 4-7 shows a second example using two negative numbers that produce a positive result.

Table 4-7 Multiplication of Two Negative Real Numbers			
Register	**Hexadecimal Value**	**Integer Value**	**Fractional Value**
X1	FFFFBF	-65	
X0	933334		-0.85
Y1	FFFFE9	-23	
Y0	ECCCCD		-0.15
A10	000000 : 0005F4		
B10	6D7064 : 75C290		

4.8 Multiplication of a Mixed Number with a Mixed Number

Assume that the mixed numbers are stored in the MN format. Multiplying two mixed numbers is simply a multiplication using the MPY instruction or the MAC instruction for a multiply and accumulate. The multiplication will be a 48-bit result, which will be in the format shown in Figure 4-10 only after a one-bit right shift to compensate for the zero-fill bit introduced by the fractional multiplication. After this shift, the most significant 16 bits will be the signed integer part, and the least significant 32 bits will be the unsigned fractional part.

Consider the example given in Table 4-8. If the result is desired in the real-number format, where the integer part is separated from the unsigned fractional part, then an additional right shift by eight bits must be performed on the product. This shift can be performed using the REP instruction or the appropriate shift multiplier as shown in **SECTION 2.5 Data Shifting**. By performing the shift, the integer part of the product is stored in the most significant word of the accumulator, and the unsigned fractional part is stored in the least significant word of the accumulator.

If the magnitude of the result is less than 128, it can be stored back in a 24-bit register in the MN format, which is performed by a left shift by eight bits on the result in accumulator A (already shifted one bit to the right, implying a net shift of seven bits left). The result will reside in A1 as shown in Table 4-8.
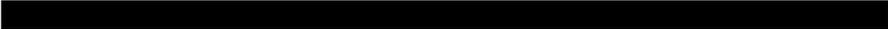
Table 4-8 Multiplication of Two Mixed Numbers

Register	Hexadecimal Value	Mixed-Number Value
X1	068000	6.5
X0	044000	4.25
A	00 : 003740 : 000000	
A10[1]	1BA000 : 000000	27.625
A10[2]	0001BA : 000000	27.625

1. In the MN format after the left shift by seven bits net.
2. In the real-number format after the right shift by nine bits net.

s i i i i i i i . f f f f f f f f f f f f f f f f

s i i i i i i i . f f f f f f f f f f f f f f f f

Two 24-bit Mixed Numbers

47   32   31   0

| Signed Integer Part | · | Unsigned Fractional Part |

Figure 4-10 Multiplication of Two Mixed Numbers

■
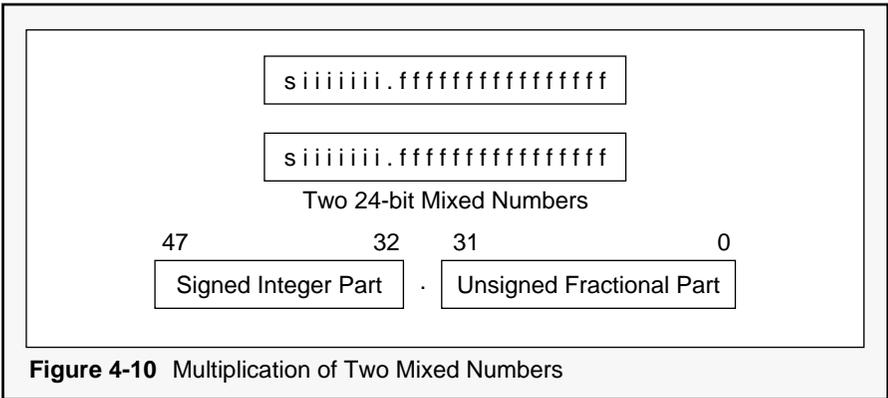
SECTION 5

Signed Division

Even though division is the inverse mathematical process of multiplication, it differs from multiplication in many aspects. Division is a shift and subtract divisor operation in contrast to multiplication, which is a shift and add multiplicand operation. In division, the results of one subtraction determine the next operation in the sequence; thus, division is inherently iterative and data dependent. The answer consists of a quotient and a remainder, both of which can have variable word lengths. In multiplication, the number of bits in the product is known a priori, which means that division is not a deterministic process, but rather a trial-and-error process. This fact makes implementing divide routines a challenge. There are, however, additional data- and hardware-related factors that must be considered, such as:

- Input Data
  Signed Unsigned
  Integer
  Fractional Normalized
  Fractional Unnormalized

