Hierarchical Verification of the Implementation of The IEEE-754 Table-Driven Floating-Point Exponential Function using HOL

Amr T. Abdel-Hamid[1], Sofiéne Tahar[1], John Harrison[2]

[1] Electrical and Computer Engineering Department

Concordia University, Montreal, Canada

Email: {at_abdel, tahar}@ece.concordia.ca

[2] Intel Corporation,

EY2-03, 5200 NE Elam Young Parkway Hillsboro, OR 97124, USA

Email: johnh@ichips.intel.com

Technical Report

April 2001

Abstract

The IEEE-754 floating-point standard, used in nearly all floating-point applications, is considered one of the most important standards. Deep datapath and algorithm complexity have made the verification of such 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 report we have 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 the gate level implementation of the circuit up to a higher level behavioral specification. To achieve this goal, we have used both hierarchical and modular approaches for modeling and verifying the floating-point exponential function in HOL.

Keywords: Formal Hardware Verification, Higher-order Logic, Theorem Proving, Design Refinement.

1 Introduction

Designs are getting larger and larger, more complex every day. Simulation, although widely used, could never give the verification coverage needed. There are two full coverage approaches, brute-force and special-purpose simulation [4], which fail to give even a fair coverage ratio for a moderate design. Usually a test pattern (vector) is either generated randomly or by an algorithm that tries to cover the areas that could be faulty and insure that the design is working correctly. But this process, although less time consuming, is also less accurate, as sometimes faults occur where they are least expected. It is becoming clear that the ”quality” of the validation achieved by traditional simulation is rapidly deteriorating as the VLSI technology progresses.

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 [10]. 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. Formal verification can be generally divided into two main categories [14]: reachability analysis, and deductive methods. Model checkers and equivalence checkers are examples of the first approach. Many different theorem provers (as HOL [20]) have been used for deductive verification [14]. 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.

2 Related Work

There exists some related work on the verification of the floating-point algorithms and designs in the open literature. For instance, Miner [21] formalized the IEEE-854 floating-point standard in PVS. He used this formalization to verify abstract mathematical descriptions of the main real operations and their relation to the corresponding floating-point implementations. Carreno [3] defined and then formalized the same IEEE-854 standard in HOL. Miner et al. [14] parameterized the definition of the subtractive floating-point division algorithm and proved it to satisfy a formal definition of the IEEE standard. Leeser et al. [18] verified a radix-2 square root algorithm and hardware implementation. Russinoff [24, 25] used the ACL2 prover to verify the compliance of the floating-point multiplication, division, and square root algorithms of the AMD-K7 Processor to the IEEE standard.

Harrison [10] defined and formalized real numbers using HOL. Then he developed a generic floating-point library to define and verify the most fundamental terms and lemmas of IEEE-754 standard [9]. This former library was used by him to formalize and verify floating-point algorithms against behavioral specification as the square root [12] and the exponential function [9].

Another approach for verification is combining a theorem prover with a model checker or a simulation tool [16]. In this approach, theorem provers handle the high-level proof, while the low-level properties are handled by the model checker or simulation. Aagaard et al [1] used Voss and a theorem prover to verify an IEEE double precision floating-point multiplier. Cornea-Hasegen

[3] used different approaches, mathematical proofs, C language and assembly language programs, to verify the IEEE correctness of the floating-point square root, divide and remainder algorithms. O'Leary et al. [23] formalized and verified some of the Pentium Pro processor's floating-point units, such as addition and division, at the gate level using the FORTE system which integrates model checking and theorem proving techniques. Compared with the theorem proving approach, this approach is much more automatic, but still requires user guidance.

Formal verification methods have sometimes been accused of a lack of ability to get into a whole industrial product design cycle. In most of the work above, it can be noticed that it is either deeply 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. Working on the same design path of most electronic products, we will discuss in this paper the formalization and verification of the IEEE-754 table-driven exponential function in all abstraction levels of the design flow. In contrast to the above related work, we will start by verifying both RTL and gate level implementation of the IEEE-754 exponential function against the behavioral model specification developed before by [9] using HOL theorem prover.

The organization of the paper is as follows. We will present briefly the HOL theorem prover in Section 2. Section 3 describes the Table-Driven exponential function algorithm developed by Tang [26]. Section 4 introduces our methodology, and shows how hierarchical and modular techniques were used to ease the verification task. Section 5 then shows the formalized specification of the exponential function and its HOL formalized model. Section 6 describes the VHDL implementation of the algorithm then introduces its HOL formalization. Section 7 summarizes the experimental results of the whole verification task. Finally, Conclusions and future work are presented in Section 8.

3 HOL

The HOL theorem prover is an interactive proof assistant for higher-order logic, developed by Gordon [7] based on the ideas from the Edinburgh LCF project [1]. 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.

Following the LCF approach, HOL implements a small set of primitive inference rules, and all theorems must be derived using only these rules. This guards against the assertion of false "theorems". However, by ML programming it is possible to automate the translation of higher-level proof techniques into the low-level primitives. In this way, HOL users can call on an extensive selection of automated tools or write special-purpose inference rules for a given application domain. Among the higher-level inference rules provided with the system are so-called tactics which allow the user to organize proofs in a mixture of a forwar and goal-directed fashion.

In the present work, several features of HOL are particularly significant. The higher-order logic allows circuit modules to be expressed simply as predicates over inputs and outputs, allowing a very natural and direct mapping from the gate level and RTL descriptions into the logic. In addition, the extensive infrastructure of real analysis is essential to verify (or even state) the highest level of specification [9]. Finally, the adherence to the LCF methodology gives us a high confidence that the final result is indeed valid.

4 THE IEEE-754 Exponential Function

Using an approximate polynomial expansion, Tang [26] has developed an algorithm for computing the floating-point exponential function using what he calls a Table-Driven approach. In this approach, the input is first reduced to a certain working range, where L is an integer larger than or equal to 1, chosen [-log $2/2^{L+1}$, log $2/2^{L+1}$] where L is an integer larger than or equal to 1, chosen beforehand, (for instance, L = 4 for single precision [26]). Then this input x is considered to be composed of:

$$x = \frac{(32 * m + j) * (log2)}{32} + (r1 + r2)$$

where m and j are integers, and $r1$ and $r2$ are real numbers, $\mid r1 + r2 \mid <$ (log 2/64) .

