Chapter 1

An FPGA-based Floating Point Unit for Rounding Error Analysis

Michael Frechtling and Philip H.W. Leong
School of Electrical and Information Engineering,
The University of Sydney,
NSW 2006,
Australia
michael.frechtling@sydney.edu.au, philip.leong@sydney.edu.au

Detection of floating-point rounding errors normally requires run-time analysis in order to be effective and software-based tools are seldom used due to the extremely high computational demands. In this paper we present a field programmable gate array (FPGA) based floating-point co-processor which supports standard IEEE-754 arithmetic, user selectable precision and Monte Carlo Arithmetic (MCA). This co-processor enables the detection of catastrophic cancellation and minimizing required floating-point precision in reconfigurable computing applications

1.1. Introduction

IEEE-754[1] has long been the standard for computing using floating-point (FP) numbers, however, as a finite precision arithmetic system it is capable of anomalous results. Rounding error during computation can significantly reduce the accuracy of a computation, and result in errors many times larger than expected.[2] In order to properly implement and verify numerical software, techniques to determine the effects of such errors are required. Monte Carlo Arithmetic (MCA)[3] can track rounding errors at run-time by force inputs and outputs to behave like random variables. Analysis of repeated operations turns an execution into trials of a Monte Carlo simulation allowing statistics on the effects of rounding errors to be obtained. MCA is typically performed using SW routines and as such its implementation involves a drastic reduction in performance. Field programmable gate arrays (FPGAs) offer a platform in which hardware (HW) acceleration can

be applied to arbitrary algorithms.

In this work, we describe a complete HW accelerator for run-time error analysis. A novel MCA co-processor architecture is employed which is implemented entirely using standard floating-point cores. System-level performance measurements are described, and a comparison with an existing (SW) implementation made.

This work was influenced by Prof. Cheung's seminal research on optimising floating-point bitwidths to advantageously utilise the reconfigurable nature of FPGAs. It is complementary to his approach to floating-point sensitivity analysis based on automatic differentiation;[4] and can be applied to customised hardware floating-point[5] and dual-fixed point[6] implementation schemes.

1.2. Background

1.2.1. *IEEE-754 Floating Point*

The binary IEEE-754[1] floating-point number system $\mathbb{F}(\beta, p, e_{min}, e_{max})$ is a subset of real numbers with elements of the form:

$$x = (-1)^s m\beta^e \tag{1.1}$$

The system is characterized by the radix β, which is assumed to be 2 in this paper, precision p, the exponent values $e_{min} \leq e \leq e_{max}$, the sign bit $s \in \{0, 1\}$ and the mantissa $m \in [0, \beta)$. Normalized values are most commonly used and are represented as a non-zero $x \in \mathbb{F}$ with $|m| \in [1, \beta)$ and $e_{min} < e < e_{max}$. De-normalized numbers are also supported and represent values of smaller magnitude than normalized numbers with $m \in [0, 1)$ and exponent $e = e_{min}$. Other classes of numbers including $+/-$ Zero, Infinity and Not a Number (NaN) are available with special formats. Without loss of generality, we assume the 32-bit IEEE single precision format in this paper with $p = 24, e_{min} = -125$, and $e_{max} = 128$. Real numbers are generally not exactly representable as FP numbers due to a number of factors including errors of measurement or estimation, quantization error or errors propagated from earlier parts of a computation. Although the IEEE-754 standard is used in all types of applications mostly without issue, if one or more non-exact numbers are subtracted, a loss of significant digits can occur due to normalization of the result.[2] This phenomena is called *catastrophic cancellation* and is one of the major causes of loss of significance.

