

The Role of Distributed Arithmetic in FPGA-based Signal Processing

Introduction

Distributed Arithmetic (DA) plays a key role in embedding DSP functions in the Xilinx 4000 family of FPGA devices. In this document the DA algorithm is derived and examples are offered that illustrate its effectiveness in producing gate-efficient designs.

Distributed Arithmetic Revisited

Distributed Arithmetic, along with Modulo Arithmetic, are computation algorithms that perform multiplication with look-up table based schemes. Both stirred some interest over two decades ago but have languished ever since. Indeed, DA specifically targets the sum of products (sometimes referred to as the vector dot product) computation that covers many of the important DSP filtering and frequency transforming functions. Ironically, many DSP designers have never heard of this algorithm. Inspired by the potential of the Xilinx FPGA look-up table architecture, the DA algorithm was resurrected in the early 90's and shown to produce very efficient filter designs [1].

The derivation of the DA algorithm is extremely simple but its applications are extremely broad. The mathematics includes a mix of boolean and ordinary algebra and require no prior preparation - even for the logic designer.

[1] Mintzer, L. "FIR filters with the Xilinx FPGA " FPGA '92 ACM/SIGDA Workshop on FPGAs pp. 129-134

Distributed Arithmetic at a Glance

The arithmetic sum of products that defines the response of linear, time-invariant networks can be expressed as:

equ. 1
$$y(n) = \sum_{k=1}^K A_k x_k(n)$$

where:

$y(n)$ = response of network at time n .

$x_k(n)$ = k th input variable at time n .

A_k = weighting factor of k th input variable that is constant for all n , and so it remains time-invariant.

In filtering applications the constants, A_k , are the filter coefficients and the variables, x_k , are the prior samples of a single data source (for example, an analog to digital converter). In frequency transforming - whether the discrete Fourier or the fast Fourier transform - the constants are the sine/cosine basis functions and the variables are a block of samples from a single data source. Examples of multiple data sources may be found in image processing.

The multiply-intensive nature of equ 1 can be appreciated by observing that a single output response requires the accumulation of K product terms. In DA the task of summing product terms is replaced by table look-up procedures that are easily implemented in the Xilinx configurable logic block (CLB) look-up table architecture.

We start by defining the number format of the variable to be 2's complement, fractional - a standard practice for fixed-point microprocessors in order to bound number growth under multiplication. The constant factors, A_k , need not be so restricted, nor are they required to match the data word length, as is the case for the microprocessor. The constants may have a mixed integer and fractional format; they need not be defined at this time. The variable, x_k , may be written in the fractional format as shown in equ. 2

$$\text{equ.2} \quad x_k = -x_{k0} + \sum_{b=1}^{B-1} x_{kb} 2^{-b}$$

where x_{kb} is a binary variable and can assume only values of 0 and 1. A sign bit of value -1 is indicated by x_{k0} . Note that the time index, n, has been dropped since it is not needed to continue the derivation. The final result is obtained by first substituting equ.2 into equ.1 -

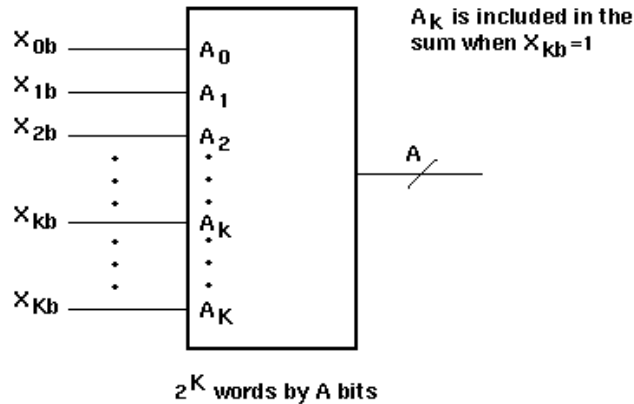
$$\text{equ.3} \quad y = \sum_{k=1}^K A_k \left[-x_{k0} + \sum_{b=1}^{B-1} x_{kb} 2^{-b} \right] = \sum_{k=1}^K x_{k0} \bullet A_k + \sum_{k=1}^K \sum_{b=1}^{B-1} x_{kb} \bullet A_k 2^{-b}$$

and then explicitly expressing all the product terms under the summation symbols:

$$\begin{aligned} \text{equ.4} \quad y = & - [x_{10} \bullet A_1 + x_{20} \bullet A_2 + x_{30} \bullet A_3 + \dots + x_{K0} \bullet A_K] \\ & + [x_{11} \bullet A_1 + x_{21} \bullet A_2 + x_{31} \bullet A_3 + \dots + x_{K1} \bullet A_K] 2^{-1} \\ & + [x_{12} \bullet A_1 + x_{22} \bullet A_2 + x_{32} \bullet A_3 + \dots + x_{K2} \bullet A_K] 2^{-2} \\ & \vdots \\ & + [x_{1(B-2)} \bullet A_1 + x_{2(B-2)} \bullet A_2 + x_{3(B-2)} \bullet A_3 + \dots + x_{K(B-2)} \bullet A_K] 2^{-(B-2)} \\ & + [x_{1(B-1)} \bullet A_1 + x_{2(B-1)} \bullet A_2 + x_{3(B-1)} \bullet A_3 + \dots + x_{K(B-1)} \bullet A_K] 2^{-(B-1)} \end{aligned}$$

Each term within the brackets denotes a binary AND operation involving a bit of the input variable and all the bits of the constant. The plus signs denote arithmetic sum operations. The exponential factors denote the scaled contributions of the bracketed pairs to the total sum. You can now construct a look-up table that can be addressed by the same scaled bit of all the input variables and can access the sum of the terms within each pair of brackets. Such a table is shown in fig. 1 and will henceforth be referred to as a Distributed Arithmetic look-up table or DALUT. The same DALUT can be time-shared in a serially organized computation or can be replicated B times for a parallel computation scheme, as described later. This ends our derivation of the DA algorithm, but, before we continue with our application examples, note the following observations.

DALUT Addressing



DALUT Content

0	0	If X_{0b} is least significant address bit
1	A_0	
2	A_1	A_K may be bipolar
3	$A_0 + A_1$	
4	A_2	
5	$A_0 + A_2$	
6	$A_1 + A_2$	
7	$A_0 + A_1 + A_2$	
8	A_3	
	\vdots	
	\vdots	

X7719

Fig. 1. The Distributed Arithmetic Look-up Table (DALUT)

The arithmetic operations have now been reduced to addition, subtraction, and binary scaling. With scaling by negative powers of 2, the actual implementation entails the shifting of binary coded data words toward the least significant bit and the use of sign extension bits to maintain the sign at its normal bit position. The hardware implementation of a binary full adder (as is done in the CLBs) entails two operands, the addend and the augend to produce sum and carry output bits. The multiple bit-parallel additions of the DALUT outputs expressed in equ. 4 can only be performed with a single parallel adder if this adder is time-shared. Alternatively, if simultaneous addition of all DALUT outputs is required, an array of parallel adders is required. These opposite goals represent the classic speed-cost tradeoff.

