

A Hybrid Number System And Its Application In FPGA-DSP Technology

Reza. Hashemian and Bipin Sreedharan
Northern Illinois University
Department of Electrical Engineering
reza@ceet.niu.edu

Abstract

New advancement in FPGA-DSP technology demands a new number system as well as a new data treatment that best utilizes the strengths and avoid the shortcomings of the new technology. To achieve this goal, here we introduce a customized floating-point number system that it can be reformatted as a fixed-point number when needed. In addition, a scheme is proposed for the use of both formats in DSP applications, such as in digital filter design

1. Introduction

New advancement in FPGA-DSP technology demands a new number system that adjusts to its strength and avoids its limitations. Rapid hardware reconfigurability for different applications and architectures is a major strength in FPGA technology, whereas, relatively high cost of hardware is one of its limitations. Hence an ideal number system can be a hybrid one that is adjusted based on applications, or even different phases of an application.

Various representations of number systems are used in digital systems [1]. In particular, fixed-point numbers are predominantly used in systems that have limited hardware and are relatively slow, such as FPGAs. Although the arithmetic operations become much simpler and faster with fixed-point numbers, they suffer from limited range and/or precision. Computations must be “scaled” to ensure that values remain representable and that they do not lose too much precision. A major drawback in fixed-point representation is that it allows sign extension that leads to space reduction needed for the magnitude, and hence, causing the truncation error to increase. In fact, in fixed-point representation word size, dynamic range, and precision are tightly related. For instance, with fixed word size the precision can be restored only when the dynamic range is reduced, and visa versa.

Floating-point number system, on the other hand, offers a wide dynamic range with maximum precision. In floating-point representation the entire word is used for precision (magnitude), with the expense of an exponent, of course. Here we don't deal with any sign extension.

The down side of a floating-point number is that it consists of multiple parts, and in an arithmetic operation each part needs to be differently treated, and this leads to reduction in the operational speed.

In the newly advanced FPGA technology where the emphasis is shifting toward combined FPGA-DSP systems [2] we need to think of a more customized number system that best fits in this new development. Ideally, this number system should be able to provide enough dynamic range and precision in order to perform rather complex arithmetic operations, and at the same time, it must be simple enough to keep the high-speed provided by fixed-point arithmetic. To gain the benefits of both worlds (fixed-point and floating-point numbers) we need to present a hybrid scheme that can take advantage of the field programmability of the new technology, and could be adjusted based on different applications. This hybrid number can take the form of a fixed-point number when 1) the size of the number stays relatively fixed, and 2) the operation needs decimal point alignment, such as addition and subtraction. On the other hand, the hybrid number can take the form of a floating-point number when the size and the magnitude of the number substantially changes in the process, and decimal point alignment is not essential, such as in multiplication. Another property expected from hybrid number system is to gain speed in arithmetic operations. There is certainly a price to pay for converting between two modes of a hybrid number, but it ultimately pays off, as long as the overall time saved in the process surpasses the time delay and extra hardware used for the conversion operation.

2. Floating-Point Number Representation

A binary-based floating-point number has three components: the sign s , the magnitude (mantissa) m , and the exponent e . For example, number x can be represented as

$$x = s \times m \times 2^e$$

The magnitude indicates a positive or negative number in signed-digit format, with m being its magnitude and s its sign. The exponent specifies the range, and it is typically based on the biased

representation that converts signed numbers into unsigned numbers [1].

There are two representation formats in IEEE standard for binary floating-point numbers, formally known as “ANSI/IEEE Std 754-1985”. The first representation is short, or single-precision, format being 32 bits wide, and the second representation is long, or double-precision, version that requires 64 bits, as shown in Fig. 1. The two formats have 8-and 11-bit exponent fields and use exponent biases of 127 and 1023, respectively [1]. The magnitude is in the range $[1, 2)$, with its single whole bit, which is always 1, removed and only the fractional part shown. The notation “23 + 1” or “52 + 1” for the length of the magnitude is meant to explicate the role of the hidden bit, which does contribute to the precision without taking up space.

