Chapter 1

MECHANICAL VERIFICATION OF REGISTER-TRANSFER LOGIC: A FLOATING-POINT MULTIPLIER

David M. Russinoff
Arthur Flatau
Advanced Micro Devices, Inc.
Austin, TX
david.russinoff@amd.com
arthur.flatau@amd.com

Abstract We describe a mechanical proof system for designs represented in the AMD[1] RTL language, consisting of a translator to the ACL2 logical programming language and a methodology for verifying properties of the resulting programs using the ACL2 prover. As an illustration, we present a proof of correctness of a simple floating-point multiplier.

Introduction

In order for a hardware design to be provably correct, it must be represented in a language that has an unambiguous semantic definition. Unfortunately, commercial hardware description languages such as VHDL and Verilog, which are intended for a variety of purposes other than formal verification, are large, complicated, and poorly specified. Attempts to develop formal semantics for these languages [Gordon, 1995, Russinoff, 1995] have generally been limited to small, manageable subsets that are inadequate for modeling real industrial designs. Consequently, a "proof of correctness" of a real VHDL or Verilog design is generally based on an alternative encoding of the underlying algorithm in some

[1]AMD, the AMD logo and combinations thereof, and AMD Athlon are trademarks of Advanced Micro Devices, Inc.

simpler formal language. The utility of such a proof rests on the unproved assumption that the two implementations are equivalent.

As an alternative to these commercial languages, Advanced Micro Devices, Inc. has adopted a special-purpose hardware language for the design of the AMD Athlon^{TM} processor and future AMD microprocessors. The language syntactically resembles Verilog, but is considerably simpler. While Verilog includes extensive features to support the design and testing of a wide variety of digital systems at various levels of abstraction, the AMD language is intended solely for modeling microprocessor designs at the level of register-transfer logic (RTL). Moreover, (although this was not a consideration in its design) our language was constructed carefully enough to allow formal verification as a realistic objective.

The subject of this paper is a methodology for mechanical verification of real hardware designs written in the AMD RTL language, using the ACL2 prover. The underlying theory of floating-point arithmetic and its bit-level implementation, developed through the course of our prior work on the verification of floating-point algorithms, is embodied in an ACL2 library, consisting of several books of definitions and lemmas, which is available on the Web [Russinoff, 1999b]. This library is briefly summarized below in Section 1.. Additional documentation may be found in [Russinoff, 1999a] and [Russinoff, 1998],

In Section 2., we present a precise description of the RTL language, including a rigorous definition of its semantics. This definition is the basis of a scheme for the automatic translation of RTL circuit descriptions into the logic of ACL2. The circuits that are handled by this translator include all combinational circuits as well as an important class of sequential circuits that may be characterized as *pipelines.* In particular, our methods are well suited to the verification of floating-point hardware designs, and have been applied to several of the arithmetic operations of the AMD Athlon processor, including an IEEE-compliant floating-point adder. In Section 3., as an illustration, we describe a proof of correctness of a simplified version of the Athlon multiplier. The complete proof may be found at the Web site that is associated with this book [Russinoff, 1999c].

1. A LIBRARY OF FLOATING-POINT ARITHMETIC

In this section, we list the basic definitions of our floating-point library [Russinoff, 1999b], along with some of the lemmas that are relevant to the proof described in Section 3.. In the mechanization of mathematical proofs of this sort, we have found that the most effective approach is to begin with an informal but rigorous and detailed written proof from which a formal ACL2 proof may be derived with minimal effort. Accordingly, our presentation will generally rely on traditional (informal) mathematical notation, rather than ACL2 syntax. The sets of rational numbers, nonzero rationals, integers, natural numbers (non-negative integers), and nonzero naturals will be denoted by $\mathbb{Q}$, $\mathbb{Q}^*$, $\mathbb{Z}$, $\mathbb{N}$, and $\mathbb{N}^*$, respectively. Function names will generally be printed in *italics*. Every function that we mention corresponds to an ACL2 function symbol, usually of the same name, which we denote in the `typewriter` font. In most cases, the formal definition of this function may be routinely derived from its informal specification and is therefore left to the reader.

Similarly, every lemma that we state corresponds to a mechanically verified ACL2 formula. In most cases, we omit the formal version, but include its name so that it may be easily located in the floating-point library.

Two functions that are central to our theory are *fl* and *cg*. For all $x \in \mathbb{Q}$, $fl(x)$ and $cg(x)$, abbreviated as $\lfloor x \rfloor$ and $\lceil x \rceil$, respectively, are the unique integers satisfying $\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1$ and $\lceil x \rceil \geq x > \lceil x \rceil - 1$. The corresponding formal definitions are based on the ACL2 primitive `floor`:

```
(defun fl (x) (floor x 1))
(defun cg (x) (- (fl (- x))))
```

Exercise: Prove (with the ACL2 prover) that for all $m \in \mathbb{N}$ and $n \in \mathbb{N}^*$,

$$\lfloor -(m+1)/n \rfloor = -\lfloor m/n \rfloor - 1.$$

Another important function is the integer remainder *rem* (corresponding to the ACL2 primitive `rem`), which may be characterized as follows:

Lemma 1..1 (`division`) *If $m \in \mathbb{N}$ and $n \in \mathbb{N}^*$, then*

$$n\lfloor m/n \rfloor + rem(m,n) = m.$$

The mechanically verified version of this lemma is:

```
(defthm division
  (implies (and (integerp m) (>= m 0)
                (integerp n) (> n 0))
           (equal (+ (* n (fl (/ m n)))
                     (rem m n))
                  m))).
```

1.1 BIT VECTORS

We exploit the natural correspondence between the bit vectors of length n and the natural numbers in the range $0 \leq x < 2^n$. Thus, for all $x, k \in \mathbb{N}$, we define

$$bitn(x,k) = rem(\lfloor x/2^k \rfloor, 2),$$

representing the k^{th} bit of the bit vector x. We also define, for all $x, i, j \in \mathbb{N}$, where $i \geq j$,

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

which extracts a field of bits from x, from the i^{th} down through the j^{th}. Following the standard notation of hardware description languages (see Section 2.), we shall abbreviate $bitn(x,k)$ as $x[k]$, and $bits(x,i,j)$ as $x[i:j]$.

The ACL2 formalization of both of these definitions is straightforward. However, instead of basing our definition of the ACL2 function `bitn` directly on the above, we make use of the primitive `logbitp`, for the sake of execution efficiency:

```
(defun bitn (x n) (if (logbitp n x) 1 0)).
```

After deriving the desired relation from the formal definition, the definition may be disabled:

```
(defthm bitn-def
    (implies (and (integerp x) (>= x 0)
                  (integerp k) (>= k 0))
             (= (bitn x k)
                (rem (fl (/ x (expt 2 k))) 2)))
   :rule-classes ()
   :hints ...)
(in-theory (disable bitn))
```

Among the library lemmas pertaining to `bitn` and `bits`, we shall require the following:

Lemma 1..2 **(`bit-expo-a`)** *For all $x, n \in \mathbb{N}$, if $x < 2^n$, then $x[n] = 0$;*

Lemma 1..3 **(`bit-expo-b`)** *For all $x, n, k \in \mathbb{N}$, if $k < n$ and $2^n - 2^k \leq x < 2^n$, then $x[k] = 1$.*

Lemma 1..4 **(`bit+a`)** *For all $x, n \in \mathbb{N}$, $(x + 2^n)[n] \neq x[n]$.*

Lemma 1..5 (bits-bitn) *For all* $x \in \mathbb{N}$ *and* $n \in \mathbb{N}^*$, $x[n:0] = 0 \Leftrightarrow x[n] = x[n-1:0] = 0$.

Lemma 1..6 (bit-bits-b) *For all* $x, i, j, k \in \mathbb{N}$, *if* $i \geq j + k$, *then* $x[i:j][k] = x[k+j]$;

Lemma 1..7 (bit-bits-c) *For all* $x, i, j, k, \ell \in \mathbb{N}$, *if* $i \geq j + k$, *then* $x[i:j][k:\ell] = x[k+j:\ell+j]$.

We have three binary logical operations on bit vectors, for which we again use abbreviations motivated by RTL notation: $logand(x, y) = x \;\&\; y$, $logior(x, y) = x \mid y$, and $logxor(x, y) = x \;\hat{}\; y$. These functions are most naturally defined recursively, e.g.,

$$x \;\&\; y = \begin{cases} 0 & \text{if } x = 0 \\ 2(\lfloor x/2 \rfloor \;\&\; \lfloor y/2 \rfloor) + 1 & \text{if } x \text{ and } y \text{ are both odd} \\ 2(\lfloor x/2 \rfloor \;\&\; \lfloor y/2 \rfloor) & \text{otherwise.} \end{cases}$$

However, since the functions `logand`, etc., are already implemented as ACL2 primitives, we once again derive the desired equations as consequences of the relevant axioms. For example:

```
(defthm logand-def
   (implies (and (integerp x) (>= x 0)
                 (integerp y) (>= y 0))
            (= (logand x y)
               (+ (* 2 (logand (fl (/ x 2)) (fl (/ y 2))))
                  (logand (rem x 2) (rem y 2)))))
 :rule-classes ()
 :hints ...).
```

The following library lemmas are cited in the proof of Section 3.:

Lemma 1..8 (bit-dist-a) *For all* $x, y, n \in \mathbb{N}$,

$$(x \;\&\; y)[n] = x[n] \;\&\; y[n].$$

Lemma 1..9 (bit-dist-b) *For all* $x, y, n \in \mathbb{N}$,

$$(x \mid y)[n] = x[n] \mid y[n].$$

Lemma 1..10 (and-dist-a) *For all* $x, y, n \in \mathbb{N}$, $x \;\&\; y \leq x$.

