FORMAL METHODS

APPLIED TO A

FLOATING POINT NUMBER SYSTEM

by

Geoff Barrett

Technical Monograph PRG-58

January 1987

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

Abstract

This report presents a formalisation of the IEEE standard for binary floating-point arithmetic and proofs of procedures to perform non-exceptional arithmetic calculations.

Contents

Introduction

The main aim of a standard is that "conforming" implementations should behave in the manner specified – it is, therefore, desirable that they should be proved to do so. It has long been argued that natural language specifications can be ambiguous or misleading and, furthermore, that there is no formal link between specification and program. This report sets out to formalise the standard defined in [IEEE] and present algorithms to perform the non-exceptional arithmetic operations. Conversions between binary and decimal formats and delivery of bias adjusted results in trapped underflows are not covered.

The notations used in this paper are **Z** (see [Abrial,Hayes,Z]) and OCCAM (see [inmos]). The meaning of each new piece of **Z** is explained in a footnote before an example of its use.

Using a formal specification language bridges the gap between natural language specification and implementation. Natural language specifications have two disadvantages: they can be ambiguous; and it is difficult to show their consistency. The first problem is considered to be an important source of software and hardware errors and is eliminated completely by a formal specification. Further, it is important to show that a specification is consistent (i.e. has an implementation) for obvious reasons.

Of course, it could be argued that an implementation of a solution provides a precise specification of a problem. While this is true, no one likes to read other peoples' code and the structure of a program is designed to be read by machine and not by humans. Moreover, any flexibility in the approach to the problem is hampered by the need to make concrete design decisions. Specification languages are structured in such a way that they can reflect the structure of a problem or a natural language description or even of a program. But, above all, they can also be ***non-algorithmic***. This means that one can formalise what one has to do without detailing how it is to be done.

A formal development divides the task of implementing a specification into four well-defined steps. The first is to write a formal specification using mathematics. In the second, this specification is decomposed into smaller specifications which can be recombined in such a way that it can be shown formally that the decomposition is valid. Third, programs are written to satisfy the decomposed specifications. And, lastly, program transformations can be applied to make the program more efficient or, possibly, to adapt it for implementation on particular hardware configurations.

The example presented here is part of a large body of work which has been undertaken to formally develop a complete floating-point system. This work has been taken further by David Shepherd to transform the resulting routines into a software model of the inmos IMST800 processor, and so specify its functions. Thus, the development process has been carried through from formal specification to silicon implementation.

References of the form, e.g., p.14 §6.3 are to [IEEE].

Chapter 1

Specification

1.1 Rounding

This section presents a formal description of floating-point numbers and how they are used to approximate real numbers. The description serves as a specification for a rounding procedure.

First, floating-point numbers and their representation are described. Each number has a format. This consists of the exponent and fraction widths and other useful constants associated with these – the minimum and maximum exponent and the bias: [1]

Format

$$
\begin{array}{ll}
\textit{expwidth}, \textit{fracwidth} & : \mathbb{N} \\
\textit{wordlength} & : \mathbb{N} \\
\textit{EMin}, \textit{EMax}, \textit{Bias} & : \mathbb{N}
\end{array}
$$

$$
\begin{array}{ll}
\textit{wordlength} & = \textit{expwidth} + \textit{fracwidth} + 1 \\
\textit{EMin} & = 0 \\
\textit{EMax} & = 2^{\textit{expwidth}} - 1 \\
\textit{Bias} & = 2^{\textit{expwidth}-1} - 1
\end{array}
$$

Four formats are specified – the exponent width and wordlength are constrained to have particular values:

$$
\begin{array}{lll}
\textit{Single} & \widehat{=}\ \textit{Format} \mid \textit{expwidth} = 8\ \wedge \textit{wordlength} = 32 \\
\textit{Double} & \widehat{=}\ \textit{Format} \mid \textit{expwidth} = 11 \wedge \textit{wordlength} = 64 \\
\textit{SingleExtended} & \widehat{=}\ \textit{Format} \mid \textit{expwidth} \geq 11 \wedge \textit{wordlength} \geq 43 \\
\textit{DoubleExtended} & \widehat{=}\ \textit{Format} \mid \textit{expwidth} \geq 15 \wedge \textit{wordlength} \geq 79
\end{array}
$$

Once the format is known, the sign, exponent and fraction can be extracted from the

[1] The variable names which are used are declared in a signature (the upper part of the box) and any constraints on these are described by the predicates in the lower part.

integer in which they are stored: [2]

$$
\begin{array}{l}
\textit{Fields} \\
\hline
\textit{Format} \\
\textit{nat} : \mathbb{N} \\
\textit{sign} : 0..1 \\
\textit{exp}, \textit{frac} : \mathbb{N} \\
\hline
\textit{nat} = \textit{sign} \times 2^{\textit{wordlength}-1} + \textit{exp} \times 2^{\textit{fracwidth}} + \textit{frac} \\
\textit{exp} < 2^{\textit{expwidth}} \\
\textit{frac} < 2^{\textit{fracwidth}} \\
\hline
\end{array}
$$

Some of the elements of ***Fields*** are considered to be error codes, or non-numbers. These will be denoted by *NaNF*:

$$\textit{NaNF} \mathrel{\hat=} \textit{Fields} \mid \textit{frac} \neq 0 \wedge \textit{exp} = \textit{EMax}$$

Now, there are enough definitions to give a definition of the value. This is only specified in single or double formats when the number is not a non-number: ("infinite" numbers are given a value to facilitate the definition of rounding)

$$
\begin{array}{l}
\textit{FP} \\
\hline
\textit{Fields} \\
\textit{value} : \mathbb{R} \\
\hline
(\textit{Single} \vee \textit{Double}) \wedge \neg \textit{NaNF} \Rightarrow \\
\quad \textit{exp} = \textit{EMin} \wedge \textit{value} = (-1)^{\textit{sign}} \times 2^{\textit{exp}-\textit{Bias}} \times 2 \times \textit{frac}_0 \\
\quad \vee \\
\quad \textit{exp} \neq \textit{EMin} \wedge \textit{value} = (-1)^{\textit{sign}} \times 2^{\textit{exp}-\textit{Bias}} \times (1 + \textit{frac}_0) \\
\quad \quad \textbf{where} \quad \textit{frac}_0 = 2^{-\textit{fracwidth}} \times \textit{frac} \\
\hline
\end{array}
$$

To facilitate further descriptions, *FP* is partitioned into five classes depending on how its value is calculated from its fields: (non-numbers; infinite, normal, denormal numbers; and zero)

$$
\begin{array}{ll}
\textit{NaN} & \mathrel{\hat=} \textit{FP} \mid \textit{frac} \neq 0 \wedge \textit{exp} = \textit{EMax} \\
\textit{Inf} & \mathrel{\hat=} \textit{FP} \mid \textit{frac} = 0 \wedge \textit{exp} = \textit{EMax} \\
\textit{Norm} & \mathrel{\hat=} \textit{FP} \mid \textit{EMin} < \textit{exp} < \textit{EMax} \\
\textit{Denorm} & \mathrel{\hat=} \textit{FP} \mid \textit{frac} \neq 0 \wedge \textit{exp} = \textit{EMin} \\
\textit{Zero} & \mathrel{\hat=} \textit{FP} \mid \textit{frac} = 0 \wedge \textit{exp} = \textit{EMin}
\end{array}
$$

$$\textit{Finite} \mathrel{\hat=} \textit{Norm} \vee \textit{Denorm} \vee \textit{Zero}$$ [3]

[2] This form is equivalent to declaring the variables of *Format* in the signature and conjoining its constraints with the new constraint.

The essential ingredients of rounding are as follows:

- the number to be approximated;
- a set of values in which the approximation must be;
- a rounding mode;
- a set of preferred values in case two approximations are equally good.

Because the number to be approximated may be outside the range of the approximating values, two values, *MaxValue* and *MinValue*, are introduced which are analogous to $+\infty$ and $-\infty$. The set of *Preferred* values is restricted to ensure that when two approximations are equally good, at least one of them is preferred. To ensure that rounding to zero is consistent, 0 must be in the approximating values.

$$\mathit{Mode} ::= \mathit{ToNearest} \mid \mathit{ToZero} \mid \mathit{ToNegInf} \mid \mathit{ToPosInf}$$

Round_Signature

$$\begin{array}{l}
r : \mathbb{R};\ \mathit{mode} : \mathit{Modes} \\
\mathit{ApproxValues}, \mathit{Preferred} : \mathbb{P}\mathbb{R} \\
\mathit{MinValue}, \mathit{MaxValue} : \mathbb{R} \\
\mathit{value}' : \mathbb{R} \\
\hline
\mathit{Preferred} \cup \{\mathit{value}'\} \subseteq \mathit{ApproxValues} \cup \{\mathit{MinValue}, \mathit{MaxValue}\} \\
0 \in \mathit{ApproxValues} \\
\forall \mathit{value}_1, \mathit{value}_2 : \mathit{ApproxValues} \cup \{\mathit{MinValue}, \mathit{MaxValue}\} \mid \mathit{value}_1 > \mathit{value}_2 \bullet \\
\quad \exists p : \mathit{Preferred} \bullet \mathit{value}_1 \geq p \geq \mathit{value}_2 \\
\forall \mathit{value} : \mathit{ApproxValues} \bullet \mathit{MinValue} \leq \mathit{value} \leq \mathit{MaxValue}
\end{array}$$

The following schemas describe the closest approximations from above and below. If, e.g., the number is smaller than *MinValue*, then the approximation from below is *MinValue*:

Above

$$\begin{array}{l}
\mathit{Round_Signature} \\
\hline
r > \mathit{MaxValue} \Rightarrow \mathit{value}' = \mathit{MaxValue} \\
r \leq \mathit{MaxValue} \Rightarrow \mathit{value}' \geq r \\
\qquad\qquad \forall \mathit{value} : \mathit{ApproxValues} \cup \{\mathit{MaxValue}\} \mid \mathit{value} \geq r \bullet \\
\qquad\qquad\quad \mathit{value} \geq \mathit{value}'
\end{array}$$

