FORMAL VERIFICATION OF FLOATING-POINT HARDWARE WITH ASSERTION-BASED VIP

Ravi Ram[1], Adam Elkins[1], Adnan Pratama[1] – Xilinx Inc.
Sasa Stamenkovic[2], Sven Beyer[3], Sergio Marchese[3] – OneSpin Solutions

[1]Xilinx Inc., ravi.ram@xilinx.com
2100 Logic Drive, San Jose, CA 95124-3400, USA, +14088792763

[2]OneSpin Solutions, sasa.stamenkovic@onespin.com
4820 Hardwood Road, #250, San Jose, CA, 95124, USA, +14087341900

[3]OneSpin Solutions, {firstname.lastname}@onespin.com
Nymphenburger Strasse, 20a, 80335, Munich, Germany, +4989990130

***Abstract*-Hardware for integer or fixed-point arithmetic is relatively simple to design, at least at the register-transfer level. If the range of values and precision that can be represented with these formats is not sufficient for the target application, floating-point hardware might be required. Unfortunately, floating-point units are complex to design, and notoriously challenging to verify. Since the famous 1994 Intel Pentium bug, many companies have investigated and successfully applied formal methods to this task. However, solutions often rely on a mix of the following: hard-to-use formal tools; highly specialized engineering skills; availability of a suitable executable model of the hardware; and significant, design-specific engineering effort. In this paper, we present an alternative floating-point hardware verification approach based on a reusable, IEEE 754 compliant SystemVerilog arithmetic library. While not addressing all verification challenges, this method enables engineers to set up a formal testbench and uncover deep corner-case bugs with minimal effort. Results from industrial applications are reported.**

I. INTRODUCTION

Floating-point (FP) representations of real numbers have significant advantages over fixed-point, particularly as, given a certain bit-width for their binary encoding, they may cover a much wider range of values without losing precision. FP arithmetic hardware units (FPUs) are crucial to support efficient execution of software programs in a variety of applications. The IEEE 754 standard [1] provides an unambiguous, established specification of how FP numbers can be represented in binary format (see Fig. 1), and how basic arithmetic operations should work on those representations. IEEE 754 compliant FPUs are often integrated in CPUs and DSPs designs.

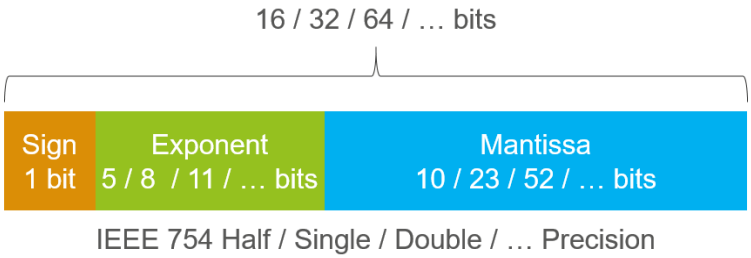

Figure 1. Binary representation of floating-point number

The functional verification of FPUs is a critical task and notoriously difficult. Corner-case bugs may cause a variety of issues, from completely wrong results when handling non-ordinary values, as +0, -0 or NaNs (not a number), to minor rounding errors that could still cause major system malfunctions when accumulated over many iterations.

A. Related work

Since the famous 1994 Intel Pentium FP division bug, which had an estimated cost of $475M [3], a number of academic institutions and semiconductor heavyweights, including Intel, AMD and IBM, have carried out a considerable amount of research in the application of formal methods to FP hardware verification [4,6,7]. Although these methods have been successful in the verification of complex industrial designs, they suffer from drawbacks that have hindered widespread adoption.

Methods relying on the availability of non-commercial proprietary tools are obviously not an option for companies with no easy access to those tools. Methods relying on complex tools, requiring highly specialized skills, such as theorem provers, may not be a viable option in projects with tight budget and timing constraints. Methods based on combinational and sequential equivalence checking between high-level models and register-transfer level (RTL) implementation, leveraging commercially available formal tools, have been used successfully to address some of the challenges of FPU verification. However, they also present drawbacks. The user needs to have access to a suitable reference model, at the appropriate level of abstraction, typically written in SystemC or C++ [2,5]. Although some teams may have access to trusted architectural models, changes are often required, for example to clock and reset in order to solve synchronization issues, and to non-synthesizable code. Moreover, engineers must have intimate knowledge of both the high-level and RTL hardware models in order to partition the overall comparison problem, and set up intermediate verification steps. Finally, work products are largely not reusable across different FPU implementations and may require significant porting effort, from days for simple operations as addition, to months for more challenging ones as multiplication.

