Formal Verification of Floating-Point RTL at AMD Using the ACL2 Theorem Prover

David Russinoff
Matt Kaufmann
Eric Smith
Robert Sumners
Advanced Micro Devices, Inc., Austin, TX
Phone: 512-602-6741, Fax: 512-602-6970, E-mail: david.russinoff@amd.com

Abstract - We describe a methodology for the formal verification of the correctness, including IEEE-compliance, of register-transfer level models of floating-point hardware designs, and its application to the floating-point units of a series of commercial microprocessors produced by Advanced Micro Devices, Inc. The methodology is based on a mechanical translator from a synthesizable subset of the Verilog hardware description language, in which the models are coded, to the formal logic of the ACL2 theorem prover. Behavioral specifications of correctness, coded in essentially the same language as the designs, are translated as well, and ultimately checked with the ACL2 prover.

Keywords— **Formal verification, Floating-point arithmetic, IEEE-compliance, Theorem proving, ACL2**

I. Introduction

The use of conventional simulation dominates the verification of modern microprocessors. However, it is widely accepted that as design complexity continues to grow dramatically, simulation alone is insufficient to establish confidence in the correctness of hardware designs. Consequently, formal verification methods have found increasing use in industrial applications.

This paper describes ongoing work, begun in 1996, in the use of theorem proving in the verification of register-transfer logic (RTL) models of the floating-point units of microprocessors developed by Advanced Micro Devices, Inc. [MOO98, RUS99, RUS98, RUS00a, RUS00b, KAU00d] Our methodology is based on the mechanical translation of a synthesizable subset of the Verilog hardware description language to the formal logic of the ACL2 prover [ACL2, KAU00a].

ACL2 is based on an applicative subset of Common LISP [STE90] with list structures and numerical data types, in which system designs may be encoded along with their behavioral specifications. It is supported by a heuristic theorem prover, which may be used to establish the correctness of a design, i.e., to derive a logical proposition stating that a design satisfies its specification.

Thus, RTL modules, written in Verilog, are translated into ACL2 according to a scheme in which bit vectors are represented as natural numbers, RTL primitives correspond to LISP functions, and signal dependencies are characterized by recursive function definitions. Through suitable extensions of standard Verilog syntax, e.g., the introduction of a rational data type, high-level arithmetic functional specifications are encoded in the same language and similarly translated into ACL2.

One advantage of a Verilog behavioral specification of a design is that it is accessible to RTL writers, and in fact may be developed in collaboration between designer and verifier. Thus, it may effectively serve as formal documentation shared by the designers, verifiers, and users of a module. Since our Verilog extensions have been implemented in our simulation environment, the same specification may also be used as a basis for comprehensive testing.

Of course, the primary purpose of such a specification is formal analysis. Once a module and its behavioral specification have been translated to ACL2, a proof of correctness is established as the culmination of a sequence of lemmas, generated through a process of interaction with the prover. Along the way, any flaws existing in the RTL are naturally exposed and must be corrected in order for the proof to be completed.

Successful use of ACL2 in large-scale applications typically depends on the development of reusable libraries of definitions and proved theorems. Over the course of this work, we have established a continually growing library of theorems pertaining to RTL primitives and floating-point arithmetic, which is publicly available as part of the standard ACL2 release. (The description of the library presented here corresponds to ACL2 Version 2.9.2.)

II. RTL Designs and Specifications in Verilog

A. RTL Models

Our RTL models are coded in a limited subset of Verilog defined by a strict set of coding guidelines designed to facilitate synthesis as well as simulation. These restrictions, which preclude such anomalies as race conditions and zero-time evaluation cycles, ensure that for each signal in a synchronous circuit, a well-defined final value for each clock cycle may be computed as a function of the values of other signals. An RTL module consists of three classes of signals: *inputs*, *wires* and *registers*. Any wire or register may be designated as an *output*. The values of inputs are undefined by the module; the defining equations of wires and registers are syntactically distinguished by the assignment operators "=" and "<=", respectively. The value of a wire

for a given clock cycle is determined by the values, for the same cycle, of the other signals that occur in its defining equations. For example, the wire `out`, defined by the following three equations, is a function of the three signals `in1`, `in2`, and `in3`.

```
out[5:0] = 6'b111111;
if (in1) out[1] = in2;
if (in2) out[5:3] = in3[2:0];
```