Lemma 1..11 (and-dist-c) *For all* $x, y, n \in \mathbb{N}$,

$$rem(x \;\&\; y, 2^n) = rem(x, 2^n) \;\&\; y.$$

Lemma 1..12 (`and-dist-d`) *For all* $x, y, n \in \mathbb{N}$, *if* $x < 2^n$, *then*

$$x \;\&\; y = x \;\&\; rem(y, 2^n).$$

Lemma 1..13 (`or-dist-a`) *For all* $x, y, n \in \mathbb{N}$, *if* $x < 2^n$ *and* $y < 2^n$, *then* $x \mid y < 2^n$.

Lemma 1..14 (`or-dist-d`) *For all* $x, y, n \in \mathbb{N}$,

$$rem(x \mid y, 2^n) = rem(x, 2^n) \mid rem(y, 2^n).$$

1.2 FLOATING-POINT REPRESENTATION

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

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

where $sgn(x) \in \{1, -1\}$ (the *sign* of x), $1 \leq sig(x) < 2$ (the *significand* of x), and $expo(x) \in \mathbb{Z}$ (the *exponent* of x).

The recursive definition of `expo` requires an explicitly supplied measure:

```
(defun expo-measure (x)
  (cond ((not (rationalp x)) 0)
        ((< x 0) '(2 . 0))
        ((< x 1) '(1 . 0))
        (t (fl x))))
(defun expo (x)
  (declare (xargs :measure (expo-measure x)))
  (cond ((or (not (rationalp x)) (= x 0)) ())
        ((< x 0) (expo (- x)))
        ((< x 1) (- (expo (/ x))))
        ((< x 2) 0)
        (t (1+ (expo (/ x 2))))))
```

The definitions of `sgn` and `sig` are then straightforward:

```
(defun sgn (x) (if (< x 0) -1 +1))
(defun sig (x) (* (abs x) (expt 2 (- (expo x))))).
```

The following properties are immediate consequences of the definitions:

Lemma 1..15 (`fp-rep`) *For all* $x \in \mathbb{Q}^*$, $x = sgn(x)sig(x)2^{expo(x)}$.

Lemma 1..16 (`expo-lower-bound`) *For all* $x \in \mathbb{Q}^*$, $|x| \geq 2^{expo(x)}$.

Lemma 1..17 (`expo-upper-bound`) *For all $x \in \mathbb{Q}^*$, $|x| < 2^{expo(x)+1}$.*

Lemma 1..18 (`fp-rep-unique`) *If $x, y \in \mathbb{Q}^*$, $1 \leq y < 2$, $n \in \mathbb{Z}$, and $|x| = 2^n y$, then $y = sig(x)$ and $n = expo(x)$.*

Lemma 1..19 (`sig-expo-shift`) *If $x \in \mathbb{Q}^*$, $n \in \mathbb{Z}$, and $y = 2^n x$, then $sig(y) = sig(x)$ and $expo(y) = n + expo(x)$.*

A floating point representation of x is a bit vector consisting of three fields, corresponding to $sgn(x)$, $sig(x)$, and $expo(x)$. A *floating point format* is a pair of positive integers $\phi = (\sigma, \epsilon)$, representing the number of bits allocated to $sig(x)$ and $expo(x)$, respectively. If $z \in \mathbb{N}$, then the *sign*, *exponent*, and *significand fields* of z with respect to ϕ are

$$sgnf(z, \phi) = z[\sigma + \epsilon],$$

$$expf(z, \phi) = z[\sigma + \epsilon - 1 : \sigma],$$

and

$$sigf(z, \phi) = z[\sigma - 1 : 0],$$

respectively. If $sigf(z, \phi)[\sigma - 1] = 1$, then z is a *normal ϕ-encoding.*

The number x represented by a normal ϕ-encoding z, where $\phi = (\sigma, \epsilon)$, is given by $sgn(x) = (-1)^{sgnf(z,\phi)}$, $sig(x) = 2^{1-\sigma} sigf(z, \phi)$, and $expo(x) = expf(z, \phi) - (2^{\epsilon-1} - 1)$. Thus, we define

$$decode(z, \phi) = (-1)^{sgnf(z,\phi)} \cdot sigf(z, \phi) \cdot 2^{expf(z,\phi) - 2^{\epsilon-1} - \sigma + 2}.$$

Note that the exponent field is biased in order to provide for an exponent range $1 - 2^{\epsilon-1} \leq expo(x) \leq 2^{\epsilon-1}$.

Let $x \in \mathbb{Q}^*$ and $n \in \mathbb{N}^*$. Then the predicate $exactp(x, n)$ is true, and we shall say that x is *n-exact*, if $sig(x)2^{n-1} \in \mathbb{Z}$. The predicate $repp(x, \phi)$ is true if x is σ-exact and $-2^{\epsilon-1} + 1 \leq expo(x) \leq 2^{\epsilon-1}$. It is clear that the latter condition holds iff x is *representable* with respect to ϕ, i.e., for some $z \in \mathbb{N}$, $x = decode(z, \phi)$. We also have the following characterization of n-exact naturals:

Lemma 1..20 (`exact-bits-a-b`) *Let $x, n, k \in \mathbb{N}^*$, $2^{n-1} \leq x < 2^n$. and $k < n$. Then 2^k divides x iff x is $(n-k)$-exact.*

Another useful lemma characterizes the "successor" of an n-exact number:

Lemma 1..21 (fp+1) *Let $x, y \in \mathbb{Q}^*$ and $n \in \mathbb{N}^*$. If $y > x > 0$ and x and y are both n-exact, then $y \geq x + 2^{expo(x)+1-n}$.*

Exercise: Prove that if x is k-exact and x^2 is $2n$-exact, then x is n-exact. (Note: the hypothesis that x is k-exact may be replaced with the weaker assumption that x is rational, but the ACL2 proof then becomes more complicated.)

The IEEE standard supports three formats, $(24, 7)$, $(53, 10)$, $(64, 15)$, which correspond to *single*, *double*, and *extended* precision, respectively. In the discussion of our floating-point multipler, floating point numbers will always be represented in the extended precision format, $\mathcal{E} = (64, 15)$. We shall abbreviate $decode(z, \mathcal{E})$ as $\hat{z}$:

```
(defun extfmt () '(64 15))
(defun hat (z) (decode z (extfmt)))
```

1.3 ROUNDING

A *rounding mode* is a function $\mathcal{M}$ that computes an n-exact number $\mathcal{M}(x, n)$ corresponding to an arbitrary rational x and a degree of precision $n \in \mathbb{N}^*$. The most basic rounding mode, *truncation* (round toward 0), is defined by

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

Thus, $trunc(x, n)$ is the n-exact number y satifying $|y| \leq |x|$ that is closest to x. Similarly, rounding *away* from 0 is given by

$$away(x, n) = sgn(x)\lceil 2^{n-1} sig(x) \rceil 2^{expo(x)-n+1},$$

and three other modes are defined simply in terms of those two: $inf(x, n)$ (round toward ∞), $minf(x, n)$ (round toward $-\infty$), and $near(x, n)$ (round to the nearest n-exact number, with ambiguities resolved by selecting $(n-1)$-exact values).

The modes that are supported by the IEEE standard are *trunc*, *near*, *inf*, and *minf*. We shall refer to these as *IEEE rounding modes.*

```
(defun ieee-mode-p (mode)
  (member mode '(trunc inf minf near)))
```

If $\mathcal{M}$ is any rounding mode, $\sigma \in \mathbb{N}^*$, and $x \in \mathbb{Q}$, then we define

$$rnd(x, \mathcal{M}, \sigma) = \mathcal{M}(x, \sigma).$$

```
(defun rnd (x mode n)
```

```
(case mode
  (trunc (trunc x n))
  (inf (inf x n))
  (minf (minf x n))
  (near (near x n))))
```

Lemma 1..22 (`rnd-shift`) *If $x \in \mathbb{Q}$, $n \in \mathbb{N}^*$, and $k \in \mathbb{Z}$, then for any IEEE rounding mode $\mathcal{M}$,*

$$rnd(2^k x, \mathcal{M}, n) = 2^k rnd(x, \mathcal{M}, n).$$

Lemma 1..23 (`rnd-flip`) *If $x \in \mathbb{Q}$ and $n \in \mathbb{N}^*$, then for any IEEE rounding mode $\mathcal{M}$,*

$$rnd(-x, \mathcal{M}, n) = -rnd(x, \mathcal{M}', n),$$

where

$$\mathcal{M}' = \begin{cases} minf, & \text{if } \mathcal{M} = inf \\ inf, & \text{if } \mathcal{M} = minf \\ \mathcal{M}, & \text{if } \mathcal{M} = trunc \text{ or } \mathcal{M} = near. \end{cases}$$

The following three lemmas justify the implementation of rounding that is employed in the AMD Athlon floating-point unit:

Lemma 1..24 (`bits-trunc`) *Let $x, m, n, k \in \mathbb{N}$. If $0 < k < n \leq m$ and $2^{n-1} \leq x < 2^n$, then*

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

Lemma 1..25 (`away-imp`) *Let $x \in \mathbb{Q}$, $x > 0$, $m \in \mathbb{N}^*$, and $n \in \mathbb{N}^*$. If x is m-exact and $m \geq n$, then*

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

Lemma 1..26 (`near-trunc`) *Let $n \in \mathbb{Z}$, $n > 1$, and $x \in \mathbb{Q}$, $x > 0$. If x is $(n+1)$-exact but not n-exact, then*

$$near(x, n) = trunc(x + 2^{expo(x)-n}, n - 1);$$

otherwise,

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

2. THE RTL LANGUAGE

In this section, we present a precise syntactic and semantic definition of the AMD RTL language. We also identify a class of programs that admit a particularly simple semantic description. For these programs, called *simple pipelines*, the value of each output may be computed in a natural way as a function of the inputs.

