Formalization of Fixed-Point Arithmetic in HOL

Behzad Akbarpour   Sofiène Tahar
Abdelkader Dekdouk
Dept. of Electrical and Computer Engineering, Concordia University
1455 de Maisonneuve W., Montreal, Quebec, H3G 1M8, Canada
Email: {behzad,tahar,dekdouk}@ece.concordia.ca

October 28, 2004

Abstract

This paper addresses the formalization in higher-order logic of fixed-point arithmetic. We encoded the fixed-point number system and specified the different quantization modes in fixed-point arithmetic such as the directed and even quantization modes. We also considered the formalization of exceptions detection and their handling like overflow and invalid operation. An error analysis is then performed to check the correctness of the quantized result after carrying out basic arithmetic operations, such as addition, subtraction, multiplication and division against their mathematical counterparts. Finally, we showed by an example how this formalization can be used to enable the verification of the transition from floating-point to fixed-point algorithmic level in the signal processing design flow.

Keywords: Fixed-Point Arithmetic, Floating-Point Arithmetic, Theorem-Proving, HOL

1 Introduction

Modern signal processing chips, such as integrated cable modems and wireless multimedia terminals, are described with algorithms in floating-point precision. Often, the architectural style with which these algorithms are implemented is precision-limited, and relies on a fixed-point representation. This requires a translation of the specification from floating-point to fixed-point precision. This implementation is optimized following some application specific trade-offs such as speed, cost, area and power consumption of the chip. The optimization task is tedious and error prone due to the effects of quantization noise introduced by the limited precision of fixed-point representation. An overview of a conventional digital signal processing (DSP) design flow is depicted in Figure 1 [23].

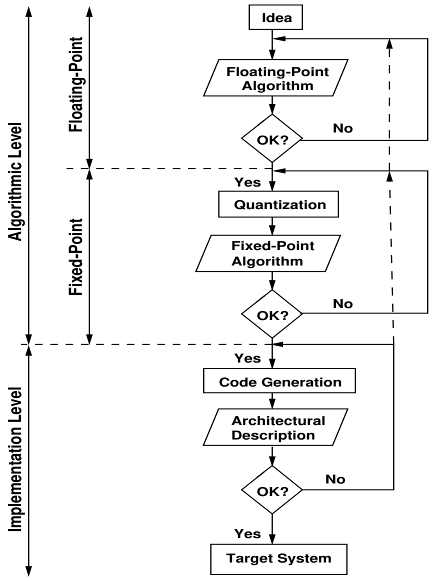


Figure 1: DSP Design Flow

Usually the conformance of the fixed-point implementation with respect to the floating-point specification is verified by simulation techniques which cannot cover the entire input space yielded by the floating-point representation. The objective of this work is to formalize the fixed-point arithmetic in higher-order logic as a basis for checking the correctness of the implementation of DSP designs against higher level algorithmic descriptions in floating-point and fixed-point representations.

Unlike floating-point arithmetic which is standardized in IEEE-754 [18] and IEEE-854 [19], current fixed-point arithmetic does not follow any particular standard and depends on the tool and the language used to design the DSP chip. Examples of such tools are SPW (Cadence) [7], Matlab-Simulink (Mathworks) [25], CoCentric (Synopsys) [37], and DSP Station (Mentor Graphics) [27]. For instance, in SPW (Signal Processing Worksystem), a fixed-point number is defined as a binary string and a set of attributes. Attributes specify how the binary string is interpreted using three arguments for the total number of bits, the number of integer bits, and the sign format. For arithmetic operations, it supports three kinds of exceptions such as *loss-of-sign* or *overflow*, two overflow modes, and five quantization modes. In Matlab Simulink Fixed-Point Blockset [26], fixed-point numbers are stored in data types that are characterized by their word size (up to 128 bits), a *radix point*, and whether they are signed or unsigned. The *radix point* is used to support integers, fractionals, and generalized fixed-point data types. The Matlab Blockset provides four quantization modes corresponding to those supported by SPW. It also supports saturation and wrapping to deal with overflow for all fixed-point data types. Another example is the Synopsys CoCentric tool, which uses fixed-point as described in the SystemC language [33]. It supports signed and unsigned fixed-point data types, as well as limited precision (53 bits mantissa) fixed-point, called *fast fixed-point* to speed up simulation. SystemC supports seven quantization modes, of which four correspond exactly to the quantization 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. With the objective of providing a general methodology for the formalization and verification of fixed-point arithmetic using higher-order logic, we define in this paper a complete common set of fixed-point arithmetic as supported by most of the DSP tools, in particular SPW and SystemC.

Based on higher-order logic, we propose to encode a fixed-point number by a pair composed of a Boolean word, and a triplet indicating the word length, the length of the integer portion, and the sign format. Then, we formalize the concepts of valuation and quantization as functions that convert respectively a fixed-point number to a real number and vice versa, taking into account different quantization 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 quantization on the real number result. Finally, we prove various lemmas regarding the error analysis of the fixed-point quantization and correctness of the basic operations like addition, multiplication, and division. The higher-order logic formalization and proof were done using the HOL theorem prover

[12]. They were developed into a full fixed-point arithmetic library, which was recently included in the last release of HOL (HOL4, Kananaskis-2).

The rest of the paper is organized as follows: Section 2 gives a review on work related to the formalization of floating-point arithmetic, some of which directly influenced our work. Section 3 describes the fixed-point arithmetic definitions adopted in this paper including the format of the fixed-point numbers, arithmetic operations, exceptions detection and their handling, and the different overflow and quantization modes. Section 4 describes in detail their formalization in HOL. In Section 5, we discuss the verification of basic fixed-point arithmetic operations, such as addition and multiplication. Section 6 presents an illustrative example on how this formalization can be used through the modeling and verification of an Integrator circuit. Finally, Section 7 concludes the paper.

2 Related Work

There exist several related work in the open literature on the formalization and verification of IEEE standard based floating-point arithmetic. For instance, Barrett [2] specified parts of the IEEE-754 standard in Z, and Miner [29] formalized the IEEE-854 floating-point standard in PVS. The latter defined the relation between floating-point numbers and real numbers, rounding, and some arithmetic operations on both finite and infinite operands. He used this formalization to verify abstract mathematical descriptions of the main operations and their relation to the corresponding floating-point implementations. His work was one of the earliest on the formalization of floating-point standards using theorem proving. His formal specification was then used by Miner and Leathrum [30] to verify in PVS a general class of IEEE compliant subtractive division algorithms.

Carreno [8] formalized the same IEEE-854 standard in HOL. He interpreted the lexical descriptions of the standard into mathematical conditional descriptions and organized them in tables, which were then formalized in HOL. He discussed different standard aspects such as precisions, exceptions and traps, and many other arithmetic operations such as addition, multiplication, and square-root of floating-point numbers.

Harrison [13] constructed the real numbers in HOL. He then developed in HOL a generic floating-point library [14] to define the most fundamental terms of the IEEE-754 standard and to prove the corresponding correctness analysis lemmas. He used this library to formalize and verify floating-point algorithms of complex arithmetic operations such as the square root, the exponential function [15], and the transcendental functions [16] against their abstract mathematical counterparts. He also used the floating-point library for the verification of the class of division algorithms used in the Intel IA-64 architecture [17].

Moore *et al.* [31] have verified the AMD-K5 floating-point division algorithm using the ACL2 theorem prover. Also, Russinoff [35] has developed a floating-point library for the ACL2 prover and applied it successfully to verify the floating-point multiplication, division, and square root algorithms of the AMD-K5 and AMD Athlon processors.

Aagaard and Seger [1] combined BDD-based model-checking and theorem proving techniques in the Voss hardware verification system to verify the IEEE compliance of the gate-level implementation of a floating-point multiplier. O'Leary *et al.* [34] reported on the specification and verification of the Intel Pentium® Pro processor's floating-point execution unit at the gate level using a combination of model-checking and theorem proving. Leeser *et al.* [24] verified a subtractive radix-2 square root algorithm and its hardware implementation using the higher-order logic theorem proving system Nuprl. Chen and Bryant [10] used word-level SMV to verify a floating-point adder. Cornea-Hasegan [9] used iterative approaches and mathematical proofs to verify the correctness of the IEEE floating-point square root, divide, and remainder algorithms.

