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 Copyright © 1987 Geoff Barrett

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

1	Spee	cification 3
	1.1	Rounding
	1.2	Addition, Subtraction, Multiplication and Division 8
	1.3	Remainder
	1.4	Square Root
	1.5	Floating Point Format Conversions
	1.6	Rounding and Converting to Integers
		I.6.1 Conversions to Integer Formats
		1.6.2 Rounding to Integer
	1.7	Comparisons
2	Imp	lementation 20
	2.1	Foreword to the Proof
	2.2	Representations of FP 22
	2.3	Representing Real Numbers
	2.4	Unpacking and Denormalising
		2.4.1 Unpacking
		2.4.2 Denormalising 26
	2.5	Rounding and Packing 26
		2.5.1 Rounding 26
		2.5.2 Packing
	2.6	Finite Arithmetic Procedures
		2.6.1 Addition and Subtraction
		2.6.2 Addition
		2.6.3 Subtraction
		2.6.4 Multiplication
		2.6.5 Division
A	Sta	ndard Functions and Procedures 41
	A.I	The Data Type
	A.2	Bit Operations
	A.3	Boolean Values
	A.4	Shift Operations
	A.5	Comparisons
	A.6	Arithmetic
	A .7	Shift Procedures
	A.8	Arithmetic Procedures

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 OGGAM (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 welldefined 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 round-ing 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 ezpwidth, fracwidth :N wordlength :N EMin, EMax, Bias :N wordlength = ezpwidth + fracwidth + 1 EMin = 0 EMax = $2^{expwidth} - 1$ Bias = $2^{expwidth} - 1$

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

Single \Rightarrow Format | expected h = 8 \land wordlength = 32Double \Rightarrow Format | expected h = 11 \land wordlength = 64SingleExtended \Rightarrow Format | expected h \ge 11 \land wordlength \ge 43DoubleExtended \Rightarrow Format | expected h \ge 15 \land wordlength \ge 79

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

¹The 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

```
Fields

Format

nat :N

sign :0..1

exp, frac: N

nat = sign × 2<sup>wordlength-1</sup> + exp × 2^{fracwadth} + frac

exp < 2^{cepoidh}

frac < 2^{fracwidth}
```

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

$$NaNF \triangleq Fields \mid frac \neq 0 \land exp = EMax$$

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)

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)

 $\begin{array}{ll} NaN & \triangleq FP \mid frac \neq 0 \land ezp = EMaz \\ Inf & \triangleq FP \mid frac = 0 \land ezp = EMaz \\ Norm & \triangleq FP \mid EMin < ezp < EMaz \\ Denorm \triangleq FP \mid frac \neq 0 \land ezp = EMin \\ Zero & \triangleq FP \mid frac = 0 \land ezp = EMin \\ Finite \triangleq Norm \lor Denorm \lor Zero^3 \end{array}$

²This 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.

Mode ::= ToNearest | ToZero | ToNegInf | ToPosInf

 $\begin{array}{l} Round_Signature \\ \hline \\ r: \mathbb{R}; mode : Modes \\ Approx Values, Preferred : P\mathbb{R} \\ Min Value, Max Value : \mathbb{R} \\ \hline \\ value' : \mathbb{R} \\ \hline \\ Preferred \cup \{value'\} \subseteq Approx Values \cup \{Min Value, Max Value\} \\ 0 \in Approx Values \\ \forall value_1, value_2 : Approx Values \cup \{Min Value, Max Value\} | value_1 > value_2 \\ \exists p : Preferred \bullet value_1 \ge p \ge value_2 \\ \hline \\ \forall value : Approx Values \bullet Min Value \le value \le Max Value \\ \end{array}$

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

 $\begin{array}{l} \textit{Round_Signature} \\ \hline r > \textit{MazValue} \Rightarrow \textit{value'} = \textit{MazValue} \\ r \leq \textit{MazValue} \Rightarrow \textit{value'} \geq r \\ \forall \textit{value} : \textit{ApproxValues} \cup \{\textit{MazValue}\} \mid \textit{value} \geq r \\ & \textit{value} \geq \textit{value'} \end{array}$

³Logical operators between schemas have the effect of merging the signatures and performing he logical operation between the pred icates.

 $\begin{array}{c} Below _ \\ \hline Round_Signature \\ \hline r < MinValue \Rightarrow value' = MinValue \\ r \ge MinValue \Rightarrow value' \le r \\ \forall value : ApproxValues \cup \{MinValue\} \mid value \le r \\ value \le value' \end{array}$

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

> Round ToZero Round _Signature mode = ToZero $(r \ge 0 \land Below$ \lor $r \le 0 \land Above)$

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

> RoundToPosInf Round_Signature mode = ToPosInf Above

RoundToNegInf _____

Round_Signature

mode = ToNegInf Below

When rounding to nearest, the closest approximation is returned, but if both are

equally good, a member of the set Preferred is returned: *

```
\begin{array}{c} RoundToNearest \\ \hline \\ Round_Signature \\ \hline \\ \hline \\ \hline \\ mode = ToNearest \\ \exists Above_1; Below_2 \mid r_1 = r = r_2 \bullet \\ \forall alue_1 - r < r - value_2 \land Above \\ \hline \\ value_1 - r > r - value_2 \land Below \\ \hline \\ value_1 - r = r - value_2 \land \\ (value_1 = value_2 \land Above \land Below \\ \hline \\ \\ value_1 \neq value_2 \land value' \in Preferred \land (Above \lor Below)) \end{array}
```

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

Round = Round To Nearest V Round To Zero V Round To PosInf V Round To NegInf

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 *Approz Values*, *Preferred*, *Min Value* and *Maz-Value*. 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 *Approz Values* and that *Min Value* and *Max Value* satisfy the constraint of *Round signature*.

