Application Report
SLAA329A–September 2006–Revised July 2018

Efficient Multiplication and Division Using MSP430™ MCUs

MSP430 Applications

ABSTRACT

Without a hardware multiplier, multiplication and division require many instruction cycles, especially in C code. This report describes a method to perform multiplication and division with only shift and add instructions without a hardware multiplier. The method is based on Horner's method. The MSP430™ microcontroller (MCU) can perform a register shift or add in one instruction cycle. This allows fast execution of multiplications and divisions using Horner's method. This method not only produces accurate results but also provides a very good dynamic range, because it does not depend on finite word length effects like some of the conventional methods. This document also describes the Canonical Signed Digit (CSD) representation of numbers, which further reduces the computational load when used with this method for multiplication and division. Note that Horner's method requires the multiplier or the divisor to be known in advance to function.

The source code that is described in this application report is available from www.ti.com/lit/zip/slaa329.

Contents

1 Introduction 2
2 Horner's Method 2
2.1 Multiplication 2
2.2 Division 4
2.3 Canonical Signed Digit Representation (CSD) 5
2.4 Summary 6
3 Additional Examples 8
3.1 Example 4: Signed Multiplication of Fractions With Multiplicand Negative 8
3.2 Example 5: Signed Multiplication of Fractions With Multiplier Negative 9
3.3 Example 6: Unsigned Multiplication of Integers 9
3.4 Example 7: Signed Multiplication of Integers With Multiplier Negative 10
3.5 Example 8: Unsigned Integer-Float Multiplication 10
3.6 Example 9: Signed Integer-Float Multiplication With Multiplier Negative 11
3.7 Example 10: Unsigned Integer Multiplication With CSD 11
3.8 Example 11: Unsigned Integer-Float Multiplication With CSD Format 12
3.9 Example 12: Signed Integer-Float Multiplication With Multiplier Negative in CSD Format 12
4 Source Code 13

Trademarks

MSP430 is a trademark of Texas Instruments.
All other trademarks are the property of their respective owners.

1 Introduction

Numbers can be broadly classified as fixed point or floating point. The fixed-point number representation is limited to positive and negative integers, while floating-point numbers can also represent fractions. MCUs typically support only fixed-point numbers and calculate only fixed-point arithmetic. Therefore, alternate methods must be devised to calculate floating-point arithmetic. This also leads to a loss in precision due to finite word length effects. Some microcontrollers do not have a hardware multiplier and rely on algorithms based on repeated addition to perform multiplication and division operations. This method consumes many instruction cycles and exhibits limited precision. The Horner's method is tailor-made for such processors without much loss in precision.

2 Horner's Method

The Horner's method requires the multiplier or the divisor to be known in advance. This is not a serious limitation, because few applications perform multiplication or division of numbers that change at runtime. After this value is established, the multiplication or division can be performed efficiently with just shift and add operations. The operand is denoted by X, the multiplier by M, and the divisor by D.

2.1 Multiplication

In explaining this method, first is considered the multiplication of two unsigned fractions.

2.1.1 Example 1: Unsigned Multiplication of Two Fractions

Let the number 0.12345 be multiplied by the constant 0.14325. The 12-bit binary representations of these numbers are:

$X = 0.12345 = 0.000111111001_b$
$M = 0.14325 = 0.001001001010_b$

The conventional method to perform this multiplication is:

$0.12345 \times 0.14325 = (0.000111111001_b)(2^{-3} + 2^{-6} + 2^{-9} + 2^{-11})$

$$
\begin{aligned}
= \; & 0.000000111111_b + \\
& 0.000000000111_b + \\
& 0.000000000001_b + \\
& 0.000000000000_b \\
\hline
& 0.000001000111_b \qquad = 0.017333984375
\end{aligned}
$$

The correct result of this multiplication is 0.0176842125, and the above method results in an absolute error of 0.000350228125, which is approximately 1.5 LSB. This error can be attributed to finite word length effects due to register width limitations. As the number of bits allocated for the fractions increase this error is reduced. The Horner's method aims to reduce this error while maintaining the same register widths.

The Horner's algorithm is based on the positions of the 1s in the multiplier and their distance to the immediate 1 to their left. This is done starting from the rightmost bit position and moving left until the last 1 before the binary point.

