FPGA Paranoia: Testing numerical properties of FPGA floating point IP-cores

Xuan You Tan, David Boland, and George Constantinides

Electrical and Electronic Engineering, Imperial College London,
London, SW7 2AZ, UK

Abstract. In the early days of computing, hardware platforms were developed independently and created their own conventions for floating point to suit their underlying hardware architecture, but this meant computer programmers had to understand these conventions when designing their algorithms, and adapt their algorithms when porting to new platforms. As a result, the IEEE-754-1985 standard was created to simplify design for computer programmers by ensuring that the same software will obtain the same results across all hardware platforms. While most computers largely adhere to the standard, sometimes corner cases can be missed. Paranoia is a test suite written by William Kahan in 1983, designed to discover obvious flaws in non-compliant floating point arithmetic. The Paranoia test suite continues to show errors and inconsistencies in modern computers and compiler libraries, and has recently found similar flaws in GPUs [1]. FPGAs have historically been used to create custom hardware designs, with a focus on performance for an application specific design, meaning such portability has not been an issue. However, transistor scaling has led to FPGAs with the potential for high floating point performance, and as such FPGA-based accelerators are increasingly adopting standard single or double precision cores within hardware accelerators for high-performance computing applications. As a result, this paper has created a framework to allow FPGA IP-cores to be tested against the Paranoia benchmark to ensure that FPGA IP-cores can been subjected to the same rigorous testing as their CPU equivalents. In this paper, we discuss this effort and provide compliance results for the main vendor and open source core generators.

Keywords: FPGA, Paranoia, IEEE 754 Floating Point Arithmetic

1 Introduction

Since computers were first invented, a fundamental issue has been how to approximate the uncountably infinite set of real numbers with a finite representation. In general, the high performance and scientific computing community has settled on floating point representation. Initially, inconsistencies between a large number of available platforms each having their own definition of floating point resulted in differing results among various floating point libraries [2]. In 1977, Robert Stewart spearheaded the first standard for floating-point arithmetic, which would eventually become IEEE-754-1985 [2, 3].

Given that GPUs and FPGAs are increasingly being used in high performance computing due to their ability to exploit inherent parallelism in an algorithm [4, 5], these devices should be subjected to the same degree of scrutiny so as to ensure any performance comparisons using a standard precision are fair and that non-compliance is both known and accounted for; we note that the well-known crash of the Ariane 5 satellite was a result of non-compliance with the IEEE 754 standard [6]. To this end, previously there has been some investigation into GPU floating point units [1]. In this paper, we develop a VHDL test suite that allows us to verify how well existing FPGA floating point operators conform to this standard and to ensure that any shortcomings are well documented for future FPGA floating point operator design.

2 Background

In this section, we first provide a basic background into floating point numbers and how they are defined using the IEEE standard, alongside some background into the field of hardware verification. We then introduce the main FPGA core generators that we test for IEEE compliance within our new framework.

2.1 Floating Point Representation

The general form of a floating point representation is given by equation (1), where s is a sign bit, R known as the radix (most commonly $R = 2$), e is the exponent, 'offset' is used to allow negative powers of the radix while keeping e non-negative in binary bit representation, and m is the mantissa [7].

$$(-1)^s \times R^{e-\text{offset}} \times 1.m \tag{1}$$

Given that floating point numbers map the infinite real number space onto a finite number of bits, certain errors must be present and tolerated. In floating point, there are three basic sources of error: rounding, underflow and overflow. Figure 1 presents a number line expressing the positive numbers for a "toy" binary floating point number system with a 2-bit exponent e which lies in the range $0 \leq e \leq 3$ (offset = 0), and 3-bit mantissa, and illustrates these errors. In this figure, numbers greater than 16 cannot be represented, showing overflow error; to reach larger numbers, a larger exponent is required. The potential round-off error is a function of the exponent, as seen in Figure 1, where the round-off error is guaranteed to be no more than 0.125 in the region $2 \leq x \leq 4$, but guaranteed to be no more than 0.5 in the region $8 \leq x \leq 16$. In the region $0 \leq x \leq 1$, no numbers can be represented and there is a large potential error compared to the neighbouring region; this is a result of underflow error.

