Formal Methods in System Design, 26, 227–266, 2005

Formal Verification of the *VAMP* Floating Point Unit*

CHRISTIAN JACOBI† cjacobi@de.ibm.com
IBM Deutschland Entwicklung GmbH, Processor Development II, D-71032 Boeblingen, Germany

CHRISTOPH BERG‡ cb@cs.uni-sb.de
Saarland University, Computer Science Department, D-66123 Saarbrücken, Germany

Abstract. We report on the formal verification of the floating point unit used in the *VAMP* processor. The dual-precision FPU is IEEE compliant and supports denormals and exceptions in hardware. The supported operations are addition, subtraction, multiplication, division, comparison, and conversions.

We have formalized the IEEE standard 754. The formalization is supplemented by a rich theory of rounding, which includes notations and theorems facilitating the verification of the actual hardware. The theory of rounding enables the separation of the hardware into smaller modules which can be verified individually. Each module is verified on the gate level against a formal specification. The combination of these formal specifications, together with the theorems from the theory of rounding, yield the overall correctness of the FPU, i.e., theorems stating that the gate-level hardware complies with the high-level formalization of the IEEE standard. The verification is done completely in the theorem prover PVS.

We further report on the implementation and test of the verified FPU on a Xilinx FPGA.

Keywords: floating point unit, formal verification, IEEE standard 754, theorem proving, PVS

1. Introduction

Our institute at Saarland University is working on the formal verification of a complete microprocessor called *VAMP*. Part of this microprocessor is a floating point unit (FPU) which is compliant to the IEEE standard 754 [28]. This article describes the verification of the FPU in the theorem prover PVS [43].

The FPU 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. The rounded result, exceptions and wrapped exponents are computed as mandated by the IEEE standard.

The FPU design is taken from the textbook on computer architecture by Müller and Paul [41], where the designs are described on the level of single gates. The correctness of the circuits is proved in a mathematical textbook fashion. We have specified and verified the designs on the gate level in PVS. The paper-and-pencil proofs in [41] served as guidelines

*This article is a revised and extended version of [6, 30].
†The work reported in this article was done while the author was affiliated with Saarland University.
‡Supported by the DFG graduate program *Leistungsgarantien für Rechnersysteme*.

for the formal proofs. Only small changes to the designs were necessary—some due to errors in [41], 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. This formalization is supplemented by a rich theory of rounding which facilitates the verification of the actual hardware. The formalization of the standard and the theory of rounding are based on [21, 38, 41].

The main ingredients of the theory of rounding are the notion of factorings, α-equivalence[1] and round decomposition. All three are based on [21, 41]. Factorings are an abstraction of bitvector-level floating point numbers. This abstraction allows us to argue about real numbers instead of bitvectors in large parts of the verification. The gate-level verification of the hardware establishes a link between the floating-point numbers represented by bitvectors and their respective factorings. Higher levels in the verification then only argue about real numbers.

The concept of α-equivalence enables the subdivision of the FPU hardware into operation-specific computation units (i.e., addition/subtraction or multiplication/division units) and a general purpose rounder. Each hardware module is verified separately against a formal specification. The specifications are then combined using the notion of α-equivalence to establish the overall correctness of the FPU.

The round decomposition theorem suggests a split of the rounding hardware into three parts, which further simplifies the verification of the rounding hardware.

The actual FPU hardware is built from a library of general purpose circuits such as adders, shifters, and decoders [7], which provide the basic building blocks for our circuits. The correctness statements of these general purpose circuits are combined to form the correctness statements of the modules within the FPU. The correctness statements of the individual modules are then further combined with the theory of rounding to yield the overall correctness of the FPU.

Hence, there can be seen two complementary approaches to the verification of our FPU: first, the FPU modules are built from general purpose, formally verified circuits. Most of the gates within the FPU belong to such a general purpose circuit, and therefore are covered by the correctness of these circuits. On the other end, the theory of rounding enables the subdivision of the complete FPU into smaller units which can be verified separately. The separation eases the verification of each unit. Furthermore, the separation enables the concurrent verification of the units by several people.

The hardware is specified in the functional PVS language. Each module corresponds to a function in PVS which maps the inputs and current state of the module to the outputs and next state. Our group has developed a translation tool which automatically converts the hardware from PVS to Verilog HDL. We have synthesized the Verilog description of the FPU, and have implemented and tested it on a Xilinx FPGA. The FPU worked on the first try.

1.1. The VAMP project

The FPU we have verified is embedded in the *VAMP* microprocessor which has been verified at our institute [9]. The *VAMP* is a variant of the DLX [26], a 32-bit RISC processor based on the MIPS instruction set. The *VAMP* processor features a Tomasulo scheduler, delayed branch, precise interrupts, and the FPU described here. The *VAMP* cache memory interface has separate instruction and data caches. Consistency is maintained by means of snooping. A rudimentary memory management unit with translation, protection, and a simple TLB is currently being verified and integrated into the *VAMP*.

Using our *pvs2hdl* tool, we have translated the FPUs and the complete *VAMP* processor including the caches to Verilog. We have synthesized the Verilog code and implemented the *VAMP* on a Xilinx FPGA located on a PCI board in a host PC. The implementation is capable of running C programs that have been compiled using gcc and glibc ports. I/O is done via the host PC.

All PVS specifications and proofs of the *VAMP* project as well as the translation tool and the Verilog files are available at the project web site [48].

1.2. Outline

In Section 2, we present the formalization of the IEEE standard. The implementation and verification of the combinatorial FPU is described in Section 3. We describe the errors in the textbook [41] we have encountered during the verification at the end of Section 3. The pipelining of the combinatorial FPU is briefly discussed in Section 4. We describe the implementation of the FPU on a Xilinx FPGA in Section 5. There we also describe the discrepancy between Intel's and AMD's FPUs compared to our FPU encountered while testing our implementation. We discuss related work in Section 6. We conclude in Section 7.

2. Theory of IEEE rounding

This section presents the theory of rounding which has been used in the verification of the floating point hardware. The theory consists of a formalization of the IEEE standard 754 [28] (often simply called "the standard" in this article), and notations and theorems facilitating the verification of the actual floating point hardware.

The theory presented in this section is primarily based on the work of Even and Paul [21] and Müller and Paul [41, Chap. 7]. The paper-and-pencil proofs in [41] served as guidelines for the formal proofs. Here, their work is extended in that we formally verify the theory in PVS. The definition of the rounding function in this section is based on Miner's formalization of the IEEE standard in PVS [38].

In Section 2.1, we define the notion of *factorings*. Factorings are a numerical abstraction of bitvector-represented floating point numbers. The abstraction eases the verification, since one may argue about numbers instead of single bits and bitvectors. We proceed in Section 2.2 with the definition of the rounding function and the proof of the *decomposition theorem of rounding*, which allows us to split the rounding process into three steps. This enables a decomposition of the actual rounding hardware in a similar fashion, which in turn simplifies the design and the verification of the rounding hardware (see Section 3). In Section 2.3 we define the floating point exceptions and exponent wrapping.

The concept of α-equivalence is defined in Section 2.4. α-equivalence is a concise way to talk about sticky-bit computations. The real numbers are partitioned into equivalence classes by means of α-equivalence. The salient property of this partitioning is that for appropriate α, α-equivalent numbers are rounded to the same floating point number, which is proved in Section 2.5. α-equivalence enables a decomposition of the FPU into computation units (e.g., adder, divider) and a rounding unit. The computation unit delivers a result to the rounder which needs not be exact but only α-equivalent to the exact result.

Together, α-equivalence and the decomposition theorem simplify the design and the verification of the FPU, since one can verify the units separately and then compose them using the theorems from this section.

2.1. Factorings

2.1.1. Basic definitions. 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 idea behind factorings is to leave the bitvector level and argue about the more abstract factorings in order to ease the verification of hardware.

The standard introduces precision formats, which we characterize by a pair (N, P) for exponent and significand width. Common values for (N, P) are $(8, 24)$ and $(11, 53)$, called single and double precision, respectively. However, the theory described here is not limited to these values of N and P; we only assume $N > 2$ and $P > 1$. For the rest of this section let N, P be arbitrary but fixed numbers with $N > 2$, $P > 1$.

From N we derive the constants $e_{\min} := -2^{N-1} + 2$ and $e_{\max} := 2^{N-1} - 1$. These 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.

The next lemma states that nonzero IEEE factorings are unique:[2]

Lemma 1. *Let (s, e, f) and (s', e', f') be IEEE factorings with nonzero value.*

$$[\![s, e, f]\!] = [\![s', e', f']\!] \iff (s, e, f) = (s', e', f').$$

Zero has two IEEE factorings $(0, e_{\min}, 0)$ and $(1, e_{\min}, 0)$, called $+0$ and -0, respectively.

2.1.2. Normalization. Next, we define the normalization algorithm. We start by defining a function $\widehat{norm}$ which maps nonzero factorings to factorings with significand between 1 and 2:

$$\widehat{norm}(s, e, f) := \left(s, e + \lfloor \log_2 f \rfloor, f \cdot 2^{-\lfloor \log_2 f \rfloor}\right).$$

We proceed with the definition of the function *norm*, which maps any (possibly zero) factoring to an IEEE factoring. Let $(\hat{s}, \hat{e}, \hat{f}) := \widehat{norm}(s, e, f)$:

$$norm(s, e, f) := \begin{cases} (\hat{s}, \hat{e}, \hat{f}) & \text{if } f \neq 0, \hat{e} \geq e_{\min}, \\ (\hat{s}, e_{\min}, \hat{f} \cdot 2^{\hat{e} - e_{\min}}) & \text{if } f \neq 0, \hat{e} < e_{\min}, \\ (s, e_{\min}, 0) & \text{if } f = 0. \end{cases}$$

The following lemma summarizes the most important properties of the normalization functions:

Lemma 2. *Let (s, e, f) be an arbitrary factoring.*[3]
(i) $[\![\widehat{norm}(s, e, f)]\!] = [\![s, e, f]\!]$ *if* $f \neq 0$,
(ii) $1 \leq \widehat{norm}_f(s, e, f) < 2$ *if* $f \neq 0$,
(iii) $[\![norm(s, e, f)]\!] = [\![s, e, f]\!]$,
(iv) $norm(s, e, f)$ *is an IEEE factoring.*

Having defined the normalization algorithm, we define conversion functions η and $\hat{\eta}$ which assign factorings to reals x:

$$\begin{aligned} \hat{\eta}(x) &:= \widehat{norm}(sign(x), 0, |x|) && \text{for } x \neq 0, \\ \eta(x) &:= norm(sign(x), 0, |x|) && \text{for arbitrary } x, \end{aligned}$$

where $sign(x) = 0$ if $x \geq 0$, and $sign(x) = 1$ otherwise.[4]

Lemma 3. *Let $x \in \mathbb{R}$.*
(i) $x = [\![\hat{\eta}(x)]\!]$ *if* $x \neq 0$,
(ii) $x = [\![\eta(x)]\!]$.

Lemma 4. *Let $x \in \mathbb{R}$.*
(i) $\hat{\eta}_e(x) = \lfloor \log_2 |x| \rfloor$ *if* $x \neq 0$,
(ii) $\hat{\eta}_f(x) = |x| \cdot 2^{-\hat{\eta}_e(x)}$ *if* $x \neq 0$,
(iii) $\eta_e(x) = \begin{cases} \lfloor \log_2 |x| \rfloor & \textit{if } x \neq 0 \textit{ and } \lfloor \log_2 |x| \rfloor \geq e_{\min}, \\ e_{\min} & \textit{otherwise,} \end{cases}$
(iv) $\eta_f(x) = |x| \cdot 2^{-\eta_e(x)}$.

Lemma 5. *Let (s, e, f) be an arbitrary factoring with value $x := [\![s, e, f]\!]$, $x \neq 0$.*

(i) $|x| \geq 2^{e_{\min}} \Longrightarrow \eta(x) = \hat{\eta}(x)$, *i.e., η and $\hat{\eta}$ coincide for normal numbers.*
(ii) $(s, e, f) = \hat{\eta}([\![s, e, f]\!])$, *if* $1 \leq f < 2$.
(iii) $(s, e, f) = \eta([\![s, e, f]\!])$ *for IEEE factorings* (s, e, f).
(iv) $\hat{\eta}_e(x) \leq \eta_e(x)$

Lemma 6. *Let $x \in \mathbb{R}$ and $(s, e, f) = \eta(x)$.*

(i) (s, e, f) *is normal iff* $|x| \geq 2^{e_{\min}}$,
(ii) (s, e, f) *is denormal iff* $|x| < 2^{e_{\min}}$.