- Output Requirements
  Quotient
  Remainder
  Quotient with Remainder
  Magnitude Only
  Signed
  Number of Bits of Accuracy
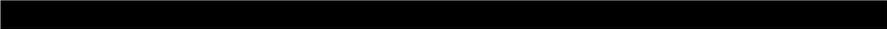
- Machine Architecture
    Fractional
    Integer
    Register Structure

The instruction set of the DSP56000/DSP56001 processors includes a divide iteration instruction, DIV. Execution of a DIV generates one quotient bit using a nonrestoring algorithm on signed fractional operands. The original dividend must occupy the low-order 48 bits of the destination accumulator and must be a positive number. Also, the divisor must be larger than the dividend so that a fractional quotient is generated. After the first DIV execution, the destination accumulator holds both the partial remainder and the formed quotient.

The partial remainder, which occupies the high-order portion of the destination operand, is a signed fraction. The partial remainder is not a true remainder and must be corrected before it may be used because of the nonrestoring nature of the division algorithm. Therefore, once the divide is complete, it is necessary to reverse the last DIV operation to restore the remainder, if the true remainder is desired.

The formed quotient, which occupies the low-order portion of the destination accumulator, is a signed fraction. One bit of the formed quotient is shifted into the LSB of the destination accumulator for each DIV execution. Thus, portions of the destination accumulator allocated to the remainder and to formed quotients depend on the number of DIV executions.
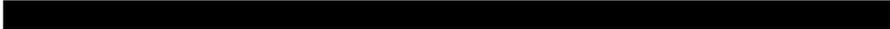
In summary, for the division to produce the correct results on the DSP56000/DSP56001, two conditions must be satisfied:

1. the dividend must be positive and sign extended

2. the magnitude of the divisor must be greater than the magnitude of the dividend so that a fractional quotient is generated except for integer division.

5.1 Division of a Signed Fraction by a Signed Fraction

The signed 24-bit divide routine for a four-quadrant divide (i.e., a signed divisor and a signed dividend) that generates a 24-bit signed quotient and a 48-bit signed remainder, SIG24DIV, is given in Figure 5-1. The dividend is assumed to be in X0, but could have been in X1, Y1, or Y0.

The first three instructions save the appropriate sign bits and ensure that the dividend is positive. The first instruction copies A1 to B1 so that the sign bit of the dividend, bit 47 of A, is saved in B1 prior to taking the absolute value of the dividend. The exclusive OR in the second instruction will result in the N bit in the status register being set if the signs of the divisor and the dividend are different. Since the DIV instruction does not affect the N bit, the N bit represents the sign of the final quotient.

```
;SIG24DIV.ASM
;This is a routine for a 4 quadrant divide (i.e., a signed divisor and a signed
;dividend) which generates a 24-bit signed quotient and a 48-bit signed
;remainder. The quotient is stored in the lower 24 bits of accumulator A,A0,
;and the remainder in the upper 24 bits, A1. The true (restored) remainder is
;stored in B1. The original dividend must occupy the low order 48 bits of the
;destination accumulator, A, and must be a POSITIVE number. The divisor (x0)
;must be larger than the dividend so that a fractional quotient is generated.
;The quotient will be in x1 and the remainder will be in B1.
;
          abs  a          a,b          ;make dividend positive, copy A1 to B1
          eor  x0,b       b,x:$0       ;save rem. sign in x:$0, quo, sign in N
          and  #$fe,ccr                ;clear carry bit C (quotient sign bit)
          rep  #$18                    ;form a 24-bit quotient
          div  x0,a                    ;form quotient in A0, remainder in A1
          tfr  a,b                     ;save remainder and quotient in B1,B0
          jpl  savequo                 ;go to SAVEQUO if quotient is positive
          neg  b                       ;complement quotient if N bit is set
savequo   tfr  x0,b       b0,x1        ;save quo. in X1, get signed divisor
          abs  b                       ;get absolute value of signed divisor
          add  a,b                     ;restore remainder in B1
          jclr #23,x:$0,done           ;go to DONE if remainder is positive
          move #$0,b0                  ;prevent unwanted carry
          neg  b                       ;complement remainder
done                                   ;end of routine
```

Figure 5-1   SIG24DIV Routine
This routine is a four quadrant divide that produces a 24-bit signed
quotient and a 48-bit signed remainder.

