Formalization of Cadence SPW Fixed-Point Arithmetic in HOL

Behzad Akbarpour, Sofiéne Tahar, Abdelkader Dekdouk

Electrical and Computer Engineering Department

Concordia University, Montreal, Canada

Email: {behzad, tahar, adekdouk}@ece.concordia.ca

Technical Report

Abstract

In this report we propose the formalization in higher-order logic of fixed-point arithmetic based on the SPW (Signal Processing WorkSystem) tool. We encoded the fixed-point number system and specified the different rounding modes in fixed-point arithmetic such as the directed and even rounding 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 rounding and to verify the basic arithmetic operations, 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 the floating-point to fixed-point algorithmic levels in the design flow of signal processors.

Keywords: Fixed-point, Floating-point, Signal Processing, Theorem Proving, HOL, SPW

Contents

1 Introduction **3**

2 Related Work **4**

3 SPW Fixed-Point Arithmetic **5**
3.1 Fixed-Point Numbers . . . 6
3.2 Fixed-Point Operations . . . 6

4 Formalizing Fixed-Point Arithmetic in HOL **8**
4.1 Fixed-Point Numbers Representation . . . 8
4.2 Fixed-Point Type . . . 9
4.3 Valuation . . . 10
4.4 Exception Handling . . . 11
4.5 Rounding . . . 11
4.6 Arithmetic Operations . . . 12

5 Verification of SPW Arithmetic Operations **12**

6 The Integrator Example **15**

7 Conclusions **17**

A Rounding to Fixed-Point Values **20**

B Lemmas for Analyzing the Fixed-Point Rounding Operation **21**

C Rounding Error in Fixed-point Arithmetic Operations **21**

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- to fixed-point precision. This implementation is optimized following some application specific trade-offs such as the speed, cost, area and the 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 [27]. 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 described by 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 higher level algorithmic descriptions of DSP designs modeled in floating- and fixed-point representations. Contrary to the floating-point arithmetic which is standardized in IEEE-754 [13] and IEEE-854 [14], the fixed-point arithmetic description depends on the tool we use to design the DSP (Digital Signal Processor) chip. Examples of such tools are SPW (Cadence) [26], Matlab-Simulink (Mathworks) [17], CoCentric (Synopsys) [25], DSP Station (Frontier Design) [18]. For instance, in SPW (Signal Processing Worksystem) [26], a fixed-point number is defined as a binary stream and a set of attributes. Attributes specify how the binary stream 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 rounding modes. In Matlab Simulink Fixed-Point Blockset [17], 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 rounding 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 [25], which uses fixed-point as described in the SystemC language [24]. 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 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. In this paper, we consider Cadence SPW [26] fixed-point arithmetic but the idea can be extended to cover other conventions.

In higher-order logic, the fixed-point number is encoded by a pair composed of a boolean stream, and a triple indicating the stream length, the length of the integer portion and the sign format. Then we formalize the concepts of valuation and rounding as functions that convert respectively a fixed-point number to a real and vice versa, taking into account different rounding and overflow modes. 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 result. Finally, we prove different lemmas regarding the error analysis of the fixed-point quantization and correctness of the basic operations like

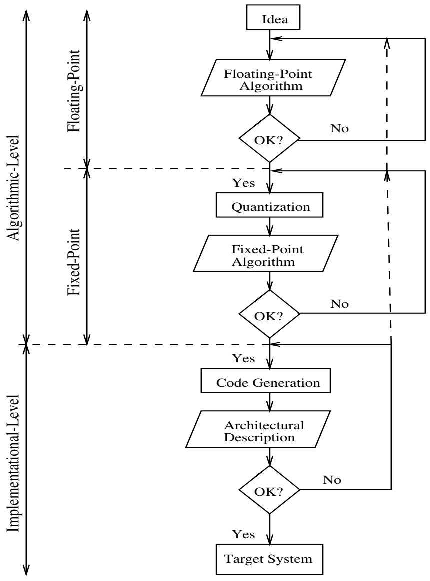

Figure 1: DSP Design Flow

addition, multiplication, and division. The higher-order logic formalization and proof were done using the HOL theorem prover [8].

The organization of this paper is as follows: Section 2 gives a review on the related work. Section 3 describes the SPW fixed-point arithmetic including the format of the fixed-point numbers, arithmetic operations, exceptions detection and their handling, and the different rounding modes. Section 4 describes in detail their formalization in HOL. In Section 5, we discuss the verification of the SPW basic arithmetic operations. 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 works in the open literature on the formalization and verification of IEEE standard based floating-point arithmetic. For instance, Barett [2] specified parts of the IEEE-754 standard in Z, and Miner [20] 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 in formalization of floating-point standards using theorem proving.

Carreno [4] 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; then these tables were 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 [9] defined and formalized real numbers using HOL. He then developed a generic floating-point library [11] 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 against behavioral specification such as the square root and the exponential function [10].

Moore et al. [21] have verified the AMD-K5 floating-point division algorithm using the ACL2 theorem prover. Also, Russinoff [23] 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.