One advantage of using our own design language is that we are free to modify its compiler to suit our needs. Thus, we have implemented an automatic translator that generates a functional representation in ACL2 of any simple pipeline, based on the compiler's internal parse tree. Our floating-point multiplier will serve as an illustration.

2.1 LANGUAGE DEFINITION

The language is based on a class of identifiers called *signals*, and a class of character strings called *numerals*. A *binary numeral* has the form $\mathtt{b}b_1 \ldots b_k$, where the b_i are binary digits; *decimal* and *hexadecimal numerals* similarly use the prefixes `d` and `h`, although `d` may be omitted. The natural number represented by a numeral ν will be denoted as $\bar{\nu}$.

A circuit description includes *input declarations*, *combinational assignments*, *sequential assignments*, and *constant definitions*, which have the forms

$$\texttt{input}\ s[\nu:0]; \tag{1.1}$$

$$s[\nu:0]\ \texttt{=}\ E; \tag{1.2}$$

$$s[\nu:0]\ \texttt{<=}\ E; \tag{1.3}$$

and

$$\texttt{`define}\ r\ \nu \tag{1.4}$$

respectively, where ν is a numeral, s is a signal, E is an *expression of size* $\bar{\nu}+1$ as defined below, and r may be any identifier. We may abbreviate $s[0:0]$ as s.

Each signal s occurring anywhere in a description must appear in exactly one of the three contexts (1.1), (1.2), and (1.3), and is called an *input*, a *wire*, or a *register*, accordingly, of *size* $\bar{\nu}+1$. In cases (1.2) and (1.3), we shall say that E is *the expression for* s. Any signal may also occur in an *output declaration*,

$$\texttt{output}\ s[\nu:0]; \tag{1.5}$$

and is then also called an *output*.

The effect of a constant definition (1.4) is simply that any subsequent occurrence of `r is taken as an abbreviation for ν.

If s is a wire, E is the expression for s, and s' is any signal, then s *depends* on s' iff either s' occurs in E or some wire occurring in E depends on s'. It is a syntactic requirement of the language that no wire depends on itself.

Let I, O, W, R, and S denote the sets of inputs, outputs, wires, registers, and signals, respectively, of a circuit description $\mathcal{D}$. Then S is the disjoint union $I \cup W \cup R$, and $O \subset S$. A mapping from I, O, or R to $\mathbb{N}$ is called an *input valuation*, an *output valuation*, or a *register state* for $\mathcal{D}$, respectively. If R is empty, then $\mathcal{D}$ admits only the null register state and we shall say that $\mathcal{D}$ is *combinational*; otherwise, $\mathcal{D}$ is *sequential*.

Next, we define the set of *expressions* of the language corresponding to the circuit description $\mathcal{D}$. For each expression E, we also define the *size* of E, as well as the *value* of E, $val_{\mathcal{D}}(E, \mathcal{I}, \mathcal{R})$, for a given input valuation $\mathcal{I}$ and register state $\mathcal{R}$:

(1) If ν and μ are numerals such that $\bar{\nu} > 0$ and $\bar{\mu} < 2^{\bar{\nu}}$, then ν'μ is a *constant expression* of size $\bar{\nu}$ and

$$val_{\mathcal{D}}(\nu\text{'}\mu, \mathcal{I}, \mathcal{R}) = \bar{\mu}.$$

(2) If s is a signal of size n, then s is an expression of size n, and

$$val_{\mathcal{D}}(s, \mathcal{I}, \mathcal{R}) = \begin{cases} \mathcal{I}(s) & \text{if } s \in I \\ \mathcal{R}(s) & \text{if } s \in R \\ val_{\mathcal{D}}(E, \mathcal{I}, \mathcal{R}) & \text{if } s \in W \text{ and } E \text{ is its expression.} \end{cases}$$

(3) If s is a signal and λ and μ are numerals with $\bar{\lambda} \geq \bar{\mu}$, then $s[\lambda : \mu]$ is an expression of size $\bar{\lambda} - \bar{\mu} + 1$, and

$$val_{\mathcal{D}}(s[\lambda : \mu], \mathcal{I}, \mathcal{R}) = bits(val_{\mathcal{D}}(s, \mathcal{I}, \mathcal{R}), \bar{\lambda}, \bar{\mu}).$$

We may abbreviate $s[\lambda : \lambda]$ as $s[\lambda]$.

(4) If E is an expression of size n, then ~E is an expression of size n, and

$$val_{\mathcal{D}}(\text{\textasciitilde}E, \mathcal{I}, \mathcal{R}) = 2^n - val_{\mathcal{D}}(E, \mathcal{I}, \mathcal{R}) - 1.$$

(5) If E_1 and E_2 are expressions of equal size, then (E_1 == E_2) is an expression of size 1, with

$$val_{\mathcal{D}}((E_1 \texttt{ == } E_2), \mathcal{I}, \mathcal{R}) = \begin{cases} 1 & \text{if } val_{\mathcal{D}}(E_1, \mathcal{I}, \mathcal{R}) = val_{\mathcal{D}}(E_2, \mathcal{I}, \mathcal{R}) \\ 0 & \text{if } val_{\mathcal{D}}(E_1, \mathcal{I}, \mathcal{R}) \neq val_{\mathcal{D}}(E_2, \mathcal{I}, \mathcal{R}). \end{cases}$$

(6) If E_1 and E_2 are expressions of size n, then $(E_1$ & $E_2)$, $(E_1$ | $E_2)$, and $(E_1$ ^ $E_2)$ are expressions of size n, with

$$val_{\mathcal{D}}((E_1 \ \& \ E_2), \mathcal{I}, \mathcal{R}) = logand(val_{\mathcal{D}}(E_1, \mathcal{I}, \mathcal{R}), val_{\mathcal{D}}(E_2, \mathcal{I}, \mathcal{R}))$$

and similar definitions for the other two operators.

(7) If E_1 and E_2 are expressions of size n, then $(E_1 + E_2)$ is an expression of size n, with

$$val_{\mathcal{D}}(E_1 + E_2), \mathcal{I}, \mathcal{R}) = rem(val_{\mathcal{D}}(E_1, \mathcal{I}, \mathcal{R}) + val_{\mathcal{D}}(E_2, \mathcal{I}, \mathcal{R}), 2^n).$$

Multiplication is defined similarly.

(8) If E_1 and E_2 are any expressions of sizes of n_1 and n_2, respectively, then $\{E_1, E_2\}$ is an expression of size $n_1 + n_2$, with

$$val_{\mathcal{D}}(\{E_1, E_2\}, \mathcal{I}, \mathcal{R}) = 2^{n_2} val_{\mathcal{D}}(E_1, \mathcal{I}, \mathcal{R}) + val_{\mathcal{D}}(E_2, \mathcal{I}, \mathcal{R}).$$

For $k > 2$, $\{E_1, \ldots, E_k\}$ is an abbreviation for $\{E_1, \{E_2, \ldots, E_k\}\}$ If $E_1 = \ldots = E_k$ and ν is a numeral with $\bar{\nu} = k$, then we may further abbreviate $\{E_1, \ldots, E_k\}$ as $\{\nu \ \{E_1\}\}$.

(9) If B is an expression of size 1 and E_1 and E_2 are expressions of size n, then $(B \ \texttt{?} \ E_1 \ \texttt{:} \ E_2)$ is an expression of size n, and

$$val_{\mathcal{D}}((B \ \texttt{?} \ E_1 \ \texttt{:} \ E_2), \mathcal{I}, \mathcal{R}) = \begin{cases} val_{\mathcal{D}}(E_1, \mathcal{I}, \mathcal{R}) & \text{if } val_{\mathcal{D}}(B, \mathcal{I}, \mathcal{R}) \neq 0 \\ val_{\mathcal{D}}(E_2, \mathcal{I}, \mathcal{R}) & \text{if } val_{\mathcal{D}}(B, \mathcal{I}, \mathcal{R}) = 0. \end{cases}$$

(10) If D, $E_1, \ldots, E_k$ are expressions of size n and $F_1, \ldots, F_k$ are expressions of size m, then

$$F = \texttt{case(}D\texttt{)} \ E_1 : \ F_1; \quad \ldots \quad E_k : \ F_k; \ \ \texttt{endcase}$$

is an expression of size m, and

$$val_{\mathcal{D}}(F, \mathcal{I}, \mathcal{R}) = \begin{cases} val_{\mathcal{D}}(F_1, \mathcal{I}, \mathcal{R}) & \text{if } val_{\mathcal{D}}(D \ \texttt{==} \ E_1, \mathcal{I}, \mathcal{R}) = 1 \\ 0 & \text{if } val_{\mathcal{D}}(D \ \texttt{==} \ E_1, \mathcal{I}, \mathcal{R}) = 0, \ k = 1 \\ val_{\mathcal{D}}(F', \mathcal{I}, \mathcal{R}) & \text{if } val_{\mathcal{D}}(D \ \texttt{==} \ E_1, \mathcal{I}, \mathcal{R}) = 0, \ k > 1, \end{cases}$$

where

$$F' = \texttt{case(}D\texttt{)} \ \ E_2 : \ F_2; \quad \ldots \quad E_k : \ F_k; \ \ \texttt{endcase}$$

The semantics of circuit descriptions are based on an underlying notion of *cycle*. Let $\mathcal{I}_1, \mathcal{I}_2, \ldots$ be a sequence of input valuations and let $\mathcal{R}_1$ be a register state for $\mathcal{D}$. We shall think of each $\mathcal{I}_k$ as representing the values of the input signals of $\mathcal{D}$ on the k^{th} cycle of an execution, and $\mathcal{R}_1$ as an initial set of register values. From these functions we shall construct a sequence of output valuations, $\mathcal{O}_1, \mathcal{O}_2, \ldots$, representing the output values produced by $\mathcal{D}$ on successive cycles.