The sign of the remainder is the sign of the divi-
dend; therefore, bit 23 of B1 contains the sign of the
remainder. B1 is stored in data memory as well as
in the second instruction so that the sign bit can be
tested using the bit manipulation instructions later in
the routine. The third instruction clears the carry bit,
C, in the condition code register. This fact ensures
the quotient will be positive because the C bit is al-
ways the next quotient bit and because the C bit is
shifted into the accumulator at the beginning of the
execution of the DIV instruction.
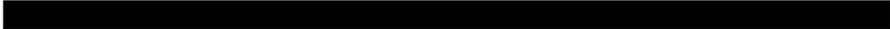
Execution of the next two instructions, REP and DIV, generates the 24-bit quotient and 48-bit remainder (the first 24 bits of which will be zero). The transfer instruction, TFR A,B, copies the quotient A0 into B0 (also A1 into B1 and A2 into B2) so that the sign of the quotient (i.e., bit 23 of B0) can be corrected. If the N bit in the status register was set, B is complemented. The only purpose is to complement B0 at this point; bit 23 of B0 is zero prior to the negation. Therefore, B0 is a valid, signed quotient that is saved in X1. The divisor in X0 is copied into B1 so that its absolute value can be generated and used to restore the remainder. If the remainder needs negating, B0 must be cleared first to prevent an unwanted carry from propagating into B1, the true remainder.

In Table 5-2 the contents of the A accumulator are shown after each iteration of the DIV instruction for the case shown in Table 5-1; the repeat instruction is not interruptible. **SECTION 5.6 Divide Routines With N $\leq$ 24 Bits** contains four divide routines that generate quotients having less than 24 bits of precision. In Table 5-1 the true remainder in B1 is shifted right by 24 bits because the quotient is 24 bits, and the remainder is always smaller than the quotient. The 24 bits are implied and are not in a register.

If the dividend is greater than the divisor, the dividend must be scaled down to be smaller than the divisor. The quotient must then be scaled up before being output, or it must be output directly and

interpreted correctly. This interpretation involves assuming the binary point has moved to compensate for the original downscaling — that is, the quotient will now be a real number. If it can be guaranteed that the divisor and dividend are normalized, then faster quadratic convergence and reciprocal methods can be used to calculate the quotient.

Table 5-1 Signed-Fraction Division		
Register	**Hexadecimal Value**	**Fractional Value**
X0 (Divisor)	600000	0.75
A (Dividend)	00 : 300000 : 000000	0.375
A0 (Quotient)	400000	0.500
A1 (Remainder)	A00000	
$000000:B1 (True Remainder)	000000 : 000000	0.0

Table 5-2 Contents of Accumulator After Signed-Fraction
Division Iterations

DIV Iteration	Contents of Accumulator A (in HEX)		
	A2	A1	A0
1	00	000000	000000
2	FF	A00000	000001
3	FF	A00000	000002
4	FF	A00000	000004
5	FF	A00000	000008
6	FF	A00000	000010
7	FF	A00000	000020
8	FF	A00000	000040
9	FF	A00000	000080
10	FF	A00000	000100
11	FF	A00000	000200
12	FF	A00000	000400
13	FF	A00000	000800
14	FF	A00000	001000
15	FF	A00000	002000
16	FF	A00000	004000
17	FF	A00000	008000
18	FF	A00000	010000
19	FF	A00000	020000
20	FF	A00000	040000
21	FF	A00000	080000
22	FF	A00000	100000
23	FF	A00000	200000
24	FF	A00000	400000

5.2 Division of a Signed Integer with a Signed Integer

Integer division can be treated in the same manner as fractional division. That is, if the dividend is positive and smaller in magnitude than the divisor, executing the SIG24DIV routine will generate the correct results. However, since the remainder is not used in integer division (the remainder is truncated), SIG24DIV can be shortened for use with integer division. Consider the example using SIG24DIV shown in Table 5-3. The contents of the A accumulator after each DIV iteration are shown in Table 5-4.

The quotient is stored in the lower 24 bits, A0, of accumulator A. The value \$1BD178 (0.21732998 decimal) is the quotient. A1 will contain the lower 24 bits of the 48-bit true remainder after the addition of the absolute value of the divisor. In this example, B1=\$0018E0 after the remainder has been restored. Therefore, the true remainder is \$000000:0018E0 or 0.000000000045247 decimal.