More recently, Daumas et al. [7] have presented a generic library for reasoning about floating-point numbers within the Coq system. Berg et al. [3] have formally verified a theory of IEEE rounding presented in [22] 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. They have used a formal definition of rounding based on Miners formalization of standard.

Aagaard and Seger [1] combined BDD based methods and theorem proving techniques to verify a floating-point multiplier. Chen and Bryant [6] used word-level SMV to verify a floating-point adder. Miner and Leathrum [19] verified a general class of subtractive division algorithms with respect to the IEEE-754 standard in PVS. Leeser et al. [15] verified a radix-2 square root algorithm and its hardware implementation using theorem proving methods. Cornea-Hasegan [5] 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. [16] reported on the verification of the Intel's FPU at the gate level using a combination of model-checking and theorem proving.

While the above works are all concerned with floating-point representation and arithmetic, there is no machine-checked formal development on 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 [10] and Carreno [4] on the floating-point formalization. Harrison's work was more oriented to verification purposes. Indeed we used an analogous set of lemmas to his work, to check the validity of operation results and to do the error analysis on our definition of the fixed-point rounding. For exception handling which is not coverd by Harrison, we followed Carreno who formalized exceptions and their handling in more details.

3 SPW Fixed-Point Arithmetic

In this section we describe SPW (Signal Processing WorkSystem) based fixed-point arithmetic. SPW is an integrated framework that provides a unified graphical interface for all aspects of system design, simulation, and implementation for developing DSP based products. It graphically represents a system as a network of functional blocks. SPW design flow supports different abstraction levels starting from floating-point algorithmic description, converting that to an optimized fixed-point specification from which an RTL (Register Transfer Level) implementation (in VHDL or Verilog) is generated (Figure 1).

3.1 Fixed-Point Numbers

A fixed-point number in SPW has a fixed number of binary digits and a fixed position for the decimal point with respect to that set 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 signal values. 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 signal values, 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 signal values are expressed as a binary stream and a set of attributes, $Stream < Set\ of\ Attributes >$. The attributes specify how the binary stream is interpreted. Generally, the attributes are specified in the following format:

$$< \#bits, \#integer\_bits, sign\_format >$$

- **#bits:** The total number of bits used to represent the fixed-point stream, including integer bits, fractional bits, and sign bit, if any. The total number of bits must be in the range of 1 to 256.

- **#integer_bits:** 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 number of bits, trailing zeroes are added to generate the equivalent binary value.

- **sign_format:** A letter specifying the sign format: "u" for unsigned, "t" for two's complement.

For example the signal 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. $1111 < 4, -1, u >$ represents the value .01111 which is 0.46875.

3.2 Fixed-Point Operations

The SPW library includes 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. In addition to the parameters corresponding to the input operands and output result, the arithmetic operations take two specific parameters defining the overflow and rounding (loss of precision) modes.

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

- **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.

The overflow-handling parameter is used to indicate how to handle overflow errors. The parameter is a string that should be set to:

- **Wrap:** Overflow bits are ignored, or
- **Clip (default):** Values are saturated. The closest representable value to the true result is returned.

The loss of precision handling parameter determines the rounding mode. The parameter names are patterned after the IEEE standards for floating-point arithmetic [13]. The loss of precision parameter values are:

- **Round (default):** The block uses the nearest representable value on either side of the original value. If the original value is exactly halfway between the two nearest representable values, the block rounds toward plus infinity.
- **Round_To_Zero:** The block uses the nearest representable value towards zero.
- **Round_To_Plus_Infinity:** The block uses the nearest representable value towards plus infinity.
- **Truncate or Round_To_Minus_Infinity:** The block discards the trailing bits or equivalently, uses the nearest representable value towards minus infinity. Both parameter settings result in the same behavior, so we can use them interchangeably.
- **Convergent_Round:** The block uses the nearest representable value on either side of the original value. If the original value is exactly halfway between the two nearest representable values, the block uses the "even" value (the value having zero as the least significant bit). Figure 2 summarizes the loss of precision modes.

For example, consider a block that serves as a primitive fixed-point multiplier that truncates the results when loss of precision occurs and wraps the result when overflow occurs. We can make the multiplier routine by calling the function *fxpMul* (*Wrap* | *Truncate*, *In1*, *In2*, *Out*).