2.1.3. Representable factorings. 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. Representable numbers exactly correspond to the representable numbers as defined in the standard.

The following lemma bounds (semi-)representable numbers.

Lemma 7. *Let (s, e, f) be a semi-representable factoring.*

(i) $f \leq 2 - 2^{1-P}$,
(ii) $|[\![s, e, f]\!]| \leq 2^i - 2^{i-P}$ *for* $i \in \mathbb{Z}, i > e$,
(iii) $X_{\max} := 2^{e_{\max}} \cdot (2 - 2^{1-P})$ *is the largest representable number.*

The following lemma characterizes the minimum distance between distinct semirepresentable factorings:

Lemma 8. *Let (s, e, f) and (s', e', f') be semi-representable factorings with values $x := [\![s, e, f]\!]$ and $x' := [\![s', e', f']\!]$, let $x \neq x'$, and let i be an integer with $e \geq i, e' \geq i$. Then*:

$$|x - x'| \geq 2^{i-(P-1)}.$$

2.2. *Rounding*

Since (semi-)representable numbers are not closed under arithmetic operations (e.g., addition, division), the IEEE standard defines four rounding modes: round to nearest, round up, round down, and round to zero. In this section, we define the rounding function, which maps arbitrary reals to semi-representable numbers according to the standard. The definition is similar to Miner's definition [38]; it only differs in cases of overflow and underflow (see Section 2.3).

2.2.1. Definition. We start with the definition of a function $r_{\text{int}}(\cdot, \mathcal{M})$ for each rounding mode $\mathcal{M} \in \{near, up, down, zero\}$, which rounds reals x to integers:

$$\begin{aligned}
r_{\text{int}}(x, up) &:= \lceil x \rceil \\
r_{\text{int}}(x, down) &:= \lfloor x \rfloor \\
r_{\text{int}}(x, zero) &:= (-1)^{sign(x)} \cdot \lfloor |x| \rfloor \\
r_{\text{int}}(x, near) &:= \begin{cases} \lfloor x \rfloor & \text{if } x - \lfloor x \rfloor < \lceil x \rceil - x, \\ \lceil x \rceil & \text{if } x - \lfloor x \rfloor > \lceil x \rceil - x, \\ x & \text{if } \lfloor x \rfloor = \lceil x \rceil, \\ 2\lfloor \lceil x \rceil / 2 \rfloor & \text{otherwise.} \end{cases}
\end{aligned}$$

Note that $x - \lfloor x \rfloor$ and $\lceil x \rceil - x$ are the fraction of x and its complement, respectively.

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

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

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

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

Since it is not obvious that this definition conforms with the IEEE standard, we prove a theorem to convince the reader of the conformance in Section 2.2.3.

2.2.2. Decomposition theorem. The decomposition theorem we prove in this section decomposes the computation of the rounding function into three steps: η-computation (sometimes called pre-normalization in the literature), significand rounding, and post-normalization. The benefit of having the decomposition theorem is that it simplifies the design and verification of rounder implementations. Furthermore, it is a powerful tool in other proofs, e.g., in Theorem 23.

The η-computation step computes the IEEE factoring $X = \eta(x)$, 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*:

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

where $X = (s, e, f)$ is an IEEE factoring, and $\mathcal{M}$ is a rounding mode. The following lemma states some properties of the *sigrd* function:

Lemma 9.

(i) $sigrd(X, \mathcal{M}) = |rd([\![X]\!], \mathcal{M})| \cdot 2^{-e}$,
(ii) $0 \leq sigrd(X, \mathcal{M}) \leq 2$,
(iii) $1 \leq f \implies 1 \leq sigrd(X, \mathcal{M})$,
(iv) $1 \geq f \implies 1 \geq sigrd(X, \mathcal{M})$,
(v) $sigrd(X, \mathcal{M}) \cdot 2^{P-1}$ *is an integer.*

In the case that significand rounding returns 0 or 2, the factoring has to be post-normalized. If significand rounding returns 0, the sign bit is forced to 0 in order to yield $\eta(0)$. In case the significand rounding returns 2, the exponent is incremented, and the significand is forced to 1. Let $f_{rd} = sigrd(X, \mathcal{M})$.

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

Note that the *postnorm* function does also significand rounding, since it uses *sigrd* as a sub-function.

Lemma 10. *The result* $postnorm(X, \mathcal{M})$ *of the post-normalization is a semi-representable IEEE factoring.*

Proof: The case $f_{rd} \in \{0, 2\}$ is trivial. Assume $0 < f_{rd} < 1$. By Lemma 9(iii) we know $1 \not\leq f$, i.e., $f < 1$, and hence $e = e_{\min}$ since X is an IEEE factoring. Therefore $postnorm(X, \mathcal{M})$ is an IEEE factoring, and with Lemma 9(v) it is a semi-representable factoring.

Now assume $1 \leq f_{rd} < 2$. Since the input X is an IEEE factoring, we know $e \geq e_{\min}$, and hence $(s, e, f_{rd}) = postnorm(X, \mathcal{M})$ is an IEEE factoring; semi-representability now follows from Lemma 9(v). □

Lemma 11. *The value computed by postnorm is the rounded result*:

$$[\![postnorm(X, \mathcal{M})]\!] = rd([\![X]\!], \mathcal{M}).$$

We now are ready to state the Decomposition Theorem:

Theorem 12 (Decomposition Theorem). *For any real* x, *and rounding mode* $\mathcal{M} \in \{near, up, down, zero\}$:

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

Proof: For nonzero rounding results, the claim follows from Lemmas 5(iii) and 11. Otherwise, the claim follows by expanding the definitions of $norm$, η, and *postnorm*. □

The IEEE factoring of the rounding result can therefore be computed by first computing the IEEE factoring $\eta(x)$ of x, then rounding the significand, and finally post-normalizing the result. This decomposition of the rounding function is well known [22], but has been (paper-and-pencil-) proved explicitly for the first time in [41]. We extend this work by formally verifying the decomposition theorem.

The following lemma is our first application of the decomposition theorem as a proof utility:

Lemma 13. *Let* $x \in \mathbb{R}$ *and* $(s, e, f) = \eta(x)$.

$$(s, e, f) \text{ is denormal} \implies \eta_e(rd(x, \mathcal{M})) = e_{\min}.$$

Proof: Since (s, e, f) is denormal, we have $e = e_{\min}$ and $f < 1$. By Lemma 9(iv) and the definition of post-normalization, $postnorm_e(\eta(x), \mathcal{M}) = e_{\min}$ follows. The claim now follows by application of the decomposition Theorem 12. □

2.2.3. Correctness of the rounding function. We now demonstrate that the definition of the IEEE rounding function rd conforms with the IEEE standard. The specification of the round to nearest mode in the standard is as follows:

(...) In this mode the representable value nearest to the infinitely precise result [of any floating point operation] shall be delivered; if the two nearest representable values are equally near, the one with its least significant bit zero shall be delivered. (...)

Since our formal definition of the function rd does not obviously coincide with this informal definition, the following theorem is proved. This theorem hopefully convinces the reader of the conformance of our rounding definition.

Theorem 14. *Let $x, x' \in \mathbb{R}$ and x' be a semi-representable number.*

(i) *For any rounding mode $\mathcal{M}$, $\eta(rd(x, \mathcal{M}))$ is semi-representable.*
(ii) *$rd(x, near)$ is a* nearest *semi-representable number*:

$$|x - x'| \geq |x - rd(x, near)|.$$

(iii) *If there are* two *nearest numbers, then the one with least significant digit zero is chosen*: *$x' \neq rd(x, near)$ and $|x - x'| = |x - rd(x, near)|$ implies $\eta_f(rd(x, near)) \cdot 2^{P-1}$ is even.*

Similar informal specifications exist in the standard for the three remaining rounding modes, and conformance theorems for these have been proved in PVS.

Proof: Part (i) is a trivial consequence of Lemma 10 and Theorem 12. Part (ii) and (iii) rely on the following fact proved by Miner in PVS [38]:

$$|x - r_{\text{int}}(x, near)| \leq \tfrac{1}{2} \quad \text{and}$$
$$|x - r_{\text{int}}(x, near)| = \tfrac{1}{2} \Longrightarrow r_{\text{int}}(x, near) \text{ is even.}$$

Let $(s, e, f) = \eta(x)$ and $(s', e', f') = \eta(x')$. It is easy to adopt the above fact to the rd-function:

$$|x - rd(x, near)| \leq 2^{e-P} \quad \text{and} \tag{1}$$
$$|x - rd(x, near)| = 2^{e-P} \Longrightarrow \left(rd(x, near) \cdot 2^{-(1+e-P)}\right) \text{ is even.}$$

We now prove part (ii). We may assume that $x' \neq rd(x, near)$, since otherwise the claim is trivial. From the decomposition theorem and the definition of the post-normalization we know that $\eta_e(rd(x, near)) \geq e$. Now assume $e' \geq e$. Using Lemma 8 (where we set $(s, e, f) = \eta(rd(x, near))$, $(s', e', f') = \eta(x')$, and $i = e$) results in

$$|rd(x, near) - x'| \geq 2^{e-(P-1)} = 2 \cdot 2^{e-P}. \tag{2}$$

Using the triangle inequality, (1) and (2) together yield

$$|x - x'| \geq 2^{e-P}. \tag{3}$$

Equations (1) and (3) yield part (ii). Assume otherwise that $e' < e$. Since $e_{\min} \leq e'$ we have $e_{\min} < e$, and therefore $f \geq 1$, since (s, e, f) is an IEEE factoring. Hence $|x| \geq 2^e$. Lemma 7(ii) with $i = e$ gives $|x'| \leq 2^e - 2^{e-P}$. Together this implies

$$|x' - x| \geq 2^{e-P}. \tag{4}$$

Again, (1) and (4) yield part (ii). The proof of part (iii) is similar. □

The following theorem states that the semi-representable numbers are exactly the fixpoints of the rounding function:

Theorem 15. *For any real x and rounding mode $\mathcal{M}$, x is semi-representable iff $rd(x, \mathcal{M}) = x$.*

Proof: If $rd(x, \mathcal{M}) = x$, x is semi-representable by Theorem 14(i). Conversely, if x is semi-representable and $\mathcal{M} = near$, then the round result must equal x by Theorem 14(ii) with $x' = x$. The claim for the remaining rounding modes follows analogously from their respective conformance theorems. □

2.3. Exceptions and wrapped exponents

The IEEE standard defines five exceptions: invalid operation (*INV*), division by zero (*DIVZ*), overflow (*OVF*), underflow (*UNF*), and inexact result (*INX*). In this section, we define the *UNF*, *OVF*, and *INX* exceptions. The *INV* and *DIVZ* exceptions are omitted since they are trivial.

2.3.1. Underflow. The standard defines the underflow exception as follows:

> Two correlated events contribute to underflow. One is the creation of a tiny nonzero result between $\pm 2^{e_{\min}}$ (...) The other is extraordinary loss of accuracy (...) When an underflow trap (...) is not enabled (...), underflow shall be signaled when both tininess and loss of accuracy have been detected. When an underflow trap (...) is enabled, underflow shall be signaled when tininess is detected regardless of loss of accuracy. (...)

For each of the contributing events, the standard leaves the choice between two different implementations. We use *tininess before rounding* (instead of after rounding) and *inexact result* as loss of accuracy (instead of denormalization loss). Tininess before rounding occurs "(...) *when a nonzero result computed as though both exponent range and the precision were unbounded would lie strictly between* $\pm 2^{e_{\min}}$." This is formalized as

$$TINY(x) := x \neq 0 \wedge |x| < 2^{e_{\min}}.$$

Here, x is the exact result of a floating point operation, and therefore is "computed as though both exponent range and the precision were unbounded." An inexact result occurs "(...) *when the delivered result differs from what would have been computed were both exponent range and precision unbounded.*" We formalize this as

$$LOSS(x, \mathcal{M}) := rd(x, \mathcal{M}) \neq x.$$

Loss of accuracy only syntactically depends on the rounding mode, since this is a required parameter to the rd-function. From Theorem 15 it easily follows $LOSS(x, \mathcal{M}_1) = LOSS(x, \mathcal{M}_)$ for distinct rounding modes $\mathcal{M}_i$.

