Formal Verification of the VAMP Floating Point Unit

Christoph Berg and Christian Jacobi

Saarland University
Computer Science Department
D-66123 Saarbrücken, Germany
Fax: +49/681/302-4290
{cb,cj}@cs.uni-sb.de

Abstract. We report on the formal verification of the floating point unit used in the VAMP processor. The FPU is fully IEEE compliant, and supports denormals and exceptions in hardware. The supported operations are addition, subtraction, multiplication, division, comparison, and conversions. The hardware is verified on the gate level against a formal description of the IEEE standard by means of the theorem prover PVS.

1 Introduction

Our institute at Saarland University is currently working on the formal verification of a complete microprocessor called VAMP. Part of this microprocessor is a fully IEEE compliant floating point unit (FPU). This paper describes the verification of the FPU in the theorem prover PVS [19].

The FPU we have verified is developed in the textbook on computer architecture by Müller and Paul [17]. The designs go down to the level of single gates. Along with the complete designs come paper proofs for the correctness of the circuits. These paper proofs served as guidelines for the formal proofs. We have specified and verified these designs on the gate level in PVS. Only small changes to the designs were necessary – some due to errors in [17], some to slightly simplify the proofs – with negligible impact on hardware cost and cycle time.

We have verified the designs with respect to a formalization of the IEEE standard 754 [10] (hereafter called "the standard"). We have partly used the formalization of the standard and the theory of rounding from [6,17], particularly the notion of factorings, round decomposition, and α-equivalence. Other parts of our IEEE formalization are influenced by Miner's formalization of the standard in PVS [14], particularly the definition of the rounding function.

The FPU we have verified supports both single and double precision. It can perform floating point addition, subtraction, multiplication, division, comparison, conversion between both floating point formats, and conversion between floating point numbers and integers. Denormal numbers are handled entirely in hardware. Exceptions and wrapped exponents are computed as mandated by the standard.

The verified VAMP processor will be implemented on a Xilinx FPGA.

Project Status. As mentioned above, the FPU we have verified is embedded in the VAMP microprocessor, which is currently being verified at our institute [12]. The VAMP is a

T. Margaria and T. Melham (Eds.): CHARME 2001, LNCS 2144, pp. 325–339, 2001.

variant of the DLX [9,17], a 32 bit RISC processor based on the MIPS instruction set. The VAMP processor features a Tomasulo scheduler, delayed branch, a cache memory interface, precise interrupts, and the FPU described in this paper.

The verification of an in-order CPU core is complete, the verification of the Tomasulo out-of-order core will be completed in a few weeks [13]. The verification of the cache has just begun. The verification of the combinatorial floating point circuits and the FPU pipeline control is complete.

Our group has developed a translation tool to automatically convert the PVS specifications to Verilog HDL. This tool is already capable of translating the combinatorial floating point adder and rounding hardware to Verilog. We have used the Xilinx software to synthesize and simulate the Verilog code. In the end, we are going to implement the complete verified VAMP processor on a Xilinx FPGA Board.

All PVS specifications and proofs as well as the Verilog files are available at our web site.[1]

Paper Outline. In Sect. 2, we sketch the formalization of the IEEE standard. The implementation and verification of the combinatorial FPU is described in Sect. 3. We describe the errors that we have encountered during the verification at the end of Sect. 3. The pipelining of the combinatorial FPU is briefly discussed in Sect. 4. We conclude in Sect. 5.

Related Work. The verification of floating point algorithms and hardware using formal methods has received considerable attention over the last years.

As mentioned above, the formalization of the IEEE standard that we use is based on [6,14,17]. The notion of factorings, round decomposition, and α-equivalence is taken from [6,17]. We have formally verified this theory in [11]. Since the definition of the rounding function is informal in [6,17], we use a formal definition of rounding, which is based on Miners formalization of the standard [14].

Harrison has formalized the IEEE standard in the theorem prover HOL Light [8]. Both Miner and Harrison have no direct counterpart to the decomposition theorem and α-equivalence (cf. Sect. 2). They do not cover the actual implementation of operations or rounding.

Aagaard and Seger combine BDD based methods and theorem proving techniques to verify a floating point multiplier [1]. Chen and Bryant [3] use word-level SMV to verify a floating point adder. Exceptions and denormals are not handled in both verification projects.

Verkest et al. verify a binary non-restoring integer division algorithm [24]. Clarke et al. [5] and Ruess et al. [20] verify SRT division algorithms. Miner and Leathrum [15] verify a general class of subtractive division algorithms with respect to the IEEE standard.

Cornea-Hasegan describes the computation of division and square root by Newton-Raphson iteration in the Intel FPUs [4]. The verification is done using *Mathematica*.

[1] `http://www-wjp.cs.uni-sb.de/projects/verification/`

O'Leary et al. report on the verification of the gate level design of Intel's FPU using a combination of model-checking and theorem proving [18]. Denormals and exceptions are not covered in the paper. Their definition of rounding is not directly related to the IEEE standard.

Moore et al. have verified the AMD K5 division algorithm [16] with the theorem prover ACL2. Russinoff has verified the K5 square root algorithm as well as the Athlon multiplication, division, square root, and addition algorithms [21,22,23]. In all his verification projects, Russinoff proves the correctness of a register transfer level implementation against his formalization of the IEEE standard using ACL2. Russinoff does not handle exceptions and denormals in his publications; however, he states that he handles denormals in unpublished work (private communication). The definition of *sticky* in [16, 23] corresponds to our rounding of representatives.

2 IEEE Floating Point Arithmetic

To formally verify the correctness of a FPU, we need a formal notion of "correctness", i.e., a formalization of the IEEE standard which the FPU shall obey. In this section, we sketch the formalization of the IEEE standard used in our verification project. The formalization is primarily based on [6,14,17]. An extended version of this section is available as [11].

