Formal Verification of Universal Numbers using Theorem Proving

Adnan Rashid[1*], Ayesha Gauhar[1], Osman Hasan[1], Sa'ed Abed[2], Imtiaz Ahmad[2]

[1]School of Electrical Engineering and Computer Science (SEECS), National University of Sciences and Technology (NUST), Islamabad, Pakistan.

[2]Computer Engineering Department, College of Engineering and Petroleum, Kuwait University, Kuwait, Kuwait.

*Corresponding author(s). E-mail(s): adnan.rashid@seecs.nust.edu.pk;
Contributing authors: 14mseeagauhar@seecs.nust.edu.pk;
osman.hasan@seecs.nust.edu.pk; s.abed@ku.edu.pk;
imtiaz.ahmad@ku.edu.pk;

Abstract

Universal number (Unum) is a number representation format that can reduce the memory contention issues in multicore processors and parallel computing systems by optimizing the bit storage in the arithmetic operations. Given the safety-critical nature of applications of Unum format, there is a dire need to rigorously assess the correctness of Unum based arithmetic operations. Unums are of three types, namely, Unum-I, Unum-II and Unum-III (commonly known as Posits). In this paper, we provide a higher-order-logic formalization of Unum-III (posits). In particular, we formally model a posit format (binary encoding of a posit), which is comprised of the sign, exponent, regime and fraction bits, using the HOL Light theorem prover. In order to prove the correctness of a posit format, we formally verify various properties regarding conversions of a real number to a posit and a posit to a real number and the scaling factors of the regime, exponent and fraction bits of a posit using HOL Light.

Keywords: Universal Numbers, Posits, Theorem Proving, Higher-order Logic, HOL Light

1 Introduction

Floating-point number format is widely used by the scientific community in application areas ranging from the aerospace, applied mathematics, physics to weather forecasting, for the representation of real numbers on a computer. Moreover, it is utilized for the execution of various arithmetic operations, i.e., addition, subtraction, division and multiplication, requiring an efficient hardware implementation. The IEEE-754 floating-point standard represents a real number as a signed fraction times an integer power of 2, i.e., $\pm(1+f)2^e$, where f is a fraction and e is an exponent, and allows the representation of real numbers in computers using various bits. This includes the handling of the rounding and fraction bits, and various invalid results, such as Not-a-Number (NaN), which is returned as a result of an invalid arithmetic operation, such as $0/0$ or $\infty \times 0$. However, the IEEE-754 floating-point standard suffers from various limitations, such as limited numerical precision as a result of allocating a fixed number of exponent and mantissa bits, failure of the associative and distributive laws of real number arithmetic due to rounding and the hardware cost for handling the denormalized numbers.

1.1 Universal Numbers and their Applications

John L. Gustafson, in 2015, proposed Universal Numbers (Unums) [1] that can overcome the above-mentioned limitations of the IEEE-754 floating point standard and provide a more precise representation of real numbers for performing computer arithmetics. There are three types of Unums, namely, Unum-I, Unum-II and Unum-III (commonly known as Posits). Unum-I [1] has a variable-length format as opposed to the fixed length floating-point number format and also provides a better numerical accuracy. However, its variable-length format makes it unexciting for hardware implementations. Unum-II [1] exhibits some interesting characteristics, such as calculating the exact reciprocal of a number and performing negation of a number simply. However, it requires pre-computed lookup tables to perform various arithmetic operations that makes it impractical for larger arithmetic word sizes. Unum-III or Posits [2, 3] are considered as the hardware-friendly version of Unums that provide an efficient utilization of fixed bit sizes, resulting in higher accuracy arithmetic for a given storage requirement, and are intended to be a drop-in replacement for the IEEE-754 format. Posit exhibits various features, such as simple rounding, larger dynamic range, better closure, no denormalized numbers to handle, and therefore simplifies the hardware and software implementations. Moreover, posit arithmetic provides identical answers on different computer systems, which is not possible using the IEEE-754 floating point arithmetic standard. Posits do not overflow to infinity or underflow to zero. Moreover, NaN provides an action rather than a bit pattern as in floating-point numbers. Also, its processing unit takes less circuitry than the IEEE Floating-point Unit (FPU) [2]. All these features lead to an improved memory bandwidth and power efficiency. Moreover, posits have been implemented as an alternative to the floating-point number format in hardware and software. For example, the hardware architecture of Unum adder/subtractor and multiplier has been designed and implemented using Field-Programmable Gate-Arrays (FPGAs). Moreover, a Verilog Hardware Description Language (HDL) generator has been constructed for performing these arithmetic operations [4–6]. Software libraries for posit-based floating-point operations are also available for C# [7], C[1], C++[2] and Julia[3] programming languages. Moreover, posits have outperformed the fixed point number system, in terms of accuracy and memory utilization, in various computational intensive applications, such as deep convolutional neural networks [8, 9].

1.2 State-of-the-art

The real number programs are widely used for analyzing the dynamics of the physical systems in various applications, such as aerospace, robotics and physics. They use the floating-point approximations resulting in the accumulation of floating-point inaccuracies that grow as the computation proceeds and thus introduce some unavoidable bugs that may lead to dire consequences. For example, an error in the Floating-point Division (FDIV) instruction of the Intel Pentium processors in 1994 resulted into a financial loss of \$475M[4]. Similarly, an uncaught floating-point exception resulted in the destruction of the Ariane 5 rocket shortly after its takeoff in 1996[5]. The cost of such errors in floating-point arithmetic is huge. The above-mentioned popular incidents due to such errors resulted into replacement of a large number of processors having FDIV instruction errors, leading to a huge financial loss of \$475M and destruction of the Ariane 5 rocket. Therefore, one can expect that similar kind of bugs today may cost tenfold of that loss without performing an exhaustive analysis of arithmetic based on posits [10]. Moreover, the conventional computer based simulation and numerical analysis techniques involve the unverified symbolic algorithms, discretization and numerical errors, and thus cannot ascertain exhaustive analysis of the safety-critical systems. Therefore, the formal verification of these number formats, performing various arithmetic, is a dire need.

1.3 Formal Verification Methods and Theorem Proving

Formal verification method [11] is a system analysis technique that mainly involves two steps; 1) developing a computer based mathematical model of the given system, 2) verifying that the system's model meets the rigorous specifications of the intended behaviour, based on deductive reasoning. Since deductive reasoning involves the use of the logical reasoning and evidence to reach a conclusion from one or more premises that are considered to be true. Therefore, the usage of this method increases the chances of catching the errors that are often ignored by the conventional simulation based and numerical analysis techniques. The idea of doing formal verification of a complex system is to identify its safety-critical components/parts that require an exhaustive analysis. For example, in the case of Ariane 5 rocket, for the identification the uncaught floating-point exception, it is sufficient to perform the formal analysis of a component providing the floating-point arithmetic. Therefore, it may not require formal verification of the whole system. Theorem proving [12] is one of the frequently

[1] https://github.com/libcg/bfp
[2] https://github.com/eruffaldi/cppPosit
[3] https://github.com/milankl/SoftPosit.jl
[4] https://www.intel.com/content/www/us/en/history/history-1994-annual-report.html
[5] https://www-users.math.umn.edu/~arnold/disasters/ariane.html

used formal verification techniques that involves constructing a mathematical model of the given system based on logic and verifying its various properties by computer programs involving automated reasoning. Here, the automated reasoning refers to the computer-based deductive reasoning process that is based on the logical reasoning and evidence. Thus, it ensures the soundness of the theorem proving technique. Theorem proving can be interactive or automatic based on the choice of the underlying logic, which can be propositional, first and higher-order logic. Higher-order logic provides more expressiveness, which is important for analyzing the dynamics of physical systems. However, it requires user interaction for developing proofs within a theorem prover. Many theorem provers (automatic and interactive), such as HOL Light [13, 14], Coq [15, 16], ACL2 [17, 18] and PVS [19–21] have been used for the formal verification of the floating point numbers and their arithmetic. Moreover, there is a research group working on the verification of the different components/operations of the Intel processors, over the years. Some of the notable contributions are from Harrison [14, 22–26], O'Leary [27], Narasimhan and Kaivola [28, 29], Slobodova [30] and Peter Tang [25] who have been working in the Intel research group. Similarly, Rockwell Collins Inc. and NASA have been successfully using formal methods for analyzing various aspects of avionics [31–34]. However, none of these contributions cater for posit, which are intended as drop-in replacement for floating-point numbers in computer systems.

1.4 Contributions of the Paper

In this paper, we provide a formalization of posits (Type III Unums) using HOL Light. In particular, we formally model a posit format, which is composed of the sign, exponent, regime and fraction bits. Moreover, we formalize a conversion of a posit to its equivalent real number (decoding) and a real number to its equivalent posit representation (encoding), which mainly uses the notion of the fraction and exponential rounding. Finally, we formally verify various properties of the posits regarding these conversions and the scaling factors of the regime, exponential and fraction bits using HOL Light.

The novel contributions of the paper are:

- A higher-order logic formal model of a posit format, which includes the sign, exponent, regime and fraction bits, using the HOL Light theorem prover. Posit has not been formalized in any of the theorem prover before this paper.
- Higher-order logic formalization of the conversion of a posit to its equivalent real number and a real number to its equivalent posit.
- Formal verification of properties regarding conversions of a real number to a posit and a posit to a real number.
- Formal verification of properties regarding the scaling factors of the regime, exponential and fraction bits of a posit using HOL Light. These properties regarding the conversions and scaling factors ensure the correctness of our formalization of posit presented in Section 4.1 of the paper. Moreover, they would be useful for performing the arithmetic based on posit.

It is important to note here that our HOL Light code for the formal verification of Unums is publicly available for download at [35] and thus can be used by other researchers in the development of a formal library for Unum arithmetic.

2 Preliminaries

This section introduces the HOL Light theorem prover and posits.