1.2.2. *Error Analysis*

Several systems have been developed for performing run-time error detection and analysis. Interval Arithmetic (IA) represents a value x by an interval $[x_{lo}, x_{hi}]$. Intervals are propagated through the calculation e.g. $[a_{lo}, a_{hi}] - [b_{lo}, b_{hi}] = [a_{lo} - b_{hi}, a_{hi} - b_{lo}]$. IA can be used to track inexact values and rounding errors during computation, however, it often produces overly pessimistic error bounds.[7] A limited number of hardware implementations of IA can be found in the literature.[8–10] The CESTAC method[11] is a special case of MCA that involves executing the same computation several times by randomly perturbing the the rounding scheme of the arithmetic operators. Comparing the results from a number of different executions, the number of significant digits can be estimated. A hardware implementation of the CESTAC method has also been published,[12] but we are not aware of any system level implementation. An FPGA based implementation of MCA addition and multiplication with an area penalty of less than 22% over IEEE-754 was recently published by Yeung et. al.[13] Compared with their implementation, this work presents a complete system for MCA rather than just an MCA core. In addition, while Yeung described a custom FPU for MCA, the FP computations in this work are performed using standard IEEE-754 FP primitives, providing significant benefits in terms of portability, flexibility and development time.

1.3. Monte Carlo Arithmetic

If x is a floating-point value of the form given in Equation 1.1 we define the *inexact* function as:

$$\begin{aligned} inexact(x, t, \xi) &= x + \beta^{e_x - t}\xi \qquad (1.2) \\ &= (-1)^{s_x}(m_x + \beta^{-t}\xi)\beta^{e_x} \qquad (1.3) \end{aligned}$$

where $x \in \mathbb{R}$, t is a positive integer representing the desired precision, ξ is a uniformly distributed random variable in the range $(-\frac{1}{2}, \frac{1}{2})$, $(\xi \in U(-\frac{1}{2}, \frac{1}{2}))$ and m_x, e_x are the mantissa and exponent of x. It is assumed that $1 < t \leq p$. An operation $\circ \in \{+, -, \times, \div\}$ is implemented as:

$$x \circ y = round(inexact(inexact(x) \circ inexact(y))) \qquad (1.4)$$

Adjustments to input operands are referred to as *precision bounding* and are used to detect catastrophic cancellation, while adjustments to outputs are referred to as *random rounding* and are used to detect round-off error.[3]

The system developed for this paper performs precision bounding using multiple floating-point computations. Using this method the operation can be performed without modifying the internal architecture of the standard IEEE-754 FPU, however, the use of standard FP operations results in FP rounding being applied multiple times within a single MCA operation. In the case of traditional MCA the average of a Monte Carlo simulation can be used to estimate the true result of an operation sensitive to rounding error, and the relative standard deviation used to detect catastrophic cancellation. In the case of MCA implemented for this paper, the average value of the results of a Monte Carlo simulation cannot be used to estimate the true result of the tested operation, as this average is affected by the use of multiple rounding stages. In this case the system is only used for the detection of catastrophic cancellation. The value t shown in equation 1.4 is the *virtual precision* of the MCA operation. This value determines the number of places the value ξ is shifted to the right of the mantissa of the floating-point value x, and is used to control the level of random fluctuations applied during MCA. The virtual precision of the MCA operations is set to a positive integer less than or equal to the machine precision of the floating-point system being used:

$$1 \leq t \leq p$$

A large t value will result in a smaller exponent value for the operand perturbation, increasing the accuracy of the operation. Similarly, a smaller t decreases the accuracy. In practice, variation of t is used to determine what effect lowering or increasing the precision has on the accuracy of an operation. The results of this analysis can then be used to determine an appropriate value for the machine precision p of a floating point operation that will maintain a required level of accuracy. The implementation developed for this paper performs *variable precision MCA*, and the value of t used by the co-processor can be modified at any time during execution. Further details are provided in Section 1.4.

1.4. System Implementation

The MCA FPU is an FPGA co-processor connected to a μBlaze soft processor through a AXI4-stream bus. The co-processor is capable of performing both standard floating-point arithmetic and MCA. The MCA FPU core was developed using the high-level C-to-RTL design software AutoESL (http://www.xilinx.com/tools/autoesl.htm). Using AutoESL, our

MCA FPU is described using standard C statements and during synthesis and implementation, floating-point operations are translated into a set of floating-point modules based on the IEEE-754 floating-point library.