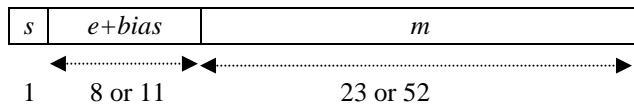


Fig. 1 – Typical Floating-Point number representation

3. Customized Hybrid Number System Representation

In our proposed representation we have adopted a slightly different floating-point number system to fit with the requirements and limitations of FPGA-DSP applications. This number system is very similar to ANSI/IEEE Std, except for the data size and the magnitude range which are different. Due to relatively costly hardware in FPGA technology the data size here is reduced from 32-bit, in ANSI/IEEE Std, to 24-bit, with “16 + 1” for the length of the magnitude, and 7 for the exponent field, as shown in Fig. 2.

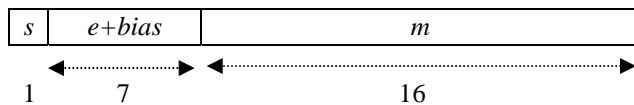


Fig. 2 – Normalized Floating-Point number representation used for FPGA-based DSP.

Secondly, the range of the magnitude is changed to $[0.5, 1)$, with its MSB removed (“hidden 1”) from the fraction. The advantage of this range is its desirability for DSP applications, such as digital filter coefficients, FFT and DCT coefficients, and so on, where the magnitudes are predominantly less than one.

A third criterion in this representation is its hybrid nature that provides a flexible formatting that is useful in many applications. As mentioned earlier, in its floating-point format the representation provides a 24-bit number.

In its fixer-point representation, however, we can have two formats: the *short format*, which is 16-bit wide, and the *long format*, which is 32-bit wide. The re-configurability of the FPGA hardware is a key element that allows us to configure the right data format for the right operations. This is done through a carefully architected hardware design.

In short fixed-point representation, or *short format*, the number occupies either the portion of the magnitude of a floating-point number (16 bits), or it takes two joint bytes of “s + e” to still represent a 16-bit fixed-point number, as shown in Fig. 3. In both cases the number is represented in 2’s complement. For long fixed-point representation, or *long format*, we joint a pair of 16-bit magnitudes to form a 32-bit 2’s complement number, as shown in Fig. 4.

This new representation can certainly offer more effective arithmetic operations in wide variety of applications, such as digital signal processors, where both data formats can be used almost simultaneously, and hardware reconfiguration, adopted for the right number format, can be performed almost on the “fly”.

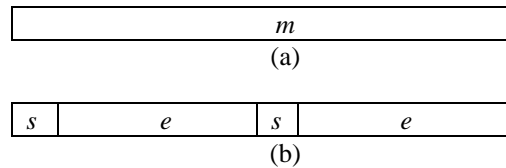


Fig. 3 – Two types of short fixed-point number formats

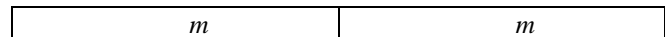


Fig. 4 – long fixed-point number format

Table 1 compares some of the important features of the ANSI/IEEE – Single precision format with the proposed Customized Hybrid number system.

Table 1

Properties	ANSI/IEEE – Single Precision	Customized Hybrid
Word size	32	24
Significant bits	23+1	16+1
Magnitude range	$[1, 2) = [1, 2 \cdot 2^{-23}]$	$[0.5, 1) = [0.5, 1 \cdot 2^{-17}]$
Exponent size	8	7
Exponent bias	127	63
Zero	$e = -127, m = 0$	$e = -63, m = 0$
Infinity/NaN	$e = 128, m = 0/-$	$e = 64, m = 0/-$
Min	2^{-126}	2^{-62}
Max	$< 2^{128}$	$< 2^{64}$

4. Efficient Floating-Point Arithmetic Operations

The major arithmetic operations involved in DSP applications are multiplication, and addition (or subtraction). In fixed-point arithmetic a multiplication takes much longer time than an addition, and hence multiply-accumulate (MAC) operations are more efficient and commonly used. In floating-point arithmetic, however, the case is slightly different, and although multiplication remains almost the same with exception of adding or subtracting the exponents, but addition (or subtraction) operation become more involved. Here the treatments applied to the magnitudes of the operands are different from those applied to their exponents. In our proposed method we present a scheme that optimally reduces the operational complexity in floating-point addition. In this scheme the adders are first clustered, then the input data (operands) are converted to fixed-point format, and finally the conventional fixed-point additions are performed in all adders in the cluster, as we will discuss next.

