Higher Radix Floating-Point Representations for FPGA-Based Arithmetic

Bryan Catanzaro, Brent Nelson
Electrical and Computer Engineering Department
Configurable Computing Laboratory
461 CB, Provo UT 84602
bryan.catanzaro@byu.edu, nelson@ee.byu.edu

Abstract

FPGA implementations of floating-point operators have historically been designed to use binary floating-point representations. The general computing world settled on binary floating-point representations over three decades ago, and more recently, the FPGA community followed their example. Binary representations were chosen to maximize numerical accuracy per bit of data, however, the unique nature of FPGA-based computation makes numerical accuracy per unit of FPGA resources a more important measure of the usefulness of a given floating-point representation. In this paper, we show that higher radix floating-point representations are well suited to FPGA-based computations, especially high precision calculations which require the support of denormalized numbers. Higher radix representations use FPGA resources more efficiently. For example, a hexadecimal floating-point adder has a 30% smaller area-time product than its binary counterpart, while still delivering equal worst-case and better average-case numerical accuracy. Contrary to established belief, higher radix representations are useful for FPGA applications requiring IEEE 754 compliance, since they can deliver superior numerical performance while still using less FPGA resources.

1 Introduction

Recent increases in FPGA capacity and capability have led to broader use of custom floating-point datapaths. When configurable resources were scarce, floating-point arithmetic could not be practically implemented on FPGAs, due to its large area and latency cost compared to fixed point arithmetic. The steady and rapid growth of FPGA resources has increased FPGA floating-point throughput to match or beat conventional floating-point processors, and FPGA floating-point throughput is growing at a faster rate. Indeed, it has been forecasted that FPGAs will enjoy an order of magnitude higher throughput on double precision floating-point arithmetic than conventional CPUs by the year 2009 [25].

Recently, there has been much work done on floating-point for FPGAs, ranging from investigating hardware architectures [20] to implementation specific optimizations [23], [19]. Several parameterizeable floating-point libraries have been developed specifically for FPGAs [1], [5]. However, the cost of floating-point arithmetic on FPGAs is still high enough that novel alternatives such as Dual Fixed Point continue to be considered [8].

The general computing world has settled on floating-point representations which conform to IEEE standards 754 and 854 ([15], [16]). These standards play a crucial role in ensuring numerical robustness and code compatibility among machines of vastly different architectures. However, the choice of floating-point representation has such a dominant impact on FPGA implementation cost that the standards are often bent, giving the designer freedom to choose a custom floating-point representation in order to spend FPGA resources as efficiently as possible. For example, work has been done to automatically determine custom floating-point bitwidths for each node of a computation [10], and others have demonstrated the suitability of very tiny floating-point representations with much less precision and range than IEEE single precision [6].

Choosing non-standard floating-point representations by manipulating bitwidths is natural for the FPGA community, since bitwidth has such an obvious effect on circuit implementation costs. Besides the non-standard bitwidths, FPGA-based floating-point units often save hardware cost by omitting support for denormalized numbers or some of the rounding modes specified by the IEEE standard.

Although the impact of non-standard bitwidth floating-point representations on FPGA implementation is well known, the effect of non-standard radix floating-point representations has not been examined. The word "radix" in the context of computer arithmetic has acquired several meanings, which can be confusing. We use the word radix to refer to the numerical base of the floating-point representation, meaning that the mantissa is interpreted to be composed of digits of some base greater than 2. This is not to be confused with high radix Booth encoding for multiplication or high radix division algorithms, as found in references to "high-radix" floating-point operators such as [26].

In this paper, we show that higher radix floating-point representations, especially hexadecimal floating-point, are uniquely suited for FPGA-based computation, especially when denormalized numbers are supported. Choosing a higher radix floating-point representation can reduce adder area by 25% and multiplier area by 12%, while still providing equal worst-case and better average-case numerical accuracy than the standard binary representation. This paper justifies higher radix representations from a numerical perspective as well as presents implementation results (Xilinx Virtex-II) for arithmetic operators which operate on higher radix number representations.

2 Mathematical Terminology

Floating-point arithmetic approximates a real number x by choosing an element of a finite set of exactly representable real numbers S, called the significance space [21]. Elements of S_β^u have the form

$$s\beta^e\beta^{\delta-u}\sum_{i=0}^{u-1} d_i\beta^i \tag{1}$$

where $s = \pm 1$ represents the sign, β is the base, or radix, u is the number of β-ary digits in the mantissa, $d_{u-1}\cdots d_0$ are the digits of the mantissa, with d_{u-1} being the most significant digit, e is the exponent, and $\beta^{\delta-u}$ is a term that accounts for the placement of the implied radix point. With this notation, the radix point is placed δ digits into the mantissa, from the most significant side. Equivalently, we can understand the $\beta^{\delta-u}$ term as a scaling factor which leads to interpreting the mantissa to be in the range $[\beta^{\delta-1}, \beta^{\delta})$.

We consider radices of the form $\beta = 2^\nu$, which ensures that each digit d_i is efficiently representable in binary. Expanding (1) into binary, with $\beta = 2^\nu$, elements of S have the form

$$s2^{\nu e}2^{\nu\delta-t}\sum_{j=0}^{t-1} b_j 2^j \tag{2}$$

where each β-ary digit d_i from (1) is expanded into its binary form $b_{\nu(i+1)-1}\cdots b_{\nu i}$, and $t = \nu u$ is the number of bits in the binary encoding of the mantissa ($d_{u-1}\cdots d_0$). The term $2^{\nu\delta-t}$ accounts for the placement of the implied binary point. We require t to be an integer, which ensures that the mantissa is representable with an integral number of bits, but make no such restriction on u, allowing fractional digits of radix β. Similarly, we require $\nu\delta$ to be an integer, but allow fractional δ. With this representation, the radix point is placed $\nu\delta$ bits into the mantissa, which may fall in the middle of a β-ary digit. In other words, we allow the radix point to function as a binary point, regardless of radix, positioning it between any bit of the mantissa, not just at the boundaries of radix β digits.

