Formal Verification of a Fully IEEE Compliant Floating Point Unit

Dissertation

zur Erlangung des Grades
Doktor der Ingenieurswissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I der
Universität des Saarlandes

Christian Jacobi

cj@cs.uni-sb.de

Saarbücken, April 2002

Dekan:	Prof. Dr. Philipp Slusallek
Erstgutachter:	Prof. Dr. Wolfgang J. Paul
Zweitgutachter:	Prof. Dr. Harald Ganzinger
Tag des Kolloquiums:	25. Oktober 2002

Hiermit erkläre ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Saarbrücken, im April 2002

Mathematical proofs, like diamonds, are hard as well as clear, and will be touched with nothing but strict reasoning.
— John Locke

There's always one more bug.
— Murphy

Danke

An dieser Stelle möchte ich allen danken, die zum Gelingen der vorliegenden Arbeit beigetragen haben.

Mein Dank gilt zunächst meinen Eltern, die mich während der gesamten Zeit meiner Ausbildung gefördert haben.

Herrn Prof. Paul danke ich für die Unterstützung während meines Studiums und meiner Promotion.

Danken möchte ich meinen Freunden

- Christoph Berg und Jochen Preiß für viele hilfreiche Diskussionen, das Korrekturlesen der Arbeit und die vielen Skat-Abende;
- Christoph Berg, Sven Beyer, Daniel Kröning, Dirk Leinenbach und Carsten Meyer für die hervorragende Zusammenarbeit im *VAMP* Projekt;
- allen Mitarbeitern des Lehrstuhls Paul für das gute Arbeits-Klima;
- Jan Pessenlehner für den Gas-Herd.

Abstract

In this thesis we describe the formal verification of a fully IEEE compliant floating point unit (FPU). The hardware is verified on the gate-level against a formalization of the IEEE standard. The verification is performed using the theorem proving system PVS. The FPU supports both single and double precision floating point numbers, normal and denormal numbers, all four IEEE rounding modes, and exceptions as required by the standard.

Beside the verification of the combinatorial correctness of the FPUs we pipeline the FPUs to allow the integration into an out-of-order processor. We formally define the correctness criterion the pipelines must obey in order to work properly within the processor. We then describe a new methodology based on combining model checking and theorem proving for the verification of the pipelines.

Kurzzusammenfassung

Die vorliegende Arbeit behandelt die formale Verifikation einer vollständig IEEE-konformen Floating Point Unit (FPU). Die Hardware wird auf Gatter-Ebene gegen eine Formalisierung des IEEE Standards verifiziert. Zur Verifikation wird das Beweis-System PVS benutzt. Die FPU unterstützt Fließkommazahlen mit einfacher und doppelter Genauigkeit, normale und denormale Zahlen, alle vier Rundungsmodi und alle Exception-Signale.

Neben der Verifikation der kombinatorischen Schaltkreise werden die FPUs gepipelined, um sie in einen Out-of-order Prozessor zu integrieren. Die Korrektheits-Kriterien, die die gepipelineten FPUs befolgen müssen, um im Prozessor korrekt zu arbeiten, werden formal definiert. Es wird eine neue Methode zur Verifikation solcher Pipelines beschrieben. Die Methode beruht auf der Kombination von Model-Checking und Theorem-Proving.

Extended Abstract

In this thesis we report on the verification of a fully IEEE compliant floating point unit (FPU). The verification is performed on the gate level against a formalization of the IEEE standard by means of the theorem proving system PVS [OSR92]. The design of the FPU and the formalization of the IEEE standard are based on the textbook on computer architecture by Müller and Paul [MP00]. We extend their work by formally verifying the designs and the formalization of the standard. We have found several errors in the designs as well as in the theory.

The verification is divided into three parts. We first describe the formalization of the IEEE standard. This includes a formalization of normal and denormal numbers and the normalization algorithm. We then define the rounding function. All four rounding modes from the IEEE standard are captured. We prove that the rounding function conforms to the standard. Next, we define the five exceptions from the standard and exponent wrapping. We then describe the concept of α-equivalence from [EP97, MP00]. α-equivalence partitions the real numbers into equivalence classes such that equivalent numbers are rounded the same and yield the same IEEE exceptions, which is also formally proved. We then describe the encoding of floating point numbers in bitvectors and formally define the correctness of the supported floating point operations.

The second part of this thesis covers the verification of the actual floating point hardware against the formalization presented before. We verify three separate FPUs, one for addition and subtraction, one for multiplication and division, and one for comparisons, format conversions, and various miscellaneous operations.

Each FPU is divided into three parts which are verified separately and then are combined to the complete FPU. The first part is an unpacker which converts the operands into some more convenient internal format. It follows the computation unit which performs the actual operation, e.g., an addition or division. The computation units do not need to compute an exact result, but an α-equivalent approximation. The approximations are fed into the rounding unit which computes the correctly rounded result and the exception flags. By the properties of α-equivalence it follows that the approximation is rounded to the same result as the exact result would have been.

The decomposition of the FPUs into unpacker, computation unit, and rounding unit eases the verification, since each part can be verified separately. Using the precise, mathematical specifications of each part, the parts can then be composed in a rigorous way.

The verified FPUs are used in an out-of-order processor. In order to exploit the capabilities of this processor, the FPUs are pipelined. The pipelines may process multiple instructions simultaneously, may have variable latency, and may reorder instructions internally. For the iterative division algorithm, the pipeline has a cycle in the pipeline structure. We formally describe the correctness criterion the pipelined FPUs shall obey in order to work properly inside the processor. We have

developed a new methodology based on combining model checking and theorem proving for the verification of the pipelines.

This thesis is part of a larger project at Saarland University which aims at the formal verification of a complete microprocessor including caches and the floating point units from this thesis. Our group has developed a tool which automatically translates hardware specifications from the theorem prover PVS to the hardware description language Verilog. Using this tool, we have implemented and tested the FPU and the complete processor on a Xilinx FPGA. We give a detailed project description and status at the end of this thesis.

Zusammenfassung

Die vorliegende Arbeit behandelt die formale Verifikation einer vollständig IEEE-konformen Floating Point Unit (FPU). Die Hardware wird auf Gatter-Ebene gegen eine Formalisierung des IEEE Standards verifiziert. Zur Verifikation wird das Beweis-System PVS [OSR92] benutzt. Das FPU-Design und die Formalisierung des IEEE Standards basieren auf dem Lehrbuch über Computer-Architektur von Müller und Paul [MP00]. Wir erweitern die Arbeit von Müller und Paul, indem wir die Designs und die Formalisierung des Standards formal verifizieren. Wir haben mehrere Fehler in den Designs und in der Theorie gefunden.

Die Verifikation ist in drei Teile aufgeteilt: zunächst beschreiben wir die Formalisierung des IEEE Standards. Diese beinhaltet eine Formalisierung von normalen und denormalen Zahlen und des Normalisierungsalgorithmus'. Danach definieren wir die Rundungsfunktionen. Alle vier Rundungsmodi aus dem Standard werden behandelt. Wir beweisen, dass die Rundungsfunktionen dem Standard entsprechen. Anschließend definieren wir alle fünf Exceptions aus dem Standard und Exponent Wrapping. Wir beschreiben dann das Konzept der α-Äquivalenz aus [EP97, MP00]. α-Äquivalenz partitioniert die reellen Zahlen in Äquivalenz-Klassen, so dass äquivalente Zahlen gleich gerundet werden und die selben Exceptions auslösen. Dies wird ebenfalls formal bewiesen. Danach beschreiben wir die Einbettung von Fließkommazahlen in Bitvektoren und definieren formal die Korrektheit der unterstützten Operationen.

Der zweite Teil der Arbeit behandelt die Verifikation der eigentlichen Fließkomma-Hardware bezüglich der vorher beschriebenen Spezifikation. Wir verifizieren drei getrennte FPUs: eine für Addition/Subtraktion, eine für Multiplikation/Division, und eine für Vergleich/Konvertierung und einige weitere Operationen.

Jede FPU ist in drei Teile zerlegt, die separat verifiziert werden und später zur kompletten FPU zusammengesetzt werden. Der erste Teil ist der Unpacker, der die Operanden in ein geeigneteres internes Format umwandelt. Es folgt die Berechnungs-Einheit, die die eigentliche Operation ausführt. Die Berechnungs-Einheit braucht nicht das exakte Ergebnis zu berechnen, sondern nur eine α-äquivalente Approximation. Diese Approximation wird dann an den Runder übergeben, der das korrekt gerundete Ergebnis und die Exceptions berechnet. Die Eigenschaften der α-Äquivalenz garantieren, dass die Approximation genauso gerundet wird und die selben Exceptions auslöst, wie es dass exakte Ergebnis würde.

Die Zerlegung der FPUs in Unpacker, Berechnungs-Einheit und Runder erleichtert die Verifikation, da jeder Teil einzeln verifiziert werden kann. Die Teile werden dann mit Hilfe ihrer präzisen mathematischen Spezifikation zusammengesetzt.

Die verifizierten FPUs werden in einen Out-of-order Prozessor eingebettet. Um die Möglichkeiten dieses Prozessors auszunutzen, werden die FPUs gepipelined. Die Pipelines können mehrere Instruktionen gleichzeitig ausführen, haben variable Latenz, und können die Instruktionen intern umordnen. Die Pipelines haben Zyklen

in ihrer Struktur, um den iterative Divisions-Algorithmus zu implementieren. Wir definieren formal die Korrektheits-Kriterien, die die gepipelineten FPUs erfüllen müssen, um innerhalb des Prozessors korrekt zu funktionieren. Wir beschreiben eine neue Methode zur Verifikation solcher Pipelines. Die Methode beruht auf der Kombination von Model-Checking und Theorem-Proving.

Die vorliegende Arbeit ist Teil eines größeren Projekts an der Universität des Saarlandes, welches die formale Verifikation eines kompletten Prozessors zum Ziel hat. Der Prozessor beinhaltet Caches und die FPUs aus dieser Arbeit. Unsere Gruppe hat ein Programm entwickelt, welches Hardware-Spezifikationen in PVS automatisch in die Hardware-Beschreibungssprache Verilog übersetzt. Mit Hilfe dieses Programms haben wir die FPU und den gesamten Prozessor auf einem Xilinx FPGA implementiert. Wir beschreiben das Projekt ausführlich am Ende dieser Arbeit.

Contents

List of Figures

Chapter 1

Introduction

Over the last decades, microprocessors have become commonly used within many applications. In particular, microprocessors are being used in life-critical environments such as automobiles, air planes, power plants and medical instrumentations. Hence, the correctness of microprocessors is of vital importance.

Simultaneously to the upcoming of microprocessors in everyday's life, the complexity of the processors grew so large that the traditional way of asserting correctness by testing and simulation is now unsatisfactory, at least for safety-critical applications. Furthermore, the cost of errors in microprocessors is gigantic. Probably the most popular example is the Pentium bug [Pra95], which cost Intel nearly half a billion dollar in 1995. One may expect that a similar bug today would cost the tenfold.

Formal verifcation offers a means to rigorously check the correctness of processors. Our group at Saarland University is currently working on the formal verification of a microprocessor called *VAMP* (for *Verified Architecture MicroProcessor*). The *VAMP* has many complex features also found in contemporary commercial micropocessors: it features pipelined out-of-order execution, precise interrupts, a cache hierarchy, and a floating point unit.

In this thesis, we consider the verification of the *VAMP* floating point unit (FPU). The FPU is developed in the textbook on computer architecture by Müller and Paul [MP00]. Along with the complete designs come paper-and-pencil proofs for the correctness of the circuits. These proofs served as guidelines for the formal verification in the theorem proving system PVS [OSR92]. The FPU is verified on the gate-level against a formalization of the IEEE standard 754 for binary floating point arithmetic [IEEE]. The formalization of the IEEE standard is based on [Min95, EP97, MP00].

The FPU is fully IEEE compliant. It features both single and double precision operations. All four rounding modes specified in the IEEE standard are implemented. Denormal numbers are handled completely in hardware, and floating point exceptions are computed as required by the standard. The operations supported by the FPU are addition, subtraction, multiplication, division, comparison,

conversions, and various others.

In order to implement the FPU with reasonable cycle time, the FPUs are pipelined. The pipelined FPUs may process multiple operations simultaneously, and the operations have variable latency. Furthermore, the operations may be reordered internally, i.e., they need not leave the pipeline in the order they enter it. We have developed a new methodology combining model checking and theorem proving to verify the correctness of the pipelines.

The presented FPU is the first formally verified, fully IEEE compliant floating point unit which is publicly available. The PVS files can be found at our web site[1].

Outline

Chapter 2 briefly describes the theorem prover PVS. We present a summary of PVS's bitvector library, and describe how combinatorial and clocked hardware is designed and specified in PVS. As an example, we present the construction of a simple carry-chain adder.

The main work of this thesis is presented in chapters 3–5. Chapter 3 presents the formalization of the IEEE standard against which the floating point hardware is verified. Furthermore, theorems and notations facilitating the hardware verification are presented. Chapter 4 presents the verification of the combinatorial floating point hardware. In chapter 5, we describe the verification of the pipelining of the combinatorial FPUs.

We present an overview of the *VAMP* project in chapter 6. Chapter 7 summarizes the thesis and discusses benefits and drawbacks of our approach to the verification and implementation of complex hardware. Finally, we give a brief outlook to future work.

Related work is discussed at the end of each chapter.

[1] `http://www-wjp.cs.uni-sb.de/~cj/PhD/`, see also the *VAMP* homepage `http://www-wjp.cs.uni-sb.de/projects/verification/`

Chapter 2

The Prototype Verification System

The Prototype Verification System (PVS) [OSR92] is a general-purpose interactive theorem prover developed at SRI International. The PVS system is based on typed higher-order logic within a Genzen-like sequent calculus [Gen35]. PVS features an expressive specification language, powerful decision procedures, e.g., for linear arithmetic, and a μ-calculus model-checker [RSS95]. We will not describe PVS in detail here; we refer the reader to various tutorials and manuals on PVS [COR$^+$95, OSRSC99a, OSRSC99b, SORSC99].

In the following, we will give a brief overview of PVS's bitvector library which we use throughout this thesis. We then describe how combinatorial and sequential hardware is modeled in PVS. Exemplarily, we describe the construction of a carry-chain adder. This is not intended to provide a deep understanding of PVS, but only to give an idea of the way we design and verify hardware.

Except for the construction of the carry-chain adder in section 2.2 we use standard mathematical notation instead of the PVS syntax throughout this thesis. The proofs in this thesis are proofs in mathematical textbook fashion which are extracted from the actual PVS proofs. Using standard mathematical notation eases readability of the definitions and proofs. For reference, we list the PVS names of all lemmas and theorems from this thesis in Appendix C.

2.1 Bits and Bitvectors

In this section, we give a short summary of PVS's bitvector library [BMS$^+$96]. The type of bits is defined as $\mathbb{B} := \{\mathbf{0}, \mathbf{1}\}$. The set of bitvectors of length $n \in \mathbb{N}^+$ is denoted by $\mathbb{B}^n$. PVS distinguishes bitvectors of length 1 from single bits; for the sake of readability we ignore this distinction in this thesis.

Let bv, bv' be bitvectors. The i^{th} bit of bitvector bv is denoted by $bv[i]$. The sub-bitvector $bv[j] \ldots bv[i]$ is denoted by $bv[j : i]$. The concatenation of bv and

bv' is denoted by $bv \circ bv'$. A bitvector of length n consisting solely of b's ($b \in \mathbb{B}$) is denoted by b^n. The bit-wise connectives $\wedge, \vee, \oplus$, and $\neg$ on bitvectors of equal length are defined in the usual way.

Number Representations. The natural number represented by $bv \in \mathbb{B}^n$ is defined as

$$\langle bv \rangle := \sum_{i=0}^{n-1} 2^i \cdot bv[i]. \tag{2.1}$$

The two's complement value of bv is

$$[bv] := \begin{cases} \langle bv \rangle & \text{if } \langle bv \rangle < 2^{n-1}, \\ \langle bv \rangle - 2^n & \text{otherwise.} \end{cases} \tag{2.2}$$

The range of the n-bit two's complement numbers is

$$T_n := \{-2^{n-1}, \ldots, 2^{n-1} - 1\}. \tag{2.3}$$

The proof that T_n indeed is the range of the n-bit two's complement numbers can be found in the bitvector library.

The PVS bitvector library provides a large number of lemmas on bitvectors and the numbers represented by them. For instance, one of the most-often used lemmas states for bitvectors $bv \in \mathbb{B}^n, bv' \in \mathbb{B}^m$:

$$\langle bv \circ bv' \rangle = \langle bv \rangle \cdot 2^m + \langle bv' \rangle. \tag{2.4}$$

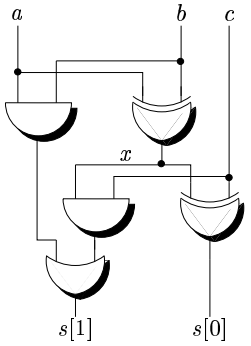
In the following, we will often use the lemmas from the bitvector library without explicitly quoting them.

The rest of this section is not part the PVS bitvector library. For the definition of IEEE floating point numbers in section 3.6 we need one further integer number format, namely *biased integer* format. For n-bit numbers, let $\mathit{bias}_n := 2^{n-1} - 1$. The biased integer value of $bv \in \mathbb{B}^n$ is defined as

$$[bv]_{\text{bias}} := \langle bv \rangle - \mathit{bias}_n. \tag{2.5}$$

Furthermore, we need a notion of binary fractions. Let $bv \in \mathbb{B}^n$. When bv shall be interpreted as a number with k digits behind the binary point, its value is $\langle bv \rangle \cdot 2^{-k}$. In order to use the large number of bitvector lemmas, we do not introduce a new type for binary fractions, but reuse the standard definition (2.1) and scale by 2^{-k}.

We often write $bv \in \mathbb{B}^{m+k}$ to denote bitvectors which are interpreted as binary fractions with m bits before and k bits behind the binary point. The formal meaning of $\mathbb{B}^{m+k}$ is exactly the same as $\mathbb{B}^n$ for $n = m + k$; $\mathbb{B}^{m+k}$ is only a notational hint that bv should be interpreted as a binary fraction.




```
fulladder(a, b, c: bit): bvec[2] =
  LET x = a XOR b IN
      ((a AND b) OR (c AND x)) o
      (x XOR c);

fa_correct: LEMMA
   FORALL(a,b,c: bit):
     bv2nat(fulladder(a,b,c)) =
       bv2nat(a) + bv2nat(b) + bv2nat(c)
```

'o' means bit-concatenation. `bvnat(·)` reflects the binary value, i.e., $\langle\cdot\rangle$.

Figure 2.1: Construction and correctness statement of a full adder

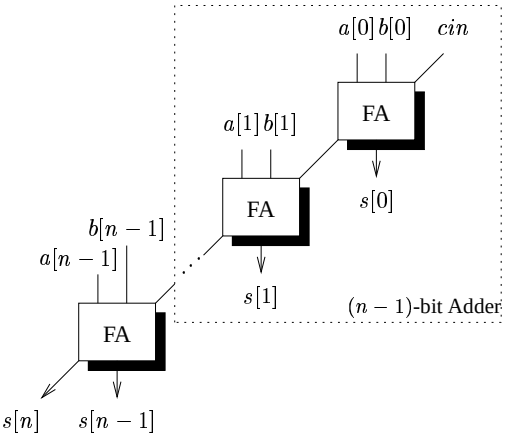
Notation. For reasons of readability, we often intermix bitvectors and the numbers represented by them in the text. For example, f may denote both the significand's value or the signficand's bitvector representation of a floaiting point number. If the precise distinction of numbers and their bitvector representation is beneficial for the understanding, we use narrow letters f for numbers, and bold letters $\mathbf{f}$ for bitvectors.

2.2 Designing Combinatorial Hardware in PVS

In this section, we briefly describe how combinatorial hardware is designed in PVS. As an example we use a simple carry-chain adder.

The PVS language supports the (recursive) definition of functions which call other functions similar to a functional programming language. We use functions to model combinatorial hardware modules. For example, a full adder can be seen as a function which maps three inputs $a, b, c \in \mathbb{B}$ to a 2-bit output $s \in \mathbb{B}^2$. Figure 2.1 compares the construction of such a full adder using schematics and using the PVS language. The correctness of the full adder is asserted in the lemma `fa_correct` also shown in figure 2.1.

Analogously, an n-bit carry-chain adder can be seen as a function which maps the input bitvectors $a, b \in \mathbb{B}^n$ and the carry-in $c_{in} \in \mathbb{B}$ to a sum-bitvector $s \in \mathbb{B}^{n+1}$. Figure 2.2 shows schematic and PVS constructions of a carry-chain adder.




```
carry_chain(n: posnat, a, b: bvec[n], cin: bit):
RECURSIVE bvec[n+1] =
  IF n = 1 THEN
    fulladder(a(0), b(0), cin)
  Else
    LET
    chain = carry_chain(n-1, a^(n-2, 0), b^(n-2, 0), cin)
    IN
      fulladder(a(n-1), b(n-1), chain(n-1)) o
      chain^(n-2,0)
 ENDIF
 MEASURE n;

 cc_adder_correct: THEOREM
   FORALL(n: posnat, a,b: bvec[n], cin:bit):
   bv2nat(carry_chain(n, a, b, cin)) =
     bv2nat(a) + bv2nat(b) + bv2nat(cin)
```

In the definition of `carry_chain`, 'o' means bit-concatenation, and '^(·,·)' means sub-bitvector extraction. `bvnat(·)` reflects the binary value.

Figure 2.2: Construction and correctness statement of a carry-chain adder

Note that the function `carry_chain` has an additional parameter `n` used to parameterize the size of the adder. The lemma `cc_adder_correct` asserts the correctness of the adder for all widths n. The transcript of the PVS proof of this lemma can be found in Appendix B.

The correctness statements of the full adder and the carry-chain adder relate the hardware implementations to mathematical specifications of the form $\langle sum \rangle = \langle a \rangle + \langle b \rangle + \langle c_{in} \rangle$. Having precise specifications of the modules enables the composition of components to more and more complex hardware, and the rigorous mathematical reasoning about these compositions.

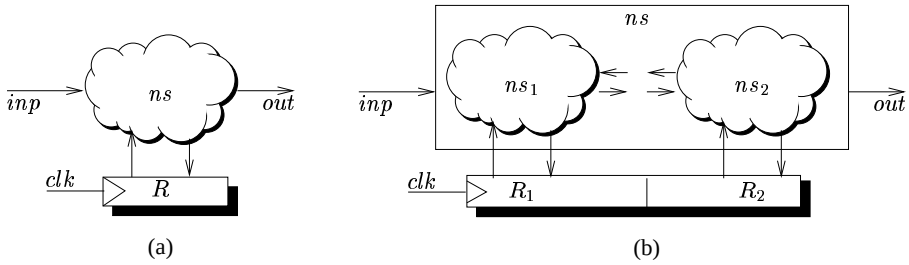


Figure 2.3: Modeling clocked circuits

2.3 Modeling Clocked Circuits

The subset of PVS which we use to model combinatorial hardware is similar to a functional programming language, thus offering no direct support for state-holding variables, as opposed to conventional programming or hardware description languages. Therefore, the concept of registers needs some extra consideration in PVS.

We may regard a clocked circuit as a circuit with only one state-holding register R (which may consist of many bits), and a *combinatorial* circuit ns (cf. Fig. 2.3a). The circuit ns takes as inputs the current state of register R and some external inputs, and computes some outputs and the next state of register R. The combinatorial circuit ns can be represented as a PVS function as described in the previous section:

$$\begin{array}{rcl} ns : State \times Input & \rightarrow & State \times Output \\ (current\_state, inp) & \mapsto & (next\_state, out) \end{array}$$

The $State$, $Input$, and $Output$ types may be arbitrarily nested records of bitvectors.

Multiple clocked circuits can be combined to one larger clocked circuit by interconnecting inputs and outputs, and using the cartesian product of the state types as new state type (cf. figure 2.3b). In this way, e.g., we embed the FPU into the processor. The result is one single combinatorial next-state function operating on the state of the processor and the state of the FPU.

2.4 Related Work

Formalizing combinatorial and clocked circuits in a functional programming language style, and verification of the hardware using a theorem prover is by no means new. For example, [HD85,KSK93,Mel93,SRC97] design, specify and verify hardware in PVS or other theorem provers in a very similar way.

Chapter 3

Theory of IEEE Rounding

This chapter 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 [IEEE] (mostly simply called "the standard" in this thesis), and notations and theorems facilitating the verification of the actual floating point hardware.

The theory presented in this chapter is primarily based on the work of Even and Paul [EP97] and Müller and Paul [MP00, Chap. 7]. The paper-and-pencil proofs in [MP00] 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 chapter is based on Miner's formalization of the IEEE standard in PVS [Min95].

In section 3.1, 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 3.2 with the definition of the rounding function and the proof of the *decomposition theorem of rounding*, which allows 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 chapter 4). In section 3.3 we define the floating point exceptions and exponent wrapping.

The concept of α-equivalence is defined in section 3.4. 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 3.5. 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. The rounder therefrom computes the correct floating point result and the exception signals. The decomposition simplifies the design and the verification of the FPU, since one can handle

the units separately and then compose them using the theorems on α-equivalence and rounding.

In section 3.6 we describe the encoding of floating point numbers in the bitvector representation defined in the standard. We also introduce the special values *infinity* and *Not-a-Number* (NaN). The supported floating point operations are described in section 3.7. We give a correctness predicate for the basic operations $(+, -, \times, \div)$ on non-special operands. We then define the result of comparisons between floating point numbers, and of conversions between different floating point formats and between floating point numbers and integers. Section 3.8 discusses related work.

Sections 3.1–3.5 of this chapter are a revised version of [Jac01], which has been presented as a poster at TPHOLs 2001.

3.1 Factorings

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

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

The following lemmas list some basic facts about factorings. We omit the proofs since they are fairly simple.

Lemma 3.1 *A factoring (s, e, f) has zero value, iff $f = 0$.*

Lemma 3.2 *Let (s, e, f) and (s', e', f') be factorings with $1 \leq f, f' < 2$. It holds*

$$e > e' \implies |[\![s, e, f]\!]| > |[\![s', e', f']\!]|$$

The property also holds for IEEE factorings.

Lemma 3.3 *Let (s, e, f) and (s', e', f') be IEEE factorings. It holds*

$$|[\![s, e, f]\!]| > |[\![s', e', f']\!]| \iff \left(e > e' \vee (e = e' \wedge f > f')\right).$$

The next lemma states that nonzero IEEE factorings are unique:

Lemma 3.4 *Let (s, e, f) and (s', e', f') be IEEE factorings with nonzero value. It holds*

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

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

3.1.2 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:

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

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

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

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

Lemma 3.5 *Let (s, e, f) be an arbitrary factoring. It holds:*[1]

(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:

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

where $sign(x) = 0$ if $x \geq 0$, and $sign(x) = 1$ otherwise.[2]

Lemma 3.6 *Let $x \in \mathbb{R}$. It holds:*

(i) $x = [\![\hat{\eta}(x)]\!]$ *if* $x \neq 0$,

[1] $\widehat{norm}_f(s, e, f)$ denotes the f-component of the triple $\widehat{norm}(s, e, f)$; analogous for other functions and components.

[2] 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 $+0$.

(ii) $x = [\![\eta(x)]\!]$

Lemma 3.7 *Let $x \in \mathbb{R}$ with $x \neq 0$ in the context of $\hat{\eta}$. It holds:*

(i) $\hat{\eta}_e(x) = \lfloor \log_2 |x| \rfloor$

(ii) $\hat{\eta}_f(x) = |x| \cdot 2^{-\hat{\eta}_e(x)}$

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

(iv) $\eta_f(x) = |x| \cdot 2^{-\eta_e(x)}$

The above lemmas all follow easily by expanding definitions and applying some basic arithmetic.

Lemma 3.8 *Let (s, e, f) be an arbitrary factoring with value $x := [\![s, e, f]\!]$, $x \neq 0$. It holds*

(i) $|x| \geq 2^{e_{\min}} \Longrightarrow \eta(x) = \hat{\eta}(x)$*, i.e., η and $\hat{\eta}$ coincide for normal numbers.*

(ii) If $1 \leq f < 2$, it holds $(s, e, f) = \hat{\eta}([\![s, e, f]\!])$.

(iii) If (s, e, f) is an IEEE factoring, it holds $(s, e, f) = \eta([\![s, e, f]\!])$.

(iv) $\hat{\eta}_e(x) \leq \eta_e(x)$

Proof: Statements *(i),(ii)* and *(iv)* are simple consequences of lemma 3.7. Statement *(iii)* is proved by using lemma 3.4 with $(s', e', f') = \eta([\![s, e, f]\!])$. □

Lemma 3.9 *Let $x \in \mathbb{R}$ and $(s, e, f) = \eta(x)$. It holds:*

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

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

Proof: It suffices to prove the first part, the second then follows directly, since $\eta(x)$ is either normal or denormal by definition of IEEE factorings. If $x = 0$, the claim holds trivially. If (s, e, f) is normal, it holds $e \geq e_{\min}$ and $f \geq 1$, hence $2^e \cdot f \geq 2^{e_{\min}}$. From lemma 3.6(*ii*) and the definition of $[\![\cdot]\!]$ we conclude $|x| \geq 2^{e_{\min}}$. Assume otherwise that $|x| \geq 2^{e_{\min}}$. From lemma 3.8(*i*) we know $(s, e, f) = \hat{\eta}(x)$. The claim now follows from lemmas 3.5(*ii*) and 3.7(*i*). □

3.1.3 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 (cf. lemmas 3.36 and 3.37). 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 $P > 1$. The standard defines an encoding of single and double precision IEEE factorings into bitvectors of length 32 and 64, respectively (cf. section 3.6). 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 following lemma bounds (semi-)representable numbers.

Lemma 3.10 *Let (s, e, f) be a semi-representable factoring, and $i > e$ be an integer. It holds*

(i) $f \leq 2 - 2^{1-P}$,

(ii) $|[\![s, e, f]\!]| \leq 2^i - 2^{i-P}$,

(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 semi-representable factorings:

Lemma 3.11 *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 i be an integer. It holds*

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

The following lemma states that semi-representability is not disturbed by multiplication with powers of 2:

Lemma 3.12 *Let (s, e, f) be a semi-representable factoring, and $n \in \mathbb{N}_0$. Then $\eta(2^n \cdot [\![s, e, f]\!])$ is a semi-representable factoring.*

3.2 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 [Min95]; it only differs in cases of overflow and underflow (Sect. 3.3).

3.2.1 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 simply 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:

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

It is not obvious that this definition conforms with the IEEE standard. In section 3.2.3 we prove a theorem to convince the reader of the conformance.

3.2.2 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 a 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 3.28.

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*:

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

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 3.13

(i) $\mathit{sigrd}(X, \mathcal{M}) = |rd([\![X]\!], \mathcal{M})| \cdot 2^{-e}$,

(ii) $0 \leq \mathit{sigrd}(X, \mathcal{M}) \leq 2$,

(iii) $1 \leq f \Longrightarrow 1 \leq \mathit{sigrd}(X, \mathcal{M})$,

(iv) $1 > f \implies 1 \geq \mathit{sigrd}(X, \mathcal{M})$,

(v) $\mathit{sigrd}(X, \mathcal{M}) \cdot 2^{P-1}$ *is an integer.*

Proof: Part *(i)* follows by expanding the definitions of *sigrd* and rd. For parts *(ii)–(iv)* one expands the definition down to r_{int} and applies basic properties of the floor and ceiling functions. Part *(v)* is a direct consequence of the definition of r_{rat}. □

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

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

Lemma 3.14 *The result* $\mathit{postnrom}(X, \mathcal{M})$ *of the post-normalization is a semi-representable IEEE factoring.*

Proof: The case $\mathit{sigrd}(X, \mathcal{M}) \in \{0, 2\}$ is trivial. Assume $0 < \mathit{sigrd}(X, \mathcal{M}) < 1$. By lemma 3.13(*iii*) we know $f < 1$, and hence $e = e_{\min}$ since X is an IEEE factoring. Therefore $\mathit{postnrom}(X, \mathcal{M})$ is an IEEE factoring, and with lemma 3.13(*v*) it is a semi-representable factoring.

Now assume $1 \leq \mathit{sigrd}(X, \mathcal{M}) < 2$. Since the input X is an IEEE factoring, we know $e \geq e_{\min}$, and hence $(s, e, \mathit{sigrd}(X, \mathcal{M})) = \mathit{postnrom}(X, \mathcal{M})$ is an IEEE factoring; semi-representability now follows from lemma 3.13(*v*). □

Lemma 3.15 $[\![\mathit{postnrom}(X, \mathcal{M})]\!] = rd([\![X]\!], \mathcal{M})$.

Proof: Apply lemma 3.13(*i*) and expand definitions. □

Theorem 3.16 (Decomposition Theorem) *For any real* x*, and rounding mode* $\mathcal{M} \in \{near, up, down, zero\}$*, it holds*

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

Proof: For nonzero rounding results, the claim follows from lemmas 3.8(*iii*) and 3.15. Otherwise, the claim follows by expanding the definitions of $norm, \eta$, and *postnrom*. □

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 [Gol96], but has been (paper-and-pencil) proved explicitly for the first time in [MP00]. 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 3.17 *Let $x \in \mathbb{R}$ and $(s, e, f) = \eta(x)$. It holds:*

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

Proof: Since (s, e, f) is denormal, it holds $e = e_{\min}, f < 1$. By lemma 3.13(*iv*) and the definition of post-normalization, it follows $\textit{postnrom}_e(\eta(x), \mathcal{M}) = e_{\min}$. The claim now follows by application of the decomposition theorem 3.16. □

3.2.3 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* [*digit*] *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 3.18 *Let $x, x' \in \mathbb{R}$ and x' be a semi-representable number.*

(i) For any rounding mode $\mathcal{M}$, $rd(x, \mathcal{M})$ 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.*

Proof: Part *(i)* is a trivial consequence of lemma 3.14 and theorem 3.16. Part *(ii)* and *(iii)* rely on the following fact proved by Miner in PVS [Min95]:

$$\begin{aligned} |x - r_{\text{int}}(x, near)| &\leq \tfrac{1}{2} \text{ and} \\ |x - r_{\text{int}}(x, near)| &= \tfrac{1}{2} \implies r_{\text{int}}(x, near) \text{ is even.} \end{aligned}$$

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

$$\begin{aligned} |x - rd(x, near)| &\leq 2^{e-P} \text{ and} \\ |x - rd(x, near)| &= 2^{e-P} \Longrightarrow \left(rd(x, near) \cdot 2^{-(1+e-P)}\right) \text{ is even.} \end{aligned} \tag{3.1}$$

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 3.11 (where we set $(s, e, f) = \eta(rd(x, near))$, $(s', e', f') = \eta(x')$, and $i = e$) results in

$$|rd(x, near) - x'| \geq 2^{e-(P-1)} = 2 \cdot 2^{e-P}. \tag{3.2}$$

Using the triangle inequality, (3.1) and (3.2) together yield

$$|x - x'| \geq 2^{e-P}. \tag{3.3}$$

Equations (3.1) and (3.3) 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) and (s', e', f') are IEEE factorings. Hence $|x| \geq 2^e$. Lemma 3.10(*ii*) with $i = e$ gives $|x'| \leq 2^e - 2^{e-P}$. Together this implies

$$|x' - x| \geq 2^{e-P}. \tag{3.4}$$

Again, (3.1) and (3.4) yield part *(ii)*. The proof of part *(iii)* is similar. □

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

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

Theorem 3.19 *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 3.18(*i*). Conversely, if x is semi-representable and $\mathcal{M} = near$, then the round result must equal x by theorem 3.18(*ii*) with $x' = x$. The claim for the remaining rounding modes follows analogously from their respective conformance theorems. □

3.3 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 *OVF*, *UNF*, and *INX* exceptions. The *INV* and *DIVZ* exceptions will be defined later.

The standard requires that each occurrence of an exception shall set a status flag and call a trap handler. The trap handler can be disabled on user request. We do

not describe the actual handling of the status flags and the trap handling, since this is part of the CPU instead of the FPU. However, since the detection of exceptions as well as the final result of floating point operations depend on whether the trap handlers are enabled or disabled, we need the enable flags for the overflow and underflow exceptions *OVFen* and *UNFen*, respectively. They are provided by the CPU.

3.3.1 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. (...)*

In lemma 3.10 we proved that $X_{\max} = 2^{e_{\max}} \cdot (2 - 2^{1-P})$ is the format's largest representable value. Since our rounding function by definition rounds as if the exponent range were unbounded above, we can define the *OVF* exception as follows:

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

Here, x is the exact result of a floating point operation. The *OVF* exception depends on the rounding mode, since different rounding modes round numbers slightly outside the representable range ($|x| = X_{\max} + \varepsilon$) differently to either $X_{\max}$, or to the next value outside the format's range.

Lemma 3.20 *It holds*

$$\mathit{OVF}(x, \mathcal{M}) \iff \eta_e(rd(x, \mathcal{M})) > e_{\max}.$$

Proof: The $\Rightarrow$ direction follows from lemma 3.10 and theorem 3.18(*i*), the $\Leftarrow$ direction from lemma 3.2. □

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

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

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

Lemma 3.21 *An overflow occurs, iff it occurs before or after rounding:*

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

Proof: By lemma 3.20 we have $\mathit{OVF}(x, \mathcal{M}) \iff \eta_e(rd(x, \mathcal{M})) > e_{\max}$. The claim now follows from the decomposition theorem 3.16 and the definition of *postnrom*. □

3.3.2 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 if 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

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

Here again, 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

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

Loss of accuracy only syntactically depends on the rounding mode, since this is a required parameter to the rd-function. From theorem 3.19 it easily follows $\mathit{LOSS}(x, \mathcal{M}_1) = \mathit{LOSS}(x, \mathcal{M}_2)$ for distinct rounding modes $\mathcal{M}_i$.

Lemma 3.22 *Let* $x \in \mathbb{R}$ *and* $(s, e, f) = \eta(x)$. *It holds*

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

Proof: By definition of *LOSS* and lemma 3.4 we have

$$\mathit{LOSS}(x, \mathcal{M}) \iff (\eta(rd(x, \mathcal{M}) \neq eta(x)).$$

The claim now follows from the decomposition theorem 3.16 and the definition of *postnrom*. □

Having defined tininess and loss of accuracy, we can define the underflow exception:

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

As mentioned above, the standard leaves other choices for the definition of *TINY* and *LOSS*. We refer the reader to [Har99, MP00] for lemmas about the relations between the different definitions.

3.3.3 Wrapped Exponent

In case of an overflow or underflow with corresponding trap enabled, the standard requests 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 $A = 3 \cdot 2^{N-2}$ with exponent width $N = 8$ and $N = 11$, respectively. 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. Again, x is the exact result of a floating point operation:

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

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

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

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 however requests a final result of either $\pm X_{\max}$ or $\pm\infty$, depending on the sign and the rounding mode. This will be specified as a case-split in section 3.7.1.