More recently, Daumas *et al.* [11] have presented a generic library for reasoning about floating-point numbers within the Coq system. This library was then used in the verification of IEEE-compliant floating-point arithmetic algorithms [5] and hardware units [6]. Berg *et al.* [3] have formally verified a theory of IEEE rounding presented in [32] using the theorem prover PVS. They have used a formal definition of rounding based on Miner's formalization of the standard [29]. This theory was then used to prove the correctness of a fully IEEE compliant floating-point unit used in the VAMP processor [4]. Sawada and Gamboa [36] formally verified the correctness of a floating-point square root algorithm used in the IBM Power4^{TM} processor. The verification was carried out with the ACL2(r) theorem prover which is an extension of the ACL2 theorem prover that performs reasoning on real numbers using non-standard analysis. The proof required the analysis of the approximation error on Chebyshev series by proving Taylor's theorem. Kaivola *et al.* [20, 21, 22] presented the formal verification of the floating-point multiplication, division, and square root units of the Intel IA-32 Pentium® 4 microprocessor. The verification was carried out using the Forte verification framework, a combined model-checking and theorem-proving system built on top of the Voss system. Model checking was done via symbolic trajectory evaluation (STE), and theorem proving was done in the ThmTac proof tool.

While all of the above work are concerned with floating-point representation and arithmetic, there is no report in the open literature on any machine-checked formalization of properties of fixed-point arithmetic. Therefore, the formalization presented in this paper is to our best knowledge, the first of its kind. Our formalization of the fixed-point arithmetic has been inspired mostly by the work done by Harrison [15] and Carreno [8] on floating-point. Harrison's work was more oriented towards verification purposes. Indeed, we 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 quantized fixed-point result. For exception handling which is not covered by Harrison [15], we followed Carreno [8] who formalized floating-point exceptions and their handling in more details.

3 Fixed-Point Arithmetic

In this section we describe the fixed-point arithmetic definitions on which we base our formalization. While we tried to keep these definitions as general as possible, the fixed-point numbers format, arithmetic operations, overflow and quantization modes, and exception handling adopted are to some extent influenced by the fixed-point arithmetic defined by Cadence SPW [7] and Synopsys SystemC [33].

3.1 Fixed-Point Numbers

A fixed-point number has a fixed number of binary digits and a fixed position for the decimal point with respect to that sequence of digits. Fixed-point numbers can be either unsigned (always positive) or signed (in two's complement representation). For example, consider the case of four bits being used to represent the fixed-point numbers. If the numbers are unsigned and if the decimal point or, more properly, the binary point is fixed at the position after the second digit ($XX.XX$), the representable real values range from 0.0 to 3.75. In two's complement format, the most significant bit is the sign bit. The remaining bits specify the magnitude. If four bits represent the fixed-point numbers, and the binary point is fixed at the position after the second digit following the sign bit ($SXX.X$), the real values range from -4.0 to $+3.5$.

Fixed-point numbers are expressed as a pair consisting of a binary string and a set of attributes, $(Binary\ String, Attributes)$. The attributes specify how the binary string is interpreted. Generally, the attributes are specified in the following format:

$$(wl, iwl, sign) \tag{1}$$

which consists of the following parameters:

- **wl:** Total word length, specifying the total number of bits used to represent the fixed-point binary string, including integer bits, fractional bits, and sign bit, if any. Word length must be in the range of 1 to 256.
- **iwl:** Integer word length, specifying the number of integer bits (the number of bits to the left of the binary point, excluding the sign bit, if any). If this number is negative, repeated leading sign bits or zeros are added to generate the equivalent binary value. If this number is greater than the total word length, trailing zeroes are added to generate the equivalent binary value.
- **sign:** A letter specifying the sign format: "u" for unsigned, and "t" for two's complement.

Example: According to the above definitions, the real value -0.75 is represented by $(111101, (6, 3, t))$. If we consider the same bit string with unsigned attributes $(111101, (6, 3, u))$, then the equivalent number is 111.101 or $+7.625$. On the other hand, $(111101, (6, -3, u))$ represents the value .000111101 which is $+0.119140625$.

3.2 Fixed-Point Operations

A DSP design tool usually provides a library including basic fixed-point signal processing blocks such as adders, multipliers, delay blocks, and vector blocks. It also supports fixed-point hardware blocks such as multiplexers, buffers, inverters, flip-flops, bit manipulation and general-purpose combinational logic blocks. These blocks accurately model the behavior of fixed-point digital signal processing systems. In this paper, we will focus on the arithmetic and logic operations, but the idea can be generalized to the remaining operations. 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. In addition to the parameters corresponding to the input operands and output result, the arithmetic operations take specific parameters defining the overflow and quantization (loss of precision) modes. These parameters are as follows:

- **q_mode:** Quantization mode. This parameter determines the behavior of the fixed-point operations when the result generates more precision in the least significant bits (LSB) than is available.
- **o_mode:** Overflow mode. This parameter determines the behavior of the fixed-point operations when the result generates more precision in the most significant bits (MSB) than is available.
- **n_bits:** Number of saturated bits. This parameter is only used for overflow mode and specifies how many bits will be saturated if a saturation behavior is specified and an overflow occurs.

Example: Consider a block that serves as a primitive fixed-point multiplier, which truncates the results when loss of precision occurs and wraps the result when overflow occurs. We can make a call to the multiplier routine through the function *fxpMul* $(Wrap \mid Truncate, In1, In2, Out)$, in which *In1* and *In2* are the input fixed-point operands, *Out* is a parameter corresponding to the output attributes, and *Wrap* and *Truncate* indicate the overflow and quantization modes, respectively.

3.2.1 Fixed-Point Exception Handling

Fixed-point arithmetic operations that do not compute and return an exact result resort to an exception-handling procedure. This procedure is controlled by the exception flags. There are three kinds of exceptions that can be tested [7]:

- **Loss of Sign:** The result was negative but the result storage area was unsigned. Zero is stored.
- **Overflow:** The result was too big to be represented in the result storage area. The overflow mode determines the returned value.
- **Invalid:** No result can be meaningfully represented (e.g., divide by zero). This error can also occur if the fixed-point number itself is invalid.

3.2.2 Fixed-Point Quantization Modes

Quantization effects are used to determine what happens to the LSBs of a fixed-point type when more bits of precision are required than are available. The quantization modes are listed in Table 1.

Table 1. Fixed-Point Quantization Modes

Quantization Mode	**Name**
Quantization to Plus Infinity	RND
Quantization to Zero	RND_ZERO
Quantization to Minus Infinity	RND_MIN_INF
Quantization to Infinity	RND_INF
Convergent Quantization	RND_CONV
Truncation	TRN
Truncation to Zero	TRN_ZERO

Figure 2 shows the behavior of each quantization mode. The X axis is the result of the previous arithmetic operation and the Y axis is the value after quantization. The diagonal line represents the ideal number representation given infinite bits. The small horizontal lines show the effect of the quantization. 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. Each non integer value on the X axis is located in a quantization interval surrounded by two successive integer multiples of q as its closest representable quantized numbers, one greater and one smaller than the original value. If the value is exactly in the middle of the quantization interval, then the two closest representable numbers are equally distanced apart from the original value. As shown in this figure modes *RND*, *RND_ZERO*, *RND_MIN_INF*, *RND_INF*, and *RND_CONV* will quantize a value to the closest representable number if the two nearest representable numbers are not equally distanced apart from the original value. Otherwise, quantization 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 2 (a-e)). The *TRN* mode is the default for fixed-point types and will be used if no other value is specified. The result is always quantized towards minus infinity (Figure 2 (f)). In other words, the result value is the first representable number lower than the original value. Finally, for *TRN_ZERO* the result is the nearest representable value to zero (Figure 2 (g)) [33].