If the leading one is found within the most significant $\nu = \log_2\beta$ bits, the number is considered normalized. Otherwise, the number is considered denormalized, which is permitted only when representing exceptionally small numbers. Normalization is essential to floating-point accuracy because it keeps the mantissa bits significant and enables easy comparison of two floating-point numbers. However, it is also expensive to implement in FPGA hardware. Higher radix representations simplify normalization: with conventional binary representations, the leading non-zero bit must be exactly located and positioned, whereas with a radix 2^ν representation, the leading non-zero bit is located and positioned less precisely - only to within ν bits. This simplification results in hardware savings.

3 Background

Before the advent of floating-point standards, various radices greater than 2 were in use. For example, the Illiac II used $\beta = 4$, the Burroughs 5500 used $\beta = 8$, and the IBM 360 used $\beta = 16$ [2]. IBM mainframes still support hexadecimal floating-point ($\beta = 16$) for compatibility reasons [11]. The designers of these systems chose higher radix representations because of area and latency savings for higher radix floating-point arithmetic units, which come primarily through reductions in the size of the shifters and leading one detection circuitry due to relaxed normalization procedures.

During the late 1960s and early 1970s, there was tension between hardware designers and numerical analysts as to the choice of radix. Hardware designers wanted to use higher radix representations to reduce the hardware cost of floating-point functional units, and numerical analysts were set on radix 2 because of its numerical advantages. The numerical analysts won the battle, because the cost of a floating-point arithmetic unit decreased so quickly that hardware penalties incurred by the use of radix 2 ceased to be a concern. IEEE standard 754 mandates the use of radix 2, and although IEEE 854 is entitled "IEEE Standard for Radix-Independent Floating-Point Arithmetic", it forbids the use of radices other than $\beta = 2$ and $\beta = 10$ [16]. Decimal representations are required for financial calculations, in order to produce exactly the same results as those done by hand [4], but their inefficient implementation causes them to be avoided whenever possible.

Despite the hardware advantages of higher radix floating-point, radix 2 has been chosen as the standard over other commensurable radices because radix 2 systems always have the best numerical accuracy when given a fixed number of bits to encode the entire floating-point number, including mantissa, exponent, and sign [3]. This comes about because there are no leading zeros in normalized radix 2 mantissas, which means that all mantissa bits are always significant. With higher radices of the form $\beta = 2^{\nu}$, up to $\nu - 1$ bits may be leading zeros. These leading zeros can be understood as exponent information which has been encoded into the mantissa, which has the effect of reducing the number of significant bits in the mantissa. Additionally, the first digit of a normalized radix 2 mantissa is always 1. Since this is known for every normalized radix 2 number, the leading digit can be implied, freeing one extra bit of precision in the actual representation. Because the first digit of a normalized floating-point number lies in the range $[1, \beta - 1]$, none of the leading bits can be implied for representations with $\beta > 2$. This fact gives radix 2 representations an extra bit of precision over other representations.

Because memory and register file oriented computing systems must represent floating-point data in a convenient, fixed number of bits, numerical accuracy per bit of representation is the dominant measure of a floating-point representation's usefulness for the general computing world. The studies which led to the choice of radix 2 as the standard were all based on this underlying premise, and so they kept the bitwidth of the floating-point word constant as they determined which radix was most advantageous (e.g., [2], [3]). To our knowledge, this fundamental assumption has not been questioned in light of the unique capabilities and limitations of FPGAs.

In the ASIC community, hexadecimal floating-point has been recently advocated for use in lightweight, low power ASIC designs [9], where the authors found that it reduced the size of the floating-point adder by 11%, but increased the size of the multiplier by 43% for very small (14-15 bit) floating-point word sizes. Our work shows a greater benefit for hexadecimal floating-point operators because we include support for denormalized numbers, we are implementing on an FPGA instead of an ASIC, and because we present results from larger floating-point formats (equivalent to IEEE single, double, and quadruple precision).

4 Higher Radix Representations for FPGAs

In contrast to conventional computing systems, custom floating-point datapaths implemented on FPGAs are not as limited by memory concerns. Data being processed on an FPGA is more likely to stay on chip until the application has finished processing it [25]. This, along with the use of distributed state in pipeline registers instead of a central register file, frees FPGA-based computation systems from rigid restrictions on floating-point word size imposed by memory interfaces. Instead, FPGA performance is constrained by circuit area, since FPGAs gain their high performance by exploiting spatial parallelism, unrolling a computation to fill the available compute fabric. Non-standard bitwidth floating-point formats are common on FPGAs because their use may enable the implementation of a particular computation or increase performance, with "acceptable" numerical accuracy.

Since FPGA performance is constrained by circuit area instead of memory interface, the fundamental assumption which led to the choice of radix 2 and exclusion of higher radix representations is not of primary importance. Instead of numerical accuracy per bit of representation, FPGA-based computing systems aim to maximize numerical accuracy and performance per unit of circuit area. From this perspective, higher radix representations are more efficient for FPGAs, even when their binary forms must be enlarged slightly in order to equalize numerical performance with their radix 2 counterparts. Also, the implied bit touted as a unique advantage of radix 2 representations is not a compelling advantage from this perspective, since it saves less than 1% of circuit area in FPGA implementations of floating-point operators.