In the binary equivalent of the multiplier $0.14325 = 0.001001001010_b$, starting from the right the first 1 occurs at bit position 2^{-11}. The difference in position of this 1 to its immediate 1 to the left is two. Similarly the difference for the 1 in bit position 2^{-9} is three and so on.

If the number to be multiplied is denoted as X, the design equations can be written as:

Step 1: Set the intermediate result equal to the operand X and start with the rightmost 1.

$X \times 2^{-2} + X = X_1$ For the first iteration, the weight 2^{-2} is applied to the intermediate result as the distance of the rightmost 1 (bit position 2^{-11}) in the multiplier to its next 1 (bit position 2^{-9}) is two. The operand X is added to account the occurrence of the 1 at bit position 2^{-9}. The result of this addition is now stored as the intermediate result after this step.

Step 2: Proceed to the next 1 in bit position 2^{-9}

$X_1 \times 2^{-3} + X = X_2$ The weight 2^{-3} is now applied to the intermediate result (updated in Step 1), because the distance of the 1 in bit position 2^{-9} to its next 1 (bit position 2^{-6}) is three. The operand is again added for the reason explained in Step 1.

Step 3: Proceed to the next 1 in bit position 2^{-6}

$X_2 \times 2^{-3} + X = X_3$ The weight 2^{-3} is applied to the intermediate result (updated in Step 2) and the operand added.

Step 4: Proceed to the last 1 in bit position 2^{-3}

Final result = $X_3 \times 2^{-3}$ The factor 2^{-3} is applied to the intermediate result (updated in Step 3), as it is the weight at the position of the leftmost 1. The operand is not added this time, because all of the 1s have been taken into account.

This approach can be validated by back substituting the intermediate results.

$$\begin{aligned}\text{Final result} &= X_3 \times 2^{-3} = (X_2 \times 2^{-3} + X) \times 2^{-3} = [(X_1 \times 2^{-3} + X) \times 2^{-3} + X] \times 2^{-3} \\ &= \{[(X_2 \times 2^{-2} + X) \times 2^{-3} + X] \times 2^{-3} + X\} \times 2^{-3} \\ &= X \times (2^{-11} + 2^{-9} + 2^{-6} + 2^{-3})\end{aligned}$$

This result matches with the multiplication discussed earlier. The algorithm with $X = 0.000111111001_b$ when implemented gives:

$$\begin{aligned} X_1 = X \times 2^{-2} + X = \; & 0.000001111110_b + \\ & 0.000111111001_b \\ \hline X_1 \rightarrow \; & 0.001001110111_b \end{aligned}$$

Similarly,

$$\begin{aligned} X_2 = X_1 \times 2^{-3} + X = \; & 0.000001001110_b + \\ & 0.000111111001_b \\ \hline X_2 \rightarrow \; & 0.001001000111_b \end{aligned}$$

And,

$$\begin{aligned} X_3 = X_2 \times 2^{-3} + X = \; & 0.000001001000_b + \\ & 0.000111111001_b \\ \hline X_3 \rightarrow \; & 0.001001000001_b \end{aligned}$$

Final result = $X_3 \times 2^{-3} = 0.000001001000_b = 0.017578125$

This has an absolute error of 0.0001060875 which is just 0.434534 LSB. Thus Horner's method gives a better result for the same width limitations. Also, the method described uses only shift and add operations. The procedure remains the same if the operand is a negative fraction. Numerical examples for various combinations of the operand (X) and multipliers (M) are given in Section 3.

2.2 Division

Now that Horner's method is shown to work for multiplication with good accuracy, it is easily extended to division. The fact that division by a number is a multiplication by its reciprocal is used to explain the approach. The examples shown in Section 3 indicate that Horner's method works flawlessly for all types of multipliers. Only one example that implements division is shown.

2.2.1 Example 2: Unsigned Division of an Integer by a Floating Point Number

Consider the division of the number X = 441 by the factor 41.8375.

$X = 441 = 0110111001_b$
$D = 41.8375 = 0101001.1101011001_b$
$M = 1/41.8375 = 0.0239020018 = 0.0000011000011110_b$

The design equations for this division are:

Step 1: Set the intermediate result equal to the operand X and start with the rightmost 1

$X \times 2^{-1} + X = X_1$ For the first iteration, the weight 2^{-1} is applied to the intermediate result as the distance of the rightmost 1 (bit position 2^{-15}) in the multiplier to its next 1 (bit position 2^{-14}) is one. The operand X is added to account the occurrence of the 1 at bit position 2^{-14}. The result of this addition is now stored as the intermediate result after this step.