Starting from this equation, the exponential function can be constructed as follows [2]:

$$x = (m * log2) + \frac{(j * log2)}{32} + r$$

where *r = r1 + r2*. The exponential of x will hence be equal to

$$\begin{aligned} exp(x) &= exp((m * log2) + \frac{(j * log2}{32} + r) \\ &= exp(log2^m) + log2^{\frac{j}{32}}) + r \\ &= 2^m + 2^{\frac{j}{32}} + exp(r) \end{aligned}$$

Here, $exp(r)$ can be represented using Taylor expansion as follows:

$$p(r) = r + (a1 * r) + (a2 * r^2) + (a3 * r^3) + ...$$

where *p(r) = exp (r) -1*, and *a1, a2,...* are the coefficients of Taylor expansion.

Returning to the exponential function:

$$exp(x) = 2^m * 2^{\frac{j}{32}} * (p(r) + 1)$$

The main objective of the algorithm is isolating m and j, and evaluating the approximating polynomial. According to Tang [3], four steps are needed to compute this exponential function:

Step 1. Filter any out-of-bounds inputs that occur. As mentioned before, x should be a number between $[-(log2/32), (log2/32)]$. So, if *x* is NaN (not-a-number, invalid IEEE-754 format), out of range, zero, positive or negative infinity, the algorithm would either be able to compute it by an approximated arithmetic operations (as in the case of positive or negative infinity), or not able to solve it at all (as for NaN).

Step 2. Start the computation by first calculating *N*,

$$N = INTEGER(X * INVL)$$

where *INVL* is a floating-point constant approximately equal to $32/log2$ in our case, and INTEGER is the default IEEE-754 round-to-nearest mode. This *N* is composed of two parts,

$$N = N1 + N2$$

where $N1 = 32 * m$, and $N2 = j$

So, the variables *m* and *j* can be derived from the previous result as follows [4]:

$$j = N2$$

$$m = \frac{N1}{32}$$

With the value of *N*, *r1* and *r2* can be calculated as follows [26]:

If the absolute value of $N \geq 2^9$ then

$$r1 = (x - N * L1)$$

else

$$r1 = (x - N1 * L1) - N2 * L2$$

and

$$r2 = -N * L2$$

L1 and *L2* are constants, where *L1* + *L2* approximates $log2/32$ to a precision higher than that of single precision (the working one) [2].

Step 3. Compute the polynomial *p(r)*, similar to the Taylor expansion, as follows [26]:

$$r = r1 + r2$$

$$Q = r * r * (a1 + r * a2)$$

$$p(r) = r1 + (r2 + Q)$$

where the coefficients (*a1* and *a2*) are obtained from a Remez algorithm calculated by Tang [26].

Step 4. The values of $2^{j/32}$, $j = 0, 1, ...32$, are calculated beforehand and represented by two working-precision numbers [26] (single precision in our case), $Slead$ and $Strail$. The sum approximates $2^{j/32}$ to roughly double the working precision [2]. Finally $exp(x)$ is calculated as follows:

$$S = Slead(j) + Strail(j)$$

$$exp(x) = 2^m + (Slead(j) + (Strail(j) + S * p(r)))$$

5 Verification Methodology

Usually there is a large abstraction gap between specification and implementation. This gap cannot be bridged in one move. Hence, a series of design step (levels) is performed, reducing the abstraction levels until a realizable description is available. These levels, as mentioned by Kropf [17], are *Architecture, Register-Transfer, Gate, Transistor, and Layout* levels (Figure 1).

Errors can occur in any of these levels or in the transfer of one level to another. Usually the implementation of a higher level is considered the specification of the lower one as shown in Figure 1. Design faults may result from erroneous transformation of the specification, given on a certain abstraction level, into an implementation on the next lower level. Three main fault classes can be distinguished according to [17]. The first class encompasses *design* faults. The second class resides

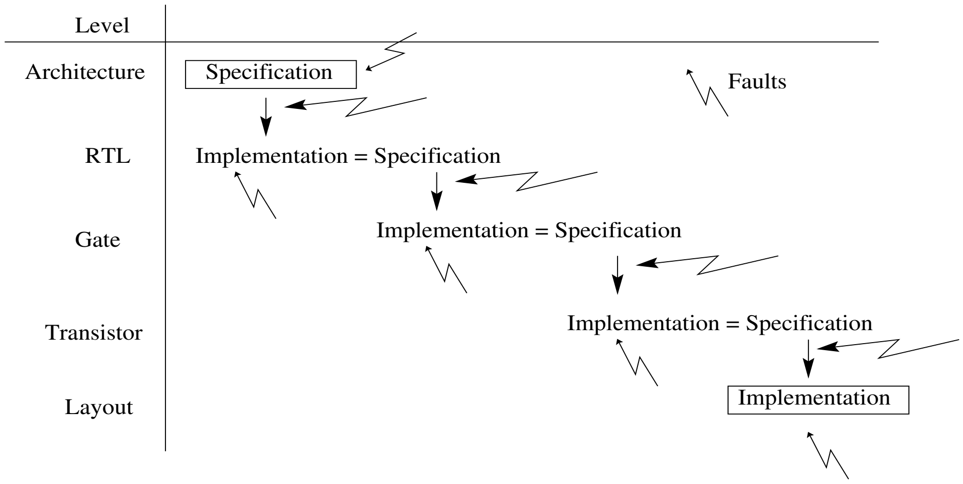

Figure 1: Modeling and Verification Stages

in it local optimizations that can be made in the same level. The third is inherited implementation faults.

The verification process for the *Table-Driven exponential function* was performed on many levels, as shown in Figure 2. Harrison [9] formalized and verified that a behavioral specification, an abstract algorithmic description he developed for the design, implies an abstract mathematical description of the IEEE-754 Table-Driven floating-point exponential function [26]. Starting form this behavioral specification, written in a "while-language", Bui et al. [1] developed an RTL implementation of the design. The goal of this work is the modeling and verification of the latter implementation with respect to the behavioral specification designed by [9] using HOL. We were also interested in the development of a formal proof that the gate implementation, machine synthesized using the Synopsys tool, implies the RTL implementation.