FP_Round1

FP_Round2 = FP_Round1 \{ Approz Values, Preferred, Min Value, Maz Value }

⁴Decorating 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:

(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:

 $FP_Round \cong FP_Round2 \land Error_Spec$

1.2 Addition, Subtraction, Multiplication and Division

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

$$Ops ::= add | sub | mul | div$$

The essential ingredients of an arithmetic operation are two numbers, FP_{\star} and FP_{μ} , 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

 FP_z ; FP_p ; op : Ops

 FP'

 wordlength' \geq wordlength,

 wordlength' \geq wordlength,

When both FP_s and FP_s 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:

Value_Spec A rit_Signature \land Finite, \land Finite, $r : \mathbb{R}$ $op = add \land r = value, + value,$ $\lor op = sub \land r = value, - value,$ $\lor op = mul \land r = value, \times value,$ $\lor op = div \land value, \neq 0 \land r = value_{z} \div value,$

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:

Sign_Bit

 $\begin{array}{l} Arit_{Signature} \\ mode: Modes; r : \mathbb{R} \\ \hline \\ (-1)^{nign'} \times abs \ r = r \\ (op = mul \lor op = div) \Rightarrow (-1)^{nign'} = (-1)^{nign_{x} + ngn_{y}} \\ (op = add \lor op = sub) \land r = 0 \Rightarrow \\ (Zero_{x} \land Zero_{y} \land (sign_{x} = sign_{y} \Leftrightarrow op = add) \land sign' = sign_{x} \\ & \lor \\ \neg (Zero_{x} \land Zero_{y}) \land (sign_{x} = sign_{y} \Leftrightarrow op = add) \land (sign' = 1 \Leftrightarrow mode = ToNegInf)) \end{array}$

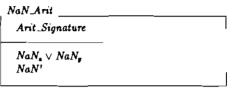
Rounding has already been described so operations on finite numbers may be defined by using FP_Round to specify the relation of r to FP':

 $Finite_Arit \stackrel{\circ}{=} (Value_Spec \land FP_Round \land Sign_Bit) \setminus \{r\}$

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

 Div_By_Zero $Arit_Signature$ $Finite_{z} \land Zero_{y}$ op = div $(Zero_{z} \land NaN') \lor (\neg Zero_{z} \land Inf' \land (-1)^{nign'} = (-1)^{nign_{z} - nign_{y}})$

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):



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 ¬(NaN_s ∨ NaN_g) Inf_s ∨ Inf_g

Inf_Add _____

Inf_Arit_Signature

op = add $(infsigns = \{sign'\} \land Inf') \lor (infsigns = \{0, 1\} \land NaN')$ where infsigns = {Inf | Inf = FP_* \lor Inf = FP_* • sign}

Inf_Sub _____

Inf Arit_Signature

 $\begin{array}{l} op = sub \\ (infsigns = \{sign'\} \land Inf') \lor (infsigns = \{0, 1\} \land NaN') \\ \textbf{where } infsigns = \{Inf \mid Inf = FP_s \lor Inf = FP_s[-value_s/value_s] \bullet sign\} \end{array}$

Inf_Div Inf_Arit_Signature op = div $(Inf_{a} \wedge Inf_{y} \wedge NaN'$ \vee Finite_y $\wedge Inf' \wedge (-1)^{ngn'} = (-1)^{ngn_{a} - ngn_{y}}$ \vee Finite_a $\wedge Zero' \wedge (-1)^{ngn'} = (-1)^{ngn_{a} - ngn_{y}}$

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

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

 $No_Round_Errors \cong round_errors' : PRound_Errors | round_errors' = \{ \}$

FP_Arit ≈ Finite_Arit ∨ No_Round_Errors ∧ (Div_By_Zero ∨ NaN_Arit ∨ Inf_Arit)

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 | mul_Zero_Inf | div_Zero | div_Inf_Inf | Mag_sub

Error_Spec

Finally, the whole specification is:

```
Arit = FP_Arit 		 Error_Spec
```

1.3 Remainder

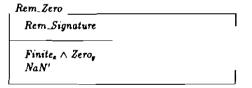
To calculate remainder, all that is necessary is a divisor and a dividend, FP_s and FP_g . The result will be given by FP'. The signature is:



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

Fin_Rem _____ Rem_Signature Finite, \wedge Finite, $\neg Zero$, $2 \times abs value' \leq abs value$, $\exists n : \mathbb{Z} \bullet$ $value_{s} = n \times value$, + value' $2 \times abs value' = abs value$, $\Rightarrow n \text{ MOD } 2 = 0$