1.4.1. *MCA FPU Implementation*

The MCA FPU is able to perform four basic arithmetic operations; add, subtract, multiply and divide. A fifth configure operation allows the precision value, t, and the MCA flag to be modified at run-time. The final operation combines the add and multiply operations to perform an FMA (Fused Multiply-Add) operation, which calculates the result of the operation $r = (a * b) + c$. The arithmetic operations are implemented by coupling standard IEEE-754 floating-point operator primitives with a configuration register and a perturbation generation module. Each perturbation module is used to determine a value that will be added to the operation based on the value of the operands and a random number. Random numbers are generated using *Maximally Distributed Tausworthe Generators* (TRNG) .[14] The configuration information consists of a 1-bit boolean flag indicating MCA or IEEE-754 mode, and a 32-bit unsigned value for t. These are stored in the configuration register for access during subsequent operations. In IEEE mode, the FPU fully supports the standard. A description of how the operators are implemented is given below.

1.4.1.1. *MCA Addition/Subtraction:*

Addition and subtraction operations are performed in terms of the *inexact* function (Equation 1.3) as follows:

$$\begin{aligned} x \pm y &= round(inexact(x) \pm inexact(y)) \\ &= round((x + \xi_x) \pm (x + \xi_y)) \\ &= round(x \pm y + \xi) \end{aligned}$$

where $\xi_x, \xi_y \in U(-\frac{1}{2}, \frac{1}{2})$ and $\xi = \xi_x \beta^{e_x - t} + \xi_y \beta^{e_y - t}$. The magnitude of ξ can be calculated using only positive values via:

$$|\xi| = \beta^{e_x - t}(|\xi_x| \pm |\xi_y| \beta^{-(e_x - e_y)})$$

and a equal probability choice of addition or subtraction is made. Note that $|\xi| \in \beta^{e_x - t}[0, 1)$ and its distribution depends on the value of x and y. The floating-point value ξ can be calculated by first using fixed point arithmetic to produce separate values for the e_ξ and m_ξ then combining these values

along with a randomly selected value for s_ξ in the correct format to produce a floating-point number:

$$m_\xi = \xi_x + [\xi_y \beta^{-(e_x - e_y)}],\ e_\xi = e_x - t$$

To perform this operation two random values for ξ_x and ξ_y must be calculated. These values will be used to form m_ξ and as such must be 24-bit normalized fixed point values. Each value must also be in $U[0, \frac{1}{2})$. Two 32-bit values are produced (one from each TRNG) and the lower 22-bits assigned as the fixed point value of ξ_x or ξ_y. The MSBs of each 32-bit number are used to calculate the sign bit s_ξ. Once the fixed point values of ξ_x and ξ_y have been produced the value of m_ξ can be calculated using fixed point arithmetic At this point we have produced a value $m_\xi \in [0, 1)$. To produce the final value for ξ this value must be normalized, the $\beta^{e_x - t}$ shift applied and the value converted to IEEE-754 single precision format. This is done in the following stages:

(1) Determine the number of leading zeroes λ_ξ. The leading zero detector (LZD) used for the Monte Carlo FPU is based on the LZD found in.[15] The mantissa value m_ξ is then shifted left or right depending on the value of λ_ξ forming the final 24 bit normalized mantissa.
(2) Calculate the 8-bit exponent value based on the value of e_x, λ_ξ and t
(3) Merge the sign, exponent and mantissa values to form the single precision floating-point value using left-shifts to move the sign and exponent values to the correct location.

Once a value for ξ has been produced the second addition operation is performed, producing the final result:

1.4.1.2. *MCA Multiplication:*

The multiplication operation can be represented in terms of the *inexact* function shown in Equation 1.4 as follows:

$$\begin{aligned} xy &= round((x + \xi_x)(y + \xi_y)) \\ &= round(xy + x\xi_y + y\xi_x + \xi_x\xi_y) \end{aligned}$$

