Verifying a Synthesized Implementation of IEEE-754 Floating-Point Exponential Function using HOL

Behzad Akbarpour[1], Amr T. Abdel-Hamid[2], Sofiène Tahar[3], and John Harrison[4]

[1] Computer Laboratory, University of Cambridge, England
ba265@cl.cam.ac.uk
[2] German University in Cairo, Tagamoa El-Khamis, Cairo, Egypt
amr.talaat@guc.edu.eg
[3] Concordia University, Montreal, Quebec, H3G 1M8, Canada
tahar@ece.concordia.ca
[4] Intel Corporation, Hillsboro, OR 97124, USA
johnh@ichips.intel.com

Abstract. Deep datapath and algorithm complexity have made the verification of floating-point units a very hard task. Most simulation and reachability analysis verification tools fail to verify a circuit with a deep datapath like most industrial floating-point units. Theorem proving, however, offers a better solution to handle such verification. In this paper, we have hierarchically formalized and verified a hardware implementation of the IEEE-754 table-driven floating-point exponential function algorithm using the HOL theorem prover. The high ability of abstraction in the HOL verification system allows its use for the verification task over the whole design path of the circuit, starting from gate level implementation of the circuit up to a high level mathematical specification.

1 Introduction

The verification of floating-point circuits has always been an important part of processor verification. The importance of arithmetic circuit verification was illustrated by the famous floating-point division bug in Intel's Pentium processor [1]. Floating-point algorithms are usually very complicated. They are composed of many modules where the smallest flaw in design or implementation can cause a very hard to discover bug, as happened in the Intel's case. Traditional approaches to verifying floating-point circuits are based on simulation. However, these approaches cannot exhaustively cover the input space of the circuits. A solution to these problems is one of the goals of formal methods [2] for verification of the correctness of hardware designs, sometimes just called hardware verification. With this approach, the behavior of hardware devices is described mathematically, and formal proof is used to verify that they meet rigorous specifications of intended behavior.

However, formal verification is not the golden rule in circuit testing because of some limitations. A correctness proof cannot guarantee that the real device will never malfunction; the design model of the device may be proved correct, but the hardware actually built can still behave in a way unintended by the designer (this is the case for simulation too). Wrong specification can play a major role in this, because it has been verified that the system will function as specified, but it has not been verified that it will work correctly. Defects in physical fabrication can cause this problem too. In formal verification a model of the design is verified, not the real physical implementation. Therefore, a fault in the modeling process can give false negatives (errors in the design which do not exist). Because of these limitations we can consider simulation and formal verification as complementary techniques, the methods have to play together.

Formal verification can be generally divided into two main categories [3]: reachability analysis and deductive methods. Model checkers and equivalence checkers are examples of the first approach. Many different theorem provers (as HOL [4]) have been used for deductive verification. To verify floating-point arithmetic circuits, model checkers would encounter some difficulties as noted in [5]. First, the specification languages are not powerful enough to express arithmetic properties; for arithmetic circuits, the specifications must be expressed as Boolean functions, which is not suitable for complex circuits. Second, these model checkers cannot represent arithmetic circuits efficiently in their models. It is hence to no surprise that most related work in the area of formal specification and verification of floating-point arithmetic circuits were done using theorem proving.

Formal verification methods [3] have sometimes been accused of a lack of ability to get into a whole industrial product design cycle. Working on the same design path of most electronic products, we discuss in this paper the formalization and verification of the IEEE-754 [6] table-driven exponential function in all abstraction levels of the design flow. The IEEE-754 exponential function was specified first formally by Harrison in [7]. This behavioral specification was written in a high level *while language*, and was intended mainly to be verified against a more abstract mathematical description of the exponential function [8]. Starting from this behavioral specification Bui *et al.* [9] developed an RTL (Register Transfer Level) implementation of the design using VHDL and Verilog. In a previous paper [10], Abdel-Hamid *et al.* have introduced design changes to the code produced by Bui *et al.*, to be able to verify this code. They have developed a modular specification and verified the same module, yet this modular specification failed to connect easily to the higher level algorithmic specification developed by Harrison. The goal of this work is to use formal verification in modeling and verification of the synthesized table driven exponential function gate level implementation against the higher level algorithmic model previously developed by Harrison. In this exercise, we extend Harrison's verification of the exponential function [7] performed as an error analysis between *real* and *algorithmic* levels, to *RTL* and then *gate* level; therefore, we close the gap between these levels. In contrast to [10] that reconstructed the RTL implementation and established a modular proof between the RTL and behavioral level, we propose a direct verification methodology without any changes to the lower level designs. We will explain in details how the verification of the synthesized gate level and RTL designs is linked to the algorithmic level for each and every module in the system.

In this work, we use the HOL theorem proving system [4] for specifying and verifying the floating-point design at hand. The HOL theorem prover is an interactive proof assistant for higher-order logic developed at Cambridge University by Gordon *et al.* [4]. It was explicitly designed for the formal verification of hardware, though it has also been applied to other areas including software verification and formalization of pure mathematics. To the best of our knowledge, this is the first attempt to close the verification gap between abstract mathematical specification and a synthesized gate level implementation using one single formalism and tool, namely HOL.

The organization of the paper is as follows: Section 2 gives a review on work related to the formalization and verification of floating-point algorithms and designs, some of which directly influenced our work. Section 3 describes the table-driven exponential function algorithm, which formal specification and implementation are discussed throughout this paper. Section 4 introduces our modeling and verification methodology and shows the main goal we are trying to reach. Section 5 shows the formalized specification of the exponential function in HOL. It also describes the VHDL implementation of the algorithm and introduces its HOL formalization. Section 6 describes the formal verification of the exponential function. We first describe the verification of the exponential function in the transition from the algorithmic level to RTL, using one of the building blocks, namely the floating-point multiplication. The details of the algorithmic to RTL verification of other blocks such as floating-point addition or rounding are given in Appendix A. We then describe the verification of the exponential function in the transition from RTL to gate level, using one of the primitive building blocks, namely the `n-bit Multiplier`. The details of the RTL to gate level verification of other blocks such as `n-bit Adder` and `n-bit Shifter` are given in Appendix B. Finally, conclusions are drawn in Section 7.

2 Related Work

There exist several related work in the open literature on the formalization and verification of IEEE standard based floating-point arithmetic. For instance, Barrett [11] specified parts of the IEEE-754 standard in Z, and Miner [12] formalized the IEEE-854 [13] floating-point standard in PVS. The latter was one of the earliest on the formalization of floating-point standards using theorem proving. This formal specification was then used by Miner and Leathrum [14] to verify in PVS a general class of IEEE compliant subtractive division algorithms. Carreno [15] formalized the same IEEE-854 standard in HOL. He interpreted the lexical descriptions of the standard into mathematical conditional descriptions and organized them in tables, which were then formalized in HOL.

The most related work among these efforts, however, is the one of Harrison

[16] who constructed the real numbers in HOL. He then developed in HOL a generic floating-point library [17] to define the most fundamental terms of the IEEE-754 standard and to prove the corresponding correctness analysis lemmas. He used this library to formalize and verify floating-point algorithms of complex arithmetic operations such as the square root, the exponential function [7], and the transcendental functions [18] against their abstract mathematical counterparts. He also used the floating-point library for the verification of the class of division algorithms used in the Intel IA-64 architecture [19].

In [20], Moore *et al.* verified the AMD-K5 floating-point division algorithm using the ACL2 theorem prover. Also, Russinoff [21] has developed a floating-point library for the ACL2 prover and applied it successfully to verify the floating-point multiplication, division, and square root algorithms of the AMD-K5 and AMD Athlon processors.

In most of the work above, the scope of the researchers was concentrated in two main fields: first, the formalization of the IEEE floating-point standards and the verification of their relations to the unbounded real numbers as in [16], [15], and [12]; second, the behavioral modeling of floating-point algorithms and verifying their correctness against their main mathematical models as in [7], and [18].

In [22], Leeser *et al.* verified a radix-2 square-root algorithm and its hardware implementation, used in many processors such as HP PA7200, and Intel Pentium [21]. They used theorem proving to bridge the abstraction gap between the algorithm and the implementation. The Nuprl proof development system was used for proof automation. This work discusses the proof of the above algorithm starting from RTL and progressing down to gate level implementation.

Another approach for verification is combining a theorem prover with a model checker or a simulation tool. In these approaches, theorem provers handle the high-level proof, while the low-level properties are handled by the model checker or simulation. Aagaard and Seger [23] used the Voss hardware verification system to verify the IEEE compliance of a floating-point multiplier. O'Leary *et al.* [24] reported on the specification and verification of the Intel Pentium® Pro processor's floating-point execution unit at the gate level using a combination of model-checking and theorem proving. Chen and Bryant [25] used word-level SMV to verify a floating-point adder. Cornea-Hasegan [26] used iterative approaches and mathematical proofs to verify the correctness of the IEEE floating-point square root, divide, and remainder algorithms. Compared with theorem proving, this approach is much more automatic, but still requires user guidance.

More recently, Daumas *et al.* [27] have presented a generic library for reasoning about floating-point numbers within the Coq system. This library was then used in the verification of IEEE-compliant floating-point arithmetic algorithms [28] and hardware units [29]. Berg *et al.* [30] have formally verified a theory of IEEE rounding presented in [31] using the theorem prover PVS. This theory was then used to prove the correctness of a fully IEEE compliant floating-point unit used in the VAMP processor [32]. Sawada and Gamboa [33] formally verified the correctness of a floating-point square root algorithm used in the

IBM Power4^{TM} processor. The verification was carried out with the ACL2(r) theorem prover. Kaivola *et al.* [34–36] presented the formal verification of the floating-point multiplication, division, and square root units of the Intel IA-32 Pentium® 4 microprocessor. The verification was carried out using the Forte verification framework. Both the IBM and Intel floating-point verification efforts use symbolic simulation (via ACL2 at IBM and STE (Symbolic Trajectory Evaluation) at Intel) for verification of optimized gate-level designs against clean register-transfer level models. The automation provided by symbolic simulation is a necessity to keep the amount of human effort down to a reasonable level. However, in our case, it is difficult to describe and verify mathematical circuits using automated tools except for a very limited set of the generated sub-goals, so we decide to solve all different goals interactively. Yet, the produced proof is highly modular and this would allow people to use it as a general framework and change the verification method safely for some of such sub-goals. On top of that, we want to link the correctness proof of the RTL to gate level transition, to the correctness proof of the algorithmic to RTL transition, and also to the error analysis between real and algorithmic levels, and prove a single theorem that connects the floating-point exponential function at the gate level to its abstract mathematical counterpart.