Step 2: Proceed to the next 1 in bit position 2^{-14}

$X_1 \times 2^{-1} + X = X_2$ The weight 2^{-1} is now applied to the intermediate result (updated in Step 1), because the distance of the 1 in bit position 2^{-14} to its next 1 is one. The operand is again added for the reason explained in Step 1.

Step 3: Proceed to the next 1 in bit position 2^{-13}

$X_2 \times 2^{-1} + X = X_3$ The weight 2^{-1} is applied to the intermediate result (updated in Step 2) and the operand added.

Step 4: Proceed to the next 1 in bit position 2^{-12}

$X_3 \times 2^{-5} + X = X_4$ The weight 2^{-5} is applied to the intermediate result (updated in Step 3) and the operand added.

Step 5: Proceed to the next 1 in bit position 2^{-7}

$X_4 \times 2^{-1} + X = X_5$ The weight 2^{-1} is applied to the intermediate result (updated in Step 4) and the operand added.

Step 6: Proceed to the last 1 in bit position 2^{-6}

Final result = $X_5 \times 2^{-6}$ The factor 2^{-6} is applied to the intermediate result (updated in Step 5) as it is the weight at the position of the leftmost 1. The operand is not added this time, because all the 1s have been taken into account.

The correct result for this division is 10.5407827, and the result obtained is 10 with the fractional part discarded with an error of 0.5407827, similar to the error in conventional methods.

All types of operands and multipliers shown in the previous section and Section 3 can be extended to the process of division to produce the desired results.

2.3 Canonical Signed Digit Representation (CSD)

The efficiency of Horner's method can be further improved by using the Canonical Signed Digit (CSD) format to represent the multiplier or divisor. The CSD format aims to reduce the number of add operations during multiplication and division. The CSD format has a ternary set as opposed to a binary set in number representation. The symbols used in this format are {0, 1, $\bar{1}$}, with $\bar{1}$ representing –1. The goal is to group consecutive 1s and change them to a ternary representation from binary representation. This is done starting from the rightmost 1 and proceeding left until the last 1. By doing so, the final CSD representation never has adjacent 1s or $\bar{1}$s. This representation is effective when there are many adjacent 1s in the binary representation.

Consider the 12-bit representation of 0.12345 = 0.000111111001_b.

The CSD format can be used here to group the consecutive 1s. Starting from the rightmost bit, the first 1 occurs at the position 2^{-12}. This 1 does not have an adjacent 1 to its left and hence not modified. The next 1 at position 2^{-9} has five adjacent 1s to the left of it. These six 1s are combined and $\bar{1}$ is placed at the rightmost bit position (2^{-9}), zeros at the remaining positions (2^{-8} to 2^{-4}) and a 1 at the bit position 2^{-3} (one position left to leftmost 1 of the original sequence). This process is repeated for additional groups of 1s that are encountered. The CSD representation now becomes:

$0.12345 = 0.00100000\bar{1}001_{CSD}$

Horner's method using CSD format reduces the number of add operations during multiplication. Section 2.3.1 is the same as Section 2.1.1, with the roles of the multiplier and multiplicand interchanged. The method gives exact results with reduced number of add operations.

2.3.1 Example 3: Unsigned Multiplication of Fractions Using Horner's Method and CSD Representation

$X = 0.14325 = 0.001001001010_b$

$M = 0.12345 = 0.00100000\bar{1}001_{CSD}$

The ternary representation results in a slight modification to Horner's method. The design equation now for an operand X becomes:

Step 1: Set the intermediate result equal to the operand X and start with the rightmost 1.

$X \times 2^{-3} - X = X_1$ For the first iteration, the weight 2^{-3} is applied to the intermediate result as the distance of the rightmost 1 (bit position 2^{-12}) in the multiplier to its next 1 or –1 (bit position 2^{-9}) is three. The operand X is now subtracted (instead of being added) to account the occurrence of the –1 (instead of a 1) at bit position 2^{-9}. The result of this subtraction is now stored as the intermediate result after this step.