The perturbation values in the above equation can be expanded and simplified to the following:

$$\xi = \beta^{e_x + e_y - t}[m_x\xi_y + m_y\xi_x + \xi_x\xi_y\beta^{-t}]$$

From the above equation it can be seen that the $\xi_x\xi_y$ term will be shifted to the right by $2t$ places during the operation. Calculation of the perturbation

value including this term would require the precision of the FPU to be extended, either by modifying the internal architecture of the FPU core or by performing the Monte Carlo calculation in a higher precision format. This can be avoided by not including the $\xi_x\xi_y$ term in the calculation, which can be done without significantly affecting the results as the large right shift results in an extremely small value for $\xi_x\xi_y$ relative to $m_x\xi_y + m_y\xi_x$. The Monte Carlo multiplication operation can therefore be simplified to the following:

$$xy = round(xy + \xi)$$

where $\xi_x, \xi_y \in U[0, \frac{1}{2})$ and $\xi = \beta^{e_x+e_y-t}[m_x\xi_y + m_y\xi_x]$. The magnitude of ξ can be calculated using only positive values via:

$$|\xi| = \beta^{e_x+e_y-t}(|m_x\xi_y| \pm |m_y\xi_x|) \qquad (1.5)$$

where a randomized, equal probability choice of addition or subtraction is made. Note that $|\xi| \in \beta^{e_x+e_y-t}[0, 2)$ and it's distribution depends on the value of x and y. In MCA multiplication a similar method to addition is employed to produce the perturbation value. Two TRNGs are first used to produce 24-bit fixed point random numbers and the corresponding sign bits. These represent $\xi_x, \xi_y \in (-\frac{1}{2}, \frac{1}{2})$. Using Equation 1.5, each value is then multiplied by the mantissa of the relevant operand and the resulting values added together, resulting in a value for m_ξ. This process produces a value $m_\xi \in (-2, 2)$. This value is used to produce a single precision floating-point value for ξ as follows:

(1) Determine the number of leading zeroes λ_ξ in m_ξ and normalize
(2) Calculate the exponent value e_ξ
(3) Merge the sign, exponent and mantissa values to form the 32-bit floating-point perturbation value

Once the final perturbation value ξ has been produced it is added to the initial result r' to produce the final result of the operation:

1.4.1.3. *MCA Division*

The division operation differs from addition, subtraction and multiplication in that two individual floating-point perturbation values ξ_x and ξ_y are produced rather than a single perturbation value ξ. This operation can be

described in terms of Equation 1.4 as follows:

$$\frac{x}{y} = round(\frac{inexact(x)}{inexact(y)}) = round(\frac{x + \xi_x}{y + \xi_y})$$

The above equation cannot be easily simplified to a point where a combined value for ξ can be calculated as for previously discussed operators. Separate perturbation values (ξ_x and ξ_y) are therefore calculated and applied to the x and y operands, requiring the precision of the division operation to be extended. This is done by performing single precision (32-bit) MCA division using double precision (64-bit) floating-point division. The perturbation values are calculated as follows:

$$\xi_x = \beta^{e_x - t} U(-\frac{1}{2}, \frac{1}{2}),\ \xi_y = \beta^{e_y - t} U(-\frac{1}{2}, \frac{1}{2})$$

Each perturbation value is applied to the relevant operand using a standard IEEE-754 floating-point addition operation, after which a standard IEEE-754 double precision division operation is performed. Although this calculation requires a total of three FP operations, calculation and addition of the two perturbation values can be performed in parallel, and the only increase in overhead over addition/multiplication is from the double precision division operation. MCA division is performed as follows. Two TRNGs are used to produce 24-bit fixed point mantissa values for ξ_x and ξ_y and their corresponding sign bits. The values are then converted to single precision floating-point format

(1) Determine the number of leading zeroes λ_{xi} in each mantissa and normalize.
(2) Calculate the exponent values
(3) Merge the sign, exponent and mantissa values.

Once the perturbation values ξ_x and ξ_y have been calculated the final result is calculated:

1.5. Testing Methods

Testing of the FPU was conducted by comparing the performance of the co-processor to a SW implementation of MCA. The test routines used are based on routines used by Parker in reference,[16] downloadable from http://www.cs.ucla.edu/∼/stott/mca. Details of equipment and parameters are in Table 1.5. In order to compile unmodified C source code to use MCA, two different versions of the gcc software floating-point library

