Modeling SystemC Fixed-Point Arithmetic in HOL

Behzad Akbarpour and Sofiène Tahar

Dept. of Electrical & Computer Engineering, Concordia University
1455 de Maisonneuve W., Montreal, Quebec, H3H 1M8, Canada
{behzad,tahar}@ece.concordia.ca

Abstract. SystemC is a new C-based system level design language whose ultimate objective is to enable System-on-a-Chip (SoC) design and verification. Fixed-point design based on the SystemC data types is rapidly becoming the standard for optimizing DSP systems. In this paper, we propose to create a formalization of SystemC fixed-point arithmetic in the HOL theorem proving environment. The SystemC fixed-point number representation which contains a new generalized format and different rounding and overflow modes is described, and then it is formalized in higher-order logic. This formalization is then compared with the formalization of IEEE standard based floating-point arithmetic in HOL. A set of theorems are proved to bound the error in fixed-point rounding and to verify the fixed-point arithmetic operations against their abstract mathematical counterparts. Finally, we show by an example how this formalization can be used in verification of the translation from floating-point and fixed-point algorithmic, down to register transfer and netlist gate levels in the design flow of SoC systems.

1 Introduction

High complexity of modern digital signal processing systems versus increasing demand for a short time-to-market are current challenges of today's VLSI designers. With improvements in silicon technology and the increase in the number of logic gates that can be implemented on a single chip, various functionalities such as memories, logic gates, analogue blocks, CPU and digital signal processing (DSP) cores can be integrated into a single silicon chip. These functionalities are implemented by using System-on-a-Chip (SoC) [7] solutions that generally integrate diverse hardware and software. On the other hand, the use of inexpensive, high speed, and low power DSPs is on the rise. For DSP the problem is to decide whether a fixed-point or a floating-point math unit should be used [17]. Several factors should be taken into account in this regard. An important first step is to gain an understanding of how the hardware representations differ and how they affect precision and range. Also needed is a grasp of the types of applications to which particular chips are best suited and which hardware vendors provide these chips. Performance is also a driving factor behind the use of DSPs for which cost, speed, and power consumption are key ingredients.

The final consideration is the availability of development tools and the programming paradigms they support. Recently, significant effort has gone into building high level languages for both fixed- and floating-point DSPs. The most popular language has been C. Since C has a built-in type for floating-point, this is an attractive solution for those chips. The standard ANSI C language, however, does not support fixed-point data types, thus forcing programmers to write in assembly language and to deal with complicated and error-prone scaling issues. A significant breakthrough to allow a systematic approach for fixed-point design has been achieved by the Open SystemC Initiative. Fixed-point design based on the SystemC [31] data types is rapidly becoming the standard for optimizing DSP systems, and Electronic Design Automation (EDA) tools supporting this design flow are available today.

With ever increasing complexity of the design of digital systems the role of design verification has gained a lot of importance. Design errors can cause serious failures, resulting in the loss of time and money. It takes a very large amount of time and effort to correct the error, especially when the error is discovered late in the process. For these reasons, we need approaches that enable us to discover errors and validate designs as early as possible. Verification is defined as the validation of the circuit for its correctness. The verification of floating-point hardware has always been an important part of processor verification. The importance of arithmetic circuit verification was illustrated by the famous floating-point division bug in Intel's Pentium processor [18]. Floating-point algorithms are usually very complicated. They are composed of many modules where the smallest flaw in the design or the implementation can cause a very hard-to-discover bug, as occurred in Intel's case. Traditional approaches for verifying floating-point circuits are based on simulation. However, these approaches cannot exhaustively cover the input space of the circuits. Therefore, new methods are needed for the economical and reliable verification of digital systems. Formal verification [19] have recently paved a path, showing the utility of finding bugs early in the design cycle. Formal verification techniques are usually classified in two categories: interactive theorem proving and automatic decision diagram based model checking and equivalence checking. Theorem proving consists in expressing the specification and implementation in a formal logic. Their relationship, stated as equivalence or implication, is regarded as a theorem to be proven within the logic system, using axioms and inference rules. Powerful mathematical techniques such as induction and abstraction are strengths of theorem proving and make it a very flexible verification technique. In model checking, one checks if the design satisfies some properties (formal specification). With equivalence checking, we check if two designs exhibit the same behavior. The latter techniques have been successfully applied to real industrial designs. However, since most of the tools are based on Binary Decision Diagrams (BDDs), they require the design to be described at the Boolean level. In practice, they often fail to verify a large-scale design because of the so-called state space explosion.

There exist several related works in the open literature on the formalization and verification of floating-point arithmetic. For instance, Barett [3] specified parts of the IEEE-754 [15] standard in Z, and Miner [25] formalized the IEEE-854 [16] floating-point standard in PVS. Carreno [6] formalized the same IEEE-854 standard in HOL. Harrison [12] defined and formalized real numbers using HOL. He then developed a generic floating-point library [14] to define and verify the most fundamental terms and lemmas of the IEEE-754 standard. This former library was used by him to formalize and verify floating-point algorithms such as the square root and the exponential function [13] against their behavioral specification.

Moore et al. [26] have verified the AMD-K5 floating-point division algorithm using the ACL2 theorem prover. Also, Russinoff [28] has developed a library for ACL2 prover and applied it successfully to verify the K5 square root, and the Athlon multiplication, division, square root, and addition algorithms. Daumas et al. [10] have presented a generic library for reasoning about floating-point numbers within the Coq system. Berg et al. [4] have formally verified a theory of IEEE rounding presented in [27] using the theorem prover PVS, and then used the theory to prove the correctness of a fully IEEE compliant floating-point unit used in the VAMP processor.

Aagaard and Seger [1] combined BDD based methods and theorem proving techniques to verify a floating-point multiplier. Chen and Bryant [9] used word-level SMV to verify a floating-point adder. Miner and Leathrum [24] verified a general class of subtractive division algorithms with respect to the IEEE-754 standard in PVS. Leeser et al. [20] verified a radix-2 square root algorithm and its hardware implementation using theorem proving methods. Cornea-Hasegan [8] used iterative approaches and mathematical proofs to verify the correctness of the IEEE floating-point square root, division and remainder algorithms. O'Leary et al. [21] reported on the verification of the Intel's floating-point unit at the gate level using a combination of model-checking and theorem proving.

