8-2010

Synthesizing Optimal Fixed-point Arithmetic for Embedded Signal Processing

Kenneth J. Hass
Bucknell University, kjh016@bucknell.edu

Synthesizing Optimal Fixed-Point Arithmetic for Embedded Signal Processing

K. Joseph Hass
Electrical Engineering
Bucknell University
Lewisburg, PA 17837
Email: k.j.hass@bucknell.edu

***Abstract*— Using fixed-point arithmetic rather than floating-point for data processing can significantly reduce the cost and power consumption of embedded systems. Unfortunately, this also shifts the burden of managing the data representation from run time to compile time, and in many cases the task of compile-time optimization must be done manually. A number of attempts have been made to formalize this process, and fixed-point methods have even been codified into an industry standard for a popular hardware-definition language, VHDL, in recent years. While the standard fixed-point libraries are certainly correct in the strict sense, they overlook an important practical consideration and may often produce results that are far from optimal. This paper discusses methods for maximizing the efficiency of fixed-point operations by careful use of the standard libraries.**

I. Introduction

Embedded digital signal processing applications, where there is a strong incentive to reduce the cost and power consumption of the computing resources, often use fixed-point arithmetic rather than floating-point. For a given target system the use of fixed-point arithmetic can increase the data processing rate by two orders of magnitude or more [1]. When used in programmable hardware, such as FPGAs, fixed-point arithmetic also allows the bit width of each variable to be optimized for the dynamic range and accuracy requirements of the algorithm, which in turn means that the utilization of hardware resources can also be optimized. Unfortunately, the use of fixed-point arithmetic still depends on a variety of ad hoc methods and tools along with considerable manual intervention. Recent attempts to standardize and formalize fixed-point arithmetic include standard libraries for VHDL, a design language codified as IEEE standard 1076 [2]. This paper discusses how these libraries can be used most effectively, and how a simple restriction on the input data values is a key factor in this strategy.

II. Fixed-Point Numbers

Any nomenclature for describing fixed-point binary numbers must, at a minimum, specify the location of the binary point. As with decimal numbers, bits to the right of the decimal point represent the fractional part of a value while bits to the left represent the integer part. Several systems of notation have been used in practice, and the simplest of these is commonly called the *Q* notation [3]. The format of each quantity is designated as *Qf*, where *f* is simply the number of bits to the right of the binary point. If we assume that there is also an implied sign bit then the total number of bits in the data word is just $f+1$, and the range of real values that can be represented is approximately $-1 < x < 1$. A common example is the *Q15* format used with 16-bit processors. This approach has been used successfully since the early years of digital computing [4].

An enhanced version of the *Q* format allows for values greater than 1 by specifying some number, *i*, of integer bits with a *Qi.f* notation [5]. Note that the *Q* format assumes that each value has exactly one non-redundant sign bit, which can be an undesirable simplification for processors that use a fixed word length. In this case a third field is often added to the format notation to indicate the number of redundant sign bits, resulting in descriptors such as the *(S/I/F)* notation system [6], [7]. This is the notation that will be used for this paper, and we will say that a signed, fixed-point value *A* has S_A sign bits, I_A bits representing the integer portion of the value, and F_A bits representing the fractional portion. The word length for *A* is then simply $S_A + I_A + F_A$. Note that if $S_A > 1$ then the additional sign bits are redundant and must have the same value.

The latest revision of the VHDL standard has formalized the definition of fixed-point values and provides overlaid arithmetic operators [2]. Fixed-point data types in VHDL make clever use of the range specifier for bit vectors to embed information about the data format. For example, an 8-bit two's-complement fixed-point number with 1 sign bit, 3 integer bits, and 4 fraction bits would be declared as:

```
signal MySignal: sfixed (3 downto -4);
```

so that $-8.0 \leq$ `MySignal` ≤ 7.9375 with a resolution of 0.0625. Using the *(S/I/F)* notation we say that this variable has a (1/3/4) format. The VHDL standard also defines arithmetic operators for the `sfixed` datatype. Of particular interest here is how these standard operators determine the required number of integer bits in the result, and how this decision can have a detrimental effect on the synthesized hardware.