Table 1.1. System Parameters

Item	**Version / Description**
FPGA Parameters	
ISE Version	13.2
FPGA	Virtex-6 LX240T (Speed Grade 3)
FPGA Board	Xilinx ML-605 Development Board
Processor Clock Speed	150 MHz
MCA Core Clock Speed	150 MHz
PC Parameters	
CPU	Intel Core 2 Duo 3 GHz
Memory	4 GB
OS	Ubuntu 12.04 32-bit
GCC Version	4.7.0

were developed. For the FPGA case, a library in which the addsf3, subsf3, mulsf3 and divsf3 were changed to utilise the MCA co-processor was created. FP operations can then be redirected to the appropriate subroutine by invoking gcc with the -msoft-float option. PC implementations of MCA were compiled in a similar fashion using a different, software-only MCA library. For each test case discussed below three tests were performed. The first was on a PC using a floating-point unit (SW-FP); the second a PC with software MCA (SW-MCA); and finally, an FPGA using the Monte Carlo co-processor (HW-MCA).

1.5.1. *Cancellation (Knuth) Test*

The Cancellation test performs a simple associativity test by calculating

$$u = (x + y) + z, v = x + (y + z)$$

Over the real numbers, u should equal v, however, for the values $x = 11111113.0, y = -11111111.0, z = 7.5111111$ catastrophic cancellation occurs. Using these values the difference between $|u|$ and $|v|$ is calculated over 1000 samples and the standard deviation of the results is used to determine the accuracy of the calculation.

1.5.2. *Cosine Test*

The Cosine test calculates the cosine function using a power series expansion:

$$cos(z) = 1 - \frac{1}{2!}z^2 + \frac{1}{4!}z^4 - \frac{1}{6!}z^6 + \frac{1}{8!}z^8 - ...$$

for $z \in [0, \pi]$. For each value of z over n steps a set of 100 samples are calculated and compared to the value of a single precision FP calculation of the same value, and the accuracy of the calculation measured at each step.

1.5.3. *Kahan Test*

The Kahan test performs an evaluation of a rational polynomial:

$$rp(x) = \frac{622 - x(751 - x(324 - x(59 - 4x)))}{112 - x(151 - x(72 - x(14 - x)))}$$

The polynomial $rp(u)$ is first evaluated for $u = 1.60631924$ using single precision IEEE arithmetic, then n results for $rp(x)$ are calculated using MCA for increasing values of x:

$$x = u, (u + \epsilon), (u + 2\epsilon), (u + 3\epsilon), ..., (u + n\epsilon)$$

with $\epsilon = 2^{-23}$. The difference value $d = rp(x) - rp(u)$ is then calculated for each iteration. Results of both the MCA test and the standard IEEE-754 test can then be compared to determine the difference in result distribution

1.5.4. *LINPACK*

The LINPACK benchmark determines system performance by measuring the time taken to solve a dense $n \times n$ system of linear equations $Ax = b$.[17] Using this benchmark performance of the system can be easily measured using an industry benchmark tool, and compared to the performance of an equivalent SW solution. Statistical measurements of the results for x have also been made, and precision testing performed using the benchmark to demonstrate the use of variable precision MCA. In this work, $n = 300$ was used.

1.6. Results

1.6.1. *System Performance & Size*

Tables 1.6.1 and 1.6.1 provide performance results and logic utilization figures for the MCA co-processor. The primary test used to determine system performance is the LINPACK benchmark, as this is an industry standard example of a FP benchmark tool. In order to achieve maximum performance the LINPACK benchmark was profiled to determine which

Table 1.2. System Performance (Measured)

Test Type	**MFLOPS**	**Mean**	**Std Dev**
Cosine			
SW-FP	990	0.0	0.0
SW-MCA	5.5	0.69	0.000009
HW-MCA	4.7	0.69	0.000009
Cancellation			
SW-FP	1900	0.0	0.0
SW-MCA	5.2	0.51	0.59
HW-MCA	2.7	0.51	0.59
Kahan			
SW-FP	1800	0.0	0.0
SW-MCA	4.2	0.000014	0.000033
HW-MCA	1.5	0.000012	0.000047
LINPACK			
SW-FP	2350	1.0	0.0
SW-MCA	4.4	1.0	0.0000000004
HW-MCA	3.5	1.0	0.0000000004