While the above works are concerned with floating-point representation and arithmetic, in [2] we proposed the first machine-checked formal development on properties of fixed-point arithmetic according to Cadence SPW (Signal Processing WorkSystem) tool. Unlike floating-point arithmetic which is standardized in IEEE-754 [15] and IEEE-854 [16], current fixed-point arithmetic does not follow any particular standard and depends on the tool and the language used to design the DSP chip. Based on higher-order logic, we proposed to encode a fixed-point number by a pair composed of a boolean word, and a triple indicating the word length, the length of the integer portion and the sign format. Then, we formalized the concepts of valuation and rounding as functions that convert respectively a fixed-point number to a real number and vice versa, taking into account different rounding and overflow modes. Fixed-point arithmetic operations are formalized as functions performing operations on the real numbers corresponding to the fixed-point operands and then applying the rounding on the real number result. We supported three kinds of exceptions, two overflow modes and five rounding modes as described in SPW documentation. Finally, we proved different lemmas regarding the error analysis of the fixed-point quantization and correctness of the basic operations like addition, multiplication, and division. The formalization of the fixed-point arithmetic has been inspired mostly by the work done by Harrison [13]. Indeed we followed similar steps as in formalization of floating-point arithmetic for modeling fixed-point arithmetic, and used an analogous set of lemmas to his work to check the validity of operation results and to carry out the error analysis of the fixed-point rounding.

In this paper, we significantly extend this work to the SystemC fixed point description. In comparison to SPW, SystemC represents the numbers in a different more comprehensive format. SystemC also covers a more complete set of overflow, rounding, and exception handling parameters. SystemC supports seven rounding modes, of which four correspond exactly to the rounding modes of SPW. The other three modes are specific to SystemC and are not supported by the other tools. SystemC supports five overflow modes covering those of SPW. These features motivated EDA companies, including Cadence, to adapt SystemC for fixed-point design and verification[1]. In the new fixed-point theory, we have included the parameters representing the overflow, rounding mode, and the number of saturation bits which have been introduced in SPW theory in the definition of arithmetic operations, directly in the format to make a generalized SystemC fixed-point attributes. Also new enumerated data types are defined to cover the SystemC rounding and overflow modes. Specific functions are then defined to handle the overflow in SystemC wrap around modes. Finally new theorems are proved to bound the error in SystemC special rounding modes. The modularity of SPW theory has facilitated the extension process. This is of great importance since the design of modular and reusable theories remains a big challenge in the theorem proving era.

The organization of this paper is as follows: Section 2 describes the SystemC fixed-point arithmetic including the format of the fixed-point numbers, and overflow and quantization modes. Section 3 describes in detail their formalization in HOL in parallel with the formalization of IEEE-754 based floating-point arithmetic in HOL. In Section 4, we discuss the rounding error analysis and the verification of the SystemC fixed-point arithmetic operations. Section 5 presents an illustrative example on how this formalization can be used through the modeling and verification of a Notch filter algorithm. Finally, Section 6 concludes the paper.

2 Fixed-Point Types in SystemC

In this section we describe SystemC based fixed-point arithmetic. SystemC is a C++ based modeling platform supporting design abstractions at the register-transfer, behavioral, and system levels. Consisting of a class library and a simulation kernel, the language is an attempt at standardization of a C/C++ design methodology, and is supported by the Open SystemC Initiative (OSCI), a consortium of a wide range of system houses, semiconductor companies, intellectual property (IP) providers, embedded software developers, and design automation

[1] In fact the latest release of the Cadence SPW tool supports both the old SPW fixed-point arithmetic as well as the SystemC one.

tool vendors. The advantages of SystemC include the establishment of a common design environment consisting of C++ libraries, models and tools, thereby setting up a foundation for hardware-software co-design; the ability to exchange IP easily and efficiently; and the ability to reuse test benches across different levels of modeling abstraction. An important element of SystemC is the support for fixed-point data-types, which is essential for the refinement of complex algorithms to a hardware or software implementation.

The SystemC fixed-point library contains basic types for both unconstrained, constrained, signed and unsigned fixed-point data types [30]. Constrained data types use static arguments to specify the functionality of the type while unconstrained data types can use argument types that are nonstatic. Static arguments must be known at compile time, while nonstatic arguments can be variables. In addition to the standard fixed-point types which use arbitrary precision in calculations, SystemC also provides limited precision fixed-point types to speed simulation when limited precision is all that is required. With standard fixed-point types the mantissa can be virtually any size. With limited precision fixed-point types the mantissa is limited to 53 bits. Limited precision fixed-point types are implemented with double precision floating-point values. The fixed-point format used by the fixed-point data types consists of the following parameters:

- *wl:* Total word length, used for fixed-point representation. Equivalent to the total number of bits used in the type. Word length must be greater than 0.
- *iwl:* Integer word length, specifies the number of bits that are to the left of the binary point (.) in a fixed-point number. Integer word length can be positive or negative, and larger than the word length. If this number is negative, repeated leading sign bits or zeros are added to the object. If this number is greater than the total number of bits, trailing zeros are added to generate the equivalent binary value.
- *q_mode:* Quantization mode, determines the behavior of the fixed point type when the result of an operation generates more precision in the least significant bits (LSB) than is available as specified by the word length and integer word length parameters.
- *o_mode:* Overflow mode, determines what happens when the result of an operation generates more bits on the most significant bits (MSB) side than are available for representation.
- *n_bits:* Number of saturated bits, only used for overflow mode and specifies how many bits will be saturated if a saturation behavior is specified and an overflow occurs.

In comparison with the fixed-point format defined in SPW [29], the parameters *wl* and *iw* in SystemC correspond to parameters *#bits* and *#integer_bits* in SPW fixed-point attributes. The parameters *q_mode* and *o_mode* which have been used in SPW during the definition of arithmetic operations, are inserted directly in the format to make a generalized fixed-point attributes for SystemC. Also the argument *n_bits* is not used by SPW and is specific to SystemC. In SPW the type of the fixed-point numbers as *signed* or *unsigned* are defined by the parameter *sign_format* in the attributes; however, in SystemC there is not such a parameter in the format and separate types are defined for signed and unsigned fixed-point numbers.

Operations performed on fixed-point data types are done using arbitrary and full precision. After the operation is complete, the resulting operand is cast to fit the fixed-point data type object. The casting operation applies the quantization behavior of the target object to the new value and assigns the new value to the target object. Then, the appropriate overflow behavior is applied to the result of the process which gives the final value.

Quantization effects are used to determine what happens to the LSBs (Least Significant Bits) of a fixed-point type when more bits of precision are required than are available. The quantization modes available in SystemC are shown in Table 1:

Table 1. SystemC Quantization Modes

Quantization Mode	**Name**
Rounding to plus infinity	SC_RND
Rounding to zero	SC_RND_ZERO
Rounding to minus infinity	SC_RND_MIN_INF
Rounding to infinity	SC_RND_INF
Convergent Rounding	SC_RND_CONV
Truncation	SC_TRN
Truncation to zero	SC_TRN_ZERO

Figure 1 shows the behavior of each quantization mode. The diagonal line represents the ideal number representation given infinite bits. The small horizontal lines show the effect of the rounding. Any value of the X axis within the range of the line will be converted to the value of the Y axis. The symbol q in the figure refers to the quantization step, that is, the resolution of the data type. As shown in this figure modes *SC_RND*, *SC_RND_ZERO*, *SC_RND_MIN_INF*, *SC_RND_INF*, and *SC_RND_CONV* will round the value to the closest representable number if the two nearest representable numbers are not an equal distance apart. Otherwise, rounding towards plus infinity, to zero, towards minus infinity, towards plus infinity if positive or minus infinity if negative, and towards nearest even will be performed respectively (Figure 1 (a-e)). *SC_TRN* mode is the default for fixed-point types and will be used if no other value is specified. The result is always rounded towards minus infinity (Figure 1 (f)). In other words, the result value is the first representable number lower than the original value. Finally, for *SC_TRN_ZERO* the result is the nearest representable value towards zero (Figure 1 (g)). Rounding modes *SC_RND*, *SC_RND_CONV*, *SC_TRN*, and *SC_TRN_ZERO* in SystemC correspond exactly to *Round*, *Convergent_Round*, *Truncate*, and *Round_To_Zero* loss of precision modes in SPW, respectively. The other three rounding modes are specific to SystemC and are not supported by SPW.

