FEATURE ARTICLE

by Kripasagar Venkat

Efficient Mirco Mathematics

Multiplication and Division Techniques for MCUs

Most inexpensive microcontrollers don't have a hardware multiplier module, and they typically require numerous of instruction cycles to perform multiplication and division operations. Kripasagar describes techniques based on Horner's method for performing efficient multiplication and division without a hardware multiplier.

Low-cost microcontrollers are typically targeted at applications with low levels of complexity and are optimized for cost and performance. The devices should offer low power consumption and be simple to design. They should also provide easy interfaces to external peripherals. As technology advances, the demand for processing capabilities and efficiency increases. This demand calls for an upgrade to existing microcontrollers and an introduction of newer peripherals. The design tools that support enhanced hardware and software must also go through a process of considerable development. Manufacturers continuously strive to provide on-chip solutions for complex algorithms, thus increasing cost. Some microcontrollers choose to go through limited enhancements and adhere to low cost and low power consumption. Complex algorithms and functions such as digital filtering become impossible on those devices. This forces designers to search for efficient methods and make do with what is available. This article focuses on one such efficient method for microcontrollers.

ALGORITHMS

Processors are broadly classified as fixed point or floating point. Fixed-point processors support only integers, whereas floating-point processors have additional circuitry to support integers and fractions. Various standards for floating-point formats have been established to maintain uniformity among processors and their associated tools. All microcontrollers fall under fixed-point processors and are either 8-bit or 16-bit devices. Fixed-point processors suffer from the effects of finite word length, round-off, and truncation.[1] These issues have a direct impact on the accuracy of the results obtained during mathematical operations. A hardware multiplier is a module that supports multiplication and multiply and accumulate (MAC) operations via dedicated central processing unit (CPU) instructions. Most low-cost microcontrollers do not have a hardware multiplier module. They usually require a lot of instruction cycles to perform multiplication using alternate algorithms.

Many algorithms have been devised for fast multiply and divide using only shift and add instructions for fixed-point processors.[2] All of these handle only integers and become trivial when the multipliers or divisors are pure integers. This does not mean that they are unable to support multipliers or divisors that are pure fractions or real numbers. The solution is a form of scaling that converts all real numbers to integers. Different standards—such as Q formats—have been introduced to accomplish this on fixed-point machines. There is, however, a potential loss in accuracy with such formats due to truncation of the real number on registers with fixed widths. Horner's algorithm attempts to reduce this error and improve accuracy. A method for multiplication is easily extended to division, which is a multiplication by the divisor's reciprocal. An innovative scaling-free method to implement integer-real multiplications will be described in this article.

Horner's algorithm is based on the position of the bits with a value of 1 and their distance to the neighboring 1

$$
\begin{aligned}
0.2468 \times 0.1357 &= \left(0.001111110010111_b\right)\left(2^{-3} + 2^{-7} + 2^{-9} + 2^{-11} + 2^{-12} + 2^{-13} + 2^{-14}\right)\\
&= 0.0000011111\ 10010_b\\
&+\ 0.0000000001\ 11111_b\\
&+\ 0.0000000000\ 01111_b\\
&+\ 0.0000000000\ 00011_b\\
&+\ 0.0000000000\ 00001_b\\
&+\ 0.0000000000\ 00000_b\\
&+\ 0.0000000000\ 00000_b\\
\hline
&\ 0.000010001000100_b = 0.03332519\ 53125
\end{aligned}
$$

Figure 1—This is the binary equivalent of a 15 × 15-bit fractional multiplication with the multiplier known in advance. Dedicated code would replace each of these steps to perform a multiplication on a 16-bit CPU.