2.1 HOL Light Theorem Prover

HOL Light [36] is a widely used interactive proof assistant for higher-order logic. The HOL Light is written in the strongly-typed functional programming language ML [37]. Theorems are formalized as axioms or inferred from the already verified theorems available in theories by inference rules. A theorem consists of a finite set Ω of Boolean terms (assumptions) and a Boolean term S as a conclusion. A new theorem is verified using any previously proved theorems and the primitive inference rules or applying existing axioms/inference rules in the HOL Light theorem proving environment that preserves the soundness of this approach. Many mathematical concepts have been formalized as HOL Light theories. A theory consists of a collection of valid HOL Light types, constants, axioms, definitions, and theorems. The HOL Light theorem proving system offers a wide range of theories, such as Boolean algebra, arithmetic, real numbers and list theories, which are extensively used in our formalization. Various automatic proof procedures [38] are also available in HOL Light to help and guide the user in conducting a proof effectively, efficiently and professionally. HOL Light has been used for the formal verification of floating-point numbers, the arithmetic involving these numbers and the associated algorithms. Some of the notable contributions are the formal verification of IA-64 division algorithms [23], square root algorithms [26], floating-point trigonometric functions [23] and floating-point exponential functions [13], development of a machine-checked theory of floating point arithmetic for the IA-64 architecture [14], parameterized floating-point formalization [39] and hierarchical verification of the IEEE-754 table-driven floating-point exponential function [40]. Similarly, some notable contributions in PVS related to arithmetic systems include the formalization of IEEE-854 floating-point standard [19], and the formal verification of IEEE rounding [41], IEEE compliant subtractive division algorithms [42], VAMP floating point unit [20] and IEEE floating point adder [43]. However, the HOL Light theorem prover supports automated reasoning of a larger set of computer arithmetic foundations that are widely used for analyzing the continuous dynamics of the engineering and physical systems, which is one of the motivations for choosing it for our proposed formalization of posit. This set of computer arithmetic foundational libraries will be extensively used in making a comparison between formal analysis of posits and floating-point numbers, which is one of our future directions. Moreover, the HOL Light theorem prover has the smallest trusted core (i.e., approximately 400 lines of Ocaml code) amongst all other higher-order-logic theorem prover and the underlying logic kernel has been verified in the CakeML project [44, 45].

Some standard symbols, their meanings and their HOL Light representations used in this paper are presented in Table 1.

Table 1: HOL Light Symbols

HOL Light Symbols	Standard Symbols	Meanings
~	not	Logical *negation*
<=>	=	Equality in Boolean domain
`num`	$\mathbb{N}$	Natural numbers data type
`real`	$\mathbb{R}$	Real data type
`SUC n`	$(n+1)$	Successor of natural number
&a	$\mathbb{N} \to \mathbb{R}$	Casting from a Natural number a to a Real number a
@f	Hilbert choice operator	Returns f if it exists
`k DIV m`	*quotient*	Returns the quotient of the division of two real numbers k and m
`-- x`	$-x$	Negative x
`EL n l`	*element*	Extracts n^{th} element of List l
`LAST l`	*last element*	Last element of List l
`[a; b; c]`	$[a, b, c]$	List having elements as a, b and c

In order to facilitate the understanding of the paper, we presented majority of the formalization of posits (Sections 4.1and 4.2) in simple Math notation. However, for some of the HOL Light functions/symbols, we used the mathematical notations presented in Table 2. Some of these notations may not correlate with the traditional conventions. However, they have been considered only to facilitate the understanding of the paper.

2.2 Posits (Unum-III)

Posits (Unum-III) [2], utilize a fixed number of bits as opposed to Type I Unums. The precise number may be chosen for a particular implementation, ranging from two bits up to many thousand bits. Posits may be simply implemented both in hardware and software. Moreover, they employ the similar type of low-level circuit building blocks that IEEE-754 floating-point numbers utilize for performing various arithmetic operations, such as integer addition and multiplication, and those also cover less chip area. Figure 1 presents a structure of an n-bit posit representation.

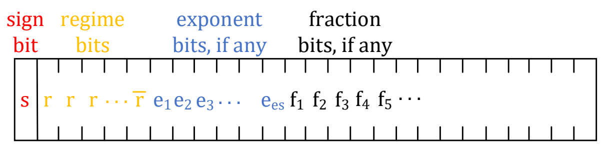

Fig. 1: Generic Posit Format for Finite, Nonzero Values

Posit representation consists of sign, regime, exponent and fraction bits. It is to be highlighted that the only boundary is shown between the sign bit and the rest of the bits since the other boundaries are flexible and depend on number of the regime bits. The regime bits are a sequence of identical bits r (all 1s or 0s), which are terminated by the opposite bit $\overline{r}$ for the case of non-zero exponent and fraction bits. In the case of

Table 2: Conventions used for HOL Light Functions

HOL Light Functions	Mathematical Conventions	Descriptions
/\	$\wedge$	Logical *and*
\/	$\vee$	Logical *or*
$\sim$(a = b)	$a \neq b$	a is not equal to b
`!x.t`	$\forall$x.t	For all $x : t$
`?x.t`	$\exists$x.t	There exists $x : t$
`\x.t`	λx.t	Function that maps x to $t(x)$
==>	$\Rightarrow$	Implication
&a	$\dot{a}$	Casting from a Natural number a to a Real number a
`int_of_num a`	$\hat{a}$	Casting from a Natural number a to an Integer a
`num_of_int a`	$\tilde{a}$	Casting from an Integer a to a Natural number a
`z pow n`	z^n	z raise to power Natural number n
`x ipow y`	x^y	x raise to power integer y
`a EXP b`	a^b	a raise to power b, where a and b are the natural numbers
`nb_num`	nb_n	Casting from an Integer nb to a Natural number using `int_of_num`
`es_num`	es_n	Casting from an integer es to a Natural number using `int_of_num`
`TL l`	$\underline{l}$	Tail of List l
`CONS h t`	h::t	Concatenates head h of a List with its tail t
`HD l`	$\bar{l}$	Head of List l
`APPEND l1 l2`	l_1 ++ $l2$	Append List l_1 with List l_2
`MEM m l`	$m \in l$	m is a member of List l
$\sim$(MEM m l)	$m \notin l$	m is not a member of List l
`NIL l`	[ ]	List l is empty
`LENGTH l`	$\lvert l \rvert$	Length of List l
`real_to_posit_ check3`	posit_{real}	Conversion of a real number to its corresponding posit representation
`add_zero_real`	real_{posit}	Conversion of a posit representation to its corresponding real number
`exponential_ rounding1`	round_e	Exponential rounding of a posit representation
`exponential_ round_ check1`	cond_e	Condition on the exponent bits in case of exponential rounding
`scale_factor_e`	$\mathrm{scaling}_e$	Scaling factor of the exponent bits
`fraction_ rounding1`	round_f	Fractional rounding of a posit representation
`fraction_ residue_set1`	$\mathrm{residue}_f$	Condition on the residue value in case of fractional rounding
`scale_factor_f`	$\mathrm{scaling}_f$	Scaling factor of the fraction bits
`scale_factor_r`	$\mathrm{scaling}_r$	Scaling factor of the regime bits

zero exponent and fraction bits, identical bits r in a regime are terminated by the end of the posit. The sign bit serves the purpose of representing the positive and negative numbers, i.e., it is 0 for the positive numbers and 1 for the negative numbers. Moreover, we need to take the 2s complement for the negative numbers before decoding the regime, exponent, and fraction bits.

To capture the idea of regime, Figure 2 provides some binary strings and their corresponding interpretations as real numbers k determined by the run length of the regime bits. Here, the symbol x in a bit string models the *don't care* condition, i.e., the interpretation does not depend on the value of that bit.

The leading bits in all bit strings (Figure 2) are known as the regime of the number. All binary strings start with some sequence of all 0 and all 1 bits in a row and terminate

Binary	0000	0001	001x	01xx	10xx	110x	1110	1111
Numerical meaning, k	-4	-3	-2	-1	0	1	2	3

Fig. 2: Regime Bit Illustration

by either the complementary bit or the end of the posit. The identical bits r of the regime bits are color-coded in amber, whereas, the opposite bit $\overline{r}$ that terminates the run, if any, is color-coded in brown. Assume m represents the number of identical bits in a run. If the identical bits in a regime bit are 1, then $k = m - 1$, otherwise, it is k = -m as given in Figure 2. The regime provides a scale factor of $useed^k$, where $useed$ = $2^{2^{es}}$ with es representing maximum exponent bits.

The next bits in a posit structure are the exponent e bits that are color-coded in blue (Figure 1) and are considered as an unsigned integer. They model a scaling factor of 2^e. There can be a maximum of es exponent bits depending on the bits remaining on the right side of the regime.

Any bits left after the regime and the exponent bits in a posit model the fraction f and it is quite similar to the fraction $1.f$ in a floating-point number, with 1 as a hidden bit. Moreover, there are no subnormal numbers with a hidden bit of 0 as they are in floating-point numbers. The two exception values for posit are 0 and $\pm\infty$. When all bits of a posit are zero, it represents the number 0. Whereas, the first bit as 1 and the remaining bits as 0s represent the value $\pm\infty$.

Now, we illustrate the posit format (structure of a posit representation) described above, using an example of decoding a 16-bit posit 0000110111011101 (Figure 3).

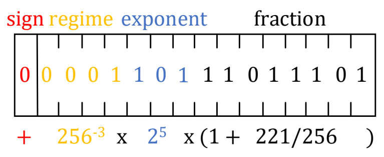
sign regime exponent fraction
0 0 0 0 1 1 0 1 1 1 0 1 1 1 0 1
+ 256^-3 x 2^5 x (1 + 221/256)

Fig. 3: Decoding of a 16-bit Posit

We have picked $es = 3$, which causes the value represented by the regime bits to provide a scaling factor between the negative and positive powers of $2^{2^3} = 256$. It is important to note here that the standard 16-bit posit consist of es of size 1. However, we have taken it as 3 for illustration purposes as shown in Figure 3. The sign bit of 0 asserts that it is a positive value/number. The regime bits consist of a run of three 0s that is terminated by a 1, making the power of useed equal to -3. The regime bits present a scale factor of 256^{-3}. The exponent bits, 101, represent the decimal number 5 as an unsigned binary integer, and introduce another scale factor 2^5. Finally, the fraction bits 11011101 represent 221 as an unsigned binary integer, so the fraction becomes $1 + \frac{221}{256}$. The overall value decoded by a 16-bit posit is given as follows:

$$256^{-3} \times 2^5 \times \left(1 + \frac{221}{256}\right) = 477 \times 2^{-27} \approx 3.55393 \times 10^{-6}$$

3 Related Work

This section provides some related work regarding the formal verification of floating-point numbers, and hardware and software implementations of posits.