[3]Logical operators between schemas have the effect of merging the signatures and performing the logical operation between the predicates.

$$
\begin{array}{|l}
\textit{Below} \\
\hline
\textit{Round\_Signature} \\
\hline
r < \textit{MinValue} \Rightarrow \textit{value}' = \textit{MinValue} \\
r \geq \textit{MinValue} \Rightarrow \textit{value}' \leq r \\
\quad \forall \textit{value} : \textit{ApproxValues} \cup \{\textit{MinValue}\} \mid \textit{value} \leq r \bullet \\
\quad\quad \textit{value} \leq \textit{value}' \\
\hline
\end{array}
$$

Finally, we are in the position to define rounding in its various different modes. Rounding toward zero gives the approximation with the least modulus:

$$
\begin{array}{|l}
\textit{RoundToZero} \\
\hline
\textit{Round\_Signature} \\
\hline
\textit{mode} = \textit{ToZero} \\
(r \geq 0 \land \textit{Below} \\
\quad \lor \\
\;r \leq 0 \land \textit{Above}) \\
\hline
\end{array}
$$

Rounding to positive or negative infinity returns the approximation which is respectively greater or less than the given number:

$$
\begin{array}{|l}
\textit{RoundToPosInf} \\
\hline
\textit{Round\_Signature} \\
\hline
\textit{mode} = \textit{ToPosInf} \\
\textit{Above} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\textit{RoundToNegInf} \\
\hline
\textit{Round\_Signature} \\
\hline
\textit{mode} = \textit{ToNegInf} \\
\textit{Below} \\
\hline
\end{array}
$$

When rounding to nearest, the closest approximation is returned, but if both are equally good, a member of the set *Preferred* is returned: [4]

RoundToNearest

$$
\begin{array}{l}
Round\_Signature \\
\hline
mode = ToNearest \\
\exists Above_1 ; Below_2 \mid r_1 = r = r_2 \bullet \\
\quad value_1 - r < r - value_2 \ \wedge \ Above \\
\qquad \vee \\
\quad value_1 - r > r - value_2 \ \wedge \ Below \\
\qquad \vee \\
\quad value_1 - r = r - value_2 \ \wedge \\
\qquad (value_1 = value_2 \ \wedge \ Above \wedge Below \\
\qquad \vee \\
\qquad value_1 \neq value_2 \ \wedge \ value' \in Preferred \wedge (Above \vee Below))
\end{array}
$$

These specifications can be disjoined to give the full specification as follows.

$$Round \mathrel{\widehat{=}} RoundToNearest \vee RoundToZero \vee RoundToPosInf \vee RoundToNegInf$$

So far, the specification is suitable for describing rounding into any format – be it integer or floating-point. To adapt *Round* specifically for floating-point format, all that is necessary is to fill in the definitions of *ApproxValues*, *Preferred*, *MinValue* and *MaxValue*. This inevitably involves the format of the destination, so *FP′* must be conjoined with *Round*. Once the definitions are filled in, they are no longer needed outside the specification and can be hidden (by existential quantification). It is not difficult to show that the definition of *Preferred* is consistent with the constraint in *Round_Signature*, but this will be left until section 2.3 where a result is proved which makes it even simpler. It is also simple to verify that 0 is an element of *ApproxValues* and that *MinValue* and *MaxValue* satisfy the constraint of *Round*
signature.

FP_Round1

$$
\begin{array}{l}
Round \wedge FP' \\
\hline
\begin{array}{lcll}
ApproxValues & = & \{Finite \mid Format = Format' \bullet value\} \\
Preferred & = & \{Finite \mid Format = Format' \wedge frac \ \mathrm{MOD}\ 2 = 0 \bullet value\} \\
MinValue & \in & \{Inf \mid Format = Format' \wedge sign = 1 \bullet value\} \\
MaxValue & \in & \{Inf \mid Format = Format' \wedge sign = 0 \bullet value\}
\end{array}
\end{array}
$$

$$FP\_Round2 \mathrel{\widehat{=}} FP\_Round1 \setminus \{ApproxValues, Preferred, MinValue, MaxValue\}$$

[4] Decorating the name of a schema with, e.g., $_1$, $'$ has the effect of decorating the names of the variables in the signature of that schema throughout.

The resulting error-conditions have not yet been specified. The conditions resulting in overflow and underflow exceptions are specifically related to a floating-point format and can be described as follows:

$$\begin{array}{lcl} Errors & ::= & inexact \mid overflow \mid underflow \\ Error\_Signature & \widehat{=} & r : \mathbb{R};\ errors' : \mathbb{P}\,Errors;\ FP' \end{array}$$

Error_Spec

$$\begin{array}{l} Error\_Signature \\ \hline \begin{array}{llcl} inexact & \in errors' & \Leftrightarrow & r \neq value' \\ overflow & \in errors' & \Leftrightarrow & Inf' \vee \exists Inf \bullet \text{abs}\, r \geq \text{abs}\, value \\ (underflow & \in errors' & \Leftrightarrow & 0 \neq \text{abs}\, r < 2^{EMin' - Bias'} \\ & \vee & & \\ \ underflow & \in errors' & \Leftrightarrow & Denorm') \end{array} \end{array}$$

(The two alternative conditions under which *underflow* is included in the set *errors'* mean that there is a choice about which condition to implement.)

Finally, the whole specification is:

$$FP\_Round \mathrel{\widehat{=}} FP\_Round2 \wedge Error\_Spec$$

1.2 Addition, Subtraction, Multiplication and Division

In order to discuss these operators, they must be introduced into the mathematics:

$$Ops ::= add \mid sub \mid mul \mid div$$

The essential ingredients of an aritbmetic operation are two numbers, FP_x and FP_y, and an operation $op : Ops$; the number FP' is the result – its format must be at least as wide as each of the operands:

Arit_Signature

$$\begin{array}{l} FP_x;\ FP_y;\ op : Ops \\ FP' \\ \hline wordlength' \geq wordlength_x \\ wordlength' \geq wordlength_y \end{array}$$

When both FP_x and FP_y are finite numbers, the specification is straightforward. A real number is specified which can be rounded to give the correct result – the result of division by zero is described separately:

$$
\begin{array}{l}
\textit{Value\_Spec} \\
\hline
\textit{Arit\_Signature} \land \textit{Finite}_x \land \textit{Finite}_y \\
r : \mathbb{R} \\
\hline
\begin{array}{lcl}
op = add & \land & r = value_x + value_y \\
 & & \lor \\
op = sub & \land & r = value_x - value_y \\
 & & \lor \\
op = mul & \land & r = value_x \times value_y \\
 & & \lor \\
op = div & \land & value_y \neq 0 \land r = value_x \div value_y
\end{array} \\
\hline
\end{array}
$$

If the result after rounding will be zero, some additional information is necessary to specify the sign completely (p.14 §6.3). If zero results from rounding a small number, the sign is that of the small number. If zero is the accurate result then it is the exclusive or of the signs of the arguments when the operation is multiplication or division and is best described by the maths otherwise:

$$
\begin{array}{l}
\textit{Sign\_Bit} \\
\hline
\textit{Arit\_Signature} \\
mode : Modes; r : \mathbb{R} \\
\hline
(-1)^{sign'} \times \mathrm{abs}\, r = r \\
(op = mul \lor op = div) \Rightarrow (-1)^{sign'} = (-1)^{sign_x + sign_y} \\
(op = add \lor op = sub) \land r = 0 \Rightarrow \\
\quad \begin{array}{lcl}
(Zero_x \land Zero_y & \land & (sign_x = sign_y \Leftrightarrow op = add) \land sign' = sign_x \\
 & & \lor \\
\neg(Zero_x \land Zero_y) & \land & (sign_x = sign_y \Leftrightarrow op = add) \land (sign' = 1 \Leftrightarrow mode = ToNegInf))
\end{array} \\
\hline
\end{array}
$$

Rounding has already been described so operations on finite numbers may be defined by using *FP_Round* to specify the relation of r to FP':

$$\textit{Finite\_Arit} \mathrel{\widehat{=}} (\textit{Value\_Spec} \land \textit{FP\_Round} \land \textit{Sign\_Bit}) \setminus \{r\}$$

Division of a finite, non-zero number by zero gives infinity; but, division of zero by zero is not a number:

$$
\begin{array}{l}
\textit{Div\_By\_Zero} \\
\hline
\textit{Arit\_Signature} \\
\hline
\textit{Finite}_x \land \textit{Zero}_y \\
op = div \\
(Zero_x \land NaN') \lor (\neg Zero_x \land Inf' \land (-1)^{sign'} = (-1)^{sign_x - sign_y}) \\
\hline
\end{array}
$$

If one of the operands is not a number, then the result is not a number (the standard demands that the result be equal to the offending operand but that is not always possible, p.13, §6.2):

NaN_Arit

$Arit\_Signature$

$NaN_x \vee NaN_y$
NaN'

Now, arithmetic with infinity is considered. This is defined to be the limit of finite arithmetic. However, certain cases do not have a limit, and these result in a NaN:

$Inf\_Arit\_Signature$

$Arit\_Signature$

$\neg(NaN_x \vee NaN_y)$
$Inf_x \vee Inf_y$

Inf_Add

$Inf\_Arit\_Signature$

$op = add$
$(infsigns = \{sign'\} \wedge Inf') \vee (infsigns = \{0, 1\} \wedge NaN')$
where $infsigns = \{Inf \mid Inf = FP_x \vee Inf = FP_y \bullet sign\}$

Inf_Sub

$Inf\_Arit\_Signature$

$op = sub$
$(infsigns = \{sign'\} \wedge Inf') \vee (infsigns = \{0, 1\} \wedge NaN')$
where $infsigns = \{Inf \mid Inf = FP_x \vee Inf = FP_y[-value_y/value_y] \bullet sign\}$

