A Monte-Carlo Floating-Point Unit for Self-Validating Arithmetic

Jackson H. C. Yeung
Department of Computer Science and Engineering
The Chinese University of Hong Kong
hcyeung@cse.cuhk.edu.hk

Evangeline F. Y. Young
Department of Computer Science and Engineering
The Chinese University of Hong Kong
fyyoung@cse.cuhk.edu.hk

Philip H. W. Leong
School of Electrical and Information Engineering
The University of Sydney
philip.leong@sydney.edu.au

ABSTRACT

Monte-Carlo arithmetic is a form of self-validating arithmetic that accounts for the effect of rounding errors. We have implemented a floating point unit that can perform either IEEE 754 or Monte-Carlo floating point computation, allowing hardware accelerated validation of results during execution. Experiments show that our approach has a modest hardware overhead and allows the propagation of rounding error to be accurately estimated.

Categories and Subject Descriptors

B.2.0 [**Arithmetic and Logic Structures**]: General

General Terms

Design

Keywords

FPGA, Floating-Point, FPU, Monte Carlo Arithmetic

1. INTRODUCTION

Rounding error is inevitable for all finite precision computations. The most common solution is to perform each arithmetic operation at sufficiently high precision so the accumulation of error in the result is within an acceptable limit. Static analysis using affine arithmetic can also be used to estimate the propagation of rounding error in fixed point computations [3, 9] and this technique is widely used in bit-width optimization of digital circuits.

Unfortunately, analysis of rounding error propagation in floating point computations is not as straightforward. Unlike fixed point operations, the error is not bounded in a fixed interval and depends on the magnitude of the operands. A method for static analysis of floating point error based on affine arithmetic is proposed in references [4] and [5]. These methods depend on input range information and can produce overly pessimistic error bounds unless the range is bound to a small interval. In a general computing problem, the range of possible input values can either be large, or simply unavailable before runtime. Moreover, associativity of mathematical operations does not hold in floating point computations and the analysis of rounding error is highly dependent on the sequence of operations. In practice, to produce correct results, numerical computing applications implemented in software often rely on the inherent stability of the algorithm rather than careful error analysis. When such an algorithm is implemented in reconfigurable hardware, any change to the floating point implementation and sequence of operations can compromise its stability.

A different, complementary approach for dealing with rounding errors is to track their propagation at runtime. Such self-validating numerical methods can produce not only the required result, but also an error bound. A traditional approach is interval arithmetic [8] which produces strict upper and lower bounds by operating on an interval instead of a point. Unfortunately, the bound is overly pessimistic in most cases, a major reason being that this technique does not take correlations into account. Affine arithmetic [14] overcomes this problem, however, the length of the error term grows as the computation proceeds, and estimating the propagation of rounding errors at runtime is very expensive. Another approach suitable for runtime implementation is the CESTAC method [16] in which the computation is repeated using three different rounding modes and the part of the result that is the same for all rounding modes are assumed to be the significant digits.

Monte Carlo Arithmetic (MCA) [11] can track rounding errors at runtime by applying randomization to make rounding errors behave like random variables. Over a number of trials, a normal computation is turned into a Monte Carlo simulation and hence statistics on the effect of rounding errors can be obtained. Apart from floating point, MCA has also been applied to logarithmic number systems [17]. In this paper we focus on applying MCA to detect catastrophic cancellation which is the major cause of loss of significant digits in a computation.

While self-validating numerical methods produce valuable information on how rounding error affects the accuracy of the result, their implementation to date has mainly been in software which suffers from poor performance compared with hardware. This is especially problematic since applications that require high accuracy often require high performance.

FPGA'11, February 27–March 1, 2011, Monterey, California, USA.

In a field-programmable computing device, the flexibility to use custom floating point units to improve performance is present. A limited number of hardware implementations of interval arithmetic can be found in the literature [1, 12, 13]. A hardware implementation of the CESTAC method has also been published [2]. We are not aware of any hardware implementations of affine or Monte Carlo floating point arithmetic.

In this paper we describe a novel self-validating floating point unit (FPU) which uses MCA to track rounding error propagation. We also show that the area and performance overheads are modest compared to a standard FPU. We believe that self-validating numerical methods are important in reconfigurable computing for the follow reasons:

- The ability to produce an error estimate at runtime allows more aggressive optimizations to be used.
- For applications where the accuracy of the result is critical, a hardware generated error bound on the result is very useful.
- The ability to analyze rounding error at runtime enables the construction of computer systems that can dynamically tune themselves according to the input data. Such an ability would fully capitalize on the strength of reconfigurable hardware.

The remainder of the paper is organised as follows. In Section 2, we provide background on floating point numbers and MCA. A modified hardware algorithm for MCA addition and multiplication is described in Section 3. The implementation of the MCA arithmetic unit is described in Section 4. Results are shown in Section 5 and conclusions drawn in Section 6.

2. BACKGROUND

2.1 Floating Point Numbers

A binary floating point number x_{fp} can be represented as a 3-tuple $< n, f, e >$, where $n \in \{0, 1\}$ is the sign, f is an unsigned fraction referred to as the significand, and e is the integer exponent. Such a floating point number represents the real value:

$$x = -1^n \cdot f \cdot 2^e.$$

The significand of a normalized floating point number has a range of $1 \leq f < 2$. It has an implicit most significant bit of 1, called the hidden bit and so the actual value stored in the binary representation of the significand is $f - 1$. e is represented as a signed binary integer in excess format.