3.1 Formal Verification of Floating-point Numbers

Many theorem provers, such as HOL Light, Coq, ACL2 and PVS have been used for the formal verification of the floating point numbers and their arithmetic. Miner [19] employed the PVS theorem prover for a formalization of ANSI/IEEE-854 standard for Radix-Independent floating-point arithmetic. It mainly involves the mapping of floating-point numbers to reals, mapping of reals to floating-point numbers, rounding and various arithmetic operations, such as addition, subtraction, multiplication, division and square root operations. Similarly, Berg et al. [20] developed a formal library for IEEE rounding [41] in PVS while utilizing the formal definition of rounding provided by Miner. Moreover, the authors used it to formally verify the correctness of a fully IEEE compliant floating-point unit used in the VAMP processor. Some more notable contributions in PVS related to arithmetic systems include the formal verification of IEEE compliant subtractive division algorithms [42], VAMP floating point unit [20] and IEEE floating point adder [43].

Daumas et al. [15] provided a generic library to formally reason about the floating-point numbers using the Coq theorem prover. The proposed formal library for the floating-point arithmetic caters for an arbitrary floating-point format and an arbitrary base, i.e., it accommodates both bases 2 and 10 for the IEEE-784 standard. Similarly, Boldo et al. [16] proposed a framework for formally verifying the floating-point C programs. The authors extracted the verification conditions from C programs annotated at the source code level that are discharged using Coq.

Harrison [13] provided the formal verification of an algorithm for computation of the exponential function in IEEE-754 standard binary floating-point arithmetic using the HOL Light theorem prover. Later, Harrison [14] generalized the formal library of floating point arithmetic, by incorporating a wide variety of floating point formats. Moreover, the authors used their proposed formalization for the verification of the floating point arithmetic performed in Intel Itanium Architecture (IA)-64. Similarly, Harrison [23] provided a number of formally verified algorithms for the evaluation of the transcendental functions, such as sine and cosine, for Intel IA-64 using double-extended precision floating point arithmetic. Some more notable contributions in HOL Light are the formal verification of IA-64 division algorithms [23], square root algorithms [26], parameterized floating-point formalization [39] and hierarchical verification of the IEEE-754 table-driven floating-point exponential function [40].

O'Leary et al. [46] proposed a hybrid verification approach, based on theorem proving and model checking, for formally verifying the Intel's FPU at the gate level. Akbarpour et al. [47] presented a formalization of fixed-point arithmetic using the HOL theorem prover. The authors formally modeled the fixed-point number system and provided specifications of various rounding modes, such as the directed and even rounding modes. Moreover, they performed an error analysis for verifying rounding and various arithmetic operations, such as addition, subtraction, division and multiplication.

Moore et al. [17] used the automated theorem prover ACL2 for formally verifying the AMD-K5 floating-point division unit using ACL2. Similarly, Rusinoff [18] formally verified the correctness of the floating point arithmetics, such as multiplication, division, and square root instructions of the AMD-K7 microprocessor using ACL2.

Intel has been applying formal verification after the incident of the infamous Intel processor bug. Indeed, there is a research group working on the verification of the different components/operations of the Intel processors, over the years. Some of the notable contributions are from Harrison [14, 22–26], O'Leary [27], Narasimhan and Kaivola [28, 29], Slobodova [30] and Peter Tang [25] who have been working in the Intel research group. Moreover, to the best of our knowledge, no bugs have been reported regarding Intel processors in literature after the infamous Pentium IV bug. The application of the formal verification could be one of the main reasons behind this as well. For example, Bentley [10] describes the steps for the identification of the bugs in the Pentium IV processor design prior to initial silicon. The author claims that he identified over 100 logic bugs and about 20 of them were *high quality* bugs that would not have been found using any other pre-silicon validation processes. Two out of those 20 bugs were classic floating point data space problems. In particular, the Floating ADD (FADD) instruction had a bug, where the 72-bit Floating Point (FP) adder was setting the carryout bit to 1 for a specific combination of source operands when there was no actual carryout. The author believes that if this error had not been caught, it may have resulted into a bug similar to the Floating Divide (FDIV) problem of the Pentium processor. More details about the validation of the Intel Pentium 4 microprocessor by the Intel research group can be found at [10]. Moreover, some experiences of O'Leary at Intel about the verification of the floating point arithmetic of the Intel Pentium 4 and Core $i7$ processors can be found at [48]. Similarly, many bugs have been found in different software over the past years. For example, Gesellensetter et al. [49] found a bug in the scheduler of the GNU Compiler Collection (GCC) compiler for a Very Long Instruction Word (VLIW) processor during its verification using the Isabelle/HOL theorem prover. Johnson [50] provides some natural history of bugs and discusses about the usage of the formal methods for analyzing the software issues in space related applications. Similarly, Fitzgerald et al. [51] discusses about the deployment of the formal verification method in industrial applications. Moreover, Zhang et al. [52] survey the successful deployment of formal methods in the industrial settings. Since posits are intended as drop-in replacement for floating-point numbers in computer systems, their formal verification is of utmost important to ensure the absence of any bugs before they are used in the processors.

Rockwell Collins Inc., a famous multinational company providing products and services regarding the aerospace applications, and NASA have been successfully using formal methods for analyzing various aspects of avionics. Whalen et al. [31] integrated formal methods with the model-based development tools, i.e., Simulink and

SCADE Suite for the verification of the software during the design cycle for safety-critical avionics applications. The authors developed a set of tools that translate the Simulink models to formal models that can be used by the model checkers and theorem provers for the automatic analysis of these models. Moreover, they formally analyzed an Unmanned Aerial Vehicle (UAV) controller modeled in Simulink. During the analysis of the controller, the authors formally verified over 60 properties and they identified 10 modelling errors and 2 requirement errors in the relatively mature models of the system. Similarly, Miller et al. [32] performed the formal analysis of Flight Critical Software (FCS) 5000, a new family of flight control systems developed by Rockwell Collins Inc that is widely used in business and regional jet aircraft. Moreover, the research team at NASA has worked on the formal verification of a flight critical software [32], software safety analysis of a flight guidance system [33] and safety analysis of software intensive systems [34]. More details about their research contributions in this direction can be found at [53, 54].

Similarly, Barnat et al. [55] integrated the DIVINE model checker and HiLiTE, a tool for requirements-based verification of aerospace system components developed and used by Honeywell for the formal verification of the avionics Simulink models. The authors used their proposed framework for formally analyzing the Voter Core that is a sub-system of the common avionics triplex sensor voter. Cao et al. [56] presented a framework for formally verifying the airborne software based on DO-333 and thus providing guidance in the integration of formal methods in the development and analysis of the software. The authors used their proposed methodology for the formal verification of Air Data Computer (ADC) software. Moreover, the authors claim that they identified 16 errors during the verification of ADC software [56]. Some more notable contributions regarding application of formal methods in avionics and aerospace are [57–61]. However, none of these contributions cater for posit.

3.2 Hardware and Software Implementations of Posits

Posits have been implemented as an alternative to the floating-point number format in hardware and software. Lehoczky et al. [7] presented the software and hardware implementations of posits. The authors used C# programming language to implement posits on the .NET platform. Moreover, they used Hastlayer, which is a tool for converting .NET models to a language that can be implemented on FPGA, to develop a hardware based implementation of posit. Similarly, the hardware architecture of Unum adder/subtractor and multiplier has been designed and implemented in FPGAs. Moreover, a Verilog HDL generator has been constructed for performing these arithmetic operations [4–6, 62]. The software implementations of posits are also available in C# [7], C[6], C++[7] and Julia[8] programming languages. Moreover, it has been experimentally shown that posits perform better than the fixed point number system, in terms of accuracy and memory utilization, for both training and inferences of deep convolutional neural networks [8, 9, 63]. However, none of the above-mentioned works

[6]https://github.com/libcg/bfp
[7]https://github.com/eruffaldi/cppPosit
[8]https://github.com/milankl/SoftPosit.jl

(presented in Sections 3.1 and 3.2) provide the verification of Unums, which is the main scope of the paper.

4 Results

4.1 Formalization of Posits

This section provides a higher-order-logic based formalization of posits using the HOL Light theorem prover. It mainly involves a conversion from a posit representation having a bit pattern to its corresponding real number and vice versa. Moreover, a conversion from a posit representation to its corresponding real number is mainly based on extracting the regime, exponent and fraction bits.