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



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

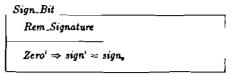
NaN_Rem _____ Rem_Signature NaN_s ∨ NaN_y NaN'

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

number by infinity is the original number:

 $Inf_Rem ______ Rem_Signature ______ Inf_* \land \neg NaN_y \land NaN' \lor \lor Finite_* \land Inf_y \land FP' = FP_*$

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



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:

Rem_Errors ::= NaN_Op | rem_Zero_Inf

Error_Spec

Putting all the pieces together gives the full specification:

 $Rem \triangleq (Fin_Rem \lor Rem_Zero \lor NaN_Rem \lor Inf_Rem) \land Sign_Bit \land Error_Spec$

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:

 Exact_Sqrt

 FP

 $r : \mathbb{R}$

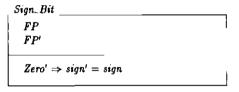
 Finite

 value ≥ 0
 $r \times r = value$
 $r \geq 0$

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

 $Pos_Sqrt \doteq (Exact_Sqrt \land FP_Round) \mid wordlength \leq wordlength' \setminus \{r\}$

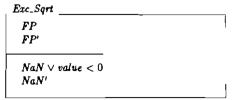
The sign of zero is unchanged:



The square root of positive infinity is infinity:

Inf_Sqrt	
Inf FP'	1
<i>FT</i>	_
sign = 0	
FP' = Inf	1

In all other cases, the result is a NaN:



There are two errors - NaN. Op and when the operand is less than zero:

 $Sqrt_Errors = NaN_Op \mid OpLT0$

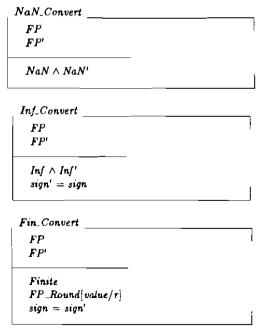
 $Error_Spec _ FP \\ FP'; errors' : P Sqrt_Errors \\ NaN_Op \in errors' \Leftrightarrow NaN \\ OpLT0 \in errors' \Leftrightarrow \neg NaN \land value < 0 \\ \end{bmatrix}$

Putting the pieces together:

Sqrt = (Pos_Sqrt \ Inf_Sqrt \ Ezc_Sqrt) \ Sign_Bit \ 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:



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.

 $Integer_Round1 ______Round1 ______Round1 ______Round1 ______Round1 _____Round1 ______Round1 _____Round1 ______Round1 _____Round1 ______Round1 Round1 ______ROUND1 Round1 _______ROUND1 Round1 =_____ROUND1 Round1 =_____ROUND1 Round1 =______ROUND1 Round1 =_____ROUND1 Round1 =______ROUND1 Round1 =_______ROUND1 Round1 =______ROUND1 Round1 =______ROUND1 Round1 =_______ROUND1 Round1 =______ROUND1 Round1 =______ROUND1 Round1 =______ROUND1 Round1 =______ROUND1 Round1 =______ROUND1 Round1 =______ROUND1 ROUND1 ROUND1$

```
Integer_Round \triangleq Integer_Round1\{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:

$Int_Conv_Round \cong$

Integer_Round | Min Value = MinInt \ MaxValue = MaxInt \ {Min Value, MaxValue}

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

Exc_Conv _____ FP Integer' ______ NaN ∨ Inf ∨ value < MinInt ∨ MaxInt > value

The specification is:

```
Convert\_Integer \cong 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:

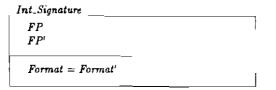
Rnd_Int_Round1

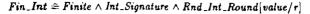
Integer_Round

 $\begin{array}{rcl} \textit{MaxValue} &=& \sup \left\{ \textit{FP'} \mid \textit{Format} = \textit{Format'} \bullet \textit{value} \right\} \cap \mathbb{Z} \\ \textit{MinValue} &=& \inf \left\{ \textit{FP'} \mid \textit{Format} = \textit{Format'} \bullet \textit{value} \right\} \cap \mathbb{Z} \end{array}$

Rnd_Int_Round = Rnd_Int_Round1 \{MinValue, MaxValue}

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





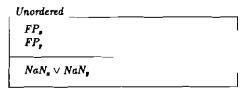
In this case, Inf and NaN are preserved:

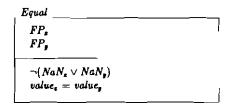
 $Inf_NaN_Int ______Inf_Signature$ $Inf \lor NaN$ FP = FP'

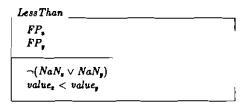
The whole specification:

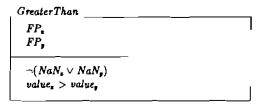
1.7 Comparisons

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









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

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

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

Bool ::= true | false

Compare_Bool $FP_{z}; FP_{y}; op : P Conditions$ result' : Bool $op \neq \{\}$ $op \neq Conditions$ result' = true $\Leftrightarrow \exists condition' : op \bullet Compare_Condition$

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:

 $Compare_Bool_Error _____Compare_Bool_exception : Bool = NaN_Op \in errors' \Leftrightarrow Unordered \land exception = true \\ (op = {EQ} \lor op = Conditions - {EQ}) \Rightarrow exception = false$