For a register, such as either of the signals `sum and carry` defined by the single equation below, the value for a given cycle is computed from the values of its defining signals—in this case, `a` and `b`—on the preceding cycle.

```
{carry, sum[2:0]}
  <= {1'b0, a[2:0]} + {1'b0, b[2:0]};
```

B. Behavioral Specifications

One consequence of the simplicity of our modeling language is that it admits a straightforward semantics-preserving translation to the functional language of ACL2 (as described in Section III). Statements of behavioral correctness of RTL modules may then be formulated as logical propositions pertaining to the functions generated by this translation, the proofs of which may be mechanically checked by the ACL2 prover.

From the perspective of formal verification, the most natural approach is to write these behavioral specifications directly in the ACL2 logic, a language designed to provide a level of expressiveness suitable for this task. However, there are two important potential advantages of a specification written in the same language as the design itself:

(1) If a specification can be written in Verilog, then it is immediately accessible to design engineers, who are generally unacquainted with logical and functional languages. In fact, such a specification can be written in collaboration between engineers and verifiers, providing a relatively high level of confidence that it successfully captures the intended functionality of the design.

(2) A Verilog specification may be executed in simulation along with the design. This provides a practical means of exposing bugs even before any formal analysis is performed, both in the RTL and in the specification itself.

Although no conventional hardware description language is ideally suited for formal specification, we have identified and implemented a small number of extensions to our synthesizable subset of Verilog that have been found to provide adequate expressiveness for our purpose.

C. Assertions

The first problem to be addressed is the need to formulate predicates that represent various types of constraints on the interface behavior of a module. Thus, a module specification consists of a supplementary set of *specification signals*, each of which is defined as a wire or a register in the extended language, together with a set of interface constraints, each of which is classified as an *input* or *output assertion*. All signals and assertions of the specification are defined in terms of the inputs and outputs of the module.

Every signal occurring in an input assertion must be derived from inputs and registers. For example, the following is taken from the specification of a multiplier that performs several operations of different latencies.

```
assert(DivSqrtOpIn4 ==> ~MulOpInit);
```

Here the signal `DivSqrtOpIn4` is a register (of the specification) that indicates that the result of a divide or square root operation is to be returned in four cycles; `MulOpInit` means that a multiplication is being initiated. The implication operator "`==>`" is a minor syntactic extension: `a ==> b` may be read as `~a | b`. Since this operation has a latency of four, the effect is to preclude the simultaneous termination of two operations.

Every signal occurring in an output assertion must be derived from outputs and registers. The following (which pertains to the same multiplier) asserts that whenever a valid result is returned in the absence of any microarchitectural fault, the value of the data output `FRes_15` agrees with that of the specification signal `DataSpec` at all relevant bit locations, which are determined by the signal `DataMask`.

```
assert
 (ResultValid & ~Fault  ==>
  (FRes_15[75:0] & DataMask[75:0])
   ==
  (DataSpec[75:0] &  DataMask[75:0]));
```

Each of the above constraints may be classified as a *safety* property, i.e., a statement that some condition is satisfied continuously throughout every execution. Our assertion language also provides for *liveness* properties. A liveness condition is one that is guaranteed not to fail permanently, i.e., if it ever becomes false during an execution, then it must become true at some later time. Such assertions are required to specify the behavior of variable-latency operations. The following, for example, states that if the signal `DivSqrtPending` is asserted, which occurs whenever a divide or square root operation is issued, it must eventually be deasserted, indicating that the operation has terminated:

```
eventually (~DivSqrtPending);
```

D. Rational-Valued Signals

In addition to the assertion facility, our specification language includes several data types that are foreign to standard Verilog. For the purpose of floating-point specification, the most important of these is the rational number type, without which it would be impossible to describe the high-level behavior of any floating-point operation in a natural way. The built-in operators associated with this type include the usual basic arithmetic operators as well as various special functions pertaining to floating-point rep-

resentation and rounding. In the following illustration, `Val1` and `Val2` are rational specification signals derived from the operands of a double-precision (53-bit) multiplication. The signal `RndRes` represents the prescribed rational value of the rounded result, computed according to the rounding mode indicated by the control input `RC`. This signal is used in the definition of the signal `DataSpec` appearing in the output assertion above.

```
// Unrounded product:

Unrnd = Val1 * Val2;

// Rounded product:

case (RC[1:0])
 'RC_RZ  : RndRes = $Trunc(Unrnd, 53);
 'RC_RP  : RndRes = $Inf(Unrnd, 53);
 'RC_RM  : RndRes = $Minf(Unrnd, 53);
 'RC_RN  : RndRes = $Near(Unrnd, 53);
endcase
```