In addition to quantization modes, we can use overflow modes to approximate a higher range for fixed-point operations. Usually, overflow occurs when the result

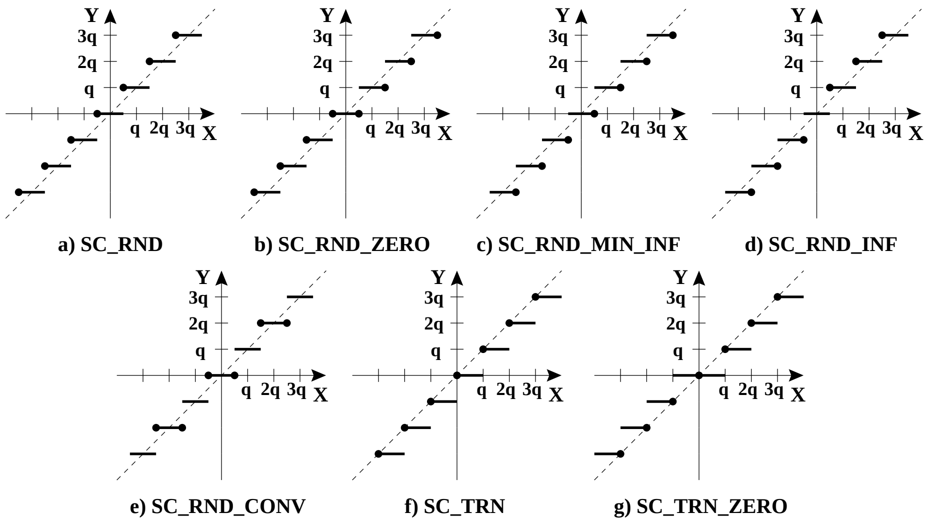


Fig. 1. The Behavior of SystemC Quantization Modes

of an operation is too large or too small for the available bit range. Specific overflow modes can then be implemented to reduce the loss of data. Overflow modes are specified by the *o_mode* and *n_bits* parameters to a fixed point type. The supported overflow modes are listed in the Table 2.

Table 2. SystemC Overflow Modes

Overflow Mode	**Name**
Saturation	SC_SAT
Saturation to zero	SC_SAT_ZERO
Symmetrical Saturation	SC_SAT_SYM
Wrap-around	SC_WRAP
Sign magnitude wrap-around	SC_WRAP_SM

Figure 2 shows the behavior of each overflow mode for a 3 bit type. The diagonal line represents the ideal value if infinite bits are available for representation. The dots represent the values of the result. The X axis is the original value and the Y axis is the result. From this figure it can be seen that $MAX = 3$ and $MIN = -4$ for a 3 bit type. *SC_SAT* mode will convert the specified value to *MAX* for an overflow or *MIN* for an underflow condition (Figure 2 (a)). *SC_SAT_ZERO* mode will set the result to 0 for any input value that is outside the representable range of the fixed point type. If the result value is greater than *MAX* or smaller than *MIN* the result will be 0 (Figure 2 (b)). In *SC_SAT_SYM* mode, positive overflow will generate *MAX* and negative overflow will generate $-MAX$ for signed numbers or *MIN* for unsigned numbers (Figure 2 (c)). With *SC_WRAP* mode the value of an arithmetic operand will wrap around from *MAX* to *MIN* as *MAX* is reached. There are two different cases within this

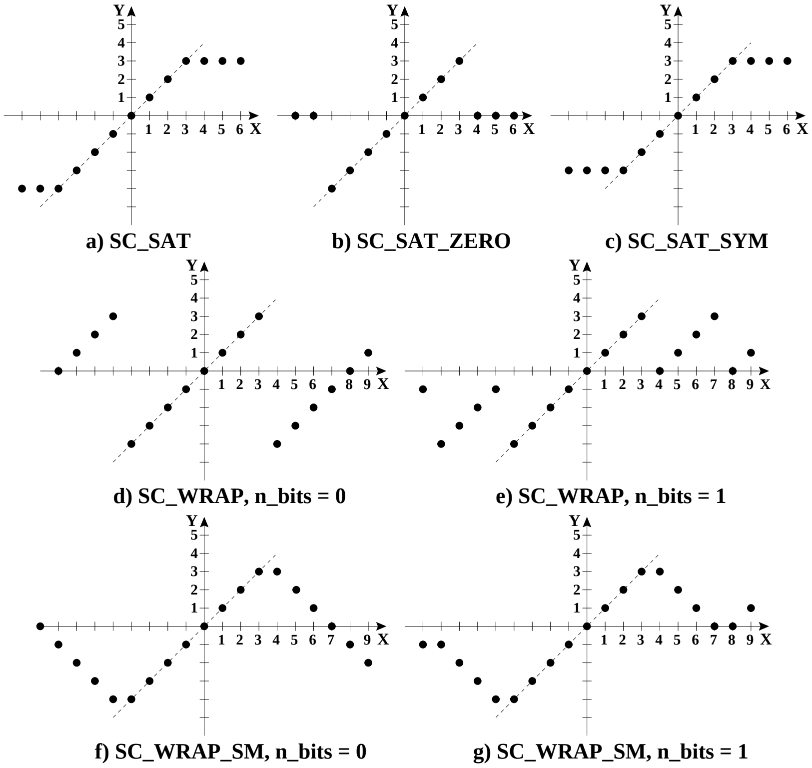


Fig. 2. The Behavior of SystemC Overflow Modes

mode. The first is with the *n_bits* parameter set to 0 or having a default value of 0. All bits except for the deleted bits are copied to the result number (Figure 2 (d)). The second is when the *n_bits* parameter is a nonzero value. In this case the specified number of most significant bits of the result number are saturated with preservation of the original sign, the other bits are simply copied. Positive numbers remain positive and negative numbers remain negative. A graph showing this behavior with $n\_bits = 1$ is shown in Figure 2 (e). Notice that positive numbers wrap around to 0 while negative values wrap around to -1. The *SC_WRAP_SM* overflow mode uses sign magnitude wrapping. This overflow mode behaves in two different styles depending on the value of parameter *n_bits*. When *n_bits* is 0 no bits are saturated. This mode will first delete any MSB bits that are outside the result word length. The sign bit of the result is set to the value of the least significant deleted bit. If the most significant remaining bit is different from the original MSB then all the remaining bits are inverted. If MSBs are the same, the other bits are copied from the original value to the result value. A graph showing the result of this overflow mode is shown in Figure 2 (f). As the value of X increases, the value of Y increases to MAX and then slowly starts to decrease until MIN is reached. The result is a sawtooth like waveform. With *n_bits* greater than 0, *n_bits MSB* bits are saturated to 1. A graph showing this behavior with $n\_bits = 1$ is shown in Figure 2 (g). Notice that while the graph looks somewhat like a sawtooth waveform, positive numbers do not dip below 0 and negative numbers do not cross -1. Overflow modes *SC_SAT* and *SC_Wrap* in SystemC cover the two overflow modes *Clip* and *Wrap* in SPW. The other three overflow modes are not supported by SPW and are specific to SystemC.