functions would provide the most benefit from optimization. These functions were optimized by performing operations of the type $(a * b) + c$ using the co-processor FMA operation. The results of the LINPACK test show that the MCA co-processor achieved a speed of 3.5 MFLOPS for the LINPACK test, and this figure corresponds with the average performance of the system during all test routines. This performance can be compared to the PC performing MCA in software, which can be seen to have achieved an average speed of 4.4 MFLOPS during the LINPACK test, which is again similar to the average speed of 4.75 MFLOPS achieved across all test routines. From this comparison two things are noted, firstly, there is a $200\times$ to $600\times$ decrease in performance for software-based MCA over standard FP. Secondly, the FPGA implementation has comparable performance to the SW implementation. Table 1.6.1 shows that this performance has been achieved with a 5x increase in logic utilization over a single precision IEEE FP unit capable of performing the same arithmetic operations. Compared with Yeung et. al.,[13] this design has similar throughput but a considerable area overhead due to implementing/interfacing each MCA operation separately in order to reduce I/O overhead

1.6.2. *Improving Performance*

The FPGA MCA core implementation has not been fully optimised. Performance is limited by I/O overhead, a conflict between the -msoft-float and

Table 1.3. System Logic Utilization

Operation	**DSP48E**	**FF**	**LUT**	**% Inc. (Avg)**
ADD	2	595	2067	301%
SUB	2	595	2067	301%
MUL	9	905	2299	725%
DIV	6	4476	7383	761%
FMA	9	1263	3618	383%
TOTAL	28	7834	17434	495%

optimisation flags in gcc and the maximum clock speed of the implementation. In order to overlap communication with computation the LINPACK benchmark was profiled, with results indicating that 92% of computation time was spent in the `daxpy` subroutines. In addition, analysis of the co-processor I/O overhead showed 80% of execution time spent transferring data and only 20% on computation. The maximum speedup, S, for LINPACK can be calculated using Amdahl's Law:[18]

$$\begin{aligned} S &\leq \frac{P}{1 + f(P-1)} \\ &\leq \frac{5}{1 + [0.08 * (5-1)]} \\ &\leq 3.79 \end{aligned}$$

where $P = 5$ is the maximum speed increase achievable by minimizing I/O, and $f = (1 - 0.92) = 0.08$ is the ratio of non-daxpy to daxpy computation. This value could be approached since the daxpy operations are vectorisable. Further performance improvements can be obtained using the gcc optimization flags, which were not used in this case due to conflicts with the -msoft-float option. Testing of routines described in this paper by directly modifying the source code showed that a consistent 2x speed improvement is achieved with -O1 optimization. Finally the μBlaze maximum clock frequency limits the overall system to a clock frequency to 150MHz, while the Zynq family of processors recently released by Xilinx are capable of clock frequencies of 800MHz. In addition Xilinx LogiCORE IP FP operator cores can achieve a maximum frequency of 400MHz. Thus a conservative estimate for the maximum frequency of an optimized design is 400MHz, a $2.5\times$ speedup over the reported design. The optimizations in this section are orthogonal and, taken togegther, an additional $\sim 20\times$ speedup may be possible. This would improve the performance of our approach to 60 MFLOPS.