The Speed Tradeoff

Any new device that can be software configured to perform DSP functions must contend with the well entrenched standard DSP chips, i.e. the programmable fixed point microprocessors that feature concurrently operating hardware multipliers and address generators, and on-chip memories. The first challenge is speed. If the FPGA doesn't offer higher speed why bother. For a single filter channel the bother is worth it - particularly as the filter order increases. And the FPGA advantage grows for multiple filter channels.

Alas, a simple advantage may not be persuasive in all cases - an overwhelming speed advantage may be needed for FPGA acceptance. Is it possible to reach 50 megasamples/sec data sample rates? Yes but at a high cost in gate resources. The first two examples will show the end points of the serial/parallel tradeoff continuum.

The Ultimate in Speed

Conceivably, with a fully parallel design the sample speed could match the system clock rate. This is the case where all the add operations of the bracketed values (the DALUT outputs) of equ. 4 are performed in parallel. We can gain implementation guidance by rephrasing equ. 4, and to facilitate this process let us abbreviate the contents within each bracket pair by the data bit position. Thus

$$[x_{12} \bullet A_1 + x_{22} \bullet A_2 + \dots + x_{K2} \bullet A_K] \text{ reduces to } [\text{sum } 2]$$

and, similarly,

$$[x_{1(B-1)} \bullet A_1 + x_{2(B-1)} \bullet A_2 + \dots + x_{K(B-1)} \bullet A_K] \text{ reduces to } [\text{sum}(B-1)]$$

For B = 16, equ. 4 becomes:

$$\begin{aligned} \text{equ.5} \quad y = & - [\text{sum0}] \\ & + [\text{sum1}]2^{-1} \\ & + [\text{sum2}]2^{-2} \\ & \cdot \\ & \cdot \\ & \cdot \\ & + [\text{sum14}]2^{-14} \\ & + [\text{sum15}]2^{-15} \end{aligned}$$

The decomposition of Equ. 5 into an array of two input adders is given below:

$$\begin{aligned} \text{equ.6} \quad y = & - [\text{sum0}] + [\text{sum1}]2^{-1} + \{[\text{sum2}] + [\text{sum3}]2^{-1}\}2^{-2} \\ & + \{[\text{sum4}] + [\text{sum5}]2^{-1} + \{[\text{sum6}] + [\text{sum7}]2^{-1}\}2^{-2}\}2^{-4} \\ & + \{[\text{sum8}] + [\text{sum9}]2^{-1} + \{[\text{sum10}] + [\text{sum11}]2^{-1}\}2^{-2}\} \\ & + \{[\text{sum12}] + [\text{sum13}]2^{-1} + \{[\text{sum14}] + [\text{sum15}]2^{-1}\}2^{-2}\}2^{-4}\}2^{-8} \end{aligned}$$

Equations 5 and 6 are computationally equivalent, but equ. 6 can be mapped in a straight-forward way into a binary tree-like array of summing nodes with scaling effected by signal routing as shown in fig. 2. Each of the 15 nodes represents a parallel adder, and while the computation may can yield responses that include both the double precision (B+C bits) of the implicit multiplication and the attendant processing gain, these adders can be truncated to produce single precision (B bits) responses. The selection of binary value levels along the data path will be discussed later. We can, nonetheless, discuss some of the hardware needed to realize this computation.

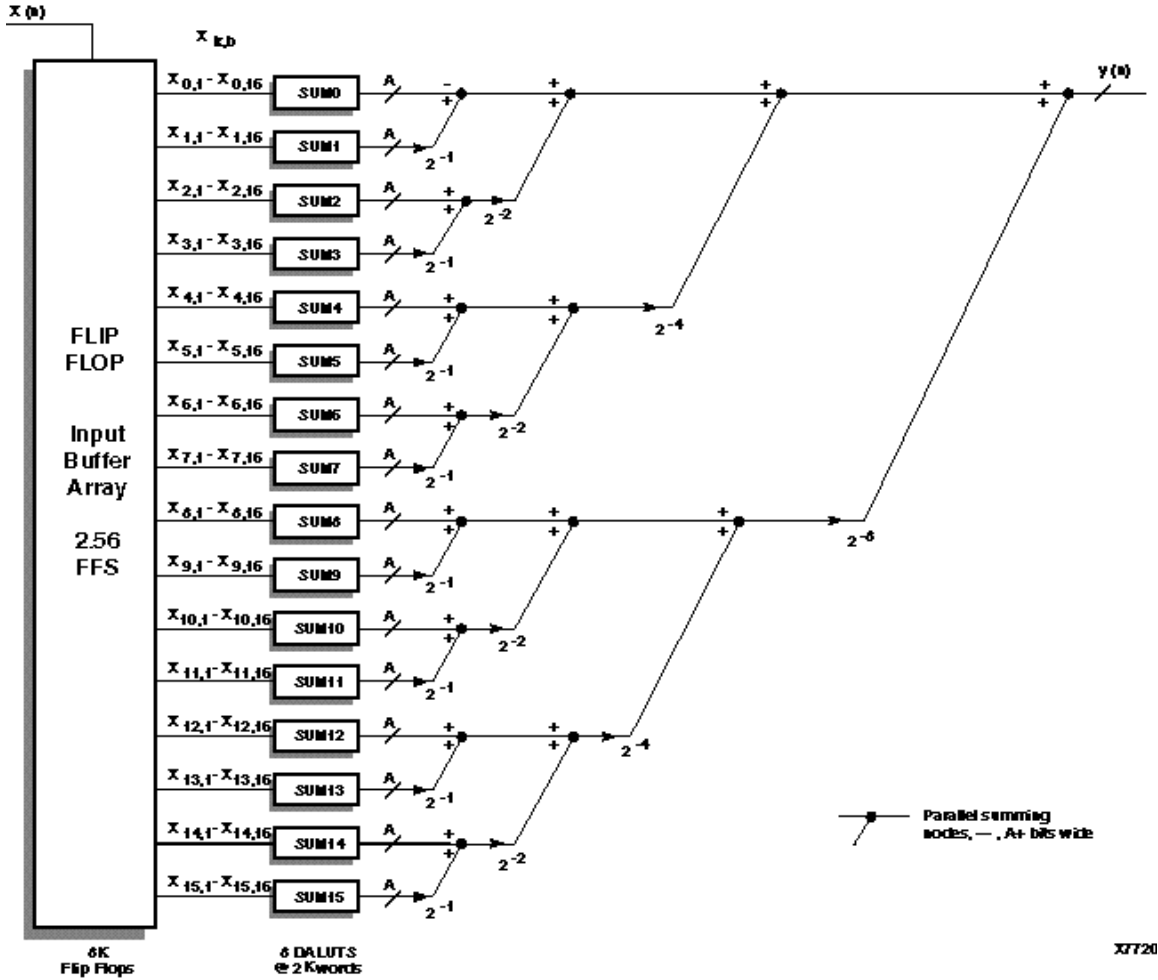


Fig. 2. Example of Fully Parallel DA Model (K=16, B=16)

