FPV

a Floating-Point Validation package

Release 1

USER'S GUIDE

Numerical Algorithms Group NP1201

Correspondence between the files you have have received and the programs described in the FPV installation note.

You have received the FPV package in one of five formats. These are:

A - Unlabelled fixed block tape. 9 track, phase encoded, 1600 bits per inch. Recorded at 80 bytes per record, 4000 bytes per block, ASCII.

B - Unlabelled fixed block tape. 9 track, phase encoded, 1600 bits per inch. Recorded at 80 bytes per record, 4000 bytes per block, EBCDIC.

C - ANSI standard labelled tape. 9 track, phase encoded, 1600 bits per inch.

D - Unix 'tar' format with all files in a tree under a single directory. 9 track, phase encoded, 1600 bits per inch.

E - IBM PC DOS 5.25 inch diskettes, single sided, double density.

All the programs and data files described in the installation note have been supplied to you in both Fortran-77 and ISO standard Pascal versions. Please go to the appropriate section below.

A - Unlabelled ASCII.

The files are written on the tape in the order specified in the installation note, first the seven Fortran versions, then the seven Pascal. It may be convenient for you to name them as in section C.

B - Unlabelled EBCDIC.

The files are written on the tape in the order specified in the installation note, first the seven Fortran versions, then the seven Pascal. It may be convenient for you to name them as in section C.

C - ANSI standard labelled.

The supplied tape contains fourteen files, which correspond to those described in the FPV Installation Note in the following way:

Installation Note description	Tape files
file 1: source-text of FPVGEN	FPVGEN.FOR, FPVGEN.PAS
file 2: source-text of FPVTGT	FPVTGT.FOR, FPVTGT.PAS
file 3: source-text of FPVPAR	FPVPAR.FOR, FPVPAR.PAS
file 4: sample driving file for FPVGEN	GENDRIVE.FOR, GENDRIVE.PAS
file 5: sample test-set file	TESTSET.FOR, TESTSET.PAS
file 6: sample driving file for FPVTGT	TGTDRIVE.FOR, TGTDRIVE.PAS
file 7: sample report file	REPORT.FOR, REPORT.PAS

D - Unix 'tar' format.

The supplied tape contains fourteen files, which are named as in section C.

E - PC DOS diskettes.

You have received four single sided discs. Discs 1 and 2 contain the Fortran version of FPV, discs 3 and 4 the Pascal version.

On the Fortran discs one file, FPVGEN.FOR, is too large to fit on one disc. It has therefore been split into two files, FPVGEN1.FOR, which takes up the whole of disc 1, and FPVGEN2.FOR, which consists of just two subroutines and is placed on disc 2. It will be more convenient if you merge these two files into one file called FPVGEN.FOR. All the other files, Fortran and Pascal, are named as in section C.

FPV Release 1 - User's Guide

Printed and produced by the Numerical Algorithms Group
1st Edition May 1986.

ISBN 1-85206-30-1

Numerical Algorithms Group Ltd
NAG Central Office
Mayfield House
256 Banbury Road
Oxford
United Kingdom OX2 7DE

Tel: National (0865) 511245
International +44 865 511245
Telex: 83354 NAG UK G

In North America:

Numerical Algorithms Group Inc
1101 31st Street, Suite 100
Downers Grove, IL 60515-1263
USA

Tel: National (312) 971 2337
International +312 971 2337
Telex: 704743 NUMALGGRP UD

In Australasia:

Siromath Sydney Pty
Level 3, St Martin's Tower
31 Market Street
Sydney NSW 2000
Australia

Tel: National (02) 29 5352
International +61 2 29 5352
Telex: AA 26 282

Contents

Acknowledgements

FPV was developed by N.A.G. Ltd in collaboration with Dr. B.A. Wichmann of the National Physical Laboratory, under a contract jointly funded by N.A.G. Ltd and the Department of Trade and Industry. FPV is based on ideas contained in a program FPTST, which was developed by Dr. N.L. Schryer of AT&T Bell Laboratories. We are grateful to AT&T Bell Laboratories for permission to use these ideas, and to Dr. Schryer for his advice and encouragement. We are also grateful for the comments of Professor W. Kahan.