3 Modeling SystemC Fixed-Point Arithmetic in HOL

In this section, we present the formalization of SystemC based fixed-point arithmetic in higher-order logic, based on the general purpose HOL theorem prover [11]. HOL's basic types include the natural numbers and booleans. It also includes other specific extensions like John Harrison's reals library [12] which proved to be essential for our fixed-point arithmetic formalization.

Fixed point numbers are modeled in HOL as a pair of elements composed of a bit string (*string*) and a set of attributes (*attrib*). The bit string is represented by a boolean word and the set of attributes is itself a combination of six elements representing the word length (*wordlength*), integer word length (*integerwordlength*), sign type (*signtype*), rounding mode (*roundmode*), overflow mode (*overflowmode*), and the number of saturation bits (*satbits*), respectively. In comparison to the SPW formalization we have included three extra parameters to define a generalized fixed-point format. The fixed-point numbers are then partitioned using special predicates into signed (*is_signed*) and un-signed (*is_unsigned*) numbers. The validity of a fixed-point number (*is_valid*) and a set of attributes (*validAttr*) is defined using special predicates. In a valid set of attributes the word length is in the range of 1 and 53 corresponding to fast fixed-point data types, in comparison to 256 in SPW. Also, the sign type in a valid set of attributes is either 0 or 1, and the number of saturation bits is less than the word length. The fixed-point data types are defined in bijection with the appropriate subset of $boolword \times \mathbb{N}^3 \times roundingmode \times overflowmode \times \mathbb{N}$ using functions *Fxp* and *deFxp*. Then, we defined the valuation function (*value*) to specify a real value to fixed-point numbers using separate formulas for signed and unsigned numbers. The constants for the smallest (*bottomfxp*) and largest (*topfxp*) fixed-point numbers for a given format together with their corresponding real values (*MIN,MAX*) are also defined using specific functions. Then, we defined enumerated data types for seven rounding modes and five overflow modes in SystemC fixed-point arithmetic. The rounding function (*fxp_round*) is then defined case by case on the rounding modes and special functions are defined to handle the overflow in Wrap-around (*WRAP*) and Sign magnitude wrap-around (*WRAP_SM*) modes. Then, we defined the operations on fixed-point numbers (*fxpAdd,fxpSub,fxpMul,fxpDiv*) which are performed using the arbitrary precision in real domain and then the result is casted to the output format.

Our effort in formalization of fixed-point arithmetic can be compared to the formalization of IEEE standard based floating-point arithmetic in HOL [13] which is performed as in the following steps:

– **Floating Point Numbers:** A floating-point number is modeled as a triple of natural numbers interpreted as a sign, an exponent, and a fraction. The exponent is usually added to a constant (bias) to make the biased exponent's range nonnegative. The floating-point numbers are partitioned into not a numbers (NaN), infinities, normalized numbers, denormalized but nonzero numbers, and zeros as specified in IEEE-754 standard. Predicates for testing the validity and finiteness of a triple for a given format are defined. Also extractors for the three fields of a floating-point number together with constants for convenient values such as largest representable positive number and the most negative number in a format are defined.
– **Format Parameters:** The floating-point format for single, double precision, and extended numbers is defined as a pair of two natural numbers representing the width in bits of the exponent field, and the width in bits of the significand field. From these parameters three other characteristic numbers are defined for the total word length, the maximum exponent value, and the bias in the exponent.
– **Representation and Valuation:** The next step in formalization of floating-point numbers is the definition of the concrete representation of the numbers as the fields are laid out with the sign as the most significant bit, the exponent in the middle and the fraction in the bottom. Then, a real value is specified to non exceptional numbers. The valuation is meaningless when applied to infinities and NaNs. The denormalized numbers and normalized numbers are treated separately. Then, a few significant real values such as the real value of the largest representable number, the overflow threshold, and the notion of the unit in the last place for a given floating-point number are defined.
– **Rounding:** The definition of the valuation function is fundamental of the definition of the inverse operation of rounding which coerces a real number into a given floating-point format. The rounding is controlled by a rounding mode, specifying whether a real number is to be mapped to the nearest floating-point number (using round to even to choose a unique number if necessary), towards zero, or towards positive or negative infinity. The modes are represented in HOL via an enumerated type definition.
– **Arithmetic Operations:** Then, the arithmetic operations are defined where they first deal with the exceptional cases, either where the arguments involve a NaN or infinity, or are invalid for other reasons (e.g. $\infty - \infty$) and generate a NaN. Apart from that, they basically just take the values of the arguments, perform the mathematical operations and then round the result according to the desired rounding mode.
– **Float and Double Types:** Finally, the above considerations are specified to actual HOL type of single precision and double precision numbers called *float* and *double*. These types are defined to be in bijection with the appropriate subset of $\mathbb{N}^3$, with the bijections written in HOL as *float* and *defloat*. The operations are defined by mapping out of the type, performing the operations, and mapping back.

4 Verification of SystemC Arithmetic Operations

The correctness of fixed-point operations can be specified by comparing the operation's output with the true mathematical result. Since the operations are defined as if they first performed using infinite precision and then the result is rounded to fit in the destination format, the verification of operations is closely related to bounding the error in rounding function. On the other hand, the analysis of error in fixed-point rounding is very similar to the error analysis in floating-point rounding. In the following discussion, we first explain the details of floating-point rounding error analysis and then describe how similar steps are followed and analogous theorems are proved to bound the error in fixed-point rounding and to verify the fixed-point arithmetic operations.