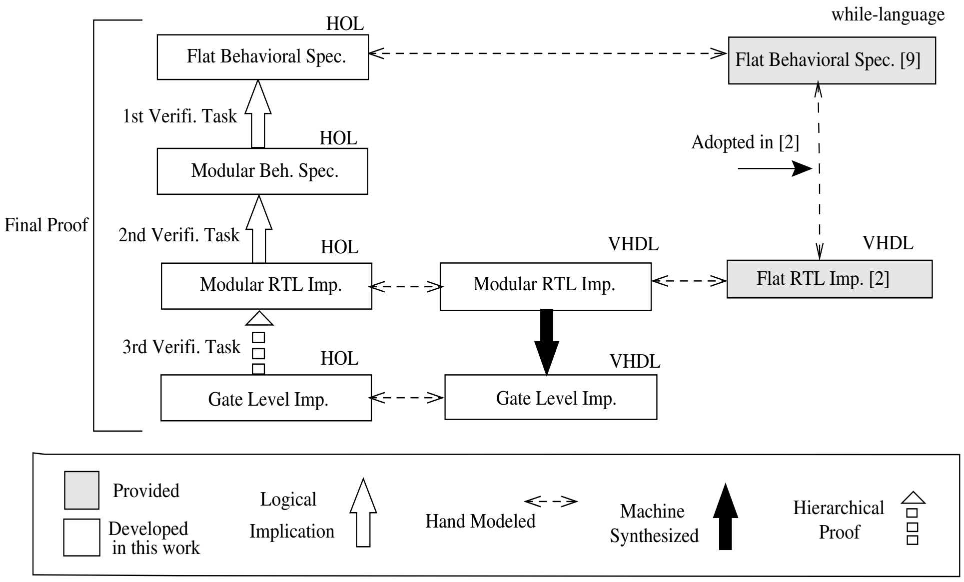

Figure 2: The Exponential Function Verification Project

From the beginning of the verification project, it was noticed that the behavioral specification of [9] is too flat for a lower level verification process. So, a new modular behavioral specification was introduced, where its modules are shown in Figure 4 with corresponding source as given in Figure 5. The same had to be performed with the RTL implementation where a newer modular VHDL code was driven from the older flat one. The newer modular implementation and specification were essentially considered to ease the verification task. The overall verification process is now composed of four main tasks (Figure 2):

First, we prove the correctness of the modular behavioral specification against the flat specification provided in [9].

Second, we prove that the modular RTL implementation implies the modular behavioral specification.

Third, we prove hierarchically that the synthesized gate level implementation implies the modular RTL description.

Finally, The whole proof should be linked in one global proof covering the whole design cycle of the floating-point exponential function. Starting from the gate level implementation, we should be ending by the abstract mathematical description of the function. These verification paths are shown in Figure 2, where the shaded boxes are the material provided by [9], [26] and [1], while the white ones represent those developed in this work.

In this paper, we have adopted a hierarchical approach for the modeling and verification of our design. In this approach, the design is structured into a hierarchy of components and subcomponents, and specifications that describe "primitive components" at one level of the hierarchy then become specifications of the intended behavior at the next level down [5] (Figure 3). Hierarchical organization not only makes the verification process natural, it also makes it tractable [6]. By breaking large problems into smaller pieces that can be handled individually. It effectively increases the range of circuit sizes that can be handled in practice [6].

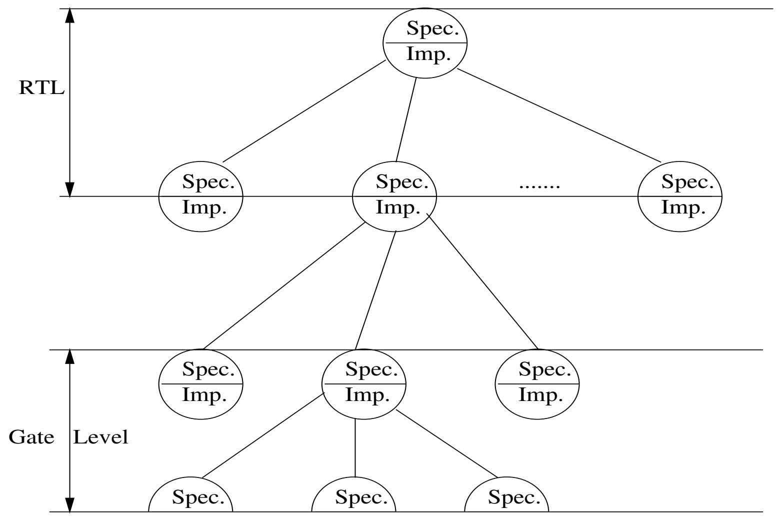

Figure 3: Hardware Hierarchical Verification approach

6 Formal Specification of IEEE-754 Exponential Function

In this section we focus on the detailed model specification we formalized for the Table-Driven exponential function. As discussed in the previous section, the scope of this report is verifying that the RTL hardware implementation of the exponential function conforms with the behavioral description written in [9] as shown in Figure 4, which we used as our main specification.

As discussed above, we were faced by the flatness problem of the specification, which was not directly useful for a hardware synthesis and/or verification. We have divided this specification into six intermediate blocks (modules) where the conjunction of these blocks (Figures 4 and 5) represents the full specification of the code described below. It should be noted, that the mathematical operators (such as *, +, and /) in the while code are bit vector operations and not mathematical real numbers operators. Trying to achieve maximum modularity for the design, we have tried to minimize the interfaces between different modules. This helps us to divide the verification tasks into well-defined smaller ones. Each of these blocks was also divided into smaller specifications giving us smaller sub-specifications clearly related to the goals needed to be proved.

