A Case Study in Formal Verification of Register-Transfer Logic with ACL2: The Floating Point Adder of the AMD Athlon^{TM} Processor

David M. Russinoff

Advanced Micro Devices, Inc.
Austin, TX

Abstract. As an alternative to commercial hardware description languages, AMD[1] has developed an RTL language for microprocessor designs that is simple enough to admit a clear semantic definition, providing a basis for formal verification. We describe a mechanical proof system for designs represented in this 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 IEEE compliance of the floating-point adder of the AMD Athlon processor.

1 Introduction

The formal hardware verification effort at AMD has emphasized theorem proving using ACL2 [3], and has focused on the elementary floating-point operations. One of the challenges of our earlier work was to construct accurate formal models of the targeted circuit designs. These included the division and square root operations of the AMD-K5 processor [4, 6], which were implemented in microcode, and the corresponding circuits of the AMD Athlon processor [7], which were initially modeled in C for the purpose of testing. In both cases, we were required to translate the designs by hand into the logic of ACL2, relying on an unrigorous understanding of the semantics of the source languages.

Ultimately, however, the entire design of the Athlon was specified and validated at the register-transfer level in a hardware description language that was developed specifically for that purpose. Essentially a small synthesizable subset of Verilog with an underlying cycle-based execution model, this language is simple enough to admit a clear semantic definition, providing a basis for formal analysis and verification. Thus, we have developed a scheme for automatically translating RTL code into the ACL2 logic [8], thereby eliminating an important possible source of error. Using this scheme, we have mechanically verified a number of operations of the Athlon floating-point unit at the register-transfer level,

[1] AMD, the AMD logo and combinations thereof, AMD-K5, and AMD Athlon are trademarks of Advanced Micro Devices, Inc.

including all addition, subtraction, multiplication, and comparison instructions. As an illustration of our methods, this paper describes the proof of correctness of the Athlon floating-point adder, a state-of-the-art adder with leading one prediction logic [5].

Much of the effort involved in the projects mentioned above was in the development and formalization of a general theory of floating-point arithmetic and its bit-level implementation. The resulting ACL2 library [9] is available as a part of ACL2 Version 2.6. Many of the included lemmas are documented in [6] and [7], and some of the details of the formalization are described in [8]. In Sections 2 and 3 below, we present several extensions of the library that were required for the present project.

In Sections 4 and 5, we demonstrate the utility of our floating-point theory, applying it in a rigorous derivation of the correctness of the adder. The theorem reported here is a formulation of the main requirement for IEEE compliance, as stipulated in Standard 754-1985 [1]:

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

In fact, we have also verified that the adder design conforms to other aspects of the behavior prescribed by [1] pertaining to overflow, underflow, and other exceptional conditions, as well as the refinements necessary for Pentium compatibility, as defined in [2]. However, since our main purpose here is to describe a verification methodology rather than to present the details of a specific proof, these secondary results have been omitted from this report.

All of our theorems have been formally encoded as propositions in the logic of ACL2, based on the ACL2 translation of the RTL code, and their proofs have all been mechanically checked with the ACL2 prover. For this purpose, we have developed a proof methodology based on some features of ACL2. In Section 5, we describe both the translation scheme and our proof methodology, using the adder as an illustration.

The use of mechanical theorem proving in the validation of hardware designs is still uncommon in the computer industry, mainly because of the effort that it entails. The work reported here, including the development of the translator and other reusable machinery, consumed some twenty weeks of the author's time, which was divided approximately equally between deriving the informal proofs and checking them mechanically. However, as has been noted before, the cost of formal methods is far outweighed by its potential benefits. In this case, our analysis of the adder exposed a logical error that would have, under certain conditions, resulted in reversing the sign of the sum of zero and an arbitrary nonzero number. This flaw had already survived extensive testing and was unlikely to be detected by conventional validation methods. It was easily repaired in the RTL, but could have been very expensive if not discovered until later.

2 Bit Vectors and Logical Operations

Bit vectors are the fundamental data type of our RTL language as well as the basis of our theory of floating-point arithmetic. We identify the bit vectors of length n with the natural numbers in the range $0 \leq x < 2^n$. Accordingly, we define the k^{th} *bit* of x to be

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

while the *slice* of x from the i^{th} bit down through the j^{th} is given by

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

The standard binary logical operations, x `&` y, x `|` y, and x `^` y, are defined recursively, e.g.,

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

If x is a bit-vector of length n, then its *complement* with respect to n is

$$comp1(x, n) = 2^n - x - 1.$$

Following conventional notation, we shall use the abbreviation

$$\tilde{}x[i:j] = comp1(x[i:j], i - j + 1).$$

The properties of these functions are collected in the ACL2 floating-point library [9]. Many of the basic lemmas are documented in [7] and [8]. Here we shall present several of the more specialized library lemmas that pertain to floating-point addition, in order to illustrate our methods of proof, especially the use of mathematical induction.

The design of the adder involves several computational techniques that are motivated by the observation that while the time required for integer addition increases logarithmically with the inputs, the logical operations defined in Section 2 may be executed in constant time. Thus, for example, the following result, which is readily proved by induction based on the recursive definitions of the logical operations, provides an efficient method for adding three vectors using a two-input adder:

Lemma 1. *For all* $x, y, z \in \mathbb{N}$,

$$x + y + z = (x \,\hat{}\, y \,\hat{}\, z) + 2((x \,\&\, y) \mid (x \,\&\, z) | (y \,\&\, z)).$$

A more interesting optimization, known as *leading one prediction*, allows the result of a subtraction to be normalized efficiently (in the event of cancellation) by performing the required left shift in advance of the subtraction itself. This requires a prediction of the highest index at which a 1 occurs in the difference.

Although the precise computation of this index appears generally to be as complex as the subtraction itself, a useful approximate solution may be obtained more quickly.

For any $x \in \mathbb{Z}^*$, $expo(x)$ will denote the index of the leading one of x, i.e., the greatest integer satisfying $2^{expo(x)} \leq |x|$. Let a and b be integers with $0 < b < a$ and $expo(a) = e$. We shall compute, in constant time (independent of a and b), a positive integer λ such that $expo(a - b)$ is either $expo(\lambda)$ or $expo(\lambda) - 1$. We begin by defining a function that returns the desired exponent $\phi = expo(\lambda)$. If $expo(b) < e - 1$, then $a/2 < a - b \leq a$ and we have the trivial solution $\phi = e$. In the remaining case, $e - 1 \leq expo(b) \leq e$, ϕ may be computed as follows: First, let m be the largest index such that $a[m] > b[m]$, i.e., $a[m] = 1$ and $b[m] = 0$. If $a[m : 0] = 2^m$ and $b[m : 0] = 2^m - 1$, then $\phi = 0$. Otherwise, ϕ is the largest index such that $\phi \leq m$ and $a[\phi - 1] \geq b[\phi - 1]$.

The correctness of this computation is established by the following lemma, in which ϕ is represented as a recursive function:

Lemma 2. *Let $a, b, n \in \mathbb{N}$. For all $d \in \mathbb{Z}$ and $k \in \mathbb{N}$, let $c_k = a[k] - b[k]$ and*

$$\phi(a, b, d, k) = \begin{cases} 0 & \text{if } k = 0 \\ \phi(a, b, c_{k-1}, k - 1) & \text{if } k > 0 \text{ and } d = 0 \\ \phi(a, b, d, k - 1) & \text{if } k > 0 \text{ and } d \neq 0 \text{ and } d = -c_{k-1} \\ k & \text{if } k > 0 \text{ and } d \neq 0 \text{ and } d \neq -c_{k-1}. \end{cases}$$

If $a < 2^n$, $b < 2^n$, and $a \neq b$, then $\phi(a, b, 0, n) - 1 \leq expo(a - b) \leq \phi(a, b, 0, n)$.

Proof: It is easy to show, by induction on k, that $\phi(a, b, d, k) = \phi(b, a, -d, k)$. Therefore, we may assume that $a > b$. Note also that if $a[k-1 : 0] = a'[k-1 : 0]$ and $b[k-1 : 0] = b'[k-1 : 0]$, then $\phi(a, b, d, k) = \phi(a', b', d, k)$.

In the case $n = 1$, we have $a = 1$, $b = 0$, and

$$expo(a - b) = 0 = \phi(a, b, 1, 0) = \phi(a, b, 0, 1).$$

We proceed by induction, assuming $n > 1$.

Suppose first that $c_{n-1} = 0$. Let $a' = a[n - 2 : 0]$ and $b' = b[n - 2 : 0]$. Then by inductive hypothesis,

$$\phi(a', b', 0, n - 1) - 1 \leq expo(a' - b') \leq \phi(a', b', 0, n - 1).$$

But $a - b = a' - b'$, hence $expo(a - b) = expo(a' - b')$, and

$$\phi(a, b, 0, n) = \phi(a, b, c_{n-1}, n - 1) = \phi(a, b, 0, n - 1) = \phi(a', b', 0, n - 1).$$

Now suppose that $c_{n-1} = 1$ and $c_{n-2} = -1$. Then $a[n - 1] = b[n - 2] = 1$ and $a[n - 2] = b[n - 1] = 0$. It follows that

$$2^{n-1} + 2^{n-2} > a \geq 2^{n-1} > b \geq 2^{n-2}.$$

Let $a' = a - 2^{n-2}$ and $b' = b - 2^{n-2}$. Then

$$2^{n-1} > a' \geq 2^{n-2} > b' \geq 0.$$

Once again,

$$\phi(a', b', 0, n-1) - 1 \leq expo(a' - b') \leq \phi(a', b', 0, n-1)$$

and $expo(a - b) = expo(a' - b')$. But

$$\begin{aligned}\phi(a, b, 0, n) &= \phi(a, b, 1, n-1) = \phi(a, b, 1, n-2) = \phi(a', b', 1, n-2) \\ &= \phi(a', b', 0, n-1).\end{aligned}$$

In the remaining case, $c_{n-1} = 1$ and $c_{n-2} \geq 0$. Now

$$2^n > a \geq a - b \geq 2^{n-1} - b[n-3:0] > 2^{n-1} - 2^{n-2} = 2^{n-2},$$

hence $n - 2 \leq expo(a - b) \leq n - 1$, while

$$\phi(a, b, 0, n) = \phi(a, b, 1, n-1) = n - 1. \ \square$$

Thus, we require a general method for computing a number λ such that $expo(\lambda) = \phi(a, b, 0, e+1)$. First, we handle the relatively simple case $expo(b) < expo(a)$:

Lemma 3. *Let $a, b \in \mathbb{N}^*$ with $expo(b) < expo(a) = e$, and let*

$$\lambda = 2a[e-1:0] \mid \tilde{}(2b)[e:0].$$

Then $\lambda > 0$ and $expo(\lambda) - 1 \leq expo(a - b) \leq expo(\lambda)$.