Lemma 16. *Let $x \in \mathbb{R}$ and $(s, e, f) = \eta(x)$. The LOSS signal can be computed using sigrd*:

$$LOSS(x, \mathcal{M}) \iff sigrd((s, e, f), \mathcal{M}) \neq f.$$

Proof: The claim follows immediately from the decomposition Theorem 12. □

Having defined tininess and loss of accuracy, we can define the underflow exception. *UNFen* is the *UNF* trap enable flag provided by the CPU:

$$UNF(x, \mathcal{M}, UNFen) := TINY(x) \wedge (LOSS(x, \mathcal{M}) \vee UNFen).$$

As mentioned above, the standard leaves other choices for the definition of *TINY* and *LOSS*. We have not defined or otherwise considered these definitions in our PVS formalization, though we believe that this would be straightforward in our setting. In [41], the alternative definitions are given, and theorems describing the relationships of the alternative definitions to those given above are proven with paper-and-pencil mathematics. In [25], similar definitions and theorems are formalized in the theorem prover HOL.

2.3.2. Overflow. The standard defines the overflow exception as follows:

The overflow exception shall be signaled whenever the destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result were the exponent range unbounded. (...)

We formalize the *OVF* exception as follows:

$$OVF(x, \mathcal{M}) := |rd(x, \mathcal{M})| > X_{\max}.$$

For the implementation of the *OVF* test in the actual hardware, it is beneficial to differentiate between overflows that are apparent before rounding and overflows that arise during rounding:

$$\begin{aligned} \mathit{OVF}_{\mathrm{bef}}(x) &:= \eta_e(x) > e_{\max}, \\ \mathit{OVF}_{\mathrm{aft}}(x, \mathcal{M}) &:= \eta_e(x) = e_{\max} \wedge \mathit{sigrd}(\eta(x), \mathcal{M}) = 2. \end{aligned}$$

In the first case we say the overflow occurs *before rounding*, in the latter case we say *after rounding*.

Lemma 17. *An overflow occurs iff it occurs before or after rounding*:

$$\mathit{OVF}(x, \mathcal{M}) \iff \mathit{OVF}_{\mathrm{bef}}(x) \vee \mathit{OVF}_{\mathrm{aft}}(x, \mathcal{M}).$$

Proof: The claim follows from Lemma 4 and Theorem 12. □

2.3.3. Wrapped exponent. In case of an overflow or underflow with corresponding trap enabled, the standard requests an implementation to deliver a biased result to the trap handler:

Trapped overflows (...) shall deliver to the trap handler the result obtained by dividing the infinitely precise result by 2^{A} and then rounding. The bias adjust A is 192 in the single, 1536 in the double format. (...)

Note that $A = 3 \cdot 2^{N-2}$ for exponent widths $N = 8$ and $N = 11$. Analogously to overflows, trapped underflows shall deliver the result obtained by multiplying the exact result with 2^{A} and then rounding. This is captured in the following definition.

$$\mathit{wrapped}(x, \mathcal{M}, \mathit{OVFen}, \mathit{UNFen}) := \begin{cases} x \cdot 2^{-A} & \text{if } \mathit{OVF}(x, \mathcal{M}) \text{ and } \mathit{OVFen}, \\ x \cdot 2^{A} & \text{if } \mathit{UNF}(x, \mathcal{M}, \mathit{UNFen}) \text{ and } \mathit{UNFen}, \\ x & \text{otherwise.} \end{cases}$$

Now we are ready to define the floating point result of operations with exact result x:

$$\mathit{result}(x, \mathcal{M}, \mathit{OVFen}, \mathit{UNFen}) := \mathit{rd}\big(\mathit{wrapped}(x, \mathcal{M}, \mathit{OVFen}, \mathit{UNFen}), \mathcal{M}\big)$$

For the sake of conciseness, we sometimes omit the *OVFen* and *UNFen* parameters in applications of the *wrapped* and *result* function.

The idea behind exponent wrapping is that multiplying the result with $2^{\pm A}$ before rounding scales the result into the representable range. The FPU returns the wrapped and rounded result to the trap handler, which can use the result in subsequent operations.

If an overflow is detected with disabled trap, the *result* definition above returns a result exceeding $X_{\max}$. The standard requests a final result of either $\pm X_{\max}$ or $\pm\infty$, depending on the sign and the rounding mode. This is specified as a case-split in PVS; we omit the details, since they are trivial.

2.3.4. Inexact. The standard defines the inexact exception as follows:

> If the rounded result of an operation is not exact or if it overflows without an overflow trap, then the inexact exception shall be signaled. (...)

It is not clear if the "rounded result" is meant to be $rd(x, \mathcal{M})$ without being wrapped, or $\mathit{result}(x, \mathcal{M})$, which potentially has been wrapped. In Harrison's formalization of the IEEE standard [25] exponent wrapping is not considered, and thus the inexact exception is defined as

$$\mathit{INX}(x, \mathcal{M}, \mathit{OVFen}) := \mathit{LOSS}(x, \mathcal{M}) \vee (\mathit{OVF}(x, \mathcal{M}) \wedge \overline{\mathit{OVFen}}).$$

In contrast, a test[5] on Intel's Pentium II with the operation $x := X_{\text{min}}/2$ with enabled underflow trap and $\mathcal{M} = up$ did not yield an *INX* signal (where X_{min} is the smallest representable value). If x is not being wrapped before rounding, then rounding up x yields X_{min}. Hence, if the *INX* signal was computed as $rd(x, \mathcal{M}) \neq x$, the rounded result would differ from x and so the *INX* signal should be set. Otherwise, $x \cdot 2^A$ is a representable number, and hence rounding does not change $x \cdot 2^A$. Consequently, if the "rounded result" in the IEEE standard is meant to be the wrapped and rounded result, then no *INX* signal should be set.

In contrast to Harrison [25], we define the inexact exception as

$$\begin{aligned}\mathit{INX}(x, \mathcal{M}, \mathit{OVFen}, \mathit{UNFen}) := {}& \mathit{LOSS}(\mathit{wrapped}(x, \mathcal{M}, \mathit{OVFen}, \mathit{UNFen}), \mathcal{M})\\ & \vee (\mathit{OVF}(x, \mathcal{M}) \wedge \overline{\mathit{OVFen}}).\end{aligned}$$

This is the definition also used in IBM's S/390 [27, pp. 19–22] and in [41], for example. It has the advantage that programs can distinguish exact (except for exponent wrapping) from inexact computations in case of trapped overflows and underflows.

We believe that the IEEE standard is ambiguous in this point.

2.4. *α-equivalence*

We now formalize the concept of α-equivalence and α-representatives from [41]. This concept is a 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

$$(x = y) \vee \exists q \in \mathbb{Z} : q \cdot 2^\alpha < x, \quad y < (q + 1) \cdot 2^\alpha, \tag{5}$$

i.e., if x and y are equal or both lie in the same open interval between two consecutive integral multiples of 2^α (cf. figure 1). Clearly, such a q in (5) must be $q_\alpha(x) := \lfloor x \cdot 2^{-\alpha} \rfloor$.

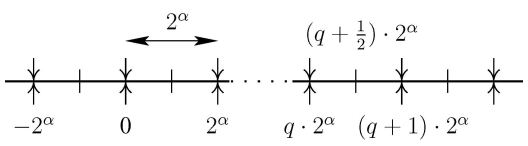

Figure 1. α-equivalence.

The α-representative of x is defined as

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

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 18. *Let x, y be reals, and α, k be integers.*

(i) $\equiv_\alpha$ *is an equivalence relation,*
(ii) $x \equiv_\alpha [x]_\alpha$,
(iii) $x \equiv_\alpha y \iff [x]_\alpha = [y]_\alpha$, (*representative equivalence*)
(iv) $x \equiv_\alpha y \iff -x \equiv_\alpha -y$, *and* $[-x]_\alpha = -[x]_\alpha$, (*negative value*)
(v) $x \equiv_\alpha y \iff 2^k \cdot x \equiv_{\alpha+k} 2^k \cdot y$, *and* $[2^k \cdot x]_{\alpha+k} = 2^k \cdot [x]_\alpha$, (*scaling*)
(vi) $x \equiv_\alpha y \iff x + k \cdot 2^\alpha \equiv_\alpha y + k \cdot 2^\alpha$, (*translation*)
(vii) $x \equiv_\alpha y \implies x \equiv_{\alpha+k} y$ *if* $k \geq 0$, (*coarsening*)
(viii) $x = 0 \iff x \equiv_\alpha 0 \iff [x]_\alpha = 0$, (*zero value*)
(ix) $0 < x < 2^\alpha \implies [x]_\alpha = 2^{\alpha-1}$. (*small value*)

Proof: Parts (i)–(iv), (viii)–(ix) are simple consequences of the definition, parts (v)–(vii) are proved by induction on k. □

Lemma 19. *Let $x, y \in \mathbb{R}$, $\alpha, k \in \mathbb{Z}$ such that $x \equiv_\alpha y$ and $k \geq \alpha$.*

$$x < 2^k \iff y < 2^k.$$

Proof: The claim is trivial if $x = y$. We therefore may assume that there is a $q \in \mathbb{Z}$ such that

$$q \cdot 2^\alpha < x, y < (q+1) \cdot 2^\alpha. \tag{6}$$

It cannot hold that $q < 2^{k-\alpha} < q + 1$ since this would enclose the integer $2^{k-\alpha}$ in between the two consecutive integers q and $q+1$. This implies that either $q \geq 2^{k-\alpha}$ or $2^{k-\alpha} \geq q+1$ holds. First assume $q \geq 2^{k-\alpha}$, hence $q \cdot 2^\alpha \geq 2^k$. Equation (6) now implies $x, y > 2^k$. Assume otherwise $2^{k-\alpha} \geq q+1$, i.e., $2^k \geq (q+1) \cdot 2^\alpha$. Now (6) implies $2^k > x, y$. □

The following theorem describes equivalence on factorings:

Lemma 20. *Let $x, y \in \mathbb{R}$ nonzero, $e := \eta_e(x)$, $e' := \eta_e(y)$, $\hat{e} := \hat{\eta}_e(x)$, $\hat{e}' := \hat{\eta}_e(y)$, and α be an integer.*

(i) $x \equiv_\alpha y \implies sign(x) = sign(y)$,
(ii) $\alpha \leq \hat{e}$ *and* $x \equiv_\alpha y \implies \hat{e} = \hat{e}'$,
(iii) $\alpha \leq e$ *and* $x \equiv_\alpha y \implies e = e'$,

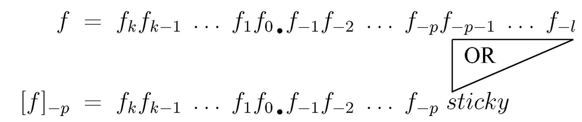
$f = f_k f_{k-1} \dots f_1 f_0 . f_{-1} f_{-2} \dots f_{-p} f_{-p-1} \dots f_{-l}$
OR
$[f]_{-p} = f_k f_{k-1} \dots f_1 f_0 . f_{-1} f_{-2} \dots f_{-p}$ sticky

Figure 2. Computing representatives by sticky-computation.

(iv) $|x| \geq 2^{e_{\min}}$ *and* $\alpha \leq e \Longrightarrow \hat{e} = \hat{\eta}_e([x]_\alpha)$,

(v) $|x| < 2^{e_{\min}}$ *and* $\alpha \leq e \Longrightarrow \hat{\eta}_e([x]_\alpha) < e_{\min}$.

Proof: We only prove part (ii). With Lemma 18(vii) it suffices to proof the claim for $\alpha = \hat{e}$. By part (i) and Lemma 18(iv) we may assume $x, y \geq 0$.