First, we define a function $next_{\mathcal{D}}$, which represents the dependence of the register state for a given cycle on the input valuation and register state for tyhe preceding cycle. Given an input valuation $\mathcal{I}$ and a register state $\mathcal{R}$, the register state $next_{\mathcal{D}}(\mathcal{I}, \mathcal{R}) = \mathcal{R}'$ is defined as follows: if $s \in R$ and E is the expression for s, then

$$\mathcal{R}'(s) = val_{\mathcal{D}}(E, \mathcal{I}, \mathcal{R}).$$

Now, for each $k \geq 2$, let $\mathcal{R}_k = next_{\mathcal{D}}(\mathcal{I}_{k-1}, \mathcal{R}_{k-1})$. The output valuations $\mathcal{O}_1, \mathcal{O}_2, \ldots$ are computed as follows: for each output signal s,

$$\mathcal{O}_k(s) = val_{\mathcal{D}}(s, \mathcal{I}_k, \mathcal{R}_k).$$

2.2 SIMPLE PIPELINES

If $\mathcal{D}$ is combinational, then we may write $val_{\mathcal{D}}(s, \mathcal{I})$ unambiguously, omitting the third argument, and consequently, the output valuation $\mathcal{O}_k$, as defined above, is completely determined by $\mathcal{I}_k$. Thus, the external behavior of a combinational circuit may be described by a functional dependence of outputs on inputs. The same is true of a certain class of sequential circuits, which we describe below. For any circuit in this class, there is a number n such that for each $k \geq n$, the output valuation $\mathcal{O}_k$ is completely determined by the input valuation $\mathcal{I}_{k-n+1}$.

We shall say that a circuit description $\mathcal{D}$ is an *n-cycle simple pipeline* if there exists a function $\psi : S \to \{1, \ldots, n\}$ such that

(1) if $s \in I$, then $\psi(s) = 1$;

(2) if $s \in W$ and E is the expression for s, then $\psi(s') = \psi(s)$ for each signal s' occurring in E;

(3) if $s \in R$ and E is the expression for s, then $\psi(s) > 1$ and $\psi(s') = \psi(s) - 1$ for each signal s' occurring in E;

(4) if $s \in O$, then $\psi(s) = n$.

Note that a 1-cycle simple pipeline is just a combinational circuit.

The main consequences of the above definition are given by Lemmas 2..1 and 2..2 below. The proofs of these lemmas use an induction scheme based on a well-founded partial ordering of the set of expressions of $\mathcal{D}$, defined as follows: For any expression E, let $\Psi(E)$ be the maximum, over all signals s occurring in E, of $\psi(s)$, and let $\Lambda(E)$ be the maximum, over all signals s occurring in E, of the number of signals on which s depends. Then for any two expressions E_1 and E_2, E_1 *precedes* E_2 iff

(a) $\Psi(E_1) < \Psi(E_2)$, or

(b) $\Psi(E_1) = \Psi(E_2)$ and $\Lambda(E_1) < \Lambda(E_2)$, or

(c) $\Psi(E_1) = \Psi(E_2)$, $\Lambda(E_1) = \Lambda(E_2)$, and E_1 is a subexpression of E_2.

According to our first lemma, every n-cycle simple pipeline has the property that the values of the inputs on any cycle determine the values of the outputs $n-1$ cycles later:

Lemma 2..1 *Let $\mathcal{I}_1, \ldots, \mathcal{I}_n, \mathcal{I}'_1, \ldots, \mathcal{I}'_n$ be input valuations and let $\mathcal{R}_1$ and $\mathcal{R}'_1$ be register states for an n-cycle simple pipeline $\mathcal{D}$. For $k = 2, \ldots, n$, let $\mathcal{R}_k = next_{\mathcal{D}}(\mathcal{I}_{k-1}, \mathcal{R}_{k-1})$ and $\mathcal{R}'_k = next_{\mathcal{D}}(\mathcal{I}'_{k-1}, \mathcal{R}'_{k-1})$. If $\mathcal{I}_1 = \mathcal{I}'_1$, then for every output s of $\mathcal{D}$,*

$$val_{\mathcal{D}}(s, \mathcal{I}_n, \mathcal{R}_n) = val_{\mathcal{D}}(s, \mathcal{I}'_n, \mathcal{R}'_n).$$

Proof: We shall show that for all k, $1 \leq k \leq n$, if E is any expression of $\mathcal{D}$ such that $\psi(s) = k$ for every signal s occurring in E, then $val_{\mathcal{D}}(E, \mathcal{I}_k, \mathcal{R}_k) = val_{\mathcal{D}}(E, \mathcal{I}'_k, \mathcal{R}'_k)$. The proof is by induction, based on the partial ordering of expressions defined above. Assume that the claim holds for all expressions that precede a given expression E. To show that the claim holds for E as well, we shall examine the only nontrivial case: E is a signal s.

If s is an input, then $k = 1$ and

$$val_{\mathcal{D}}(s, \mathcal{I}_1, \mathcal{R}_1) = \mathcal{I}_1(s) = \mathcal{I}'_1(s) = val_{\mathcal{D}}(s, \mathcal{I}'_1, \mathcal{R}'_1).$$

Thus, we may assume that s is a wire or a register. Let F be the expression for s.

Suppose s is a wire. Then $\psi(r) = k$ for each signal r occurring in F. Therefore, $\Psi(s) = k = \Psi(F)$ and $\Lambda(s) \geq \Lambda(F) + 1$, hence F precedes s and by our inductive hypothesis,

$$val_{\mathcal{D}}(s, \mathcal{I}_k, \mathcal{R}_k) = val_{\mathcal{D}}(F, \mathcal{I}_k, \mathcal{R}_k) = val_{\mathcal{D}}(F, \mathcal{I}'_k, \mathcal{R}'_k) = val_{\mathcal{D}}(s, \mathcal{I}'_k, \mathcal{R}'_k).$$

Finally, suppose s is a register. Then $k > 1$ and $\psi(r) = k - 1$ for each signal r occurring in F. Thus, $\Psi(F) = k-1 < k = \Psi(s)$, so F precedes s, and we may conclude that $val_{\mathcal{D}}(F, \mathcal{I}_{k-1}, \mathcal{R}_{k-1}) = val_{\mathcal{D}}(F, \mathcal{I}'_{k-1}, \mathcal{R}'_{k-1})$. But since $\mathcal{R}_k = next_{\mathcal{D}}(\mathcal{I}_{k-1}, \mathcal{R}_{k-1})$,

$$val_{\mathcal{D}}(s, \mathcal{I}_k, \mathcal{R}_k) = \mathcal{R}_k(s) = val_{\mathcal{D}}(F, \mathcal{I}_{k-1}, \mathcal{R}_{k-1}),$$

and similarly,

$$val_{\mathcal{D}}(s, \mathcal{I}'_k, \mathcal{R}'_k) = \mathcal{R}'_k(s) = val_{\mathcal{D}}(F, \mathcal{I}'_{k-1}, \mathcal{R}'_{k-1}). \ \Box$$

Now, let $\mathcal{D}, \mathcal{I}_1, \ldots, \mathcal{I}_n$, and $\mathcal{R}_1, \ldots, \mathcal{R}_n$ be as described in Lemma 2..1. Let $\mathcal{O} : O \to \mathbb{N}$ be defined by $\mathcal{O}(s) = val_{\mathcal{D}}(s, \mathcal{I}_n, \mathcal{R}_n)$. Then according to the lemma, $\mathcal{O}$ is determined by $\mathcal{I} = \mathcal{I}_1$ alone, and we may define $out_{\mathcal{D}}(\mathcal{I}) = \mathcal{O}$. Thus, for an n-cycle simple pipeline, there is a natural mapping from input valuations to output valuations.

If we are interested only in the mapping $out_{\mathcal{D}}$, then any n-cycle simple pipeline may be replaced with a combinational circuit:

Lemma 2..2 *Let $\mathcal{D}$ be an n-cycle simple pipeline, and let $\tilde{\mathcal{D}}$ be the circuit description obtained from $\mathcal{D}$ by replacing each sequential assignment (1.3) by the corresponding combinational assignment (1.2). Then $\tilde{\mathcal{D}}$ is a combinational circuit description and $out_{\mathcal{D}} = out_{\tilde{\mathcal{D}}}$.*

Proof: To prove that $\tilde{\mathcal{D}}$ is a combinational circuit description, it will suffice to show that $\tilde{\mathcal{D}}$ is a well-formed circuit description. If not, then there must be signals $s_1, \ldots, s_k$ such that $s_1 = s_k$ and for $i = 1, \ldots, k-1$, s_i occurs in the expression for s_{i+1}. But since $\psi(s_1) \leq \ldots \leq \psi(s_k) = \psi(s_1)$, we would then have $\psi(s_1) = \ldots = \psi(s_k)$, which would imply that each s_i is a wire of $\mathcal{D}$, contradicting the assumption that $\mathcal{D}$ is well-formed.

Now, given an input valuation $\mathcal{I}$ for $\mathcal{D}$ (and thus for $\tilde{\mathcal{D}}$), let $\mathcal{O} = out_{\mathcal{D}}(\mathcal{I})$ and $\tilde{\mathcal{O}} = out_{\tilde{\mathcal{D}}}(\mathcal{I})$. We must show that $\mathcal{O}(s) = \tilde{\mathcal{O}}(s)$ for every output signal s. Let $\mathcal{I}_1, \ldots, \mathcal{I}_n$ be input valuations for $\mathcal{D}$, where $\mathcal{I}_1 = \mathcal{I}$, and let $\mathcal{R}_1, \ldots, \mathcal{R}_n$ be register states such that $\mathcal{R}_{k+1} = next_{\mathcal{D}}(\mathcal{I}_k, \mathcal{R}_k)$ for $k = 1, \ldots n-1$. Then $\mathcal{O}(s) = val_{\mathcal{D}}(s, \mathcal{I}_n, \mathcal{R}_n)$. On the other hand, since $\tilde{\mathcal{D}}$ is combinational, $\tilde{\mathcal{O}}(s) = val_{\tilde{\mathcal{D}}}(s, \mathcal{I}_1)$. Thus, we may complete the proof by showing that if E is any expression such that $\psi(s) = k$ for every signal s occurring in E, then $val_{\mathcal{D}}(E, \mathcal{I}_k, \mathcal{R}_k) = val_{\tilde{\mathcal{D}}}(E, \mathcal{I}_1)$.