We define machine precision, p, to be the number of bits in the significand, excluding the hidden bit. For an IEEE 754 single precision floating point number, $p = 23$.

The IEEE-754 floating point standard also supports denormalized floating point numbers which represent those below the range representable by normalized numbers. For these numbers, the exponent is set to its smallest value and no hidden bit is assumed. In this case, assuming single precision, $< n, f, e >$ represents the real value:

$$x = -1^n \cdot (f - 1) \cdot 2^{-126}.$$

Our implementation is fully IEEE 754 compliant and hence supports denormalized numbers. Without loss of generality, normalized floating point numbers are assumed in the rest of this paper unless otherwise specified.

2.2 MCA

MCA, proposed by Parker [10], is a way to detect catastrophic cancellation and overcome several arithmetic anomalies in floating point calculations. Parker employs a high precision floating point unit to perform low precision Monte Carlo floating point computation. In his experiments, a double precision floating point unit is used to perform single precision Monte Carlo floating point computations.

Exact values are real numbers that can be represented exactly within the floating point precision whereas *inexact values* are rounded due to finite precision or real values that are not completely known. In MCA, an inexact value x is modeled with a random variable that agrees with x to s digits:

$$\tilde{x} = inexact(x, s, \xi) = x + 2^{e-s+1}\xi$$

where e is the base 2 exponent of x, s is a positive integer, and ξ is a random variable in the interval $(-\frac{1}{2}, \frac{1}{2})$, representing the uncertainty. Exact values are represented as their floating point value and are not random variables.

A full MCA floating point operation is computed as:

$$op(x, y) = round(inexact(op(\\ inexact(x, t, \xi), inexact(y, t, \xi))), t, \xi)$$

where t is the virtual precision, emulating a precision less than the actual machine precision ($t \leq p$), and ξ is a random variable uniformly distributed in the interval $(-\frac{1}{2}, \frac{1}{2})$. The function $round()$ is any floating point rounding function. In this paper, we assume round to the nearest.

In a full MCA floating point operation, the function $inexact()$ is applied: (1) once to each of the operands, and (2) also to the result of the floating point operation before rounding. The former is called *precision bounding* and can be used for detection of catastrophic cancellation. The latter is *random rounding*, which improves the statistical properties of floating point rounding and can be used to address anomalies in floating point arithmetic such as non-associativity and bias of round-off errors. This is because, if random rounding is applied, the expected value of the result converges to the correct value.

Precision bounding and random rounding can be used independently. In this work, we only consider the precision bounding operation since our objective is to estimate the propagation of rounding error. The techniques described in this paper could also be applied to random rounding,

In MCA, a computation is performed n times with the same input, forming a Monte Carlo simulation. The output is an n-tuple $X =< x_1, x_2, \ldots, x_n >$. The arithmetic mean of X is used as the result, while the distribution of X can be used to estimate the rounding error. Instability in rounding is reflected by an X with large variance.

3. MCA ARITHMETIC UNIT

In Parker's work, double precision floating point arithmetic was used to perform single precision MCA so that finite precision effects were negligible. This approach is obviously very inefficient for hardware implementations. In this section, we propose a modified algorithm for MCA floating point that uses lower precision arithmetic.

The key idea is to use a random perturbation of operands and operator result to model rounding error. For a round to nearest scheme, the rounding error is at most $\frac{1}{2} \cdot 2^{e-p}$, where

e is the exponent of the number. This is approximately half the difference of the two adjacent floating point numbers. Therefore, the rounding error can be modeled as a random variable:

$$round(x_{real}) = x_{real} + \epsilon$$

where ϵ is a random number distributed in $[-2^{e-p-1}, 2^{e-p-1}]$. If ϵ is assumed to be uniformly distributed, the forward error, δ, for a floating point operation can be computed as:

$$fop(x, y) + \delta = fop(x + \xi_x, y + \xi_y)$$

where ξ_x and ξ_y are random numbers uniformly distributed in the interval $[-2^{e_x-p-1}, 2^{e_x-p-1}]$ and $[-2^{e_y-p-1}, 2^{e_y-p-1}]$. Since ξ_x and ξ_y are less than $\frac{1}{2}$ a unit in the last place (ulp), computing $fop(x, y) + \delta$ requires extending the precision of the arithmetic unit. Our algorithm avoids this by computing a result that matches the statistical distribution of $fop(x, y) + \delta$ without directly evaluating the expression. Since our objective is estimating the propagation of rounding error in IEEE floating point computation, we do not apply random rounding.

3.1 MCA Addition

We define a function *unround* as the hardware equivalent of the precision bounding operation:

$$unround(< n, f, e >) =< n, f + 2^{-p} \cdot \Xi, e >$$

where Ξ is a uniformly distributed random variable with a probability density function

$$f_\Xi(x) = \begin{cases} 1, & \text{if } -\frac{1}{2} \le x < \frac{1}{2} \\ 0, & \text{otherwise} \end{cases}$$

For implementation purposes, we use the equivalent function:

$$unround(< n, f, e >) =< n, f + 2^{-p} \cdot \Phi - 2^{-p-1}, e > \quad (1)$$

where Φ is a random variable uniformly distributed in the interval $[0, 1)$. The probability density function for Φ is:

$$f_\Phi(x) = \begin{cases} 1, & \text{if } 0 \le x < 1 \\ 0, & \text{otherwise} \end{cases}$$