$$\begin{aligned}
\text{Final product} &= x_6 \times 2^{-3} = \left(x_5 \times 2^{-4} + x\right) \times 2^{-3} \\
&= \left(\left(x_4 \times 2^{-2} + x\right) \times 2^{-4} + x\right) \times 2^{-3} \\
&= \left(\left(\left(x_3 \times 2^{-2} + x\right) \times 2^{-2} + x\right) \times 2^{-4} + x\right) \times 2^{-3} \\
&= \left(\left(\left(\left(x_2 \times 2^{-1} + x\right) \times 2^{-2} + x\right) \times 2^{-2} + x\right) 2^{-4} + x\right) \times 2^{-3} \\
&= \left(\left(\left(\left(\left(x_1 \times 2^{-1} + x\right) \times 2^{-1} + x\right) \times 2^{-2} + x\right) \times 2^{-2} + x\right) \times 2^{-4} + x\right) \times 2^{-3} \\
&= \left(\left(\left(\left(\left(\left(x \times 2^{-1} + x\right) \times 2^{-1} + x\right) \times 2^{-1} + x\right) \times 2^{-2} + \times\right) \times 2^{-2} + \times\right) \times 2^{-4} + x\right) \times 2^{-3} \\
&= x \times \left(2^{-3} + 2^{-7} + 2^{-9} + 2^{-11} + 2^{-12} + 2^{-13} + 2^{-14}\right)
\end{aligned}$$

Figure 2—This is an example of a back substitution in the steps of Horner's method for a fractional multiplier. This is done to verify if Horner's method conforms to the conventional binary multiplication routine for fractions.

in a multiplier. For this method to work, the multiplier or divisor should be known in advance. This method also relies on dedicated code for any multiplies or divides with a potential increase in code size. These are no serious limitations in applications in which speed is of prime concern or when the multiplier or divisor does not change run time. The canonical signed digit (CSD) format is introduced to further reduce the CPU overhead.

FILTERING

Filtering forms the crux of many digital processing algorithms. Filtering can be viewed as a weighted multiply and accumulate (MAC) process. Digital filters come in two flavors: finite impulse response (FIR) and the infinite impulse response (IIR). A mathematician's view of a FIR is a transfer function with only a numerator polynomial. An IIR filter is a transfer function with both numerator and denominator polynomials. FIR filters use only the present and past samples of the input signal, whereas IIR filters also use the previous output samples.[1] Stability of these filters is important because it tries to preserve the sanity of the signal being filtered. A FIR filter is inherently stable with no restriction on the roots (also known as zeros) of its polynomial. It can also exhibit linear phase to reduce the phase distortion of any input signal. The IIR filter performs much better and requires a significantly smaller order compared to a FIR filter for the same set of filtering specifications. The IIR filter would be stable only if the roots of the denominator polynomial (also known as poles) had a magnitude of less than one. Special care must be taken when scaling the IIR filter coefficients because it could easily render a stable IIR filter unstable. Digital filter coefficients almost always are real numbers. An implementation of them on fixed-point processors must support scaling to integers and additional software to track any sort of overflow associated with them.[3] An alternative to all of this is to use high-level languages such as C along with a floating-point library to support integer real number multiplies. The C floating-point library would produce accurate results but with an increased CPU overhead, rendering real-time processing at times would be impossible.

Texas Instruments's ultra-low-power MSP430 microcontrollers are an example of modern 16-bit microcontrollers that support single-cycle shift and add instructions.[4] Although some of the devices offer the hardware multiplier module, this method can be used to accomplish an integer real multiply. Comparisons of Horner's method and Horner's method using CSD are made with existing integer-integer multiply algorithms and a C floating-point library. Results showing good accuracy adhere to reduced CPU bandwidth and power consumption. Performance of FIR digital filters on the MSP430 has also been shown to have excellent accuracy.

FRACTIONAL MULTIPLIER

The data to be processed is usually an analog signal converted to digital samples using an on-chip ADC, for example an ADC12, a 12-bit ADC on the MSP430. The digital samples can represent analog signals such as temperature captured by a sensor or audio signals that need some sort of digital filtering.