Using the same induction scheme as in Lemma 2..1, we again note that in the only nontrivial case, E is a signal s. If s is an input, then $k = 1$ and

$$val_{\mathcal{D}}(s, \mathcal{I}_1, \mathcal{R}_1) = \mathcal{I}_1(s) = val_{\tilde{\mathcal{D}}}(s, \mathcal{I}_1).$$

If s is a wire of $\mathcal{D}$, and hence of $\tilde{\mathcal{D}}$, and F is the expression for s, then

$$val_{\mathcal{D}}(s, \mathcal{I}_k, \mathcal{R}_k) = val_{\mathcal{D}}(F, \mathcal{I}_k, \mathcal{R}_k) = val_{\tilde{\mathcal{D}}}(F, \mathcal{I}_1) = val_{\tilde{\mathcal{D}}}(s, \mathcal{I}_1).$$

In the remaining case, s is a register of $\mathcal{D}$ and a wire of $\tilde{\mathcal{D}}$. If F is the expression for s (in both contexts), then

$$\begin{aligned} val_{\mathcal{D}}(s, \mathcal{I}_k, \mathcal{R}_k) &= \mathcal{R}_k(s) = val_{\mathcal{D}}(F, \mathcal{I}_{k-1}, \mathcal{R}_{k-1}) \\ &= val_{\tilde{\mathcal{D}}}(F, \mathcal{I}_1) = val_{\tilde{\mathcal{D}}}(s, \mathcal{I}_1). \ \Box \end{aligned}$$

2.3 TRANSLATION TO ACL2

One of the functions of the RTL-ACL2 translator is to analyze the dependencies among the signals of a circuit description to determine whether it satisfies the definition of a simple pipeline. Once this is established, an ACL2 function is constructed from each wire and register definition, ignoring the distinction between the two, in accordance with Lemma 2..2. This function computes the value of the signal for a given input valuation in terms of the values of the signals that occur in its defining expression. Thus, each RTL construct in the expression for the signal is replaced with the corresponding ACL2 construct, as determined by the definition of evaluation given in Subsection 2.1.

For example, the combinational assignment

```
sig_of[128:0] = {1'b0, carry_of, 127'b0} |
                (add_of[128:0] & {1'b0, mask_of[127:0]});
```

of the circuit `FMUL` (Fig. 1.4) generates the definition

```
(defun sig_of (carry_of add_of mask_of)
  (logior (cat carry_of 0 127)
          (logand add_of mask_of)))
```

while the sequential assignment

```
sticky_of <= case(pc_C3)
               `SNG : ~(prod[102:0] == 103'b0);
               `DBL : ~(prod[73:0] == 74'b0);
             endcase;
```

(Fig. 1.3) produces

```
(defun sticky_of (pc_c3 prod)
  (cond ((equal pc_c3 0)
         (if (equal (bits prod 102 0) 0) 0 1))
        ((equal pc_c3 1)
         (if (equal (bits prod 73 0) 0) 0 1)))).
```

Finally, an additional function is defined for each output signal, which binds each non-input signal in succession to its value for a given set of input vaues, and returns the value of the output. For the circuit `FMUL`, which has only one output, z, a single function is generated as follows:

```
(defun fmul (x y rc pc)
  (let* ((sgnx (sgnx x))
```

```
  (sgny (sgny y))
  (expx (expx x))
  (expy (expy y))
  (sigx (sigx x))
  (sigy (sigy y))
  (sgnz (sgnz sgnx sgny))
  (exp_sum (exp_sum expx expy))
    ...
  (carry_nof (carry_nof add_nof))
  (sig_of (sig_of carry_of add_of mask_of))
  (sig_nof (sig_nof carry_nof add_nof mask_nof))
  (sigz (sigz overflow sig_of sig_nof))
  (exp_of (exp_of exp_sum_c4 carry_of))
  (exp_nof (exp_nof exp_sum_c4 carry_nof))
  (expz (expz overflow exp_of exp_nof))
  (z (z sgnz_c4 expz sigz)))
 z)).
```

It is evident that this function accurate represents the dependence of the output `z` on the inputs, i.e., if the bindings of `x`, `y`, `rc`, and `pc` are given by an input valuation $\mathcal{I}$, then the value computed by `fmul` is $out_{\mathtt{FMUL}}(\mathcal{I})(\mathtt{z})$.

3. CORRECTNESS OF THE MULTIPLIER

Let $\mathcal{I}$ be a fixed input valuation for `FMUL`. We shall adopt the convention of italicizing each signal to denote its value for $\mathcal{I}$, e.g.,

$$val_{\mathtt{FMUL}}(\mathtt{sigz}, \mathcal{I}) = sigz$$

and since `rc` in an input,

$$val_{\mathtt{FMUL}}(\mathtt{rc}, \mathcal{I}) = \mathcal{I}(\mathtt{rc}) = rc.$$

Note that `FMUL` has four inputs: x and y are $\mathcal{E}$-encodings of the numbers to be multiplied, rc is a 2-bit encoding of the mode to be used in rounding the result, and pc is a 1-bit encoding of the desired degree of precision, corresponding to either single (24-bit) or double (53-bit) precision.

We would like to show that the circuit meets the main requirement for IEEE compliance, as stipulated in the floating-point standard [IEEE, 1985]:

> [Multiplication] shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result ...

Thus, the output z must satisfy the following:

```
module FMUL;

//*****************************************************************
// Declarations
//*****************************************************************

//Precision and rounding control:

`define SNG   1'b0        // single precision
`define DBL   1'b1        // double precision
`define NRE   2'b00       // round to nearest
`define NEG   2'b01       // round to minus infinity
`define POS   2'b10       // round to plus infinity
`define CHP   2'b11       // truncate

//Parameters:

input x[79:0];            //first operand
input y[79:0];            //second operand
input rc[1:0];            //rounding control
input pc;                 //precision control
output z[79:0];           //rounded product


//*****************************************************************
// First Cycle
//*****************************************************************

//Operand fields:

sgnx = x[79]; sgny = y[79];                       //signs
expx[14:0] = x[78:64]; expy[14:0] = y[78:64];     //exponents
sigx[63:0] <= x[63:0]; sigy[63:0] <= y[63:0];     //significands

//Sign of result:

sgnz <= sgnx ^ sgny;

//Biased exponent sum:

exp_sum[14:0] <= expx[14:0] + expy[14:0] + 15'h4001;

//Registers:

rc_C2[1:0] <= rc[1:0];
pc_C2 <= pc;
```

Figure 1.1 Module FMUL

```
//******************************************************************
// Second Cycle
//******************************************************************

//Rounding Constants//

//Overflow case -- single precision:

rconst_sing_of[127:0] =
  case(rc_C2[1:0])
    `NRE : {25'b1, 103'b0};
    `NEG  : sgnz ? {24'b0, {104 {1'b1}}} : 128'b0;
    `POS  : sgnz ? 128'b0 : {24'b0, {104 {1'b1}}};
    `CHP : 128'b0;
  endcase;

//Overflow case -- double precision:

rconst_doub_of[127:0] =
  case(rc_C2[1:0])
    `NRE : {54'b1, 74'b0};
    `NEG  : sgnz ? {53'b0, {75 {1'b1}}} : 128'b0;
    `POS  : sgnz ? 128'b0 : {53'b0, {75 {1'b1}}};
    `CHP : 128'b0;
  endcase;

//General overflow case:

rconst_of[127:0] <= case(pc_C2)
                      `SNG : rconst_sing_of[127:0];
                      `DBL : rconst_doub_of[127:0];
                    endcase;

//No overflow:

rconst_nof[126:0] = rconst_of[127:1];

//Registers:

sgnz_C3 <= sgnz;
exp_sum_C3[14:0] <= exp_sum[14:0];
sigx_C3[63:0] <= sigx[63:0];
sigy_C3[63:0] <= sigy[63:0];
rc_C3[1:0] <= rc_C2[1:0];
pc_C3 <= pc_C2;
```

Figure 1.2 `Module FMUL` (continued)

```
//*******************************************************************
// Third Cycle
//*******************************************************************

//The output of an integer multiplier actually consists of two vectors,
//the sum of which is the product of the inputs sigx and sigy.  These
//vectors become available in the third cycle, when they are processed
//in parallel by three distinct adders.  The first of these produces
//the unrounded product, which is used only to test for overflow.
//The other two include rounding constants, assuming overflow and no
//overflow, respectively.  Thus, at the (hypothetical) implementation
//level, these three sums are actually generated in parallel:

prod[127:0] = {64'b0, sigx_C3[63:0]} * {64'b0, sigy_C3[63:0]};

add_of[128:0] <= {1'b0, prod[127:0]} + {1'b0, rconst_of[127:0]};

add_nof[127:0] <= prod[127:0] + {1'b0, rconst_nof[126:0]};


//overflow indicator:

overflow <= prod[127];


//Sticky bit:

sticky_of <= case(pc_C3)
               `SNG : ~(prod[102:0] == 103'b0);
               `DBL : ~(prod[73:0] == 74'b0);
             endcase;

sticky_nof <= case(pc_C3)
                `SNG : ~(prod[101:0] == 102'b0);
                `DBL : ~(prod[72:0] == 73'b0);
              endcase;

//Registers:

rc_C4[1:0] <= rc_C3[1:0];
pc_C4 <= pc_C3;
sgnz_C4 <= sgnz_C3;
exp_sum_C4[14:0] <= exp_sum_C3[14:0];
```