Proof: Since

$$\phi(a, b, 0, e+1) = \phi(a, b, 1, e) = \phi(a[e-1:0], b, 1, e),$$

it will suffice to show, according to Lemma 2, that

$$\phi(a[e-1:0], b, 1, e) = expo(\lambda).$$

Using induction, we shall prove the following more general result: For all $a, b, k \in \mathbb{N}$, if $a < 2^k$ and $b < 2^k$, then

$$\phi(a, b, 1, k) = expo(2a \mid \tilde{}(2b)[k:0]).$$

If $k = 0$, then $a = b = 0$ and

$$expo(2a \mid \tilde{}(2b)[k:0]) = expo(0 \mid 1) = 0 = \phi(a, b, 1, k).$$

Suppose $k > 0$. If $a[k-1] = 0$ and $b[k-1] = 1$, then

$$\begin{aligned}\phi(a, b, 1, k) &= \phi(a, b, 1, k-1) \\ &= \phi(a, b[k-2:0], 1, k-1) \\ &= \phi(a, b - 2^{k-1}, 1, k-1) \\ &= expo(2a \mid \tilde{}(2b - 2^k)[k-1:0]) \\ &= expo(2a \mid \tilde{}(2b)[k:0]).\end{aligned}$$

In the remaining case, $\phi(a,b,1,k) = k$ and either $a[k-1] = 1$ or $b[k-1] = 0$. Since

$$expo(2a \mid \tilde{}(2b)[k:0]) = max(expo(2a), expo(\tilde{}(2b)[k:0])) \leq k,$$

we need only show $max(expo(2a), expo(\tilde{}(2b)[k:0])) \geq k$. But if $a[k-1] = 1$, then $2a \geq 2 \cdot 2^{k-1} = 2^k$, and if $b[k-1] = 0$, then $b < 2^{k-1}$, which implies $\tilde{}(2b)[k:0] = 2^{k+1} - 2b - 1 > 2^k - 1$. □

The next lemma covers the more complicated case $expo(b) = expo(a)$:

Lemma 4. *Let $a, b \in \mathbb{N}^*$ such that $a \neq b$ and $expo(a) = expo(b) = e > 1$. Let*

$$\begin{aligned}
\lambda_t &= a \hat{} \tilde{}b[e:0], \\
\lambda_g &= a \,\&\, \tilde{}b[e:0], \\
\lambda_z &= \tilde{}(a \mid \tilde{}b[e:0])[e:0],
\end{aligned}$$

$$\begin{aligned}
\lambda_0 = {} & (\lambda_t[e:2] \,\&\, \lambda_g[e-1:1] \,\&\, \tilde{}\lambda_z[e-2:0]) \mid \\
& (\tilde{}\lambda_t[e:2] \,\&\, \lambda_z[e-1:1] \,\&\, \tilde{}\lambda_z[e-2:0]) \mid \\
& (\lambda_t[e:2] \,\&\, \lambda_z[e-1:1] \,\&\, \tilde{}\lambda_g[e-2:0]) \mid \\
& (\tilde{}\lambda_t[e:2] \,\&\, \lambda_g[e-1:1] \,\&\, \tilde{}\lambda_g[e-2:0]),
\end{aligned}$$

and

$$\lambda = 2\lambda_0 + 1 - \lambda_t[0].$$

Then $\lambda > 0$ and $expo(\lambda) - 1 \leq expo(a-b) \leq expo(\lambda)$.

Proof: Let c_k and ϕ be defined as in Lemma 2. Since $c_e = 0$,

$$\phi(a,b,0,e+1) = \phi(a,b,0,e) = \phi(a,b,c_{e-1},e-1),$$

and therefore it will suffice to show that $\lambda \neq 0$ and $\phi(a,b,c_{e-1},e-1) = expo(\lambda)$. In fact, we shall derive the following more general result: For all $n \in \mathbb{N}$, if $n \leq e-1$ and $a[n:0] \neq b[n:0]$, then $\lambda[n:0] \neq 0$ and

$$expo(\lambda[n:0]) = \begin{cases} \phi(a,b,c_n,n) & \text{if } c_n = 0 \text{ or } c_{n+1} = 0 \\ \phi(a,b,-c_n,n) & \text{otherwise.} \end{cases}$$

For the case $n = 0$, note that $a[0] \neq b[0]$ implies $\lambda[0:0] = 1$, hence $expo(\lambda[0:0]) = 0$, while $\phi(a,b,c_0,0) = \phi(a,b,-c_0,0) = 0$.

We proceed by induction. Let $0 < n \leq e-1$. Note that for $0 \leq k \leq e-2$,

$$\begin{aligned}
\lambda_0[k] = 1 \Leftrightarrow {} & \lambda_t[k+2] = 1 \text{ and } \lambda_g[k+1] = 1 \text{ and } \lambda_z[k] = 0, \text{ or} \\
& \lambda_t[k+2] = 0 \text{ and } \lambda_z[k+1] = 1 \text{ and } \lambda_z[k] = 0, \text{ or} \\
& \lambda_t[k+2] = 1 \text{ and } \lambda_z[k+1] = 1 \text{ and } \lambda_g[k] = 0, \text{ or} \\
& \lambda_t[k+2] = 0 \text{ and } \lambda_g[k+1] = 1 \text{ and } \lambda_g[k] = 0.
\end{aligned}$$

For $0 \le k \le e$,

$$\lambda_t[k] = 1 \Leftrightarrow c_k = 0, \quad \lambda_g[k] = 1 \Leftrightarrow c_k = 1, \quad \text{and} \quad \lambda_z[k] = 1 \Leftrightarrow c_k = -1.$$

It follows that for $0 \le k \le e-2$,

$$\begin{aligned} \lambda_0[k] = 1 \Leftrightarrow\ & c_{k+1} \neq 0, \text{ and} \\ & \text{if } c_{k+2} = 0 \text{ then } c_k \neq -c_{k+1}, \text{ and} \\ & \text{if } c_{k+2} \neq 0 \text{ then } c_k \neq c_{k+1}. \end{aligned}$$

But since $n > 0$, $\lambda[n] = \lambda_0[n-1]$, and since $n \le e-1$,

$$\begin{aligned} \lambda[n] = 1 \Leftrightarrow\ & c_n \neq 0, \text{ and} \\ & \text{if } c_{n+1} = 0 \text{ then } c_{n-1} \neq -c_n, \text{ and} \\ & \text{if } c_{n+1} \neq 0 \text{ then } c_{n-1} \neq c_n. \end{aligned}$$

If $c_n = 0$, then $\lambda[n] = 0$, hence $\lambda[n:0] = \lambda[n-1:0] \neq 0$ and

$$expo(\lambda[n:0]) = expo(\lambda[n-1:0]) = \phi(a, b, c_{n-1}, n-1) = \phi(a, b, c_n, n).$$

Next, suppose $c_n \neq 0$ and $c_{n+1} = 0$. If $c_{n-1} = -c_n$, then $\lambda[n] = 0$, hence $\lambda[n:0] = \lambda[n-1:0] \neq 0$ and

$$\begin{aligned} expo(\lambda[n:0]) &= expo(\lambda[n-1:0]) = \phi(a, b, -c_{n-1}, n-1) = \phi(a, b, -c_{n-1}, n) \\ &= \phi(a, b, c_n, n). \end{aligned}$$

But if $c_{n-1} \neq -c_n$, then $\lambda[n] = 1$ and

$$expo(\lambda[n:0]) = n = \phi(a, b, c_n, n).$$

Finally, suppose $c_n \neq 0$ and $c_{n+1} \neq 0$. If $c_{n-1} = c_n$, then $\lambda[n] = 0$, $\lambda[n:0] = \lambda[n-1:0] \neq 0$, and

$$\begin{aligned} expo(\lambda[n:0]) &= expo(\lambda[n-1:0]) = \phi(a, b, -c_{n-1}, n-1) = \phi(a, b, -c_{n-1}, n) \\ &= \phi(a, b, -c_n, n). \end{aligned}$$

But if $c_{n-1} \neq c_n$, then $\lambda[n] = 1$ and

$$expo(\lambda[n:0]) = n = \phi(a, b, -c_n, n). \ \square$$

Finally, for the purpose of efficient rounding, it will also be useful to predict the *trailing one* of a sum or difference, i.e., the least index at which a 1 occurs. The following lemma provides a method for computing, in constant time, an integer τ that has precisely the same trailing one as the sum or difference of two given operands. As usual, subtraction is implemented through addition, by incrementing the sum of one operand and the complement of the other. Thus, the two cases $c = 0$ and $c = 1$ of the lemma correspond to addition and subtraction, respectively. We omit the proof, which is similar to that of Lemma 4.

Lemma 5. *Let $a, b, c, n, k \in \mathbb{N}$ with $a < 2^n$, $b < 2^n$, $k < n$, and $c < 2$. 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

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

3 Floating Point Numbers and Rounding

Floating point representation of rational numbers 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).

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 is a bit vector of length $\sigma + \epsilon + 1$, then the *sign*, *exponent*, and *significand fields* of z with respect to ϕ are $s = z[\sigma + \epsilon]$, $e = z[\sigma + \epsilon - 1 : \sigma]$, and $m = z[\sigma - 1 : 0]$, respectively. The rational number represented by z is given by

$$decode(z, \phi) = (-1)^s \cdot m \cdot 2^{e - 2^{\epsilon-1} - \sigma + 2}.$$

If $z[\sigma - 1] = 1$, then z is a *normal ϕ-encoding*. In this case, if $x = decode(z, \phi)$, then $sgn(x) = (-1)^s$, $sig(x) = 2^{\sigma-1} m$, and $expo(x) = e - (2^{\epsilon-1} - 1)$. 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 x is *n-exact* iff $sig(x) 2^{n-1} \in \mathbb{Z}$. It is easily shown that x is *representable with respect to ϕ*, i.e., there exists $z \in \mathbb{N}$ such that $x = decode(z, \phi)$, iff x is σ-exact and $-2^{\epsilon-1} + 1 \leq expo(x) \leq 2^{\epsilon-1}$.

The AMD Athlon floating-point unit supports four formats, corresponding to *single*, *double*, and *extended* precision as specified by IEEE, and a larger *internal* format:

$$(24, 7),\ (53, 10),\ (64, 15),\ \text{and}\ (68, 18).$$

In our discussion of the adder, floating point numbers will always be represented in the internal (68,18) format. If z is a bit vector of length 87, then we shall abbreviate $decode(z,(68,18))$ as $\hat{z}$.

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 that is closest to x and satisfies $|y| \leq |x|$. 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 the three other modes discussed in [6] 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.

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

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

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

As showed in [7], a number can be rounded according to any IEEE rounding mode by adding an appropriate constant and truncating the sum. In particular, if x is a positive integer with $expo(x) = e$, then the *rounding constant* for x corresponding to a given mode $\mathcal{M}$ and degree of precision σ is

$$\mathcal{C}(e,\mathcal{M},\sigma) = \begin{cases} 2^{e-\sigma} & \text{if } \mathcal{M} = near \\ 2^{e-\sigma+1} - 1 & \text{if } \mathcal{M} = inf \\ 0 & \text{if } \mathcal{M} = trunc \text{ or } \mathcal{M} = minf. \end{cases}$$

Lemma 6. *Let $\mathcal{M}$ be an IEEE rounding mode, $\sigma \in \mathbb{Z}$, $\sigma > 1$, and $x \in \mathbb{N}^*$ with $expo(x) \geq \sigma$. Then*

$$rnd(x,\mathcal{M},\sigma) = trunc(x + \mathcal{C}(expo(x),\mathcal{M},\sigma),\nu),$$

where

$$\nu = \begin{cases} \sigma - 1 & \textit{if } \mathcal{M} = near \textit{ and } x \textit{ is } (\sigma+1)\textit{-exact but not } \sigma\textit{-exact} \\ \sigma & \textit{otherwise.} \end{cases}$$

An additional rounding mode is critical to the implementation of floating-point addition: If $x \in \mathbb{Q}$, $n \in \mathbb{N}$, and $n > 1$, then

$$sticky(x,n) = \begin{cases} x & \text{if } x \text{ is } (n-1)\text{-exact} \\ trunc(x,n-1) + sgn(x)2^{expo(x)+1-n} & \text{otherwise.} \end{cases}$$

The significance of this operation is that the result of rounding a number x to σ bits, according to any IEEE rounding mode, can always be recovered from $sticky(x,\sigma+2)$:

Lemma 7. *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*

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

Proof: We may assume that $x > 0$ and x is not $(n-1)$-exact; the other cases follow trivially. First, note that since $sticky(x, n)$ is n-exact but not $(n-1)$-exact,

$$\begin{aligned} trunc(sticky(x, n), n-1) &= sticky(x, n) - 2^{expo(sticky(x,n))-(n-1)} \\ &= sticky(x, n) - 2^{expo(x)+1-n} \\ &= trunc(x, n-1). \end{aligned}$$

Thus, for any $m < n$,

$$trunc(sticky(x, n), m) = trunc(trunc(x, n-1), m) = trunc(x, m),$$

and the corresponding result for *away* may be similarly derived.

