Verifying Relative Error Bounds using Symbolic Simulation

Jesse Bingham and Joe Leslie-Hurd

Intel Corporation, Hillsboro, U.S.A
jesse.d.bingham@intel.com
joe.leslie-hurd@intel.com

Abstract. In this paper we consider the problem of formally verifying hardware that is specified to compute reciprocal, reciprocal square root, and power-of-two functions on floating point numbers to within a given *relative error*. Such specifications differ from the common case in which any given input is specified to have *exactly one* correct output. Our approach is based on symbolic simulation with binary decision diagrams, and involves two distinct steps. First, we prove a lemma that reduces the relative error specification to several inequalities that involve reasoning about natural numbers only. The most complex of these inequalities asserts that the product of several naturals is less-than/greater-than another natural. Second, we invoke one of several customized algorithms that decides the inequality, without performing the expensive symbolic multiplications directly. We demonstrate the effectiveness of our approach on a next-generation Intel® processor design and report encouraging time and space metrics for these proofs.

1 Introduction

Formal verification of hardware data path designs is by now standard practice for many design organizations, see e.g. [8, 16, 12, 17]. Typically the specifications for such circuits are *functional*, meaning that there is exactly one correct output for any given input. In principle, verification can be carried out by writing an executable specification and checking that for all inputs, the output of the design is equal to that of the specification. Symbolic simulation allows one to verify this for all inputs in one fell swoop.[1]

In this paper, we consider designs with specifications that are not functional since a given input can correctly produce any one of a *multitude* of possible outputs. These specifications only require that the design result *approximates* the true mathematical result, in the sense that the relative error is less than some bound. Note that this is distinct from many functional specifications that allow approximate results via rounding; in that case the rounding is precisely defined so that there is still exactly one correct answer.

[1] Although for some operations, one must employ case splitting and/or decomposition due to exponential blow up.

We consider three unary operations in this work: reciprocal (`RCP`), reciprocal square root (`RSQRT`), and power-of-two (`EXP2`). The first two take one IEEE floating point number as input, while power-of-two takes a fixed-point number; all three *produce* one IEEE floating point number as output. The common thread in verifying these three operations is summarized by the following two elements:

1. Express the relative error specification as two inequalities, each of the form

$$B \;\langle\rangle\; \prod_{i=1}^{n} M_i \tag{1}$$

   where $\langle\rangle$ is either $<$ or $>$, along with some simpler conditions also involving only integer reasoning. Here B and $M_1, \ldots, M_n$ are positive integers that are specific to the operation under consideration, and each have tractable symbolic representations. The equivalence of these inequalities to the desired relative error bounds are stated and proven as meta-theorems in this paper.
2. Use one of several custom algorithms for deciding the inequalities (1). These algorithms are optimized for efficient symbolic computation; though $M_1, \ldots, M_n$ have tractable representations, the product typically does not, so directly computing the product and checking the inequality can be prohibitively expensive. The other conditions involving integers are simple enough that they don't require any specially optimized algorithms to decide symbolically.

The general technique of reducing problems involving floating point numbers to problems involving integers is well-known, and for example has been used to find test vectors for floating point units where the outputs are very close to rounding boundaries [11]. The chief novelty of this paper is the specific recipes for reducing the relative error specifications of three families of floating point operations—`RCP`, `RSQRT` and `EXP2`—to a form that can be proved by symbolic computation techniques.

The primary contribution of the paper presents these novel reductions and demonstrates how they can be integrated with standard symbolic simulation tools for RTL. This facilitates formal verification of relative error bounds for our three instruction classes on a next-generation Intel® processor. This is the first verification approach for relative error bounds that uses symbolic simulation instead of theorem proving, which offers the advantage of providing counter-examples whenever the verification fails, shortening debugging time. A secondary contribution of this paper is the technique for verifying the relative error bounds of the `EXP2` floating point operation, which computes an approximation to 2^x for an input x. We present a recipe for verifying bounds of this transcendental function using symbolic arithmetic operations.

The rest of the paper is organized as follows. Background notions and notations are given in Sect. 2. The lemmas that reduce the relative error specifications to integer reasoning are give in Sect. 3. Sect. 4 presents the three symbolic decision procedures for (1). Our case study results, paper summary, and a discussion of related work correspond to Sects. 5, 6, and 7, respectively.

2 Background

2.1 Relative Error

Let $\mathbb{B}$, $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{R}$ represent the set of booleans $\{0,1\}$, naturals $\{0,1,2,\ldots\}$, integers, and reals, respectively. For any $x \in \mathbb{R}$, we use the usual floor and ceiling operations that map $x \in \mathbb{R}$ to $\mathbb{Z}$: $\lfloor x \rfloor$ is the maximum integer n such that $n \leq x$ and $\lceil x \rceil$ is the minimum integer n such that $n \geq x$. We also define $\langle x \rangle = x - \lfloor x \rfloor$, i.e. $\langle x \rangle$ is the "fractional" part of x.

Let y and y^* be reals; here y can be thought of as a mathematically precise result, whereas y^* is an approximation of y. For real $\epsilon > 0$, we say that y^* *approximates* y *with relative error* ϵ if

$$\left| \frac{y^* - y}{y} \right| < \epsilon$$

For a natural $p \geq 2$, we use the notation $y^* \approx_p y$ to assert that y^* approximates y with relative error 2^{-p}. We make the assumption that $p \geq 2$, and hence the relative error is at most $\frac{1}{4}$, to rule out some pathological cases in our proofs.[2] In this paper we will be interested in establishing

$$\forall x.\ h(x) \approx_p f(x)$$

where $h(x)$ is the output of a hardware design given input x, and f is the mathematical function that the hardware is designed to approximate.