In a floating-point multiplication operation we only need to multiply the magnitudes and add the exponents. Evidently, due to the range $[0.5, 1)$ of the magnitude in both multiplicand and multiplier the range of the product become $[0.25, 1)$. This suggests that to maintain the same precision (16+1 bits) we need to temporarily append an extra bit to the magnitude, just in case, if we need to make a left shift for the precision adjustment. Evidently, this extra bit short lives, and it is needed only to perform the shift operation, if needed. As we notice, the multiplication in floating-point number system is very similar to that of fixed-point, except for performing two extra operations, one (possibly) shift in the magnitude of the product, and an addition performed on the exponents. For instance, suppose we need to multiply $x = -19.108$ by $y = 0.2793$ to get $z = x*y = -5.3369$. In the proposed floating-point $x = -1 \times 11E00 \times 2^5$, $y = 1 \times 131BA \times 2^{-1}$ and $z = -1 \times 1558F \times 2^3$. Now if we multiply the fraction and add the exponent we get $11E00 * 131BA = 558DCC00$, which is truncated to $558E$ with $e = 4$. For final result we need to do one shift to the right and reduce the exponent by 1 to get $z = -1 \times 1558E \times 2^3$. Notice that the difference is reduced to the LSB.

Floating-point addition/subtraction, however, proves to be more complex and time consuming. In the addition operation the two magnitudes need to be aligned first, and for this alignment the smaller (unsigned) number shifts to the right until the exponents of the two operands become identical. The two magnitudes are then added in a fixed-point adder, with again a possible shift. In this situation the exponent for the sum term takes the value of that of

the larger number. Other data treatments must also follow to ensure the correct result. For example, suppose four numbers $p = 25.802$, $q = -4.883$, $r = 0.9496$, and $s = -17.138$ are going to be added. Number p has definitely the largest exponent $e = 5$. Now if we make e the fraction all four numbers then their fractions are $P = 19CD4$, $Q = -4E20$, $R = F31$, and $S = -11235$. We then add them as fixed-point numbers to get $T = 4BB0$. Next, by making two shifts to the left we normalize the result to get $T = 12EC2$, and the exponent $e = 5-2 = 3$. This is precisely the result $t = 4.7306$.

Due to the significant difference in two operations, as stated before, we try to separate addition operations from multiplications, in our proposed method, as much as possible. Naturally, we are going to avoid the popular Multiply-Accumulate (MAC) operations practiced in conventional methods. In fact, we use floating-point arithmetic for multiplications, but for addition we switch to the modified fixed-point arithmetic adopted in this article. To make the operation effective we need to reduce the number of conversions from one system to another, and as few as possible. This is done in applications that clustering (additions or multiplications) is possible and presented as a non-penalized option. Hence, we can stay with one format for the number representation as long as we are operating within one cluster (additions or multiplications) and switch to the other format as required. We still need to present an efficient interface method that links the two clusters of operations and leads us to the final result. In this stage, we need to perform some special treatments that bring the two formats together, and prepare the resulting data in desirable format.

To make it more clear we take the example of an eight-tap FIR filter, as depicted in Fig. 5.

Figure 5 shows a direct form FIR structure [2], where, $x(n)$ is the input, $y(n)$ the output, R the registers, and C_0 to C_7 are the filter coefficients. Intersection AB separates the multiplications from the additions/subtractions. To smoothly perform the entire filter operation the data format is separated by the intersection AB. The data above the intersection AB is in floating-point format, and that below is in fixed-point format.

As discussed earlier, the input data $x(n)$ is in 24-bit floating-point format. The filter coefficients, C_0 to C_7 , are also in 24-bit floating-point format. The operation starts by getting delayed input signals, still in floating-point, from the registers R and multiplying them by the filter coefficients. The result is evidently in 24-bit floating-point format when it reaches to intersection line AB. In intersection AB the data must go through the following steps and convert:

1. The exponents of all the data (eight of them) entering AB intersection are compared and the

values, if needed, are changed to match the largest exponent e_{max} . Obviously, the magnitudes are also shifted accordingly to maintain the correct values.