Table 5-3 Signed-Integer Division

Register	Hexadecimal Value	Fractional Value
X0 (Divisor)	600162E	5678
A (Dividend)	00 : 0004D2 : 000000	1234
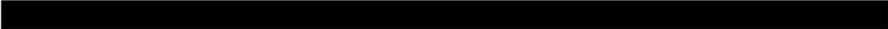
**Table 5-4** Contents of Accumulator After Each Signed-Integer Division Iteration

DIV Iteration	Contents of Accumulator A (in HEX)		
	A2	A1	A0
1	FF	FFF376	000000
2	FF	FFFD1A	000000
3	00	001062	000000
4	00	000A96	000001
5	FF	FFFEFE	000003
6	00	00142A	000006
7	00	001226	00000D
8	00	000E1E	00001B
9	00	00060E	000037
10	FF	FFF5EE	00006F
11	00	00020A	0000DE
12	FF	FFEDE6	0001DB
13	FF	FFF1FA	00037A
14	FF	FFFA22	0006F4
15	00	000A72	000DE8
16	FF	FFFEB6	001BD1
17	00	00139A	0037A2
18	00	001106	006F45
19	00	000BDE	00DE8B
20	00	00018E	01BD17
21	FF	FFECE	037A2F
22	FF	FFF00A	06F45E
23	FF	FFF642	0DE8BC
24	FF	0002B2	1BD178

MOTOROLA

Shown in Figure 5-2, the INTDIV macro performs a
signed-integer divide without the extra instructions
that SIG24DIV uses to generate the remainder.
This routine reduces the number of operative in-
structions by about one-half.

```
integer divide macro.
 Registers used: a,b,x0
 Input: macro pass parameters "dividend, divisor"
 Output: Quotient --> a0

 macro dividend, divisor
 move #>dividend, a            ;sign extend a2
 move a2,a1                    ;and A1
 move #>dividend,a0            ;move the dividend into A
 asl a   #>divisor,x0          ;prepare for divide, and
                              ;move divisor into x0 (24 bit)
 abs a   a,b                  ;make dividend positive, save in B
 and #$fe, ccr                ;clear the carry flag
 rep #$18                     ;form a 24-bit quotient
 div x0,a                     ;for quotient in a0, remainder a1
 eor x0,b                     ;save quotient sign in N
 jpl done                     ;go to done if quotient is positive
 neg a                        ;complement quotient if N bit is set
 nop                          ;finished, the quotient is in a0
 endm
```

-2   INTDIV Routine
    This routine is a four quadrant divide that produces a 24-bit signed
    quotient with no remainder.

5.3 Double-Precision Division

Division of a 48-bit number by another 48-bit num-
ber is not possible using the REP,DIV instruction
sequence because the divisor is restricted to be a
24-bit fraction. Therefore, to perform double-preci-
sion division producing a 48-bit quotient and 96-bit
remainder, the DIV48 routine is introduced. This
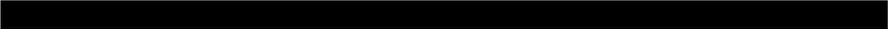
routine implements the nonrestoring divide algorithm that the DIV instruction implements in hardware. The flowchart for this algorithm is shown in Figure 5-3.

Figure 5-3   Double-Precision Divide Flowchart
This chart describes the division of two signed 48-bit numbers
that is performed by the DIV48 routine.
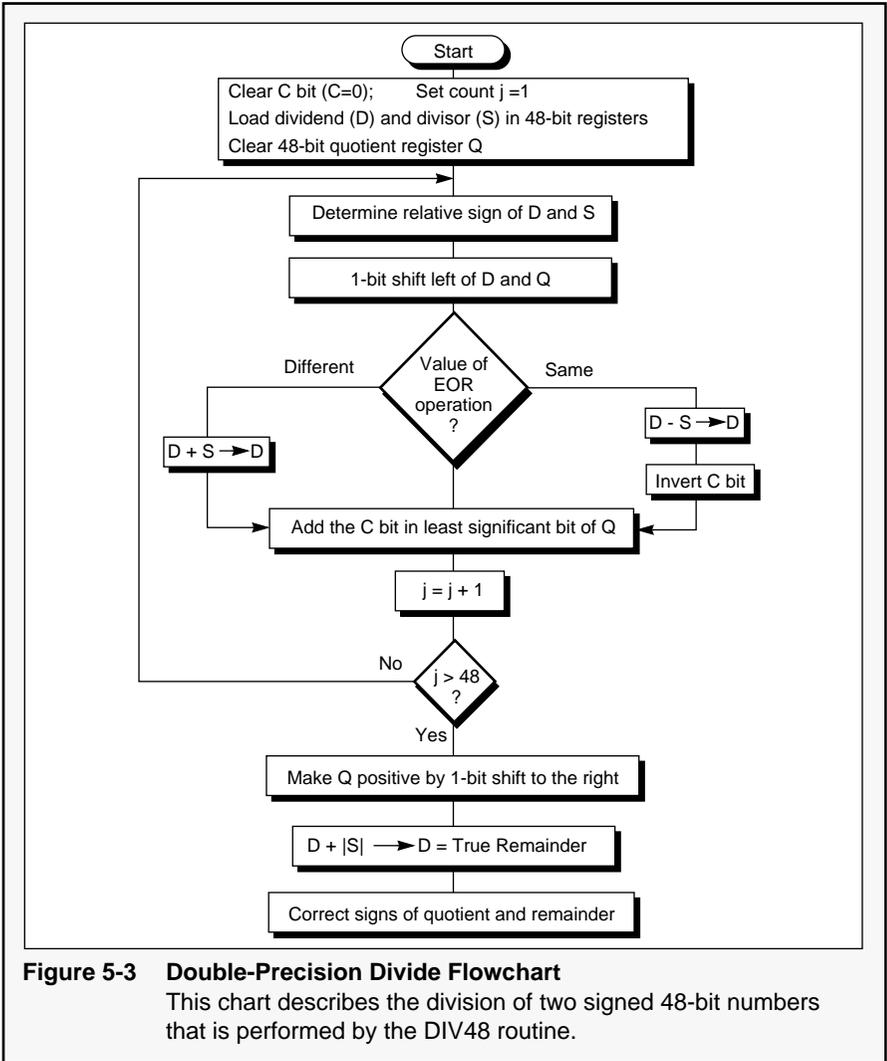
The DIV48 dividend is stored in B10; the divisor is
stored in X; and the quotient is developed in A10. After
iterating the loop 48 times, B10 will contain the least sig-