Step 2: Proceed to the next 1 or –1 (in this example) at bit position 2^{-9}

$X_1 \times 2^{-6} + X = X_2$ The weight 2^{-6} is applied to the intermediate result (updated in Step 1) and operand added, because 1 is encountered at bit position 2^{-3}.

Step 3: Proceed to the last 1 in bit position 2^{-3}

Final result = $X_2 \times 2^{-3}$ The factor 2^{-3} is applied to the intermediate result (updated in Step 2) as it is the weight at the position of the leftmost 1 (or –1). The operand is not added this time, because all the 1s (or –1s) have been taken into account.

The algorithm with $X = 0.001001001010_b$ when implemented gives:

$$X_1 = X \times 2^{-3} - X = \quad 0.000001001001_b - 0.001001001010_b$$

$$X_1 \rightarrow \quad \tilde{1}.110111111111_b$$

$\tilde{1}$ = Sign bit

Similarly,

$$X_2 = X_1 \times 2^{-6} + X = \quad \tilde{1}.111111110111_b + 0.001001001010_b$$

$$X_2 \rightarrow \quad 0.001001000001_b$$

Final result = $X_2 \times 2^{-3} = 0.000001001000_b = 0.017578125$

By grouping consecutive 1s, the number of add operations is reduced from 6 to 2. A few things can be observed with the results obtained. In this example, interchanging the roles of multiplier and multiplicand produced similar results. Hence, the absolute error in this case is also 0.0001060875, which is just 0.434534 LSB.

The CSD representation holds for integers as well, with a similar procedure. Additional care must be taken while grouping the 1s in doing the conversion to CSD of fractions and integers. The grouping of 1s must be done taking each group independently, one at a time. An example of the integer 441 represented in CSD is shown below:

$441 = 0110111001_b = 100\bar{1}00\bar{1}001_{CSD}$

Step 1 $441 = 0110111001_b = 011100\bar{1}001_{CSD}$

Step 2 $441 = 011100\bar{1}001_{CSD} = 100\bar{1}00\bar{1}001_{CSD}$

Additional examples using the CSD format in conjunction with Horner's method is shown in Section 3.

2.4 Summary

The performance of Horner's method with and without CSD representations has been implemented on the MSP430 architecture. Comparisons of the number of instruction cycles for each multiply and divide is made for Horner's method against its C equivalent and the existing algorithms. The C code and the associated assembly files have been included in a zip file accompanying this application report. Their descriptions are provided in Section 4. The examples considered are (41 × 441), (41 × 441.8375), (9280/41), and (1500/37.12345) for integer-integer and integer-float multiplication and division, respectively.

2.4.1 Results

Table 1 shows a comparison among the methods. For the integer-float multiplication example, the accuracy of Horner's method is almost perfect, which is not the case with existing conventional algorithms. Also, if the multiplier were a pure fraction, scaling of the multiplier to the nearest integer must be done before the use of conventional algorithms.

Table 1. Observations for the Methods Discussed

Type	Methods	Instruction Cycles	Code Size (bytes)	Results	Absolute Error
Integer-Integer Multiplication (41 × 441)	CLIB[1]	77	50	18081	0
	Existing methods[2]	107	54	18081	0
	Horner	15	32	18081	0
	Horner+CSD	13	30	18081	0
Integer-Integer Division (9280 / 41)	CLIB	183	80	226	0
	Existing methods	191	28	226	0
	Horner	23	48	226	0
	Horner+CSD	21	44	226	0
Integer-Float Multiplication (41 × 441.8375)	CLIB	427[3]	322	18115.3375	0
	Existing methods	107	54	18081	34.3375
	Horner	32	66	18115	0.3375
	Horner+CSD	29	60	18115	0.3375
Integer-Float Division (1500 / 37.12345)	CLIB	476	500	40.4057	0
	Existing methods	191	28	40	0.4057
	Horner	24	50	40	0.4057
	Horner+CSD	22	46	40	0.4057

(1) The C library is part of the IAR Embedded Workbench Ver. 3.41A, written for the MSP430 family of devices.
(2) The algorithms have been explained in the book *Computer Organization*, Carl Hamacher, Zvonko Vranesic, and Safawat Zaky, 3rd Edition, McGraw Hill Publication, 1990.
(3) Includes cycles for type conversion from float to integer as part of a requirement of the algorithm used.