Let a, b, c be 3 floating point numbers such that

$$\begin{aligned} a &= -1^{n_a} \cdot f_a \cdot 2^{e_a} \\ b &= -1^{n_b} \cdot f_b \cdot 2^{e_b} \\ c &= -1^{n_c} \cdot f_c \cdot 2^{e_c} \\ a &> b. \end{aligned}$$

MCA addition can be defined as

$$mc\_add(a, b) = c = unround(a) + unround(b).$$

This function is realized in hardware by making some changes to the standard floating point addition algorithm. In the standard algorithm, operand significands are aligned by shifting the smaller operand to the right by an amount equal to the difference of the exponents. The aligned operands are then added together. If the normalization step is not considered, this step can be represented by the following equation:

$$f_c = f_a + f_b \cdot 2^{e_b-e_a} + \phi_a \cdot 2^{-p} + \phi_b \cdot 2^{-p} \cdot 2^{e_b-e_a}$$

Our goal is to compute this expression without extending the precision of the adder. In a floating point adder, the adder is $p + 4$ bits wide, where $p + 1$ is the width of the significand, and an additional 3 bits are used for rounding. We will compute f_c under this bit-width constraint. During normalization, the rightmost $e_b - e_a - 3$ bits of f_b are shifted out. In normal floating point addition, rounding of these lost bits is tracked via the sticky bit. Since there are an infinite sequence of random bits to the right of the least significant bit, the sticky bit is not necessary. We denote those bits shifted out f_{b0} and the remaining ones f_{b1}. Hence, $f_b = f_{b0} + f_{b1}$, and f_c can be computed as:

$$\begin{aligned} f_c &= f_a + f_{b1} \cdot 2^{e_b-e_a} + \varepsilon \\ \varepsilon &= f_{b0} \cdot 2^{e_b-e_a} + \phi_a \cdot 2^{-p} + \phi_b \cdot 2^{-p} \cdot 2^{e_b-e_a} \end{aligned}$$

Computing ε exactly would require enlarging the adder to a width of $2p-3$ bits. Instead, we approximate f_{b0} using a uniform random number. This approximation only produces a small error (as confirmed experimentally later in the paper) since the bits shifted out consist of the lower order bits of the number, which are roughly randomly distributed. Under this assumption, we can compute ε using the expression:

$$\varepsilon = \phi_a \cdot 2^{-p} + \phi_x \cdot 2^{-p}$$

where ϕ_x is a random number taken from the distribution Φ, representing all lower order bits of f_b that are not visible. The lower order bits are the sum of two random numbers distributed in the interval $[0, 1]$. Since the lower order bits are not recorded, we only need to know how the sum of the random numbers affect rounding. If A, B are independently distributed uniform random variables $U[0, 1]$, $P(A + B \ge 1) = \frac{1}{2}$ and the uncomputed part of the addition generates a carry $\frac{1}{2}$ of the time, during rounding a carry input is added to the least significant place with probability $\frac{1}{2}$. After taking the carry operation into account, the unrecorded bits are uniformly distributed and its value is simply rounded to the nearest floating point number.

3.2 MCA Multiplication

Let a, b, c be 3 floating point numbers such that $a = -1^{n_a} \cdot f_a \cdot 2^{e_a}$, $b = -1^{n_b} \cdot f_b \cdot 2^{e_b}$, and $c = -1^{n_c} \cdot f_c \cdot 2^{e_c}$. Using the function *unround*() defined in Equation 1, MCA multiplication can be defined as

$$mc\_mult(a, b) = c = unround(a) \times unround(b).$$

This function is realized in hardware by making some changes to the standard floating point multiplication algorithm. The first step is to compute the product of the significand. Ignoring the normalization step, the significand of c can be computed by the following equation:

$$f_c = (f_a + \phi_a \cdot 2^{-p}) \cdot (f_b + \phi_b \cdot 2^{-p})$$

A random error is injected to both operands. Since a floating point multiplier does not use any guard bit in the multiplication stage, the multiplier width must be increased to accommodate the error injected. The number of extra bits added affects how precise the error is propagated after the multiplication. The absolute error of a multiplication is $(f_a\phi_b + f_b \cdot \phi_a) \cdot 2^{-p} + \phi_a\phi_b 2^{-2p}$. Since $\phi_a\phi_b 2^{-2p}$ is a very small value, it is ignored. Taking normalization into account, the relative error can be represented by the equation:

$$\frac{(f_a + f_b) \cdot 2^{-p}}{\lfloor f_a \cdot f_b \rfloor}$$

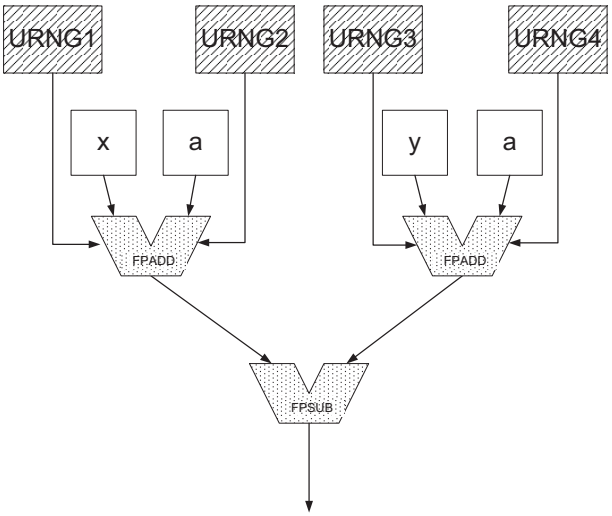

Figure 1: Example showing how correlated error is handled for the expression $(x + a) - (y + a)$. URNG2 and URNG4 are initialized with the same seed and hence have identical sequences.