This disposes of all but the case $\mathcal{M} = near$. For this last case, it suffices to show that if $trunc(x, \sigma+1) = trunc(y, \sigma+1)$ and $away(x, \sigma+1) = away(y, \sigma+1)$, then $near(x, \sigma) = near(y, \sigma)$. We may assume $x \leq y$. Suppose $near(x, \sigma) \neq near(y, \sigma)$. Then for some $(\sigma+1)$-exact a, $x \leq a \leq y$. But this implies $x = a$, for otherwise $trunc(x, \sigma+1) \leq x < a \leq trunc(y, \sigma+1)$. Similarly, $y = a$, for otherwise $away(x, \sigma+1) \leq a < y \leq away(y, \sigma+1)$. Thus, $x = y$, a contradiction. □

The following property is essential for computing a rounded sum or difference:

Lemma 8. *Let $x, y \in \mathbb{Q}$ such that $y \neq 0$ and $x + y \neq 0$. Let $k \in \mathbb{Z}$, $k' = k + expo(x) - expo(y)$, and $k'' = k + expo(x+y) - expo(y)$. If $k > 1$, $k' > 1$, $k'' > 1$, and x is $(k'-1)$-exact, then*

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

Proof: Since x is $(k'-1)$-exact, $2^{k-2-expo(y)}x = 2^{(k'-1)-1-expo(x)}x \in \mathbb{Z}$. Thus,

$$\begin{aligned} y \text{ is } (k-1)\text{-exact} &\Leftrightarrow 2^{k-2-expo(y)}y \in \mathbb{Z} \\ &\Leftrightarrow 2^{k-2-expo(y)}y + 2^{k-2-expo(y)}x \in \mathbb{Z} \\ &\Leftrightarrow 2^{k''-2-expo(x+y)}(x+y) \in \mathbb{Z} \\ &\Leftrightarrow x + y \text{ is } (k''-1)\text{-exact}. \end{aligned}$$

If y is $(k-1)$-exact, then

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

Thus, we may assume that y is not $(k-1)$-exact. Now in [6] it was proved, with k, k', and k'' as defined above, and under the weaker assumptions that $k > 0$, $k' > 0$, $k'' > 0$, and x is k'-exact, that

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

Hence, if $sgn(x+y) = sgn(y)$, then

$$\begin{aligned} x + sticky(y,k) &= x + trunc(y,k-1) + sgn(y)2^{expo(y)+1-k} \\ &= trunc(x+y,k''-1) + sgn(x+y)2^{expo(x+y)+1-k''} \\ &= sticky(x+y,k''). \end{aligned}$$

On the other hand, if $sgn(x+y) \neq sgn(y)$, then

$$\begin{aligned} x + sticky(y,k) &= x + trunc(y,k-1) + sgn(y)2^{expo(y)+1-k} \\ &= away(x+y,k''-1) - sgn(x+y)2^{expo(x+y)+1-k''} \\ &= trunc(x+y,k''-1) + sgn(x+y)2^{expo(x+y)+1-k''} \\ &= sticky(x+y,k''). \ \square \end{aligned}$$

4 Description of the Adder

A simplified version of the Athlon floating-point adder is represented in the AMD RTL language as the circuit description $\mathcal{A}$, displayed in Figs. 1–6. As defined precisely in [8], a program in this language consists mainly of *input declarations*, *combinational assignments*, and *sequential assignments*, which have the forms

$$\texttt{input}\ \ s[k:0]; \tag{1}$$

$$s[k:0]\ \ \texttt{=}\ \ E; \tag{2}$$

and

$$s[k:0]\ \ \texttt{<=}\ \ E; \tag{3}$$

respectively, where $k \in \mathbb{N}$, s is a *signal* representing a bit vector of length $k+1$, and E is an expression constructed from signals and standard logical connectives. Each signal s occurring anywhere in a description must appear in exactly one of the three contexts (1), (2), and (3), and is called an *input*, a *wire*, or a *register*, accordingly. Any signal may also occur in an *output declaration*,

$$\texttt{output}\ \ s[k:0]; \tag{4}$$

and is then also called an *output*. Note that the circuit $\mathcal{A}$ has five inputs, `a`, `b`, `op`, `rc`, `pc` (Fig. 1), and one output, `r` (Fig. 2), which happens to be a wire (Fig. 6).

A circuit description may also contain *constant definitions* of the form

$$\texttt{`define}\ r \quad C$$

where r is an identifier and C is either a numerical constant or a pattern representing a set of constants. generalized constant expression. For example, according to the definition of `FSUB0` (Fig. 1), the value computed for the assignment

statement to `sub` (Fig. 2) is 1 whenever the value of the 11-bit vector `op` matches the string `1100000x10x`.

Any signal that occurs in the defining expresssion E for a (non-input) signal s is called a *direct supporter* of s. If s is a wire and s' is any signal, then s *depends* on s' iff s' is a direct supporter either of s or of some wire on which s depends. It is a syntactic requirement of the language that no wire depends on itself.

A *combinational* circuit is one that is free of registers. The semantics of a combinational circuit are particularly simple: the behavior of each output may be described as a function of the inputs. In fact, the same is true of a more general class of circuits, which we define as follows: A circuit description is an *n-cycle simple pipeline* if each of its signals s may be assigned a *cycle number* $\psi(s) \in \{1, \ldots, n\}$ such that

(1) if s is an input, then $\psi(s) = 1$;
(2) if s is a wire, then $\psi(s') = \psi(s)$ for each direct supporter s' of s;
(3) if s is a register, then $\psi(s') = \psi(s) - 1$ for each direct supporter s' of s;
(4) if s is an output, then $\psi(s) = n$.

In [8], we present a general semantic definition of the RTL language. associating a function with each signal. This function returns a sequence of values, interpreted as the values assumed by the signal on successive cycles, for a given set of sequences of values of the input signals. It is shown that for a simple pipeline, the value of each signal s on cycle $\psi(s)$ is determined by the values of the inputs on cycle 1. Moreover, this functional dependence on inputs is the same as for the combinational circuit that results from collapsing the pipeline by replacing each sequential assignment (3) by the corresponding combinational assignment (2)

The actual RTL model of the AMD Athlon floating-point adder is a 4-cycle simple pipeline. In order to simplify our analysis of the circuit as well as this presentation, the circuit description $\mathcal{A}$ listed below was derived by modifying the original as follows:

(1) All sequential assignments have been replaced with combinational assignments, yielding an equivalent combinational circuit.
(2) All code pertaining to functions other than addition and subtraction, which are performed by the same hardware, has been deleted.
(3) All code pertaining to the reporting of exceptional conditions, including overflow and underflow, has been deleted.
(4) The remaining code has been simplified by eliminating signals and combining assignments when possible.
(5) Signal names have been changed to promote readability.

The resulting circuit $\mathcal{A}$ is shorter and simpler than the original, and bears less resemblance to the intended gate-level implementation, but the two may easily be shown to be semantically equivalent with respect to the computation of sums and differences. In fact, this equivalence has been established mechanically as discussed in Section 6.

```
module A;

//*****************************************************************************
// Definitions
//*****************************************************************************

// CLASS DEFINITIONS
`define UNSUPPORTED       3'b000
`define SNAN              3'b001
`define NORMAL            3'b010
`define INFINITY          3'b011
`define ZERO              3'b100
`define QNAN              3'b101
`define DENORM            3'b110
`define MMX               3'b111

//OPCODE DEFINITIONS//
`define FADD 11'b1100xx0x000
`define FADDU 11'b11010000000
`define FSUB0 11'b1100000x10x
`define FSUB1 11'b1100100x10x
`define FSUB2 11'b1100110x10x
`define FSUBU 11'b11010000100
`define FADDT64 11'b11010010001
`define FSUBT64 11'b11010010000
`define FADDT68 11'b11010010010
`define FSUBT68 11'b11010010110

//PRECISION DEFINITIONS//
`define PC_32 2'b00 // single
`define PC_64 2'b10 // double
`define PC_80 2'b11 // extended
`define PC_80R 2'b01 // extended (reserved)

//ROUNDING DEFINITIONS//
`define RC_RN 2'b00  // round to nearest
`define RC_RM 2'b01 // round to minus infinity
`define RC_RP 2'b10 // round to plus infinity
`define RC_RZ 2'b11 // truncate

//*****************************************************************************
// Parameters
//*****************************************************************************

//INPUTS//
input a[89:0];        //first operand
input b[89:0];        //second operand
input op[10:0];       //opcode
input rc[1:0];        //rounding control
input pc[1:0];        //precision control
```

Fig. 1. Circuit $\mathcal{A}$

```
//OUTPUT//
output r[89:0];           //sum or difference

//OPERAND FIELDS//
mana[67:0] = a[67:0]; manb[67:0] = b[67:0];     //significand
expa[17:0] = a[85:68]; expb[17:0] = b[85:68];   //exponent
signa = a[86]; signb = b[86];                   //sign
classa[2:0] = a[89:87]; classb[2:0] = b[89:87];      //class
azero = (classa[2:0] == `ZERO); bzero = (classb[2:0] == `ZERO);

//OPERATION//
int_op = (op == `FADDT68) | (op == `FSUBT68);
ext_op = (op == `FADDT64) | (op == `FSUBT64);
sub = casex(op[10:0])
        `FSUB0,`FSUB1,`FSUB2,`FSUBU,`FSUBT68,`FSUBT64 : 1'b1;
        default : 1'b0;
      endcase;
esub = sub ^ signa ^ signb;    //effective subtraction

//ROUNDING CONTROL//
rc_near = (rc[1:0] == `RC_RN) | int_op;      // round to nearest
rc_minf = (rc[1:0] == `RC_RM) & ~int_op;     // round to minus infinity
rc_inf = (rc[1:0] == `RC_RP) & ~int_op;      // round to plus infinity
rc_trunc = (rc[1:0] == `RC_RZ) & ~int_op;    // truncate

//PRECISION CONTROL//
pc_32 = (pc == `PC_32) & ~ext_op & ~int_op;      // single
pc_64 = (pc == `PC_64) & ~ext_op & ~int_op;      // double
pc_80 = (((pc == `PC_80) | (pc == `PC_80R)) & ~int_op) | ext_op; // extended
pc_87 = int_op;      // internal

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

// SELECT CLOSE OR FAR PATH//
diffpos[18:0] = {1'b0,expa[17:0]} + {1'b0,~expb[17:0]} + 19'b1;
diffneg[17:0] = expb[17:0] + ~expa[17:0] + 18'b1;
swap = ~diffpos[18];
expl[17:0] = bzero | (~azero & ~swap) ? expa : expb;
rsa[6:0] = swap ? diffneg[6:0] : diffpos[6:0];
overshift = (swap & (|diffneg[17:7])) | (~swap & (|diffpos[17:7])) |
            (rsa[6] & ((|rsa[5:3]) | (&rsa[2:1])));
far = ~esub | azero | bzero | overshift | (|rsa[6:1]);

// CLOSE PATH//
shift_close = expa[0] ^ expb[0];
swap_close = ~(expa[0] ^ expa[1] ^ expb[1]);
ina_shift_close[68:0] = shift_close  ?  {1'b0,mana[67:0]} : {mana[67:0] ,1'b0};
inb_shift_close[68:0] = shift_close  ?  {1'b0,manb[67:0]} : {manb[67:0] ,1'b0};
```