Inf_Mul

$Inf\_Arit\_Signature$

$op = mul$
$((Zero_x \vee Zero_y) \wedge NaN'$
$\vee$
$\neg(Zero_x \vee Zero_y) \wedge Inf' \wedge (-1)^{sign'} = (-1)^{sign_x + sign_y})$

$$
\begin{array}{|l}
\hline
\textit{Inf\_Div} \\
\quad \textit{Inf\_Arit\_Signature} \\
\hline
\quad op = div \\
\quad (Inf_x \wedge Inf_y \wedge NaN' \\
\quad \vee \\
\quad Finite_y \wedge Inf' \wedge (-1)^{sign'} = (-1)^{sign_x - sign_y} \\
\quad \vee \\
\quad Finite_x \wedge Zero' \wedge (-1)^{sign'} = (-1)^{sign_x - sign_y}) \\
\hline
\end{array}
$$

These partial specifications can be disjoined to give the complete specification of arithmetic with infinity:

$$Inf\_Arit \mathrel{\widehat=} Inf\_Add \vee Inf\_Sub \vee Inf\_Mul \vee Inf\_Div$$

None of the exceptional cases return the rounding errors; *No_Round_Errors* describes this, and *FP_Arit* describes the complete relation on *Arit_Signature*:

$$No\_Round\_Errors \mathrel{\widehat=} round\_errors' : \mathbb{P}\, Round\_Errors \mid round\_errors' = \{\ \}$$

$$
\begin{array}{ll}
FP\_Arit \mathrel{\widehat=} & \qquad\qquad Finite\_Arit \\
& \qquad\qquad\quad \vee \\
& No\_Round\_Errors \wedge (Div\_By\_Zero \vee NaN\_Arit \vee Inf\_Arit)
\end{array}
$$

Five different errors can occur during the operations. These cover all the different cases when the finite operations do not extend to infinite numbers; division by zero; and when one operand is not a number:

$$Arit\_Errors ::= NaN\_Op \mid mul\_Zero\_Inf \mid div\_Zero \mid div\_Inf\_Inf \mid Mag\_sub$$

$$
\begin{array}{|lll}
\hline
\textit{Error\_Spec} & & \\
\quad Arit\_Signature & & \\
\quad arit\_errors' : \mathbb{P}\, Arit\_Errors & & \\
\hline
\quad NaN\_Op & \in arit\_errors' \Leftrightarrow & NaN_x \vee NaN_y \\
\quad mul\_Zero\_Inf & \in arit\_errors' \Leftrightarrow & op = mul \wedge ((Zero_x \wedge Inf_y) \vee (Inf_x \wedge Zero_y)) \\
\quad div\_Zero & \in arit\_errors' \Leftrightarrow & (op = div \wedge \neg NaN_x \wedge Zero_y) \\
\quad div\_Inf\_Inf & \in arit\_errors' \Leftrightarrow & (op = div \wedge Inf_x \wedge Inf_y) \\
\quad Mag\_sub & \in arit\_errors' \Leftrightarrow & (Inf_x \wedge Inf_y \wedge ((sign_x = sign_y \wedge op = sub) \\
& & \qquad\qquad\qquad\qquad \vee \\
& & \qquad\qquad (sign_x \neq sign_y \wedge op = add)) \\
\hline
\end{array}
$$

Finally, the whole specification is:

$$Arit \mathrel{\widehat=} FP\_Arit \wedge Error\_Spec$$

1.3 Remainder

To calculate remainder, all that is necessary is a divisor and a dividend, FP_x and FP_y. The result will be given by FP'. The signature is:

$$
\begin{array}{l}
\textit{Rem\_Signature} \\
\hline
FP_x; FP_y \\
FP' \\
\hline
\end{array}
$$

In the general case, in which both numbers are finite and the divisor is not zero, the result is defined as follows:

$$
\begin{array}{l}
\textit{Fin\_Rem} \\
\hline
\textit{Rem\_Signature} \\
\hline
\textit{Finite}_x \wedge \textit{Finite}_y \\
\neg \textit{Zero}_y \\
2 \times \text{abs } \textit{value}' \leq \text{abs } \textit{value}_y \\
\exists n : \mathbb{Z} \bullet \\
\quad \textit{value}_x = n \times \textit{value}_y + \textit{value}' \\
\quad 2 \times \text{abs } \textit{value}' = \text{abs } \textit{value}_y \Rightarrow n \text{ MOD } 2 = 0 \\
\hline
\end{array}
$$

Remainder of a finite number by zero is a non-number:

$$
\begin{array}{l}
\textit{Rem\_Zero} \\
\hline
\textit{Rem\_Signature} \\
\hline
\textit{Finite}_x \wedge \textit{Zero}_y \\
\textit{NaN}' \\
\hline
\end{array}
$$

As ever, when one of the operands is a non-number, the result is a non-number:

$$
\begin{array}{l}
\textit{NaN\_Rem} \\
\hline
\textit{Rem\_Signature} \\
\hline
\textit{NaN}_x \vee \textit{NaN}_y \\
\textit{NaN}' \\
\hline
\end{array}
$$

The remainder of infinity by any number is not a number. The remainder of a finite number by infinity is the original number:

$$
\begin{array}{|l}
\hline
\textit{Inf\_Rem} \\
\hline
\quad \textit{Rem\_Signature} \\
\hline
\quad \textit{Inf}_x \wedge \neg \textit{NaN}_y \wedge \textit{NaN}' \\
\quad \vee \\
\quad \textit{Finite}_x \wedge \textit{Inf}_y \wedge \textit{FP}' = \textit{FP}_x \\
\hline
\end{array}
$$

When the result is zero, the sign is the sign of the dividend:

$$
\begin{array}{|l}
\hline
\textit{Sign\_Bit} \\
\hline
\quad \textit{Rem\_Signature} \\
\hline
\quad \textit{Zero}' \Rightarrow \textit{sign}' = \textit{sign}_x \\
\hline
\end{array}
$$

There are three errors possible with remainder – when one of the operands is not a number, or the divisor is zero or the dividend is infinity. The second two give rise to the same exception:

$$\textit{Rem\_Errors} ::= \textit{NaN\_Op} \mid \textit{rem\_Zero\_Inf}$$

$$
\begin{array}{|l}
\hline
\textit{Error\_Spec} \\
\hline
\quad \textit{Rem\_Signature} \\
\quad \textit{errors}' : \mathbb{P}\,\textit{Rem\_errors} \\
\hline
\quad \textit{NaN\_Op} \in \textit{errors} \Leftrightarrow \textit{NaN}_x \vee \textit{NaN}_y \\
\quad \textit{rem\_Zero\_Inf} \in \textit{errors} \Leftrightarrow (\textit{Inf}_x \wedge \neg \textit{NaN}_y) \vee (\neg \textit{NaN}_x \wedge \textit{Zero}_y) \\
\hline
\end{array}
$$

Putting all the pieces together gives the full specification:

$$\textit{Rem} \mathrel{\hat{=}} (\textit{Fin\_Rem} \vee \textit{Rem\_Zero} \vee \textit{NaN\_Rem} \vee \textit{Inf\_Rem}) \wedge \textit{Sign\_Bit} \wedge \textit{Error\_Spec}$$

1.4 Square Root

As with addition etc., an exact result is specified then rounded using *FP_Round*. The exact square root is defined as follows:

$$
\begin{array}{|l}
\textit{Exact\_Sqrt} \\
\hline
FP \\
r : \mathbb{R} \\
\hline
\textit{Finite} \\
\textit{value} \geq 0 \\
r \times r = \textit{value} \\
r \geq 0 \\
\hline
\end{array}
$$

This is rounded and r is hidden. The destination must have a format at least as wide as the argument:

$$\textit{Pos\_Sqrt} \mathrel{\hat{=}} (\textit{Exact\_Sqrt} \land \textit{FP\_Round}) \mid \textit{wordlength} \leq \textit{wordlength}' \setminus \{r\}$$

The sign of zero is unchanged:

$$
\begin{array}{|l}
\textit{Sign\_Bit} \\
\hline
FP \\
FP' \\
\hline
\textit{Zero}' \Rightarrow \textit{sign}' = \textit{sign} \\
\hline
\end{array}
$$

The square root of positive infinity is infinity:

$$
\begin{array}{|l}
\textit{Inf\_Sqrt} \\
\hline
\textit{Inf} \\
FP' \\
\hline
\textit{sign} = 0 \\
FP' = \textit{Inf} \\
\hline
\end{array}
$$

In all other cases, the result is a *NaN*:

$$
\begin{array}{|l}
\textit{Exc\_Sqrt} \\
\hline
FP \\
FP' \\
\hline
\textit{NaN} \lor \textit{value} < 0 \\
\textit{NaN}' \\
\hline
\end{array}
$$

There are two errors – NaN_Op and when the operand is less than zero:

$$Sqrt\_Errors \mathrel{\widehat=} NaN\_Op \mid OpLT0$$

$$
\begin{array}{|l}
\textbf{Error\_Spec} \\
\hline
FP \\
FP';\ errors' : \mathbb{P}\,Sqrt\_Errors \\
\hline
NaN\_Op \in errors' \Leftrightarrow NaN \\
OpLT0 \quad \in errors' \Leftrightarrow \neg NaN \wedge value < 0 \\
\hline
\end{array}
$$

Putting the pieces together:

$$Sqrt \mathrel{\widehat=} (Pos\_Sqrt \vee Inf\_Sqrt \vee Exc\_Sqrt) \wedge Sign\_Bit \wedge Error\_Spec$$

1.5 Floating Point Format Conversions

When converting to a different format, *Inf* and *NaN* must be preserved, and *Finite* numbers may have to be rounded:

$$
\begin{array}{|l}
\textbf{NaN\_Convert} \\
\hline
FP \\
FP' \\
\hline
NaN \wedge NaN' \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\textbf{Inf\_Convert} \\
\hline
FP \\
FP' \\
\hline
Inf \wedge Inf' \\
sign' = sign \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\textbf{Fin\_Convert} \\
\hline
FP \\
FP' \\
\hline
Finite \\
FP\_Round[value/r] \\
sign = sign' \\
\hline
\end{array}
$$

$$Convert \mathrel{\widehat=} ((NaN\_Convert \lor Inf\_Convert) \land No\_Round\_Errors) \lor Fin\_Convert$$