Since both f_a and f_b are both normalized, $f_a \in [1, 2)$ and $f_b \in [1, 2)$. The maximum relative error is $2(2^{-p})$, and this translates to an error of at most 2 ulp in the result. Since the final result is rounded to 1 ulp, the error cannot be represented in high precision in the result, so there is little advantage to computing the error propagated in high precision. The multiplier width is increased by 1 bit to accommodate the injected error. Zero error is injected with a probability of $\frac{1}{2}$, an error of -1 and $+1$ is injected with a probability of $\frac{1}{4}$.

Rounding is unchanged except that the extra 2 bits from the multiplier output are included the calculation of sticky bit.

3.3 Handling correlated rounding error

In a datapath circuit, when a intermediate result becomes the input of more then one subsequent operation, the rounding error of the input is necessarily correlated. In such a case, our scheme can partially account for the correlation between rounding errors by arranging for all PRNGs corresponding to the same variable to be initialized with the same random seed. This causes all operations using the same intermediate result to perturbate the input using the same random number. Figures 3 and 5 show the modifications made to the adder and the multiplier for handling correlated error.

Figure 1 shows an example of how correlation can be handled. In this example, the expression $(x + a) - (y + a)$ is computed, where a is the result from some previous computation. URNG2 and URNG4 are initialized with the same random seed so that the same random sequence is used in the all unround operations associated with the variable a.

4. IMPLEMENTATION

A floating point adder/subtractor and multiplier incorporating the MCA algorithm described in the previous section is implemented. The floating point unit can operate in either MCA mode or IEEE 754 single precision mode. Our test design is based the floating point unit used in reference [6], which itself is derived from an open source floating point unit [15]. The floating point unit is a IEEE 754 compliant single precision one, supporting all IEEE 754 rounding modes and denormalized numbers. Four pipeline stages are employed, this being optimized for latency rather than maximum clock frequency. It would be possible to add additional pipeline stages to operate at a higher clock frequency. We will highlight the major changes made to support MCA operation.

Listing 1: Combined-Tausworthe Generator

```
unsigned s1,s2,s3,b;

unsigned taus88()
{
  b = (((s1 << 13) ^ s1) >> 19);
  s1 = (((s1 & 4294967294) << 12) ^ b);
  b = (((s2 << 2) ^ s2) >> 25);
  s2 = (((s2 & 4294967288) << 4) ^ b);
  b = (((s3 << 3) ^ s3) >> 11);
  s3 = (((s3 & 4294967280) << 17) ^ b);
  return (s1 ^ s2 ^ s3);
}
```

The architecture for the adder is shown in Figure 2. A 32-bit combined Tausworthe pseudo-random number generator [7] is used to generate the random numbers, and is described in Listing 1. A 32-bit random number is produced each clock cycle by combining the output of 3 Tausworthe generators s1, s2 and s3.

The floating point adder is composed from 4 major modules: the prenormalization unit, the fixed point adder/subtractor, the post-normalization unit and the rounding unit. The parts that are modified for MCA are highlighted in Figure 2. Here we list the modifications made to each of the modules.

- In the prenormalization unit, 3 bits from the URNG are appended to the significand of each operand. A binary value '100' is then subtracted from the significand of each operand.
- In the fixed point adder/subtractor, 1 is added to the sum with a probability $\frac{1}{2}$ for addition. For subtraction, 1 is subtracted from the sum with a probability $\frac{1}{2}$. This is implemented by feeding a random bit to the carry in so no additional adder is required.
- In the normalization left shift, random digits are filled into the least significant bit (LSB) instead of zero.
- In Monte Carlo mode, the sticky bit is ignored, the result is rounded up if and only if the round bit is 1.

Figure 3 shows the modifications made to the adder to account for rounding error correlation. Different PRNGs are used for each operand. The PRNG corresponding to each variable is initialized with a different random seed so the same pseudo-random sequence is used by each fan-out of a variable, as shown in Figure 1.

A diagram for a floating point multiplier using the proposed algorithm is shown in Figure 4. A 32-bit combined

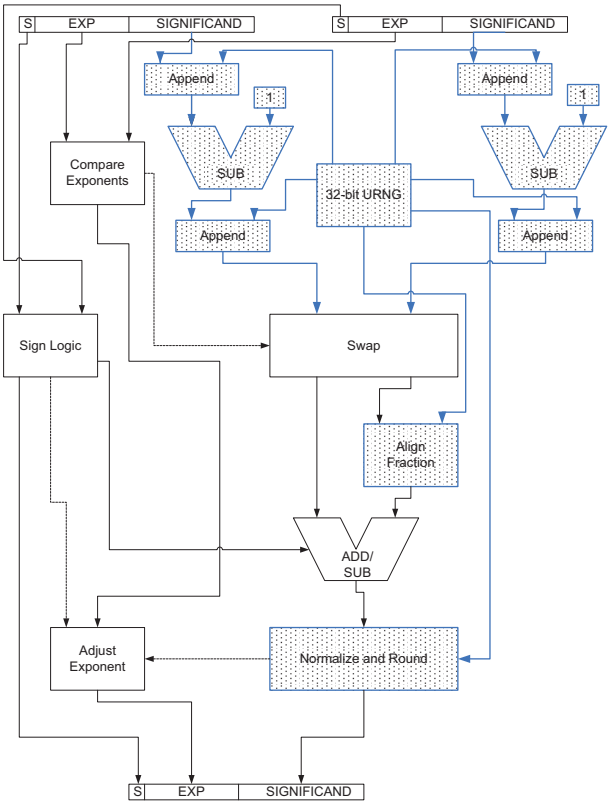

Figure 2: MCA adder data-path.