Figure 1.3 Module FMUL (continued)

```
//*****************************************************************
// Fourth Cycle
//*****************************************************************

//Significand mask:

mask_of[127:0] =
  case (pc_C4)
    `SNG : (rc_C4[1:0] == `NRE) & ~sticky_of & ~add_of[103] ?
               {{23 {1'b1}}, 105'b0} : {{24 {1'b1}}, 104'b0};
    `DBL : (rc_C4[1:0] == `NRE) & ~sticky_of & ~add_of[74] ?
               {{52 {1'b1}}, 76'b0} : {{53 {1'b1}}, 75'b0};
  endcase;

mask_nof[126:0] =
  case (pc_C4)
    `SNG : (rc_C4[1:0] == `NRE) & ~sticky_nof & ~add_nof[102] ?
               {{23 {1'b1}}, 104'b0} : {{24 {1'b1}}, 103'b0};
    `DBL : (rc_C4[1:0] == `NRE) & ~sticky_nof & ~add_nof[73] ?
               {{52 {1'b1}}, 75'b0} : {{53 {1'b1}}, 74'b0};
  endcase;

//Carry bit:

carry_of = add_of[128];
carry_nof = add_nof[127];

//Significand and exponent:

sig_of[128:0] = {1'b0, carry_of, 127'b0} |
                (add_of[128:0] & {1'b0, mask_of[127:0]});
sig_nof[127:0] = {1'b0, carry_nof, 126'b0} |
                 (add_nof[127:0] & {1'b0, mask_nof[126:0]});
sigz[63:0] = overflow ? sig_of[127:64] : sig_nof[126:63];

exp_of[14:0] = exp_sum_C4[14:0] + {14'b0, carry_of} + 15'b1;
exp_nof[14:0] = exp_sum_C4[14:0] + {14'b0, carry_nof};
expz[14:0] = overflow ? exp_of[14:0] : exp_nof[14:0];

//Final result:

z[79:0] = {sgnz_C4, expz[14:0], sigz[63:0]};

endmodule
```

Figure 1.4 Module FMUL (continued)

Theorem 1 (`correctness-of-fmul`) *Assume that x and y are normal $\mathcal{E}$-encodings, $rc \in \{0, 1, 2, 3\}$, and $pc \in \{0, 1\}$. Let*

$$\mathcal{M} = \begin{cases} near, & \text{if } rc = 0 \\ minf, & \text{if } rc = 1 \\ inf, & \text{if } rc = 2 \\ trunc, & \text{if } rc = 3, \end{cases}$$

$$\mu = \begin{cases} 24 & \text{if } pc = 0 \\ 53 & \text{if } pc = 1, \end{cases}$$

and $\mathcal{A} = rnd(\hat{x}\hat{y}, \mathcal{M}, \mu)$. If $\mathcal{A}$ is representable, then z is a normal encoding and $\hat{z} = \mathcal{A}$.

The ACL2 formalization is straightforward:

```
(defun mode (rc)
  (case rc (0 'near) (1 'minf) (2 'inf) (3 'trunc)))
(defun precision (pc) (case pc (0 24) (1 53)))
(defthm correctness-of-fmul
    (let ((ideal (rnd (* (hat x) (hat y))
                      (mode rc)
                      (precision pc)))
          (z (fmul x y rc pc)))
      (implies (and (normal-encoding-p x (extfmt))
                    (normal-encoding-p y (extfmt))
                    (member rc (list 0 1 2 3))
                    (member pc (list 0 1))
                    (repp ideal (extfmt)))
               (and (normal-encoding-p z (extfmt))
                    (= (hat z) ideal))))
  :hints ...)
```

In the next subsection, we sketch an informal proof of Theorem 1, illustrating the application of the library of Section 1.. Once again, each lemma listed below includes the name of a corresponding ACL2 `defthm` event, which may be found in [Russinoff, 1999c]. Finally, in Subsection 3.2, we clarify the nature of this correspondence and describe the methodology that we have developed to derive the formal theorem `correctness-of-fmul` from the informal proof.

3.1 INFORMAL PROOF

For convenience, we introduce several auxiliary variables. First, we define

$$sticky = \begin{cases} sticky\_of & \text{if } overflow = 1 \\ sticky\_nof & \text{if } overflow = 0. \end{cases}$$

Each of the variables $rconst$, add, $carry$, $mask$, and sig is defined in the analogous manner. We also define

$$P = \begin{cases} 128 & \text{if } overflow = 1 \\ 127 & \text{if } overflow = 0, \end{cases}$$

$$\rho = rem(sig, 2^P),$$

and

$$\mathcal{M}' = \begin{cases} minf, & \text{if } \mathcal{M} = inf \text{ and } sgnz = 1 \\ inf, & \text{if } \mathcal{M} = minf \text{ and } sgnz = 1 \\ \mathcal{M}, & \text{otherwise.} \end{cases}$$

Our first four lemmas may be derived by case analysis as immediate consequences of these definitions:

Lemma 3..1 (`CARRY-REWRITE`) $carry = add[P]$.

Lemma 3..2 (`sig-rewrite`) $sig = (2^{P-1}carry) \mid (add \,\&\, mask)$.

Lemma 3..3 (`mask-rewrite`)

$$mask = \begin{cases} 2^P - 2^{P-\mu+1} & \textit{if } \mathcal{M} = near,\ sticky = add[P-\mu-1] = 0 \\ 2^P - 2^{P-\mu} & \textit{otherwise.} \end{cases}$$

Lemma 3..4 (`rconst-rewrite`)

$$rconst = \begin{cases} 2^{P-\mu-1} & \textit{if } \mathcal{M}' = near \\ 2^{P-\mu} - 1 & \textit{if } \mathcal{M}' = inf \\ 0 & \textit{otherwise.} \end{cases}$$

Lemma 3..5 (`expo-prod`) $expo(prod) = P - 1$.

Proof: Since x and y are normal encodings,

$$2^{126} \leq prod = sigx \cdot sigy < 2^{128},$$

and the lemma follows from Lemmas 1..2 and 1..3. □

Lemma 3..6 (`sig-prod`) *$sig(prod) = sig(\hat{x})sig(\hat{y})/2^{overflow}$.*

Proof: By Lemma 3..5,

$$\begin{aligned} prod &= 2^{63}sig(\hat{x})2^{63}sig(\hat{y}) \\ &= sig(\hat{x})sig(\hat{y})2^{-overflow}2^{126+overflow} \\ &= sig(\hat{x})sig(\hat{y})2^{-overflow}2^{expo(prod)}. \end{aligned}$$

The claim now follows from Lemma 1..15. □

Lemma 3..7 (`expo-xy`) *$expo(\hat{x}\hat{y}) = expo(\hat{x}) + expo(\hat{y}) + overflow$.*

Proof: By Lemmas 1..15 and 3..6,

$$\begin{aligned} \hat{x}\hat{y} &= sgn(\hat{x})sig(\hat{x})2^{expo(\hat{x})}sgn(\hat{y})sig(\hat{y})2^{expo(\hat{y})} \\ &= sgn(\hat{x}\hat{y})\left[sig(\hat{x})sig(\hat{y})/2^{overflow}\right]2^{expo(\hat{x})+expo(\hat{y})+overflow} \\ &= sgn(\hat{x}\hat{y})sig(prod)2^{expo(\hat{x})+expo(\hat{y})+overflow}. \end{aligned}$$

The result now follows from Lemma 1..18. □

Lemma 3..8 (`sig-xy`) *$sig(\hat{x}\hat{y}) = sig(prod)$.*

Proof: This is another consequence of the proof of Lemma 3..7. □

Lemma 3..9 (`sticky-exact`) *$sticky = 0$ iff $prod$ is $(\mu + 1)$-exact.*

Proof: It is clear that in all cases, $sticky = 0$ iff $2^{P-(\mu+1)}$ divides *prod*, and the lemma follows from Lemmas 1..20 and 3..5. □

Lemma 3..10 (`add-rewrite`) *$add = prod + rconst$.*

Proof: By Lemmas 3..4 and 3..5, $0 \leq prod + rconst < 2^P + 2^P = 2^{P+1}$, hence by the definition of *add*,

$$add = rem(prod + rconst, 2^{P+1}) = prod + rconst. \Box$$

Lemma 3..11 (`sig-bit`) *$sig[P-1] = 1$.*

Proof: By Lemmas 1..9 and 3..2, we may assume $carry = 0$ and hence by Lemmas 3..2, 3..3, 3..6, and 1..8,

$$\begin{aligned} sig[P-1] &= (add \mathbin{\&} mask)[P-1] = add[P-1] \mathbin{\&} mask[P-1] \\ &= add[P-1]. \end{aligned}$$

But then since

$$2^{P-1} \leq prod \leq prod + rconst = add < 2^{P+1}$$

and $carry = add[P] = 0$, Lemma 1..3 implies $add < 2^P$ and hence, by the same lemma, $add[P-1] = 1$. □

Lemma 3..12 (`sig-add-expo`) *$expo(sig) \leq expo(add) = P-1+carry$.*

Proof: If $carry = 0$, then

$$sig = add \mathbin{\&} mask \leq add < 2^P,$$

by Lemma 1..10, and Lemma 3..11 implies $sig \geq 2^{P-1}$, hence

$$expo(sig) = expo(add) = P - 1.$$

On the other hand, if $carry = add[P] = 1$, then $expo(add) = P$, while $sig < 2^{P+1}$ by Lemma 1..13, hence $expo(sig) \leq P$. □