2.1 Factorings

We abstract IEEE numbers as defined in the standard to *factorings*. A factoring is a triple (s, e, f) with sign bit $s \in \{0, 1\}$, exponent $e \in \mathbb{Z}$, and significand $f \in \mathbb{R}_{\geq 0}$. Note that exponent range and significand precision are unbounded. The value of a factoring is

$$[\![s, e, f]\!] := (-1)^s \cdot 2^e \cdot f.$$

The standard introduces an exponent width N, from which constants $e_{min} := -2^{N-1} + 2$ and $e_{max} := 2^{N-1} - 1$ are derived. These constants are used to bound the exponent range.

We call a factoring (s, e, f) *normal* if $e \geq e_{min}$ and $1 \leq f < 2$. A factoring is called *denormal* if $e = e_{min}$ and $0 \leq f < 1$. We call a factoring an *IEEE factoring* if it is either normal or denormal.

Lemma 1. *Each $x \in \mathbb{R}_{\neq 0}$, has a unique factoring $(\hat{s}, \hat{e}, \hat{f})$ with $1 \leq \hat{f} < 2$ and $[\![\hat{s}, \hat{e}, \hat{f}]\!] = x$. Each $x \in \mathbb{R}_{\neq 0}$ has a unique IEEE factoring (s, e, f) with $[\![s, e, f]\!] = x$. Zero has two IEEE factorings $(0, e_{min}, 0)$ and $(1, e_{min}, 0)$, called $+0$ and -0, respectively.*

Let $\hat{\eta}$ and η be the functions that map (non-zero) reals x to their corresponding factorings $(\hat{s}, \hat{e}, \hat{f})$ and (s, e, f), respectively. We define $\eta(0) := (0, e_{min}, 0)$.

Lemma 2. *Let $x \in \mathbb{R}$ with $x \neq 0$ in the context of $\hat{\eta}$. It holds:*[2]

$$\hat{\eta}_e(x) = \lfloor \log_2 |x| \rfloor, \qquad \eta_e(x) = \begin{cases} \lfloor \log_2 |x| \rfloor & \text{if } x \neq 0 \text{ and } \lfloor \log_2 |x| \rfloor \geq e_{min} \\ e_{min} & \text{otherwise,} \end{cases}$$

$$\hat{\eta}_f(x) = |x| \cdot 2^{-\hat{\eta}_e(x)}, \quad \eta_f(x) = |x| \cdot 2^{-\eta_e(x)}.$$

Let P be the significand precision as defined in the standard. A significand f is called *representable*, if f has at most $P-1$ digits behind the binary point, i.e., if $2^{P-1} \cdot f \in \mathbb{N}_0$. We call an IEEE-factoring (s, e, f) *semi-representable*, if f is representable. We call an IEEE-factoring *representable*, if it is semi-representable, and furthermore $e \leq e_{max}$ holds. We call a real x (semi-)representable, if $\eta(x)$ is (semi-) representable.

We will only investigate semi-representable factorings in the following (i.e., we allow e to exceed e_{max}). In order to "round" semi-representable factorings to representable ones, one has to decide whether to round to infinity or to the largest representable number in case $e > e_{max}$. This decision depends only on the sign and the rounding mode, and therefore is trivial.

Representable numbers exactly correspond to the representable numbers as defined in the standard. Common values for (N, P) are $(8, 24)$ and $(11, 53)$, called single and double precision, respectively. The standard defines an encoding of single and double precision IEEE factorings into bit strings of length 32 and 64, respectively. To enhance the readability of our formulas in the following, we consider factorings instead of their bit string encodings.

2.2 Rounding

We proceed with the definition of the rounding function. The IEEE standard defines four rounding modes: round to nearest, up, down, and to zero. We define a function $r_{int}(\,\cdot\,, \mathcal{M})$ for each rounding mode $\mathcal{M} \in \{near, up, down, zero\}$, which rounds reals x to integers [14]:

$$r_{int}(x, up) := \lceil x \rceil \qquad r_{int}(x, near) := \begin{cases} \lfloor x \rfloor & \text{if } x - \lfloor x \rfloor < \lceil x \rceil - x \\ \lceil x \rceil & \text{if } x - \lfloor x \rfloor > \lceil x \rceil - x \\ x & \text{if } \lfloor x \rfloor = \lceil x \rceil \\ 2 \lfloor \lceil x \rceil / 2 \rfloor & \text{otherwise} \end{cases}$$

$$r_{int}(x, down) := \lfloor x \rfloor \qquad r_{int}(x, zero) := (-1)^{sign(x)} \cdot \lfloor |x| \rfloor$$

By scaling by 2^{P-1}, reals are rounded to rationals with $P-1$ fractional bits:

$$r_{rat}(x, \mathcal{M}) := 2^{-(P-1)} \cdot r_{int}(x \cdot 2^{P-1}, \mathcal{M}).$$

Further scaling with 2^e, $e := \eta_e(x)$, yields the IEEE rounding function:

$$rd(x, \mathcal{M}) := 2^e \cdot r_{rat}(x \cdot 2^{-e}, \mathcal{M}).$$

[2] $\eta_e(x)$ denotes the e-component of the factoring $\eta(x) = (s, e, f)$; analogous for other components and $\hat{\eta}$.

It is not obvious that this definition conforms with the IEEE standard. We prove the theorems stating this conformance in [11].

The rounding of reals x can be decomposed into three steps: η-computation (sometimes called pre-normalization in the literature), significand rounding, and a post-normalization.