nificant 48 bits of the 96-bit remainder. The sign of the remainder must be made the same as that of the dividend. The quotient will be negative if the signs of the dividend and the divisor are different. The borrow bit generated as the result of a subtraction (i.e., N1-N2) is the complement of the carry bit generated as the result of the addition of the negative (i.e., N1+(-N2)) of an operand. Therefore, the carry bit must be inverted if the divisor is subtracted from the dividend. The carry is introduced into the quotient using the add long with carry (ADC) instruction with one of the addends, Y, equal to zero. Consider the example in Table 5-5. The result, which is stored in accumulator A, is obtained using the DIV48 routine shown in Figure 5-4.

Table 5-5 Double-Precision Division			
Register	**Decimal Hexadecimal Value**	**Integer Value**	**Decimal Fractional Value**
X (Divisor)	000078 : 123450	+2014458960	0.000014313591805
B10 (Dividend)	00000F : 02468A	+251807370	0.000001789198976
A10 (Quotient)	100000 : 000000		0.125
B10 (Remainder before Correction)	FFFF87 : EDCBB0		
B10 (True Remainder after Correction)	000000 : 000000 000000 : 000000	0	0.0

```
;DIV48.ASM
;This routine performs double precision division on two 48-bit operands.
;The operands can be either fractions or integers. The dividend must be positive
;the magnitude of the divisor must be greater than the magnitude of the
;dividend. The dividend and divisor are assumed to be in memory as
;         L:$divaddr     dividend
;         L:$divaddr+1   divisor
;         L:$divaddr+2   quotient
;The dividend is loaded in the long word operand B10 and the divisor is loaded
;in the long operand word x. The 48-bit true remainder is stored in B.
;
        move   #divaddr,r0      ;initialize r0
        move   #0,y0            ;clear y0
        move   r0,r1            ;initialize r1
        and    #$fe,ccr         ;clear carry bit C
        move   1:(r0)+,b        ;load dividend into B
        abs    b   1:(r0)+,x    ;make the dividend positive, divisor in X
        clr    a   #0, y1       ;clear a and y1
        do     #48,endloop      ;execute the loop 48 times
        eor    x1,b             ;do operands have the same sign?
        jmi    opp              ;if opposite sign jump to location one
        eor    x1,b             ;restore the value of the operand in B
        asl    a                ;prepare A to receive a quotient bit
        asl    b                ;multiply the dividend by 2
        sub    x,b              ;subtract the divisor from the dividend
        move   sr,x:$0          ;process to invert the carry bit
        bchg   #0,x:$0          ;invert the carry bit
        move   x:$0,sr          ;restore the SR with inverted carry bit
        jmp    quo              ;jump to location "quo"
opp     eor    x1,b             ;operands have opposite sign
        asl    a                ;prepare A to receive a quotient bit
        asl    b                ;multiply the dividend by 2
        add    x,b              ;add the divisor to the dividend
quo     adc    y,a              ;add the carry bit to develop quotient
endloop                         ;end of the 48 iteration loop
        asr    a                ;introd. The positive sign bit in quotient
        tfr    x1,a   a,1:(r0)  ;divisor in A, save quotient
        move   x0,a0            ;lower part of divisor in A
        abs    a                ;get the absolute value of divisor
        add    a,b              ;generate lower 48 bits of true remainder
        jclr   #23,x:(r1),done  ;if dividend is positive, finished
        neg    b                ;if dividend is negative, negate remainder
done    nop                     ;end of routine
```

Figure 5-4   DIV48 Routine
This routine divides a 48-bit signed number by a second 48-bit
signed number, and produces a 48-bit quotient and 48-bit remainder.