3.3.4 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 *result*$(x, \mathcal{M})$, which potentially has been wrapped. In Harrison's formalization of the IEEE standard [Har99] 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[3] on Intel's Pentium II with the operation $x := X_{\min}/2$ 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 [Har99], we define the inexact exception as

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

This is the definition also used in IBM's S/390 [IBM00, Pg. 19-22] and in [MP00], e.g. It has the advantage that programs can distinguish exact (except for exponent wrapping) from inexact computations in case of trapped overflows and underflows. For example, the above computation $x := X_{\min}/2$ can be represented exactly after having been multiplied with 2^A.

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

3.4 α-Equivalence

We now formalize the concept of α-equivalence[4] and α-representatives from [EP97, MP00]. This concept is a very concise way to speak about sticky-bit computations.

Let α be an integer. Two reals x and y are said to be *α-equivalent* ($x \equiv_\alpha y$), if $x = y$ or if there exists some $q \in \mathbb{Z}$ with $q \cdot 2^\alpha < x, y < (q+1) \cdot 2^\alpha$, i.e., if both x

[3] The test-program is available at our website:
`http://www-wjp.cs.uni-sb.de/~cj/PhD/`

[4] The term α-equivalence is not related to the term as used in λ-calculus (see e.g. [Bar90]).

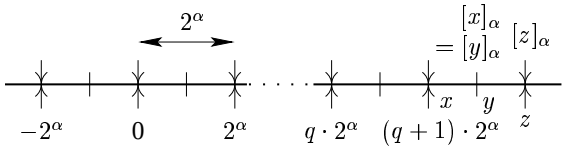


Figure 3.1: α-equivalence

and y lie in the same open interval between two consecutive integral multiples of 2^α (cf. figure 3.1). Clearly, if such a q exists, it must be $q_\alpha(x) := \lfloor x \cdot 2^{-\alpha} \rfloor$. The α-representative of x is defined as

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

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 3.23 *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)*

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

Lemma 3.23(*iii*) is used in the following to conclude the validity of statements on α-equivalent numbers x, y from the validity of the same statement on x and its representative $[x]_\alpha$. For example, we will prove in theorem 3.28 that x and $[x]_\alpha$ round to the same value for appropriate α. From this, one can conclude using lemma 3.23(*iii*) that α-equivalent x, y also round to the same number: it holds $[x]_\alpha = [y]_\alpha$ and hence $rd(x, \mathcal{M}) = rd([x]_\alpha, \mathcal{M}) = rd([y]_\alpha, \mathcal{M}) = rd(y, \mathcal{M})$. We will not explicitly reference any further usage of this proof idea.

Lemma 3.24 *Let $x, y \in \mathbb{R}, \alpha, k \in \mathbb{Z}$ such that $x \equiv_\alpha y$ and $k \geq \alpha$. It holds*

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

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

$$q \cdot 2^\alpha < x, y < (q+1) \cdot 2^\alpha. \tag{3.5}$$

It cannot hold $q < 2^{k-\alpha} < q+1$ 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 $2^{k-\alpha} \geq q+1$. First assume $q \geq 2^{k-\alpha}$, hence $q \cdot 2^\alpha \geq 2^k$. Equation (3.5) now implies $x > 2^k$ and $y > 2^k$. Assume otherwise $2^{k-\alpha} \geq q+1$, i.e., $2^k \geq (q+1) \cdot 2^\alpha$. Now (3.5) implies $2^k < x$ and $2^k < y$. □

The following theorem describes equivalence on factorings:

Lemma 3.25 *Let $x, x' \in \mathbb{R}$ nonzero, $e := \eta_e(x), e' := \eta_e(x'), \hat{e} := \hat{\eta}_e(x), \hat{e}' := \hat{\eta}_e(x')$, and α be an integer. It holds*

(i) $x \equiv_\alpha y \Longrightarrow sign(x) = sign(x')$,

(ii) $\alpha \leq \hat{e}$ *and* $x \equiv_\alpha x' \Longrightarrow \hat{e} = \hat{e}'$,

(iii) $\alpha \leq e$ *and* $x \equiv_\alpha x' \Longrightarrow e = e'$,

(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)*. Part *(i)* is easy, parts *(iii)–(v)* are similar to *(ii)*.

With lemma 3.23(*vii*) it suffices to proof the claim for $\alpha = \hat{e}$. By part *(i)* and lemma 3.23(*iv*) we may assume $x, x' \geq 0$.

Since the claim is trivial for $x = x'$, we further assume that $q_{\hat{e}}(x) \cdot 2^{\hat{e}} < x, x' < (q_{\hat{e}}(x)+1) \cdot 2^{\hat{e}}$ by definition of α-equivalence. From lemma 3.5(*ii*), we know $1 \leq x \cdot 2^{-\hat{e}} < 2$, and therefore $q_{\hat{e}}(x) = \left\lfloor x \cdot 2^{-\hat{e}} \right\rfloor = 1$. We then have $2^{\hat{e}} < x, x' < 2^{\hat{e}+1}$, and therefore $\hat{e} = \lfloor \log x \rfloor = \lfloor \log x' \rfloor$. Lemma 3.7 proves the claim. □

We now are ready to prove an important theorem, which allows the easy computation of IEEE factorings corresponding to representatives:

Theorem 3.26 *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:*

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

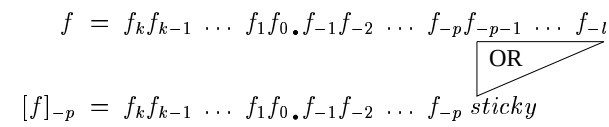


Figure 3.2: Computing representatives by sticky-computation

Proof: From lemma 3.25(*i*) and 3.25(*iii*) we have $\eta_s([x]_{e-p}) = s$ and $\eta_e([x]_{e-p}) = e$. From lemma 3.7 we know $\eta_f([x]_{e-p}) = |[x]_{e-p}| \cdot 2^{-e}$. With lemma 3.23(*iv*) and 3.23(*v*), we have $|[x]_{e-p}| \cdot 2^{-e} = [|x| \cdot 2^{-e}]_{-p}$. Lemma 3.7 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*. Let $f \geq 0$ be a real in binary format $f_k, \ldots, f_0, f_{-1} \ldots, f_{-l} \in \{\mathbf{0}, \mathbf{1}\}^{(k+1)+l}$ 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 $f_{-p-1}, \ldots, f_{-l}$ (cf. figure 3.2):

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

Theorem 3.27 *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 3.26 and 3.27 together allow 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 [Gol96], but the formalism with α-representatives allows for a very concise argumentation about these sticky computations. The verification of the adder circuitry in [BJ01, Ber01], e.g., relies heavily on the concept of α-equivalence.

3.5 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 [EP97, MP00]. There, the proofs are by geometrical arguments which are not suitable for formal verification.

Theorem 3.28 *Let $x \in \mathbb{R}$, $(s, e, f) := \eta(x)$, and $\mathcal{M}$ be a rounding mode. It holds*

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

Proof: It is technically very tedious to prove this theorem in PVS. We only give a sketch of the PVS proof. By theorems 3.16 and 3.26 it suffices to show

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

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

$$r_{\text{int}}\big((-1)^s\cdot f\cdot 2^{P-1},\mathcal{M}\big) \;=\; r_{\text{int}}\big((-1)^s\cdot [f]_{-P}\cdot 2^{P-1},\mathcal{M}\big). \tag{3.6}$$

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 $[f]_{-P}=(q+0.5)\cdot 2^{-P}$ with $q:=q_{-P}(f)=\lfloor f\cdot 2^P\rfloor$. Hence $[f]_{-P}=(\lfloor f\cdot 2^P\rfloor+0.5)\cdot 2^{-P}$ holds. Substituting this in (3.6) yields

$$\begin{aligned} r_{\text{int}}((-1)^s\cdot f\cdot 2^{P-1},\mathcal{M}) &= r_{\text{int}}\big((-1)^s\cdot\big((\lfloor f\cdot 2^P\rfloor+0.5)\cdot 2^{-1}\big),\mathcal{M}\big)\\ &= r_{\text{int}}\big((-1)^s\cdot\big(\tfrac14+\tfrac12\lfloor f\cdot 2^P\rfloor\big),\mathcal{M}\big). \end{aligned} \tag{3.7}$$

The theorem now follows from the next two lemmas. Lemma 3.29 proves that the claim is correct if $\mathcal{M}\neq near$. Lemma 3.30 proves that the same cases apply in the definition of $r_{\text{int}}(\,\cdot\,,near)$ on both sides of equation (3.7). Then the claim again follows by lemma 3.29. □

Lemma 3.29 *For all $z\in(\mathbb{R}^+\setminus\mathbb{N})$ and $s\in\{0,1\}$, it holds*

$$\lfloor(-1)^s\cdot z\rfloor=\big\lfloor(-1)^s\cdot\big(\tfrac14+\tfrac12\lfloor 2z\rfloor\big)\big\rfloor,$$

$$\lceil(-1)^s\cdot z\rceil=\big\lceil(-1)^s\cdot\big(\tfrac14+\tfrac12\lfloor 2z\rfloor\big)\big\rceil.$$

Lemma 3.30 *For all $z\in\mathbb{R}^+$, $2z\notin\mathbb{Z}$ and $s\in\{0,1\}$, set $z':=(-1)^s\cdot z$. It holds*

$$\begin{aligned} &\lceil z'\rceil-z'>z'-\lfloor z'\rfloor\iff\\ &\qquad\lceil z'\rceil-(-1)^s\cdot\big(\tfrac14+\tfrac12\lfloor 2z\rfloor\big)>(-1)^s\cdot\big(\tfrac14+\tfrac12\lfloor 2z\rfloor\big)-\lfloor z'\rfloor, \end{aligned}$$

$$\begin{aligned} &\lceil z'\rceil-z'<z'-\lfloor z'\rfloor\iff\\ &\qquad\lceil z'\rceil-(-1)^s\cdot\big(\tfrac14+\tfrac12\lfloor 2z\rfloor\big)<(-1)^s\cdot\big(\tfrac14+\tfrac12\lfloor 2z\rfloor\big)-\lfloor z'\rfloor. \end{aligned}$$

Lemma 3.29 can be proved by induction on $\lfloor z\rfloor$, and some basic properties of the floor and ceiling-functions from the PVS library. The proof, however, is technical and tedious. Lemma 3.30 is proved automatically by the PVS command `(grind)`.

Corollary 3.31 *Let $x\in\mathbb{R}$, $\alpha\le\eta_e(x)-P$, and $\mathcal{M}$ be a rounding mode. It holds*

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

In particular, the claim holds for $\alpha=\hat\eta_e(x)-P$.

Proof: The claim follows from theorem 3.28 and lemmas 3.23 and 3.8(*iv*). □

Corollary 3.32 *Let (s, e, f) be an IEEE factoring. It holds*

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

Proof: The claim follows from lemmas 3.13(*i*) and theorems 3.26 and 3.28. □

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

(i) $\mathit{OVF}(x, \mathcal{M}) \iff \mathit{OVF}([x]_{e-P}, \mathcal{M})$,

(ii) $\mathit{TINY}(x) \iff \mathit{TINY}([x]_{e-P})$,

(iii) $\mathit{LOSS}(x, \mathcal{M}) \iff \mathit{LOSS}([x]_{e-P}, \mathcal{M})$,

(iv) $\mathit{UNF}(x, \mathcal{M}, \mathit{UNFen}) \iff \mathit{UNF}([x]_{e-P}, \mathcal{M}, \mathit{UNFen})$,

Analogously to corollary 3.31, the claim holds for finer representatives.

Proof: Part *(i)* is an immediate consequence of theorem 3.28. Part *(ii)* follows from lemmas 3.25(*iv*) and 3.25(*v*). Part *(iii)* is slightly more complicated. We have to prove

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

By theorem 3.19, this is equivalent to

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

By theorem 3.26 and by definition of representability, this is equivalent to

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

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 have $[f]_{-P} = \left(q_{-P}(f) + \frac{1}{2}\right) \cdot 2^{-P}$. 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 $\mathit{result}(x, \mathcal{M})$ 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 3.34 *Let $x \in \mathbb{R}$, $\hat{e} := \hat{\eta}_e(x)$, and $\mathcal{M}$ be a rounding mode. It holds*

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

Proof: Theorem 3.33 shows that exponent wrapping occurs on x iff it occurs on $[x]_{\hat{e}-P}$. The claim follows trivially from corollary 3.31 if no wrapping occurs. Otherwise, assume first that an trapped underflow occurs, i.e., $\mathit{UNF}(x, \mathcal{M}, \mathit{UNFen}) \wedge \mathit{UNFen}$. We have to prove

$$rd(x \cdot 2^A, \mathcal{M}) \quad = \quad rd([x]_{\hat{e}-P} \cdot 2^A, \mathcal{M}). \tag{3.8}$$

By lemma 3.23(*v*) it holds $[x]_{\hat{e}-P} \cdot 2^A = [x \cdot 2^A]_{\hat{e}-P+A}$. Replacing this in (3.8) yields

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

This follows from corollary 3.31 if we prove

$$\hat{e} - P + A \quad = \quad \hat{\eta}_e(x \cdot 2^A) - P, \tag{3.9}$$

which follows from 3.7(*i*).

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

Note that in order to prove (3.9), the higher precision of $[x]_{\hat{e}-P}$ compared to $[x]_{e-P}$ is needed: (3.9) 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 significant which have been "shifted out" by denormalizing the significant. These digits, however, are present in the representative of the scaled significant, since by scaling these digits are "shifted back". Therefore, one needs the $(\hat{e} - P)$-representative, since this does not erase these "shifted out" digits.

We now are ready to prove that the *INX* exception can be computed on representatives, too. Analogously to theorem 3.34, we need the more precise $(\hat{e} - P)$-representative:

Theorem 3.35 *Let $x \in \mathbb{R}$, $\hat{e} := \hat{\eta}_e(x)$. It holds*

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

Proof: The proof is a combination of the proofs of 3.33(*iii*) and 3.34. We omit the details. □

Theorems 3.33–3.35 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 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.

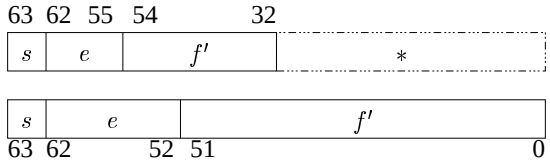


Figure 3.3: Embedding of (s, e, f') in one bitvector

3.6 IEEE Number Format

So far, we have only considered factorings which consist of numbers. In order to implement floating point units in hardware, a definition of a representation of numbers using bits and bitvectors is needed. The IEEE standard defines these representations for floating point numbers. The definition in the standard also features the special values *infinity* (∞) and *not-a-number* (NaN) used to represent results of certain operations, e.g., division by zero. In this section we describe how we formalize the number format definitions from the standard in PVS.

For the following, we remind the reader of the notations and definitions of bitvectors and the numbers represented by them from section 2.1.

An IEEE floating point format is defined by a pair of parameters (N, P) analogously to the parameters used in the previous sections on factorings. An IEEE number consists of a sign bit $s \in \mathbb{B}$, an N-bit exponent $e \in \mathbb{B}^N$, and a $(P-1)$-bit "almost-significant" $f' \in \mathbb{B}^{P-1}$. The actual P-bit significant f is defined as

$$f = \begin{cases} \mathbf{0} \circ f' & \text{if } e = \mathbf{0}^N, \\ \mathbf{1} \circ f' & \text{otherwise.} \end{cases}$$

The additional bit $f[P-1]$ is called *hidden bit*.

The most important floating point formats are single precision $(N, P) = (8, 24)$ and double precision $(N, P) = (11, 53)$. The components s, e, f' are embedded into a 64-bit bitvector w according to figure 3.3. We call w the *IEEE bitvector* of (s, e, f') and identify w and (s, e, f').

In the IEEE standard, single precision floating point numbers are embedded into 32-bit bitvectors. The above embedding equals the embedding in the standard, except that we append 32 non-specified bits.

IEEE Numbers. If the exponent e satisfies $e \neq \mathbf{1}^N$, the IEEE bitvector corresponds to a factoring

$$\textit{bv2fact}(w) := \begin{cases} (s, e_{\min}, \langle f \rangle \cdot 2^{-(P-1)}) & \text{if } e = \mathbf{0}^N \\ (s, [e]_{\text{bias}}, \langle f \rangle \cdot 2^{-(P-1)}) & \text{otherwise,} \end{cases}$$

where $e_{\min} = -2^{N-1} + 2$ as in the previous sections[5]. Note the subtle difference between the bitvectors (s, e, f) and the corresponding factoring *bv2fact*(w)

consisting of *numbers* instead of bitvectors. According to the definitions in section 3.1 we call the bitvector w normal or denormal, if the corresponding factoring *bv2fact*(w) is normal or denormal, respectively.

Lemma 3.36 *Let $w = (s, e, f')$ be a (single or double) IEEE bitvector with $e \neq \mathbf{1}^N$. Then bv2fact(w) is a representable IEEE factoring (with respect to the appropriate parameters N and P).*

Proof: Let $(\mathsf{s}, \mathsf{e}, \mathsf{f}) =$ *bv2fact*(w) be the factoring represented by w. Assume $e \notin \{\mathbf{0}^N, \mathbf{1}^N\}$. It holds $\mathsf{e} = [e]_{\text{bias}} = \langle e \rangle - \textit{bias}_N \in \{1 - \textit{bias}_N, \ldots, 2^n - 2 - \textit{bias}_N\}$. From the definitions of $e_{\min}, e_{\max}, \textit{bias}_N$ it follows $\mathsf{e} =\in \{e_{\min}, \ldots, e_{\max}\}$.

It holds $0 \leq \mathsf{f} < 2$, and $\mathsf{f} < 1$ only if $e = \mathbf{0}^N$. If $e = \mathbf{0}^N$ it holds $\mathsf{e} = e_{\min}$. Hence, $(\mathsf{s}, \mathsf{e}, \mathsf{f})$ is an IEEE factoring, and $\mathsf{e} \leq e_{\max}$. From the definition of *bv2fact* follows that $2^{P-1} \cdot \mathsf{f} \in \mathbb{N}$; hence $(\mathsf{s}, \mathsf{e}, \mathsf{f})$ is a representable IEEE factoring. □

Lemma 3.37 *Every representable factoring $(\mathsf{s}, \mathsf{e}, \mathsf{f})$ has an IEEE bitvector representation.*

Proof: It is easy to construct an IEEE bitvector w with $[\![\textit{bv2fact}(w)]\!] = [\![\mathsf{s}, \mathsf{e}, \mathsf{f}]\!]$. □

We extend the value operator $[\![\cdot]\!]$ to IEEE bitvectors w satisfying $e \neq \mathbf{1}^N$:

$$[\![w]\!] := [\![\textit{bv2fact}(w)]\!].$$

Infinity. IEEE bitvectors $w = (s, e, f')$ with $e = \mathbf{1}^N$ have the special meaning *infinity* or *Not-a-Number* (NaN). If $e = \mathbf{1}^N$ and $f' = \mathbf{0}^{P-1}$, then w represents infinity; depending on the sign bit s, w is either plus infinity ($+\infty$) or minus infinity ($-\infty$). The central statement about infinity in the IEEE standard is

> *Infinities shall be interpreted in the affine sense, that is, $-\infty <$ (every finite number) $< +\infty$.*

This defines the result of most operations involving infinite operands.

NaN. If $e = \mathbf{1}^N$ and $f' \neq \mathbf{0}^{P-1}$, then w represents *Not-a-Number* (NaN). There are two kinds of NaNs: *signaling* NaNs where $f'[P-2] = \mathbf{0}$, and *quiet* NaNs where $f'[P-2] = \mathbf{1}$. Operations involving signaling NaN operands shall signal the invalid-exception *INV*. Operations involving NaN operands shall return one of the input NaNs as output. Note that this is not possible in all operations due to different formats of the operands and results (e.g., in conversions).

For later use, we introduce predicates *number*(w), *inf*(w), *inf*$_+(w)$, *inf*$_-(w)$, *nan*(w), *nan*$_s(w)$ and *nan*$_q(w)$ in order to distinguish IEEE bitvectors w representing numbers, infinity, plus infinity, ..., respectively.

[5] Actually, $e_{\min}$ and $e_{\max}$ are explicitly defined only for single and double precision in the standard. For other floating point formats, the standard leaves the choice of $e_{\min}$ and $e_{\max}$ (within some bounds) to the implementor. We uniformly choose $e_{\min} = -2^{N-1} + 2$ and $e_{\max} = 2^{N-1} - 1$ as defined in section 3.1.

3.7 Floating Point Operations

In this section, we define the result of the supported floating point operations. The operations are addition, subtraction, multiplication, division, comparison, and conversions. We assume (N, P) to be either $(8, 24)$ for single or $(11, 53)$ for double precision, respectively.

3.7.1 Basic Operations

We start by defining the result of an operation $a \circ b$ where $\circ \in \{+, -, \times, \div\}$. We first assume a and b to be IEEE numbers, i.e., no special operands, and $b \neq 0$ in case of divisions. Let $\mathcal{M}, \mathit{OVFen}, \mathit{UNFen}$ be the current rounding mode and exception masks for overflow and underflow, respectively.

Let $x := a \circ b$ be the exact result of the operation, and w be the output of the floating point unit. Let $\mathit{ovf}, \mathit{unf}, \mathit{inx}$ be the exception signals computed by the FPU. We define the predicate *FPU-result-correct* stating the correctness of the FPU result:

$\mathit{FPU\text{-}result\text{-}correct}(x, \mathcal{M}, \mathit{OVFen}, \mathit{UNFen})(w, \mathit{ovf}, \mathit{unf}, \mathit{inx}) :=$

1. If no untrapped overflow occurs, then the bitvector w represents a number, and the value of w is the (possibly wrapped) rounded result as defined in section 3.3 (pg. 20).

$$\neg(\mathit{OVF}(x, \mathcal{M}) \wedge \overline{\mathit{OVFen}}) \implies \mathit{number}(w) \wedge [\![w]\!] = result(x, \mathcal{M}, \mathit{OVFen}, \mathit{UNFen}).$$

   Note that this definition does *not* define the sign bit if the rounded result is 0. The sign of 0 is handled as special case in the sections on the hardware. Furthermore, note that the definition implicitly requires that the (wrapped) rounded result lies in the range of representable numbers. That this is true for the basic operations will be proved in later sections.

2. If an untrapped overflow occurs, the FPU shall return either $\pm\infty$ or $\pm X_{\max}$, depending on the rounding mode and the sign of the exact result. The IEEE standard defines this explicitly:

   > *The result, when no trap occurs, shall be determined by the rounding mode and the sign of the intermediate [exact] result as follows:*
   > *(1) Round to nearest carries all overflows to ∞ with the sign of the intermediate result.*
   > *(2) Round toward 0 carries all overflows to the format's largest finite number with the sign of the intermediate result.*
   > *(3) Round toward $-\infty$ carries positive overflows to the format's largest finite number, and carries negative overflows to $-\infty$.*

(4) Round toward $+\infty$ carries negative overflows to the format's most negative finite number, and carries positive overflows to $+\infty$.

The formalization is as follows:

$$
\begin{array}{l}
(\mathit{OVF}(x, \mathcal{M}) \wedge \overline{\mathit{OVFen}}) \implies \\
\quad \text{IF } \mathcal{M} = near \vee (\mathcal{M} = up \wedge x \geq 0) \vee (\mathcal{M} = down \wedge x \leq 0) \text{ THEN} \\
\quad\quad \text{IF } x \geq 0 \text{ THEN } \mathit{inf}_+(w) \text{ ELSE } \mathit{inf}_-(w) \text{ ENDIF} \\
\quad \text{ELSE} \\
\quad\quad \mathit{number}(w) \wedge \\
\quad\quad [\![\mathit{bv2fact}(w)]\!] = \text{IF } x \geq 0 \text{ THEN } X_{\max} \text{ ELSE } -X_{\max} \text{ ENDIF} \\
\quad \text{ENDIF},
\end{array}
$$

3. The overflow, underflow, and inexact exceptions are computed according to their specification in section 3.3:

$$
\begin{aligned}
\mathit{ovf} &= \mathit{OVF}(x, \mathcal{M}), \\
\mathit{unf} &= \mathit{UNF}(x, \mathcal{M}, \mathit{UNFen}), \\
\mathit{inx} &= \mathit{INX}(x, \mathcal{M}, \mathit{OVFen}, \mathit{UNFen}).
\end{aligned}
$$

The correctness of the division by zero- and invalid-exceptions are handled separately below.

The following theorem combines theorems 3.33–3.35:

Theorem 3.38 *Let $x, x' \in \mathbb{R}$, $\hat{e} = \hat{\eta}_e(x)$, and $x \equiv_{\hat{e}-P} x'$. It holds*

$$
\begin{array}{l}
\mathit{FPU\text{-}result\text{-}correct}(x, \mathcal{M}, \mathit{OVFen}, \mathit{UNFen})(w, \mathit{ovf}, \mathit{unf}, \mathit{inx}) \iff \\
\quad \mathit{FPU\text{-}result\text{-}correct}(x', \mathcal{M}, \mathit{OVFen}, \mathit{UNFen})(w, \mathit{ovf}, \mathit{unf}, \mathit{inx})
\end{array}
$$

This theorem will be used to combine the computation units (e.g., adder) with the rounder in the next chapter. The computation unit provides a result x' which is α-equivalent to the exact result x, but has a shorter bitvector-representation. The rounder then computes the result w and the exception bits from the intermediate x' result.

The above definition of *FPU-result-correct* covers all possible inputs to the FPU except for

- the result of floating point operations on special operands ($\pm\infty$,NaNs),
- comparison and conversion results,
- and the *DIVZ* and *INV* exceptions.

The result of operations on special operands is explicitly defined in the standard. We give examples on the transliterations of these definitions to PVS theorems in the chapter on the verification of the actual hardware, but do not give the full details in this thesis, since the details are long and tedious.

The division-by-zero exception is signaled on divisions $a/0$ where $a \neq 0$. That the hardware implementation fulfills this requirement is proved in the verification of the actual hardware.

The invalid exception is signaled on any operation involving signaling NaNs as operands, on additions (and subtractions) on infinities with opposing (same) sign, on $0 \times \pm\infty$, on $0/0$ and $\pm\infty/\pm\infty$, and on some comparisons and conversions as specified below.

3.7.2 Comparison

The IEEE standard defines four relations for the comparison of floating point numbers: *less than, equal, greater than,* and *unordered.* Two floating point numbers are *unordered,* if at least one of them is a NaN. The three other relations have their obvious meaning for non-special values. For special values, it holds $-\infty < \infty$, $-\infty \not< -\infty$, $-\infty = -\infty$, and so on. Let w_1, w_2 be IEEE bitvectors. The four relations are formalized as follows:

$$
\begin{aligned}
\mathit{unordered}(w_1, w_2) &:= \mathit{nan}(w_1) \vee \mathit{nan}(w_2),\\
\mathit{less}(w_1, w_2) &:= \neg\mathit{unordered}(w_1, w_2) \wedge\\
&\quad \big((\neg\mathit{inf}_+(w_1) \wedge \mathit{inf}_+(w_2)) \vee\\
&\quad\ (\mathit{inf}_-(w_1) \wedge \neg\mathit{inf}_-(w_2)) \vee\\
&\quad\ (\mathit{number}(w_1) \wedge \mathit{number}(w_2) \wedge [\![w_1]\!] < [\![w2]\!])\big),\\
\mathit{greater}(w_1, w_2) &:= \neg\mathit{unordered}(w_1, w_2) \wedge\\
&\quad \big((\mathit{inf}_+(w_1) \wedge \neg\mathit{inf}_+(w_2)) \vee\\
&\quad\ (\neg\mathit{inf}_-(w_1) \wedge \mathit{inf}_-(w_2)) \vee\\
&\quad\ (\mathit{number}(w_1) \wedge \mathit{number}(w_2) \wedge [\![w_1]\!] > [\![w2]\!])\big),\\
\mathit{equal}(w_1, w_2) &:= \neg\mathit{unordered}(w_1, w_2) \wedge\\
&\quad \big((\mathit{inf}_-(w_1) \wedge \mathit{inf}_-(w_2)) \vee\\
&\quad\ (\mathit{inf}_+(w_1) \wedge \mathit{inf}_+(w_2)) \vee\\
&\quad\ (\mathit{number}(w_1) \wedge \mathit{number}(w_2) \wedge [\![w_1]\!] = [\![w2]\!])\big).
\end{aligned}
$$

The above definitions ignore the sign of zeros, as it is explicitly demanded for comparisons in the standard.

The actual comparison operation is controlled by four bits *FCONun, FCONlt, FCONgt,* and *FCONeq.* Each bit names the relation which shall be tested on the

operands. Thus, the result *fcc* of the comparison is defined as

$$\begin{aligned} \mathit{fcc} :=& (\mathit{unordered}(w_1, w_2) \wedge \mathit{FCONun}) \vee (\mathit{less}(w_1, w_2) \wedge \mathit{FCONlt}) \vee \\ & (\mathit{greater}(w_1, w_2) \wedge \mathit{FCONgt}) \vee (\mathit{equal}(w_1, w_2) \wedge \mathit{FCONeq}). \end{aligned}$$

Additionally to the comparison result, the FPU shall signal an invalid operation if

> *unordered operands are compared using one of the predicates involving "<" or ">" but not* unordered. [IEEE]

This condition is made formal in the predicate *FCON-sig-unordered*:

$$\begin{aligned} \mathit{FCON\text{-}sig\text{-}unordered}(w_1, w_2) \quad := \quad & (\mathit{FCONgt} \vee \mathit{FCONlt}) \wedge \neg \mathit{FCONun} \\ & \wedge \mathit{unordered}(w_1, w_2). \end{aligned}$$

The following lemmas show how to implement the comparison operation in hardware.

Lemma 3.39 *For all IEEE bitvectors w_1 and w_2, exactly one of the predicates unordered(w_1, w_2), less(w_1, w_2), greater(w_1, w_2) and equal(w_1, w_2) holds.*

Proof: This lemma is proved automatically using `(grind)`. □

Lemma 3.40 *Let w, w' be non-special IEEE bitvectors, and let $(s, e, f), (s', e', f')$ be the corresponding factorings. It holds*

$$\begin{aligned} \mathit{less}(w_1, w_2) \quad \Longleftrightarrow \quad & \neg([\![s, e, f]\!] = 0 \wedge [\![s', e', f']\!] = 0) \wedge \\ & \big((s = 1 \wedge s' = 0) \vee \\ & \quad (s = 0 \wedge s' = 0 \wedge (e < e' \vee (e = e' \wedge f < f'))) \vee \\ & \quad (s = 1 \wedge s' = 1 \wedge (e > e' \vee (e = e' \wedge f > f')))\big) \\ \mathit{equal}(w_1, w_2) \quad \Longleftrightarrow \quad & ([\![s, e, f]\!] = 0 \wedge [\![s', e', f']\!] = 0) \vee (s, e, f) = (s', e', f') \end{aligned}$$

Proof: The lemma is proved by case-splitting on the sign-bits, and applying lemma 3.3. □

3.7.3 Conversion

The IEEE standard demands that instructions for the conversions 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 3.2. All four rounding modes must be supported.

In case of conversion from single precision floating point numbers or integers to double precision, rounding does not affect the value, i.e., it is always exact; however, the rounding algorithm normalizes the number and thus yields an IEEE

factoring. This is necessary, since denormal single precision numbers have a normal double precision representation due to the larger exponent range in double precision. In conversion to a floating point format, the conversion unit signals exceptions as specified in section 3.3. We therefore may use the *FPU-result-correct–* predicate as defined in section 3.7.1 in order to define correct conversion from any format to floating point formats. Conversions of infinity or NaN between floating point formats shall preserve infinity or NaN, respectively. Note that the conversion of a NaN cannot return the same NaN since the widths of the IEEE bitvectors do not match.

Conversion to Integers. The rounding function and algorithm described in section 3.2 was designed to return IEEE factorings where the significand has $P-1$ fractional digits. For the conversion to integer format, we therefore need to adjust the rounding algorithm to return integers. We start by defining the result of rounding reals x to integers:

$$\textit{rd2int}(x, \mathcal{M}) := \eta(r_{\text{int}}(x, \mathcal{M})),$$

where r_{int} was defined in section 3.2.1. Note that the function *rd2int* returns a factoring instead of an integer. This has the advantage of being compatible with the rest of the theory, and later allows the use of the standard rounding unit for the rounding to integers.

The correctness of the *rd2int* function is proved similarly to the correctness of the overall rounding function in section 3.2.3. For the to-nearest rounding mode, e.g., the correctness statement is:

Theorem 3.41 *Let $x \in \mathbb{R}$, i be an arbitrary integer, and $r := [\![\textit{rd2int}(x, near)]\!]$ be the rounding result. It holds:*

(i) *r is an integer: $r \in \mathbb{Z}$,*

(ii) *r is a nearest integer: $|x - i| \geq |x - r|$,*

(iii) *In case of a tie, r is even: $|x - r| = \frac{1}{2} \implies even(r)$.*

We now partition the range of semi-representable factorings into large ones (with exponent $e \geq P-1$), small ones ($e < 0$), and the rest. The following two lemmas show that rounding to integer is easy if the operand is either large or small.

Lemma 3.42 *Let (s, e, f) be a semi-representable IEEE factoring with $e \geq P-1$, and let $x = [\![s, e, f]\!]$ be its value. It holds:*

$$\textit{rd2int}(x, \mathcal{M}) = (s, e, f).$$

Proof: By definition of semi-representability, we know that $f \cdot 2^{P-1}$ is an integer. Since $e \geq P-1$, it follows that $x = [\![s, e, f]\!]$ is an integer. Hence, $r_{\text{int}}(x) = x$ by definition, which proves the lemma. □

Lemma 3.43 *Let (s, e, f) be an IEEE factoring with $e < 0$, and $x = [\![s, e, f]\!]$. If $f = 0$, then rd2int$(x, \mathcal{M}) = \eta(0) = (0, e_{\min}, 0)$. If $f \neq 0$, then*

$$\textit{rd2int}(x, \mathcal{M}) = \begin{cases} (0, e_{\min}, 0) & \textit{if } \mathcal{M} = zero, \\ (0, e_{\min}, 0) & \textit{if } \mathcal{M} = pos \wedge s = 1, \\ (0, 0, 1) & \textit{if } \mathcal{M} = pos \wedge s = 0, \\ (1, 0, 1) & \textit{if } \mathcal{M} = neg \wedge s = 1, \\ (0, e_{\min}, 0) & \textit{if } \mathcal{M} = neg \wedge s = 0, \\ (0, e_{\min}, 0) & \textit{if } \mathcal{M} = near \wedge e < -1, \\ (0, e_{\min}, 0) & \textit{if } \mathcal{M} = near \wedge e = -1 \wedge f = 1, \\ (s, 0, 1) & \textit{if } \mathcal{M} = near \wedge e = -1 \wedge f \neq 1. \end{cases}$$

Proof: It holds $(0, e_{\min}, 0) = \eta(0), (0, 0, 1) = \eta(1), (1, 0, 1) = \eta(-1)$. The claim follows by case-splitting, expanding definitions, and applying properties of floor- and ceiling-functions. □

Using lemmas 3.42 and 3.43, it is easy to implement the conversion to integer for large and small floating-point numbers.

The next theorem allows the conversion for the mid-range numbers using the standard rounding function rd. One first scales x by multiplication with $2^{e_{\min}+1-P}$. Intuitively, this scaling denormalizes x and thereby moves the binary digit of weight 1 into the least significand representable position. This denormalized x is then rounded using the standard rounding function rd. The significant resulting from the rounding hence carries the *rd2int*(x) result in its least significand digits.

Theorem 3.44 *Let (s, e, f) be an IEEE factoring with $0 \leq e < P - 1$, let $x = [\![s, e, f]\!]$, and $(s_r, e_r, f_r) = \eta(rd(x \cdot 2^{e_{\min}+1-P}, \mathcal{M}))$ be the result of first scaling x and then rounding. It holds:*

$$[\![\textit{rd2int}(x, \mathcal{M})]\!] = (-1)^{s_r} \cdot f_r \cdot 2^{P-1}. \tag{3.10}$$

Note that the exponent e_r is not part of the right-hand side of the equation. That means that the rounded integer value of x is obtained by interpreting the rounded significand f_r as a natural number, and taking the negative of this number if $s_r = 1$.

In order to prove theorem 3.44, we first prove the following two lemmas:

Lemma 3.45 *Let (s, e, f) be an IEEE factoring with $0 \leq e < P - 1$ and $x = [\![s, e, f]\!]$. The factoring $\eta(x \cdot 2^{e_{\min}+1-P})$ is denormal.*

Proof: First notice that (s, e, f) is normal since $e_{\min} < 0 \leq e$; hence $1 \leq f < 2$ holds. From lemma 3.9(*ii*) it suffices to prove $|x \cdot 2^{e_{\min}+1-P}| < 2^{e_{\min}}$. It holds $\lfloor \log_2 |x \cdot 2^{e_{\min}+1-P}| \rfloor = \lfloor \log_2 (2^e \cdot f \cdot 2^{e_{\min}+1-P}) \rfloor = e + e_{\min} + 1 - P + \lfloor \log_2 f \rfloor = e - (P-1) + e_{\min} < e_{\min}$, since $1 \leq f < 2$ and $e < P - 1$. □

Lemma 3.46 *Let (s, e, f) be an IEEE factoring with $0 \leq e < P - 1$ and $x = [\![s, e, f]\!]$. It holds:*

$$[\![\textit{rd2int}(x, \mathcal{M})]\!] = 2^{P-1-e_{\min}} \cdot rd(x \cdot 2^{e_{\min}-P-1}, \mathcal{M}).$$

Proof: Expanding the definitions of *rd2int*, rd, and r_{rat} and applying lemma 3.45 yields the claim. □

Proof of Theorem 3.44: We have to prove (3.10). By lemma 3.46 this is equivalent to

$$2^{P-1-e_{\min}} \cdot rd(x \cdot 2^{e_{\min}-P-1}, \mathcal{M}) \quad = \quad (-1)^{s_r} \cdot f_r \cdot 2^{P-1}. \tag{3.11}$$

From lemmas 3.45 and 3.17 we know $e_r = e_{\min}$. Rewriting (3.11) with lemma 3.7(*iv*) yields

$$2^{P-1-e_{\min}} \cdot rd(x \cdot 2^{e_{\min}-P-1}, \mathcal{M}) \quad = \quad (-1)^{s_r} \cdot |rd(x \cdot 2^{e_{\min}-P-1}, \mathcal{M})|/2^{e_{\min}} \cdot 2^{P-1},$$

which follows by case-splitting on the sign s_r. □

3.8 Related Work

As mentioned before, the central concepts in this chapter are taken from [EP97, MP00]. The paper-and-pencil proofs in [EP97, MP00] served as guidelines in our formal verification.