Horner's algorithm is explained in two parts, pure fractional multipliers and pure integer multipliers. Steps have been shown to distinguish their procedures to achieve efficient multiplies.

A fractional multiplier M falls in the range –1 < M < 1. Multiplication by such a number can be accomplished on fixed-point machines only by scaling M to the nearest integer. Alternatively, the bits of value 1 can be identified and shift and add operations on any multiplicand x can be done. This approach would lead to a dedicated code for each multiplier. Consider the example with x = 0.2468 and M = 0.1357 with 15-bit resolution:

$$\begin{aligned} x &= 0.2468 = 0.001111110010111_b \\ M &= 0.1357 = 0.001000101011110_b \end{aligned} \qquad [1]$$

$$x_1 = \left\{\begin{array}{r} 0.000111111001011_b \\ +\ 0.001111110010111_b \\ \hline 0.010111101100010_b \\ \hline \end{array}\right. \quad x_2 = \left\{\begin{array}{r} 0.001011110110001_b \\ +\ 0.001111110010111_b \\ \hline 0.011011101001000_b \\ \hline \end{array}\right.$$

$$x_3 = \left\{\begin{array}{r} 0.001101110100100_b \\ +\ 0.001111110010111_b \\ \hline 0.011101100111011_b \\ \hline \end{array}\right. \quad x_4 = \left\{\begin{array}{r} 0.000111011001110_b \\ +\ 0.001111110010111_b \\ \hline 0.010111001100101_b \\ \hline \end{array}\right.$$

$$x_5 = \left\{\begin{array}{r} 0.000101110011001_b \\ +\ 0.001111110010111_b \\ \hline 0.010101100110000_b \\ \hline \end{array}\right. \quad x_6 = \left\{\begin{array}{r} 0.000001010110011_b \\ +\ 0.001111110010111_b \\ \hline 0.010001001001010_b \\ \hline \end{array}\right.$$

Figure 3—Binary representation and mapping of the implementation steps of Horner's method for Equation 1. The result of the intermediate steps x1 to x6 shows its progression to the final result.

$$\begin{aligned}\text{Final product} &= x_3 \times 2^0 = \left(x_2 \times 2^2 + x\right)\\ &= \left(\left(x_1 \times 2^1 + x\right) \times 2^2 + x\right)\\ &= \left(\left(\left(x \times 2^3 + x\right) \times 2^1 + x\right) \times 2^2 + x\right)\\ &= x \times \left(2^0 + 2^2 + 2^3 + 2^6\right) = x \times 77\end{aligned}$$

Figure 4—Back substitution in the steps of Horner's method for an integer multiplier. This is done to verify if Horner's method conforms to the conventional binary multiplication routine for integers.

Figure 1 shows the exact bit-wise binary addition for this multiplication.

The correct result for this multiplication using floating-point math is 0.03349076 and the absolute error using the conventional method is 0.0001655646875, an error of approximately 5.4 LSB. This error is due to finite word-length effects on the multiplier. Horner's algorithm attempts to reduce this error.

Horner's algorithm identifies the positions of bits with a value of 1 in the multiplier and their distance to the nearest 1 to the left. This is done starting from the rightmost 1 and moving left to the last 1 before the binary point. For Equation 1 with M = 0.1357, the position of the bits with a value of 1 in the multiplier are {2^{-14}, 2^{-13}, 2^{-12}, 2^{-11}, 2^{-9}, 2^{-7}, 2^{-3}}. The distance of the closest binary 1 to the left for each of the bits is {1, 1, 1, 1, 2, 2, 2, 4}. Once this has been established, Horner's algorithm generates a set of design equations using only shift and add operations. The design equations are written in terms of the multiplicand x. It is assumed that the reader is aware that 2^{-1} is a right shift by 1 and 2^1 is a left shift by 1.