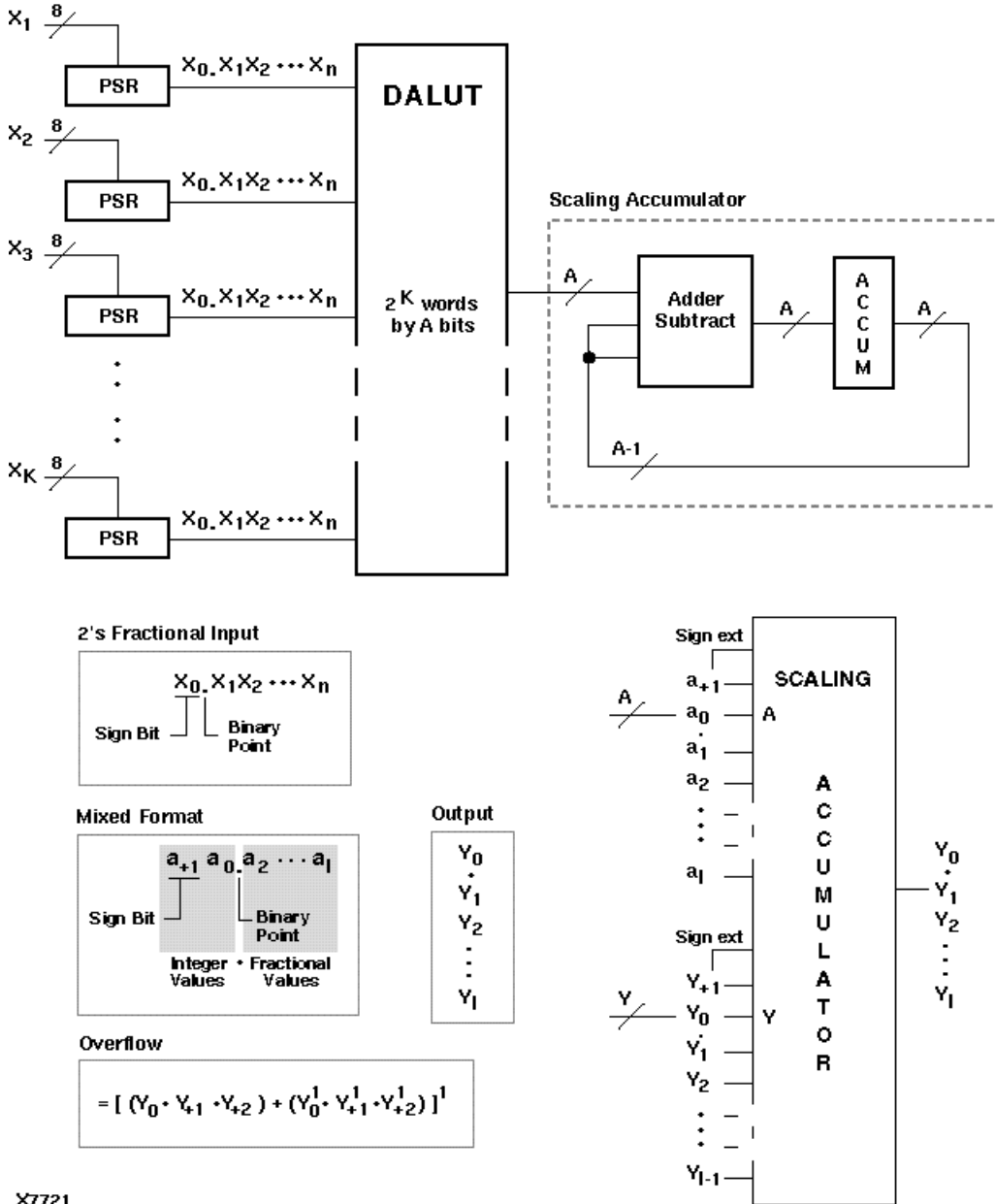
All B bits of all K data sources must be present to address the B DALUTS. A BxK array of flip-flops is required. Each of the B identical DALUTS contains 2^K words with C bits per word where C is the “cumulative” coefficient accuracy. (This, too, will be discussed later). The data flow from the flip-flop array can be all combinatorial; the critical delay path for B=16 is not inordinately long - signal routing through 5 CLB stages and a carry chain embracing 2C adder stages. A system clock in the 10 Mhz range may work. Certainly with internode pipelining a system clock of 50 Mhz appears feasible. The latency would, in many cases be acceptable; however, it would be problematic in feedback networks (e.g., IIR filters).

The Ultimate in Gate Efficiency

The ultimate in gate efficiency would be a single DALUT, a single parallel adder, and, of course, fewer flip-flops for the input data source. Again with our B=16 example, a rephrasing of equ.4 yields the desired result:

$$\text{equ. 7} \quad y = \{ \dots 14 \dots \{ [\text{sum15}]2^{-1} + [\text{sum14}]2^{-1} + [\text{sum13}]2^{-1} + [\text{sum12}]2^{-1} + [\text{sum11}]2^{-1} \\ + [\text{sum10}]2^{-1} + [\text{sum9}]2^{-1} + [\text{sum8}]2^{-1} + [\text{sum7}]2^{-1} + [\text{sum6}]2^{-1} \\ + [\text{sum5}]2^{-1} + [\text{sum4}]2^{-1} + [\text{sum3}]2^{-1} + [\text{sum2}]2^{-1} + [\text{sum1}]2^{-1} \\ - [\text{sum0}] \}$$

Starting from the least significant end, i.e. addressing the DALUT with the least significant bit of all K input variables the DALUT contents, [sum15], are stored, scaled by 2^{-1} and then added to the DALUT contents, [sum14] when the address changes to the next-to-the-least-significant bits. The process repeats until the most significant bit addresses the DALUT, [sum0]. If this is a sign bit a subtraction occurs. Now a vision of the hardware emerges. A serial shift register, B bits long, for each of the K variables addresses the DALUT least significant bit first. At each shift the output is applied to a parallel adder whose output is stored in an accumulator register. The accumulator output - scaled by 2^{-1} - is the second input to the adder. Henceforth, the adder, register and scalar shall be referred to as a scaling accumulator. The functional blocks are shown in fig. 3. All can be readily mapped into the Xilinx 4000 CLBs. There is a performance price to be paid for this gate efficiency - the computation takes at least B clocks.