4.1 Floating-Point Verification

The steps in analysis of floating-point rounding error in HOL [13] are as follows[2]:

- **Lemmas for Analyzing the Rounding Operation:** In the first step, prove some lemmas about the properties of the approximating a real number with a floating-point number. First, prove a theorem that ensures the existence of the best approximation to a given real number in a finite non empty set of floating-point numbers. Then, prove that the chosen best approximation to a real number satisfying a property p from a finite and non empty set of floating-point numbers is unique and is itself a member of the set and is itself the best approximation of the real number. Then, prove that the set of all valid and finite floating-point numbers are finite and non empty. Then, prove that the chosen best approximation to a real number satisfying a property p from the set of all finite floating-point numbers is a finite and valid floating-point number. Finally, prove that the result of rounding a real number to a floating-point number is valid.
- **Preliminary Theorems about Rounding Error:** In the second step, define the error as the difference between a real number and the value of its rounding result for rounding to nearest even. Then, prove that if the absolute value of a real number is less than the threshold value of a given floating-point format, then the rounding result is the nearest value to the real number and the corresponding error is minimum comparing to the other floating-point numbers. Also, if for a given real number we can find a floating-point number with equal value then the rounding-error is zero.
- **General Error Bound Theorems:** Next, prove two main theorems quantifying the error. In the first theorem, prove that if the absolute value of a real number is in the representable range of normalized floating-point numbers and located in the j'th binade, i.e. its absolute value is less than $2^{j+1}/2^{126}$

[2] This analysis is performed using the HOL Light theorem prover which is an older version of the tool. The code is recently ported by the first author from HOL Light to HOL4 which is the latest version of HOL tool.

and greater than or equal to $2^j/2^{126}$, then the absolute value of error is less than or equal to $2^j/2^{150}$. In the second theorem, prove that if the real number is in the denormal range, i.e. its absolute value is less than 2^{-126}, then the error is less than or equal to 2^{-150}. To prove these main theorems, a set of eight lemmas and four theorems about general rounding error are established.

The error bounding theorems can be explained as follows. The single precision format in IEEE standard for binary floating-point numbers is 32 bits wide and has an 8 bit exponent field with the exponent bias of 127 and has a 23 bit significand considering the hidden bit which is always 1. The single precision floating-point numbers are distributed on the real axis as shown in Figure 3. Figure 3(a) shows the number distribution pattern and the various subranges in this format. Figure 3(b) illustrates the relative magnitudes of normalized and denormalized numbers. In the context of numbers of a specific precision, it is useful to speak of rounding in terms of units in the last place (*ulp*). A *ulp* is naturally understood as the magnitude of the least significant digit, or in the other words, the distance between the floating point number a and the next floating point number of greater magnitude. For example, one *ulp* of the denormalized region in the single precision format of IEEE standard is 2^{-149}, and one *ulp* for the j'th binade in the normalized region is equal to $2^{j+1}/2^{150}$ as shown in Figure 3(b). The rounding error can be easily bounded in term of *ulp*s. For rounding to nearest the absolute value of error is less than or equal to half a *ulp*. This means that the absolute value of error is less than or equal to 2^{-150} for denormalized region, and less than or equal to $2^j/2^{150}$ for the j'th binade in the normalizerd region as stated in the last two main theorems mentioned before.

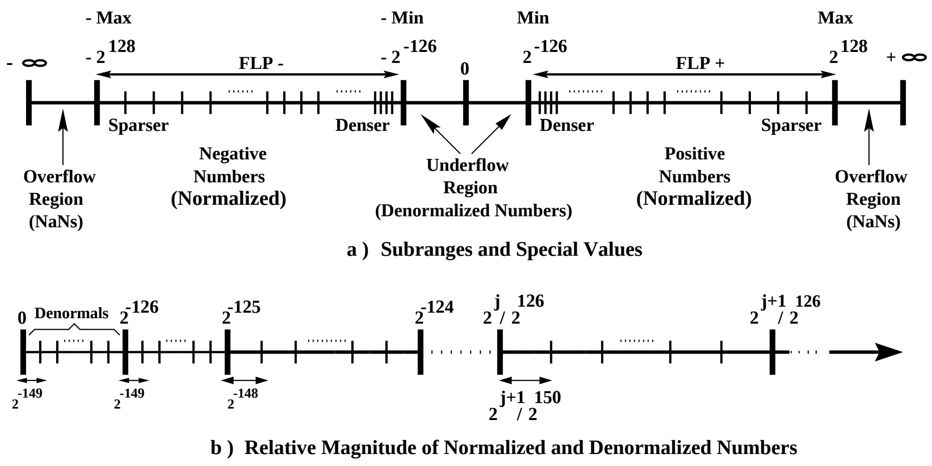


Fig. 3. Single Precision Floating-Point Numbers on the Real Number Line

– **Rounding Error in Arithmetic Operations:** At the end, prove theorems that relate the arithmetic operations such as addition, subtraction, multiplication, division, reminder, square root, negation and absolute value to their abstract mathematical counterparts according to the corresponding errors. The theorems are composed of two parts. In the first part which is about the finiteness of the floating-point operation output prove that for each pair of finite floating-point numbers, if the real result is less than the overflow threshold value then the output result is also finite. In the second part of the theorems, prove that the value of the floating-point result is equal to the value of the real result plus an error which is already quantified using the previous error bound theorems.

4.2 Fixed-Point Verification

Similar steps are followed for the error analysis of fixed-point rounding:

– **Lemmas for Analyzing the Fixed-Point Rounding Operation:** We first proved lemmas concerning with the approximation of a real number with a fixed-point number. We proved (*FXP_IS_CLOSEST_EXISTS*) that in a finite nonempty set of fixed-point numbers we can find the best approximation to a real number based on a given valuation function. Then, we proved that the chosen best approximation to a real number satisfying a property p from a finite and non empty set of fixed-point numbers is unique (*FXP_CLOSEST_IS_EVERYTHING*) and is itself a member of the set (*FXP_CLOSEST_IN_SET*) and is itself the best approximation of the real number (*FXP_CLOSEST_IS_CLOSEST*). Finally, we proved (*FXP_IS_VALID_CLOSEST*) that the chosen best approximation to a real number satisfying a property p from the set of all valid fixed-point numbers with a given attributes is itself a valid fixed-point number. Since in the definition of fixed-point rounding we have used the same approximating functions (***is_closest, closest***) as in floating-point case, the proof of these theorems are very close to their corresponding floating-point lemmas. Then, we proved that the set of all valid fixed-point numbers with a given attributes is finite (*FINITE_VALID_ATTRIB*). We also proved (*FXP_IS_VALID_NONEMPTY*) that the set of all valid fixed-point numbers is nonempty. The proof of the first lemma is a bit complicated. For this purpose we made use of some built-in theorems about the finite sets in HOL ***pred_sets*** library [22]. Among these are the two fundamental theorems *FINITE_EMPTY* and *FINITE_INSERT*, which state that the empty set is indeed finite and the insertion of an element to a finite set constructs a finite set. Other theorems state that the union of two finite sets (*FINITE_UNION*), the image of a function on a finite set (*IMAGE_FINITE*), a singleton set[3] (*FINITE_SING*), the cross combination of two finite sets (*FINITE_CROSS*), and any subset of a finite set (*SUBSET_FINITE*) is itself a finite set. Using these theorems together with the definition of a valid fixed-point number helped us

[3] a set that contains precisely one element

to break down the proof of the finiteness of all valid fixed-point numbers to the proof of finiteness of the set of all boolean words with a given word length (*WORD_FINITE*) and the set of all natural numbers less than a given value (*FINITE_COUNT*). The last theorems are proved by induction on the word length of the boolean word and the maximum limit of the natural numbers, respectively. For SystemC fixed-point, we also need to prove that the set of all elements of type *roundmode* and *overflowmode* are finite (*FINITE_ROUNDMODE*, *FINITE_OVERFLOWMODE*). This is obvious since these sets contain only seven and five elements, respectively. Finally, we proved (*FXP_IS_VALID_ROUND*) that the result of rounding a real number which is in the range representable by a given valid attributes is a valid fixed-point number.