The first step is to initialize the intermediate result to x and proceed to the rightmost bit (2^{-14}). The nearest 1 to its left is at bit position 2^{-13}. The difference in weight 2^{-1} is applied to the intermediate result. The multiplicand x is then added to the weighted result for the occurrence of the 1 at bit position 2^{-13}. The result of this addition is now stored as the intermediate result x_1 for the next step:

$$x \times 2^{-1} + x = x_1 \qquad [2]$$

The next step is to proceed to the next bit with a value of 1 (2^{-13}). The nearest 1 to its left is at bit position 2^{-12}. The difference in weight 2^{-1} is applied to the intermediate result. The multiplicand x is again added to the weighted result for the occurrence of the 1 at bit position 2^{-12}. The result of this addition is now stored as the intermediate result x_2 for the next step:

$$x_1 \times 2^{-1} + x = x_2 \qquad [3]$$

The third step is to proceed to the next bit with a value of 1 (2^{-12}). The nearest 1 to its left is at bit position 2^{-11}. The difference in weight 2^{-1} is applied to the intermediate result. The multiplicand x is again added to the weighted result for the occurrence of the 1 at bit position 2^{-11}. The result of this addition is now stored as the intermediate result x_3 for the next step:

$$x_2 \times 2^{-1} + x = x_3 \qquad [4]$$

The fourth step is to proceed to the next bit with a value of 1 (2^{-11}). The nearest 1 to its left is at bit position 2^{-9}. The difference in weight 2^{-2} is applied to the intermediate result. The multiplicand x is again added to the weighted result for the occurrence of the 1 at bit position 2^{-9}. The result of this addition is stored as the intermediate result x_4 for the next step:

$$x_3 \times 2^{-2} + x = x_4 \qquad [5]$$

The fifth step is to proceed to the next bit with a value of 1 (2^{-9}). The nearest 1 to its left is at bit position 2^{-7}. The difference in weight 2^{-2} is applied to the intermediate result. The multiplicand x is again added to the weighted result for the occurrence of the 1 at bit position 2^{-7}. The result of the addition is stored as the intermediate result x_5 for the next step:

$$x_4 \times 2^{-2} + x = x_5 \qquad [6]$$

At step six, proceed to the next bit with a value of 1 (2^{-7}). The nearest 1 to its left is at bit position 2^{-3}. The difference in weight 2^{-4} is applied to the intermediate result. The multiplicand x is again added to the weighted result for the occurrence of the 1 at bit position 2^{-3}. The result of this addition is now stored as the intermediate result x_6 for the next step:

$$x_5 \times 2^{-4} + x = x_6 \qquad [7]$$

The seventh step is to proceed to the last bit with a value of 1 (2^{-3}). Because it is the last binary 1, it does not have any ones to the left, therefore, only its bit position is applied as the weight to the intermediate result to give the final product (i.e., 2^{-3}). The result is the final product:

$$\text{Final product} = x_6 \times 2^{-3} \qquad [8]$$

The procedure can be validated by back substitution to give the same result as the conventional multiply (see Figure 2).

The bit-wise realization of design Equations 2 through 7 is shown in Figure 3. It indicates the exact operations at each stage and gives a brighter picture of Horner's method. The final product is 0.000010001001001_b.

This results in an absolute error of 0.000012976796875, an error of approximately 0.42522368 LSB. Thus, Horner's algorithm is extremely accurate and does not suffer much from finite word length

$$M = 0.1357 = 0.001000101\underbrace{0111}_{\text{group}}10_b$$

$$= 0.00100010\,\underbrace{11}_{\text{group}}\,00\bar{1}0_b = 0.001000\,\underbrace{11}_{\text{group}}\,0\bar{1}000\bar{1}0_b$$

$$= 0.0010010\bar{1}0\bar{1}000\bar{1}0_{CSD} = 2^{-3} + 2^{-6} - 2^{-8} - 2^{-10} - 2^{-14}$$