X7721

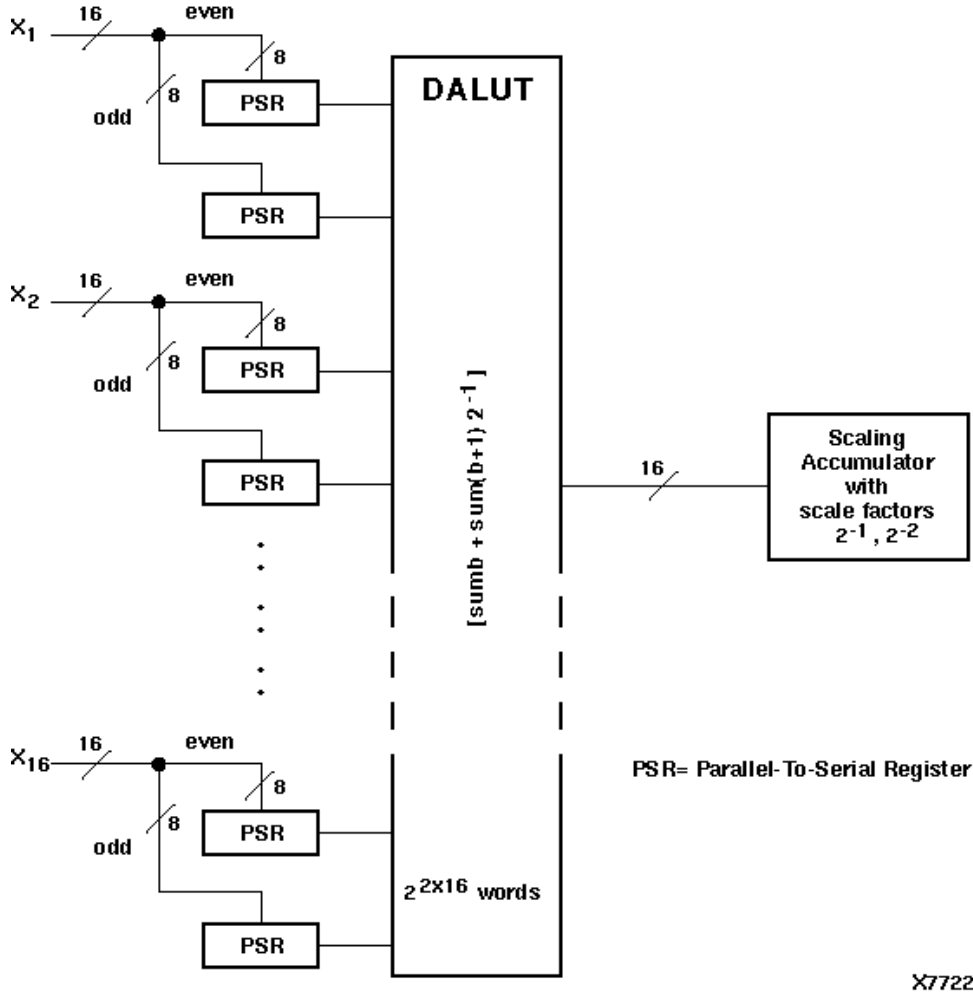
Fig. 3. Serially Organized DA processor

Between the Extremes

While there are a number of speed-gate count tradeoffs that range from one bit per clock (the ultimate in gate efficiency) to B bits per clock (the ultimate in speed) the question of their effectiveness under architectural constraints remains. We can start this study with the case of 2 bit-at-a-time processing; the computation lasts B/2 clocks and the DALUT now grows to include two contiguous bits, i.e. [sumb + {sum(b+1)}2⁻¹]. Again consider the case of B = 16 and rephrasing equ. 5:

$$\begin{aligned} y = & - [\text{sum0}] \\ & + [\text{sum1} + \{\text{sum2}\}2^{-1}]2^{-1} \\ & + [\text{sum3} + \{\text{sum4}\}2^{-1}]2^{-3} \\ & + [\text{sum5} + \{\text{sum6}\}2^{-1}]2^{-5} \\ & + [\text{sum7} + \{\text{sum8}\}2^{-1}]2^{-7} \\ & + [\text{sum9} + \{\text{sum10}\}2^{-1}]2^{-9} \\ & + [\text{sum11} + \{\text{sum12}\}2^{-1}]2^{-11} \\ & + [\text{sum13} + \{\text{sum14}\}2^{-1}]2^{-13} \\ & + [\text{sum15}]2^{-15} \end{aligned}$$

The terms within the rectangular brackets are stored in a common DALUT which can also serve [sum0] and [sum15]. Note that the computation takes B/2 + 1 or 9 clock periods. The functional blocks of the data path are shown in fig. 4a. The odd valued scale factors outside the rectangular brackets do introduce some complexity to the circuit, but it can be managed; various scaling techniques will be discussed in a later section.



X7722

Fig. 4a. Two-bit-at-a-time Distributed Arithmetic Data Path (B=16, K=16)

The scaling simplifies with magnitude-only input data. Furthermore, the two bit processing would last for 8 clock periods. Thus:

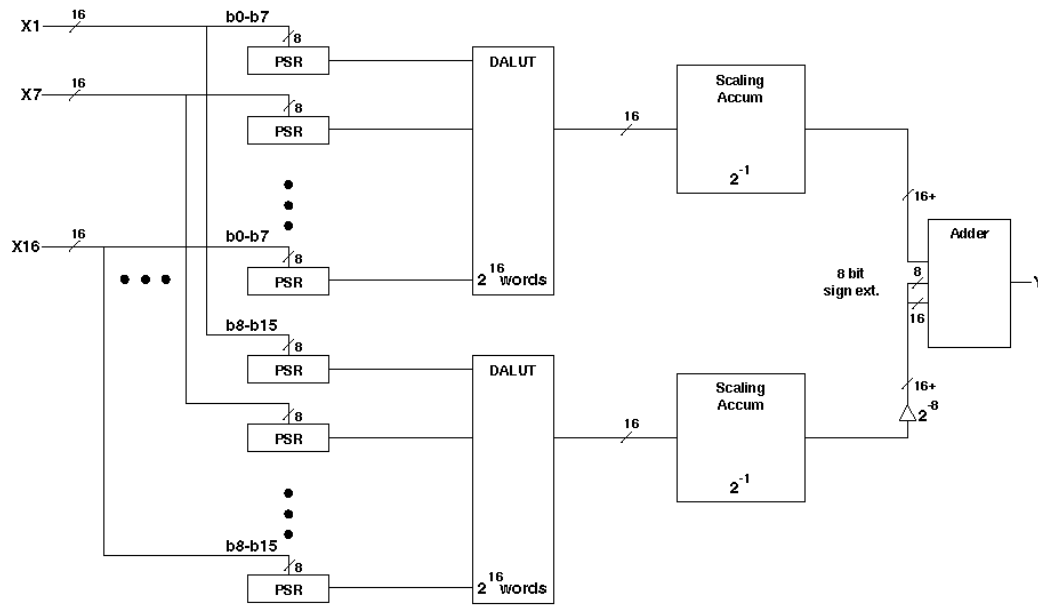
$$\begin{aligned}
 y = & [\text{sum0} + \{\text{sum1}\}2^{-1}] \\
 & + [\text{sum2} + \{\text{sum3}\}2^{-1}]2^{-2} \\
 & + [\text{sum4} + \{\text{sum5}\}2^{-1}]2^{-4} \\
 & + \text{etc.} \quad \dots\dots
 \end{aligned}$$

There is another way of rephrasing or partitioning equ. 5 that maintains the B clock computation time:

$$y = \begin{bmatrix} -[\text{sum0}] \\ +[\text{sum1}]2^{-1} \\ +[\text{sum2}]2^{-2} \\ +[\text{sum3}]2^{-3} \\ +[\text{sum4}]2^{-4} \\ +[\text{sum5}]2^{-5} \\ +[\text{sum6}]2^{-6} \\ +[\text{sum7}]2^{-7} \end{bmatrix} + \begin{bmatrix} [\text{sum8}] \\ +[\text{sum9}]2^{-1} \\ +[\text{sum10}]2^{-2} \\ +[\text{sum11}]2^{-3} \\ +[\text{sum12}]2^{-4} \\ +[\text{sum13}]2^{-5} \\ +[\text{sum14}]2^{-6} \\ +[\text{sum15}]2^{-7} \end{bmatrix} \bullet 2^{-8}$$

Here two identical DALUTs, two scaling accumulators, and a post-accumulator adder (fig.4b) are required. While the adder in the scaling accumulator may be single precision, the second adder stage may

be double precision to meet performance requirements (this point will be clarified when the detailed circuits are developed).



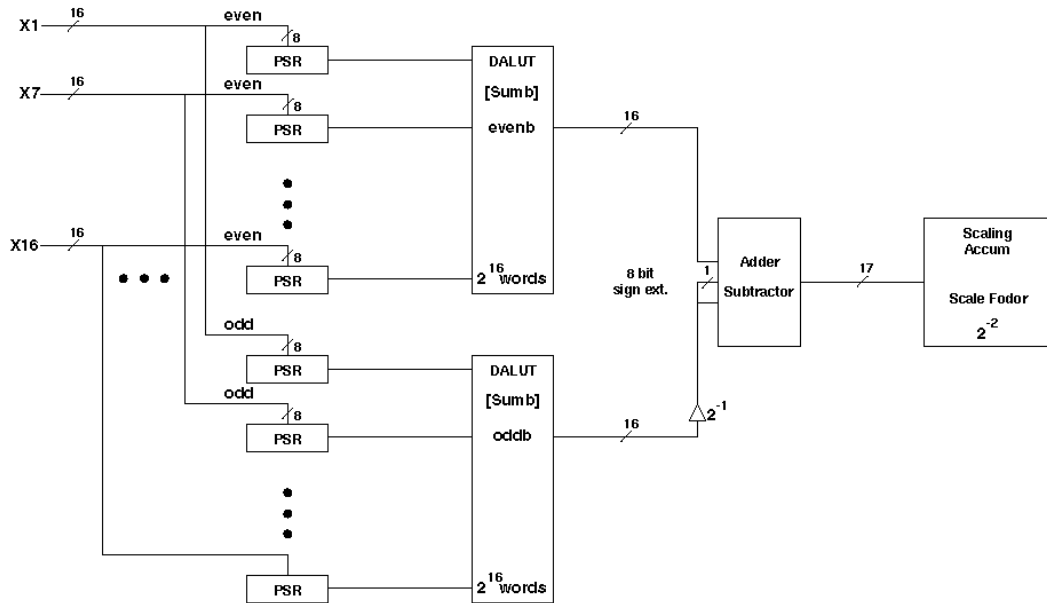
X7723

Fig. 4b. Two-bit-at-a-time Distributed Arithmetic Data Path (B=16, K=16)

Are there any other two-bit-at-a-time possibilities? The answer, not surprisingly, is yes. Each possibility implies a different circuit arrangement. Consider a third rephrasing of equ. 5.

$$\begin{aligned}
 y = & + [-\text{sum0}] + [\text{sum1}]2^{-1} \big] 2^{-0} \\
 & + [\text{sum2}] + [\text{sum3}]2^{-1} \big] 2^{-2} \\
 & + [\text{sum4}] + [\text{sum5}]2^{-1} \big] 2^{-4} \\
 & + [\text{sum6}] + [\text{sum7}]2^{-1} \big] 2^{-6} \\
 & + [\text{sum8}] + [\text{sum9}]2^{-1} \big] 2^{-8} \\
 & + [\text{sum10}] + [\text{sum11}]2^{-1} \big] 2^{-10} \\
 & + [\text{sum12}] + [\text{sum13}]2^{-1} \big] 2^{-12} \\
 & + [\text{sum14}] + [\text{sum15}]2^{-1} \big] 2^{-14}
 \end{aligned}$$

Here the inner brackets denote a DALUT output while the larger, outer brackets denote the scaling of the scaling accumulator.. Two parallel odd-even bit data paths are indicated (fig.4c) with two identical DALUTs. The DALUT addressed by the even bits has its output scaled by 2^{-1} and then is applied to the parallel adder. The adder sum is then applied to the scaling accumulator which yields the desired response, $y(n)$. Here a single precision pre-accumulator adder replaces the double precision post accumulator double precision adder.



X7724

Fig. 4c. Two-bit-at-a-time Distributed Arithmetic Data Path (B=16, K=16)

Each of these approaches implies a different gate implementation. Certainly one of the most important concerns is DALUT size which is constrained by the look-up table capacity of the CLB.