Since the claim is trivial for $x = y$, we further assume that $q_{\hat{e}}(x) \cdot 2^{\hat{e}} < x, y < (q_{\hat{e}}(x) + 1) \cdot 2^{\hat{e}}$ by definition of α-equivalence. From Lemma 2(ii), we know $1 \leq x \cdot 2^{-\hat{e}} < 2$, and therefore $q_{\hat{e}}(x) = \lfloor x \cdot 2^{-\hat{e}} \rfloor = 1$. We then have $2^{\hat{e}} < x, y < 2^{\hat{e}+1}$, and therefore $\hat{e} = \lfloor \log x \rfloor = \lfloor \log y \rfloor$. Lemma 4 proves the claim. □

We now are ready to prove two important theorems which enable the computation of IEEE factorings corresponding to representatives:

Theorem 21. *Let* $x \in \mathbb{R}$, *let* $(s, e, f) := \eta(x)$ *be the corresponding IEEE factoring, and let* $p \geq 0$ *be an integer. The IEEE factoring of* $[x]_{e-p}$ *can be computed by computing the representative* $[f]_{-p}$ *of* f*:*

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

Proof: From Lemma 20(i) and 20(iii) we have $\eta_s([x]_{e-p}) = s$ and $\eta_e([x]_{e-p}) = e$. From Lemma 4 we know $\eta_f([x]_{e-p}) = |[x]_{e-p}| \cdot 2^{-e}$. With Lemma 18(iv) and 18(v), we have $|[x]_{e-p}| \cdot 2^{-e} = [|x| \cdot 2^{-e}]_{-p}$. Lemma 4 gives $|x| \cdot 2^{-e} = f$, and hence $\eta_f([x]_{e-p}) = [f]_{-p}$. □

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

$$sticky_{-p}(f) := f_{-p-1} \vee \dots \vee f_{-l}.$$

Theorem 22. *With the above definitions, the representative* $[f]_{-p}$ *of* f *can be computed* *by replacing the less significant bits by the sticky bit*:

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

Proof: By definition, $q_{-p}(f) = \lfloor f \cdot 2^p \rfloor$, and therefore $q_{-p}(f) = \sum_{i=-p}^{k} f_i \cdot 2^{i+p}$. Furthermore, $f = q_{-p}(f) \cdot 2^{-p}$, iff $sticky_{-p}(f) = 0$. Applying this in the definition of $[\,\cdot\,]_{-p}$ proves the claim. □

Theorems 21 and 22 together enable an easy 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 [22], but the formalism with α-representatives enables a very concise argumentation about these sticky computations. The verification of the adder circuitry in Section 3.3, e.g., relies heavily on α-equivalence.

2.5. *Rounding representatives*

The most important property of α-representatives is that rounding and exception-computation yield the same result on α-equivalent x, x' for appropriate α. This will be proved in this section. The proofs in this section are completely different from the proofs in [21,41]. There, the proofs are by geometrical arguments which are not well supported by PVS.

Theorem 23. *Let $x \in \mathbb{R}$, $(s, e, f) := \eta(x)$, and $\mathcal{M}$ be a rounding mode.*

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

Proof: We only give a sketch of the proof. By Theorems 12 and 21 it suffices to show

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

By unfolding the definitions of *sigrd* and r_{rat}, this is equivalent to

$$r_{\text{int}}\big((-1)^s \cdot f \cdot 2^{P-1}, \mathcal{M}\big) = r_{\text{int}}((-1)^s \cdot [f]_{-P} \cdot 2^{P-1}, \mathcal{M}). \tag{7}$$

Since the claim is trivial if $[f]_{-P} = f$, we can assume by the definition of α-equivalence that $f \cdot 2^P \notin \mathbb{Z}$, and $[f]_{-P} = (q + \frac{1}{2}) \cdot 2^{-P}$ with $q := q_{-P}(f) = \lfloor f \cdot 2^P \rfloor$. Hence $[f]_{-P} = (\lfloor f \cdot 2^P \rfloor + \frac{1}{2}) \cdot 2^{-P}$ holds. Substituting this in (7) yields

$$\begin{aligned} & r_{\text{int}}((-1)^s \cdot f \cdot 2^{P-1}, \mathcal{M}) \\ &= r_{\text{int}}((-1)^s \cdot \big((\lfloor f \cdot 2^P \rfloor + \tfrac{1}{2}) \cdot 2^{-1}\big), \mathcal{M}) \\ &= r_{\text{int}}((-1)^s \cdot \big(\tfrac{1}{4} + \tfrac{1}{2}\lfloor f \cdot 2^P \rfloor\big), \mathcal{M}). \end{aligned}$$

In PVS, this is verified by induction on $\lfloor f \cdot 2^P \rfloor$ in a very technical proof. □

Corollary 24. *Let $x \in \mathbb{R}$, $\alpha \leq \eta_e(x) - P$, and $\mathcal{M}$ be a rounding mode.*

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

In particular, the claim holds for $\alpha = \hat{\eta}_e(x) - P$.

Proof: By Lemma 18(ii,vii) we have $x \equiv_{\eta_e(x)-P} [x]_\alpha$. Hence, by Theorem 23 and by Lemma 18(iii):

$$rd(x, \mathcal{M}) = rd([x]_{\eta_e(x)-P}, \mathcal{M}) = rd([[x]_\alpha]_{\eta_e(x)-P}, \mathcal{M}). \tag{8}$$

By Lemma 20(iii) we have $\eta_e(x) = \eta_e([x]_\alpha)$. Replacing this in (8) gives

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

A second application of Theorem 23 with $[x]_\alpha$ gives

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

The claim for $\alpha = \hat{\eta}_e(x) - P$ follows from Lemma 5(iv). □

Corollary 25. *Let (s, e, f) be an IEEE factoring.*

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

Proof: The claim follows from lemmas 9(i) and theorems 21 and 23. □

Not only the rounding can be accomplished by using the representative, but also the detection of exceptions. We first prove this for *OVF* and *UNF*:

Theorem 26. *Let $x \in \mathbb{R}$, $(s, e, f) := \eta(x)$, and $\mathcal{M}$ be a rounding mode.*

(i) $OVF(x, \mathcal{M}) \iff OVF([x]_{e-P}, \mathcal{M})$,
(ii) $TINY(x) \iff TINY([x]_{e-P})$,
(iii) $LOSS(x, \mathcal{M}) \iff LOSS([x]_{e-P}, \mathcal{M})$,
(iv) $UNF(x, \mathcal{M}, UNFen) \iff UNF([x]_{e-P}, \mathcal{M}, UNFen)$,

Analogously to Corollary 24, the claim holds for finer representatives.

Proof: Part (i) is an immediate consequence of Theorem 23. Part (ii) follows from Lemmas 20(iv) and 20(v). Part (iii) is slightly more complicated. We have to prove

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

By Theorem 15, this is equivalent to

$$\eta(x) \text{ is semi-representable } \iff \eta([x]_{e-P}) \text{ is semi-representable.}$$

By Theorem 21 and by definition of representability, this is equivalent to

$$f \cdot 2^{P-1} \in \mathbb{Z} \iff [f]_{-P} \cdot 2^{P-1} \in \mathbb{Z}. \tag{9}$$

Assume $f \cdot 2^{P-1} \in \mathbb{Z}$. Then $q_{-P}(f) = \lfloor f \cdot 2^P \rfloor = f \cdot 2^P$ and hence $[f]_{-P} = f$. Thus, $[f]_{-P} \cdot 2^{P-1} \in \mathbb{Z}$ as well.

In the other case $f \cdot 2^{P-1} \notin \mathbb{Z}$ we either have $[f]_{-P} = f$ in which case (9) is trivially true, or $[f]_{-P} = \left(q_{-P}(f) + \frac{1}{2}\right) \cdot 2^{-P}$. Hence, $[f]_{-P} \cdot 2^{P-1} = \frac{1}{2}\lfloor f \cdot 2^P \rfloor + \frac{1}{4} \notin \mathbb{Z}$.

Part *(iv)* is a trivial consequence of the former parts. □

From the above theorem, one can conclude that the wrapped and rounded result *result* $(x, \mathcal{M})$ can be computed using equivalence, too. However, in case of trapped underflow one needs more precision, namely $(\hat{e} - P)$-equivalence instead of $(e - P)$-equivalence:

Theorem 27. *Let $x \in \mathbb{R}$, $\hat{e} := \hat{\eta}_e(x)$, and $\mathcal{M}$ be a rounding mode.*

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

Proof: Theorem 26 shows that exponent wrapping occurs on x iff it occurs on $[x]_{\hat{e}-P}$. The claim follows trivially from Corollary 24 if no wrapping occurs. Otherwise, assume first that a trapped underflow occurs, i.e., $\mathit{UNF}(x, \mathcal{M}, \mathit{UNFen}) \wedge \mathit{UNFen}$. We have to prove

$$rd(x \cdot 2^A, \mathcal{M}) = rd([x]_{\hat{e}-P} \cdot 2^A, \mathcal{M}). \tag{10}$$

By Lemma 18(v) we have $[x]_{\hat{e}-P} \cdot 2^A = [x \cdot 2^A]_{\hat{e}-P+A}$. Replacing this in (10) yields

$$rd(x \cdot 2^A, \mathcal{M}) = rd([x \cdot 2^A]_{\hat{e}-P+A}, \mathcal{M}).$$

This follows from Corollary 24 if we prove

$$\hat{e} - P + A = \hat{\eta}_e(x \cdot 2^A) - P, \tag{11}$$

which follows from Lemma 4(i).

The proof for *OVF* is literally the same with $-A$ for A. □

Note that in order to prove (11), the higher precision of $[x]_{\hat{e}-P}$ compared to $[x]_{e-P}$ is needed: (11) would not follow for e and η replaced for $\hat{e}$ and $\hat{\eta}$, respectively: if $\hat{e} < e_{\min}$, then $e = e_{\min}$, and it may happen that $e_{\min} < \eta_e(x \cdot 2^A) = \hat{\eta}(x \cdot 2^A) = \hat{e} + A \neq e + A$. Intuitively, computing the $(e - P)$-representative of x kills digits in the significand which have been "shifted out" by denormalizing the significand. These digits, however, are present

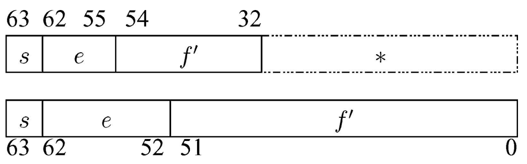

Figure 3. Embedding of (s, e, f') in one bitvector.

in the representative of the wrapped significand, since by scaling these digits are "shifted back".

The *INX* exception can be computed on representatives, too. Analogously to Theorem 27, we need the more precise $(\hat{e} - P)$-representative:

Theorem 28. *Let $x \in \mathbb{R}$, $\hat{e} := \hat{\eta}_e(x)$.*

$$INX(x, \mathcal{M}, OVFen, UNFen) \iff INX([x]_{\hat{e}-P}, \mathcal{M}, OVFen, UNFen).$$

Proof: The proof is a combination of the proofs of 26(iii) and 27. We omit the details. □

Theorems 26–28 enable a subdivision of a complete FPU into computation units (e.g., adder, multiplier) and a rounder. The computation units compute a result which need not be exact but only an $(\hat{e}-P)$-equivalent approximation of the exact result. The rounder therefrom rounds to the correct floating point number, and computes the exceptions. The passing of an $(\hat{e} - P)$-equivalent approximation of the exact result saves very large intermediate results, e.g., during addition of the format's smallest and largest representable numbers. Furthermore, the sub-division of the FPU into smaller parts eases the verification of the hardware, since the parts can be verified separately.

2.6. *IEEE number format*

The IEEE standard defines bitvector representations for floating point numbers (figure 3). The definition in the standard also features the special values *infinity* (∞) and *not-a-number* (*NaN*). We omit the details of the formalization since they are straightforward.

2.7. *Comparison, conversion*

The IEEE standard demands that instructions for the comparison of floating point numbers, and for the conversion between all supported floating point formats, and between all supported floating point formats and integer formats are available. Conversions are subject to rounding as specified in Section 2.2. All four rounding modes must be supported. We have formalized comparisons and conversions in PVS but omit the details here. They can be found in [31].

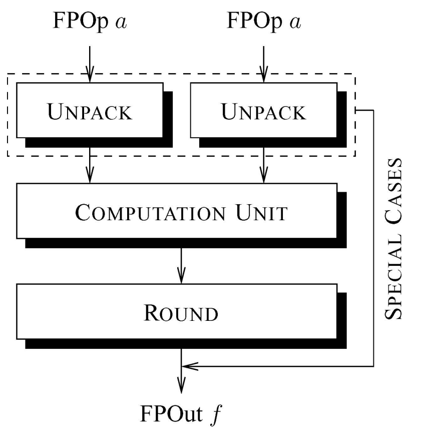

Figure 4. Top-level view of the floating point units.

3. Verifying the *VAMP* FPU

In this section, we describe the design and verification of the floating point hardware. The hardware is verified with respect to the specification given in the previous section. We build three separate floating point units: the *additive unit* for addition/subtraction, the *multiplicative unit* for multiplication/division, and the *miscellaneous unit* that supports conversions, comparisons, and some trivial operations like negation and absolute value computation.