A posit format representation comprises four components, namely, sign, exponent, regime and fraction bits. It is sufficient to define a computing environment for posits, i.e., a posit configuration, using the total number of bits (nb) having an integer value of greater than or equal to 2 and the number of exponent bits (es) with an integer value of greater than or equal to 0 [64]. For example, for the case of $nb = 2$ and $es = 0$, the two bits comprising the posit are the sign and the regime bits, respectively. We model a valid posit configuration as the following HOL Light function:

Definition 1: *Valid Posit*

$\vdash_{def} \forall(\mathsf{nb{:}int})\ (\mathsf{es{:}int}).\ \mathsf{is\_valid\_posit}\ (\mathsf{nb,es}) = (\mathsf{nb} \geq \widehat{2}) \wedge (\mathsf{es} \geq \widehat{0})$

The function is_valid_posit accepts a pair of integers (nb,es), describing the total number of bits nb and the number of exponent bits es and returns a valid posit configuration providing the constraints on these number of bits. We model a posit configuration using *new type definition* feature of HOL Light as follows:

```
let posformat_tpbij = new_type_definition "posit" ("mk_posit", "dest_posit")
(prove(`?(pst:int#int). is_valid_posit pst`, REWRITE_TAC [PROOF_TYPE]));;
```

where posit models a new type by providing its name and bijection alongwith a theorem asserting that bijection. The function mk_posit projects a pair of integers to a posit type and dest_posit maps a posit to a pair of integers. Next, to model a valid length of a posit, we first extract the elements of the pair (nb, es) in HOL Light as follows:

Definition 2: *Extraction of the Elements of Pair (nb, es)*

$\vdash_{def} \forall(\mathsf{P{:}posit}).\ \mathsf{nb}\ \mathsf{P} = \mathsf{FST}\ (\mathsf{dest\_posit}\ \mathsf{P})$

$\vdash_{def} \forall(\mathsf{P{:}posit}).\ \mathsf{es}\ \mathsf{P} = \mathsf{SND}\ (\mathsf{dest\_posit}\ \mathsf{P})$

$\vdash_{def} \forall(\mathsf{P{:}posit}).\ \mathsf{nb_n}\ \mathsf{P} = \widetilde{(\mathsf{nb}\ \mathsf{P})}$

$\vdash_{def} \forall(\mathsf{P{:}posit}).\ \mathsf{es_n}\ \mathsf{P} = \widetilde{(\mathsf{es}\ \mathsf{P})}$

The function nb accepts a posit configuration P and extracts the total numbers of bits (nb) of a posit as an integer. Similarly, the function es extracts the numbers of exponent bits (es) of a posit as an integer. The functions $\mathsf{nb_n}$ and $\mathsf{es_n}$ use num_of_int to cast the integers nb and es to natural numbers.

Now, we model a valid posit length as follows:

Definition 3: *Valid Posit Length*

$\vdash_{def} \forall(\mathsf{P{:}posit})\ (\mathsf{L{:}bool\ list}).\ \mathsf{is\_valid\_posit\_length}\ \mathsf{P}\ \mathsf{L} = (|\mathsf{L}| = \mathsf{nb_n}\ \mathsf{P})$

The function is_valid_posit_length accepts a posit configuration `P` and a list `L:bool list`, i.e., a posit representation, capturing the bit values of a posit format, and returns a valid length of a posit.

Next, we model the two exception values for posits [64] in HOL Light as follows:

Definition 4: *Exceptions (Zero and Infinity)*

$\vdash_{def} \forall(\mathsf{L{:}bool\ list}).\ \mathsf{zero\_exception\ L} = \mathsf{T} \notin \mathsf{L}$

$\vdash_{def} \forall(\mathsf{L{:}bool\ list}).\ \mathsf{inf\_exception\ L} = \overline{\mathsf{L}} \land (\mathsf{T} \notin \underline{\mathsf{L}})$

The functions zero_exception and inf_exception present the exception values *zero* and $\pm\infty$, respectively. If all bits of a posit representation are 0, it represents an exception value *zero*. Similarly, if the first bit is 1 and the rest of the bits are 0, it provides an exception value $\pm\infty$.

Next, we model the seed value for the posit P, described in Section 2.2, as the following HOL Light function:

Definition 5: *Seed Value*

$\vdash_{def} \forall\mathsf{P}.\ \mathsf{useed\ P} = \dot{2}^{\dot{2}^{(\mathsf{es_n\ P})}}$

Definition 6: *Minimum and Maximum Positive Value of Posits*

$\vdash_{def} \forall(\mathsf{P{:}posit}).\ \mathsf{maxpos\ P} = (\mathsf{useed\ P})^{(\mathsf{nb_n\ P\ -\ 2})}$

$\vdash_{def} \forall(\mathsf{P{:}posit}).\ \mathsf{minpos\ P} = \dfrac{1}{\mathsf{maxpos\ P}}$

The functions maxpos and minpos model the largest and smallest positive real numbers (values) expressible as a posit P, respectively [2].

Definition 7: *Check Extreme Values of Posit*

$\vdash_{def} \forall\mathsf{L}.\ \mathsf{checkmax\ L} = (\mathsf{F} \notin \underline{\mathsf{L}})$

The HOL Light function checkmax accepts a posit representation L, containing all bit values of a posit format, and returns a Boolean value true (T) if all bits are equal to 1 except the first (leading) bit, which can be either 0 or 1. For the case of first bit, i.e., sign bit equal to 0, it captures the largest positive value, whereas, it models the largest negative value for a sign bit 1.

Next, to calculate the scale factor of the regime bits, i.e., *useed*^{k}, we first formalize k (power of the variable *useed*) in HOL Light as the following recursive function:

Definition 8: *Value of k for Scaling Factor of the Regime Bits*

$$\begin{aligned}
&\vdash_{def} \forall(\mathsf{h{:}bool})\ (\mathsf{t{:}bool\ list}).\ \mathsf{value\_of\_k}\ [\ ]\ = 0\ \land \\
&\quad \mathsf{value\_of\_k}\ (\mathsf{h{:}t}) = \mathsf{if}\ \bar{\mathsf{t}}\ \mathsf{then}\ (\mathsf{if}\ \mathbf{[C_1]}(\bar{\mathsf{t}} = \overline{(\underline{\mathsf{t}})}) \land \mathbf{[C_2]}(1 < |\mathsf{t}|) \\
&\qquad\qquad\qquad\qquad \mathsf{then}\ ((\mathsf{value\_of\_k}\ \mathsf{t}) + 1)\ \mathsf{else}\ 0) \\
&\qquad\qquad\qquad \mathsf{else}\ (\mathsf{if}\ \mathbf{[C_3]}(\bar{\mathsf{t}} = \underline{\overline{(\mathsf{t})}}) \land \mathbf{[C_4]}\ (1 < |\mathsf{t}|) \\
&\qquad\qquad\qquad\qquad \mathsf{then}\ ((\mathsf{value\_of\_k}\ \mathsf{t}) - 1)\ \mathsf{else}\ -1)
\end{aligned}$$

The function value_of_k accepts a posit representation L: bool list and returns k (power of the variable *useed*) for the scaling factor of the regime bits. If the identical bits in a regime for a run are 1, then value_of_k is equal to *run length* - 1, otherwise it is equal to the negation of *run length*. Here, *run length* corresponds to the variable m presented in Section 2.2 and models the length of the regime bits.

Now, the scaling factor of the regime bits is formalized in HOL Light as follows:

Definition 9: *Scaling Factor of the Regime Bits*

$\vdash_{def} \forall(\mathsf{P{:}posit})\ (\mathsf{L{:}bool\ list}).\ \mathsf{scaling_r\ P\ L} = (\mathsf{useed\ P})^{(\mathsf{value\_of\_k\ L})}$

To convert a posit to its equivalent real number, we require the scaling factors of the exponential and the fraction bits, which further need an extraction of these bits from a given posit representation. Moreover, for both these extractions, we need to pick elements (bit values) from a posit representation, which is formalized as the following HOL Light function:

Definition 10: *Pick Elements From a List*

$\vdash_{def} \forall(\mathsf{L{:}bool\ list})\ (\mathsf{l{:}num})\ (\mathsf{u{:}num}).$

$\qquad \mathsf{pick\_elements\ L\ l\ u} = \mathsf{pick\_elements\_simp\ L\ l\ ((u - l) + 1)}$

The function pick_elements accepts a list L, a lower index l and an upper index u and returns a list containing the elements of the input list from l to u indices. It uses a recursive function pick_elements_simp to extract the required elements from a given list.

Next, we extract the exponent bits of a posit representation as follows:

Definition 11: *Extracting Exponent Bits*

```
⊢def ∀(P:posit) (L:bool list). exp_bits P L =
  if [C1]((regime_length L) + 1) < (nb_n P) ∧ [C2](1 ≤ (eb_n P)) then
            (if [C3]((regime_length L) + 1 + (eb_n P)) ≤ |L| then
          pick_elements L ((regime_length L) + 1) ((regime_length L) + (eb_n P))
             else pick_elements L ((regime_length L) + 1) (|L| − 1))
   else [ ]
```

The function exp_bits accepts a posit configuration P and a posit representation L and returns the exponent bits of the posit. Here, the function regime_length provides the length of the regime bits.

Now, the scaling factor of the exponent bits is formalized as the following HOL Light function:

Definition 12: *Scaling Factor of Exponent Bits*

$\vdash_{def} \forall(\mathsf{P{:}posit})\ (\mathsf{L{:}bool\ list}).\ \mathsf{scaling_e\ P\ L} = \dot{2}^{\mathsf{BV\_n\ (exp\_bits\ P\ L)\ *\ (2^{eb_n\ P\ -\ exp\_length\ P\ L})}}$

where the function BV_n provides a natural number representation of the bit values. There can be a maximum of es exponent bits depending on the bit left on the right side of the regime in a posit representation. Therefore, the function $\mathsf{scaling_e}$ provides a scale factor of the exponent bits by incorporating the scenario, where the exponent bits exp_bits e are less than es. Moreover, the exponent bits scales from 0 to 2^{es}.

Next, we extract the fraction bits of a posit representation as follows:

Definition 13: *Extracting Fraction Bits*

```
⊢def ∀(P:posit) (L:bool list). fraction_bits P L =
   if [C]((regime_length L) + (exp_length P L) + 1 < (nb_n P)) then
      pick_elements L ((regime_length L) + (exp_length P L) + 1) ((nb_n P) − 1)
   else [ ]
```

The function fraction_bits accepts a posit configuration P and a posit representation L and returns the fraction bits of a posit representation, if any.