FPGAs can implement any precision and have the ability to trade these errors for the performance and area of the circuit, but because of the increasing number of FPGA accelerators for scientific computing applications that adopt a standard representation [8], in this paper, we restrict to single precision results to focus on how closely existing FPGA IP-cores adhere to the IEEE floating point

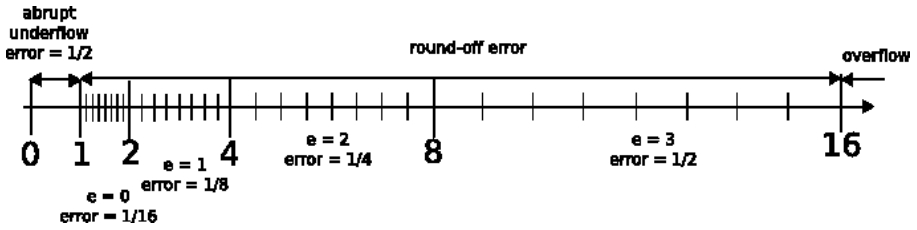

Fig. 1. Number line for an unsigned floating point representation where the exponent is 2 bits over the range $0 \leq e \leq 3$ and mantissa is 3 bits.

standards; we describe the main additional specifications within the IEEE 754 floating point standard in comparison to a basic floating point implementation in the following section.

2.2 The IEEE standard

Rounding: Guard, Round and Sticky Bits The IEEE-754 standard defines four rounding modes: "round to nearest even", "round towards $+\infty$" (ceil), "round towards $-\infty$" (floor), and "round to zero" (truncate) [9], with most software adopting the first mode. In order to ensure correct rounding, as defined in [10], additional bits are necessary within the internal floating point units. For example, suppose we were trying to compute the result of $16 - 15 = 1.000 \cdot 2^4 - 1.111 \cdot 2^3$. After normalisation, this becomes $(1.000 - 0.111) \cdot 2^4 = 0.001 \cdot 2^4 = 1.000 \cdot 2^1$. Even though the correct result is representable with three bits ($1.000 \cdot 2^0$), the result returned is incorrect because a bit was lost during the right shift operation.

To ensure correct rounding, the IEEE Standard defines three bits additional bits to be suffixed to a floating point representation, namely the guard, round, and sticky bits. The guard and round bits are used as classic bits to locally increase the mantissa precision, after normalisation, while the sticky bit represents whether the result is exact, to ensure correct rounding for the basic arithmetic functions.

The IEEE standard: Subnormal numbers Kahan proposed and helped standardise a representation called subnormal (or denormal) numbers, where if the exponent is all zeros, the "implied leading one" in equation (1) in the mantissa is turned into 0, changing the floating point representation to (2). This removes the "abrupt underflow" gap, seen in Figure 1, and ensures that the maximum rounding error does not increase when the number represented approaches 0 [11].

$$(-1)^s \times R^{-\text{offset}} \times 0.m \tag{2}$$

Exceptions and Flags The standard defines five types of exception which should be flagged, and either create a default result or pass the an argument back to an exception handler; these are shown in Table 1.

Table 1. IEEE 754 exceptions and default values [10].

Exception	Default Result
overflow	$\pm\infty$
underflow	0 or denormal
divide by zero	$\begin{cases} \infty & \text{if } a \div 0, a \in \mathbb{R}^+ \\ -\infty & \text{if } a \div 0, a \in \mathbb{R}^- \\ \text{NAN} & \text{if } 0 \div 0 \end{cases}$
invalid	NAN
inexact	round(result)

Single and Double Precision While the above specifications within the IEEE-754 floating point standard are defined across all precisions in radix-2 and radix-10 arithmetic, most users will only be aware of two specific precisions: single and double. Single precision consists of a sign bit, 8-bit exponent and 23-bit mantissa, double precision consists of a sign bit, 11-bit exponent and 52-bit mantissa. In our tests in Section 5, we only examine behaviour in IEEE-754 single precision arithmetic, which is often used in FPGA and GPU accelerators.