Barrett [Bar89] has formalized parts of the IEEE standard in the specification language Z. However, his work does not include any verified theorems, but only the translation of the standard to Z.

Miner [Min95] has formalized the IEEE standard 854 in PVS. The IEEE standard 854 is an extension of the standard 754 with which we deal in this thesis. The main extension is that 754 only covers binary representations, whereas 854 covers arbitrary bases. Miner has proved some simple lemmas in his work. Our definition of the rounding function and the proof of its correctness is based on Miner's work. Miner's formalization does not comprise theorems related to α-equivalence and round decomposition.

Another formalization of the IEEE standard was given by Harrison [Har97, Har99] in the theorem prover HOL Light. Harrison does not discuss exponent wrapping, which introduces some ambiguities in the definition of the inexact exception (cf. section 3.3). Harrison's formalization has no counterpart to round decomposition. He has theorems related to the computation of exceptions of α-equivalent numbers [Har99, Sect. 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.

In [MLK98], Moore et al. verify the AMD K5 floating point division algorithm. They have a definition of sticky bit computations that is similar to our α-equivalence. They do not cover exceptions and round decomposition.

In [Rus98, Rus99, Rus00], Russinoff proves the correctness of some components of AMD floating point units against a formal specification. His formalization of the rounding function and sticky bit computations is similar to [MLK98]. Russinoff does not cover denormals, exceptions, and round decomposition; however, he states that he handles denormals in unpublished work (private communication).

Chapter 4

Verification of the Floating Point Hardware

In this chapter, we describe the design and verification of the floating point hardware with respect to the specification given in the previous chapter. 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 build as depicted in figure 4.1: 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 and outputs the result as an IEEE bitvector. Special cases such as operations on special operands (∞, NaN), or divisions

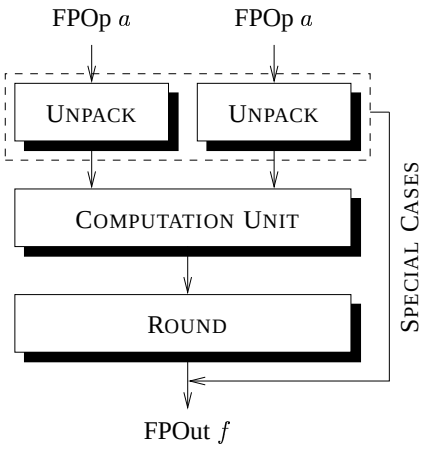


Figure 4.1: Top-level view of the floating point units.

by zero bypass the computation and rounding unit.

Most of the designs are taken virtually unchanged from [MP00]. We therefore mostly omit the construction of the circuits, and describe only their interface and precise correctness statement. We describe the detailed design only where our design differs significantly from [MP00], or where the exact design is needed to follow the correctness arguments. We give detailed proofs of the statements only if they are wrong or incomplete in [MP00].

The design and formal verification of the circuits in this chapter makes heavy use of our library of verified general-purpose circuits such as adders, shifters, decoders, etc. [BJK01a].

We have reported on the formal verification of the *VAMP* floating point hardware previously in [BJ01].

This chapter is structured as follows: we describe the different hardware components in sections 4.1–4.5. Section 4.6 discusses some minor discrepancies of our FPUs to the IEEE standard. Section 4.7 discusses related work.

4.1 Unpacker

In this section, we describe the unpacker circuits. There are two kinds of unpackers:

- The floating point unpacker takes as input a floating point number in IEEE format and some control variables, and returns the floating point number in a more convenient format: the exponent format is changed from biased integer to two's complement format, and the hidden significant bit is revealed. For multiplication and division, the unpacker normalize denormal operands. Furthermore, the unpacker outputs some auxiliary information about the operand, e.g., whether the operand is zero, ∞ or NaN.
- The fixed-point unpacker takes as argument a 32-bit two's complement integer, and returns the bitvector representation of a factoring with the same value as the integer.

The output format of the unpackers is the same for single and double precision, since both precisions are processed (nearly) the same in the computation units.

4.1.1 Floating Point Unpacker

Circuit 4.1 (FP-UNPACK) The floating point unpacker is a circuit with inputs

- $F \in \mathbb{B}^{64}$: the floating point operand in IEEE format.
- $dbl \in \mathbb{B}$: if set, F represents a double precision number, otherwise a single precision number.
- $normal \in \mathbb{B}$: if set, the unpacker normalizes denormal inputs. This is needed for multiplications and divisions.

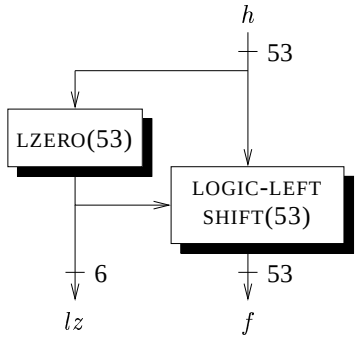


Figure 4.2: Normalization shift in the unpacker

The outputs of the unpacker are

- $s \in \mathbb{B}, e \in \mathbb{B}^{11}, f \in \mathbb{B}^{1+52}$: the unpacked sign, exponent, and significand. The exponent is in two's complement format, the significand represents a fraction with 52 bits behind the point.
- $lz \in \mathbb{B}^{6}$: the number of leading zeros of the significand before normalization. This is unspecified if $normal = \mathbf{0}$.
- *einf* $\in \mathbb{B}$: active if the exponent equals $\mathbf{1}^N$, i.e., if F is a special value.
- *ZERO, INF, pINF, nINF, QNAN, SNAN* $\in \mathbb{B}$: active if F is zero, $\pm\infty$, $+\infty$, $-\infty$, a quiet or signaling NaN, respectively.

The construction of the floating point unpacker closely follows [MP00, pg. 354ff]. We therefore omit the details of the construction. ◇

The following three lemmas are the correctness statements of the floating point unpacker:

Lemma 4.1 *The output bits ZERO,INF,pINF,... are set iff the input F represents zero, infinity, plus infinity, ..., respectively.*

Lemma 4.2 *Let $normal = \mathbf{0}$. If einf $= \mathbf{0}$, then F represents a number, and the corresponding factoring is represented by the output components s, e, f:*

$$\mathit{bv2fact}(F) = (s, [e], \langle f \rangle \cdot 2^{-52})$$

Lemma 4.3 *Let $normal = \mathbf{1}$. If einf $= \mathbf{0}$ and ZERO $= \mathbf{0}$, then F represents a nonzero number, and it holds*

$$\widehat{\mathit{norm}}(\mathit{bv2fact}(F)) = (s, [e] - \langle lz \rangle, \langle f \rangle \cdot 2^{-52}).$$

Proof: All three lemmas follow easily from the construction of the unpacker. There is only one non-trivial part in the proof missing in [MP00]: with $normal = \mathbf{1}$, the output of the unpacker is in fact the normalized operand. Let $h \in \mathbb{B}^{1+52}$ be the bitvector representation of the un-normalized significand, i.e.,

$$\langle h \rangle \cdot 2^{-52} = \mathit{bv2fact}_f(F). \tag{4.1}$$

Figure 4.2 shows the part of the unpacker which performs the normalization. The leading-zero counter counts the number of leading zeros of h, and a logical-left shifter shifts the leading zeros out. It may seem obvious that this yields the normalized factoring, but it is not trivial to prove in PVS.

From the correctness of the leading-zero counter [BJK01a] we know that

$$\langle lz \rangle = lzero(h), \tag{4.2}$$

where $lzero$ is the function counting leading zeros. The following equations (4.3) and (4.4) are lemmas on the $lzero$ function from the library [BJK01a]:

$$\forall n \in \mathbb{N}, b \in \mathbb{B}^n : lzero(b) = n - 1 - \lfloor \log_2 \langle b \rangle \rfloor . \tag{4.3}$$

$$\forall n \in \mathbb{N}, b \in \mathbb{B}^n : \langle b \rangle = \langle b[n - 1 - lzero(b) : 0] \rangle. \tag{4.4}$$

The following equation is a lemma on the logical-left-shift function lls from the library:

$$\forall n \in \mathbb{N}, b \in \mathbb{B}^n, sa \in \mathbb{B}^{\lceil \log_2 n \rceil} : \langle lls(b, sa) \rangle = \langle b[n - 1 - \langle sa \rangle : 0] \rangle \cdot 2^{\langle sa \rangle}, \tag{4.5}$$

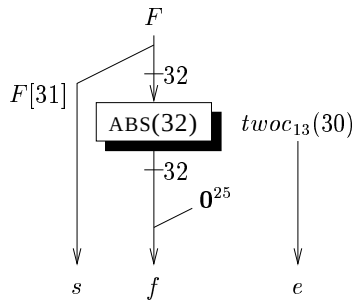
The correctness of the unpacker now follows from the above four lemmas:

$$\begin{aligned} \langle f \rangle &\overset{(4.5)}{=} \langle h[52 - \langle lz \rangle : 0] \rangle \cdot 2^{\langle lz \rangle} \\ &\overset{(4.2)}{=} \langle h[52 - lzero(h) : 0] \rangle \cdot 2^{lzero(h)} \\ &\overset{(4.4)}{=} \langle h \rangle \cdot 2^{lzero(h)} \\ &\overset{(4.3)}{=} \langle h \rangle \cdot 2^{52 - \lfloor \log_2 \langle h \rangle \rfloor} \end{aligned} \tag{4.6}$$

Let $\mathsf{f} := \mathit{bv2fact}_f(F)$. Replacing (4.1) in (4.6), multiplying with 2^{-52}, and applying arithmetic yields

$$\begin{aligned} \langle f \rangle \cdot 2^{-52} &= \mathsf{f} \cdot 2^{-\lfloor \log_2 \mathsf{f} \rfloor} \text{ and} \\ \langle lz \rangle &= -\lfloor \log_2 \mathsf{f} \rfloor . \end{aligned}$$

Hence, the significand $\langle f \rangle \cdot 2^{-52}$ and the exponent $[e] - \langle lz \rangle$ of the right-hand side factoring in lemma 4.3 is computed as in the definition of $\widehat{\mathit{norm}}$ in section 3.1.2 (pg. 11). □



The circuit ABS computes the binary representation of the absolute value of its input F [BJK01a]. $twoc_{13}(30)$ is the 13-bit two's complement bitvector with value 30.

Figure 4.3: Design of the fixed point unpacker

4.1.2 Fixed Point Unpacker

The design of the fixed-point unpacker is slightly different from [MP00], since the conversion unit using this unpacker is different from [MP00] (cf. section 4.5).

Circuit 4.2 (FXUNPACK**)** The fixed point unpacker is a circuit with a 32-bit two's complement operand $F \in \mathbb{B}^{32}$ as input. The outputs of the unpacker are $s \in \mathbb{B}, e \in \mathbb{B}^{13}, f \in \mathbb{B}^{2+55}$ representing a factoring with the value $[F]$ of the operand. The construction of the fixed point unpacker is shown in figure 4.3. ◇

Note that the exponent and significand output of the fixed point unpacker have a different length than the corresponding outputs of the floating point unpacker. This is because the floating point unpacker is connected to the computation units. In contrast, the fixed point unpacker is directly connected to the rounder in the conversion unit. The outputs of the fixed point unpacker therefore equal the inputs of the rounding unit.

The following lemma states the correctness of the fixed point unpacker. The proof is straightforward.

Lemma 4.4 *For all inputs* $F \in \mathbb{B}^{32}$ *holds* $[F] = [\![s, [e], \langle f \rangle \cdot 2^{-55}]\!]$.

How both the floating point and the fixed point unpackers are connected with the rest of the FPU will be described in later sections.

4.2 Rounder

In this section, we describe the design and the verification of the floating point rounder.

Circuit 4.3 (FP-ROUNDER**)** The floating point rounder has the following inputs:

- $s_r \in \mathbb{B}, e_r \in \mathbb{B}^{13}, f_r \in \mathbb{B}^{2+55}$: the bitvector representation of the input factoring; e_r is the two's complement exponent, f_r is the significand with 55 bits behind the binary point.
- $RM \in \mathbb{B}^2$: encodes the rounding mode. The encoding is defined as

$$\mathcal{M} = \begin{cases} zero & \text{if } RM = \mathbf{00}, \\ near & \text{if } RM = \mathbf{01}, \\ up & \text{if } RM = \mathbf{10}, \\ down & \text{if } RM = \mathbf{11}. \end{cases} \tag{4.7}$$

- $dbl \in \mathbb{B}$: specifies whether the rounder shall round to single ($dbl = \mathbf{0}$) or double precision.
- $\mathit{OVFen}, \mathit{UNFen} \in \mathbb{B}$: the enable bits for the *OVF* and *UNF* exceptions, respectively.

The rounder outputs are

- $R \in \mathbb{B}^{64}$: the IEEE bitvector of the result.
- $\mathit{ovf}, \mathit{unf}, \mathit{inx} \in \mathbb{B}$: the exception signals. ◇

Let $(s, e_0, f_0) := (s, [e_r], \langle f_r \rangle \cdot 2^{-55})$ be the factoring represented by the input, and let $x := [\![s, e_0, f_0]\!]$ be its value. We require the rounder output to satisfy the correctness statement as specified in section 3.7.1:

$$\mathit{FPU\text{-}result\text{-}correct}(x, \mathcal{M}, \mathit{OVFen}, \mathit{UNFen})(R, \mathit{ovf}, \mathit{unf}, \mathit{inx}).$$

From theorem 3.38 it immediately follows that the input to the rounder does not need to be the exact result of the floating point operation, i.e., x has to be an α-equivalent approximation of the exact result.

In order to prove the correctness of the rounder outputs, we assume three properties of the inputs. The computational units which compute the rounder inputs will guarantee these properties:

1. The value to round is not zero, i.e., $x \neq 0$. This implies $f_0 > 0$.
2. For denormal input significands, the exponent does not exceed $e_{\max}$:

$$f_0 < 1 \implies e_0 \leq e_{\max} \tag{4.8}$$

   This requirement is different from [MP00]. There it is required that $f_0 < 1 \implies \neg \mathit{OVF}(x, \mathcal{M})$. This requirement is not strong enough for the correctness of the given rounder construction, as will be shown in the proof of theorem 4.6.

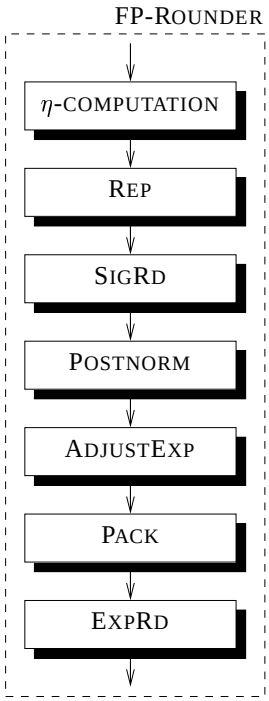


Figure 4.4: Top-level view of the rounder.

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 $2^{e_{\max}}$. Formally, define $\mathit{wrapped}_{\text{bef}}$ as

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

where $A = 3 \cdot 2^{N-2}$ as defined in section 3.3. Note that the function $\mathit{wrapped}_{\text{bef}}$ performs exponent wrapping the same as the function *wrapped*, except for the case that $\mathit{OVF}_{\text{aft}}(x)$ occurs. In the following, let

$$y := \mathit{wrapped}_{\text{bef}}(x, \mathit{OVFen}, \mathit{UNFen}).$$

With these definitions, the third input requirement for the rounder formally reads

$$(\mathit{TINY}(x) \wedge \mathit{UNFen}) \vee (\mathit{OVF}_{\text{bef}}(x) \wedge \mathit{OVFen}) \implies 2^{e_{\min}} < |y| < 2^{e_{\max}}. \quad (4.9)$$

Figure 4.4 shows the top-level design of the rounder. The upper four stages of the design arise from the decomposition theorem. The η-COMPUTATION stage

computes the IEEE factoring of y. The REP stage computes the $(-P)$-representative of the resulting significant. This representative is then rounded in the SIGRD stage. The result of significant rounding is then post-normalized in POSTNORM.

The ADJUSTEXP stage adjusts the result in case of trapped overflows after rounding, i.e., performs exponent wrapping not already performed in $\mathit{wrapped}_{\text{bef}}$. The PACK stage converts the intermediate result to the IEEE format. In case of untrapped overflows, the EXPRD-stage ties the result to either infinity or $X_{\max}$. In the following, we will describe some parts of the rounder in detail. The other parts are very similar to [MP00], and we will describe them only briefly.

4.2.1 η-Computation Stage

The η-COMPUTATION circuit[1] 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. Furthermore, the η-COMPUTATION circuit computes $\mathit{TINY}(x)$ and $\mathit{OVF}_{\text{bef}}(x)$ flags. The basic algorithm for the η-computation is as follows:

1. Compute the logarithm of f_0 using a leading-zero counter on f_r; therefrom decide whether $2^{e_0} \cdot f_0 < 2^{e_{\min}}$, i.e., whether $\mathit{TINY}(x)$ holds. Furthermore compute $\mathit{OVF}_{\text{bef}}(x)$. We will describe the computation of $\mathit{OVF}_{\text{bef}}(x)$ in detail below.

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 (4.9) 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 analogously to the normalization in the unpacker (cf. section 4.1).

   (b) If an untrapped underflow occurs, it holds by definition $|x| < 2^{e_{\min}}$ 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 of two small numbers with exponents slightly greater than $e_{\min}$.

   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

[1] In [MP00], η-computation is called *normalization shift*. We find this term confusing, since η-computation does not always normalize but may de-normalize the inputs in some cases.

shift by $\approx e_{\text{min}} - e_0$. For example, the multiplication $2^{e_{\text{min}}} \cdot 2^{e_{\text{min}}}$ yields $e_0 = 2 \cdot e_{\text{min}} \ll e_{\text{min}}$. In double precision, e.g., this would require an ≈ 1024-bit shifter. Since this very far right shift would require a huge shifter, the η-computation computes only an $(-P)$-equivalent of the exact significant.

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 cyclic shifter together with a rather complex mask- and control-logic enclosing the shifter [MP00, Sect. 8.4.2].

Circuit 4.4 (η-COMPUTATION) The η-computation circuit has the same inputs as the circuit FP-ROUNDER. The outputs of the η-computation are

- $s_n \in \mathbb{B}\,, e_n \in \mathbb{B}^{\,11}, f_n \in \mathbb{B}^{\,1+127}$: represents (an approximation of) the IEEE factoring $\eta(y)$.
- $e_n^+ \in \mathbb{B}^{\,11}$: represents the incremented exponent.
- *TINY*, $\mathit{OVF}_{\text{bef}} \in \mathbb{B}$: active if *TINY*$(x)$ or $\mathit{OVF}_{\text{bef}}(x)$ occur, respectively.
- $RM \in \mathbb{B}^{\,2}, \mathit{dbl}, \mathit{OVFen}, \mathit{UNFen} \in \mathbb{B}$: forwarded from the inputs. $\diamond$

The construction of the normalization shifter and its correctness is described in [MP00, pg. 394–404]. We omit the details here. We only give the detailed correctness proof for the computation of $\mathit{OVF}_{\text{bef}}$ below, since this is wrong in [MP00]. Besides we only give the correctness statement of the η-computation for single precision:

Theorem 4.5 *For all inputs to the η-computation satisfying the rounder input conditions, it holds:*

(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:

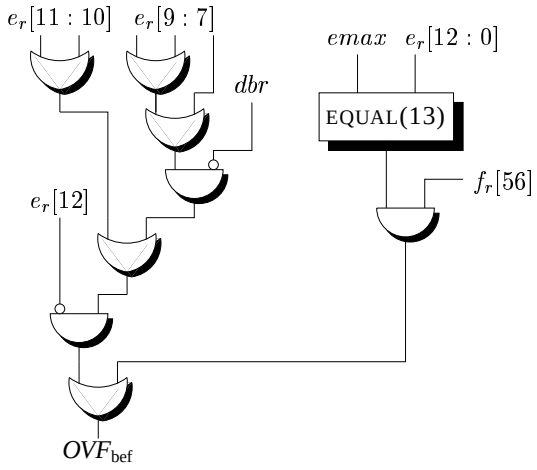
(iii) $(s_n, [e_n[7:0]]_{\text{bias}}, \langle f_n \rangle \cdot 2^{-127})$ *is an IEEE factoring.*

(iv) $[\![s_n, [e_n[7:0]]_{\text{bias}}, \langle f_n \rangle \cdot 2^{-127}]\!] \equiv_\alpha \mathit{wrapped}_{\text{bef}}(x, \mathit{OVFen}, \mathit{UNFen})$ *with* $\alpha = [e_n[7:0]]_{\text{bias}} - 24$.

(v) $[e_n^+[7:0]]_{\text{bias}} = [e_n[7:0]]_{\text{bias}} + 1$

The statement for double precision is analogous.

The proof of correctness of the above theorem is one of the most complex proofs in [MP00]. 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 manual prover commands; in [MP00], the proof is 10 pages long.



EQUAL is an equality-tester from the library [BJK01a]. $emax$ is – depending on the precision – the two's complement representation $\mathbf{0}^3 dbl^3 \mathbf{1}^7$ of $e_{\max}$.

Figure 4.5: Computation of OVF_{bef}

Verification of OVF_{bef}. We exemplarily describe the construction and verification of the OVF_{bef} circuit in detail. The correctness proof in [MP00] is wrong, as will become apparent below. Figure 4.5 shows the circuit computing the OVF_{bef} signal.

Theorem 4.6 *Let $f_r \in \mathbb{B}^{57}, e_r \in \mathbb{B}^{13}, dbl \in \mathbb{B}$ be as in the definition of the inputs of* FP-ROUNDER*, and let x be the value of the rounder input. The OVF_{bef} output of the circuit in figure 4.5 is active, iff $OVF_{\text{bef}}(x)$ holds.*

Proof: It holds

$$\begin{aligned} OVF_{\text{bef}}(x) &\overset{\text{def.}}{\iff} \eta_e(x) > e_{\max} \\ &\overset{\text{Lemma 3.7}(iii)}{\iff} \lfloor \log_2 |x| \rfloor > e_{\max} \\ &\iff e_0 + \lfloor \log_2 f_0 \rfloor > e_{\max} \qquad (4.10) \\ &\overset{(*)}{\iff} e_0 > e_{\max} \vee (e_0 = e_{\max} \wedge f_0 \geq 2). \qquad (4.11) \end{aligned}$$

The last transformation $(*)$ holds because of the input conditions $0 < f_0 < 4$ and $f_0 < 1 \implies e_0 \leq e_{\max}$. To prove $(*)$ we distinguish three cases:

1. $0 < f_0 < 1$: we have $\lfloor \log_2 f_0 \rfloor < 0$ and $e_0 \leq e_{\max}$. Hence, both (4.10) and (4.11) evaluate to false.

2. $1 \leq f_0 < 2$: we have $\lfloor \log_2 f_0 \rfloor = 0$. Hence both (4.10) and (4.11) are true, iff $e_0 > e_{\max}$ holds.

3. $2 \leq f_0 < 4$: we have $\lfloor \log_2 f_0 \rfloor = 1$. Hence (4.10) holds iff $e_0 > e_{\max} - 1$, i.e., iff $e_0 = e_{\max} \vee e_0 > e_{\max}$, which is equivalent to (4.11).

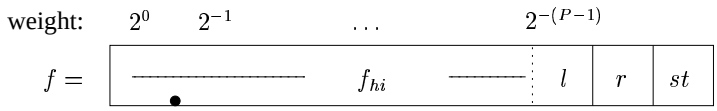


Figure 4.6: Decomposition of the significand into f_{hi}, *least-*, *round-*, and *sticky*-bit.

It is easy to verify that the left side of figure 4.5 computes $e_0 > e_{\max}$, and that the right side computes $e_0 = e_{\max} \wedge f_0 \geq 2$. □

In [MP00], the rounder input condition (4.8) is different from ours. There it is required that $f_0 < 1 \Rightarrow \neg \mathit{OVF}(x, \mathcal{M})$. However, the transformation $(*)$ becomes wrong with this requirement. Assume, for example, $f_0 = 0.5$ and $e_0 = e_{\max} + 1$. Then $|x| = 2^{e_{\max}}$ is representable and does not overflow on rounding, although equation (4.11) evaluates to true in this situation.

4.2.2 Rep, SigRd and Postnorm Stages

The result of the η-computation is passed into the REP stage, where the $(-P)$-representative of f_n is computed. This is done using an OR-tree as suggested by theorem 3.27. We omit the details, since they are fairly simple.

The SIGRD stage rounds the significand as computed by the REP stage. By corollary 3.32 this yields the same result as rounding f_n as returned by the η-computation. For significand rounding, the significand is split according to figure 4.6. The last three bits are called *least-*, *round-*, and *sticky*-bit. Significand rounding is performed by chopping the round- and sticky-bits off the significand, and incrementing the chopped significand in some cases depending on l, r and st, the sign s and the rounding mode $\mathcal{M}$. Whether the significand has to be incremented is determined by

$$\mathit{sigrd\text{-}incr} = \begin{cases} 0 & \text{if } \mathcal{M} = zero, \\ \neg s \wedge (r \vee st) & \text{if } \mathcal{M} = up, \\ s \wedge (r \vee st) & \text{if } \mathcal{M} = down, \\ r \wedge (st \vee l) & \text{if } \mathcal{M} = near. \end{cases}$$

Example: In mode $\mathcal{M} = up$ the significant is never incremented if the number is negative ($s = \mathbf{1}$), since this would decrease the number. If $s = \mathbf{0}$, the significand is incremented if the number was not already representable, i.e., if there are bits behind the least representable bit. That is checked by $r \vee st$.

The correctness of the significand round algorithm is asserted by the following lemmas. The arguments in these lemmas are missing in [MP00]. The first lemma shows how to compute floor and ceiling of the decomposed significand:

Lemma 4.7 *Let $f \in \mathbb{R}$ be decomposed into a sign s and $f_{hi} \in \mathbb{N}, l, r, st \in \{0,1\}$ such that $f = (-1)^s \cdot \left(2f_{hi} + l + \frac{1}{2}r + \frac{1}{4}st\right)$. Note that this is the numerical counterpart to the bitvector decomposition in figure 4.6. The floor and ceiling of f can be computed by*

$$\lfloor f \rfloor = \begin{cases} 2f_{hi} + l & \text{if } s = 0, \\ -(2f_{hi} + l) & \text{if } s = 1 \wedge r = 0 \wedge st = 0, \\ -(2f_{hi} + l + 1) & \text{otherwise.} \end{cases}$$

$$\lceil f \rceil = \begin{cases} -2(f_{hi} + l) & \text{if } s = 1, \\ 2f_{hi} + l & \text{if } s = 0 \wedge r = 0 \wedge st = 0, \\ 2f_{hi} + l + 1 & \text{otherwise.} \end{cases}$$

Proof: The claim follows by case-splitting on s, r, l, st and applying properties of the floor- and ceiling-functions. In PVS, this is done automatically by the strategy `(grind)`. □

This lemma can now be used to prove that the significand rounding algorithm (chopping & incrementing) is correct:

Lemma 4.8 *Let (s, e, f) be an IEEE factoring, and let $f_{hi} \in \mathbb{N}, l, r, st \in \{0,1\}$ be such that*

$$f \cdot 2^{P-1} = 2f_{hi} + l + \tfrac{1}{2}r + \tfrac{1}{4}st.$$

Then it holds

$$\mathit{sigrd}((s,e,f), \mathcal{M}) = 2^{-(P-1)} \cdot \begin{cases} 2f_{hi} + l + 1 & \text{if } \mathit{sigrd\text{-}incr}, \\ 2f_{hi} + l & \text{otherwise}. \end{cases}$$

Proof: By definition of *sigrd* and r_{rat} we have

$$\mathit{sigrd}((s,e,f), \mathcal{M}) = 2^{-(P-1)} \cdot \left| r_{\text{int}}\left((-1)^s \cdot f \cdot 2^{P-1}, \mathcal{M}\right) \right|.$$

The claim now follows by expanding the definition of r_{int} and application of lemma 4.7 to replace the floor- and ceiling-applications in r_{int}. □

The construction of the SIGRD circuit in PVS closely follows [MP00, pp. 406f]. The correctness of the circuit immediately follows from the above lemma. However, in [MP00] there are two bugs in the SIGRD circuit:

1. The circuit for the increment-decision is wrong. The *XOR* gate has to be replaced by an *XNOR* gate.

2. In case of chopping in single precision, the circuit forwards the bits r and st unchanged to the output, although these bits should be tied to $\mathbf{0}$. This makes the arguments on pg. 408 in [MP00] wrong.

Both bugs have been fixed easily.

Besides significand rounding, the circuit SIGRD also computes the signal *inx* as

$$\mathit{inx} \iff (f \neq \mathit{sigrd}((s,e,f),\mathcal{M})),$$

i.e., *inx* is active if significand rounding effectively changes the significand. This is correct by lemma 3.22.

The next stage in the rounding process is post-normalization. This is performed in a straightforward way in stage POSTNORM. We omit the details. However, there is one subtle difference between our post-normalization and the one in [MP00]: our definition of post-normalization ties the sign to 0 in the case that significand rounding yields 0 (cf. section 3.2.2). This is necessary to comply with the definition of $\eta(0)$. Tying the sign to 0 in this case is also implemented in the post-normalization circuit POSTNORM. This allows concise statements such as theorem 4.9(*iii*). However, the IEEE standard explicitly defines the sign bit for the final result of operations in case that rounding yields 0. In order to comply with the specification from the IEEE standard, our rounder implementation saves the input sign bit s_r and replaces the newly computed sign with this original sign in the last rounder stage (see below). The detour of computing a new sign-bit has the only purpose of having concise correctness statements. This is one of the few circuits which have been altered solely to ease verification.

In contrast, in [MP00] the sign is not defined for $\eta(0)$. The arguments on the sign bit of $\eta(0)$ in various places of [MP00] are therefore either fuzzy, missing, or simply wrong. For example, the statement of the decomposition theorem [MP00, Thm. 7.4, pg. 331] is void if the rounded result is 0, since $\eta(0)$ is not well-defined.

Circuit 4.5 (REP, SIGRD, POSTNORM) The combination of the stages REP, SIGRD, and POSTNORM takes as inputs the outputs of the η-COMPUTATION circuits. The outputs are:

- $s_p \in \mathbb{B}$, $e_p \in \mathbb{B}^{11}$, $f_p \in \mathbb{B}^{1+52}$: the post-normalized factoring.
- $s_r, \mathit{TINY}, \mathit{OVF}_{\text{bef}}, \mathit{dbl}, \mathit{OVFen}, \mathit{UNFen} \in \mathbb{B}$, $RM \in \mathbb{B}^2$: forwarded from the η-COMPUTATION outputs.
- *INX* $\in \mathbb{B}$: equals $\mathit{INX}(x, \mathcal{M}, \mathit{OVFen}, \mathit{UNFen})$.
- *SIGovf* $\in \mathbb{B}$: active if the significand round yielded 2. $\diamond$

The correctness statement of the rounder stages so far for single precision is:

Theorem 4.9 *For all inputs to the rounder satisfying the rounder input conditions, it holds:*

(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) $[\![s_p, [e_p[7:0]]_{\text{bias}}, \langle f[52:29]\rangle \cdot 2^{-23}]\!] = \eta(rd(\mathit{wrapped}_{\text{bef}}(x, \mathit{OVFen}, \mathit{UNFen}), \mathcal{M}))$,

(iv) $f[28:0] = \mathbf{0}^{29}$,

(v) SIGovf is active, iff $\mathit{sigrd}((s_0, e_0, f_0), \mathcal{M}) = 2$.

The correctness statement for double precision is analogous.

4.2.3 AdjustExp, Pack and ExpRd Stages

After the post-normalization, the most complex parts of the rounding process are done. It follows the ADJUSTEXP stage, which ties the exponent to $e_{\max} + 1 - A$ in the case that an overflow after rounding with enabled trap occurs. Such overflows are easily detected by testing if both the exponent e_p represents $e_{\max} + 1$ and the significand round yielded 2, i.e., if *SIGovf* is active.

The next stage is the PACK stage, which transforms the intermediate result to IEEE format by tying the exponent of denormal numbers to $\mathbf{0}^N$, and hiding the most-significant significand bit.

Finally, the EXPRD stage computes the result for untrapped overflows. This is a straightforward implementation by some multiplexers of the second part of the definition of *FPU-result-correct* (section 3.7.1, pg. 30). Furthermore, EXPRD ties the sign of the output to the input sign s_r.

Circuit 4.6 (ADJUSTEXP, PACK, EXPRD) The combination of circuits ADJUSTEXP, PACK, and EXPRD takes as inputs the outputs of the POSTNORM circuit. Its outputs are the outputs of the complete rounder as specified in circuit FP-ROUNDER.

Putting it all together, we have the correctness statement of the complete floating point rounder:

Theorem 4.10 *For all inputs to the circuit* FP-ROUNDER *satisfying the rounder input conditions, it holds*

$$\mathit{FPU\text{-}result\text{-}correct}(x, \mathcal{M}, \mathit{OVFen}, \mathit{UNFen})(R, \mathit{ovf}, \mathit{unf}, \mathit{inx}),$$

where $R, \mathit{ovf}, \mathit{unf}, \mathit{inx}$ *are the outputs of* FP-ROUNDER.

In order to pipeline the floating point units, the rounder circuit FP-ROUNDER is decomposed into two stages RD-STG1 and RD-STG2. The stage RD-STG1 comprises the η-computation and the representative computation, and the stage RD-STG2 comprises all other parts of the rounder. However, the actual intersection point is not important for the verification, but only for balancing the depth of the pipeline stages for the later implementation.

4.3 Multiplicative Floating Point Unit

In this section, we describe the multiplicative floating point unit. We first explain the multiplication and division algorithm, before we proceed to present the hardware implementing these algorithms. As in the previous sections, we will not describe the hardware in detail, since the hardware closely follows [MP00]. We conclude the section by combining the unpacker, the multiplicative computation unit, and the rounder to the complete multiplicative FPU.

4.3.1 Multiplication/Division Algorithm

Basic Algorithm

The multiplicative computation unit gets as input the two operands a and b from two separate floating point unpackers. For now, we assume that the operands are non-special, nonzero floating point numbers. The other cases are handled as special cases in section 4.3.4. The unpackers normalize denormal operands.

We denote the normalized factorings of a and b by (s_a, e_a, f_a) and (s_b, e_b, f_b), respectively. For the description of the algorithm, it is convenient to see these as numbers and ignore that in hardware these numbers are represented by bitvectors. We return to bitvectors in the description of the actual hardware.

The basic algorithm for multiplication is to add up the exponents and multiply the significands. This yields a result significand in the interval $[1, 4)$, and hence the rounder input condition (4.8) on page 44 is trivially fulfilled.

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

A major bug of [MP00] is that the multiplication with 2 is missing. If this multiplication is omitted, a significand in the interval $(1/2, 2)$ is passed to the rounder. Since the difference of the operand exponents may be less than $e_{\min}$, the rounder input condition (4.8) 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 described in this section 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 4.11 *Let (s_a, e_a, f_a) and (s_b, e_b, f_b) be normal factorings with nonzero values $a = [\![s_a, e_a, f_a]\!]$ and $b = [\![s_b, e_b, f_b]\!]$. 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:

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

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 $[1, 4)$:

$$1 \leq [f_a \cdot f_b]_{-P} < 4$$
$$1 \leq 2 \cdot [f_a/f_b]_{-(P+1)} < 4.$$

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 (4.9) *on page 45):*

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

Proof: We prove the theorem for division; the proof for multiplication is analogous. It holds

$$\begin{array}{lll} a/b & = & ((-1)^{s_a} \cdot 2^{e_a} \cdot f_a) \,/\, ((-1)^{s_b} \cdot 2^{e_b} \cdot f_b) \\ & = & (-1)^{s_a \oplus s_b} \cdot 2^{e_a - e_b} \cdot (f_a/f_b) \\ & \equiv_{e_a-e_b-(P+1)} & [(-1)^{s_a \oplus s_b} \cdot 2^{e_a - e_b} \cdot (f_a/f_b)]_{e_a-e_b-(P+1)} \\ & & \qquad \text{(by lemma 3.23(}ii\text{))} \\ & = & (-1)^{s_a \oplus s_b} \cdot 2^{e_a - e_b - 1} \cdot 2 \cdot [f_a/f_b]_{-(P+1)} \\ & & \qquad \text{(by lemma 3.23(}iv\text{,}v\text{)).} \end{array}$$

It holds $\hat{e} = \lfloor \log_2 |a/b| \rfloor = e_a - e_b + \lfloor \log_2 f_a/f_b \rfloor \geq e_a - e_b - 1$ since $f_a/f_b \geq 1/2$. We therefore may coarsen the relation by applying lemma 3.23(*vii*), yielding

$$a/b \equiv_{\hat{e}-P} (-1)^{s_a \oplus s_b} \cdot 2^{e_a - e_b - 1} \cdot 2 \cdot [f_a/f_b]_{-(P+1)}.$$

This proves the first claim. The second claim follows easily from lemma 3.24. For the third claim, observe that since the input factorings are normalized and representable, it holds $e_a, e_b \in \{e_{\min} - P, \ldots, e_{\max}\}$ and hence $e_{\min} - P - e_{\max} \leq e_a - e_b \leq e_{\max} - e_{\min} + P$. Evaluation of $e_{\min}, e_{\max}$ and A proves the claim. □

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 $[f_a/f_b]_{-(P+1)}$. The remainder of this subsection will describe this algorithm. We start by explaining the lookup table from which the initial approximation is obtained. We then briefly describing the Newton-Raphson iteration. Finally, we describe how the representative $[f_a/f_b]_{-(P+1)}$ is computed from the approximation r.

Initial Approximation.

The initial approximation x_0 is loaded from a lookup table, which is implemented as a ROM in hardware. The lookup table has 256 entries, each 8 bits in width. Let $\mathbf{f_b} \in \mathbb{B}^{1+52}$ denote the bitvector representation of f_b, i.e.,

$$\langle \mathbf{f_b} \rangle \cdot 2^{-52} = f_b.$$

Since f_b is normal, the most significand bit $\mathbf{f_b}[52]$ satisfies $\mathbf{f_b}[52] = \mathbf{1}$ for all operands. $\mathbf{f_b}[52]$ is therefore not suited as "information carrier" for the lookup table. Therefore the next 8 bits $\mathbf{f_b}[51:44]$ are used to address the lookup table. Let $i := \langle \mathbf{f_b}[51:44] \rangle$ denote the value of these address bits.