1.6 Rounding and Converting to Integers

This section covers both converting to an integer format and rounding to an integer in floating-point format. The basic adaptation of the rounding predicate is the same for both operations. The approximating values are all the numbers from *MinValue* to *MaxValue* and the preferred values are the even integers. When converting to an integer format, the minimum and maximum values can be defined to be the minimum and maximum integers of the format. When rounding to an integer value in floating-point format, these values will be the greatest and smallest integers available in the destination format.

$$\begin{array}{|l}
\textit{Integer_Round1} \\
\hline
Round \\
\hline
ApproxValues = MinValue..MaxValue \\
Preferred = ApproxValues \cap \{n : \mathbb{Z} \mid n \bmod 2 = 0\} \\
\hline
\end{array}$$

$$Integer\_Round \mathrel{\widehat=} Integer\_Round1 \setminus \{ApproxValues, Preferred\}$$

1.6.1 Conversions to Integer Formats

All that we need to know of an integer format are the minimum and maximum integers. These can be used to adapt *Integer_Round* to describe rounding into an integer format:

$$\begin{array}{|l}
\hline
MaxInt, MinInt : \mathbb{Z} \\
\hline
\end{array}$$

$$Int\_Conv\_Round \mathrel{\widehat=} Integer\_Round \mid MinValue = MinInt \land MaxValue = MaxInt \setminus \{MinValue, MaxValue\}$$

When the operand is not a *Finite* number or is out of range of the integer format, the result is not specified:

$$\begin{array}{|l}
\textit{Exc_Conv} \\
\hline
FP \\
Integer' \\
\hline
NaN \lor Inf \lor value < MinInt \lor MaxInt > value \\
\hline
\end{array}$$

The specification is:

$$Convert\_Integer \mathrel{\widehat=} Exc\_Conv \lor Int\_Conv\_Round[value/r]$$

1.6.2 Rounding to Integer

There is a small problem in using *Integer_Round* to specify rounding to an integer value in a given floating-point format as there may be some integer values between the maximum and minimum values which cannot be obtained. The following definition assumes (as is the case with the formats specified in the standard) that if there exist two integers m and n such that there is an intermediate integer which cannot be obtained in the destination format, then no other value between m and n can obtained in that format. Although it is not difficult to give a definition in the general case, it is felt that the assumption is not unreasonable. Hence, *MinValue* and *MaxValue* can be defined to be the minimum and maximum integer available in that format:

Rnd_Int_Round1

$$\begin{array}{|l} Integer\_Round \\ \hline MaxValue = \sup \{FP' \mid Format = Format' \bullet value\} \cap \mathbb{Z} \\ MinValue = \inf \{FP' \mid Format = Format' \bullet value\} \cap \mathbb{Z} \end{array}$$

$$Rnd\_Int\_Round \mathrel{\widehat{=}} Rnd\_Int\_Round1 \setminus \{MinValue, MaxValue\}$$

The destination format is restricted to be the same as that of the argument:

Int_Signature

$$\begin{array}{|l} FP \\ FP' \\ \hline Format = Format' \end{array}$$

$$Fin\_Int \mathrel{\widehat{=}} Finite \wedge Int\_Signature \wedge Rnd\_Int\_Round[value/r]$$

In this case, *Inf* and *NaN* are preserved:

Inf_NaN_Int

$$\begin{array}{|l} Int\_Signature \\ \hline Inf \vee NaN \\ FP = FP' \end{array}$$

The whole specification:

$$Int \mathrel{\widehat{=}} Fin\_Int \vee Inf\_NaN\_Int$$

1.7 Comparisons

There are four mutually exclusive comparisons. Unordered when one is a non-number; equal; less than; or greater than:

Unordered

FP_x
FP_y

$NaN_x \vee NaN_y$

Equal

FP_x
FP_y

$\neg(NaN_x \vee NaN_y)$
$value_x = value_y$

LessThan

FP_x
FP_y

$\neg(NaN_x \vee NaN_y)$
$value_x < value_y$

GreaterThan

FP_x
FP_y

$\neg(NaN_x \vee NaN_y)$
$value_x > value_y$

The result of a comparison can be a condition code identifying one of the four disjoint relations:

$$Conditions ::= UO \mid EQ \mid LE \mid GE$$

Compare_Condition

$FP_x ; FP_y$
$condition' : Conditions$

$$
\begin{aligned}
& condition' = UO \;\wedge\; Unordered \\
& \quad\vee \\
& condition' = EQ \;\wedge\; Equal \\
& \quad\vee \\
& condition' = LE \;\wedge\; LessThan \\
& \quad\vee \\
& condition' = GE \;\wedge\; GreaterThan
\end{aligned}
$$

Alternatively, it may return a true-false result depending on one of the useful comparisons listed below:

$$Bool ::= true \mid false$$

Compare_Bool

$FP_x ; FP_y ; op : \mathbb{P}\, Conditions$
$result' : Bool$

$$
\begin{aligned}
& op \neq \{\} \\
& op \neq Conditions \\
& result' = true \Leftrightarrow \exists condition' : op \bullet Compare\_Condition
\end{aligned}
$$

An exception can be raised when one of the operands is not a number. If this exception is to be raised, the flag *exception* must be set:

Compare_Bool_Error

$Compare\_Bool$
$exception : Bool$

$$
\begin{aligned}
& NaN\_Op \in errors' \Leftrightarrow Unordered \wedge exception = true \\
& (op = \{EQ\} \vee op = Conditions - \{EQ\}) \Rightarrow exception = false
\end{aligned}
$$

Chapter 2

Implementation

2.1 Foreword to the Proof

Much of the proof relies on OCCAM specifications given in the appendix. Informal specifications can be found in [inmos]. The proof of the arithmetic procedures is largely routine manipulation of equations. These parts will be treated somewhat briefly with statements of the theorems necessary. Hints to the proof of theorems will be indicated, e.g., ***Routine manipulation. (This hint is omitted.)*** For the non-exceptional cases, the algorithm uses the following scheme:

1. Unpack both operands into their sign, exponent and fraction fields.
2. Denormalise both by shifting in the leading bit of the fraction when necessary.
3. Perform the relevant operation.
4. Pack the result.
5. Round the packed result.

Error conditions are set during packing and rounding. The more difficult parts of the proof are caused by changes in the representation of numbers (e.g. packing, denormalising, etc.). The first section of the proof is concerned with specifying the relationship between *FP* or $\mathbb{R}$ and these representations. The second section contains procedures for changing representations along with their proofs. Later sections contain procedures for the arithmetic operations. The proofs of these are much simpler than for the others and only an informal outline of why they are correct is given.

The following is a brief description of how specifications and programs are related and how it is possible to assert formally that a program meets its specification. The predicates in braces, e.g. $\{\phi\}\boxed{\texttt{P}}\{\psi\}$, mean that if P is executed in a state satisfying ϕ, then it is guaranteed to terminate in a state satisfying ψ. Some of the conjuncts of the assertions are omitted for the sake of clarity. The first assertion is called the precondition of the program – if this does not hold on entry to the program, neither is it guaranteed to terminate nor, if it does, to terminate in any sensible state. The rules relating the program to the assertions are described in [Gries], [Dijkstra] and [Hoare]. A brief description follows:

Rule 1 *The program SKIP does nothing but terminate:*

$$\vdash \{\phi\}\boxed{\mathtt{SKIP}}\{\phi\}$$

Rule 2 *If the expression e can be evaluated correctly (i.e. there is no division by zero etc.), then if the state is required to satisfy ϕ after termination, it must satisfy ϕ with e substituted for x before:*

$$\vdash \{\mathcal{D}\mathtt{e} \wedge \phi[\mathtt{e}/\mathtt{x}]\}\boxed{\mathtt{x\ :=\ e}}\{\phi\}$$

Rule 3 *If P starts in state ϕ and terminates in state ψ and Q starts in state ψ and terminates in state χ, then P followed by Q starts in state ϕ and terminates in state χ:*

$$\{\phi\}\boxed{\mathtt{P}}\{\psi\} \wedge \{\psi\}\boxed{\mathtt{Q}}\{\chi\} \vdash \{\phi\}\boxed{\begin{array}{l}\mathtt{SEQ}\\ \quad\mathtt{P}\\ \quad\mathtt{Q}\end{array}}\{\chi\}$$

Rule 4 *The rule for conditionals is that, if P starts in a state satisfying ϕ and its guard and terminates in state ψ and similarly for Q, then the conditional composition can start in a state which satisfies one or other of the guards and ϕ and terminate in a state satisfying ψ:*

$$\{b_P \wedge \phi\}\boxed{\mathtt{P}}\{\psi\} \wedge \{b_Q \wedge \phi\}\boxed{\mathtt{Q}}\{\psi\} \vdash \{(b_P \vee b_Q) \wedge \phi\}\boxed{\begin{array}{l}\mathtt{IF}\\ \quad b_P\\ \qquad\mathtt{P}\\ \quad b_Q\\ \qquad\mathtt{Q}\end{array}}\{\psi\}$$

Rule 5 *The precondition of a program may be strengthened:*

$$(\chi \Rightarrow \phi) \wedge \{\phi\}\boxed{\mathtt{P}}\{\psi\} \vdash \{\chi\}\boxed{\mathtt{P}}\{\psi\}$$

Rule 6 *The postcondition of a program may be weakened:*

$$(\chi \Leftarrow \psi) \wedge \{\phi\}\boxed{\mathtt{P}}\{\psi\} \vdash \{\phi\}\boxed{\mathtt{P}}\{\chi\}$$