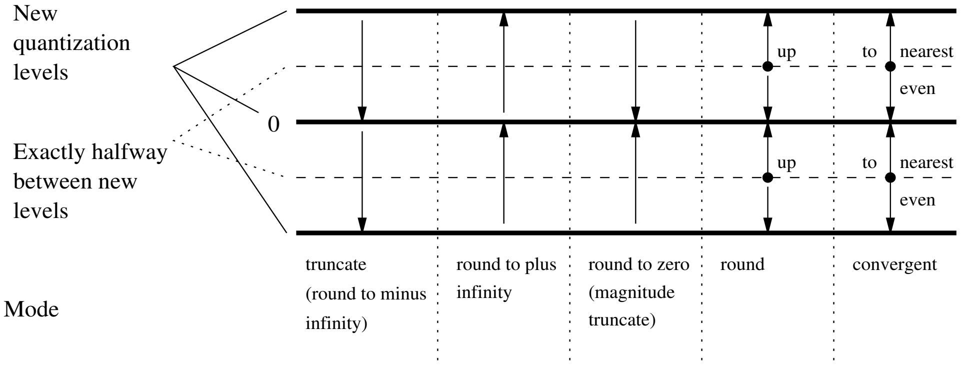

Figure 2: Loss of Precision Modes

4 Formalizing Fixed-Point Arithmetic in HOL

In this section, we present the formalization of SPW based fixed-point arithmetic in higher-order logic, based on the general purpose HOL theorem prover [8]. The 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 John Harrison's reals library [9] which proved to be essential for our fixed-point arithmetic formalization. Table 1 summarizes some of the HOL symbols used in this paper and their meanings.

Table 1. HOL Symbols

Standard symbol	**HOL notation**	**Meaning**
ε	@	Hilbert choice
(none)	&	Natural map $\aleph \to \Re$
(none)	$c \to a \mid b$	if c then a else b
$-x$	$\neg x$	Unary negation of x
$x^{-}1$	inv (x)	Multiplicative inverse of x
$\mid x \mid$	$abs\ (x)$	Absolute value of x
x^n	x pow n	Real x raised to natural number power n

4.1 Fixed-Point Numbers Representation

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

$\vdash_{def}$ `stream (v,a) = v`
$\vdash_{def}$ `attrib (v,a) = a`

The binary stream 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 [28] to facilitate the manipulation of binary words. The attributes are represented by a triple 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  streamlength (b,ib,st) = b
⊢def  intbits (b,ib,st) = ib
⊢def  signtype (b,ib,st) = st
```

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

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

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

```
⊢def  fracbits X = (is_unsigned X →  (streamlength X − intbits X) |
     (streamlength 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 *streamlength* should be in the range of 1 and 256, the *signtype* can be either 0 or 1, and the number of integer bits is less than or equal to the *streamlength*. A valid fixed-point number must have a valid set of attributes and the length of the binary stream must be equal to *streamlength*.

```
⊢def  validAttr X = (streamlength X > 0) ∧ (streamlength X < 257) ∧
     (signtype X < 2) ∧ (intbits X < (streamlength X + 1))

⊢def  is_valid (a: (bool word,(num,num,num))) = (validAttr (attrib a)) ∧
     (WORDLEN (stream a) = streamlength (attrib a))
```

4.2 Fixed-Point Type

Now we define the actual HOL type for fixed-point numbers. The type is defined to be in bijection with the appropriate subset of $bool\ word \times \aleph^3$, with the bijections written in HOL as $Fxp : (bool\ word\ ,\ (num\ ,\ num\ ,\ num)) \rightarrow fxp$ and $deFxp : fxp \rightarrow (bool\ word\ ,\ (num\ ,\ num\ ,\ num))$:

```
⊢def  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, e.g:

```
⊢def  Stream (a: fxp) = stream (deFxp a)
⊢def  Issigned (a:fxp) = is_signed (Attrib a)
⊢def  Isvalid (a:fxp) = is_valid (deFxp a)
```

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

4.3 Valuation

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

Unsigned:

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

Signed:

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

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

```
⊢def  value (a: fxp) =
     Isunsigned a ⟹  & (BNVAL (Stream a)) / (&2 pow Fracbits a) |
     (& (BNVAL (Stream a)) − &(((BV (MSB (Stream a)))) *
     (2 EXP (Streamlength a)))) / (&2 pow Fracbits a)
```

BNVAL is a predefined function of HOL *word* library 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 most significant bit of a word.

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:

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

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:

$$MIN \; = \; - \; 2^a$$

So, we obtain the corresponding functions in HOL.

```
⊢def  MAX (X) = &2 pow intbits (X) − inv (&2 pow fracbits (X))
⊢def  MIN (X) = is_unsigned (X) →  &0 | ¬(&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 =
     (is_unsigned (X) →  Fxp (WORD (REPLICATE (streamlength (X)) T),X) |
     Fxp (WCAT ((WORD [ F ]),
     (WORD (REPLICATE ((streamlength (X)) − 1) T))),X))

⊢def  bottomfxp X =
     (is_unsigned (X) →  Fxp (WORD (REPLICATE (streamlength (X)) F),X) |
     Fxp (WCAT ((WORD [ T ]),
     (WORD (REPLICATE ((streamlength (X)) − 1) F))),X))
```

WCAT, the concatenation of two words, is a predefined HOL function.

[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 a result of performing the operation. Exceptions are described in Section 3.2 and declared as a new HOL data type:

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

no_except is reserved for the case without exception.

Two overflow modes are also represented via an enumerated type definition:

$\vdash_{def}$ `overflowmode = Wrap | Clip`

According to the definition of overflow modes in Section 3.2 for clipping, 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 wrapping, we must first convert the real number to binary format, and then discard the extra bits according to the output attributes.

4.5 Rounding

Rounding takes an infinitely precise real number and converts it into a fixed-point number. Five rounding modes are specified in Section 3.2 according to SPW documentation, which we formalize using the following datatype.

$\vdash_{def}$
```
roundmode =
  Round | To_zero | To_plus_infinity | Truncate | Convergent
```

The rounding operation is defined by a family of functions which are defined case by case on the rounding modes. The functions take as arguments a real number, rounding and overflow modes, and output attributes. They return a fixed-point number and an exception flag. The real number value may be outside the representable range of finite fixed-point numbers. In this case, the rounding function must check for overflow and return an overflow exception flag if detected. Similar to the floating-point case [10], 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:

$\vdash_{def}$
```
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, using the Hilbert choice operator, a best approximation in case there are more than one closest number, based on a given property like *even*.