2.2 Floating Point Numbers

A *floating point number* [5], or simply *float*, is a triple (s, e, m) where $s \in \{-1, 1\}$ is called the *sign*, $e \in \mathbb{Z}$ is called the *exponent*, and $m \in \mathbb{N}$ is called the *mantissa*.[3] The mantissa must satisfy a range constraint $2^\ell \leq m < 2^{\ell+1}$ where $\ell \in \mathbb{N}$ is a constant called the *mantissa fraction length*.[4] In this paper we will deal with *single precision* and *double precision* floats, which have $\ell = 23$ and $\ell = 52$, respectively. If x is a floating point number, we write $s(x)$, $e(x)$, and $m(x)$ for the sign, exponent, and mantissa of x, respectively. The real number represented by x is defined to be

$$s(x)m(x)2^{e(x)-\ell}$$

and in a minor abuse of notation we will use x and the represented real interchangeably.

---

[2] The relative errors used in our hardware verification case studies have $p \in \{11, 14, 23, 28\}$.

[3] Here we abstract slightly away from bit-level floating point encodings, e.g. as defined in IEEE Standard 754 [7].

[4] In practice e also satisfies a range constraint $e_{min} \leq e \leq e_{max}$, where e_{min} and e_{max} are maximal and minimal exponents. However, the results in this paper do not depend on exponent range constraints and so we omit them.

2.3 Symbolic Simulation

Let V be a finite set of boolean-valued variables. An *assignment* (to V) is a function $\alpha : V \to \mathbb{B}$. For any set S (which we'll call the *base type*), a function of type $(V \to \mathbb{B}) \to S$ is called a *symbolic* S; if S is unspecified we will simply refer to this as a *symbolic object*. Thus a symbolic S is a function that takes an assignment and produces an element of the base type S. In this paper we will be interested in symbolic booleans (a.k.a *boolean functions*), symbolic integers/naturals, and symbolic floats. To represent and manipulate boolean functions we will use the well-known binary decision diagram (BDD) [2] data structure. One can then represent a symbolic integer b using a finite list of boolean functions $b_n, \ldots, b_0$ and twos-complement encoding; i.e. for an assignment α,

$$b(\alpha) \;=\; -b_n(\alpha)2^n + b_{n-1}(\alpha)2^{n-1} + \cdots + b_0(\alpha)2^0$$

Once equipped with symbolic integers, we can represent symbolic floats as (s, e, m), where s is a boolean function indicating the sign, and e and m are symbolic integers. Furthermore, any function involving the various base types of interest can be extended to take and return symbolic objects. In code, this typically involves simply replacing primitive operations with symbolic variants. One fundamental operation that we will use symbolically is if-then-else, explained as follows. Let X_i and X_e be symbolic objects having the same base type, and let c be a boolean function. Then we define

$$\mathbf{ite}(c, X_i, X_e) = \lambda\alpha.\ \mathbf{if}\ c(\alpha)\ \mathbf{then}\ X_i(\alpha)\ \mathbf{else}\ X_e(\alpha)$$

For the rest of the paper, we assume availability of symbolic variants of other fundamental operations, such as addition, subtraction, multiplication, exponentiation, and constants, and will not notationally distinguish the symbolic from the non-symbolic operations.

Symbolic simulation is a well-known approach wherein symbolic objects are propagated through the primitives of a hardware (or software) design [4]. In this paper we employ BDD-based symbolic simulation, e.g. [15]. Here, a hardware description language representation of the design is compiled down to a gate-level implementation, which operates on wires carrying boolean values. Roughly, symbolic simulation involves associating to each input wire a unique boolean variable from V (represented by a BDD), and then propagating the symbolic booleans through the gates according to the gate's function. Symbolic simulation proper completes when the resulting BDDs on the output wires of interest have been computed. These output BDDs are then fed into a specification-checking phase that either proves correctness or returns a counter-example in the form of an assignment to V. In the framework in which we did our work, the specification refers to inputs and output being naturals, integers, or floats; i.e. the BDDs seen by the symbolic simulator are packaged into symbolic objects before evaluating the specification. Hence, even though symbolic simulation works on a "bit-blasted", gate-level representation, we can meaningfully construct a specification that relates the input float to the output float (or other type, as appropriate).

3 Bounded Product Reduction

In this section we present the meta-theorems needed to reduce the relative error verification problem for `RCP` (Sect. 3.1), `RSQRT` (Sect. 3.2), and `EXP2` (Sect. 3.3) to integer inequalities. Note that the theorem (and proof) for the first two are quite similar, though sufficiently different as to warrant separate theorems, whereas the reduction for `EXP2` is somewhat more elaborate. Though the reduction for `RSQRT` involves reasoning about irrational numbers, these can be eliminated by squaring; however the irrationality of `EXP2` cannot be disposed of in such an easy manner and requires more sophisticated techniques.

3.1 Reciprocal

Suppose we wish to establish $y \approx_p 1/x$, where x and y are floating point numbers. To reduce the problem to purely integer reasoning, we invoke the following key lemma.

Lemma 1 (Reduction for `RCP`).
Let x and y be floating point numbers. Then we have $y \approx_p 1/x$ if and only if all of the following three conditions hold:

$$\begin{aligned}
&\textit{(i)}\ s(x) = s(y)\\
&\textit{(ii)}\ e(x) + e(y) \in \{-2, -1, 0\}\\
&\textit{(iii)}\ 2^{2\ell+2} - 2^{2\ell+2-p} \ <\ m(x)m(y)2^{e(x)+e(y)+2} \ <\ 2^{2\ell+2} + 2^{2\ell+2-p}
\end{aligned}$$