The numerical disadvantages of higher radix representations can be resolved by adding a few bits to the mantissa, which is not practical in the general computing world because of the constraints imposed by memory interfaces. For a radix 2^{ν} representation, an additional $\nu - 1$ bits of mantissa are sufficient to equalize worst case numerical accuracy, while providing increased average accuracy [9]. Because FPGAs are architected with bit-level granularity, the penalty for a few extra mantissa bits is minimal.

Storing slightly wider intermediate results in embedded block memories on FPGAs should not pose a problem, since most FPGA block memories can be configured in multiples of 9 bits wide, and thus have a few extra bits to store data. These extra bits were originally intended to store parity information, however, they are often used to store data. For example, the internal single precision floating-point datatype used in [12] is 34 bits wide, and another single precision floating-point datatype provided commercially by Nallatech is 36 bits wide [24]. Our higher radix floating-point representations still fit conveniently in FPGA embedded memories, despite being a few bits wider than the standard datatypes.

Some people may feel that a higher radix implementation is not acceptable for FPGA designs which aim to replace an IEEE compliant CPU. Although it is true that a higher radix design will not produce bit-for-bit the same output as a standard IEEE design, the IEEE specification does not require identical output from all IEEE compliant operators. For example, the Intel x87 floating-point unit performs all calculations in an internal 80-bit double extended format, converting down to single or double precision only on command [17]. The results from an x87 FPU will thus be more accurate and therefore not identical to the results from a 64-bit double precision unit which satisfies the bare minimum of the IEEE specification. Similarly, the widespread use of fused multiply-add units, such as those on IBM and Motorola's PowerPC and Intel's Itanium processors, also results in more accurate computation than the IEEE standard requires [13]. This occurs because only one rounding operation is required in a multiply-add operation, as opposed to the two which are necessary to do a multiply and then an add, using standard operators. Systems which use a fused multiply-add unit will therefore produce slightly different, more accurate results than those which do not.

Analogously, FPGA-based systems which use IEEE formats externally and compute internally with a higher radix are acceptable for applications requiring IEEE compliance, since they have higher numerical accuracy and equal dynamic range.

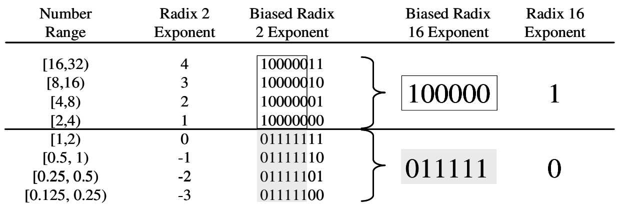
5 The Radix Point and Dynamic Range

Changing the radix of a floating-point representation affects both the mantissa and the exponent value of a floating-point number. Since the radix is exponentiated by the exponent value, higher radix representations need smaller values of exponent to represent the same number. Essentially, we divide the radix 2 exponent by ν to yield the radix 2^ν exponent. Thus, the exponent of a radix 2^ν representation can be restricted in range by a factor of ν compared to a radix 2 representation, while still keeping a dynamic range equal to that of the radix 2 representation. This allows us to represent the higher radix exponent with $\lfloor \log_2 \nu \rfloor$ fewer bits and keep roughly the same dynamic range. However, there are some subtleties that should be explained.

According to the IEEE standards, exponents are represented in biased form, where an n bit exponent has bias $BIAS = 2^{n-1} - 1$, and the actual encoded exponent value is $e + BIAS$. This particular bias allows floating-point comparison to be performed as a signed integer comparison when the floating-point number is structured as [sign, exponent, mantissa] [22], which is useful, and so we choose the standard bias for our higher radix representations.

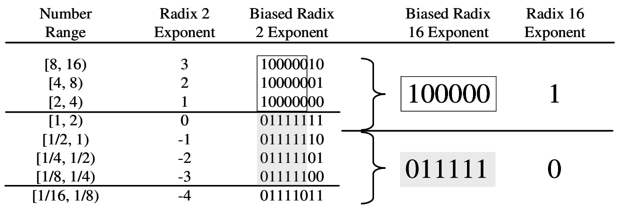
Along with the biased exponent, another feature of IEEE standard floating-point is that the mantissa is interpreted to be within the range $[1, 2)$. This means that the standard places the binary point 1 binary digit into the mantissa, or utilizing our earlier notation, defines $\delta = 1$. We envision that many applications will require input and output data in a conventional, IEEE compliant form, so we choose the parameters of our higher radix representation to keep translation hardware to a minimum. The placement of the binary point affects both dynamic range and the translation hardware necessary to map from standard representations to higher radix representations, so we need to choose it carefully.

Conversions between radix 2 and radix $\beta = 2^\nu$ will involve division and multiplication by ν, as explained earlier, so we are interested in simplifying the conversions for radices such that $\nu = 2^k$, which allows the division and multiplication to be accomplished by shifts alone.



Number Range	Radix 2 Exponent	Biased Radix 2 Exponent	Biased Radix 16 Exponent	Radix 16 Exponent
[16,32)	4	10000011	100000	1
[8,16)	3	10000010		
[4,8)	2	10000001		
[2,4)	1	10000000		
[1,2)	0	01111111	011111	0
[0.5, 1)	-1	01111110		
[0.25, 0.5)	-2	01111101		
[0.125, 0.25)	-3	01111100		

Figure 1. Exponent Mapping, Mantissa in Range $[\frac{2}{\beta}, 2)$: Binary Point Placement 1.111...