The η-computation step computes the IEEE factoring $\eta(x)$, where x is the number to be rounded. The significand round step then rounds the significand computed in the η-computation to $P-1$ digits behind the binary point. This is formalized in the function $sigrd$:

$$sigrd(X, \mathcal{M}) := \left| r_{rat}\big((-1)^s \cdot f, \mathcal{M}\big) \right|,$$

where $X = (s, e, f)$ is an arbitrary IEEE factoring, and $\mathcal{M} \in \{near, up, down, zero\}$ is a rounding mode.

In the case that the significand round returns 0 or 2, the factoring has to be post-normalized. If the significand round returns 2, the exponent is incremented, and the significand is forced to 1; if the significand round returns 0, the sign bit is forced to 0 in order to yield $\eta(0)$. The post-normalization is defined as follows:

$$postnorm(X, \mathcal{M}) = \begin{cases} (s, e, sigrd(X, \mathcal{M})) & \text{if } 0 < sigrd(X, \mathcal{M}) < 2, \\ (s, e+1, 1) & \text{if } sigrd(X, \mathcal{M}) = 2, \\ (0, e_{min}, 0) & \text{if } sigrd(X, \mathcal{M}) = 0. \end{cases}$$

Theorem 1 (Decomposition Theorem). *For any real x, and rounding mode $\mathcal{M} \in \{near, up, down, zero\}$, it holds*

$$postnorm\big(\eta(x), \mathcal{M}\big) = \eta\big(rd(x, \mathcal{M})\big).$$

The benefit of having the decomposition theorem is that it simplifies the design and verification of the rounder (cf. Sect. 3.3).

2.3 Exceptions

The IEEE standard defines five exceptions: invalid operation (INV), division by zero (DIVZ), overflow (OVF), underflow (UNF), and inexact result (INX). Our formalization of these exceptions is taken from [17], as the implementation in the actual hardware is. For example, the inexact result exception is formalized as

$$INX(x, \mathcal{M}) := \big(x \neq rd(x, \mathcal{M})\big),$$

where x is the infinitely precise result of a floating point operation. The definition of the other exceptions is similiar.

Lemma 3. *Let x be a real. It holds $INX(x, \mathcal{M}) \iff \eta_f(x) \neq sigrd(\eta(x), \mathcal{M})$.*

In case of underflow or overflow with the respective trap handler enabled, the standard mandates scaling the result into the representable range, and passing the scaled result to the trap handler. This is called *wrapped exponent*. The handling of wrapped exponents is as in [17].

2.4 α-Equivalence

We now define the concept of α-equivalence and α-representatives [17]. As we will see in theorem 3, this concept is a very concise way to speak about sticky-bit computations.

Let α be an integer. Two reals x and y are said to be *α-equivalent* ($x \equiv_\alpha y$), if $x = y$ or if there exists some $q \in \mathbb{Z}$ with $q \cdot 2^\alpha < x, y < (q+1) \cdot 2^\alpha$, i.e., if both x and y lie in the same singleton $\{2^\alpha\}$ or in the same open interval between two consecutive integral multiples of 2^α. Clearly, if such an q exists, it must be $q_\alpha(x) := \lfloor x \cdot 2^{-\alpha} \rfloor$. The α-representative of x is defined as

$$[x]_\alpha = \begin{cases} x & x = q_\alpha(x) \cdot 2^\alpha \\ \left(q_\alpha(x) + \frac{1}{2}\right) \cdot 2^\alpha & \text{otherwise,} \end{cases}$$

i.e., if x is an integral multiple of 2^α, the representative of x is x itself, and the midpoint of the interval between the surrounding multiples of 2^α otherwise. The following lemma summarizes some important facts:

Lemma 4. *Let $x, y \in \mathbb{R}$, and $\alpha, k \in \mathbb{Z}$.*

1. $\equiv_\alpha$ *is an equivalence relation,*
2. $x \equiv_\alpha [x]_\alpha$,
3. $x \equiv_\alpha y \iff [x]_\alpha = [y]_\alpha$, *(representative equivalence)*
4. $x \equiv_\alpha y \iff -x \equiv_\alpha -y$, *and* $[-x]_\alpha = -[x]_\alpha$, *(negative value)*
5. $x \equiv_\alpha y \iff 2^k \cdot x \equiv_{\alpha+k} 2^k \cdot y$, *and* $[2^k \cdot x]_{\alpha+k} = 2^k \cdot [x]_\alpha$, *(scaling)*
6. $x \equiv_\alpha y \iff x + k \cdot 2^\alpha \equiv_\alpha y + k \cdot 2^\alpha$, *(translation)*
7. $x \equiv_\alpha y \implies x \equiv_{\alpha+k} y$ *if* $k \geq 0$, *(coarsening)*
8. $x = 0 \iff x \equiv_\alpha 0 \iff [x]_\alpha = 0$, *(zero value)*
9. $0 < x < 2^\alpha \implies [x]_\alpha = 2^{\alpha-1}$. *(small value)*

The following theorem describes the computation of IEEE-factorings corresponding to representatives:

Theorem 2. *Let $x \in \mathbb{R}$, $(s, e, f) = \eta(x)$, and $p \in \mathbb{N}$. The IEEE-factoring of $[x]_{e-p}$ can be computed by computing the representative $[f]_{-p}$ of f:*

$$\eta([x]_{e-p}) = (s, e, [f]_{-p}).$$

Next, we show that the representative of f can be computed by a *sticky-bit computation*. Let $f \geq 0$ be a real in binary format $f_k, \ldots, f_0, f_{-1}, \ldots, f_{-l} \in \{0,1\}^{k+l+1}$ such that $f = \sum_{i=-l}^{k} f_i \cdot 2^i$. Let $p \in \mathbb{Z}$, $k \geq -p > -l$. The $(-p)$-sticky-bit of f is the logical OR of all bits $f_{-p-1}, \ldots, f_{-l}$:

$$sticky_{-p}(f) := \bigvee_{i=-l}^{-p-1} f_i.$$

Theorem 3. *The representative $[f]_{-p}$ of f can be computed by replacing the less significant bits by the sticky bit:*

$$[f]_{-p} = \left(\sum_{i=-p}^{k} f_i \cdot 2^i \right) + 2^{-p-1} \cdot sticky_{-p}(f)$$

Theorems 2 and 3 together allow a very efficient computation of representatives (respectively their IEEE-factorings) by or-ing the less significant bits in an OR-tree, and replacing them by the sticky-bit. This technique is well known [7], but introducing the formalism with α-representatives allows for a very concise argumentation about these sticky-computations.