III. The Most-Negative Number

When performing binary arithmetic there may be a strong temptation to assume that the normal rules of integer arithmetic will always apply, and to act accordingly. One of these fundamental rules states that for every number *a* in the set of

978-1-4244-7773-9/10/$26.00 ©2010 IEEE

integers $\mathbb{Z}$ there must be a solution to the equation $a + x = 0$ such that $x \in \mathbb{Z}$ [8]. In simple terms, for every positive value in the set there must be a corresponding negative value.

When we chose to represent integers in two's-complement binary form using a finite number of bits than this rule no longer applies. Unlike a sign-magnitude representation which may have both a positive zero and a negative zero, any set of two's-complement values will have just one representation for the integer zero, which consists of all of the bits being zero. Since the sign bit of integer zero is zero this value consumes one of the possible representations for positive values, so there must always be one negative value that has no corresponding positive value. This value is the *most-negative number* (MNN).

For example, a 4-bit two's-complement integer can have seven positive values (0001_2 through 0111_2) and one representation for zero (0000_2). There are eight negative values, 1111_2 through 1000_2, that represent the decimal integers -1 through -8 respectively. Performing unary negation on the most-negative number cannot possibly return a valid 4-bit two's-complement value for $+8$ since there is no such representation. Unfortunately, negating the MNN returns the same value, the MNN itself.

The standard fixed-point arithmetic packages defined in the VHDL-2008 standard allow for the possibility that the operands for any calculation may include the MNN. As we will see, this can lead to a significant loss in precision (if the word length is fixed) or an undesirable addition of logic resources and power in order to maintain a desired level of precision.

IV. Fixed-Point Addition and Subtraction

In general, binary addition and subtraction must allow for the possibility of an overflow or underflow from the most significant bit. The number of integer bits in the sum or difference must therefore be greater by one than the largest number of integer bits in either operand:

$$I_{A+B} = \max(I_A, I_B) + 1$$
$$I_{A-B} = \max(I_A, I_B) + 1$$

This well-known result can lead to unexpected behavior when we include the most-negative number in the allowed set of operands. For example, unary negation is commonly performed by taking the one's-complement (i.e. inverting) the operand and adding a 1 to the least significant bit. In most cases the number of integer bits in the negated value will be the same as in the original operand, but if the operand is the MNN then we must *increase the word length* to accommodate the result. If we are using 4-bit signed integers and wish to negate 1000_2 (-8) the result cannot be correctly represented with four bits, and we must prepend one bit to obtain 01000_2 $(+8)$. This situation also arises when finding the absolute value of the MNN. Consequently, the VHDL-2008 operators for unary negation and absolute value have the following rules:

$$I_{-A} = I_A + 1$$
$$I_{|A|} = I_A + 1$$

If we consider the case where $I_B \geq I_A$ we find that the number of integer bits required to represent $A + (-B)$ is *one greater* than the number of integer bits required to represent $A - B$. Likewise, the representation for $-(-B)$ must have *two* more integer bits than *B* itself (see [2], p. 524). In most cases the added integer bits carry no useful information and are in fact redundant sign bits, and it is only when *B* is the MNN that they may be needed. Unfortunately, these added bits require very real hardware resources for their storage and computation.

A further consideration for addition arises when calculating the sum of N values. The final sum will require no more than $\log_2 N$ additional integer bits to ensure that the result is properly represented. However, the result produced by a synthesizer depends largely on how the summation is coded, and as many as $N-1$ integer bits may be added if the N values are summed into a simple accumulator. Recognizing this situation can be difficult for any optimization tool, so the responsibility for writing HDL that will infer the desired operators falls on the designer, as we will see in the example below.