In most of these works, except for [22], hardware implementations were discussed in more details. Usually, RTL implementation was proved against predefined properties for the IEEE floating-point standard used. This may cover compatibility of the floating-point implementations under investigation to the IEEE standard, but it would not cover the correctness of the implementation against the main circuit behavioral specification. Also, it can be noticed that most of these works are either concerned with the verification of the abstract mathematical description of an IEEE floating-point standard, or is only concerned with the RTL verification against a higher behavioral specification. In this work, we will discuss the formalization and verification of the IEEE-754 table-driven exponential function in all abstraction levels of the design flow.

3 The IEEE-754 Exponential Function Algorithm

In this section, we give an introduction to the IEEE-754 exponential function algorithm which formal specification and design are discussed in the rest of the paper.

Using an approximate polynomial expansion, Tang [8] has developed an algorithm for computing the floating-point exponential function using what he calls a *table-driven* approach. In this approach, given an input argument x, exceptional cases such as NaN (not-a-number), infinities (or simply very large arguments) and zeros are dealt with first. For example, $\exp(-\infty) = +0$. Furthermore, if the argument x is small enough for this to be a satisfactory approximation, the exponential function is calculated simply as $1 + x$. The main part of the algorithm covers the remaining cases. Mathematically, the procedure is simple. First we obtain a reduced argument r such that for some integer n:

$$x = n\frac{\ln(2)}{32} + r$$

and $-\frac{\ln(2)}{64} \leq r \leq \frac{\ln(2)}{64}$. This n is found by rounding $x\frac{32}{\ln(2)}$ to the nearest integer. Now we decompose n into its quotient and remainder when divided by 32, i.e., $n = 32m + j$ with $0 \leq j \leq 31$. Hence

$$e^x = e^{(32m+j)\frac{\ln(2)}{32}+r} = e^{\ln(2)m}e^{\frac{\ln(2)j}{32}}e^r = 2^m 2^{\frac{j}{32}}e^r$$

Values of $2^{\frac{j}{32}}$ for $0 \leq j \leq 31$ are prestored constants, and multiplication by 2^m is fast. Hence we just need to calculate e^r for $r \in [-\frac{\ln(2)}{64}, \frac{\ln(2)}{64}]$. This is done by a lower-order polynomial approximation $p(r) \approx e^r - 1$, where:

$$p(r) = r + \frac{8388676}{2^{24}}r^2 + \frac{11184876}{2^{26}}r^3$$

The actual reconstruction of e^x, for reasons of accuracy, is done by:

$$e^x = 2^m(2^{\frac{j}{32}} + 2^{\frac{j}{32}}p(r))$$

In fact, in order to achieve good accuracy, the above mathematical description is complicated slightly. The value r is broken down into $r_1 + r_2$ where $r_2 \ll r_1$. Similarly the prestored constants $2^{\frac{j}{32}}$ are all stored as two separate arrays S_{lead} and S_{trail} with $2^{\frac{j}{32}} \approx S_{lead}(j) + S_{trail}(j)$ and $S_{trail}(j) \ll S_{lead}(j)$. This would avoid rounding errors as well as take care of the ordering of operations, hence making the actual code look a bit more complicated than the above mathematical description.

4 Modeling and Verification Methodology

The verification process for the table-driven floating-point exponential function will be performed on many levels. Harrison [7] formalized and verified using the HOL Light theorem prover that a behavioral specification of the IEEE-754 table-driven floating-point exponential function implies its abstract mathematical counterpart. He also performed an error analysis between these two levels. For this, he first developed theories in HOL on construction of real numbers [16], and formalization of IEEE-754 standard based floating-point arithmetic [17, 7]. Then he used valuation functions to find the real value of the floating-point exponential function output, and defined the error as the difference between this value and the corresponding output of the ideal real exponential function. Then he established fundamental lemmas on error analysis of floating-point rounding and arithmetic operations against their abstract mathematical counterparts. Finally based on these lemmas, he proved that the floating-point exponential function algorithm has the correct overflow behavior and, in absence of overflow, the error in the result is less than 0.54 units in the last place compared against the exact mathematical exponential function. He confirmed and strengthened the main results of the previously published error analysis in [8], though he uncovered a minor error in the hand proof and located a few subtle corners in the proof that a less careful worker might easily have overlooked. The error in postulated theorems was related to forgetting of special or degenerate cases in IEEE floating-point such as NaNs and negative zeros.

After handling the transition from real to floating-point levels, we move to the RTL design. At this point, we use the standard higher-order logic predicate approach to model the floating-point exponential function at the RTL, as developed by Bui *et al.* [9] using VHDL and Verilog, within the HOL environment. The last step is to verify this level using a classical hierarchical proof approach in HOL [37]. In this way, we hierarchically prove that the floating-point exponential function RTL implementation implies the high level algorithmic specification which has already been related to the ideal real specification through the error analysis. The verification can be extended in HOL, following a similar approach, down to gate level netlist implementation, machine synthesized using the Synopsys tool.

The overall modeling and verification process is described in Figure 1, where the white boxes are the material provided by [7], [9], and [8], while the shaded ones represent those developed in this work.

Let X be the input variable and E the corresponding output of the floating-point exponential function in the gate level, then our final goal is:

$$\begin{aligned} \vdash_{thm} \quad & \forall\ X\ \ E.\ FP\_EXP\_GATE\ XE \implies \\ & \quad Val\ (float\ (E)) = exp\ (Val\ (float\ (X))\ +\ Error\ XE\ \ \wedge \\ & \qquad\qquad abs\ (Error\ X\ E) \leq Bound\ X\ E \end{aligned} \tag{1}$$

Here FP_EXP_GATE is a predicate [5] describing the floating-point exponential function in gate level, and its input and output signals X and E are Boolean words. To relate these signals to the corresponding specifications in floating-point and real domains, we make use of the bijection function $float$, and the valuation function $valof$. Also, *exp* is the exponential function in real domain available in HOL transcendental functions theory (*transc*). The theorem states that the real value of the floating-point exponential function in gate level is equal to the real value of the exponential function in real domain plus an error, and also the absolute value of the error is bounded to a certain value which depends on the range of the input and output numbers. This goal cannot be reached directly, due to the very high abstraction gap between the gate and abstract mathematics levels as described above. So, the proof scheme was changed

[5] A predicate is simply a function in which you cannot distinguish between input and output variables.

Fig. 1. Floating-Point Exponential Function Specification and Verification Approach

to hierarchically prove that the gate level implies the more abstract RTL. Then this RTL was related, by a formal proof, to the behavioral specification. The latter was proved to imply the high level real specification plus the error. This can be formalized as follows in HOL:

$$\vdash_{thm} \ \ \forall\, X \ \ E.\ \ FP\_EXP\_GATE\ (X, E) \implies FP\_EXP\_RTL\ (X, E) \tag{2}$$

$$\vdash_{thm} \ \ \forall\, X \ \ E.\ \ FP\_EXP\_RTL\ (X, E) \implies FP\_EXP\_ALGORITHM\ (float\ (X), float\ (E)) \tag{3}$$

$$\begin{aligned}\vdash_{thm} \ \ \forall\, X \ \ E.\ \ & FP\_EXP\_ALGORITHM\ (float\ (X), float\ (E)) \implies \\ & valof\ (float\ (E)) = exp\ (valof\ (float\ (X))\ +\ error\ (X, E) \ \ \wedge \\ & abs\ (error\ (X, E)) \leq error\_bound\ (X, E)\end{aligned} \tag{4}$$

In these formulas, FP_EXP_RTL and $FP\_EXP\_ALGORITHM$ are predicates describing the floating-point exponential function in RTL and algorithmic levels, respectively. Note that the inputs and outputs in RTL level are still Boolean, however in algorithmic level they have floating-point type and we use the data conversion function $float$ to convert the variables from the Boolean type to IEEE-754 standard based floating-point type. Also, as can be understood from the theorems, there are no finite precision effects in transition from gate level to RTL level, and also from RTL level to algorithmic level; therefore, the corresponding correctness theorems are described as purely logical implications. However, for transition from algorithmic level to abstract mathematical real numbers domain we should consider the effects of finite precision between floating-point numbers and real numbers and conduct an error analysis to bound the corresponding error. Finally using Equations (2), (3) and (4) we can reach the final goal stated in Equation (1).

Due to the high modularity of the design, the goals of Equations (2) and (3) could be extended to sub-level modules' specification and implementation, and then the verification continues with these sub-level modules. These proofs were then composed to yield the original goals.

5 Formal Specification and Implementation of the Exponential Function

In this section we describe the formal specification and implementation of the IEEE-754 floating-point exponential function in HOL theorem prover. The verification details will be discussed in the next section.

5.1 Formal Specification of the Exponential Function

The original analysis of the floating-point exponential function in algorithmic level was performed by Harrison [7] using the HOL Light theorem prover. In this work, we ported the code from HOL Light to HOL4, Kananaskis-3. We modeled the algorithmic specification of the floating-point exponential function as a predicate in higher-order logic as follows:

```
⊢_def Int_32 = Int(32)
⊢_def Int_2e9 = Int(2 EXP 9)
⊢_def Plus_one = float(0,127,0)
⊢_def THRESHOLD_1 = float(0,134,6056890)
⊢_def THRESHOLD_2 = float(0,102,0)
⊢_def Inv_L = float(0,132,3713595)
⊢_def L1 = float(0,121,3240448)
⊢_def L2 = float(0,102,4177550)
⊢_def A1 = float(0,126,68)
⊢_def A2 = float(0,124,2796268)
⊢_def FP_EXP_ALGORITHM X E =
          ∃ R1 R2 R P Q S E1 N N1 N2 M J S_Lead S_Trail.
            TABLES_OK S_Lead S_Trail ∧
             (if Isnan X then E = X
             else (if X = Plus_infinity then E = Plus_infinity
             else (if X = Minus_infinity then E = Plus_zero
             else (if float_abs X > THRESHOLD_1 then
                  (if X > Plus_zero then E = Plus_infinity
                   else E = Plus_zero)
             else (if float_abs X < THRESHOLD_2 then E = Plus_one + X
             else
              (N = INTRND (X * Inv_L)) ∧
              (N2 = % N Int_32) ∧
              (N1 = N - N2) ∧
              (if Int_abs N ≥ Int_2e9 then
                 R1 = X - Tofloat N1 * L1 - Tofloat N2 * L1
               else
                 R1 = X - Tofloat N * L1) ∧

              (R2 = Tofloat ¬N * L2) ∧
              (M = N1 / Int_32) ∧
              (J = N2) ∧
              (R = R1 + R2) ∧
              (Q = R * R * (A1 + R * A2)) ∧
              (P = R1 + (R2 + Q)) ∧
              (S = S_Lead J + S_Trail J) ∧
              (E1 = S_Lead J + (S_Trail J + S * P)) ∧
               E = Scalb (E1,M))))))
```

where the constant TABLES_OK is used to abbreviate a large set of assumptions about the values of table entries taken from Tang's paper [8]. In addition to IEEE 754 standard single-precision format floating-point numbers, the algorithm uses the formalization of machine integers which are defined as 2's complement 32-bit integers in HOL.