2.3 Hardware Test Suites

Aside from exhaustive testing across every possible input value, the only way to confirm that hardware performs exactly as specified is to perform some sort of formal verification of the hardware implementation [12–14]. However, such a process typically requires a lot of time and effort, which is unacceptable, especially when new algorithms that save time and area are being developed all the time. As such a more tractable approach is to create a test suite that identifies corner cases that are most likely to cause errors in the output, and test the hardware against these cases, alongside further random tests, to ensure it works as desired. In terms of floating point verification, the TestFloat [15], the IeeeCC754 test suite [16] and Paranoia [17] are three such test suites. In this project, we have chosen to focus on the latter, for it is the most well-known, as can be seen by its various translations – from its original BASIC program [17] into various languages, including Pascal [18] and C [19]. The popularity of Paranoia has ensured it is still used on modern CPU hardware to show flaws in many floating point arithmetic implementations [20], as well as to benchmark GPU floating

point arithmetic [1], and for this reason, we wish to create the same level of scrutiny towards FPGA IP-cores.

2.4 Existing FPGA IP-cores

FPGA manufacturers and developers have developed their own hardware designs that perform individual floating point operations. In this paper, we test the latest cores from the two major FPGA manufacturers Altera [21] and Xilinx [22], as well as the open source platform FloPoCo [23]. These cores typically offer a myriad of customisations exploiting the freedom of an FPGA to create specialised hardware, for example, the ability to tune variable widths of exponent and mantissa to obtain a superior hardware design to meet a designers specification, but also offer IEEE standard single and double precision cores. The specifications to which these cores adhere are detailed as follows:

- Altera Floating Point Megafunctions v11.0 (part of Quartus II v11.0 service pack 1) [21] only support round-to-nearest-even rounding mode, the default of IEEE-754-1985, and do not support subnormal numbers, flushing them to zero. However, there is support for exception signals for underflow and overflow.
- Xilinx DS816 Floating Point Operator v6.0 (part of Xilinx ISE v13.3) [22] supports round-to-nearest-even rounding mode, and both handshaking and exception signals for underflow or overflow are implemented.
- FloPoCo version 2.2.1 [23] has a special floating point format, with an additional two-bit prefix. The two bits are used only to signal exceptions, namely 00 for zero, 01 for normal numbers, 10 for infinities, and 11 for NaN. These differ from IEEE exception signals where zero (exponent = 00...00) or Infinity or NaN (exponent = 11...11), meaning FloPoCo cores can use these values to represent additional normal floating point numbers. FloPoCo also does not aim to support subnormal numbers.

3 Test Framework

The general framework for our approach is illustrated in Figure 2, using the example of Altera floating point units. In this framework, we create preprocessor and post-processor blocks and instantiate all the individual floating point operator cores. The preprocessor and post-processor blocks pass the input data and receive the output data from the desired core respectively, and additionally translate traditional floating point numbers in the testbench into the format desired by the floating point cores. For example, in the case of FloPoCo cores, we must translate the exception signals to and from their IEEE equivalents. This allows the testbench to be independent of all details of the hardware implementation.

While our framework is general enough to apply any tests to any hardware floating point cores, in our analysis, we focus only on Altera, Xilinx, and FloPoCo cores using the Paranoia benchmark. In the rest of this section, we detail the choices we have made to perform these tests.

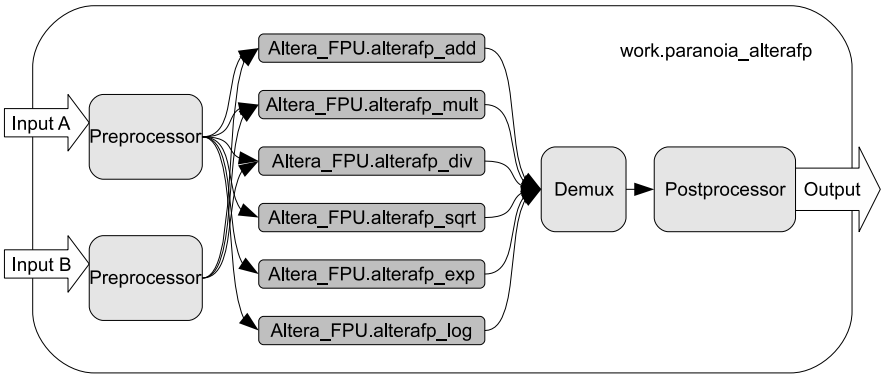

Fig. 2. Paranoia test framework as implemented with Altera IP-cores.