The valuable property of α-representatives is that rounding x and its representative $[x]_{e-P}$ yields the same result:

Theorem 4. *Let x be an arbitrary real, $(s, e, f) = \eta(x)$, and $\mathcal{M}$ be a rounding mode. It holds*

$$rd(x, \mathcal{M}) = rd([x]_{e-P}, \mathcal{M}).$$

The significand round can be performed on the representative $[f]_{-P}$ of f:

$$sigrd\big((s, e, f), \mathcal{M}\big) = sigrd\big((s, e, [f]_{-P}), \mathcal{M}\big).$$

Corollary 1. *By lemma 4.7, theorem 4 also holds for any $\alpha \leq e - P$:*

$$rd(x, \mathcal{M}) = rd([x]_{\alpha}, \mathcal{M}).$$

As a consequence, one can detect the OVF, UNF and INX exceptions by analysis of the representative of x:

Corollary 2. *Again, let $\alpha \leq e - P$. It holds*

$$INX(x, \mathcal{M}) \iff INX([x]_{\alpha}, \mathcal{M}),$$

and analogously for UNF and OVF.

Corollaries 1 and 2 facilitate the verification of the FPU in that they allow the decomposition of the FPU into computation units and a rounding unit. The computation unit performs the operation, e.g., a multiplication, and delivers a result to the rounder which is α-equivalent to the infinitely precise result of the operation (with the appropriate α). The rounder therefrom computes the correctly rounded result and the exception signals. During the verification of the computation units, the rounding algorithm and exceptions do not matter, and during the verification of the rounder, the operations do not matter. In fact, using the α-equivalence interface, the first author has verified the addition unit independently of the rounding unit, which was verified at the same time by the second author.

2.5 Correctness of the FPU

The standard requests that every floating point operation shall return a result obtained as if one first computed the exact result with infinite precision, and then rounded this exact result. We therefore call the FPU correct, if for each operation $\circ \in \{+, -, \times, \div\}$ on all representable numbers x and y, the FPU returns the IEEE bit string encoding of the factoring

$$\eta\big(rd(x \circ y, \mathcal{M})\big).$$

Furthermore, the FPU must compute the correct exception signals.

3 Verifying the VAMP FPU

Figure 1 shows the top-level schematic of the FPU. Floating point operands are passed into the floating point unpacker FPUnpack, integer operands are passed into the fixed point unpacker FXUnpack. Integer operands are used in conversion from integers to floating point numbers.

The floating point unpacker converts the operands from the IEEE format into a more convenient format. It translates the exponent from biased integer into two's complement format, and reveals the hidden significand bit. In case of multiplication and division, the unpacker normalizes denormal significands and adjusts the exponents accordingly. Single and double precision operands are embedded into the same internal format. Furthermore, the floating point unpacker handles special cases such as operations on $\pm\infty$, NaN, zeros etc.

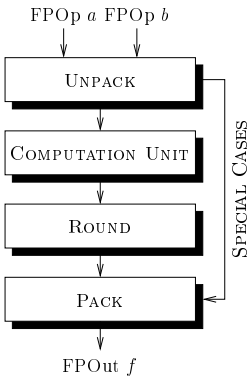

Fig. 1. VAMP FPU

From the floating point unpacker, non-special operands are fed into one of the computation units, namely addition and subtraction, multiplication and division, comparison, and the conversion unit.

Let $x = a \circ b$ with $\circ \in \{+, -, \times, \div\}$ be the exact result of an operation to be performed by the functional units. Instead of feeding x into the rounder, the functional units compute a factoring (s_i, e_i, f_i) which rounds to the same floating point number as x does (cf. corollary 1):

$$[\![s_i, e_i, f_i]\!] \equiv_\alpha x \text{ with } \alpha \leq \eta_e(x) - P.$$

This factoring needs not to be an IEEE-factoring. The rounder computes the floating point result and the exceptions from (s_i, e_i, f_i). After rounding, the circuit Pack transforms the rounded floating point result into the IEEE format.

Together with the conversion unit, the rounder is capable to convert between single and double precision floating point numbers, and to convert floating point numbers into the integer format.

The comparison unit outputs a flag indicating the result of the comparision performed. We have implemented and verified the comparison and conversion units, but do not discuss this further in this paper.

In the following sections, we describe the construction and verification of the computation units and the rounder. Exemplarily, we prove the correctness of the addition algorithm. The proof is a transcript of the actual PVS proof using standard mathematical notation instead of PVS notation for the sake of readability. The proof is similar to the proof given in [17] which, however, has larger gaps then the proof given here. *The significance of the proof presented here is that it is formally verified.*

We do not describe the proofs of the other components due to lack of space.

3.1 Adder

The floating point adder has IEEE-factorings (s_a, e_a, f_a) and (s_b, e_b, f_b) as inputs. The adder therefrom computes the sum (or difference in case of subtraction), (s_s, e_s, f_s),

which is fed into the rounder. Let $a := [\![s_a, e_a, f_a]\!]$ and $b := [\![s_b, e_b, f_b]\!]$ be the values of the operands.