The following two functions are useful, they return the integer part and the fractional part of a real number:

$$\begin{array}{|l}
\hline
\mathit{int} \quad : \mathbb{R} \rightarrow \mathbb{N}\\
\mathit{nonint} : \mathbb{R} \rightarrow \mathbb{R}\\
\hline
r = \mathit{int}\ r + \mathit{nonint}\ r\\
\mathrm{abs}\,(\mathit{nonint}\ r) < 1\\
\mathrm{abs}\,(\mathit{int}\ r) \leq \mathrm{abs}\ r\\
\hline
\end{array}$$

2.2 Representations of FP

The aim of this section is to specify the relation between *FP* and its representations in the program. We will only be concerned with the implementation of single-length numbers on a machine whose wordlength is 32 :

$$FPS32 \mathrel{\widehat=} FP \mid Single \wedge wordlength = wl$$

Externally to the program, each number is represented as a single *Word* corresponding to the value of its field *nat*. Thus, the relationship of FP to its external representation is given by:

$$External \mathrel{\widehat=} FPS32; word : Word \mid nat = word.nat$$

Internally to the program, *FP* is represented by three words giving its sign, exponent and fraction. The exact relation between these words and *FP* is discussed further below:

$$Internal \mathrel{\widehat=} FPS32; wsign, wexp, wfrac : Word$$

To distinguish the five different classes of number, they are first unpacked into the sign, exponent and fraction fields. The words *wsign*, *wexp*, and *wfrac* correspond to the fields *sign*, *exp*, and *frac*:

```
Unpacked ____________________________
| Internal
|__________________
| wsign.nat = sign × 2^(wl−1)
| wexp.int  = exp
| wfrac.nat = frac × 2^(expwidth+1)
|____________________________________
```

To perform the arithmetic operations, *Finite* numbers are given a representation which bears a uniform relation to their *value* – the first equation in the following implicitly defines *wfrac.nat*:

```
Unnormalised ________________________
| Internal
|__________________
| Finite
| value     = (−1)^sign × 2^(wexp.int−Bias−wl+1) × wfrac.nat
| wsign.nat = sign × 2^(wl−1)
| wexp.int  = exp
|____________________________________
```

2.3 Representing Real Numbers

The aim of this section is to specify the relation between $\mathbb{R}$ and its representations in the program.

First, notice the simple result that the order on the absolute value of a number is the same as the usual order on the less significant bits of its representation as a word:

$$FP; FP' \mid Format = Format' \land \neg(NaN \lor NaN')$$

$$\vdash \text{abs}\, value \leq \text{abs}\, value' \Leftrightarrow nat \text{ MOD } 2^{wordlength-1} \leq nat' \text{ MOD } 2^{wordlength-1}$$

This can be used to see that the number of least modulus with modulus greater than a given finite number is obtained by incrementing its representation as a word:

Succ

$$FP;\ FP_0$$

$$\begin{array}{l} Finite \\ \text{abs}\, value < \text{abs}\, value_0 \\ \forall FP' \mid \text{abs}\, value < \text{abs}\, value' \bullet \text{abs}\, value_0 \leq \text{abs}\, value \end{array}$$

$$FP; FP_0 \mid Finite \land nat_0 = nat + 1 \vdash Succ$$

From this result, the consistency of *Preferred* in section 1.1 can be deduced.

In turn, this means that if the approximation of less modulus is known, only enough extra information to determine the four predicates in *RoundToNearest* is needed to return the correct value. This is, of course, the familiar *guard* and *sticky* bits defined below:

Bounds

$$\begin{array}{ll} r & : \mathbb{R} \\ Succ; guard, sticky & : 0..1 \end{array}$$

$$\begin{array}{lcl} r > 0 & \Rightarrow & sign = 0 \land Below[value/value'] \\ r = 0 & \Rightarrow & Zero \\ r < 0 & \Rightarrow & sign = 1 \land Above[value/value'] \\ & & \\ guard = 0 & \Leftrightarrow & r - value < value_0 - r \\ sticky = 0 & \Leftrightarrow & r - value = value_0 - r \lor r = value \end{array}$$

$$\begin{array}{ll} Bounds \vdash & \exists Above_1; Below_2 \mid r_1 = r = r_2 \bullet \\ & value_1 - r < r - value_2 \Leftrightarrow guard = 0 \land sticky = 1 \\ & value_1 - r > r - value_2 \Leftrightarrow guard = 1 \land sticky = 1 \\ & value_1 - r = r - value_2 \Leftrightarrow sticky = 0 \\ & value_1 = value_2 \Leftrightarrow guard = 0 \land sticky = 0 \end{array}$$

This is, however, not quite enough information to return the correct overflow condition. If $r \geq 2^{EMax' - Bias'}$, this information is lost. Conversely, it is not possible to determine the overflow condition before rounding as the condition *Inf'* cannot be tested until the final result is calculated. Thus, it is necessary to divide Error_Spec into two parts. The *inexact* and *underflow* conditions can be determined before or after rounding. The design decision is made that so many error conditions as possible will be determined after rounding in order that the precondition of the module is simpler. Thus, the following decomposition is valid (the validity is demonstrated by the theorem):

Error_Before

Error_Signature
$overflow \in errors' \Leftrightarrow \mathrm{abs}\ r \geq 2^{EMax' - Bias'}$

Error_After

$Error\_Signature;\ errors : \mathbb{P}\, Errors$		
$overflow$	$\in errors \Leftrightarrow \mathrm{abs}\ r \geq 2^{EMax' - Bias'}$	
$inexact$	$\in errors' \Leftrightarrow r \neq value'$	
$overflow$	$\in errors' \Leftrightarrow overflow \in errors \lor Inf'$	
$underflow$	$\in errors' \Leftrightarrow Denorm'$	

$$\vdash Error\_Spec \sqsubseteq (Error\_Before;\ Error\_After)$$ [1]

If we have the approximation of less modulus, the *guard* and *sticky* bits and an overflow indication, there is enough information to determine the correct result and the correct error conditions. Thus, a real number may be represented prior to rounding as follows:

$$Packed \mathrel{\widehat{=}} ((\exists Succ \bullet Bounds) \land External \land Error\_After) \setminus \{errors'\}$$

This representation is too complicated for the immediate result of a calculation – we require a form which has a sign, exponent and fraction but which contains enough information to produce a *Packed* number. If the exponent is considered to be unbounded above (this assumption causes no problems since the largest exponent which can be produced from finite arithmetic is less than 2^{wl}), and demand that the fraction be at least 2^{wl-1} when the exponent is not *EMin*, a condition for an extra digit of accuracy is easy to formulate. The condition given here is stronger than necessary but simpler than

[1] If a schema is thought of as a function from its unprimed to its primed components, the sequential composition (;) is analogous to the right composition of the two functions. The symbol $\sqsubseteq$ is used to indicate that a design decision has been made.

the weakest condition:

Normal

$r : \mathbb{R}$
Internal

$wexp.int \geq EMin$
$wexp.int > EMin \Rightarrow wfrac.nat \geq 2^{wl-1}$
$\text{abs}(approx - exact) < 0$
$nonint\ approx = 0 \Leftrightarrow nonint\ exact = 0$
$\textbf{where}\ approx = (-1)^{wsign.nat} \times 2^{1-expwidth} \times wfrac.nat$
$\qquad exact = 2^{Bias-wexp.int+2+fracwidth} \times r$

2.4 Unpacking and Denormalising

The object of this section is to specify and prove the procedures which will be used to perform changes of representation of *FP*. First, the numbers are unpacked from their *External* representation into the *Unpacked* representation. Second, numbers are converted into their *Unnormalised* representation.

Some useful constant words:

Zero, One, MSB, INF : *Word*

Zero.*nat*	=	0
One.*nat*	=	1
MSB.*bitset*	=	$\{wl - 1\}$
INF.*nat*	=	$2^{fracwidth} \times EMax$

2.4.1 Unpacking

The specification of the procedure:

$$Unpack \mathrel{\hat{=}} \{word\} \lhd External; Unpacked' \mid FP = FP' \rhd \{wsign, wexp, wfrac\}^{2}$$

The most significant bit of the *word* is stored in *wsign*, then the sign bit is shifted out and the exponent and fraction fields are shifted into the appropriate *Words*:

[2] The symbols $\lhd$ $\rhd$ indicate that the procedure is to take its input from the variables to the left of $\lhd$, filling the other fields consistently, and put its output into the variables on the right of $\rhd$ Formally, $\lhd$ hides all unprimed variables except those in the set to its left; $\rhd$ hides the primed form of all variables except those to its right.

```
PROC Unpack (VALUE word, VAR wsign, wexp, wfrac) =
  {External}
  SEQ
    wsign := word ∧ MSB
    {Unpacked\{wexp, wfrac}}
    SHIFTLEFT (wexp, wfrac, Zero, word << One, expwidth)   :
  {Unpacked}
```

The following three theorems about integers are useful in the details of the proof:

$$a, b, c : \mathbb{N} \mid c \neq 0 \vdash a = b \Leftrightarrow a \,\mathrm{DIV}\, c = b \,\mathrm{DIV}\, c \land a \,\mathrm{MOD}\, c = b \,\mathrm{MOD}\, c$$
$$\vdash a \times (b \,\mathrm{MOD}\, c) = (a \times b) \,\mathrm{MOD}\, (a \times c)$$
$$\vdash (a \times b) \,\mathrm{DIV}\, (a \times c) = b \,\mathrm{DIV}\, c$$

2.4.2 Denormalising

The specification:

$$Denormalise \mathrel{\widehat{=}} \{wexp, wfrac\} \lhd Unpacked; Unnormalised' \mid FP = FP' \rhd \{wexp, wfrac\}$$

If the number is in *Norm* then the implicit leading bit is shifted in, otherwise it is left unchanged:

```
PROC Denormalise (VAR wexp, wfrac) =
  {Unpacked}
  IF
    wexp = EMin
      {Denorm}
      SKIP
    wexp <> EMin
      {Norm}
      wfrac := MSB ∨ (wfrac >> One) :
  {Unnormalised}
```

2.5 Rounding and Packing

This section aims to specify and prove procedures for converting between representations of **R**.