Based on Tang's algorithm, the above HOL code implements the exponential function in the following four steps:

Step 1. Filter out the exceptional cases. When the input argument X is a NaN, a quite NaN should be returned. When X is $+\infty$, $+\infty$ should be returned without any exception. When X is $-\infty$, $+0$ should be returned without any exception. When the magnitude of X is larger than THRESHOLD_1, a $+\infty$ with an overflow signal, or a $+0$ with underflow and inexact signals, should be returned. When the magnitude of X is smaller than THRESHOLD_2, $1 + X$ should be returned.

Step 2. Reduce the input argument X to $[-\frac{\log 2}{64}, \frac{\log 2}{64}]$. Obtain integers M and J, and working-precision floating-point numbers R_1 and R_2 such that (up to roundoff)

$$X = (32M + J)\frac{\log 2}{32} + (R_1 + R_2),$$

$|R_1 + R_2| \leq \frac{\log 2}{64}$.

To perform the argument reduction accurately, do the following:

– Calculate N as follows:

$$\begin{aligned} N &:= \text{INTRND}(X * \text{INV\_L}) \\ N_2 &:= N \bmod 32 \\ N_1 &:= N - N_2 \end{aligned}$$

INV_L is $\frac{32}{\log 2}$ rounded to working precision. Note that $N_2 \geq 0$, regardless of N's sign. INTRND rounds a floating-point number to the nearest integer in the manner prescribed by the IEEE standard [6].

– The reduced argument is represented in two working-precision numbers, R_1 and R_2. We compute them as follows. First, the value of $\frac{\log 2}{32}$ is represented in two working-precision numbers, L_1 and L_2, such that the leading part, L_1, has a few trailing zeros and $L_1 + L_2$ approximates $\frac{\log 2}{32}$ to a precision much higher than the working one. If the single-precision exponential is requested and $|N| \geq 2^9$, then calculate R_1 by

$$R_1 := (X - N_1 * L_1) - N_2 * L_1.$$

Otherwise, calculate R_1 by

$$R_1 := (X - N * L_1).$$

R_2 is obtained by

$$R_2 := -N * L_2.$$

– To complete this step, we decompose N into M and J, thus:

$$\begin{aligned} M &:= \frac{N_1}{32} \\ J &:= N_2. \end{aligned}$$

Step 3. Approximate $\exp(R_1 + R_2) - 1$ by a polynomial $p(R_1 + R_2)$, where

$$p(t) = t + a_1t^2 + a_2t^3 + ... + a_nt^{n+1}.$$

The polynomial is computed by a standard recurrence:

$$\begin{aligned} R &:= R_1 + R_2 \\ Q &:= R * R * (A_1 + R * (A_2 + R * (... + R * A_n)...)) \\ P &:= R_1 + (R_2 + Q) \end{aligned}$$

The coefficients are obtained from a Remez algorithm implemented by Tang [8]. Our method for bounding the approximation error in this polynomial [7] is post-hoc, and works equally well if the polynomial is derived in other ways, e.g., via Chebyshev expansions [38] or more delicate means [39].

Step 4. Reconstruct $\exp(X)$ via

$$\exp(X) = 2^M(2^{\frac{j}{32}} + 2^{\frac{j}{32}}\ p(R_1 + R_2)).$$

Each of the values $2^{\frac{j}{32}}, j = 0, 1, ..., 31$, is calculated beforehand and represented by two working-precision numbers $S\_lead(J)$ and $S\_trail(J)$. The sum approximates $2^{\frac{j}{32}}$ to roughly double the working precision. Thus, we may consider $2^{\frac{j}{32}} = S\_lead(J) + S\_trail(J)$ for all practical purposes. The Reconstruction is as follows:

$$\begin{aligned} S &:= S\_lead(J) + S\_trail(J) \\ \exp &:= 2^M * (S\_lead(J) + (S\_trail(J) + S * P)) \end{aligned}$$

5.2 Formal Implementation of the Exponential Function

The implementation of the algorithm in RTL was done by Bui *et al.* [9] using two different hardware description languages, namely, Verilog and VHDL.

A block diagram of the whole system is shown in Figure 2. In this diagram, we use the same labels as in the algorithm specification.

The part constructed using VHDL made use of the sequential mode in contrast to the Verilog implementation that used combinational logic. Both essentially implement the same algorithm outlined in the previous section.

The VHDL design is composed of numerous procedures that perform IEEE-754 floating-point operations. These operations include the addition, multiplication, division by 32, rounding to the nearest integer, modulo 32, comparison and powers of 2. To ensure that the code is synthesizable, the program was made primitive and the length was much greater than it needed to be.

We modeled this implementation as a predicate in higher-order logic as follows:

Fig. 2. Floating-Point Exponential Function Main Block Diagram

```
⊢def FP_EXP_RTL xs xe xm outs oute outm =
      ∃ inv temp temp2 temp3 twoe9 flag slead strail
        n n1 n2 r1 r2 l1 l2 a1 a2 e1 m q s p r j.
      MULT1 invs xs stemp inve xe etemp invm xm mtemp ∧
      ROUND1 stemp ns etemp ne mtemp nm ∧
      MOD32 ns n2s ne n2e nm n2m ∧
      ADDER1 ns (¬n2s) n1s ne n2e n1e nm n2m n1m ∧
      COMP F twoe9s ne twoe9e nm twoe9m flag ∧
      (if flag = WORD [F; F; T] then
         MULT1 ns l1s stemp2 ne l1e etemp2 nm l1m mtemp2 ∧
         ADDER1 (¬stemp2) xs r1s etemp2 xe r1e xm mtemp2 r1m
       else
         MULT1 ns l1s stemp2 ne l1e etemp2 nm l1m mtemp2 ∧
         ADDER1 (¬stemp2) xs stemp3 etemp2 xe etemp3 mtemp2 xm mtemp3 ∧
         MULT1 stemp2 l1s r1s etemp2 l1e r1e mtemp2 l1m r1m ∧
         ADDER1 (¬n2s) stemp3 stemp2 n2e etemp3 etemp2 n2m mtemp3 mtemp2) ∧
      MULT1 (¬ns) l2s r2s ne l2e r2e nm l2m r2m ∧
      D32 n1s ms n1e me n1m mm ∧
      ADDER1 r1s r2s rs r1e r2e re r1m r2m rm ∧
      MULT1 rs a2s stemp re a2e etemp rm a2m mtemp ∧
      ADDER1 stemp a1s stemp2 etemp a1e etemp2 mtemp a1m mtemp2 ∧
      MULT1 rs rs stemp re re etemp rm rm mtemp ∧
      MULT1 stemp stemp2 qs etemp etemp2 qe mtemp mtemp2 qm ∧
      ADDER1 r2s qs stemp r2e qe etemp r2m qm mtemp ∧
      ADDER1 stemp r1s ps etemp r1e pe mtemp r1m pm ∧
      GET_J n2s n2e n2m j ∧
      TABLES_OK j sleads sleadm sleade ∧
      ADDER1 sleads strails ss sleade straile se sleadm strailm sm ∧
      MULT1 ss ps stemp se pe etemp sm pm mtemp ∧
      ADDER1 stemp strails stemp2 etemp straile se sleadm strailm sm ∧
      ADDER1 sleads stemp2 e1s sleade etemp2 e1e sleadm mtemp2 e1m ∧
      TWOPOWERM ms me mm stemp etemp mtemp ∧
      MULT1 stemp e1s outs etemp e1e oute mtemp e1m outm
```

The design is composed of numerous primitive building blocks including the addition (`ADDER1`), multiplication (`MULT1`), division by 32 (`D32`), rounding to nearest integer (`ROUND1`), modulo 32 (`MOD32`), comparison (`COMP`), powers of 2 (`TWOPOWERM`), and get J (`Get_J`), which will be explained in next section.

6 Formal Verification of the Exponential Function

In this section we describe the verification of the floating-point exponential function using HOL according to the methodology described in Section 4. We first describe the verification of the exponential function in the transition from the algorithmic level to RTL, using one of the building blocks, namely the floating-point multiplication. The details of the algorithmic to RTL verification of other blocks such as floating-point addition, division by 32, round to nearest integer, modulo 32, comparison, powers of two, and get J blocks are given in Appendix A. We then describe the verification of the exponential function in the transition from RTL to gate level, using one of the primitive building blocks, namely the `n-bit Multiplier`. The details of the RTL to gate level verification of other blocks such as `n-bit Adder`, `n-bit Subtracter`, `n-bit Concatenator`, `n-bit Multiplexer`, and `n-bit Shifter` are given in Appendix B.

6.1 Verification of RTL to Algorithmic Level

In this section we describe the algorithmic level to RTL verification of the floating-point exponential function. The whole RTL design is segmented into different blocks and then modeled using HOL. The resulting model is in turn set against the algorithmic specification and the HOL tool is used interactively to prove its correctness.

The Main Theorem We established the correctness of the RTL implementation of the floating-point exponential function against its algorithmic specification in HOL as the following main theorem:

```
Theorem 1: FP_EXP_RTL_TO_ALGORITHM_THM
⊢ FP_EXP_RTL xs xe xm outs oute outm ⟹
            FP_EXP_ALGORITHM (float (BV xs,BNVAL xe,BNVAL xm))
                             (float (BV outs,BNVAL oute,BNVAL outm))
```

where `float` is the bijection function that converts a triplet of natural numbers to the floating-point type, and `BV` and `BNVAL` are predefined functions of the HOL *word* library mapping a single bit and a Boolean word into a natural number, respectively.