Number Range	Radix 2 Exponent	Biased Radix 2 Exponent	Biased Radix 16 Exponent	Radix 16 Exponent
[8, 16)	3	10000010	100000	1
[4, 8)	2	10000001		
[2, 4)	1	10000000		
[1, 2)	0	01111111		
[1/2, 1)	-1	01111110	011111	0
[1/4, 1/2)	-2	01111101		
[1/8, 1/4)	-3	01111100		
[1/16, 1/8)	-4	01111011		

Figure 2. Exponent Mapping, Mantissa in Range $[\frac{1}{\beta}, 1)$: Binary Point Placement .1111...

Figure 1 illustrates the simplest exponent mapping process for a conversion between a radix 2 representation with 8 bits of exponent and a radix $16 = 2^{2^2}$ representation with 6 bits of exponent: the upper 6 bits of the radix 2 exponent become the radix 16 exponent. The information from the truncated exponent bits is encoded by introducing up to 3 leading zeros into the radix 16 mantissa. In order for the exponent mapping to be accomplished by a simple truncation, figure 1 shows that the mantissa of the higher radix representation should be interpreted to be within the range $[\frac{2}{\beta}, 2)$. This requires placing the implied radix point within the first β-ary digit of the mantissa. For our earlier notation, this choice corresponds to $\delta = \frac{1}{\nu}$.

This choice of radix point placement is unorthodox: other higher-radix floating-point representations such as the hexadecimal formats used by IBM [11], or the CMU lightweight floating-point project [9], place the radix point to the left of the mantissa. The standard choice leads to a more complicated exponent mapping, as shown by figure 2.

When the implied binary point is selected as outlined, the dynamic range of the higher radix format is as close as possible to standard radix 2. For other radices of the form

Representation	Desired Value	Represented Value	Exponent	Mantissa
S_2^4, 4 bit exponent	2.0	2.0	1000	1.000
S_{16}^1, 2 bit exponent	2.0	2.0	10	0.001
S_2^4, 2 bit exponent	3.25	3.25	1000	1.101
S_{16}^1, 2 bit exponent	3.25	2.0	10	0.001
$S_{16}^{1.75}$, 2 bit exponent	3.25	3.25	10	0.001101

Table 1. Encoded Numbers in Different Representations

$\beta = 2^\nu \neq 2^{2^k}$, it is not possible to equalize the dynamic range. These representations will cover either a considerably larger or smaller range than standard binary representations.

6 Encoding

Now that we have explained how the radix point should be placed, we can illustrate how changing the radix affects bit-level encoding. The first row of table 1 shows how the number 2.0 is encoded in a radix 2 representation with 4 bits of exponent and 4 bits of mantissa, explicitly showing the leading one of the mantissa that is usually implicit. The second row shows how the same number is encoded in radix 16 with 4 bits of mantissa and 2 bits of exponent, given the binary point is placed as we described earlier. Notice that in this case, no precision is lost, and both systems are able to exactly represent the number.

The third row of the table shows how the number 3.25 is encoded in the example radix 2 representation. Row 4 shows how encoding 3.25 in the hexadecimal representation causes precision to be lost. Since 3 leading zeros were introduced, the bottom 3 significant bits of the mantissa were lost, leading to a significant representation error - instead of 3.25 as desired, we end up with 2.0! Row 5 shows how adding an additional 3 bits to the mantissa is sufficient for the hexadecimal representation to capture all the precision of its binary counterpart. Since the worst possible scenario for hexadecimal floating-point introduces 3 leading zeros, if the mantissa is extended by 3 bits, every number representable in binary floating-point is *exactly* represented in hexadecimal format.

7 Numerical Accuracy

Higher radix floating-point representations are currently unpopular because of a perceived lack of numerical accuracy. When the number of bits in the floating-point word is kept constant, high radix representations do lack accuracy, as we just illustrated, but if the mantissa is allowed to grow slightly in a higher radix representation, worst case accuracy can be equalized. A normalized radix $\beta = 2^\nu$ representation introduces up to $\nu - 1$ leading zero bits, which can be understood as an encoding of exponent information from the radix 2 representation. Thus, if the mantissa is extended by $\nu - 1$ bits, worst case accuracy will be exactly equal to that of the corresponding radix 2 representation.

Radix	Floating-Point Word Size
2	n bits
4	$n + 1$ bits
8	$n + 2$ bits
16	$n + 2$ bits
256	$n + 6$ bits
$\beta = 2^\nu$	$n + \log_2 \beta - \lfloor \log_2 \log_2 \beta \rfloor$

Table 2. Floating-point Word Size

Table 2 illustrates how the overall floating-point word size changes as a function of radix, while keeping worst case accuracy and dynamic range equal or better to radix 2, taking into account the loss of the implied leading bit, the reduction in exponent size, and the expansion of the mantissa which come with higher radix representations.

Interestingly, when worst case accuracy is equalized, the higher radix representation has better average case accuracy. To see this, we compare the significance space density of a higher radix representation with an extra $\nu - 1$ bits of mantissa to that of the corresponding radix 2 representation. Relative significance space density is the ratio of the amount of distinct numbers which can be exactly represented in 2 different significance spaces. Matula found [21] that the relative significance space density for two floating-point representations S_β^u (radix β with u β-ary digits of mantissa) and S_ϕ^r is

$$\left| \frac{S_\phi^r}{S_\beta^u} \right| = \frac{(\phi - 1)\phi^{r-1}}{(\beta - 1)\beta^{u-1}} \log_\phi \beta \tag{3}$$

Illustrating the meaning of this equation, figure 3 shows the 16 members of S_2^3 (radix 2 with 3 bits of mantissa) and the 60 members of $S_{16}^{1.5}$ (radix 16 with 6 bits of mantissa) over the interval $[2, 32)$ when the radix point has been positioned as outlined earlier. According to equation 3,

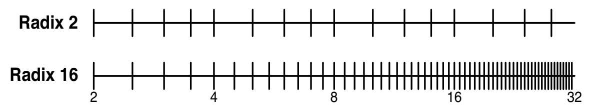


Figure 3. Density Comparison

$\left|\frac{S_2^3}{S_{16}^{1.5}}\right| = 3.75$, and indeed we see that the ratio of the number of members of those two significance spaces over this range is $\frac{60}{16} = 3.75$. Figure 3 illustrates that every number in a binary floating-point format is exactly represented in its hexadecimal counterpart, which justifies our claim that higher radix representations can provide equal worst case accuracy to standard binary representations.