- **Rounding Error in Fixed-Point Arithmetic Operations:** Then, we defined the error resulting from rounding a real number to a fixed-point value (*fxperror*). Then, we established the first main theorems (*FXP_ADD_THM, FXP_SUB_THM, FXP_MUL_THM, FXP_DIV_THM*) on the correctness of fixed-point arithmetic operations. According to these theorems, if the input fixed-point operands and the output attributes are valid then the result of fixed-point operations is valid. Also the result of the operations is related to the real result considering the error.
- **General Fixed-Point Error Bound Theorem:** In the next step, we established the second main theorem on fixed-point rounding error analysis which concerns bounding the error. The error is absolutely quantified as in the theorem *FXP_ERROR_BOUND_THM*. According to this theorem, the error in rounding a real number which is in the range representable by a given set of attributes X is less than the quantity $1/2^{fracbits(X)}$. To explain the theorem, we consider the following fact which relates the definition of the fixed-point numbers to the rationals. An N-bit binary word, when interpreted as an unsigned fixed-point number, can take on values from a subset P of rationals of the form $p/2^b$ in which p is an integer in the range $0 \leq p \leq 2^N - 1$ for unsigned, and $-2^{N-1} \leq p \leq 2^{N-1} - 1$ for signed numbers, respectively. Note that P contains 2^N elements and b represents the fractional bits in each case. Based on this fact, we can depict the range of values covered by each case as shown by Figure 4. Thereafter, the representable range of fixed-point numbers is divided into 2^N equispaced quantization steps with the distance between two successive steps equal to $1/2^b$. Suppose that $x \in \mathbb{R}$ is approximated by a fixed-point number a. The position of these values are labeled in the figure. The error $\mid x - a \mid$ is hence less than the length of one interval, or $1/2^b$, as mentioned in the second theorem. In comparison to floating-point case, the fixed-point representation leads to equal spacing in the set of representable numbers. Thus the maximum absolute error is the same throughout (*ulp* with truncation and *ulp*/2 with rounding).
- **Lemmas about General Fixed-Point Rounding Error:** To prove the general fixed-point error bound theorem, a set of five lemmas is established. We first proved that the rounding result is the nearest value to a real number (*FXP_BOUND_AT_WORST_LEMMA*) and the corresponding error is min-

imum (*FXP_ERROR_AT_WORST_LEMMA*) comparing to the other fixed-point numbers. Then, we proved (*FXP_ERROR_BOUND_LEMMA1*) that each representable real value x can be surrounded by two successive rational numbers. Also we proved (*FXP_ERROR_BOUND_LEMMA2*) that the difference between the real number and the surrounding rationals is less than $1/2^{fracbits(X)}$. Finally, we proved (*FXP_ERROR_BOUND_LEMMA3*) that for each real value we can find a fixed-point number with the required error characteristics. Since the rounding produces the minimum error as stated in *FXP_ERROR_AT_WORST_LEMMA*, the proof of the second main theorem (*FXP_ERROR_BOUND_THM*) is a direct consequence of *FXP_ERROR_BOUND_LEMMA3*. In these proofs, we have treated the case of signed and unsigned numbers separately since they have different definitions for *MAX*, *MIN*, and *value* functions. For signed numbers special attention needs also to be paid to dealing with the negative numbers.

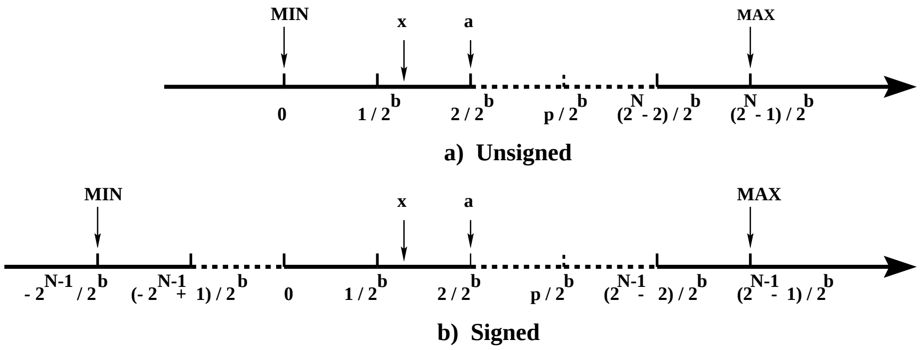
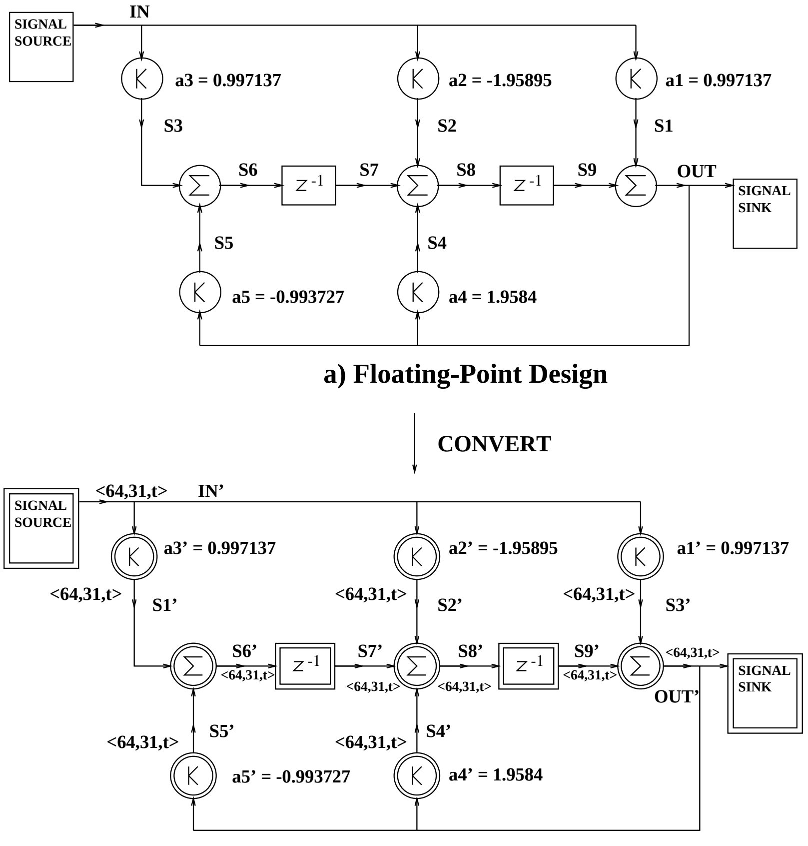