$\vdash_{def}$
```
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 rounding function for an arbitrary fixed-point output attributes. We only show the clause for *Convergent* rounding mode [2].

$\vdash_{def}$
```
(fxp_round (X) Convergent mode x =
  (x > MAX X) ∧ (mode = Clip) →  (topfxp (X) , overflow) |
  (x < MIN X) ∧ (mode = Clip) →  (bottomfxp (X) , overflow) |
  (x > MAX X) ∧ (mode = Wrap) →  (Wrapp (X) x , overflow) |
  (x < MIN X) ∧ (mode = Wrap) →  (Wrapp (X) x , overflow) |
  (closest (value) (λ a. LSB (Straem a) = F)
  {(a: fxp) | Isvalid a ∧ (Attrib a = X)} x, no_except))
```

[2]see Appendix A

4.6 Arithmetic Operations

In accordance with the SPW description of fixed-point arithmetic functions, operations such as addition or multiplication take two fixed-point input signals and store their 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 an exception flag *invalid* is raised.

Null is a constant that represents the result of an invalid operation, as follows:

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

If the result is negative but the output is unsigned then zero is returned and exception flag is *loss_sign*. 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 round the result according to the desired rounding and overflow modes. Formally, the operations for addition and multiplication are defined as follows:

```
⊢def  fxpAdd (X) rnd over a b =
     ¬(Isvalid a ∧ Isvalid b) →  (Null,invalid) |
     (value a + value b < &0) ∧ (is_unsigned X) →
     (Fxp (WORD (REPLICATE (streamlength X) F), X), loss_sign) |
     (fxp_round (X) rnd over (value a + value b))
```

```
⊢def  fxpMul (X) rnd over a b =
     ¬(Isvalid a ∧ Isvalid b) →  (Null,invalid) |
     (value a * value b < &0) ∧ (is_unsigned X) →
     (Fxp (WORD (REPLICATE (streamlength X) F), X), loss_sign) |
     (fxp_round (X) rnd over (value a * value b))
```

5 Verification of SPW Arithmetic 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* which 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 3 commutes. In following, we discuss the addition operation. Other operations can be handled similarly [3].

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

```
⊢def  fxperror (X) rnd over x = (value (FST (fxp_round (X) rnd x over)) − x)
```

and then establish the correctness theorem as follows:

[3] see Appendix C

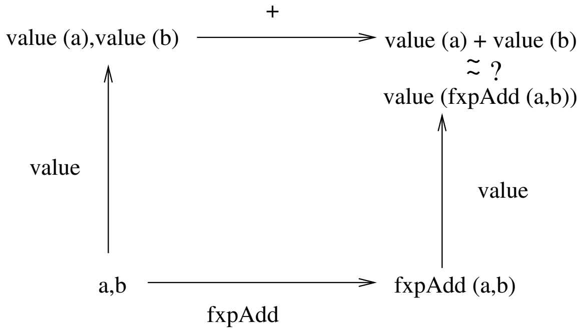

Figure 3: Correctness Criteria in Fixed-Point Addition

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

The theorem is composed of two parts. The first part is about the validity of fixed-point addition output and states that if the input fixed-point numbers and the output attributes are valid then the result of fixed-point addition is valid. The second part of the theorem relates the result of fixed-point addition to the real result based on the corresponding error function. To prove this main theorem, a number of lemmas have been established[4].

The following ensures the existence of the best approximation to a given real number in a finite non empty set of fixed-point numbers:

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

Next lemma states that the chosen best approximation satisfying a property p is itself the best approximation of a given real number:

```
Lemma 2: FXP_CLOSEST_IS_CLOSEST
⊢  FINITE (s) ⟹  ¬(s = EMPTY) ⟹  is_closest v s x (closest v p s x)
```

Then we prove that the set of all valid fixed-point numbers with a given attributes is finite:

```
Lemma 3: FINITE_VALID_ATTRIB
⊢  FINITE {(a: fxp) | Isvalid a ∧ (Attrib a = X)}
```

Finally, the following lemma establishes the validity of the rounding result:

```
Lemma 4: IS_VALID_ROUND_CONV_CLIP
⊢  (validAttr (X)) ⟹  Isvalid (FST (fxp_round X Convergent Clip x))
```

The validity of the rounding directly implies the validity of fixed-point addition output, and this completes the proof of the first part of the theorem. The second part of the theorem is proved using the properties of real arithmetic in HOL and rewriting with the definitions of *fxpAdd* and *fxperror* functions.

The second main theorem on error analysis concerns bounding the rounding error. The error can be absolutely quantified as follows:

[4] For a complete list of lemmas refer to Appendix B.

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

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 the non-negative rationals given by:

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

Similarly, for signed two's complement representation:

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

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.

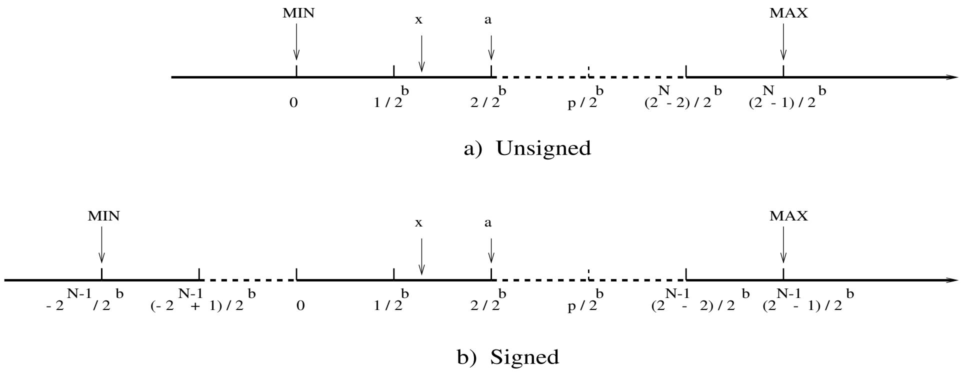

Figure 4: Fixed-Point Values on the Real Axis

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 $x \in \Re$ 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 HOL we first proved that the rounding result is the nearest value to a real number and the corresponding error is minimum comparing to the other fixed-point numbers:

```
Lemma 5: FXP_ERROR_AT_WORST_LEMMA
⊢ (validAttr X) ∧ ¬(x > MAX (X)) ∧ ¬(x < MIN (X)) ∧
  (Isvalid a) ∧ (Attrib a = X) ⟹
  abs (fxperror X Convergent Clip x) ≤ abs (value a − x)
```

Then we proved that each representable real value x can be surrounded by two successive rational numbers:

```
Lemma 6: FXP_ERROR_BOUND_LEMMA1
⊢ (validAttr X) ∧ ¬(x > MAX (X)) ∧ ¬(x < MIN (X)) ⟹
  ∃k. (k < 2 EXP streamlength X) ∧ (&k / (&2 pow fracbits X) ≤ x) ∧
  (x < (&(SUC k) / (&2 pow fracbits (X))))
```

Also we proved that the differnce between the real number and the surrounding rationals is less than 1 / $2^{fracbits\ (X)}$:

```
Lemma 7: FXP_ERROR_BOUND_LEMMA2
⊢  (validAttr X) ∧ ¬(x > MAX (X)) ∧ ¬(x < MIN (X)) ⟹
   ∃k. (k ≤ 2 EXP streamlength 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 8: 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 = streamlength X)
```

Since the rounding produces the minimum error as stated in *Lemma 5*, the proof of the second theorem (FXP_ERROR_BOUND_THM) is a direct consequence of *Lemma 8*. 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 for dealing with the negative numbers. Although the two main theorems in this section are proved for *Convergent* and *Clip* modes, the proof for other modes can be handled in a similar manner.

6 The Integrator Example

In this section we demonstrate how the formalization presented in the previous sections can be used for the verification of the transition from floating-point to fixed-point algorithmic levels in SPW. As an example consider the case of an *Integrator* as shown in Figure 5. The *Integrator* is first designed and simulated using predefined floating-point blocks and parameters (Figure 5(a)). The design is composed of *Add*, *Gain* (multiply by a constant), and *Delay* blocks together with signal source and sink elements. Figure 5(b) shows the converted fixed-point design 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 the floating-point blocks. The attributes of all fixed-point block outputs are set to $< 64, 31, 1 >$ to ensure that overflows and quantization do not affect the system operation.

We first model the design in 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 = (∀n. c n = a n float_mul b)
⊢def  Float_Delay_Block a b = (∀ n. b n = a (n − 1))
⊢def  Float_Add_Block a b c = (∀ n. c n = a n float_add b n)

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

In these definitions we have used Harrison's formalization of floating-point arithmetic [10]. Floating-point data types are stored in SPW as standard IEEE 64 bit double precision format.

The corresponding predicates for the fixed-point implementation are as follows:

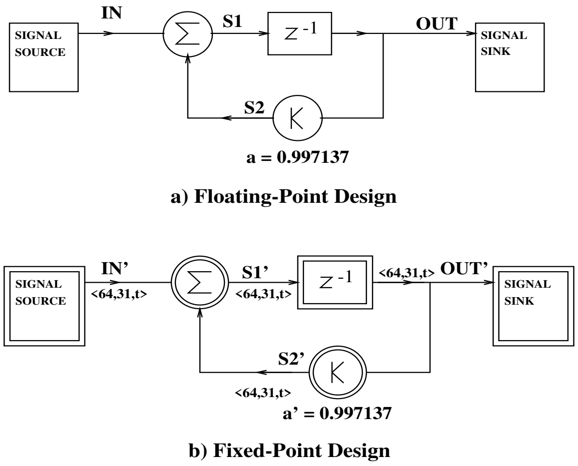

Figure 5: SPW Design of an Integrator

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