3.1 Adapting Paranoia to FPGA core generators

Hardware Operators In our implementation of Paranoia, as mentioned above, we instantiated individual floating point cores for operations whenever possible. We note however that using this methodology, we cannot test any more complex cores, such as the 3-Input Adder, Accumulator, or Multiply-Accumulator of the FloPoCo block. Instead, any complex expressions were handled by first breaking the operations into a two-input static single assignment form.

Software functions While we choose to use dedicated hardware operators whenever possible, the core generators are not capable of implementing every type of floating point operation required in Paranoia, so we implemented some operations, such as floor(x), $\log_2(x)$ and random(x), in software. Similarly, while no hardware core generator provides a specific exponentiation unit x^y, this function could be re-written as shown in (3), so we created a function that reads what is available from the hardware core generator and decides whether to implement it by multiplying a value initialised to 1 by the value x y times, or to use exponential and logarithm hardware blocks to approximate x^y; we note that if y is non-integer, it is only possible to apply the latter approach. Furthermore, given that neither approach would work in the case where $y = 0$, where the result must always be 1, this special case must be added to prevent the blocks failing the desired test.

$$z = x^y = e^{y \cdot \ln x} \tag{3}$$

As well as specific floating point operations, we also used software within our pre and post-processor blocks to simplify some basic operations. For example, while Altera and Xilinx provide a block to negate or find the absolute value of a float, this can be achieved cheaply by manipulating the sign bit, so we implement this directly. Furthermore, in the case of FloPoCo, two additional

software changes had to be added in the pre and post-processor blocks. Firstly, because it does not have a subtract core, we had to first negate one of its inputs, and secondly it does not implement comparison, so for simplicity and speed we chose to uses the Altera compare core to perform comparisons for the results of operations from FloPoCo cores.

4 Paranoia on an FPGA

While the previous section described the overall framework to which one could apply any software testbench, in this section, we describe the set of tests taken from the paranoia benchmark, as well as any modifications that we made to these tests to make it suitable to test the desired hardware.

4.1 Basic Arithmetic

Listing 1.1 demonstrates the initial few lines of Paranoia, first defining the variables Zero as 0 and One as 1, from which most other floating point values are obtained through arithmetic operations on these two variables, and no other literals are used; we have recreated this style in our tests. Paranoia then runs several tests on basic arithmetic, for example, $3 \stackrel{?}{=} 2 + 1$. Particularly important is the zero comparison test, which tests if -0.0 is equal to 0.0, because if they are unequal, several later tests cannot be run. While FloPoCo has no compare core, as we mentioned in Section 3.1, meaning that it could not alone verify the difference between $+0$ and -0, we did add tests to ensure it created both positive and negative zero correctly. Eventually, more complex arithmetic tests are run.

```
Zero = 0;
One = 1;
Two = One + One;
TwoForty = Four * Five * Three * Four;
MinusOne = -One;
Half = One / Two;
```

Listing 1.1. paranoia.c(402-404,412-414) Initial tests.

One set of arithmetic tests which we have removed are those that test for extra-precise sub-expressions, which are typically additional bits stored for intermediate results, because we have assumed throughout that the result is rounded before passing to the next; the subject of creating application specific fused data-paths on FPGAs that require additional internal bits to achieve equivalent results is a separate field [24, 25]. However, we do include Paranoia's tests that search for the presence of a guard, round and sticky bit in succession, to check that rounding is correct under several corner cases.

After tests for addition and subtraction, Paranoia tests for correct multiplication by evaluating if $X*Y \stackrel{?}{=} Y*X$ for several random values. Division is tested for several extreme values as described in the Paranoia benchmark, this

includes values such as $\frac{1}{0}$ and $\frac{0}{0}$. We note that the latter tests should return exception signals which we aim to detect ($\pm\infty$ and/or NaN), as described in Section 2.2. Finally $\sqrt{x}$ is also tested for several corner case values as in the traditional version of Paranoia.