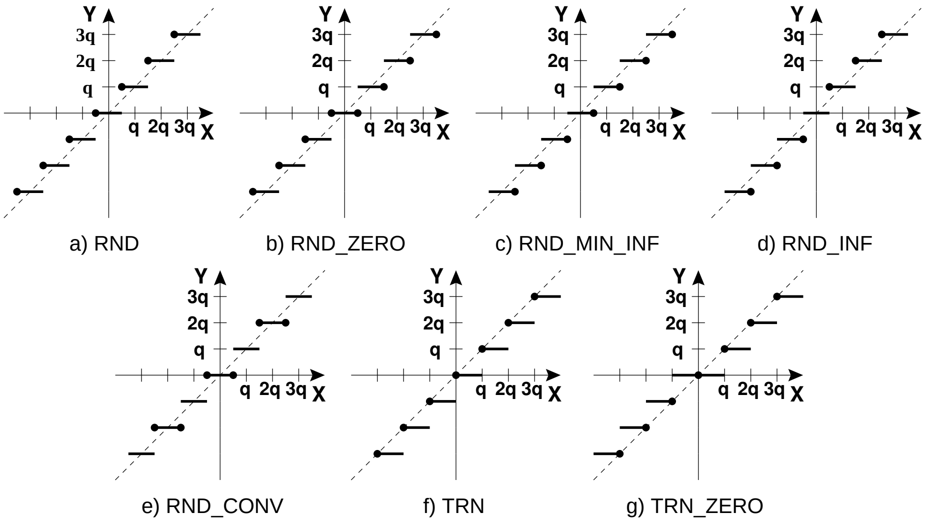


Figure 2: The Behavior of Fixed-Point Quantization Modes

3.2.3 Fixed-Point Overflow Modes

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 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, and are listed in Table 2.

Table 2. Fixed-Point Overflow Modes

Overflow Mode	**Name**
Saturation	SAT
Saturation to Zero	SAT_ZERO
Symmetrical Saturation	SAT_SYM
Wrap-Around	WRAP
Sign Magnitude Wrap-Around	WRAP_SM

Figure 3 shows the behavior of each overflow mode for a 3 bit fixed-point data 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 fixed-point data type. The *SAT* mode will convert the specified value to *MAX* for an overflow or *MIN* for an underflow condition (Figure 3 (a)). The *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 3 (b)). In the *SAT_SYM* mode, positive overflow will generate *MAX* and negative overflow will generate $-MAX$ for signed numbers or *MIN* for unsigned numbers (Figure 3 (c)). With the *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 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 3 (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 given in Figure 3 (e). Note that positive numbers wrap around to 0 while negative values wrap around to -1. The *WRAP_SM* overflow mode uses sign magnitude wrapping. This overflow mode behaves in two different styles depending on the value of the *n_bits* parameter. 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 the 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 provided in Figure 3 (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 given in Figure 3 (g). Note that while the graph looks somewhat like a sawtooth waveform, positive numbers do not dip below 0 and negative numbers do not cross -1 [33].

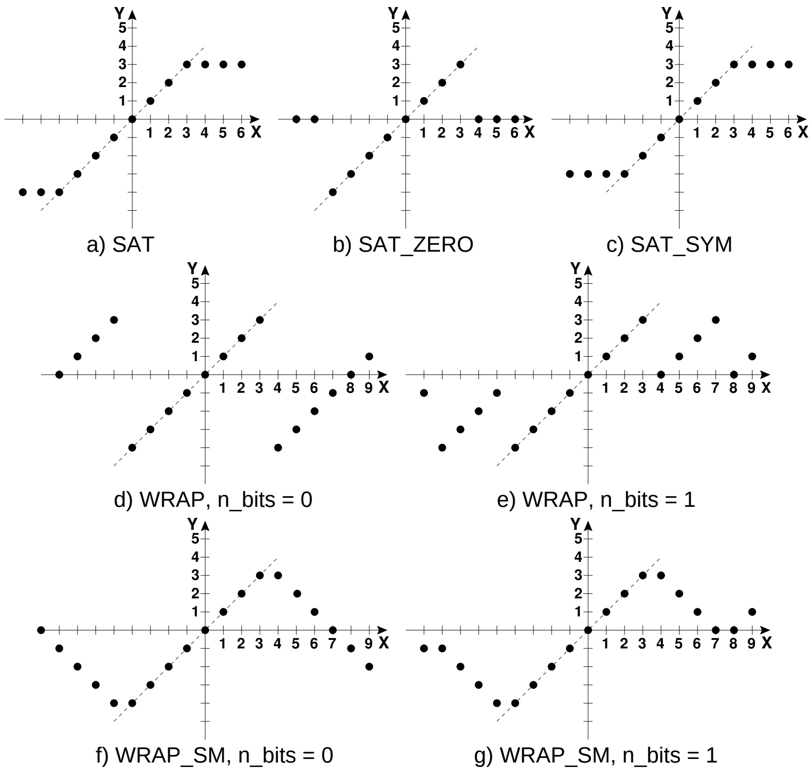


Figure 3: The Behavior of Fixed-Point Overflow Modes

4 Formalizing Fixed-Point Arithmetic in HOL

In this section, we present formalization of the fixed-point arithmetic in higher-order logic, based on the general purpose HOL theorem prover. The

HOL system supports both forward and backward proofs. The forward proof style applies inference rules to existing theorems to obtain new theorems and eventually the desired theorem. Backward or goal oriented proofs start with the goal to be proven. Tactics are applied to the goal and subgoals until the goal is decomposed into simpler existing theorems or axioms. The system basic language includes the natural numbers and Boolean type. It also includes other specific extensions like reals library [13], which was proved to be essential for our fixed-point arithmetic formalization. Table 3 summarizes some of the HOL symbols used in this paper and their meanings [12].

Table 3. HOL Symbols

HOL Symbol	**Standard Symbol**	**Meaning**
$@x.\ t$	$\varepsilon x.\ t$	An x such that $t\ (x)$ holds
$\lambda x.\ t$	$\lambda x.\ t$	Function that maps x to $t\ (x)$
&	(none)	Natural map operator ($\mathbb{N} \rightarrow \mathbb{R}$)
$\neg\ t$	$\neg\ t$	Not t
$\neg\ x$	$-\ x$	Unary negation of x
$inv\ (x)$	x^{-1}	Multiplicative inverse of x
$abs\ (x)$	$\mid x \mid$	Absolute value of x
$x\ pow\ n$	x^n	Real x raised to natural number power n
$m\ EXP\ n$	m^n	Natural number m raised to exponent n

The HOL type system does not support subtypes, so the real numbers ($\mathbb{R}$) have formally a different type from the natural numbers ($\mathbb{N}$). Therefore, the unary operator *ampersand* (&) is used to map between them. Thus the real number *numerals* can be written as $\&0, \&1$, etc [15].

4.1 Fixed-Point Numbers Representation

The actual fixed-point numbers are represented in HOL by a pair of elements representing the binary string and the set of attributes. The extractors for the two fields of a fixed-point number are defined as follows:

$\vdash_{def}$ `string (s,a) = s`
$\vdash_{def}$ `attrib (s,a) = a`

The binary string is treated as a Boolean word (type: *bool word*). For example, the bit string 1010 is represented by *WORD [T;F;T;F]*. In this way, we use the definitions and theorems already available in the HOL word library [39] to facilitate the manipulation of binary words. The attributes are represented by a triplet of natural numbers for the total number of bits, the integer bits and the sign format.

In HOL, we define functions to extract the primitive parameters for arbitrary attributes.

```
⊢def  wordlength (w,iw,s) = w
⊢def  intbits (w,iw,s) = iw
⊢def  sign (w,iw,s) = s
```

We also define predicates partitioning the fixed-point numbers into signed and unsigned numbers.

```
⊢def  is_signed X = (sign X = 1)
⊢def  is_unsigned X = (sign X = 0)
```

The number of digits on the right hand side of the binary point of a fixed-point number is defined as *fracbits*. It can be derived as the difference between the total number of bits and the number of integer bits, considering the sign bit in the case of signed numbers.

```
⊢def  fracbits X =
       if (is_unsigned X) then (wordlength X - intbits X)
       else (wordlength X - intbits X - 1)
```

Two useful derived predicates test the validity of a set of attributes and a fixed-point number based on the definition in Section 3.1. In a valid set of attributes, the *wordlength* should be in the range of 1 and 256, the *sign* can be either 0 or 1, and the number of integer bits is less than or equal to the *wordlength*. A valid fixed-point number must have a valid set of attributes and the length of its binary string must be equal to the *wordlength*.

```
⊢def  validAttr X =
       wordlength X > 0 ∧ wordlength X < 257 ∧
       intbits X < wordlength X + 1 ∧ sign X < 2

⊢def  is_valid a =
       validAttr (attrib a) ∧ (WORDLEN (string a) = wordlength (attrib a))
```

where *WORDLEN* is a predefined function of the HOL *word* library, which returns the size of a word.