- The adders only deal with the magnitudes, which are now converted to 16-bit $2s'$ -complement fixed-point numbers. Note that, once data is converted along the intersection line AB, no other conversion is needed as long as the addition process takes place in fixed-point arithmetic.
- In the final stage the response ($y(n)$) is converted into a floating-point number again to match the input signal format. In fact to perform the final conversion operation, the final summation, after minor adjustments, become the magnitude of $y(n)$, and e_{max} its exponent.

Figure 6 represents a 16-tap low-pass FIR filter, implemented on a Xilinx Virtex II FPGA. As shown, the frequency response of the filter, the input and the output signals are displayed. The time domain waveforms for the sample input and the output are also specified.

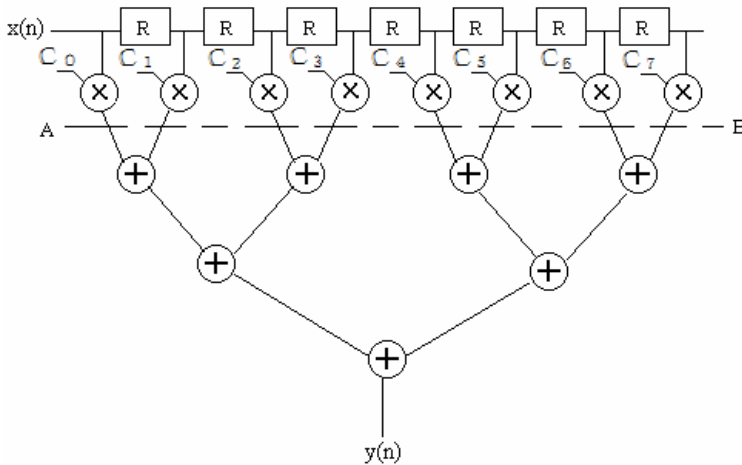
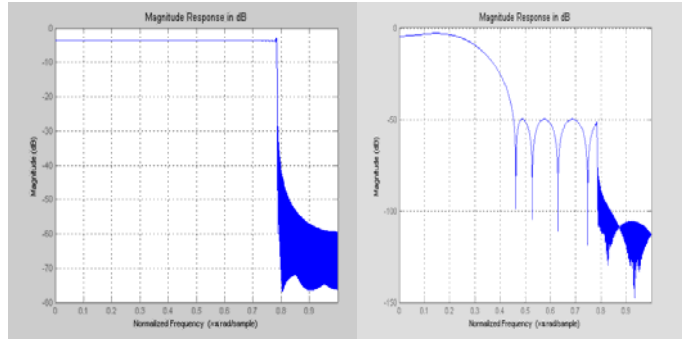
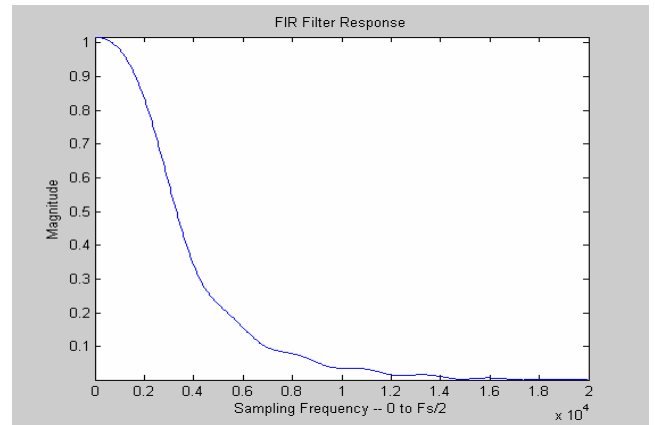
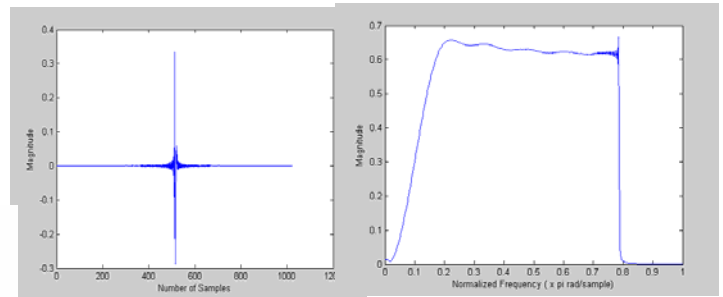


Fig. 5 – A direct form eight-tap FIR filter



Input Output

Fig. 6 (b) – Input and output frequency spectrums.