4.2 Exponentiation

Exponentiation (x^y) is tested using extreme values of x and y, as well as the value 0^0, which by convention should return 1 exactly, to enable compact representation of various series and sequences, especially polynomial or power series. This is followed by a special test to throw up inaccuracies in the exponentiation function for non-integer values. In this test, the value e^2 is first computed in the reference precision by an iterative refinement process, then this value is compared against the evaluation of $x^{\frac{x+1}{x-1}}$ for several values close to 1, noting that these values should be almost equivalent given the result in (4). While these tests are performed as described in Paranoia, because none of the core generators create an exponentiation core, as we mentioned in Section 3.1, we replace this using the various alternatives; this enables us to perform additional multiplication tests, as well as implementing the desired function.

$$\lim_{x \to 1} x^{\frac{x+1}{x-1}} = e^2 \tag{4}$$

4.3 Underflow and Overflow

Our tests for underflow and overflow are restricted to those which check that underflow and overflow are correctly flagged, as opposed to any handling of subnormals, because as mentioned in Section 2.4, subnormals are not supported by any of the core generators.

5 Results

We ran our hardware version of Paranoia on single precision (8-bit exponent, 23-bit mantissa) Altera v11.0, FloPoCo v2.2.1, and Xilinx v6.0 cores. Results were then compared with the results from a Intel Core i7 running single precision Paranoia compiled with gcc 4.4.3 (no errors). These tests revealed many differences between the FPGA floating point IP-cores and the Core i7 FPU, as summarised in Table 2. In this section, we highlight and discuss the differences detected between the FPGA IP-cores and the general purpose processor.

5.1 Basic Arithmetic

The Paranoia tests for basic arithmetic were in general satisfied, there were some differences from the IEEE 754 standard in the corner cases for division.

Table 2. Success of FPGA IP-cores against Paranoia benchmark.

Test Name	Altera v11.0	Flopoco v2.2.1	Xilinx v6.0
Basic Arithmetic			
Basic Arithmetic	☑	☑	☑
Division by Zero	☑	☒	☑
Add/Sub Rounding	☑	☑	☑
Multiplication Rounding	☑	☑	☑
Division Rounding	☒	☑	☑
Guard Digits	☑	☑	☑
Sticky Bit	☒	☑	☑
Sqrt Rounding	☑	☑	☑
Exponentiation			
x^y where $x, y \in \mathbb{Z}$	☑	☑	☑
$\lim_{x\to 1} x^{\frac{x+1}{x-1}} = e^2$	☒	☑	NI
Underflow and Overflow			
Thresholds	☑	☑	☑
PseudoZero	☑	☒	☑
X!=Z but X-Z=0	☑	☒	☑
Gradual Underflow	☒	☒	☒

☑	Passed this test
☒	Did not pass this test
NI	Not implemented in hardware

FloPoCo v2.2.1 division by zero The IEEE 754 standard states that any value divided by zero should result in the exception signal $\pm\infty$ with the sign equal to the sign of the numerator, except in the case of $0 \div 0$, where the result should be signalled as NAN. Using the Paranoia benchmark, while the Xilinx and Altera blocks worked as expected, the FloPoCo block stated that $0 \div 0 = \infty$. While this could be argued to be a technicality, this incorrect exception signal could result in errors during future computations, for example when computing $e^{-(0\div 0)}$.

Altera v11.0's division core rounding Altera's documentation states that the blocks adhere to round-to-nearest (even). However, the example of Table 3 shows that Altera calculates division differently to other cores. By performing the operation in double precision, one can confirm that it is the Altera core that rounds incorrectly. This also causes the Altera v11.0 cores to fail the Sticky Bit test, as seen in Table 2.

Table 3. Results after rounding for Paranoia's division test with various division cores.