Fig. 2. Circuit $\mathcal{A}$ (continued)

```
ina_swap_close[68:0] = (shift_close & swap_close) ?
                        {manb[67:0] ,1'b0} : {mana[67:0] ,1'b0};
inb_swap_close[68:0] = (shift_close & swap_close) ?
                        ina_shift_close[68:0] : inb_shift_close[68:0];
lop0[68:0] = {mana[66:0],2'b0} | {1'b0,~manb[66:0],1'b1};
lop1_t[67:0] = mana[67:0] ^ ~manb[67:0];
lop1_g[67:0] = mana[67:0] & ~manb[67:0];
lop1_z[67:0] = ~(mana[67:0] | ~manb[67:0]);
lop1[67:0] = {1'b0,
              (lop1_t[67:2] & lop1_g[66:1] & ~lop1_z[65:0]) |
               (~lop1_t[67:2] & lop1_z[66:1] & ~lop1_z[65:0]) |
               (lop1_t[67:2] & lop1_z[66:1] & ~lop1_g[65:0]) |
               (~lop1_t[67:2] & lop1_g[66:1] & ~lop1_g[65:0]),
              ~lop1_t[0]};
lop2[68:0] = {manb[66:0],2'b0} | {1'b0,~mana[66:0],1'b1};
lop[68:0] = shift_close ? (swap_close ? lop2[68:0] : lop0[68:0]) : {lop1[67:0],1'b0};
found = 1'b0;
for (i=68; i>=0; i=i-1)
  if (lop[i] & ~found)
    begin
      found = 1'b1;
      lsa[6:0] = 7'h44 - i[6:0];
    end

//FAR PATH//
rshiftin_far[67:0] = swap ? mana[67:0] : manb[67:0];
ina_far[67:0] = azero | (swap & ~bzero) ? manb[67:0] : mana[67:0];


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

//PREDICT EXPONENT OF RESULT//
lshift[17:0] = far ? (esub ? 18'h3ffff : 18'b0) : ~{11'b0,lsa[6:0]};
exp[17:0] = expl[17:0] + lshift[17:0];

//ALIGN OPERANDS//
ina_close[68:0] = ~shift_close & (mana < manb) ? inb_swap_close[68:0] << lsa[6:0] :
                                                 ina_swap_close[68:0] << lsa[6:0];
ina_add[70:0] = far ? {ina_far[67:0], 3'b0} : {ina_close[68:0], 2'b0};
inb_close[68:0] = ~shift_close & (mana < manb) ? ina_swap_close[68:0] << lsa[6:0] :
                                                 inb_swap_close[68:0] << lsa[6:0];
rshiftout_far[69:0] = overshift | azero | bzero ?
                      70'b0 : {rshiftin_far[67:0],2'b0} >> rsa[6:0];
sticky_t[194:0] = {rshiftin_far[67:0],127'b0} >> rsa[6:0];
sticky_far = ~(azero | bzero) & (overshift | (|sticky_t[124:58]));
inb_far[70:0] = {rshiftout_far[69:0],sticky_far};
inb_add_nocomp[70:0] = far | azero | bzero ? inb_far[70:0] : {inb_close[68:0],2'b0};
inb_add[70:0] = esub ? ~inb_add_nocomp[70:0] : inb_add_nocomp[70:0];
```

Fig. 3. Circuit $\mathcal{A}$ (continued)

```
//DETERMINE SIGN OF RESULT//
sign_tmp = swap | (~far & ~shift_close & (mana < manb)) ? signb ^ sub : signa;
abequal = esub & (mana == manb) & (expa == expb);
sign_reg = ((~azero & ~bzero & ~abequal & sign_tmp) |
             (~azero & ~bzero & abequal & rc_neg) |
             (azero & ~bzero & (signb ^ sub)) |
             (~azero & bzero & signa) |
             (azero & bzero & signa & (signb ^ sub)) |
             (azero & bzero & (signa ^ (signb ^ sub)) & rc_neg)) & ~(ainf | binf) |
           (ainf & signa) | (binf & (signb ^ sub));

//COMPUTE ROUNDING CONSTANT//
int_noco[70:0] = {68'b0,1'b1,2'b0}; // 71'h4
ext_noco[70:0] = case(1'b1)
                   rc_trunc : 71'b0;            rc_inf  : {64'h0, ~{7 {sign_reg}}};
                   rc_near  : {65'b1,6'b0};     rc_minf : {64'h0,{7 {sign_reg}}};
                 endcase;

doub_noco[70:0] = case(1'b1)
                    rc_trunc : 71'h0;           rc_inf  : {53'h0,~{18 {sign_reg}}};
                    rc_near  : {54'b1,17'b0};   rc_minf : {53'h0,{18 {sign_reg}}};
                  endcase;
sing_noco[70:0] = case(1'b1)
                    rc_trunc : 71'h0;           rc_inf  : {24'h0,~{47 {sign_reg}}};
                    rc_near  : {25'b1,46'b0};   rc_minf : {24'h0,{47 {sign_reg}}};
                  endcase;

rconst_noco[70:0] = case(1'b1)
                      pc_87 : int_noco;    pc_80 : ext_noco;
                      pc_64 : doub_noco;   pc_32 : sing_noco;
                    endcase;


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

//CHECK FOR OVERFLOW OR CANCELLATION//
sum[71:0] = {1'b0,ina_add[70:0]} + {1'b0,inb_add[70:0]} + {71'b0,esub};
overflow = sum[71];
ols = ~sum[70];

//COMPUTE SUM ASSUMING NO OVERFLOW OR CANCELLATION, CHECK FOR CARRYOUT//
sum_noco[70:0] = rconst_noco[70:0] ^ ina_add[70:0] ^ inb_add[70:0];
carry_noco[71:0] = {(rconst_noco[70:0] & ina_add[70:0]) |
                     (rconst_noco[70:0] & inb_add[70:0]) |
                     (ina_add[70:0] & inb_add[70:0]),
                    1'b0};
sum71_noco[72:0] = {2'b0,sum_noco[70:0]} + {1'b0,carry_noco[71:0]} + {72'b0,esub};
overflow_noco = sum71_noco[71];
```

Fig. 4. Circuit $\mathcal{A}$ (continued)

```
//COMPUTE SUM ASSUMING OVERFLOW OR CANCELLATION, CHECK FOR CARRYOUT//
rconst_co[70:0] = esub ? {1'b0,rconst_noco[70:1]} : {rconst_noco[69:0],rconst_noco[0]};
sum_co[70:0] = rconst_co[70:0] ^ ina_add[70:0] ^ inb_add[70:0];
carry_co[71:0] = {(rconst_co[70:0] & ina_add[70:0]) |
                  (rconst_co[70:0] & inb_add[70:0]) |
                  (ina_add[70:0] & inb_add[70:0]),
                 1'b0};
sum71_co[72:0] = {2'b0,sum_co[70:0]} + {1'b0,carry_co[71:0]} + {72'b0,esub};
overflow_co = sum71_co[72];
ols_co = ~sum71_co[70];

//COMPUTE STICKY BIT OF SUM FOR EACH OF THREE CASES//
sticksum[47:0] = esub ? ina_add[47:0] ^ inb_add[47:0] : ~(ina_add[47:0] ^ inb_add[47:0]);
stickcarry[47:0] = esub ? {ina_add[46:0] & inb_add[46:0],1'b0} :
                          {ina_add[46:0] | inb_add[46:0],1'b0};
stick[47:0] = ~(sticksum[47:0] ^ stickcarry[47:0]);
sticky_ols = (|stick[44:16] & pc_32) | (|stick[15:5] & (pc_32 | pc_64)) |
             (|stick[4:1] & ~pc_87) | stick[0] ;
sticky_noco = sticky_ols | (stick[45] & pc_32) | (stick[16] & pc_64) |
              (stick[5] & pc_80) | stick[1] ;
sticky_co =   sticky_noco | (stick[46] & pc_32) | (stick[17] & pc_64) |
              (stick[6] & pc_80) | stick[2] ;


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

//COMPUTE SIGNIFICAND//
man_noco[67:0] =
  {sum71_noco[72] | sum71_noco[71] | sum71_noco[70],
   sum71_noco[69:48],
   sum71_noco[47] & ~(~sum71_noco[46] & ~sticky_noco & pc_32 & rc_near),
   sum71_noco[46:19] & {28 {~pc_32}},
   sum71_noco[18] & ~(pc_32 | (~sum71_noco[17] & ~sticky_noco & pc_64 & rc_near)),
   sum71_noco[17:8] & ~{10{pc_32 | pc_64}},
   sum71_noco[7] & ~(pc_32 | pc_64 | (~sum71_noco[6] & ~sticky_noco & pc_80 & rc_near)),
   sum71_noco[6:4] & ~{3{pc_32 | pc_64 | pc_80}},
   sum71_noco[3] & ~(pc_32 | pc_64 | pc_80 | (~sum71_noco[2] & ~sticky_noco & rc_near))
  };

man_co[67:0] =
  {sum71_co[72] | sum71_co[71],
   sum71_co[70:49],
   sum71_co[48] & ~(~sum71_co[47] & ~sticky_co & pc_32 & rc_near),
   sum71_co[47:20] & {28 {~pc_32}},
   sum71_co[19] & ~(pc_32 | (~sum71_co[18] & ~sticky_co & pc_64 & rc_near)),
   sum71_co[18:9] & ~{10{pc_32 | pc_64}},
   sum71_co[8] & ~((pc_32 | pc_64) | (~sum71_co[7] & ~sticky_co & pc_80 & rc_near)),
   sum71_co[7:5] & ~{3{pc_32 | pc_64 | pc_80}},
   sum71_co[4] & ~(pc_32 | pc_64 | pc_80 | (~sum71_co[3] & ~sticky_co & rc_near))
  };
```