4.2 Fixed-Point Type

Now we define the actual HOL type for the fixed-point numbers. The type is defined to be in bijection with the appropriate subset of *(bool word* $\times \mathbb{N}^3$*)*, with the bijections written in HOL as *fxp : (bool word* $\times \mathbb{N}^3$*)* $\rightarrow$ *fxp*, and *defxp : fxp* $\rightarrow$ *(bool word*$\times\mathbb{N}^3$*)*. The bijection maps the set of all elements of type *(bool word*$\times\mathbb{N}^3$*)* to the set of valid fixed-point numbers specified by the function *is_valid* as defined in the previous section. For this purpose, we make use of built-in facilities in HOL for defining new bijection types [38]. A similar technique was used in [15] for defining type bijections for the floating-point numbers (*float,defloat*) in HOL.

```
   fxp_tybij =
⊢  (∀a. fxp (defxp a) = a) ∧ (∀r. is_valid r = (defxp (fxp r) = r))
```

We specialize the previous functions and predicates to the *fxp* type, as follows:

```
⊢def  String a = string (defxp a)
⊢def  Attrib a = attrib (defxp a)
⊢def  Wordlength a = wordlength (Attrib a)
⊢def  Intbits a = intbits (Attrib a)
⊢def  Fracbits a = fracbits (Attrib a)
⊢def  Sign a = sign (Attrib a)
⊢def  Issigned a = is_signed (Attrib a)
⊢def  Isunsigned a = is_unsigned (Attrib a)
⊢def  Isvalid a = is_valid (defxp a)
```

Note that we start the name of the functions manipulating fixed-point numbers by capital letters to distinguish them from those taking pairs and triplets as argument.

4.3 Fixed-Point Valuation

Now we specify the real number valuation of fixed-point numbers. We use two separate formulas for signed and unsigned numbers:

- **Unsigned:**

$$(1/2^M) * (\sum_{n=0}^{N-1} 2^n \ * \ v_n) \tag{2}$$

- **Signed:**

$$(1/2^M) * [\sum_{n=0}^{N-1} 2^n \ * \ v_n - 2^N * v_{N-1}] \tag{3}$$

where v_n represents the n^{th} bit of the binary string in the fixed-point number[1], and M and N are respectively *fracbits* and *wordlength*. In HOL, we define the valuation function *value* that returns the corresponding real value of a fixed-point number.

```
⊢def  value a =
       if (Isunsigned a) then &(BNVAL (String a)) / 2 pow Fracbits a
       else (&(BNVAL (String a)) − &((2 EXP Wordlength a) *
            BV (MSB (String a)))) / 2 pow Fracbits a
```

where *BNVAL* is a function which returns the numeric value of a Boolean word, *BV* is a function for mapping between a single bit and a number, and *MSB* is a constant for the most significant bit of a word, available in the HOL *word* library.

We also define the real value of the smallest (*MIN*) and largest (*MAX*) representable numbers for a given set of attributes. The maximum is defined for both signed and unsigned numbers using the following formula:

$$MAX \;=\; 2^a \;-\; 2^{-b} \tag{4}$$

where a is the *intbits* and b the *fracbits*. The minimum value for unsigned numbers is zero and for signed numbers is computed using the following formula:

$$MIN \;=\; -\,2^a \tag{5}$$

Thereafter, we obtain the corresponding functions in HOL.

```
⊢def  MAX X = 2 pow intbits X − inv (2 pow fracbits X)
⊢def  MIN X = if (is_unsigned X) then 0 else ¬(2 pow intbits X)
```

The constants for the smallest (*bottomfxp*) and largest (*topfxp*) representable fixed-point numbers for a given set of attributes can be defined as follows:

```
⊢def  topfxp X =
       if (is_unsigned X) then fxp (WORD (REPLICATE (wordlength X) T),X)
       else fxp (WCAT (WORD [F],WORD (REPLICATE (wordlength X − 1) T)),X)

⊢def  bottomfxp X =
       if (is_unsigned X) then fxp (WORD (REPLICATE (wordlength X) F),X)
       else fxp (WCAT (WORD [T],WORD (REPLICATE (wordlength X − 1) F)),X)
```

where *WCAT* denotes the concatenation of two words, and *REPLICATE* makes a list consisting of a value replicated a specified number of times, which are predefined functions in HOL.

[1] We adopt the convention that bits are indexed from the right hand side.

4.4 Exception Handling

Operations on fixed-point numbers can signal exceptions as described in Section 3.2. These are declared as a new HOL data type.

$\vdash_{def}$ `Exception = no_except | overflow | invalid | loss_sign`

where *no_except* is reserved for the case without exception.

Five overflow modes are also represented via an enumerated type definition.

$\vdash_{def}$ `overflow_mode = SAT | SAT_ZERO | SAT_SYM | WRAP | WRAP_SM`

According to the definition of overflow modes in Section 3.2.3 for *Saturation*, if the number is greater than *MAX* or less than *MIN*, we return *topfxp* and *bottomfxp*, as the closest representable values to the right result, respectively. For *Saturation to Zero* overflow, we will return zero in any case. For *Symmetrical Saturation*, if the number is greater than *MAX*, we return *topfxp*. If the number is less than *MIN*, we return the two's complement of the maximum value, defined by the function *minustopfxp* for signed, and *bottomfxp* for unsigned numbers, respectively. For *Wrap-around* and *Sign magnitude*, we must first convert the real number to a binary format. Then we discard the extra bits according to the output attributes, and saturate the required bits based on the parameter *n_bits*. The details are defined as functions *WRAP_AROUND* and *WRAP_AROUND_SM*. Therefore, we define the fixed-point overflow function in HOL as follows:

```
⊢def fxp_overflow X o_mode n_bits x =
      if (x > MAX X) then
            if (o_mode = SAT) then topfxp X
            else if (o_mode = SAT_ZERO) then
                  fxp (WORD (REPLICATE (wordlength X) F),X)
            else if (o_mode = SAT_SYM) then topfxp X
            else if (o_mode = WRAP) then
                  WRAP_AROUND X n_bits x
            else WRAP_AROUND_SM X n_bits x
      else if (x < MIN X) then
            if (o_mode = SAT) then bottomfxp X
            else if (o_mode = SAT_ZERO) then
                  fxp (WORD (REPLICATE (wordlength X) F),X)
            else if (o_mode = SAT_SYM) then
                  if (is_unsigned X) then bottomfxp X
                  else minustopfxp X
            else if (o_mode = WRAP) then
                  WRAP_AROUND X n_bits x
            else WRAP_AROUND_SM X n_bits x
      else Null
```

where *Null* is a constant that represents the result of an invalid operation, defined as:

```
⊢def  Null = @a. ¬ (Isvalid a)
```

Note that if the number is in the representable range of the given attributes, i.e. its value is neither greater than MAX nor less than MIN, then the overflow is meaningless and *Null* will be returned as the result.

4.5 Quantization

Fixed-point quantization takes an infinitely precise real number and converts it into a fixed-point number. Seven quantization modes are specified in Section 3.2.2, which we formalize using the following data type.

```
⊢def  quantization_mode =
       RND | RND_ZERO | RND_MIN_INF | RND_INF | RND_CONV | TRN | TRN_ZERO
```

Then we define the fixed-point quantization operation by a function, which is defined case by case on the quantization modes as follows:

```
⊢def  fxp_quantize X q_mode x =
       if (q_mode = RND) then
             closest value (λ a. value a ≥ x)
             {a | (Isvalid a) ∧ (Attrib a = X)} x
       else if (q_mode = RND_ZERO) then
             closest value (λ a. abs (value a) ≤ abs x)
             {a | (Isvalid a) ∧ (Attrib a = X)} x
       else if (q_mode = RND_MIN_INF) then
             closest value (λ a. value a ≤ x)
             {a | (Isvalid a) ∧ (Attrib a = X)} x
       else if (q_mode = RND_INF) then
             closest value
             (λ a. (if 0 ≤ x then value a ≥ x else value a ≤ x))
             {a | (Isvalid a) ∧ (Attrib a = X)} x
       else if (q_mode = RND_CONV) then
             closest value (λ a. LSB (String a) = F)
             {a | (Isvalid a) ∧ (Attrib a = X)} x
       else if (q_mode = TRN) then
             closest value (λ a. T)
             {a | (Isvalid a) ∧ (Attrib a = X) ∧ (value a ≤ x)} x
       else closest value (λ a. T)
            {a | (Isvalid a) ∧ (Attrib a = X) ∧
            (abs (value a) ≤ abs x)} x
```