Since the unpacker embeds single and double precision inputs into the same internal format, we do not distinguish between single and double precision in the adder. The final rounding stage will round the result to the appropriate precision. We therefore fix $P = 53$ in this section.

To simplify the description, we assume that the adder shall perform an addition. If it shall perform a subtraction, b is replaced with $-b$ by inverting the sign bit s_b.

The exact sum is denoted by $S := a + b$. We assume that $a \neq 0, b \neq 0$, and $S \neq 0$, since these are special cases handled by the unpacker.

Addition Algorithm. The informal description of the addition algorithm is

1. The larger exponent of e_a and e_b is the result's exponent e_s.
2. Assume that $e_a \geq e_b$, otherwise swap a and b.
3. Align the significand f_b by shifting it $\delta := e_a - e_b$ to the right: $f'_b := 2^{-\delta} \cdot f_b$.
4. Add both significands with respect to the sign bits: $f'_s := (-1)^{s_a} \cdot f_a + (-1)^{s_b} \cdot f'_b$. The absolute value of f'_s is the result's significand, $f_s := |f'_s|$.
5. The result's sign s_s can be computed as $s_s := s_a \oplus (f'_s < 0)$.

As the alignment shift in step 3 would require a shifter of size $e_{max} - e_{min} \approx 2^{11}$, this is impractical. We therefore approximate the shifted significand by its $(-P-1)$-representative:

$$f'_b := [2^{-\delta} \cdot f_b]_{-(P+1)}.$$

This does not change the result of the operation, since both values are rounded to the same value by the rounder:

$$rd(S, \mathcal{M}) = rd\big(2^{e_a} \cdot \big((-1)^{s_a} \cdot f_a + (-1)^{s_b} \cdot f'_b\big), \mathcal{M}\big).$$

From corollary 1 we know that it suffices to supply a value to the rounder that is α-equivalent to the sum S, where $\alpha \leq \eta_e(x) - P$ must hold. From lemma 2 we know that $\hat{\eta}_e(x) \leq \eta_e(x)$. Therefore it suffices to prove the following theorem:

Theorem 5. *Let $\hat{e} := \hat{\eta}_e(S)$. It holds*

$$S \equiv_{\hat{e}-P} 2^{e_a} \cdot \big((-1)^{s_a} \cdot f_a + (-1)^{s_b} \cdot f'_b\big). \tag{1}$$

Proof. By definition, we have

$$\begin{aligned} S = [\![s_a, e_a, f_a]\!] + [\![s_b, e_b, f_b]\!] &= (-1)^{s_a} \cdot 2^{e_a} \cdot f_a + (-1)^{s_b} \cdot 2^{e_b} \cdot f_b \\ &= 2^{e_a} \cdot \big((-1)^{s_a} \cdot f_a + (-1)^{s_b} \cdot 2^{-\delta} \cdot f_b\big). \end{aligned}$$

The claim (1) is therefore equivalent to

$$2^{e_a} \cdot \big((-1)^{s_a} \cdot f_a + (-1)^{s_b} \cdot 2^{-\delta} \cdot f_b\big) \equiv_{-(P-\hat{e})} 2^{e_a} \cdot \big((-1)^{s_a} \cdot f_a + (-1)^{s_b} \cdot f'_b\big). \tag{2}$$

Assume $\delta < 2$. Since f_b is a representable significand with at most $P-1$ fractional digits, it holds

$$f'_b = [2^{-\delta} \cdot f_b]_{-(P+1)} = 2^{-\delta} \cdot f_b.$$

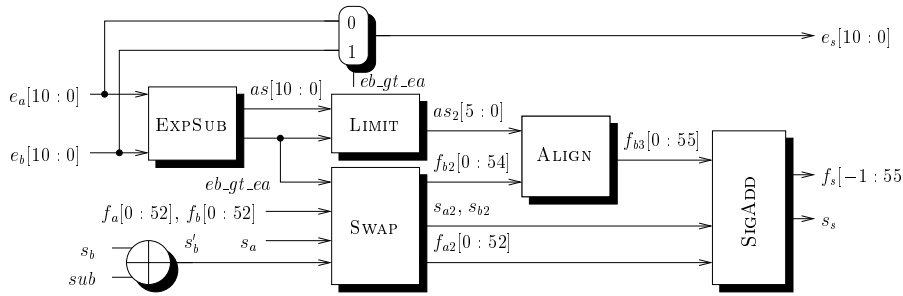

Fig. 2. The adder

This proves (2) for this case. Now let $\delta \geq 2$. By the definition of f'_b, we have

$$2^{-\delta} \cdot f_b \equiv_{-(P+1)} f'_b.$$

Successively rewriting with lemma 4 (parts 4,5,6) yields

$$(-1)^{s_b} \cdot 2^{e_a - \delta} \cdot f_b \equiv_{e_a-(P+1)} (-1)^{s_b} \cdot 2^{e_a} \cdot f'_b,$$
$$(-1)^{s_a} \cdot 2^{e_a} \cdot f_a + (-1)^{s_b} \cdot 2^{e_a-\delta} \cdot f_b \equiv_{e_a-(P+1)} (-1)^{s_a} \cdot 2^{e_a} \cdot f_a + (-1)^{s_b} \cdot 2^{e_a} \cdot f'_b.$$

Now assume $\hat{e} - P \geq e_a - (P+1)$. Lemma 4.7 then implies

$$(-1)^{s_a} \cdot 2^{e_a} \cdot f_a + (-1)^{s_b} \cdot 2^{e_a-\delta} \cdot f_b \equiv_{\hat{e}-P} (-1)^{s_a} \cdot 2^{e_a} \cdot f_a + (-1)^{s_b} \cdot 2^{e_a} \cdot f'_b.$$