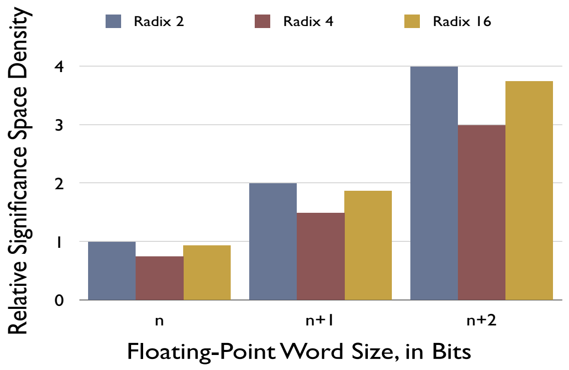


Figure 4. Significance Space Density per Bit

Figure 4 shows how significance space density changes for radices 2, 4 and 16 as a function of overall floating-point word size. At equal word size, hexadecimal representations have 94% of the density of binary representations. When the hexadecimal representation has 2 more bits, and therefore equal worst-case accuracy, it represents 3.75 times as many numbers as its binary counterpart. Since rounding ensures that the closest element of S to the exact result of the computation is selected as the output of that computation, the denser significance space of worst-case accuracy normalized higher radix representations translates into better average-case accuracy.

Indeed, the authors of [9] found that using a hexadecimal floating-point format with only 1 extra bit, instead of the 2 that are required for worst-case accuracy normalization, gave more accurate results in their calculation than the standard radix 2 format, despite the fact that their hexadecimal format had worse worst-case accuracy. Accordingly, for some applications, fully extending the mantissa to equalize worst case accuracy may not be required.

7.1 Rounding

Moving to a higher radix also affects rounding. There are several different types of rounding defined in the IEEE specification - the default and most numerically accurate is unbiased rounding to nearest even, and since it is also the most complicated rounding procedure, we will focus on how this mode must be implemented to preserve its numerical properties with higher radix representations.

The most important property of a good rounding procedure is that the rounded result of an arithmetic operator is the same result as if the operation had been accomplished with infinite precision, then rounded to the given representation [18]. We want our rounding procedure, adapted for higher radices, to preserve this property.

During a floating-point add operation, before the add occurs, the radix points of the two operands must be aligned. This is accomplished by shifting the mantissa of the smaller operand to the right as dictated by the difference in their exponents. During this shifting, significant bits may be shifted away into oblivion. Later on, during normalization, the result of the add may be shifted to the left, which ideally should reintroduce the bits which were lost at alignment. In order to do this, in radix 2 addition there are three extra bits which are added to the least significant end of the smaller addend, which are usually called the Guard, Round and Sticky bits [7].

For higher radix addition, the Guard bit must be turned into a Guard digit in order for the operation to retain all the significant bits that may be shifted out during alignment, and later shifted back in during normalization. The function of the Round and Sticky bits doesn't change, and so they remain unchanged in higher radix rounding procedures.

Thus, instead of the 3 extra round bits needed for radix 2, we now have $\nu + 2$ round bits. We have included this rounding procedure in our adder.

For multiplication, there is no need for a guard digit. Unbiased rounding requires one round bit to determine whether the result should be rounded up or down, and the sticky bit to signal whether all other bits of the result are 0. The rounding procedures remain as they are in radix 2 operations.

8 Implementation and Results

Using the parameterization capability of JHDL [14], we have implemented an adder and multiplier which are parameterizeable in both bitwidth and radix, as well as conversion circuitry between radix 2 and radix 16. The parameterized circuits are unpipelined, so we also implemented pipelined radix 2 and radix 16 single precision adders and multipliers to show that the efficiency gains seen in the unpipelined operators remain after pipelining.

All experiments were placed and routed on a Xilinx Virtex-II 6000, speed grade 6, with embedded multiplier stepping 1. No hand or relative placement was used. We present results for radix 16 and radix 4, since they are easily convertable to radix 2 and are therefore of greatest interest. All circuits implement the round to nearest even rounding procedure, as well as support for denormalized numbers. When reference is made to single precision, etc., the high radix circuits have equal worst case accuracy and equal dynamic range as their IEEE radix 2 counterparts, i.e. they use the formats described above, including the extension of the mantissa by $\nu - 1$ bits, and the contraction of the exponent by $\lfloor \log_2 \nu \rfloor$ bits. Thus, the hexadecimal representation compared against IEEE single precision has 6 bits of exponent and 27 bits of mantissa, while its radix 2 counterpart has 8 bits of exponent and 24 bits of mantissa.

8.1 Priority Encoder

The priority encoder is one of the critical circuits in the adder and multiplier. It finds the leading non-zero digit using fast carry logic. Using a higher radix significantly reduces the size and critical path of the priority encoder.