```
⊢def  Fxp_Integrator_Imp a′ IN′ OUT′ =
      ∃ S1′ S2′.
      Fxp_Add_Block IN′ S2′ S1′ ∧
      Fxp_Delay_Block S1′ OUT′ ∧
      Fxp_Gain_Block OUT′ a′ S2′
```

The functions *fxp_add* and *fxp_mul* are defined as follows:

```
⊢def  a′ fxp_add b′ = FST (fxpAdd (64,31,1) Convergent Clip a′ b′)
⊢def  a′ fxp_mul b′ = FST (fxpMul (64,31,1) Convergent Clip a′ b′)
```

In the next step, we describe each design as a difference equation relating the input and output samples.

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

⊢def  FXP_Integrator_Spec a′ IN′ OUT′ =
      ∀n. OUT′ n = (IN′ (n − 1) fxp_add (OUT′ (n − 1) fxp_mul a′))
```

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

```
Lemma 9: FLOAT_INTEGRATOR_IMP_SPEC
⊢  Float_Integrator_Imp a IN OUT ⟹  Float_Integrator_Spec a IN OUT

Lemma 10: FXP_INTEGRATOR_IMP_SPEC
⊢  Fxp_Integrator_Imp a′ IN′ OUT′ ⟹  Fxp_Integrator_Spec a′ IN′ OUT′
```

Based on the theorems FXP_ADD_THM and FXP_MUL_THM and the corresponding ones in floating-point theory [10], we prove the following theorem that states the error between the real values of the floating and fixed-point precision output samples.

```
Theorem 3: INTEGRATOR_THM
⊢ Float_Integrator_Imp a IN OUT ∧ Fxp_Integrator_Imp a′ IN′ OUT′ ∧
  Finite IN (n − 1) ∧ Finite OUT (n − 1) ∧ Finite a ∧
  Isvalid IN′(n − 1) ∧ Isvalid OUT′ (n − 1) ∧ Isvalid a′ ∧
  abs (Val OUT (n − 1) ∗ Val a) < threshold float_format ∧
  abs (Val (IN (n − 1)) + Val (OUT (n − 1) float_mul a)) <
  threshold float_format ∧
  (value (OUT′ (n − 1)) ∗ value a′) ≤ MAX (64,31,1) ∧
  (value (OUT′ (n − 1)) ∗ value a′) ≥ MIN (64,31,1) ∧
  (value (IN′ (n − 1)) + value (OUT′ (n − 1) fxp_mul a′)) ≤ MAX (64,31,1) ∧
  (value (IN′ (n − 1)) + value (OUT′ (n − 1) fxp_mul a′)) ≥ MIN (64,31,1)
  ⟹


  Finite (OUT n) ∧ Isvalid (OUT′ n) ∧
  (Val (OUT n) − value (OUT′ n)) =
  (Val (IN (n − 1)) − value (IN′ (n − 1))) +
  ((Val (OUT (n − 1)) ∗ Val a) − (value (OUT′ (n − 1)) ∗ value a′)) +
  error (Val (OUT (n − 1)) ∗ Val a) +
  error (Val (IN (n − 1)) + Val (OUT (n − 1) float_mul a)) +
  fxperror (64,31,1) Convergent Clip (value (OUT′ (n − 1)) ∗ value a′) +
  fxperror (64,31,1) Convergent Clip
  (value (IN′ (n − 1)) + value (OUT′ (n − 1) fxp_mul a′))
```

As shown by 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 assumptions are set to guarantee the validity of output samples. The errors are already quantified using the second theorem in this paper (FXP_ERROR_BOUND_THM) and the corresponding theorems for error analysis in floating-point case.

7 Conclusions

In this paper, we proposed the formalization of SPW based fixed-point arithmetic in the HOL theorem prover. We started first by encoding the fixed-point arithmetic in HOL considering different rounding and overflow modes and exception handling, then we 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 also showed 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 systems. The formalization presented in this paper can be considered as a complement to the floating-point formalizations which are widely available in the literature. Our future work will focous on the verification of the transition from the floating-point algorithmic level to RTL or gate level implementations in 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 ICCAD, pages 7-10. IEEE, Nov. 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, Proc. 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods, CHARME 2001:325-339.

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

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

[6] Y.-A. Chen and R. E. Bryant, Verification of Floating Point Adders, In CAV'98, Volume 1427 of LNCS, 1998.

[7] M. Daumas, L. Rideau, L. Thry, A Generic Library for Floating-Point Numbers and Its Application to Exact Computing, Proc. TPHOLs'2001:169-184, November 2001.