Fig. 5. Circuit $\mathcal{A}$ (continued)

```
man_ols[67:0] =
  {sum71_co[70] | sum71_co[69],
   sum71_co[68:47],
   sum71_co[46] & ~(~sum71_co[45] & ~sticky_ols & pc_32 & rc_near),
   sum71_co[45:18] & {28 {~pc_32}},
   sum71_co[17] & ~(pc_32 | (~sum71_co[16] & ~sticky_ols & pc_64 & rc_near)),
   sum71_co[16:7] & ~{10{pc_32 | pc_64}},
   sum71_co[6] & ~((pc_32 | pc_64) | (~sum71_co[5] & ~sticky_ols & pc_80 & rc_near)),
   sum71_co[5:3] & ~{3{pc_32 | pc_64 | pc_80}},
   sum71_co[2] & ~(pc_32 | pc_64 | pc_80 | (~sum71_co[1] & ~sticky_ols & rc_near))
  };

man_reg[67:0] = case(1'b1)
                  (~esub & ~overflow) | (esub & ~ols) : man_noco[67:0];
                  ~esub & overflow                    : man_co[67:0];
                  esub & ols                          : man_ols[67:0];
                endcase;

//ADJUST EXPONENT://
exp_noco[17:0] = overflow_noco ? exp[17:0] + 18'h1 : exp[17:0];
exp_co[17:0]   = overflow_co   ? exp[17:0] + 18'h2 : exp[17:0] + 18'b1;
exp_noco_sub[17:0] = overflow ^ overflow_noco ?
                     exp[17:0] + 18'h2 : exp[17:0] + 18'h1;
exp_ols[17:0]  = ols_co ? exp[17:0] : exp[17:0] + 18'h1;

exp_reg[17:0] = case(1'b1)
                  (~esub & ~overflow) : exp_noco[17:0];
                  (~esub & overflow)  : exp_co[17:0];
                  (esub & ~ols)       : exp_noco_sub[17:0];
                  (esub & ols)        : exp_ols[17:0];
                endcase;

//DETERMINE CLASS//
class_reg[2:0] = case(1'b1)
                   (azero & bzero) | abequal  : `ZERO;
                   default                    : `NORMAL;
                 endcase;
//FINAL RESULT//
r[89:0] = {class_reg[2:0], sign_reg, exp_reg[17:0], man_reg[67:0]};
endmodule
```

Fig. 6. Circuit $\mathcal{A}$ (continued)

Although it is combinational, our listing of $\mathcal{A}$ reflects the adder's 4-cycle structure insofar as its signals are grouped according to their cycle numbers with respect to the original RTL specification, and our analysis will be guided by this organization. As a first step toward understanding the 4-cycle structure, consider the following procedure, which represents a naive approach to floating point addition and subtraction:

(1) Compare the exponent fields of the summands to determine the right shift necessary to align the significands;
(2) Perform the required right shift on the significand field that corresponds to the lesser exponent;
(3) Add (or subtract) the aligned significands, together with the appropriate rounding constant;
(4) Determine the left shift required to normalize the result;
(5) Perform the left shift and adjust the exponent accordingly;
(6) Compute the final result by assembling the sign, exponent, and significand fields.

Under the constraints imposed by contemporary technology and microprocessor clock rates, each of the above operations might reasonably correspond to a single cycle, resulting in a six-cycle implementation. It is possible, however, to improve on this cycle count by executing some of these operations in parallel.

The most important optimization of the above algorithm is based on the observation that while a large left shift might be required (in the case of subtraction, if massive cancellation occurs), and a large right shift might be required (if the exponents are vastly different), only one of these possibilities will be realized for any given pair of inputs. Thus, the Athlon adder includes two data paths: on one path, called the *far* path, the right shift is determined and executed; on the other, called the *close* path, the left shift is performed instead. As noted in Section 2, the left shift may be determined in advance of the subtraction. Consequently, steps (4) and (5) may be executed concurrently with steps (1) and (2), respectively, resulting in a four-cycle implementation. In Section 5, we shall examine the code corresponding to each cycle in detail.

In the subsequent discussion, we shall assume a fixed execution of $\mathcal{A}$ determined by a given set of values corresponding to the inputs. We adopt the convention of italicizing the name of each signal to denote its value for these inputs. Thus, `r` denotes the output value determined by the inputs `a`, `b`, `op`, `rc`, and `pc`.

The input values a and b are the operands. Each operand is a vector of ninety bits, including a three-bit encoding of its class, along with sign, exponent, and significand fields, according to the AMD Athlon internal $(68, 18)$ format. The fields of a are assigned to $classa$, $signa$, $expa$, and $mana$; those of b are similarly recorded. While all eight classes are handled by the actual RTL, we consider here only the case $classa = classb =$ `NORMAL`, and assume that a and b are normal $(68, 18)$-encodings, i.e., $mana[67] = manb[67] = 1$. Further, in order to ensure that all computed exponent fields are representable in the allotted 18 bits (see,

for example, the proof of Lemma 14(a)), we assume that *expa* and *expb* are both in the range from 69 to $2^{18} - 3$.

The operation to be performed is encoded as the input *op*, which we shall assume to be one of the 11-bit opcodes listed in Figure 1. This opcode may indicate either addition or subtraction, as reflected by the 1-bit signal *sub* (Figure 2). Let $\mathcal{E}$ denote the exact result of this operation, i.e.,

$$\mathcal{E} = \begin{cases} \hat{a} + \hat{b} \text{ if } sub = 0 \\ \hat{a} - \hat{b} \text{ if } sub = 1. \end{cases}$$

For simplicity, we shall assume that $\mathcal{E} \neq 0$.

Rounding control is determined by *rc* along with *op*. According to these two values, exactly one of the bits *rc_near*, *rc_minf*, *rc_inf*, and *rc_trunc* is 1, as shown in Figure 2. We shall introduce a variable $\mathcal{M}$, which we define to be corresponding rounding mode. For example, if *op* is neither `FADDT68` nor `FSUBT68` and *rc* = `RC_RZ`, then $rc\_trunc = 1$ and $\mathcal{M} = trunc$.

Similarly, *pc* and *op* together determine which one of the bits *pc*_32, *pc*_64, *pc*_80, and *pc*_87 is set. We define σ to be the corresponding degree of precision: 24, 53, 64, or 68, respectively.

The result prescribed by the IEEE standard will be denoted as $\mathcal{P}$, i.e.,

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

Our goal may now be stated as follows:

Theorem 1. *Suppose that* $a[89:77]$ *and* $b[89:87]$ *are both* `NORMAL` *and that* $a[86:0]$ *and* $b[86:0]$ *are normal* $(68,18)$*-encodings such that* $69 \leq a[85:68] \leq 2^{18} - 3$ *and* $69 \leq b[85:68] \leq 2^{18} - 3$. *Assume that* $\mathcal{E} \neq 0$. *Then* $r[89:87] =$ `NORMAL`, $r[86:0]$ *is a normal* $(68,18)$*-encoding, and* $\hat{r} = \mathcal{P}$.

Before proceeding with the proof of Theorem 1, we must note that the circuit description $\mathcal{A}$ contains a construct of the RTL language that was not described in [8], namely, the `for` loop (Fig. 3):

```
found = 1'b0;
for (i=68; i>=0; i=i-1)
  if (lop[i] & ~found)
    begin
      found = 1'b1;
      lsa[6:0] = 7'h44 - i[6:0];
    end
```

If an assignment to a signal `s` occurs within a `for` loop, then its value s is computed by a recursive function determined by the loop. In this example, the recursive function Θ for the signal `lsa` is defined by

$$\Theta(lsa, found, lop, i) = \begin{cases} lsa & \text{if } i < 0 \\ \Theta(68 - i, 1, i - 1, lop) & \text{if } lop[i] = 1 \text{ and } found = 0 \\ \Theta(lsa, found, i - 1, lop) & \text{otherwise,} \end{cases}$$

and the value of `lsa` is given by

$$lsa = \Theta(0, 0, val_{\mathcal{A}}(\texttt{lop}, \mathcal{I}, \mathcal{R}), 68).$$

5 Proof of Correctness

The proof of Theorem 1 may be simplified by noting that we may restrict our attention to the case $\mathcal{E} > 0$. In order to see this, suppose that we alter the inputs by toggling the sign bits $a[86]$ and $b[86]$ and replacing rc with rc', where

$$rc' = \begin{cases} \texttt{RC\_RM} \text{ if } rc = \texttt{RC\_RP} \\ \texttt{RC\_RP} \text{ if } rc = \texttt{RC\_RM} \\ rc \quad \text{otherwise.} \end{cases}$$

It is clear by inspection of the code that the only signals affected are $signa$, $signb$, sgn_tmp, and sgn_reg, each of which is complemented, and rc_inf and rc_minf, which are transposed. Consequently, $\hat{r}$ is negated and $\mathcal{M}$ is replaced by $\mathcal{M}'$, where

$$\mathcal{M}' = \begin{cases} minf \text{ if } \mathcal{M} = inf \\ inf \quad \text{if } \mathcal{M} = minf \\ \mathcal{M} \quad \text{otherwise.} \end{cases}$$

Now, from the simple identity $rnd(-x, \mathcal{M}', \sigma) = -rnd(x, \mathcal{M}, \sigma)$, it follows that if Theorem 1 holds under the assumption $\mathcal{E} > 0$, then it holds generally.

The proof proceeds by examining the signals associated with each cycle in succession.

First Cycle

In the initial cycle, the operands are compared, the path is selected, and the left and right shifts are computed for the close and far paths, respectively. We introduce the following notation:

$$\Delta = |expa - expb|,$$

$$\alpha = \begin{cases} mana \text{ if } expa > expb \vee (expa = expb \wedge (far = 1 \vee mana \geq manb)) \\ manb \text{ otherwise,} \end{cases}$$

and

$$\beta = \begin{cases} manb \text{ if } \alpha = mana \\ mana \text{ if } \alpha = manb. \end{cases}$$

We begin by comparing the exponents of the operands:

Lemma 9. *(a)* $swap = 1$ *iff* $expa < expb$; *(b)* $expl = max(expa, expb)$.

Proof: Since $\mathit{diffpos} = \mathit{expa} + \tilde{}\mathit{expb}[17:0] + 1 = 2^{18} + \mathit{expa} - \mathit{expb} < 2^{19}$,

$$\mathit{swap} = 1 \Leftrightarrow \mathit{diffpos}[18] = 0 \Leftrightarrow \mathit{diffpos} < 2^{18} \Leftrightarrow \mathit{expa} < \mathit{expb}. \ \square$$

The path selection is determined by the signal far, which depends on $esub$ and Δ. For the far path, the magnitude of the right shift is given by Δ. We distinguish between the cases $\Delta \geq 70$ and $\Delta < 70$.

Lemma 10.
(a) $overshift = 1$ *iff* $\Delta \geq 70$;
(b) if $overshift = 0$, *then* $rsa = \Delta$;
(c) $far = 0$ *iff* $esub = 1$ *and* $\Delta \leq 1$.

Proof: If $\mathit{swap} = 0$, then $\mathit{diffpos}[17:0] = \mathit{diffpos} - 2^{18} = \mathit{expa} - \mathit{expb}$, and if $\mathit{swap} = 1$, then $\mathit{diffneg}[17:0] = \mathit{diffneg} = \mathit{expb} - \mathit{expa}$. Thus, in either case, $rsa = \Delta[6:0] = rem(\Delta, 128)$. It follows that $overshift = 1$ iff either $\Delta[17:7] = \lfloor \Delta/128 \rfloor \neq 0$ or $rsa \geq 70$, which implies (a), from which (b) and (c) follow immediately. $\square$

The next lemma is an immediate consequence of Lemma 9:

Lemma 11. *Assume* $far = 1$.
(a) $ina\_far = \alpha$; *(b)* $rshiftin\_far = \beta$.