In [MP00], the content of the lookup table is given implicitly, i.e., an algorithm is defined describing the content of every ROM cell. In PVS, we have defined the lookup table explicitly as a large *case*-statement mapping addresses from $\{0,\ldots,255\}$ to bitvectors from $\mathbb{B}^8$. Let *lookup*(i) denote the PVS function comprising this *case*-statement. The lookup table delivers an bitvector of length 8, which is extended to the actual initial approximation. The binary representation of the initial approximation is defined as

$$\mathbf{x_0} := \mathbf{0.1} \circ \mathit{lookup}(i) \circ \mathbf{0}^{48} \in \mathbb{B}^{1+57}, \tag{4.12}$$

hence the value of the initial approximation is

$$\begin{aligned} x_0 &= \langle \mathbf{x_0} \rangle \cdot 2^{-57} \\ &= \tfrac{1}{2} + \langle \mathit{lookup}(i) \rangle \cdot 2^{-9}. \end{aligned} \tag{4.13}$$

Before we give the correctness statement of the complete initial approximation, we state a lemma on the content of the actual lookup table:

Lemma 4.12 *Let $i \in \{0,\ldots,255\}$, and let $f := 1 + i \cdot 2^{-8} + 2^{-9}$. It holds*

$$\left|\left(\tfrac{1}{2} + \langle \mathit{lookup}(i) \rangle \cdot 2^{-9}\right) - 1/f\right| < 2^{-9},$$

that is, the content of the lookup table approximates the reciprocal of f.

Proof: The claim is proved in PVS by separately analyzing the content of each lookup table entry. Each case is proved by applying basic arithmetic. □

The above lemma characterizes the approximation error for significands which are of the special form $1+i\cdot 2^{-8}+2^{-9}$. Every representable significand $f_b \in [1,2)$ is approximated by such a number, since with $i = \langle \mathbf{f_b}[51:44] \rangle$ it holds

$$\left|f_b - (1 + i \cdot 2^{-8} + 2^{-9})\right| \leq 2^{-9},$$

and some basic arithmetic yields

$$\left|1/f_b - 1/(1 + i \cdot 2^{-8} + 2^{-9})\right| \leq 2^{-9}.$$

Together with lemma 4.12 this proves the following theorem on the error of the complete initial approximation:

Theorem 4.13 *The initial approximation x_0 as defined in* (4.12) *and* (4.13) *satisfies*

$$0 < |1/f_b - x_0| < 2^{-8}.$$

Newton-Raphson Iteration.

Starting from the initial approximation x_0 one defines the sequence x_i by

$$x_{i+1} := x_i \cdot (2 - f_b \cdot x_i).$$

It is easy to show that the sequence x_i converges quadratically to $1/f_b$ if the initial approximation is precise enough (see [MP00, pg. 374f], e.g.). The problem with this algorithm is that the intermediate results have ever larger binary representations. In order to implement the algorithm in hardware, one chops all bits after the 57^{th} bit behind the binary point of every intermediate result. This is mathematically represented by the function $\lfloor\cdot\rfloor_\sigma$ which chops all digits after the σ^{th} digit behind the binary point:

$$\lfloor z \rfloor_\sigma := 2^{-\sigma} \cdot \lfloor z \cdot 2^\sigma \rfloor .$$

In hardware, the computation of $2 - \lfloor f_b \cdot x_i \rfloor_{57}$ would require an incrementer to compute the two's complement. Therefore, one deliberately introduces one further approximation error and computes

$$A_i := 2 - \lfloor f_b \cdot x_i \rfloor_{57} - 2^{-57} \tag{4.14}$$

instead of $2 - \lfloor f_b \cdot x_i \rfloor_{57}$. In hardware, this can be implemented by simply inverting the bitvector representation of $\lfloor f_b \cdot x_i \rfloor_{57}$, which saves the delay of the incrementer. The approximated sequence x_i is hence defined as

$$x_{i+1} := \left\lfloor x_i \cdot (2 - \lfloor f_b \cdot x_i \rfloor_{57} - 2^{-57}) \right\rfloor_{57}. \tag{4.15}$$

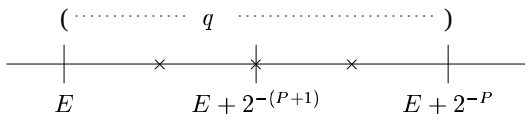
The approximation error is defined as

$$\delta_i := 1/f_b - x_i. \tag{4.16}$$

In particular, δ_0 denotes the error of the initial approximation. The following theorem summarizes the error analysis of the Newton-Raphson algorithm with finite-precision intermediate results. The arguments closely follow [MP00], we therefore omit the proof:

Theorem 4.14 *Let $1 \leq f_b < 2$, and x_0 be the initial approximation such that $0 < |\delta_0| < 2^{-8}$. It holds*

$$0 < \delta_2 < 1.1 \cdot 2^{-29},$$
$$0 < \delta_3 < 2^{-55}.$$



The bars are the integral multiples of $2^{-(P+1)}$. The crosses indicate the three possible $(-(P+1))$–representatives of q.

Figure 4.7: Computation of $[q]_{-(P+1)}$

Computation of the Representative.

By theorems 4.14 and 4.13, x_2 and x_3 are approximations of the reciprocal $1/f_b$. Let $r = x_2$ in single and $r = x_3$ in double precision, respectively. Note that r by definition has a binary representation with 57 bits behind the binary point, since it is the result of a $\lfloor\cdot\rfloor_{57}$–application. In order to compute $[f_a/f_b]_{-(P+1)}$ from the approximation r we define

$$\begin{aligned} E &:= \lfloor f_a \cdot r \rfloor_{P+1}\,, \\ E_b &:= E \cdot f_b. \end{aligned}$$

The following lemma states the important property of E:

Lemma 4.15 *It holds* $E < f_a/f_b < E + 2^{-P}$.

Proof: The claim follows from theorem 4.14 and properties of $\lfloor\cdot\rfloor_\sigma$. The proof is as in [MP00, pg. 380]. □

By lemma 4.15, E is an approximation of the quotient f_a/f_b. The remaining problem is to compute a $(-(P+1))$-representative of the quotient. Figure 4.7 illustrates this problem. The exact quotient q lies between E and $E + 2^{-P}$. There are three possible positions for the $(-(P+1))$–representatives of q. The task is to decide which of the three positions is the representative of q, i.e., in which part of the interval $(E, E + 2^{-P})$ the exact quotient q lies.

Why this problem is non-trivial if the approximation is computed by Newton-Raphson iteration, and why it is not sufficient to simply obtain some more precision by an additional iteration step is, e.g., described in [OF97]. We omit this discussion.

Lemma 4.16 *Let* $f_a, f_b, E \in \mathbb{R}$, $q = f_a/f_b$, $E_b = E \cdot f_b$. *Assume* $E < q < E + 2^{-P}$ *and* $E \cdot 2^{P+1} \in \mathbb{Z}$. *It holds*

$$[q]_{-(P+1)} = \begin{cases} E + 2^{-(P+2)} & \text{if } q < E + 2^{-(P+1)}, \\ E + 2^{-(P+1)} & \text{if } q = E + 2^{-(P+1)}, \\ E + 3 \cdot 2^{-(P+2)} & \text{if } q > E + 2^{-(P+1)}. \end{cases}$$

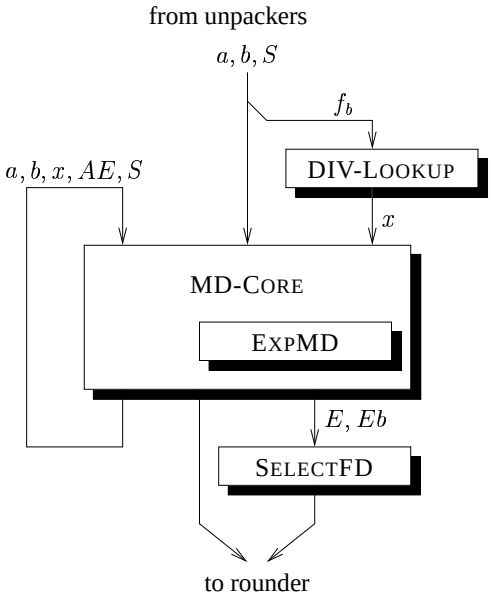


Figure 4.8: Top-level schematics of the multiplicative funtional unit

By multiplying the comparisons on the right side with f_b, and replacing the definition of E_b, it holds

$$[q]_{-(P+1)} = \begin{cases} E + 2^{-(P+2)} & \text{if } f_a < E_b + f_b \cdot 2^{-(P+1)}, \\ E + 2^{-(P+1)} & \text{if } f_a = E_b + f_b \cdot 2^{-(P+1)}, \\ E + 3 \cdot 2^{-(P+2)} & \text{if } f_a > E_b + f_b \cdot 2^{-(P+1)}. \end{cases} \quad (4.17)$$

Proof: The claim follows easily from the definition of $[\cdot]_{-(P+1)}$ and properties of $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$. □

Equation 4.17 allows the computation of $[q]_{-(P+1)}$ without using the exact quotient f_a / f_b.

4.3.2 Hardware Implementation

We briefly describe the hardware which implements the above algorithms. From here on let f_b, E etc. denote bitvector representations of the numbers used in the previous sections. Figure 4.8 shows the top-level schematics of the multiplicative unit. The input operands a and b are received from two floating point unpackers as four-tuples (s_a, e_a, lz_a, f_a) and (s_b, e_b, lz_b, f_b) (see section 4.1).

The circuit DIV-LOOKUP contains the ROM for the lookup table and generates the initial approximation from f_b.

The circuit MD-CORE performs one multiplication, which either accounts for one of the multiplications in the Newton-Raphson iteration step (4.15), the computation of E or E_b, or for the multiplication of the significands in case the operation

is a multiplication. In case of divisions, the results are fed back to loop several times through the MD-CORE circuit. During iteration, the most interesting outputs of circuit MD-CORE are x and AE: x is the binary representation of the current approximation, i.e., x represents x_i. The intermediate result A_i of the inner multiplication (4.14) is represented by AE. When the iteration is finished and E and E_b are to be computed, AE is used to represent E.

The circuit EXPMD computes the result exponent. EXPMD is a sub-circuit of MD-CORE.

The output of the circuit MD-CORE is fed to the rounder in the case that the operation is a multiplication. In case of division, the result is fed to the circuit SELECTFD, which computes the representative $[f_a/f_b]_{-(P+1)}$ from E and E_b.

The number of iterations for divisions depends on the precision: two Newton/Raphson iterations for single and three for double precision operations are needed. Each Newton/Raphson iteration step takes two multiplications (cf. equation (4.15)). The computation of E and E_b takes two further multiplications. Altogether, this yields 6/8 multiplications for single/double precision divisions, respectively. Each of the multiplications corresponds to one iteration through the MD-CORE circuit.

In order to distinguish between multiplications and divisions, and to count the number of remaining iterations, each operation traversing through MD-CORE is assigned a state variable S holding information on the progress of the operation. The state type *MD-State* is defined as *MD-State* $= \{$MUL, DIV21, DIV20, DIV11, DIV10, DIV01, DIV00, DIVE, DIVEB$\}$. The state MUL indicates that the operation is a multiplication. The remaining states are used for divisions to count the number of iterations. The states DIVij indicate that i further iteration steps are needed, including the current step. If $j = 1$ ($j = 0$), the current iteration step performs the inner (outer) multiplication of the iteration step. The states DIVE and DIVEB indicate that E and E_b are currently being computed, respectively.

Single precision divisions proceed through the states DIV11 to DIVEB in the above order, while double precision divisions proceed through DIV21 to DIVEB. The next-state function *md-nxtstate*(S) is defined as

$$\textit{md-nxtstate}(S) = \begin{cases} \text{MUL} & \text{if } S = \text{MUL} \\ \text{DIV20} & \text{if } S = \text{DIV21} \\ \text{DIV11} & \text{if } S = \text{DIV20} \\ \text{DIV10} & \text{if } S = \text{DIV11} \\ \text{DIV01} & \text{if } S = \text{DIV10} \\ \text{DIV00} & \text{if } S = \text{DIV01} \\ \text{DIVE} & \text{if } S = \text{DIV00} \\ \text{DIVEB} & \text{if } S = \text{DIVE} \\ \text{DIVEB} & \text{if } S = \text{DIVEB}. \end{cases}$$

The computation of the next state *md-nxtstate*(S) of the operation is integrated into the circuit MD-CORE.

Circuit 4.7 (DIV-LOOKUP) The initial approximation lookup table takes as argument the normalized significand $f_b \in \mathbb{B}^{1+52}$ and returns $x_0 \in \mathbb{B}^{1+57}$ as defined in (4.12). $\diamond$

Circuit 4.8 (MD-CORE) The circuit MD-CORE has the following inputs:

- $s_a, s_b \in \mathbb{B}, e_a, e_b \in \mathbb{B}^{11}, lz_a, lz_b \in \mathbb{B}^{6}, f_a, f_b \in \mathbb{B}^{1+52}$: the unpacked, normalized input operands. Note that the unpackers deliver the exponents as a combination of e_a and lz_a, i.e., the normalized exponent is $[e_a] - \langle lz_a \rangle$ (analogous for b).
- $S \in$ *MD-State*: the state of the operation. The states are represented by distinct 4-bit bitvectors. Initially, the state S is computed from the op-code of the operation (cf. Appendix A), during the iterations the next state is taken from the feedback loop.
- $x \in \mathbb{B}^{1+57}$: the representation of the current approximation x_i. Initially, x comes from DIV-LOOKUP, during the iterations x comes from the feedback loop.
- $AE \in \mathbb{B}^{1+57}$: During the Newton/Raphson iteration ($S \notin \{$MUL,DIVEB$\}$), AE represents A_i, i.e., the intermediate result of the inner product of the current Newton/Raphson iteration step. If the current state is DIVEB, AE represents E.
- $RM, dbl,$ *OVFen*, *UNFen*: the usual flags.

The circuit has three sets of outputs: the first set is fed back to the input and is used for the iteration, the second set is fed directly to the rounder and is used for multiplications, and the third set is fed to the representative computation in the circuit SELECTFD, which is described below.

1. The outputs which are fed back have the same format as the inputs of circuit MD-CORE. All inputs except x, AE and S are fed back unchanged. If the state S is of the form DIVi1, then x remains unchanged, and AE becomes A_i, i.e, the inner multiplication of the iteration step is performed. If the state S is of the form DIVi0, then AE remains unchanged, and x becomes $\lfloor x \cdot AE \rfloor_{57}$, i.e., the outer multiplication of the iteration step is performed. If $S =$ DIVE, x remains unchanged and AE becomes E, i.e., $\lfloor f_a \cdot x \rfloor_{P+1}$. The new state S is computed as *md-nxtstate*(S).
2. In case of multiplications, the outputs to the rounder are
   - $s_r \in \mathbb{B}$: the result sign, computed as $s_a \oplus s_b$.

- $e_r \in \mathbb{B}^{13}$: the result exponent, computed by the circuit EXPMD described below.
- $f_r \in \mathbb{B}^{2+55}$: the result significand, computed as $f_a \cdot f_b$ followed by a sticky-bit computation.
- $RM, dbl,$ *OVFen*, *UNFen*: passed unchanged from the inputs.

3. The outputs to the representative computation in SELECTFD are
   - $s_r \in \mathbb{B}$: the result sign, computed as $s_a \oplus s_b$.
   - $e_r \in \mathbb{B}^{13}$: the exponent, computed by the circuit EXPMD.
   - $f_a, f_b \in \mathbb{B}^{1+52}$: passed unchanged from the inputs.
   - $AE \in \mathbb{B}^{1+57}$: the representation of E is passed unchanged from the input.
   - $Eb \in \mathbb{B}^{1+114}$: the representation of $E_b = E \cdot f_b$.
   - $RM, dbl,$ *OVFen*, *UNFen*: passed unchanged from the inputs.

The heart of the MD-CORE circuit is a (58×58)-bit multiplier. This multiplier is built from two (29×29)-bit and one (30×30)-bit multipliers and four adders using the scheme of Karatsuba/Ofman [KO63]. In [MP00], the multiplier is implemented as a Wallace tree [Wal64]. However, our multiplier implementation is better suited for the implementation of the FPU on an FPGA (cf. section 6.5). The change of the multiplier implementation has virtually no impact on the correctness proof. $\diamond$

Circuit 4.9 (EXPMD) The circuit EXPMD has the following inputs:

- $e_a, e_b \in \mathbb{B}^{11}, lz_a, lz_b \in \mathbb{B}^{6}$: the exponent and leading-zero outputs of the unpackers.
- *fdiv*: indicates whether a multiplication or division is being computed.

The output of the circuit is $e_r \in \mathbb{B}^{13}$. The computation of e_r is performed as suggested by theorem 4.11. In particular, the circuit EXPMD adds the additonal -1 in case of divisions (cf. theorem 4.11). In order to implement this, the implementation from [MP00] had to be extended by an additonal 3/2-adder stage. $\diamond$

The correctness statement of the exponent computation is:

Lemma 4.17 *It holds:*

$$\begin{aligned} \mathit{fdiv} = \mathbf{0} &\implies [er] = ([e_a] - \langle lz_a \rangle) + ([e_b] - \langle lz_b \rangle) \\ \mathit{fdiv} = \mathbf{1} &\implies [er] = ([e_a] - \langle lz_a \rangle) - ([e_b] - \langle lz_b \rangle) - 1 \end{aligned}$$

Circuit 4.10 (SELECTFD) The circuit SELECTFD takes as inputs the respective outputs of circuit MD-CORE. The outputs of the circuit are

- $s_r \in \mathbb{B}\,, e_r \in \mathbb{B}^{13}$: passed unchanged
- $f_r \in \mathbb{B}^{2+55}$: the representation of $2 \cdot [f_a/f_b]_{-(P+1)}$ computed as described in lemma 4.16. Note that the multiplication by 2 is done here.
- $RM, dbl,$ *OVFen*, *UNFen*: passed unchanged from the inputs.

The construction of the circuit is as in [MP00, pg. 386], except that f_r is shifted one to the left in order to implement the multiplication by 2. ◇

Altogether, the circuits satisfy the following correctness statements. The theorems follow from the construction of the hardware in conjunction with the lemmas and theorems above.

Theorem 4.18 *Let (s_a, e_a, lz_a, f_a) and (s_b, e_b, lz_b, f_b) be nonzero, non-special, unpacked operands with values a and b which are fed into the circuit* MD-CORE. *Let the state input S of* MD-CORE *be* MUL. *Consider the multiplication outputs s_r, e_r, f_r of* MD-CORE. *Let $\hat{e} = \hat{\eta}_e(a \cdot b)$. It holds*

$$a \cdot b \equiv_{\hat{e}-P} [\![s_r, [e_r], \langle f_r \rangle \cdot 2^{-55}]\!],$$

that is, the circuit MD-CORE *computes an appropriate approximation of the exact product of a and b.*

Theorem 4.19 *Let (s_a, e_a, lz_a, f_a) and (s_b, e_b, lz_b, f_b) be nonzero, non-special, unpacked operands with values a and b which are fed into the circuit* MD-CORE. *Let $dbl = \mathbf{1}$, i.e., the operation be a double precision operation. Let the state input S be* DIV21, *and the x input be the initial approximation obtained from the* DIV-LOOKUP *circuit. Iterate the circuit* MD-CORE *8 times and feed the outputs to the circuit* SELECTFD. *Let s_r, e_r, f_r be the outputs of* SELECTFD *obtained in this way. Let $\hat{e} = \hat{\eta}_e(a/b)$. It holds*

$$a/b \equiv_{\hat{e}-53} [\![s_r, [e_r], \langle f_r \rangle \cdot 2^{-55}]\!],$$

that is, the outputs are an appropriate approximation of the exact quotient of a and b. The correctness statement for single precision is analogous. Single precision divisions started with $S =$ DIV11 *take 6 iterations of the circuit* MD-CORE.

4.3.3 Special Cases

The circuits described in the previous sections can only handle nonzero, non-special operands. In the case that one of the operands is zero or a special value (∞, NaN), the corresponding floating point unpacker signals that by activating the appropriate output signal as described in section 4.1.1. The circuit MD-SPECIAL therefrom computes the result of the operation according to table 4.1. The circuit MD-SPECIAL is implemented by a multiplexer-tree. The construction is trivial though error-prone; here formal verification helps avoiding errors even in the design phase.

$a \cdot b$	b				
a	y	0	∞	qNaN	sNaN
x		0	∞		
0	0	0	qNaN		
∞	∞	qNaN	∞		
qNaN	qNaN*				
sNaN	qNaN				

a/b	b				
a	y	0	∞	qNaN	sNaN
x		∞	0		
0	0	qNaN	0		
∞	∞	∞	qNaN		
qNaN	qNaN*				
sNaN	qNaN				

qNaN (sNaN) denotes quite (signalling) NaNs; qNaN* denotes one of the input NaNs. In any case, the output sign is the XOR of the input signs.

Table 4.1: Result of special cases during multiplication/division

4.3.4 Putting It All Together

We now are ready to combine all the circuits described so far in this chapter to the complete, yet combinatorial, multiplicative floating point unit. In chapter 5, we will descibe how this floating point unit is pipelined.

Let MD-UNP denote the combination of one unpacker FP-UNPACK for each of the two operands a and b, the circuit MD-SPECIAL for handling special operands, and the circuit DIV-LOOKUP performing initial approximation lookup for divisions. The circuits are connected in the obious way. The output of circuit MD-UNP is either the result of the special operation, or the input for the next stage.

Let MD-STG1 and MD-STG2 be two circuits obtained by dividing MD-CORE into two stages. In our implementation, MD-STG1 consists of two multipliers and two adders from the Karatsuba/Ofman scheme, MD-STG2 consists of one multiplier and two adders for Karatsuba/Ofman, and the rest of the logic in the circuit MD-CORE. However, the exact subdivision is not important in the following, its only purpose is to balance the delay of the parts for later pipelining.

The remaining stages of the multiplicative FPU are the circuit SELECTFD and the rounder stages RD-STG1 and RD-STG2 (cf. section 4.2).

Circuit 4.11 (MD-COMB) The combinatorial multiplicative FPU MD-COMB has the following inputs:

- $a, b \in \mathbb{B}^{64}$: the operands are IEEE bitvectors,

- *OVFen*, *UNFen* $\in \mathbb{B}$, $RM \in \mathbb{B}^2$: the exception masks and rounding mode, respectively.
- $opcode \in \mathbb{B}^9$: the operation code. The encoding is listed in appendix A.

The outputs are

- $r \in \mathbb{B}^{64}$: the result is an IEEE bitvector,
- *ovf*, *unf*, *inx*, *inv*, *divz* $\in \mathbb{B}$: the five exception signals.

The functionality is defined as

$$\text{MD-COMB} := \begin{cases} \text{MD-UNP} & \text{if special operands} \\ \text{RD-STG2} \circ \text{RD-STG1} \circ \text{MD-STG2} \circ \text{MD-STG1} \circ \text{MD-UNP} & \\ \qquad \text{if operation is multiplication} & \\ \text{RD-STG2} \circ \text{RD-STG1} \circ \text{SELECTFD} \circ & \\ \qquad (\text{MD-STG2} \circ \text{MD-STG1})^6 \circ \text{MD-UNP} & \\ \qquad \text{if operation is single precision division} & \\ \text{RD-STG2} \circ \text{RD-STG1} \circ \text{SELECTFD} \circ & \\ \qquad (\text{MD-STG2} \circ \text{MD-STG1})^8 \circ \text{MD-UNP} & \\ \qquad \text{if operation is double precision division} & \end{cases}$$

Here, $\circ$ means composition of the circuits; the inputs/outputs of the circuits are connected in the obvious way. The construct $(\ldots)^i$ stands for i-fold composition. $\diamond$

We now prove the overall correctness of the FPU.

Theorem 4.20 *Let a and b be nonzero, non-special IEEE bitvectors, let a rounding mode RM and flags OVFen and UNFen be given. Let the operation to be performed be a multiplication (respectively a division). Let $p := [\![a]\!] \cdot [\![b]\!]$ be the exact result (respectively $p := [\![a]\!]/[\![b]\!]$). Let w be the result of the operation as computed by circuit* MD-COMB*, and let ovf, unf, and inx be the computed exception flags. It holds:*

$$\textit{FPU-result-correct}(p, RM, \textit{OVFen}, \textit{UNFen})(w, \textit{ovf}, \textit{unf}, \textit{inx}),$$

with FPU-result-correct as defined in section 3.7.1.

Proof: We only prove the case of multiplications; divisions are completly analogous. By definition of circuit MD-COMB, the result is computed by RD-STG2 $\circ$ RD-STG1 $\circ$ MD-STG2 $\circ$ MD-STG1 $\circ$ MD-UNP. By lemma 4.3, the unpacker passes the normalized operands a and b to stages MD-STG1 and MD-STG2. By theorems 4.11 and 4.18, theses stages compute a representative of the exact product which satisfies the rounder input conditions. Hence, by theorem 4.10, the rounder computes the correct result and exception flags. $\square$

Note that in order to satisfy the rounder input condition (4.8) on page 44 in case of divisions, the result significand has to be multiplied by 2 to yield a significand ≥ 1 (cf. Theorem 4.11). As mentioned before, this is missing in [MP00]. There, the formal "putting it all together" is not performed. We believe that this is the reason why this bug has been overlooked. This further shows that the verification of (even large) sub-parts of a system does not give ultimate confidence in the correctness of the system; the system has to be verified as a whole.

The correctness of operations on special operands is covered by a series of theorems, one for each entry in table 4.1. We exemplarily state one of the theorems:

Theorem 4.21 *Let a and b be IEEE bitvectors such that inf(a) and $textzero(b)$ hold. Let the operation be a multiplication. Then the result of circuit* MD-COMB *is a quiet NaN, and the INV signal is raised.*

Proof: The correctness follows from lemma 4.1 which ensures that the special operands are correctly recognized, and from the construction of circuit MD-SPECIAL which delivers the result. □

4.4 Additive Floating Point Unit

In this section, we describe the floating point unit for addition and subtraction. The core of this FPU has been verified by Christoph Berg in his master thesis [Ber01]. The design and the verification of the core is described in detail in [Ber01], we therefore restate only the interface and the correctness statement. We then proceed as in the section on the multiplicative unit by describing the special cases, and by combining the parts to form the complete additive unit.

4.4.1 Additive FPU Core

The additive FPU core is implemented by the circuit FP-ADDER from [Ber01]:

Circuit 4.12 (FP-ADDER) The circuit FP-ADDER has the following inputs:

- $s_a, s_b \in \mathbb{B}, e_a, e_b \in \mathbb{B}^{11}, f_a, f_b \in \mathbb{B}^{1+52}$: the unpacked operands. The operands do not get normalized by the unpackers.
- $sub \in \mathbb{B}$: If $sub = \mathbf{0}$ an addition is performed, otherwise a subtraction.

The outputs of the circuit are $s_s \in \mathbb{B}, e_s \in \mathbb{B}^{11}, f_s \in \mathbb{B}^{2+55}$ representing the result factoring. ◇

Theorem 4.22 *Let a, b be non-special, possibly zero, operands, let $S := [\![a]\!] + [\![b]\!]$ if $sub = \mathbf{0}$, otherwise $S := [\![a]\!] - [\![b]\!]$, assume $S \neq 0$, and let $\hat{e} = \hat{\eta}_e(S)$. The*

$a+b$	b				
a	y	$+\infty$	$-\infty$	qNaN	sNaN
x		$+\infty$	$-\infty$	qNaN*	
$+\infty$	$+\infty$	$+\infty$	qNaN		
$-\infty$	$-\infty$	qNaN	$-\infty$		
qNaN	qNaN*				
sNaN	qNaN				

$a-b$	b				
a	y	$+\infty$	$-\infty$	qNaN	sNaN
x		$-\infty$	$+\infty$	qNaN*	
$+\infty$	$+\infty$	qNaN	$+\infty$		
$-\infty$	$-\infty$	$-\infty$	qNaN		
qNaN	qNaN*				
sNaN	qNaN				

qNaN (sNaN) denotes quite (signaling) NaNs; qNaN* denotes one of the input NaNs. The sign of operations with result 0 will be defined in section 4.4.3

Table 4.2: Result of special cases during addition/subtraction

outputs of circuit FP-ADDER *satisfy*

$$\begin{aligned} [\![s_s, e_s, f_s]\!] &\equiv_{\hat{e}-P} S, \\ 2^{e_{\min}-A} \leq [\![s_s, e_s, f_s]\!] &\leq 2^{e_{\max}+A}, \\ [e_s] &\leq e_{\max}, \end{aligned}$$

that is, the rounder input requirements (section 4.2) are fulfilled.

Proof: The proof is given in [Ber01, Chap. 5] □

4.4.2 Special Cases

A special case for the additive FPU occurs if one of the operands is a special operand (∞, NaN), or if the exact result of the operation is zero. Table 4.2 shows the result of operations in the former case. The case of exact results zero has to be handled as a special case because of the sign of zero results. This will be described in more detail in the next section.

As in the multiplicative FPU, the special operand cases are handled by a circuit ADD-SPECIAL which is implemented by a multiplexer-tree to implement the table 4.2. The circuit ADD-SPECIAL also detects whether the operation yields zero as exact result, and outputs a correctly signed zero in this case. The correct sign is

defined in the next section. Whether two operands a, b yield zero as exact result is determined according to

$$\begin{aligned} [\![a]\!] + [\![b]\!] = 0 \quad &\iff \quad [\![a]\!] = -[\![b]\!] \\ &\iff \quad (s_a = \neg s_b) \wedge (e_a = e_b) \wedge (f_a = f_b) \end{aligned}$$

according to lemma 3.4 (analogously for subtraction $[\![a]\!] - [\![b]\!]$).

4.4.3 The Sign of Addition/Subtraction

The definition of the sign of the result of additions and subtractions is one of the most confusing parts of the IEEE standard [IEEE]:

> *(. . .) the sign of a sum, or a difference $x-y$ regarded as sum $x+(-y)$, differs from at most one of the addends signs (. . .) When the sum of two operands with opposite signs (or the difference of two operands with like signs) is exactly zero, the sign of that sum (or difference) shall be $+$ in all rounding modes except round toward $-\infty$, in which mode that sign shall be $-$. However, $x + x = x - (-x)$ retains the same sign as x even when x is zero.*

This is (hopefully) captured in the following formalization. Let $S = a \pm b$ be the exact result of the operation, and let s_a be the sign of the first operand. The sign bit of the result is defined as

$$sign := \begin{cases} \mathbf{0} & \text{if } S > 0, \\ \mathbf{1} & \text{if } S < 0, \\ s_a & \text{if } S = 0, a = 0, \\ \mathbf{0} & \text{if } S = 0, a \neq 0, \mathcal{M} \neq down, \\ \mathbf{1} & \text{if } S = 0, a \neq 0, \mathcal{M} = down. \end{cases} \tag{4.18}$$

As mentioned above, operations with exact result zero are handled as special cases. As the following theorem asserts, these are exactly the cases where the *rounded* result equals zero:

Theorem 4.23 *Let (s_a, e_a, f_a) and (s_b, e_b, f_b) be representable IEEE factorings with values a and b. Let $\mathcal{M}$ be a rounding mode. It holds*

$$a + b = 0 \iff rd(a + b, \mathcal{M}) = 0.$$

Proof: The proof was developed by Berg, but is not part of his master thesis [Ber01]. We therefore sketch the proof: the $\Rightarrow$ direction follows directly from the definition of the rounding function. For the other direction assume $a + b \neq 0$. Now note that every representable number is an integral multiple of $X_{\min}$, where $X_{\min}$ is the smallest representable number. Hence $a = q_1 \cdot X_{\min}, b = q_2 \cdot X_{\min}$ for

some $q_1, q_2 \in \mathbb{Z}$, and thus $a + b = (q_1 + q_2) \cdot X_{\text{min}} \neq 0$. Hence $|a + b| \geq X_{\text{min}}$. Such numbers cannot be rounded to 0. □

The theorem asserts that the only non-trivial decision on the sign of addition/subtraction results are those where the exact result is zero. Otherwise, the rounded result is not zero, and hence the sign has to be the algebraic sign of the exact result. In the case that the exact result is zero, the sign is computed by circuit ADD-SPECIAL as defined in (4.18).

In [MP00], the above theorem is missing, and hence the argument why the sign is correct is not complete. In fact, the computation of the sign of nonzero results in [MP00, pg. 369] is even wrong, as it is explained in [Ber01, pg. 66]. Another bug in [MP00] is that exactly zero results are not treated as special case, but that a value of zero is fed to the rounder in such cases, although the rounder is not specified for zero inputs, cf. section 4.2.

4.4.4 Putting It All Together

We now combine the unpacker, adder circuits, and the rounder to the complete additive floating point unit.

Let ADD-UNP denote the combination of an unpacker for each of the two operands, and the circuit ADD-SPECIAL. The output of this circuit is either the result of special operations as computed by ADD-SPECIAL, or the unpacked operands as computed by the unpackers.

The core FP-ADDER of the FPU is divided into two stages called ADD-STG1 and ADD-STG2. The remaining stages of the additive FPU are the two rounder stages RD-STG1 and RD-STG2. As with the multiplicative FPU, we now define the combinatorial additive FPU as the circuit ADD-COMB:

Circuit 4.13 (ADD-COMB**)** The combinatorial additive FPU MD-COMB has the following inputs:

- $a, b \in \mathbb{B}^{64}$: the operands are IEEE bitvectors,
- *OVFen*, *UNFen* $\in \mathbb{B}$, $RM \in \mathbb{B}^2$: the exception masks and rounding mode, respectively.
- $opcode \in \mathbb{B}^9$: the operation code. The encoding is listed in appendix A.

The outputs are

- $r \in \mathbb{B}^{64}$: the result is an IEEE bitvector,
- *ovf*, *unf*, *inx*, *inv*, *divz* $\in \mathbb{B}$: the five exception signals.

The functionality is defined as

$$\text{ADD-COMB} := \begin{cases} \text{ADD-UNP} & \text{if special operation} \\ \text{RD-STG2} \circ \text{RD-STG1} \circ \text{ADD-STG2} \circ & \\ \quad \text{ADD-STG1} \circ \text{ADD-UNP} & \text{otherwise} \end{cases}$$

$\diamond$

The following theorem gives the correctness statement for the additive FPU with non-special operands, but potentially exact result zero:

Theorem 4.24 *Let a and b be non-special (potentially zero) operands, let a rounding mode RM and flags OVFen and UNFen be given. Let the operation to be performed be an addition (respectively subtraction). Let $S := [\![a]\!] + [\![b]\!]$ be the exact result (respectively $S := [\![a]\!] - [\![b]\!]$). Let w be the result of the operation as computed by circuit* ADD-COMB*, and let ovf, unf, and inx be the computed exception flags. It holds:*

$$\textit{FPU-result-correct}(S, RM, \textit{OVFen}, \textit{UNFen})(w, \textit{ovf}, \textit{unf}, \textit{inx}),$$

with FPU-result-correct as defined in section 3.7.1.

Proof: Assume first that the operation yields an exact result $S = 0$. Then the circuit ADD-COMB outputs the result as computed by circuit ADD-SPECIAL, which is 0 in this case with all exception signals disabled. By definition of *FPU-result-correct*, the claim of the theorem holds.

Now assume that the operation yields an exact result $S \neq 0$. Then the output of circuit ADD-COMB is computed as RD-STG2 $\circ$ RD-STG1 $\circ$ ADD-STG2 $\circ$ ADD-STG1$\circ$ADD-UNP. By lemma 4.2 the unpacker passes the unpacked operands a and b to stages ADD-STG1 and ADD-STG2. By theorem 4.22, these stages compute a representative of the exact result which satisfies the rounder input conditions. By theorem 4.10, the rounder computes the correctly rounded result and the correct exception flags. $\square$

The *FPU-result-correct*–predicate does not cover the sign of the result w if this result is zero (cf. section 3.7.1). Since the correctness of the sign of the result is a non-trivial statement for addition/subtraction, we have proved a seperate theorem on the sign bit:

Theorem 4.25 *Let a and b be non-special (potentially zero) operands, and let S be the exact result of the addition (or subtraction) of a and b. The sign bit computed by the circuit* ADD-COMB *matches the definition of the correct sign in equation* (4.18).

Proof: If $S = 0$, then this is recognized by the circuit ADD-SPECIAL and the correctly signed result is generated.

Now assume that $S \neq 0$. By theorem 4.23, it holds $rd(S, \mathcal{M}) \neq 0$. This propagates to the potentially wrapped and then rounded result, hence it holds $\textit{result}(x, \mathcal{M}, \textit{OVFen}, \textit{UNFen}) \neq 0$. By theorem 4.24, and by the definition of *FPU-result-correct*, circuit ADD-COMB computes $\textit{result}(x, \mathcal{M}, \textit{OVFen}, \textit{UNFen})$ if no untrapped overflow occurs. Consequently, circuit ADD-COMB computes the correct sign, since the sign is unique for nonzero results. If an untrapped overflow

occurs, MD-COMB outputs either $\pm\infty$ or $\pm X_{\max}$ with the correct sign by definition of *FPU-result-correct*. □

The correctness of the special operand cases is asserted by a theorem for each of the entries in table 4.2. We exemplarily state one of the theorems:

Theorem 4.26 *Let a and b be IEEE bitvectors such that $inf_+(a)$ and $inf_+(b)$ hold. Let the operation be an addition. Then the result w of circuit* ADD-COMB *is $+\infty$, i.e., it holds $inf_+(w)$, and no exception signals are raised.*

Proof: The correctness follows from lemma 4.1 which ensures that the special operands are correctly recognized, and from the construction of circuit ADD-SPECIAL which delivers the result according to table 4.2. □

4.5 Comparison, Conversion and Miscellaneous Operations

In this section we describe the third floating point unit, which we refer to as "Misc-FPU". The Misc-FPU is capable of the following operations:

- comparisons between two floating point numbers of the same format,
- conversion between the two floating point formats,
- conversion of the two floating point formats from/to integer format,
- negation and computation of absolute value,
- and moves between floating point registers, and between floating point and integer registers.

The design of this FPU significantly differs from the design in [MP00], in particular for the conversions with integer destiniation format. We therefore describe the Misc-FPU in more detail than the multiplicative and additive FPUs in the previous sections.

Circuit 4.14 (FP-MISC) The Misc-FPU has the following inputs:

- $a, b \in \mathbb{B}^{64}$: the operands. a is either an IEEE bitvector, or the upper half $a[63:32]$ represents a two's complement integer, depending on the operation. b is always an IEEE bitvector. The b operand is needed only for comparisons.
- *OVFen*, *UNFen* $\in \mathbb{B}$, $RM \in \mathbb{B}^2$: the exception masks and rounding mode, respectively.
- $opcode \in \mathbb{B}^9$: the operation code. The encoding is listed in appendix A.