For the purpose of simulation, each new feature of our specification language must be translated or otherwise incorporated into standard Verilog. The safety assertions present no difficulty, as they are readily translated into the pre-existing assertion language of our simulation environment. For liveness assertions, however, the translation involves some work. For each such assertion, a counter is defined. If the indicated condition ever fails to hold, the counter in initialized to a value that may be set in the specification. If the counter is decremented to 0 before the condition becomes true, then the assertion fails. Although the choice of the counter's initial value is arbitrary, this implementation has generally produced acceptable results in simulation.

Since Verilog does not provide built-in support for (arbitrarily precise) rational arithmetic, we have implemented a C library for this purpose, based on the GNU Multiple Precision Arithmetic Library [GMP], to which calls are generated during Verilog simulation. The C code maintains the values of all rational signals, while special Verilog variables are used are used to trigger the evaluation of any blocks that are sensitive to changes in these values.

III. Translation of Verilog to ACL2

A. Primitive Operators

Our translation scheme is based on the natural correspondence between bit vectors and integers, the primary data types of Verilog and ACL2, respectively. Thus, a bit vector of width n, $b_{n-1}b_{n-2}\cdots b_1b_0$, where each $b_k \in \{0,1\}$,

corresponds to the integer $x = \sum_{k=0}^{n-1} 2^k b_k$, in the range $0 \le x < 2^n$. The set of such integers is recognized by the ACL2 predicate `bvecp`:

```
(defun bvecp (x n)
  (and (integerp x)
       (>= x 0)
       (< x (expt 2 n)))
```

We also define an ACL2 function corresponding to each RTL primitive. The Verilog operator that extracts the slice of bits between indices i and j from a vector x, which may be defined as

$$x[i:j] = \lfloor rem(x, 2^{i+1})/2^j \rfloor,$$

and the single bit operator, $x[k] = x[k:k]$, are formalized in ACL2 as follows:

```
(defun bits (x i j)
  (mbe :logic
       (fl (/ (mod x (expt 2 (1+ i)))
              (expt 2 j)))
       :exec
       (logand
         (ash x (- j))
         (1- (ash 1 (1+ (- i j)))))))

(defun bitn (x k)
  (bits x k k))
```

Note that the definition of `bits` makes use of the ACL2 "must be equal" feature, which provides for two (provably equivalent) formulations of the same function: one for logical analysis and the other for efficient execution.

The concatenation operator,

$$\{x[m-1:0], y[n-1:0]\} = 2^n x[m-1:0] + y[n-1:0],$$

is formalized as a function of four arguments:

```
(defun cat (x m y n)
  (+ (* (expt 2 n) (bits x (1- m) 0))
     (bits y (1- n) 0)))
```

The RTL binary logical operators roughly correspond to the built-in LISP functions `logand`, etc. Thus,

$$x[n-1:0] \;\&\; y[n-1:0]$$

is represented by

```
(defun land (x y n)
  (logand (bits x (1- n) 0)
          (bits y (1- n) 0)))
```

while

$$x[n-1:0] \mid y[n-1:0]$$

and

$$x[n-1:0] \;\hat{}\; y[n-1:0]$$

are similarly formalized by functions `lior` and `lxor`, defined in terms of the LISP primitives `logior` and `logxor`, respectively. For the unary complement operator,

$$\tilde{}\,x[n-1:0] = 2^n - x[n-1:0] - 1,$$

we have

```
(defun lnot (x n)
  (+ (expt 2 n)
     (- (bits x (1- n) 0))
     -1))
```

Addition and multiplication of bit vectors are formalized by composing `bits` with the built-in integer arithmetic functions:

```
(defun mod+ (x y n)
  (bits (+ x y) (1- n) 0))
```

As for the extra constructs associated with our rational data type, the elementary arithmetic operations all correspond to primitive LISP functions, while the ACL2 definitions of the functions pertaining to floating-point arithmetic (`$trunc`, etc.) are straightforward.