As explained before, there is a high level of regularity and modularity in the design of the floating-point exponential function so that primitive blocks such as adders and multipliers are used to build the larger and complicated design. Also, the main verification goal of the whole design can be broken down to the verification proofs of the sub-level modules. These proofs are then composed to yield the original goals. Therefore the main theorem FP_EXP_RTL_TO_ALGORITHM_THM was proven in HOL using the following tactic:

```
e (REPEAT GEN_TAC THEN
   REWRITE_TAC [FP_EXP_ALGORITHM, FP_EXP_RTL]
   REPEAT STRIP_TAC THEN
   .
   .
   ARW_TAC [MULT1_RTL_TO_ALGORITHM_Correct, ADDER1_RTL_TO_ALGORITHM_Correct,
   D32_RTL_TO_ALGORITHM_Correct, ROUND1_RTL_TO_ALGORITHM_Correct,
   MOD32_RTL_TO_ALGORITHM_Correct, COMP_RTL_TO_ALGORITHM_Correct,
   TWOPOWERM_RTL_TO_ALGORITHM_Correct, GET_J_RTL_TO_ALGORITHM_Correct])
```

where lemmas such as MULT1_RTL_TO_ALGORITHM_Correct, ADDER1_RTL_TO_AL GORITHM_Correct, etc. are about the correctness of the sub-level modules which relate the RTL implementation of each module with the corresponding algorithmic specification.

In the following sections we will describe in details the verification of one of the primitive building blocks, namely the floating-point multiplication. The rest is given in Appendix A. For all the blocks described, the RTL descriptions, the corresponding HOL models, and parts of the proof strategy are provided to explain the verification in its entirely.

Verification of Floating-Point Multiplication Block Multiplication is an operation that is quite straight-forward. Its algorithm is divided into three main parts corresponding to the three parts of the single precision format. The first part, the sign, is determined by an exclusive-OR function of the two input signs. The exponent of the output, the second part, is calculated by adding the two input exponents. And finally, the significand is determined by multiplying the two input significands each with a “1” concatenated to it. The result obtained will have about twice as many bits as the significand should normally have and so, the result will be truncated, normalized and the implied “1” will be removed (see Figure 3 for the block diagram). The normalization process will be fairly simple knowing that the multiplication of two 24 bit numbers with a one at the most significant bit position will yield a result with a one at the most significant bit (bit 47) or at bit 46. Depending on the situation, the result will either be shifted once or twice. At the beginning of the algorithm, there is an IF statement that checks for exceptional cases where there is a zero in at least one of the inputs. It is important to note that this implementation of the floating-point multiplier does not handle subnormal numbers; therefore, it is not a fully fledged floating-point multiplier. It is a perfect block for the proposed exponential function, as the subnormal numbers are not allowed to reach the multiplier block in this design.

In HOL, we modeled this algorithm as follows:

```
⊢def MULT1_RTL s1 s2 s3 e1 e2 e3 m1 m2 m3 =
      ∃ imp1 imp2 count mbuff1 mbuff2 mbuff3 mbuff4 mbuff5.
        (if (BNVAL e1 = 0) ∨ (BNVAL e2 = 0) then
           (e3 = NBWORD 8 0) ∧ (m3 = NBWORD 23 0) ∧ (s3 = F)
         else
           (s3 = s1 xor s2) ∧ (mbuff3 = BNVAL e1 − 127) ∧
           (mbuff4 = BNVAL e2 − 127) ∧ (mbuff2 = mbuff3 + mbuff4) ∧
           (imp1 = WCAT (WORD [T],m1)) ∧ (imp2 = WCAT (WORD [T],m2)) ∧
           (mbuff1 = NBWORD 48 (BNVAL imp1 * BNVAL imp2)) ∧
           (if BIT 47 mbuff1 = T then count = 1 else count = 2) ∧
           (mbuff5 = SND (SHL F mbuff1 F)) ∧
           (m3 = WSEG 23 25 mbuff5) ∧
           (e3 = NBWORD 8 (mbuff2 − count + 127)))
```

where `BV` and `BNVAL` are predefined functions of the HOL *word* library mapping a single bit and a Boolean word into a natural number, respectively. `NBWORD` is the reverse function mapping a natural number into a Boolean word with a given word length. `BIT`, `WSEG`, and `WCAT` are the basic constants denoting the functions of indexing, segmenting and concatenation of words, respectively, and `SHL` is the generic shift left operator.

Then we established the correctness of the RTL implementation of the floating-point multiplication function against its algorithmic specification in HOL as the following lemma:

```
Lemma 1: MULT1_RTL_TO_ALGORITHM_Correct
⊢ MULT1_RTL s1 s2 s3 e1 e2 e3 m1 m2 m3 ⟹
      (float (BV s3,BNVAL e3,BNVAL m3) =
       float (BV s1,BNVAL e1,BNVAL m1) * float (BV s2,BNVAL e2,BNVAL m2))
```

where `float` is the bijection function that converts a triplet of natural numbers to the floating-point type. Note that we used the conventional symbols for arithmetic operations on floating-point numbers at the algorithmic level using the operator overloading feature of HOL. The arithmetic operations on floating-point numbers are defined where they first deal with the exceptional cases, either where the arguments involve a NaN or infinity, or are invalid for other reasons (e.g., $\infty - \infty$) and generate a NaN. Apart from that, they basically just take the real value of the arguments, perform the mathematical operations using the arbitrary precision in real domain and then round the result according to the desired rounding mode. Therefore, our main task in the proof of the above mentioned theorem was to show that the result of the operation following the RTL algorithm is the best approximation to the real result. These are established in HOL as the following lemmas:

Fig. 3. Multiplication Block Diagram

```
Lemma 2:
⊢ MULT1_RTL s1 s2 s3 e1 e2 e3 m1 m2 m3 ⟹
   ((BV s3,BNVAL e3,BNVAL m3) = round float_format To_nearest
        (valof float_format (BV s1,BNVAL e1,BNVAL m1) *
                           valof float_format (BV s2,BNVAL e2,BNVAL m2)))

Lemma 3:
⊢ MULT1_RTL s1 s2 s3 e1 e2 e3 m1 m2 m3 ⟹
   ((BV s3,BNVAL e3,BNVAL m3) =
    closest (valof float_format) (λa. T) {a | is_finite float_format a}
      (valof float_format (BV s1,BNVAL e1,BNVAL m1) *
                           valof float_format (BV s2,BNVAL e2,BNVAL m2)))
```

where `round` is the floating-point rounding function, `float_format` is the floating-point format, `To_nearest` is the rounding to nearest mode, `valof` is the valuation function, `closest` is the function that picks out the best approximation to a real value from a set of floating-point numbers, and `is_finite` defines the finiteness criteria for the floating-point numbers. The proof is done by rewriting with the definition of `MULT1_RTL` and a search in the range of all finite floating-point numbers to check if the result of multiplication using this function is the closest value to the real value resulting from multiplication of the real values of two input floating-point numbers.

Following a similar approach, we have verified other building blocks of the floating-point exponential function such as floating-point addition, division by 32, round to nearest integer, modulo 32, comparison, powers of two, and get J blocks in the transition from algorithmic level to RTL using HOL. For more details, please refer to Appendix A.

6.2 Verification of Gate Level to RTL

Following similar approach to the verification of the RTL to algorithmic level as described in the previous section, we established the correctness of the gate level implementation of the floating-point exponential function against its RTL specification in HOL as the following main theorem:

```
Theorem 2: FP_EXP_GATE_LEVEL_TO_RTL_THM
⊢ FP_EXP_GATE xs xe xm outs oute outm ⟹
   FP_EXP_RTL xs xe xm outs oute outm
```

To prove this theorem, we have proved the following lemmas regarding the correctness of each module:

```
Lemma 12: MULT1_GATE_TO_RTL_Correct
⊢ MULT1_GATE s1 s2 s3 e1 e2 e3 m1 m2 m3 ⟹
   MULT1_RTL s1 s2 s3 e1 e2 e3 m1 m2 m3
Lemma 13: ADDER1_GATE_TO_RTL_Correct
⊢ ADDER1_GATE s1 s2 s3 e1 e2 e3 m1 m2 m3 ⟹
   ADDER1_RTL s1 s2 s3 e1 e2 e3 m1 m2 m3
Lemma 14: D32_GATE_TO_RTL_Correct
⊢ D32_GATE s1 s2 e1 e2 m1 m2 ⟹
   D32_RTL s1 s2 e1 e2 m1 m2
Lemma 15: ROUND1_GATE_TO_RTL_Correct
⊢ ROUND1_GATE s1 s2 e1 e2 m1 m2 ⟹
   ROUND1_RTL s1 s2 e1 e2 m1 m2
Lemma 16: MOD32_GATE_TO_RTL_Correct
⊢ MOD32_GATE s1 s2 e1 e2 m1 m2 ⟹
   MOD32_RTL s1 s2 e1 e2 m1 m2
Lemma 17: COMP_GATE_TO_RTL_Correct
⊢ COMP_GATE s1 s2 e1 e2 m1 m2 ⟹
   COMP_RTL s1 s2 e1 e2 m1 m2
Lemma 18: TWOPOWERM_GATE_TO_RTL_Correct
⊢ TWOPOWERM_GATE s1 e1 m1 s3 e3 m3 ⟹
   TWOPOWERM_RTL s1 e1 m1 s3 e3 m3
Lemma 19: GET_J_GATE_TO_RTL_Correct
⊢ GET_J_GATE s1 e1 m1 j ⟹
   GET_J_RTL s1 e1 m1 j
```

The gate level specification of the modules is very similar to their RTL specification so that they are composed of the same number of sub-modules at the lower level. As can be seen from Figures 3 to 10, there are seven main primitive building block sub-modules in these levels namely `n-bit Adder`, `n-bit Subtracter`, `n-bit Multiplier`, `n-bit Comparator`, `n-bit Concatenator`, `n-bit Multiplexer`, and `n-bit Shifter`. We use these intermediate sub-modules to cover the gap between the RTL and the gate level. In the following we will describe the details of the verification of one such sub-modules, `n-bit Multiplier`. The others are given in Appendix B.