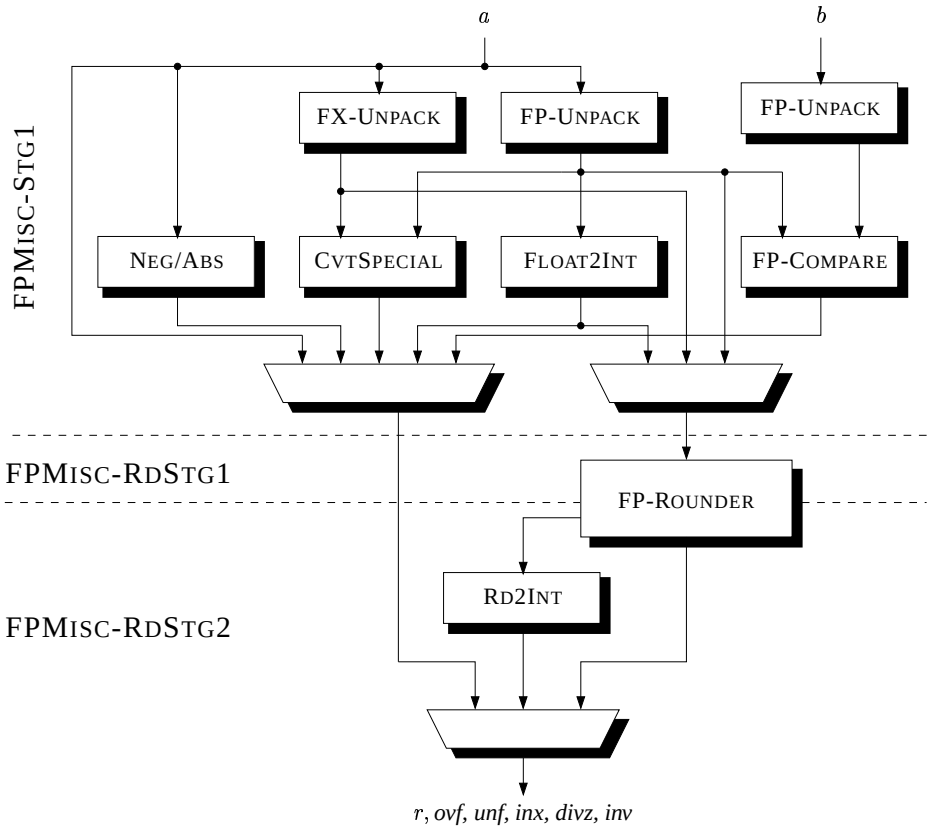


Figure 4.9: Top-level schematics of the Misc-FPU.

The outputs are

- $r \in \mathbb{B}^{64}$: the result; r is, depending in the operation, an IEEE bitvector, or $r[63:32]$ encodes an integer, or $r[32]$ is the comparison result. The embedding of integers and comparison result in the 64-bit bitvector r is arranged to fit with the *VAMP* processor.
- $ovf, unf, inx, inv, divz \in \mathbb{B}$: the five exception signals. In the Misc-FPU, *divz* is never active.

Figure 4.9 shows the top-level schematics of the Misc-FPU. ◇

We now describe informally how the different operations are processed in the Misc-FPU, before we describe the design and verification of some of the operations in more detail.

- *Compares*: for comparisons between two floating point numbers of same format, the two operands are unpacked in the two FP-UNPACK circuits. The numbers are then compared in the circuit FP-COMPARE according to the specification in section 3.7.2.

- *Conversion between floating point formats*: to convert a non-special, nonzero operand a from one floating point format to the other, the operand is unpacked and then fed to the rounder FP-ROUNDER in order to round and IEEE-normalize the result. Furthermore, the rounder computes the flags *ovf*, *unf* and *inx*. Note that for conversions from single to double precision, rounding does not change the value of the operand, but it may be necessary to normalize a single precision de-normal number during conversion in order to yield a double precision IEEE factoring. Vice-versa, for the conversion from double to single precision, the operand is rounded and potentially denormalized in order to fit the smaller exponent range of single precision.

  If the a operand is a special or zero value, the conversion is performed in the circuit CVTSPECIAL.

- *Conversion from integer to floating point format*: the integer operand a is unpacked using the circuit FX-UNPACK (see section 4.1.2). If the integer has zero value, the conversion is performed in circuit CVTSPECIAL. Otherwise, the unpacked integer is fed to the rounder in order to yield the correctly rounded and normalized floating point number. The exception signals are generated as usual.

- *Conversion from floating point to integer format*: if the floating point operand a has a large exponent $e \geq P-1$, or a small exponent $e < 0$, then rounding to integer is easy according to lemmas 3.42 and 3.43. Detection and handling of these cases is performed in circuit FLOAT2INT. In all other cases, the circuit FLOAT2INT computes an input to the rounder. The rounding result is then post-processed in circuit RD2INT in order to yield the correctly rounded integer according to theorem 3.44. This is described in more detail below.

- *Negation and absolute value:* for negation and absolute value computation, the sign bit of the operand a is flipped or tied to **0**, respectively. This is performed in circuit NEG/ABS. Negation and absolute value are not considered to be arithmetic operations, and hence do not signal any exceptions. If applied to special values, the sign is changed as if applied to numbers. This conforms with the standard [IEEE, Appendix].

- *Moves:* In order to allow fast copying of floating point values between different floating point registers, and between floating point and integer registers, the instruction set comprises move instructions. We include these instructions in the Misc-FPU. The operands are simply passed unchanged from the input to the output of the Misc-FPU.

4.5.1 Comparisons

Circuit 4.15 (FP-COMPARE) The circuit FP-COMPARE has the following inputs:

- $s_a, s_b \in \mathbb{B}\,, e_a, e_b \in \mathbb{B}^{\,11}, f_a, f_b \in \mathbb{B}^{\,1+52}$: the unpacked operands,

- $ZERO_a$, $pINF_a$, $nINF_a$, $QNAN_a$, $SNAN_a$, $ZERO_b$, $pINF_b$, $nINF_b$, $QNAN_b$, $SNAN_b \in \mathbb{B}$: the special-operand flags,
- $FCONun, FCONlt, FCONgt, FCONeq \in \mathbb{B}$: the compare-operation control bits as defined in section 3.7.2.

The circuit outputs

- $fcc \in \mathbb{B}$: the comparison result,
- $inv \in \mathbb{B}$: the invalid signal as specified in section 3.7.2.

The construction of circuit FP-COMPARE is straightforward according to lemmas 3.39 and 3.40. We omit the details. ◇

Theorem 4.27 *Given two IEEE bitvectors a and b as inputs, the output of the Misc-FPU performing a comparison satisfies the comparison specification from section 3.7.2, i.e.,*

$$\begin{aligned} r[63:32] &= \mathbf{0}^{31} fcc, \\ ovf, unf, inx, divz &= \mathbf{0}, \\ inv &= FCON\text{-}sig\text{-}unordered(a, b). \end{aligned}$$

Note the embedding convention that the signal fcc is returned as $r[32]$. This is arranged to fit with the VAMP CPU.

Proof: The claim follows from the correctness of the unpackers (lemmas 4.1 and 4.2), and the construction of circuit FP-COMPARE together with lemmas 3.39 and 3.40. □

4.5.2 Conversion to Floating-Point Formats

In order to perform a conversion with a floating point destination format, the operand is unpacked and then fed to the rounder. Unpacking is either performed in the floating or fixed point unpacker, depending to the source format. Special operands are handled in the circuit CVTSPECIAL. The correctness follows from the correctness of the unpackers and the correctness of the rounder for non-special operands, and the correctness of CVTSPECIAL for special operands. The details are tedious and therefore omitted.

There is one non-trivial part in the conversion from double to single precision floating point numbers: the double precision number x to be converted may be so tiny or so large that it does not fit into single precision even after exponent wrapping. Hence, the rounder input condition (4.9) on page 45 might not be satisfied. This does only apply if the corresponding trap is enabled, since otherwise x is rounded to zero or infinity in such cases. The standard requests:

> *Trapped overflow on conversion from a binary floating-point format shall deliver to the trap handler a result in that ... format, possibly with the exponent bias adjusted, but rounded to the destination's precision.*

The case of trapped underflows on conversion is defined analogously in the standard. However, our rounder is not capable of doing so because it cannot round to 24 significant bits within a double precision format. Therefore, on trapped underflow/overflow on conversion, the Misc-FPU delivers to the trap-handler the original operand; the trap-handler can compute the rounded significand in software in these cases.

This is implemented as follows: on conversion from double to single precision, the *OVFen* and *UNFen* inputs to the rounder are tied to **0** in order to disable exponent wrapping. If the rounder signals *OVF* or *UNF*, and originally the corresponding trap was enabled, the original operand is returned instead of the rounded result, and the trap-handler is activated by the CPU.

To the best of our knowledge, this is the only discrepancy of our FPUs to the IEEE standard. Adopting the rounder so that it can handle the described case is probably not too hard.

4.5.3 Conversion to Integer Format

The conversion from floating point to integer format is slightly more complex. For conversion to integer, a correctly rounded integer has to be computed from the floating point operand. The invalid exception is signaled if this is not possible due to overflow or special operands. This is specified in the standard as follows:

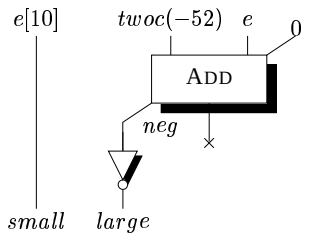
> *The invalid operations are ... conversion of a binary floating-point number to an integer or decimal format when overflow, infinity, or NaN precludes a faithful representation in that format ...*

Hence, the circuit FLOAT2INT signals invalid on infinity and NaN operands, and outputs an unspecified integer in these cases. Assume otherwise that the floating point operand a is non-special. In the following, we treat unpacked single precision operands as if they had double precision. This will not affect the correctness of the conversion from single precision to integer numbers.

We first have to distinguish whether the exponent e is large ($\geq P - 1 = 52$) or small (< 0). This is performed in circuit F2I-DECIDE from figure 4.10. The correctness of this circuit is asserted in the following lemma:

Lemma 4.28 *Let $e \in \mathbb{B}^{11}$ be the exponent in two's complement as delivered by the unpacker. Let $small$ and $large$ be the outputs of circuit* F2I-DECIDE. *It holds:*

$$\begin{aligned} small = \mathbf{1} &\iff [e] < 0, \\ large = \mathbf{1} &\iff [e] \geq 52. \end{aligned}$$



$twoc(-52)$ denotes the 11-bit two's complement representation of -52.

Figure 4.10: Circuit F2I-DECIDE

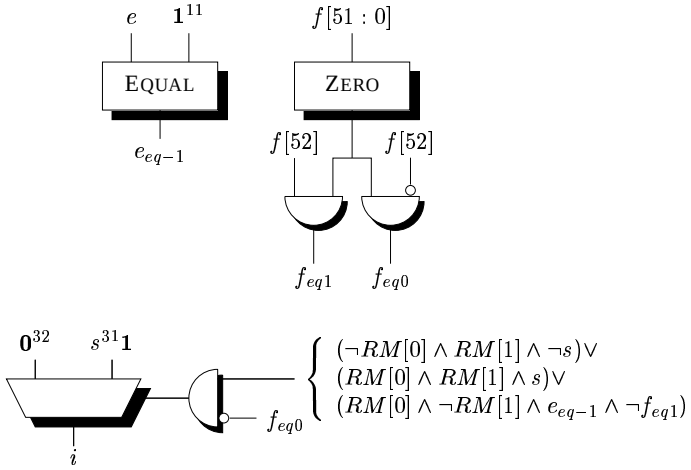


Figure 4.11: Circuit F2I-SMALL

Proof: A two's complement number is negative iff its sign bit is **1**. Hence, $small$ is correct. The neg output of the adder is active iff the sum of $[e]$ and -52 is negative, i.e., iff $[e] < 52$. Hence, $large = \mathbf{1}$ iff $[e] \geq 52$. □

If the exponent e is small, the result can be computed according to lemma 3.43. This is performed in circuit F2I-SMALL in figure 4.11.

Lemma 4.29 *Let a be a non-special operand with sign s and exponent e. Let $[e] < 0$, let $x := [\![a]\!]$ be the value of a, and let $\mathcal{M}$ be a rounding mode with bit-encoding $RM \in \mathbb{B}^2$ (cf. page 44). Let i be the output of circuit* F2I-SMALL *applied to the unpacked operand a. It holds*

$$[i] = [\![\mathit{rd2int}(x, \mathcal{M})]\!].$$

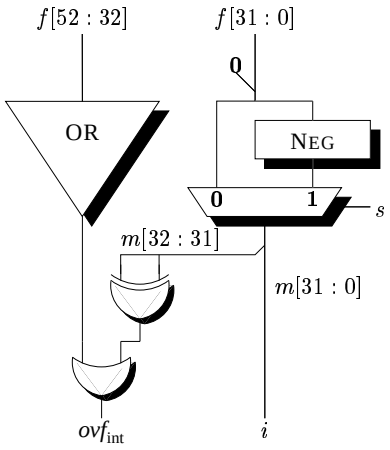


Figure 4.12: Circuit RD2INT

Proof: It holds

$$\begin{aligned} e_{eq-1} = \mathbf{1} &\iff [e] = -1, \\ f_{eq1} = \mathbf{1} &\iff \langle f \rangle \cdot 2^{-52} = 1, \\ f_{eq0} = \mathbf{1} &\iff \langle f \rangle \cdot 2^{-52} = 0. \end{aligned}$$

The correctness now follows directly from lemma 3.43 and the bit-encoding of rounding modes (see (4.7) on page 44). □

If the exponent e is large, i.e., $[e] \geq 52$, it holds $[\![a]\!] \geq 2^{52}$, which is outside the range of 32-bit integers. In this case, a non-specified integer is returned, and the *inv* signal is activated according to the IEEE standard.

If the exponent e is mid-range, i.e., $0 < [e] < 52$, the conversion is performed according to theorem 3.44: the operand is multiplied with $2^{e_{\min}+1-P}$, then rounded, and finally the integer is extracted from the rounding result.

Multiplication of the operand with $2^{e_{\min}+1-P}$ is performed by adding $e_{\min} + 1 - P = -1074$ to e before feeding the operand to the rounder. The needed adder is incorporarted into circuit FLOAT2INT.

The rounding is performed in the standard rounder FP-ROUNDER with disabled exception masks *OVFen* and *UNFen*. The final integer result is extracted in circuit RD2INT. In order to allow the easy extraction of the integer result, the circuit RD2INT gets as input not the packed rounding result, but the intermediate rounded result as computed by the ADJUSTEXP stage right before packing in the rounder (see section 4.2). The construction of circuit RD2INT is shown in figure 4.12.

Lemma 4.30 *Let* $f \in \mathbb{B}^{53}, s \in \mathbb{B}$ *be inputs to the circuit* RD2INT, *and* $ovf_{\text{int}} \in$

$\mathbb{B}, i \in \mathbb{B}^{32}$ *be its outputs. Let* $x := (-1)^s \cdot \langle f \rangle$. *It holds:*

$$\begin{aligned} x \in T_{32} \quad &\Longrightarrow \quad [i] = x, \\ \mathit{ovf}_{\text{int}} \quad &\Longleftrightarrow \quad x \notin T_{32}, \end{aligned}$$

where $T_n = \{-2^{n-1}, \ldots, 2^{n-1} - 1\}$ *is defined in section 2.1 as the range of* n*-bit two's complement numbers.*

Proof: If one of the bits $f[52:32]$ is $\mathbf{1}$, x is not in the range T_{32} and hence $\mathit{ovf}_{\text{int}}$ is asserted. Otherwise, $x = (-1)^s \cdot \langle f[31:0] \rangle$. The bits $f[31:0]$ are extended by a $\mathbf{0}$ in order to yield a two's complement representation. This is fed to a negater NEG which computes $-\langle f[31:0] \rangle$. The result is multiplexed with the non-negated f yielding $m \in \mathbb{B}^{33}$ with $[m] = x$. This 33-bit two's complement bitvector is in 32-bit range iff $m[32] = m[31]$; in this case, it holds $[i] = [m[31:0]] = x$. Otherwise, x is not in the correct range, and hence $\mathit{ovf}_{\text{int}}$ is asserted. □

If the RD2INT circuit asserts the $\mathit{ovf}_{\text{int}}$ signal, then x lies outside the two's complement-representable range an hence the *inv* exception is signaled as mandated by the standard.

The following theorem asserts the overall correctness of the convert to integer circuits:

Theorem 4.31 *Let* a *be an IEEE bitvector and* $\mathcal{M}$ *be a rounding mode with encoding* RM*, let the circuit* FP-MISC *perform a conversion from floating point to integer format. Let* r *be the output of this operation, and* $\mathit{ovf}, \mathit{unf}, \mathit{inx}, \mathit{inv}, \mathit{divz}$ *be the computed exception signals. In the case that* a *is non-special, let* $I := [\![\mathit{rd2int}([\![a]\!], \mathcal{M})]\!]$ *be the correctly rounded integer. It holds*

$$\begin{aligned} [r[63:32]] \quad &= \quad I \text{ if } a \text{ is non-special and } I \in T_{32}, \\ \mathit{ovf}, \mathit{unf}, \mathit{inx}, \mathit{divz} \quad &= \quad \mathbf{0}, \\ \mathit{inv} = \mathbf{1} \quad &\Longleftrightarrow \quad a \text{ is infinity or NaN, or } I \notin T_{32} \end{aligned}$$

Note the embedding convention that the 32-bit integer result is encoded in the bits $r[63:32]$*. This is arranged to fit with the VAMP CPU.*

Proof: If a is special, then *inv* is signaled by construction of circuit FLOAT2INT. Otherwise, if a has a small or a large exponent, this is correctly detected according to lemma 4.28. Lemma 4.29 asserts that the case of small exponents is correctly processed. In the case of a large exponent, FLOAT2INT correctly signals *inv* by construction.

It remains the case of operands with exponent between 0 and $P - 1$. These operands are multiplied by $2^{e_{\min}+1-P}$, i.e., their exponent is decreased by 1074. The operand is then fed to the rounder. By theorem 4.9, the output (s_r, e_r, f_r) of the ADJUSTEXP stage satisfies[2]

$$[\![s_r, [e_r]_{\text{bias}}, \langle f_r \rangle \cdot 2^{-52}]\!] \quad = \quad \eta(rd(\mathit{wrapped}_{\text{bef}}([\![a]\!], \mathit{OVFen}, \mathit{UNFen}), \mathcal{M})),$$

and since both *OVF* and *UNF* exceptions are disabled, it holds

$$[\![s_r, [e_r]_{\text{bias}}, \langle f_r \rangle \cdot 2^{-52}]\!] \quad = \quad \eta(rd([\![a]\!], \mathcal{M})).$$

By theorem 3.44, this implies

$$I \quad = \quad (-1)^{s_r} \cdot \langle f_r \rangle.$$

By lemma 4.30, the circuit RD2INT correctly computes I from s_r and f_r if I is in-range, and signals ovf_{int} otherwise. In this case FP-MISC signals *inv*. □

4.6 Discrepancies to the IEEE Standard

As described in section 4.5.2, the Misc-FPU handles trapped overflows and underflows on conversion from double to single precision different than 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 [IEEE, Sect. 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 some 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 that the former two operations could be designed and verified with small effort given the experience and techniques presented in this chapter. Conversion between floating point and decimal formats might be slightly more complex [Coo80, Cli90]. All three operations raise an unimplemented-trap in the *VAMP* CPU and may be implemented in a trap handler.

4.7 Related Work

Aagaard and Seger combine BDD based methods and theorem proving techniques to verify a floating point multiplier [AS95]. Chen and Bryant [CB98] use word-level model checking to verify a floating point adder. Exceptions and denormals are not handled in both verification projects.

Verkest et al. verify a non-restoring integer division algorithm [VCDM94]. Clarke et al. [CGZ96] and Ruess et al. [RSS96] verify SRT division algorithms. Miner and Leathrum [ML96] verify a general class of subtractive division algorithms with respect to the IEEE formalization of Miner [Min95]. Mechanized proofs of SRT integer division are reported in [Bry96, KS97].

In [Har97], Harrison proves the correctness of an algorithm for the exponential function against his IEEE formalization. He assumes that IEEE correct addi-

[2] Theorem 4.9 asserts the correctness of the outputs of the POSTNORM stage. In the PVS proof, there is a similar theorem for the ADJUSTEXP stage.

tion, multiplication, and rounding to integer are provided. In [AHTH01, AH01a], Abdel-Hamid et al. verify an implementation of this algorithm against a formal specification. However, there is a large gap between their specification and the IEEE standard.

O'Leary et al. [OZGS99] 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. This will be described in more detail in section 6.5.2.

In [AJK00], 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 [KK01], 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.

Cornea-Hasegan [CH98, CH99] 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*, an computer algebra system. Computer algebra systems are usually not considered to be formal verification tools [ADG+01].

Moore et al. have verified the AMD K5 division algorithm [MLK98] with the theorem prover ACL2. Russinoff has verified the K5 square root algorithm as well as the AMD Athlon multiplication, division, square root, and addition algorithms [Rus98, Rus99, Rus00]. In all his verification projects, Russinoff proves the correctness of a register transfer level implementation against his formalization of the IEEE standard using ACL2. Russinoff does not handle exceptions and denormals in his publications; 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, cf. section 6.5.2.

In [CCH+96], 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[3].

The bug has escaped the verification in [CCH+96] because Chen et al. did not formally compose all parts of the system [OZGS99]. It is therefore comparable to

[3] see `http://support.intel.com/support/processors/flag/tech.htm`

the Müller/Paul division bug described in section 4.3.1, which is also due to not "putting it all together" in a formal way.

Summarizing, our work is the first formal verification of a complete floating point unit with the supported operations on the gate-level against a direct formalization of the IEEE standard. In particular, our work includes denormal operands and results, and the correct computation of the exception signals as an integral part of the floating point unit.

Chapter 5

Pipelining the FPUs

In this chapter we describe how the floating point units presented in the previous chapter are pipelined in order to work as execution units in the *VAMP* processor. In order to exploit the benefits of the out-of-order Tomasulo scheduler [Tom67] used in the *VAMP* processor, the FPU execution units may process multiple instructions simultaneously, may have branches and cycles in the pipeline structure (e.g., for special cases and the division algorithm), may have variable latency, and may reorder instructions internally, i.e., instructions do not need to leave the pipeline in the order they entered it.

We describe a general methodology for the verification of pipelined execution units with these features. As an example we describe the verification of our multiplicative FPU. Its pipeline can process up to six instructions simultaneously. The difficulty in the verification of such complex pipelines arises from the fact that pipelines consist of a control-dominated part which schedules the processing of the instructions in the pipeline, while simultaneously the effect of the datapaths on the data of each instruction has to be considered in order to guarantee *functional* correct behavior of the execution unit.

The sole use of theorem proving for the verification of complex pipelines would involve the construction of an inductive invariant to cope with the control-dominated part. The construction usually has to be performed manually, which is considered the hard part of the verification of out-of-order systems [HGS00, SH98, Kro01]. On the other hand, model checking is suitable for the automatic verification of control-dominated systems, but becomes infeasible for the verification of complete pipelines due to the data part. Even if one uses abstract datapaths, e.g., uninterpreted functions [BD94], the state space grows huge due to the large number of (nested) function applications (e.g., due to possible cycles in the pipeline structure).

Our methodology combines the best of both worlds: we use the PVS built-in model-checker [RSS95] to verify the control part of the pipelines, and then use theorem proving to conclude overall correctness, including data correctness.

In order to use model-checked properties for the further verification by theorem

proving, the model-checked properties have to be translated into a form which is easy to use for theorem proving. In PVS, the FairCTL operators are defined as fixpoints in μ-calculus, which in turn are defined in terms of higher-order logic. These definitions are hard to use in theorem proving. It is more suitable for theorem proving to define computation traces explicitly, and to express temporal properties using standard mathematical quantifiers, e.g., $\forall t\colon p(t)$ to express a property p to hold for all times t along a computation trace. In order to translate model-checked properties safely from FairCTL to $\forall t$ form, we have proved theorems which relate the FairCTL operators defined in μ-calculus with their intended semantics expressed in $\forall t$ form. These relations are well known [CGP99], but have not been verified using formal methods before.

This chapter is structured as follows. In section 5.1 we formally define the correctness criterion which our execution units shall obey. In section 5.2 we exemplarily sketch the pipeline design of our multiplicative FPU, which is our most complex execution unit.

In section 5.3 we describe a failed approach to the verification of complex pipelines based on using solely theorem proving. Contrary, we describe the verification of the pipelining using solely model checking in section 5.4. This approach failed as well.

In section 5.5 we prove theorems which allow the safe translation of model-checked properties to $\forall t$ form. This is used in section 5.6 to combine model checking and theorem proving for the verification of the pipelines. In section 5.7, the new methodology is applied to the pipelines of our FPUs. We discuss related work in section 5.8.

In [MP00], the FPU is integrated into an in-order variant of the DLX-processor. In our work, the FPUs are integrated into the out-of-order *VAMP* processor. 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. Hence, the work presented in this chapter does not base on [MP00].

This chapter is an extended version of [Jac02].

5.1 Pipeline Correctness Criterion

In this section we describe the correctness criterions which our execution units (EU, also called simply *pipelines* in this thesis) shall obey. An execution unit can be seen as a black box with inputs and outputs interconnecting the EU with the Tomasulo scheduled *VAMP* processor core. The core *dispatches* instructions by passing the instruction data (operands, op-code, etc.) to the EU along with a tag used to identify the instruction. The EU executes the instruction and returns the result with the corresponding tag to the core. The EU may process several instructions simultaneously, instructions may have variable latency, and the EU may reorder instructions internally, i.e., instructions do not need to leave the pipeline

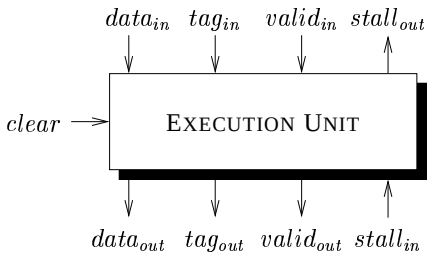


Figure 5.1: Execution unit interface

in the order they have entered it. The *VAMP* processor core can cope with these possibilities.

The Tomasulo scheduler only dispatches instructions whose operands are available. Therefore, the pipelines do not have to cope with *data hazards*. The only hazards occurring in the pipelines are *structural hazards*, i.e., multiple instructions requiring the same resources in the pipeline. All others hazards are dealt with in the processor core.

Figure 5.1 shows a black-box view of an execution unit. The $clear$ input is activated at power-up and during interrupts in order to clear the pipeline. Instructions are dispatched into the EU by activating the $valid_{in}$ signal along with the instruction's $data_{in}$[1]and tag_{in}. The EU then computes the result and returns it by activating $valid_{out}$ along with the proper $data_{out}$ and tag_{out}. The $stall_{out}$ signal is activated if the EU cannot take further instructions; in this case, the scheduler must not dispatch instructions. Analogously, if the core activates the $stall_{in}$ signal, the EU must not return any instructions.

In the following, we ignore the $clear$ signal since the implementation and verification of $clear$ is simple.

5.1.1 Formalization of the EU Interface

Let $\mathcal{S}$ denote the state set of the EU (usually the set of possible contents of the registers within the EU). Let $\mathcal{D}_i$, $\mathcal{D}_o$, and $\mathcal{T}$ denote the set of the input data, output data, and tags, respectively. The $valid$ and $stall$ signals are booleans. The EU is specified by the following five functions:

1. $ns(S_{cur}, data_{in}, tag_{in}, valid_{in}, stall_{in}) \rightarrow \mathcal{S}$: the next-state function; it computes the next state given the current state S_{cur} and the current inputs.
2. $data_{out}(S_{cur}, data_{in}, valid_{in}, stall_{in}) \rightarrow \mathcal{D}_o$: computes the data output of the EU given current state and inputs.
3. $tag_{out}(S_{cur}, tag_{in}, valid_{in}, stall_{in}) \rightarrow \mathcal{T}$: computes the output-tag.

[1]In this chapter, $data$ is always meant to be the inputs of the combinatorial circuit needed to execute the instruction. In our case of FPUs, this includes op-codes, rounding-mode, flags, and operands, i.e., the inputs of the combinatorial FPUs from chapter 4.

4. $valid_{out}(S_{cur}, valid_{in}, stall_{in}) \rightarrow \mathbb{B}$: computes the valid output.

5. $stall_{out}(S_{cur}, stall_{in}) \rightarrow \mathbb{B}$: computes the stall output.

The functions $data_{out}$, tag_{out}, $valid_{out}$, and $stall_{out}$ model the combinatorial circuits which compute the corresponding outputs from the (registered) state and the current inputs. Note that not all outputs may depend on all inputs. This is necessary to model absence of *combinatorial* dependencies between some inputs and outputs. For example, $stall_{out}$ only depends on the state and the current $stall_{in}$, i.e., whether the EU accepts a further instruction may not depend on the instruction data or tag.

Let $\mathcal{I} := \mathcal{D}_i \times \mathcal{T} \times \mathbb{B} \times \mathbb{B}$ denote the combination of the inputs of the EU. We recursively define the behavior of a pipeline under an infinite input sequence $I := (i_0, i_1, \ldots) \in \mathcal{I}^\infty$. We assume the pipeline to be in some initial state $init \in \mathcal{S}$ at time $t = 0$. The state $s^t(I)$ at time t is recursively defined as

$$\begin{aligned} s^0(I) &:= init, \\ s^{t+1}(I) &:= ns(s^t(I), i_t). \end{aligned}$$

We define $data^t_{out}(I)$, $tag^t_{out}(I)$, $valid^t_{out}(I)$, and $stall^t_{out}(I)$ to be the outputs of the pipeline during cycle t, e.g.,

$$stall^t_{out}(I) := stall_{out}(s^t(I), i_t.stall_{in}).$$

For the sake of convenience, we omit the parameter I if it is clear from the context.

We say a tag $tg \in \mathcal{T}$ is dispatched at time t (denoted by $disp(tg, t)$), if $valid^t_{in}$ and $tag^t_{in} = tg$ hold. The tag is returned at time t (denoted by $ret(tg, t)$), if $valid^t_{out}$ and $tag^t_{out} = tg$ hold. The tag is in use at time t (denoted by $inuse(tg, t)$), if the tag was dispatched and not yet returned, i.e.,

$$inuse(tg, t) := \exists t' < t \colon disp(tg, t') \text{ and } \forall t'' \in \{t', \ldots, t-1\} \colon \neg ret(tg, t'').$$

5.1.2 Correctness Criterion

We can now define the correctness criterions for execution units. First, if $stall_{in}$ is active, $valid_{out}$ may not be signaled:

$$\forall t \colon stall^t_{in} \implies \neg valid^t_{out}. \tag{P1}$$

The $stall_{out}$ signal is live, i.e., at each point in time t, it will eventually become inactive (at time t'):

$$\forall t \colon \exists t' \geq t \colon \neg stall^{t'}_{out}. \tag{P2}$$

Instructions dispatched into the EU at time t will eventually be returned (at time t'). We call this property *liveness* of the EU:

$$\forall t \colon disp(tg, t) \implies \exists t' \geq t \colon ret(tg, t'). \tag{P3}$$

The last property, called *tag-consistency*, requires that instructions returned at time t by the EU have been dispatched before (at time t'), and have not already been returned in between (at time t''):

$$\forall t\colon ret(tg,t) \implies \exists t' \leq t\colon disp(tg,t') \text{ and } \forall t'' \in \{t',\ldots,t-1\}\colon \neg ret(tg,t''). \quad \text{(P4)}$$

Note that the right side of the above definition does nearly but not exactly match $inuse(tg,t)$, since here $t' = t$ is allowed in contrast to the $inuse$ definition. However, it is sufficient to prove $\forall t\colon ret(tg,t) \implies inuse(tg,t)$ in order to assert tag-consistency. Note further that liveness and consistency together yield a one-to-one mapping between dispatched and returned instructions.

Of course the execution unit cannot satisfy these properties if the inputs do not satisfy some properties themself. The first required input property is that no instruction is dispatched if the $stall_{out}$ is active, analogously to (P1):

$$\forall t\colon stall^t_{out} \implies \neg valid^t_{in}. \quad \text{(I1)}$$

The analogue to (P2) is that the $stall_{in}$ signal is live:

$$\forall t\colon \exists t' \geq t\colon \neg stall^{t'}_{in}. \quad \text{(I2)}$$

The third input property is called *tag-uniqueness* and requires that no tag tg is dispatched into the EU if it is already in use:

$$\forall t\colon disp(tg,t) \implies \neg inuse(tg,t). \quad \text{(I3)}$$

We call an execution unit correct iff for all input sequences I and tags tg the properties (P1) to (P4) hold under the assumptions (I1) to (I3), where not all properties need all assumptions:

$$\begin{aligned} EUcorrect \;:=\; & (I1) \implies (P1) \text{ and} \\ & (I1) \wedge (I2) \implies (P2) \wedge (P3) \text{ and} \\ & (I1) \wedge (I2) \wedge (I3) \implies (P4). \end{aligned} \quad \text{(C)}$$

This definition of correctness only covers the correct termination of instructions. In order to cover the input/output data relation, we introduce the notion of *functional correct execution units*. An EU is called *functional correct* with respect to a function $dp : \mathcal{D}_i \to \mathcal{D}_o$, iff $dp(data_{in}) = data_{out}$ holds for corresponding inputs and outputs. The function dp is the function computed by the combinatorial datapaths. The pipelined hardware shall compute this function. In order to

model functional correctness, we strengthen the liveness property (P3) to cover the relation between data input and output of an instruction:

$$\forall t\colon disp(tg,t) \implies \big(\exists t' \geq t\colon ret(tg,t') \text{ and } dp(data^t_{in}) = data^{t'}_{out}\big). \quad \text{(P3}')$$

Formally, we call an execution unit *functional correct* with respect to dp iff (C) holds where (P3) is replaced by (P3′).

Note that the definition of (functional) correctness allows multiple instructions (with distinct tags) in the EU simultaneously, and that no restriction on the order in which instructions leave the EU is imposed. Note further that not all EUs have a functional description; a memory unit, e.g., cannot be described by a function dp, since functions are by definition memory-less.

The correctness criterions of the EUs have been arranged with Kröning in order to allow the integration of our EUs into Kröning's Tomasulo core [Kro01].

5.2 Example Pipeline

In section 4.3 we have verified the combinatorial correctness of the multiplicative FPU with respect to the IEEE standard. Here we describe the pipelining of this FPU as an example. Figure 5.2 shows the structure of the pipeline. The pipeline stages correspond to the sub-circuits of circuit MD-COMB as defined in section 4.3.

We briefly recap the multiplicative FPU from the perspective of the pipeline: the first pipeline stage performs unpacking of floating point operands, handles special cases, and performs initial approximation lookup in case of divisions. The next two stages comprise a pipelined multiplier. For divisions, the instructions have to iterate through these stages six or eight times, depending on the precision of the floating point operation. The SELECTFD stage computes the representative of the quotient, multiplications skip this stage. Finally, the results are rounded by the two-stage rounder. Special cases do not flow through the pipeline, but are bypassed from the unpacker to the output.

Out-of-order completion in this pipeline can occur in various ways: for example, an operation involving special cases is bypassed to the output while other operations are still in the pipeline. Other examples are a multiplication which overtakes a division that iterates through the multiplier stages, or a single precision division which overtakes a double precision division.

The functional behavior of the FPU pipeline is prescribed by the combinatorial circuit MD-COMB. In section 4.3, MD-COMB is composed from sub-circuits corresponding to the datapaths of the individual pipeline stages. For the verification of the pipeline, the actual implementation of these datapaths is not important, i.e., can be left uninterpreted (in the sense of uninterpreted functions [BD94]). We only have to prove that instructions take the correct path through the pipeline. Then, by definition of MD-COMB, the data output of the pipeline equals the data output of

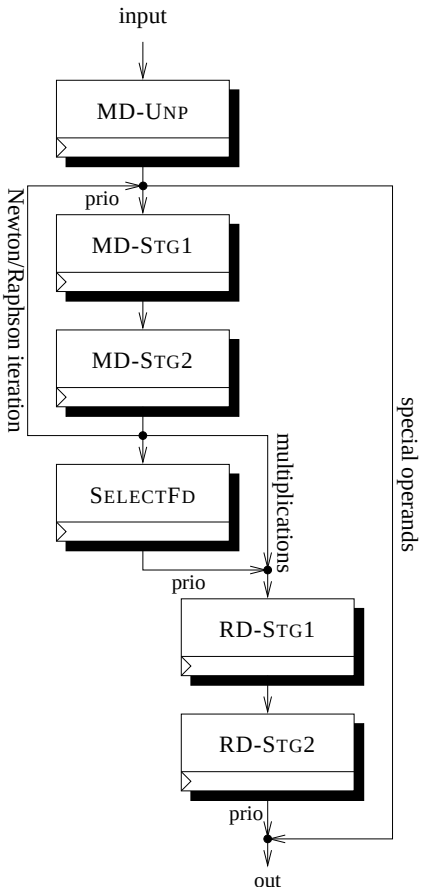


Figure 5.2: FPU pipeline

the MD-COMB circuit, as it is required for functional correctness of the execution unit.

Design of the Pipeline Control.

In the following, we briefly describe the idea behind the construction of the pipeline control. Instructions flow through the pipeline along with their associated control information. The control information consists of a valid bit, the instruction tag, and some auxiliary control data. The auxiliary data is used, e.g., to distinguish multiplications and divisions, and to count the number of remaining iterations for divisions.

Each register stage in the pipeline can hold one instruction along with its control information. If the valid bit of a register stage is inactive, the stage is empty, i.e., the instruction in this stage is void. Assume that register stage R contains a valid instruction I. There may be several possible stages to which instruction I has to be fed in the next step. For example, the next stage of a valid instruction in

stage MD-STG2 of the multiplicative FPU pipeline may be either MD-STG1, SELECTFD, or RD-STG1. The correct next stage can be decided from the auxiliary control data.

Each register stage R is assigned a clock-enable signal ce_R which controls the clocking of the stage. The stage is clocked whenever possible without loosing an instruction. More precisely, R is clocked whenever R is empty, or R is full and the valid instruction I currently in R can be fed to its next stage R'. The instruction I can be fed to R', if R' is clocked itself, and no other instruction with higher priority simultaneously aims for R'.

For example, if the MD-STG2 stage contains a multiplication, and the SELECTFD stage contains a division, both instructions aim for the RD-STG1 stage. The division is statically prioritized in our example (see below). Assume that the RD-STG1 is being clocked in the next cycle. Then the SELECTFD stage may be clocked, too, since its instruction is fed to the RD-STG1. The MD-STG2 stage, however, may not be clocked, since the multiplication would otherwise be overwritten from MD-STG1, and hence the multiplication would be lost. Precisely, the clock-enable for stage MD-STG2 is defined as

$$
\begin{array}{lcl}
ce_{\text{MD-STG2}} & := & \neg valid_{\text{MD-STG2}} \\
 & & \text{the MD-STG2 stage is empty, hence no instruction is lost if MD-STG2 is clocked;} \\
 & \vee & (\text{MD2-Nxt} = \text{SELECTFD} \wedge ce_{\text{SELECTFD}}) \\
 & & \text{the instruction currently in MD-STG2 has SELECTFD as next stage, and SELECTFD will be clocked, i.e., SELECTFD will accept the instruction;} \\
 & \vee & (\text{MD2-Nxt} = \text{RD-STG1} \wedge ce_{\text{RD-STG1}} \wedge \neg valid_{\text{SELECTFD}}) \\
 & & \text{the instruction in MD-STG2 has RD-STG1 as next stage, and RD-STG1 will be clocked. Additionally, the SELECTFD stage may not contain a valid instruction, since otherwise this instruction would have priority for the RD-STG1 stage;} \\
 & \vee & (\text{MD2-Nxt} = \text{MD-STG1}) \\
 & & \text{the instruction in MD-STG2 has MD-STG1 as destination, i.e., has to be fed back. In our multiplicative pipeline, it is always ensured that MD-STG1 can accept the instruction from MD-STG2. This is because a valid instruction in MD-STG1 can always proceed to MD-STG2 if MD-STG2 is being fed back.}
\end{array}
$$

Note that the register stage is clocked even if no instructions pass through the stage, since the stage is always empty is this case.