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.
- 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 PP or 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. $\{\phi\} [P] \{\psi\}$, 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:

$$\vdash \{\phi\}$$
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 z before:

$$\vdash \{\mathcal{D}\mathbf{e} \land \phi[\mathbf{e}/\mathbf{x}]\} \mathbf{x} := \mathbf{e} \{\phi\}$$

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 :

$$\{\phi\}\mathbb{P}\{\psi\} \land \{\psi\}\mathbb{Q}\{\chi\} \vdash \{\phi\} \left| \begin{array}{c} \operatorname{SEQ} \\ \mathbb{P} \\ \mathbb{Q} \\ \end{array} \right| \{\chi\}$$

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 :

$$\{b_{P} \land \phi\} \mathbb{P}\{\psi\} \land \{b_{Q} \land \phi\} \mathbb{Q}\{\psi\} \vdash \{(b_{P} \lor b_{Q}) \land \phi\} \begin{bmatrix} \mathsf{IF} \\ b_{P} \\ P \\ b_{Q} \\ Q \end{bmatrix} \{\psi\}$$

Rule 5 The precondition of a program may be strengthened:

$$(\chi \Rightarrow \phi) \land \{\phi\} \mathbb{P} \{\psi\} \vdash \{\chi\} \mathbb{P} \{\psi\}$$

Rule 6 The postcondition of a program may be weakened:

$$(\chi \leftarrow \psi) \land \{\phi\} \mathbb{P} \{\psi\} \vdash \{\phi\} \mathbb{P} \{\chi\}$$

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

int :
$$\mathbb{R} \to \mathbb{N}$$

nonint : $\mathbb{R} \to \mathbb{R}$
 $r = int r + nonint r$
abs (nonint r) < 1
abs (int r) \leq abs r

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:

 $FPS32 \cong FP \mid Single \land wordlength = wl$

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:

 $External \triangleq FPS32$; word : Word | 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 = FPS32; wsign, werp, 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^{el-1} wexp.int = exp wfrac.nat = frac × 2^{enpwidth+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 winc.nat:

Unnormalised ______ Internal ______ Finite $value = (-1)^{min} \times 2^{max-ml-Bin-ml+1} \times wfrac.nat$ $wsign.nat = sign \times 2^{ml-1}$ 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:

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

+ abs value ≤ abs value' ⇔ nat MOD 2 condicate -1 < nat' MOD 2 condicate -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
Finite

abs value < abs value_0

\forall FP' \mid abs value < abs value' \cdot abs value_0 \le abs value
```

```
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 r :R Succ; guard, sticky :0..1 $r > 0 \Rightarrow sign = 0 \land Below[value/value']$ $r = 0 \Rightarrow Zero$ $r < 0 \Rightarrow sign = 1 \land Above[value/value']$ guard = 0 \Leftrightarrow $r - value < value_0 - r$ sticky = 0 \Leftrightarrow $r - value = value_0 - r \lor r = value$

This is, however, not quite enough information to return the correct overflow condition. If $r \ge 2^{EMet'-Bist'}$, 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_Befo Error_Sig	
overflow	\in errors' \Leftrightarrow abs $r \geq 2^{EMas' - Bias'}$
Error_After	
Error_Sign	ature; errors : PErrors
overflow	\in errors \Leftrightarrow abs $r \geq 2^{EM_{GS}' - Bias'}$
inezact	\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:

$Packed \cong ((\exists Succ \bullet Bounds) \land External \land Error_After) \setminus \{errors'\}$

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^{el}), and demand that the fraction be at least 2^{el-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

¹It 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 \subseteq is used to indicate that a design decision has been made.

the weakest condition:

```
Normal

r: R

Internal

wexp.int \geq EMin

wexp.int \geq EMin \Rightarrow wfrac.nat \geq 2^{ul-1}

abs (approx - exact) < 0

nonint approx = 0 \Leftrightarrow nonint exact = 0

where approx = (-1)^{unips.ad} \times 2^{1-cepuidt} \times wfrac.nat

exact = 2^{Biss-ucep.int+2+freewordt} \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, Dne, NSB, INF : Word Zero.nat = 0 Dne.nat = 1 NSB.bitset = $\{wl - 1\}$ INF.nat = $2^{precently} \times EMaz$

2.4.1 Unpacking

The specification of the procedure:

 $Unpack \cong \{word\} \triangleleft External; Unpacked' \mid FP = FP' \triangleright \{wsign, wexp, wfmc\}^2$

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:

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

```
PROC Unpack (VALUE word, VAR wsign, we:p, wfrac) =
  {External}
  SEQ
  wsign := word A NSB
  {Unpacked\{wexp, wfrac}}
  SHIFTLEFT (wexp, wfrac, Zero, word << One, expwidth) :
  {Unpacked}
  The following thms theorems about integers are useful in the details of the pro-
  The following time theorems about integers are useful in the details of the pro-
  The following time theorems about integers are useful in the details of the pro-
  The following time theorems about integers are useful in the details of the pro-
  Section 2012 and 2012 an
```

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

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

2.4.2 Denormalising

The specification:

```
Denormalise \triangleq \{wexp, wfrac\} \triangleleft Unpacked; Unnormalised' \mid FP = FP' \triangleright \{wexp, wfrac\}
```

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

```
PROC Demormalise (VAR werp, wfrac) =
  { Unpacked}
  IF
  werp = ENin
    {Denorm}
    SKIP
  werp <> ENin
    {Norm}
    wfrac := NSE ∀ (wfrac >> One) :
    {Unnormalised}
```

2.5 Rounding and Packing

This section aims to specify and prove procedures for converting between representations of \mathbf{R} .

2.5.1 Rounding

Specification:

```
Round \Rightarrow {word, guard, sticky, mode, errors} \triangleleft Packed \land FP_Round \land Error_After \triangleright {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:

 $\vdash Round_Proc \Leftrightarrow (\exists r : R; FP_0 \bullet FP_Round2 \land Bounds); Error_After$

And the second observation can be formalised as:

$$\exists r : \mathbb{R}; FP_0 \bullet FP_Round2 \land Bounds$$

$$\Leftrightarrow$$

$$\exists r : \mathbb{R}; FP_0 \bullet FP_Round2 \mid mode = ToNearest \land Bounds$$

$$\lor$$

$$\exists r : \mathbb{R}; FP_0 \bullet FP_Round2 \mid mode = ToPosInf \land Bounds$$

$$\lor$$

$$\exists r : \mathbb{R}; FP_0 \bullet FP_Round2 \mid mode = ToNegInf \land Bounds$$

$$\lor$$

$$\exists r : \mathbb{R}; FP_0 \bullet FP_Round2 \mid mode = ToZero \land Bounds$$

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 \land (sticky \lor nat)). This is derived from: nat := nat + ((guard \land sticky) \lor (guard \land (nat \land One))). Using guard = guard \land One and the commutativity and associativity of \land , the last part of the expression reduces to guard \land nat. Now, \land distributes through \lor 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 mat, errors) =
\{overflow \in errors \Leftrightarrow r \geq 2^{EMax-Bas}\}
\{r > 0 \Rightarrow Below[FP/FP']\}
\{r \leq 0 \Rightarrow Above[FP/FP']\}
SEQ
  IF
     mode = ToZero
        SKIP
        {FP_Round2[FP/FP']}
     mode = ToNegInf
        İF
          sign = Zero
             SKIP
          sign ≠ Zero
             nat := nat + One
             \{FP\_Round2[FP/FP']\}
     moda = ToPosInf
        TF
          sign = Zero
             nat := nat + One
          sign ≠ Zero
             SKIP
             \{FP\_Round2[FP/FP']\}
     mode = ToNearest
        nat := nat + (guard \land (sticky \lor nat))
        {FP_Round2[FP/FP']}
  \{overflow \in errors \Leftrightarrow r \geq 2^{EMax-Bias}\}
  errors := errors ∩ {overflow}
  {underflow, inexact \notin errors}
  \{overflow \in errors \Leftrightarrow r > 2^{EMax - Bas}\}
  IF
     Inf
        errors := errors U {overflow}
     ⊐ Inf
        SKIP
  \{overflow \in errors \Leftrightarrow Inf \lor r \geq 2^{EMas-Bias}\}
  {underflow ∉ errors}
  IF
     Denom
        errors := errors \cup {underflow}
     - Denorm
        SKIP
  \{underflow \in errors \Leftrightarrow Denorm\}
```

```
{inezact ∉ errors}
IF
 (sticky ∨ guard) ≠ Zero
 errors := errors ∪ {ineract}
 (sticky ∨ guard) = Zero
    SKIP
{inezact ∈ errors ⇔ r ≠ value}
{FP_Round|FP/FP']}
```

2.5.2 Packing

Specification:

 $Pack \cong \{wsign, wezp, wfrac\} \\ \triangleleft (Normal; Packed') \land Error_Before \mid r = r' \triangleright \\ \{word, guard, sticky, errors\}$

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, werp, wfrac, VAR word, guard, sticky, errors) =
{Normal}
SE0
  1F
    wexp = ENin
      SKIP
     werp <> EMin
       wfrac := wfrac << One
  1E
     werp >= ENar
       SEO
         errors := {overflow}
         werp := EMax-One
         wfrac := NOT Zero
     weip < EMax
       errors := {}
  \{overflow \in errors \Leftrightarrow r > 2^{EMax - Bas}\}
  \{Below[abs r/r] \land FP' \Rightarrow exp' = wexp.int \land frac' = w[rac.nat DIV 2^{coprists+1}\}
  SHIFTLEFT (word, sticky, wexp, wfrac, fracwidth+One)
  \{Below[abs r/r] \land FP'|(word >> One), bitset/bitset']\}
  guard := word \wedge One
  IF
     sticky = Zero
       SXIP
     sticky <> Zero
       sticky := One
  word := weign V (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 = {wsign, wezp, wfrac, op, wsign, wezp, wfrac,}

⊲(Unnormalised; Unnormalised; Normal) ∧ Value_Spec ⊳

{wsign, wezp, 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:

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

$$AddSub \triangleq FiniteArit \mid op = add \lor op = sub$$

First, consider the sum of two numbers:

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

and the difference:

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

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:

```
 \begin{array}{c} Aligned \\ \hline Internal_{s}; Internal_{y} \\ carry : 0..1 \\ wexp : Word \\ \hline \\ Unnormalised_{s} \lor Unnormalised_{y} \\ wexp.int = max {exp_{s}, exp_{y} } \\ wsign_{s}.nat = 2^{wl-1} \times sign_{s} \\ wsign_{s}.nat = 2^{wl-1} \times sign_{g} \\ wfrac_{s}.nat = frac_{s} DIV 2^{wexp.ind-exp_{s}} \\ wfrac_{y}.nat = frac_{y} DIV 2^{wexp.ind-exp_{y}} \\ carry.nat = 0 \Leftrightarrow \begin{pmatrix} frac_{s} \text{ MOD } 2^{wexp.ind-exp_{y}} = 0 \\ \wedge \\ frac_{s} \text{ MOD } 2^{wexp.ind-exp_{y}} = 0 \end{pmatrix}
```

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_{\mathbf{x}} \land \mathbf{z}_0 = Internal_{\mathbf{x}}\}
  SEO
     IF
        verp_r >= werp_y
          SEQ
             weip := weip_r
             \{wexp.int = max\{exp_s, exp_s\}\}
             IF
                (wexp_x-wexp_y) <= wl
                  SHIFTRIGHT (wfrac_y, carry, wfrac_y, Zero, wexp_x-wexp_y)
                (weip_r-weip_y) > wl
                  SEQ
                     Carry
                                := wfrac_y
                     wfrac_y := Zero
          \{z_0 = Internal_n\}
        werp_y >= werp_x
          SEQ
             wexp := wexp_y
             \{wezp.int = max\{ezp_s, ezp_s\}\}
             IF
                (werp_v-werp_r) <= wl
                  SHIFTRIGHT (wfrac_x, carry, wfrac_x, Zero, wexp_y-wexp_x)
                (weip_y-weip_z) > wl
                  SEQ
                     carry := wfrac_x
                     wfrac_z := Zero
          \{carry = 0 \Leftrightarrow frac_n \text{ MOD } 2^{wcap.ind-capt} = 0\}
          \{w frac_s.nat = frac_s \text{ DIV } 2^{\operatorname{ecsp.ssd}-c.sp_s}\}
     IF
        carry = 0
          SKIP
        carry <> 0
          carry := 1:
     \{ carry = 0 \Leftrightarrow frac_x \text{ MOD } 2^{scap, ind-cap_x} = 0 \land carry \in 0..1 \}
```

2.6.2 Addition

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

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 \land sign_{u} \neq sign_{v}) \lor (op = add \land sign_{u} = sign_{v})\}
   \{wexp.int > EMin\}
  VAR carryl:
  SEO
     LONGSUN (carryl,wfrac,wfrac_x,wfrac_y,Zero)
     \{2^{m} \times carry1.nat + wfrac.nat = wfrac_s.nat + wfrac_s.nat\}
     carry := carry V (wfrac A One)
     \{carry \in 0..1\}
     wsign := wsign_r
     wexp := wexp+carryl
     SHIFTRIGHT (carryl,wfrac,carryl,wfrac,carryl)
     \{nonint \ (2^m \times r) = 0 \Leftrightarrow carry = 0 \land w frac << (fracwidth + 2) = 0\}
     wfrac := wfrac V carry :
   \{Normal \mid abs r = abs value, + abs value, \}
```

2.6.3 Subtraction

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

$$Sub \doteq \{wsign_{z}, wfrac_{s}, op, wsign_{g}, wfrac_{g}, wezp\} \\ \lhd Aligned; Normal' \land Value_Spec | \\ (op = add \land sign_{z} \neq sign_{y} \lor op = sub \land sign_{s} = sign_{y}) \triangleright \\ \{wsign, wezp, wfrac\}$$

```
+ Align; Sub = {wsign<sub>s</sub>, wexp<sub>s</sub>, wfrac<sub>s</sub>, op, wsign<sub>y</sub>, wexp<sub>y</sub>, wfrac<sub>y</sub>, }
<(Unnormalised<sub>s</sub>; Unnormalised<sub>y</sub>; Normal') ∧ Value_Spec |
(op = add ∧ sign<sub>s</sub> ≠ sign<sub>y</sub> ∨ op = sub ∧ sign<sub>s</sub> = sign<sub>y</sub>) ▷
{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}
      verp :≃ ENin
    (werp < ENin) AND (wfrac <> Zero)
      {Denorm}
      SEQ
        sticky := sticky ∨ (wfrac ∧ (NOT ((NOT Zero) << (-wexp))))</pre>
        wfrac := wfrac >> (-werp)
              :⇒ ENin
        werp
    (werp >= ENin) AND (wfrac <> Zero)
      \{Norm \lor Inf\}
      SXTP
    TF.
      sticky = Zero
        SXIP
      sticky <> Zero
        wfrac := wfrac V Dne :
  {Normal}
```