Verification of n-bit Multiplier The `n-bit Multiplier` in RTL is specified as follows:

```
⊢def CELL_MUL_SPEC a b c p co po =
     (BV po = (if (BV (a ∧ b) + BV c + BV p < 2) then
                 (BV(a ∧ b)  + BV c + BV p)
               else
                 (BV (a ∧ b) + BV c + BV p) − 2)) ∧
     (co = ¬ (BV (a ∧ b) + BV c  + BV p  < 2))
```

```
⊢def ShiftLeFT_Spec n X Y = ∀ n. ((Y 0 = F) ∧ (Y (SUC n) = X n))
```

```
⊢def ROW_MUL_SPEC n A b C P CO PO Aout=
         ∃ c.
           (BV (PO n) = if (BV ((A n) ∧ b) + BV (C n) + BV (P n) < 2) then
                          (BV ((A n) ∧ b)  + BV (C n) + BV (P n))
                        else
                          (BV ((A n) ∧ b) + BV (C n) + BV (P n)) - 2) ∧
            ((c n) = ¬ (BV ((A n) ∧ b) + BV (C n)  + BV (P n)  < 2)) ∧
            ShiftLeFT_Spec n A Aout ∧
            ShiftLeFT_Spec n c CO ∧
            (C (SUC n) = F) ∧
            (P (SUC n) = F)

⊢def (ARRAY_MUL_spec 0 A B C P Co Po Aout =
         ROW_MUL_SPEC 0 A (B 0) C P Co Po Aout) ∧
    (ARRAY_MUL_spec (SUC n) A B C P Co Po Aout = ∃ a p c.
    ARRAY_MUL_spec n A B C P  c p a ∧
    ROW_MUL_SPEC n  a (B (SUC n)) c p Co Po Aout)

⊢def MUL_SPEC n A B C P MULout =
         ∃ Co Po Aout.
           ARRAY_MUL_spec n A B C P Co Po Aout ∧
           nadd_spec  ((2 * n) - 1) Co Po F MULout (MULout (2 * n))
```

The `n-bit Multiplier` at the gate level is implemented as follows:

```
⊢def CELL_MUL_IMP a b c p co po =
         ∃ s1.
           (and2 a b s1) ∧
           (fa_imp s1 c p po co)

⊢def ROW_MUL_IMP n A b C P CO PO Aout  =
         ∃ c.
           CELL_MUL_IMP (A (n)) b (C (n)) (P (n)) (c n) (PO (n)) ∧
           ShiftLeFT_Imp n  A Aout ∧
           ShiftLeFT_Imp n c CO ∧
           (C (SUC n) = F) ∧
           (P (SUC n) = F)

⊢def ARRAY_MUL_IMP 0 A B C P Co Po Aout =
         ROW_MUL_IMP 0 A (B 0) C P Co Po Aout) ∧
    (ARRAY_MUL_IMP (SUC n) A B C P Co Po Aout = ∃ a p c.
    ARRAY_MUL_IMP n A B C P  c p a ∧
    ROW_MUL_IMP n a (B (SUC n)) c p Co Po Aout)

⊢def MUL_IMP n A B C P MULout =
         ∃ Co Po Aout.
           ARRAY_MUL_IMP n A B C P Co Po Aout ∧
           nadd_imp  ((2 * n) - 1) Co Po F MULout (MULout (2 * n))
```

The correctness of the `n-bit Multiplier` block is proved in HOL as in the following theorems:

```
Theorem: N_MUL_GATE_LEVEL_TO_RTL_Correct
⊢ MUL_IMP n A B C P MULout ⇒  MUL_SPEC n A B C P MULout
```

This goal can be tackled by dividing it into smaller sub-goals, where every sub-goal represents the verification of one of its sub-modules. This was done by starting with verifying the cell, then the row and then the array multiplier.

Following a similar approach, we have verified other primitive building blocks of the floating-point exponential function such as `n-bit Adder`, `n-bit Subtracter`, `n-bit Concatenator`, `n-bit Multiplexer`, and `n-bit Shifter` in the transition from RTL to gate level in HOL. For more details, please refer to Appendix B.

6.3 Summary

Having proved the Theorems 1 and 2 which state the correctness of the floating-point exponential function in the transition from gate level to RTL and algorithmic levels, together with the final correctness theorem proved in [7] about the error analysis of the algorithmic level to real numbers, we can prove the following theorem which bridges the gap between gate level and ideal real numbers considering the error analysis:

```
Theorem 3: FP_EXP_GATE_LEVEL_TO_REAL_THM
⊢ FP_EXP_GATE xs xe xm outs oute outm ∧
   Finite (float (BV xs,BNVAL xe,BNVAL xm)) ∧
   exp(Val (float (BV xs,BNVAL xe,BNVAL xm)) < threshold (float_format) ⟹
   Isnormal (float (BV outs,BNVAL oute,BNVAL outm)) ∧
   abs(Val(float (BV outs,BNVAL oute,BNVAL outm)) -
   exp(Val(float (BV xs,BNVAL xe,BNVAL xm)))) <
   (&54 / &100) * Ulp(float (BV outs,BNVAL oute,BNVAL outm)) ∨
   (Isdenormal(float (BV outs,BNVAL oute,BNVAL outm)) ∨
    Iszero (float (BV outs,BNVAL oute,BNVAL outm))) ∧
   abs(Val(float (BV outs,BNVAL oute,BNVAL outm)) -
   exp(Val(float (BV outs,BNVAL oute,BNVAL outm))) <
   (&77 / &100) * Ulp(float (BV outs,BNVAL oute,BNVAL outm)))
```

This main theorem connects the floating-point exponential function at the gate level to its abstract mathematical counterpart. The specification it proves is that the function has the correct overflow behavior and, in absence of overflow, the error in the result is less than 0.54 units in the last place (`Ulp`) (0.77 if the answer is denormalized) compared against the exact mathematical exponential function. One `Ulp` is defined as the magnitude of the least significant bit of the value concerned.

7 Conclusions

Most verification and testing tools will fail short to verify a circuit with a deep datapath. The IEEE-754 table-driven exponential function with its 32 bit input and 32 bit output implementation would be considered an impossible task for exhaustive simulation. For full coverage with simulation we would have 2^{32} cases, which means that even a 2 or 3 percent coverage would take very long simulation time. Model checking techniques will not go a lot further as the deep datapath means a huge state space causing a *state space explosion* [40], making it impossible to verify such a circuit. The main module and most of its sub-modules properties cannot be covered easily with, e.g., CTL properties [40].

In this paper, we have demonstrated the use of HOL to establish a complete proof between the lower gate level and RTL implementations and the higher level algorithmic specifications previously developed by Harrison for the IEEE-754 table-driven floating-point exponential function. To establish this proof, we had to formally specify and verify many floating-point smaller modules, such as floating-point addition, and floating-point multiplication, as well as, many other primitive building blocks. The project was first defined as a two years master thesis of the second author and then completed by the first author as a half man-year postdoctoral research. The whole code was composed of nearly 5000 lines.

One of the very important advantages of the hierarchical verification lies in the fact that the change of a module or more will not mean the re-proof of the whole system. It only means the re-proof that the new module meets the same specification that the older version did. This may mean a lot for tight time-to-market in a fast moving technology like electronics. As an example, our proof can always be used with the changing technology as long as we prove that the lower modules, gates for instance, are still satisfying the same properties.

References

1. CS-013007 (1994) Statistical analysis of floating point flaw, description of the flaw. Intel White Paper, Santa Clara, USA.
2. Kropf, T. (2000) Introduction to Formal Hardware Verification, Springer-Verlag, Berlin.
3. Kern, C. and Greenstreat, M.R. (1999) Formal verification in hardware design: a survey. ACM Transactions on Design Automation of Electronic Systems, 4 (2), 123-193.
4. Gordon, M.J.C. and Melham, T.F. (1993) Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic. Cambridge University Press, Cambridge.
5. Chen, Y. (1998) Arithmetic Circuit Verification Based on Word-Level Decision Diagrams. PhD Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA.
6. IEEE STD-754 (1985) IEEE standard for binary floating-point arithmetic. The Institute of Electrical and Electronics Engineers, Inc., New York, USA.
7. Harrison, J.R. (2000) Floating-point verification in HOL light: The exponential function. Formal Methods in System Design, 16 (3), 271-305.
8. Tang, P.T.P. (1989) Table-driven implementation of the exponential function in IEEE floating-point arithmetic. ACM Transactions on Mathematical Software, 15 (2), 144-157.

9. Bui, H.T., Khalaf, B., and Tahar, S. (1999) Table-driven floating-point exponential function. Proceedings of CCECE 99, Edmonton, Alberta, 9-12 May, pp. 450-455, IEEE Canada, Dundas.
10. Abdel-Hamid, A.T., Tahar, S., and Harrison, J. (2002) Enabling hardware verification through design changes. Proceedings of ICFEM 02, Shanghai, China, 21-25 October, LNCS 2495, pp. 459-470, Springer-Verlag, Berlin.
11. Barrett, G. (1989) Formal methods applied to a floating point number system. IEEE Transactions on Software Engineering, SE-15 (5), 611-621.
12. NASA-95-tm110167 (1995) Defining the IEEE-854 floating-point standard in PVS. NASA Langley Technical Report Server, Hampton, Virginia, USA.
13. IEEE STD-854 (1987) IEEE standard for radix-independent floating-point arithmetic. The Institute of Electrical and Electronics Engineers, Inc., New York, USA.
14. Miner, P.S. and Leathrum, J.F. (1996) Verification of IEEE compliant subtractive division algorithms. Proceedings of FMCAD 96, Palo Alto, California, 6-8 November, LNCS 1166, pp. 64-78, Springer-Verlag, Berlin.
15. NASA-95-tm110189 (1995) Interpretation of IEEE-854 floating-point standard and definition in the HOL system. NASA Langley Technical Report Server, Hampton, Virginia, USA.
16. Harrison, J.R. (1994) Constructing the real numbers in HOL. Formal Methods in System Design, 5 (1/2), 35-59.
17. Harrison, J.R. (1999) A machine-checked theory of floating-point arithmetic. Proceedings of TPHOLs 99, Nice, France, 14-17 September, LNCS 1690, pp. 113-130, Springer-Verlag, Berlin.
18. Harrison, J.R. (2000) Formal verification of floating point trigonometric functions. Proceedings of FMCAD 00, Austin, Texas, USA, 1-3 November, LNCS 1954, pp. 217-233, Springer-Verlag, Berlin.
19. Harrison, J.R. (2000) Formal verification of IA-64 division algorithms. Proceedings of TPHOLs 00, Portland, Oregon, 14-18 August, LNCS 1869, pp. 234-251, Springer-Verlag, Berlin.
20. Moore, J.S., Lynch, T., and Kaufmann, M. (1998) A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating-point division algorithm. IEEE Transactions on Computers, 47 (9), 913-926.
21. Russinoff, D.M. (2000) A case study in formal verification of register-transfer logic with ACL2: the floating-point adder of the AMD Athlon processor. Proceedings of FMCAD 00, Austin, Texas, USA, 1-3 November, LNCS 1954, pp. 3-36, Springer-Verlag, Berlin.
22. Leeser, M. and O'Leary, J. (1995) Verification of a subtractive radix-2 square root algorithm and implementation. Proceedings of ICCD 95, Austin, Texas, USA, 2-4 October, pp. 526-531, IEEE Computer Society, Washington, DC, USA.
23. Aagaard, M.D. and Seger, C.-J.H. (1995) The formal verification of a pipelined double-precision IEEE floating-point multiplier. Proceedings of ICCAD 95, San Jose, California, USA, 5-9 November, pp. 7-10, IEEE Computer Society, Washington, DC, USA.
24. O'Leary, J., Zhao, X., Gerth, R., and Seger, C.-J.H. (1999) Formally verifying IEEE compliance of floating-point hardware. Intel Technology Journal, Q1, 1-14.
25. Chen, Y.A. and Bryant, R.E. (1998) Verification of floating point adders. Proceedings of CAV 98, Vancouver, BC, 28 June - 2 July, LNCS 1427, pp. 488-499, Springer-Verlag, Berlin.
26. Cornea-Hasegan, M. (1998) Proving the IEEE correctness of iterative floating-point square root, divide, and remainder algorithms. Intel Technology Journal, Q2, 1-11.