The first approach, defined by equ.5b, describes a DALUT of 2^{2K} words that feeds a single scaling accumulator, while the second, defined by equ.5c, describes 2 DALUTs -each 2^K words - that feed separate scaling accumulators. An additional parallel adder is required to sum (with the $2^{-B/2}$ scaling indicated) the two output halves. The difference in memory sizes between 2^{2K} and 2×2^K is very significant particularly when we confront reality, namely the CLB memory sizes of 32x1 or 2x(16x1) bits. Consider the cases for K = 4, 6, 8, and 10, and for A = 16. Table 1 tells the story.

Table 1. DALUT Size vs. Input Variables

K	2^K	x16b	CLBs	2^{2K}	x16b	CLBs	2×2^K	x16b	CLBs
4	16	256	8	256	4096	128	32	512	16
6	64	1024	32	4096	65,536	2048	128	2048	64
8	256	4096	128	65,536			512	8192	256
10	1024	16384	512				2048	32768	1024

The table indicates that for a 16 bit distributed arithmetic process, the single bit-at-a-time computation can accommodate 10 and, possibly, 12 input variables. For the 2^{2K} two bit-at-a-time DALUT organization fewer than 6 input variables can be handled by the XC4025 FPGA, while for the 2×2^K scheme 9 input variables may be possible. Are there any partitioning techniques that could effectively increase the number of variables on a single device? Again the answer is yes. The distributed arithmetic computation can be configured and shaped to overcome many architectural constraints. Remember the FPGA was not designed with signal processing in mind. Yet with a good understanding of the DA computation process, with an

appreciation of the constraints inherent in the CLB and the interconnect environment, and with good knowledge of the DSP algorithm of interest, an FPGA-embedded design can offer the system designer a very attractive alternative to more convention solutions.

By the Numbers

Digital Signal Processing is fundamentally the high-speed computation of the sum of arithmetic product terms as defined by equ. 1. The process is commonly referred to as “number crunching.” The designer of DA-based DSP algorithms in FPGAs is confronted with more “number” design choices than the DSP programmer faces with the standard DSP microprocessor chips. Indeed, the number formats of the DSP microprocessor are pre-ordained - either fixed-point 16 or 24 bit, or 32 bit floating point word lengths. The FPGA design choice ranges from 2 bits to as many bits as are required; data word size and coefficient (constant) word size may be independently set, and may be modified along the data path - all subject to the constraint of a fixed point number format. The choices available to the FPGA designer present both challenges and opportunities. The arithmetic processes must be well understood - pitfalls such as overflow must be avoided. Designs can be optimized in both performance and number of gate and memory resources. It is thus appropriate to review some of the basic concepts underlying number formats and arithmetic operations.

Number formats

Numbers are a systematic representation of the counting process. Large numbers are efficiently represented by a weighted number system wherein a few symbols are used repeatedly with different weights or radices. Thus, in our well-known decimal system there are 10 symbols or digits, 0 through 9, which define the value within each decimal (radix 10) position. The digits in a decimal number have position weightings of 1, 10, 100, 1000, etc. Western civilization was weaned on the radix 10. It, of course, is not the only radix; the ancient Egyptians knew the radix 2, the Babylonians the radix 60, the Mayans radices 18 and 20. The advent of digital computers has renewed our interest in radix 2 (binary) number systems.

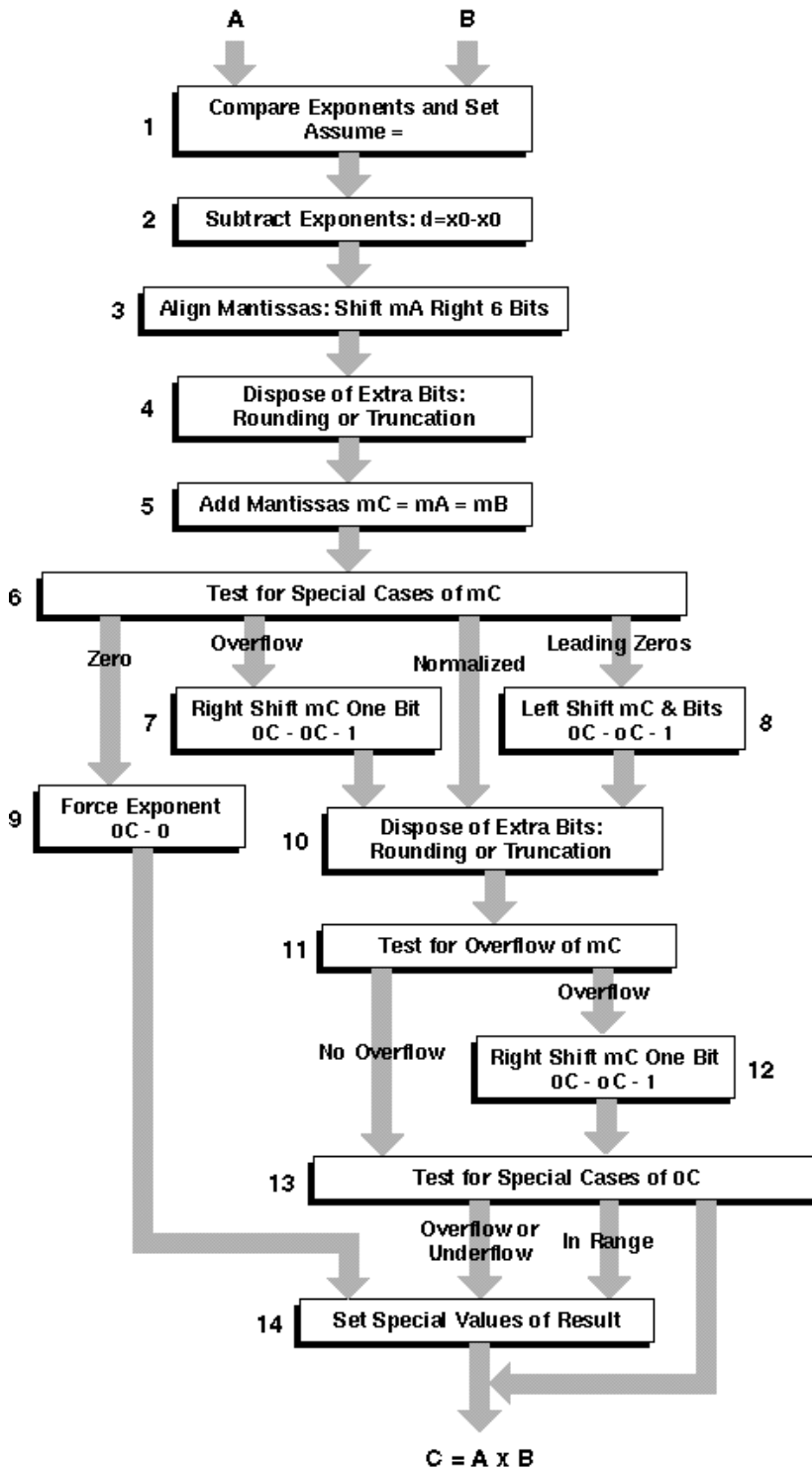
The radix-2 or binary number system fits nicely with two level (expressed by symbols 0 and 1) logic circuits. Any integer, N, can be represented in binary notation as:

equ. 8
$$N = \sum_{j=0}^M b_j 2^j$$

where the coefficients b_j denote the binary values 0,1 and the j th power of 2 denotes the weight at the j th position. The weighted binary representation serves as the basis for both fixed and floating point number formats. While our focus will be on fixed binary point formats it should be noted that floating point is used widely, and a standardized format (IEEE 754) has been established. Since many DSP microprocessors offer this format, a brief background description follows.