B. Contribution

This paper presents an alternative FPU formal verification approach leveraging a SystemVerilog assertion-based verification intellectual property (ABVIP). With this solution, engineers do not need to provide a reference model, and can set up assertions for FP operations with minimal effort and design knowledge. Furthermore, intended deviations from the IEEE 754 standard can be accommodated using high-level functions, making them easy to isolate and review. This method covers most FP operations, including multiplication. However, it does not feature out-of-the-box support for operations that are iterative in nature, such as division and square root.

II. IEEE 754 FORMAL ABVIP

The core component of the method presented in the paper is the formal FP ABVIP developed by OneSpin Solutions. The ABVIP is coded in standard SystemVerilog. In a nutshell, it consists of a set of packages defining data types encoding IEEE 754 FP operands, exceptions flags, rounding modes, and functions capturing expected results of FP operations.

A. Features

The FP ABVIP supports half, single and double-precision FP formats. Other custom precision formats can be supported with little effort. High-level record data types encapsulate sign, mantissa and exponent bit slices. All four rounding modes are supported (roundTowardZero, roundTiesToEven, roundTowardPositive and roundTowardNegative), as well as all five exceptions flags (invalid operation, division by zero, inexact result, underflow and overflow). As per the standard, the underflow flag, also called tininess, can be computed either before or after rounding. Checking of exception flags can be selectively switched off. Moreover, the package contains a set of query functions to isolate and accurately specify intended deviation from the standard, e.g., for the handling of denormal numbers.

The operations supported out of the box are addition, subtraction, multiplication, absolute value, negation, and all comparison operations (less, equal, greater and unordered).

The FP ABVIP also includes type conversion functions between FP numbers of different precision, as well as between FP numbers and integers of the same bit width. All the ABVIP functions are prefixed with the string *ieee*.

B. Quality assurance

An engineering team with extensive expertise in formal verification, computer arithmetic, and the IEEE 754 standard has developed the ABVIP SystemVerilog code. The team has verified the model against three OpenCores FPUs, namely: double_fpu, accessible at https://opencores.org/project,double_fpu; fpu100, accessible at https://opencores.org/project,fpu100; and fpu, accessible at https://opencores.org/project,fpu. Additionally, the ABVIP has been verified against the VAMP FPU [8], accessible at http://www-wjp.cs.uni-saarland.de/forschung/projekte/VAMP/downloads.php. There is also ongoing effort to verify the ABVIP against the SoftFloat C library [9], which is considered an unofficial reference implementation of IEEE 754. Finally, deployment in industrial projects provides further confidence on the correctness of the ABVIP.

Overall, the authors have high confidence that the FP ABVIP correctly implements the IEEE 754 standard. Moreover, thanks to the exhaustive nature of formal verification, a bug could go unnoticed only if both the ABVIP and the design under test (DUT) would have exactly the same error. Considering that hardware FPUs are coded by a completely different team, from a different company, and using different means and goals, this risk is low.

C. User interface

Fig. 2 shows a high-level view of the FPU formal verification flow using ABVIP. The FPU design and the ABVIP are read in the formal tool. The user is responsible to configure the ABVIP, for example selecting the precision. End-to-end assertions are set up with minimal effort. The formal tool provides a counter-example trace if an assertion fails, or a proof, either unbounded or bounded. A bounded proof requires additional effort. Users may need to tune formal tool options, introduce abstractions, or adopt a divide-and-conquer approach to turn a bounded proof into an unbounded one, and thus achieve full confidence in the correctness of the design. However, this is not necessary for initial formal bug hunting, which by itself provide a significant boost to design quality and high return on investment (ROI).