5.4 Real-Number Division

This type division is not possible by simply using the DIV instruction repeatedly because a real number is a 48-bit number, and the DIV operation only operates on 24-bit operands. A real-number division cannot be broken down into four divisions and subtractions like the real-number multiplication because division is nondeterministic. One way to divide two real numbers is to scale the numbers to form integers, fractions, or mixed numbers by multiplying them by the appropriate constant. Consider the following example:

$$\frac{123.750}{837.875} = 0.14769506191257$$

To perform the previous division using the DSP56000/DSP56001, multiply both numbers by 1000 to change them to integers. The new values with the results after executing the SIG24DIV routine are shown in Table 5-6.

The second and most accurate method of real-number division is by using the DIV48 routine shown in Figure 5-4. The real number is kept in the real-number format and stored according to the requirements of the routine. The values used are shown in Table 5-7. Although the result is more accurate than the result obtained by the first method, it is slower than the first method.

Table 5-6 Result of Real-Number Division Using DIV24 Routine

Register	Hexadecimal Value	Integer Value	Fractional Value
X1 (Divisor)	0CC8F3	837875	

Table 5-7 Result of Real-Number Division Using DIV48 Routine

Register	Hexadecimal Value	Real Number	Fractional Value
X (Divisor)	000345 : E00000	+837.875	
B10 (Dividend)	00007B : C00000	+123.750	
A10 (Quotient)	12E7AB : FA58FC		0.147695061912572
B10 (Remainder)	000129 : 200000		
True Remainder	000000 : 000000 00046F : 000000		4.8069E-19

5.5 Mixed-Number Division

The mixed number is stored in the appropriate register or accumulator in the MN format. Since the number is represented in 24 bits, the SIG24DIV routine can be used. The mixed-number division must satisfy the same two conditions as the fractional division. Since the magnitude of the divisor must be greater than the magnitude of the dividend, the quotient will be a fraction represented in the signed-fraction format.

Consider the example shown in Table 5-8. After executing the SIG24DIV routine, the quotient will be in A0, and the lower part of the true remainder will be in B1. The results obtained after the execution of each DIV instruction are given in Table 5-9.

Table 5-8 Result of Mixed-Number Division Using SIG24DIV Routine

| Register	Hexadecimal Value	Decimal Value
X0 (Divisor)	3FC000	63.75
A (Dividend)	00 : 188000 : 000000	24.50
A0 (Quotient)	313131	0.3843137
A1 (Remainder)	D8C000	
$000000:B1 (True Remainder)	000000 : 188000	0.000000011408702
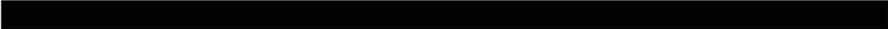
Table 5-9 Contents of Accumulator After Each Mixed-Number Division Iteration

DIV Iteration	Contents of Accumulator A (in HEX)		
	A2	A1	A0
1	FF	F14000	000000
2	00	224000	000000
3	00	04C000	000001
4	FF	C9C000	000003
5	FF	D34000	000006
6	FF	E64000	00000C
7	00	0C4000	000018
8	FF	D8C000	000031
9	FF	F14000	000062
10	00	224000	0000C4
11	00	04C000	000189
12	FF	C9C000	000313
13	FF	D34000	000626
14	FF	E64000	000C4C
15	00	0C4000	001898
16	FF	D8C000	003131
17	FF	F14000	006262
18	00	224000	00C4C4
19	00	04C000	018989
20	FF	C9C000	031313
21	FF	D34000	062626
22	FF	E64000	0C4C4C
23	00	0C4000	189898
24	FF	D8C000	313131

5.6 Divide Routines with N ≤ 24 Bits

Four distinct routines for the division of fractional numbers where an N-bit (N<24) quotient is required are given in the following paragraphs.

5.6.1 Positive Operands with Remainder Where N Is Variable

For positive fractional operands, the code in Figure 5-5 may be used to perform a divide operation, which generates an N-bit quotient and a 48-bit remainder having 48 N bits of precision for N<24.