Input Output

Fig. 6 (c) – Input and output time domain signals.

More example designs for other types of FIR filters are shown in the Appendix.

5. Conclusion

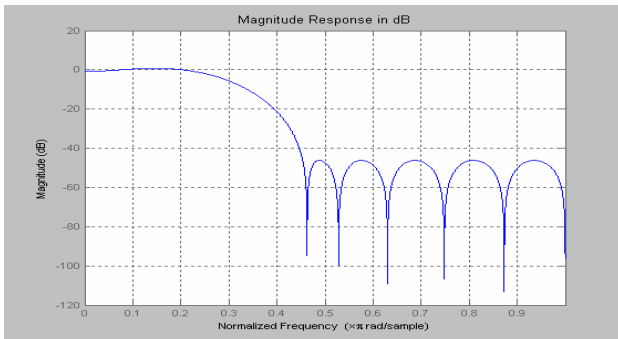


Fig. 6 (a) – Low-pass filter characteristic

A new hybrid number system is introduced that depending on the arithmetic operation (multiplication or addition) it can be reformatted from a floating-point to a fixed-point number and visa versa. This is certainly very desirable for the new advancement in FPGA-DSP technology where hardware reconfiguration can be performed even on “fly”. Examples of FIR digital filter designs clearly show the strength of the scheme. The main idea here is to be able to cluster the multiplications and additions separately, use different number formats for each cluster (floating-point for multiplication and fixed-point for addition), and make conversion once and at the end of the arithmetic operation.

6. References

- [1] B. Parhami, *Computer Arithmetic, Algorithms and Hardware Design*. Oxford University Press, 2000.
- [2] S. K. Mitra, *Digital Signal Processing*. Mc Graw Hill, 2001.
- [3] N. R. Scott, *Computer Number Systems & Arithmetic*. Prentice-Hall, 1985.

Appendix

Similar to the FIR filter discussed earlier (Fig. 6) Figures also show the design of Butterworth filters using 16-tap, implemented on a Xilinx Virtex II FPGA. Here again, the filters characteristics and the outputs for the same input signal are displayed.

BUTTERWORTH LOWPASS FILTER
Cut off Frequency – 5KHz
Sampling Frequency – 40KHz

Fig.7(a) – Filter Spectrum.

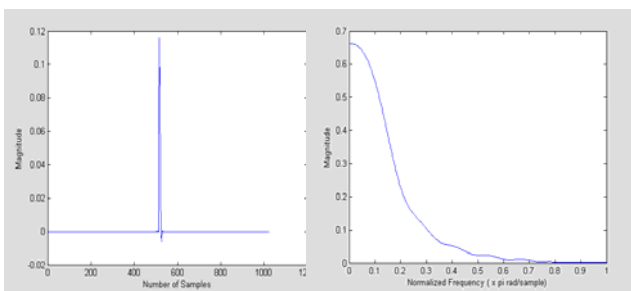
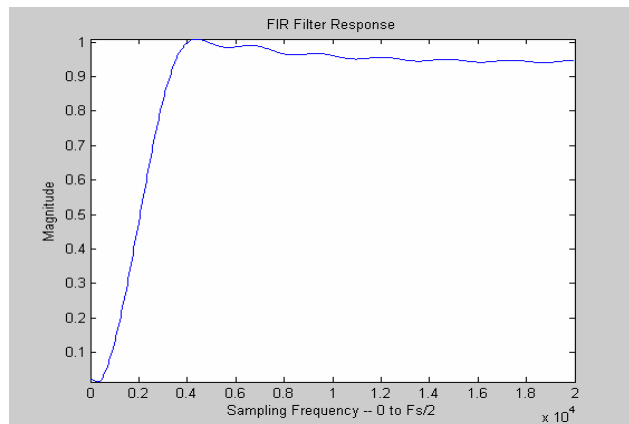


Fig 7(b) - Time response and frequency response of the output

BUTTERWORTH HIGHPASS FILTER



Cut off Frequency – 7KHz
Sampling Frequency – 40KHz

Fig.8(a) – Filter Spectrum.