```
PROC Sub =
  {Aligned}
  \{(op = sub \land sign_s = sign_s) \lor (op = add \land sign_s \neq sign_s)\}
  IF
     (word_x \land (NOT NSB)) = (word_y \land (NOT NSB))
       \{abs value_s = abs value_s\}
       IF
          (mode = ToNegInf) AND (wfrac_x <> Zero)
            SEQ
              wsign := MSB
              werp := Zero
              wfrac := Zero
            {Sign_Of_Zero Zero Zero'}
          (mode <> ToNegInf) OR (wfrac_r = Zero)
            SEQ
              weign := wsign_x \wedge weign_y
              werp := Zero
              wfrac := Zero
            {Sign_Of_Zero[Zero/Zero']}
       {Sign_Of_Zero|Zero/Zero']}
     (word_x \land (NOT MSB)) \iff (word_y \land (NOT MSB))
       {abs value, \neq abs value, }
       SED
         IF
            (word_x \land (NOT MSB)) < (word_y \land (NOT MSB))
              \{abs value_n < abs value_n\}
              SEO
                 wsign := wsign_y
                 wfrac := wfrac_y - wfrac_z - carry
            (word_x \land (NOT NSB)) > (word_y \land (NOT NSB))
              {abs value_ > abs value_}
              SEQ
                wsign := wsign_x
                 wfrac := wfrac_x - wfrac_y - carry
              \{w \text{frac.nat} \geq 2^{v^{l-2}} \lor (abs value_{x} - abs value_{y} = 2^{vccp, int-Bios-vl+1} \land carry = 0)\}
         VAR places, zero:
         SEQ
            NORNALISE (places, wfrac, zero, wfrac, Zero)
            werp := werp - places
         Normal (carry) :
  \{Normal \mid abs r = abs value_s - abs value_s\}
```

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

```
PROC AddSub =
  {Aligned}
  VAR carry:
  SEO
    Align
    TF.
       op = sub
        wsign_y := wsign_y X MSB
       op = add
         SKIP
     IF
       wsign_x = wsign_y
         Add
       wsign_x <> wsign_y
         Sub :
  \{Normal \land Value\_Spec\}
```

2.6.4 Multiplication

Specification:

Multiply
$$\hat{=}$$
 FiniteArit | $op = mul$

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 Wultiply =
  { Unnormalised_ ^ Unnormalised_ }
  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
   NORNALISE (places,wfrac,lo,wfrac.lo)
   wexp := wexp - places
   Normal (lo) :
   {Normal (r = value_ × value_)
}
```

2.6.5 Division

Specification:

$Divide = FiniteArit \mid op = div$

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_ \land Unnormalised_}
  \{value, \neq 0\}
  SEQ
    wsign := wsign_x X wsign_y
     SEQ
       {Unnormalised_}
       VAR places, zero:
       SEO
         NORNALISE (places, wfrac_x, zero, wfrac_x, Zero)
         wexp_x := wexp_x - places
       \{w | rac_s.nat \geq 2^{-l-1} \lor w | rac_s.nat = 0\}
       \{value_z = 2^{wexp_z.int-Bios-wl+1} \times w[rac_z.nal\}
       {Unnormalised}
       VAR places, zero:
       SEO
         NORNALISE (places.wfrac_y.zero.wfrac_y.Zero)
         werp_y := werp_y - places
       \{w|rac_n, nat \geq 2^{wl-1}\}
       {value, = 2 very int - Bias - vi+1 × w(rac, nat}
       VAR rem:
       SEQ
          werp := (werp_x+Bias) - werp_y
          LONGDIV (wfrac.rem.wfrac_x >> One.Zero.wfrac_y)
          \{value_s = 0 \lor w frac.nat \ge 2^{sl-2}\}
          VAR places, zero:
          SEQ
            NORWALISE (places, wfrac, zero, wfrac, Zero)
            werp := werp - places
       Normal (rem) :
  \{Normal \mid r = value_{r} \div value_{r}\}
```

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

```
PROC FiniteArit =
  { Unnormalised_ ∧ Unnormalised_ ∧ value, ≠ 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 witbout 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 witbout 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, Itd. 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
bitset:
$$P(0..(wl - 1))$$

nat: $0..(2^{ol} - 1)$
int: $(-2^{ol-1})..(2^{ol-1} - 1)$
nat: $\Sigma i: bitset \cdot 2^{i}$
int: $(2 \times nat) \text{ MOD } 2^{ol} - nat$

A.2 Bit Operations

NOT : Word \rightarrow Word (NOT w).bitset = 0..(wl - 1) - w.bitset

 $\begin{array}{l} \bigwedge, \bigvee, X : Word \rightarrow Word \\ \hline \\ (w_1 \wedge w_2).bitset = w_1.bitset \cap w_2.bitset \\ (w_1 \vee w_2).bitset = w_1.bitset \cup w_2.bitset \\ (w_1 X w_2).bitset = w_1.bitset \Delta w_2.bitset \end{array}$

A.3 Boolean Values

TRUE, FALSE : Word TRUE .bitset = 0..wl - 1 FALSE .bitset = {} Bool \approx { TRUE , FALSE } \vdash NOT TRUE = FALSE NDT FALSE = TRUE AND , OR : Bool \times Bool \rightarrow Bool FALSE AND b = FALSE TRUE AND b = b TRUE OR b = TRUE FALSE OR b = b

A.4 Shift Operations

>>, <<: Word × Word $\not\prec$ Word n.int ≥ 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)