Now, we formalize the scaling factor of the fraction bits as the following HOL Light function:

Definition 14: *Scaling Factor of Fraction Bits*

$$\vdash_{def} \forall(\mathsf{P{:}posit})\ (\mathsf{L{:}bool\ list}).\ \mathsf{scaling_f\ P\ L} = \dot{1} + \frac{\dot{(\mathsf{BV\_n\ (fraction\_bits\ P\ L)})}}{\dot{2}^{(\mathsf{fraction\_length\ P\ L})}}$$

where the function fraction_length provides the length of the fraction bits. The fraction bits serves the same functionality as they do in the floating-point numbers.

Finally, we formalize a conversion of a posit to its equivalent real number as follows:

Definition 15: *Posit to Real Number Conversion*

$\vdash_{def} \forall$(P:posit) (L:bool list). posit_to_signed_real P L =

if **[C$_1$]** zero_exception L then $\dot{0}$

else (if **[C$_2$]** (checkmax L) then (if **[C$_3$]** sign_bit L then $-\dfrac{\&1}{\text{maxpos P}}$

else maxpos P)

else (if **[C$_4$]** (sign_bit L) then $-$(scaling$_r$ P L' $*$ scaling$_e$ P L' $*$ scaling$_f$ P L')

else (scaling$_r$ P L) $*$ (scaling$_e$ P L) $*$ (scaling$_f$ P L)))

where L' = (sign_bit L)::(two_complement $\underline{\text{L}}$).

The function posit_to_signed_real accepts a posit configuration P and a posit representation L and returns a real value corresponding to the given posit. The first conditional statement of the function posit_to_signed_real (Condition C$_1$) checks all bits of a posit representation using a function add_zero_real. For the case of all bits equal to zero, it returns a real number/value 0. Otherwise, the second conditional statement (Condition C$_2$) uses the function checkmax (Definition 7) to confirm if the given posit representation provides a largest positive or a largest negative real value for the sign bit values of 0 and 1, respectively. For the scenario when a posit representation does not capture any largest positive or negative values, it returns the corresponding real number, which can be any positive or negative value depending on the sign bit of the given posit representation. For example, for a sign bit 1, it uses the notion of 2s complement to represent a negative real number.

Now, we provide the formalization of the conversion function from a real number to posit, which is mainly based on the notion of the exponential and the fractional rounding. The approach for converting a real number to its corresponding posit representation is quite similar to the method used for transforming any real number to float in floating-point arithmetic. For the case of the floating-point numbers, the first step involves checking for the exception values, which are only 0 and $\pm\infty$ for the case of posits. If the number does not represent any extreme values, it is divided by 2 or multiplied by 2 until it is in the interval $[1, 2)$, and thus determining the fraction bits for the corresponding floating-point number. For the case of posits, the given real number is, first, repeatedly divided or multiplied by *useed* until it is in the interval $[1, useed)$. Then, the non-negative exponent for the posit is determined by repeatedly divided by 2 until it is in the interval $[1, 2)$. The fraction always consists of a leading 1 bit to the left of the binary point and does not require handling any subnormal exception values that have a 0 bit to the left of the binary point.

Definition 16: *Negative Real Number (Sign Bit)*

$\vdash_{def} \forall$(x:real). sign_real x = (x < $\dot{0}$)

The function sign_real accepts a real number x and returns true if it is negative.

First, we formalize the regime bits (regime field) for a posit corresponding to a real number in HOL Light as follows:

Definition 17: *Regime Bits (Regime Field)*

$\vdash_{def} \forall$(x:real) (P:posit). regime_bits x P =

```
    if [C_1](1̇ ≤ x) then (get_regime_ones x P ((nb_n P) − 2))
                        else ( get_regime_zeros  x P ((nb_n P) − 2))
⊢_def ∀(x:real) (P:posit) (n:num).
    get_regime_zeros x P 0 = if [C_2](x = 0̇) then [F] else [T] ∧
    get_regime_zeros x p (SUC n) = if [C_3](1̇ ≤ x) then (T::[ ]) else
        F::(get_regime_zeros (x ∗ (useed P)) P n)
⊢_def ∀(x:real) (P:posit) (n:num). get_regime_ones x P 0 = [T] ∧
   get_regime_ones x P (SUC n) = if [C_4](1̇ ≤ x < useed P) then T::(F::[ ]) else
       T::(get_regime_ones  x/(useed P)  P n)
```

The function regime_bits accepts a real number x and a posit configuration P and provides the regime bits of a posit representation corresponding to the given real number x. It mainly asserts a condition on the value of x, i.e., if $1 \leq x$, then it uses the function get_regime_ones to obtain identical regime bits 1 terminated with a 0. If the condition on x is false, it uses the function get_regime_zeros to generate a sequence of 0s in the regime field terminated by a 1.

Generally, three distinct cases arise during a conversion of a real number to its corresponding posit, i.e., 1) the resultant posit consists of the regime, exponent and fraction bits; 2) it has only the regime and the exponent bits (no fraction bits); 3) it has only regime bits (no exponent and fraction bits). These cases depend on the fact if the notion of rounding is involved in the conversion or not, i.e., if a real number is exactly expressible as a posit using the number of bits mentioned in a posit configuration or it requires more bits, where it is rounded to a nearest valid posit representation. The notion of fractional rounding is used for the case when we need more fraction bits than the number of bits left after the regime and the exponent bits in a posit representation, to express the given real value as a posit. Whereas, the exponential rounding captures the scenario, where the number of exponent bits e in a posit representation is less than the value es given in a posit configuration P. Moreover, in both types of rounding, i.e., the fractional and the exponential, if the input real value is at the tie-breaking point, it is rounded to the nearest even posit having the last bit equal to zero. The fractional rounding is quite similar to that of the floating-point numbers. In fractional rounding, the tie-breaking point is the arithmetic mean of the two choices (lower and upper bounds) for the rounding and the posit is rounded to the nearest fraction. However, in the case of the exponential rounding, the tie-breaking point is the geometric mean of the two choices (lower and upper bounds). Moreover, in the case of a real number not exactly expressible as a posit, i.e., the exponent bits are truncated and the real value is either rounded above or rounded down to a valid posit representation given in Equations (1) and (2) [65]. For example, for two posits 32 and 128, the tie-breaking point is 64. Therefore, any value greater than 64 maps to 128 and a value less than 64 is rounded to 32.

$$e^{+} = \left(\left\lfloor \frac{e}{2^t} \right\rfloor + 1\right) 2^t \tag{1}$$

$$e^{-} = \left\lfloor \frac{e}{2^t} \right\rfloor 2^t \tag{2}$$

Similarly, the fractional rounding is based on the residue left after the computation of the fraction bits and the tie-breaking point, which is $\frac{1}{2}$. If the residue is less than $\frac{1}{2}$, the corresponding posit is rounded down, otherwise it is rounded up.

Now, we formalize the exponential rounding in HOL Light as follows:

Definition 18: *Exponential Rounding*

$\vdash_{def} \forall (\mathsf{x:real})\ (\mathsf{P:posit}).\ \mathsf{round_e}\ \mathsf{x}\ \mathsf{P} =$

$\quad \mathsf{if}\ \mathbf{[C_1]}\ \tilde{}(\mathsf{exp\_residue}\ \ \mathsf{x}\ \mathsf{P}) \land \mathbf{[C_2]}\ \mathsf{M} = \mathsf{N}\ \mathsf{then}\ (\mathsf{exp\_posit\_tie}\ \mathsf{x}\ \mathsf{P})$

$\quad \mathsf{else}\ \ (\mathsf{if}\ ((\mathbf{[C_3]}\ \mathsf{exp\_residue}\ \mathsf{x}\ \mathsf{P} \land \mathbf{[C_4]}\ (\mathsf{M} = \mathsf{N})) \lor \mathbf{[C_5]}\ (\mathsf{M} > \mathsf{N})\ \mathsf{then}$

$\qquad (\mathsf{exp\_posit\_up}\ \mathsf{x}\ \mathsf{P})\ \mathsf{else}\ (\mathsf{exp\_posit\_down}\ \mathsf{x}\ \mathsf{P}))$

where $\mathsf{M} = \mathsf{te\_rounded\_bits}\ \mathsf{x}\ \mathsf{P}$ and $\mathsf{N} = 2^{\mathsf{te\_bits\ x\ P\ -\ 1}}$.

The function $\mathsf{round_e}$ accepts a real number x and a posit configuration P and returns the exponential rounding of the corresponding posit representation. It mainly asserts a condition on the exponential residue exp_residue (checks if there is any value/residue left after extracting the regime and exponent bits) and the exponent bits te_rounded_bits (returns the value of the exponent bits) to be truncated. Moreover, the function te_bits provides the number of truncated bits. The HOL Light functions exp_posit_up accepts a posit configuration P and a real number x, and returns a list containing the regime bits and the binary representation of the value of the exponent of the rounded up posit. Similarly, exp_posit_down provides a list by appending the regime bits with the binary representation of the value of the exponent of the rounded down posit. More details about the exponential rounding and these functions can be found in a detailed technical report of our work [35].

Next, we model the fractional rounding as, if the residual value after computing the regime exponent and fraction bits is greater than $\frac{1}{2}$, then it is rounded up and it is rounded down for a residual value less than $\frac{1}{2}$. Moreover, if the residual value is equal to $\frac{1}{2}$, then it is rounded to the nearest even posit. We model it using the function $\mathsf{round_f}$ as follows:

Definition 19: *Fractional Rounding*

$\vdash_{def} \forall (\mathsf{x:real})\ (\mathsf{P:posit}).\ \mathsf{round_f}\ \mathsf{x}\ \mathsf{P} =$

$\mathsf{if}\ \mathbf{[C_1]}\ \mathsf{residue_f}\ \mathsf{x}\ \mathsf{P} = \dot{0}$

$\qquad \mathsf{then}\ \ (\ \mathsf{regime\_bits}\ \ \mathsf{x}\ \mathsf{P}){+}{+}((\mathsf{exp\_list}\ \mathsf{x}\ \mathsf{P}){+}{+}(\ \mathsf{set\_fraction\_list}\ \ \mathsf{x}\ \mathsf{P}))$

$\mathsf{else}\ \ (\mathsf{if}\ \mathbf{[C_2]}\ \mathsf{residue_f}\ \mathsf{x}\ \mathsf{P} = \dfrac{1}{2}\ \mathsf{then}\ (\mathsf{frac\_posit\_tie}\ \mathsf{x}\ \mathsf{P})$

$\quad \mathsf{else}\ \ (\mathsf{if}\ \mathbf{[C_3]}\ \mathsf{residue_f}\ \mathsf{x}\ \mathsf{P} > \dfrac{1}{2}\ \mathsf{then}\ (\mathsf{frac\_posit\_up}\ \mathsf{x}\ \mathsf{P})$

$\qquad \mathsf{else}\ \ (\ \mathsf{frac\_posit\_down}\ \ \mathsf{x}\ \mathsf{P})))$

The function $\mathsf{round_f}$ accepts a real number x and a posit configuration P and returns the fractional rounding of the corresponding posit representation. More details about the formalization of the fractional rounding can be found in the technical report [35].