2.5.1 Rounding

Specification:

$$Round \mathrel{\widehat{=}} \{word, guard, sticky, mode, errors\} \lhd Packed \land FP\_Round \land Error\_After \rhd \{word, error$$

There are two things to notice about the specification:

- the specification of errors is conjoined in such a way that the unprimed variable, *errors*, upon which it depends is not restricted by the the other conjuncts; thus the specification decomposes into a sequential composition of a specification on *FP* and a specification on *errors*;
- *Round*, and hence *FP_Round*, is a disjunction of specifications and thus may be implemented by a conditional.

The first observation can be formalised as:

$$\vdash \mathit{Round\_Proc} \Leftrightarrow (\exists r : \mathbb{R}; \mathit{FP}_0 \bullet \mathit{FP\_Round2} \wedge \mathit{Bounds}); \mathit{Error\_After}$$

And the second observation can be formalised as:

$$\begin{array}{lc} \vdash & \exists r : \mathbb{R}; \mathit{FP}_0 \bullet \mathit{FP\_Round2} \wedge \mathit{Bounds} \\ & \Leftrightarrow \\ & \exists r : \mathbb{R}; \mathit{FP}_0 \bullet \mathit{FP\_Round2} \mid \mathit{mode} = \mathit{ToNearest} \wedge \mathit{Bounds} \\ & \vee \\ & \exists r : \mathbb{R}; \mathit{FP}_0 \bullet \mathit{FP\_Round2} \mid \mathit{mode} = \mathit{ToPosInf} \wedge \mathit{Bounds} \\ & \vee \\ & \exists r : \mathbb{R}; \mathit{FP}_0 \bullet \mathit{FP\_Round2} \mid \mathit{mode} = \mathit{ToNegInf} \wedge \mathit{Bounds} \\ & \vee \\ & \exists r : \mathbb{R}; \mathit{FP}_0 \bullet \mathit{FP\_Round2} \mid \mathit{mode} = \mathit{ToZero} \wedge \mathit{Bounds} \end{array}$$

The first observation has the obvious implication that the module can be implemented as the sequence of two smaller programs, the first of which sets the correct approximation and the second of which returns the correct error conditions.

The second observation leads to a decomposition because each of the disjuncts is disjoint (i.e. the conjunction of any two is not satisfiable). Thus, a conditional can be formed in which the guards discriminate according to the rounding mode.

The most obscure line is the following: `nat := nat + (guard ∧ (sticky ∨ nat))`. This is derived from: `nat := nat + ((guard ∧ sticky) ∨ (guard ∧ (nat ∧ One)))`. Using `guard = guard ∧ One` and the commutativity and associativity of ∧, the last part of the expression reduces to `guard ∧ nat`. Now, ∧ distributes through ∨ to give the optimised expression.

The original expression can be seen to be correct by studying the inequalities used to define *RoundToNearest*.

```
PROC Round (VALUE mode, guard, sticky, VAR nat, errors) =
{overflow ∈ errors ⇔ r ≥ 2^(EMax-Bias)}
{r ≥ 0 ⇒ Below[FP/FP']}
{r ≤ 0 ⇒ Above[FP/FP']}
SEQ
  IF
    mode = ToZero
      SKIP
      {FP_Round2[FP/FP']}
    mode = ToNegInf
      IF
        sign = Zero
          SKIP
        sign ≠ Zero
          nat := nat + One
          {FP_Round2[FP/FP']}
    mode = ToPosInf
      IF
        sign = Zero
          nat := nat + One
        sign ≠ Zero
          SKIP
          {FP_Round2[FP/FP']}
    mode = ToNearest
      nat := nat + (guard ∧ (sticky ∨ nat))
      {FP_Round2[FP/FP']}
  {overflow ∈ errors ⇔ r ≥ 2^(EMax-Bias)}
  errors := errors ∩ {overflow}
  {underflow, inexact ∉ errors}
  {overflow ∈ errors ⇔ r ≥ 2^(EMax-Bias)}
  IF
    Inf
      errors := errors ∪ {overflow}
    ¬ Inf
      SKIP
  {overflow ∈ errors ⇔ Inf ∨ r ≥ 2^(EMax-Bias)}
  {underflow ∉ errors}
  IF
    Denorm
      errors := errors ∪ {underflow}
    ¬ Denorm
      SKIP
  {underflow ∈ errors ⇔ Denorm}
```

$\{inexact \notin errors\}$

```
IF
  (sticky ∨ guard) ≠ Zero
    errors := errors ∪ {inexact}
  (sticky ∨ guard) = Zero
    SKIP
```

$\{inexact \in errors \Leftrightarrow r \neq value\}$
$\{FP\_Round[FP/FP']\}$

2.5.2 Packing

Specification:

$$Pack \mathrel{\widehat{=}} \begin{array}{c}\{wsign, wexp, wfrac\} \\ \lhd (Normal; Packed') \wedge Error\_Before \mid r = r' \rhd \\ \{word, guard, sticky, errors\}\end{array}$$

The fraction is adjusted to remove the leading bit if the exponent is large enough. The exponent is checked for overflow. If overflow has occurred then the appropriate error condition is set and the exponent and fraction are set to give the largest finite modulus and to ensure that the guard and sticky bits will be correct; if overflow has not occurred, no change is made. Then, the fraction and exponent are packed and the guard and sticky bits set appropriately. The proof of this procedure is very much like that of *Unpack* and *Denormalise*:

```
PROC Pack (VALUE wsign, wexp, wfrac, VAR word, guard, sticky, errors) =
{Normal}
SEQ
  IF
    wexp = EMin
      SKIP
    wexp <> EMin
      wfrac := wfrac << One
  IF
    wexp >= EMax
      SEQ
        errors := {overflow}
        wexp := EMax-One
        wfrac := NOT Zero
    wexp < EMax
      errors := {}
  {overflow ∈ errors ⇔ r ≥ 2^(EMax-Bias)}
  {Below[abs r/r] ∧ FP' ⇒ exp' = wexp.int ∧ frac' = wfrac.nat DIV 2^(expwidth+1)}
  SHIFTLEFT (word,sticky,wexp,wfrac,fracwidth+One)
  {Below[abs r/r] ∧ FP'[(word >> One).bitset/bitset']}
  guard := word ∧ One
  IF
    sticky = Zero
      SKIP
    sticky <> Zero
      sticky := One
  word := wsign ∨ (word >> One) :
{Packed}
```

2.6 Finite Arithmetic Procedures

These procedures will take two *Unnormalised* numbers and calculate the result into an *External* . Their specification:

$$FiniteArit \mathrel{\hat{=}} \{wsign_x, wexp_x, wfrac_x, op, wsign_y, wexp_y, wfrac_y\} \lhd (Unnormalised_x; Unnormalised_y; Normal') \wedge Value\_Spec \rhd \{wsign, wexp, wfrac\}$$

The procedures for each operation will be considered separately in the following sections.

2.6.1 Addition and Subtraction

Since adding a number is the same as subtracting the number with its sign changed, the two procedures are combined into one:

$$\vdash Add = Sub[sign_y / 1 - sign_y]$$

$$AddSub \mathrel{\widehat{=}} FiniteArit \mid op = add \lor op = sub$$

First, consider the sum of two numbers:

$$\begin{aligned} d, e : \mathbb{Z}; f, g : \mathbb{N} \mid d \geq e \vdash 2^d \times f + 2^e \times g &= 2^d \times (f + 2^{e-d} \times g) \\ &= 2^d \times (f + int(2^{e-d} \times g) + nonint(2^{e-d} \times g)) \end{aligned}$$

and the difference:

$$\begin{aligned} carry : 0..1 \vdash 2^d \times f - 2^e \times g &= 2^d \times (f - 2^{e-d} \times g) \\ &= 2^d \times (f - int(2^{e-d} \times g) - carry + (carry - nonint(2^{e-d} \times g))) \end{aligned}$$

If $carry$ is 0 or 1 as $nonint(2^{e-d} \times g)$ is zero or non-zero then simple manipulations show that we have enough information to calculate the sum or difference accurately Thus, the first step in both operations is to align the fractions: the least significant bit of $carry$ is set if and only if any set bits are shifted out; the exponent of the result is set to the greater of the two arguments. Its specification:

Aligned

$$\begin{array}{|l} Internal_x ; Internal_y \\ carry : 0..1 \\ wexp : Word \\ \hline Unnormalised_x \lor Unnormalised_y \\ wexp.int = max\{exp_x, exp_y\} \\ wsign_x.nat = 2^{wl-1} \times sign_x \\ wsign_y.nat = 2^{wl-1} \times sign_y \\ wfrac_x.nat = frac_x \text{ DIV } 2^{wexp.int - exp_x} \\ wfrac_y.nat = frac_y \text{ DIV } 2^{wexp.int - exp_y} \\ carry.nat = 0 \Leftrightarrow \left(\begin{array}{c} frac_x \text{ MOD } 2^{wexp.int - exp_x} = 0 \\ \land \\ frac_y \text{ MOD } 2^{wexp.int - exp_y} = 0 \end{array} \right) \end{array}$$

$$\begin{array}{rc} Align \mathrel{\widehat{=}} & \{wexp_x, wfrac_x, wexp_y, wfrac_y\} \\ & \lhd\ Unnormalised_x ; Unnormalised_y ; Aligned' \mid \\ & value_x = value_x{}' \land value_y = value_y{}' \rhd \\ & \{wfrac_x, wfrac_y, wexp, carry\} \end{array}$$