This proves (2). It remains to show that $\hat{e} - P \geq e_a - (P+1)$. By lemma 2, this is equivalent to

$$\hat{e} = \hat{\eta}_e(S) = \left\lfloor \log_2 \left| (-1)^{s_a} \cdot 2^{e_a} \cdot f_a + (-1)^{s_b} \cdot 2^{e_b} \cdot f_b \right| \right\rfloor \geq e_a - 1. \tag{3}$$

Since the operands are IEEE factorings, $f_b < 2$. Since $\delta \geq 2$, we have $2^{-\delta} \leq \frac{1}{4}$. Together, this yields

$$2^{-\delta} \cdot f_b < \tfrac{1}{2}.$$

Since $e_b \geq e_{min}$, and $\delta = e_a - e_b \geq 2$, we know that $e_a > e_{min}$, and hence $f_a \geq 1$. We now have

$$\left| (-1)^{s_a} \cdot f_a + (-1)^{s_b} \cdot 2^{-\delta} \cdot f_b \right| \geq \tfrac{1}{2}.$$

Multiplying with 2^{e_a} and taking logarithms yields (3). The floor brackets $\lfloor \cdot \rfloor$ may be dropped since $e_a - 1$ is integer. □

The representative $[2^{-\delta} \cdot f_b]_{-(P+1)}$ can be computed with a shift distance limited to B (we later fix $B = 63$). This avoids the need for a very large shifter.

Lemma 5. *For $B > P$, let $\delta' = \min(\delta, B)$. It holds*

$$[2^{-\delta} \cdot f_b]_{-(P+1)} = [2^{-\delta'} \cdot f_b]_{-(P+1)}$$

Proof. The case $\delta \leq B$ is trivial. Let $\delta > B > P$. Then by lemma 4.9, it holds

$$[2^{-\delta'} \cdot f_b]_{-(P+1)} = 2^{-(P+2)} = [2^{-\delta} \cdot f_b]_{-(P+1)}.$$ □

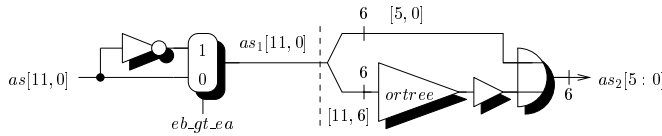

Fig. 3. Circuit LIMIT

Adder Sardware. The adder (Fig. 2) is a straightforward implementation of the described algorithm using basic components [2]. If a subtraction is to be performed, s_b is negated, yielding s'_b. Circuit EXPSUB computes the difference $as := e_a - e_b$ and the flag $eb\_gt\_ea := (e_b > e_a)$. The result's exponent e_s is selected by a multiplexer. Circuit SWAP swaps a and b in case $e_b > e_a$. The shift distance is limited in circuit LIMIT to $B := 63$. Circuit ALIGN performs the actual alignment shift. It primaly consists of a 64-bit shifter and a sticky computation, which collects the bits shifted out during the alignment. Circuit SIGADD performs the addition, i.e., steps 4 and 5 from our informal description.

The verification of the adder is straightforward: prove the correctness of the subcircuits, and combine them using the above lemma and theorem.

Verifying the Gate Level. As an example for the detail level our proofs operate on, we present the LIMIT circuit (Fig. 3) that calculates the shift distance as_2 for circuit ALIGN. First, an approximation of the absolute value of the shift distance $as = e_a - e_b$ is computed.

$$as_1 = \begin{cases} as & \text{if } as \geq 0 \\ -as + 1 & \text{if } as < 0. \end{cases}$$

If one of the high order bits $as_1[10:6]$ is set, then $as_1 > 63$. In this case, the low order bits of as_1 are forced to one by the OR-gates. Otherwise, the shift distance as_1 is unchanged. It holds

$$as_2 = \min\{as_1, 63\}.$$

Both statements are easily verified in PVS.

In case $e_b > e_a$, the described computation introduces an error of 1 in the shift distance. This is compensated by pre-shifting the significand by one bit in circuit SWAP in this case. This detour is done to reduce the cycle time of the adder. The approximation of the absolute value can be computed with the delay of a single inverter. If one computed the exact absolute value of as, one would introduce the delay of an incrementer that would increase the length of the critical path of the adder.

3.2 Multiplier and Divider

The product of two floating point numbers can be computed by adding the exponents and multiplying the significands. The less significant bits of the significand's product are then compressed by means of a sticky bit computation. The so computed representative of the product is then passed to the rounder. Implementation and verification of this algorithm are straightforward.

In order to compute the quotient of two floating point numbers, one subtracts the exponents, and computes the quotient of the significands. The latter is the interesting part of the MULT/DIV unit.

Let f_a and f_b be the two significands. We may assume $1 \leq (f_a, f_b) < 2$, since the unpacker provides normalized significands. In our FPU, we use Newton-Raphson iteration to compute an approximation of f_b^{-1}. We start with an initial approximation x_0 with $0 < |x_0 - f_b^{-1}| < 2^{-8}$, which is loaded from a lookup table with 256 entries. In PVS, the lookup table is defined as a function mapping addresses $a \in \{0, \ldots, 255\}$ to bitvectors $b \in \{0,1\}^8$. We have verified the content of the lookup table by automatically checking all 256 entries. The verification takes 5 minutes on a 500 MHz AMD Athlon.

The analysis of the actual Newton-Raphson iteration and the following computation of the representative $[f_a/f_b]_{-P}$ of the significand quotient is described very detailed in [17]. The translation of the proofs to PVS is therefore straightforward.

Before passing the result to the rounder, the significand is left-shifted by one bit to yield a significand in the range $[1, 4)$ as required by the rounder. The exponent is adjusted accordingly.