Definition 20: *Minimum Positive Real Number in a Posit Representation*

$\vdash_{def} \forall (\mathsf{P:posit}).\ \mathsf{minpos\_posit}\ \mathsf{P} = \mathsf{num\_BV\_f}\ ((\mathsf{nb_n}\ \mathsf{P}) - 1)\ (1)$

The function minpos_posit accepts a posit configuration P and returns a posit representation of a minimum positive real number expressible in a posit.

Similarly, the function maxpos_posit captures a posit representation of a maximum (largest) positive real number expressible in a posit.

Definition 21: *Maximum Positive Real Number in a Posit Representation*
$\vdash_{def} \forall$(P:posit). maxpos_posit P = num_BV_f ((nb$_n$ P) − 1) ($2^{(\text{nb}_n\ \text{P})-1}$ − 1)

Finally, we use Definitions 16 - 21 to formalize the conversion of a real number to its corresponding posit representation as the following HOL Light function:

Definition 22: *Real to Posit Conversion*

```
⊢def ∀(x:real) (P:posit). posit_real x P =
if ([C1] x = 0̇ ∨ [C2] |x| ≥ maxpos P ∨ [C3] |x| ≤ minpos P) then
  (if [C4] x = 0̇ then [F]++(regime_bits x P)
   else
    (if [C5] |x| ≥ maxpos P then (sign_real x)::(if [C6] sign_real x then
        two_complement A else A)
     else ( sign_real x) ::( if [C7] sign_real x then two_complement B else B)))
else
  (if [C8] x > 0̇ then (if [C9] cond_e x P then [F]++H else [F]++K)
   else (if [C10] (cond_e |x| P) then [T]++(two_complement M) else
       [T]++(two_complement N)))
where A = maxpos_posit P,   B = minpos_posit P,   H = round_e x P,
      K = round_f x P,      M = round_e |x| P,    N = round_f |x| P.
```

The function posit$_{\text{real}}$ accepts a real number x and a posit configuration P and returns its equivalent posit representation. The satisfaction of the first conditional statement (Conditions $\mathsf{C}_1 - \mathsf{C}_7$) provides the posit representations for zero, minimum and maximum posits. Whereas, the satisfaction of the second conditional statement (Conditions $\mathsf{C}_8 - \mathsf{C}_{10}$) provides other posits corresponding to any positive or negative real numbers using the notions of the exponential and the fractional rounding. Moreover, for the case of the negative real numbers, it provides the 2s complement of the corresponding posit representation.

We use our higher-order-logic based formalization of posits, a conversion from a posit to its corresponding real number and a real number to its corresponding posit, presented in this section, to formally verify various properties providing the correctness of these conversions and the scaling factors of various bits, such as regime, exponential and the fraction bits in Section 4.2 of the paper.

4.2 Formal Verification of Posits

In this section, we present the formal verification of various properties of posits regarding the conversions and the scaling factors of the regime, exponential and the fraction bits using HOL Light. The verification of these properties not only ensures the correctness of our formal definitions presented in Section 4.1 but they are also quite vital for performing various arithmetic operations based on posits.

We formally verify various properties regarding bounds on scaling factors of the exponent, fraction and regime bits and are given in Table 3. For example, Theorem 4 provides an upper bound of $2^{2^{es}-1}$ on the scaling factor of the exponent bits. Similarly, Theorem 7 ensures that the scaling factor of the fraction bits is equal to 1, which indicates that no fraction bits are left after the exponential rounding. More details about the verification of these theorems can be found in the technical report [35].

Table 3: Formally Verified Properties regarding Scaling Factors of Various Bits

Theorem 1: *Positive Value of k* $\vdash_{thm} \forall(\mathsf{L{:}bool\ list}).\ [\mathsf{A}]\ \overline{(\underline{\mathsf{L}})} \Rightarrow (\mathsf{value\_of\_k\ L}) \geq 0$
Theorem 2: *Minimum Number of Bits of a Posit* $\vdash_{thm} \forall(\mathsf{P{:}posit}).\ 2 \leq \mathsf{nb_n\ P}$
Theorem 3: *Upper Bound on Length of the Exponent Bits* $\vdash_{thm} \forall(\mathsf{P{:}posit})\ (\mathsf{L{:}bool\ list}).\ [\mathsf{A}]\ \mathsf{is\_valid\_posit\_length\ \ P\ L} \Rightarrow \mathsf{exp\_length\ P\ L} \leq \mathsf{eb_n\ P}$
Theorem 4: *Upper Bound on Scaling Factor of the Exponent Bits* $\vdash_{thm} \forall(\mathsf{P{:}posit})\ (\mathsf{L{:}bool\ list}).\ [\mathsf{A}]\ \mathsf{is\_valid\_posit\_length\ \ P\ L} \Rightarrow \mathsf{scaling_e\ P\ L} \leq \left(2^{(2^{\mathsf{nb_n}}\ \dot{}\ \mathsf{P}\text{-}\ 1)}\right)$
Theorem 5: *Lower Bound on Scaling Factor of the Exponent Bits* $\vdash_{thm} \forall(\mathsf{P{:}posit})\ (\mathsf{L{:}bool\ list}).\ [\mathsf{A}]\ \mathsf{is\_valid\_posit\_length\ \ P\ L} \Rightarrow \dot{0} < \mathsf{scaling_e\ P\ L}$
Theorem 6: *Upper Bound on Scaling Factor of the Fraction Bits* $\vdash_{thm} \forall(\mathsf{P{:}posit})\ (\mathsf{L{:}bool\ list}).\ [\mathsf{A}]\ \mathsf{is\_valid\_posit\_length\ \ P\ L} \Rightarrow \mathsf{scaling_f\ P\ L} < \dot{2}$
Theorem 7: *Scaling Factor of the Fraction Bits in Exponential Rounding* $\vdash_{thm} \forall(\mathsf{P{:}posit})\ (\mathsf{L{:}bool\ list}).\ [\mathsf{A_1}]\ (\ \mathsf{is\_valid\_posit\_length\ \ P\ L}) \land$ $[\mathsf{A_2}]\ \mathsf{exp\_length\ \ P\ L} < \mathsf{eb_n\ P} \Rightarrow \mathsf{scaling_f\ P\ L} = \dot{1}$
Theorem 8: *Upper Bound on the Value of k* $\vdash_{thm} \forall(\mathsf{L{:}bool\ list}).\ [\mathsf{A_1}]\ \tilde{}(\mathsf{checkmax\ L}) \land [\mathsf{A_2}]\ (2 \leq \lvert\mathsf{L}\rvert) \Rightarrow \mathsf{value\_of\_k\ L} < (\lvert\mathsf{L}\rvert \dot{-} 2)$
Theorem 9: *Upper Bound on Scaling Factor of the Regime Bits* $\vdash_{thm} \forall(\mathsf{P{:}posit})\ (\mathsf{L{:}bool\ list}).\ [\mathsf{A_1}]\ \mathsf{is\_valid\_posit\_length\ \ P\ L} \land$ $[\mathsf{A_2}]\ \mathsf{is\_valid\_posit}\ \ (\ \mathsf{dest\_posit\ \ P}) \Rightarrow \mathsf{scaling_r\ P\ L} \leq \mathsf{maxpos\ P}$
Theorem 10: *Lower Bound on Scaling Factor of the Regime Bits* $\vdash_{thm} \forall(\mathsf{P{:}posit})\ (\mathsf{L{:}bool\ list}).\ [\mathsf{A}]\ \mathsf{is\_valid\_posit\_length\ \ P\ L} \Rightarrow \dot{0} < \mathsf{scaling_r\ P\ L}$
Theorem 11: *Total Length of a Posit* $\vdash_{thm} \forall(\mathsf{P{:}posit})\ (\mathsf{L{:}bool\ list}).\ [\mathsf{A}]\ \mathsf{is\_valid\_posit\_length\ \ P\ L}$ $\Rightarrow (\mathsf{exp\_length\ P\ L}) + (\mathsf{regime\_length\ L}) + (\mathsf{fraction\_length\ \ P\ L}) + 1 = \mathsf{nb_n\ P}$
Theorem 12: *Upper Bound on a Real Value Obtained from its Equivalent Posit* $\vdash_{thm} \forall(\mathsf{L{:}bool\ list})\ (\mathsf{P{:}posit}).\ [\mathsf{A_1}]\ \mathsf{is\_valid\_posit}\ \ (\ \mathsf{dest\_posit\ \ P}) \land$ $[\mathsf{A_2}]\ \ \mathsf{is\_valid\_posit\_length\ \ \ P\ L} \land [\mathsf{A_3}]\ \tilde{}(\mathsf{zero\_exception\ L}) \land [\mathsf{A_4}]\ \tilde{}(\mathsf{inf\_exception\ \ L})$ $\Rightarrow \mathsf{posit\_to\_signed\_real\ \ P\ L} \leq \mathsf{maxpos\ P}$

Next, we verify some important properties about the conversion of a real number to its equivalent posit (encoding) and a conversion of a posit to its equivalent real number (decoding). Some of these properties are presented in Table 4. For example, Theorem 14 ensures that every real number greater than the largest negative value of its corresponding posit representation is mapped to -minpos. More details about the verification of these properties alongside other formally verified properties can be found in our technical report [35].