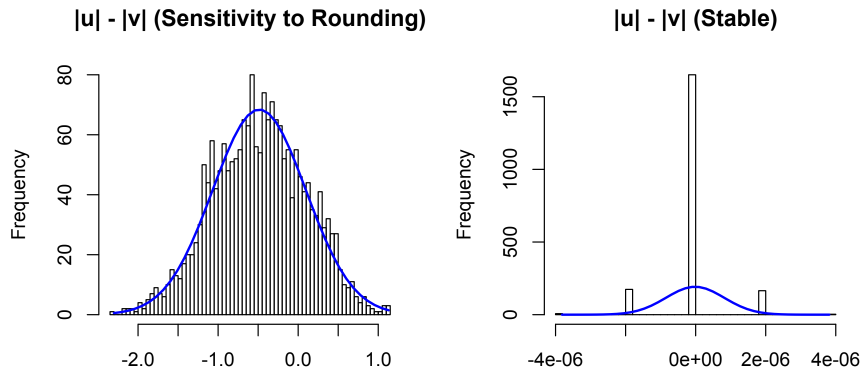

Fig. 1.1. Distribution of results for $[(x + y) + z] - [x + (y + z)]$

1.6.3. *Error Detection*

Figure 1.1 shows the results of two runs of the Cancellation test. The histogram on the right shows results using the values given in section 1.5.1 and demonstrates distribution of results for operations susceptible to round-off error. From table 1.6.1 it can be seen that the results of this test have a standard deviation of 0.51, this being of the same order as the mean, 0.59. It can thus be concluded that, for the given inputs, the co-processor detected catastrophic cancellation. The distribution of these results can also be compared to the histogram on the right side of Figure 1.1 which show results for another execution of the cancellation test using different inputs. From the histogram, it can be seen that the standard deviation and relative standard deviation of the results are much lower and hence, for these inputs, lower sensitivity to rounding error is indicated. The results of the Kahan test are shown in Figure 1.2. The plot on the right side of the figure shows the results of the Kahan test when performed using standard IEEE-754 FP operations, and as can be seen in the plot the results do not show random rounding. The plot on the left side of the figure shows results obtained using the MCA co-processor. Comparing these two sets of results it can be seen that MCA operations performed by the co-processor produce randomly rounded results, and that statistical analysis of these results can be used to determine the sensitivity of the system to rounding error.

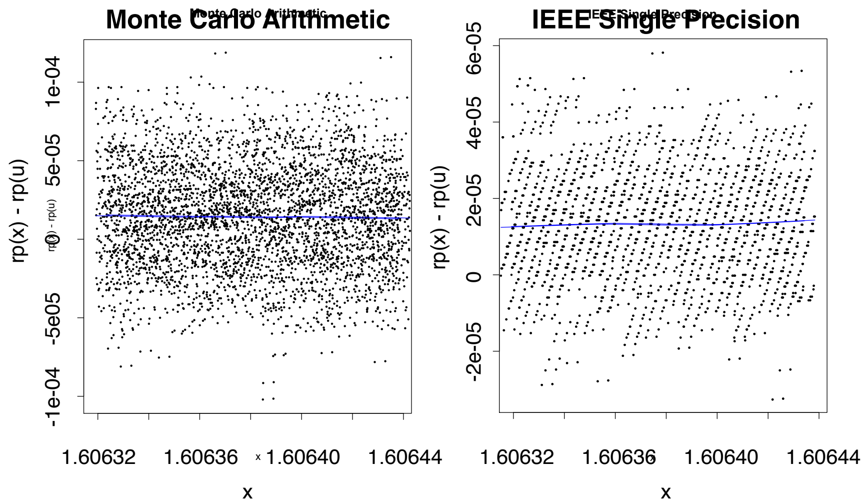

Fig. 1.2. Kahan Test Results

1.6.4. *Precision Testing*

The final set of testing and results demonstrate the ability of the MCA co-processor to perform variable precision MCA, and to determine the minimum precision required to perform an operation to a specified accuracy. The LINPACK benchmark is executed using $n = 300$ and the virtual precision of MCA operations varied between $1 \leq t \leq 24$. The value of the result vector x is then analysed to determine the accuracy of the results, which are given in Figure 1.3. For a required output accuracy, (specified in terms of either the minimum number of significant figures or the minimum realtive standard deviation), the minimum required machine precision is the corresponding abscissa in Figure 1.3.

1.7. Conclusion

A floating-point unit for the run-time detection of round-off errors was designed and implemented using high-level synthesis tools. It was integrated in a μBlaze soft processor system, and verified to give results of accuracy equivalent to previously published software implementations. Measurements showed that the performance of the current implementation is similar to an equivalent PC-based SW implementation, and that a further