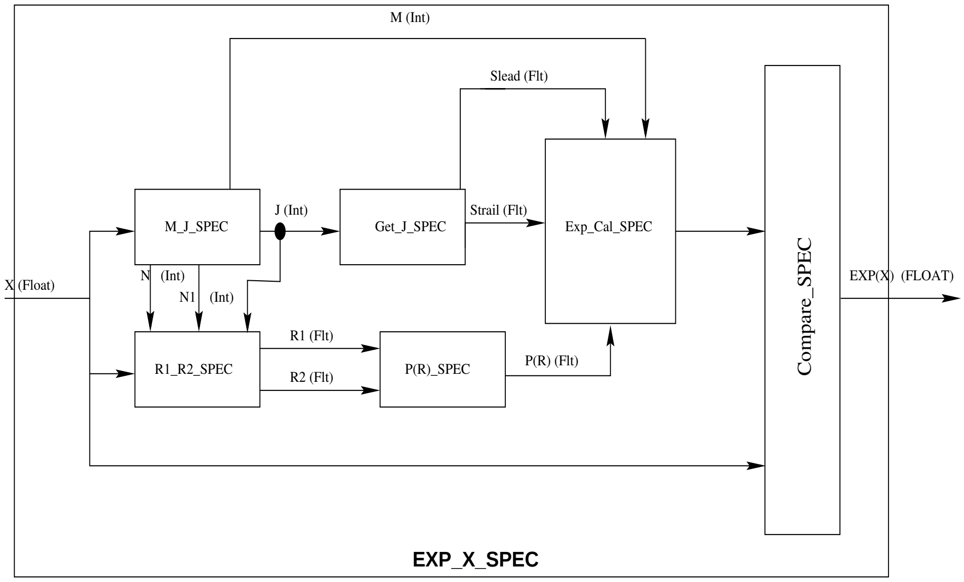

Figure 4: The Modular Organization of the Exponential Function Specification

The six modules composing the system (Figure 4) are completely responsible for checking the input X value and computing the result of the exponential function. These blocks are:

1) *m and j computing block* (M_J_SPEC) : responsible for half of Step 2 (cf. Section 3) by computing the values of m and j. Its input is the number X and its outputs are J, M, N, and $N1$.

2) *r1* and *r2* block ($R1\_R2\_SPEC$): responsible for the second half of Step 2, it computes the values of $r1$ and $r2$. Its inputs are X, $N1$, $N2$ (equal to J) and its outputs are the two floating point numbers $R1$ and $R2$.

3) *p(r) block* (P_R_SPEC) : computes the value of $p(r)$, it takes $R1$ and $R2$ and outputs P_R.

4) *Slead and Strail block* (Ge_J_SPEC) : a floating-point multiplexer module, where the value of J decides which values for Strail and Slead should be chosen. Its input is just the number

J and its outputs are Slead and Strail.

5) *Exponent calculation block* (Exp_Cal_SPEC) : the main computational block where finally the exponential function is computed. It takes $Slead$, $Strail$, M and P_R as its inputs and output is the EXP_X.

6) *Checking block* (Compare_SPEC): the compare and decision module. According to the value of input X, this module decides whether to choose the computed value of the exponent or another output as NAN (Not A Number). It takes X and the computed exponent as its inputs and the final answer is the output (OUT_EXP_X).

The highest-level system specification in HOL is as follows:

```
⊢def IEEE_EXP_SPEC Xs Xe Xm EXP_s EXP_e EXP_m =
∃ Ns Ne Nm Ms Me Mm Js Je Jm N1s N1e N1m R1s R1e R1m...
(M_J_SPEC Xs Xe Xm Ns Ne Nm Ms Me Mm Js Je Jm..)∧
(R1_R2_MOD_SPEC Xs Xe Xm Ns Ne Nm ..)∧
(Get_J_SPEC Js Je Jm Strail_s Strail_e Strail_m ..)∧
(P_R_SPEC R1_s R1_e R1_m R2_sR2_eR2_mPR_sPR_ePR_m)∧
(EXP_CAL_MOD_SPEC Stail_s Stail_e Stail_m ..)∧
(Compare_SPEC EXP_1_s EXP_1_e EXP_1_m Xs Xe Xm EXP_s EXP_e EXP_m)
```

There is a high level of regularity in these six modules where floating-point operations, like addition, multiplication, are the main sub-modules in all of them. This will help us in the reuse of the developed models and theories in building the higher levels of the specification. Each of the six modules was modeled as a conjunction of lower level components. As an example, we will describe the m-and-j module specification (M_J_SPEC).

The M_J_SPEC is responsible for computing the m and j values mentioned in the description section. It is composed of the conjunction of five sub-specifications as shown in Figure 6. These sub-specifications are the floating point multiplication module (FP_MUL_SPEC), floating-point to integer approximation module (FP_INT_SPEC), Modulo 32 module (Mod_32_SPEC), floating-point subtraction module (FP_Sub_SPEC), and division by 32 module (Div_32_SPEC) . Here m and j are integer valued numbers, even though they were represented in the IEEE-754 floating-point format, as it is easier to use them afterwards in this format. In short, this module would have one input (x) , composed of three parts (sign, exponent and mantissa) and four outputs (J, M, N, and N1), each composed of the same three parts. This was modeled in HOL as follows:

```
⊢def M_J_SPEC Xs Xe Xm Ns Ne Nm N1s N1e N1m Ms Me Mm Js Je Jm =
∃ const_s const_e const_m s1_s s1_e s1_m.
(const_s = F) ∧ valu const_e 7 = 132) ∧ (valu const_m 22 = 3713595) ∧
(FP_MUL_SPEC Xs Xe Xm const_s const_e const_m s1_s s1_e s1_m) ∧
(FP_to_INT_SPEC s1_s s1_e s1_m Ns Ne Nm) ∧ (Mod_32_SPEC Ns Ne Nm Js Je Jm) ∧
(FP_Sub_SPEC Ns ... N1s N1e N1m) ∧ (DIV_32_SPEC N1s N1e N1m Ms Me Mm)
```

Each of these main modules is then hierarchically top-down specified to reach the full specification of this system. As an example, for the next levels specifications, we consider the floating-point multiplier sub-module (Figure 7). This sub-module has three datapaths: the sign, the exponent and

```
Int_32 = Int(32)
Int_2e9 = Int (2 EXP 9)
Int_2e9 = Int (2 EXP 9)
Plus_one = float (0, 127, 0)
THRESHOLD_1 = float (0, 134, 6066890)
THRESHOLD_2 = float (0, 102, 0)
Inv_L = float (0, 121, 3240448)
L1 = float (0, 121, 3240448)
L2 = float (0, 102, 4177550)
A1 = float (0, 126, 68)
A2 = float (0, 124, 2796268)
var x:float, E:float, R1:float, R2:float,
R:float, P:float,  Q:float,S:float, E1:float,
N:Int, N1:Int, N2:Int, M:Int, J:Int;

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 (abs(x) > THRESHOLD_1 then Checking Block
if X > Plus_Zero then E:= Plus_infinity
else E:= Plus_Zero  else if abs(X) < THRESHOLD_2
then E:= Plus_one + X
else ( N:= INTRND (X * Inv_L);
N2:= N \% Int_32;   m and j computing block
N1:= N - N2;  M:= N1 / Int_32;   J:= N2;
if abs (N) Int_2e9 then
R1:= (X-Tofloat(N1) * L1) - Tofloat (N2) * L1 r1 and r2 block
else   R1:= X - Tofloat(N) * L1;
R2:= Tofloat(N) * L2;
R:= R1 + R2;
Q:= R * R (A1 + R * A2);
p(r) block P:= R1 + (R2 + Q);
S:= S_Lead(J) + S_Trial(J);      Slead and Stail block
E1:= S_Lead(J) + (S_Trial(J) + S * P); Exponent calculation block
```