Basically, each unit is built as depicted in figure 4: the operands are passed to the unpackers, where they are converted to some more convenient internal format. The computation unit then performs the actual computation. Instead of computing the exact result, it computes an α-equivalent approximation. This approximation is then fed to the rounding unit which rounds this value and outputs the result in IEEE format. The rounding units for the different operations are identical, i.e., there are three instances of the same rounding unit. Special cases such as operations on special operands (∞, *NaN*), or divisions by zero bypass the computation and rounding units.

3.1. Unpacker

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. Single and double precision operands are embedded into the same internal format by the unpacker.

In case of multiplication and division, the unpacker normalizes denormal significands and adjusts the exponents accordingly. For this purpose, the unpacker of the multiplicative FPU has a leading-zero counter and a shifter for each of the two operands.

Furthermore, the floating point unpacker detects special cases such as operations on $\pm\infty$ and *NaN*. Other special cases are multiplication/division by 0, and an exact addition/

subtraction result of 0. The result for special cases is selected by multiplexers and directly passed to the output, bypassing the computation unit and the rounder.

We omit the details of the construction and verification of the unpackers, since they are relatively simple; we refer the reader to [31, 41] for details.

3.2. Rounder

The purpose of the floating point rounder is to generate a correctly rounded result from an approximation of the exact result. Furthermore, the rounder computes the overflow, underflow, and inexact exception signals. In this section we briefly describe the design of the rounder, and show how the different parts of the rounder are composed in a formal way using the theorems presented in Section 2. We do not describe the construction and verification in detail, since this would be too long; again, we refer the reader to [31, 41].

3.2.1. Rounder interface. Let (s, e_0, f_0) be the factoring represented by the rounder inputs (i.e., the input bitvectors interpreted as numbers in the appropriate way), and let $x := [\![s, e_0, f_0]\!]$ be its value. x does not need to be the exact result of the operation, but only an α-equivalent approximation with $\alpha \leq \hat{\eta}_e(x) - P$, see Section 2.5.

The input factoring (s, e_0, f_0) does not need to be a normalized factoring. We only require the following three input conditions in order to prove the correctness of the rounder.

1. The value to round is not zero, i.e., $x \neq 0$. This implies $f_0 > 0$. If an operation yields zero as exact result, this has to be handled as special case by the unpacker.
2. If the exponent exceeds $e_{\max}$, the significand must be normal:

$$e_0 > e_{\max} \implies f \geq 1 \tag{12}$$

   The purpose of this requirement is to facilitate the detection of overflows before rounding: if the exponent e exceeds $e_{\max}$, an overflow before rounding has occurred regardless of the significand f, since by (12) the overflow is not compensated by a denormal significand.
3. The result x lies in a range such that in case of trapped underflows or trapped overflows before rounding the wrapped result lies strictly between $2^{e_{\min}}$ and $2^{e_{\max}}$. This guarantees that after wrapping the rounded result is in the range of representable numbers. In the rest of this section, let

$$y := \mathit{wrapped}(x, \mathcal{M}, \mathit{OVFen}, \mathit{UNFen}).$$

   We require:

$$(\mathit{TINY}(x) \wedge \mathit{UNFen}) \vee (\mathit{OVF}_{\mathrm{bef}}(x) \wedge \mathit{OVFen}) \implies 2^{e_{\min}} < |y| < 2^{e_{\max}}. \tag{13}$$

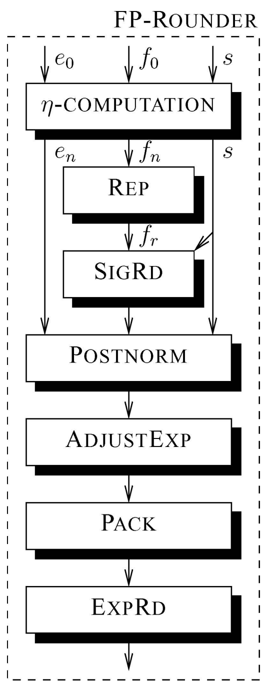

Figure 5. Rounder.

Figure 5 depicts the top-level schematics of the rounding unit. As suggested by the decomposition theorem, the rounding is performed by first computing $\eta(x)$ in circuit η-COMPUTATION, then rounding the significand, followed by a post-normalization in the case that significand rounding yields a significand of 2. In the actual implementation, a sticky-bit computation is performed on the significand after the η-computation (circuit REP). The hard part of the rounding is finished after post-normalization; the following three circuits are simple. ADJUSTEXP detects and handles trapped overflows after rounding, PACK converts the result to IEEE format, and EXPRD ties the result to $\pm\,\infty$ or $\pm\,X_{\max}$ in case of untrapped overflows, depending on the sign and rounding mode.

3.2.2. η-Computation. The η-COMPUTATION circuit[6] is the most complex circuit in the rounder. Its task is to compute an approximation of the IEEE factoring $\eta(y)$ under the above rounder input constraints and the further condition that no untrapped overflow before rounding occurs. Note that $\eta(y)$ is the IEEE factoring of the potentially wrapped result.

The η-COMPUTATION circuit furthermore computes *TINY*(x) and *OVF*$_{\mathrm{bef}}(x)$ flags. The basic η-computation algorithm is as follows:

1. Compute the logarithm of f_0 using a leading-zero counter; therefrom decide whether $2^{e_0} \cdot f_0 < 2^{e_{\min}}$, i.e., whether *TINY*$(x)$ holds. Furthermore compute *OVF*$_{\mathrm{bef}}(x)$ as $e_0 > e_{\max} \vee (e_0 = e_{\max} \wedge f_0 \geq 2)$.

2. Compute the exponent $e_1 = \eta_e(y)$ from $\lfloor \log_2 f_0 \rfloor$, $\mathit{TINY}(x)$ and $\mathit{OVF}_{\text{bef}}(x)$. Furthermore compute $e_1^+ := e_1 + 1$. Both e_1 and e_1^+ are returned in biased integer format.
3. For the computation of the significand $\eta_f(y)$, two cases have to be distinguished:

   (a) If no untrapped underflow occurs, then input constraint (13) asserts that $|y| \geq 2^{e_{\min}}$, hence $\eta(y)$ is normal. If the input significand f_0 is denormal, it has to be normalized by means of left-shifting it.

   (b) If an untrapped underflow occurs, we have $|x| < 2^{e_{\min}}$ and $x = y$, and hence $\eta(y)$ is denormal, and therefore $e_1 = e_{\min}$. The significand $\eta_f(y)$ is then computed as $f_0 \cdot 2^{e_0 - e_{\min}}$. If $e_{\min} < e_0$, then f_0 is already "more denormal" than the required result, and therefore f_0 has to be shifted left. This may, e.g., occur due to cancellation during addition.
   If $e_0 < e_{\min}$, f_0 needs to be de-normalized, i.e., right-shifted. If $e_0 \ll e_{\min}$, the exact computation of $f_0 \cdot 2^{e_0 - e_{\min}}$ would require a very far right shift by $\approx e_{\min} - e_0$. For example, the multiplication $2^{e_{\min}} \cdot 2^{e_{\min}}$ yields $e_0 = 2 \cdot e_{\min} \ll e_{\min}$. In double precision, e.g., this would require an $\approx 2^{11}$-bit shifter. Since this very far right shift would require a huge shifter, the η-computation computes only an $(-P)$-equivalent of the exact significand.

   Summarizing, a left-shift is required in case (a) and sometimes in case (b), or a right shift is required in case (b) combined with a sticky-bit computation for the $(-P)$-equivalence. All these situations can be handled by a single 64-bit cyclic shifter combined with a rather complex mask- and control-logic [41, Section 8.4.2].

The construction of the normalization shifter and its correctness are described in [41, pp. 394–404]. We only give the correctness statement of the η-computation for single precision:

Theorem 29. *Let TINY, $\mathit{OVF}_{\text{bef}}$, s, e_n, f_n be the outputs of the η-computation (appropriately interpreted as numbers). For all inputs to the η-computation satisfying the rounder input conditions:*

(i) $\mathit{TINY} = \mathit{TINY}(x)$,
(ii) $\mathit{OVF}_{\text{bef}} = \mathit{OVF}_{\text{bef}}(x)$.

The following statements also require that no untrapped overflow before rounding occurs:

(iii) $(s, e_n[7:0], f_n)$ *is an IEEE factoring*,
(iv) $[\![s, e_n[7:0], f_n]\!] \equiv_\alpha \mathit{wrapped}(x, \mathit{OVFen}, \mathit{UNFen})$ *with* $\alpha = e_n[7:0] - 24$,
(v) $e_n^+[7:0] = e_n[7:0] + 1$.

The statement for double precision is analogous.

The proof of correctness of the above theorem is one of the most complex proofs in [41]. Consequently, the proof was very hard to verify in PVS. The correctness of the η-computation for single and double precision takes 34 lemmas requiring 1480 manually entered prover commands; in [41], the proof is 12 pages long.

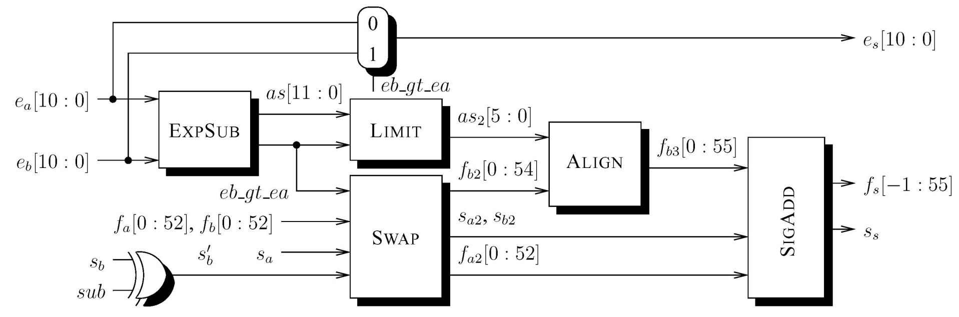

Figure 6. Adder.

3.2.3. Rounder correctness. The 128-bit significand f_n computed by the η-computation is then compressed to the 55-bit f_r by a sticky-bit computation. Next, significand rounding on f_r is performed in circuit SigRd by investigating the 3 least significant bits of f_r, and either chopping or incrementing the higher order bits [31,41].

If significand rounding changes the significand, the inexact result exception *INX* is signaled. The correctness of the *INX* signal follows directly from lemma 16. The *INX* signal is a good example of how a rich theory on IEEE rounding as in Section 2 enables a very simple hardware-level verification.

Putting it all together, we get the following correctness theorem for the rounding unit:

Theorem 30. *For all inputs obeying the rounder input requirements, the rounder outputs the correctly rounded (and potentially wrapped) result, and the correct exception signals.*

3.3. Adder

The floating point adder (figure 6) 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.

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 rounder will round the result to the appropriate precision. We therefore fix $P = 53$ in this section.

3.3.1. Addition algorithm. Let $a := [\![s_a, e_a, f_a]\!]$ and $b := [\![s_b, e_b, f_b]\!]$ be the values of the operands. 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 $S \neq 0$. The special case $S = 0$ is handled by the unpacker.

The informal description of the addition algorithm is

1. The larger 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: $f'_s := (-1)^{s_a} \cdot f_a + (-1)^{s_b} \cdot f'_b$.
5. The result's significand is $f_s := |f'_s|$.
6. The result's sign is $s_s := s_a \oplus (f'_s < 0)$.

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:

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

From Theorem 27 we know that it suffices to supply a value to the rounder that is α-equivalent to the sum S, where $\alpha = \hat{\eta}_e(x) - P$. We prove the following theorem to show that the addition algorithm satisfies the rounder input requirements:

Theorem 31. *Let* $\hat{e} := \hat{\eta}_e(S)$.

$$S \equiv_{\hat{e}-P} 2^{e_a} \cdot ((-1)^{s_a} \cdot f_a + (-1)^{s_b} \cdot f'_b). \tag{14}$$

The adder also fulfills the requirements (12) *and* (13).

Proof: By definition, we have

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

The claim (14) is therefore equivalent to

$$2^{e_a} \cdot ((-1)^{s_a} \cdot f_a + (-1)^{s_b} \cdot 2^{-\delta} \cdot f_b) \equiv_{-(P-\hat{e})} 2^{e_a} \cdot ((-1)^{s_a} \cdot f_a + (-1)^{s_b} \cdot f'_b). \tag{15}$$

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

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

This proves (15) 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 18*(iv,v,vi)* yields

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

By lemma 3.3.2 below we have $\hat{e} - P \geq e_a - (P + 1)$. Lemma 18*(vii)* now 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 (15). The requirement (12) is fulfilled trivially. Requirement (13) is proved by bounding the magnitude of the sum (or difference) of representable numbers. □

Lemma 32. *Let* $\delta := e_a - e_b \geq 2$.

$$\hat{e} - P \geq e_a - (P + 1). \tag{16}$$

Proof: By Lemma 4, (16) is equivalent to

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

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

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

Multiplying with 2^{e_a} and taking logarithms yields (17). 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 $B = 63$).