References

Brown W.S. (1981). A simple but realistic model of floating-point computation. ACM Trans. Math. Software **7**, pp.445-480.

Cody W.J. and Waite W. (1980). Software Manual for the Elementary Functions. Prentice Hall, Englewood Cliffs.

Coonen J.T. (1984). A compact test suite for P754 arithmetic - Version 2.0. Chapter 10 of Ph.D. Thesis, University of California, Berkeley.

IEEE (1985). Standard for Binary Floating-point Arithmetic. ANSI/IEEE Std 754-1985.

Karpinski R. (1985). Paranoia: a floating-point benchmark. Byte **10**, No. 2, pp.223-235.

Schryer N.L. (1981). A test of a computer's floating-point unit. Computer Science Technical Report No. 89. AT&T Bell Laboratories, Murray Hill, N.J.

Schryer N.L. (1986). A case study in testing: floating-point arithmetic. To be submitted to Comm. ACM.

Sterbenz P.H. (1974). Floating-point Computation. Prentice Hall, Englewood Cliffs.

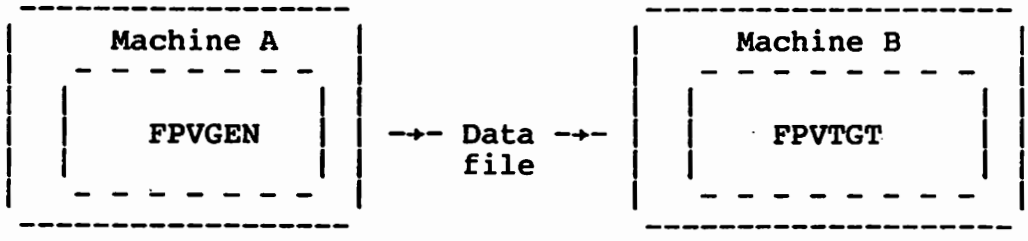
1. Introduction

1.1 Summary of Features of FPV

FPV is a software package for validating an implementation of floating-point arithmetic. It is primarily intended to check for design errors in floating-point arithmetic, but may also be used to check for intermittent errors (caused by a transient malfunction in the hardware).

By 'validation' we mean simply an experimental verification that floating-point arithmetic has been correctly implemented according to its specification. FPV must be supplied with the essential parameters of the specification - base, precision, exponent range, and rounding rule - and then attempts to verify that the arithmetic conforms to these parameters by probing for errors as best we know how. FPV does not attempt to judge the quality of the design of an implementation of floating-point arithmetic. Almost any implementation can satisfy the tests performed by FPV if the criteria for acceptance are suitably relaxed. The 'best' implementations satisfy the most stringent criteria.

On many systems, application of FPV need only involve running a single program, FPVGEN. This program can generate operands, perform the floating-point operations and check the results, all on the machine to be tested. However, in order to facilitate testing in as wide a variety of environments as possible, FPV allows the testing procedure to be split into two phases. In 'two-phase' mode, the program FPVGEN generates a file of test data; a second program FPVTGT, usually running on a different machine, reads the file and performs the tests.



The program FPVTGT is considerably shorter than FPVGEN, and is much easier to adapt to different environments (even if this involves translating it into a different language). The programs FPVGEN and FPVTGT are currently written in both standard Fortran 77 and ISO standard Pascal, level 1. They therefore require a suitable compiler to be available and they test the arithmetic as 'seen' through those languages. A few machine-specific modifications may be needed to make the programs completely robust. In order to test arithmetic on machines which do not have a Fortran or Pascal compiler, or to test arithmetic as seen through a different language (e.g. Basic, Ada), it is necessary to translate all or part of the program FPVTGT into a suitable language.