Number of reducations in add is two

$$M = 891 = 11011110\,\underbrace{11}_{\text{group}}{}_b = 110\underbrace{11111}_{\text{group}}0\bar{1}_b = \underbrace{111}_{\text{group}}\,0000\bar{1}0\bar{1}_b$$

$$= 100\bar{1}0000\bar{1}0\bar{1}_{CSD} = 2^{10} - 2^7 - 2^{-2} - 2^0 = 1{,}024 - 128 - 4 - 1 = 891$$

Number of reductions in add is four

Figure 5—For any integer or fractional multiplier M, the CSD conversion of multipliers is done to reduce overhead. Stepwise grouping of adjacent binary 1s with the number of reductions in each case is shown. The integrity of the multiplier after conversion to CSD is also verified.

effects. The design equations involve only right shifts and add operations making it an extremely fast algorithm. The design equations are unique for this multiplier and a 15-bit register width. This also implies that the multiplicand x can be any number (integer/fraction) of any sign (positive/negative). If the multiplier is a negative number, 2's complement format should be used for its binary representation and should follow similar steps to obtain the design equations.

INTEGER MULTIPLIER

Horner's method is easily extended to integer multipliers using the same concept. The only difference is in the search for ones. The search is now from the leftmost bit to the rightmost bit before the binary point. The multiplier M in this case is any integer. Additional care must be taken to ensure that the result of the multiply does not exceed the range for representation in the microcontroller. The second example shows a similar scheme when the multiplier is an integer type. Only design equations have been provided for this multiplier with x taking any form (integer/fraction). Consider the multiplier M = 77 = 1001101_b. The following are design equations:

$$x \times 2^3 + x = x_1 \quad [9]$$

$$x_1 \times 2^1 + x = x_2 \quad [10]$$

$$x_2 \times 2^2 + x = x_3 \quad [11]$$

$$\text{Final product} = x_3 \times 2^0 \quad [12]$$

The weights in design Equations 9 through 11 are all positive powers of two rather than negative, as the direction is now from left to right, as opposed to right to left for fractional multipliers. The result of back substitution for the integer multiplier is shown in Figure 4.

Methods	CPU cycles	Code size	Result	Absolute error
Horner's method	33	68 bytes	10656	0.38979
Horner with CSD	27	56 bytes	10656	0.38979
Existing method (14)	107	54 bytes	9954	702.38979
Existing method (15)	107	54 bytes	10665	8.61021
C Floating-point library	427	322 bytes	10656.38979	0

Table 1—This evaluation of Horner's method is done with comparisons to other methods on the MSP430 platform. For consistency, the same multipliers are chosen and the result for each multiply is shown. Cycle count directly relates to the overhead on the CPU and absolute error indicates the accuracy of each method.

Listing 1—This is a section of typical MSP430 microcontroller code for an integer-real multiply. The dedicated code mainly consists of add/subtract and shift instructions only. For an MSP430 microcontroller, the add/subtract and shift instructions are single-cycle.

```
inv.w R13            ; 2's compliment since the last digit is -1
add.w #1,R13
rra.w R13
rra.w R13
rra.w R13
add.w R12,R13        ; X1=-X*2^-3+X
rra.w R13
rra.w R13
sub.w R12,R13        ; X2=X1*2^-2-X
rra.w R13
rra.w R13
rra.w R13
rra.w R13
rra.w R13
rra.w R13
sub.w R12,R13        ; X3=X2*2^-6-X
rra.w R13
rra.w R13
rra.w R13
rra.w R13
add.w R12,R13        ; X4=X3*2^-4+X
rla.w R13            ; Upscale final result by 16
rla.w R13
rla.w R13
rla.w R13            ; Final result of multiplication
mov.w R13, R12       ; Value returned to calling function
ret
END
```

REAL MULTIPLIER

Procedures for pure fraction and pure integer multipliers are shown separately to distinguish their implementation. Once this is established, a real number multiplication can easily be realized using either of the two approaches. The multiplier is scaled up or scaled down to either pure integers or pure fractions, and Horner's method applied to them. Once the multiplication is complete, the result must be scaled accordingly. The resulting error is extremely small, similar to Equation 1. This makes filtering an easy task without the overhead of a C floating-point library.