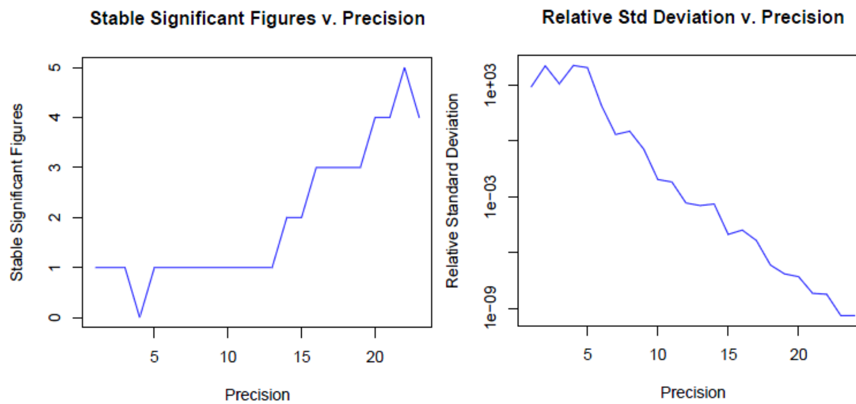

Fig. 1.3. Precision Test results

speedup of 20× is possible.

This work shows that HW accelerated implementations of error detection algorithms can provide accurate measurements of the effects of rounding error while not dramatically impacting performance. Future work will focus on better integration between the FPU and processor, improving performance.

References

1. IEEE Task P754, *IEEE 754-2008, Standard for Floating-Point Arithmetic*. IEEE, Aug. 2008.
2. D. Goldberg, “Computer Arithmetic,” *Computer Architecture: A Quantitative Approach*, 1990.
3. 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*, vol. 2, no. 4, pp. 58–68, 2000.
4. A. A. Gaffar, O. Mencer, W. Luk, and P. Y. K. Cheung, “Unifying bit-width optimisation for fixed-point and floating-point designs,” in *In 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM04*. Society Press, 2004, pp. 79–88.
5. A. A. Gaffar, W. Luk, P. Y. K. Cheung, N. Shirazi, and J. Hwang, “Automating customisation of floating-point designs,” in *FPL*, ser. Lecture Notes in Computer Science, M. Glesner, P. Zipf, and M. Renovell, Eds., vol. 2438. Springer, 2002, pp. 523–533.
6. C. T. Ewe, P. Y. K. Cheung, and G. A. Constantinides, “Dual fixed-point: An efficient alternative to floating-point computation,” in *FPL*, ser. Lecture

Notes in Computer Science, J. Becker, M. Platzner, and S. Vernalde, Eds., vol. 3203. Springer, 2004, pp. 200–208.

7. W. Kahan, "How futile are mindless assessments of round-off in floating point computation," 2006, http://www.cs.berkeley.edu/∼wkahan/Mindless.pdf.
8. 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.*, july 2009, pp. 12 –15.
9. M. Schulte and E. Swartzlander, "A Family of Variable-precision Interval Arithmetic Processors," *Computers, IEEE Transactions on*, vol. 49, no. 5, pp. 387 –397, may 2000.
10. J. Stine and M. Schulte, "A Combined Interval and Floating Point Multiplier," in *VLSI, 1998. Proceedings of the 8th Great Lakes Symposium on*, feb 1998, pp. 208 –215.
11. J. Vignes and R. Alt, "An efficient stochastic method for round-off error analysis," in *Accurate Scientific Computations*, ser. Lecture Notes in Computer Science, W. L. Miranker and R. A. Toupin, Eds., vol. 235. Springer, 1985, pp. 183–205.
12. 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*, vol. 2, 2002, pp. 603 – 606 vol.2.
13. J. H. C. Yeung, E. F. Y. Young, and P. H. W. Leong, "A Monte-Carlo Floating-Point Unit for Self-validating Arithmetic," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2011, pp. 199–208.
14. P. L'Ecuyer, "Maximally Equidistributed Combined Tausworthe Generators," *Mathematics of Computation*, 1996.
15. V. G. Oklobdzija, "An Algorithmic and Novel Design of a Leading Zero Detector Circuit: Comparison with Logic Synthesis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 2, no. 1, pp. 1–5, Mar. 1994.
16. D. S. Parker, "Monte Carlo Arithmetic," Oct. 2003, http://www.cs.ucla.edu/∼stott/mca/.
17. J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK Benchmark: Past, Present and Future," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 9, 2003.
18. G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485.