FPV allows arbitrary values for the base, precision and range of floating-point numbers (though currently the base must not exceed 16). FPV can test the following floating-point operators:

addition and subtraction	(x+y, x-y)
multiplication	(x*y)
division	(x/y)
negation	(-x)
absolute value	($|x|$)
square root	($\sqrt{x}$)
comparisons	($x=y$, $x\neq y$, $x<y$, $x>y$, $x\leq y$, $x\geq y$)

Square root is included because it is sometimes provided as a basic hardware instruction; exponentiation is not included because in most cases it involves calls to the exp and log functions. FPV does not test mixed-precision operations, nor does it test conversion between floating-point numbers and integers or decimal strings. FPV tests arithmetic on operands stored in memory. On some systems registers are provided with extended precision and range. In principle, it should be possible to modify FPV to test the full scope of register-arithmetic, but the details must depend on what language facilities are available to access such registers.

FPV can either test that the results are correct according to one of a choice of commonly used rounding rules; or, if the rounding rule is unknown or not one of those provided, it can test that the results lie within the narrow bounds defined in the model of floating-point arithmetic developed by W.S. Brown (the 'Brown model'). FPV can also be used to test whether the overflow and underflow flags are set correctly, though for this purpose machine-specific modifications must be made to the programs.

Because the number of combinations of floating-point operands on any realistic computer is enormous (e.g. of the order of 10^{18} or more), any testing must be extremely selective. The selection strategy used by FPV has been demonstrated to be effective in practice, but there is no guarantee that errors might not exist which cannot be detected by FPV. Moreover the stringency of the testing performed by FPV is under the control of the user. A user can select a subset of the tests of which FPV is capable; indeed he will normally wish to do so to ensure that the tests can be completed in a reasonable length of time, or sometimes to focus attention on a particular feature that is suspect. Effective use of FPV is the user's responsibility and requires a reasonable degree of understanding. The FPV User's Guide aims to explain the necessary background and to give suitable advice.

1.2 Advice to Readers

Sections 3 and 4 of this guide present the theoretical basis of FPV, and **must** be understood by anyone who wishes to undertake serious testing. Users who wish to gain quick

experience of using FPV may proceed to Sections 5 and 6, referring back to Sections 3 and 4 as necessary, but this approach is suitable only for initial familiarisation. Section 7 is relevant only if FPV is being used in two-phase mode; Section 8 gives additional guidance that can be referred to if needed; Section 9 gives detailed advice on modifications to FPV that may be necessary (the reasons for making them are explained in subsections 4.6 and 3.5); Section 10 gives advice on translating FPV into other languages.

2. Why Test Floating-Point Arithmetic?

The question may be asked, 'Why do we need to test computer arithmetic? Surely the computer manufacturers test their machines thoroughly; errors in arithmetic on production machines are very rare; if their effects are serious, they will quickly be noticed; if not (e.g. if they only affect the less significant digits), are they all that important?'

There are several answers to this question:-

(1) Unfortunately not all manufacturers do test their machines thoroughly - partly because it is a non-trivial task. The number of possible floating-point numbers that can be stored in a computer word is so large that to test every possible combination of two numbers under addition, subtraction, multiplication and division might require thousands or millions of years of processor time. Any test of the arithmetic therefore must be very selective, and if the wrong choices are made, errors may be missed. It is not enough to take a million pairs of real numbers at random, and check that results of operations on them are correct. Errors are likely to arise in rather special circumstances, for example near the ends of the range of valid floating-point numbers, and a random search for them is unlikely to be effective. Schryer (1981, 1986) developed a program FPTST for testing floating-point arithmetic which discovered errors in 14 out of the first 21 machines on which it was run.

(2) On many machines floating-point arithmetic is implemented in software. This is especially so on microcomputers, but also true on larger machines where double or quadruple precision arithmetic may be performed by software, although single precision is implemented in hardware. It seems to be a fact of life that errors occur more frequently in software implementations than in hardware, perhaps because errors are more easily corrected in software, or because the writers of the software do not know enough about the underlying hardware. Preliminary versions of FPV have revealed errors on 5 different machines, 4 of which were errors in software.

(3) Many new machines are produced by relatively small companies who may be unwilling or unable to invest heavily in rigorous testing of computer arithmetic.

(4) Manufacturers often provide incomplete or inaccurate documentation of their floating-point arithmetic, for example of the method of rounding, or of the thresholds for overflow and underflow. It is important to be able to diagnose the precise behaviour of the arithmetic on a computer, even though discrepancies from the expected behaviour may only occur in the least significant bits. (Such information is needed by NAG, for example, in

order to assign with confidence correct values for the machine-dependent constants in the X02 chapter of the NAG Library, upon which many other Library routines depend.)