The floating point number format is largely inspired by the need to represent large dynamic range numbers ($10^{\pm 99}$) without resorting to extremely large word lengths. Similar to the logarithm, the floating point number is represented by both an exponent and a mantissa. The standard 32 bit format has a sign bit in the most significant bit position. The sign bit is followed by an eight bit exponent and a 23 bit mantissa. The exponent defines the powers of 2 prior to the binary point. The exponent covers a magnitude range of 2^{255} or, equivalently, a signal range of 1530 db (to be explained later) - an extremely large value since the upper end of the analog-to-digital converter range is 100 db. The binary point of the mantissa denotes the most significant “1” bit of the number. In fact, it is not represented but implied in the floating point format, with

the mantissa containing the 23 lower bits. Following an arithmetic operation, the mantissa, M , is “normalized” to the range $\frac{1}{2} \leq M < 1$. Floating point arithmetic entails complex procedures as shown in fig. 5. The additional hardware needed to execute these procedures in a manner that is transparent to the programmer adds significantly to the device cost. The fixed-point number format is a compelling choice for the FPGA.



X7725

Fig. 5a. Floating-Point Addition

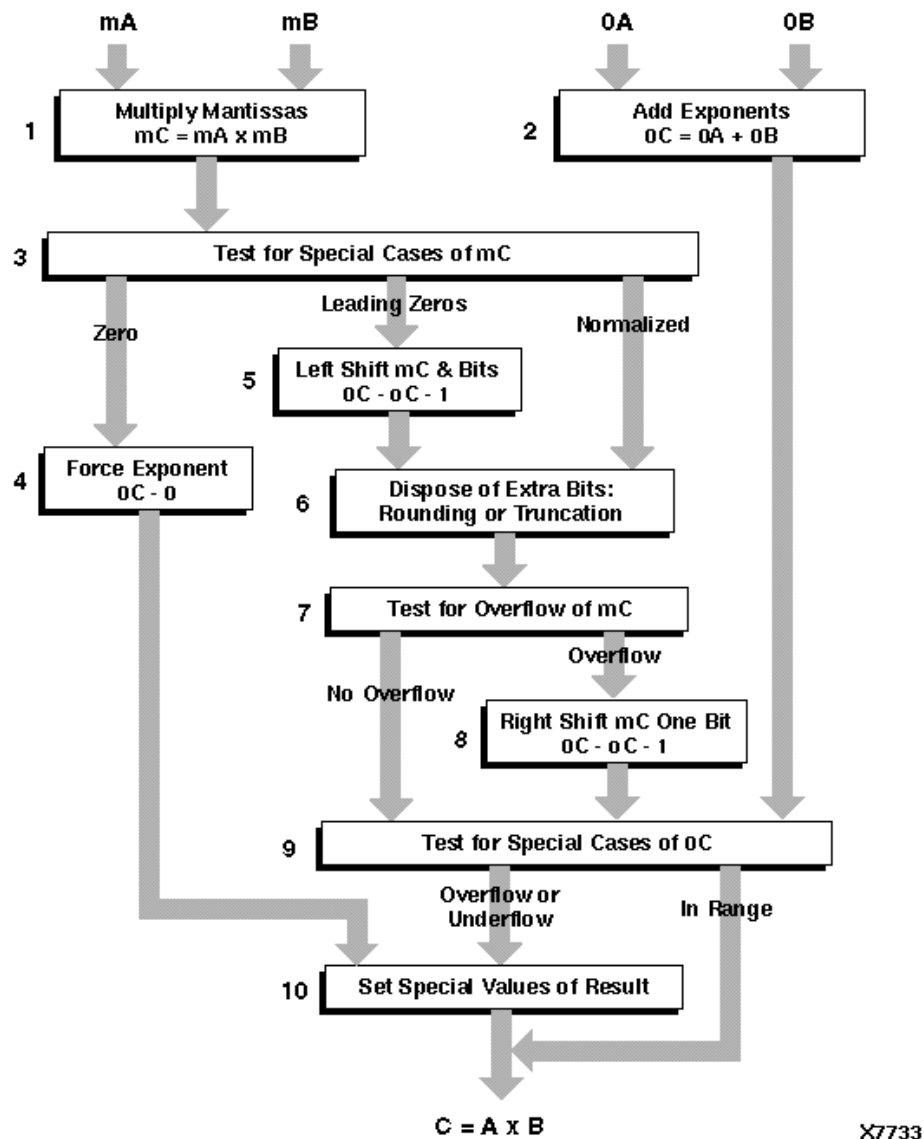


Fig. 5b. Floating-Point Multiplication

Fractional 2's Complement Representation

Number values are unbounded but computer representations are not. The basic DSP equation 1 generates word growth. The multiplication of two B bit factors produces a $2B$ bit product; addition often produces a carry bit. Repeated additions can produce multiple carry bits. Multiplication can be bounded by using the fractional 2's complement number representation. The DA development that was presented earlier is based on the use of this format; equation 2 defines the 2's complement fractional format. The sign bit, x_0 , is 1 for negative numbers. This is followed by the binary point and then a positive component made up of a decreasing order of $B-1$ binary weighted fractions. Two important features of this form are:

- The number 0 is represented unambiguously with all coefficients, x_b , equal to 0 (this is not the case for the 1's complement).

- Subtraction is easily realized by complementing the subtrahend bit by bit, doing a binary addition of the two operands, and then adding $1 \times 2^{-(B-1)}$ (i.e. 1 in the least significant bit position) to the results.

The range of values, -1 to $1-2^{-(B-1)}$, is not symmetric about the origin. This may introduce undesirable artifacts in the output signal when full-scale limiting occurs. Clamping to symmetrical end points $-1 + 2^{-(B-1)}$ and $1 - 2^{-(B-1)}$ eliminates this problem.