Proof. We have

$$\begin{aligned}
& y \approx_p 1/x\\
\Leftrightarrow\ & |xy - 1| < 2^{-p}\\
\Leftrightarrow\ & \left|\left(s(x)m(x)2^{e(x)-\ell}\right)\left(s(y)m(y)2^{e(y)-\ell}\right) - 1\right| < 2^{-p}\\
\Leftrightarrow\ & \left|s(x)s(y)m(x)m(y)2^{-2\ell}2^{e(x)+e(y)} - 1\right| < 2^{-p}\\
\Leftrightarrow\ & 1 - 2^{-p} < s(x)s(y)m(x)m(y)2^{-2\ell}2^{e(x)+e(y)} < 1 + 2^{-p}\\
\Leftrightarrow\ & 2^{2\ell+2}(1 - 2^{-p}) < s(x)s(y)m(x)m(y)2^{e(x)+e(y)+2} < 2^{2\ell+2}(1 + 2^{-p})
\end{aligned}$$

which is equivalent to

$$2^{2\ell+2} - 2^{2\ell+2-p} < s(x)s(y)m(x)m(y)2^{e(x)+e(y)+2} < 2^{2\ell+2} + 2^{2\ell+2-p} \qquad (2)$$

Since $2^{2\ell+2} - 2^{2\ell+2-p}$ is positive, we must have $s(x) = s(y)$. Thus, since $s(x)s(y) = 1$, the above is equivalent to Condition (iii) of the lemma statement. Also, from the definition of floating point number, we have $2^{2\ell} \le m(x)m(y) < 2^{2\ell+2}$. If $e(x) + e(y) \le -3$, then

$$m(x)m(y)2^{e(x)+e(y)+2} \ \le\ m(x)m(y)/2 \ <\ 2^{2\ell+1} \ \le\ 2^{2\ell+2} - 2^{2\ell+2-p}$$

(since p is a positive integer), contradicting the lower bound of (2). Similarly, if $e(x) + e(y) \ge 1$, then

$$m(x)m(y)2^{e(x)+e(y)+2} \ \ge\ m(x)m(y)2^3 \ >\ 2^{2\ell+3} \ >\ 2^{2\ell+2} + 2^{2\ell+2-p}$$

which violates the upper bound of (2). □

Conditions (i) and (ii) of Lemma 1 clearly only involve integers; furthermore, assuming $p \leq 2\ell + 2$, so too does (iii).[5] Hence, we have reduced $y \approx_p 1/x$ to two instances of (1) with $n = 2$, where $M_1 = m(x)2^{e(x)+e(y)+2}$ and $M_2 = m(y)$, and the bound $B = 2^{2\ell+2}(1 - 2^{-p})$ (resp. $B = 2^{2\ell+2}(1 + 2^{-p})$) in the first (resp. second) instance. Note that we choose to multiply $m(x)$ by $2^{e(x)+e(y)+2}$ to create M_1, rather than have $n = 3$. The BDD complexity introduced by multiplying $m(x)$ by $2^{e(x)+e(y)+2}$ is relatively insignificant, since under condition (ii) the latter ranges over just $\{1, 2, 4\}$.

3.2 Reciprocal Square Root

Reciprocal square root involves a similar derivation as reciprocal, except we can disregard the sign, since the operation is only defined on non-negative floats.

Lemma 2 (Reduction for `RSQRT`). *Let x and y be positive floating point numbers. Then we have $y \approx_p 1/\sqrt{x}$ if and only if both of the following conditions hold:*

$$\begin{aligned} &\textit{(i)}\ -3 \leq e(x) + 2e(y) \leq 0 \\ &\textit{(ii)}\ 2^{3\ell+3} - 2^{3\ell+4-p} + 2^{3\ell+3-2p} \ < \ m(x)m(y)^2 2^{e(x)+2e(y)+3} \\ &\qquad\qquad\qquad\qquad\qquad\qquad < \ 2^{3\ell+3} + 2^{3\ell+4-p} + 2^{3\ell+3-2p} \end{aligned}$$

Proof. We have

$$\begin{aligned} & y \approx_p 1/\sqrt{x} \\ \Leftrightarrow\ & |y\sqrt{x} - 1| < 2^{-p} \\ \Leftrightarrow\ & -2^{-p} < y\sqrt{x} - 1 < 2^{-p} \\ \Leftrightarrow\ & 1 - 2^{-p} < y\sqrt{x} < 1 + 2^{-p} \\ \Leftrightarrow\ & (1 - 2^{-p})^2 < xy^2 < (1 + 2^{-p})^2 \\ \Leftrightarrow\ & (1 - 2^{-p})^2 < \left(m(x)2^{e(x)-\ell}\right)\left(m(y)2^{e(y)-\ell}\right)^2 < (1 + 2^{-p})^2 \\ \Leftrightarrow\ & (1 - 2^{-p})^2 < m(x)m(y)^2 2^{e(x)+2e(y)-3\ell} < (1 + 2^{-p})^2 \\ \Leftrightarrow\ & 2^{3\ell+3}(1 - 2^{-p})^2 < m(x)m(y)^2 2^{e(x)+2e(y)+3} < 2^{3\ell+3}(1 + 2^{-p})^2 \\ \Leftrightarrow\ & 2^{3\ell+3} - 2^{3\ell+4-p} + 2^{3\ell+3-2p} < m(x)m(y)^2 2^{e(x)+2e(y)+3} \\ & \qquad\qquad\qquad\qquad\qquad\qquad < 2^{3\ell+3} + 2^{3\ell+4-p} + 2^{3\ell+3-2p} \end{aligned}$$

Note that since $p \geq 2$, we have $2^{3\ell+2} < 2^{3\ell+3} - 2^{3\ell+4-p} + 2^{3\ell+3-2p}$ and $2^{3\ell+3} + 2^{3\ell+4-p} + 2^{3\ell+3-2p} < 2^{3\ell+4}$, and from the definition of floating point number, we have $2^{3\ell} \leq m(x)m(y)^2 < 2^{3\ell+3}$. If $e(x)+2e(y) \leq -4$, the we get the contradiction

$$2^{3\ell+2} \ < \ m(x)m(y)^2 2^{e(x)+2e(y)+3} \ \leq \ m(x)m(y)^2 2^{-1} \ < \ 2^{3\ell+2}$$

If $e(x) + 2e(y) \geq 1$, then we get the contradiction

$$2^{3\ell+4} \ > \ m(x)m(y)^2 2^{e(x)+2e(y)+3} \ \geq \ m(x)m(y)^2 2^4 \ \geq \ 2^{3\ell+4}$$

□

[5] In all our hardware verification case studies we have $p \leq 2\ell + 2$, however if this does not hold, one need only multiply all three quantities by $2^{p-2\ell-2}$ to obtain integers.