For the close path, the ordering of exponents is determined by $\mathit{shift\_close}$ and $\mathit{swap\_close}$:

Lemma 12. *Assume* $far = 0$.
(a) $\mathit{shift\_close} = 0$ *iff* $\mathit{expa} = \mathit{expb}$;
(b) if $\mathit{shift\_close} = 1$, *then* $\mathit{swap\_close} = 1$ *iff* $\mathit{expa} < \mathit{expb}$;
(c) $\mathit{ina\_swap\_close} = \begin{cases} 2\mathit{mana} & \text{if } \mathit{expa} = \mathit{expb} \\ 2\alpha & \text{if } \mathit{expa} \neq \mathit{expb}; \end{cases}$
(d) $\mathit{inb\_swap\_close} = \begin{cases} 2\mathit{manb} & \text{if } \mathit{expa} = \mathit{expb} \\ \beta & \text{if } \mathit{expa} \neq \mathit{expb}. \end{cases}$

Proof: (a) is a consequence of Lemma 10; (c) and (d) follow from (a) and (b). To prove (b), first note that

$$\begin{aligned} \mathit{expa} > \mathit{expb} &\Leftrightarrow \mathit{expa} - \mathit{expb} = 1 \\ &\Leftrightarrow rem(\mathit{expa} - \mathit{expb}, 4) = 1 \\ &\Leftrightarrow rem(\mathit{expa}[1:0] - \mathit{expb}[1:0], 4) = 1. \end{aligned}$$

Now suppose that $\mathit{expa}[0] = 0$. Then $\mathit{expb}[0] = 1$ and

$$\mathit{swap\_close} = 1 \Leftrightarrow \mathit{expa}[1] = \mathit{expb}[1] \Leftrightarrow rem(\mathit{expa}[1:0] - \mathit{expb}[1:0], 4) = 1.$$

On the other hand, if $\mathit{expa}[0] = 1$, then $\mathit{expb}[0] = 0$ and

$$\mathit{swap\_close} = 1 \Leftrightarrow \mathit{expa}[1] \neq \mathit{expb}[1] \Leftrightarrow rem(\mathit{expa}[1:0] - \mathit{expb}[1:0], 4) = 1. \ \square$$

The magnitude of the left shift is determined by the signal lsa:

Lemma 13. *If* $far = 0$, *then* $66 - lsa \leq expo(\alpha - 2^{-\Delta}\beta) \leq 67 - lsa$.

Proof: As noted in Section 4, $lsa = \Theta(0, 0, lop, 68)$, where

$$\Theta(lsa, found, lop, i) = \begin{cases} lsa & \text{if } i < 0 \\ \Theta(68 - i, 1, i - 1, lop) & \text{if } lop[i] = 1 \text{ and } found = 0 \\ \Theta(lsa, found, i - 1, lop) & \text{otherwise.} \end{cases}$$

It is easily shown by induction on i that if $0 < i \leq 68$ and $0 < lop < 2^{i+1}$, then $\Theta(0, 0, lop, i) = 68 - expo(lop)$. In particular, if $0 < lop < 2^{69}$, then $lsa = 68 - expo(lop)$. Thus, it will suffice to show that $lop > 0$ and that

$$expo(lop) - 2 \leq expo(\alpha - 2^{-\Delta}\beta) \leq expo(lop) - 1.$$

First consider the case $expa > expb$. We may apply Lemma 3, substituting $2mana$, $manb$, and $lop = lop0$ for a, b, and λ, respectively, and conclude that $lop > 0$ and

$$expo(lop) - 1 \leq expo(2mana - manb) \leq expo(lop).$$

But in this case,

$$expo(2mana - manb) = expo(2(\alpha - 2^{-1}\beta)) = expo(\alpha - 2^{-\Delta}\beta) + 1,$$

and the desired inequality follows. The case $expa < expb$ is similar.

Now suppose $expa = expb$. Here we apply Lemma 4, substituting $mana$, $manb$, and $lop1$ for a, b, and λ, which yields $lop1 > 0$ and

$$expo(lop1) - 1 \leq expo(mana - manb) \leq expo(lop1).$$

But

$$expo(mana - manb) = expo(\alpha - \beta) = expo(\alpha - 2^{-\Delta}\beta)$$

and $expo(lop) = expo(2lop1) = expo(lop1) + 1$. □

Second Cycle

In the next cycle, the sign of the result is computed, its exponent is approximated, and the significand fields of the operands are aligned by performing the appropriate shifts. The results of this alignment are the signals *ina_add* and *inb_add_nocomp*. In the case of subtraction, *inb_add_nocomp* is replaced by its complement.

In all cases, α and β are first padded on the right with three 0's, and β is shifted right according to Δ. On the close path, both inputs are then shifted left as determined by lsa, and the exponent field of the result is predicted as $expl - lsa - 1$:

Lemma 14. *Assume* $far = 0$.

(a) $exp = expl - lsa - 1$;
(b) $ina\_add = rem(2^{lsa+3}\alpha, 2^{71})$;
(c) $inb\_add\_nocomp = rem(2^{lsa+3-\Delta}\beta, 2^{71})$.

Proof: (a) The constraints on *expa* and *expb* ensure that $69 \le expl \le 2^{18} - 3$. Since $0 \le lsa \le 68$, it follows that $0 \le expl - lsa - 1 < 2^{18}$, hence

$$exp = rem(expl + 2^{18} - lsa - 1, 2^{18}) = expl - lsa - 1.$$

(b) It follows from Lemma 12 that $ina\_close = rem(2\alpha \cdot 2^{lsa}, 2^{69})$, hence

$$ina\_add = 4rem(2\alpha \cdot 2^{lsa}, 2^{69}) = rem(2^{lsa+3}\alpha, 2^{71}).$$

(c) This is similar to (b). □

On the far path, β is rounded after it is shifted to the right. The predicted exponent field is *expl* for addition and $expl - 1$ for subtraction:

Lemma 15. *Assume* $far = 1$.

(a) $exp = expl - esub$;
(b) $ina\_add = 8\alpha$;
(c) $inb\_add\_nocomp = \begin{cases} sticky(2^{3-\Delta}\beta, 71-\Delta) & \text{if } \Delta < 70 \\ 1 & \text{if } \Delta \ge 70. \end{cases}$

Proof: (a) and (b) are trivial, as is (c) in the case $\Delta \ge 70$. Suppose $\Delta < 70$. Since $rshiftin\_far = \beta$ and $expo(\beta \cdot 2^{2-\Delta}) = 69 - \Delta$,

$$rshiftout\_far = \lfloor \beta \cdot 2^{2-\Delta} \rfloor = trunc(\beta \cdot 2^{2-\Delta}, 70 - \Delta)$$

and $sticky\_t = \beta \cdot 2^{127-\Delta}$. Thus,

$$\begin{aligned} sticky\_far = 0 &\Leftrightarrow (\beta \cdot 2^{127-\Delta})[124:58] = 0 \\ &\Leftrightarrow (\beta \cdot 2^{127-\Delta})[124:0] = 0 \\ &\Leftrightarrow \beta \cdot 2^{127-\Delta} \text{ is divisible by } 2^{125} \\ &\Leftrightarrow \beta \cdot 2^{2-\Delta} \in \mathbb{Z} \\ &\Leftrightarrow \beta \text{ is } (70-\Delta)\text{-exact.} \end{aligned}$$

Thus, if $sticky\_far = 0$, then

$$\begin{aligned} inb\_add\_nocomp &= 2 \cdot rshiftout\_far + 0 \\ &= trunc(\beta \cdot 2^{3-\Delta}, 70 - \Delta) \\ &= sticky(\beta \cdot 2^{3-\Delta}, 71 - \Delta), \end{aligned}$$

and otherwise,

$$\begin{aligned} inb\_add\_nocomp &= 2 \cdot rshiftout\_far + 1 \\ &= trunc(\beta \cdot 2^{3-\Delta}, 70 - \Delta) + 2^{(3-\Delta+67)+1-(71-\Delta)} \\ &= sticky(\beta \cdot 2^{3-\Delta}, 71 - \Delta). \ \square \end{aligned}$$

It is clear that the sign of the final result is correctly given by *sign_reg*:

Lemma 16. *If $\mathcal{E} > 0$, then $sign\_reg = 0$.*

The rounding constant is also computed in this cycle. The following lemma is easily verified for all possible values of $\mathcal{M}$ and σ:

Lemma 17. *If $\mathcal{E} > 0$, then $rconst\_noco = \mathcal{C}(70, \mathcal{M}, \sigma)$.*

Third Cycle

In this cycle, three sums are computed in parallel. The first of these is the unrounded sum

$$sum = ina\_add + inb\_add + esub.$$

In the case of (effective) addition, the leading bit $sum[71]$ is examined to check for overflow, i.e., to determine whether $expo(sum)$ is 70 or 71. In the subtraction case, only $sum[70:0]$ is of interest—the leading bit $sum[70]$ is checked to determine whether cancellation occurred. Thus, one of three possible rounding constants is required, depending on whether the exponent of the unrounded sum is 69, 70, or 71. The second adder computes the rounded sum assuming an exponent of 70; the third assumes 71 in the addition case and 69 for subtraction. Concurrently, the relevant sticky bit of the unrounded sum is computed for each of the three cases.

Lemma 18. *Assume $\mathcal{E} > 0$ and $esub = 0$.*

(a) $expo(sum) = \begin{cases} 70 \text{ if } overflow = 0 \\ 71 \text{ if } overflow = 1; \end{cases}$

(b) $rnd(sum, \mathcal{M}, \sigma) = 2^{2^{17}+69-exp}\mathcal{P}$.

Proof: In this case,

$$sum = ina\_add + inb\_add.$$

Suppose $\Delta < 70$. Then by Lemma 8,

$$\begin{aligned} sum &= 8\alpha + sticky(2^{3-\Delta}\beta, 71-\Delta) \\ &= 8sticky(\alpha + 2^{-\Delta}\beta, expo(\alpha + 2^{-\Delta}\beta) + 4), \end{aligned}$$

where $67 \le expo(\alpha + 2^{-\Delta}\beta) \le 68$.

On the other hand, if $\Delta \ge 70$, then since $expo(\alpha) = expo(\beta) = 67$, we have $0 < 2^{2-\Delta}\beta < 1$ and $expo(\alpha + 2^{-\Delta}\beta) = 67$, which imply

$$\begin{aligned} trunc(\alpha + 2^{-\Delta}\beta, 70) &= \lfloor 2^{70-1-67}(\alpha + 2^{-\Delta}\beta)\rfloor 2^{67+1-70} \\ &= \lfloor 2^2\alpha + 2^{2-\Delta}\beta\rfloor 2^{-2} \\ &= \alpha \end{aligned}$$

and

$$\begin{aligned} sum &= 8\alpha + 1 \\ &= 8(trunc(\alpha + 2^{-\Delta}\beta, 70) + 2^{expo(\alpha+2^{-\Delta}\beta)+1-71}) \\ &= 8sticky(\alpha + 2^{-\Delta}\beta, 71) \\ &= 8sticky(\alpha + 2^{-\Delta}\beta, expo(\alpha + 2^{-\Delta}\beta) + 4). \end{aligned}$$

In either case, $70 \le expo(sum) \le 71$, which yields (a).

To complete the proof of (b), note first that

$$|\hat{a}| + |\hat{b}| = 2^{expa-2^{17}-66}mana + 2^{expb-2^{17}-66}manb = 2^{expl-2^{17}-66}(\alpha + 2^{-\Delta}\beta),$$

and hence, since $exp = expl$,

$$rnd(\alpha + 2^{-\Delta}\beta, \mathcal{M}, \sigma) = 2^{-exp+2^{17}+66}rnd(|\hat{a}| + |\hat{b}|, \mathcal{M}, \sigma) = 2^{-exp+2^{17}+66}\mathcal{P}.$$

Now since $\sigma \le 68$, Lemma 7 implies

$$\begin{aligned} rnd(sum, \mathcal{M}, \sigma) &= 2^3 rnd(sticky(\alpha + 2^{-\Delta}\beta, expo(\alpha + 2^{-\Delta}\beta) + 4), \mathcal{M}, \sigma) \\ &= 2^3 rnd(\alpha + 2^{-\Delta}\beta, \mathcal{M}, \sigma) \\ &= 2^{2^{17}+69-exp}\mathcal{P}. \ \square \end{aligned}$$

Lemma 19. *Assume $\mathcal{E} > 0$ and $esub = 1$.*

(a) $expo(sum[70:0]) = \begin{cases} 70 \text{ if } ols = 0 \\ 69 \text{ if } ols = 1; \end{cases}$

(b) $rnd(sum[70:0], \mathcal{M}, \sigma) = 2^{2^{17}+68-exp}\mathcal{P}$.