B. Signal Definitions

As noted in Section II-A, a well-defined value may be computed for each signal of a synchronous module for each clock cycle. Consequently, the behavior of a signal may be represented as a function of a single integer argument n, representing the number of elapsed clock cycles. (For a design that involves several instantiations of various modules, a second argument is required in order to distinguish between signals associated with different module instances, but this complication is ignored in this paper.) The ACL2 translation of a module consists mainly of a mutually recursive set of such function definitions.

Since we have no formal semantic definition of the Verilog source, there can be no "proof of correctness" of the translation. However, in order to maximize our confidence that the simulation behavior of a signal is faithfully represented by the corresponding ACL2 function, the translation is performed as a two-step process [KAU03]. The first step is designed to be as simple as possible, so that the process itself may be easily analyzed and understood, but without regard for the readability or convenience of application of the resulting "raw" ACL2 definitions. In the second step, the ACL2 simplifier is used to reduce these functions to a more manageable form and to *prove* the equivalence of the two forms. The prover is also used during this stage to derive lemmas that characterize the values returned by these signal functions, to be used later in generating proofs of correctness.

As an illustration, the wire `out` of Section II-A generates the following raw definition during the first translation stage:

```
(defun out$tmp (n)
 (if1 (bitn (in1 n) 0)
   (setbits 63 6 1 1 (bitn (in2 n) 0))
   63))

(defun out (n)
  (if1 (bitn (in2 n) 0)
     (setbits (out$tmp n) 6 5 3
              (bits (in3 n) 2 0))
     (out$tmp n)))
```

Here `if1` is a variant of the standard branching function `if` that tests whether its first argument is nonzero; `setbits` replaces a slice of its first argument as indicated by the remaining arguments. In the second stage, the calls to `setbits` are eliminated:

```
(defun out$tmp (n)
  (if1 (in1 n)
     (cat 15 4 (cat (in2 n) 1 1 1) 2)
     63))

(defun out (n)
  (if1 (in2 n)
       (cat (in3 n) 3 (out$tmp n) 3)
       (out$tmp$1 n)))
```

Note that the function generated for a wire includes calls to other signal functions with the same cycle number n. For a register, the supporting signals are evaluated on the preceding cycle. The raw and simplified definitions for the signal `sum` of the preceding section follow:

```
(defun sum (n)
  (if (zp n)
    (reset 'sum 3)
    (bits (mod+
            (cat 0
                 1
                 (bits (a (1- n)) 2 0)
                 3)
            (cat 0
                 1
                 (bits (b (1- n)) 2 0)
                 3)
            4)
          2 0)))

(defun sum (n)
  (if (zp n)
    (reset 'sum 3)
    (bits (+ (a (+ -1 n)) (b (+ -1 n)))
          2 0)))
```

The undefined function `reset`, which appears in all register functions, is used to model unknown values in our two state logic. By means of the ACL2 `encapsulate` feature, it is constrained to return a bit vector of unspecified value but fixed width. Thus, all that is known about the value of `sum` in the initial state at $n = 0$, i.e., when `(zp n)` is true, is that it is a bit vector of width 3.

C. Assertions

The same procedure by which ACL2 functions are derived from Verilog equations is also used to translate assertions into ACL2 predicates, which are also functions of the cycle number n. The main result required to establish that a module satisfies its specification is a theorem stating that all output assertions hold on every cycle, under the assumption that no input assertion has been violated on any earlier cycle. In order to formulate this theorem in ACL2, we define a predicate *input-assertions* as the conjunction of all predicates derived from the input safety assertions of a module specification. Similarly, a predicate *output-assertions* is derived from the output safety assertions. The main theorem states that for every `n`, if `(input-assertions k)` is true for all `k` $<$ `n`, then `(output-assertions n)` must also be true.

If a specification includes liveness assertions as well, then an additional result must be proved for each output liveness assertion, stating that for all `n`, the corresponding predicate holds for some `k` $>$ `n`. Typically, it is necessary to construct an expression for `k` as a function of `n` and the values of various specification signals at cycle `n`. For example, consider the case of the multiplier liveness property of the Section II-A. If `DivSqrtPending` is deasserted at cycle `n`, then we may take `k = n`; otherwise, it must be shown that `FPM_DoneInSeven_divsqrt_8` is asserted at some `k` $>$ `n`, the expression for which involves various other signals derived from the inputs, which determine the latency of the operation.