3.3 Power-of-Two

In this section, we consider relative error bounds for an instruction `EXP2` that takes an input x and returns an approximation of 2^x. Unlike the preceding instructions, `EXP2` does not take a floating point number as input, but rather a *fixed point* number. A *fixed point number with precision q* is a real number x such that $x2^q \in \mathbb{Z}$. Though `EXP2` takes a fixed point number as input, it produces a floating point number as output.

Lemma 3. *Let x be a fixed point number with precision q and let y be a positive floating point number, and let $p \geq 2$ be an integer. Then we have $y \approx_p 2^x$ if and only if both of the following conditions hold:*

$$\begin{aligned} &\textit{(i)}\ e(y) - \lfloor x \rfloor \in \{-1, 0, 1\} \\ &\textit{(ii)}\ 2^{\langle x \rangle} 2^\ell (2^p - 1) \ < \ m(y) 2^{p + e(y) - \lfloor x \rfloor} \ < \ 2^{\langle x \rangle} 2^\ell (2^p + 1) \end{aligned}$$

Proof. Letting $d = e(y) - \lfloor x \rfloor$, we have

$$\begin{aligned} & y \approx_p 2^x \\ \Leftrightarrow\ & \left| y 2^{-x} - 1 \right| < 2^{-p} \\ \Leftrightarrow\ & \left| m(y) 2^{e(y) - \ell - \lfloor x \rfloor - \langle x \rangle} - 1 \right| < 2^{-p} \\ \Leftrightarrow\ & \left| m(y) 2^{-\ell} 2^{-\langle x \rangle} 2^d - 1 \right| < 2^{-p} \\ \Leftrightarrow\ & 1 - 2^p < m(y) 2^{-\ell} 2^{-\langle x \rangle} 2^d < 1 + 2^{-p} \\ \Leftrightarrow\ & 2^{\langle x \rangle} 2^\ell (2^p - 1) < m(y) 2^{p+d} < 2^{\langle x \rangle} 2^\ell (2^p + 1) \end{aligned}$$

Since $0 \leq \langle x \rangle < 1$, we have $\frac{1}{2} < 2^{-\langle x \rangle} \leq 1$; we also have $1 \leq m(y) 2^{-\ell} < 2$. Thus,

$$\tfrac{1}{2} \ < \ m(y) 2^{-\ell} 2^{-\langle x \rangle} \ < \ 2$$

and therefore if $2^d \leq \frac{1}{4}$ or $4 \leq 2^d$, the left-hand side of the inequality becomes strictly greater than $\frac{1}{2}$, and thus the inequality cannot hold since the right-hand side is less than or equal to $\frac{1}{2}$. Thus $d \in \{-1, 0, 1\}$. □

All quantities involved in the inequalities *(ii)* above are integers, *except* the value $2^{\langle x \rangle}$, which in general is an irrational in $[1, 2)$. Hence we cannot hope to simply scale all values by some power of two to make an equi-satisfiable integer inequality, as was done in Lemmas 1 and 2. However, if we are equipped with a means of computing $\left\lfloor 2^k 2^{\langle x \rangle} \right\rfloor$ and $\left\lceil 2^k 2^{\langle x \rangle} \right\rceil$ precisely, for any $k \in \mathbb{N}$, we can still obtain an equivalent computable inequality. This is afforded by the following lemma.

Lemma 4. *Let r be a real and m and n be naturals. Then $rm < n$ (resp. $n < rm$) if and only if there exists some natural k such that $\left\lceil r2^k \right\rceil m < n2^k$ (resp. $n2^k < \left\lfloor r2^k \right\rfloor m$)*

Proof. The $\Leftarrow$ direction is easy. For the $\Rightarrow$ direction, suppose $rm < n$. Then $rm + q = n$ for some real $q > 0$, and thus $r + q/m = n/m$. Choose k such that $2^{-k} < q/m$. Then $n = rm + q > rm + m2^{-k}$, and thus $n2^k > r2^k m + m = (r2^k + 1)m > \left\lceil r2^k \right\rceil m$. The respective statement is proven analogously. □

We now exploit Lemma 4 to create a "computable" version of Lemma 3:

Lemma 5. *Let x be a fixed point number with precision q, let y be a positive floating point number, and let $p > 0$ be an integer. Then we have $y \approx_p 2^x$ if and only if $e(y) - \lfloor x \rfloor \in \{-1, 0, 1\}$ and there exists natural k such that*

$$\left\lceil 2^{k+\langle x \rangle} \right\rceil 2^{\ell}(2^p - 1) \;<\; m(y)2^{k+p+e(y)-\lfloor x \rfloor} \;<\; \left\lfloor 2^{k+\langle x \rangle} \right\rfloor 2^{\ell}(2^p + 1) \qquad (3)$$

Proof. Follows from Lemmas 3 and 4 □

Although the condition (3) from Lemma 5 only involves integers, it still requires a means of symbolically computing $\lceil 2^{k+\langle x \rangle} \rceil$ and $\lfloor 2^{k+\langle x \rangle} \rfloor$. Such computations are possible, however we chose to merely compute upper- and lower-bounds, respectively, on these two quantities. We now elaborate on this scheme.