As mentioned above, our concrete pipeline statically prioritizes the longer path, i.e., divisions in the stage SELECTFD have priority over multiplications in the MD-STG2. That the static prioritization is fair involves the following tricky argument: a multiplication in the MD-STG2 can only be postponed by a division in the SELECTFD stage. This division will eventually proceed to the RD-STG1. From then on the SELECTFD stage is empty, and no new division can reach SELECTFD

until the multiplication in MD-STG2 proceeds to RD-STG2, since the new division would have to pass the occupied MD-STG2. Hence, the multiplication is not stalled infinitely.

This argument would be hard to verify by theorem proving. In our combined approach of model-checking and theorem-proving, the fairness problem is automatically resolved by model-checking (cf. equations (5.1) and (5.2)).

One could also use other prioritization schemes, as long as fairness is guaranteed. For example, we have also designed a pipeline where fair arbiters are used to schedule such conflicts. For the FPU pipelines, however, static prioritization of the longer pipeline paths is preferable, since thereby the older instructions receive higher priority.

We omit the details of the complete construction of the pipeline control because they are too lengthy. We lack a formalism which allows the concise and mathematically rigorous presentation of such a pipeline control. Such a formalism is presented in [MP00, JK00, Kro01] for in-order pipelines without branches and cycles in the pipeline structure. We believe that one could extend this formalism to cope with complex out-of-order pipelines. However, this is considered future work, and is beyond the scope of this thesis.

It is worth mentioning that we have *designed* the pipeline control in the model-checker SMV [McM93]. The design of the pipeline control took about one week which also included verification and debugging using SMV. We believe that *simultaneously designing and verifying* the control significantly helped in designing such a complex pipeline with relatively small effort.We have used SMV instead of the PVS built-in model-checker, since SMV—in contrast to PVS—is capable of constructing counter-examples if it encounters a bug. This is of immeasurable value for the debugging of the pipelines. The SMV description of the multiplicative pipeline control is listed in Appendix D.

5.3 Pipeline Verification by Theorem Proving

Our initial approach to the verification of complex pipelines as the one described in the previous section was to decompose the pipeline into smaller segments, prove each segment to be functional correct, and then to re-compose and conclude functional correctness of the complete pipeline. For this, we have verified a library of functional correct basic segments, and of theorems allowing the composition of such segments.

As primitives, we have verified that *combinatorial circuits* and *single registers* (without assigned datapaths) are functional correct segments. We then proved a composition theorem stating that two arbitrary functional correct segments (with respect to functions dp_1 and dp_2) may be concatenated yielding a larger functional correct segment (with respect to function $dp_2 \circ dp_1$).

From these primitives and the concatenation theorem, it is possible to build simple sequential pipelines of arbitrary depth. The verification of combinatorial

circuits and single registers was relatively simple. However, the verification of the concatenation theorem was considerably complex and took about 2 weeks.

In order to build a pipeline as complex as described in the previous section we need further primitives: we have verified a *splitter* which splits the pipeline into two paths, e.g., for bypassing special results. The verification of this splitter took another 2 weeks. The last primitive we need is an *iterator* for the cycle in the pipeline. Obviously, this is the most complex primitive. For example, in order to prove liveness of the $stall_{out}$ signal of the iterator, one would have to prove that the pipeline inside the iterator would drain empty eventually if no new instructions enter the pipeline. We have thought about how the proof had to be structured for the verification in PVS, but have not tried to prove it in PVS due to the complexity. Instead, we have developed our approach to the verification of pipelines described in the following sections.

Together, the primitives we have verified needed more than 1000 proof commands and took more than a month of proof development. The difficulty in the verification arose from the irregularity of the arguments, and from the need to manually construct inductive invariants. These invariants have to be much stronger than the actually needed invariants, and therefore are hard to find. Finding inductive invariants is considered to be the hard part of the verification of out-of-order systems [HGS00, SH98, Kro01].

5.4 Pipeline Verification by Model Checking

We have modeled the pipeline of the FPU in the model-checker SMV. The datapaths have been abstracted using uninterpreted functions [BD94]. Given an uninterpreted function $f : D \rightarrow C$ modeling a pipeline stage, SMV verifies the specified properties of the pipeline for *all* functions $f' : D \rightarrow C$ with the domain and codomain of f. Datatype and symmetry reduction [ID96, McM00] are used in order to reduce the state space and the number of functions f' which have to be verified separately. However, in the case of our FPU, the symmetry is small due to the number of function applications, in particular because of nested function applications for the cycle in the pipeline. This results in a very large number of different cases, and each case has a large state-space.

We have tried to verify the pipeline including abstracted datapaths. We gave up when a run-time of 4 days and a memory usage of more than 1GB was reached. The SMV code is available at our web-page.

5.5 Translating FairCTL to $\forall t$ form

Since the verification of the functional correctness of the pipelines failed using solely theorem proving or model checking, we tried to combine both techniques.

Our goal is to use the PVS built-in model-checker for the verification of temporal properties of the pipeline control, and then to use the theorem prover to conclude

overall correctness of the pipeline, including the datapaths. In PVS, the FairCTL operators are defined as fixpoint in μ-calculus [RSS95], whereas we have used temporal properties in $\forall t$ form in section 5.1 to define pipeline correctness. We believe that temporal properties expressed in $\forall t$ form are more suitable for theorem proving.

In order to transform model-checked statements from FairCTL to $\forall t$ form, we formally verify that the FairCTL operators defined as fixpoints in μ-calculus match their intended semantics expressed in $\forall t$ form. These theorems have first been proved in [EC80] and are well known. However, they have not been verified using formal methods, which is necessary to transform between μ-calculus and $\forall t$ form in a formally safe way. For the formal verification in PVS, the "paper & pencil" proofs from [CGP99] served as guidelines. The formal verification depends on the definition of fixpoints and FairCTL operators in PVS [RSS95].

In this section, systems are described by a state set $\mathcal{S}$ and a total next-state *relation* $N \subseteq \mathcal{S} \times \mathcal{S}$ which models a non-deterministic choice of the next state. In contrast, in section 5.1 systems were modeled by a next state *function* which deterministically computes the next state from the current state and inputs. It is easy to transform between deterministic systems with inputs, and non-deterministic systems without inputs by "simulating" inputs by non-deterministic choice and vice versa. We come back to this difference in section 5.5.4.

5.5.1 Fixpoints

Let $2^{\mathcal{S}}$ denote the set of monadic predicates. Let $pp : 2^{\mathcal{S}} \rightarrow 2^{\mathcal{S}}$ be a so-called *predicate transformer*. The predicate transformer pp is called monotone, if

$$\forall Q, Q' \in 2^{\mathcal{S}} : Q \subseteq Q' \implies pp(Q) \subseteq pp(Q').$$

A predicate $Q \in 2^{\mathcal{S}}$ is called a fixpoint of pp iff $Q = pp(Q)$. A predicate Q is called the least fixpoint of pp iff for all fixpoints Q' holds: $Q \subseteq Q'$. A predicate Q is called the greatest fixpoint of pp iff for all fixpoints Q' holds: $Q \supseteq Q'$.

In PVS, operators $\mu(pp)$ and $\nu(pp)$ are defined to compute the least and greatest fixpoints of pp, respectively. Both operators are defined in terms of higher-order logic:

$$\mu(pp) := \{s \in \mathcal{S} \mid \forall Q \in 2^{\mathcal{S}} : pp(Q) \subseteq Q \implies Q(s)\},$$
$$\nu(pp) := \{s \in \mathcal{S} \mid \exists Q \in 2^{\mathcal{S}} : Q \subseteq pp(Q) \wedge Q(s)\}.$$

Intuitively, an element $s \in \mathcal{S}$ is in the least fixpoint $\mu(pp)$, if s is in all predicates Q which are "lessened" by the predicate transformer pp. The greatest fixpoint operator has an analogous intuition. The correctness of these definitions is asserted in the following theorem:

Theorem 5.1 *Let $pp : 2^{\mathcal{S}} \rightarrow 2^{\mathcal{S}}$ be a monotonic predicate transformer. Then $\mu(pp)$ is the least fixpoint, and $\nu(pp)$ is the greatest fixpoint of pp.*

The theorem has been verified in PVS in [RSS95], we therefore omit the proof.

5.5.2 The FairCTL Operators

Let $N \subseteq \mathcal{S} \times \mathcal{S}$ be a total next-state relation. An *N-path* is an infinite sequence $(p_0, p_1 \ldots) \in \mathcal{S}^\infty$ where successive states respect the next-state relation, i.e., $\forall t\colon N(p_t, p_{t+1})$ holds.

Let $f, g, \mathit{fair} \in 2^\mathcal{S}$ be predicates. The basic FairCTL operators are defined in terms of fixpoints [EC80]:

$$\begin{aligned}
\mathbf{EX}(N, f) &:= \{s \in \mathcal{S} \mid \exists s' \in \mathcal{S}\colon f(s') \wedge N(s, s')\},\\
\mathbf{EG}(N, f) &:= \nu(\lambda Q \in 2^\mathcal{S}\colon f \wedge \mathbf{EX}(N, Q)),\\
\mathbf{EU}(N, f, g) &:= \mu(\lambda Q \in 2^\mathcal{S}\colon g \vee (f \wedge \mathbf{EX}(N, Q))),\\
\mathbf{fairEG}(N, f)(\mathit{fair}) &:= \nu(\lambda Q \in 2^\mathcal{S}\colon \mathbf{EU}(N, f, f \wedge \mathit{fair} \wedge \mathbf{EX}(N, Q))).
\end{aligned}$$

There are several other FairCTL operators inferred from these basic operators. In later sections, we will need the following:

$$\begin{aligned}
\mathbf{AG}(N, f) &:= \neg\mathbf{EU}(N, \mathit{TRUE}, \neg f),\\
\mathbf{fairAF}(N, f)(\mathit{fair}) &:= \neg\mathbf{fairEG}(N, \neg f)(\mathit{fair}).
\end{aligned}$$

Each of the operators yields a predicate on $\mathcal{S}$. Their intention is

- $\mathbf{EX}(N, f)(s)$ iff state s has a successor s' such that $f(s)$ holds.
- $\mathbf{EG}(N, f)(s)$ iff there exists an N-path starting from s such that f holds globally along the path.
- $\mathbf{EU}(N, f, g)(s)$ iff there exists an N-path starting from s such that f holds along the path until g holds, and g holds eventually.
- $\mathbf{fairEG}(N, f)(\mathit{fair})(s)$ iff there exists an N-path starting from s such that f holds globally along the path, and the fairness predicate *fair* holds infinitely often along the path.
- $\mathbf{AG}(N, f)(s)$ iff on all N-paths starting in s, f holds globally.
- $\mathbf{fairAF}(N, f)(\mathit{fair})(s)$ iff on all N-paths, on which *fair* holds infinitely often, f holds eventually.

For the definition and intention of additional FairCTL operators, we refer the reader to [CGP99].

5.5.3 Proof of μ-Calculus $\equiv \forall t$-Form

In the following, we prove that the FairCTL operators defined in μ-calculus match their intended semantics as described informally above. This is trivial for the $\mathbf{EX}$ operator and hence omitted.

Theorem 5.2 *It holds $\mathbf{EG}(N, f)(s)$ iff there exists an N-path $p_0, p_1, \ldots$ starting in s, i.e. $p_0 = s$, where all states satisfy f, i.e., $\forall t\colon f(p_t)$.*

The prove of theorem 5.2 follows [CGP99]. We prove the theorem using the following lemmas. For the rest of this section let

$$\tau := \lambda Q \in 2^{\mathcal{S}}\colon f \wedge \mathbf{EX}(N, Q).$$

Lemma 5.3 *τ is a monotonic predicate transformer.*

Proof: Trivial by definition of $\mathbf{EX}$. In PVS, the claim is proved automatically by the strategy `(grind)`. □

Lemma 5.4 *It holds $\mathbf{EG}(N, f)(s)$ iff $f(s) \wedge \mathbf{EX}(N, \mathbf{EG}(N, f))(s)$ holds.*

Proof: Expanding the definition of $\mathbf{EG}$ and substituting the predicate transformer τ, we have to prove $(\nu(\tau))(s) = (f(s) \wedge \mathbf{EX}(N, \nu(\tau))(s))$. From the monotony of τ and theorem 5.1 we know that $\nu(\tau) = \tau(\nu(\tau))$. By the definition of τ we have $\nu(\tau)(s) = (f(s) \wedge \mathbf{EX}(N, \nu(\tau))(s))$, which proves the claim. □

The following lemma corresponds to the $\Rightarrow$ direction of theorem 5.2, but comprises a stronger invariant.

Lemma 5.5 *Let $\mathbf{EG}(N, f)(s)$ hold. There exists an N-path $p_0, p_1, \ldots$ starting in s, and $\forall t\colon f(p_t) \wedge \mathbf{EX}(N, \mathbf{EG}(N, f))(p_t)$.*

Proof: The N-path is constructed inductively. We set $p_0 := s$. From lemma 5.4 we know that the induction base $f(p_0) \wedge \mathbf{EX}(N, \mathbf{EG}(N, f))(p_0)$ holds. Assume we have already defined $p_0, \ldots, p_k$. From the induction hypotheses we know $\mathbf{EX}(N, \mathbf{EG}(N, f))(p_k)$; expanding the definition of the $\mathbf{EX}$ operator yields $\exists s'\colon \mathbf{EG}(N, f)(s') \wedge N(p_k, s')$. We set $p_{k+1} := s'$. Lemma 5.4 yields the induction step $f(p_{k+1}) \wedge \mathbf{EX}(N, \mathbf{EG}(N, f))(p_{k+1})$. □

Lemma 5.6 *Define the predicate $\widehat{\mathbf{EG}} := \{s \in \mathcal{S} \mid \exists p_0, p_1, \ldots\colon p_0 = s \wedge \forall t\colon f(p_t)\}$. The predicate $\widehat{\mathbf{EG}}$ is a fixpoint of τ, i.e., $\widehat{\mathbf{EG}} = \tau(\widehat{\mathbf{EG}})$.*

Proof: We first show $\widehat{\mathbf{EG}} \supseteq \tau(\widehat{\mathbf{EG}})$. Let $s \in \tau(\widehat{\mathbf{EG}})$; by definition of τ we have $f(s) \wedge \mathbf{EX}(N, \widehat{\mathbf{EG}})(s)$, hence by definition of $\mathbf{EX}$ we have $\exists s'\colon N(s, s') \wedge \widehat{\mathbf{EG}}(s')$, hence $\exists p_0, p_1, \ldots\colon p_0 = s' \wedge \forall t\colon f(p_t)$ by definition of $\widehat{\mathbf{EG}}$. We define the path $p'_0 := s$, $p'_i := p_{i-1}$; this path proves $\widehat{\mathbf{EG}}(s)$.

Now we prove $\widehat{\mathbf{EG}} \subseteq \tau(\widehat{\mathbf{EG}})$. Let $s \in \widehat{\mathbf{EG}}$; we then have a path $p_0, p_1, \ldots$ with $p_0 = s$ and $\forall t\colon f(p_t)$. We have to show $f(s) \wedge \mathbf{EX}(N, \widehat{\mathbf{EG}})(s)$. $f(s)$ holds obviously, and the path $p_1, p_2, \ldots$ proves $\mathbf{EX}(N, \widehat{\mathbf{EG}})(s)$. □

Proof of Theorem 5.2. The $\Rightarrow$ direction follows directly from lemma 5.5. For the other direction, assume there exists a path $p_0, p_1, \ldots$ with $s = p_0$ and $\forall t\colon f(p_t)$,

i.e., $\widehat{\mathbf{EG}}(s)$ holds. We have to prove $\mathbf{EG}(N, f)(s)$. By definition, this is equivalent to $\nu(\tau)(s)$. By lemma 5.6, $\widehat{\mathbf{EG}}$ is a fixpoint of τ. Since τ is monotonic, we know that $\nu(\tau)$ is the greatest fixpoint (theorem 5.1), hence $\widehat{\mathbf{EG}} \subseteq \nu(\tau)$. Since $\widehat{\mathbf{EG}}(s)$ holds, we conclude $\mathbf{EG}(N, f)(s)$. □

We omit the correctness proofs for the other FairCTL operators. The proofs follow the same idea as for the $\mathbf{EG}$ operator. The proof for $\mathbf{fairEG}$ is slightly more complex due to the nested fixpoint operators. The proofs for $\mathbf{AG}$ and $\mathbf{fairAF}$ are simple using the correctness of $\mathbf{EU}$ and $\mathbf{fairEG}$. We refer the reader to [CGP99] for details. We give only the precise correctness statements in the following theorems:

Theorem 5.7 *Let $f, g \in 2^{\mathcal{S}}$ be predicates. It holds $\mathbf{EU}(N, f, g)(s)$ iff there exists an N-path $p_0, p_1, \ldots$ starting in s, where g holds eventually, and f holds until then:*

$$p_0 = s \wedge \exists t\colon g(p_t) \wedge \forall t' \in \{0, \ldots t-1\}\colon f(p_{t'}).$$

Theorem 5.8 *Let $f, \mathit{fair} \in 2^{\mathcal{S}}$ be predicates. It holds $\mathbf{fairEG}(N, f)(\mathit{fair})(s)$ iff there exists an N-path $p_0, p_1, \ldots$ starting in s, where f holds globally, and the fairness predicate fair holds infinitely often:*

$$p_0 = s \wedge \forall t\colon f(p_t) \wedge \forall t\colon \exists t' \geq t\colon \mathit{fair}(p_{t'}).$$

Theorem 5.9 *It holds $\mathbf{AG}(N, f)(s)$ iff for all N-paths $p_0, p_1, \ldots$ starting in s the predicate f holds globally:*

$$p_0 = s \implies \forall t\colon f(p_t).$$

Theorem 5.10 *It holds $\mathbf{fairAF}(N, f)(\mathit{fair})(s)$ iff for all N-paths $p_0, p_1, \ldots$ starting in s, along which fair holds infinitely often, the predicate f holds eventually:*

$$\big(p_0 = s \wedge \forall t\colon \exists t' \geq t\colon \mathit{fair}(p_{t'})\big) \implies \exists t\colon f(p_t).$$

5.5.4 Non-Determinism versus Input Sequences

As mentioned above, we have used non-deterministic systems without inputs in the context of FairCTL, whereas deterministic systems with inputs have been used in section 5.1 to define the correctness of execution units. The use of deterministic next state functions is better suited for the definition of execution units since it is closer to the actual implementation; furthermore, we believe it is simpler to handle in theorem proving. However, the definition of FairCTL in PVS imposes the use of non-deterministic systems for model checking. It is easy to bridge this gap:

Let $\mathcal{S}$ be the state type, $\mathcal{I}$ be the input type, and $ns : \mathcal{S} \times \mathcal{I} \to \mathcal{S}$ be the deterministic next-state function of a system as in section 5.1. Further, let $Ip \subseteq \mathcal{S} \times \mathcal{I}$ be an input predicate (e.g., $Ip \equiv stall_{out} \Rightarrow \neg valid_{in}$ to model the pipeline

input property (I1)). Let $init \in \mathcal{S}$ be the initial state. We define a new state type $\mathcal{S}' := \mathcal{S} \times \mathcal{I}$ and a non-deterministic next-state relation $N \subseteq \mathcal{S}' \times \mathcal{S}'$ by

$$N\left((s_1, i_1), (s_2, i_2)\right) := \left(s_2 = ns(s_1, i_1) \wedge Ip(s_2, i_2)\right).$$

Regard the new state type as current state and input. Then there is a transition from (s_1, i_1) to (s_2, i_2), iff the next-state function ns takes the transition $s_1 \rightarrow s_2$ under input i_1. Furthermore, the next-state relation N non-deterministically chooses the next input i_2, which has to satisfy the input-predicate Ip. We define $init' := \{(s, i) \mid s = init \wedge Ip(s, i)\}$ as the initial state set of the new system.

It is easy to see that computations in both systems are equivalent. We can restate the above theorems with respect to the input sequence semantics. For the sake of brevity, we only restate theorems 5.9 and 5.10:

Corollary 5.11 *It holds* $(\forall s' \in init' \colon \mathbf{AG}(N, f)(s'))$ *iff for all input sequences* $I := (i_0, i_1, \ldots) \in \mathcal{I}^\infty$ *satisfying the input predicate, the predicate* f *holds globally:*

$$\left(\forall t \colon Ip(s^t(I), i_t)\right) \implies \left(\forall t \colon f(s^t(I))\right),$$

where s^t *is defined as in section 5.1.*

Corollary 5.12 *It holds* $(\forall s' \in init' \colon \mathbf{fair}\mathbf{AF}(N, f)(\mathit{fair})(s'))$ *iff for all input sequences* $I := (i_0, i_1, \ldots) \in \mathcal{I}^\infty$ *satisfying the input predicate and yielding a path on which fair holds infinitly often, the predicate* f *holds eventually. Formally: for all input sequences* I *holds:*

$$\left(\left(\forall t \colon Ip(s^t(I), i_t)\right) \wedge \left(\forall t \colon \exists t' \geq t \colon \mathit{fair}(s^t(I))\right)\right) \implies \left(\exists t \colon f(s^t(I))\right).$$

The proofs of corollaries 5.11 and 5.12 from theorems 5.9 and 5.8 are straightforward. In PVS, they are proved using the `(grind)` command.

In the following, we do not explicitly distinguish between systems stated as next-state function or relation. Of course, one has to deal with the differences in PVS, but this is easy and hence omitted in the rest of this chapter.

5.6 Pipeline Verification using Model Checking and Theorem Proving

5.6.1 Separating Pipeline Control and Datapaths

In order to use model checking on the pipeline control we have to separate the control and datapath circuits in the pipeline. Figure 5.3 shows a simple pipeline example. The control registers consist of valid bits indicating that a stage contains a valid instruction, the tags, and some auxiliary control data, e.g., a counter to keep track of the number of iterations to go through during divisions. The control circuit maintains the control registers, and computes the control outputs $valid_{out}, tag_{out}$,

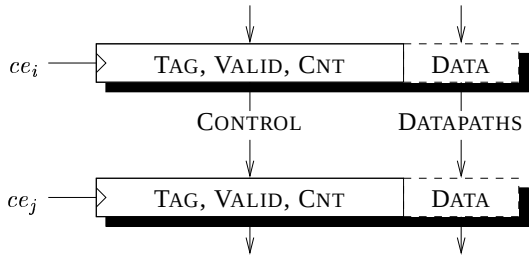


Figure 5.3: Separating Control and Datapaths

and $stall_{out}$. The control interacts with the datapaths by computing the clock-enables ce for each stage and the multiplexer control signals where multiple inputs lead to the same pipeline stage (e.g, to the MD-STG1 stage in Fig. 5.2). According to the separation of control and data in the pipeline, we split the next-state function ns of the pipeline into a next-state function ns_{ctrl} of the control part, and a next-state function ns_{data} of the data part.

5.6.2 Verification of the Pipeline

In the following, we describe how we verify the liveness (P3) and tag-consistency (P4) properties of pipelines. We will not discuss the (P1) and (P2) properties, since these are fairly simple in comparison. Furthermore, we will only give the idea of the actual verification, since the mathematical details are tedious and straightforward.

Liveness

We start with the verification of liveness. In order to prove *functional* correctness of the pipelines, we prove the strengthened liveness property (P3′) covering the functionality of the pipeline. We first use model checking to verify the liveness of each clock-enable signal. The liveness of $stall_{in}$ is presumed. No other input-predicate is used, i.e., $Ip := \mathit{TRUE}$. We model-check the following property for each stage i and arbitrary, not necessarily reachable or initial control state s:

$$\mathbf{fair}\,\mathbf{AF}(ns_{ctrl}, ce_i)(\neg stall_{in})(s). \tag{5.1}$$

Using corollary 5.12 we conclude that the clock-enable ce_i is live in all computations starting from an arbitrary state s under all input sequences where $stall_{in}$ is live, i.e., for all $I := (i_0, i_1, \ldots)$ holds

$$\left(\forall t\colon \exists t' \geq t\colon \neg stall_{in}^{t'}\right) \implies \left(\exists t\colon ce_i^t\right). \tag{5.2}$$

Note that the left-hand side of the equation matches the pipeline input property (I2). Note further that (5.3) asserts *only one* activation of ce_i in the path starting from s. The following corollary extends this statement to an infinite number of ce_i activations.

Corollary 5.13 *Let s_0 be an initial state, and let $I = (i_0, i_1, \ldots)$ be an input sequence. It holds*

$$\big(\forall t\colon \exists t' \geq t\colon \neg stall_{in}^{t'}\big) \implies \big(\forall t\colon \exists t' \geq t\colon ce_i^t\big). \tag{5.3}$$

Proof: The corollary is proved from (5.3) using theorem proving. Let a time t be given. We have to show $\exists t' \geq t\colon ce_i^{t'}$. The system is in some state $s^t(I)$ at time t. Equation (5.2) holds for arbitrary input sequences and arbitrary initial states. Hence, we may apply (5.2) with initial state $s^t(I)$ and with input sequence $I^{\geq t} := (i_t, i_{t+1}, \ldots)$, i.e., the part of the input sequence I laying in the future. Equation (5.2) yields a time $\hat{t}$ where $ce_i^{\hat{t}}$ holds with respect to input sequence $I^{\geq t}$ and initial state $s^t(I)$. Hence, with $t' := t + \hat{t}$ holds $ce_i^{t'}$ with respect to input sequence I and initial state s_0. This proves the claim. □

Having proved the liveness of the clock-enables, it is relatively easy to verify liveness of the complete pipeline including the datapaths by pushing instructions through the pipeline stage by stage. This is done using theorem proving. We exemplarily prove the liveness property (P3′) of the multiplicative FPU for multiplication instructions:

Theorem 5.14 *Assume that the input properties (I1) and (I2) hold. Assume further that a multiplication with tag tg is dispatched at time t, i.e., $disp(tg, t)$ holds. Then there exists $t' \geq t$ such that $ret(tg, t')$ and $data_{out}^{t'} = \text{MD-COMB}(data_{in}^t)$ hold, i.e., the multiplication eventually terminates with the correct data.*

Proof: We only sketch the proof, because its details are long and tedious. By input property (I1) we know that $stall_{out}^t$ is inactive, since otherwise the instruction cannot be dispatched. Since the definition of $stall_{out}$ directly depends on $ce_{\text{MD-UNP}}$ (cf. Appendix D), one trivially concludes that the instruction is clocked into the register stage MD-UNP at time t. The data in this register are the outputs of the combinatorial MD-UNP circuit.

From corollary 5.13 we know that there exists a (minimal) time $t_1 > t$ such that $ce_{\text{MD-UNP}}^{t_1}$ is active, i.e., the MD-UNP stage is clocked at time t_1, and is not clocked in between. Hence, the data at time $t_1 - 1$ in the register stage MD-UNP is the same as at time t.

The MD-UNP stage can only be clocked if its valid instruction proceeds to the next stage (this is concluded trivially from the definition of $ce_{\text{MD-UNP}}^t$). Hence, we can conclude that the instruction with tag tg is clocked from the MD-UNP stage into stage MD-STG1 at time t_1. The data at this time is computed from MD-STG1 ∘ MD-UNP, i.e., the composition of the first two combinatorial stages.

Analogously, we derive times $t_2 > t_1$, $t_3 > t_2$, and $t_4 > t_3$ where the instruction proceeds to MD-STG2, RD-STG1, and RD-STG2, respectively. When the instruction is in stage RD-STG2, it is returned to the CPU immediately when the $stall_{in}$ signal becomes inactive. Hence, there exists $t' > t$ where the instruction

is returned with $data^{t'}_{out}$ computed from $data^{t}_{in}$ by the combinatorial circuits between the register stages, i.e., RD-STG2 $\circ$ RD-STG1 $\circ$ MD-STG2 $\circ$ MD-STG1 $\circ$ MD-UNP $=$ MD-COMB by definition of MD-COMB. $\square$

Tag-Consistency

The verification of tag-consistency is slightly more complicated. We want to express tag-consistency (P4) in FairCTL in order to allow model checking. Therefore we need a FairCTL formalization of "tag has been dispatched previously", and a formalization of tag-uniqueness. It would be useful to have temporal operators reaching in the past; however, FairCTL does not provide such operators.

In order to circumvent this problem, we introduce an auxiliary variable $inuse_{tg}$ for each tag $tg \in \mathcal{T}$ representing that an instruction with tag tg is currently in the pipeline. The meaning of this variable is exactly the same as the predicate $inuse$ from section 5.1. The variable $inuse_{tg}$ is set whenever an instruction with tag tg enters the pipeline, and it is cleared whenever the tag tg leaves the pipeline. Tag-uniqueness can hence be modeled as input predicate Ip asserting that the tag tg is not dispatched when the variable $inuse_{tg}$ is already set. Vice versa, tag-consistency can be modeled as an invariant stating that a tag tg can only leave the pipeline if $inuse_{tg}$ is set.

Let $\tilde{nst}_{ctrl}$ denote the next-state function of the modified model including the $inuse$ variables, and let Ip denote the input predicate modeling tag-uniqueness (I3), i.e., $Ip := \forall tg\colon valid_{in} \wedge tag_{in} = tg \Rightarrow \neg inuse_{tg}$. Using model checking, we verify the property

$$\forall tg\colon \mathbf{AG}\big(\tilde{nst}_{ctrl}, (valid_{out} \wedge tag_{out} = tg) \implies inuse_{tg}\big)(init),$$

where $init$ is an initial state in which all pipeline stages are empty (i.e., $valid_i = 0$), and all $inuse_{tg}$ variables are cleared. From this we conclude using corollary 5.11: for all input sequences $I = (i_0, i_1, \ldots) \in \mathcal{I}^\infty$ and for all tags tg, it holds

$$\begin{aligned}\big(\forall t\colon \big(valid^t_{in} \wedge tag^t_{in} = tg\big) &\implies \neg inuse^t_{tg}\big) \implies \\ &\big(\forall t\colon \big(valid^t_{out} \wedge tag^t_{out} = tg\big) \implies inuse^t_{tg}\big)\,.\end{aligned}$$

Rewriting this with the definitions of $disp(t, tg)$ and $ret(t, tg)$ (cf. section 5.1.1) yields

$$\begin{aligned}\big(\forall t\colon disp(t, tg) &\implies \neg inuse^t_{tg}\big) \implies \\ &\big(\forall t\colon ret(t, tg) \implies inuse^t_{tg}\big)\,.\end{aligned}$$

As one can see (and easily verify in PVS), the left-hand side of the implication matches tag-uniqueness, and that the right-hand side implies tag-consistency.

5.6.3 Some Practical Considerations

In order to verify tag-consistency, we have changed the model and added the auxiliary variables $inuse_{tg}$. It is easy to prove that these auxiliary variables do not affect the outputs of the actual pipeline implementation and hence can be omitted in the implementation. They are solely used to prove the correctness of the pipeline.

The state-space for model checking becomes very large due to the tags and the $inuse_{tg}$ variables. Of course, one can abstract the tags by means of scalar-sets [ID96] in the sense of data-type reduction as in SMV [McM00]. Model-checkers such as SMV support this as a built-in feature. However, in PVS the abstraction has to be done manually. We have abstracted the $inuse_{tg}$ variables to only one variable, but we have not abstracted the width of the tags themselves. To abstract the tags would not be overly hard, but is not necessary for our purpose.

A major disadvantage of the PVS model-checker is that it is not capable of providing counter-examples when the verification of a FairCTL formula fails. Since the design of complex pipelines is very error-prone and debugging is hard, such counter-examples are very useful. We therefore developed and debugged the pipelines (without datapaths) in SMV, and then manually translated the pipeline control to PVS. We then used the PVS model-checker to re-check the properties.

We have manually performed the "pushing through the pipeline" stage by stage during liveness verification. The proofs for each stage are very similar. We therefore believe that it is possible to create a proof strategy which performs the "pushing through the pipeline" automatically. This would result in a mostly automatic method for the verification of complex pipelines.

The presented methodology does not cope with pipelines where the liveness of the clock-enables or the tag-consistency depends on signals computed by the datapaths which are fed into the pipeline control. In our FPUs, the datapaths only compute whether the operations are special cases, and if they are double or single precision operations. Liveness and tag-consistency do not depend on these signals. Hence, these signals can be left un-specified during model checking.

It is imaginable that one could build pipelines where the liveness or tag-consistency depends on complex computations within the datapaths, e.g., for self-timed division algorithms (see, e.g., [CL93]). This could signifcantly complicate the model checking step of our method. It would be interesting future work to extend our methodology to such pipelines.

A principle idea to cope with such pipelines would be to "guess" the values passed from the datapaths to the control in an "oracle" outside the actual pipeline. This could eliminate the need to incorporate the datapaths into the model-checked model. Another idea is to prove upper bounds on the number of iterations of a self-timed circuit using theorem proving, and to incorporate an additional counter into the control which counts up to this maximum number of iterations. However, we have not considered such self-timed pipelines in detail.

5.7 Putting It All Together

In this section, we present the formal correctness statement of the pipelined multiplicative FPU. The correctness statement for the other FPUs is completely analogous. We remind the reader that clocked circuits are described in PVS by a next-state/output circuit (cf. section 2.3).

Circuit 5.1 (MD-PIPE) The next-state/output circuit of the pipelined multiplicative FPU has the following inputs:

- *STATE*: the current state of the execution unit, i.e., the content of the registers within the pipeline,
- $clear, valid_{in}, stall_{in} \in \mathbb{B}\,, tag_{in} \in \mathbb{B}^3$: the execution unit control inputs,
- $data_{in}$: the inputs of the datapaths. These are the same inputs as those of the circuit MD-COMB on page 63.

The next-state/output circuit computes

- *nSTATE*: the next state of the execution unit, i.e., the new content of the registers,
- $valid_{out}, stall_{out} \in \mathbb{B}\,, tag_{out} \in \mathbb{B}^3$: the execution unit control outputs,
- $data_{out}$: the outputs of the datapaths; the same as the outputs of MD-COMB.

Note that we have fixed the tags to be three bits wide. This is arranged to fit with the *VAMP* processor, and can be adjusted easily. ◇

The following theorem asserts the correctness of the multiplicative FPU execution unit:

Theorem 5.15 *The circuit* MD-PIPE *is a functional correct execution unit (cf. section 5.1.2) with respect to the function computed by the combinatorial multiplicative FPU* MD-COMB.

The theorem is proved using the techniques presented in the previous sections.

5.8 Related Work

There are a couple of papers which report on the verification of out-of-order processors, e.g., by Hosabettu et al. [HGS00], by Sawada and Hunt [SH98], by McMillan [McM00], and by Berezin et al. [BBCZ98]. None of the cited papers mentions execution units which have a cycle in the pipeline structure or may reorder instructions internally. Kröning is the first who reports on the verification of a Tomasulo scheduler capable of handling such complex pipelines [Kro01], although the design

and the verification of the actual pipelines is not part of Kröning's work. In this chapter we have presented a general methodology to verify complex pipelines, and have presented the pipeline of the multiplicative floating point unit as an example.

Aagaard and Leeser [AL94] propose a methodology for the verification of complex pipelines: they decompose pipelines into smaller segments, and then further decompose the correctness proof of individual segments into smaller proof goals. However, their work describes only how one *could* employ a theorem prover for the verification of pipelines, but they do not actually use formal methods (in the sense of a computer tool). We have described a similar approach to the verification of our pipelines using solely theorem proving in section 5.3, but failed because very complex inductive invariants had to be constructed manually.

Another approach to the verification of pipelines is the use of a logic with uninterpreted functions [BD94] that are used to model the datapath functionality. The use of uninterpreted functions is comparable to the separation of the EU into pipeline control and datapaths, since the actual datapath implementation has no impact on the pipeline verification (cf. section 5.6). Bryant et al. [BGV01] describe how a logic with equality and uninterpreted functions can be reduced to propositional logic. In [VB00], Velev and Bryant describe how this reduction can be used to verify in-order microprocessors with variable-latency EUs. However, they do not verify the actual EU, but use an abstract execution unit model in order to verify the processor core. The EUs modeled by the abstraction process only one instruction at a time, and hence do not reorder instructions internally. Velev and Bryant only verify in-order processors; the verification of out-of-order designs would probably require the manual construction of a complex inductive invariant, and hence automation would be lost. In our approach, this is not the case due to the use of model checking.

Another approach is the use of uninterpreted functions within a model-checker such as SMV. Data-type reduction and case-splitting is used to reduce the state space [McM00]. This is used in [McM00] to verify a Tomasulo scheduler, where the functionality of the EUs is defined by uninterpreted functions. However, the state space and the number of cases to be checked grows rapidly in the number of function applications, which is large in our example due to the cycle in the pipeline structure, cf. section 5.4.

In [BBCZ98], Berezin et al. prove the correctness of a simple Tomasulo processor by combining model checking with uninterpreted functions and theorem proving. They use SMV to verify an invariant of an abstraction of the processor, and then use PVS to conclude overall correctness of the concrete machine. Their translation from SMV to PVS is not formally safe in the sense that they introduce a new, manually written axiom in PVS which hopefully reflects exactly the model-checked property. In contrast, we use the PVS built-in model-checker, and then use the theorems from section 5.5 to safely translate the model-checked properties to a form suitable for theorem proving.

In [HIK98], Ho et al. use the abstraction of the datapaths of pipelines to token nets for the automatic verification of pipeline control properties. Their approach is not applicable to pipelines with cycles in the pipeline structure, and is not suitable to verify functional correctness of the pipelines.

In [AJK00], Aagaard et al. verify iterative circuits using Intel's Forte system. They use symbolic simulation and model checking for the verification of bit-level invariants of iterative floating point circuits, and then use theorem proving to conclude "numerical" correctness of the floating point results. Though Intel's circuits are most probably much more complex than ours in terms of gate count, the verified pipelines are simple in the sense that they seem to support only one instruction at a time and hence do not reorder instructions. Details are not described in [AJK00]. Moreover, the work from [AJK00] is not reproducible in a scientific sense since Intel's Forte system is not publicly available.

Schneider and Hoffmann [SH99] report on the definition of the temporal logic LTL in the theorem prover HOL [GM93], and on the automatic translation of LTL to ω-automata within HOL. The ω-automata are used as input for a model-checker. Their definition of LTL is close to our $\forall t$ form. Hence, their work could be used to verify pipelines in HOL in a similar way as described here.