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

[9] J. R. Harrison, Theorem Proving with the Real Numbers, Technical Report Number 408, University of Cambridge Computer Laboratory, December 1996.

[10] J. R. Harrison, Floating-Point Verification in HOL Light: The Exponential Function, Technical Report Number 28, University of Cambridge Computer Laboratory. UK, June 1997.

[11] J. R. Harrison, A Machine-Checked Theory of Floating-Point Arithmetic, Proc. TPHOLs'99, August 1999.

[12] G. Huet, G. Kahn, and C. Paulin-Mohring, The Coq Proof Assistant: A Tutorial: Version 6.1 Technical Report 204, INRIA, 1997.

[13] IEEE, IEEE Standard for Binary Floating-Point Arithmetic, 1985. ANSI/IEEE Std 754-1985.

[14] IEEE, IEEE Standard for Radix-Independent Floating-Point Arithmetic, 1987. ANSI/IEEE Std 854-1987.

[15] M. Leeser, and J. O'Leary, Verification of a Subtractive Radix-2 Square Root Algorithm and Implementation, Proc. ICCD'95, October 1995.

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

[17] Fixed-Point Blockset User's Guide (ver. 2.0), Mathworks, Inc., 1999.

[18] DSP Station User's Guide, Mentor Graphics Corp., 1993.

[19] P. S. Miner, and J. F. Leathrum, Verification of IEEE Compliant Subtractive Division Algorithms, In FMCAD-96, Volume 1166 of LNCS, pages 64-, November 1996.

[20] 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.

[21] J. S. Moore, T. Lynch, and M. Kaufmann, A Mechanically Checked Proof of the AMD5K86 Floating Point Division Program, IEEE Transactions on Computers, 47(9):913-926, 1998.

[22] S. M. Mueller and W. J. Paul, Computer Architecture. Complexity and Correctness, Springer 2000.

[23] 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 Proceedings of FMCAD-00, volume 1954 of LNCS. Springer, 2000.

[24] SystemC User's Guide, Synopsys, Coware, Frontier Design, Inc., 2000.

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

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

[27] M. Willems, V. Bursgens, H. Keding, and H. Meyr, "FRIDGE: An Interactive Fixed-point Code Generation Environment for HW/SW Codesign," in Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, Munchen, Germany, April 1997, pp. 687-690.

[28] W. Wong, The HOL word Library, University of Cambridge, Computer Laboratory, May 1993.

A Rounding to Fixed-Point Values

```
floor = ⊢  ∀ x. floor x = @n. &n ≤ abs x ∧ ∀i. &i ≤ abs x ⟹  i ≤ n
fraction = ⊢  ∀x. fraction x = x − &(floor x)
BDIG = ⊢  ∀n a. BDIG n a = (VB ((a DIV (2 EXP n)) MOD 2))
BDIGFUN = ⊢  ∀a. BDIGFUN a = λ n. BDIG n a
BWORD = ⊢  ∀n m. BWORD n m = WORD (GENLIST (BDIGFUN n) m)
TWOEXPONENT = ⊢  ∀x. TWOEXPONENT x = @n. &(2 EXP n) ≥ abs x ∧
∀m. (& (2 EXP m) ≥ abs x) ⟹  n ≤ m
POSITIVE = ⊢  ∀x. POSITIVE x = &(2 EXP (TWOEXPONENT x)) + x
Wrapp = ⊢  ∀X x. Wrapp (X) x = (x ≥ 0 →  fxp (WCAT ((BWORD (floor x)
(intbits X)),(BWORD (floor (& (2 EXP fracbits X) * (fraction (POSITIVE x))))
(fracbits X))), X)) | fxp (WCAT (((BWORD (floor (POSITIVE x)) (intbits X))
WOR (BWORD ((2 EXP (TWOEXPONENT x))) (intbits X))), (BWORD (floor (& (2 EXP
fracbits X) * (fraction (POSITIVE x)))) (fracbits X))), X))


fxp_round = ⊢  ∀X x mode.
     (fxp_round (X) Convergent mode x =
     (x > MAX X) ∧ (mode = Clip) →  (topfxp X, overflow) |
     (x < MIN X) ∧ (mode = Clip) →  (bottomfxp X , overflow) |
     (x > MAX X) ∧ (mode = Wrap) →  (Wrapp X x , overflow) |
     (x < MIN X) ∧ (mode = Wrap) →  (Wrapp X x , overflow) |
     (closest (value) (λ a. LSB (Stram a) = F)
     {(a: fxp) | Isvalid a ∧ (Attrib a = X)} x, no_except)) ∧
(fxp_round (X) To_zero mode x =
     (x > MAX X) ∧ (mode = Clip) →  (topfxp X, overflow) |
     (x < MIN X) ∧ (mode = Clip) →  (bottomfxp X , overflow) |
     (x > MAX X) ∧ (mode = Wrap) →  (Wrapp X x , overflow) |
     (x < MIN X) ∧ (mode = Wrap) →  (Wrapp X x , overflow) |
     (closest (value) (λ a. T) {a | Isvalid a ∧ (Attrib a = X) ∧
     abs (value a) ≤ abs x} x, no_except)) ∧
(fxp_round (X) To_plus_infinity mode x =
     (x > MAX X) ∧ (mode = Clip) →  (topfxp X, overflow) |
     (x < MIN X) ∧ (mode = Clip) →  (bottomfxp X , overflow) |
     (x > MAX X) ∧ (mode = Wrap) →  (Wrapp X x , overflow) |
     (x < MIN X) ∧ (mode = Wrap) →  (Wrapp X x , overflow) |
     (closest (value) (λ a. T)
     {a | Isvalid a ∧ (Attrib a = X) ∧ (value a ≥ x)} x, no_except)) ∧
(fxp_round (X) Truncate mode x =
     (x > MAX X) ∧ (mode = Clip) →  (topfxp X, overflow) |
     (x < MIN X) ∧ (mode = Clip) →  (bottomfxp X , overflow) |
     (x > MAX X) ∧ (mode = Wrap) →  (Wrapp X x , overflow) |
     (x < MIN X) ∧ (mode = Wrap) →  (Wrapp X x , overflow) |
     (closest (value) (λ a. T)
     {a | Isvalid a ∧ (Attrib a = X) ∧ (value a ≤ x)} x, no_except)) ∧
(fxp_round (X) Round mode x =
     (x > MAX X) ∧ (mode = Clip) →  (topfxp X, overflow) |
     (x < MIN X) ∧ (mode = Clip) →  (bottomfxp X , overflow) |
     (x > MAX X) ∧ (mode = Wrap) →  (Wrapp X x , overflow) |
     (x < MIN X) ∧ (mode = Wrap) →  (Wrapp X x , overflow) |
     (closest (value) (_ a. value (fxp (Stream a, X)) >= x)
     {a | Isvalid a / (Attrib a = X)} x, no_except))
```