(5) It is true that errors in arithmetic on production machines are rare - so rare that users naturally trust the arithmetic and tend to blame suspect results on their own (or other people's) programs. It is only when they are led to investigate further, investing large amounts of time, that the computer may be proved to be at fault. It is also true that such errors as do occur are often not gross errors, but constitute an occasional loss of accuracy. For example occasional double precision results may only be accurate to single precision; but such a phenomenon undermines the validity of using double precision computation as a check on the accuracy of single precision results. More generally, while an occasional inaccurate result may be compensated for by subsequent computation and pass unnoticed, it is also possible for its effect to be critical, e.g. it may prevent convergence to the expected accuracy.

(6) At the heart of the matter, however, lies the drive for correctness and reliability in numerical computing. Any success in proving the correctness of numerical algorithms is invalidated if the underlying arithmetic is incorrect. Before testing any complex piece of numerical software, it makes sense to check the fundamental components of the computing environment such as the floating-point arithmetic and the elementary functions. (For testing the latter, programs have been provided by Cody and Waite (1980).)

Two recent developments have highlighted the importance of validating floating-point arithmetic.

The first is the IEEE standard for binary floating-point arithmetic (IEEE, 1985). This is an excellent design and may well lead to a greater uniformity in the specification of floating-point arithmetic on different machines. However, a standard is incomplete unless there is some means of checking conformity with the standard. The IEEE standard is complex and provides among other things for: extended precision and range, different rounding modes, gradual underflow, and exception handling. To validate a full implementation of the standard requires a very elaborate test package; many existing implementations are in fact partial and here again manufacturers' information can be misleading, so it is important to be able to check which features of the standard have been implemented and which have not. FPV can test all the arithmetic operations specified in the standard except: remainder, round to integer, conversion between floating-point formats, and binary-decimal conversion. It can test all of the specified rounding modes. It cannot test operations on denormalised numbers, nor the full range of exception-handling

facilities. FPV would normally test arithmetic on numbers in either of the basic formats, assuming that these are the only formats in which numbers are stored in memory; however it should not be difficult to modify FPV, using non-standard language facilities or machine language, so that it can test arithmetic on numbers in an extended format. Coonen (1984) has developed a test suite specifically for the IEEE standard.

The second development is the programming language Ada whose definition includes a detailed specification of floating-point arithmetic. A complete validation of an Ada compiler should therefore include a validation of floating-point arithmetic (as 'seen' by an Ada program). An Ada version of the program FPVTGT meets this requirement.

3. A Model of Floating-Point Arithmetic

Floating-point arithmetic is available in a wide variety of computing environments. It may be implemented in hardware, firmware or software, and different implementations may co-exist on one machine (e.g. single precision and double precision, binary and decimal). In this Guide the terms 'arithmetic' or 'an arithmetic' are used for short to denote 'an implementation of floating-point arithmetic'.

In order to deal with the wide variety of arithmetics that have been implemented, FPV needs a generally applicable **model** of floating-point arithmetic. We have followed Schryer (1981) in using the model developed by W.S. Brown (1981) (sometimes referred to as 'the Brown model') as our starting point, but have made some variations and extensions. (Brown developed his model as a framework for portable numerical computing, rather than for testing, so our requirements are somewhat different.) The model provides:

- a simple characterisation of an implementation of floating-point arithmetic in terms of four parameters;
- a generally applicable criterion for the correctness of floating-point arithmetic operations.

The model is an idealisation and simplification of the actual behaviour of floating-point arithmetics, but it provides a sufficiently close description for the practical requirements of testing.

Schryer's program FPTST tests specifically whether or not arithmetic conforms to the Brown model with given values of the parameters. FPV is not so closely tied to the Brown model, but can:

- either test whether arithmetic is exactly correct according to one of a limited choice of rounding rules;
- or test whether arithmetic conforms to the criteria of the Brown model.

The latter is a less stringent test, but one that is more generally applicable.