V. Fixed-Point Multiplication

A generally accepted rule for binary multiplication is that the number of bits in the product is equal to the sum of the number of bits in the two operands. This maxim may be a vestige of our early experience programming conventional processors with a fixed word length, where the product of two "single precision" integers is expected to be a "double precision" result, and it certainly holds true for unsigned operands. If we consider two unsigned four-bit integers their maximum decimal value is 15 and their product is 225 which does indeed require 8 bits to represent (1110_0001_2). This rule will also apply to the integer portion of fixed-point numbers:

$$I_{A\times B} = I_A + I_B$$

For an N-bit *unsigned* binary integer, the most positive value is $2^N - 1$. Taking the product of two such values we find that

$$A \times B = (2^N - 1)(2^N - 1) = 2^{2N} - 2^{N+1} + 1$$
$$< (2^{2N} - 1) \text{ for } N \geq 1$$

We observe that the magnitude of the product is strictly less than $(2^{2N} - 1)$ for all interesting values of N so, as expected, the product is properly represented in a $2N$-bit unsigned representation.

On the other hand, an N-bit *two's-complement* representation reserves at least one bit to indicate the sign of the number. As a result the most positive value is $2^{N-1} - 1$ for signed numbers. Taking the product of two such values we find that

$$A \times B = (2^{N-1} - 1)(2^{N-1} - 1) = 2^{2N-2} - 2^N + 1$$
$$< (2^{2N-2} - 1) \text{ for } N \geq 1$$

This leads to the surprising conclusion that the result of this two's-complement multiplication can be correctly represented

in a $(2N-1)$-bit word ($2N-2$ bits for the magnitude of the product plus a sign bit). As a general rule, the product of an N-bit two's-complement value multiplied by an M-bit two's-complement value will require $M+N-1$ bits, one fewer than the product of two unsigned numbers of the same size. We can easily rationalize this rule by observing that each of the operands contributes a sign bit while only one sign bit is needed in the product.

For example, suppose we multiply 0111_2 (+7), which is the most-positive integer we can represent in a 4-bit signed word, by itself. The result will be 0110001_2 (+49) and can be correctly represented using 6 bits for the integer itself and one additional bit for the sign.

Unfortunately, there is an exception to the signed multiplication rule developed above. For an N-bit two's-complement representation, the most negative value is -2^{N-1}. Taking the product of two such values we find that

$$A \times B = (-2^{N-1})(-2^{N-1}) = 2^{2N-2} \not< (2^{2N-2} - 1) \text{ for any } N \geq 1$$

Now the result is not strictly less than 2^{2N-2} and cannot be accommodated in a word length of $2N-1$ bits. Therefore, if we allow the possibility that both operands to a multiplication can be the MNN then the computation of the product must provide one more bit in the result than would be necessary if this special case could be avoided.

VI. Digital Filter Example

As an example of optimizing fixed-point arithmetic, consider a simple finite-impulse response (FIR) filter with 8 taps. The output of the filter at for any given sampling time can be described by the equation

$$Y = \sum_{i=0}^{K} A_i X_i$$

where K is the number of taps in the filter, Y is the filter output, X is the set of K input samples, and A is a set of filter coefficients. The straightforward implementation of the filter in VHDL is based on a single line of code that captures the filter equation:

```
Y <= A0*X0+A1*X1+...+A6*X6+A7*X7;
```

Suppose that the coefficients and input samples are represented as 12-bit signed values, with one bit reserved for the sign and 11 bits of fraction information. Using the *SIF* notation this is a (1/0/11) format, while in VHDL it would be declared as type `sfixed (0 downto -11)`. Note that the range of decimal values that can be represented is

$$-1 \leq X \leq 1 - 2^{-11}$$

The product of an input sample and a coefficient, both using this format, must be strictly less than 1 *unless* both the coefficient and the input sample are exactly equal to -1. Allowing for this highly unlikely case requires that we provide for an integer bit in the product to express the proper two's-complement representation of $+1$. This is exactly the assumption that is made by the VHDL fixed-point package, which requires that the format of the product be (1/1/22).