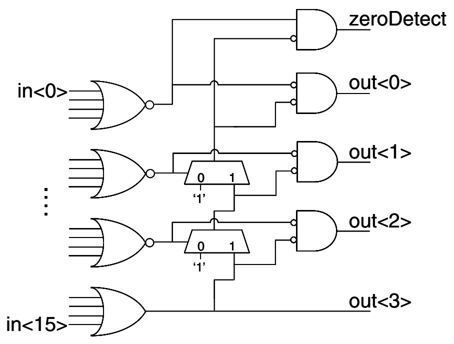


Figure 5. 16 Bit Priority Encoder for Radix 16

Figure 5 illustrates a radix 16 priority encoder. The incoming word is divided into radix-16 digits, which are then priority encoded conventionally. The key is that the number of digits is reduced by a factor of 4, significantly reducing the complexity of the operation. The priority encoder for a single precision radix 2 adder is 25 bits long: 24 bits for the mantissa, plus 1 for the guard bit where the leading one may be located. The carry chain for such an encoder is then 23 bits long, since the top and bottom bits do not require carry propagation. In contrast, the corresponding priority encoder for the single precision radix 16 adder has a 6 bit long carry chain. This arises because there are 27 bits in the mantissa and 4 guard bits, making 31 bits or 8 radix-16 digits, since the incomplete digit must be counted as a full digit. The priority encoder is a relatively small circuit, so it doesn't contribute much to the size reduction. However, it is in the critical path of the normalizing circuitry, which makes the encoder critical path length reduction more prominent.

8.2 Normalizing and Aligning Shifters

The bulk of the hardware savings comes from reducing the size of the normalizing and aligning shifters. Since a radix 2^ν shifter only has to shift to within ν bits, the amount of shifting which must be performed is reduced significantly.

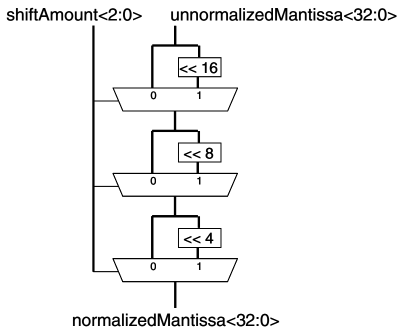


Figure 6. Radix 16 Normalizing Shifter

Figure 6 illustrates this benefit for the normalizing shifter of the single precision radix 16 adder. The shifter must shift 0 to 7 radix-16 digits, requiring 3 stages of 2 input muxes. The corresponding radix 2 shifter must shift 0 to 24 bits, requiring 5 stages of 2 input muxes[1]. Since these shifters occupy a relatively large area, reducing their cost creates most of the area benefit of high radix multipliers and adders.

8.3 Unpipelined Adder

Our adder implements the canonical single path floating-point adder architecture as outlined [22], [7].

Precision	Radix 2	Radix 4	Radix 16
Single	521 LUTs	465 LUTs	416 LUTs
Double	1176 LUTs	989 LUTs	903 LUTs
Quadruple	2581 LUTs	2251 LUTs	1945 LUTs

Table 3. Adder Area Comparison

Table 3 shows radix 4 gaining from 11% at single precision to 13% at quadruple precision over the standard binary

[1]Some have reported that using 4 input muxes reduces latency in normalizing shifters, although the area remains the same [19]. However, we found that the increased routing complexity which results from using 4 input muxes creates a substantial latency penalty.

adder, while radix 16 benefits from 20% at single precision to 25% at quadruple precision.

Precision	Radix 2	Radix 4	Radix 16
Single	51.5 ns	46.6 ns	48.1 ns
Double	66.9 ns	62.7 ns	61.4 ns
Quadruple	89.4 ns	88.2 ns	83.0 ns

Table 4. Adder Timing Comparison

Table 4 illustrates that the combinatorial critical path through high-radix adders is reduced slightly, around 5% for radix 4 and 7% for radix 16.

The benefits we have seen using radix 16 are greater than those observed in [9] for several reasons. Firstly, shifters are relatively cheaper in VLSI technology than in FPGA fabric, since they can use more efficient transistor level structures specifically designed for shifting. This reduces the impact of minimizing the shifters, in contrast to FPGAs, on which shifters are expensive. Secondly, [9] examines the benefit of hexadecimal floating-point representations at very small word sizes. As can be seen in table 3, the benefit from higher radix representations increases with word size.

8.4 Unpipelined Multiplier

Our multiplier uses the single-path architecture outlined in [25], and supports denormalized numbers. The multiplier makes use of embedded block multipliers for the mantissa multiplication.

Precision	Radix 2	Radix 4	Radix 16
Single	452 LUTs	445 LUTs	392 LUTs
Double	1312 LUTs	1245 LUTs	1139 LUTs
Quadruple	3559 LUTs	3431 LUTs	3130 LUTs

Table 5. Multiplier Area Comparison

Table 5 shows that radix 4 multipliers are slightly smaller than their radix 2 counterparts, while radix 16 multipliers are around 12% smaller. Higher radix operators used exactly the same number of block multipliers as the binary multiplier.

Interval arithmetic reminds us that the result of a normalized radix 2 multiplication with both mantissas in the range $[1, 2)$ will have a mantissa in the range $[1, 4)$, while the result of a normalized higher radix multiplication with both mantissas in the range $[\frac{2}{\beta}, 2)$ will have a mantissa in the range $[\frac{4}{\beta^2}, 4)$. Thus, the radix 2 multiplier has 2 ranges to select between to produce a normalized result: $[1, 2)$ and $[2, 4)$, while the higher radix multiplier has 3 ranges to choose from: $[\frac{4}{\beta^2}, \frac{2}{\beta})$, $[\frac{2}{\beta}, 2)$, and $[2, 4)$. This results in an extra layer of muxing, which along with the increased adder tree necessary to form the mantissa product reduces the benefit of higher radix representations for multipliers.