Proof: In this case,

$$sum = ina\_add + inb\_add + 1 = 2^{71} + ina\_add - inb\_add\_nocomp.$$

Also note that

$$||\hat{a}| - |\hat{b}|| = 2^{expl-2^{17}-66}(\alpha - 2^{-\Delta}\beta),$$

hence

$$rnd(\alpha - 2^{-\Delta}\beta, \mathcal{M}, \sigma) = 2^{-expl+2^{17}+66}rnd(||\hat{a}| - |\hat{b}||, \mathcal{M}, \sigma) = 2^{-expl+2^{17}+66}\mathcal{P}.$$

Suppose first that $far = 0$. By Lemmas 13 and 14,

$$\begin{aligned} sum[70:0] &= rem(rem(2^{lsa+3}\alpha, 2^{71}) - rem(2^{lsa+3-\Delta}\beta, 2^{71}), 2^{71}) \\ &= rem(2^{lsa+3}(\alpha - 2^{-\Delta}\beta), 2^{71}) \\ &= 2^{lsa+3}(\alpha - 2^{-\Delta}\beta), \end{aligned}$$

where $69 \le expo(2^{lsa+3}(\alpha - 2^{-\Delta}\beta)) \le 70$. Thus, since $exp = expl - lsa - 1$,

$$\begin{aligned} rnd(sum[70:0], \mathcal{M}, \sigma) &= 2^{lsa+3}rnd(\alpha - 2^{-\Delta}\beta, \mathcal{M}, \sigma) \\ &= 2^{lsa+3}2^{-expl+2^{17}+66}\mathcal{P} \\ &= 2^{2^{17}+68-exp}\mathcal{P}. \end{aligned}$$

Next, suppose $far = 1$. Then $\Delta \geq 2$, and it follows that $66 \leq expo(\alpha - 2^{-\Delta}\beta) \leq 67$. If $\Delta < 70$, then

$$\begin{aligned} sum[70:0] &= 8\alpha - sticky(2^{3-\Delta}\beta, 71-\Delta) \\ &= 8sticky(\alpha - 2^{-\Delta}\beta, expo(\alpha - 2^{-\Delta}\beta) + 4). \end{aligned}$$

But if $\Delta \geq 70$, then $0 < 2^{2-\Delta}\beta < 1$, and hence

$$\begin{aligned} &trunc(\alpha - 2^{-\Delta}\beta, expo(\alpha - 2^{-\Delta}\beta) + 3) \\ &= \lfloor 2^2(\alpha - 2^{-\Delta}\beta)\rfloor 2^{-2} = \lfloor 2^2\alpha - 2^{2-\Delta}\beta\rfloor 2^{-2} = (2^2\alpha - 1)2^{-2} = \alpha - 2^{-2}, \end{aligned}$$

which implies

$$\begin{aligned} &sticky(\alpha - 2^{-\Delta}\beta, expo(\alpha - 2^{-\Delta}\beta) + 4) \\ &= trunc(\alpha - 2^{-\Delta}\beta, expo(\alpha - 2^{-\Delta}\beta) + 3) + 2^{-3} = \alpha - 2^{-3}, \end{aligned}$$

and again

$$sum[70:0] = 8\alpha - 1 = 8sticky(\alpha - 2^{-\Delta}\beta, expo(\alpha - 2^{-\Delta}\beta) + 4).$$

Thus,

$$69 \leq expo(sum[70:0]) = expo(\alpha - 2^{-\Delta}\beta) + 3 \leq 70.$$

Since

$$expo(\alpha - 2^{-\Delta}\beta) + 4 \geq 70 \geq \sigma + 2$$

and $exp = expl - 1$, Lemma 7 implies

$$\begin{aligned} rnd(sum[70:0], \mathcal{M}, \sigma) &= 2^3 rnd(sticky(\alpha - 2^{-\Delta}\beta, expo(\alpha - 2^{-\Delta}\beta) + 4), \mathcal{M}, \sigma) \\ &= 2^3 rnd(\alpha - 2^{-\Delta}\beta, \mathcal{M}, \sigma) \\ &= 2^{2^{17}+68-exp}\mathcal{P}. \ \square \end{aligned}$$

The next lemma is a straightforward application of Lemma 1:

Lemma 20. *$sum71\_noco = rconst\_noco + sum$.*

If the exponent of the sum is not 70, a different rounding constant must be used. Applying Lemma 1 again, and referring to the definition of $\mathcal{C}$, we have the following:

Lemma 21. *$sum71\_co = rconst\_co + sum$, where if $\mathcal{E} > 0$,*

$$rconst\_co = \begin{cases} \mathcal{C}(71, \mathcal{M}, \sigma) \text{ if } esub = 0 \\ \mathcal{C}(69, \mathcal{M}, \sigma) \text{ if } esub = 1. \end{cases}$$

The next lemma is required for the *near* case:

Lemma 22. *Let* $S = \begin{cases} sum & \text{if } esub = 0 \\ sum[70:0] & \text{if } esub = 1. \end{cases}$ *Then* S *is* $(\sigma+1)$*-exact iff any of the following holds:*

(a) $esub = 0$, $overflow = 0$, *and* $sticky\_noco = 0$;
(b) $esub = 0$, $overflow = 1$, *and* $sticky\_co = 0$;
(c) $esub = 1$, $ols = 0$, *and* $sticky\_noco = 0$;
(d) $esub = 1$, $ols = 1$, *and* $sticky\_ols = 0$.

Proof: S is $(\sigma+1)$-exact iff S is divisible by $2^{expo(S)-\sigma}$, i.e., $sum[expo(S) - \sigma - 1 : 0] = 0$. Invoking Lemma 5 with $a = ina\_add[47:0]$, $b = inb\_add[47:0]$, $c = esub$, and $n = 48$, we conclude that for all $k < 48$, $sum[k:0] = 0$ iff $stick[k:0] = 0$. In particular, since $expo(S) - \sigma - 1 \leq 71 - 24 - 1 = 46$, S is $(\sigma+1)$-exact iff $stick[expo(S) - \sigma - 1 : 0] = 0$.

Suppose, for example, that $esub = ols = 1$ and $\sigma = 24$. In this case,

$$\begin{aligned} sticky\_ols = 0 &\Leftrightarrow stick[44:16] = stick[15:5] = stick[4:1] = stick[0] = 0 \\ &\Leftrightarrow stick[44:0] = 0. \end{aligned}$$

But $expo(S) - \sigma - 1 = 69 - 24 - 1 = 44$, hence S is $(\sigma+1)$-exact iff $sticky\_ols = 0$. All other cases are similar. □

Fourth Cycle

In the final cycle, the significand field is extracted from the appropriate sum, and the exponent field is adjusted as dictated by overflow or cancellation. The resulting output r is an encoding of the prescribed result $\mathcal{P}$, as guaranteed by Theorem 1.

We now complete the proof of the theorem. As noted earlier, we may assume that $\mathcal{E} > 0$. By Lemma 16, $sign\_reg = 0$, hence our goal is to show that

$$2^{exp\_reg - 2^{17} - 66} man\_reg = \mathcal{P}.$$

We shall present the proof for the case $esub = 1$, $ols = 0$; the other three cases are similar.

By Lemma 19, $expo(sum[70:0]) = 70$ and

$$rnd(sum[70:0], \mathcal{M}, \sigma) = 2^{2^{17} + 68 - exp} \mathcal{P}.$$

Since $man\_reg = man\_noco$ and $exp\_reg = exp\_noco\_sub$, we must show that

$$\begin{aligned} man\_noco &= 2^{-exp\_noco\_sub + 2^{17} + 66} \mathcal{P} \\ &= 2^{-exp\_noco\_sub + 2^{17} + 66} 2^{-2^{17} - 68 + exp} rnd(sum[70:0], \mathcal{M}, \sigma) \\ &= 2^{exp - exp\_noco\_sub - 2} rnd(sum[70:0], \mathcal{M}, \sigma). \end{aligned}$$

Let

$$\nu = \begin{cases} \sigma - 1 & \text{if } \mathcal{M} = near \text{ and } sum[70:0] \text{ is } (\sigma+1)\text{-exact but not } \sigma\text{-exact} \\ \sigma & \text{otherwise.} \end{cases}$$

By Lemmas 6 and 17,

$$rnd(sum[70:0], \mathcal{M}, \sigma) = trunc(sum[70:0] + rconst\_noco, \nu).$$

By Lemma 22, $sum[70:0]$ is $(\sigma+1)$-exact iff $sticky\_noco = 0$. If $sticky\_noco = 0$ and $\mathcal{M} = near$, then

$$\begin{aligned} sum[70:0] \text{ is } \sigma\text{-exact} &\Leftrightarrow sum[70-\sigma] = 0 \\ &\Leftrightarrow (sum + 2^{70-\sigma})[70-\sigma] = sum71\_noco[70-\sigma] = 1. \end{aligned}$$

Thus,

$$\nu = \begin{cases} \sigma - 1 \text{ if } \mathcal{M} = near \text{ and } sticky\_noco = sum71\_noco[70-\sigma] = 0 \\ \sigma \qquad \text{otherwise,} \end{cases}$$

and it is easy to check, for each possible value of σ, that

$$man\_noco[66:0] = sum71\_noco[69:71-\nu] \cdot 2^{68-\nu}.$$

Since $expo(sum[70:0]) = 70$ and $expo(rconst\_noco) \leq 70 - \sigma$,

$$70 \leq expo(sum[70:0] + rconst\_noco) \leq 71,$$

and therefore

$$\begin{aligned} expo(sum[70:0] + rconst\_noco) = 70 &\Leftrightarrow (sum[70:0] + rconst\_noco)[71] = 0 \\ &\Leftrightarrow (sum + rconst\_noco)[71] = sum[71] \\ &\Leftrightarrow overflow\_noco = overflow. \end{aligned}$$

Suppose first that $overflow\_noco \neq overflow$. Since $expo(sum[70:0]) = 70$ and

$$expo(rnd(sum[70:0], \mathcal{M}, \sigma)) = expo(sum[70:0] + rconst\_noco) = 71,$$

$rnd(sum[70:0], \mathcal{M}, \sigma)) = 2^{71}$. In this case, $exp\_noco\_sub = exp + 2$, so we must prove that

$$man\_noco = 2^{-4} rnd(sum[70:0], \mathcal{M}, \sigma) = 2^{67}.$$

Since

$$2^{71} \leq sum[70:0] + rconst\_noco < 2^{71} + 2^{71-\sigma},$$

we have

$$sum71\_noco[70:0] = (sum[70:0] + rconst\_noco)[70:0] < 2^{71-\sigma},$$

which implies $sum71\_noco[70:71-\sigma] = 0$, and therefore $man\_noco[66:0] = 0$. But since $2^{70} \leq sum71\_noco < 2^{73}$,

$$man\_noco[67] = sum71\_noco[72] \mid sum71\_noco[71] \mid sum71\_noco[70] = 1$$

and $man\_noco = 2^{67}$.