Figure 5: Full Specifications of the Exponential Function in While-Langauge

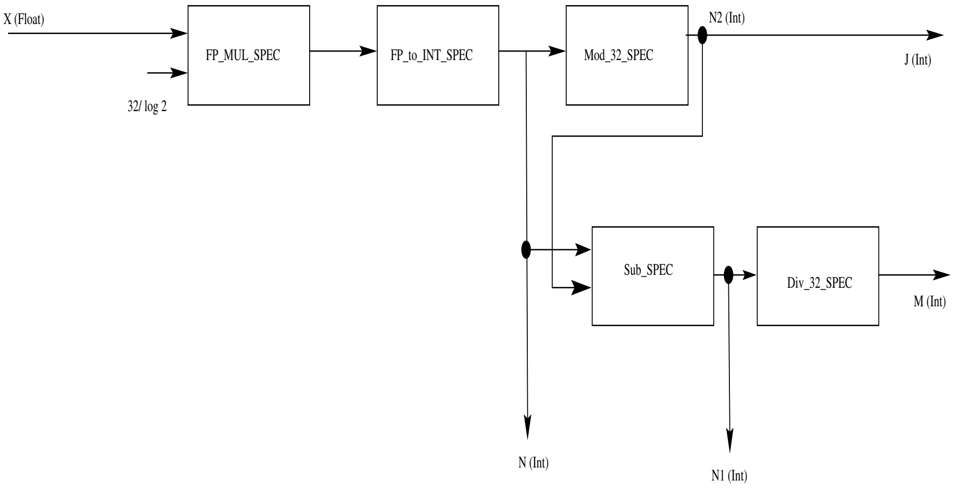

Figure 6: Specification of the m_and_j module (M_J_SPEC)

the mantissa. For the sign and the exponent, the specification could be done directly on this level, but for the mantissa we have to build another level of hierarchical modules and start lower level specifications. This can be formalized as follows using HOL:

$\vdash_{def}$ `FP_MUL_SPEC A_s A_e A_m B_s B_e B_m`
`MULout_s MULout_e MULout_m overFlow =`
$\exists$`check.`
`(Mantissa_SPEC A_m B_m MULout_m check)` $\wedge$
`(Exp_SPEC A_e B_e MULout_e check overFlow)` $\wedge$
`(Sign_SPEC A_s B_s MULout_s)`

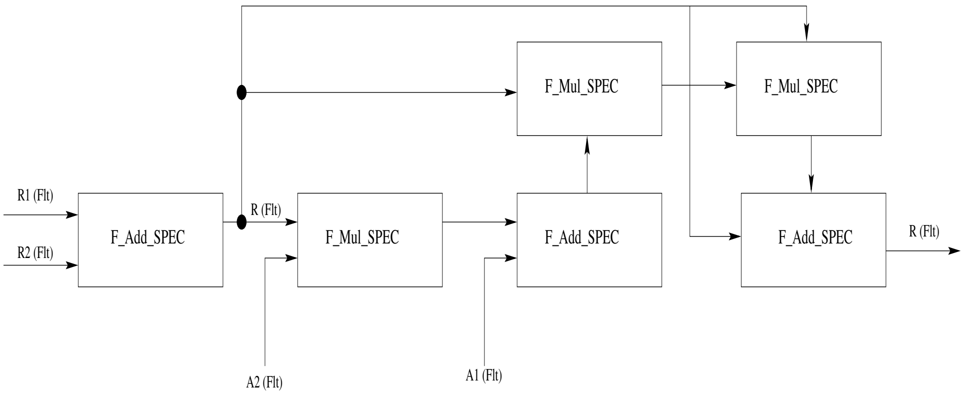

Figure 7: Specifying the Floating-Point Multiplier

where Sign, Exponent and Mantissa modules were specified as follows:

$\vdash_{def}$ `Sign_SPEC i1 i2 out = (out = ((i1 = i2) => F|T))`

```
⊢def  EXP_SPEC e1 e2 e3 e4 overFlow  =
((valu e4 7 = ((((((valu e1 7 − 127) + (valu e2 7 − 127)) − valu e3 7))
+ 127  <  2 EXP SUC 7)∧
(((((valu e1 7 − 127) + (valu e2 7 − 127)) − valu e3 7))
+127)|((((((valu e1 7 − 127)+
(valu e2 7 − 127)) − valu e3 7)) + 127) − 2 EXP SUC 7)))∧
(overFlow =  ((((((valu e1 7 − 127)+
(valu e2 7 − 127)) − valu e3 7)) + 127 < 2 EXP SUC 7))))
```

```
⊢def  Mantissa_SPEC A B MULout const =
∃ A_1 B_1 MULout_pre MULout_pre_1 C P check.
(Concatenate_SPEC 23 A A_1 T)∧
(Concatenate_SPEC 23 B B_1 T)∧
(MUL_SPEC 24 A_1B_1 C P MULout_pre)∧
(Check_SPEC MULout_pre const check)∧
(Shifter_SPEC MULout_pre MULout_pre_1 check)∧
(Truncate_SPEC MULout_pre_1 MULout 25)
```

This latter is composed of a number of sub-modules at the lower level. As an example, we have shown the specification of one of the main modules which is the integer multiplier:

```
⊢def  CELL_MUL_SPEC a b c p co po  =
bv po  = ((bv(ab) + bv c + bv p < 2)  =>
(bv(ab) + bv c + bv p) | (bv(ab) + bv c + bv p) − 2))∧
(co  =∼ (bv(ab) + bv c + bv p < 2))
```

```
⊢def  LEFT_SHIFT_SPEC X Y = ∀ n.((Y 0 = F) ∧ (Y(SUC n) = X n))
ROW_MUL_SPEC A b C P CO PO Aout  = ∀n.
(!c.(((bv(PO n) = (bv((A n) b) + bv(C n) + bv(P n) < 2  =>
(bv((A n) b) + bv(C n) + bv(P n))|(bv((A n) b) + bv (C n) + bv(P n)) − 2))∧
((c n) =  (bv((A n) b) + bv(C n) + bv(P n) < 2))))∧
LEFT_SHIFT_SPEC A Aout  ∧  LEFT_SHIFT_SPEC c CO∧
(C (SUC n)  = F)(P (SUC n) = F))
```

```
(∗ Recursive definition of the Array module ∗)
(ARRAY_MUL_SPEC 0 A B C P Co Po Aout
   = ROW_MUL_SPEC A (B 0) C P Co Po Aout)∧
(ARRAY_MUŁSPEC (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 a (B (SUC n)) c p Co Po Aout)
```

```
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))
```