Observe that since x is a fixed-point number with precision q, we have that $\langle x \rangle = \sum_{i=1}^{q} x_i 2^{-i}$, where $x_i \in \mathbb{B}$, and hence

$$2^{k+\langle x \rangle} \;=\; 2^k \prod_{i=1}^{q} 2^{x_i 2^{-i}}$$

Now let us suppose we have a pair of functions $\mathit{sqrt2L}, \mathit{sqrt2U} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that for all $n, i \in \mathbb{N}$ we have $\mathit{sqrt2L}(n, i) \leq 2^{n+2^{-i}} \leq \mathit{sqrt2U}(n, i)$. Here we may think of n as a bit-precision used to approximate the 2^ith-root of 2. Taking $k = nq$ and replacing the exponent x_i with an **ite** operator yields

$$\begin{array}{rcl} \left\lfloor 2^{nq+\langle x \rangle} \right\rfloor & \geq & \prod_{i=1}^{q} \mathbf{ite}(x_i, \mathit{sqrt2L}(n, i), 2^n) \\ \left\lceil 2^{nq+\langle x \rangle} \right\rceil & \leq & \prod_{i=1}^{q} \mathbf{ite}(x_i, \mathit{sqrt2U}(n, i), 2^n) \end{array} \qquad (4)$$

The introduction of the ceiling and floor operators on the LHSs of (4) are justified since the RHSs are naturals. Condition (3) is hence implied by

$$\begin{array}{rcl} 2^{\ell}(2^p - 1) \prod_{i=1}^{q} \mathbf{ite}(x_i, \mathit{sqrt2U}(n, i), 2^n) & < & m(y)2^{k+p+e(y)-\lfloor x \rfloor} \\ 2^{\ell}(2^p + 1) \prod_{i=1}^{q} \mathbf{ite}(x_i, \mathit{sqrt2L}(n, i), 2^n) & > & m(y)2^{k+p+e(y)-\lfloor x \rfloor} \end{array} \qquad (5)$$

We have obtained adequate functions for *sqrt2L* and *sqrt2U* via some straightforward modifications of a pre-existing function that performs (floor of) square-root on symbolic naturals. Therefore, when verifying `EXP2`, we need only decide inequalities of the form (5), with the "precision" parameter n selected large enough for the verification to succeed.

4 Deciding Symbolic Product Inequalities

Section 3 showed how the relative error specification for `RCP`, `RSQRT`, and `EXP2` can be reduced to two inequalities of the form (1): $B \;\langle\rangle\; \prod_{i=1}^{n} M_i$, where $\langle\rangle$ is either $<$ or $>$ and each $M_i \in \mathbb{N}$. In this section we describe three algorithms for deciding symbolic inequalities of this form. Technically, these algorithms

return the symbolic boolean characterizing the space of assignments for which the inequality holds; verification is successful iff this is the constant function *True*. Let us abbreviate $\prod_{i=1}^{n} M_i$ by Π, and let us refer to our problem as $<$-*bounding* (resp. $>$-*bounding*) when $\lessgtr$ is $<$ (resp. $>$).

A common feature of the three algorithms is that all involve a loop that iteratively computes closer and closer approximations $a_0, a_1, \ldots$ of Π. When $<$-bounding, this sequence is such that for all i, $a_i \leq a_{i+1} \leq \Pi$; thus if we reach an i such that $B < a_i$, we have proven $B < \Pi$. The analogous statement with all inequalities reversed holds for $>$-bounding. Let sat_i denote the symbolic boolean $B \lessgtr a_i$. Assuming it exists, let $v \in \mathbb{N}$ be minimal such that $sat_v = \mathit{True}$. Clearly, after iteration v the algorithm can safely return *True*. Furthermore, if $a_v \neq \Pi$, we have proven the bound without computing Π exactly. This *early termination* saves significant time and space, since v can be much smaller than the total number of iterations the algorithm would otherwise execute, and the BDD sizes in the representation of a_v are much smaller than that of Π.

Since the sequence $a_0, a_1, \ldots$ is monotonic, so too is $sat_0, sat_1, \ldots$, in the sense that $sat_i \Rightarrow sat_{i+1}$ for all i. Let $u \in \mathbb{N}$ be minimal such that $sat_u \neq \mathit{False}$. Typically, u is somewhat smaller than v, which implies there are iterations i wherein $\mathit{False} \neq sat_i \neq \mathit{True}$ (i.e., sat_i is a non-constant boolean function). This reveals a certain redundancy in these later iterations; even though we have completed the proof for the space sat_i, we continue to do computationally complex operations to go from a_i to a_{i+1}, which implicitly involve *all* assignments. We hence investigated the use of an optimization called *sat-space restriction* (SSR), in which, at the end of the ith iteration, we replace a_i with $\mathbf{ite}(sat_i, 0, a_i)$. SSR thus zeros out the approximations a_i in the space wherein the bound is already established. Our intuition suggests that SSR might be an impactful optimization, but its efficacy is an empirical question. Experiments have confirmed that it is indeed useful. For instance, for single precision `RCP` with $p = 28$ and using the algorithm of Sect. 4.3, computing the relative error specification took 8,771 and 7,038 seconds when SSR was off and on respectively, giving a 20% runtime improvement.[6]

The SSR optimization also improves the robustness of our symbolic product algorithms in the presence of hardware bugs, which can cause many more iterations of the Π-approximating loop. SSR ensures that the extra iterations only perform symbolic computations within the space of the buggy inputs. As an extreme example, if this space contains a single input vector, then the extra iterations will involve BDDs that are either constants or the minterm corresponding to the buggy input, and hence are immune to blow-up.

An orthogonal optimization to SSR is *truncation* (Tr), which involves truncating a certain number of lower order "bits" from each a_i. For a natural t and symbolic natural a, define $\mathit{truncL}_t(a) = 2^t \lfloor a2^{-t} \rfloor$ and $\mathit{truncU}_t(a) = 2^t(\lfloor a2^{-t} \rfloor + 1)$. Clearly $\mathit{truncL}_t(a) \leq a \leq \mathit{truncU}_t(a)$; and we may safely apply *truncL* (resp. *truncU*) when $<$-bounding (resp. $>$-bounding). Truncating can be useful, since the lower order t BDDs in intermediate computations might

[6] These results were averaged over 3 runs

introduce significant complexity, while negligibly contributing to the magnitude of the value.