IV. An ACL2 Library of Floating-Point Arithmetic

An ACL2 proof of any theorem of practical significance requires considerable guidance from the user. Typically, the prover must be guided, by means of user-supplied hints, through a lengthy sequence of lemmas culminating in the desired result. In particular, a proof of correctness of even the simplest floating-point operation involves the formulation and proof of at least several hundred lemmas, while a more complicated module may involve tens of thousands. Some of these lemmas are specific to the RTL design at hand, pertaining to the values of particular signals; others are results of more general interest, dealing with relevant properties of bit vectors, arithmetic, etc. During the course of our RTL verification effort, we have endeavored to distinguish between these two classes of lemmas, and have collected those belonging to the latter class in an evolving reusable library. Thus, once a theoretical arithmetic result or a property of a standard technique has been established during the analysis of a particular module, it remains available for use in the proofs of future designs.

There are two ways in which an ACL2 lemma may be applied to the proof of another lemma. First, once the user becomes aware of the applicability of an existing lemma to a proof in progress, he may supply a hint to the prover, instructing it to instantiate the lemma as appropriate and to treat the resulting formula as an additional hypothesis of the conjecture under consideration. On the other hand, a lemma may be declared to belong to one or more *rule classes*, which will allow it to be applied automatically according to the prover's heuristics, even without the user's knowledge. The most important of these classes is that of *rewrite* rules, which are used in the process of reducing an expression to a simpler form. Rewrite rules, when they worked as planned, provide the benefit of relieving the user of some of the details of a proof, but often have the disadvantage of depriving him of control. Our experience has convinced us of the prudence of a conservative approach whereby most lemmas are supplied manually via hints from the user, but if the application of a simplifying lemma is considered to be generally desirable, it may be classified as a rewrite rule. In some cases, a lemma may be listed as a rewrite rule but left in a disabled state, offering the user the choice of using it in a hint or enabling it for automatic application. Illustrations will be provided by the examples below.

The library comprises six ACL2 "books", i.e., files of lemmas and definitions. Of the combined set of approximately 600 library lemmas, we shall list only a few representative examples in the space allowed here. For the sake of readability, we shall use conventional mathematical notation rather than the formal syntax of LISP, but the reader is encouraged to consult the actual ACL2 source files.

The books of the library are logically partitioned into three sections:

A. Bit Vectors

Much of the library pertains to the basic operations on bit vectors and the underlying general theory of integer arithmetic. These results are organized into two books:

- `basic`: Definitions and basic properties of the relevant built-in arithmetic functions, especially floor (also known as the *greatest integer* function), quotient and remainder, and exponentiation (87 lemmas).
- `bits`: Behavior of the primitive operations on bit vectors, including bit extraction, concatenation, and logical operations, as described in Section II-A (189 lemmas).

The book `bits` is the largest and most widely used of the library. It contains a relatively large number of rewrite rules. These include associative and commutative laws of the primitive operations, for which the prover has special heuristics for establishing equalities between expressions constructed from these functions. Various other rewrite rules have a similar effect, namely, the transformation of certain types of expressions into a canonical form:

Lemma $(x \mathbin{\&} y)[n] = x[n] \mathbin{\&} y[n]$.

The following is an example of a disabled rewrite rule that may be enabled as appropriate:

Lemma If $i < m + n$, then

$$\{x[m-1:0], y[n-1:0]\}[i:j]$$

$$= \begin{cases} y[i:j] & \text{if } i < n \\ x[i-n:j-n] & \text{if } j > n \\ \{x[i-n:0], y[n-1:j]\} & \text{if } j < n \leq i. \end{cases}$$

In general, a rewrite rule is appropriate when the right side of an equality is invariably a simpler expression than the left side, as in the examples below:

Lemma If $j \leq i$, $\ell \leq k$, and $j = k + 1$, then

$$\{x[i:j], x[k:\ell]\} = x[i:\ell].$$

Lemma If $k \leq i - j$, then

$$x[i:j][k:\ell] = x[k+j:\ell+j].$$

Lemma $(2^n - 1)$ & $x = x[n-1:0]$.