Chapter 6

The *VAMP* Project

The work presented in the previous chapters is part of the *VAMP* project at Saarland University. The goal of the *VAMP* project is the formal verification of a complete microprocessor called *VAMP* (for Verified Architecture Microprocessor). The *VAMP* is a variant of the ubiquitous DLX processor [HP96]. The *VAMP* features a 5-stage pipeline, out-of-order execution by means of a Tomasulo scheduler [Tom67], precise interrupts, delayed branch, a memory unit with caches, and the IEEE compliant floating point units presented in the previous chapters. Our group is planning to enhance the memory unit with virtual memory management.

The *VAMP* processor is designed and verified completely in PVS. We have developed a tool called *pvs2hdl* which automatically translates the PVS hardware designs to the hardware description language *Verilog* [Ver96, Cil99]. Using this tool, the *VAMP* processor is implemented on a Xilinx Virtex-E FPGA [Xil02].

We have ported the GNU C-compiler *gcc* and the C library *glibc* for the *VAMP* processor. The ports are based on the *gcc* and *glibc* ports for the Hennessy-Patterson DLX [O'K97]. We have developed an interface which allows to run programs on the *VAMP* implementation on the Xilinx FPGA.

People. Several persons participate in the *VAMP* project (alphabetically):

- Christoph Berg: verification of the floating point adder core,
- Sven Beyer: verification of the memory unit, and development of the *pvs2hdl* tool,
- Christian Jacobi: verification of the FPUs, and implementation of the hardware on the FPGA,
- Daniel Kröning: verification of the processor core and integer ALU,
- Dirk Leinenbach: development of the *pvs2hdl* tool, and implementation of the hardware on the FPGA,
- Carsten Meyer: development of the software environment for the *VAMP*,

- Wolfgang J. Paul: supervisor.

The *VAMP* project resulted in several scientific publications [BBJ+02, Bey02, BJ01, BJK01a, BJKL02, Jac01, Jac02, JK00, KMP99, Kro01, KP01, MPK00], partly still in the publication process. Several doctoral and diploma theses are in progress at the time of this writing (April 2002).

In the following sections, we will give a more detailed overview of the *VAMP* project. Section 6.1 is taken from [BJK01b, Sect. 2]. Sections 6.4–6.5 are based on [BJKL02].

6.1 The *VAMP* Processor Core

The *VAMP* processor is an out-of-order variant of the DLX processor [HP96]. Out-of-order execution allows for high performance even in case of long latency instructions such as floating point or memory instructions. One of the most popular out-of-order execution algorithm is the Tomasulo scheduling algorithm [Tom67]. It is one of the most competitive scheduling algorithms and provides CPI rates down to 1.1 on a single-instruction issue machine [MLD+99]. The algorithm is widely used, e.g., in the IBM PowerPC [Mot97] and Intel's Pentium-Pro [CS95]. The original Tomasulo scheduler uses out-of-order termination and therefore does not support precise interrupts. The *VAMP* supports precise interrupts by means of a *reorder buffer* [SP88]. The reorder buffer sorts the instructions in program order before termination. Figure 6.1 depicts the basic structure of the *VAMP* microprocessor. Stage IF does the instruction fetch; we implement one branch delay slot using the Delayed PC technique [MPK00]. The delayed branch semantics is, for example, used in the MIPS instruction set architecture (ISA) [KH92]. The hardware for the instruction fetch is taken from the in-order machine described in [MP00, JK00].

In the next stage, the instruction is decoded. This includes fetching the operands if available. The instruction and the operands are then passed to a reservation station (RS). This is called *issue*. The reservation stations act as queue for the instructions and are located between the decode/issue stage and the execution units (EUs).

As soon as all operands are available, the instruction is passed from the reservation station to the EU in the execute stages (dispatch). The *VAMP* features five EUs: the ALU handles integer instructions such as `add`. The second EU is the interface to the data memory. The remaining EUs are the FPUs presented in the previous chapters. As described in chapter 5, the EUs may process multiple instructions simultaneously, and may return instructions out-of-order. Currently, only the FPUs use these possibilities. However, one could extend the *VAMP* with pipelined integer ALUs (e.g., including an integer divider), or with non-blocking caches, see e.g. [Pre02].

After the EU has finished the execution, the result of the instruction is passed to the producer registers. In case the producer holds an instruction, it requests the

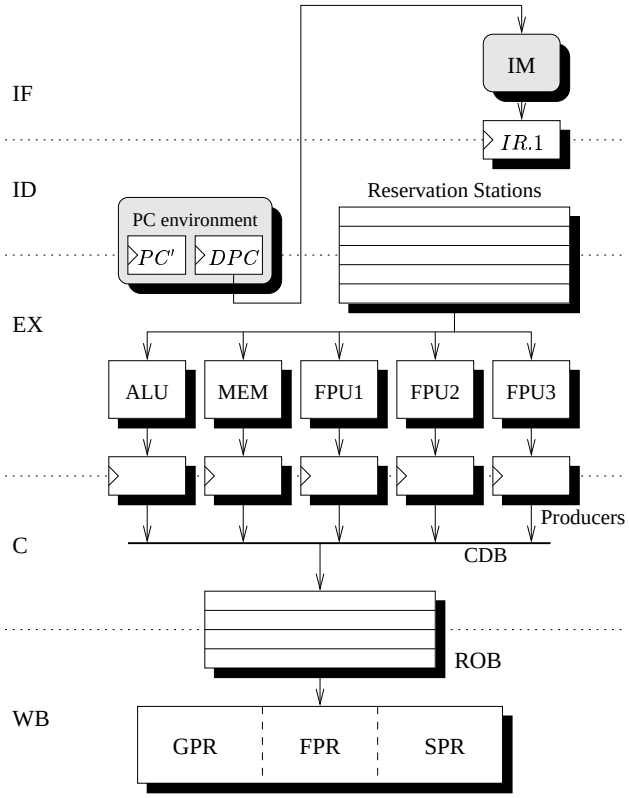


Figure 6.1: Overview of the *VAMP* microprocessor

common data bus (CDB). As soon as the request is acknowledged, the result is put on this bus (completion). In contrast to commercial designs such as the IBM's PowerPC, we support only one CDB. The bus is used for two purposes: 1) The instruction is passed to the reservation stations that wait for the result because of a data dependency, and 2) the result is passed to the reorder buffer.

The reorder buffer re-sorts the instructions back to program order. The benefit of this is that we can write the results into the register file in program order (in-order termination). This allows precise interruptions of the instruction stream.

Status. The processor core is work by Daniel Kröning [Kro01]. The verification of the Tomasulo algorithm and of the ALU is complete. The gate-level implementation of the processor core is complete, the verification of the gate-level is expected to be complete in May 2002.

6.2 The Memory Unit

The memory unit connects the main memory to the *VAMP* processor. The unit supports split instruction and data caches, backed up by the shared main memory.

Data-consistency between instruction- and data-cache is guaranteed by snooping. The split instruction and data caches allow simultaneous memory-accesses for instruction fetch and load/store instructions. Our group is planning to enhance the memory unit with a second-level cache, virtual memory, and address translation.

The cache designs are parameterized, i.e., the size and associativity of the caches can easily be changed by adopting parameters. The caches support both write-back and write-through mode. The main memory to which the caches connect is assumed to work with the bus protocol from [MP00, Sect. 6]. However, this could be easily adopted to any other synchronous protocol.

In the actual VAMP implementation, the first-level instruction cache is a 2-way set-associative 8 KB read-only cache. The data cache is a 4-way set-associative 16 KB read-write cache with write-back policy. The least-recently-used (LRU) policy is implemented as replacement strategy.

Status. The verification of the memory unit is performed by Sven Beyer. The verification is nearly finished (expected May 2002). The implementation and verification of virtual memory and address translation is planned, but not yet started. We also plan to enhance the VAMP to properly work with self-modifying code. The VAMP shall detect whether an instruction in the pipeline has been overwritten by a preceeding data memory access, and shall signal an interrupt in this case.

6.3 Verification Effort

Table 6.1 lists the effort needed for the verification of the different parts of the *VAMP* FPU. As one sees, there is a large gap between the PVS proofs and those from [MP00] in the number of lemmas and theorems. This is due to two reasons: first, many seemingly trivial things are not proved in [MP00]. This, in particular, includes the width of busses, adders, etc. The lack of verification of these "trivial" things was source of several bugs in [MP00]. Second, a lot of the mathematics in [MP00] is scattered over the continuous text. For example, the η-computation circuit within the rounder is described and verified over 12 pages in [MP00] (the proof is incorrect, cf. section 4.2.1), but has only 3 explicit lemmas. A large part of our work was to divide the mathematics in the text from [MP00] into lemmas. The η-computation, for example, has 34 lemmas in PVS.

However, it should be noted that the explanations and proofs in [MP00] originally were not intended to serve as guidelines for formal verification, but for human readability, in particular for students in computer architecture. It is remarkable that

[1] Joint work with Christoph Berg and Sven Beyer.

[2] Verified by Christoph Berg.

[3] Joint work with Christoph Berg.

[4] The proofs are all very similar. The large number of 2541 proof commands is due to a lot of "copy & paste".

	PVS		MP00		
	Lemmas	Steps	Lemmas	Pages	Steps/Page
Basic Circuits[1]	107	4032	4	23	175
Theory of Rounding	266	4808	9	33	146
Unpacker	13	361	0	5	72
Add/Sub[2]	180	3928	1	14	280
Mul/Div	106	2817	5	18	157
Rounder	98	4008	5	22	182
Compare/Convert	33	1616	1	20	81
Misc. Lemmas[3]	123	895			
CTL $\equiv \forall t$	30	930			
Pipelining of the FPUs[4]	90	2541			
$\sum$	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 [MP00]. All in all, the PVS specifications of the FPU take 374 KB of source, and the proof scripts take 1.1 MB (excluding whitespace).

Table 6.1: Verification effort for the FPUs,

VAMP part	Lemmas	Steps
Processor Core & ALU	521	14367
Caches	625	23359
FPU	1046	25936

Table 6.2: Overall effort of the different *VAMP* parts (FPU&Caches not yet complete).

the FPU from [MP00] has so few bugs considering that the hardware has not been implemented and tested, but "only" proved by hand.

The total verification effort for the different parts of the *VAMP* processor is listed in table 6.2. The complexity of the proofs in the individual parts of the *VAMP* have different sources. The main challenge for the verification of the Tomasulo scheduler and the caches is finding an inductive invariant for the data consistency. 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 [DV02]; 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. Contrary, the composition of parts was mostly straightforward given the decomposition facilities provided by the theory of rounding.

6.4 Translating PVS to Verilog

We have developed a tool named *pvs2hdl* which allows the automatic translation of PVS hardware designs to the hardware description language Verilog [Cil99]. As described in section 2, the functional language within PVS is used to model combinatorial hardware designs in PVS. Clocked circuits are modeled by input, output and register types and a next-state/output function. The next-state function computes from the current input and state the next state and the outputs of the circuit.

The translation of combinatorial hardware from PVS to Verilog works as follows: for each PVS function, a Verilog module is generated. If the PVS function has integer parameters to represent parameterized circuits (e.g, the carry-chain adder from section 2.2), a Verilog module for each different occurring parameterization is generated. If the PVS function is recursive, the recursion is unrolled. This is necessary since Verilog does not support recursion.

The translation of the bitvector operators *and, or, not, xor, if, cond*, and bitvector concatenation and extraction is straightforward, since these constructs have their literal counterparts in Verilog. Verilog does not support records, thus records are flattened into bitvectors during the translation.

PVS function calls are translated to module instantiations in Verilog. While there may be multiple instantiations of the same module—resulting in multiple occurrences of the module in the actual hardware—, the module itself is translated to Verilog only once.

In order to translate clocked circuits to Verilog, the name of the next-state/output function (say ns) and of the state type (say $State$) are passed to *pvs2hdl*. The tool first translates the function ns to a combinatorial Verilog module `ns` as described above. The tool then creates a Verilog module `ns_clk` with a clock input and a local register variable R of type $State$ (in case $State$ is a record type, there may be multiple registers due to flattening). The clocked module `ns_clk` has a single sub-module `ns`. The inputs and outputs of `ns_clk` are connected to the corresponding inputs of `ns`, and the register R is connected to the state input/output of the `ns` module.

The *pvs2hdl* tool does currently not support the automatic generation of RAM and ROM. However, RAM and ROM are needed within the *VAMP*, e.g., for the register files or the division lookup table. The RAM and ROM modules in Verilog are manually incorporated into the automatically generated *VAMP* Verilog sources. The content of the ROM is extracted from the PVS sources using a simple script.

6.5 Experimental Results

In this section we present experimental results. We have translated various combinatorial and clocked circuits from PVS to Verilog, and implemented them for usage on a Xilinx Virtex-E FPGA [Xil02]. We compare the cost and delay of the

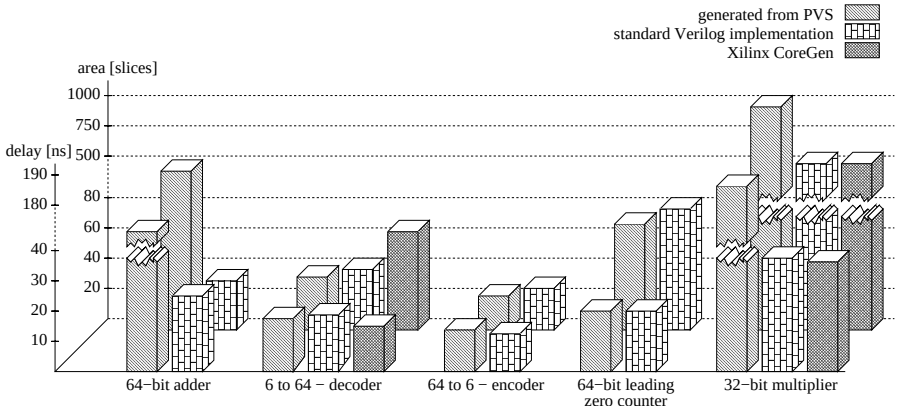


Figure 6.2: Comparison of the cost of translated designs and optimized macros

generated circuits to implementations in Verilog using native Xilinx macros. The cost and delay of the circuits are determined using the Xilinx Foundation software. The unit of cost is a *Virtex-E slice*, which reflects FPGA area. The unit of delay is a nano-second (ns).

6.5.1 Implementation of General-Purpose Circuits

Adder. We have translated a 64-bit carry-chain adder as defined in section 2.2. The implementation results in a cost of 106 slices and a delay of 79 ns, while the use of a standard Xilinx adder macro results in a cost of 33 slices and a delay of 23.4 ns. The large gap is due to special hardware resources called *fast carry logic* used by the optimized adder macro. Since the *pvs2hdl* translator does not trigger the usage of these architecture-dependent resources, our implementation performs very poorly. This phenomenon shows up only for adders, incrementers, and multipliers.

Decoder. We have implemented a recursive decoder in PVS. The implementation of the 6-to-64–bit decoder has a cost of 36 slices and a delay of 18.2 ns. Using a standard Verilog implementation as in [Cil99, pg. 328] yields 40 slices and 19.5 ns. A decoder generated by the Xilinx CoreGen software yields 64 slices and 16.3 ns. So our implementation is the cheapest, and is nearly as fast as the Xilinx CoreGen variant, although this has nearly twice the cost.

Others. Figure 6.2 shows comparisons of various circuits implemented in PVS with implementations in Verilog or with special Xilinx macros. Except for adders and multipliers, the translated PVS circuits can compete with the other implementations. The PVS implementations even outperform the usual implementations in some cases.

Since the *VAMP* processor comprises large adders and multipliers (e.g., in the multiplicative FPU), we have added support for predefined Verilog modules to the *pvs2hdl* tool. This allows the replacement of basic modules (e.g, adders and multipliers) by cheaper and faster Xilinx macros. Of course, the correctness of the complete processor then depends on the correctness of the Xilinx macros. However, the *VAMP* processor would not fit onto the Xilinx FPGA if we would not use the special macros. If we would implement the *VAMP* processor in a full-custom process, we could use our verified components.

In the following sections, experimental results are always generated by replacing all adders and multipliers in the designs by Xilinx macros. All circuits were implemeted without floorplanning, although this could have a significant impact on size and delay. However, our main objective is on correct rather than fast or small hardware. We therefore have not considered floorplanning so far.

6.5.2 Implementation of the Floating Point Units

We have translated our three FPUs to Verilog, and have implemented them on a Xilinx FPGA. In this section, we consider the pipelined multiplicative FPU as an example. The largest components of the FPU are a 58-bit multiplier, each one 64-bit shifter and 64-bit leading zero counter for each of the two unpackers, a further 64-bit shifter and a 58-bit half-decoder for the η-COMPUTATION within the rounder, and various adders and incrementers. As described in section 4.3, the 58-bit multiplier is built from two 29-bit and one 30-bit multiplier using the Karatsuba-Ofman scheme [KO63]. This is beneficial, since the Xilinx multiplier macros support only multipliers up to 32 bits. The multipliers and all adders and incrementers in the FPU are replaced by Xilinx macros.

Cost and Delay. The translation of the multiplicative floating point unit from PVS to Verilog yields a design requiring 4243 slices. This accounts for $\sim 25\%$ of the complete 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 [MP00] estimated a gate-count of 87.000, hence 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 RD-STG1. It involves a leading-zero counter on the input significand, a 13-bit adder, and a 53-bit cyclic shifter in the circuit η-COMPUTATION, and a 103-bit or-tree for the sticky-bit computation in the circuit REP. Nearly 80% of the delay of 59.4 ns are due to routing, only 20% are due to logic delay.

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

Testing the FPU. We have run several 100.000 random test-vectors on the FPGA implementation of the FPU. After having debugged the *pvs2hdl* tool with the gen-

eral purpose circuits described above, the FPU worked on the first try. We have compared the results of the test-vectors with the Intel FPU inside the Pentium II processor. We found two sources of discrepancies between our and Intel's FPU. First, some operations involving NaNs are handled differently. However, the IEEE standard [IEEE] is under-specifying in these cases, so both our and Intel's FPU conform to the standard.

The second source of discrepancies revealed a non-conformance of Intel's FPU to the IEEE standard. Internally, the Intel FPU always operates with the extended precision exponent width of 15 bit. The error 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 $e_{\min}$. In correct IEEE rounding, the significant of this result has to be denormalized with respect to the 11-bit $e_{\min}$ before rounding. In our rounding unit this is accomplished by circuit η-COMPUTATION (section 4.2). On Intel's FPU, however, the result is first rounded to the target 53-bit significant 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. It is well known that this twofold rounding is not IEEE compliant [Lee89].

For example, assume that the normalized significand f is of the form $f = \ldots 001\llcorner 01$, where the '$\llcorner$' sign denotes the least representable bit, i.e., the significand bit with weight 2^{-52}. Let the corresponding denormalized significand f' be $f' = \ldots 00\llcorner 101$, i.e., f' is obtained by denormalizing f by one bit. In round to nearest mode, f' is rounded up to $f'_{rd} = \ldots 01\llcorner$. In contrast, the normalized f is first rounded down to $f_{rd1} = \ldots 001\llcorner 0$. This intermediate result is then denormalized to $\ldots 00\llcorner 10$. This results in a tie for rounding to nearest, and hence is rounded to the nearest representable number with least-significand bit zero, which is $\ldots 00\llcorner 00$. This differs from the correct f'_{rd} in the least significant representable bit.

The described problem is known to Intel, as a talk by Roger Golliver (Intel) shows [Gol98]. In the next generation of Intel's FPU, the internal exponent width will be adjustable in software (personal communication with John Harrison, Intel). It should be noted that we have also tried these operations on AMD processors, which yielded the same result as Intel's FPU.

6.5.3 Implementation of the Complete *VAMP* Processor

We have implemented the complete *VAMP* processor on the FPGA, including caches and FPUs. Beside the verified *VAMP* circuits, some auxiliary circuits are needed to run the *VAMP* processor on the FPGA. Figure 6.3 gives an overview of the complete *VAMP* implementation. The FPGA resides on a PCI board within a host PC. The FPGA is connected to four SDRAM chips, and to a bridge to the PCI bus. This bridge is used for communication with the host PC. On the FPGA is the actual *VAMP* processor, the interface to the bridge, and a memory controller.

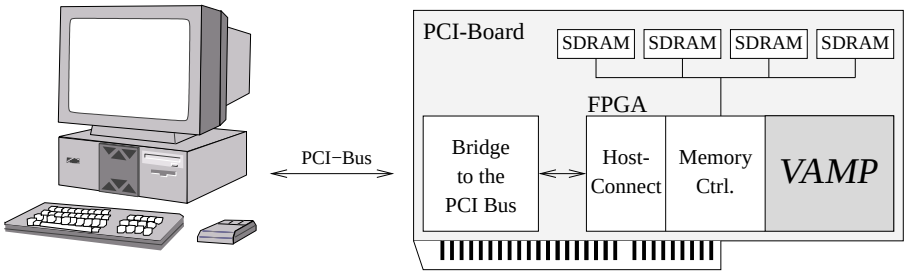


Figure 6.3: Overview of the *VAMP* processor implementation

The memory controller connects to four 16 MB SDRAM chips. On the FPGA side, the memory controller has two interfaces which work with the bus protocol from [MP00, Sect. 5]. Hence, the memory controller acts as a protocol bridge between the [MP00] bus protocol and the SDRAM protocol. One of the interfaces is used for connecting the cache of the *VAMP* processor to the main memory. The other interface is used to access the *VAMP* main memory from the host PC. This is used to transfer programs and data between the host PC to the *VAMP* memory. Simultaneous memory accesses of the *VAMP* processor and the host PC are arbitrated by the memory controller.

The host connection and the memory controller are designed and tested by Dirk Leinenbach. The memory controller is based on an SDRAM controller from Isytec GmbH, Germany, which also sells the PCI board. The host connection and the memory controller are not formally verified. The formal verification of the *VAMP* processor reaches only to the memory-boundary. One could also verify the auxiliary circuits; however, this is currently not intended for the *VAMP* project.

Cost and Delay. The complete *VAMP* processor occupies 18053 slices, which accounts for 94% of the FPGA area. The maximum clock frequency is $\sim 10MHz$. The *VAMP* processor has 9109 bits of registers, not counting the register file and caches.

Testing the Implementation. In order to test the *VAMP* implementation, we have ported the GNU C compiler *gcc* and the C library *glibc* for the *VAMP* processor. We are currently testing the *VAMP* processor. The processor is already capable of running simple test programs. We are currently running more complex tests on the *VAMP* processor. Since the processor is not yet fully verified, we are simultaneously debugging the *VAMP* using conventional debugging methods and using PVS.

The software tests of the *VAMP* processor are performed by Carsten Meyer.

6.6 Related Work

There are several other microprocessor verification projects in the formal methods community. Hunt and Brock have verified the FM9001 microprocessor [HB92]. The verification is performed on the net-list level. The FM9001 was the first formally verified processor which has been implemented based on the formal specification. Brock and Hunt claim that they have never encountered a situation where the processor implementation deviates from its specification.

Other verified processors include Windley's AVM-1 processor [Win95], and the AAMP5 processor by Miller and Srivas [MS95]. More recent projects, e.g, by Hunt and Sawada [HS99, Saw99], and Hosabettu et al. [Hos99, HGS00], verify processors on higher levels of abstraction, and hence do not yield actual implementations.

We are not aware of any formally verified processor of the complexity of the *VAMP* processor, in particular with caches and floating point units which are common in today's microprocessors. So, our group is the first which has verified and implemented such a complex microprocessor.

Chapter 7

Summary, Discussion and Future Work

In this chapter we give a short summary of the thesis, discuss our approach to the formal verification and implementation of complex hardware systems, and provide an outlook to possible future research.

Section 7.2 is based on [BJKL02, Sect. 6].

7.1 Summary

This thesis covers the formal verification of a fully IEEE compliant floating point unit in the theorem proving system PVS. The FPU supports addition, subtraction, multiplication, division, comparsion, conversion and various other operations. Both single and double precision numbers are supported. Denormal numbers are completely handled in hardware. The exception signals are computed as required by the IEEE standard.

In chapter 3 we presented a formalization of the IEEE standard in PVS. We have verified concepts and theorems which ease the verification of the actual hardware. Most notably, the decomposition theorem of rounding allows for the decomposition of the rounding hardware into smaller parts, and the concept of α-equivalence makes the decomposition of the FPU into computational and rounding unit possible.

In chapter 4 we have verified the actual floating point hardware with respect to the specification from chapter 3. We have separately verified unpacker, rounder, and computational units, and then combined these units to the complete FPUs. Decomposition of the system into smaller parts based on the theorems from chapter 3 considerably simplified the verification, since the parts could be verified separately with respect to local specifications. The parts were then re-composed, and the local specifications were proved to yield the desired overall specification.

In chapter 5 we have pipelined the combinatorial FPUs from chapter 4. The

FPUs have been prepared to work in the out-of-order *VAMP* processor. To exploit the out-of-order scheduler, the pipelines may process multiple instructions simulatiously, have variable latency, and may reorder instructions internally. The pipelines have branches and cycles in the pipeline structure, e.g., for the iterative division algorithm. We have presented a new methodology based on the combination of model checking and theorem proving for the verification of such complex pipelines.

The verification of the FPUs is part of the *VAMP* project at Saarland University. Chapter 6 provided a detailed project description and status of the *VAMP* project.

7.2 Discussion

Advantages. We have presented the design and verification of complex hardware in PVS. Using our tool *pvs2hdl* we have translated the hardware to Verilog and implemented it on a Xilinx FPGA. There are several benefits to this approach compared to the traditional way of hardware design and verification using Verilog and testing:

1. The use of high-level constructs such as recursion and λ-expressions allows for the concise description of structured hardware.

2. The description of hardware in PVS enables the formal verification of the hardware descriptions against a formal specification.

3. The PVS system offers support for both theorem proving as well as model checking. Thus, we can exploit both techniques in our proofs without a tedious and error-prone translation between two different verification systems.

4. The verification can exploit the structured and modular description of the hardware; one can verify general purpose circuits for arbitrary bit widths, and use the correctness results in the verification of larger and larger circuits. In this way, it is possible to design, verify and implement hardware of almost arbitrary complexity.

The latter points are particularly important, as the design of complex hardware systems is very error prone, and verification is therefore an increasingly important part of the development cycle. The feasibility of our approach is proved by the verification presented in this thesis.

The verification heavily exploits the structure of the hardware. We have verified a library of parameterized general purpose circuits [BJK01a], upon which hierarchically more and more complex circuits are built and verified. The verification of each circuit uses the correctness statements of its sub-circuits, so that the hierarchy is maintained during verification. This considerably eases the verification task.

The hardware is specified and verified in PVS on the gate level. In order to obtain real hardware, we have developed the *pvs2hdl* tool to automatically translate the PVS hardware descriptions to Verilog. Several other tools (synthesizer, place & route tools, etc.) then transform Verilog to real hardware. Each of the steps involved is not formally verified and could introduce new errors into the design. In fact, even the PVS proof checker could have bugs which hide errors in the "verified" hardware.

However, there is a great benefit in having verified the PVS gate-level description of the hardware: the design is free of *logical* errors (if we have not been trapped by bugs in PVS). Nowhere an *and*-gate is used where an *or*-gate would have been correct, no adder is too small in size, no entry in lookup tables is missed, etc. Although each of the tools mentioned above could introduce new errors, the confidence in the logical correctness of the gate-level greatly improves the confidence in the correctness of the ultimate hardware. The famous Pentium bug, for example, was a logical bug where an entry in a lookup table was omitted [Pra95], which would have been discovered in our verification. There are approaches to verified synthesis tools [AL95, ML01]. However, the formal verification of real-size synthesis tools is far beyond the capabilities of current software verification techniques.

The FPU verified in this thesis has been implemented on a Xilinx FPGA. The FPU worked on the first try. No debugging of the FPU circuits was necessary after having verified the gate-level in PVS. We have run hundred-thousands of test-vectors without discovering a bug in the *VAMP* FPU.

Drawbacks. There are drawbacks in the use of PVS as hardware development & verification system which we do not want to be left unmentioned:

1. Designing combinatorial circuits in a functional programming language and our notion of clocked circuits is not common practice for hardware designers.

2. The support for fast simulation and visualization is common in modern development systems, but not available in PVS. In the design phase, many obvious errors can be found by simulation. The harder errors could then be found during formal verification. The theorem prover ACL2 [KM96], for example, offers efficient LISP-based support for simulation. However, ACL2 cannot handle higher-order logic in contrast to PVS. The use of higher-order logic sometimes streamlines theories and hence simplifies the verification. This particularly applies to the definition of CTL and the verification of pipelines in chapter 5.

3. We support only a single clock domain. For example, we cannot directly model an SDRAM interface of a CPU where the SDRAM is clocked independently of the CPU. An extension of our PVS hardware model to cover multiple clock domains in the style of [AH01b] is possible, but we have not yet investigated this possibility.

4. Our PVS hardware model maps to a small subset of the Verilog hardware description language which is sufficient to design any combinatorial circuit or clocked circuit with one single clock domain. However, by designing hardware in PVS, we disallow any "dirty" design tricks employed in common HDLs in order to optimize the design. Therefore, it may not be possible to design hardware as thoroughly optimized for speed as contemporary commercial processors.

   However, it is not our project goal to compete with modern microprocessors in performance, but to offer formally verified correctness guarantees for microprocessors in safety-critical devices. Many of these safety-critical devices do not need a clock frequency of more than 100 MHz, which could be achieved by our approach in a full-custom process. We see a considerable market for formally verified microprocessor of comparably modest performance, e.g., in medical devices, nuclear reactors, and military applications.

5. A considerable part of the verification effort is needed for very low-level circuits for which appropriate automatic methods are available (see, e.g., [BC95, CB96]. One could save a great deal of time by automatically verifying small sub-circuits, and restrict interactive proof development to the composition of such sub-circuits to larger circuits which are too large for automatic verification. However, these automatic methods are not available in PVS.

There are publicly available tools supporting some of these features, but none integrates all features needed for an integrated development & verification system. There are such tools in industry, e.g. Intel's Forte system [OZGS99], but these tools are not publicly available, they are not even sold. In order to develop and formally verify large hardware systems against a high-level specification, we believe our approach is currently the only feasible that deploys only publically available tools.

7.3 Future Work

We see several directions in which the work presented in this thesis could be extended. First, modern floating point units in commercial processors support a variety of operations not considered in this thesis, e.g., transcendential functions. One could incorporate such operations into the *VAMP* FPUs.

As described above, the verification of the hardware implementation using theorem proving is very time-consuming and partly tedious. One could incorporate more automatic verification methods into the verification environment in order to ease future verification projects. Furthermore, one could extend the verification environment with debugging facilities such as simulation and visualization.

In order to further improve confidence in the correctness of the design, one could verify the PVS hardware specification against the netlist generated by the

Verilog synthesizer. This would close the verification gap involving our *pvs2hdl* tool and Verilog tools. The verification of the netlist could be performed using equivalence-checkers; however, these tools probably do not scale to the required circuit size. We have not investigated the verification of netlists yet.

As mentioned in section 5.2, we lack a general formalism for the design and presentation of complex out-of-order pipelines. One could try to enhance the formalism for in-order pipelines from [MP00, JK00, Kro01] to out-of-order pipelines. We have just started investigating the modeling of pipelines as (potentially cyclic) graphs, for which a pipeline control and its formal correctness proof should be automatically generated.

One could further enhance our verification methodology for out-of-order pipelines to cope with complex control/data dependencies, e.g., for self-timed division algorithms (cf. section 5.6.3). Furthermore, one could automate the "pushing through the pipeline" for liveness-verification.

Last, but by no means least, one could move the ladder one step up and verify system software running on the verified *VAMP* processor. Our group at Saarland University is currently setting up a project aiming for the formal verifcation of the L4 operation system micro-kernel [Lie95].

Appendix A

Floating Point Instruction Set

Table A.1 lists the op-codes of the floating point instructions supported by our FPUs. The *mov.s* and *mov.d* instructions move single respectively double precision floating point numbers from one register to another. The *mf2i* and *mi2f* instructions move 32 bit of a floating point register to an integer register or vice-versa, respectively. The *cmp.s* and *cmp.d* instructions perform comparisons between floating point numbers according to table A.2.

Table A.1: Op-codes of the supported floating point instructions.

Mnemonic	9-bit op-code	FPU
add.s	0x000	additive
add.d	0x040	additive
sub.s	0x001	additive
sub.d	0x041	additive
mul.s	0x002	multiplicative
mul.d	0x042	multiplicative
div.s	0x003	multiplicative
div.d	0x043	multiplicative
neg.s	0x004	misc
neg.d	0x044	misc
abs.s	0x005	misc
abs.d	0x045	misc
cmp.s*	0x03*	misc
cmp.d*	0x07*	misc
mov.s	0x008	misc
mov.d	0x048	misc
mf2i	0x009	misc
mi2f	0x00A	misc

continued on next page

continued from previous page		
Mnemonic	9-bit op-code	FPU
cvt.s2d	0x060	misc
cvt.s2i	0x120	misc
cvt.d2s	0x021	misc
cvt.d2i	0x121	misc
cvt.i2s	0x024	misc
cvt.i2d	0x026	misc

Table A.2: Encoding of the different comparisons.