We now present the three algorithms we use for deciding (1).

4.1 Brute Force

This algorithm does full symbolic multiplications, but can apply Tr on intermediate results. In terms of the above characterization, the approximation sequence is degenerate and has just the single element $a_0 = b_n$, where $b_0 = 1$ and $b_{i+1} = b'_i M_i$ and b'_i is either $truncL_t(b_i)$ or $truncU_t(b_i)$ for $<$-bounding or $>$-bounding, respectively. We then simply symbolically evaluate and return $B \mathrel{<\!\!>} b_n$. We call this *brute force* since the individual multiplications $b'_i M_i$ are done with an off-the-shelf symbolic multiplication algorithm that is oblivious to the fact that we only wish to *bound* the final product. This is not the case for the next two algorithms, wherein multiplication is aware of B and $\mathrel{<\!\!>}$.

4.2 Partial Product Summation

The partial product summation is only used when $n = 2$; we will write x and y for M_1 and M_2, respectively. Let y_i be the the ith "bit" of the symbolic natural y, i.e. $y = \sum_{i=1}^{r} y_i 2^i$ where r is selected to be large enough to accommodate all values in y's range. The approximations $a_0, a_1, \ldots$ are based on the "partial product" expansion $xy = \sum_{j=0}^{r} y_j x 2^j$. In particular, a_i involves summing the first $i+1$ terms of this expansion, and replacing the remaining terms by a (symbolically simpler) natural ϕ_i.

$$a_i \;=\; \phi_i + \sum_{j=r-i}^{r} y_j x 2^j$$

When $<$-bounding, we simply use $\phi_i = 0$; while for $>$-bounding, $\phi_i = x2^{r-i}$.[7]

Fig. 1 depicts the algorithmic expression of the $>$-bounding partial product summation. The approximation a_i is computed on line 6; this is separate from *acc*, which is simply the sum of the first $i+1$ term of the partial product summation. Line 7 updates the *sat* space, handling the final iteration (wherein $acc = xy$, but is typically not reached) with a special case. Lines 8-10 check for and do early termination, which invariably happens in our case studies that use this algorithm. Lines 11 and 12 are the optional Tr and SSR optimizations, respectively.

4.3 Polynomial Expansion

Though this approach can be generalized for any n, we only use it for `RCP` and `RSQRT`, and hence $n \in \{2, 3\}$. Here we explain the $n = 3$ case and denote our three multiplicands by x, y, and z. Let us fix a natural $b \geq 1$, and assume that

[7] One can safely tighten this slightly to $x(2^{r-i} - 1)$, but we used $x2^{r-i}$ since its representation as a symbolic natural is not more complex than that of x.

```
1: function PP_Bound_Upper(B, x, y)
2:     acc := 0
3:     sat := false
4:     for i := 0 upto r do
5:         acc := acc + ite(y_{r-i}, x2^{r-i}, 0)
6:         a := acc + x2^{r-i}
7:         sat := sat ∨ ite(i = r, B > acc, B ≥ a)
8:         if sat = True then
9:             return True
10:        end if
11:        acc := truncU_t(acc)
12:        acc := ite(sat, 0, acc)
13:    end for
14:    return sat
15: end function
```

Fig. 1: The partial product summation algorithm (>-bounding)

each of x, y, and z is representable using rb bits; i.e. each of the three symbolic naturals is in the range $[0, 2^{rb})$. Let us express x as $x = \sum_{j=0}^{r} x_j d^j$, where $d = 2^b$ and each x_i is a symbolic natural with range $\{0, \ldots, d-1\}$. Note that in the symbolic natural representation discussed in Sect. 2.3, obtaining the x_i's from x is trivial, since each x_i is represented by a "bit slice" of x. We express y and z similarly, respectively yielding $y_r, \ldots, y_0$ and $z_r, \ldots, z_0$. Our approach is based on the identity $xyz = \sum_{h,j,k} x_h y_j z_k d^{h+j+k}$, where the sum ranges over all triples $(h, j, k) \in \{0, \ldots, r\}^3$.

Let $\tau_0, \tau_1, \ldots$ be a total ordering of the triples $\{0, \ldots, r\}^3$, and let $T_i = \{\tau_j : j \leq i\}$. For <-bounds, we form a_i by simply summing the terms corresponding to the triples of T_i, which clearly is a lower bound, since each term is nonnegative.

$$a_i = \sum_{(h,j,k) \in T_i} x_h y_j z_k d^{h+j+k} \leq xyz \tag{6}$$

For >-bounds, the analogous a_i is somewhat more involved:

$$\begin{aligned} a_i &= \left(d^{r+1} - 1\right)^3 - \sum_{(h,j,k) \in T_i} \left((d-1)^3 - x_h y_j z_k\right) d^{h+j+k} \\ &\geq \left(d^{r+1} - 1\right)^3 - \sum_{h,j,k} \left((d-1)^3 - x_h y_j z_k\right) d^{h+j+k} \\ &= \sum_{h,j,k} (d-1)^3 d^{h+j+k} - \sum_{h,j,k} ((d-1)^3 - x_h y_j z_k) d^{h+j+k} \\ &= \sum_{h,j,k} \left((d-1)^3 - (d-1)^3 + x_h y_j z_k\right) d^{h+j+k} \\ &= xyz \end{aligned} \tag{7}$$

The natural choice of $\tau_0, \tau_1, \ldots$ (for either direction of bounding) is one that orders terms with higher powers of d first. In other words, whenever $h + j + k >$