Fig. 4. Fixed-Point Values on the Real Axis

– **SystemC Fixed-Point Error Bound Theorem:** The theorem *FXP_ERR OR_BOUND_THM* is a general theorem for bounding the error in fixed-point rounding which is valid for all rounding modes. This theorem can then be extended to prove new theorems for different rounding modes in SystemC fixed-point arithmetic. For instance, for *SC_TRN, SC_RND_ZERO, SC_RND_MIN_INF,SC_RND_INF* and *SC_RND_CONV* modes which round to nearest representable values, the error is less than *ulp/2*. For these modes the error is bounded to $1/2^{fracbits(X)+1}$. This fact is proved as in theorem *SYSTEMC_FXP_ERROR_BOUND_THM.*

5 The Notch Filter Example

In this section we demonstrate how to apply the formalization of SystemC fixed-point arithmetic presented in the previous sections for the verification of DSP systems. We have chosen CoCentric Fixed-Point Designer [33] as the application tool and the case of a second order 60 Hz *Notch Filter* as an example circuit (Figure 5). The filter is first designed and simulated using floating-point operations and parameters (Figure 5(a)). The design is composed of *Add* (adder), *Gain* (multiply by a constant), and *Delay* blocks together with signal source and sink elements. The design is then converted to a fixed-point design (Figure 5(b)) in which each block is replaced with the corresponding fixed-point block. Fixed-point blocks are shown by double circles and squares to distinguish from floating-point blocks. The attributes of all fixed-point block outputs are set to $< 64, 31, t >$ to ensure that overflows and quantization do not affect the system operation. This means that we have used sixty four bits to represent the signal values, the numbers are in two's complement format in which the most significant bit is the sign bit, and the binary point is fixed at the thirty first position following the sign bit.



b) Fixed-Point Design

Fig. 5. A Second Order Notch Filter

Figure 6 shows the proposed verification methodology. Based on this methodology, we first modeled the design in different abstraction levels such as floating-point and fixed-point levels as predicates in higher order logic (*NOTCH_FILTER_FLOAT_IMP,NOTCH_FILTER_FXP_IMP*). The process of specifying a hardware description language in higher-order logic is commonly known as semantic embedding. There are two main approaches [5]: deep embedding and shallow embedding. In deep embedding, the abstract syntax of a design description is represented by terms, which are then interpreted by semantic functions defined in the logic that assign meaning to the design. With this method, it is possible to reason about classes of designs, since one can quantify over the syntactic structures. However, setting up HOL types of abstract syntax and semantic functions can be very tedious. In a shallow embedding on the other hand, the design is modeled directly by a formal specification of its functional behavior. This eliminates the effort of defining abstract syntax and semantic functions, but it also limits the proofs to functional properties. In this example, since our main concern is to check the correctness of the design based on its functionality, we propose shallow embedding: translate the intended meaning of the design blocks into HOL and then complete the formal proof in HOL theorem prover. Primitive blocks are defined using the corresponding functions in floating-point and fixed-point theories in HOL. The whole filter is then implemented as a conjunction of these blocks.

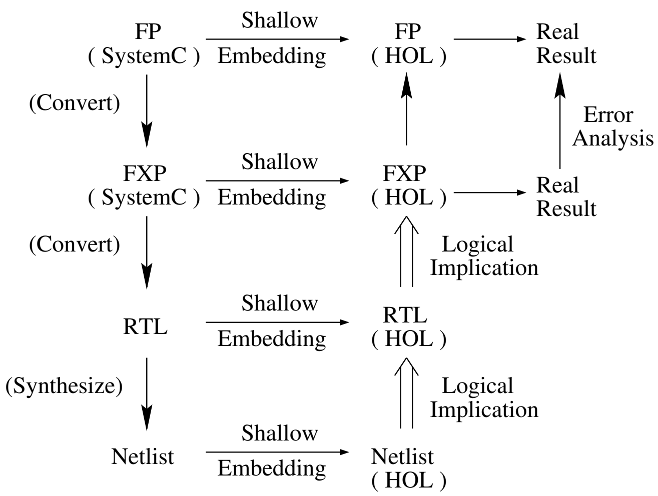


Fig. 6. Verification Methodology

In the next step, separately and independently from the actual implementations, we described the designs as a difference equation relating the input and output samples (*NOTCH_FILTER_FLOAT_SPEC,NOTCH_FILTER_FXP_SPEC*). Then, we established lemmas that ensure the implementation at each level satisfies the corresponding specification (*NOTCH_FILTER_FLOAT_IMP_SPEC, NOTCH_FILTER_FXP_IMP_SPEC*). For the error analysis of transition from floating-point to fixed-point levels, and based on the theorems *FXP_ADD_THM*, *FXP_MUL_THM*, and the corresponding ones in floating-point theory, we proved a theorem (*NOTCH_FILTER_FXP_TO_FLOAT_THM*) that states the error between the real values of the floating and fixed-point precision output samples. According to this theorem, for a valid and finite set of input and output sequences at times n - *1* and n - *2*, we will have finite and valid outputs at time n. Also, the difference between the output real values at each sample time can be expressed as the difference in input and output values at previous sample times multiplied by the corresponding coefficients, taking into account the effects of finite precision in coefficients and arithmetic operations. Proper assumptions are set for both floating-point and fixed-point designs to guarantee the validity of output samples. Based on this theorem, three sources of error can be distinguished: errors due to the quantization of input samples, errors due to the rounding in arithmetic operations, and errors due to quantization of coefficients. The errors are already quantified using the theorem *SYSTEMC_FXP_ERROR_BOUND_THM* and the corresponding theorems for error analysis in floating-point case.

Next, we generated with CoCentric System Studio [32] the VHDL code corresponding to the Filter design, and used Synopsys to synthesize the code to reach to the logic gate level netlist. At this point, we used the well known formal techniques to model the design in each of these levels in higher-order logic within the HOL environment (*NOTCH_FILTER_RTL_IMP*, *NOTCH_FILTER_NETLIST_IMP*). The next step is to verify these different levels using a classical hierarchical proof approach in HOL [23]. Our final goal is to prove that the gate level implementation implies the floating-point algorithmic design considering the errors (*NOTCH_FILTER_NETLIST_TO_FLOAT_THM*). This goal cannot be reached directly, due to the very high abstraction gap between the gate and floating-point algorithmic levels. The proof scheme need hence to be changed to hierarchically prove that the gate level implies the more abstract RTL (*NOTCH_FILTER_NETLIST_TO_RTL_THM*). The latter is used to imply the high level fixed-point algorithmic specification (*NOTCH_FILTER_RTL_TO_FXP_THM*) which has already been related to the floating-point description through the error analysis. This can be formalized in HOL using *float* and *Fxp* data abstraction functions which map binary words to floating-point and fixed-point numbers, respectively. In the proof of these theorems we used the regular and modular behavior of the design, so that we proved separate lemmas for different primitive modules such as adder, multiplier, and delay and then used these lemmas in the proof of the original theorems.