7 Implementation of IEEE-754 Exponential Function

In this section, we describe briefly the implemented VHDL code developed by Bui *et al.* [2] of the IEEE-754 exponential function and its modular model in HOL. We were faced with the same problem in the specification, which was the flatness of the design. The code was so flat as to make it nearly impossible to be modeled, let alone verified. So we had to make some design changes, which keep the same code properties but make the design easier to model and verify. We have aimed mainly in our changes to attack the following criteria:

1) Logic Complexity: Hierarchical designs reduce the logic complexity in the circuit. Also, in some modules the code could be changed to perform the same function, although it is less complex.

2) Verification time and effort: it is very hard to model and verify a very large flat design. Redesigning the VHDL code has saved a lot of time and effort needed in the verification process. Also, this helped in emphasizing that the modules needed to be verified.

These goals were reached by deriving a newer VHDL implementation based on modules from the code written in [2]. The new high-level RTL implementation is composed of modules corresponding to the high level specification. We have shown in Figure 6 the synthesis of this code as resulting from the Synopsys design analyzer tool. This figure only shows the exponential computation module.

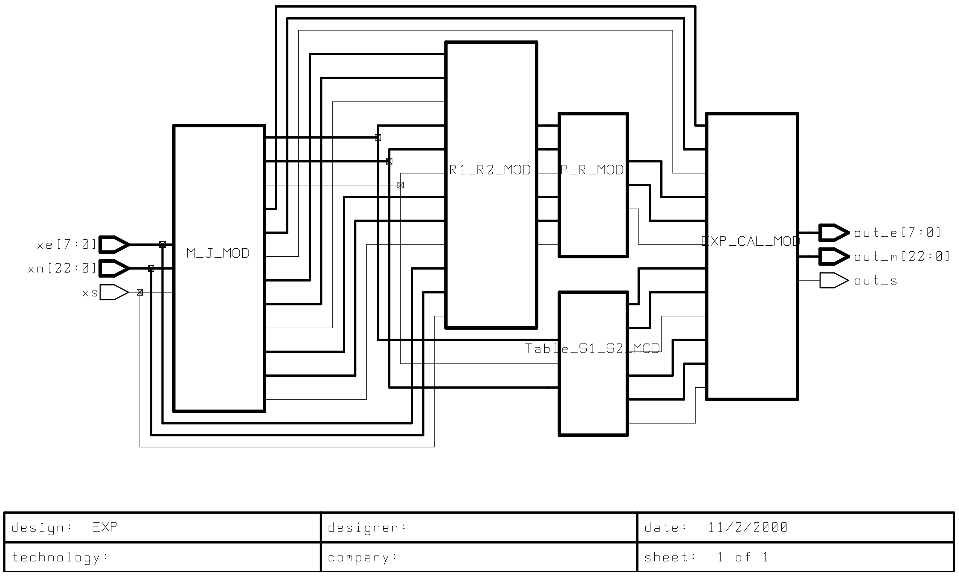

Figure 8: Top-level VHDL Implementation Built in Synopsys Design Analyzer

It was not that difficult to move this code into HOL notation. The HOL high level model of our implementation was as follows, which was nearly a one-to-one mapping to VHDL:

```
⊢def IEEE_EXP_IMP Xs Xe Xm EXP_s EXP_e EXP_m =
∃ Ns Ne Nm Ms Me Mm Js Je Jm N1s N1e N1m...
(M_J_IMP Xs Xe Xm Ns Ne Nm Ms Me Mm Js Je Jm N1s N1e N1m) ∧
(R1_R2_MOD_IMP Xs Xe Xm Ns Ne Nm N1s N1e N1m ..) ∧
(Get_J_IMP Js Je Jm Strail_s Strail_e Strail_m ..) ∧
(P_R_IMP R1_s R1_e R1_m R2_s R2_eR2_m PR_s PR_e PR_m) ∧
(EXP_CAL_MOD_IMP Slead_s Slead_e Slead_m ..) ∧
(Compare_IMP EXP_1_s EXP_1_e EXP_1_m Xs Xe Xm EXP_s EXP_e EXP_m)
```

To show how the hierarchical implementation of these modules were modeled in HOL, we will stick to the same example we gave before in the specification: we will discuss the VHDL code of the M_J module and its HOL model all the way down to the gate level implementation.

As discussed before, the module is composed of primitive floating-point functions, such as floating-point addition (called Adder1 in the VHDL chart below) which were implemented in the lower levels. The M_J module was implemented in VHDL. This code was then synthesized using the Synopsys tool. This was directly modeled in HOL as follows:

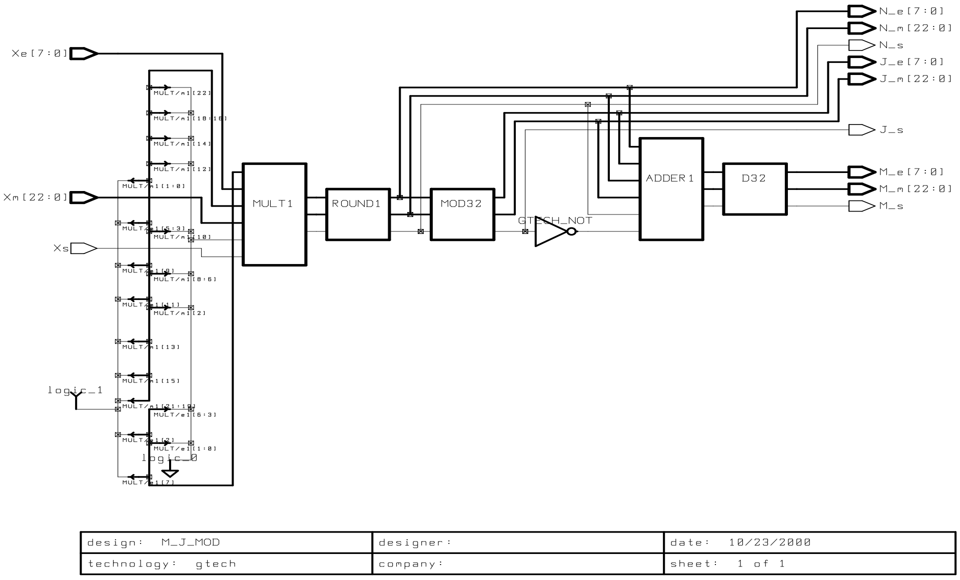

Figure 9: Synthesis of the M_J Module VHDL Implementation

```
⊢def  M_J_IMP Xs Xe Xm  Ns Ne Nm  N1s N1e N1m  Ms Me Mm  Js Je Jm  =
∃ const_s const_e const_m s1_s s1_e s1_m.
(const_s = F) ∧
(valu const_e 7 = 132) ∧
(valu const_m 22 = 3713595) ∧
(FP_MUL_IMP Xs Xe Xm  const_s const_e const_m  s1_s s1_e s1_m) ∧
(FP_to_INT_IMP s1_s s1_e s1_m  Ns Ne Nm) ∧
(Mod_32_IMP Ns Ne Nm  Js Je Jm) ∧
(FP_Sub_IMP Ns Ne Nm  Js Je Jm  N1s N1e N1m) ∧
(DIV_32_IMP N1s N1e N1m  Ms Me Mm)
```

Similarly, we would proceed to more and more sub-modules in order to reach the gate level implementation. We use intermediate sub-modules to cover the gap between the RTL and the gate level. This ends with simple gate building blocks of all the modules. These gates are considered the elementary building blocks for the whole architecture. Examples of these primitives are AND, OR, NOT, XOR, etc.

After fully modeling the specification and implementation of the system, the next task was the verification process, which will be discussed in the next section along with the experimental results. The high similarity between the implementation at high levels and the specification considerably eased the verification task.

8 Formal Verification of the Exponential Function

So far, we modeled the behavioral and RTL specifications for the IEEE-754 Table-Driven exponential function in a modular form. We also modeled the gate level implementation of the same function. The next step is to verify these different levels using a hierarchical proof approach in HOL.

Let X be the input and EXP the exponential output, our final goal is:

$$\begin{array}{l} \forall\ X\ \ EXP. \\ \qquad if\ \frac{-log2}{32} \leq X \leq \frac{log2}{32}. \\ \qquad (EXP\_GATE\_IMP\ \ X\ \ EXP\ \ \Longrightarrow\ \ EXP\_BEH\_SPEC\ \ X\ \ EXP) \end{array}$$

This goal can be written differently in HOL, where X_s, X_e and X_m are the three parts of the input floating-point number and EXP_s, EXP_e and EXP_m denote the floating-point output. The assumption on X is made by the boundary module within both the specification and the implementation. So the goal would be the following:

$$\begin{array}{l} \forall\, X\_s\ \ X\_e\ \ X\_m\ \ EXP\_s\ \ EXP\_e\ \ EXP\_m. \\ \qquad IEEE\_EXP\_GATE\_IMP\ \ X\_s\ \ X\_e\ \ X\_m\ \ EXP\_s\ \ EXP\_e\ \ EXP\_m \\ \qquad\ \ \Longrightarrow IEEE\_EXP\_SPEC\ \ X\_s\ \ X\_e\ \ X\_m\ \ EXP\_s\ EXP\_e\ \ EXP\_m \end{array}$$

This goal cannot be reached directly, due to the very high abstraction gap between the gate and behavioral levels as described above. So, the proof scheme was changed to hierarchically prove that the gate level implies the more abstract RTL. Then this RTL was related, by a formal proof, to a modular behavioral specification. The latter was proved to imply the high level flat behavioral specification. This can be formalized as follows in HOL:

$$\begin{array}{l} \vdash_{thm}\ \ \forall\, X\ \ EXP.\ IEEE\_EXP\_GATE\_IMP\ \ X\ \ EXP \\ \qquad\qquad\qquad\qquad \Longrightarrow\ \ IEEE\_EXP\_RTL\_IMP\ \ X\ \ EXP \end{array} \tag{1}$$

$$\begin{array}{l} \vdash_{thm}\ \ \forall\, X\ \ EXP.\ IEEE\_EXP\_RTL\_IMP\ \ X\ \ EXP \\ \qquad\qquad\qquad\qquad \Longrightarrow\ IEEE\_EXP\_MOD\_BHV\_SPEC\ \ X\ \ EXP \end{array} \tag{2}$$

$$\begin{array}{l} \vdash_{thm}\ \ \forall\, X\ \ EXP.\ IEEE\_EXP\_MOD\_BHV\_SPEC\ \ X\ \ EXP \\ \qquad\qquad\qquad\qquad \Longrightarrow\ \ IEEE\_EXP\_BHV\_SPEC\ \ X\ \ EXP \end{array} \tag{3}$$

Finally using equations (1), (2) and (3) we can reach the final goal stated again in equation (4):

$$\begin{array}{l} \vdash_{thm}\ \ \forall\ X\ \ EXP.\ IEEE\_EXP\_GATE\_IMP\ X\ \ EXP \\ \qquad\qquad\qquad\qquad \Longrightarrow\ IEEE\_EXP\_BHV\_SPEC\ \ X\ \ EXP \end{array} \tag{4}$$

Due to the high modularity of the two intermediate blocks, the goals (1), (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 goal. As an illustrative example let's consider the M_J module at the RTL level, whose specification and implementation have been shown above. In the following theorem the goal was set as:

```
e( REPEAT GEN_TAC THEN
   REWRITE_TAC[M_J_Imp,M_J_SPEC] THEN
   REPEAT STRIP_TAC THEN
   EXISTS_TAC (--`const_s:bool`--) THEN
   .
   .
   ARW_TAC[DIV_32_correct, Mod_32_correct ,
           FP_to_INT_correct, FP_MUL_correct,
           FP_Sub_correct, FP_Add_correct ])
```

Theorems like $DIV\_32\_correct$, $Mod\_32\_correct$, etc. are lemmas that were already proved in previous verification steps. Usually in lower modules different strategies were developed. For instance, induction (INDUCT_TAC) was mostly used in proving recursive functions. Automatic tactics as PROVE_TAC and ARW_TAC were also used when the goal was straightforward. These tactics take more machine time but they shorten the proof and decrease the manpower needed.