Lemma 3..13 (`rem-sig`) *sig is divisible by 2^{P-64}.*

Proof: Since 2^{P-64} divides $mask$, the result follows from Lemmas 1..11 and 1..14. □

Lemma 3..14 (`sgnf-z`) *$sgnf(z, \mathcal{E}) = sgnz$.*

Proof: Note that

$$z = 2^{79} sgnz + 2^{64} expz + sigz,$$

where $0 \leq sgnz < 2$, $0 \leq expz < 2^{15}$, and $0 \leq sigz < 2^{64}$. Thus,

$$sgnf(z, \mathcal{E}) = z[79] = rem(\lfloor z/2^{79} \rfloor, 2) = rem(sgnz, 2) = sgnz. \square$$

Lemma 3..15 (`expf-z`) *$expf(z, \mathcal{E}) = expz$.*

Proof: As in the proof of Lemma 3..14,

$$\begin{aligned} expf(z, \mathcal{E}) &= z[78:64] = \lfloor rem(z, 2^{79})/2^{64} \rfloor = \lfloor expz + sigz/2^{64} \rfloor \\ &= expz. \square \end{aligned}$$

Lemma 3..16 (`sigf-z`) *$sigf(z, \mathcal{E}) = sigz$.*

Proof: As in the proof of Lemma 3..14,

$$sigf(z, \mathcal{E}) = z[63:0] = rem(z, 2^{64}) = sigz. \square$$

Lemma 3..17 (`z-normal`) *z is a normal encoding.*

Proof: It is clear that $z \in \mathbb{N}$ and

$$sigf(z, \mathcal{E}) = sigz = sig[P-1 : P-64].$$

Thus, by Lemmas 1..6 and 3..11, $sigz[63] = sig[P-1] = 1$, and hence $sigz \geq 2^{63}$. $\square$

Lemma 3..18 (`sgn-z`) $sgn(\hat{z}) = sgn(\hat{x}\hat{y})$.

Proof: By Lemma 3..14, $sgn(\hat{z}) = (-1)^{sgnz}$. Thus, $sgn(\hat{z}) = 1 \Leftrightarrow sgnz = 0 \Leftrightarrow sgnx = sgny \Leftrightarrow sgn(\hat{x}) = sgn(\hat{y}) \Leftrightarrow sgn(\hat{x}\hat{y}) = 1$. $\square$

Lemma 3..19 (`sig-z`) $sig(\hat{z}) = \rho/2^{P-1}$.

Proof: Since sig is divisible by 2^{P-64}, so is $\rho = rem(sig, 2^P)$. Thus,

$$sigz = sig[P-1 : P-64] = \lfloor \rho/2^{P-64} \rfloor = \rho/2^{P-64}$$

and $sig(\hat{z}) = sigz/2^{63} = \rho/2^{P-1}$. $\square$

Lemma 3..20 (`expo-z`) $expo(\hat{z}) = expo(\hat{x}\hat{y}) + carry + 2^{15}k$, *for some* $k \in \mathbb{Z}$.

Proof: We have

$$expx = expf(x, \mathcal{E}) = expo(\hat{x}) + 2^{14} - 1,$$

$$expy = expf(y, \mathcal{E}) = expo(\hat{y}) + 2^{14} - 1,$$

and by Lemma 3..7,

$$\begin{aligned} expz &= rem(exp\_sum + carry + overflow, 2^{15}) \\ &= rem(expx + expy + 2^{14} + 1 + carry + overflow, 2^{15}) \\ &= rem(expo(\hat{x}) + expo(\hat{y}) + overflow + 2^{14} - 1 + carry, 2^{15}) \\ &= rem(expo(\hat{x}\hat{y}) + 2^{14} - 1 + carry, 2^{15}). \end{aligned}$$

Hence, for some $k \in \mathbb{Z}$,

$$expf(z, \mathcal{E}) = expz = expo(\hat{x}\hat{y}) + 2^{14} - 1 + carry + 2^{15}k.$$

But then

$$expo(\hat{z}) = expf(z, \mathcal{E}) - (2^{14} - 1) = expo(\hat{x}\hat{y}) + carry + 2^{15}k. \square$$

Lemma 3..21 (`rho-rewrite`) $\rho = rnd(prod, \mathcal{M}', \mu)2^{-carry}$.

Proof: We consider the following cases:

Case 1: $carry = 0$

Since $sig < 2^P$ by Lemma 3..12, we must show

$$sig = rnd(prod, \mathcal{M}', \mu).$$

Subcase 1.1: $\mathcal{M}' = near$

First suppose $sticky = add[P-\mu-1] = 0$. Then Lemmas 1..4, 3..4, and 3..10 imply

$$prod[P-\mu-1] = 1,$$

and by Lemmas 3..9, 1..20, and 1..5, $prod$ is $(\mu+1)$-exact but not μ-exact. Thus, by Lemmas 1..24, 1..26, 3..2, 3..3, 3..5, 3..10, and 3..12,

$$\begin{aligned} sig &= (prod + 2^{P-\mu-1}) \,\&\, (2^P - 2^{P-\mu+1}) \\ &= trunc(prod + 2^{P-\mu-1}, \mu - 1) \\ &= near(prod, \mu) \\ &= rnd(prod, \mathcal{M}', \mu). \end{aligned}$$

In the remaining case, $prod$ is either μ-exact or not $(\mu+1)$-exact, and the same lemmas yield

$$\begin{aligned} sig &= (prod + 2^{P-\mu-1}) \,\&\, (2^P - 2^{P-\mu}) \\ &= trunc(prod + 2^{P-\mu-1}, \mu) \\ &= near(prod, \mu) \\ &= rnd(prod, \mathcal{M}', \mu). \end{aligned}$$

Subcase 1.2: $\mathcal{M}' = inf$

By Lemmas 1..24 and 1..25,

$$\begin{aligned} sig &= (prod + 2^{P-\mu} - 1) \,\&\, (2^P - 2^{P-\mu}) \\ &= trunc(prod + 2^{P-\mu} - 1, \mu) \\ &= away(prod, \mu) \\ &= rnd(prod, \mathcal{M}', \mu). \end{aligned}$$

Subcase 1.3: $\mathcal{M}' = trunc$ or $\mathcal{M}' = minf$

Lemma 1..24 yields

$$\begin{aligned} sig &= prod \,\&\, (2^P - 2^{P-\mu}) \\ &= trunc(prod, \mu) \\ &= rnd(prod, \mathcal{M}', \mu). \end{aligned}$$

Case 2: carry $= 1$

In this case, by Lemmas 3..1 and 3..12,

$$2^P \leq add = prod + rconst < 2^P + rconst,$$

which, with Lemma 3..4, implies

$$0 \leq rem(add, 2^P) < rconst < 2^{P-\mu}.$$

Applying Lemmas 1..14, 1..11, and 1..12, we have

$$\begin{aligned} rem(sig, 2^P) &= rem(2^{P-1} \mid (add \,\&\, mask), 2^P) \\ &= 2^{P-1} \mid (rem(add, 2^P) \,\&\, mask) \\ &= 2^{P-1} \mid (rem(add, 2^P) \,\&\, rem(mask, 2^{P-\mu})) \\ &= 2^{P-1} \mid (rem(add, 2^P) \,\&\, 0) \\ &= 2^{P-1}. \end{aligned}$$

Thus, it suffices to show that $rnd(prod, \mathcal{M}', \mu) = 2^P$.

Subcase 2.1: $\mathcal{M}' = near$

Since

$$prod + 2^{P-1-\mu} = prod + rconst \geq 2^P,$$

we must have $near(prod, \mu) = 2^P$.

Subcase 2.2: $\mathcal{M}' = inf$

Let $a = 2^P - 2^{P-\mu}$. Then

$$prod \geq 2^P - rconst = 2^P - 2^{P-\mu} + 1 > a,$$

and since a is μ-exact,

$$away(prod, \mu) \geq a + 2^{expo(a)+1-\mu} = a + 2^{P-\mu} = 2^P$$

by Lemma 1..21, and it follows that $away(prod, \mu) = 2^P$.

Subcase 2.3: $\mathcal{M}' = trunc$ or $\mathcal{M}' = minf$

This case is precluded by Lemma 3..4 and our earlier observation that $0 < rconst$. □

We may now complete the proof of Theorem 1. By Lemmas 3..5 and 3..8,

$$prod = sig(prod)2^{expo(prod)} = sig(\hat{x}\hat{y})2^{P-1},$$

and hence by Lemmas 3..19, 3..21 and 1..22,

$$\begin{aligned} sig(\hat{z}) &= \rho/2^{P-1} = rnd(prod, \mathcal{M}', \mu)/2^{carry+P-1} \\ &= rnd(sig(\hat{x}\hat{y}), \mathcal{M}', \mu)/2^{carry}. \end{aligned}$$

Now, applying Lemmas 3..20 and 1..22, we have

$$\begin{aligned} \hat{z} &= sgn(\hat{z})sig(\hat{z})2^{expo(\hat{z})} \\ &= sgn(\hat{z})rnd(sig(\hat{x}\hat{y}), \mathcal{M}', \mu)2^{expo(\hat{x}\hat{y})+2^{15}k} \\ &= sgn(\hat{z})rnd(sig(\hat{x}\hat{y})2^{expo(\hat{x}\hat{y})}, \mathcal{M}', \mu)2^{2^{15}k}, \end{aligned}$$

where $k \in \mathbb{Z}$. If $sgnz = 0$, then $\mathcal{M}' = \mathcal{M}$ and by Lemma 3..14, $sgn(\hat{z}) = 1$. On the other hand, if $sgnz = 1$, then $\mathcal{M}' = flip(\mathcal{M})$ and $sgn(\hat{z}) = -1$. In either case, by Lemmas 1..23 and 3..18,

$$\begin{aligned} \hat{z} &= rnd(sgn(\hat{z})sig(\hat{x}\hat{y})2^{expo(\hat{x}\hat{y})}, \mathcal{M}, \mu)2^{2^{15}k} \\ &= rnd(sgn(\hat{x}\hat{y})sig(\hat{x}\hat{y})2^{expo(\hat{x}\hat{y})}, \mathcal{M}, \mu)2^{2^{15}k} \\ &= rnd(\hat{x}\hat{y}, \mathcal{M}, \mu)2^{2^{15}k}. \end{aligned}$$

But since $rnd(\hat{x}\hat{y}, \mathcal{M}, \mu)$ is representable, i.e.,

$$1 - 2^{-14} \leq expo(rnd(\hat{x}\hat{y}, \mathcal{M}, \mu)) \leq 2^{14},$$

and the same is true of $\hat{z}$, Lemma 1..19 yields

$$|2^{15}k| = |expo(\hat{z}) - expo(rnd(\hat{x}\hat{y}, \mathcal{M}, \mu))| < 2^{15},$$

and hence $k = 0$. □

3.2 FORMAL PROOF

In the design of a formal computational model, the ACL2 user is often faced with conflicting criteria. For example, a model that is intended primarily for formal analysis may not provide the desired execution efficiency. It is a common practice to define two or more models to serve distinct purposes and then prove them to be equivalent. This is the approach that we take here.