Lemma $(2^n - 1)$ | $x[n-1:0] = 2^n - 1$.

In cases such as the following, in which an equation might be used as a transformation in either direction, control is best retained by the user:

Lemma x & $(2^i - 2^j) = 2^j x[i-1:j]$.

Lemma $(2^m x)$ & $y[n-1:0] = 2^m(x$ & $y[n-1:m])$.

Lemma If $m \leq n$, $y < 2^m$, and $x < 2^{n-m}$, then

$$(2^m x) \mid y = 2^m s + y.$$

B. Floating-Point Arithmetic

The general theory of floating-point arithmetic is covered in three books:

- `float`: Characterization of representable rational numbers and their decomposition into sign, exponent, and significand (51 lemmas).
- `reps`: Encoding and decoding of rationals with respect to various (IEEE and other) representation schemes, including both implicit and explicit leading-one formats; denormals; characterization of representable numbers; conversion between formats and rebiasing of exponents (50 lemmas).
- `round`: Properties of IEEE rounding as well as various rounding modes used internally by AMD floating-point units; results pertaining to the implementation and correctness of a number of standard implementations of rounding (193 lemmas).

Floating point representation is based on the observation that every nonzero rational number x admits a unique factorization,

$$x = sgn(x)sig(x)2^{expo(x)},$$

where $sgn(x) \in \{1, -1\}$, $1 \leq sig(x) < 2$, and $expo(x)$ is an integer. The lemmas of the book `float` deal with the properties of these functions, such as the following:

Lemma If $x > 0$, then $2^{expo(x)} \leq x < 2^{expo(x)+1}$.

Lemma $sig(sig(x)) = sig(x)$.

Lemma $sig(2^n x) = sig(x)$.

Lemma $expo(2^n x) = expo(x) + n$.

Lemma $expo(x) + expo(y) \leq expo(xy) \leq expo(x) + expo(y) + 1$.

The theory of representability is based on the predicate *exactp*, which determines whether a rational x admits a binary expansion consisting of n bits. More formally, *exactp(x,n)* is true if and only if $2^{n+1-expo(x)}x$ is an integer. Among the lemmas pertaining to this predicate are the following:

Lemma If $exactp(x, n)$, then $exactp(2^k x, n)$.

Lemma If $m < n$ and $exactp(x, m)$, then $exactp(x, n)$.

Lemma If x is a bit vector of width n, then $exactp(x, n)$.

Lemma If $exactp(x, m)$ and $exactp(y, n)$, then $exactp(xy, m + n)$.

Lemma If $x > 0$ and $exactp(x, n)$, then the least $y > n$ such that $exactp(y, n)$ is $y = x + 2^{expo(x)+1-n}$.

Our formalization of the IEEE rounding modes is based the three functions *trunc* (round-to-zero), *away* (round-away-from-zero), and *near* (round-to-nearest-even), which are defined in terms of the floor function. For example,

$$trunc(x, n) = sgn(x)\lfloor 2^{n-1-expo(x)}|x|\rfloor 2^{expo(x)+1-n}.$$

The book `round` includes lemmas characterizing the behavior of these functions:

Lemma If $x > 0$, then

$$x - 2^{expo(x)+1-n} < trunc(x, n) \leq x.$$

Lemma If $m \leq n$, then

$$trunc(trunc(x, n), m) = trunc(x, m).$$

Lemma $exactp(x, n) \Leftrightarrow trunc(x, n) = x$.

Lemma $trunc(2^k x, n) = 2^k trunc(x, n)$.

A number of lemmas relate rounding to the bit vector primitives:

Lemma If $0 < k < n \leq m$ and $2^{n-1} \leq x < 2^n$, then

$$trunc(x, k) = x \text{ \& } (2^m - 2^{n-k}).$$

Lemma If $0 < k < m$ and $2^{n-1} \leq x < 2^n$, then

$$trunc(x, m) = trunc(x, k) + 2^{n-m}x[n-k-1:n-m].$$

One of the non-standard rounding modes used in our analysis is *sticky rounding*. If $exactp(x, n)$, then $sticky(x, n) = x$, and otherwise,

$$sticky(x, n) = trunc(x, n-1) + sgn(x)2^{expo(x)+1-n}.$$

The significance of this function is that the result of rounding x to n bits, according to any IEEE rounding mode, can always be recovered from $sticky(x, n+2)$:

Lemma If $m > n$, then

$$trunc(x, n) = trunc(sticky(x, m), n).$$

Lemma If $m > n$, then

$$away(x, n) = away(sticky(x, m), n).$$

Lemma If $m \geq n + 2$, then

$$near(x, n) = near(sticky(x, m), n).$$

Here is a property of sticky rounding that is in dealing with rounded sums:

Lemma Let $k' = k + expo(x) - expo(y)$ and $k'' = k + expo(x + y) - expo(y)$. If $k > 1$, $k' > 1$, $k'' > 1$, and $exactp(x, k' - 1)$, then

$$x + sticky(y, k) = sticky(x + y, k'').$$

C. Special-Purpose Techniques

The final book of the library, `fadd` (30 lemmas), deals with a number of standard techniques that are commonly used in the implementation and optimization of floating-point operations, including methods of carry propagation and generation, sticky bit computation, leading-one prediction, and multiplier encoding. The contents of this book do not properly belong to the general theory of floating-point arithmetic, but they are relevant to a variety of FPU designs and are therefore suitable for inclusion in the library.
For example, various optimizations in the addition of bit vectors are based on the functions *gen* and *prop*, defined recursively as follows:

$$gen(x, y, i, j) = \begin{cases} 0 \text{ if } i < j \\ x[i] \text{ if } i \geq j \text{ and } x[i] = y[i] \\ gen(x, y, i-1, j) \text{ otherwise} \end{cases}$$

and

$$prop(x, y, i, j) = \begin{cases} 1 \text{ if } i < j \\ 0 \text{ if } i \geq j \text{ and } x[i] = y[i] \\ prop(x, y, i-1, j) \text{ otherwise.} \end{cases}$$

The value of $gen(x, y, i, j)$ indicates whether a carry bit is *generated* by the sum $x[i : j] + y[i : j]$. If $gen(x, y, i, j) = 0$, then $prop(x, y, i, j)$ determines whether an incoming carry is *propagated* across the same slice. We have the following equivalent formulations:

Lemma
$gen(x, y, i, j) = 1 \Leftrightarrow x[i : j] + y[i : j] \geq 2^{i+1-j}$.

Lemma
$prop(x, y, i, j) = 1 \Leftrightarrow x[i : j] + y[i : j] = 2^{i+1-j} - 1$.

Lemma
$prop(x, y, i, j) = 1 \Leftrightarrow x[i : j] \hat{} y[i : j] = 2^{i+1-j} - 1$.

Various properties of *gen* and *prop* allow them to be computed efficiently:

Lemma If $i > k \geq j$, then

$$\begin{aligned} gen(x, y, i, j) \quad = \quad & gen(x, y, i, k+1) \mid \\ & gen(x, y, k, j) \,\&\, prop(x, y, i, k+1). \end{aligned}$$

Lemma If $i > k \geq j$, then

$$prop(x, y, i, j) = prop(x, y, i, k+1) \,\&\, prop(x, y, k, j).$$

The addition of bit vectors may be optimized by means of lemmas such as the following, which allows a sum to be decomposed into bit slices:

Lemma
$(x+y)[i : j] = (x[i : j]+y[i : j]+gen(x, y, i, j\text{-}1))[i\text{-}j : 0]$.

Another example of an efficient addition technique, the following result is used as an optimization in the computation of rounded sums. Suppose that two bit vectors a and b are to be added, along with a carry bit c. For the purpose of rounding, we would like to predict the exactness, i.e, the number of trailing zeroes, of $a + b + c$. The following lemma provides a quantity τ that may be computed in constant time (independent of the width of a and b) and has the same number of trailing zeroes as the sum.

Lemma Let a and b be bits vectors of width n, let c be 0 or 1, and let $k < n$. Let

$$\sigma = \begin{cases} \tilde{}(a \hat{} b)[n-1 : 0] & \text{if } c = 0 \\ a \hat{} b & \text{if } c = 1, \end{cases}$$

$$\kappa = \begin{cases} 2(a \mid b) & \text{if } c = 0 \\ 2(a \,\&\, b) & \text{if } c = 1, \end{cases}$$

and

$$\tau = \tilde{}(\sigma \hat{} \kappa)[n : 0].$$

Then

$$(a + b + c)[k : 0] = 0 \Leftrightarrow \tau[k : 0] = 0.$$

V. Summary of Results

Following the methodology outlined above, we have verified the correctness, including IEEE-compliance [IEEE85], of floating-point modules of the AMD microprocessor series that perform addition, multiplication, comparison, division, and square root extraction. Our efforts have exposed a number of design flaws, many of which had survived extensive testing through simulation, and all of which were corrected before fabrication in silicon.