$h'+j'+k'$, the triple (h,j,k) comes before (h',j',k') and triples with equal sums are ordered arbitrarily. Fig 2 gives the $<$-bounding variant of the algorithm; $>$-bounding is similar, but uses (7) instead of (6). In particular, line 3 is replaced with $a := (d^{r+1}-1)^3$, and line 6 is replaced with $a := a-((d-1)^3 - x_h y_j z_k)d^\sigma$; lines 7 and 11 are modified in the obvious way. Similar to Fig. 1, lines 11 and 12 are the optional optimizations Tr and SSR, respectively.

```
 1: function Poly_Expansion_Bound_Lower(B, x, y, z)
 2:     sat := False
 3:     a := 0
 4:     for σ := 3r downto 0 do
 5:         for all (h, j, k) ∈ ℕ³ such that h + j + k = σ do
 6:             a := a + x_h y_j z_k d^σ
 7:             sat := sat ∨ B < a
 8:             if sat = True then
 9:                 return True
10:             end if
11:             a := truncL_t(a)
12:             a := ite(sat, 0, a)
13:         end for
14:     end for
15:     return sat
16: end function
```

Fig. 2: The polynomial expansion algorithm ($<$-bounding)

5 Case Studies

Our method has been implemented in reFLect, the lazy functional language used to program Intel's *Forte* tool suite [14], and sits as a specification layer on top of the *Relational STE* [10] symbolic simulator. The design under verification was from a next-generation many-core CPU under development at Intel®. The `RCP` and `RSQRT` instructions analyzed in the paper are used as initial approximations in the implementation of division and squareroot computations; it is therefore crucial that they satisfy the specified relative error for the final result to be correct. Each core on the CPU is equipped with a SIMD unit that implements a fused-multiply-add (FMA) datapath, which computes $x + yz$ with only a single rounding for floats x, y, and z, as well as special-purpose hardware for our three approximate instruction families. The instruction classes `RCP` and `RSQRT` have instances for the three relative errors 2^{-11}, 2^{-14} and 2^{-28}; most of which are supported for both single precision (SP) and double-precision (DP) floats, while `EXP2` has only relative error 2^{-23}, but has an instance that produce each of SP and DP results. The input for the SP (resp. DP) `EXP2` flavor is a fixed-point integer with precision 24 and an 8-bit (resp. 11-bit) integer part, i.e. they fall in

the range $[-2^7, 2^7)$ (resp. $[-2^{10}, 2^{10})$). All instructions in our three classes are implemented using a similar method. Roughly, a selection of bits from the input are used to map into a instruction-specific ROM to obtain coefficients to use in a quadratic approximation. The FMA hardware is then used to perform the operations (multiplication, addition, normalization and rounding) necessary for evaluating the quadratic formula into a floating point result.

Op.	Tot. Time	Spec. Time.	Mem.	Alg.	Case split
`RCP` 11S	58	3	1.8	P	No
`RCP` 14S	103	49	1.8	P	No
`RCP` 14D	135	51	1.8	P	No
`RCP` 28S	14,972	7,038	17.4	E(4,0)	No
`RCP` 28D	2.7 days	1.3 days	3.6	E(5,0)	512-way
`RSQRT` 11S	68	4	1.8	P	No
`RSQRT` 14S	124	69	1.8	P	No
`RSQRT` 14D	139	55	1.8	P	No
`RSQRT` 28S	18,301	13,173	6.0	E(5,0)	16-way
`RSQRT` 28D	22.7 days	16.7 days	9.0	E(5,110)	1,024-way
`EXP2` 23S	72,759	63,428	2.9	B(30)	128-way
`EXP2` 23D	59,706	51,152	2.8	B(30)	128-way

Table 1: Verification Results

Table 1 gives the verification results.[8] The *Op* column gives the instruction type, along with the value of p and an indication of single precision (S) or double precision (D) floats.[9] The *Tot Time* column gives the total (wall clock) run time for the proof; the units are seconds except for the entries measured in days. *Spec Time* is the time for just computing the relative error specification; the time for symbolic simulation is not included.[10] *Mem* is the maximum virtual memory, in GB, the Forte process used during execution. *Alg* indicated which of the decision procedures from Sect. 4 was used: *B(t)* is the brute force algorithm from Sect. 4.1 with parameter t, *P* is the partial product approach from Sect. 4.2 with SSR enabled, while *E(b,t)* is the algorithm of Sect. 4.3 with SSR and parameters b and t. Some instructions require *case splitting* [1], which partitions the input space into a number of cases; the *Case split* column indicates if this was used, and if so how many cases. The case splits were obvious and involved holding

[8] All runs used the BDD variable order of sign, exponent, and then mantissa source variables.

[9] The instructions `RCP`28S and `RSQRT`28S are oddities since the minimum relative error allowed by the single precision format is 2^{-23}. The specification says to do the computation in the double precision domain, and then round to the nearest single precision. We were able to verify that the relative error bound was 2^{-22} and 2^{-23}, respectively, for these instructions.

[10] The time accounted to symbolic simulation also involves a non-negligible component for a cone-of-influence reduction.

constant some of the input bits used to index into the coefficient ROMs in the circuit. It is important to note that the multi-day runs were in reality performed by grouping the cases into 10 buckets and running them on different machines concurrently—case splits are embarrassingly parallelizable—so the real time used for even `RSQRT` 28D was just over 2 days.

The 2^{-14} flavors of `RCP` and `RSQRT` are interesting in that, unlike the others, they support *denormal* inputs and outputs. Denormal floats are very small values that have the minimum possible exponent, and have $m(x) < 2^{\ell}$. Though our theory assumes normal floats, it is still applicable to denormals since we have not assumed any lower bound on the exponent. Our specification code simply "normalizes" the float before doing the relative error check, this means that we map the denormal float (s, e, m) to $(s, e - j, m2^j)$, where $j \in \mathbb{N}$ is selected so that $2^{\ell} \leq m2^j < 2^{\ell+1}$. This operation clearly preserves the value represented by the float. This step did not introduce any significant verification complexity.