The translation scheme described in Subsection 2.3 is conceptually simple and provides an accurate representation of the RTL model that may be executed fairly efficiently. This is an important consideration in many applications, as it allows the formal model to be validated against the RTL through testing. However, this model is not amenable to formal analysis—it would be awkward to attempt to use it directly to formalize the argument presented in Subsection 3.1. Every reference to a signal would necessarily mention all the signals on which it depends, and the derived properties of those signals would have to be listed repreatedly. For example, a formal statement of Lemma 3..6 based on the definition

```
(defun prod (sigx_c3 sigy_c3)
   (bits (* sigx_c3 sigy_c3) 127 0))
```

would have to include all relevant properties of `sigx_c3` and `sigy_c3` as explicit hypotheses.

For the purpose of verification, therefore, we shall use an alternative translation scheme, and establish a method for converting theorems pertaining to the resulting model to theorems about the original model. Using our multiplier as an illustration, we begin by defining two functions, representing the constraints on inputs and desired properties of outputs, respectively:

```
(defun input-spec (x y rc pc)
  (and (normal-encoding-p x (extfmt))
       (normal-encoding-p y (extfmt))
       (member rc (list 0 1 2 3))
       (member pc (list 0 1))
       (repp (rnd (* (hat x) (hat y))
                  (mode rc)
                  (precision pc))
             (extfmt))))
(defun output-spec (x y rc pc)
  (let ((z (fmul x y rc pc)))
    (and (normal-encoding-p z (extfmt))
         (= (hat z)
            (rnd (* (hat x) (hat y))
                 (mode rc)
                 (precision pc))))))
(in-theory (disable input-spec output-spec))
```

Next, we introduce constants corresponding to the inputs, constrained to satisfy the input specification:

```
(encapsulate ((x* () t) (y* () t) (rc* () t) (pc* () t))
  (local (defun x* () (encode 1 (extfmt))))
  (local (defun y* () (encode 1 (extfmt))))
  (local (defun rc* () 0))
  (local (defun pc* () 1))
  (local (in-theory (disable input-spec*)))
```

```
(defthm input-spec* (input-spec (x*) (y*) (rc*) (pc*))))
```

Constants are then defined corresponding to all remaining signals. In fact, for convenience, these functions are automatically generated by our translator and placed in a separate file. This file contains, for example,

```
(defun sgnx* nil (sgnx (x*)))
```

and

```
(defun z* nil (z (sgnz_c4*) (expz*) (sigz*))).
```

Formal versions of the lemmas appearing in Subsection 3.1, based on these constant functions, may now be proved in a natural way by faithfully following their informal proofs (see [Russinoff, 1999c]). Thus, we obtain the following theorem:

```
(defthm z*-spec
    (and (normal-encoding-p (z*) (extfmt))
         (= (hat (z*))
            (rnd (* (hat (x*)) (hat (y*)))
                 (mode (rc*))
                 (precision (pc*)))))
  :rule-classes())
```

Now, our goal is to derive the theorem `correctness-of-fmul` from `z*-spec`. First, we establish this relationship between the two models:

```
(defthm fmul-star-equivalence
        (equal (z*)
               (fmul (x*) (y*) (rc*) (pc*)))
        :rule-classes nil)
```

The last two theorems now yield the following:

```
(defthm output-spec*
    (output-spec (x*) (y*) (rc*) (pc*))
  :hints (("goal" :in-theory (enable output-spec)
                  :use (z*-spec fmul-star-equivalence))))
```

The next step is critical, employing functional instantiation:

```
(defthm fmul-input-output
    (implies (input-spec x y rc pc)
             (output-spec x y rc pc))
  :hints
  (("goal" :in-theory (enable input-spec*)
           :use ((:functional-instance output-spec*
```

```
                   (x* (lambda ()
                         (if (input-spec x y rc pc)
                             x (x*))))
                   (y* (lambda ()
                         (if (input-spec x y rc pc)
                             y (y*))))
                   (rc* (lambda ()
                           (if (input-spec x y rc pc)
                               rc (rc*))))
                   (pc* (lambda ()
                          (if (input-spec x y rc pc)
                              pc (pc*))))))))
  :rule-classes ())
```

The final theorem now follows easily:

```
(defthm correctness-of-fmul
    (let ((ideal (rnd (* (hat x) (hat y))
                      (mode rc)
                      (precision pc)))
          (z (fmul x y rc pc)))
      (implies (and (normal-encoding-p x (extfmt))
                    (normal-encoding-p y (extfmt))
                    (member rc (list 0 1 2 3))
                    (member pc (list 0 1))
                    (repp ideal (extfmt)))
               (and (normal-encoding-p z (extfmt))
                    (= (hat z) ideal))))
  :hints (("goal" :in-theory (enable input-spec output-spec)
                  :use (fmul-input-output))))
```

Exercise: The hypothetical implementation of our floating-point multiplier relies on the efficient computation of the sum of three bit vectors (see the comments in Fig. 1.3), using several logical operations (which are executed in constant time) and a single addition. In order to establish the correctness of this computation, prove that for all $x, y, z \in \mathbb{N}$,

$$x + y + z = x \text{ ^ } y \text{ ^ } z + 2\,[(x \;\&\; y) \;\mathtt{|}\; (x \;\&\; z) \;\mathtt{|}\; (y \;\&\; z)]\,.$$

Exercise: A rounding mode used in the AMD Athlon floating-point adder, called *sticky* rounding, is defined as follows, for $x \in \mathbb{Q}^*$ and $n \in \mathbb{N}^*$:

(a) $sticky(x, 1) = sgn(x)2^{expo(x)}$.

(b) If $n > 1$ and x is $(n-1)$-exact, then $sticky(x, n) = x$.
(c) If $n > 1$ and x is not $(n-1)$-exact, then

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

Derive the following properties of sticky rounding:
(1) Let $\mathcal{M}$ be an IEEE rounding mode, $\sigma \in \mathbb{N}^*$, $n \in \mathbb{N}$, and $x \in \mathbb{Q}^*$. If $n \geq \sigma + 2$, then

$$rnd(x, \mathcal{M}, \sigma) = rnd(sticky(x, n), \mathcal{M}, \sigma).$$

(2) Let $x, y \in \mathbb{Q}$ such that $y \neq 0$ and $x + y \neq 0$. Let $k, k', k'' \in \mathbb{Z}$ such that $k' = k + expo(x) - expo(y)$, and $k'' = k + expo(x + y) - expo(y)$.
(a) If $k > 0$, $k' > 0$, $k'' > 0$, and x is k'-exact, then

$$x + trunc(y, k) = \begin{cases} trunc(x + y, k'') & \text{if } sgn(x + y) = sgn(y) \\ away(x + y, k'') & \text{if } sgn(x + y) \neq sgn(y); \end{cases}$$

(b) If $k > 1$, $k' > 1$, $k'' > 1$, and x is $(k' - 1)$-exact, then

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

References

[Gordon, 1995] Gordon, M. (1995). The semantic challenge of Verilog HDL. In *Tenth Annual IEEE Symposiom on Logic in Computer Science*. IEEE Computer Society Press.

[IEEE, 1985] IEEE (1985). Standard for binary floating point arithmetic. IEEE Standard 754-1985.

[Russinoff, 1995] Russinoff, D. (1995). Specification and verification of gate-level vhdl models of synchronous and asynchronous circuits. In Börger, E., editor, *Specification and Validation Methods*. Oxford University Press.

[Russinoff, 1998] Russinoff, D. (1998). A Mechanically Checked Proof of IEEE Compliance of a Register-Transfer-Level Specification of the AMD-K7 Floating-Point Multiplication, Division, and Square Root Instructions. *London Mathematical Society Journal of Computation and Mathematics*, 1:148–200.

[Russinoff, 1999a] Russinoff, D. (1999a). A Mechanically Checked Proof of Correctness of the AMD-K5 Floating-Point Square Root Microcode. *Formal Methods in System Design*, 14:75–125.

[Russinoff, 1999b] Russinoff, D. (1999b). An acl2 library of floating-point arithmetic. `http://www.cs.utexas.edu/users/-moore/publications/others/fp-README.html`.

[Russinoff, 1999c] Russinoff, D. (1999c). Mechanical verification of register-transfer logic: A floating-point multiplier. `http://-www.cs.utexas.edu/users/moore/acl2-book-99/russinoff/-index.html`.