3.3 Rounder

Let x be the exact result of an operation, and let (s_i, e_i, f_i) be the input factoring to the rounder. This factoring is not necessarily an IEEE-factoring. Let $(s, e, f) = \eta(x)$ be the IEEE factoring of x. The rounder specification requires the input factoring to satisfy $[\![s_i,e_i,f_i]\!] \equiv_\alpha x$, where $\alpha \leq e - P$. Here, P is the precision of the operation's destination format.

The rounding unit is decomposed into the η-shifter, the representative computation, the significand round, and the post-normalization stage (Fig. 4). The η-shifter computes an IEEE-factoring (s_n, e_n, f_n) with $s_n = s, e_n = e, f_n \equiv_{-P} f$. Two cases have to be distinguished:

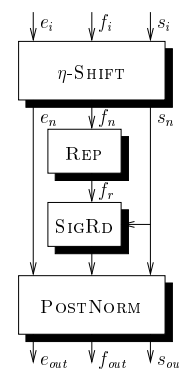

Fig. 4. Rounder

1. In case of an addition/subtraction, the exponent e_i satisfies $e_i \geq e_{min}$ by construction (Sect. 3.1). However, the significand f_i lies in the interval $(0, 4)$, and can – due to cancellation – be less than 1 even if $e_i > e_{min}$. In the latter case, the significand has to be shifted left.

2. In case of multiplication and division, the input significand f_i lies in the interval $[1, 4)$, since the inputs to the multiplier/divider were normalized by the unpacker. The exponent e_i, however, does not necessarily satisfy $e_i \geq e_{min}$.[3] In the case where $e_i < e_{min}$, the significand f_i has to be shifted right by $e_{min} - e_i$ digits. Since this shift could be very far, the shift distance is limited similarly to the adder alignment shift explained in Sect. 3.1.

The η-shifter outputs f_n with 128 binary digits. The circuit REP computes the representative $f_r := [f_n]_{-P}$. This is done using an OR-tree, as suggested by theorem 3. We then have $(s_n, e_n, f_r) = \eta([x]_{e-P})$ by theorem 2.

The next circuits SIGRD and POSTNORM exactly correspond to the functions $sigrd$ and $postnorm$ from Sect. 2.2. The significand round on f_r is performed by investigation

[3] For example, the multiplication $2^{e_{min}} \times 2^{e_{min}}$ generates inputs $(s_i, e_i, f_i) = (0, e_{min} + e_{min}, 1)$ to the rounder. Note that $e_{min} < 0$, and therefore $2 \cdot e_{min} < e_{min}$.

of the 3 least significant bits of f_r, and either chopping or incrementing the higher order bits [17]. If this effectively changes the significand, then the inexact result exception INX is signalled according to lemma 3.

The post-normalization increments the exponent and forces the significand to 1 if normalization is necessary.

Theorems 1 and 4 together imply the correctness of the rounder:

$$(s_{out}, e_{out}, f_{out}) = \eta\big(rd(x, \mathcal{M})\big).$$

After rounding, the circuit PACK outputs the IEEE bit string encoding of this factoring. In case of an untrapped overflow, however, the circuit PACK outputs either the format's maximal value, or infinity, depending on the sign and the rounding mode.

The correctness of the unpacker, the computation units, the rounder, and the packer together imply the correctness of the whole FPU.

3.4 Errors Encountered

We briefly describe some of the errors in [17] that we have encountered during the verification of the FPU in PVS:

The specification of the rounder interface (pg. 392) is wrong. There it is required that an overflow does not occur if a denormal significand f_i is fed into the rounder, i.e., $f_i < 1 \Rightarrow \neg OVF(x, \mathcal{M})$. This is necessary to detect overflows correctly (pg. 397). However, the requirement is not strong enough: it must hold that the input exponent e_i is less than e_{max} in case of a denormal input significand, i.e., $f_i < 1 \Rightarrow e_i \leq e_{max}$. Otherwise, the proof on page 397 fails.

The divider does not obey the rounder specification (neither the old nor the new one). A division of $1 \cdot 2^{e_{max}}$ by $(2 - 2^{-P+1}) \cdot 2^{e_{min}}$ overflows, but the input significand into the rounder $f_i \approx 1/2$ is denormal. This bug can be fixed by left-shifting f_i by 1 and appropriately adjusting the exponent e_i in case of divisions (cf. Sect. 3.2).

On page 400, a carry-in is fed into a compound adder, although compound adders do not feature a carry-in. A similar error was found in the exponent addition circuit in the multiplier (pg. 383).

In circuit SIGRD (pg. 406), chopping the significand in single precision mode leaves non-zero digits after the least significand bit. The claims in Sect. 8.4.5 are therefore wrong. This can be fixed by tying the bits after the least significand bit to zero.

In the significand round, the circuit for the decision whether to chop or to increment the significand is wrong (pg. 407). The XOR has to be replaced by an XNOR gate.

In the adder, the computation of the sign bit is wrong (pg. 371).

The proofs in [17] partly have large gaps. These gaps had to be filled during the verification in PVS. Most proof gaps could be filled without revealing errors in [17], but some proof gaps hid errors, e.g., the errors listed above. Having formally verified the proofs in PVS ultimately gives us the certainty that the design of the FPU is correct – under the assumption that PVS is sound.

4 FPU Control

So far we have verified combinatorial circuits. In order to implement the FPU in hardware with reasonable cycle time, one has to insert pipelining registers. Since multipliers are very expensive, one cannot fully pipeline the iterative Newton-Raphson algorithm. A loop has to be incorporated into the pipeline structure to reuse the multiplier in each iteration. This saves hardware costs, but considerably complicates control and the correctness proof.