Multipliers which support denormalized numbers must have both a normalizing and a denormalizing shifter, the size of which are reduced by high-radix representations. This results in the area benefit we have observed - if our multiplier did not support denormalized numbers, we would see a small area penalty rather than a savings, due to the added mux and slightly enlarged mantissa multiplier. However, FPGAs see a smaller penalty from the mantissa extension than ASIC implementations because of the discrete area scaling behavior of multipliers constructed from smaller block multipliers. Thus, block multipliers and support for denormalized numbers explain why we observe an area benefit, as opposed to the area penalty seen by [9].

Precision	Radix 2	Radix 4	Radix 16
Single	49.0 ns	56.3 ns	52.6 ns
Double	73.5 ns	86.9 ns	74.7 ns
Quadruple	108.0 ns	122.2 ns	116.5 ns

Table 6. Multiplier Timing Comparison

The combinatorial critical path through our high radix multipliers was increased from 2-8% for the hexadecimal multiplier, and somewhat more for the radix 4 multiplier. This is primarily due to the enlarged mantissa multiplier.

8.5 Pipelined Operators

Operator	Radix 2	Radix 16
SP Adder	350 Slices	281 Slices
SP Multiplier	435 Slices	393 Slices

Table 7. Pipelined Area Comparison

Table 7 shows that the area benefits observed earlier are not changed significantly by pipelining. The radix 16 single precision adder is 20% smaller than its radix 2 counterpart, while the radix 16 multiplier is 10% smaller than the radix 2 multiplier. This is an expected result, since the topology and architecture of the operators does not change with radix, and therefore, the costs of pipelining should not be significantly different for higher radix operators.

Table 8 shows the clock periods of the pipelined operators. The radix 16 adder sees a 13% smaller clock period at the same pipeline depth. This is due to the priority encoder stage, which is significantly less complicated in the radix 16 adder. The multiplier sees an insignificantly reduced clock period.

Operator	Radix 2	Radix 16
SP Adder	6.2 ns	5.4 ns
SP Multiplier	6.4 ns	6.1 ns

Table 8. Pipelined Timing Comparison

Combining the area and time savings, the radix 16 adder has a 30% smaller area-time product, while the radix 16 multipler has a 14% smaller area-time product.

8.6 Converter Hardware

As explained earlier, the hardware necessary to convert a radix 2 representation to a radix β representation is simplified when $\beta = 2^{2^k}$. Of radices that satisfy this condition, radix 16 seems to be optimal, since it yields more hardware savings than radix 4, yet doesn't require the floating-point word size to be lengthened excessively to compensate for reduced accuracy, as do large radices such as 256.

Since a hexadecimal floating-point representation is 2 bits longer than its corresponding binary counterpart, some applications will require keeping the datapath externally radix 2 but internally radix 16, stationing converters at the gateways to the circuit. Although converter circuitry may be necessitated by higher radix representations, it is worth noting that FPGA-based floating-point datapaths gain performance by keeping data on chip as much as possible, especially since FPGAs are very pin-limited compared with the parallelism that can be accomodated internally. These two facts combined support the assertion that relatively few of these converters should be needed, and the overall system cost should be reduced by using a higher radix representation.

We chose the implied binary point placement to simplify conversion between standard radix 2 and radix 16. Because of this choice, conversion from radix 2 to radix 16 requires only a shifter which shifts the mantissa 0-3 places to the right, as determined by the bottom 2 bits of the exponent, which are then discarded to form the radix 16 exponent. A small bit of logic is required to handle exponent corner cases. No rounding is necessary, since no significant bits are lost in the conversion.

The conversion from radix 16 back to radix 2 requires a shifter to shift the mantissa 0-3 places to the left, eliminating the leading zeros. Since the radix 16 format can represent more numbers than the radix 2 format, a round operation is required to choose the closest representable radix 2 number, and some logic must be included for exponent corner cases. In order to avoid instantiating a rounder in this converter, we integrate the converter into the normalization and rounding steps of the arithmetic operators, making hybrid radix operators which accept hexadecimal numbers and output binary, IEEE results.

Precision	Radix 2 $\rightarrow$ Radix 16	Radix 16 $\rightarrow$ Radix 2
Single	50 LUTs	104 LUTs
Double	108 LUTs	229 LUTs
Quadruple	229 LUTs	484 LUTs

Table 9. Converter Circuitry Area

The cost of these converters is reasonable: in the worst case scenario with a datapath comprised of a radix 2 $\rightarrow$ radix 16 converter, a single radix 16 adder, and a radix 16 $\rightarrow$ radix 2 converter, the aggregate cost is between from 2-9% more than the cost of a single radix 2 adder. Since FPGAs gain their performance by performing multiple calculations and limiting I/O, few of these converters should be needed compared to the number of arithmetic operators in the datapath. Thus, using hexadecimal floating-point internally and binary floating-point externally should reduce overall system cost, despite the use of converter circuitry.

8.7 Future Work

We have not examined the impact of higher radix representations on divider or square-root circuitry.

We expect high radix representations to reduce power consumption similar to or slightly better than they reduce area, although this is as of yet unproven. Choosing a higher radix representation may thus be another chance to lower power consumption. Future work will explore these questions on pipelined versions of our higher radix operators.

9 Conclusion

The choice of floating-point representation has a major impact on FPGA based floating-point datapaths. Choosing a higher radix representation can yield implementations with better numerical accuracy, while still reducing area cost. Radix 16 is a particularly good choice, since it provides good area savings, and converters to and from radix 2 are simplified. Designs that are heavily constrained by memory interfaces can either sacrifice some accuracy to fit the representation within a convenient number of bits, or they can use converters at the gateways to the floating-point datapath.