2.4.2 Tradeoffs

Table 1 results are used as a comparison of speed, accuracy and memory requirements. The MSP430 CPU implements Horner's algorithms extremely quickly with its single cycle shift and add operations. The Horner's method, when compared to the existing methods, is extremely fast and maintains the same level of accuracy for integer-integer multiplications and better accuracy for integer-float multiplications. Division is also performed extremely fast with limited accuracy. The error in Horner's method is similar to existing methods for integer-integer division. The results for integer-float divisions can be improved by representing the divisor by a higher number of bits, with an insignificant increase in cycle count. There is no provision to obtain the remainder during division in Horner's method, which is not the case with the existing algorithms. The most important benefit of Horner's method is the multipliers and divisors suffer very little from finite word length effects. Although the examples considered had only 12- or 16-bit word lengths, each of them can be implemented with large precision with proportional increase in the number of shift and add operations. The memory requirement is much higher for Horner's method, because the code is different for each multiplier or divisor. In cases where speed is of prime concern, this is not a serious limitation. Also Horner's method requires the multiplier or the divisor to be known in advance, which is not the case with the other methods. This does not pose any limitation to filtering operations and other standard conversions, where the multipliers and divisors do not change during runtime.

3 Additional Examples

This section includes examples to help you better understand Horner's method for various types of operands and multipliers.

3.1 Example 4: Signed Multiplication of Fractions With Multiplicand Negative

Consider the example when –0.12345 is multiplied by the constant 0.14325. The representation of this negative fraction in the 2s complement format is:

$$X = -0.12345 = \tilde{1}.111000000111_b$$
$$M = 0.14325 = 0.001001001010_b$$

where $\tilde{1}$ is the sign bit representing –1.

The design equations for this numerical example are shown.

$$X_1 = X \times 2^{-2} + X = 1.111110000001_b + 1.111000000111_b$$
$$X_1 \rightarrow 1.110110001000_b$$
$$X_2 = X_1 \times 2^{-3} + X = 1.111110110001_b + 1.111000000111_b$$
$$X_2 \rightarrow 1.110110111000_b$$
$$X_3 = X_2 \times 2^{-3} + X = 1.111110110111_b + 1.111000000111_b$$
$$X_3 \rightarrow 1.110110111110_b$$

Final result = $X_3 \times 2^{-3} = 1.111110110111_b = –0.017822265625$, which has an absolute error of 0.000138053125, which is just 0.565466 LSB.

The above example shows that the method is accurate, regardless of the sign of the operand X. If the multiplier were a negative number, the above procedure remains exactly the same, except that in the end the operand X is subtracted due to the sign bit of the multiplier. The design equations for this are shown in Example 5.

3.2 Example 5: Signed Multiplication of Fractions With Multiplier Negative

If the multiplier is now considered to be –0.12345 and the operand X to be 0.14325. The binary representations are:

$M = –0.12345 = \tilde{1}.111000000111_b$
$X = 0.14325 = 0.001001001010_b$

where $\tilde{1}$ is the sign bit representing –1.

$X \times 2^{-1} + X = X_1$
$X_1 \times 2^{-1} + X = X_2$
$X_2 \times 2^{-7} + X = X_3$
$X_3 \times 2^{-1} + X = X_4$
$X_4 \times 2^{-1} + X = X_5$

Final result = $X_5 \times 2^{-1} – X$

With the present operands the absolute error obtained is 0.000138053125, which is 0.565466 LSB.

The previous examples considered fraction-fraction multiplication exhibiting the better accuracy of this method over others. This method can also be employed when the multiplier is any integer or real number. The procedure for this is very similar to the previous method, with a slight change. In microcontrollers, the operand X is usually data coming from the analog-to-digital converter (ADC), which is a fixed-point number. Typical examples are considered with the operand X a fixed point and the multiplier an integer or a real number.

3.3 Example 6: Unsigned Multiplication of Integers

Consider the value of X to be 41 and the multiplier is 441.