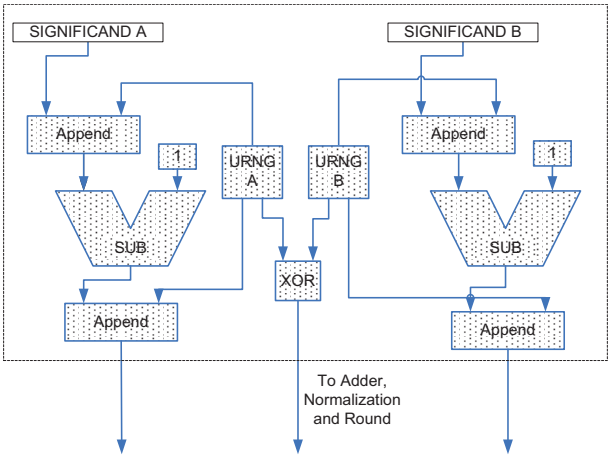

Figure 3: Modification made to adder to account for correlation of rounding error.

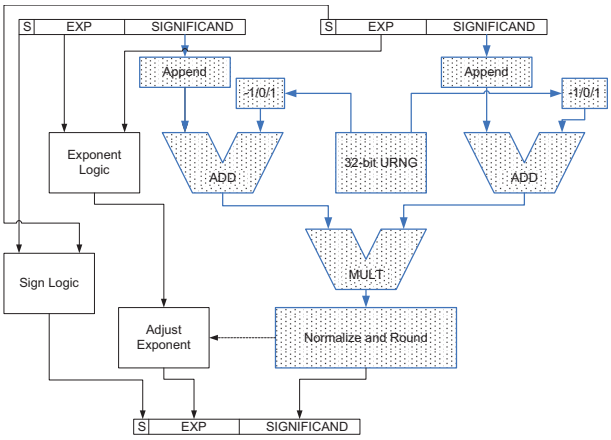

Figure 4: MCA multiplier data-path.

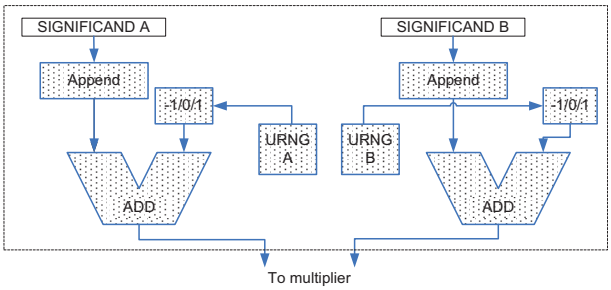

Figure 5: Modification made to multiplier to account for correlation of rounding error.

Tausworthe pseudo-random number generator is used and only 4 bits of the output are used. The modifications for MCA are highlighted in Figure 2. For clarity, only relevant signals are shown. Here we list the modifications made.

- Both operands are extend by 1 bit by appending a zero. Then 1 is either subtracted from or added to the least significant bit with a probability of $\frac{1}{4}$ for addition and $\frac{1}{4}$ for subtraction.
- The multiplier is extended by one bit to 25×25.
- During rounding, the addition output bits from the multiplier is included in the calculation of the sticky bit.

Figure 5 shows the modifications made to the multiplier to account for rounding error correlation. Different PRNGs are used for each operand. The PRNG corresponding to each variable is initialized with a random seed so that the same pseudo-random sequence is used by each fan-out of a variable.

The adder and multiplier are synthesized on a Xilinx Virtex 5 XC5VLX50T FPGA using ISE 11.1 using the default settings of optimising for timing performance with IOB packing. In the current implementation, a single Monte Carlo floating point adder occupied 460 slices and runs at 86 MHz while the unmodified adder occupied 304 slices and runs at 110 MHz. The Monte Carlo multiplier occupies 419 slices, and runs at 90 MHz, while the unmodified multiplier occupies 284 slices, and runs at 86 MHz. Note that the

adder uses only 26 of the 32 output bits of the URNG while the multiplier uses only 4 of the 32 bits. If both the adder and multiplier are implemented, the URNG can be shared, eliminating a URNG.

5. RESULTS

5.1 Logarithmically Distributed Inputs

To test the effectiveness of the algorithm, pairs of double precision floating point numbers are drawn by independently generating the significand and exponent from uniform distributions in the required range. The resulting floating point numbers are logarithmically distributed. The numbers are then rounded to single precision and fed to the MCA floating point unit. A total of 64 Monte Carlo iterations are performed for each pair and the standard deviation of the output computed. The floating point unit is also run in non-MCA mode for 1 iteration to obtain an IEEE 754 single precision result. The results are compared by performing the same operation in double precision using the original pair of double precision floating point number. The rounding error is the difference between the MCA result and the double precision calculation.