Now suppose that $overflow\_noco = overflow$. Since $exp\_reg = exp + 1$, we must show

$$man\_noco = 2^{-3} rnd(sum[70:0], \mathcal{M}, \sigma).$$

But since $expo(sum[70:0] + rconst\_noco) = 70$,

$$\begin{aligned} man\_noco &= sum71\_noco[70 : 71 - \nu] \cdot 2^{68-\nu} \\ &= 2^{-3}(sum71\_noco[70:0] \;\&\; (2^{71} - 2^{71-\nu})) \\ &= 2^{-3} trunc(sum71\_noco[70:0], \nu) \\ &= 2^{-3} trunc(sum[70:0] + rconst\_noco, \nu) \\ &= 2^{-3} rnd(sum[70:0], \mathcal{M}, \sigma). \; \square \end{aligned}$$

6 ACL2 Formalization

In this section, we describe formalization of Theorem 1, including the automatic translation of the RTL model to the ACL2 logic, the formal statement of the theorem, and the mechanization of its proof. Naturally, a prerequisite for all of this is the formalization of the general theory of bit vectors and floating-point arithmetic. This is described in some detail in [8], and the reader may also refer to the complete on-line floating-point library [9].

Our translator is an ACL2 program that accepts any simple pipeline (as defined in Section 4) and automatically produces an equivalent set of executable ACL2 function. For the present purpose, the translator was applied to two circuit descriptions. The first was the actual register-transfer logic for the AMD Athlon adder, the main object of interest. The second was a simplified version, an extension of the circuit $\mathcal{A}$ shown in Section 4, including additional inputs and outputs pertaining to overflow, underflow, and other exceptional conditions. The proof of equivalence of the corresponding resulting ACL2 functions was a critical step in verifying the correctness of the adder.

The first task of the translation is to check that the given circuit description is indeed a simple pipeline, so that it may be replaced by an equivalent combinational circuit. An ordering of the signals of the circuit is then constructed, with respect to which each signal is preceded by those on which it depends.

The main enterprise of the translator is the definition of an ACL2 function corresponding to each signal s, excluding inputs, based on the RTL expression for s. This requires a translation from each primitive RTL operation to an appropriate primitive or defined function of ACL2. For example, the function definition generated for the signal `sticksum` of Fig. 4, constructed from the assignment

```
sticksum[47:0] = esub ?
                   ina_add[47:0] ^ inb_add[47:0]
                 : ~(ina_add[47:0] ^ inb_add[47:0]);
```

is

```
(defun sticksum (inb-add ina-add esub)
  (if (equal esub 0)
      (comp1 (logxor (bits ina-add 47 0)
                     (bits inb-add 47 0))
             48)
    (logxor (bits ina-add 47 0)
            (bits inb-add 47 0)))).
```

Iteration presents a minor complication to this scheme, but the RTL loop construct may be effectively translated into LISP recursion. For example, the iterative definition of the signal `lsa` of Fig. 3 generates the following ACL2 code:

```
(defun lsa-aux (lsa found lop i)
  (declare (xargs :measure (1+ i)))
  (if (and (integerp i) (>= i 0))
      (if (not (equal (logand (bitn lop i)
                              (comp1 found 1))
                      0))
          (lsa-aux (- 68 i) found lop (1- i))
        (lsa-aux lsa 1 lop (1- i)))
    lsa))

(defun lsa (lop) (lsa-aux 0 0 lop 68))
```

Note that a *measure* declaration was inserted by hand as a hint to the prover in establishing the admissibility of the recursive definition of `lsa-aux`, but this was the only modification required of the automatically generated code.

Finally, an ACL2 function corresponding to each output of the circuit is generated, with the circuit inputs as arguments. This function is defined by means of the `let*` operator, calling in succession the functions corresponding to the circuit's wires and binding their values to variables that represent the corresponding signals. Finally, the binding of the selected output signal is returned. The function corresponding to the sole output of our simplified adder takes the following form:

```
(defun adder (a b op rc pc)
  (let* ((mana (mana a))
         (manb (manb b))
         (expa (expa a))
         (expb (expb b))
         (signa (signa a))
         (signb (signb b))
         .....
         (r (r class-reg sign-reg exp-reg man-reg)))
    r))
```

The number of bindings in this definition (i.e., the number of wires in the circuit) is 209. The translation of the actual adder RTL is similar, but much longer, involving over 700 bindings. However, the proof of equivalence of these two functions was fairly straightforward (using the ACL2 prover), and we were then able to restrict our attention to the simpler circuit without compromising our objective of establishing the correctness of the actual RTL.

While there are a number of other feasible translation schemes, the one described above was selected because (a) the correspondence between the RTL and the resulting ACL2 code is easily recognizable, and (b) the ACL2 code may be executed (and thus tested) fairly efficiently. The disadvantage of this scheme, however, is that it produces functions that are not amenable to direct formal analysis. For this purpose, some reformulation of these functions is required.

Our goal is to generate a mechanical proof of Theorem 1 by formalizing the reasoning of Section 5. Thus, we would like to be able to state and prove a lemma pertaining to a given signal, invoking previously proved results concerning other signals, without explicitly listing these previous results or stating the dependence on these other signals in the hypothesis of the lemma. This does not seem possible if our lemmas are to be statements about the ACL2 functions described above.

Our solution to this problem is based on two features of ACL2: *encapsulation*, which allows functions to be characterized by constraining axioms rather than complete definitions, and *functional instantiation*, which allows lemmas pertaining to constrained functions to be applied to other functions that can be shown to satisfy the same constraints.

Suppose that the hypothesis and conclusion of Theorem 1 are formally represented by the functions `input-spec` and `output-spec`, respectively, so that the theorem is encoded as the formula

```
(implies (input-spec a b op rc pc)
         (output-spec a b op rc pc (adder a b op rc pc))).
```

Through encapsulation, we introduce constant functions `a*`, `b*`, etc. corresponding to the inputs by executing the following ACL2 event:

```
(encapsulate ((a* () t) (b* () t) ...)
   (local (defun a* () ...))
   (local (defun b* () ...))
    ...
   (defthm inputs* (input-spec (a*) (b*) (op*) (rc*) (pc*)))).
```

Here, the definitions of `a*`, `b*`, etc. are irrelevant as long as they allow the proof of the formula `inputs*`. The result of this event is that the functions that it introduces are undefined, but constrained to satisfy `inputs*`.

Next, we define a second set of functions corresponding to the wires of the circuit. These functions are constants, derived from the first set of functions by replacing each occurrence of a signal with the corresponding constant. For example:

```
(defun sticksum* ()
  (if (equal (esub*) 0)
      (comp1 (logxor (bits (ina-add*) 47 0)
                     (bits (inb-add*) 47 0))
             48)
    (logxor (bits (ina-add*) 47 0)
            (bits (inb-add*) 47 0)))).
```

(In fact, the translator has been modified to generate these definitions as well.) The purpose of these functions is to facilitate formal reasoning about the signals of our circuit, allowing us to prove a lemma about the behavior of a signal by invoking previously proved lemmas about the signals on which it depends. Thus, to prove a lemma pertaining to the constant `(sticksum*)`, we may expand its definition and invoke any relevant lemmas about `(ina-add*)` and `(inb-add*)`. In this manner, tracing the proofs of Section 5 step by step, we arrive at the following result:

```
(defthm r*-spec
  (output-spec (a*) (b*) (op*) (rc*) (pc*) (r*))).
```

But simply by expanding definitions, we may also easily prove

```
(defthm r*-adder
  (equal (r*) (adder (a*) (b*) (op*) (rc*) (pc*))))
```

and combining the last two lemmas, we trivially deduce

```
(defthm outputs*
  (output-spec (a*) (b*) (op*) (rc*) (pc*)
               (adder (a*) (b*) (op*) (rc*) (pc*))))
```

Finally, our desired theorem may be derived from the constraint `inputs*` and the theorem `outputs*` by functional instantiation:

```
(defthm correctness-of-adder
   (implies (input-spec a b op rc pc)
            (output-spec a b op rc pc (adder a b op rc pc)))
   :hints (("goal" :use
            ((:functional-instance outputs*
               (a* (lambda ()
                     (if (input-spec a b op rc pc) a (a*))))
               (b* (lambda ()
                     (if (input-spec a b op rc pc) b (b*))))
                ...)))))
```

In this final ACL2 event, a hint is provided to the prover: use the *fuctional instance* of the lemma `outputs*` that is produced by replacing each of the functions `a*`,`b*`, ... with a certain zero-argument *lambda expression*. Thus, the function `a*` is to be replaced by the lambda expression

```
(lambda () (if (input-spec a b op rc pc) a (a*))),
```

the value of which is

```
(if (input-spec a b op rc pc) a (a*)),
```

and the constant corresponding to each of the other inputs is similarly instantiated. Then, according to the principle of functional instantiation, the desired theorem may be established by proving two subgoals. The first is the implication that the statement of the theorem follows from the instantiated lemma:

```
(implies
  (output-spec a b op rc pc
               (adder (if (input-spec a b op rc pc) a (a*))
                      (if (input-spec a b op rc pc) b (b*))
                      ...))
  (implies (input-spec a b op rc pc)
           (output-spec a b op rc pc (adder a b op op rc pc)))
```

The second subgoal is the corresponding functional instance of the constraint `inputs*`:

```
(input-spec (if (input-spec a b op rc pc) a (a*))
            (if (input-spec a b op rc pc) b (b*))
            ...).
```

But the first subgoal is trivial, second follows from `inputs*` itself, and the theorem `correctness-of-adder` follows.

Acknowledgements

Several people have contributed to this project. The RTL-ACL2 translator was implemented by Art Flatau. Stuart Oberman designed the Athlon adder and explained it to the author. Matt Kaufmann and J Moore provided some helpful modifications of ACL2 and advice in its use.

References

1. Institute of Electrical and Electronic Engineers, "IEEE Standard for Binary Floating Point Arithmetic", Std. 754-1985, New York, NY, 1985.
2. Intel Corporation, *Pentium Family User's Manual, Volume 3: Architecture and Programming Manual*, 1994.
3. Kaufmann, M., Manolios, P., and Moore, J, *Computer-Aided Reasoning: an Approach*, Kluwer Academic Press, 2000.
4. Moore, J, Lynch, T., and Kaufmann, M., "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.

5. Oberman, S., Hesham, A., and Flynn, M., "The SNAP Project: Design of Floating Point Arithmetic Units", Computer Systems Lab., Stanford U., 1996.
6. Russinoff, D., "A Mechanically Checked Proof of IEEE Compliance of the AMD-K5 Floating Point Square Root Microcode", *Formal Methods in System Design* **14** (1):75-125, January 1999. See URL `http://www.onr.com/user/russ/david/fsqrt.html`.
7. Russinoff, D., "A Mechanically Checked Proof of IEEE Compliance of the AMD-K7 Floating Point Multiplication, Division, and Square Root Algorithms". See URL `http://www.onr.com/user/russ/david/k7-div-sqrt.html`.
8. Russinoff, D. and Flatau, A., "RTL Verification: A Floating-Point Multiplier", in Kaufmann, M., Manolios, P., and Moore, J, eds., *Computer-Aided Reasoning: ACL2 Case Studies*, Kluwer Academic Press, 2000. See URL `http://www.onr.com/user/russ/david/acl2.html`.
9. Russinoff, D., "An ACL2 Library of Floating-Point Arithmetic", 1999. See URL `http://www.cs.utexas.edu/users/moore/publications/others/fp-README.html`.