The fixed-point quantization function takes as arguments a real number, a quantization mode, and an output attributes, and returns the corresponding fixed-point number. Similar to the floating-point case [15], its definition is based on the following predicate meaning that a is an element of the set s that provides a best approximation to x, assuming a valuation function v:

```
⊢def  is_closest v s x a =
       ((a IN s) ∧ ∀b. (b IN s) ⟹ (abs (v a − x) ≤ abs (v b − x)))
```

However, we still need to define a function that picks out a best approximation in case there are more than one closest number, based on a given property like *even*. This can be done in HOL as follows:

```
⊢def  closest v p s x =
       @a. ((is_closest v s x a) ∧
            ((∃b. (is_closest v s x b) ∧ (p b)) ⟹ (p a)))
```

Finally, we define the actual fixed-point rounding function for an arbitrary output attributes.

```
⊢def  fxp_round X o_mode q_mode n_bits x =
       if (x > MAX X ∨ x < MIN X) then
             ((fxp_overflow X o_mode n_bits x),overflow)
       else ((fxp_quantize X q_mode x),no_except)
```

where *fxp_overflow* is the fixed-point overflow function as defined in the previous section and supports all overflow modes, and *fxp_quantize* is the fixed-point quantization function that supports all quantization modes. The fixed-point rounding function takes as argument a real number, an output attributes, the quantization and overflow modes, and the number of saturated bits. It returns a fixed-point number and an exception flag. The function first checks for overflow, and in case of overflow returns the result based on the overflow mode, and sets the exception flag to *overflow*. Otherwise, it performs the quantization based on the quantization mode, and sets the exception flag to *no_except*.

4.6 Fixed-Point Arithmetic Operations

Fixed-point arithmetic operations such as addition or multiplication take two fixed-point input operands and store the result into a third. The attributes of the inputs and output need not match one another. Both unsigned and two's complement inputs and output are allowed. The result is formatted into the output as specified by the output attributes and by the overflow and loss of precision mode parameters. In our formalization, we first deal with exceptional cases such as invalid operation and loss of sign. If any of the input numbers is invalid, then the result is *Null* and the exception flag *invalid* is raised. If the result is negative but the output is unsigned then zero is returned and the exception flag *loss_sign* is raised. Also in the case of division by zero, the output value is forced to zero and the *invalid* flag is raised. Otherwise, we take the real value of the input arguments, perform the operation as infinite precision, then quantize the result according to the desired quantization and overflow modes. Formally, the operations for addition, subtraction, multiplication, and division are defined as follows:

```
⊢def fxpAdd X o_mode q_mode n_bits a b =
      if ¬(Isvalid a ∧ Isvalid b) then (Null,invalid)
      else if (value a + value b < 0 ∧ is_unsigned X) then
            (fxp (WORD (REPLICATE (wordlength X) F),X),loss_sign)
      else fxp_round X o_mode q_mode n_bits (value a + value b)

⊢def fxpSub X o_mode q_mode n_bits a b =
      if ¬(Isvalid a ∧ Isvalid b) then (Null,invalid)
      else if (value a − value b < 0 ∧ is_unsigned X) then
            (fxp (WORD (REPLICATE (wordlength X) F),X),loss_sign)
      else fxp_round X o_mode q_mode n_bits (value a − value b)

⊢def fxpMul X o_mode q_mode n_bits a b =
      if ¬(Isvalid a ∧ Isvalid b) then (Null,invalid)
      else if (value a * value b < 0 ∧ is_unsigned X) then
            (fxp (WORD (REPLICATE (wordlength X) F),X),loss_sign)
      else fxp_round X o_mode q_mode n_bits (value a * value b)

⊢def fxpDiv X o_mode q_mode n_bits a b =
      if ¬(Isvalid a ∧ Isvalid b) then (Null,invalid)
      else if (value b = 0) then
            (fxp (WORD (REPLICATE (wordlength X) F),X),invalid)
      else if (value a / value b < 0 ∧ is_unsigned X) then
            (fxp (WORD (REPLICATE (wordlength X) F),X),loss_sign)
      else fxp_round X o_mode q_mode n_bits (value a / value b)
```

5 Verification of Fixed-Point Operations

According to the discussion in Section 4.3, each fixed-point number has a corresponding real number value. The correctness of a fixed-point operation can be specified by comparing its output with the true mathematical result, using the valuation function *value* that converts a fixed-point to an infinitely precise number.

For example, the correctness of a fixed-point adder *fxpAdd* is specified by comparing it with its ideal counterpart $+$. That is, for each pair of fixed-point numbers *(a,b)*, we compare *value (a)* $+$ *value (b)* and *value (fxpAdd (a,b))*. In other words, we check if the diagram in Figure 4 commutes.

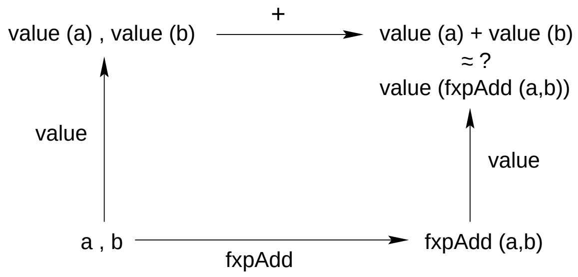


Figure 4: Correctness Criteria for Fixed-Point Addition

For this purpose we define the error resulting from quantizing a real number to a fixed-point value as follows:

```
⊢def  fxperror X o_mode q_mode n_bits x =
       value (FST (fxp_round X o_mode q_mode n_bits x)) − x
```

and then establish the correctness theorems for all four fixed-point arithmetic operations.

```
Theorem 1: FXP_ADD_THM
⊢  (Isvalid a) ∧ (Isvalid b) ∧ validAttr (X) ⟹
   (Isvalid (FST (fxpAdd (X) o_mode q_mode n_bits a b))) ∧
   (value (FST (fxpAdd (X) o_mode q_mode n_bits a b)) =
   value (a) + value (b) +
   (fxperror (X) o_mode q_mode n_bits (value (a) + value (b))))

Theorem 2: FXP_SUB_THM
⊢  (Isvalid a) ∧ (Isvalid b) ∧ validAttr (X) ⟹
   (Isvalid (FST (fxpSub X o_mode q_mode n_bits a b))) ∧
   (value (FST (fxpSub X o_mode q_mode n_bits a b)) =
   value (a) − value (b) +
   (fxperror X o_mode q_mode n_bits (value a − value b)))

Theorem 3: FXP_MUL_THM
⊢  (Isvalid a) ∧ (Isvalid b) ∧ validAttr (X) ⟹
   (Isvalid (FST (fxpMul X o_mode q_mode n_bits a b))) ∧
   (value (FST (fxpMul X o_mode q_mode n_bits a b)) =
   (value a * value b) +
   (fxperror X o_mode q_mode n_bits (value a * value b)))
```

```
Theorem 4: FXP_DIV_THM
⊢  (Isvalid a) ∧ (Isvalid b) ∧ validAttr (X) ⟹
   (Isvalid (FST (fxpDiv X o_mode q_mode n_bits a b))) ∧
   (value (FST (fxpDiv X o_mode q_mode n_bits a b)) =
   (value a / value b) +
   (fxperror X o_mode q_mode n_bits (value a / value b)))
```

The theorems are composed of two parts. The first part is about the validity of the fixed-point arithmetic operation output and states that if the input fixed-point numbers and the output attributes are valid then the result of the fixed-point operation is valid. The second part of the theorem relates the result of the fixed-point arithmetic operations to the real result based on the corresponding error function. To prove these main theorems, a number of lemmas have been established. We first proved lemmas concerning the approximation of a real number with a fixed-point number. We proved that in a finite non-empty set of fixed-point numbers, we can find the best approximation to a real number based on a given valuation function (*Lemma 1*).

```
Lemma 1: FXP_IS_CLOSEST_EXISTS
⊢  FINITE (s) ⟹  ¬(s = EMPTY) ⟹  ∃ (a: fxp). is_closest v s x a
```