27. Daumas, M., Rideau, L., and Théry, L. (2001) A generic library for floating-point numbers and its application to exact computing. Proceedings of TPHOLs 01, Edinburgh, Scotland, 3-6 September, LNCS 2152, pp. 169-184, Springer-Verlag, Berlin.
28. Boldo, S., Daumas, M., and Théry, L. (2003) Formal proofs and computations in finite precision arithmetic. Proceedings of CALCULEMUS 03, Rome, Italy, 10-12 September, pp. 101-111.
29. Boldo, S. and Daumas, M. (2004) Properties of two's complement floating point notations. Software Tools for Technology Transfer, 5 (2-3), 237-246.
30. Berg, C. and Jacobi, C. (2001) Formal verification of the VAMP floating point unit. Proceedings of CHARME 01, Livingston, Scotland, 4-7 September, LNCS 2144, pp. 325-339, Springer-Verlag, Berlin.
31. Mueller, S.M. and Paul, W.J. (2000) Computer Architecture. Complexity and Correctness. Springer-Verlag, Berlin.
32. Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., and Paul, W.J. (2003) Instantiating uninterpreted functional units and memory system: functional verification of the VAMP. Proceedings of CHARME 03, L'Aquila, Italy, 21 - 24 October, LNCS 2860, pp. 51-65, Springer-Verlag, Berlin.
33. Sawada, J. and Gamboa, R. (2002) Mechanical verification of a square root algorithm using Taylor's theorem. Proceedings of FMCAD 02, Portland, Oregon, 6-8 November, LNCS 2517, pp. 274-291, Springer-Verlag, Berlin.
34. Kaivola, R. and Aagaard, M.D. (2000) Divider circuit verification with model checking and theorem proving. Proceedings of TPHOLs 00, Portland, Oregon, 14-18 August, LNCS 1869, pp. 338-355, Springer-Verlag, Berlin.
35. Kaivola, R. and Kohatsu, K.R. (2003) Proof engineering in the large: formal verification of Pentium® 4 floating-point divider. Software Tools for Technology Transfer, 4 (3), 323-334.
36. Kaivola, R. and Narasimhan, N. (2002) Formal verification of the Pentium® 4 floating-point multiplier. Proceedings of DATE 02, Paris, France, 4-8 March, pp. 20-27.
37. Melham, T. (1993) Higher Order Logic and Hardware Verification. Cambridge University Press, Cambridge.
38. Li, R.-C. (2004) Near Optimality of Chebyshev Interpolation For Elementary Function Computations. IEEE Transactions on Computers, 53, 678-687.
39. Brisebarre, N., Muller, J.-M., and Tisserand, A. (2006) Computing Machine-Efficient Polynomial Approximations. ACM Transactions on Mathematical Software, 32, 236-256.
40. Baier, C. and Katoen, J.-P. (2008) Principles of model checking. The MIT Press, Cambridge.

A Details of RTL to Algorithmic Verification

Verification of Addition Block Figure 4 shows the block diagram of the addition function.

The addition procedure covers both the addition and the subtraction operations. The idea is mainly the same for both but handling both cases together is an added degree of complexity. The algorithm puts both numbers to the same exponent, adds or subtracts the numbers and then normalizes. The first part of the addition procedure checks which input is greater (`onebigger`). This is especially important in cases where the inputs are of opposite signs. If the inputs

Fig. 4. Addition Block Diagram

carry the same sign, the output sign will then be the same. When the signs are different, the input with the greater magnitude will impose its sign. The next step is to denormalize both inputs and perform the addition. However, before going on to that step, “01” has to be concatenated to both numbers (`mbuff1, mbuff3`). The reason for this is that the 1 is the *implicit* 1 contained in the IEEE 754 format. The 0 is there to make sure that the carry bit is not lost. Denormalizing is done by right-shifting the smaller input by an amount determined by the difference in exponents (`Counter`). The exponent is unbiased by removing 127 (“01111111”) from its biased value (`mbuff6`). Addition is then performed normally and the last part is normalizing. It would have been more convenient to use FOR loops for denormalizing purposes but the code would have been more dense and significantly more complex. In HOL, we modeled this algorithm as follows:

```
⊢def ADDER1_RTL s1 s2 s3 e1 e2 e3 m1 m2 m3 =
    ∃ onebigger counter count mbuff1 mbuff2 mbuff3 mbuff4 mbuff5 mbuff6.
         (if BNVAL e1 > BNVAL e2 then onebigger = T
          else
            (if BNVAL e2 > BNVAL e1 then onebigger = F
             else
               (if BNVAL m1 > BNVAL m2 then onebigger = T
                else onebigger = F))) ∧
         (if s1 = s2 then
            s3 = s1
          else
            (if onebigger = T then s3 = s1 else s3 = s2)) ∧
         (if onebigger = F then
            (counter = BNVAL e2 - BNVAL e1) ∧
            (mbuff1 = WCAT (WORD [F; T],m1)) ∧
            (mbuff3 = WCAT (WORD [F; T],m2)) ∧
            (mbuff6 = BNVAL e2 - 127)
          else
            (counter = BNVAL e1 - BNVAL e2) ∧
            (mbuff1 = WCAT (WORD [F; T],m2)) ∧
            (mbuff3 = WCAT (WORD [F; T],m1)) ∧
            (mbuff6 = BNVAL e1 - 127)) ∧
            (mbuff2 = SHRN mbuff1 counter) ∧
         (if s1 = s2 then
            BNVAL mbuff4 = BNVAL mbuff2 + BNVAL mbuff3
          else
            BNVAL mbuff4 = BNVAL mbuff3 - BNVAL mbuff2) ∧


         (if BIT 24 mbuff4 = T then
            (mbuff5 = SHLN mbuff4 1) ∧ (BNVAL e3 = mbuff6 + 128) ∧
             ... ∧
         (if BIT 0 mbuff4 = T then
            (mbuff5 = SHLN mbuff4 25) ∧ (BNVAL e3 = mbuff6 + 104) ∧
          else
            (mbuff5 = WORD (REPLICATE 25 F)) ∧ (BNVAL e3 = 0) ∧
            (s3 = F) ∧ (m3 = WSEG 23 2 mbuff5))
```

where SHRN and SHLN are functions that shift right and left a Boolean word to a given number of bits, respectively. Following a similar approach to the floating-point multiplier, we proved the following lemma that checks the correctness of the RTL implementation of the floating-point addition function against its algorithmic specification in HOL.

```
Lemma 4: ADDER1_RTL_TO_ALGORITHM_Correct
⊢ ADDER1_RTL s1 s2 s3 e1 e2 e3 m1 m2 m3 ⟹
   (float (BV s3,BNVAL e3,BNVAL m3) = float (BV s1,BNVAL e1,BNVAL m1) +
                                      float (BV s2,BNVAL e2,BNVAL m2))
```

Verification of Division by 32 Block This function is only required to be used on a specific type of numbers: multiples of 32. Knowing this fact, the procedure does not need to support all possible ranges of inputs. The operations performed can be explained as follows: the algorithm will output zero if the input exponent is less than five and will simply subtract five from the exponent if it is not the case. This can be seen in Figure 5.

Fig. 5. Division by 32 Block Diagram

In HOL, we modeled this algorithm as follows:

```
⊢def D32_RTL s1 s2 e1 e2 m1 m2 =
        (s2 = s1) ∧
        (if BNVAL e1 > 131 then
           (e2 = NBWORD 8 (BNVAL e1 − 5)) ∧
           (m2 = m1)
         else
           (e2 = NBWORD 8 0) ∧ (m2 = NBWORD 23 0))
```

Then we established the correctness of the algorithmic to RTL transition of the division by 32 block in HOL as follows:

```
Lemma 5: D32_RTL_TO_ALGORITHM_Correct
⊢ D32_RTL s1 s2 e1 e2 m1 m2 ⟹ (Toint (float (BV s2,BNVAL e2,BNVAL m2)) =
                              Toint (float (BV s1,BNVAL e1,BNVAL m1)) / Int_32)
```

where `Toint` is the coercion for mapping floating-point numbers into machine integers, and "/" denotes the division operation on 2's complement 32 bit machine integers. The lemma is proved by rewriting on definitions of the division function in RTL (`D32`) and algorithmic ("/") level, and valuations on floating-point numbers and machine integers.

Verification of Round to Nearest Integer Block Figure 6 illustrates the round to nearest algorithm.

Fig. 6. Round to Nearest Integer Block Diagram

The algorithm starts by checking if the exponent is of the order of −2 or less. This would result in an output of zero. The second case is to check if the exponent is −1 in which case the output would be equal to 1. These two IF statements are for negative exponent handling since the main algorithm cannot deal with these cases.

The basic idea is to verify the bit at the 0.5 position. If the bit is set, the decimal positions are filled with zero and we add one to the resulting integer. If the bit is reset, the bits located to the right of the decimal point will be reset. To accomplish this, the input is first shifted right by a number of positions corresponding to the exponent (so that all fraction bits are shifted out). The number obtained should be an integer. This number is then incremented by one if the bit at 0.5 is set else it should be left the same.