CSD REPRESENTATION

The CSD format or representation is tailor-made for algorithms that rely on bits with a value of 1 for their design.[3] The CSD format for its representation uses a ternary set {–1, 0, 1} compared to a binary set {0, 1}. If looked at carefully, the number of steps in Horner's algorithm depends on the number of 1s present in the multiplier. The CSD format attempts to reduce the steps by grouping consecutive 1s in the multiplier and replacing them with a combination of the ternary set {–1, 0, 1}. This modification reduces the number of add operations for multipliers that have groups of consecutive 1s. By examination, the CSD representation would never have adjacent 1s or –1s. One must not be confused with the term "ternary set" because the process of introducing –1s in a number is merely to replace some of the additions by subtractions. The grouping is done starting from the rightmost bit and proceeding left. Figure 5 shows an example with the steps involved in

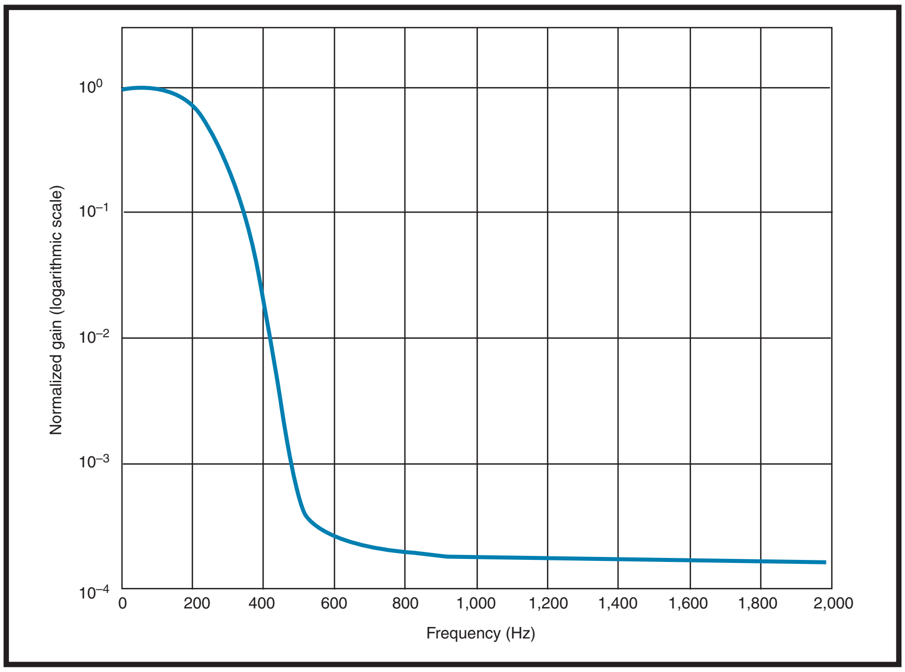

Figure 6—Horner's method when used to implement a low-pass filter with a cut-off of 300 Hz. The response is depicted as a gain versus frequency plot and is shown to conform to the design.

binary to CSD conversion and subsequent reduction in add operations. The ternary element –1 is represented as $\overline{1}$ in Figure 5.