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 (*Lemma 2*), and is itself a member of the set (*Lemma 3*), and is itself the best approximation of the real number (*Lemma 4*).

```
Lemma 2: FXP_CLOSEST_IS_EVERYTHING
⊢  FINITE (s) ⟹  ¬(s = EMPTY) ⟹
   is_closest v s x (closest v p s x) ∧
   ((∃b. is_closest v s x b ∧ p b) ⟹  p (closest v p s x))
```

```
Lemma 3: FXP_CLOSEST_IN_SET
⊢  FINITE (s) ⟹  ¬(s = EMPTY) ⟹  (closest v p s x) IN s
```

```
Lemma 4: FXP_CLOSEST_IS_CLOSEST
⊢  FINITE (s) ⟹  ¬(s = EMPTY) ⟹  is_closest v s x (closest v p s x)
```

Finally, we proved 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 (*Lemma 5*).

```
Lemma 5: IS_VALID_CLOSEST
⊢  (validAttr X) ⟹
   Isvalid (closest v p {a | Isvalid a ∧ ((Attrib a) = X)} x)
```

Besides, we proved that the set of all valid fixed-point numbers with a given attributes is finite (*Lemma 6*).

```
Lemma 6: FINITE_VALID_ATTRIB
⊢  FINITE {a | Isvalid a ∧ (Attrib a = X)}
```

The proof of this lemma is a bit complicated. For this purpose we made use of some built-in theorems about finite sets in the HOL *pred_sets* library [28]. 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[2] (*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 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 lemmas are proved by induction on the word length of the Boolean word and the maximum limit of the natural numbers, respectively.

We also proved that the set of all valid fixed-point numbers is nonempty (*Lemma 7*).

```
Lemma 7: IS_VALID_NONEMPTY
⊢  (validAttr X) ⟹  ¬({a | Isvalid a ∧ (Attrib a = X)} = EMPTY)
```

Finally, we proved that the result of quantizing a real number, which is in the range representable by a given valid attributes, is a valid fixed-point number (*Lemma 8*).

```
Lemma 8: IS_VALID_QUANTIZATION
⊢  (validAttr X) ⟹  Isvalid (FST (fxp_round X o_mode q_mode n_bits x))
```

The validity of the quantization directly implies validity of the fixed-point operation output, and this completes the proof of the first parts of the theorems. The second parts of the theorems are proved using the properties of the real arithmetic in HOL and rewriting with the definitions of the *fxpAdd, fxpSub, fxpMul, fxpDiv,* and *fxperror* functions.

The second main theorem on fixed-point error analysis concerns bounding the quantization error. The error can be absolutely quantified as follows:

[2] a set that contains precisely one element.

```
Theorem 5: FXP_ERROR_BOUND_THM
⊢  (validAttr X) ∧ ¬(x > MAX (X)) ∧ ¬ (x < MIN (X)) ⟹
   abs (fxperror X o_mode q_mode n_bits x) ≤ inv (&2 pow fracbits X)
```

According to this theorem, the error in quantizing a real number which is in the range representable by a given set of attributes X is less than the quantity $1 \ / \ 2^{fracbits\ (X)}$. This theorem is valid for all fixed-point quantization modes. However, for *RND, RND_ZERO, RND_MIN_INF, RND_INF*, and *RND_CONV* modes, which quantize to the nearest representable value, the error can be bounded to $1 \ / \ 2^{(fracbits\ (X)+1)}$ by extending the theorem.

To explain the theorem, we consider the following fact that 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 the non-negative rationals given by

$$P \ = \ \{p/2^b \ \mid \ 0 \ \leq \ p \ \leq \ 2^N \ - \ 1, \ p \ \in \ \mathbb{Z}\} \tag{6}$$

Similarly, for signed two's complement representation, we have

$$P \ = \ \{p/2^b \ \mid \ -2^{N-1} \ \leq \ p \ \leq \ 2^{N-1} \ - \ 1, \ p \ \in \ \mathbb{Z}\} \tag{7}$$

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 for each case as shown in Figure 5.

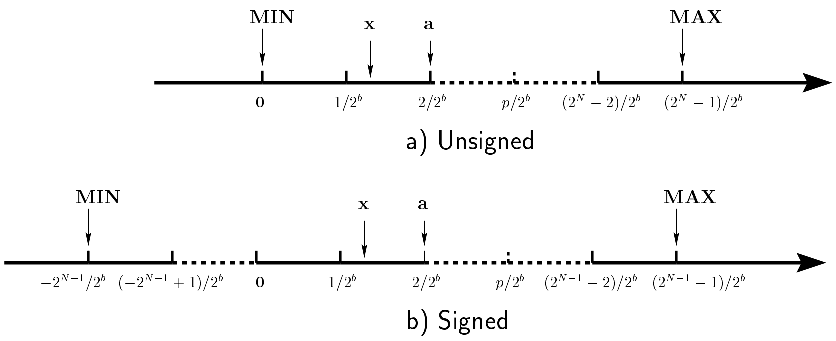


Figure 5: Fixed-Point Values on the Real Axis

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 Figure 5. 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 HOL, we first proved that the quantization result is the nearest value to a real number and the corresponding error is minimum compared to the other fixed-point numbers (*Lemma 9*).

```
Lemma 9: FXP_ERROR_AT_WORST_LEMMA
⊢  (validAttr X) ∧ ¬(x > MAX (X)) ∧ ¬(x < MIN (X)) ∧
   (Isvalid a) ∧ (Attrib a = X) ⟹
   abs (fxperror X o_mode q_mode n_bits x) ≤ abs (value a − x)
```

Then we proved that each representable real value x can be surrounded by two successive rational numbers (*Lemma 10*).

```
Lemma 10: FXP_ERROR_BOUND_LEMMA1
⊢  (validAttr X) ∧ ¬(x > MAX (X)) ∧ ¬(x < MIN (X)) ⟹
   ∃k. (k < 2 EXP wordlength X) ∧ (&k / (&2 pow fracbits X) ≤ x) ∧
   (x < (&(SUC k) / (&2 pow fracbits (X))))
```

Also we proved that the difference between the real number and the surrounding rationals is less than $1 \ / \ 2^{fracbits\ (X)}$ (*Lemma 11*).

```
Lemma 11: FXP_ERROR_BOUND_LEMMA2
⊢  (validAttr X) ∧ ¬(x > MAX (X)) ∧ ¬(x < MIN (X)) ⟹
   ∃k. (k ≤ 2 EXP wordlength X) ∧
   abs (x − &k / (&2 pow (fracbits (X)))) ≤ inv (&2 pow (fracbits (X)))
```

Finally, we proved that for each real value we can find a fixed-point number with the required error characteristics (*Lemma 12*).

```
Lemma 12: FXP_ERROR_BOUND_LEMMA3
⊢  (validAttr X) ∧ ¬(x > MAX (X)) ∧ ¬(x < MIN (X)) ⟹ ∃(w: bool word).
   abs (value (fxp (w,X)) − x) ≤ inv (&2 pow (fracbits X)) ∧
   (WORDLEN w = wordlength X)
```

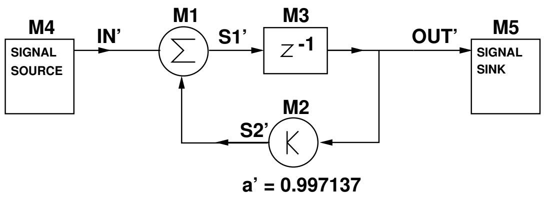
Since the quantization produces the minimum error as stated in *Lemma 9*, the proof of the second main theorem (*Theorem 5*) is a direct consequence of *Lemma 12*. In these proofs, we have treated the case of signed and unsigned numbers separately since they have different definitions for *MAX*, *MIN*, and *value* functions. For signed numbers a special attention needs also to be paid to deal with negative numbers.