In this routine, the quotient is built up by rotating the C bit into B. The correct C bit is generated by executing the DIV instruction. The remainder is built up in A. The REP Y1 and ASL B instruction sequence sets the signed-fraction format for the N-bit quotient into a signed fraction. Similarly, the REP Y0 and ASR B instruction sequence formats the 48-N-bit true remainder into a signed fraction.

5.6.2 Positive Operands without Remainder Where N Is Fixed

For positive fractional operands, the code in Figure 5-6 (or similar code) may be used to perform a divide operation yielding only an N-bit quotient without a remainder for N<24.

The quotient bits must be extracted out of the accumulator that contains both remainder and quotient bits after the execution of the DIV instructions. These bits must then be formatted as a positive fraction.

```
;This routine assumes that the 48-bit positive fractional dividend
;is stored in the A accumulator, the 24-bit positive fractional divisor
;stored in the X0 register, the value N is stored in the Y0 register an
;value 24-N is stored in the Y1 register. This routine stores the N-bit p
;fractional quotient in the X1 register and the 48-bit positive fractio
;remainder with 48-N bits of precision in the B accumulator. Note that
;routine the value of N and 24-N may be changed at run time without reass
;since they are stored in registers.
;
        clr  b                ;initialize B1 for quotient
        and  #$fe, ccr        ;clear carry, C, (quotient sign bit)
        do   y0,loop1         ;compute N-bit quotient (Y0=N)
        rol  b                ;build up N-bit quotient in B1
        div  x0,a             ;build up 48-N bit remainder in A
loop1   rep  y1               ;repeat 24-N times (Y1=24-N)
        asl  b                ;format quotient as positive fraction
        tfr  x0,b    b1,x1    ;save N-bit quotient, move divisor
        add  a,b              ;recover 48-N bit remainder in B
        rep  y0               ;repeat N times (Y0=N)
        asr  b                ;format remainder as positive fraction
```

Figure 5-5   Positive Divide: 48-Bit Operand and Remainder

```
;This routine assumes that the 48-bit positive fractional dividend is st
;in the A accumulator and that the 24-bit positive fractional divisor is
;in the X0 register. This routine stores the N-bit positive fractional qu
;in the A accumulator, A1. The value of N is not stored in a register an
;specified at the time of compilation using the CVI and POW functions buil
;the DSP56000 Cross Assembler. The quotient is stored in A1 and the rema
;is destroyed.
;
        and  #$fe,ccr                ;clear carry, C, (quotient sign bit
        rep  #n                      ;form an N-bit quotient
        div  x0,a                    ;perform divide iteration N times
        move a0,1                    ;move quotient to A1, destroy rema
        move #>(@cvi(@pow(2,n))-1,x1 ;store N-bit quotient bit mask in
        and  x1,a                    ;extract N-bit quotient in A1
        rep  #(24-n)                 ;repeat (24-N) times
        lsl  a                       ;format quotient as positive fracti
```

Figure 5-6   Positive Divide: N-Bit Quotient without Remainder

5.6.3 Signed Operands with Remainder Where N Is Variable

For signed fractional operands, the code shown in Figure 5-7 may be used to perform a divide operation yielding an N-bit quotient and a 48-bit remainder having 48 N bits of precision for N<24. Bits 0 and 1 in location X:$0 are used to save the quotient and remainder sign flags, respectively.

```
;This routine assumes that the sign extended 48-bit fractional dividend
;is stored in the A accumulator, the 24-bit signed divisor is stored in the
;X0 register, the value N is stored in the Y0 register and that the value
;24-N is stored in the Y1 register. This routine stores the N-bit signed
;fractional quotient in the X1 register and the 48-bit positive fractional
;remainder with 48-N bits of precision in the B accumulator. In this routine the
;values of N and 24-N may be changed at run time.
;
          bclr #0,x:$0            ;clear quotient sign flag (bit 0,x:$0)
          bclr #1,x:$0            ;clear remainder sign flag (bit 1,x:$0)
          tst a                   ;determine the sign of the dividend
          jpl signquo             ;go to SIGNQUO if the dividend is positive
          bset #1,x:$0            ;set remainder sign flag if negative
signquo   abs a    a,b            ;make dividend positive, copy a1 to b1
          eor x0,b                ;get sign of quotient (N bit)
          jpl start               ;go to START if the sign is positive
          bset #o,x:$0            ;set quotient sign flag if negative
start     clr b                   ;initialize B1 for quotient
          and #$fe,ccr            ;clear carry, C, (quotient sign bit=0)
          do  y0, loop1           ;compute N-bit quotient (Y0=N)
          rol b                   ;build up N-bit quotient in B1
          div x0,a                ;build up 48-N bit remainder in A
loop1     rep y1                  ;repeat 24-N times (Y1=24-N)
          asl b                   ;format quotient as positive fraction
          jclr #0,x:$0,savequo    ;go to SAVEQUO if the quot. is positive
          neg b                   ;complement quot. if sign flag is set
savequo   tfr x0,b b1,x1          ;save N-bit quotient, divisor into B
          abs b                   ;get the absolute value of divisor
          add a,b                 ;recover 48-N bit remainder in B
          rep y0                  ;repeat N times (Y0=N)
          asr b                   ;format remainder as signed fraction
          jclr #1,x:$0,done       ;go to DONE if the remainder is positive
          neg b                   ;complement remainder if negative
done
```