6 Conclusions

As system-on-a-chip (SoC) designs become a driving force in electronics systems, current verification techniques are falling behind at an increasing rate. Verification of today's SoCs occurs at low levels of abstraction, typically RTL. As the complexity of SoCs grows, it is important to move the verification to higher levels of abstraction. In this paper, we proposed the formalization of SystemC based fixed-point arithmetic in the HOL theorem prover as a basis for modeling and verification of SoC designs at floating-point and fixed-point algorithmic levels against the implementations in RTL and netlist gate levels. The formalization presented in this paper is an extension to the previous work on formalization of IEEE standard based floating-point arithmetic and Cadence SPW based fixed-point arithmetic. We modeled the generalized SystemC fixed-point data types and extended the verification to cover the different rounding and overflow modes in SystemC fixed-point arithmetic. Finally, we used our formalization for modeling and verification of a second order Notch filter system.

References

1. M. D. Aagaard, and C.-J. H. Seger, "The Formal Verification of a Pipelined Double-Precision IEEE Floating-Point Multiplier," In International Conference on Computer Aided Design, San Francisco, CA, USA, pp. 7-10, Nov. 1995.
2. B. Akbarpour, S. Tahar, and A. Dekdouk, "Formalization of Cadence SPW Fixed-Point Arithmetic in HOL," In Integrated Formal Methods, Lecture Notes in Computer Science, Vol. 2335, Springer-Verlag, pp. 185-204, 2002.
3. G. Barrett, "Formal Methods Applied to a Floating Point Number System," IEEE Transactions on Software Engineering, SE-15(5): 611-621, May 1989.
4. C. Berg and C. Jacobi, "Formal Verification of the VAMP Floating Point Unit," In Correct Hardware Design and Verification Methods, Lecture Notes in Computer Science, Vol. 2144, Springer-Verlag, pp. 325-339, 2001.
5. R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van-Tassel, "Experience with Embedding Hardware Description Languages in HOL," In Theorem Provers in Circuit Design, pages 129-156. North-Holland, 1992.
6. V. A. Carreno, "Interpretation of IEEE-854 Floating-Point Standard and Definition in the HOL System," NASA Technical Memorandum 110189, September 1995.
7. H. Chang et al., Surviving the SoC Revolution, A Guide to Platform-Based Design, Kluwer Academic Publishers, Boston, 1999.
8. M. Cornea-Hasegan, "Proving the IEEE Correctness of Iterative Floating-Point Square Root, Divide, and Remainder Algorithms," Intel Technology Journal, Q2, 1998, pp. 1-11.
9. Y.-A. Chen and R. E. Bryant, "Verification of Floating Point Adders," In Computer Aided Verification, Lecture Notes in Computer Science, Vol. 1427, Springer-Verlag, pp. 488-499, 1998.
10. M. Daumas, L. Rideau, L. Thry, "A Generic Library for Floating-Point Numbers and its Application to Exact Computing," In Theorem Proving in Higher Order Logics, Lecture Notes in Computer Science, Vol. 2152, Springer-Verlag, pp. 169-184, 2001.
11. M. J. C. Gordon and T. F. Melham, Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic, Cambridge University Press, 1993.
12. J. R. Harrison, "Theorem Proving with the Real Numbers," Technical Report 408, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK, December 1996.
13. J. R. Harrison, "Floating-Point Verification in HOL Light: The Exponential Function," Formal Methods in System Design 16(3): 271-305 (2000).

14. J. R. Harrison, "A Machine-Checked Theory of Floating-Point Arithmetic," In Theorem Proving in Higher Order Logics, Lecture Notes in Computer Science, Vol. 1690, Springer-Verlag, pp. 113-130, 1999.
15. IEEE, Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985, The Institute of Electrical and Electronic Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA, 1985.
16. IEEE, Standard for Radix-Independent Floating-Point Arithmetic, ANSI/IEEE Std 854-1987, The Institute of Electrical and Electronic Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA, 1987.
17. C. Inacio, D. Ombres, "The DSP Decision: Fixed Point or Floating?," IEEE Spectrum, 0018-9234, September 1996.
18. Intel Inc., "Pentium Processors, Statistical Analysis of Floating-Point Flaw," Intel White Paper, Sec. 3, November 1994.
19. C. Kern and M. Greenstreet, "Formal Verification in Hardware Design: A Survey," ACM Transactions on Design Automation of Electronic Systems, Vol. 4, pp. 123-193, April 1999.
20. M. Leeser, and J. O'Leary, "Verification of a Subtractive Radix-2 Square Root Algorithm and Implementation," In International Conference on Computer Design, Cambridge, MA, USA, pp. 526-531, October 1995.
21. J. O Leary, X. Zhao, R. Gerth, and C.-J.H. Seger, "Formally Verifying IEEE Compliance of Floating-Point Hardware," Intel Technology Journal, Vol. 1999-Q1, pp. 1-14.
22. T. F. Melham, "The HOL pred_sets Library," University of Cambridge, Computer Laboratory, February 1992.
23. T. Melham, Higher Order Logic and Hardware Verification, Cambridge Tracts in Theoretical Computer Science 31 (Cambridge University Press, 1993).
24. P. S. Miner, and J. F. Leathrum, "Verification of IEEE Compliant Subtractive Division Algorithms," In International Conference on Formal Methods in Computer-Aided Design, Lecture Notes in Computer Science, Vol. 1166, Springer-Verlag, pp. 64-78, 1996.
25. P.S. Miner, "Defining the IEEE-854 Floating-Point Standard in PVS," Technical Memorandum 110167, NASA, Langley Research Center, Hampton, VA 236810001, USA, June 1995.
26. J.S. Moore, T. Lynch, and M. Kaufmann, "A Mechanically Checked Proof of the Correctness of the Kernel of the AMD5K86 Floating-Point Division Algorithm," IEEE Transactions on Computers, Vol. 47, pp. 913-926, 1998.
27. S. M. Mueller and W. J. Paul, Computer Architecture. Complexity and Correctness, Springer-Verlag, 2000.
28. D. M. Russinoff, "A Case Study in Formal Verification of Register-Transfer Logic with ACL2: The Floating-Point Adder of the AMD Athlon Processor," In Formal Methods in Computer-Aided Design, Lecture Notes in Computer Science, Vol. 1954, Springer-Verlag, pp. 3-36, 2000.
29. Signal Processing WorkSystem (SPW) User's Guide, Cadence Design Systems, Inc., July 1999.
30. S. Swan, An Introduction to System Level Modeling in SystemC 2.0, Cadence Design Systems, Inc., May 2001.
31. http://www.systemc.org/.
32. CoCentric^{TM} System Studio User's Guide, Synopsys, Inc., USA, August 2001.
33. CoCentric Fixed-Point Designer User's Guide, Synopsys, Inc., USA, August 2002.