For the fractional multiplier, starting from the rightmost, replace the first group of 1s (2^{-14} to 2^{-11}) by a combination of {–1, 0, 1}. The four 1s are combined and $\overline{1}$ is placed at the rightmost bit position (2^{-14}), zeros at the remaining position (2^{-13} to 2^{-11}), and a 1 in the 1 bit position to the left of this group (2^{-10}). This procedure is repeated for subsequent groups of ones already present or created from previous groupings. A similar procedure is shown for the integer multiplier. The reduction in add operations is two in the fractional case and four in the integer case. This reduction is multiplier-dependent and effective only when there are a number of groups of consecutive ones in the multiplier. The CSD format produces the same results for multiplies or divides with a slight reduction in CPU overhead. The introduction of the CSD format introduces a small change in the way Horner's algorithm is implemented. Because there would be a ternary set, all –1s would be subtracts and all 1s would continue to remain adds. This subtract, when necessary, would merely replace the add operation implemented at the end of each step.

IMPLEMENTATION ON THE MSP430

Horner's method works most efficiently on microcontrollers that support single-cycle add and shift operations. For example, the MSP430 CPU supports this requirement and also performs single-cycle register-register move operations, enabling fast multiply. A section of the MSP430 microcontroller's assembly code is shown in Listing 1. Table 1 shows a comparison of the Horner's method, Horner's method using CSD, existing algorithms, and floating-point C library. Performance measures such as code size, CPU cycles, and the final result have been shown for a standard integer-real multiply. Results for filter implementation using Horner's method using CSD are implemented. An approximate frequency response using a simple square and add operation in the filtered signal across various frequencies is shown in Figure 6 and Figure 7.

Table 1 shows the comparison of Horner's method and its variation to alternate methods for an integer-real multiply of the number 711 and the real multiplier 14.98789. The number 14.98789 is first scaled down by 16 (right shift by four) to get 0.936743125. The result is then scaled up by the same number 16 (left shift by four). The result of Horner's method is the closest to the floating-point library implementation losing only the fractional part of the final result. This result is as good as it can get considering its implementation is on a fixed-point processor. The existing method, although generic in

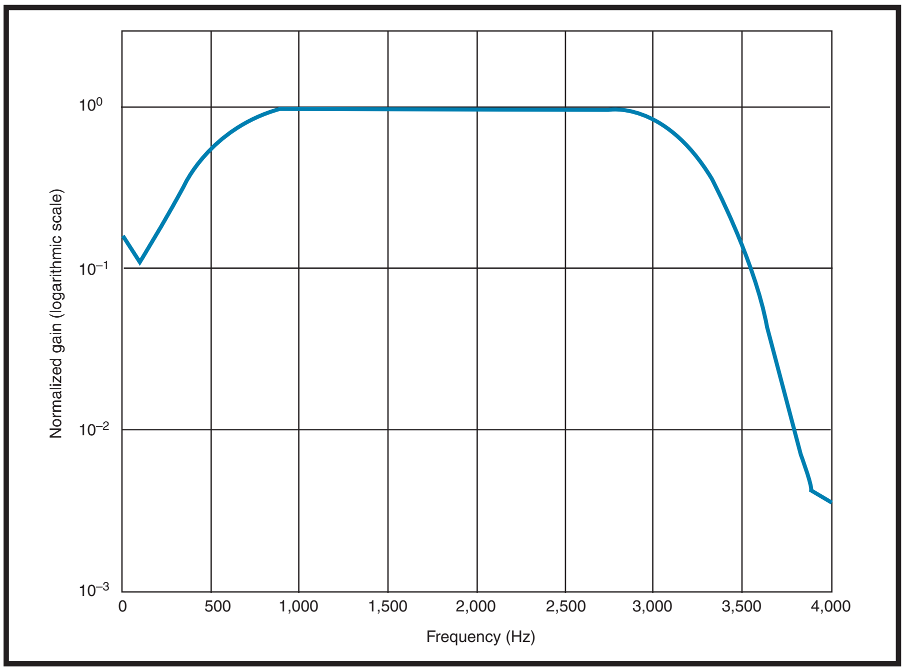

Figure 7—When Horner's method is used to implement a band-pass filter to extract the voice band of 300 to 3,400 Hz, the response is depicted as a gain versus frequency plot and it is shown to conform to the design.

implementation, loses precision due to finite-word length effects at the start of the multiply. To be fair to the integer-integer multiply algorithm, both cases of rounding up to 15 and rounding down to 14 are shown. The multiplier, even if rounded-up has significant error (see Table 1).