In [17], the FPU is integrated into an in-order variant of the DLX-processor. In our verification project, the FPU will be integrated into a Tomasulo based out-of-order DLX-variant. It is therefore necessary to design a new control automaton for the FPU in order to exploit the benefits of the out-of-order scheduler.

After pipelining, the FPU has a variable latency, and operations are finished out-of-order. The latency of the FPU is 1 cycle for comparison and for operations involving special operands. It is 5 cycles for addition, subtraction, and multiplication. The division unit has latency 16 and 20 cycles in single and double precision, respectively. Two divisions can be performed interleaved without increased latency.

We have verified the new FPU control using a combination of PVS's modelchecking and theorem proving capabilities We omit the control implementation details here, since they are not specific to FPUs.

5 Summary and Future Work

We have formally verified a fully IEEE compliant floating point unit. The supported operations are addition, subtraction, multiplication, division, comparison, and conversions. The FPU handles denormals and exceptions as required by the IEEE standard. The hardware has been verified on the gate level with respect to a formal description of the IEEE standard using the theorem prover PVS.

The proofs in PVS used paper proofs from [17] as guidelines. However, some of the proofs in [17] were erroneous, and most proofs had gaps needed to be filled in PVS. Those gaps hid errors in the design in [17]. Having formally verified the proofs (and filled the proof gaps) in PVS gives us the certainty that now the hardware is correct with respect to its specification.

To the best of our knowledge, this is the first time that a floating point unit that supports addition/subtraction, multiplication/division, comparison, conversions, denormals, and exceptions in hardware has been formally verified on the gate level, and the designs and proofs scripts are made publicly available.

The amount of work needed to develop the PVS hardware description and proofs was roughly a year for each of the authors. Since theorem proving strongly profits from experience, we think we would succeed in at most half the time now on a comparable project.

We are currently working on the integration of the FPU into the VAMP processor and the translation of the VAMP to Verilog. The VAMP processor including the FPU will then be implemented on a Xilinx FPGA.

Acknowledgements. The authors would like to thank Sven Beyer, Daniel Kröning, Dirk Leinenbach, Wolfgang Paul, and Jochen Preiß for valuable discussions.

References

1. M. D. Aagaard and C.-J. H. Seger. The formal verification of a pipelined double-precision IEEE floating-point multiplier. In *ICCAD*, pages 7–10. IEEE, Nov. 1995.
2. C. Berg, C. Jacobi, and D. Kroening. Formal verification of a basic circuits library. In *IASTED International Conference on Applied Informatics*. ACTA Press, 2001.
3. Y.-A. Chen and R. E. Bryant. Verification of floating point adders. In *CAV'98*, volume 1427 of *LNCS*, 1998.
4. M. Cornea-Hasegan. Proving the IEEE correctness of iterative floating-point square root, divide, and remainder algorithms. *Intel Technology Journal*, Q2, 1998.
5. E. M. Clarke, S. M. German, and X. Zhao. Verifying the SRT division algorithm using theorem proving techniques. In *CAV'96*, volume 1102 of *LNCS*, 1996.
6. G. Even and W. Paul. On the design of IEEE compliant floating point units. In *Proceedings of the 13th Symposium on Computer Arithmetic*. IEEE Computer Society Press, 1997.
7. D. Goldberg. *Computer Arithmetic*. In [9], 1996.
8. J. Harrison. A machine checked theory of floating point arithmetic. In *TPHOL '99*, volume 1690 of *LNCS*. Springer, 1999.
9. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.
10. Institute of Electrical and Electronics Engineers. *ANSI/IEEE standard 754–1985, IEEE Standard for Binary Floating-Point Arithmetic*, 1985.
11. C. Jacobi. A formally verified theory of IEEE rounding. Unpublished, available at `www-wjp.cs.uni-sb.de/~cj/ieee-lib.ps`, 2001.
12. C. Jacobi and D. Kroening. Proving the correctness of a complete microprocessor. In *GI Jahrestagung 2000*. Springer, 2000.
13. D. Kroening. *Formal Verification of Pipelined Microprocessors*. PhD thesis, Saarland University, Computer Science Department, 2001.
14. P. S. Miner. Defining the IEEE-854 floating-point standard in PVS. Technical Report TM-110167, NASA Langley Research Center, 1995.
15. P. S. Miner and J. F. Leathrum. Verification of IEEE compliant subtractive division algorithms. In *FMCAD-96*, volume 1166 of *LNCS*, pages 64–, 1996.
16. J Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the AMD5K86 floating point division program. *IEEE Transactions on Computers*, 47(9):913–926, 1998.
17. S. M. Mueller and W. J. Paul. *Computer Architecture. Complexity and Correctness*. Springer, 2000.
18. J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger. IA-64 floating point operations and the IEEE standard for binary floating-point arithmetic. *Intel Technology Journal*, Q4, 1999.
19. S. Owre, N. Shankar, and J. M. Rushby. PVS: A prototype verification system. In *CADE 11*, volume 607 of *LNAI*, pages 748–752. Springer, 1992.
20. H. Ruess, N. Shankar, and M. K. Srivas. Modular verification of SRT division. In *CAV'96*, volume 1102 of *LNCS*, 1996.
21. D. M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998.
22. D. M. Russinoff. A mechanically checked proof of correctness of the AMD K5 floating point square root microcode. *Formal Methods in System Design*, 14(1):75–125, Jan. 1999.
23. D. M. Russinoff. A case study in formal verification of register-transfer logic with ACL2: The floating point adder of the AMD Athlon processor. In *FMCAD-00*, volume 1954 of *LNCS*. Springer, 2000.
24. D. Verkest, L. Claesen, and H. De Man. A proof on the nonrestoring division algorithm and its implementation on an ALU. *Formal Methods in System Design*, 4, 1994.