When FPV is testing arithmetic according to a specific rounding rule, it simply requires a convenient means of describing which numbers are representable in the arithmetic. We discuss this aspect separately, before we consider the rules for the performance of basic arithmetic operations.

(For more extensive background reading on floating-point arithmetic we suggest the book by Sterbenz (1974).)

3.1 Representable Floating-Point Numbers

To describe (as closely as possible) the set of floating-point numbers which are representable in an arithmetic, FPV uses the same four integer parameters as are used by Brown. They are:

- the base, B;
- the precision, P;
- the minimum allowed exponent, EMIN;
- the maximum allowed exponent, EMAX.

The set of representable numbers defined by these parameters is assumed to consist of:

zero

and numbers of the form:

$\pm B^e f$

where:

- the integer exponent, e, satisfies EMIN $\leq$ e $\leq$ EMAX;
- the fraction, or mantissa, f, is a base-B fraction of P digits such that $1/B \leq f < 1$, i.e.

$$f = 0 . f_1 f_2 . . . f_P$$

with $1 \leq f_1 < B$, and $0 \leq f_i < B$ for $i > 1$. The f_i are the base-B digits of the fraction.

We shall refer to the values of B, e and f as the **model representation** of a representable floating-point number. The way in which numbers are represented in the machine may be different (for example, the exponent may be biassed, the point may be shifted, and so on), but that is irrelevant to FPV. FPV is only concerned with the values of the representable numbers, not with how they are stored.

In many arithmetics the set of representable numbers can be precisely described by suitable values of the parameters B, P, EMIN and EMAX: for example, in DEC VAX 11/780 single precision hardware B = 2, P = 24, EMIN = -127 and EMAX = 127. There are, however, some exceptions:

- machines which use a 2's-complement representation of negative numbers and may allow numbers which cannot be negated. Typically

  $+B^{EMIN-1}$ is representable, but not $-B^{EMIN-1}$

  $-B^{EMAX}$ is representable, but not $+B^{EMAX}$

- machines which allow gradually underflowed numbers between $\pm B^{EMIN-1}$ and zero, as are defined, for example,

in the IEEE standard (IEEE, 1985);

- machines which would require a non-integer value of P (the only such machine of which we have definite knowledge is the Telefunken TR440: it would require B = 16 and P = $9\frac{1}{2}$);

- machines which do not have a clear-cut set of representable floating-point numbers: some bit-patterns may be accepted as legitimate floating-point operands by one floating-point operator but not by another (for example CDC 7600 machines and Cray machines do not have a single underflow threshold that applies to all operators).

On such machines values for the parameters must be chosen which define the largest possible subset of the representable numbers. FPV will regard the subset so defined as its 'domain': it will test floating-point operations whose operands belong to the defined subset. In this case, then, there will be a small 'fringe' of possible operands which will not be tested by FPV. An alternative approach, which requires greater care, is to choose values of the parameters which define a larger set of values, including some non-representable numbers: the effects of including non-representable numbers (e.g. overflow or spurious invalid results) must then be discounted.

Henceforth we shall assume, for simplicity in the discussion, that the set of representable numbers is precisely defined by suitable values of the parameters. Values of B, P, EMIN and EMAX which are being used to define the set of representable numbers, will be referred to as **machine-parameters**, and the set of numbers so defined as **machine-numbers**.

The largest positive machine-number is:

$$\lambda = B^{EMAX} * (1 - B^{-P})$$

Any number whose magnitude is larger than this is said to **overflow** the range of the machine (or, for short, to overflow the machine). λ is the overflow threshold. The smallest positive machine-number is:

$$\sigma = B^{EMIN-1}$$

Any number whose magnitude lies between σ and zero is said to **underflow** the range of the machine (or to underflow the machine). σ is the underflow threshold.

In passing, we define here the term **ulp** (= 'unit in the last place'). Relative to a given non-zero machine-number, 1 ulp is the value of a digit 1 in the least significant digit-position of the fraction; it is the difference between the given number and the next largest machine-number (in magnitude). In terms of the model representation, relative

to a given number $\pm B^e f$:

$$1 \text{ ulp} = B^{e-p}$$

3.2 Rounding Rules for Arithmetic Operations

FPV provides a limited number of **rounding rules** according to which the correct computed result of an operation can be determined unambiguously. The available rules all have the following properties:

- if the exact (mathematical) result of an operation is a machine-number, then this must be the computed result;
- otherwise the exact result lies in an interval between two machine-numbers, and the computed result must be one of these two machine-numbers: the rounding rule determines which.