The following is a proof of the procedure which ignores the values of variables associated with y. The proof can be extended simply to include these:

```
PROC Align (VALUE wexp_x, wexp_y, VAR wfrac_x, wfrac_y, wexp, carry) =
  {Unnormalised_x ∧ x0 = Internal_x}
  SEQ
    IF
      wexp_x >= wexp_y
        SEQ
          wexp := wexp_x
          {wexp.int = max{exp_x, exp_y}}
          IF
            (wexp_x-wexp_y) <= wl
              SHIFTRIGHT (wfrac_y,carry,wfrac_y,Zero,wexp_x-wexp_y)
            (wexp_x-wexp_y) > wl
              SEQ
                carry    := wfrac_y
                wfrac_y := Zero
        {x0 = Internal_x}
      wexp_y >= wexp_x
        SEQ
          wexp := wexp_y
          {wexp.int = max{exp_x, exp_y}}
          IF
            (wexp_y-wexp_x) <= wl
              SHIFTRIGHT (wfrac_x,carry,wfrac_x,Zero,wexp_y-wexp_x)
            (wexp_y-wexp_x) > wl
              SEQ
                carry := wfrac_x
                wfrac_x := Zero
        {carry = 0 ⇔ frac_x MOD 2^(wexp.int-exp_x) = 0}
        {wfrac_x.nat = frac_x DIV 2^(wexp.int-exp_x)}
    IF
      carry = 0
        SKIP
      carry <> 0
        carry := 1 :
    {carry = 0 ⇔ frac_x MOD 2^(wexp.int-exp_x) = 0 ∧ carry ∈ 0..1}
```

2.6.2 Addition

This procedure will deal with addition of numbers with like signs or subtraction of numbers with opposite signs:

$$
\begin{array}{rc}
Add \mathrel{\hat=} & \{wsign_x, wfrac_x, op, wsign_y, wfrac_y, wexp\} \\
& \lhd\ Aligned; Normal' \wedge Value\_Spec \mid \\
& (op = add \wedge sign_x = sign_y \vee op = sub \wedge sign_x \neq sign_y) \rhd \\
& \{wsign, wexp, wfrac\}
\end{array}
$$

$$
\begin{array}{rc}
\vdash Align; Add = & \{wsign_x, wexp_x, wfrac_x, op, wsign_y, wexp_y, wfrac_y,\} \\
& \lhd (Unnormalised_x; Unnormalised_y; Normal') \wedge Value\_Spec \mid \\
& (op = add \wedge sign_x = sign_y \vee op = sub \wedge sign_x \neq sign_y) \rhd \\
& \{wsign, wexp, wfrac\}
\end{array}
$$

Once the fractions have been aligned, they are added together. If the sum overflows, the result is shifted down by one – its least significant bit is preserved in *carry* and replaced after shifting. The sign of the result will be the same as both arguments.

```
PROC Add =
  {Aligned}
  {(op = sub ∧ sign_x ≠ sign_y) ∨ (op = add ∧ sign_x = sign_y)}
  {wexp.int ≥ EMin}
  VAR carry1:
  SEQ
    LONGSUM (carry1,wfrac,wfrac_x,wfrac_y,Zero)
    {2^wl × carry1.nat + wfrac.nat = wfrac_x.nat + wfrac_y.nat}
    carry := carry ∨ (wfrac ∧ One)
    {carry ∈ 0..1}
    wsign := wsign_x
    wexp  := wexp+carry1
    SHIFTRIGHT (carry1,wfrac,carry1,wfrac,carry1)
    {nonint (2^·· × r) = 0 ⇔ carry = 0 ∧ wfrac<<(fracwidth + 2) = 0}
    wfrac := wfrac ∨ carry :
  {Normal | abs r = abs value_x + abs value_y}
```

2.6.3 Subtraction

This procedure deals with subtraction of numbers with like signs or addition of numbers with different signs. Its specification:

$$
\begin{array}{rc}
Sub \mathrel{\hat=} & \{wsign_x, wfrac_x, op, wsign_y, wfrac_y, wexp\} \\
& \lhd\ Aligned; Normal' \wedge Value\_Spec \mid \\
& (op = add \wedge sign_x \neq sign_y \vee op = sub \wedge sign_x = sign_y) \rhd \\
& \{wsign, wexp, wfrac\}
\end{array}
$$

$$\vdash Align; Sub = \quad \{wsign_x, wexp_x, wfrac_x, op, wsign_y, wexp_y, wfrac_y,\} \\ \lhd(Unnormalised_x; Unnormalised_y; Normal') \wedge Value\_Spec \mid \\ (op = add \wedge sign_x \neq sign_y \vee op = sub \wedge sign_x = sign_y) \rhd \\ \{wsign, wexp, wfrac\}$$

An exception is made if the result will be zero so that the sign can be given correctly. Otherwise, the smaller argument is subtracted from the larger. The following procedure is useful to ensure that the exponent is in the correct range.

```
PROC Normal (VAR sticky) =
  IF
    wfrac = Zero
      {Zero}
      wexp := EMin
    (wexp < EMin) AND (wfrac <> Zero)
      {Denorm}
      SEQ
        sticky := sticky ∨ (wfrac ∧ (NOT ((NOT Zero) << (-wexp))))
        wfrac  := wfrac >> (-wexp)
        wexp   := EMin
    (wexp >= EMin) AND (wfrac <> Zero)
      {Norm ∨ Inf}
      SKIP
    IF
      sticky = Zero
        SKIP
      sticky <> Zero
        wfrac := wfrac ∨ One :
  {Normal}
```

```
PROC Sub =
  {Aligned}
  {(op = sub ∧ sign_x = sign_y) ∨ (op = add ∧ sign_x ≠ sign_y)}
  IF
    (word_x ∧ (NOT MSB)) = (word_y ∧ (NOT MSB))
      {abs value_x = abs value_y}
      IF
        (mode = ToNegInf) AND (wfrac_x <> Zero)
          SEQ
            wsign  := MSB
            wexp   := Zero
            wfrac  := Zero
          {Sign_Of_Zero[Zero/Zero']}
        (mode <> ToNegInf) OR (wfrac_x = Zero)
          SEQ
            wsign  := wsign_x ∧ wsign_y
            wexp   := Zero
            wfrac  := Zero
          {Sign_Of_Zero[Zero/Zero']}
      {Sign_Of_Zero[Zero/Zero']}
    (word_x ∧ (NOT MSB)) <> (word_y ∧ (NOT MSB))
      {abs value_x ≠ abs value_y}
      SEQ
        IF
          (word_x ∧ (NOT MSB)) < (word_y ∧ (NOT MSB))
            {abs value_x < abs value_y}
            SEQ
              wsign := wsign_y
              wfrac := wfrac_y - wfrac_x - carry
          (word_x ∧ (NOT MSB)) > (word_y ∧ (NOT MSB))
            {abs value_x > abs value_y}
            SEQ
              wsign := wsign_x
              wfrac := wfrac_x - wfrac_y - carry
            {wfrac.nat ≥ 2^(wl-2) ∨ (abs value_x - abs value_y = 2^(wexp.int-Bias-wl+1) ∧ carry = 0)}
        VAR places, zero:
        SEQ
          NORMALISE (places,wfrac,zero,wfrac,Zero)
          wexp := wexp - places
        Normal (carry) :
  {Normal | abs r = abs value_x - abs value_y}
```

These procedures are combined in the following procedure which deals with all non-exceptional addition and subtraction:

```
PROC AddSub =
  {Aligned}
  VAR carry:
  SEQ
    Align
    IF
      op = sub
        wsign_y := wsign_y X MSB
      op = add
        SKIP
    IF
      wsign_x = wsign_y
        Add
      wsign_x <> wsign_y
        Sub :
  {Normal ∧ Value_Spec}
```

2.6.4 Multiplication

Specification:

$$Multiply \mathrel{\hat{=}} FiniteArit \mid op = mul$$

After multiplying the fractions, the result is determined exactly The fraction and exponent of the result are then adjusted to satisfy *Normal*. Details of the proof are left as an exercise:

```
PROC Multiply =
  {Unnormalised_x ∧ Unnormalised_y}
  VAR lo:
  SEQ
    wsign := wsign_x X wsign_y
    wexp  := (wexp_x + wexp_y + One) - Bias
    LONGPROD (wfrac,lo,wfrac_x,wfrac_y,Zero)
    VAR places:
    SEQ
      NORMALISE (places,wfrac,lo,wfrac,lo)
      wexp := wexp - places
    Normal (lo) :
  {Normal | r = value_x × value_y}
```

2.6.5 Division

Specification:

$$Divide \mathrel{\hat=} FiniteArit \mid op = div$$

An exception is made when dividing by zero. Both arguments are normalised so that the arguments to *LONGDIV* are in the required range and that the resulting quotient has enough significant digits. The quotient is then adjusted to satisfy *Normal*:

```
PROC Divide =
  {Unnormalised_x ∧ Unnormalised_y}
  {value_y ≠ 0}
  SEQ
    wsign := wsign_x X wsign_y
    SEQ
      {Unnormalised_x}
      VAR places, zero:
      SEQ
        NORMALISE (places,wfrac_x,zero,wfrac_x,Zero)
        wexp_x := wexp_x - places
      {wfrac_x.nat ≥ 2^(wl-1) ∨ wfrac_x.nat = 0}
      {value_x = 2^(wexp_x.int-Bias-wl+1) × wfrac_x.nat}
      {Unnormalised_y}
      VAR places, zero:
      SEQ
        NORMALISE (places,wfrac_y,zero,wfrac_y,Zero)
        wexp_y := wexp_y - places
      {wfrac_y.nat ≥ 2^(wl-1)}
      {value_y = 2^(wexp_y.int-Bias-wl+1) × wfrac_y.nat}
      VAR rem:
      SEQ
        wexp := (wexp_x+Bias) - wexp_y
        LONGDIV (wfrac,rem,wfrac_x >> One,Zero,wfrac_y)
        {value_x = 0 ∨ wfrac.nat ≥ 2^(wl-2)}
        VAR places, zero:
        SEQ
          NORMALISE (places,wfrac,zero,wfrac,Zero)
          wexp := wexp - places
      Normal (rem) :
  {Normal | r = value_x ÷ value_y}
```

Finally, the component parts can be assembled by the following procedure which performs all non-exceptional arithmetic:

```
PROC FiniteArit =
  {Unnormalised_x ∧ Unnormalised_y ∧ value_y ≠ 0}
  VAR wsign, wexp, wfrac:
  IF
    (op = add) OR (op = sub)
      AddSub
    op = mul
      Multiply
    op = div
      Divide :
  {Normal ∧ Value_Spec}
```

Conclusions

It is often heard said that formal methods can only be applied to practically insignificant problems, that development costs in large products are too high, and that the desired reliability is still not achieved. The problem presented here is only a part of a large body of work which has been undertaken to implement a proven-correct floating-point system. This work develops the system from a Z specification to silicon implementation – an achievement which cannot be considered insignificant. The formal development was started some time after the commencement of an informal development and has since overtaken the informal approach. The reason for this was mainly because of the large amount of testing involved in the intermediate stages of an informal development – a process which becomes less necessary with a formal development.