The multiplication of two B bit fractional numbers yields a fractional, double precision product of $2B$ bits. If the product is subjected to further multiplications, word growth would soon exceed the size of the processing resources. Reduction to single precision through rounding or truncation is a standard procedure. The carries resulting from addition cause word growth at the high end. The result is a mixed format with integer values to the left of the binary point and fractional values to the right. The largest integer bit defines the value of the sign. If the add operation yields result that exceed the capacity of the adder circuit, overflow occurs and the value of the sum “wraps around” - a transition from maximum positive to maximum negative, or vice versa. If an intermediate stage of a multi-stage filter were to overflow, a very large transient would be introduced into the signal path. Note that there is forgiveness to overflow; if partial filter response results produce a single overflow, the wrap-around can be canceled if the final result remains within bounds. One way of ensuring this is to design filters with unity gain in the passband(s). Plots of the dynamic range of fixed and floating point word sizes are offered in fig. 6. While the dynamic range for the 32 bit floating point word is off the chart, the computation accuracy is limited to the 24 bits of the mantissa.

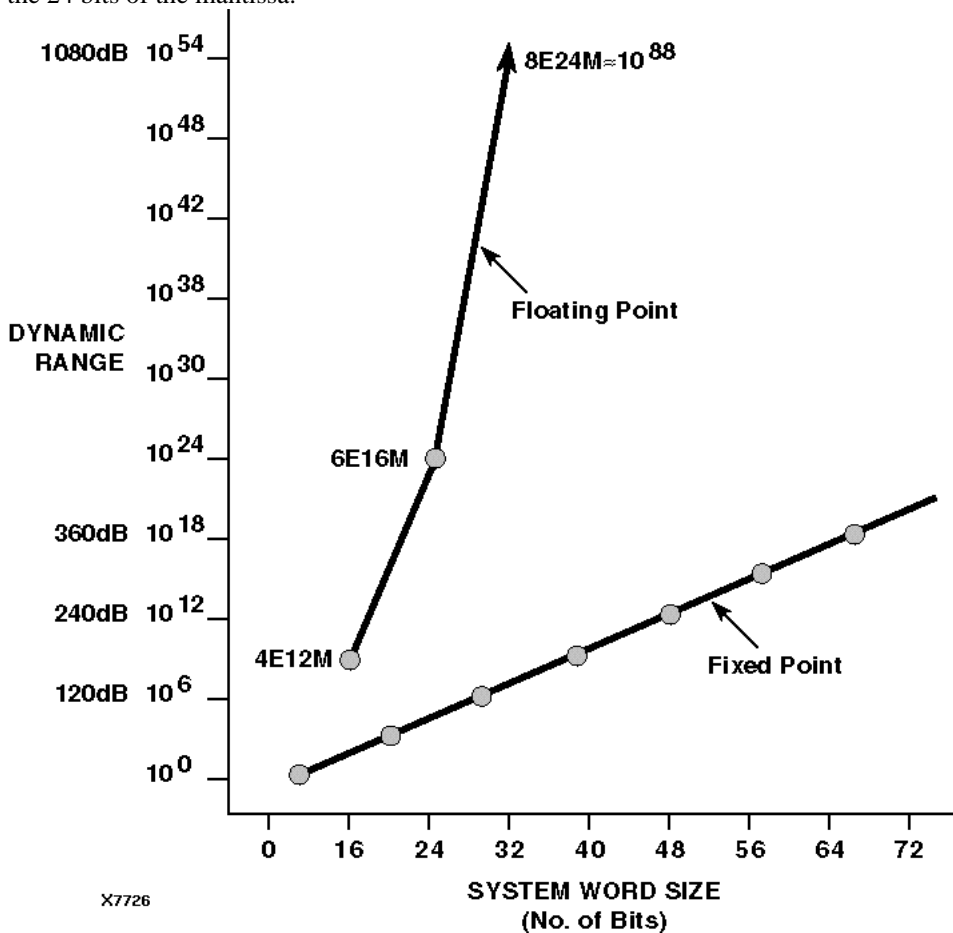


Fig. 6. Word Size vs. Dynamic Range

A Bit's Worth

Signals flowing through analog networks are subject to losses and gains. The operations affecting signal level are expressed in logarithmic form (specifically, decibels) so that cascaded gain changes need not be multiplied but, rather, can be added when expressed in decibels (dB). The signal gain (or loss) in one processing stage may be expressed as:

$$G = 20 \log (S_o/S_i) \text{ dB} \quad \text{where } S_i = \text{input signal, } S_o = \text{output signal}$$

This notation can be carried over to the digital domain where signals are expressed as binary weighted numbers. Consider a signal of M bits. In passing through a digital stage with a signal gain of 2, the output signal grows to M+1 bits. The stage gain due to the addition of a single bit is:

$$20 \log (2/1) = 20 \log 2 = 6.02 \text{ dB}$$

Thus each bit added to a binary word extends the signal representation range by approximately 6 dB. An 8-bit word has a 48 bit signal range, while a 16 bit word has a 96 dB signal range. This signal carrying capacity is also referred to as dynamic range, linear range, noise floor, and signal-to-noise ratio. These and other signal-defining terms, as well as the consequences of finite word computations, will be covered in a later section.

Tracking the Binary Point in the DA Data Path

Since gate efficiency is an important goal in FPGA-based designs, it is important that scaling discipline be maintained - that word sizes be minimized to ensure adequate network accuracy, and dynamic range while avoiding overflow. Fortunately, it is very easy to track word size changes in the distributed arithmetic data path - even with mixed word formats. We shall now trace the binary point track of the serially organized DA Processor of fig. 3.

The input data are 2's complement fractional and address the DALUT bit serial. The DALUT may contain mixed format words with several bits above the binary point. This is particularly the case for IIR filter stages. The Scaling Accumulator will also match the DALUT in the number of bits above the binary point. An additional integer bit may be added to guard against intermediate overflow conditions during the recursive shift and add operations. In scaling down the Accumulator output to the adder the least significant bit is discarded (or saved in a serial shift register if a double precision answer is desired). The most significant bit is extended up one bit position.

After the input sign bits have addressed the DALUT, and after the final subtraction, the Scaling Accumulator will produce a mixed format output with the binary point set by the DALUT. With unity gain (0dB) passband response the final output should be 2's complement fractional. Thus the sign bit reverts to the first bit to the left of the binary point. All other inter bits should match the sign bit (whether 1 or 0); any differences indicate an overflow in the final computation. During simulation or prototyping it may be wise to monitor these integer bits.

There are cases such as the symmetrical FIR filter where serial adders precede the DALUT. The ensuing carry does move the sign bit another position to the left of the binary point. This does not change the binary point track; the final subtraction just occurs one shift pulse later.