TABLE I
Synthesis Results for 8-tap Filter

Implementation	LUTs	Critical Path
Default VHDL	196	20.05 ns
Discard Y MSBs	181	19.65 ns
...and group adders	178	11.46 ns
Discard product MSB, group adders	172	11.51 ns

After all of the products are computed they are summed, and the typical approach used by logic synthesizers is to add any two of the products and form a partial sum. The remaining product terms are then added, one at a time, to the partial sum until $K-1$ additions have been performed. Using the VHDL `sfixed` datatype requires that each of these additions also increments the number of integer bits in the data format, so the required format for Y becomes `sfixed(8 downto -22)` or (1/8/22) and consumes a total of 31 bits. However, it is easy to see that the sum of eight values, each strictly less than 1, must have a value less than 8 and can be correctly represented with just 3 integer bits. In other words, the casual use of the `sfixed` operators results in the inclusion of 5 redundant sign bits that will carry no useful information.

A. Synthesis Results

This straightforward implementation of the filter was synthesized to the target architecture of a commercial FPGA using the FPGA vendor's tools, with results shown in the first entry in Table I. The chosen FPGA provides very fast multiplier macrocells, and 8 of these were used in the filter while the adders were constructed from look-up table (LUT) logic blocks. The critical path delay from the filter inputs to the output is shown in the final column, where the filter has been placed between register banks so delays associated with chip input/output pads is not a factor.

Since we know that the result contains redundant sign bits, an easy optimization step is to simply discard the 5 most-significant bits (MSBs) of the filter output. The synthesis tool should then recognize that all of the logic elements used to calculate the discarded bits can also be pruned away, resulting in an overall simplification of the circuit. The synthesis results for this change are shown in the second entry of Table I, where we see that the number of LUTs is reduced by 7.7% and the critical path delay is shortened by 2%.

A more significant improvement can be obtained by also forcing the synthesizer to sum the products using an adder tree, where the 8 products are first grouped into four pairs of products and each pair is added. The format of each of these partial sums is (1/2/22). The partial sums are then grouped into two pairs and the pairs are again added, with a resulting format of (1/3/22). The final pair of partial sums is now added and the format of the final sum is (1/4/22). The VHDL code can be easily modified to enforce this sequence of operations by grouping the operands with parentheses:

```
Y <= ((A0*X0+A1*X1)+(A2*X2+A3*X3))
     +((A4*X4+A5*X5)+(A6*X6+A7*X7));
```

In this case there is just 1 redundant sign bit, the one contributed by the multiplication operator, which can be discarded. This implementation was synthesized to give the results in the third row of the table. We have reduced the number of LUTs by 9.2% and trimmed the critical path delay by nearly 43%.

As a final experiment, the MSB of each product term was discarded before the partial sums were computed. An adder tree was again used to accumulate all of the products. Since both the multiplications and the additions were optimized to discard redundant sign bits there was no need to discard any sign bits from the final output value, and this is the limit of optimizations that can be made without knowledge of the coefficient values or input data statistics. As we can see from the last entry in Table I the critical path delay did not change significantly, which is to be expected because discarding a redundant bit in the products has no real affect on the path delay for the actual information bits. However, there was a small decrease in the required logic resources because the width of each adder could be reduced by one bit.

B. Other DSP Structures

The FIR filter example presented above emphasizes optimizations related to the summation, but other algorithms will be more strongly affected by the addition of redundant bits during multiplication. The Fast-Fourier Transform (FFT), for example, requires repeated multiplications by complex coefficients to obtain a result. Calculating an N-point FFT requires the use of $\log_2 N$ butterfly stages, and all but the first of these includes a multiplication by a non-trivial complex coefficient [9]. Thus, using the standard `sfixed` operators to calculate a 1024-point FFT would accumulate at least 9 redundant sign bits in the result. The multipliers and adders in the later butterfly stages would see progressively wider operands, requiring additional logic resources. If the FFT is pipelined then the pipeline registers will also be unnecessarily wider, consuming further resources with no return in useful information.