Lemma 33. *For* $B > P$, *let* $\delta' = \min(\delta, B)$.

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

Proof: The case $\delta \leq B$ is trivial, since $\delta = \delta'$. Let $\delta > B > P$. Then by Lemma 18(ix), we have

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

since both $2^{-\delta} \cdot f_b$ and $2^{-\delta'} \cdot f_b$ are smaller than $2^{-(P+1)}$. □

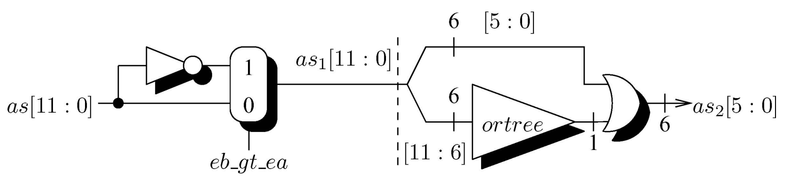

Figure 7. Circuit LIMIT.

3.3.2. Adder hardware. The adder (figure 6) is a straightforward implementation of the described algorithm using the general purpose components from [7]. 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 alignment shift. It primarily consists of a 64-bit shifter and a sticky-bit computation which collects the bits shifted out during the alignment. Circuit SIGADD performs the addition and sign selection, i.e., steps 4–6 from our informal description.

The verification of the adder is straightforward: prove the correctness of the sub-circuits, and combine them using Theorem 31 and Lemma 33.

3.3.3. Verifying the gate level. As an example for the detail level our proofs operate on, we present the LIMIT circuit (figure 7) that calculates the shift distance as_2 for circuit ALIGN. The PVS code for the circuit and Lemmas 34 and 35 are shown in figure 8. In this section, we strictly distinguish numbers and their bitvector representations and denote the binary and 2's complement value of bitvectors by $\langle\cdot\rangle$ and $[\cdot]$, respectively. The set of bits is $\mathbb{B} := \{\mathbf{0}, \mathbf{1}\}$.

The first part of the circuit computes an approximation of the absolute value of the shift distance $[as] = [e_a] - [e_b]$. In case $[e_b] > [e_a]$, the computation introduces an error of 1 in the shift distance. This is done to save the delay of an incrementer that would increase the length of the critical path of the adder. The error introduced by using the 1's complement of as instead is compensated by pre-shifting the significand by one bit in circuit SWAP in this case.

Lemma 34. *Let $eb\_gt\_ea \in \mathbb{B}$, $e_a, e_b \in \mathbb{B}^{11}$, $as, as_1 \in \mathbb{B}^{12}$. With $eb\_gt\_ea = ([e_b] > [e_a])$, the as_1 signal in* LIMIT *satisfies*

$$\langle as_1\rangle = abs([as]) - \begin{cases} 1 & \textit{if } eb\_gt\_ea = \mathbf{1}, \\ 0 & \textit{if } eb\_gt\_ea = \mathbf{0}. \end{cases}$$

Proof: We present a transcript of the PVS proof. We have two cases. For $[as] < 0$, we have to show

$$\langle\neg as\rangle = abs([as]) - 1.$$

```
%%----------------------- HARDWARE -----------------------
or_tree(n: posnat, b: bvec[n]): Recursive bit =
  If n = 1 Then b(0)
  Else Let a = floor(n/2) In
    or_tree(n - a, b^(n-1, a)) Or or_tree(a, b^(a-1, 0))
  Endif
  Measure n;

limit_approx(i): bvec[12] =
  If i'eb_gt_ea Then Not i'as Else i'as Endif;

limit_limit(as1): bvec[6] =
  Let ortree = or_tree(6, as1^(11,6)) In
    Lambda (i: below(6)): as1(i) Or ortree;

%%------------------------ LEMMAS ------------------------
or_tree_correct: Lemma
  Forall (n: posnat, b: bvec[n]):
    Not or_tree(n, b) Iff b = fill[n](False);

limit_approx_correct: Lemma  %% 34 %%
  i'eb_gt_ea = (bv2int(i'as) < 0) Implies
    bv2nat(limit_approx(i)) =
      abs(bv2int(i'as)) - If i'eb_gt_ea Then 1 Else 0 Endif;

limit_limit_correct: Lemma   %% 35 %%
  bv2nat(limit_limit(as1)) =
    If bv2nat(as1) > B Then B Else bv2nat(as1) Endif;
```

Hardware part: The *pvs2hdl* tool translates the underlined parts of the hardware description to Verilog. The remaining parts are used for recursive circuit construction; numerical expressions are evaluated by constant propagation during the translation process. $b(n)$ extracts bit n of bitvector b, $b\hat{}(n, m)$ extracts bits $[n : m]$. `If` is translated to a multiplexer. `Lambda` defines a bitvector by an expression for each individual bit. The expression is translated bit-wise to Verilog, and the individual bits are then concatenated.
Lemma part: `fill`$[n](b)$ is bit b replicated n times. `bv2nat(b)` and `bv2int(b)` are the numerical evaluation of b in unsigned binary format ($\langle b\rangle$) and two's complement format ($[b]$), respectively. The *pvs2hdl* tool ignores all lemmas for the translation.

Figure 8. PVS implementation of circuit LIMIT and correctness lemmas.

This is equivalent to

$$2^{12} - 1 - \langle as\rangle = -[as] - 1.$$

Replacing $[\cdot]$ with $\langle\cdot\rangle$, we have

$$2^{12} - \langle as\rangle = -(\langle as\rangle - 2^{12} \cdot \langle as[11]\rangle).$$

This is true because $as[11] \iff [as] < 0$. For $[as] \geq 0$, we have to prove $[as] = \langle as\rangle$ where $as[11] = \mathbf{0}$, which is trivial. □

The second part of the LIMIT circuit limits $\langle as_1\rangle$ to 63. If one of the high order bits $as_1[10:6]$ is set, then $\langle as_1\rangle > 63$. In this case, the low order bits of as_1 are forced to **1** by the OR gate. Otherwise, the shift distance $\langle as_1\rangle$ is unchanged.

Lemma 35. *The output* $as_2 \in \mathbb{B}^6$ *of* LIMIT *satisfies*

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

Proof: The first case we look at is $\langle as_1\rangle > 63$. We have to show

$$\left\langle as_1[5:0] \vee \bigvee as_1[11:6]\right\rangle = 63.$$

In this case, $as_1[11:6] \neq \mathbf{000000}$, hence $\bigvee as_1[11:6] = \mathbf{1}$.

$$\langle as_1[5:0] \vee \mathbf{1}\rangle = 63.$$

This is trivial. In the case $\langle as_1\rangle \leq 63$, we have to show

$$\left\langle as_1[5:0] \vee \bigvee as_1[11:6]\right\rangle = \langle as_1\rangle.$$

Since $as_1[11:6] = \mathbf{000000}$, and $\bigvee as_1[11:6] = \mathbf{0}$, we have

$$\langle as_1[5:0] \vee \mathbf{0}\rangle = \langle as_1\rangle.$$

□

3.4. Multiplier and divider

The basic algorithm for multiplication is to add up the exponents and multiply the significands. Since denormal input operands are normalized within the unpacker (cf. Section 3.1), this yields a result significand in the interval $[1, 4)$, and hence the rounder input condition (12) is trivially fulfilled.

For divisions, one subtracts the exponents and divides the significands. This yields a quotient significand in the interval $(\frac{1}{2}, 2)$. In order to yield a significand in $[1, 4)$ to match (12), the significand is multiplied by 2, and to compensate for this the exponent is decremented by one.

A major bug of [41] is that the multiplication with 2 is missing for divisions. If this multiplication is omitted, a significand in the interval $(\frac{1}{2}, 2)$ is passed to the rounder. Since the difference of the operand exponents may be less than $e_{\min}$, the rounder input condition (12) may not be satisfied. This leads to unspecified results of the rounding unit. In order to implement the multiplication with 2, some circuits and theorems in the multiplicative unit had to be adjusted.

The above algorithms for multiplication and division may lead to significands with long or even infinite binary representations. We therefore compute α-equivalent approximations of the result significands.

For both operations, the result's sign is the XOR of the operands' signs.

The correctness of the algorithms is asserted by the following theorem:

Theorem 36. *Let (s_a, e_a, f_a) and (s_b, e_b, f_b) be factorings with nonzero values $a = [\![s_a, e_a, f_a]\!]$ and $b = [\![s_b, e_b, f_b]\!]$. Assume $f_a, f_b \in [1, 2)$. Let $\hat{e} = \hat{\eta}_e(a \cdot b)$ for multiplications, and $\hat{e} = \hat{\eta}_e(a/b)$ for divisions. The algorithm described above is correct*:

$$\begin{aligned} a \cdot b &=_{\hat{e}-P} [\![s_a \oplus s_b, e_a + e_b, [f_a \cdot f_b]_{-P}]\!], \\ a/b &=_{\hat{e}-P} [\![s_a \oplus s_b, e_a - e_b - 1, 2 \cdot [f_a/f_b]_{-(P+1)}]\!]. \end{aligned}$$

The representative of the quotient significand has to have one more bit of precision compared to multiplication, since it is multiplied with 2. The resulting significand lies in the interval $[1, 4)$*:*

$$\begin{aligned} &1 \le [f_a \cdot f_b]_{-P} < 4, \\ &1 \le 2 \cdot [f_a/f_b]_{-(P+1)} < 4. \end{aligned}$$

If the operands are representable numbers, the value of the result lies in a range such that exponent wrapping scales the result into the representable range (cf. rounder input condition (13)*):*

$$\begin{aligned} 2^{e_{\min}-A} &< |[\![s_a \oplus s_b, e_a + e_b, [f_a \cdot f_b]_{-P}]\!]| < 2^{e_{\max}+A}, \\ 2^{e_{\min}-A} &< |[\![s_a \oplus s_b, e_a - e_b - 1, 2 \cdot [f_a/f_b]_{-(P+1)}]\!]| < 2^{e_{\max}+A}. \end{aligned}$$

It is easy to implement multiplication with the described algorithm. For the implementation of the division, the problem of computing $[f_a/f_b]_{-(P+1)}$ remains. This is done using Newton-Raphson iteration: starting from an initial approximation of $1/f_b$, one iteratively computes a better approximation $r \approx 1/f_b$. From this approximation r one computes the representative $[f_a/f_b]_{-(P+1)}$.

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 $a \in \{0, \dots, 255\}$ to bitvectors $b \in \mathbb{B}^8$. We have verified the content of the lookup table by automatically checking all 256 entries.

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 [41]. The translation of the proofs to PVS is therefore straightforward.

3.5. Putting it all together

We have verified the unpacker, rounder, and computation units completely independent from each other. Using the formal correctness statement of each unit, it is simple to put all parts together in a completely rigorous way, leaving no holes for errors to slip in.

The correctness of the unpacker, the computation units, and the rounder together imply the correctness of the whole FPU. For example, the adder computes by Theorem 31 a value that is $(\hat{e} - P)$-equivalent to the exact result. By Theorem 30, the rounder correctly rounds this approximation, and computes the exception signals for this approximation. The rounded result and the exception signals are the same as for the exact result by Theorem 27. Hence, the additive FPU as a whole is correct.

A formal theorem about the complete correctness of the FPU involves a lot of special cases like operations on infinity and NaNs, untrapped overflows (Section 2.3.3) etc. We omit the details since they are straightforward; they can be found in [31].

3.6. Errors encountered

We briefly describe some of the errors in [41] that we have encountered during the verification of the FPU in PVS:

The specification of the rounder interface (pp. 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 (pp. 397). However, the requirement is not strong enough; $f_i < 1 \Rightarrow e_i \leq e_{\max}$ must hold. 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 $f_i \approx \frac{1}{2}$ 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. Section 3.4).

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 (pp. 383).

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

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

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

The proofs in [41] 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 [41], 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. While some of the errors would certainly have been found by simply implementing the design and testing it, formal verification is the only way to ensure that no errors are left.

The writing of [41] was preceded and accompanied by lectures Müller and Paul gave at Saarland University. In turn, the errors found during the formal verification as well design improvements have influenced the lectures, and PVS is used in computer architecture courses now.