In HOL we modeled this algorithm as follows:

```
⊢def ROUND1_RTL s1 s2 e1 e2 m1 m2 =
        ∃ mbuff1 imp1 imp2 imp3 imp4 imp5 imp6 imp7 count.
          (if BNVAL e1 < 126 then
             (e2 = NBWORD 8 0) ∧ (m2 = NBWORD 23 0) ∧ (s2 = F)
           else
             (if BNVAL e1 = 126 then
                (e2 = NBWORD 8 127) ∧ (m2 = NBWORD 23 0) ∧ (s2 = s1)
              else
                (s2 = s1) ∧ (mbuff1 = BNVAL e1 − 127) ∧
                (imp1 = WCAT (WORD [F],m1)) ∧
                (count = 23 − mbuff1) ∧
                (imp2 = SHRN imp1 (count − 1)) ∧
                (imp3 = BIT 0 imp2) ∧
                (imp4 = SHRN imp2 1) ∧
                (imp5 = NBWORD 24 (BNVAL imp4 + BV imp3)) ∧
                (if mbuff1 < 23 then
                   (if BIT (mbuff1 + 1) imp5 = T then
                      (BNVAL e2 = mbuff1 + 1 + 127) ∧
                      (imp6 = SHLN imp5 (count − 1))
                    else
                      (BNVAL e2 = mbuff1 + 127) ∧
                      (imp6 = SHLN imp5 count)) ∧
                      (m2 = WSEG 23 0 imp6)
                 else
                   (e2 = e1) ∧ (m2 = m1))))
```

Then we established the correctness of the algorithmic to RTL transition of the round to nearest integer block in HOL as follows:

```
Lemma 6: ROUND1_RTL_TO_ALGORITHM_Correct
⊢ ROUND1_RTL s1 s2 e1 e2 m1 m2 ⟹
                     (Toint (float (BV s2,BNVAL e2,BNVAL m2)) =
                                INTRND (float (BV s1,BNVAL e1,BNVAL m1)))
```

where the function INTRND is the composition of the round-to-integer-value operation on floating-point numbers (ROUNDFLOAT) and the coercion function Toint. Therefore, our main task in the proof of the above mentioned theorem was to show that the result of the rounding operation following the RTL algorithm is the best approximation to the real value of the input number. This is established in HOL as the following theorem:

```
Lemma 7:
⊢ ROUND1_RTL s1 s2 e1 e2 m1 m2 ⟹
                 ((BV s2,BNVAL e2,BNVAL m2) =
   closest (valof float_format) (λa. T) {a | is_integral float_format a}
                          (valof float_format (BV s1,BNVAL e1,BNVAL m1)))
```

where is_integral checks if a floating-point number has a finite and integer value.

Verification of Modulo 32 Block Modulo 32 is an operation that is done by simply taking the five first bits located to the left of the decimal point. The result will then be an unsigned 5-bit integer that will have to be converted to single precision format. For the block diagram, refer to Figure 7. The lower right portion of the figure shows a loop that was not actually implemented in the algorithm. It was drawn like that in order to reduce the complexity of the diagram. The variable "I" is used as a loop variable.

Fig. 7. Modulo-32 Block Diagram

The procedure is somewhat similar to that of rounding to the nearest integer. The input is first shifted right by the number of bits corresponding to the exponent. The result is then ANDed with the "11111" bit pattern in order to isolate the five bits. The conversion process checks where the first 1 is located starting from the most significant position. An exponent is then assigned accordingly and the result is shifted left to comply with the rules of normalization.

In HOL we modeled this algorithm as follows:

```
⊢def MOD32_RTL s1 s2 e1 e2 m1 m2 =
        ∃ mbuff1 imp1 imp2 imp3 imp4 imp5 imp6 count n2 n3.
          (s2 = s1) ∧ (BNVAL mbuff1 = BNVAL e1 − 127) ∧
          (imp1 = WCAT (WORD [T],m1)) ∧
```

```
(if BNVAL mbuff1 > 23 then
   (count = BNVAL mbuff1 - 23) ∧ (imp2 = SHLN imp1 count)
 else
   (count = 23 - BNVAL mbuff1) ∧ (imp2 = SHRN imp1 count)) ∧
(if BIT 7 mbuff1 = T then
   n2 = NBWORD 24 0
 else
   n2 = NBWORD 24 31 WAND imp2) ∧
(if BIT 4 n2 = T then
   (BNVAL e2 = 131) ∧ (n3 = SHLN n2 19)
 else
   (if BIT 3 n2 = T then
      (BNVAL e2 = 130) ∧ (n3 = SHLN n2 20)
    else
      (if BIT 2 n2 = T then
         (BNVAL e2 = 129) ∧ (n3 = SHLN n2 21)
       else
         (if BIT 1 n2 = T then
            (BNVAL e2 = 128) ∧ (n3 = SHLN n2 22)
          else
            (if BIT 0 n2 = T then
               (BNVAL e2 = 127) ∧ (n3 = SHLN n2 23)
             else
               (BNVAL e2 = 0) ∧
               (n3 = WORD (REPLICATE 24 F))))))) ∧
               (m2 = WSEG 23 0 n3)
```

Then we established the correctness of the algorithmic to RTL transition of the modulo 32 block in HOL as follows:

```
Lemma 8: MOD32_RTL_TO_ALGORITHM_Correct
⊢ MOD32_RTL s1 s2 e1 e2 m1 m2 ⟹
          (Toint (float (BV s1,BNVAL e1,BNVAL m1)) =
                     % (Toint (float (BV s2,BNVAL e2,BNVAL m2))) Int_32)
```

where % is the modulus operation on machine integers which always returns a positive answer whatever the sign of its arguments. The lemma is proved by rewriting on definitions of the functions MOD32 and %, and valuations on floating-point numbers and machine integers.

Verification of Comparison Block Unlike the other procedures, the comparison does not output a number in the IEEE 754 format. Instead, it outputs three bits that give a comparative indication of the size of the first input with respect to the second. If the first input is greater than the second one, then the most significant bit is set. If the second input is greater than the first, then it is the least significand bit that is set. If the two inputs are equal then the middle bit is set. Only one bit can be set at any given time. In HOL we modeled this algorithm as follows:

```
⊢def COMP_RTL s1 s2 e1 e2 m1 m2 flag =
         ∃ sign expo magn.
           (sign = WCAT (WORD [s1],WORD [s2])) ∧
           (if BNVAL e1 > BNVAL e2 then
              expo = WORD [T; F]
            else
              (if BNVAL e2 > BNVAL e1 then
                 expo = WORD [F; T]
               else
                 (expo = WORD [F; F]) ∧
                 (if BNVAL m1 > BNVAL m2 then
                    magn = WORD [T; F]
                  else
                    (if BNVAL m2 > BNVAL m1 then
                       magn = WORD [F; T]
                     else
                       magn = WORD [F; F])))) ∧
           (if sign = WORD [F; F] then
              (if expo = WORD [T; F] then
                 flag = WORD [T; F; F]
               else
                 (if expo = WORD [F; T] then
                    flag = WORD [F; F; T]
                  else
                    (if magn = WORD [T; F] then
                       flag = WORD [T; F; F]
                     else
                       (if magn = WORD [F; T] then
                          flag = WORD [F; F; T]
                        else
                          flag = WORD [F; T; F]))))
            else
              (if sign = WORD [T; T] then
                 (if expo = WORD [T; F] then
                    flag = WORD [F; F; T]
                  else
                    (if expo = WORD [F; T] then
                       flag = WORD [T; F; F]
                     else
                       (if magn = WORD [T; F] then
                          flag = WORD [F; F; T]
                        else
                          (if magn = WORD [F; T] then
                             flag = WORD [T; F; F]
                           else
                             flag = WORD [F; T; F]))))
               else
                 (if sign = WORD [T; F] then
                    flag = WORD [F; F; T]
                  else
                    flag = WORD [T; F; F])))
```

Figure 8 shows the block diagram of the comparison function.

Fig. 8. Comparison Block Diagram

The correctness of the algorithmic to RTL transition of the comparison block is established in HOL as follows:

```
Lemma 9: COMP_RTL_TO_ALGORITHM_Correct
⊢ COMP_RTL s1 s2 e1 e2 m1 m2 flag ⟹
    (if flag = WORD [T; F; F] then
       Toint (float (BV s1,BNVAL e1,BNVAL m1)) >
       Toint (float (BV s2,BNVAL e2,BNVAL m2))
     else
       (if flag = WORD [F; T; F] then
          Toint (float (BV s1,BNVAL e1,BNVAL m1)) =
          Toint (float (BV s2,BNVAL e2,BNVAL m2))
        else
          Toint (float (BV s1,BNVAL e1,BNVAL m1)) <
          Toint (float (BV s2,BNVAL e2,BNVAL m2))))
```

Note that we use conventional symbols for comparison operations on machine integers in algorithmic level using the operator overloading feature of HOL. The proof is straightforward by rewriting on definitions of the functions `COMP`, and $<, =, >$, and valuation on machine integers.

Verification of Powers of Two Block The powers of two function can be implemented by realizing that the value of the input is the value of the output exponent. For example, placing four as an input would result in two to the power of four, yielding four in the exponent field. The objective of the function would then be to convert the input, being an IEEE 754 number, to a 2's complement number. The bias of 127 would then be added to the result and the sum would be placed in the exponent field. The sign and significand fields will be filled with zeros because the result will always be positive and will always be an integer multiple of two. A detailed block digram of the Power of Two function is given in Figure 9. In this figure, there is a loop in the lower left section that is used to provide a concise description of the algorithm. It uses the variable "I" as a loop variable.

Fig. 9. Powers of Two Block Diagram

In HOL we modeled this algorithm as follows:

```
⊢def TWOPOWERM_RTL s1 e1 m1 s3 e3 m3 =
       ∃ expo magn buff buff2.
         (if e1 = WORD [F; F; F; F; F; F; F; F] then
            (e3 = WORD [F; T; T; T; T; T; T; T]) ∧
            (m3 = WORD (REPLICATE 23 F)) ∧ (s3 = F)
          else
            ((expo = NBWORD 8 (BNVAL e1 - 127)) ∧
             (magn = WCAT (WORD [F; F; F; F; F; F; F; T],m1))) ∧
            (if s1 = F then
               (if expo = WORD [F; F; F; F; F; T; T; T] then
                  buff = WSEG 8 16 magn
                else
                  (if expo = WORD [F; F; F; F; F; T; T; F] then
                     buff = WSEG 8 17 magn
                   else
                     (if ... else
                                    buff = WSEG 8 23 magn)))))))
             else
               (if expo = WORD [F; F; F; F; F; T; T; T] then
                  buff2 = WSEG 8 16 magn
                else
                  (if expo = WORD [F; F; F; F; F; T; T; F] then
                     buff2 = WSEG 8 17 magn
                   else
                     (if ... else
                                    buff2 = WSEG 8 23 magn))))))) ∧
               (buff = NBWORD 8 (0 - BNVAL buff2))) ∧ (s3 = F) ∧
            (e3 = NBWORD 8 (BNVAL buff + 127)) ∧ (m3 = NBWORD 23 0))
```

The correctness of the algorithmic to RTL transition of the powers of two block is established in HOL as follows:

```
Lemma 10: TWOPOWERM_RTL_TO_ALGORITHM_Correct
⊢ TWOPOWERM_RTL s1 e1 m1 s3 e3 m3 ⟹
   (valof float_format (BV s3,BNVAL e3,BNVAL m3) =
   exp (Ival (Toint (float (BV s1,BNVAL e1,BNVAL m1))) * ln 2))
```

where `Ival` is the valuation function on machine integers, and `exp` and `ln` are the exponential and natural logarithmic functions defined in HOL real library, respectively. The proof is done by rewriting on the definition of the function `TWOPOWERM` and valuations on floating-point numbers and machine integers.