As for reliability, that remains to be seen. However, the existence of a proof of correctness means that mistakes are less likely and can be corrected with less danger of introducing further mistakes. Errors can arise in two ways: first, a simple mistype in the program; or a genuine error in the proof. Because of the steps in the development, the effect of this can be limited. Either, a fragment of program is wrong and can be corrected without affecting any larger scale properties of the program; or, the initial decomposition was at fault, in which case most of the development may have to be reworked. If the last scenario seems a little dire, remember that decomposition is a prerequisite of any structured programming methodology but errors at this stage are more likely to be discovered in a formal development. Furthermore, there are now two ways to discover bugs and a way to show that they are not present. The possibility of automatic proof-checkers gives some hope that programmers will be able to guarantee the quality of a program more reliably than an architect can guarantee the robustness of a house.

This example, however, does demonstrate some of the advantages which can be gained from a formal specification. Specifications often become modified – either the customer changes her mind or the original description of the problem is found to be at fault.

Trying to modify a badly documented system is disastrous. Trying to modify a well documented system is, at best, error prone. Using a formal specification, it is possible to determine which parts of the system to change and, moreover, how to change them without affecting unmodified parts. For instance, if the specification of error conditions were to change, it would be possible to prove that only the second part of the rounding module and, perhaps, its precondition need be changed. The modifications can take place without having to resort to various pieces of code. Likewise, in the development stage, the formalism exists to reason about how proposed modules will fit together. Moreover, modules may be reused with greater confidence because there is a precise description of what each one does.

The advantages of a non-algorithmic formalism speak for themselves. The language used here bears a formal relation to its implementation and can be transformed to emulate the structure of a program. On the other hand, the high-level specification can be written to bear a close relationship to a natural language description – there are many mathematical idioms which already exist to formalise seemingly intractable descriptions. This paper has assumed some familiarity with the IEEE Standard, but it is desirable to use the formalism as a supplement to a natural language specification to which reference can be made in case of ambiguity.

Acknowledgements

Thanks are due to David Shepherd, Michael Goldsmith, Bill Roscoe, Tony Hoare and Jim Woodcock for comments and encouragement. This work was carried out under an Alvey Research Project in collaboration with inmos.

Bibliography

[Abrial] Abrial, J-R., Schumann, S.A. & Meyer, B. Specification Language Z. *Massachusetts Computer Associates, Inc.* 1979.

[Dijkstra] Dijkstra, E.W. A discipline of programming. *Prentice-Hall,* 1976

[Gries] Gries, D. The science of programming. *Springer-Verlag,* 1981

[Hayes] Hayes, I. (ed.) Specification Case Studies. *Prentice Hall,* 1987

[Hoare] Hoare, C.A.R. An axiomatic Basis for Computer Programming. *CACM* 12 *(*1969*). pp.*576-580,583.

[IEEE] IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Std* 754-1985, *New York. August,* 1985.

[inmos] inmos, ltd. The OCCAM Programming Manual. *Prentice Hall.* 1984

[Z] Sufrin, B.A., editor. The Z Handbook. *Programming Research Group, Oxford University. March, 1986.*

Appendix A

Standard Functions and Procedures

A.1 The Data Type

$$wl : \mathbb{N}$$

Word

$$\begin{array}{l} bitset : \mathbb{P}(0..(wl - 1)) \\ nat \quad : 0..(2^{wl} - 1) \\ int \quad : (-2^{wl-1})..(2^{wl-1} - 1) \end{array}$$

$$\begin{array}{lcl} nat & = & \Sigma\, i : bitset \bullet 2^{i} \\ int & = & (2 \times nat) \,\mathrm{MOD}\, 2^{wl} - nat \end{array}$$

A.2 Bit Operations

$$\mathtt{NOT} : Word \rightarrow Word$$

$$(\mathtt{NOT}\ w).bitset = 0..(wl - 1) - w.bitset$$

$$\wedge, \vee, \mathrm{X} : Word \rightarrow Word$$

$$\begin{array}{l} (w_1 \wedge w_2).bitset = w_1.bitset \cap w_2.bitset \\ (w_1 \vee w_2).bitset = w_1.bitset \cup w_2.bitset \\ (w_1\ \mathrm{X}\ w_2).bitset = w_1.bitset \Delta w_2.bitset \end{array}$$

A.3 Boolean Values

$$\mathtt{TRUE}, \mathtt{FALSE} : Word$$

$$\begin{aligned} \mathtt{TRUE}.bitset &= 0..wl - 1 \\ \mathtt{FALSE}.bitset &= \{\} \end{aligned}$$

$$Bool \mathrel{\widehat{=}} \{\mathtt{TRUE}, \mathtt{FALSE}\}$$

$$\begin{aligned} \vdash \quad \mathtt{NOT}\ \mathtt{TRUE} &= \mathtt{FALSE} \\ \mathtt{NDT}\ \mathtt{FALSE} &= \mathtt{TRUE} \end{aligned}$$

$$\mathtt{AND}, \mathtt{OR} : Bool \times Bool \rightarrow Bool$$

$$\begin{aligned} \mathtt{FALSE}\ \mathtt{AND}\ b &= \mathtt{FALSE} \\ \mathtt{TRUE}\ \mathtt{AND}\ b &= b \\[1ex] \mathtt{TRUE}\ \mathtt{OR}\ b &= \mathtt{TRUE} \\ \mathtt{FALSE}\ \mathtt{OR}\ b &= b \end{aligned}$$

A.4 Shift Operations

$$>>, << : Word \times Word \nrightarrow Word$$

$$\begin{aligned} & n.int \geq 0 \\ & \Rightarrow \\ & (w>>n).bitset = (0..wl - 1) \cap succ^{-n.int}(w.bitset) \\ & (w<<n).bitset = (0..wl - 1) \cap succ^{n.int}(w.bitset) \end{aligned}$$

A.5 Comparisons

$$<, >, <=, >=, \equiv, <> : Word \times Word \rightarrow Bool$$

$$\begin{aligned} & w_1.int < w_2.int \Leftrightarrow w_1 < w_2 = \mathtt{TRUE} \\ & w_1 = w_2 \Leftrightarrow w_1 \equiv w_2 = \mathtt{TRUE} \\ & w_1 > w_2 = w_2 < w_1 \\ & w_1 <= w_2 = \mathtt{NOT}(w_1 > w_2) \\ & w_1 >= w_2 = w_2 <= w_1 \\ & w_1 <> w_2 = \mathtt{NOT}(w_1 \equiv w_2) \end{aligned}$$

A.6 Arithmetic

$$+, -, \times : Word \times Word \rightarrow Word$$

$$(w_1 + w_2).nat = (w_1.nat + w_2.nat) \text{ MOD } 2^{wl}$$
$$(w_1 - w_2).nat = (w_1.nat - w_2.nat) \text{ MOD } 2^{wl}$$
$$(w_1 \times w_2).nat = (w_1.nat \times w_2.nat) \text{ MOD } 2^{wl}$$

$$/, \backslash : Word \times Word \nrightarrow Word$$

$$w_2.int \neq 0$$
$$\Rightarrow$$
$$w_1.int = (w_1 / w_2).int \times w_2.int + (w_1 \backslash w_2).int$$
$$(w_2.int > 0 \land 0 \leq (w_1 \backslash w_2).int < w_2.int$$
$$\lor$$
$$w_2.int < 0 \land w_2.int < (w_1 \backslash w_2).int \leq 0)$$

A.7 Shift Procedures

SHIFTLEFT

$$hi', lo' : Word$$
$$hi, lo : Word$$
$$n : Word$$

$$n.int \geq 0$$
$$2^{wl} \times hi'.nat + lo'.nat = ((2^{wl} \times hi.nat + lo.nat) \times 2^{n}) \text{ MOD } 2^{2 \times wl}$$

SHIFTRIGHT

$$hi', lo' : Word$$
$$hi, lo : Word$$
$$n : Word$$

$$n.int \geq 0$$
$$2^{wl} \times hi'.nat + lo'.nat = (2^{wl} \times hi.nat + lo.nat) \text{ DIV } 2^{n}$$

NORMALISE

$hi', lo' : Word$
$hi, lo : Word$
$places' : Word$

$n.int \geq 0$
$2^{wl} \times hi'.nat + lo'.nat = (2^{wl} \times hi.nat + lo.nat) \times 2^{places'}$
$wl - 1 \in hi'.bitset \lor hi'.nat = 0 = lo'.nat \land places' = 2 \times wl$

A.8 Arithmetic Procedures

LONGSUM

$carry', z' : Word$
$x, y, carry : Word$

$carry.nat \in 0..1$
$2^{wl} \times carry'.nat + z'.nat = x.nat + y.nat + carry.nat$

LONGDIFF

$borrow', z' : Word$
$x, y, borrow : Word$

$borrow.nat \in 0..1$
$-2^{wl} \times borrow'.nat + z'.nat = x.nat - y.nat - borrow.nat$

LONGPROD

$hi', lo' : Word$
$x, y, carry : Word$

$2^{wl} \times hi'.nat + lo'.nat = x.nat \times y.nat + carry.nat$

LONGDIV

$quot', rem' : Word$
$hi, lo, y : Word$

$2^{wl} \times hi.nat + lo.nat < 2^{wl} \times y.nat$
$2^{wl} \times hi.nat + lo.nat = quot' \times y.nat + rem'.nat$
$0 \leq rem' < y.nat$

Index