Figure 5-7   Signed Divide: 48-Bit Operand and Remainder

The first function of this routine is to set up these flags. The quotient is built up by rotating the C bit into B. The correct C bit is generated by executing the DIV instruction. The remainder is built up in A. The REP Y1 and ASL B instruction sequence formats the N-bit quotient as a signed fraction. Similarly, the REP Y0 and ASR B instruction sequence formats the 48 N-bit true remainder into a signed fraction.

5.6.4 Signed Operands without Remainder Where N Is Fixed

For signed fractional operands, the code given in Figure 5-8 may be used to perform a divide operation, yielding only an N-bit quotient without a remainder for N<24.

```
;This routine assumes that the sign extended 48-bit fractional dividend
;is stored in the A accumulator and that the 24-bit signed fractional di
;is stored in the X0 register. This routine stores the N-bit signed frac
;quotient in the A accumulator. The value of N is not stored in a regist
;and is specified at the time of compilation using the CVI and POW funct
;built into the DSP56000 Cross Assembler. The quotient is stored in A1
;and the remainder has been destroyed.
;
      abs  a    a,b              ;make dividend positive, copy signed div
      eor  x0,b                  ;get sign of quotient and save in N bit
      and  #$fe,ccr              ;clear carry (quotient sign bit = 0)
      rep  #n                    ;form an N-bit quotient by executing
      div  x0,a                  ;the divide iteration N times
      jpl  mask                  ;go to MASK if quotient is positive
      neg  a                     ;negate quotient
mask move a0,a1                  ;destroy remainder, move quotient in A1
      move #>(@cvi(@pow(2,n))-1,x1;N-bit quot. mask in X1
      and  x1,a                  ;recover signed N-bit quotient in A1
      rep  #24-n                 ;repeat 24-N times
      lsl  a                     ;format quotient as signed fraction
```

Figure 5-8   Signed Divide: N-bit Quotient with Remainder
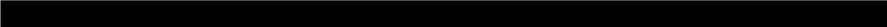
The NEG A instruction is guaranteed to negate the quotient because the first bit of the quotient has been set to zero, making it a positive fraction. Quotient bits must be extracted out of the accumulator, which contains both remainder and quotient bits after the execution of the DIV instructions. The quotient bits must then be formatted as a positive fraction. ∎

SECTION 6

Conclusion

". . . unless an application specifically demands using less than 24 bits of precision, the use of an unsigned full-precision division routine will probably result in the fastest division execution time."

For the case in which only positive fractional operands are used to compute both a full-precision (i.e., 24-bit) unsigned quotient and its 48-bit remainder, both the quotient and the remainder are correctly aligned with their respective register boundaries after the 24 divide iterations. Neither the quotient nor the remainder must be shifted to produce the correct result (see **SECTION 5.1 Division of a Signed Fraction by a Signed Fraction**). In general, unless an application specifically requires the number of bits of the precision (N) in the quotient to be variable, using a fixed value N, declared at the time of assembling, results in a significantly faster division execution time. Similarly, unless an application specifically requires that the remainder has to be computed, using a routine that computes only the quotient results in a significantly faster division execution time. Finally, unless an application specifically demands using less than 24 bits of precision, the use of an unsigned full-precision division routine will probably result in the fastest division execution time. ∎
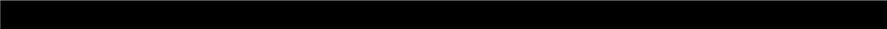
REFERENCES

1. DSP56000 Macro Assembler Reference Manual (available from DSP, Oak Hill, Texas, (512) 440-2030), Motorola Inc., 1986.

2. DSP56000 Digital Signal Processor User's Manual (DSP56000UM/AD), Motorola Inc., 1986.

3. DSP56001 56-Bit General-Purpose Digital Signal Processor (ADI1290), Motorola Inc., 1988. ∎