Verification of Get J Block The current implementation of the exponential circuit is the table-driven implementation. The table index should ideally be an unsigned integer to make the search easier. The "Get J" procedure takes care of this. It takes a number in the single-precision format and transforms it to an unsigned number. The procedure examines the exponent and extracts the corresponding bits from the significand. Even though the source code uses a series of IF statements, the block diagram in Figure 10 shows a loop that uses a variable “I” to perform the required task.

Fig. 10. Get J Block Diagram

Using an unsigned number for the search makes the task of finding a correct value for S easier (refer to the algorithm described in Section 3).

In HOL we modeled this algorithm as follows:

```
⊢def GET_J_RTL s1 e1 m1 j =
        ∃ expo magn.
          (BNVAL expo = BNVAL e1 − 127) ∧
          (magn = WCAT (WORD [F; F; F; F; T],m1)) ∧
          (if expo = WORD [F; F; F; F; F; T; F; F] then
             j = WSEG 5 19 magn
           else
             (if expo = WORD [F; F; F; F; F; F; T; T] then
                j = WSEG 5 20 magn
              else
                (if expo = WORD [F; F; F; F; F; F; T; F] then
                   j = WSEG 5 21 magn
                 else
                   (if expo = WORD [F; F; F; F; F; F; F; T] then
                      j = WSEG 5 22 magn
                    else
                      (if expo = WORD [F; F; F; F; F; F; F; F] then
                         j = WSEG 5 23 magn
                       else
                         j = WORD (REPLICATE 5 F))))))
```

The correctness of the algorithmic to RTL transition of the get J block is established in HOL as follows:

```
Lemma 11: GET_J_RTL_TO_ALGORITHM_Correct
⊢ GET_J s1 e1 m1 j ⟹
   (Int (BNVAL j) = Toint (float (BV s1,BNVAL e1,BNVAL m1)))
```

where Int is the bijection function that converts a natural number to the machine integer type. The proof is done by rewriting on the definition of the function GET_J, case analysis on the input exponent, and valuations on floating-point numbers and machine integers.

B Details of Gate Level to RTL Verification

Verification of n-bit Adder The n-bit Adder in RTL level is specified as follows:

```
⊢def nadd_spec n a b cin s c =
     (BNVAL s = ((if ((BNVAL a + BNVAL b + BV cin) < (2 EXP (SUC n))) then
                   (BNVAL a + BNVAL b + BV cin)
                 else
                   ((BNVAL a + BNVAL b + BV cin) - 2 EXP (SUC n))))) ∧
     (c = ¬((BNVAL a + BNVAL b + BV cin) < (2 EXP (SUC n))))
```

The n-bit Adder at the gate level is implemented using primitive building blocks such as AND, OR, NOT, and XOR as follows:

```
⊢def nadd_imp 0 a b cin sum1 cout =
         fa_imp (a 0) (b 0) cin (sum1 0) cout) ∧
    (nadd_imp (SUC n) a b cin sum1 cout = ∃ (cripple:bool).
    (fa_imp (a (SUC n)) (b (SUC n)) cripple (sum1 (SUC n)) cout) ∧
    (nadd_imp n a b cin sum1 cripple))
```

The correctness of the n-bit Adder block is proved in HOL as in the following theorem:

```
Lemma 20: N_ADD_GATE_LEVEL_TO_RTL_Correct
⊢ nadd_imp n a b cin sum cout = nadd_spec n a b cin sum cout
```

Verification of n-bit Subtracter The n-bit Subtracter in RTL is specified as follows:

```
⊢def nSub_spec 0 a b bin dif bout =
         fs_spec (a 0) (b 0) bin (dif 0) bout) ∧
    (nSub_spec (SUC n) a b bin dif bout = ∃ (bripple:bool).
    (fs_spec (a (SUC n)) (b (SUC n)) bripple (dif (SUC n)) bout) ∧
    (nSub_spec n a b bin dif bripple))
```

The n-bit Subtracter at the gate level is implemented as follows:

```
⊢def (nSub_imp 0 a b bin dif bout =
         fs_imp (a 0) (b 0) bin (dif 0) bout) ∧
     (nSub_imp (SUC n) a b bin dif bout = ∃ (bripple:bool).
     (fs_imp (a (SUC n)) (b (SUC n)) bripple (dif (SUC n)) bout) ∧
     (nSub_imp n a b bin dif bripple))
```

The correctness of the `n-bit Subtracter` block is proved in HOL as in the following theorem:

```
Lemma 21: N_SUB_GATE_LEVEL_TO_RTL_Correct
⊢ nSub_imp  n a b bin dif bout ⇒  nSub_spec n a b bin dif bout
```

Verification of n-bit Comparator The `n-bit Comparator` in RTL is specified as follows:

```
⊢def n_BIT_COMP_Spec  n A B lf gf ef l g e =
         (l = ((BNVAL A < BNVAL B) ∧ ef) ∨ lf) ∧
         (g = ((BNVAL A > BNVAL B) ∧ ef) ∨ gf) ∧
         (e = ((BNVAL A = BNVAL B)∧ ef))
```

The `n-bit Comparator` at the gate level is implemented as follows:

```
⊢def BIT_COMPARE_Imp a b l g e =
         ∃ anot bnot s1 s2.
           (not1 a anot) ∧
           (not1 b bnot) ∧
           (and2 anot b l) ∧
           (and2 bnot a g) ∧
           (and2 a b s1) ∧
           (and2 anot bnot s2) ∧
           (or2 s1 s2 e)

⊢def n_BIT_COMP_BULID_Imp a b lf gf ef l g e =
         ∃ l1 e1 g1 sl sg.
           (BIT_COMPARE_Imp a b l1 g1 e1) ∧
           (and2 e1 ef e) ∧
           (and2 l1 ef sl)∧
           (and2 g1 ef sg) ∧
           (or2 sl lf l) ∧
           (or2 sg gf g)

⊢def (n_BIT_COMP_Imp 0 A B lf gf ef l g e =
         n_BIT_COMP_BULID_Imp (A 0) (B 0) lf gf ef l g e) ∧
     (n_BIT_COMP_Imp (SUC n) A B lf gf ef l g e = ∃ l1 g1 e1.
     (n_BIT_COMP_BULID_Imp (A (SUC n)) (B (SUC n)) lf gf ef l1 g1 e1) ∧
     (n_BIT_COMP_Imp n A B l1 g1 e1 l g e))
```

The correctness of the `n-bit Comparator` block is proved in HOL as in the following theorem:

```
Theorem: N_BIT_COMP_GATE_LEVEL_TO_RTL_Correct
⊢ n_BIT_COMP_BULID_Imp a b lf gf ef l g e = n_BIT_COMP_BULID_Spec a b lf gf ef l g e
```

Verification of n-bit Concatenator The `n-bit Concatenator` in RTL is specified as follows:

```
⊢def CONCATINATE_Spec (n:num) (X:num−>bool) (Y:num−>bool) =
        (BNVAL Y = 2 EXP (SUC n) + BNVAL X)
```

The `n-bit Concatenator` at the gate level is implemented as follows:

```
⊢def CONCATINATE_Imp (n:num) (X:num−>bool) (Y:num−>bool) =
        (Y (SUC n) = T) ∧
        (∀ n. Y n = X n)
```

The correctness of the `n-bit Concatenator` block is proved in HOL as in the following theorem:

```
Theorem: CONCATINATE_GATE_LEVEL_TO_RTL_THM
⊢ CONCATINATE_Imp n X Y ⇒  CONCATINATE_Spec n X Y
```

Verification of n-bit Multiplexer The `n-bit Multiplexer` in RTL is specified as follows:

```
⊢def MUX_Spec n A B s OUT =
        (BNVAL OUT = (if (s = F) then BNVAL A else BNVAL B))
```

The `n-bit Multiplexer` at the gate level is implemented as follows:

```
⊢def MUX_Cell_Imp a b s out =
        ∃ s1 s2 s3.
          (not1 s s1) ∧
          (and2 s1 a s2) ∧
          (and2 s b s3) ∧
          (or2 s2 s3 out)

⊢def (MUX_Imp 0 A (B:num−> bool) s OUT =
        MUX_Cell_Imp (A 0) (B 0) s (OUT 0)) ∧
    (MUX_Imp (SUC n) A B s OUT =
    (MUX_Cell_Imp (A (SUC n)) (B (SUC n)) s (OUT (SUC n)) )  ∧
    (MUX_Imp n A (B:num−> bool) s OUT))
```

The correctness of the `n-bit Multiplexer` block is proved in HOL as in the following theorem:

```
Theorem: MUX_GATE_LEVEL_TO_RTL_Correct
⊢ MUX_Imp n A B s OUT = MUX_Spec n A B s OUT
```

Verification of n-bit Shifter The `n-bit Shifter` in RTL is specified as follows:

```
⊢def ShiftRight_Spec n M L =
        ((BNVAL L * 2) + BV (M 0) = (BNVAL M))

⊢def ShiftLeFT_Spec n M L =
        (BNVAL L = 2 * BNVAL M)
```

The `n-bit Shifter` at the gate level is implemented as follows:

```
⊢def (ShiftLeFT_Imp 0 M L =
        ((L 1) = (M 0)) ∧
        (L 0 = F) ) ∧
    (ShiftLeFT_Imp (SUC n) M L =
    (L (SUC (SUC n)) = M (SUC n)) ∧
    ShiftLeFT_Imp n M L)

⊢def (ShiftRight_Imp 0 M L =
        ((L 0) = (M 1)) ∧
        (L 1 = F) ) ∧
    (ShiftRight_Imp (SUC n) M L =
    (L (SUC n) = M (SUC (SUC n))) ∧
    ShiftRight_Imp n M L)
```

The correctness of the `n-bit Shifter` block is proved in HOL as in the following theorems:

```
Theorem: SHIFT_LEFT_GATE_LEVEL_TO_RTL_THM
⊢ ShiftLeFT_Imp n M L ⇒  ShiftLeFT_Spec n M L

Theorem: SHIFT_RIGHT_GATE_LEVEL_TO_RTL_THM
⊢ ShiftRight_Imp n M L ⇒  ShiftRight_Spec n M L
```