Op-code bit	Comparison predicate
opcode[0]	*FCONun*
opcode[1]	*FCONeq*
opcode[2]	*FCONlt*
opcode[3]	*FCONgt*

Appendix B

Proof of Carry-Chain adder

This chapter presents a transcript of the PVS proof of theorem `cc_adder_correct` from chapter 2. The proof is virtually literally the same as the original PVS proof, we have only changed some minor points to increase readability. When starting the proof, PVS confronts us with the proof goal:

```
  |-------
[1]   FORALL (n: posnat, a, b: bvec[n], cin: bit):
        bv2nat(carry_chain_impl(n, a, b, cin)) =
         bv2nat(a) + bv2nat(b) + bv2nat(cin)
```

All formulas above the line (here empty) are called antecedents and may be seen as known facts, and the formulas below the line (here $[1]$) are called consequents. The disjunction of the consequents has to be proved from the conjunction of the antecedents. We start the proof by telling PVS to induct on the length n. We therefore issuing the command `(INDUCT "n":NAME "upfrom_induction[1]")`. The induction scheme to use is `upform_induction`, which tells PVS to start a natural number induction starting from 1.

```
Rule? (INDUCT "n" :NAME "upfrom_induction[1]")
Inducting on n on formula 1 using induction scheme
upfrom_induction[1],
this yields  2 subgoals:
cc_adder_correct.1 :

  |-------
[1]   FORALL (a, b: bvec[1], cin: bit):
        bv2nat(carry_chain(1, a, b, cin)) =
         bv2nat(a) + bv2nat(b) + bv2nat(cin)
```

Now PVS confronts us with 2 cases, the first of which is the induction base shown above. The induction base is proved by skolemizing the quantified a, b, and cin, expanding definitions, and case analysis. This is all done automatically by the PVS command `(grind)`:

```
Rule? (GRIND)
/= rewrites (a(0) /= b(0))
```

```
  to NOT (a(0) = b(0))
XOR rewrites a(0) XOR b(0)
....
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of cc_adder_correct.1.

cc_adder_correct.2 :

  |-------
[1]   FORALL (n: upfrom(1)):
        (FORALL (a, b: bvec[n], cin: bit):
           bv2nat(carry_chain(n, a, b, cin)) =
            bv2nat(a) + bv2nat(b) + bv2nat(cin))
         IMPLIES
         (FORALL (a, b: bvec[1 + n], cin: bit):
            bv2nat(carry_chain(n + 1, a, b, cin)) =
             bv2nat(a) + bv2nat(b) + bv2nat(cin))
```

PVS now presents the second sub-goal of the induction, namely the induction step. We start by skolemizing and flattening in order to yield a more readable goal. The skolemized variables are indicated by an appended '`!1`'.

```
Rule? (SKOSIMP*)
Repeatedly Skolemizing and flattening,
this simplifies to:
cc_adder_correct.2 :

[-1]  FORALL (a, b: bvec[n!1], cin: bit):
        bv2nat(carry_chain(n!1, a, b, cin)) =
         bv2nat(a) + bv2nat(b) + bv2nat(cin)
  |-------
[1]   bv2nat(carry_chain(n!1 + 1, a!1, b!1, cin!1)) =
       bv2nat(a!1) + bv2nat(b!1) + bv2nat(cin!1)
```

We have to prove formula $[1]$, the induction claim, from formula $[-1]$, the induction hypthesis. We first expand the definition of `carry_chain` in formula $[1]$:

```
Rule? (EXPAND "carry_chain" 1)
Expanding the definition of carry_chain,
this simplifies to:
cc_adder_correct.2 :

[-1]  FORALL (a, b: bvec[n!1], cin: bit):
        bv2nat(carry_chain(n!1, a, b, cin)) =
         bv2nat(a) + bv2nat(b) + bv2nat(cin)
  |-------
[1]   bv2nat(fulladder(a!1(n!1), b!1(n!1),
                carry_chain(n!1, a!1^(n!1 - 1, 0),
                            b!1^(n!1 - 1, 0), cin!1)(n!1))
           o carry_chain(n!1, a!1^(n!1 - 1, 0),
                         b!1^(n!1 - 1, 0), cin!1)^(n!1 - 1, 0))
       = bv2nat(cin!1) + bv2nat(a!1) + bv2nat(b!1)
```

Next we rewrite formula [1] with lemma `bv2nat_concat` (equation (2.4)). The `SUBST` parameter tells PVS to use `n!1` for `m` when instantiating the lemma. The other all-quantified variables of the lemma are instantiated automatically by PVS.

```
Rule? (REWRITE "bv2nat_concat" :SUBST ("m" "n!1"))
Rewriting using bv2nat_concat, matching in * where
  m gets n!1,
this simplifies to:
cc_adder_correct.2 :

[-1]  FORALL (a, b: bvec[n!1], cin: bit):
        bv2nat(carry_chain(n!1, a, b, cin)) =
         bv2nat(a) + bv2nat(b) + bv2nat(cin)
  |-------
[1]   bv2nat[n!1](carry_chain(n!1, a!1^(n!1 - 1, 0),
                      b!1^(n!1 - 1, 0), cin!1)^(n!1 - 1,0))
       + bv2nat[2](fulladder(a!1(n!1), b!1(n!1),
                             carry_chain(n!1, a!1^(n!1 - 1, 0),
                                 b!1^(n!1 - 1, 0), cin!1)(n!1)))
         * exp2(n!1)
       = bv2nat(cin!1) + bv2nat(a!1) + bv2nat(b!1)
```

We now rewrite formula [1] with the correctness lemma for the full adder:

```
Rule? (REWRITE "fa_correct")
Rewriting using fa_correct, matching in *,
this simplifies to:

cc_adder_correct.2 :

[-1]  FORALL (a, b: bvec[n!1], cin: bit):
        bv2nat(carry_chain(n!1, a, b, cin)) =
         bv2nat(cin) + bv2nat(a) + bv2nat(b)
  |-------
[1]   bv2nat[n!1](carry_chain(n!1, a!1^(n!1 - 1, 0),
                      b!1^(n!1 - 1, 0), cin!1)^(n!1 - 1, 0))
       +   bv2nat(carry_chain(n!1, a!1^(n!1 - 1, 0),
                      b!1^(n!1 - 1, 0), cin!1)(n!1)) * exp2(n!1)
       + bv2nat(a!1(n!1)) * exp2(n!1)
       + bv2nat(b!1(n!1)) * exp2(n!1)
       = bv2nat(cin!1) + bv2nat(a!1) + bv2nat(b!1)
```

We now want to re-combine the most-signiciant bit with the less signifcant bits of the `carry_chain` output. We therefore use lemma `bv2nat_split_top`, which is a specialication of lemma `bv2nat_concat`. However, this time PVS does not find the correct instantiation itself, and hence we have to employ the lemma first using the `LEMMA` command, and then instantiate it manually using the `INST` command:

```
Rule? (LEMMA "bv2nat_split_top")
Applying bv2nat_split_top
this simplifies to:
cc_adder_correct.2 :
```

```
[-1]  FORALL (n: above(1), b: bvec[n]):
        bv2nat(b) =
         bv2nat(b^(n - 2, 0)) + bv2nat(b(n - 1)) * exp2(n - 1)
[-2]  FORALL (a, b: bvec[n!1], cin: bit):
        bv2nat(carry_chain(n!1, a, b, cin)) =
         bv2nat(cin) + bv2nat(a) + bv2nat(b)
  |-------
[1]   bv2nat[n!1](carry_chain(n!1, a!1^(n!1 - 1, 0),
                      b!1^(n!1 - 1, 0), cin!1)^(n!1 - 1, 0))
       +   bv2nat(carry_chain(n!1, a!1^(n!1 - 1, 0),
                      b!1^(n!1 - 1, 0), cin!1)(n!1)) * exp2(n!1)
       + bv2nat(a!1(n!1)) * exp2(n!1)
       + bv2nat(b!1(n!1)) * exp2(n!1)
       = bv2nat(cin!1) + bv2nat(a!1) + bv2nat(b!1)

Rule? (INST -1 "n!1+1" "carry_chain(n!1, a!1^(n!1 - 1, 0),
                           b!1^(n!1 - 1, 0), cin!1)")
Instantiating the top quantifier in -1.
this simplifies to:
cc_adder_correct.2 :

[-1]  bv2nat(carry_chain(n!1, a!1^(n!1 - 1, 0),
                            b!1^(n!1 - 1, 0), cin!1))
       =
       bv2nat(carry_chain(n!1, a!1^(n!1 - 1, 0),
                            b!1^(n!1 - 1, 0), cin!1)
              ^(n!1 + 1 - 2, 0))
       +
       bv2nat(carry_chain(n!1, a!1^(n!1 - 1, 0),
                           b!1^(n!1 - 1, 0), cin!1)
                          (n!1 + 1 - 1))
        * exp2(n!1 + 1 - 1)
[-2]  FORALL (a, b: bvec[n!1], cin: bit):
        bv2nat(carry_chain(n!1, a, b, cin)) =
         bv2nat(cin) + bv2nat(a) + bv2nat(b)
  |-------
[1]   bv2nat[n!1](carry_chain(n!1, a!1^(n!1 - 1, 0),
                      b!1^(n!1 - 1, 0), cin!1)^(n!1 - 1, 0))
       +   bv2nat(carry_chain(n!1, a!1^(n!1 - 1, 0),
                      b!1^(n!1 - 1, 0), cin!1)(n!1)) * exp2(n!1)
       + bv2nat(a!1(n!1)) * exp2(n!1)
       + bv2nat(b!1(n!1)) * exp2(n!1)
       = bv2nat(cin!1) + bv2nat(a!1) + bv2nat(b!1)
```

Replacing formula $[-1]$ in $[1]$ from right to left and hiding $[-1]$ results in:

```
[-1]  FORALL (a, b: bvec[n!1], cin: bit):
        bv2nat(carry_chain(n!1, a, b, cin)) =
         bv2nat(cin) + bv2nat(a) + bv2nat(b)
  |-------
[1]   bv2nat(carry_chain(n!1, a!1^(n!1 - 1, 0),
                            b!1^(n!1 - 1, 0), cin!1))
       + bv2nat(a!1(n!1)) * exp2(n!1)
       + bv2nat(b!1(n!1)) * exp2(n!1)
       = bv2nat(cin!1) + bv2nat(a!1) + bv2nat(b!1)
```

We now apply the induction hypthesis from $[-1]$. PVS finds the correct instantiation automatically by the command `(INST?)`.

```
Rule? (inst?)
Found substitution:
cin: bit gets cin!1,
b: bvec[n!1] gets b!1^(n!1 - 1, 0),
a gets a!1^(n!1 - 1, 0),
Using template: bv2nat(carry_chain(n!1, a, b, cin))
Instantiating quantified variables,
this simplifies to:
cc_adder_correct.2 :

[-1]  bv2nat(carry_chain(n!1, a!1^(n!1 - 1, 0),
                            b!1^(n!1 - 1, 0), cin!1))
       =
       bv2nat(cin!1) + bv2nat(a!1^(n!1 - 1, 0)) +
        bv2nat(b!1^(n!1 - 1, 0))
  |-------
[1]   bv2nat(carry_chain(n!1, a!1^(n!1 - 1, 0),
                            b!1^(n!1 - 1, 0), cin!1))
       + bv2nat(a!1(n!1)) * exp2(n!1)
       + bv2nat(b!1(n!1)) * exp2(n!1)
       = bv2nat(cin!1) + bv2nat(a!1) + bv2nat(b!1)
```

In the next step, we replace the induction hypthesis in the induction claim.

```
Rule? (REPLACE -1 :HIDE? T)
Replacing using formula -1,
this simplifies to:
cc_adder_correct.2 :

  |-------
[1]   bv2nat(a!1^(n!1 - 1, 0)) + bv2nat(b!1^(n!1 - 1, 0)) +
       bv2nat(a!1(n!1)) * exp2(n!1) + bv2nat(b!1(n!1)) * exp2(n!1)
       = bv2nat(a!1) + bv2nat(b!1)
```

The last steps are to decompose `a!1` and `b!1` by lemma `bv2nat_split_top`.

```
Rule? (REWRITE "bv2nat_split_top" :SUBST ("n" "n!1+1"))
Found matching substitution:
b: bvec[n] gets a!1,
n: above(1) gets n!1 + 1,
Rewriting using bv2nat_split_top, matching in * where
  n gets n!1+1,
this simplifies to:
cc_adder_correct.2 :

  |-------
[1]   bv2nat(a!1^(n!1 - 1, 0)) + bv2nat(b!1^(n!1 - 1, 0)) +
       bv2nat(a!1(n!1)) * exp2(n!1) + bv2nat(b!1(n!1)) * exp2(n!1)
       =
       bv2nat(a!1^(n!1 - 1, 0)) + bv2nat(a!1(n!1)) * exp2(n!1) +
        bv2nat(b!1)
```

```
Rule? (REWRITE "bv2nat_split_top" :SUBST ("n" "n!1+1"))
Found matching substitution:
b: bvec[n] gets b!1,
n: above(1) gets n!1 + 1,
Rewriting using bv2nat_split_top, matching in * where
  n gets n!1+1,
this simplifies to:
cc_adder_correct.2 :

  |-------
[1]   bv2nat(a!1^(n!1 - 1, 0)) + bv2nat(b!1^(n!1 - 1, 0)) +
       bv2nat(a!1(n!1)) * exp2(n!1) + bv2nat(b!1(n!1)) * exp2(n!1)
       =
       bv2nat(a!1^(n!1 - 1, 0)) + bv2nat(a!1(n!1)) * exp2(n!1) +
        bv2nat(b!1^(n!1 - 1, 0)) + bv2nat(b!1(n!1)) * exp2(n!1)

This completes the proof of cc_adder_correct.2.
```

PVS features strategies such as automatic induction with simplification and rewriting which sometimes help shortening proofs like the presented above. However, most of the proofs of the hardware in this thesis are as detailed (and hence tedious) as the one presented.

Appendix C

Circuits, Theorems and Lemmas in PVS

The following tables list the filenames, PVS theory names, and the PVS names of the circuits, theorems and lemmas in this thesis. The PVS source files as at the time of handing in this thesis are available at `http://www-wjp.cs.uni-sb.de/~cj/PhD/`. The newest sources can be found at the *VAMP* homepage: `http://www-wjp.cs.uni-sb.de/projects/verification/`.

Table C.1: Theorems and Lemmas

Number in this thesis	Filename	PVS Theory	PVS Name
3.1	ieee/factoring.pvs	factoring	val_zero
3.2	ieee/factoring.pvs	fact_props	
3.3	ieee/factoring.pvs	compare_lem	
3.4	ieee/factoring.pvs	fact_props	i3e_factoring_unique
3.5	ieee/factorings.pvs	fact_norm	
3.6	ieee/factorings.pvs	eta	
3.7	ieee/factorings.pvs	eta_props	
3.8	ieee/factorings.pvs	eta_props	
3.9	ieee/factorings.pvs	eta_props	eta_nor, eta_denor
3.10	ieee/factorings.pvs	fin_precision, fin_exponent	
3.11	ieee/factorings.pvs	fin_precision	if_gaps3
3.12	ieee/factorings.pvs	fact_mul2	if_fact_mul2
3.13	ieee/round.pvs	round_decomposition	
3.14	ieee/round.pvs	round_decomposition	ii_postnorm_typ
3.15	ieee/round.pvs	round_decomposition	ii_postnorm_val
3.16	ieee/round.pvs	round_decomposition	ii_round_decomposition
3.17	ieee/round_props.pvs	round_props	round_denormal
3.18	ieee/round.pvs	round_correct	

continued on next page

continued from previous page

Number in this thesis	Filename	PVS Theory	PVS Name
3.19	ieee/round.pvs	round_correct	round_fix
3.20	ieee/except.pvs	unf_ovf	ovf_emax
3.21	ieee/except.pvs	ovf_unf	OVF_decomp
3.22	ieee/except.pvs	ovf_unf	LOSS_sigrd
3.23	ieee/alpha_equiv.pvs	alpha_lemmas	
3.24	ieee/alpha_equiv.pvs	alpha_lemmas	alpha_rep_large
3.25	ieee/alpha_equiv.pvs	alpha_rd_lem	
3.26	ieee/alpha_equiv.pvs	alpha_rd_lem	alpha_rep_eta
3.27	ieee/alpha_sticky.pvs	alpha_sticky	sticky_comp2
3.28	ieee/round_fact.pvs	round_fact2	rnd_repr
3.29	ieee/round_fact.pvs	round_fact2	rnd_repr_l1, rnd_repr_l2
3.30	ieee/round_fact.pvs	round_fact2	rnd_repr_l3a, rnd_repr_l3b
3.31	ieee/round_fact.pvs	round_fact2	rnd_repr_hat
3.32	ieee/sigrd.pvs	sigrd_props	sigrd_repr
3.33	ieee/except.pvs	ovf_unf	
3.34	ieee/wrapped_exp.pvs	wrapped_exp	rd_result_eq
3.35	ieee/inx.pvs	inx	INX_eq
3.36	ieee/nu_format.pvs	ieee_bv	ieeebvfact_is_representable3
3.37	ieee/nu_format.pvs	ieee_bv	fact2ieeebv
3.38	ieee/fpop_result.pvs	fpop_result	result_correct_equiv
3.39	ieee/compare.pvs	fp_compare	relation_cover, relation_exclusive
3.40	ieee/compare.pvs	compare_comp	less_compute
3.41	ieee/rd2int.pvs	rd2int	
3.42	ieee/rd2int.pvs	rd2int	rd2int_large
3.43	ieee/rd2int.pvs	rd2int	rd2int_small
3.44	ieee/rd2int.pvs	rd2int	rd2int_format
3.45	ieee/rd2int.pvs	rd2int	rd2int_lem2
3.46	ieee/rd2int.pvs	rd2int	rd2int_rewr
4.1	unpack/unpack.pvs	unpack	spec_unpack_impl_sgl
4.2	unpack/unpack.pvs	unpack	unpack_TCC1, unpack_impl_sgl_add
4.3	unpack/unpack.pvs	unpack	unpack_TCC1, unpack_impl_sgl_mul
4.4	unpack/unpack.pvs	fx_unpack	fx_unpack_correct
4.5	rounder/ns.pvs	ns_impl	ns_correct_tiny_ovf_sgl, ns_correct_sgl
4.6	rounder/ns_flags.pvs	flags_impl	flags_ovf1_sgl
4.7	ieee/sigrd.pvs	sigrd_props	ceil_lem, floor_lem
4.8	ieee/sigrd.pvs	sigrd_props	sigrd_impl_lem
4.9	rounder/postnorm.pvs	postnorm_stage	
4.10	rounder/exprd.pvs	exprd	exprd_correct_sgl
4.11	muldiv/ieee_md.pvs	ieee_md	
4.12	mul_div/lookup.pvs	lookup	lookup_div_correct

continued on next page

continued from previous page

Number in this thesis	Filename	PVS Theory	PVS Name
4.13	mul_div/div_initial.pvs	div_initial	initial_ok
4.14	mul_div/div_initial.pvs	div_initial	delta_bound_57
4.15	mul_div/mul_div_comb.pvs	mul_div_comb	div_comb_E, div_comb_quot_sgl
4.16	mul_div/div_rep.pvs	div_rep	div_rep_comp
4.17	mul_div/exp_md.pvs	exp_md	exp_md_mul, exp_md_div
4.18	mul_div/md_stg2.pvs	md_stg2	md_stg_smul, md_stg_dmul
4.19	mul_div/md_stg2.pvs	md_stg2	md_stg_sdiv, md_stg_ddiv
4.20	mul_div/md_comb.pvs	md_comb	smul_rdinp, sdiv_rdinp
4.21	mul_div/md_comb.pvs	md_comb	md_inf_times_zero_s
4.22	adder/fpadd.pvs	fpadd	fpadd_correct
4.23	ieee/add_zero.pvs	add_zero	
4.24	add/add_comb.pvs	add_comb	s_add_result, s_sub_result
4.25	add/add_comb.pvs	add_comb	add_sign_s
4.26	add/add_comb.pvs	add_comb	add_pinf_s
4.27	fp_misc/fp_misc_comb.pvs	cmp_sgl_correct	
4.28	fp_misc/fp_rd2int.pvs	fp_rd2int	rd2int_range_small, rd2int_range_large
4.29	fp_misc/fp_rd2int.pvs	fp_rd2int	rd2int_small_int_dbl
4.30	fp_misc/fp_rd2int.pvs	fp_rd2int	rd2int_extract_correct
4.31	fp_misc/fp_misc_comb.pvs	fp_misc_comb	cmp_sgl_correct
5.1	pvshdl/mutheorems.pvs	mutheorems	gfp_is_gfp, lfp_is_lfp
5.2	pvsctl/ctlpath.pvs	ctlpath	EG_thm1
5.3	pvsctl/ctlpath.pvs	ctlpath	EG_monoton
5.4	pvsctl/ctlpath.pvs	ctlpath	EG_lem1
5.5	pvsctl/ctlpath.pvs	ctlpath	EG_lem2
5.6	pvsctl/ctlpath.pvs	ctlpath	EG_lem3
5.7	pvsctl/ctlpath.pvs	ctlpath	EU_thm1
5.8	pvsctl/ctlpath.pvs	ctlpath	fEG_thm1
5.9	pvsctl/ctlpath.pvs	ctlpath	AG_thm
5.10	pvsctl/ctlpath.pvs	ctlpath	fairAF_thm
5.11	pvsctl/statetrans.pvs	statetrans	AG_path
5.12	pvsctl/statetrans.pvs	statetrans	fairAF_path
5.13	fpu/md_correct.pvs	md_correct	e.g. TOMmd_unpce_fintrue
5.14	fpu/md_correct	md_correct	e.g. TOMmd_mul_live
5.15	fpu/md_correct.pvs	md_correct	TOMmd_correct

Table C.2: Circuits

Number in this thesis	Circuit Name	Filename	PVS Name
4.1	FP-UNPACK	unpack/unpack.pvs	unpack_impl, spec_unpack_impl
4.2	FXUNPACK	unpack/fx_unpack.pvs	fx_unpack
4.3	FP-ROUNDER	rounder/rd_stg.pvs	

continued on next page

continued from previous page

Number in this thesis	Circuit Name	Filename	PVS Name
4.4	ETA-COMP	rounder/ns.pvs	ns_impl
4.5	REP, SIGRD, POSTNORM	rounder/repp.pvs, rounder/sigrd.pvs, rounder/postnorm.pvs	
4.6	ADJUSTEXP, PACK, EXPRD	rounder/adjustexp.pvs, rounder/pack.pvs, rounder/exprd.pvs	
4.7	DIV-LOOKUP	mul_div/div_initial.pvs	initial_impl
4.8	MD-CORE	mul_div/md_stg1.pvs, mul_div/md_stg2.pvs	md_stg1, md_stg2
4.9	EXPMD	mul_div/exp_md.pvs	exp_md
4.10	SELECTFD	mul_div/select_fd.pvs	select_fd
4.11	MD-COMB	mul_div/md_comb.pvs	md_comb
4.12	FP-ADDER	adder/fpadder.pvs	fpadder
4.13	ADD-COMB	add/add_comb.pvs	add_comb
4.14	FP-MISC	fp_misc/fp_misc_comb.pvs	fp_misc_comb
4.15	FP-COMPARE	fp_misc/fp_compare.pvs	fp_compute_fcc
5.1	MD-PIPE	fpu/md_synth.pvs	md_synth

Appendix D

Multiplicative Pipeline Control in SMV

```
scalarset tagT undefined;

typedef md_stateT
    mul, div2_1, div2_0, div1_1, div1_0, div0_1,
          div0_0, div_E, div_Eb, sel_fd ;


module main(val_in, muldiv_in, double_in, special_in,
  tag_in, stall_in, val_out, tag_out, stall_out)

input val_in,stall_in : boolean;
input muldiv_in: boolean; /* TRUE=div */
input double_in: boolean; /* TRUE=double */
input special_in: boolean; /* TRUE=special operands */
input tag_in: tagT;

output val_out, stall_out: boolean;
output tag_out: tagT;


/* defined below */
unp_ce: boolean;
md1_ce: boolean;
md2_ce: boolean;
selfd_ce: boolean;
rd1_ce: boolean;


/*************************************/
/* UNPACKER/LOOKUP                   */
/*************************************/
```

```
/* full bits indicate, that the stage contains a valid
   instruction */

unp_full: boolean;
unp_tag: tagT;
unp_state: md_stateT;
unp_special: boolean;

init(unp_full):=FALSE;
next(unp_full):=unp_ce ? val_in : unp_full;
next(unp_tag) :=unp_ce ? tag_in : unp_tag;
next(unp_state):=unp_ce ?
          (~muldiv_in ? mul :
               (double_in ? div2_1 : div1_1))
          : unp_state;
next(unp_special):=unp_ce ? special_in : unp_special;

stall_out := ~unp_ce;

/*************************************/
/* Mul/Div1                          */
/*************************************/

md1_sel: boolean; /* COMB */

md1_full: boolean;
md1_tag: tagT;
md1_state: md_stateT;

md1_sel := md2_full & ~(md2_state=mul | md2_state=sel_fd);
/* md1_sel <=> feedback */

init(md1_full):=FALSE;
next(md1_full):= md1_ce ?
  (md1_sel | (unp_full & ~unp_special)) : md1_full;
next(md1_tag) := md1_ce ?
  (md1_sel ? md2_tag : unp_tag) : md1_tag;
next(md1_state):=md1_ce ?
  (md1_sel ? md2_state : unp_state): md1_state;

/*************************************/
/* Mul/Div2                          */
/*************************************/

nxtstate: md_stateT; /* COMB */

md2_full: boolean;
md2_tag: tagT;
md2_state: md_stateT;
```

```
nxtstate:=case       /* this is decrement */
         md1_state = mul : mul;
         md1_state = div2_1: div2_0;
         md1_state = div2_0: div1_1;
         md1_state = div1_1: div1_0;
         md1_state = div1_0: div0_1;
         md1_state = div0_1: div0_0;
         md1_state = div0_0: div_E;
         md1_state = div_E: div_Eb;
         md1_state = div_Eb : sel_fd;
         md1_state = sel_fd : sel_fd; ;

init(md2_full):=FALSE;
next(md2_full):=md2_ce ? md1_full : md2_full;
next(md2_tag) :=md2_ce ? md1_tag : md2_tag;
next(md2_state):=md2_ce ? nxtstate : md2_state;


/*************************************/
/* Select FD                         */
/*************************************/

selfd_full: boolean;
selfd_tag: tagT;

init(selfd_full):=FALSE;
next(selfd_full):=selfd_ce ?
  (md2_full & md2_state=sel_fd) : selfd_full;
next(selfd_tag):=selfd_ce ? md2_tag : selfd_tag;


/*************************************/
/* Round 1                           */
/*************************************/

rd1_full: boolean;
rd1_tag: tagT;

init(rd1_full):=FALSE;
next(rd1_full):=rd1_ce ?
  (selfd_full | md2_full & md2_state=mul) : rd1_full;
next(rd1_tag):=rd1_ce ?
  (selfd_full ? selfd_tag : md2_tag) : rd1_tag;

/*************************************/
/* Round 2/Output                    */
/*************************************/
```

```
out_sel: boolean; /* COMB */

out_sel:= ~rd1_full & unp_full & unp_special;

val_out := ~stall_in & (out_sel | rd1_full);
tag_out := out_sel ? unp_tag : rd1_tag;

/*************************************/
/* Clock Enables                     */
/*************************************/

rd1_ce := val_out | ~rd1_full;
selfd_ce :=  rd1_ce | ~selfd_full;
md2_ce := (md2_state=mul & rd1_ce & ~selfd_full) |
          (md2_state=sel_fd & selfd_ce) |
         ~(md2_state=mul | md2_state=sel_fd) | ~md2_full;
md1_ce := md2_ce | ~md1_full;
unp_ce := (md1_ce & ~md1_sel & ~unp_special) |
          (val_out & out_sel) |
          ~unp_full;


/************************************************/
/************************************************/
/************* Specifications/Lemmas ************/
/************************************************/
/************************************************/

/* LIVENESS */

stall_fair: assert G F ~stall_in;
assume stall_fair;

forall(i in tagT)
  live[i]: assert
    G ((val_in & tag_in=i & ~stall_out) ->
      F (val_out & tag_out=i));

  using stall_fair prove live[i];


/* TAG-CONSISTENCY */

tagtable: array tagT of boolean;

forall(i in tagT)

  init(tagtable[i]):=FALSE;
  next(tagtable[i]):=(val_in & tag_in=i) |
```

```
                    (tagtable[i] & ~(val_out &tag_out=i));

  tagunique[i]: assert
    G ((val_in & tag_in=i) -> ~tagtable[i]);
  assume tagunique[i];

  cons1[i]: assert G ((val_out & tag_out=i) -> tagtable[i]);
  using tagunique[i] prove cons1[i];



/* MISC */
fu_valid_out_correct: assert G (val_out -> ~stall_in);
fu_stall_outfintr: assert G F ~stall_out;
using stall_fair prove fu_stall_outfintr;
```

Bibliography

[ADG+01] A. Adams, M. Dunstan, H. Gottliebsen, T. Kelsey, U. Martin, and S. Owre. Computer Algebra meets Automated Theorem Proving: Integrating Maple and PVS. In *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001*, volume 2152 of *LNCS*, pages 27–42. Springer, 2001.

[AH01a] A. T. Abdel-Hamid. A hierarchical verification of the IEEE-754 table-driven floating-point exponential function using HOL. Master's thesis, Dpt. Electrical and Computer Engineering, Concordia University, Montréal, Québec, Canada, 2001.

[AH01b] A. R. Albrecht and A. J. Hu. Register transformations with multiple clock domains. In *Proc. 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *LNCS*, pages 126–139. Springer, 2001.

[AHTH01] 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, editors, *TPHOLs 2001: Supplemental Proceedings*, 2001. Informatics Research Report EDI-INF-RR-0046, Univ. Edinburgh, UK.

[AJK00] M. D. Aagaard, R. B. Jones, and R. Kaivola. Formal verification of iterative algorithms in microprocessors. In *Design Automation Conference (DAC) 2000*. ACM, 2000.

[AL94] M. D. Aagaard and M. Leeser. Reasoning about pipelines with structural hazards. In *Theorem Provers in Circuit Design (TPCD'94)*, volume 901 of *LNCS*. Springer, 1994.

[AL95] M. D. Aagaard and M. Leeser. Verifying a logic-synthesis algorithm and implementation: A case study in software verification. *IEEE Trans. on Software Engineering*, 21(10), Oct 1995.

[AS95] M. D. Aagaard and C.-J. H. Seger. The formal verification of a pipelined double-precision IEEE floating-point multiplier. In *ICCAD*, pages 7–10. IEEE, November 1995.

[Bar89] G. Barrett. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering*, 15(5):611–621, May 1989.

[Bar90] H. P. Barendregt. Functional programming and lambda calculus. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pages 321–363. Elsevier, Amsterdam, 1990.

[BBCZ98] S. Berezin, A. Biere, E. Clarke, and Y. Zhu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In *FMCAD '98*, LNCS 1522. Springer, 1998.

[BBJ+02] C. Berg, S. Beyer, C. Jacobi, D. Kröning, and D. Leinenbach. Formal verification of the VAMP microprocessor (project status). Technical report, Max-Planck-Institut für Informatik, April 2002.

[BC95] R. E. Bryant and Y.-A. Chen. Verification or Arithmetic Circuits with Binary Moment Diagrams. In *32nd ACM/IEEE Design Automation Conference*, June 1995.

[BD94] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *CAV'94*, LNCS 818. Springer, 1994.

[Ber01] C. Berg. Formal verification of an IEEE floating point adder. Master's thesis, Saarland University, Computer Science Department, May 2001.

[Bey02] S. Beyer. Formal verification of a cache memory interface. submitted for publication, 2002.

[BGV01] R. E. Bryant, S. German, and M. N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Trans. on Computational. Logic (TOCL)*, 2(1):1–41, Jan 2001.

[BJ01] 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*, pages 325–339. Springer, 2001.

[BJK01a] C. Berg, C. Jacobi, and D. Kröning. Formal verification of a basic circuits library. In *Proc. of the IASTED International Conference on Applied Informatics, Innsbruck (AI 2001)*. ACTA Press, 2001.

[BJK01b] C. Berg, C. Jacobi, and D. Kröning. Formal verification of the VAMP mircoprocessor (project status). Unpublished, available at `http://www-wjp.cs.uni-sb.de/~cj/vamp-status.ps`, April 2001.

[BJKL02] S. Beyer, C. Jacobi, D. Kroening, and D. Leinenbach. Correct hardware by synthesis from PVS. submitted for publication, 2002.

[BMS+96] R. W. Butler, P. S. Miner, M. K. Srivas, D. A. Greve, and S. P. Miller. A bitvectors library for PVS. Technical Report TM-110274, NASA Langley Research Center, 1996.

[Bry96] R. E. Bryant. Bit-level analysis of an SRT divider circuit. In *33rd Design Automation Conference (DAC'96)*, pages 661–665. ACM, June 1996.

[CB96] Y.-A. Chen and R. E. Bryant. ACV: An arithmetic circuit verifier. In *Proc. of IEEE ICCD '96*, pages 361–365. IEEE, 1996.

[CB98] Y.-A. Chen and R. E. Bryant. Verification of floating point adders. In *CAV'98*, volume 1427 of *LNCS*, 1998.

[CCH+96] 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*, pages 19–33. Springer, 1996.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, Massachusetts, 1999.

[CGZ96] 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.

[CH98] M. Cornea-Hasegan. Proving the IEEE correctness of iterative floating-point square root, divide, and remainder algorithms. *Intel Technology Journal*, Q2, 1998.

[CH99] M. Cornea-Hasegan. IA-64 floating point operations and the IEEE standard for binary floating-point arithmetic. *Intel Technology Journal*, Q4, 1999.

[Cil99] M. D. Ciletti. *Modeling, Synthesis, and Rapid Prototyping with the VERILOG HDL*. Prentice Hall, 1999.

[CL93] J. Cortadella and T. Lang. Division with speculation of quotient digits. In *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 87–94. IEEE, June 1993.

[Cli90] W. D. Clinger. How to read floating-point numbers accurately. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 92–101, 1990.

[Coo80] J. T. Coonen. An implementation guide to a proposed standard for floating point arithmetic. *COMPUTER*, 13(1):68–79, January 1980.

[COR+95] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, April 1995. Available, with specification files, at `http://www.csl.sri.com/wift-tutorial.html`.

[CS95] R. P. Colwell and R. L. Steck. A 0.6um bicmos processor employing dynamic execution. International Solid State Circuits Conference (ISSCC), 1995.

[DV02] B. L. Di Vito. *Manip: A PVS Prover Strategy Package for Common Manipulations*. NASA Langley Research Center, Hampton, VA, 2002. availanle at `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html`.

[EC80] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Automata, Languages and Programming*, LNCS 85. Springer, 1980.

[EP97] 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.

[Gen35] G. Gentzen. Untersuchungen über das logische Schließen. In *Mathematische Zeitschrift*, volume 1, pages 176–210, 1935.

[GM93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[Gol96] D. Goldberg. Computer arithmetic. *In [HP96]*, 1996.

[Gol98] R. Golliver. Efficiently producing default orthogonal IEEE double results using extended IEEE hardware. Talk at 3rd Meeting of the Java Study Group, 1998. available as `http://std.dkuug.dk/JTC1/SC22/JSG/docs/m3/docs/jsgn326.pdf`.

[Har97] J. Harrison. Floating point verification in HOL light: The exponential function. In *Algebraic Methodology and Software Technology*, pages 246–260, 1997.

[Har99] J. Harrison. A machine checked theory of floating point arithmetic. In *TPHOLs '99*, volume 1690 of *LNCS*. Springer, 1999.

[HB92] W. A. Hunt and B. C. Brock. A formal HDL and its use in the FM9001 verification. In *Mechanized Reasoning and Hardware Design*, pages 35–47. Prentice Hall International, 1992.

[HD85] F. K. Hanna and N. Daeche. Specification and verification using higher-order logic. In Koomen and Moto-oka, editors, *Computer Hardware Description Languages and their Applications*, pages 418–433. North Holland, unknown 1985.

[HGS00] R. Hosabettu, G. Gopalakrishnan, and M. Srivas. Verifying microarchitectures that support speculation and exceptions. In *CAV '00*, volume 1855 of *LNCS*. Springer, 2000.

[HIK98] P.-H. Ho, A. J. Isles, and T. Kam. Formal verification of pipeline control using controlled token nets and abstract interpretation. In *ICCAD-98*. ACM, 1998.

[Hos99] R. Hosabettu. *Systematic Verification of Pipelined Microprocessors*. PhD thesis, Department of Computer Science, University of Utah, 1999.

[HP96] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[HS99] W. A. Hunt, Jr. and J. Sawada. The FM9801 microprocessor verification. *IEEE Micro*, 19(3):47–55, May/June 1999.

[IBM00] IBM. *z/Architecture Principles of Operation*. Poughkeepsie, NY, December 2000.

[ID96] C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1–2):41–75, 1996.

[IEEE] Institute of Electrical and Electronics Engineers. *ANSI/IEEE standard 754–1985, IEEE Standard for Binary Floating-Point Arithmetic*, 1985.

[Jac01] C. Jacobi. Formal verification of a theory of IEEE rounding. In R. J. Boulton and P. B. Jackson, editors, *TPHOLs 2001: Supplemental Proceedings*, 2001. Informatics Research Report EDI-INF-RR-0046, Univ. Edinburgh, UK.

[Jac02] C. Jacobi. Formal verification of complex out-of-order pipelines by combining model-checking and theorem-proving. accepted for Computer Aided Verification (CAV), to appear, 2002.

[JK00] C. Jacobi and D. Kroening. Proving the correctness of a complete microprocessor. In *GI Jahrestagung 2000*. Springer, 2000.

[KH92] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[KK01] R. Kaivola and K. Kohatsu. Proof engineering in the large: Formal verification of the 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.

[KM96] M. Kaufmann and J. S. Moore. ACL2: An industrial strength version of Nqthm. In *Compass'96: Eleventh Annual Conference on Computer Assurance*, page 23, Gaithersburg, Maryland, 1996. National Institute of Standards and Technology.

[KMP99] D. Kröning, S. M. Müller, and W. Paul. A rigorous correctness proof of the Tomasulo scheduling algorithm with precise interrupts. In *Proc. of the SCI'99/ISAS'99 International Conference*, 1999.

[KO63] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7, 1963.

[KP01] D. Kröning and W. Paul. Automated pipeline design. In *Proc. of 38th Design Automation Conference (DAC)*, pages 810,815, 2001.

[Kro01] D. Kroening. *Formal Verification of Pipelined Microprocessors*. PhD thesis, Saarland University, Computer Science Department, 2001.

[KS97] D. Kapur and M. Subramaniam. Mechanizing verification of arithmetic circuits: SRT division. In *FSTTCS*, volume 1346 of *LNCS*, pages 103–, 1997.

[KSK93] R. Kumar, K. Schneider, and T. Kropf. Structuring and automating hardware proofs in a higher-order theorem-proving environment. *Formal Methods in System Design*, 2(2):165–223, 1993.

[Lee89] C. Lee. Multistep gradual rounding. *IEEE Transactions on Computers*, 38(4), 1989.

[Lie95] J. Liedtke. On micro-kernel construction. In *Symposium on Operating Systems Principles*, pages 237–250, 1995.

[McM93] K. L. McMillan. *Symbolic model checking*. Kluwer, 1993.

[McM00] K. L. McMillan. A methodology for hardware verification using compositional model checking. *Science of Computer Programming*, 37(1-3):279–309, 2000.

[Mel93] T. Melham. *Higher Order Logic and Hardware Verification*, volume 31 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1993.

[Min95] P. S. Miner. Defining the IEEE-854 floating-point standard in PVS. Technical Report TM-110167, NASA Langley Research Center, 1995.

[ML96] P. S. Miner and J. F. Leathrum. Verification of IEEE compliant subtractive division algorithms. In *FMCAD-96*, volume 1166 of *LNCS*, pages 64–, 1996.

[ML01] S. McKeever and W. Luk. Towards provably-correct hardware compilation tools based on pass separation techniques. In *Correct Hardware Design and Verification Methods CHARME 2001*, volume 2144 of *LNCS*. Springer, 2001.

[MLD+99] S. M. Mueller, H. Leister, P. Dell, N. Gerteis, and D. Kroening. The impact of hardware scheduling mechanisms on the performance and cost of processor designs. In *15th GI/ITG Conference 'Architektur von Rechensystemen' ARCS'99*, pages 65–73. VDE Verlag, 1999.

[MLK98] J. S. Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the AMD5K86 floating point division program. *IEEE Transactions on Computers*, 47(9):913–926, 1998.

[Mot97] PowerPC 750 RISC Microprocessor Technical Summary, Motorola Inc., 1997.

[MP00] S. M. Mueller and W. J. Paul. *Computer Architecture. Complexity and Correctness*. Springer, 2000.

[MPK00] S. M. Müller, W. Paul, and D. Kröning. Proving the correctness of processors with delayed branch using delayed PC. In I. Althoefer et al., editor, *Proc. Symposium on Numbers, Information and Complexity, Bielefeld*, pages 579–588. Kluwer, 2000.

[MS95] S. P. Miller and M. Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *Proceedings of the Workshop on Industrial Strength Formal Specification Techniques (WIFT'95)*, Boca Raton, Florida, 1995.

[OF97] S. F. Oberman and M. J. Flynn. Division algorithms and implementations. *IEEE Transactions on Computers*, 46(8):833–854, 1997.

[O'K97] M. O'Keefe. A GCC machine description for DLX. available at `http://www-mount.ee.umn.edu/~okeefe/mcerg/gcc-dlx.html`, 1997.

[OSR92] S. Owre, N. Shankar, and J. M. Rushby. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer, 1992.

[OSRSC99a] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[OSRSC99b] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[OZGS99] 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.

[Pra95] V. R. Pratt. Anatomy of the pentium bug. In *TAPSOFT'95*, volume 915, pages 97–107. Springer-Verlag, 1995.

[Pre02] J. Preiß. *Optimal Pipeline Depth of Out-of-order RISC processors*. PhD thesis, Saarland University, 2002. Draft.

[RSS95] S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking with automated proof checking. In *CAV'95*, volume 939. Springer, 1995.

[RSS96] H. Ruess, N. Shankar, and M. K. Srivas. Modular verification of SRT division. In *CAV'96*, volume 1102 of *LNCS*, 1996.

[Rus98] D. M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998.

[Rus99] D. M. Russinoff. A mechanically checked proof of correctness of the AMD K5 floating point square root microcode. *Formal Methods in System Design*, 14(1):75–125, January 1999.

[Rus00] 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.

[Saw99] J. Sawada. *Formal Verification of an Advanced Pipelined Machine*. PhD thesis, University of Texas at Austin, December 1999. Also available from http://www.cs.utexas.edu/users/sawada/dissertation/-diss.html.

[SH98] J. Sawada and W. A. Hunt, Jr. Processor verification with precise exceptions and speculative execution. In *CAV '98*, volume 1427 of *LNCS*. Springer, 1998.

[SH99] K. Schneider and D. W. Hoffmann. A HOL conversion for translating linear time temporal logic to ω-automata. In *TPHOLs 99*, volume 1690 of *LNCS*. Springer, 1999.

[SORSC99] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[SP88] J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, 1988.

[SRC97] M. Srivas, H. Rueß, and D. Cyrluk. Hardware verification using PVS. In T. Kropf, editor, *Formal Hardware Verification: Methods and Systems in Comparison*, volume 1287 of *Lecture Notes in Computer Science*, pages 156–205. Springer-Verlag, 1997.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11 (1):25–33, 1967.

[VB00] M. N. Velev and R. E. Bryant. Formal verification of superscalar microprocessors with multicycle functional units, exception, and branch prediction. In *DAC '00*. ACM/IEEE, 2000.

[VCDM94] D. Verkest, L. Claesen, and H. De Man. A proof on the nonrestoring division algorithm and its implementation on an ALU. *Formal Methods in System Design*, 4, 1994.

[Ver96] Institute of Electrical and Electronics Engineers. *IEEE Standrard 1364-1995 Hardware Description Language Based on the Verilog Hardware Description*, 1996.

[Wal64] C. S. Wallace. A suggestion for a fast multiplier. *IEEE Trans. on Electronic Comp.*, EC-13(1):14–17, 1964.

[Win95] P. J. Windley. Formal modeling and verification of microprocessors. *IEEE Transactions on Computers*, 44(1):54–72, 1995.

[Xil02] Xilinx, Inc. *Virtex-E Data Sheet*, 2002. available at `http://www.xilinx.com/partinfo/ds022.htm`.