A.5 Comparisons

 $<,>,<=,>=,\equiv,<>: Word \times Word \rightarrow Bool$ $w_1.int < w_2.int \Leftrightarrow w_1 < w_2 = TRUE$ $w_1 = w_2 \Leftrightarrow w_1 \equiv w_2 = TRUE$ $w_1 > w_2 = w_2 < w_1$ $w_1 <= w_2 = NOT (w_1 > w_2)$ $w_1 >= w_2 = w_2 <= w_1$ $w_1 <> w_2 = NOT (w_1 \equiv w_2)$

A.6 Arithmetic

 $\begin{array}{l} +,-,\times: \ Word \times Word \to Word \\ \hline (w_1+w_2).nat = (w_1.nat+w_2.nat) \ \text{MOD } 2^{wt} \\ (w_1-w_2).nat = (w_1.nat-w_2.nat) \ \text{MOD } 2^{wt} \\ (w_1 \times w_2).nat = (w_1.nat \times w_2.nat) \ \text{MOD } 2^{wt} \end{array}$

 $/, \backslash : Word \times Word \not\rightarrow Word$ \Rightarrow $w_{2}.int \neq 0$ \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

 $\begin{array}{l} n.int \geq 0 \\ 2^{*i} \times hi'.nat + lo'.nat = ((2^{*i} \times hi.nat + lo.nat) \times 2^{*i}) \text{ MOD } 2^{2 \times *i} \end{array}$

SHIFTRIGHT

 $\begin{array}{l} hi', lo': Word \\ hi, lo: Word \\ n: Word \\ \hline \\ n.int \geq 0 \\ 2^{el} \times hi'.nat + lo'.nat = (2^{el} \times hi.nat + lo.nat) \text{ DIV } 2^{a} \end{array}$

NORMALISE

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

 $\begin{array}{l} n.int \geq 0\\ 2^{wl} \times hi'.nat + lo'.nat = (2^{wl} \times hi.nat + lo.nat) \times 2^{places'}\\ wl - 1 \in hi'.bitset \vee hi'.nat = 0 = lo'.nat \wedge places' = 2 \times wl\end{array}$

A.8 Arithmetic Procedures

LONGSUM

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

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

LONGDIFF

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

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

LONGPROD

 $\begin{array}{l} hi', lo': Word \\ x, y, carry: Word \\ \hline \\ \hline \\ 2^{v'} \times hi'.nat + lo'.nat = x.nat \times y.nat + carry.nat \\ \end{array}$

LONGDIV

 $\begin{array}{l} quot', rem': Word \\ hi, lo, y: Word \\ \hline \\ 2^{mi} \times hi.nat + lo.nat < 2^{mi} \times y.nat \\ 2^{mi} \times hi.nat + lo.nat = quot' \times y.nat + rem'.nat \\ O \leq rem' < y.nat \end{array}$

Index

Above 5 AddSub 31 Add 33 Aligned 31 Align 31 Arit_Signature 8 Ant 11 Below 6 Bool 42 Bounds 23 Compare_Bool_Error 19 Compare_Bool 19 Compare_Condition 18 Convert Integer 16 Convert 16 Denormalise 26 Denorm 4 Div_By_Zero 9 Divide 37 DoubleExtended 3 Double 3 Equal 18 Error_After 24 Error_Before 24 Error_Signature 8 Error_Spec 11 Error_Spec 13 Error_Spec 15 Error_Spec 8 Exact_Sert 14 Exc_Conv 16 Exc. Sqrt 14 External 22 FPS32 22

FP Arit 11 FP_Round27 FP Roundl 7 FP. Round 8 FP 4 Fields 4 Fin_Convert 15 Fin_Int 17 Fin_Rem 12 FiniteArit 30 Finite_Arit 9 Finite 4 Format 3 Greater Than 18 Inf_Add 10 Inf_Arit_Signature 10 Inf_Arit 11 Inf_Convert 15 Inf_Div 11 Inf_Mul 10 Inf_NaN_Int 17 Inf_Rem 13 Inf_Sort 14 Inf_Sub 10 Inf 4 Int_Conv_Round 16 Int_Signature 17 Integer_Round1 16 Integer_Round 16 Internal 22 Int 17 LONGDIFF 44 LONGDIV 44 LONGPROD 44 LONGSUM 44

LessThan 18	Value_Spec 9
Multiply 36	Word 41
NORMALISE 43 NaNF 4 NaN_Arit 10 NaN_Convert 15 NaN_Rem 12 NaN 4 No_Round_Errors 11 Normal 25 Norm 4	Zето 4
Packed 24 Pack 29	
Pos_Sqrt 14	
Rem_Signature 12 Rem_Zero 12 Rem 13 Rnd_Int_Round 17 Rnd_Int_Round 17 RoundToNearest 7 RoundToNegInf 6 RoundToZero 6 Round_Signature 5 Round 26 Round 7	
SHIFTLEFT 43 SHIFTRIGHT 43 Sign_Bit 13 Sign_Bit 14 Sign_Bit 9 SingleExtended 3 Single 3 Sqrt_Errors 15 Sqrt 15 Sub 33 Succ 23	
Unnormalised 22 Unordered 18 Unpacked 22 Unpack 25	