3.7. Verification effort

Table 1 lists the effort needed for the verification of the different parts of the *VAMP* FPU. There is a large gap between the number of lemmas and theorems in the PVS proofs and those from [41]. This is due to two reasons: first, many seemingly trivial things are not proved in [41]. This, in particular, includes the width of busses, adders, etc. The lack of verification of these "trivial" things was source of several bugs in [41].

Second, a lot of the mathematics in [41] is scattered over the continuous text. A large part of our work was to divide the mathematics in the text from [41] into lemmas. For example, the η-computation circuit within the rounder is described over 12 pages in [41], but has only 3 lemmas, whereas it has 34 lemmas in PVS.

The complexity of the proofs in the individual parts of the FPU have different sources. The vast effort in the verification of the theory of IEEE rounding is due to PVS's limited arithmetic capabilities. This problem might be solved by new arithmetic strategies [20]; however, these were not available when we did the verification. The most time-consuming part in the verification of the FPU datapaths was the verification on the level of single bits. The verification of the lowest-level modules was often very tedious. In particular, this applies to the verification of the general purpose circuits (adders, leading zero counters,

Table 1. Verification effort for the FPUs.

	PVS proofs		Müller/Paul [41]		
	Lemmas	Steps	Lemmas	Pages	Steps/Page
General purpose circuits	107	4032	4	23	175
Theory of rounding	266	4808	9	33	146
Unpacker	13	361	0	5	72
Additive unit	180	3928	1	14	280
Multiplicative unit	106	2817	5	18	157
Miscellaneous unit	33	1616	1	20	81
Rounder	98	4008	5	22	182
Various lemmas	123	895			
Pipelining of the FPUs	120	3471			
$\sum$	1046	25936	25	135	192

The table compares the verification effort in PVS measured as the number of lemmas and the number of interactive proof commands with the number of lemmas and pages in [41]. All in all, the PVS specifications of the FPU take 374 kB of source, and the proof scripts take 1.1 MB (excluding whitespace).

decoders, etc.) [7]. The verification of such circuits could be eased by integrating more automatic verification methods like ACV [11] into PVS, e.g., for simple arithmetic circuits like the LIMIT circuit of Section 3.3.3.

The composition of modules to larger modules was mostly straightforward given the decomposition facilities provided by the theory of rounding.

3.8. Discrepancies to the IEEE standard

The Misc-FPU handles trapped overflows and underflows on conversion from double to single precision differently from the way mandated by the standard. In such cases, our FPU delivers the original operand to the trap handler which then can perform the correct operation in software. In the IEEE standard, it is explicitly allowed to implement some of the functionality in software [28, Section 1.1], so this discrepancy is not even a real discrepancy. However, we have not formally verified the software of the trap handler for these cases.

There are three floating point operations defined in the standard which we have not implemented, namely square root, rounding of floating point numbers to an integral-valued floating point number, and conversion between floating point and decimal formats. We believe the former two operations could be designed and verified with small effort given the experience and techniques presented above. Conversion between floating point and decimal formats might be slightly more complex [15, 16]. All three operations raise an *unimplemented*-trap in the *VAMP* CPU and may be implemented in a trap handler.

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 [41], the FPU is integrated into an in-order variant of the DLX-processor. In our verification project, the FPU is integrated into a Tomasulo based out-of-order DLX-variant by Kröning [36]. It was 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 may be 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. Divisions have a latency of 16 and 20 cycles for single and double precision, respectively. Two divisions can be performed interleaved without increased latency.

We have verified the FPU control using a combination of PVS's model checking and theorem proving capabilities. We omit the control implementation and verification details here, since they are not specific to FPUs. We refer the reader to [31, 32] for details on the construction and verification of the FPU pipelines.

5. Implementing the FPU on an FPGA

In this section we describe the implementation and test of our FPU on a Xilinx FPGA.

5.1. Implementation

Our group has developed a tool called *pvs2hdl* which automatically translates hardware descriptions from PVS to the hardware description language *Verilog*. The combinatorial hardware is described in PVS as exemplarily shown in figure 8. For the handling of clocked circuits and more details about the translation, we refer the reader to [8].

Using the *pvs2hdl* tool, we have implemented all three FPUs on a Xilinx Virtex-E-2000 FPGA. The complete hardware was automatically generated, except for the interface hardware needed for the communication between the FPGA and a host PC. We have replaced our PVS adders, incrementers, and multipliers with Xilinx-specific macros, since these use special FPGA resources and hence use far less area then the counterparts generated from our PVS implementations.

The translation of the multiplicative floating point unit from PVS to Verilog yields a design requiring 4243 FPGA slices. This accounts for $\sim$25% of the Virtex-E 2000 area. The registers within the FPU have 1637 bits. The Xilinx software reports a gate-count of 88,000. The gate-model from [41] estimated a gate-count of 87,000, i.e., it is pretty close to the actual FPGA size. The maximum clock frequency is 16.8 MHz.

The critical path is on the significand path in the first rounder stage. It involves a leading-zero counter on the input significand, a 13-bit adder, and a 53-bit cyclic shifter for the η-computation, and a 103-bit or-tree for the sticky-bit computation. Nearly 80% of the delay are due to routing, only 20% are due to logic delay. These results have been obtained without guiding the place-and-route with a floorplan, though this could significantly reduce the delay of the FPU. However, since our main objective is not on speed but on correctness, we have not yet considered floorplanning.

The additive FPU occupies 1545 slices and runs at 17 MHz. The Misc-FPU occupies 1211 slices and runs at 16 MHz.

5.2. Testing the implementation

We have run several 100,000 random test-vectors on the FPGA implementation of the FPU. The FPU worked correctly from the beginning; no debugging was necessary. We have compared the results of the test-vectors with the Intel FPU inside the Pentium II processor in the host PC. We found two sources of discrepancies between our and Intel's FPU. First, some operations involving *NaN*s are handled differently. However, the IEEE standard [28] is under-specifying in these cases, so that both our and Intel's FPU conform to the standard.

The second source of discrepancies revealed a non-trivial difference of Intel's FPU to ours. Internally, the Intel FPU always operates with an extended precision exponent width of 15 bits. The difference occurs if a double precision operation yields an exponent which is less than the double precision 11-bit $e_{\min}$, but greater than the 15-bit $e_{\min}$ (similarly for single precision results with 8-bit exponent). In IEEE rounding, the significand of this result has to be denormalized with respect to the 11-bit $e_{\min}$ before rounding. In our rounding unit this is accomplished in the η-COMPUTATION (Section 3.2). On Intel's FPU, however, the result is first rounded to the target 53-bit significand precision, but with the internal 15-bit exponent. In a second step, this intermediate result is denormalized. The denormalization shifts out some of the significand bits, and the denormalized significand is then rounded again. This twofold rounding can produce different results than one-step rounding [37].

For example, assume that the normalized significand f is of the form $f = \cdots 001_\lrcorner 01$, where the '$_\lrcorner$' sign denotes the least representable bit, i.e., the significand bit with weight 2^{-52}. Let the corresponding denormalized significand be $f' = \cdots 00_\lrcorner 101$, i.e., f' is obtained by denormalizing f by one bit. For IEEE rounding in round to nearest mode, f' is rounded up to $f'_{rd} = \cdots 01_\lrcorner$. In Intel's FPU, the normalized f is first rounded down to $f_{rd1} = \cdots 001_\lrcorner$. This intermediate result is then denormalized to $\cdots 00_\lrcorner 1$. This results in a tie for rounding to nearest, and hence is rounded to the nearest representable number with least significant bit zero, which is $\cdots 00_\lrcorner$. This differs from f'_{rd} in the least significant representable bit.

We also have tried these operations on AMD processors, which yielded the same results as Intel's FPU. Nevertheless, the results are still IEEE compliant because of Section 4.3 of the standard [28]:

> Normally, a result is rounded to the precision of its destination. However, some systems deliver results only to (...) extended destinations. On such systems the user (...) shall be able to specify that a result be rounded instead to single precision, though it may be stored in the (...) extended format with its wider exponent range. (...).

In our interpretation, the last sentence also applies if the actual destination is double precision which is stored in extended format. A footnote to the above further clarifies:

> Control of rounding precision is intended to allow systems whose destination is always (...) extended to mimic, in the absence of over/underflow, the precisions of systems with single and double precision.

The x86 architecture allows the control of the precision to which the significand is rounded. Only in case of underflow in the actual destination, the double rounding effect described above can occur. Hence, both Intel's/AMD's FPU behavior as well as our behavior is IEEE compliant.

The described problem is known to Intel [23]. Intel's Itanium architecture allows the programmer to control both significand precision and exponent width, thus enabling to choose real double or single precision behavior without the extended precision artifact.

6. Related work

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

As mentioned before, the formalization of the IEEE standard that we use is based on [21, 38, 41]. The notion of factorings, round decomposition, and α-equivalence is taken from [21, 41]. We have formally verified this theory. Since the definition of the rounding function is informal in [21, 41], we use a formal definition of rounding, which is based on Miner's formalization of the standard [38]. Miner's formalization does not comprise theorems related to α-equivalence and round decomposition.

Barrett [5] has formalized parts of the IEEE standard in the specification language Z. His work does not include any verified theorems, but only the translation of the standard to Z.

Harrison has formalized the IEEE standard in the theorem prover HOL Light [25]. Harrison does not discuss exponent wrapping, which introduces some ambiguities in the definition of the inexact exception (cf. Section 2.3). Harrison's formalization has no counterpart to round decomposition. He has theorems related to the computation of exceptions of α-equivalent numbers [25, Section 5.3], but does not relate them to sticky-bit computations. However, this relation is essential to subdivide the FPU into computational units and a rounder unit in our verification project. Neither Miner nor Harrison cover the actual implementation of operations or rounding.

Daumas et al. have formalized theorems for reasoning about floating point numbers in Coq [19]. They concentrate on properties of the rounding function rather than support for verification of floating point hardware.

Moore et al. have verified the AMD K5 division algorithm [40] with the theorem prover ACL2. They have a definition of sticky bit computations that is similar to our α-equivalence. They do not cover exceptions and round decomposition.

Russinoff has verified the K5 square root algorithm as well as the AMD Athlon multiplication, division, square root, and addition algorithms [45–47]. In all his verification projects, Russinoff proves the correctness of a register transfer level implementation against his formalization of the IEEE standard using ACL2. His formalization of the rounding function and sticky bit computations is similar to [40]. Russinoff does not handle exceptions and denormals in his publications; he states that he handles denormals in unpublished work (private communication). However, the above mentioned discrepancy of Intel's FPU to the IEEE standard in some cases where denormal numbers are involved also applies to AMD's FPU (Section 5.2).

Aagaard and Seger combine BDD based methods and theorem proving techniques to verify a floating point multiplier [2]. Chen and Bryant [12] use word-level model checking to verify a floating point adder. Exceptions and denormals are handled in neither verification project.

Verkest et al. verify a non-restoring integer division algorithm [49]. Clarke et al. [14] and Ruess et al. [44] verify SRT division algorithms. Miner and Leathrum [39] verify a general class of subtractive division algorithms with respect to the IEEE formalization of Miner [38]. Mechanized proofs of SRT integer division are reported in [10, 35].

In [24], Harrison proves the correctness of an algorithm for the exponential function against his IEEE formalization. He assumes that IEEE-correct addition, multiplication, and rounding to integer are provided. In [3, 4], Abdel-Hamid et al. verify an implementation of this algorithm against a formal specification. However, this specification is nearer to the gate-level implementation than to a high-level formalization of the IEEE standard.

O'Leary et al. [42] report on the verification of the gate level design of Intel's FPU using a combination of model checking and theorem proving. Their definition of rounding does not reflect the IEEE standard in an obvious way. Denormals and exceptions are not covered in the paper. In fact, in our tests of our FPU against the Intel FPU we have encountered differences in the rounding of denormal numbers, which are due to discrepancies of Intel's rounding to the IEEE standard (Section 5.2).

In [1], Aagaard et al. report on the verification of gate-level implementations of iterative algorithms. Among other circuits, they verify floating point square root, division, and remainder operations. They do not give details on the specification against which the circuits are verified.

In [33], Kaivola and Kohatsu report on the verification of Intel's Pentium 4 floating point divider. The main focus of their paper is not the actual divider verification, but the challenges formal verification has to overcome in an industrial setting. In [34], Kaivola and Narasimhan describe the formal verification of the Pentium 4 multiplier from a similar perspective. The verification covers exception signals, but does not cover denormal numbers.