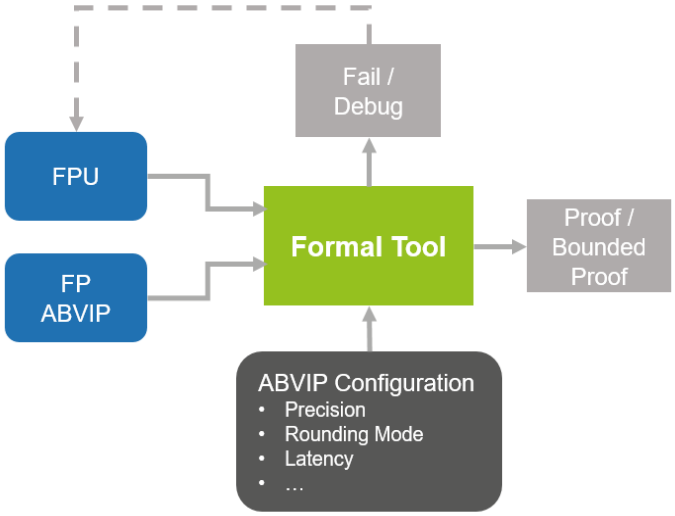

Figure 2. Formal FPU verification with ABVIP

```
property fp_multiplication_p;
   ieee_with_flags_t expected;
   disable iff (~reset_n)
     (<trigger for multiplication operation>, expected = ieee_mul(.op_a(op_a), .op_b(op_b), .round_mode(round_mode)))
   |->
     ##<latency>
     result_valid && ieee_check_result(
                                       .expected(expected),
                                       .actual(actual),
                                       .supported_flags(supported_flags));
endproperty

fp_multiplication_a : assert property (fp_multiplication_p);
```

Figure 3. SystemVerilog template assertion for the IEEE-754 FP multiplication operation

Fig. 3 shows a SystemVerilog template property, and associated assertion, that users need to adapt to the specific DUT to verify the multiplication operation. Function *ieee_mul* computes the expected results, including exception flags, of the multiplication given the input operands and the rounding mode. Function *ieee_check_resul*t compares the expected result with the result computed by the FPU. Check for unsupported exception flags can be disabled. It is worth noting that this function is more intelligent than a straightforward equality check, as there are special cases where the standard does not precisely stipulate the expected result. In such cases, a simple comparison could yield spurious failures. The same template can be adapted to other operations simply by replacing function *ieee_mul* with, for example *ieee_add*, and adjusting the latency.

The user must provide the trigger condition for the operation, which will depend on the FPU input interface. The user also needs to provide hooks to the FPU signals that capture the actual result, the current rounding mode, as well as the input operands and the exception flags. Finally, *latency* is a positive integer that expresses the number of cycles that the FPU needs to compute the result.

Pipeline-based implementations that do not have a fixed latency or that can complete out-of-order, can also be accommodated but will require additional effort. It is worth noting, though, that in many cases it might be advisable to decouple the verification of the pipeline control functionality from the FP algorithm, regardless of the particular tool or solution adopted.

III. FP ABVIP Integration with Formal Tools

OneSpin's IEEE 754 FP ABVIP is coded in standard SystemVerilog and is therefore readable by electronic design automation (EDA) tools supporting this language. Specific tools can provide additional features, particularly to improve debug and proof runtimes. Visualization of FP data types helps the analysis of signal traces in the waveform viewer, for example when debugging a failing assertion. Features for automatic and user-defined selection of proof engines and strategies optimized for arithmetic operations are crucial to achieve convergent results, or quickly uncover corner-case bugs.

In the following sections, we present results achieved using OneSpin's 360 DV-Verify™ FPU App. This app includes the FP ABVIP, as well as debug and proof optimization features. Fig. 4 shows a high-level view of the FPU formal verification flow using OneSpin's FPU App.

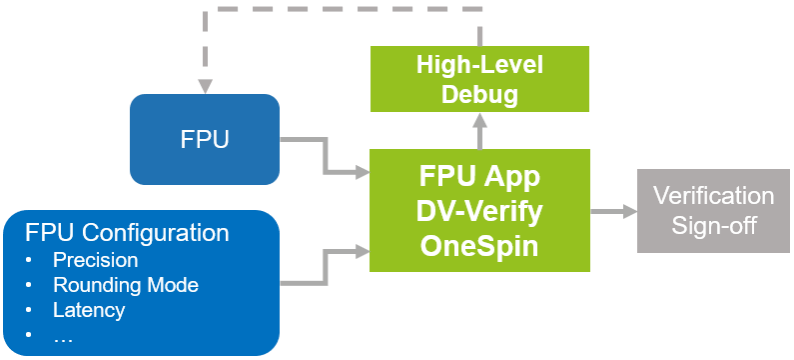

Figure 4. Formal FPU verification with OneSpin's APV App

IV. RESULTS: OPENCORES FPU

In this section, we report on experiences in applying OneSpin's FPU App to verify the addition, subtraction and multiplication operations of an OpenCores FPU. The source code and documentation for this design can be accessed at https://opencores.org/project,fpu100.