6 Summary

This paper has presented a novel technique for verifying relative error specifications using symbolic simulation, demonstrated on three operations taken from an industrial case study. For each of the three operations, the relative error specification is reduced to inequalities between products of integers, which is then symbolically evaluated using a custom procedure to avoid BDD blow-up. In addition to verifying an industry hardware design, this technique delivered additional benefits when applied in an industrial setting. We found that the ability of symbolic simulation to deliver counter-examples greatly improved communication between the verification and design teams, and as a consequence the debugging cycle was shortened.

7 Related Work

The most relevant existing work is a paper by Sawada [13] which presents a technique for verifying the relative error of approximate `RCP` and `RSQRT` instructions. The technique relies on the manual construction of a high level model of the hardware implementation, expressed in terms of bounded polynomial functions. The high level model is proved to satisfy the relative error bounds by using custom proof strategies in the ACL2 theorem prover. The advantage of this approach is that it mechanizes the high level reasoning needed to reduce the relative error specification to a form suitable for automatic analysis. Our approach currently relies on pen-and-paper meta-theorems to support this reduction, although we are confident they could be mechanized using the Goaled theorem prover integrated with Forte [10]. However, the advantage of our approach is that it works directly on the register transfer level (RTL)—there is no need to construct a high level model of its behaviour—and it can also be applied to verify the relative error bounds of `EXP2`. Sawada's paper reports results for precisions only up to $p = 14$, at which level a relative error verification of reciprocal required 13,953

seconds on a 2.93GHz processor. Our verification of RCP14 for DP float inputs required only 133 seconds (on a 3.07 GHz machine).

Another related work is a paper by Harrison [6] presenting a verification of relative error bounds for trigonometric functions implemented using software floating point operations. Although this was an interactive proof carried out using the HOL Light theorem prover, it made essential use of a custom automatic proof tactic for proving that the operations implementing the range reduction step are sufficiently accurate for every possible floating point input. This is similar to our relative error verification, although the technique presented in the paper of encoding a tailored real analysis argument as an automatic proof tactic is very different from our technique of reducing floating point numbers to integers followed by symbolic simulation using BDDs.

Our verification approach relies on performing symbolic arithmetic operations on integers represented by lists of BDDs, using a technique introduced by Minato and Somenzi [9]. The chief difficulty of performing symbolic arithmetic in this way is that the representing BDDs tend to blow up in size. For example, it was shown by Bryant [3] that any BDD representing the middle bit of a product of two symbolic integers is necessarily exponential in the number of bits of the multiplicands (regardless of the ordering of the variables). Thatchachar [18] also proves exponential bounds for RCP and square root (but does not cover RSQRT) for a general class of representations that includes BDDs. Hence a possible alternative approach that computes the "exact" RCP result and then shows that the hardware output is within the relative error would be infeasible, and our more sophisticated methods are justified.

Acknowledgement

We extend gratitude to Professor Alan Hu for agreeing to present this paper on our behalf.

References

1. M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. Formal verification using parametric representations of Boolean constraints. In *Design Automation Conference (DAC 1999)*, July 1999.
2. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, C-35(8):677–691, 1986.
3. R. E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Trans. Comput.*, 40(2):205–213, 1991.
4. J. A. Darringer. The application of program verification techniques to hardware verification. In *Proceedings of the 16th Design Automation Conference*, DAC '79, pages 375–381, Piscataway, NJ, USA, 1979. IEEE Press.
5. D. Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

6. J. Harrison. Formal verification of floating point trigonometric functions. In W. A. Hunt Jr. and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design (FMCAD 2000)*, volume 1954 of *Lecture Notes in Computer Science*, pages 254–270. Springer Berlin Heidelberg, 2000.
7. IEEE. *Standard for binary floating-point arithmetic.* ANSI/IEEE Standard 754-1985. The Institute of Electrical and Electronic Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA, 1985.
8. R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, and A. Naik. Replacing testing with formal verification in Intel® Core™ i7 processor execution engine validation. In A. Bouajjani and O. Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 414–429. Springer, 2009.
9. S.-I. Minato and F. Somenzi. Arithmetic Boolean expression manipulator using BDDs. *Formal Methods in System Design*, 10(2-3):221–242, 1997.
10. J. O'Leary, R. Kaivola, and T. Melham. Relational STE and theorem proving for formal verification of industrial circuit designs. In B. Jobstmann and S. Ray, editors, *Formal Methods in Computer-Aided Design (FMCAD 2013)*, pages 97–104. IEEE, Oct. 2013.
11. M. Parks. Number-theoretic test generation for directed rounding. *IEEE Trans. Comput.*, 49(7):651–658, July 2000.
12. V. Paruthi. Large-scale application of formal verification: From fiction to fact. In *Formal Methods in Computer-Aided Design (FMCAD 2010)*, pages 175–180, 2010.
13. J. Sawada. Automatic verification of estimate functions with polynomials of bounded functions. In *Formal Methods in Computer-Aided Design (FMCAD 2010)*, pages 151–158, 2010.
14. C. J. Seger, R. B. Jones, J. W. O'Leary, T. Melham, M. D. Aagaard, C. Barrett, and D. Syme. An industrially effective environment for formal hardware verification. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 24(9):1381–1405, 2006.
15. C.-J. H. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–189, 1995.
16. A. Slobodová, J. Davis, S. Swords, and W. A. Hunt. A flexible formal verification framework for industrial scale validation. In S. Singh, B. Jobstmann, M. Kishinevsky, and J. Brandt, editors, *MEMOCODE*, pages 89–97. IEEE, 2011.
17. D. Stewart. Formal for everyone - Challenges in achievable multicore design and verification. In G. Cabodi and S. Singh, editors, *Formal Methods in Computer-Aided Design (FMCAD 2012)*, page 186. IEEE, Oct. 2012. Slides available at `http://www.cs.utexas.edu/~hunt/FMCAD/FMCAD12/FormalForEveryone_DStewart_ARM.pdf`.
18. J. Thathachar. On the limitations of ordered representations of functions. In A. Hu and M. Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 232–243. Springer Berlin Heidelberg, 1998.