Figure 6 shows the results from the MCA floating point adder. A total of 5000 pairs of numbers distributed in the full range of single precision floating point number are tested and the rounding error is plotted against the standard deviation. Here we test how the standard deviation of the MCA simulation can be used to estimate the distribution of the rounding error. If the result of the Monte Carlo iterations are approximately normally distributed, we will expect that most of the values will lie within 3 standard deviation of the population. The dotted line indicates the point where the rounding error is equal to 3 standard deviations. It can be seen that, as expected, the data-points lie below the 3 standard deviation line. Figure 7 shows the same test with the MCA multiplier and similar results are observed.

5.2 Inputs in the range $[1.0, 2.0]$

Figures 8 and 9 show the results of a similar test with the generated range limited to $[1.0, 2.0]$. The exponent is equal to 0 and the significand is uniformly distributed over its full valid range. A total of 100000 pairs of number are tested for the MCA adder and 10000 for the multiplier. The ratio of the rounding error and standard deviation is plotted against the result and a range between 0 and 3 observed. This shows that the standard deviation of the Monte Carlo iterations give a good indication of the distribution of rounding error.

5.3 Comparison with Double Precision Simulation

The addition of pairs of single precision random numbers using MCA is compared to a double precision MCA simulation using a virtual precision of 23 bits. An example of the output distribution for $-1.0 + 1.0009765625$ is shown in Figure 10. The values were chosen to have a large relative error. The output distribution closely matches the double precision simulation. Figure 11 shows the result of a similar test for the MCA multiplier calculating 1.5×1.5, with the x-axis plotted in ulp. The sparse distribution is due to the fact that the error propagation is small compared to 1 ulp.

5.4 Catastrophic Cancellation

Catastrophic cancellation occurs when two inexact floating point values, similar in magnitude, are subtracted. Consider the example $a = 1.10000000000000000000000_2 \times 2^0$, $b = 1.10000000000000000000001_2 \times 2^0$, $c = a - b$. When computed in single precision, the value of c is $1.00000000000000000000000_2 \times 2^{-23} \approx 1.19209 \times 10^{-7}$. This would be the exact result if a and b are exact values. However, when a and b are result of some previous floating point operations, they are subjected to rounding errors. In this case, the value of c contains no more then 1 binary significant digit, the other digits being rounding errors from previous operations. When the same computation is run on our floating point unit in MCA mode over 1024 iterations, we obtain a mean of $1.18244651 \times 10^{-7}$, and standard deviation of 4.913304×10^{-8}. These values clearly indicate that the result contains large rounding errors.

6. CONCLUSION

Based on Monte Carlo Arithmetic, an hardware algorithm for randomising rounding errors was devised and an IEEE 754 compliant single precision floating point unit incorporating the algorithm implemented. Experiments show that the floating point unit gives an accurate estimation of the propagation of rounding error. In our implementation using 4 pipeline stages, the multiplier has no speed penalty and occupies 51% more area than a standard one. The adder has a 22% increase in delay and 47% increase in area. We did not make any attempt to optimize the speed and area of the implementation. Speed could be improved by better balancing of pipeline stages and area can be reduced by sharing of the PRNGs between the adder and multiplier.

This work shows our approach can effectively estimate the propagation of rounding error with a minimal impact on performance. Future work will involve applying this approach at a system level and exploring aggressive floating point optimizations that reduce hardware resources while tracking rounding errors at runtime.

7. REFERENCES

[1] A. Amaricai, M. Vladutiu, and O. Boncalo. Design of floating point units for interval arithmetic. In *Research in Microelectronics and Electronics, 2009. PRIME 2009. Ph.D.*, pages 12 –15, july 2009.

[2] R. Chotin and H. Mehrez. A floating-point unit using stochastic arithmetic compliant with the IEEE-754 standard. In *Electronics, Circuits and Systems, 2002. 9th International Conference on*, volume 2, pages 603 – 606 vol.2, 2002.

[3] J. Cong, K. Gururaj, B. Liu, C. Liu, Z. Zhang, S. Zhou, and Y. Zou. Evaluation of static analysis techniques for fixed-point precision optimization. In *FCCM '09: Proceedings of the 2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, pages 231–234, Washington, DC, USA, 2009. IEEE Computer Society.

[4] C. Fang, T. Chen, and R. Rutenbar. Floating-point error analysis based on affine arithmetic. In *Proceedings of the 2003 IEEE International Conference on Acoustics, Speech, and Signal Processing*. IEEE Computer Society, 2003.

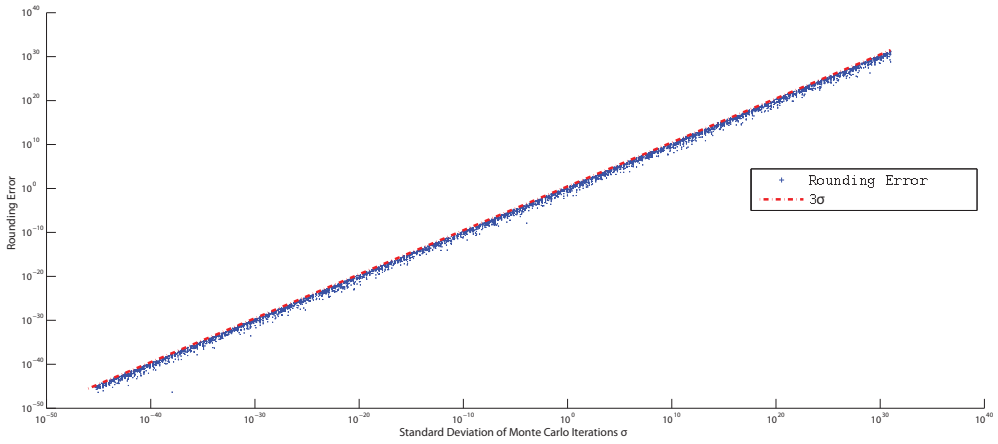

Figure 6: MCA adder - comparison of rounding error to standard deviation of Monte Carlo iterations.