Cornea-Hasegan [17, 18] describes algorithms for the computation of division and square root by Newton-Raphson iteration in the Intel FPUs. The verification is done using paper-and-pencil proofs supported by *Mathematica*, a computer algebra system.

In [13], Chen et al. verify the correctness of sub-circuits of Intel's Pentium Pro floating point unit. They leave out the composition of these sub-circuits, and the formal reasoning why this composition is correct. In fact, the "verified" Pentium Pro had a bug in conversion from floating point to integer format, the so-called *FIST* bug [29]. According to [42], the bug has escaped the verification in [13] because Chen et al. did not formally compose all parts of the system. It is therefore comparable to the Müller/Paul [41] division bug described in Section 3.4, which is also due to not "putting it all together" in a formal way.

7. Summary and discussion

We have formally verified an 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 designs and proofs scripts are publicly available [48].

We tackle the verification from three sides. First, we formalize the IEEE standard and develop a rich library of definitions and theorems about IEEE rounding. These definitions and theorems later enable the simple decomposition of the hardware into smaller building blocks that can be verified independently. For example, the concept of α-equivalence allows us to decompose the whole FPU into computation units like the adder from Section 3.3, and rounding unit (Section 3.2). The adder and the rounder have been verified independently by the two authors, using the rounder interface from Section 3.2.1.

Secondly, we verify the algorithmic correctness of large building blocks on the level of real numbers rather than on bits and bitvectors. A good example for this is Theorem 31, which states the correctness of the addition algorithm. This theorem enables the further decomposition of the adder block into smaller sub-blocks, as shown in figure 6. The correctness statements of the individual blocks, e.g., Lemma 35 about circuit LIMIT (figure 7), are then composed to the correctness theorem of the adder using the algorithmic correctness theorems.

After the decomposition of the hardware into smaller blocks, in the third track the small blocks are composed from general purpose building-blocks like adders, OR-trees, leading zero counters, etc. We have developed a library of such general purpose circuits [7]. With each circuit comes an algebraic correctness statement, stating for instance, that the leading zero counter counts the numbers of leading zeros correctly. Blocks like the LIMIT circuit are built from such general-purpose circuits. The correctness proof of LIMIT depends on the correctness theorem of the OR-tree (cf. Lemma 35). In order to establish the correctness of this lemma, the gates hidden in the OR-tree are not considered, they are encapsulated in the correctness statement of the OR-tree. Most of the gates in the FPU are hidden in general-purpose circuits in that way.

The proofs in PVS used paper proofs from [41] as guidelines. However, some of the proofs in [41] were erroneous, and most proofs had gaps which needed to be filled in PVS. Those gaps hid errors in the design in [41]. Having formally verified the proofs (and filled the proof gaps) in PVS gives us the certainty that the hardware is now correct with respect to its specification.

The amount of work needed to develop the PVS IEEE library and hardware description and the proofs was roughly three man-years. Since theorem proving strongly profits from experience, we think we would succeed in at most half the time now on a comparable project.

The FPU verification did work very well in the general purpose, interactive theorem prover PVS. However, parts of the verification were very technical and tedious. This applies in particular to verification on the level of single gates. The PVS verification of complex hardware such as our FPU could benefit from the integration of more automatic verification methods on the lowest hardware level (e.g. ACV [11]). One could verify small modules using such automatic methods, and apply interactive theorem proving only to the composition of such modules to larger units. This would turn PVS into a very powerful hardware verification system.

Acknowledgments

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

Notes

1. The term α-*equivalence* is not related to the term as used in λ-calculus.
2. We omit the proofs if they are trivial or the claims follow directly from previous lemmas.
3. $\widehat{norm}_f()$ denotes the f-component of the triple $\widehat{norm}()$; analogous for other functions and components.
4. We distinguish $+0$ and -0 in our theory of factorings, but for the conversion from reals to factorings we convert $0 \in \mathbb{R}$ to $+0$.
5. The test-program is available at the project website [48].
6. In [41], η-computation is called *normalization shift*. We believe this term is confusing, since η-computation does not always normalize but may de-normalize the inputs in some cases.

References

1. M.D. Aagaard, R.B. Jones, and R. Kaivola, "Formal verification of iterative algorithms in microprocessors," in *DAC 2000*. ACM, 2000.
2. M.D. Aagaard and C.-J.H. Seger, "The formal verification of a pipelined double-precision IEEE floating-point multiplier," in *ICCAD*, IEEE, Nov. 1995, pp. 7–10.
3. A.T. Abdel-Hamid, "A hierarchical verification of the IEEE-754 table-driven floating-point exponential function using HOL," Master's thesis, Dept. Electrical and Computer Engineering, Concordia University, Montréal, Québec, Canada, 2001.
4. A.T. Abdel-Hamid, S. Taher, and J. Harrison, "Table-driven floating-point exponential function using HOL," in R.J. Boulton and P.B. Jackson (Eds.), *TPHOLs 2001: Supplemental Proceedings*, 2001, Informatics Research Report EDI-INF-RR-0046, Univ. Edinburgh, UK.
5. G. Barrett, "Formal methods applied to a floating-point number system," *IEEE Transactions on Software Engineering*, Vol. 15, No. 5, pp. 611–621, 1989.
6. C. Berg and C. Jacobi, "Formal verification of the VAMP floating point unit," in *Proc. 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *LNCS*, Springer, 2001, pp. 325–339.
7. C. Berg, C. Jacobi, and D. Kroening, "Formal verification of a basic circuits library," in *Proc. of the IASTED International Conference on Applied Informatics, Innsbruck (AI 2001)*, ACTA Press, 2001.
8. S. Beyer, C. Jacobi, D. Kroening, and D. Leinenbach, "Correct hardware by synthesis from pvs. Unpublished," Available at `http://busserver.cs.uni-sb.de/publikationen/BJKL02.pdf`, 2002.
9. S. Beyer, C. Jacobi, D. Kroening, D. Leinenbach, and W.J. Paul, "Instantiating uninterpreted functional units and memory system: Functional verification of the VAMP processor," in *Proceedings of CHARME 2003*, volume 2860 of *LNCS*, Springer, 2003, pp. 51–65.
10. R.E. Bryant, "Bit-level analysis of an SRT divider circuit," in *33rd Design Automation Conference (DAC'96)*, ACM, June 1996, pp. 661–665.
11. Y.-A. Chen and R.E. Bryant, "ACV: An arithmetic circuit verifier," in *Proc. of IEEE ICCD '96*, IEEE, 1996, pp. 361–365.
12. Y.-A. Chen and R.E. Bryant, "Verification of floating point adders," in *CAV'98*, volume 1427 of *LNCS*, 1998.
13. Y.-A. Chen, E.M. Clarke, P.-H. Ho, Y. Hoskote, T. Kam, M. Khaira, J.W. O'Leary, and X. Zhao, "Verification of all circuits in a floating-point unit using word-level model checking," in *Formal Methods in Computer-Aided Design*, volume 1166 of *LNCS*, Springer, 1996, pp. 19–33.
14. 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.
15. W.D. Clinger, "How to read floating-point numbers accurately," in *SIGPLAN Conference on Programming Language Design and Implementation*, 1990, pp. 92–101.
16. J.T. Coonen, "An implementation guide to a proposed standard for floating point arithmetic," *COMPUTER*, Vol. 13, No. 1, pp. 68–79, 1980.
17. M. Cornea-Hasegan, "Proving the IEEE correctness of iterative floating-point square root, divide, and remainder algorithms," *Intel Technology Journal*, Q2, 1998.
18. M. Cornea-Hasegan, "IA-64 floating point operations and the IEEE standard for binary floating-point arithmetic," *Intel Technology Journal*, Q4, 1999.
19. M. Daumas, L. Rideau, and L. Théry, "A generic library for floating-point numbers and its application to exact computing," in *Proc. of TPHOLs 2001*, volume 2152 of *LNCS*, 2001, P. 169.
20. B.L. Di Vito. *Manip: A PVS Prover Strategy Package for Common Manipulations*. NASA Langley Research Center, Hampton, VA, 2002. available at `shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html`.
21. 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.
22. D. Goldberg, Computer arithmetic, 1996, in [26].
23. R. Golliver, "Efficiently producing default orthogonal IEEE double results using extended IEEE hardware," Talk at 3rd Meeting of the Java Study Group, 1998. available at `std.dkuug.dk/JTC1/SC22/JSG/docs/m3/docs/jsgn326.pdf`.

24. J. Harrison, “Floating point verification in HOL light: The exponential function,” in *Algebraic Methodology and Software Technology*, 1997, pp. 246–260.
25. J. Harrison, “A machine checked theory of floating point arithmetic,” in *Proc. of TPHOL ’99*, volume 1690 of *LNCS*, Springer, 1999.
26. J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd ed. Morgan Kaufmann, San Mateo, CA, 1996.
27. IBM, *z/Architecture Principles of Operation*, Poughkeepsie, NY, Dec. 2000.
28. Institute of Electrical and Electronics Engineers. *ANSI/IEEE standard 754–1985, IEEE Standard for Binary Floating-Point Arithmetic*, 1985.
29. *Discussion of Flag Erratum.* `support.intel.com/support/processors/flag/tech.htm`, 2002.
30. C. Jacobi, “Formal verification of a theory of IEEE rounding,” in R.J. Boulton and P.B. Jackson (Eds.), *TPHOLs 2001: Supplemental Proceedings*, 2001, Informatics Research Report EDI-INF-RR-0046, Univ. Edinburgh, UK.
31. C. Jacobi, “Formal verification of a fully IEEE compliant floating point unit,” PhD thesis, Saarland University, Germany, 2002. Available at `http://de.geocities.com/christianjacobi/`.
32. C. Jacobi, “Formal verification of complex out-of-order pipelines by combining model-checking and theorem-proving,” in *Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Springer, 2002, pp. 309–232.
33. R. Kaivola and K. Kohatsu, “Proof engineering in the large: Formal verification of pentium 4 floating-point divider,” in *Proc. 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *LNCS*, Springer, 2001.
34. R. Kaivola and N. Narasimhan, “Formal verification of the pentium 4 floating-point multiplier,” in *Design, Automation and Test in Europe Conference and Exposition (DATE).* IEEE, 2002.
35. D. Kapur and M. Subramaniam, “Mechanizing verification of arithmetic circuits: SRT division,” in *FSTTCS*, volume 1346 of *LNCS*, 1997, pp. 103–.
36. D. Kroening, “Formal verification of pipelined microprocessors,” PhD thesis, Saarland University, Computer Science Department, 2001.
37. C. Lee, “Multistep gradual rounding,” *IEEE Transactions on Computers*, Vol. 38, No. 4, 1989.
38. P.S. Miner, “Defining the IEEE-854 floating-point standard in PVS,” Technical Report TM-110167, NASA Langley Research Center, 1995.
39. P.S. Miner and J.F. Leathrum, “Verification of IEEE compliant subtractive division algorithms,” in *FMCAD-96*, volume 1166 of *LNCS*, 1996, pp. 64–.
40. J.S. Moore, T. Lynch, and M. Kaufmann, “A mechanically checked proof of the AMD5K86 floating point division program,” *IEEE Transactions on Computers*, Vol. 47, No. 9, pp. 913–926, 1998.
41. S.M. Mueller and W.J. Paul, *Computer Architecture. Complexity and Correctness*, Springer, 2000.
42. J. O’Leary, X. Zhao, R. Gerth, and C.-J.H. Seger, “Formally verifying IEEE compliance of floating-point hardware,” *Intel Technology Journal*, Q4, 1999.
43. S. Owre, N. Shankar, and J.M. Rushby, “PVS: A prototype verification system,” in *CADE 11*, volume 607 of *LNAI*, Springer, 1992, pp. 748–752.
44. H. Ruess, N. Shankar, and M.K. Srivas, “Modular verification of SRT division,” in *CAV’96*, volume 1102 of *LNCS*, 1996.
45. 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*, Vol. 1, pp. 148–200, 1998.
46. D.M. Russinoff, “A mechanically checked proof of correctness of the AMD K5 floating point square root microcode,” *Formal Methods in System Design*, Vol. 14, No. 1, pp. 75–125, 1999.
47. 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 *Proceeding of FMCAD-00*, volume 1954 of *LNCS*, Springer, 2000.
48. The VAMP Project Homepage. `www-wjp.cs.uni-sb.de/forschung/projekte/VAMP/`, 2003.
49. 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*, vol. 4, 1994.