Another common element in digital signal processing, the infinite impulse response (IIR) filter, is also quite sensitive to numerical problems. As the name implies, the response of an IIR filter to an impulse input decays over time but, at least theoretically, will last indefinitely. This behavior is due to the presence of a feedback path from the filter output back to its input. Therefore, if the filter adds unnecessary bits to the output word then the redundant bits will accumulate recursively and render the filter unusable.

VII. Preventing the Most-Negative Number

A relevant question at this point is whether it is reasonable and practical to assume that the MNN can never appear as an operand in our calculations. Suppose first that we can exclude the MNN from the set of primary inputs to our processing block. This is easily accomplished for streaming data inputs by comparing the incoming values to the MNN and incrementing them if that value appears. In a well-designed signal processing application the input data values rarely approach the most positive or most negative limits so this restriction is typically of little consequence.

Now, if the MNN is excluded then the unary negation and absolute value operators will never return the MNN and will not require an additional integer bit. Similarly, if we account for the normal possibility of overflow from an addition or underflow from a subtraction then these operators can not return the MNN. A few trial calculations will also show that a signed multiplication cannot result in the MNN, even if one of the operands is a MNN. Furthermore, if neither the multiplier nor the multiplicand are the MNN then the product can be properly represented using one *fewer* bit than the sum of the bits in the operands.

Consequently, if the MNN is removed from the set of possible input values then we can be confident that it will never be produced as an intermediate result and our processing algorithms can be optimized accordingly.

VIII. Conclusion

Evolving standards for fixed-point computation in hardware description languages are replacing the variety of ad hoc methods that were developed over the past few decades. These standards will allow digital designers to employ fixed-point techniques efficiently, but optimizing the area, power consumption, and computational throughput of fixed-point arithmetic still requires attention to detail. Proper ordering of additions and subtractions can significantly improve the design with no loss of precision or dynamic range, and in most cases the redundant sign bit produced by the standard multiplication function can be discarded. Removing the most-negative number from the set of possible operands is a key step in achieving these improvements.

References

[1] K.-I. Kum, J. Kang, and W. Sung, "AUTOSCALER for C: An optimizing floating-point to integer C program converter for fixed-point digital signal processors," *IEEE Trans. Circuits Syst. II*, vol. 47, no. 9, pp. 840–848, Sept. 2000.

[2] "IEEE standard VHDL language reference manual," IEEE Std 1076-2008, Jan. 26, 2009.

[3] W. Hohl and C. N. Hinds, "A primer on fractions," *IEEE Potentials*, pp. 10–14, March/April 2008.

[4] J. H. Wilkinson, *Rounding Errors In Algebraic Processes*. Prentice-Hall, 1963.

[5] J.-H. Sohn, J.-H. Woo, M.-W. Lee, H.-J. Kim, R. Woo, and H.-J. Yoo, "A 155-mW 50-Mvertices/s graphics processor with fixed-point programmable vertex shader for mobile applications," *IEEE J. Solid-State Circuits*, vol. 41, no. 5, pp. 1081–1091, May 2006.

[6] K. J. Hass, D. H. Lenhert, and N. Ahmed, "On a microcomputer implementation of an intrusion-detection algorithm," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. ASSP-27, no. 6, pp. 782–789, Dec. 1979.

[7] J. E. Simpson, "A block floating-point notation for signal processes," Sandia National Laboratories, Tech. Rep. SAND79-1823, 1979.

[8] T. W. Hungerford, *Abstract Algebra: An Introduction*, 2nd ed. Saunders College Publishing, 1997.

[9] N. Ahmed and K. R. Rao, *Orthogonal Transforms for Digital Signal Processing*. Springer-Verlag, 1975.