	Altera	FloPoCo	Xilinx	Intel
$\left(1.5 - 2^{-23}\right) \div \left(1 - 2^{-23}\right)$	1.5	1.5000001	1.5000001	1.5000001
3FBFFFFF÷3F7FFFFE	3FC00000	3FC00001	3FC00001	3FC00001
$\left(1.5 + 2^{-23}\right) \div \left(1 + 2^{-23}\right)$	1.4999999	1.5	1.5	1.5
3FC00001÷3F800001	3FBFFFFF	3FC00000	3FC00000	3FC00000

5.2 Exponentiation

As we mentioned in Section 3.1, none of the core generators create a hardware block to implement exponentiation, and as a result, we implemented this function using two methods. In the case where a value is raised to a positive integer power, this could be approximated using repeated multiplication, and similar to the previous tests for multiplication, this performed correctly for all cases, provided the special case for $0^0 = 1$ is added.

Exponentiation with fractional exponents in Altera v11.0's logarithm block The previous test revealed no shortcomings with the logarithm and exponentiation cores for integer arguments. However, since $\frac{x+1}{x-1}$ is, in general, non-integer, because Xilinx core generator v6.0 does not support $\ln(x)$ and e^x, only Altera v12.0 and FloPoCo v2.2.1 can perform this test, which we implemented using the $x^y = e^{y \cdot \ln x}$ identity. Interestingly, this revealed a flaw in the Altera v11.0 logarithm core while calculating ln(0.99999994), as shown in Table 4.

Table 4. Arithmetic errors in Altera $\log(x)$ cores.

	Altera	FloPoCo	IEEE Double
X	0.99999994 (0x3F7FFFFF)	0.99999994 (0x3F7FFFFF)	$1 - 2^{-24}$
ln(X)	-1.1641532e-10 (0xAF000000)	-5.9604645e-8 (0xB3800000)	-5.960464e-8

5.3 Underflow and Overflow

FloPoCo v2.2.1 and subnormal numbers None of the core generators state that they support subnormal numbers, so unsurprisingly they do not pass related tests, such as those for gradual underflow. While Xilinx v6.0 and Altera v11.0 blocks simply flushed subnormal numbers to zero, interestingly, FloPoCo v2.2.1 contained some inconsistent subnormal evaluation. Firstly, when continually halving the value 1 to find the smallest representable value, FloPoCo

returned 0x00000001. The fact that this number can be reached through repeated multiplications shows that subnormal numbers must be supported to some extent, because halving 0x00000001 would produce a number exactly between 0x00000001 and 0x00000000, although this implies the rounding mode is defaulted to round towards $+\infty$ as opposed to round-to-nearest even which requires the result to be even, i.e. 0x00000000. Furthermore, this number returns errors with division, for example (5) returns $+NaN$ instead of +2.0, and interestingly, this differs from its behaviour of division by zero described earlier, which always returns ∞.

$$Z = \text{0x00000001}, \quad \frac{Z+Z}{Z} \tag{5}$$

6 Conclusion

This paper has presented a flexible framework to test hardware cores to which we modified the well-known Paranoia testbench to detect any flaws that modern FPGA IP-cores exhibit with respect to the IEEE 754 standard. By applying these tests, we have managed to highlight some limitations in current floating point core generators. Many users will be unaware of the current differences between fully IEEE compliant hardware and the current FPGA IP-cores, and we hope that documenting these differences allows suitable care to be taken when designing circuits using these cores.

We appreciate that in a custom computing world, in many applications, it is almost always worthwhile relaxing IEEE compliant specifications in favour of smaller or faster hardware, but we argue that prospective users must be aware of any such issues. By ensuring our community makes the same rigorous efforts as their general purpose computer counterparts, it is likely to provide trust that FPGAs are safe to use to accelerate applications and hopefully to gain more prospective users of FPGAs, and as such, the current framework will be made freely available to download from:

http://cas.ee.ic.ac.uk/people/dpb03/

References

1. K. Hillesland and A. Lastra. (2004) GPU Floating-Point Paranoia. [Online]. Available: http://www.cs.unc.edu/∼ibr/projects/paranoia/gpu_paranoia.pdf
2. W. Kahan and C. Severance. (1998) An interview with the old man of floating-point. [Online]. Available: http://www.eng.auburn.edu/∼agrawvd/COURSE/E6200_Fall07/READ/Kahan_ Interview.pdf
3. IEEE, *IEEE Standard for Binary Floating-Point Arithmetic*, Std., 1985.
4. A. Jones, "Supercomputing's future: Is it CPU or GPU?" *ZDNet UK / News and Analysis / Business of IT / IT Strategy*, 2010, http://www.zdnet.co.uk/news/it-strategy/2010/06/16/supercomputings-future-is-it-cpu-or-gpu-40089202/.
5. S. Gupta, "China's Investment In GPU Supercomputing Begins to Pay Off Big Time," *NVIDIA Blog*, 2011, http://blogs.nvidia.com/2011/06/chinas-investment-in-gpu-supercomputing-begins-to-pay-off-big-time/.