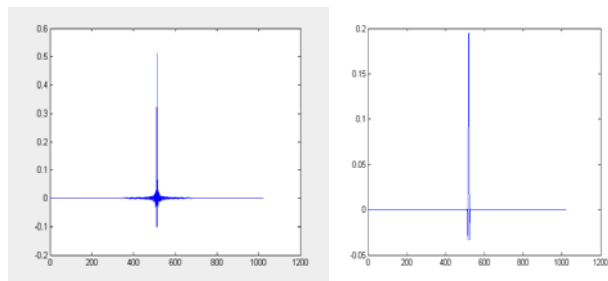
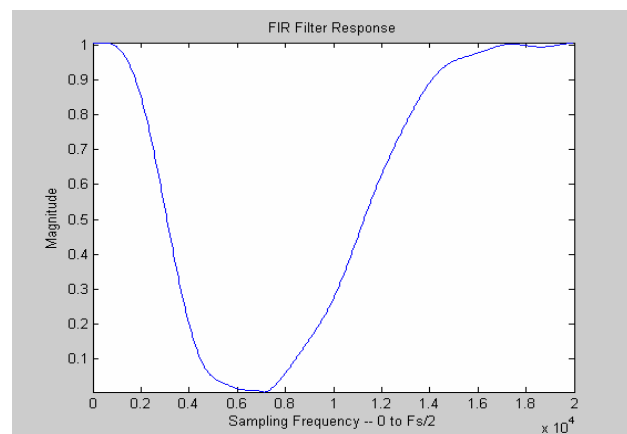


Fig 8(b) - Time response and frequency response of the output

BUTTERWORTH BAND STOP FILTER



Cut off Frequency – 5K to 25KHz
Sampling Frequency – 40KHz

Fig.9(a) – Filter Spectrum.

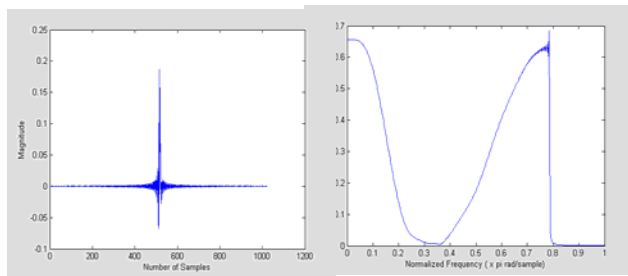
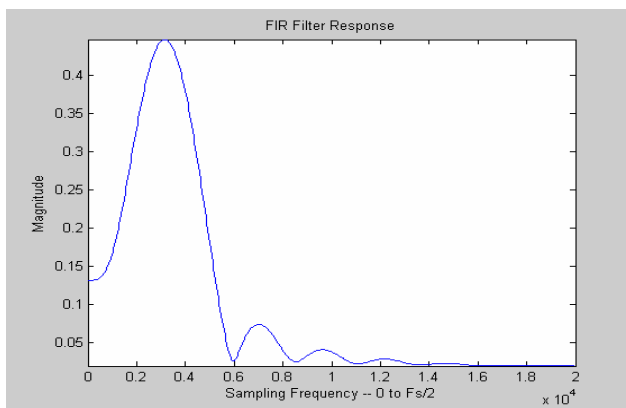


Fig 9(b) - Time response and frequency response of the output

BUTTERWORTH BANDPASS FILTER



Cut off Frequency – 5K to 7KHz
 Sampling Frequency – 40KHz

Fig.10(a) – Filter Spectrum.

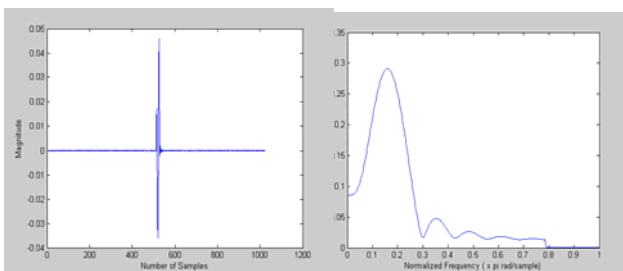


Fig10(b) - Time response and frequency response of the output