Table 4: Formally Verified Properties regarding Encoding and Decoding of Posits

Theorem 13: *Encoding and Decoding of Zero* $\vdash_{thm}$ ∀(P:posit) (L:bool list) (x:real). [A$_1$] is_valid_posit (dest_posit P) ∧ [A$_2$] is_valid_posit_length P (posit$_{real}$ x P) ∧ [A$_3$] (x = $\dot{0}$) ⇒ real$_{posit}$ P (posit$_{real}$ x P) = x
Theorem 14: *Encoding and Decoding of* `maxpos` $\vdash_{thm}$ ∀(P:posit) (L:bool list) (x:real). [A$_1$] is_valid_posit (dest_posit P) ∧ [A$_2$] is_valid_posit_length P (posit$_{real}$ x P) ∧ [A$_3$] ~(zero_exception (posit$_{real}$ x P)) ∧ [A$_4$] x = maxpos P ⇒ real$_{posit}$ P (posit$_{real}$ x P) = x
Theorem 15: *For Values Greater than the Largest Negative Number* $\vdash_{thm}$ ∀(P:posit) (L:bool list) (x:real). [A$_1$] is_valid_posit (dest_posit P) ∧ [A$_2$] is_valid_posit_length P (posit$_{real}$ x P) ∧ [A$_3$] ~(zero_exception (posit$_{real}$ x P)) ∧ [A$_4$] ~(inf_exception (posit$_{real}$ x P)) ∧ [A$_5$] (x < $\dot{0}$) ∧ [A$_6$] (x ≥ −minpos P) ⇒ real$_{posit}$ P (posit$_{real}$ x P) = −minpos P
Theorem 16: *Completely Encoded fraction* $\vdash_{thm}$ ∀n x. [A$_1$] fraction_residue1 x n = &0 ⇒ x = $\dfrac{(\text{BV\_n (fraction\_list x n)})}{(\dot{2}^{\lvert\text{fraction\_list x n}\rvert})}$
Theorem 17: *Decoding of Encoded Positive Real Numbers* $\vdash_{thm}$ ∀(P:posit) (L:bool list) (x:real). [A$_1$] is_valid_posit (dest_posit P) ∧ [A$_2$] is_valid_posit_length P (posit$_{real}$ x P) ∧ [A$_3$] ~(zero_exception (posit$_{real}$ x P)) ∧ [A$_4$] ~(inf_exception (posit$_{real}$ x P)) ∧ [A$_5$] (x > $\dot{0}$) ∧ [A$_6$] (minpos P < |x| < maxpos P) ∧ [A$_7$] ~(checkmax (posit$_{real}$ x P)) ∧ [A$_8$] ~(cond$_e$ x P) ∧ [A$_9$] residue$_f$ x P = $\dot{0}$ ∧ [A$_{10}$] regime_length ([F]++((regime_bits x P)++((exp_list x P)++ (set_fraction_list x P)))) = |regime_bits x P| ∧ [A$_{11}$] value_of_k (posit$_{real}$ x P) = value_of_k ([sign_real x]++(regime_bits x P)) ⇒ real$_{posit}$ P (posit$_{real}$ x P) = x

The formal verification of the above theorems ensures the correctness of our formalization of posits, presented in Section 4.1, i.e., the formal model of posits and conversion from a real number to posit and vice versa, and its various parameters, such as the scaling factors of the regime, exponential and the fraction bits using HOL Light. Moreover, these formalization results, presented in Sections 4.1 and 4.2, can be further used for the verification of various arithmetic operations, such as addition, subtraction, multiplication and division operators.

5 Discussion

The distinguished feature of the proposed formalization is that all the proved theorems are of generic nature, i.e., all the functions and variables are universally quantified and hence, can be specialized based on the requirement of the Unum arithmetics, like the encoding or decoding of any particular Unums. Moreover, the inherent correctness of the theorem proving approach ensures that all the necessary assumptions are explicitly present with the respective theorem. The effort spent in verification of each theorem is represented in the form of proof lines and the man-hours as shown in Table 5. The man-hours are calculated based on two factors. The first factor includes the number of lines of HOL Light code per hour by a person with an average expertise and the second factor is the complexity of the proof. Moreover, there is no direct method to access the complexity of the proof. We often consider three major factors to estimate it. 1) The complexity of the mathematical results that are used in the analysis or proof of a theorem. For example, a proof involving integrals will be more complex than that involving matrices/vectors that are easy to handle.; 2) The expertise of a researcher regarding a particular proof goal.; 3) How many lemmas that are directly used in a proof of a theorem and are not available in a library. More lemmas to prove, make a formal proof of a theorem more complex and vice versa. Therefore, lines number of the proof script do not have a direct relationship with the man-hours. For instance, the man-hours for the verification of Theorems 13 and 14 are identical, while the proof lines for the former are less than that for the later. Moreover, there are few inherent limitations of our proposed higher-order-logic theorem proving approach. 1) Our proposed approach involves a lot of human interaction due to the undecidable nature of higher-order-logic, i.e., the user is involved in the process of formal verification along with the machine.; 2). Sometimes there is a significant gap between the traditional mathematical proof and its formal proof. Therefore, we need to identify the additional steps at our own that are required for developing a complete formal proof. ; 3) We have identified all formally verified properties, presented in Section 4.2, at our own to ensure the correctness of the formalization of posit provided in Section 4.1 of the paper. Moreover, to the best of our knowledge, these properties are not mentioned in the literature.

6 Conclusions

The Universal Number (Unum) is a number representation format that provides an improved memory bandwidth and the power efficiency as compared to the floating-point numbers [1]. As a first step towards the verification of the Unum arithmetic, this paper provides a formalization of posit, which is a Type III Unums. In particular, we provide a conversion of a real number to its corresponding posit representation and a posit representation to its corresponding real number. We also verify some important properties regarding scaling factors of its regime, exponential and fraction bits using HOL Light that are widely used to perform various arithmetic operations involving posits. In future, we plan to verify different arithmetic operations [65], such as addition, subtraction, multiplication, exponential and division for posits. We also plan to utilize our proposed formalization to formally verify the computations of the transcendental

Table 5: Verification Details for Each Theorem

Formalized Theorems	Proof Lines	Man-Hours	Complexity of Proofs
Theorem 1 (Table 3)	8	1	Easy
Theorem 2 (Table 3)	20	1	Easy
Theorem 3 (Table 3)	15	1	Easy
Theorem 4 (Table 3)	40	7	Easy
Theorem 5 (Table 3)	3	0.5	Easy
Theorem 6 (Table 3)	20	2	Easy
Theorem 7 (Table 3)	26	1	Easy
Theorem 8 (Table 3)	69	19	Medium
Theorem 9 (Table 3)	64	17	Medium
Theorem 10 (Table 3)	7	0.5	Easy
Theorem 11 (Table 3)	39	2	Easy
Theorem 12 (Table 3)	300	84	Hard
Theorem 13 (Table 4)	10	2	Easy
Theorem 14 (Table 4)	25	2	Easy
Theorem 15 (Table 4)	96	27	Hard
Theorem 16 (Table 4)	45	36	Hard
Theorem 17 (Table 4)	80	46	Hard

functions, such as sine, cosine and exponential functions [1]. Another future direction is to make a comparison of the formal libraries of the floating-point number and posits.

Data Availability Statement

The HOL Light code for our proposed formalization of posits is available at https://github.com/adrashid/positsverification and it has been added as reference 35 of the paper.

References

[1] Gustafson, J.L.: The End of Error: Unum Computing. CRC Press, ??? (2017)

[2] Gustafson, J.L., Yonemoto, I.T.: Beating floating point at its own game: Posit Arithmetic. Supercomputing Frontiers and Innovations **4**(2), 71–86 (2017)

[3] Esmaeel, A.A., Abed, S., Mohd, B.J., Fairouz, A.A., *et al.*: POSIT vs. Floating Point in Implementing IIR Notch Filter by Enhancing Radix-4 Modified Booth Multiplier. Electronics **11**(1), 163 (2022)

[4] Jaiswal, M.K., So, H.K.-H.: Architecture Generator for Type-3 Unum Posit Adder/Subtractor. In: Circuits and Systems, pp. 1–5 (2018). IEEE

[5] Podobas, A., Matsuoka, S.: Hardware Implementation of POSITs and their Application in FPGAs. In: Parallel and Distributed Processing, pp. 138–145 (2018). IEEE

[6] Jaiswal, M.K., So, H.K.-H.: Universal Number Posit Arithmetic Generator on FPGA. In: Design, Automation & Test in Europe, pp. 1159–1162 (2018). IEEE

[7] Lehóczky, Z., Retzler, A., Tóth, R., Szabó, Á., Farkas, B., Somogyi, K.: High-level.