$X = 41 = 0101001_b$
$M = 441 = 0110111001_b$

The only change when the multiplier is an integer is the algorithm starts looking for 1s starting from the leftmost bit and moves right. Similar to the approach for fractions the difference in bit positions of the neighboring 1s is used as weights. Because the direction is from left to right, the weights all become positive powers of 2 rather than negative. The design equation for this example is:

$X \times 2^{1} + X = X_1$
$X_1 \times 2^{2} + X = X_2$
$X_2 \times 2^{1} + X = X_3$
$X_3 \times 2^{1} + X = X_4$
$X_4 \times 2^{3} + X = X_5$

Final result = $X_5 \times 2^{0}$

The final result becomes the result X_5 weighted by 2^0 (the bit position of the rightmost 1).

The absolute error obtained for this multiplication is zero.

3.4 Example 7: Signed Multiplication of Integers With Multiplier Negative

The design equations would slightly change if the multiplier was negative. Consider the multiplier to be –441 instead for the same X.

$X = 41 = 0101001_b$
$M = –441 = \tilde{1}\,001000111_b$

The design equations now become:

$–X \times 2^3 + X = X_1$ The sign bit causes the operand to be negative in the first equation.
$X_1 \times 2^4 + X = X_2$
$X_2 \times 2^1 + X = X_3$
$X_3 \times 2^1 + X = X_4$

Final result = $X_4 \times 2^0$

The absolute error obtained for this multiplication is zero.

If the multiplier is a real number with an integer part and fractional part, the method is just a combination of the two types discussed so far. The algorithm is now divided into two parts, one for the integer part of the multiplier and the other for the fractional part of the multiplier. This approach is explained in Example 8.

3.5 Example 8: Unsigned Integer-Float Multiplication

Consider the multiplier to be defined as 441.8375 and the operand X is 41.

$X = 41 = 0101001_b$
$M = 441.8375 = 0110111001.1101011001_b$

The fact to note here is that this method is independent of word length for the multiplier. The overhead would only be in the number of shifts and adds to be done.

The design equations for this example is given by:

$X \times 2^1 + X = X_1$
$X_1 \times 2^2 + X = X_2$
$X_2 \times 2^1 + X = X_3$
$X_3 \times 2^1 + X = X_4$
$X_4 \times 2^3 + X = X_5$

Intermediate result due to integer part = $X_5 \times 2^0$

$X \times 2^{-3} + X = X_1$
$X_1 \times 2^{-1} + X = X_2$
$X_2 \times 2^{-2} + X = X_3$
$X_3 \times 2^{-2} + X = X_4$
$X_4 \times 2^{-1} + X = X_5$

Intermediate result due to fractional part = $X_5 \times 2^{-1}$

Adding the above two results gives us the final solution. The absolute error obtained for this multiply is just the fractional part and is 0.3375 LSB.

The procedure is very similar if the multiplier is a negative real number, as shown in Section 3.6.

3.6 Example 9: Signed Integer-Float Multiplication With Multiplier Negative

Consider the multiplier to be defined as –441.8375, and the operand X is 41.

$X = 41 = 0101001_b$

$M = –441.8375 = \tilde{1}\ 001000110.0010100111_b$

The design equations for this example is given by:

$–X \times 2^3 + X = X_1$

$X_1 \times 2^4 + X = X_2$

$X_2 \times 2^1 + X = X_3$

Intermediate result due to integer part = $X_3 \times 2^1$

$X \times 2^{-1} + X = X_1$

$X_1 \times 2^{-1} + X = X_2$

$X_2 \times 2^{-3} + X = X_3$

$X_3 \times 2^{-2} + X = X_4$

Intermediate result due to fractional part = $X_4 \times 2^{-3}$

Adding the above two results gives us the final solution. The absolute error obtained for this multiply is just the fractional part and is 0.6625 LSB.

3.7 Example 10: Unsigned Integer Multiplication With CSD

Consider the example of an integer-integer multiplication using Horner's method using the CSD format, with the operand X being 41 and multiplier 441.

$X = 41 = 0101001_b$

$M = 441 = 0110111001_b = 100\bar{1}00\bar{1}001_{CSD}$

The design equations are

$X_1 = X \times 2^3 – X =\ 00101001000_b –$

00000101001_b

$X_1 \rightarrow\ 00100011111_b$

$X_2 = X_1 \times 2^3 – X =\ 100011111000_b –$

000000101001_b

$X_2 \rightarrow\ 100011001111_b$

$X_3 = X_2 \times 2^3 + X =\ 100011001111000_b +$

000000000101001_b

$X_3 \rightarrow\ 100011010100001_b$

Final result = $X_3 = (100011010100001)_b = 18081$

The absolute error obtained for this multiplication is zero.