6 Application with SPW

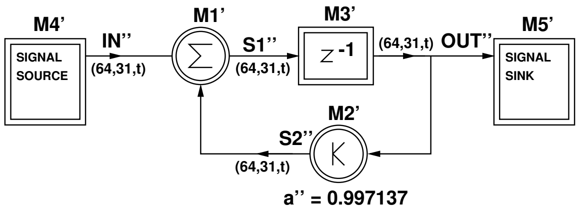
In this section we demonstrate how to apply the formalization of fixed-point arithmetic presented in the previous sections for the verification of the transition from floating-point to fixed-point algorithmic levels. We have chosen SPW as application tool and the case of an *Integrator* as an example circuit. A digital integrator is a discrete time system that transforms a sequence of input numbers into another sequence of output, by means of a specific computational algorithm. To describe the general functionality of a digital integrator, let $\{x_t\}$, $\{w_t\}$, and a denote the input sequence, output sequence, and constant coefficient of the integrator, respectively. Then the integrator can be specified by the difference equation:

$$w_t = x_{t-1} + a\ w_{t-1} \tag{8}$$

Thereafter, the output sequence at time t is equal to the input sequence at time *t - 1*, added to the output at time *t - 1* multiplied by the integrator coefficient.



a) Floating-Point Design



b) Fixed-Point Design

Figure 6: SPW Design of an Integrator

Figure 6 shows the SPW design of an integrator. The integrator is first designed and simulated using the SPW predefined floating-point blocks and parameters (Figure 6 (a)). The design is composed of an adder (*M1*), a multiplier by constant (*M2*), and a delay (*M3*) block, together with signal source (*M4*) and sink (*M5*) elements. The input signal, the output signal, and the output of the adder and multiplier blocks are labeled by *IN', OUT', S1'*, and *S2'*, respectively. Figure 6 (b) shows the converted fixed-point design in which each block is replaced with the corresponding fixed-point block (*M1', M2', M3', M4', M5'*). Fixed-point blocks are shown by double circles and squares to distinguish them from the floating-point blocks. The attributes of all fixed-point block outputs are set to $(64, 31, t)$ to ensure that overflow and quantization do not affect the system operation. The corresponding fixed-point signals are labeled by *IN", OUT", S1"*, and *S2"*.

In HOL, we first model the design at each level as predicates in higher-order logic. The predicates corresponding to the floating-point design are as follows:

```
⊢def  Float_Gain_Block a′ b′ c′ = (∀t. c′ t = a′ t float_mul b′)
⊢def  Float_Delay_Block a′ b′ = (∀t. b′ t = a′ (t − 1))
⊢def  Float_Add_Block a′ b′ c′ = (∀t. c′ t = a′ t float_add b′ t)

⊢def  Float_Integrator_Imp X a′ IN′ OUT′ =
      ∃ S1′ S2′.
      Float_Add_Block IN′ S2′ S1′ ∧
      Float_Delay_Block S1′ OUT′ ∧
      Float_Gain_Block OUT′ a′ S2′
```

where X is the floating-point format. In these definitions, we have used available formalization of floating-point arithmetic in HOL [15]. Floating-point data types are stored in SPW in the standard IEEE 64 bit double precision format.

The HOL description of the fixed-point implementation is as follows:

```
⊢def  Fxp_Gain_Block a″ b″ c″ = (∀t. c″ t = a″ t fxp_mul b″)
⊢def  Fxp_Delay_Block a″ b″ = (∀t. b″ t = a″ (t − 1))
⊢def  Fxp_Add_Block a″ b″ c″ = (∀t. c″ t = a″ t fxp_add b″ t)

⊢def  Fxp_Integrator_Imp X′ o_mode q_mode n_bits a″ IN″ OUT″ =
      ∃ S1″ S2″.
      Fxp_Add_Block IN″ S2″ S1″ ∧
      Fxp_Delay_Block S1″ OUT″ ∧
      Fxp_Gain_Block OUT″ a″ S2″
```

where *X'* is the fixed-point format, and the functions *fxp_add* and *fxp_mul* are defined as follows:

```
⊢def  a″ fxp_add b″ = FST (fxpAdd X′ o_mode q_mode n_bits a″ b″)
⊢def  a″ fxp_mul b″ = FST (fxpMul X′ o_mode q_mode n_bits a″ b″)
```

In the next step, we describe each design as a difference equation relating the input and output samples according to the equation (8).

```
⊢def  FLOAT_Integrator_Spec X a′ IN′ OUT′ =
     ∀t. OUT′ t = (IN′ (t − 1) float_add (a′ float_mul OUT′ (t − 1)))

⊢def  FXP_Integrator_Spec X′ o_mode q_mode n_bits a″ IN″ OUT″ =
     ∀t. OUT″ t = (IN″ (t − 1) fxp_add (a″ fxp_mul OUT″ (t − 1)))
```

The following lemmas ensure that the implementation at each level satisfies the corresponding specification.

```
Lemma 13: FLOAT_INTEGRATOR_IMP_SPEC
⊢  Float_Integrator_Imp X a′ IN′ OUT′ ⟹
   Float_Integrator_Spec X a′ IN′ OUT′

Lemma 14: FXP_INTEGRATOR_IMP_SPEC
⊢  Fxp_Integrator_Imp X′ o_mode q_mode n_bits a″ IN″ OUT″ ⟹
   Fxp_Integrator_Spec X′ o_mode q_mode n_bits a″ IN″ OUT″
```

Now we assume that the floating-point and fixed-point input sequences are the rounded versions of an infinite precision ideal input *IN*, and we have

```
⊢def  IN′ t = round X To_nearest (IN t)
⊢def  IN″ t = FST (fxp_round X′ o_mode q_mode n_bits (IN t))
```

where *round* is the floating-point rounding function, and *To_nearest* is the corresponding mode for rounding to nearest floating-point number [15]. We also make some other assumptions on finiteness and validity of floating-point and fixed-point inputs, coefficients, and intermediate results, in order to have finite and valid final outputs. Using these assumptions and based on the theorems *FXP_ADD_THM* and *FXP_MUL_THM* (Section 5) and the corresponding ones in floating-point theory [15], we prove the following theorem concerning the error between the real values of the floating-point and fixed-point precision integrator output samples.

```
Theorem 6: INTEGRATOR_THM
⊢  Float_Integrator_Imp X a′ IN′ OUT′ ∧
   Fxp_Integrator_Imp X′ o_mode q_mode n_bits a″ IN″ OUT″
   ⟹
   Val (OUT′ t) − value (OUT″ t) =
   Val a′ * Val (OUT′ (t − 1)) −
   value a″ * value (OUT″ (t − 1)) +
   error (IN (t − 1)) +
   error (Val a′ * Val (OUT′ (t − 1))) +
   error (Val (IN′ (t − 1)) + Val (a′ float_mul OUT′ (t − 1))) +
   fxperror X′ o_mode q_mode n_bits
   (value (value a″ * OUT″ (t − 1))) +
```

```
fxperror X′ o_mode q_mode n_bits
(value (IN″ (t − 1)) + value (a″ fxp_mul OUT″ (t − 1))) −
fxperror X′ o_mode q_mode n_bits (IN (t − 1))
```

where *Val* is the floating-point valuation function, and *error* is the floating-point rounding error function [15]. According to *Theorem 6*, for a valid and finite set of input and output sequences at time *(t - 1)* to the integrator design at the floating-point and fixed-point levels, we can have finite and valid outputs at time t, and the difference in the real values corresponding to these output samples can be expressed as the difference in input and output values multiplied by the corresponding coefficients, taking into account the effects of finite precision in coefficients and arithmetic operations. To find a constant upper bound for the difference between the outputs, we use *Theorem 5* on the fixed-point error quantification. Similarly, for the floating-point error bound analysis we proved the following lemma:

```
Lemma 15: ERROR_BOUND_NORM_STRONG_NORMALIZE
⊢ normalizes X x ⟹
   ∃ j. abs (error x) ≤ (2 pow j / 2 pow (bias X + fracwidth X))
```

where *normalizes* defines the criteria for an arbitrary real number to be in the range of normalized floating-point numbers, *bias* defines the exponent bias in the floating-point format which is a constant used to make the exponent's range non-negative, and *fracwidth* extracts the fraction width parameter from the floating-point format. According to *Lemma 15*, if the absolute value of a real number is in the representable range of the normalized floating-point numbers with the format X and located in the j'th binade (the floating-point numbers between two adjacent powers of 2), then the absolute value of the error is less than or equal to $2^j/2^{(bias\ X\ +\ fracwidth\ X)}$. The lemma is proved based on the general floating-point absolute error bound theorem developed in [15].