6. Intel, "Intel and floating point," *Cygnus Software*, 2006. [Online]. Available: http://www.intel.com/standards/floatingpoint.pdf
7. W. Kahan, "Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic," University of California, Berkeley, Tech. Rep., 1997, http://www.cs.berkeley.edu/∼wkahan/ieee754status/IEEE754.PDF.
8. K. Underwood, "FPGAs vs. CPUs: trends in peak floating-point performance," in *Proc. Int. Symp. on Field Programmable Gate Arrays*, 2004, pp. 171–180.
9. IEEE, *IEEE Standard for Floating-Point Arithmetic*, IEEE Std. 754, 2008.
10. D. Goldberg, "What every computer scientist should know about floating-point arithmetic," in *ACM Computing Surveys*, 1991, http://download.oracle.com/docs/cd/E19957-01/806-3568/ncg_goldberg.html.
11. W. Kahan. (2005) A brief tutorial on gradual underflow. [Online]. Available: http://www.cs.berkeley.edu/∼wkahan/ARITH_17U.pdf
12. S. Beyer, "Putting it all together - formal verification of the VAMP," *International journal on software tools for technology transfer*, vol. 8, no. 4, pp. 411–30, 2006.
13. C. Jacobi, "Formal verification of the VAMP floating point unit," *Formal Methods in System Design*, vol. 26, no. 3, pp. 227–66, 2005.
14. N. Kikkeri and P. M. Seidel, "An FPGA Implementation of a Fully Verified Double Precision IEEE Floating-Point Adder," in *IEEE International Conference on Application-specific Systems, Architectures and Processors*, 2007, pp. 83–88.
15. J. Hauser. (2010) Testfloat. [Online]. Available: http://www.jhauser.us/arithmetic/TestFloat.html
16. B. Verdonk, A. Cuyt, and D. Verschaeren, "A precision- and range-independent tool for testing floating-point arithmetic II: conversions," *ACM Trans. Math. Softw.*, vol. 27, no. 1, pp. 119–140, 2001.
17. W. Kahan. (1983) Paranoia in BASIC. [Online]. Available: http://netlib.org/paranoia/paranoia.b
18. B. A. Wichmann. (1985) Paranoia in Pascal. [Online]. Available: http://netlib.org/paranoia/paranoia.p
19. T. Sumner and D. Gay. (1986) Paranoia in C. [Online]. Available: http://netlib.org/paranoia/paranoia.c
20. R. Karpinski, "Paranoia - a floating point benchmark," *Byte Magazine*, vol. 10, no. 2, pp. 223–235, 1985.
21. Altera, *Floating-Point Megafunctions User Guide*, 2011, http://www.altera.com/literature/ug/ug_altfp_mfug.pdf.
22. Xilinx, *LogiCore IP Floating-Point Operator v6.0*, 2011, http://www.xilinx.com/support/documentation/ip_documentation/floating_point/v6_0/ds816_floating_point.pdf.
23. F. de Dinechin. (2010) FloPoCo, a generator of arithmetic cores for FPGAs. [Online]. Available: http://flopoco.gforge.inria.fr/
24. F. de Dinechin, B. Pasca, O. Cret, and R. Tudoran, "An FPGA-specific approach to floating-point accumulation and sum-of-products," in *Proc. Int. Conf. on Field-Programmable Technology*, 2008, pp. 33 –40.
25. A. Roldao Lopes and G. Constantinides, "A Fused Hybrid Floating-Point and Fixed-Point Dot-Product for FPGAs," in *Proc. Int. Symp. onReconfigurable Computing: Architectures, Tools and Applications*, 2010, vol. 5992, ch. 16, pp. 157 – 168.