B Lemmas for Analyzing the Fixed-Point Rounding Operation

```
FXP_IS_CLOSEST_EXISTS = ⊢ ∀v x s.
FINITE (s) ⟹ ¬(s = EMPTY) ⟹ ∃(a: fxp). is_closest v s x a
FXP_CLOSEST_IS_EVERYTHING = ⊢ ∀v p x s.
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))
FXP_CLOSEST_IN_SET = ⊢ ∀v p x s.
FINITE (s) ⟹ ¬(s = EMPTY) ⟹ (closest v p s X) IN s
FXP_CLOSEST_IS_CLOSEST = ⊢ ∀v p x s.
FINITE (s) ⟹ ¬(s = EMPTY) ⟹ is_closest v s x (closest v p s X)
MAINFINITE =
⊢ FINITE {a | (FST (SND a) < 257) ∧ (FST (SND (SND a)) < (FST (SND a) +
1)) ∧ (SND (SND (SND a)) < 2) ∧ (WORDLEN (FST a) = FST (SND a))}
FINITE_VALID_ATTRIB = ⊢ ∀X. FINITE {a | Isvalid a ∧ (Attrib a = X)}
IS_VALID_NONEMPTY = ⊢ ∀X.
validAttr X ⟹ ¬({a | Isvalid a ∧ (Attrib a = X)} = EMPTY)
IS_VALID_CLOSEST = ⊢ ∀v p X x.
validAttr X ⟹ Isvalid (closest v p {a | Isvalid a ∧
((Attrib a) = X)} x)
IS_VALID_TOPFXP = ⊢ ∀X. Isvalid topfxp X
IS_VALID_BOTTOMFXP = ⊢ ∀X. Isvalid bottomfxp X
IS_VALID_WRAPP = ⊢ ∀X x. Isvalid Wrapp X x
IS_VALID_ROUND_CONV_CLIP = ⊢ ∀X x.
validAttr X ⟹ Isvalid (FST (fxp_round X Convergent Clip x))
```

C Rounding Error in Fixed-point Arithmetic Operations

```
FXP_SUB_THM
⊢ (Isvalid a) ∧ (Isvalid b) ∧ validAttr (X) ⟹
   (Isvalid (FST (fxpSub X Convergent Clip a b))) ∧
   (value (FST (fxpSub X Convergent Clip a b)) = value (a) − value (b) +
   (fxperror X Convergent Clip (value a − value b)))
FXP_MUL_THM
⊢ (Isvalid a) ∧ (Isvalid b) ∧ validAttr (X) ⟹
   (Isvalid (FST (fxpMul X Convergent Clip a b))) ∧
   (value (FST (fxpMul X Convergent Clip a b)) = (value a * value b) +
   (fxperror X Convergent Clip (value a * value b)))
FXP_DIV_THM
⊢ (Isvalid a) ∧ (Isvalid b) ∧ validAttr (X) ⟹
   (Isvalid (FST (fxpDiv X Convergent Clip a b))) ∧
   (value (FST (fxpDiv X Convergent Clip a b)) = (value a / value b) +
   (fxperror X Convergent Clip (value a / value b)))
```