High radix approaches may not be optimal for designs with much I/O and little computation, for designs using very small, non-standard representations, or for designs with many multipliers and no support for denormalized numbers. For such applications, radix 2 may be the best choice. However, for many designs, higher radix representations can be used to maximize efficiency for floating-point datapaths implemented on FPGAs. Some designers are beginning to push for greater precision than afforded by IEEE double precision, and need support for denormalized numbers [25]. The area savings afforded by higher radix representations, especially when support for denormalized numbers is required, may enable the implementation of such extremely high precision calculations on an FPGA. Since processors with hardware quadruple precision units are rare and expensive at present, such calculations must be run in software, making them an even bigger target for FPGA implementation. Calculations requiring less precision can also benefit from higher radix representations, especially if there are proportionally many add operations in the datapath.

Due to the established consensus that binary floating-point is optimal, the choice of floating-point radix has been neglected. The unique traits of FPGAs, such as the high ratio of calculation to I/O, high shifter cost, and embedded block multipliers make higher radix floating-point representations, especially hexadecimal floating-point, particularly attractive. Designers of FPGA-based custom floating-point datapaths should consider whether a high radix representation would be better suited to their needs.

References

[1] P. Belanovic and M. Leeser. A Library of Parameterized Floating Point Modules and Their Use. *Proceedings of the 12th International Workshop on Field Programmable Logic and Applications (FPL'02)*, 2002.

[2] R. P. Brent. On the Precision Attainable with Various Floating-Point Number Systems. *IEEE Transactions on Computers*, C-22:601–607, June 1973.

[3] W. S. Brown and P. L. Richman. The Choice of Base. *Communications of the ACM*, 12(10):560–561, October 1969.

[4] M. F. Cowlishaw. Decimal Floating-Point: Algorism for Computers. *Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH'03)*, 2003.

[5] J. Detrey and F. de Dinechin. FPLibrary, a VHDL Library of Parameterisable Floating-Point and LNS Operators for FPGA. *http://perso.ens-lyon.fr/jeremie.detrey/FPLibrary/*, 2004.

[6] J. Dido et al. A Flexible Floating-Point Format for Optimizing Data-Paths and Operators in FPGA Based DSPs. *ACM/SIGDA Tenth ACM International Symposium on Field-Programmable Gate Arrays (FPGA'02)*, 2002.

[7] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, San Francisco, 2004.

[8] C. T. Ewe, P. Y. K. Cheung, and G. A. Constantinides. Dual Fixed-Point: An Efficient Alternative to Floating-Point Computation. *Proceedings of the 14th International Workshop on Field Programmable Logic and Applications (FPL'04)*, 2004.

[9] F. Fang, T. Chen, and R. A. Rutenbar. Lightweight Floating-Point Arithmetic: Case Study of Inverse Discrete Cosine Transform. *EURASIP Journal of Applied Signal Processing*, pages 879–892, September 2002.

[10] A. A. Gaffar, O. Mencer, W. Luk, P. Y. Cheung, and N. Shirazi. Floating Point Bitwidth Analysis via Automatic Differentiation. *Proceedings of the International Conference on Field Programmable Technology*, 2002.

[11] G. Gerwig et al. The IBM eServer z990 Floating-Point Unit. *IBM Journal of Research and Development*, 48(3/4), 2004.

[12] M. Gokhale et al. Monte Carlo Radiative Heat Transfer Simulation on a Reconfiguable Computer. *Proceedings of the 14th International Workshop on Field Programmable Logic and Applications (FPL'04)*, 2004.

[13] E. Hokenek, R. K. Montoye, and P. W. Cook. Second-Generation RISC Floating Point with Multiply-Add Fused. *IEEE Journal of Solid-State Circuits*, 25(5):1207–1213, 1990.

[14] B. Hutchings et al. A CAD Suite for High-Performance FPGA Design. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'99)*, pages 12–24, 1999.

[15] IEEE Standards Board. *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985 edition, 1985.

[16] IEEE Standards Board. *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, ANSI/IEEE Std 854-1987 edition, 1987.

[17] Intel Corporation, Santa Clara, California. *iAPX 86, 88, 186 and 188 User's Manual: Programmer's Reference*, 1985.

[18] D. E. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, Reading, Massachussets, third edition, 1998.

[19] B. Lee and N. Burgess. Parameterisable Floating-point Operations on FPGA. *Conference Record of the Thirty-Sixth Asilomar Conference on Signals, Systems, and Computers*, 2002.

[20] J. Liang, R.Tessier, and O. Mencer. Floating Point Unit Generation and Evalution for FPGAs. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'03)*, pages 185–194, 2003.

[21] D. W. Matula. Base Conversion Mappings. *Proceedings of the American Federation of Information Processing Societies*, 30:311–318, 1967.

[22] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, New York, 2000.

[23] E. Roesler and B. Nelson. Novel Optimizations for Hardware Floating-Point Units in a Modern FPGA Architecture. *Proceedings of the 12th International Workshop on Field Programmable Logic and Applications (FPL'02)*, pages 637–646, 2002.

[24] W. D. Smith and A. R. Schnore. Towards an RCC-based Accelerator for Computational Fluid Dynamics Applications. *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'03)*, pages 222–231, 2003.

[25] K. Underwood. FPGAs vs. CPUs: Trends in Peak Floating-Point Performance. *ACM/SIGDA Twelfth ACM International Symposium on Field-Programmable Gate Arrays (FPGA'04)*, 2004.

[26] R. K. Yu and G. B. Zyner. 167 MHz Radix-4 Floating Point Multiplier. *Proceedings of the IEEE Symposium on Computer Arithmetic*, 1995.

IEEE COMPUTER SOCIETY