The DUT supports single-precision operands, all exception flags and all rounding modes. It features a bit-level optimized implementation of the multiplication operation, making its formal verification a computationally demanding task. The design features a number of separate execution units. Addition and subtraction have a latency of 7 clock cycles, while multiplication has a latency of 12 clock cycles. The I/O interface is straightforward, with control signals marking the start and end of an operation, the operation type and rounding mode.

The developer reports that it has verified the FPU with simulation using two million test vectors created with SoftFloat. Moreover, the FPU has also been successfully implemented on a Cyclone I FPGA device.

Table I summarizes the results obtained, and includes information on tool runtimes, effort spent in configuring the FP ABVIP, and final proof result.

Table I. OpenCores FPU formal verification results

Operation	Bugs Found	Setup Effort	Runtime	Proof Result
FADD	0	30 minutes	52 seconds	Unbounded Proof
FSUB	1	5 minutes	60 seconds (to detect bug)	Fail
FMUL	2	5 minutes	1 second (to detect first bug)	Fail

A. Addition

The addition assertion is easy to configure, requiring only a minor adaptation to the template shown in Fig. 3. The DUT expects the input operands, operation selector and rounding mode signals to be stable while the operation is being executed. It is the user's responsibility to capture these requirements. The assertion is proven (unbounded) in less than a minute.

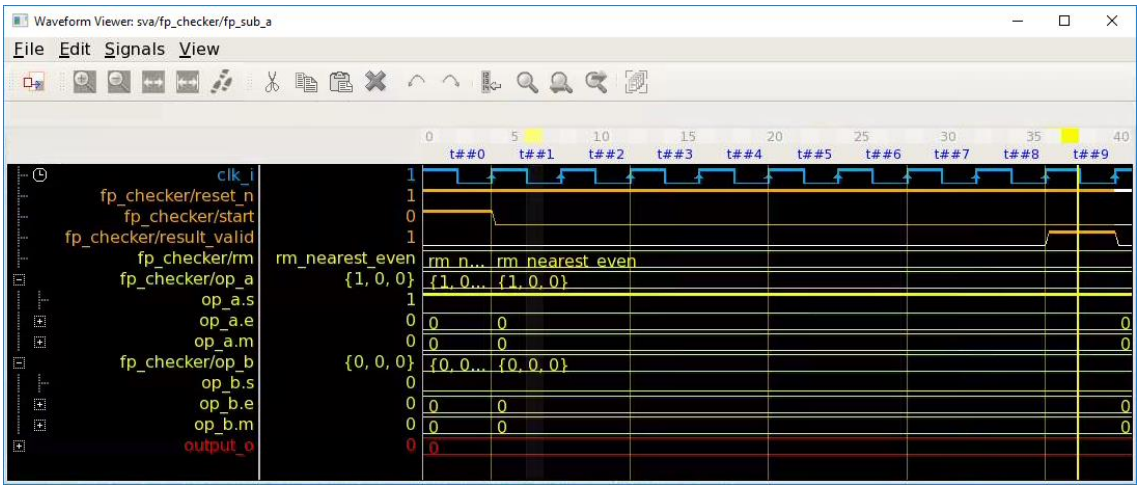

Figure 5. Counter-example trace exposing FPU subtraction bug

B. Subtraction

The subtraction assertion is trivial to set up once the addition has been proven, as the extra input constraints are the same. Within a minute, the tool finds a corner-case scenario that uncovers a DUT bug regarding the sign of zero. The addition unit correctly computes:

$$-0 + (-0) = -0$$

This corresponds to the standard's statement [1]: "However, x + x = x − (−x) retains the same sign as x even when x is zero". The subtraction unit, however, wrongly computes +0 for the corresponding case:

$$-0 - (+0) = +0 \text{ (correct result is } -0)$$

Input operands values 0x80000000 (op_a = -0) and 0x00000000 (op_b = +0), with rounding mode roundTiesToEven are shown in the counter example trace computed by the tool (see Fig. 5). As the design has not been fixed, it is not possible to prove the assertion and measure the runtime required for proof convergence. Note that other bugs could still be present in the design.