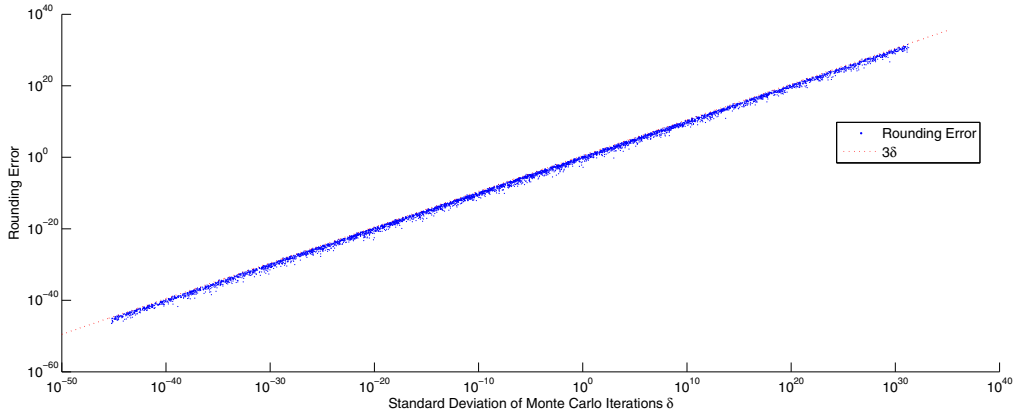

Figure 7: MCA multiplier - comparison of rounding error to standard deviation of Monte Carlo iterations.

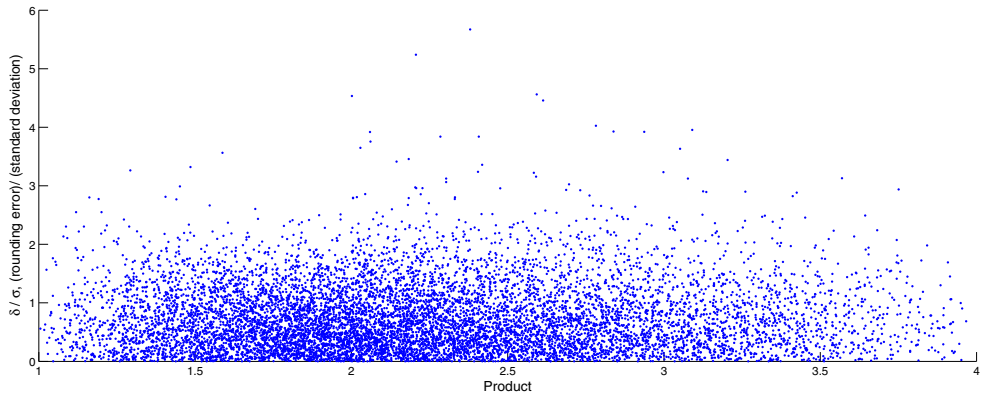

Figure 8: MCA multiplier - rounding error vs standard deviation.

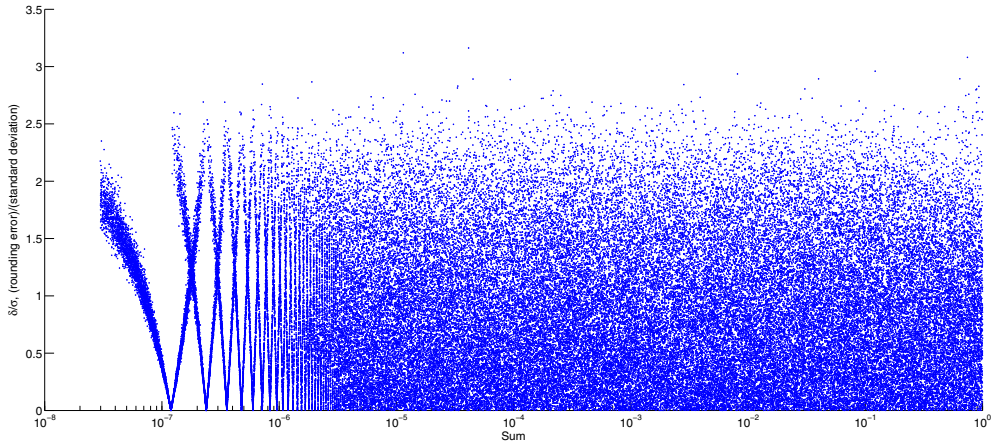

Figure 9: MCA adder - rounding error vs standard deviation.