Although Horner's method requires dedicated code for each multiplier, the accuracy it provides is too tempting to use it. The CSD format is just an additional option to further reduce the execution cycles and code size.

For real-time filtering operations, Horner's method is certainly preferred over the other method because it is at least three times faster than the existing methods. These methods have limited accuracy with the option of scaling and using 32-bit registers to improve accuracy. The floating-point library is almost not an option because it is almost 15 times slower. To exhibit the performance of filtering using Horner's method, two filter implementations have been shown. The first example is a FIR low-pass filter and the second is a FIR band-pass filter. In both cases, the normalized gain in dB versus frequency is shown. Limited frequency resolution has been chosen because the intent is to show the filter's performance for a pre-fixed frequency sweep. In a real-world application, frequency resolution does not have a role to play. The performance depends on the number of bits chosen for the coefficients and the filter order. For every increase in order, the increase in the number of CPU cycles can vary between 30 and 35 CPU cycles, which also includes memory updates. This is a huge savings when compared to a floating-point library implementation.

Figure 6 shows the response of a thirtieth-order low-pass filter with its cut-off set at 300 Hz at a sampling frequency of 4,000 Hz. The entire filter takes 1,030 CPU cycles with 31 integer-real multiplies and 30 memory-memory moves at each stage and occupying a code size of 1,696 bytes. The MSP430 microcontroller supports a real-time operation for this case. If a floating-point library was used instead, the cycle count would be a minimum of 14,000 cycles preventing any real-time filtering.

Figure 7 shows an implementation of a twentieth-order band-pass filter specifically intended to limit the frequencies to the voice band, ideal for speech applications. The passband is set from 300 to 3,400 Hz with a sampling frequency set at 8,000 Hz. The total instruction cycles in this case is just 677 CPU cycles with a code size of 1,110 bytes. Both these filters are very close to their respective floating-point implementations.

ACHIEVE EFFICIENCY

This article focused on efficient multiplication and division of microcontrollers without a hardware multiplier module. The design steps to implement Horner's algorithm were shown with examples. The CSD format was introduced and its effects on reduction of steps was portrayed. This method not only showed superior performance, but also removed the fear of CPU overhead on the real-time implementation of digital filters. The number of CPU cycles and code size entirely depends on the resolution chosen for the filter coefficients. There cannot be a compromise on the integer part, but fewer resolutions for the fractional part can significantly reduce the CPU cycles with a compromise to performance. This method is certainly not limited to processors without a hardware multiplier, instead it paves the way for integer-real multiply on fixed-point machines. With memory getting cheaper by the day, code size can never pose a limitation for such a powerful scheme!

Kripasagar Venkat (kripa@ti.com) is an applications engineer for MSP430 microcontrollers at Texas Instruments. He holds a master's degree from the University of Texas at Dallas with an emphasis on digital signal processing. His current interests include signal processing and filter design for low-power applications.

PROJECT FILES

To download code, go to ftp://ftp.circuitcellar.com/pub/Circuit_Cellar/2008/212.

REFERENCES

[1] S. K. Mitra, *Digital Signal Processing: A Computer-Based Approach*, McGraw-Hill, New York, NY, 2001.

[2] C. Hamacher, Z. Vranesic, and S. Zaky, *Computer Organization*, McGraw-Hill, New York, NY, 1990.

[3] R. M. Hewlitt & E. S. Swartzlantler, "Canonical Signed Digit Representation for FIR Digital Filters," 2000 IEEE Workshop on Signal Processing Systems, Oct. 11 to 13, 2000.

SOURCE

MSP430 Family of microcontrollers
Texas Instruments, Inc.
www.ti.com