Finally, we proved the following theorem (*Theorem 7*) that bounds the output error of the integrator design in the transition from the floating-point to fixed-point levels.

```
Theorem 7: INTEGRATOR_FP_TO_FXP_ERROR_BOUND_THM
⊢ Float_Integrator_Imp X a′ IN′ OUT′ ∧
  Fxp_Integrator_Imp X′ o_mode q_mode n_bits a″ IN″ OUT″
  ⟹
  ∃ j1 j2 j3.
  abs (Val (OUT′ t) − value (OUT″ t)) ≤
  2 * abs (a) * M +
  (2 pow j1 + 2 pow j2 + 2 pow j3) / 2 pow (bias X + fracwidth X) +
  3 / (2 pow (fracbits X′))
```

In the proof of this theorem, we have assumed that the real values of the floating-point and fixed-point integrator coefficients are equal (*Val a' = value a"* $= a$), hence ignoring the effects of inaccuracies in the integrator coefficient. We have also assumed that the floating-point and fixed-point output values are bounded to a constant value (M). The parameters *j1, j2,* and *j3* are related to the binades in which the real valued arguments of the three floating-point error expressions in *Theorem 6* are located.

7 Conclusions

In this paper, we established the formalization of fixed-point arithmetic in the HOL theorem prover. Unlike floating-point arithmetic, there is no standard for the fixed-point counterpart. We hence defined in this paper a complete common set of the fixed-point arithmetic supported by most DSP tools, in particular SPW and SystemC. We started first by encoding the fixed-point arithmetic in HOL considering different quantization and overflow modes, as well as exception handling. We then proved two main theorems stating that the operations on fixed-point numbers are closely related to the corresponding operations on infinitely precise values, considering some error. The error is bounded to a certain absolute value which is a function of the output precision. We have also shown by an example how these theorems can be used as a basis for analysis of the quantization errors in the design of fixed-point DSP subsystems. The formalization presented in this paper can be considered as a complement to the floating-point formalizations which are widely available in the literature. Based on the proposed fixed-point formalization, our immediate future work will focus on the verification of the transition from the floating-point algorithmic level to hardware implementations for DSP applications.

References

[1] M. D. Aagaard and C. -J. H. Seger, "The Formal Verification of a Pipelined Double-Precision IEEE Floating-Point Multiplier," In Proceedings International Conference on Computer Aided Design, pp. 7-10, San Jose, California, USA, November 1995.

[2] G. Barrett, "Formal Methods Applied to a Floating Point Number System," IEEE Transactions on Software Engineering, SE-15 (5): 611-621, May 1989.

[3] C. Berg and C. Jacobi, "Formal Verification of the VAMP Floating Point Unit," In Correct Hardware Design and Verification Methods, LNCS 2144, pp. 325-339, Springer-Verlag, 2001.

[4] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W. J. Paul, "Instantiating Uninterpreted Functional Units and Memory System: Functional Verification of the VAMP," In Correct Hardware Design and Verification Methods, LNCS 2860, pp. 51-65, Springer-Verlag, 2003.

[5] S. Boldo, M. Daumas, and L. Thèry, "Formal Proofs and Computations in Finite Precision Arithmetic," In Proceedings of the 11th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, pp. 101-111, Rome, Italy, September 2003.

[6] S. Boldo and M. Daumas, "Properties of Two's Complement Floating Point Notations," Software Tools for Technology Transfer, 5 (2-3): 237-246, March 2004.

[7] Cadence Design Systems, Inc., "Signal Processing WorkSystem (SPW) User's Guide," USA, July 1999.

[8] V. A. Carreno, "Interpretation of IEEE-854 Floating-Point Standard and Definition in the HOL System," NASA TM-110189, September 1995.

[9] M. Cornea-Hasegan, "Proving the IEEE Correctness of Iterative Floating-Point Square Root, Divide, and Remainder Algorithms," Intel Technology Journal, Q2: 1-11, 1998.

[10] Y. -A. Chen and R. E. Bryant, "Verification of Floating Point Adders," In Computer Aided Verification, LNCS 1427, pp. 488-499, Springer-Verlag, 1998.

[11] M. Daumas, L. Rideau, and L. Thèry, "A Generic Library for Floating-Point Numbers and Its Application to Exact Computing," In Theorem Proving in Higher Order Logics, LNCS 2152, pp. 169-184, Springer-Verlag, 2001.

[12] M. J. C. Gordon and T. F. Melham, "Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic," Cambridge University Press, 1993.

[13] J. R. Harrison, "Constructing the Real Numbers in HOL," Formal Methods in System Design, 5 (1/2): 35-59, 1994.

[14] J. R. Harrison, "A Machine-Checked Theory of Floating-Point Arithmetic," In Theorem Proving in Higher Order Logics, LNCS 1690, pp. 113-130, Springer-Verlag, 1999.

[15] J. R. Harrison, "Floating-Point Verification in HOL Light: The Exponential Function," Formal Methods in System Design, 16 (3): 271-305, 2000.

[16] J. R. Harrison, "Formal Verification of Floating Point Trigonometric Functions," In Formal Methods in Computer-Aided Design, LNCS 1954, pp. 217-233, Springer-Verlag, 2000.

[17] J. R. Harrison, "Formal Verification of IA-64 Division Algorithms," In Theorem Proving in Higher Order Logics, LNCS 1869, pp. 234-251, Springer-Verlag, 2000.

[18] The Institute of Electrical and Electronic Engineers, Inc., "IEEE, Standard for Binary Floating-Point Arithmetic," ANSI/IEEE Standard 754, USA, 1985.

[19] The Institute of Electrical and Electronic Engineers, Inc., "IEEE, Standard for Radix-Independent Floating-Point Arithmetic," ANSI/IEEE Std 854, USA, 1987.

[20] R. Kaivola and M. D. Aagaard, "Divider Circuit Verification with Model Checking and Theorem Proving," In Theorem Proving in Higher Order Logics, LNCS 1869, pp. 338-355, Springer-Verlag, 2000.

[21] R. Kaivola and N. Narasimhan, "Formal Verification of the Pentium® 4 Floating-Point Multiplier," In Proceedings Design Automation and Test in Europe Conference, pp. 20-27, Paris, France, March 2002.

[22] R. Kaivola and K. R. Kohatsu, "Proof Engineering in the Large: Formal Verification of Pentium® 4 Floating-Point Divider," Software Tools for Technology Transfer, 4 (3): 323-334, 2003.

[23] H. Keding, M. Willems, M. Coors, and H. Meyr, "FRIDGE: A Fixed-Point Design and Simulation Environment," In Proceedings Design Automation and Test in Europe Conference, pp. 429-435, Paris, France, February 1998.

[24] M. Leeser and J. O'Leary, "Verification of a Subtractive Radix-2 Square Root Algorithm and Implementation," In Proceedings International Conference on Computer Design, pp. 526-531, Austin, Texas, USA, October 1995.

[25] Mathworks, Inc., "Simulink Reference Manual," USA, 1996.

[26] Mathworks, Inc., "Fixed-Point Blockset, For Use with Simulink, User's Guide," USA, 2004.

[27] Mentor Graphics, Inc., "DSP Station User's Manual," USA, 1993.

[28] T. F. Melham, "The HOL pred_sets Library," University of Cambridge, Computer Laboratory, February 1992.

[29] P. S. Miner, "Defining the IEEE-854 Floating-Point Standard in PVS," NASA TM-110167, June 1995.

[30] P. S. Miner and J. F. Leathrum, "Verification of IEEE Compliant Subtractive Division Algorithms," In Formal Methods in Computer-Aided Design, LNCS 1166, pp. 64-78, Springer-Verlag, 1996.

[31] 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, 47 (9): 913-926, 1998.

[32] S. M. Mueller and W. J. Paul, "Computer Architecture. Complexity and Correctness," Springer-Verlag, 2000.

[33] Open SystemC Initiative, "SystemC Language Reference Manual," USA, 2004.

[34] J. O' Leary, X. Zhao, R. Gerth, and C.-J.H. Seger, "Formally Verifying IEEE Compliance of Floating-Point Hardware," Intel Technology Journal, Q1: 1-14, 1999.

[35] 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, LNCS 1954, pp. 3-36, Springer-Verlag, 2000.

[36] J. Sawada and R. Gamboa, "Mechanical Verification of a Square Root Algorithm using Taylor's Theorem," In Formal Methods in Computer-Aided Design, LNCS 2517, pp. 274-291, Springer-Verlag, 2002.

[37] Synopsys, Inc., "CoCentric^{TM} System Studio User's Guide," USA, August 2001.

[38] University of Cambridge, "The HOL System Reference," Computer Laboratory, Cambridge, UK, March 2004.

[39] W. Wong, "Modeling Bit Vectors in HOL: The Word Library," In Higher Order Logic and Its Applications, LNCS 780, pp. 371-384, Springer-Verlag, 1994.