The ability to represent behavioral specifications in the native language of hardware designers has proved to be of great value. Our earlier verification efforts were based on formulations of correctness written directly in ACL2, derived from an informal and unreliable understanding of the designers' intent. In contrast, this new approach brings the designer into the specification process and allows a considerably higher level of confidence in the result. It also provides an effective means of communication among designers and users, replacing the less dependable documentation practices on which they have traditionally relied. The collaborative formulation of these specifications has exposed a number of bugs resulting from poorly understood interface constraints much earlier in the design process than they would otherwise have been found.

The use of formal specifications in simulation has also proved beneficial. Our application of this technology has not been limited to floating-point data paths: we have written similar specifications of several blocks of control logic, which have been used for both formal analysis and testing through co-simulation. As a consequence of the high level of abstraction that characterizes the design of these specifications, they have been effective in exposing bugs that went undetected by standard testing methods, which are generally tied more closely to the implementation.

While it is recognized that theorem proving offers a higher level of assurance than other methods of hardware verification, it remains notoriously labor-intensive. We have found, however, that a significant part of the total effort of verifying a floating-point RTL design is effectively reusable. As our ACL2 library has evolved, the time required for the verification of comparable modules has noticeably decreased. Since no part of this library is specific to AMD designs or to Verilog constructs, it is our hope that it may be of use to others who are interested in verifying floating-point algorithms and RTL.

References

[STE90] Steele, G.L., Jr., Common Lisp The Language, 2nd edition, Digital Press, 1990.

[ACL2] ACL2 home page. `www.cs.utexas.edu/users/moore/acl2/.`

[GMP] GNU Multiple Precision Arithmetic Library. `www.swox.com/gmp/.`

[IEEE85] Institute of Electrical and Electronic Engineers, IEEE Standard for Binary Floating Point Arithmetic, Std. 754-1985, New York, NY, 1985.

[KAU00a] Kaufmann, M., P. Manolios, and J S. Moore, Computer-Aided Reasoning: An Approach. Kluwer Academic Press, 2000.

[KAU00b] Kaufmann, M., P. Manolios, and J S. Moore, editors, Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Press, 2000.

[KAU00d] Kaufmann, M. and D. M. Russinoff, Verification of pipeline circuits, Proceedings ACL2 Workshop 2000, Oct. 2000. `www.cs.utexas.edu/users/moore/acl2/workshop-2000/.`

[KAU03] Kaufmann, M., A tool for simplifying files of ACL2 definitions, Proceedings ACL2 Workshop 2003, July 2003. `www.cs.utexas.edu/users/moore/acl2/workshop-2003/.`

[MOO98] Moore, J, T. Lynch, and M. Kaufmann, A mechanically checked proof of the correctness of the kernel of the $AMD5_K86$ floating point division algorithm, *IEEE Transactions on Computers*, 47:9, September, 1998.

[RUS98] Russinoff, D.M., A mechanically checked proof of IEEE compliance of the AMD-K7 floating point multiplication, division, and square root instructions, London Mathematical Society Journal of Computation and Mathematics (1), 148-200, December, 1998. `www.russinoff.com/papers/k7-div-sqrt.html`.

[RUS99] Russinoff, D.M., A mechanically checked proof of IEEE compliance of the AMD-K5 floating point square root microcode, Formal Methods in System Design 14, 75-125 (1999). `www.russinoff.com/papers/fsqrt.html`.

[RUS00a] Russinoff, D.M., A case study in formal verification of register-transfer logic with ACL2: the floating point adder of the AMD Athlon processor, invited paper, FMCAD 2000. `www.russinoff.com/papers/fadd.html`.

[RUS00b] Russinoff, D.M. and A. Flatau, RTL verification: a floating-point multiplier, in [KAU00b], pp. 201–232. `www.russinoff.com/papers/acl2.html`.