C. Multiplication

Similarly to the subtraction, the multiplication assertion is also trivial to set up. Within a second, the tool finds a corner-case scenario exposing a DUT bug. This was a known bug already reported in 2014 and tracked in the OpenCores website. For further information, please refer to https://opencores.org/bug,view,2455.

With the help of input constraints, the previous known bug was worked around, and verification continued. The assertion failed again, exposing an additional, unknown bug. Multiplying a NaN with a number delivers a NaN (which is expected), but it also raises both overflow and inexact flags which is incorrect. As the design has not been fixed, it is not possible to prove the assertion and measure the runtime required for proof convergence. Note that other bugs could still be present in the design.

V. RESULTS: XILINX FPU

In this section, we report on experiences in applying OneSpin's formal FPU App to verify the addition, subtraction and multiplication operations of a Xilinx FPU. Moreover, coverage results are also briefly discussed. It is worth noting that the IP has also been verified with simulation and emulation and these efforts had started before formal verification was introduced.

The Xilinx FPU has a configurable number of pipeline stages. Operations are mostly compliant with IEEE 754, with only a few minor intended deviations.

A. Setup

The first step was to set up the IP to operate in FP mode. Separate setups and SystemVerilog constraints, connected to the IP using the *bind* statement, were developed for each supported precision mode. Once the first setup was successful, others required minimal effort.

The FP ABVIP instantiation template had to be adapted to map its signals, including input operands, rounding mode, result and exception flags, with the corresponding DUT signals.

During the bring-up phase, it was useful to be able to disable the check of some exception flags. Initial debugging of assertion failures in OneSpin revealed a bug in the specification of the setup constraints. The results reported in Table II refer to the IP configuration with no pipeline stages. Moreover, effort to familiarize with the general tool interface and the FPU App is included in the figures in contrast to the corresponding efforts in the OpenCores Table I.

B. Assertions

In addition to the assertions verifying generic FP operations, dedicated assertions covering specific scenario were developed, including:

- Operations with quiet NaN (where NaN stands for Not a Number)
- Operations with signaling NaN
- Addition of infinities with opposite sign
- Multiplication of zero with infinity

These assertions provided added confidence into the correctness of the design, and a straightforward mapping to specific items of the verification plan.

The intended deviations from the IEEE 754 standard were expressed leveraging ABVIP functions and using two approaches. The example below shows the manipulation of denormal input operands in auxiliary verification code:

```
op_a  <= ieee_isDenorm(dut_op_a) ? {dut_op_a[31], 31'h0} : dut_op_a;
op_b  <= ieee_isDenorm(dut_op_b) ? {dut_op_b[31], 31'h0} : dut_op_b;
```

The second approach consisted in adapting the assertion. In the example below, the check of the underflow exception flag is disabled when the result of the addition operation (*expected.number*) is denormal.

```
property fp_addition_p;
ieee_with_flags_t expected;
(start_i, expected = ieee_add(.op_a(op_a), .op_b(op_b), .rm(rm)))  |->
      result_valid &&
      ieee_check_result(
            .expected(denormalToZero(expected)),
            .actual(actual),
            .supported_flags(ieee_isDenorm(expected.number)?
                  '{UNF:1'b0, default:1'b1}: '{default: 1'b1});
endproperty
```

The verification of the multiplication operations made use of the type conversion functions provided by the FP ABVIP, as shown in the example below:

```
op_b_32 <= ieee_754_conversion::convert_half_to_single(op_b_16);
```

A summary of the verification effort and runtime is reported in Table II. Note that the runtime for the multiplication refers to a converging, unbounded formal proof, not the runtime to find the corner-case bug. Moreover, the formal verification environment was run on RTL versions precedent to the introduction of formal verification, which

contained known bugs that had been found in simulation and emulation. The formal environment found these bugs within seconds. Considering the ease of setup and use of the FP ABVIP and the quality of results, the development team believes that introducing the OneSpin FPU App earlier in the project would have saved significant effort.