NET Software Implementations of Unum Type I and Posit with Simultaneous FPGA Implementation using Hastlayer. In: Next Generation Arithmetic, pp. 1–7 (2018)

[8] Langroudi, H.F., Karia, V., Gustafson, J.L., Kudithipudi, D.: Adaptive Posit: Parameter Aware Numerical Format for Deep Learning Inference on the Edge. In: Computer Vision and Pattern Recognition, pp. 726–727 (2020)

[9] Murillo, R., Del Barrio, A.A., Botella, G.: Deep PeNSieve: A Deep Learning Framework based on the Posit Number System. Digital Signal Processing, 102762 (2020)

[10] Bentley, B.: Validating the Intel Pentium 4 Microprocessor. In: Design Automation, pp. 244–248 (2001)

[11] Clarke, E.M., Wing, J.M.: Formal Methods: State of the Art and Future Directions. ACM Computing Surveys **28**(4), 626–643 (1996)

[12] Harrison, J.: Handbook of Practical Logic and Automated Reasoning. Cambridge University Press, ??? (2009)

[13] Harrison, J.: Floating Point Verification in HOL Light: the Exponential Function. In: Algebraic Methodology and Software Technology, pp. 246–260 (1997). Springer

[14] Harrison, J.: A Machine-checked Theory of Floating Point Arithmetic. In: Theorem Proving in Higher Order Logics, pp. 113–130 (1999). Springer

[15] Daumas, M., Rideau, L., Théry, L.: A Generic Library for Floating-point Numbers and its Application to Exact Computing. In: Theorem Proving in Higher Order Logics. LNCS, vol. 2152, pp. 169–184 (2001). Springer

[16] Boldo, S., Filliâtre, J.-C.: Formal verification of floating-point programs. In: Symposium on Computer Arithmetic, pp. 187–194 (2007). IEEE

[17] Moore, J.S., Lynch, T.W., Kaufmann, M.: A Mechanically Checked Proof of the AMD5K86TM Floating-point Division Program. IEEE Transactions on Computers (9), 913–926 (1998)

[18] Russinoff, D.: A Mechanically Checked Proof of IEEE Compliance of a Register-transfer-level Specification of the AMD-K7 Floating-point Multiplication, Division, and Square Root Instructions. LMS Journal of Computation and Mathematics **1**, 148–200 (1998)

[19] Miner, P.S.: Defining the IEEE-854 Floating-Point Standard in PVS (1995)

[20] Berg, C., Jacobi, C.: Formal Verification of the VAMP Floating-point Unit. In: Correct Hardware Design and Verification Methods. LNCS, vol. 2144, pp. 325–339

(2001). Springer

[21] Jacobi, C.: Formal Verification of a Fully IEEE Compliant Floating Point Unit (2002)

[22] Cornea, M., Harrison, J., Anderson, C., Tang, P.T.P., Schneider, E., Gvozdev, E.: A Software Implementation of the IEEE 754R Decimal Floating-point Arithmetic using the Binary Encoding Format. Transactions on Computers **58**(2), 148–162 (2008)

[23] Harrison, J.: Formal Verification of Floating Point Trigonometric Functions. In: Formal Methods in Computer-aided Design, pp. 254–270 (2000). Springer

[24] Harrison, J.: Floating-point Verification using Theorem Proving. In: Formal Methods for the Design of Computer, Communication and Software Systems, pp. 211–242 (2006). Springer

[25] Harrison, J., Kubaska, T., Story, S., *et al.*: The Computation of Transcendental Functions on the IA-64 Architecture. In: Intel Technology Journal (1999). Citeseer

[26] Harrison, J.: Formal Verification of Square Root Algorithms. Formal Methods in System Design **22**(2), 143–153 (2003)

[27] Jones, R.B., O'Leary, J.W., Seger, C.-J., Aagaard, M.D., Melham, T.F.: Practical Formal Verification in Microprocessor Design. Design & Test of Computers **18**(4), 16–25 (2001)

[28] Narasimhan, N., Kaivola, R.: Formal Verification of the Pentium$^{\circledR}$ 4 Floating-Point Multiplier. In: Design, Automation & Test in Europe, pp. 1–8 (2002). IEEE

[29] Kaivola, R.: Intel CoreTM i7 Processor Execution Engine Validation in a Functional Language Based Formal Framework. In: Practical Aspects of Declarative Languages, pp. 414–429 (2011). Springer

[30] Slobodová, A.: Formal Verification of Hardware Support for Advanced Encryption Standard. In: Formal Methods in Computer-Aided Design, pp. 1–4 (2008). IEEE

[31] Whalen, M., Cofer, D., Miller, S., Krogh, B.H., Storm, W.: Integration of Formal Analysis into a Model-based Software Development Process. In: Formal Methods for Industrial Critical Systems, pp. 68–84 (2007). Springer

[32] Miller, S., Anderson, E., Wagner, L., Whalen, M., Heimdahl, M.: Formal Verification of Flight Critical Software. In: AIAA Guidance, Navigation, and Control Conference and Exhibit, p. 6431 (2005)

[33] Tribble, A.C., Lempia, D., Miller, S.P.: Software Safety Analysis of a Flight Guidance System. In: Digital Avionics Systems Conference, vol. 2, pp. 13–1131 (2002).

IEEE

[34] Tribble, A., Miller, S.: Safety Analysis of Software Intensive Systems. IEEE Aerospace and Electronic Systems **19**(10), 21–26 (2004)

[35] https://github.com/adrashid/posits_verification

[36] Harrison, J.: HOL Light: A Tutorial Introduction. In: Srivas, M., Camilleri, A. (eds.) Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96). Lecture Notes in Computer Science, vol. 1166, pp. 265–269. Springer, ??? (1996)

[37] Paulson, L.: ML for the Working Programmer. Cambridge University Press, ??? (1996)

[38] Harrison, J.: Formalized Mathematics. Technical Report 36, Turku Centre for Computer Science, Finland (1996)

[39] Jacobsen, C., Solovyev, A., Gopalakrishnan, G.: A Parameterized Floating-point Formalizaton in HOL Light. Electronic Notes in Theoretical Computer Science **317**, 101–107 (2015)

[40] Abdel-Hamid, A.T.: A hierarchical Verification of the IEEE-754 Table-driven Floating-point Exponential Function using HOL. PhD thesis, Concordia University (2001)

[41] Müller, S.M., Paul, W.J.: Computer Architecture: Complexity and Correctness. Springer, ??? (2013)

[42] Miner, P.S., Leathrum, J.F.: Verification of IEEE Compliant Subtractive Division Algorithms. In: Formal Methods in Computer-Aided Design. LNCS, vol. 1166, pp. 64–78 (1996). Springer

[43] Berg, C.: Formal Verification of an IEEE Floating Point Adder. Master's Thesis, Saarland University, Germany (2001)

[44] Harrison, J.: Towards self-verification of HOL Light. In: International Joint Conference on Automated Reasoning, pp. 177–191 (2006). Springer

[45] Kumar, R.: Self-compilation and Self-verification. Technical report, University of Cambridge, Computer Laboratory (2016)

[46] O'Leary, J., Zhao, X., Gerth, R., Seger, C.-J.H.: Formally Verifying IEEE Compliance of Floating-point Hardware. Intel Technology Journal **3**(1), 1–14 (1999)

[47] Akbarpour, B., Dekdouk, A., Tahar, S.: Formalization of Cadence SPW Fixed-Point Arithmetic in HOL. In: Integrated Formal Methods. LNCS, vol. 2335, pp.

185–204 (2002). Springer

[48] O'Leary, J.: Theorem Proving in Intel Hardware Design (2009)

[49] Gesellensetter, L., Glesner, S., Salecker, E.: Formal Verification with Isabelle/HOL in Practice: Finding a Bug in the GCC Scheduler. In: Formal Methods for Industrial Critical Systems, pp. 85–100 (2007). Springer

[50] Johnson, C.W.: The Natural History of Bugs: Using Formal Methods to Analyse Software Related Failures in Space Missions. In: Formal Methods, pp. 9–25 (2005). Springer

[51] Fitzgerald, J., Bicarregui, J., Larsen, P.G., Woodcock, J.: Industrial Deployment of Formal Methods: Trends and Challenges. In: Industrial Deployment of System Engineering Methods, pp. 123–143. Springer, ??? (2013)

[52] Zhang, F., Niu, W., et al.: A Survey on Formal Specification and Verification of System-level Achievements in Industrial Circles. Academic Journal of Computing & Information Science **2**(1) (2019)

[53] https://shemesh.larc.nasa.gov/fm/fm-main-research.html

[54] https://shemesh.larc.nasa.gov/fm/fm-collins-intro.html

[55] Barnat, J., Beran, J., Brim, L., Kratochvíla, T., Ročkai, P.: Tool Chain to Support Automated Formal Verification of Avionics Simulink Designs. In: Formal Methods for Industrial Critical Systems, pp. 78–92 (2012). Springer

[56] Cao, Z., Lv, W., Huang, Y., Shi, J., Li, Q.: Formal Analysis and Verification of Airborne Software Based on DO-333. Electronics **9**(2), 327 (2020)

[57] Nellen, J., Rambow, T., Waez, M.T.B., Ábrahám, E., Katoen, J.-P.: Formal Verification of Automotive Simulink Controller Models: Empirical Technical Challenges, Evaluation and Recommendations. In: Formal Methods, pp. 382–398 (2018). Springer

[58] Xu, H., Wang, P.: Real-time Reliability Verification for UAV Flight Control System Supporting Airworthiness Certification. PloS ONE **11**(12), 0167168 (2016)

[59] Cofer, D.: Formal Methods in the Aerospace Industry: Follow the Money. In: Formal Engineering Methods, pp. 2–3 (2012). Springer

[60] Chaves, L., Bessa, I.V., Ismail, H., Santos Frutuoso, A.B., Cordeiro, L., Lima Filho, E.B.: DSVerifier-aided Verification Applied to Attitude Control Software in Unmanned Aerial Vehicles. Transactions on Reliability **67**(4), 1420–1441 (2018)

[61] Wiels, V., Delmas, R., Doose, D., Garoche, P.-L., Cazin, J., Durrieu, G.: Formal Verification of Critical Aerospace Software. AerospaceLab (4), 1 (2012)

[62] Jaiswal, M.K., So, H.K.-H.: Pacogen: A Hardware Posit Arithmetic Core Generator. ACCESS **7**, 74586–74601 (2019)

[63] Langroudi, S.H.F., Pandit, T., Kudithipudi, D.: Deep Learning Inference on Embedded Devices: Fixed-point Vs Posit. In: Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications, pp. 19–23 (2018). IEEE

[64] Gustafson, J.L.: Posit Arithmetic. Mathematica Notebook Describing the Posit Number System **30** (2017)

[65] Chung, S.Y.: Provably Correct Posit Arithmetic with Fixed-point Big Integer. In: Next Generation Arithmetic, pp. 1–10 (2018)