A summary of the verification times for the whole system is given in Table 1. All the experiments have been carried out on a Sun Ultra SPARC 2 workstation with 296 MHz processor and 768 MB of memory. In the table, we have showed the verification time of two main building blocks, the floating-point adder and multiplier. These modules took a very high verification time, but due to the high reusability of pre-proven theorems and lemmas, other building blocks and even the main module required much less verification time. Also, it can be seen that the verification time for four of the six main modules are much smaller, as all these modules' building blocks were pre-proved, making the final task shorter. The sum of the times of the systems showed here will be less than the total verification time as there were some lemmas verified to achieve the final proof goal. The whole code was composed of nearly 4600 lines.

Table 1: Verification times of different system modules

Module Name	**Verification Time (Sec.)**
Floating-Point Addition	60.500
Floating-Point Multiplication	30.540
M_J Module	2.120
R1_R1 Module	5.620
P_R Module	3.420
EXP_Cal Module	1.970
IEEE_EXP Module	5.290
Total Verification Time	**214.200**

9 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* problem, 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.

In this paper, we have demonstrated the use of HOL to model the behavioral and RTL specifications for the IEEE-754 Table-Driven exponential function in a modular form, as well as the modeling of the gate level implementation of the same function. Finally, using *hierarchical verification*, we have been able to develop a formal proof indicating the correctness of the implementation using the HOL tool.

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.

As future work, we are working in linking the results of this work to the results of Harrison [9]. This will build a full proof of this design, starting from the gate level implementation and ending with the abstract mathematical description proposed by Tang, giving a clear image of the formal verification integration in the design cycle.

References

[1] M. D. Aagaard, and C.-J. H Seger, ”The formal verification of a pipelined double-precision IEEE floating-point multiplier”, Proc. ICCD’96, November 1995.

[2] H. T. Bui, B. Khalaf, and S. Tahar, ”Table-Driven Floating-Point Exponential Function”, CCECE’99, May 1999.

[3] V. A. Carreno, ”Interpretation of IEEE-854 Floating-Point Standard and Definition in the HOL system”, NASA Technical Memorandum 110189, September 1995.

[4] Y. Chen, and R. E. Bryant, ”Verification of Floating-Point Adders”, Computer Science Department, School of Computer Science, Carnegie Mellon University, 1998.

[5] Y. Chen, ”Arithmetic Circuit Verification Based on Word-Level Decision Diagrams”, School of Computer Science. Carnegie Mellon University, 1998.

[6] M.Cornea-Hasegan, ”Proving the IEEE Correctness of Iterative Floating-Point Square Root, Divide, and Remainder Algorithms”, Intel Technology Journal, 1998.

[7] M.J.C. Gordon and T.F.Melham, ”Introduction to HOL: a theorem proving environment for higher order logic”, Cambridge University Press, 1993.

[1] M.J.C. Gordon, R. Milner and C. Wadsworth, "Edinburgh LCF: A Mechanised Logic of Computation". J. R. Harrison, "Floating-point verification in HOL light: the exponential function", Technical Report number 428, University of Cambridge Computer Laboratory. UK, June 1997.

[9] J. R. Harrison, "Floating point verification in HOL light: the exponential function", Technical Report number 428, University of Cambridge Computer Laboratory. UK, June 1997.

[10] J. R Harrison, "Theorem Proving with the real Numbers", Technical Report number 408, University of Cambridge Computer Laboratory, December 1996.

[11] J. R. Harrison, "A Machine-Checked Theory of Floating Point Arithmetic", Proc. TPHOLs'99, August 1999.

[12] J. R. Harrison, "Verifying the accuracy of polynomial approximations in HOL", TPHOLs'97, August 1997.

[13] D. Hoffmann, and T. Kropf, "Verification of a GF (2m) Multiplier-Circuit for Digital Signal Processing", Technical Report 22/98, University of Karlsruhe, 1998.

[14] Intel Inc., "Pentium Processors, Statistical Analysis of Floating Point Flaw", Intel White Paper, Sec. 3, November 1994.

[15] IEEE Standards, "IEEE Standard for Binary Floating-Point Arithmetic Std. 754-1985", IEEE Standard, IEEE, 1985.

[16] C. Kern, and M. R. Greenstreat, "Formal verification in Hardware Design: A Survey", ACM transactions on Design Automation of Electronic Systems, Vol. 4 No. 2, April 1999.

[17] T. Kropf, "Introduction to Formal Hardware Verification", Springer, 1999.

[18] M. Leeser, and J. O' Leary, "Verification of a subtractive radix-2 square root algorithm and implementation", Proc. ICCD;95, October 1995.

[19] T. Melham, "Higher Order Logic and Hardware Verification", Cambridge University Press, 1993.

[20] T.F. Melham and M.J.C. Gordon, Higher Order Logic and Hardware Verification, Cambridge Tracts in Theoretical Computer Science 31, Cambridge University Press, 1993.

[21] P. S. Miner, "Defining the IEEE-854 Floating-Point Standard in PVS", NASA Technical Memorandum 110167, June 1995.

[22] P. S. Miner, and J. F. Leathrum, "Verification of IEEE Compliant Subtractive Division Algorithms", Proc. FMCAD'96, November 1996.

[23] J. O'Leary, X. Zhao, R. Gerth, and C. H. Seger, "Formally verifying IEEE Compliance of Floating-Point Hardware", Intel Technology Journal, 1999.

[24] D. M. Russinoff, "A Mechanically Checked Proof of IEEE Compliance of a Register-Transfer-Level Specification of the AMD-K7 Floating-Point Multiplication, Division, and Square Root Instructions", LMS Journal of Computation and Mathematics, December 1998.

[25] D. M. Russinoff, ”A Mechanically Checked Proof of IEEE Compliance of the AMD K5 Floating-Point Square Root Microcode”, Formal Methods in System Design, vol. 14, no. 1, January 1999.

[26] P. T. P. Tang, ”Table-Driven Implementation of the Exponential Function in IEEE Floating-Point Arithmetic”, ACM Transactions on Mathematical Software, vol. 15, no. 2, 1989.