Table II. Xilinx FPU formal verification results

Operation	Bugs Found	Setup Effort	Runtime	Proof Result
FADD	0	4 days	3 minutes	Unbounded Proof
FSUB	0	3 days	1 minute	Unbounded Proof
FMUL	1	15 days	4 minutes	Unbounded Proof

C. Coverage

A number of cover properties were implemented to ensure that specific scenarios could be reached. The formal tool can automatically generate input stimuli, including FP operands, triggering the specified scenarios. Additionally, cover properties provide confidence that the formal environment is not over constrained, and enable early design exploration as interesting traces can be generated before a simulation testbench is available. A selection of simple cover properties that were developed, including some using functions from the FP ABVIP, are reported below:

```
svmovrfl : cover property (OVF_FLAG_o);

svmundrfl : cover property (UNF_FLAG_o);

svm_zerop_zerop : cover property (
  start_i && ieee_isZero(op_a) && ieee_isZero(op_b) && !op_a.s && !op_b.s);
svm_nump_nump : cover property (
  start_i && ieee_isNumber(op_a) && ieee_isNumber(op_b) && !op_a.s && !op_b.s);
svm_numn_numn : cover property (
  start_i && ieee_isNumber(op_a) && ieee_isNumber(op_b) && op_a.s && op_b.s);
```

OneSpin provides a metric-driven verification solution based on the Quantify Coverage App. Quantify was used to compute a number of metrics for statement and branch coverage targets, including *observed*, *reached*, *constrained* and *dead*. An RTL statement that is classified as *constrained*, for instance, cannot be covered because of constraints in the formal environment. It is particularly important to review these results, as over-constraining is a primary risk in formal verification. An RTL statement classified as *observed* on the other hand, ensure that if the statement were corrupted, at least one assertion would fail, thus exposing the introduced bug. Quantify can also compute coverage for bounded proofs. More information on Quantify can be found at http://www.onespin.com/solutions/metric-driven-verification.

Quantify coverage results were used to assess the quality of assertion and cover properties, highlight potential verification gaps, for example due to over constraining, and spot unused design logic. The overall coverage achieved was in excess of 90%. This result met expectations, as the DUT is not exclusively dedicated to FP operations.

VI. CONCLUSION

The functional verification of FPUs is a challenging task. Simulation-based methods are insufficient as they provide very limited coverage of input operands. Formal methods on the other hand are exhaustive in nature but may require hard-to-use tools, and significant effort from highly specialized engineers with intimate knowledge of formal tools, computer arithmetic, and the DUT.

In this paper, we present an alternative formal verification approach based on a SystemVerilog FP ABVIP compliant with the IEEE 754 standard. The FP ABVIP is easy to set up, and can accommodate intended deviations from the standard. Results of the application of the FP ABIP as part of the OneSpin FPU App in industrial applications show that corner-case bugs can be unveiled within seconds, and unbounded proof achieved within minutes, even for the multiplication operation. These results were obtained without the use of abstractions or assume-guarantee partitioning.

The main current limitation of this approach is that operations that are iterative in nature, such as division and square root, are not supported out of the box. Moreover, achieving unbounded proofs for operations such as FMUL or FMA (fused multiply-add) without partitioning is still challenging for formal tools, particularly in the case of double or extended precision operands and bit-level optimized implementations. That said, formal technology has improved dramatically over the last few years, and the results shown in this paper demonstrate that proof strategies targeted at arithmetic operations can deliver results that were not conceivable until recently.

REFERENCES

[1] ANSI/IEEE, IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-2008. IEEE.

[2] T.W. Pouraz, V. Agrawal, Efficient and Exhaustive Floating Point Verification Using Sequential Equivalence Checking, DVCon, USA, 2016.

[3] V. R. Pratt. Anatomy of the Pentium bug. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, Proceedings of the 5th International Joint Conference on the theory and practice of software development (TAPSOFT'95), Aarhus, Denmark, 1995. Springer-Verlag.

[4] J. Harrison, "Formal verification of floating-point arithmetic at Intel," 02 06 2006. [Online]. Available: http://www.cl.cam.ac.uk/~jrh13/slides/jnao-02jun06/slides.pdf. [Accessed 27 07 2017].

[5] M. A. Kirankumar, "Leveraging ESL Approach to Formally Verify Algorithmic Implementations", DVcon, India, 2015.

[6] U. Krautz, et al., "Automatic Verification of Floating Point Units", DAC, USA, 2014.

[7] C. Jacobi, "Formal Verification of a Fully IEEE Compliant Floating Point Unit", PhD Thesis, University of Saarlandes, April 2002.

[8] S. Beyer et al., "Putting it all together - Formal Verification of the VAMP", STTT Journal, Special Issue on Recent Advances in Hardware Verification, Springer, 2006.

[9] Berleley SoftFloat Library, www.jhauser.us/arithmetic/SoftFloat.html. [Accessed 26 10 2017].