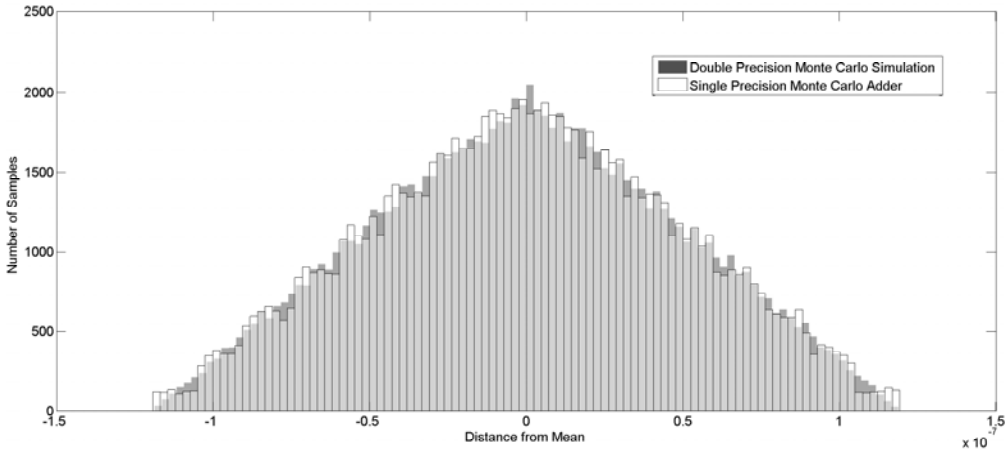

Figure 10: Comparison between hardware MCA addition and double precision MCA simulation.

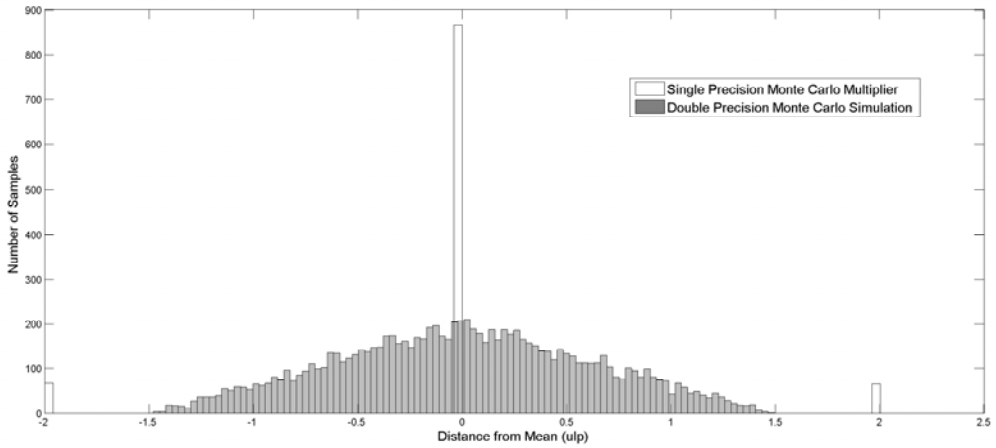

Figure 11: Comparison between hardware MCA multiplication and double precision MCA simulation.

[5] C. F. Fang, R. A. Rutenbar, M. Püschel, and T. Chen. Toward efficient static analysis of finite-precision effects in DSP applications via affine arithmetic modeling. In *Design Automation Conference (DAC 2003*, pages 496–501, 2003.

[6] C. H. Ho, C. W. Yu, P. Leong, W. Luk, and S. J. E. Wilton. Floating-point FPGA: architecture and modeling. *IEEE Trans. Very Large Scale Integr. Syst.*, 17(12):1709–1718, 2009.

[7] P. L'Ecuyer. Maximally equidistributed combined tausworthe generators. In *Math. Computation*, volume 65, pages 203 –213, 1996.

[8] R. E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[9] W. Osborne, R. Cheung, J. Coutinho, W. Luk, and O. Mencer. Automatic accuracy-guaranteed bit-width optimization for fixed and floating-point systems. In *Proc. International Conference on Field Programmable Logic and Applications (FPL)*, pages 617–620, 2007.

[10] D. S. Parker. Monte carlo arithmetic: exploiting randomness in floating-point arithmetic. 1997. http://www.cs.ucla.edu/∼stott/mca/CSD-970002.ps.gz.

[11] D. S. Parker, B. Pierce, and P. R. Eggert. Monte carlo arithmetic: How to gamble with floating point and win. *Computing in Science and Engineering*, 2(4):58–68, 2000.

[12] M. Schulte and J. Swartzlander, E.E. A family of variable-precision interval arithmetic processors. *Computers, IEEE Transactions on*, 49(5):387 –397, may 2000.

[13] J. Stine and M. Schulte. A combined interval and floating point multiplier. In *VLSI, 1998. Proceedings of the 8th Great Lakes Symposium on*, pages 208 –215, feb 1998.

[14] J. Stolfi and L. H. de Figueiredo. *Self-validated numerical methods and applications*. Brazilian Mathematics Colloquium Monograph, IMPA, Rio De Janeiro, Brazil, 1997.

[15] R. Usselmann. Floating point unit, 2009. http://opencores.org/project,fpu.

[16] J. Vignes. A stochastic arithmetic for reliable scientific computation. *Math. Comput. Simul.*, 35(3):233–261, 1993.

[17] P. Vouzis, M. Arnold, S. Collange, and M. Kothare. Monte Carlo logarithmic number system for model predictive control. In *Proc. International Conference on Field Programmable Logic and Applications (FPL)*, pages 453–458, 2007.