The design equations remain the same if the operand X were a negative number.

3.8 Example 11: Unsigned Integer-Float Multiplication With CSD Format

When the multiplier is a real number say 441.8375 and the operand X is a positive integer say 41, the design equations become:

$X = 41 = 0101001_b$

$M = 441.8375 = 0110111001.1101011001_b = 100\bar{1}00\bar{1}010.00\bar{1}0\bar{1}0\bar{1}001_{CSD}$

The design equations for this example is given by:

$X \times 2^3 - X = X_1$

$X_1 \times 2^3 - X = X_2$

$X_2 \times 2^2 + X = X_3$

Intermediate result due to integer part = $X_3 \times 2^1$

$X \times 2^{-3} - X = X_1$

$X_1 \times 2^{-2} - X = X_2$

$X_2 \times 2^{-2} - X = X_3$

Intermediate result due to fractional part = $X_3 \times 2^{-3}$

Adding the above two results gives us the final solution. The absolute error obtained for this multiplication is just the fractional part and is 0.3375 LSB.

A point to be noted here is if the rightmost bit is $\bar{1}$, the design equations start with a negative X weighted and added or subtracted with X depending on the next bit being 1 or –1, respectively.

3.9 Example 12: Signed Integer-Float Multiplication With Multiplier Negative in CSD Format

Consider the example if the multiplier was negative and the operand X a positive number 41 with the multiplier –441.8375.

$X = 41 = 0101001_b$

$M = -441.8375 = \bar{1}001000110.0010100111b = \tilde{1}\ 00100101\bar{0}.0010101001\bar{1}_{CSD}$

The design equations are:

$-X \times 2^3 + X = X_1$

$X_1 \times 2^3 + X = X_2$

$X_2 \times 2^2 - X = X_3$

Intermediate result due to integer part = $X_3 \times 2^1$

$-X \times 2^{-3} + X = X_1$

$X_1 \times 2^{-2} + X = X_2$

$X_2 \times 2^{-2} + X = X_3$

Intermediate result due to fractional part = $X_3 \times 2^{-3}$

Adding the above two results gives the final solution. The absolute error obtained for this multiplication is just the fractional part and is 0.6625 LSB.

4 Source Code

The following source code files implement multiplication and division. Some of these examples assume that the incoming operand is a 12-bit sample from the ADC. You can download these files from www.ti.com/lit/zip/slaa329.

multiply_int.c – source file calling four math functions. All the functions perform multiplication of two 16-bit integers to give a 16-bit integer result. The "main()" function calls each of these functions.

mul.s43 – assembly source file containing an existing scheme for integer-integer multiplication using only shift and add operations. This source file is called by function "main()" in the source file multiply_int.c and multiply_float.c.

horner_mul_int.s43 – assembly source file containing the Horner's scheme for integer-integer multiplication. This source file is called by function "main()" in the source file multiply_int.c.

csd_mul_int.s43 – assembly source file containing the Horner's scheme using the CSD format for integer-integer multiplication. This source file is called by function "main()" in the source file multiply_int.c.

multiply_float.c – source file calling four math functions. All the functions perform multiplication of a 16-bit integer with a floating point number to give a 16-bit integer or float result. The "main()" function calls each of these functions.

horner_mul_float.s43 – assembly source file containing the Horner's scheme for integer-float multiplication. This source file is called by function "main()" in the source file multiply_float.c.

csd_mul_float.s43 – assembly source file containing the Horner's scheme using the CSD format for integer-float multiplication. This source file is called by function "main()" in the source file multiply_float.c.

div_int.c – source file calling four math functions. All the functions perform division of two 16-bit integers to give a 16-bit integer result. The "main()" function calls each of these functions.

div.s43 – assembly source file containing an existing scheme for integer-integer division using only shift and add operations. This source file is called by function "main()" in the source file div_int.c and div_float.c.

horner_div_int.s43 – assembly source file containing the Horner's scheme for integer-integer division. This source file is called by function "main()" in the source file div_int.c.

csd_div_int.s43 – assembly source file containing the Horner's scheme using the CSD format for integer-integer division. This source file is called by function "main()" in the source file div_int.c.

div_float.c – source file calling four math functions. All the functions perform division of a 16-bit integer with a floating point number to give a 16-bit integer or float result. The "main()" function calls each of these functions.

horner_div_float.s43 – assembly source file containing the Horner's scheme for integer-float division. This source file is called by function "main()" in the source file div_float.c.

csd_div_float.s43 – assembly source file containing the Horner's scheme using the CSD format for integer-float multiplication. This source file is called by function "main()" in the source file div_float.c.

Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from September 1, 2006 to July 18, 2018	Page
• General edits throughout document	1
• Added link to download the source files in the abstract	1

IMPORTANT NOTICE FOR TI DESIGN INFORMATION AND RESOURCES

Texas Instruments Incorporated ('TI") technical, application or other design advice, services or information, including, but not limited to, reference designs and materials relating to evaluation modules, (collectively, "TI Resources") are intended to assist designers who are developing applications that incorporate TI products; by downloading, accessing or using any particular TI Resource in any way, you (individually or, if you are acting on behalf of a company, your company) agree to use it solely for this purpose and subject to the terms of this Notice.

TI's provision of TI Resources does not expand or otherwise alter TI's applicable published warranties or warranty disclaimers for TI products, and no additional obligations or liabilities arise from TI providing such TI Resources. TI reserves the right to make corrections, enhancements, improvements and other changes to its TI Resources.

You understand and agree that you remain responsible for using your independent analysis, evaluation and judgment in designing your applications and that you have full and exclusive responsibility to assure the safety of your applications and compliance of your applications (and of all TI products used in or for your applications) with all applicable regulations, laws and other applicable requirements. You represent that, with respect to your applications, you have all the necessary expertise to create and implement safeguards that (1) anticipate dangerous consequences of failures, (2) monitor failures and their consequences, and (3) lessen the likelihood of failures that might cause harm and take appropriate actions. You agree that prior to using or distributing any applications that include TI products, you will thoroughly test such applications and the functionality of such TI products as used in such applications. TI has not conducted any testing other than that specifically described in the published documentation for a particular TI Resource.

You are authorized to use, copy and modify any individual TI Resource only in connection with the development of applications that include the TI product(s) identified in such TI Resource. NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY OTHER TI INTELLECTUAL PROPERTY RIGHT, AND NO LICENSE TO ANY TECHNOLOGY OR INTELLECTUAL PROPERTY RIGHT OF TI OR ANY THIRD PARTY IS GRANTED HEREIN, including but not limited to any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information regarding or referencing third-party products or services does not constitute a license to use such products or services, or a warranty or endorsement thereof. Use of TI Resources may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

TI RESOURCES ARE PROVIDED "AS IS" AND WITH ALL FAULTS. TI DISCLAIMS ALL OTHER WARRANTIES OR REPRESENTATIONS, EXPRESS OR IMPLIED, REGARDING TI RESOURCES OR USE THEREOF, INCLUDING BUT NOT LIMITED TO ACCURACY OR COMPLETENESS, TITLE, ANY EPIDEMIC FAILURE WARRANTY AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

TI SHALL NOT BE LIABLE FOR AND SHALL NOT DEFEND OR INDEMNIFY YOU AGAINST ANY CLAIM, INCLUDING BUT NOT LIMITED TO ANY INFRINGEMENT CLAIM THAT RELATES TO OR IS BASED ON ANY COMBINATION OF PRODUCTS EVEN IF DESCRIBED IN TI RESOURCES OR OTHERWISE. IN NO EVENT SHALL TI BE LIABLE FOR ANY ACTUAL, DIRECT, SPECIAL, COLLATERAL, INDIRECT, PUNITIVE, INCIDENTAL, CONSEQUENTIAL OR EXEMPLARY DAMAGES IN CONNECTION WITH OR ARISING OUT OF TI RESOURCES OR USE THEREOF, AND REGARDLESS OF WHETHER TI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You agree to fully indemnify TI and its representatives against any damages, costs, losses, and/or liabilities arising out of your non-compliance with the terms and provisions of this Notice.

This Notice applies to TI Resources. Additional terms apply to the use and purchase of certain types of materials, TI products and services. These include; without limitation, TI's standard terms for semiconductor products http://www.ti.com/sc/docs/stdterms.htm), evaluation modules, and samples (http://www.ti.com/sc/docs/sampterms.htm).

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265