The available rules are:

- round **to nearest**, with $\frac{1}{2}$ ulp rounded to **either** of the nearest machine numbers (this rule allows either possibility for rounding $\frac{1}{2}$ ulp, in case neither of the following two rules applies);
- round **to nearest**, with $\frac{1}{2}$ ulp rounded **away from zero** (this is the conventional rule taught in school, which is used on many machines, e.g. the DEC VAX machines);
- round **to nearest**, with $\frac{1}{2}$ ulp rounded **to nearest even**, that is, to the adjacent machine-number whose least significant fraction-digit is even (this is the unbiassed default rule defined in the IEEE standard);
- round **toward zero** (often called truncation or chopping);
- round **toward minus-infinity**;
- round **toward plus-infinity**;

The last four rules are those specified in the IEEE standard (IEEE, 1985).

When testing according to a specific rounding rule, comparisons are required always to yield the correct result: they are not affected by the differences between the rules.

The rules must be qualified if there is a possibility of overflow or underflow: this is discussed in subsection 3.5.

All of the above rounding rules are consistent with the weaker criteria of the Brown model, which are described in the next two subsections.

3.3 Brown Model Bounds: Strongly Supported Operators

This subsection and the next may be omitted at first reading by users who are confident that the arithmetic being tested conforms to one of the rounding rules provided by FPV. If the arithmetic being tested does not use one of those rules, then FPV must check correctness according to the **rules of the Brown model;** these yield tight **bounds** on the computed result.

This subsection describes the rules for what Brown calls **strongly supported** operators. A less stringent set of rules for weakly supported operators is described in the next subsection.

Brown uses values of the four parameters B, P, EMIN and EMAX to define a set of **model-numbers.** The model-numbers may be a subset of the machine-numbers. We refer to the parameter values which define the set of model-numbers as **model-parameters.** We derive from them values for the model overflow threshold (sometimes called model-λ) and the model underflow threshold (sometimes called model-σ), just as in subsection 3.1.

Model-numbers must satisfy the following rules for the result of a basic arithmetic operation:

- if the exact result is also a model-number, then this must be the computed result;
- otherwise the exact result lies in an interval bounded by two model-numbers, and the computed result must then lie in the same interval (as shown in Fig. 3.1);

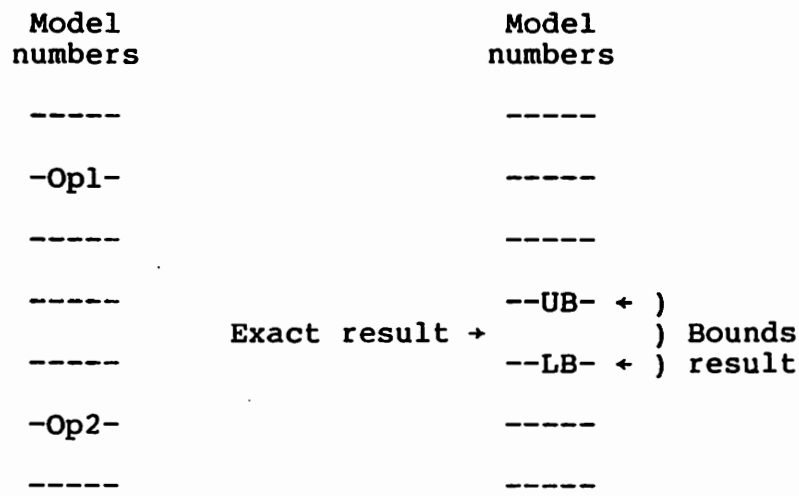


Fig. 3.1

- comparisons between any two model-numbers must always yield the correct result.

If all machine-numbers conform to the rules of the model,

then the rules imply that the computed result must be one of the machine-numbers on either side of the exact result (but may be either). Arithmetic with these properties is sometimes called 'faithful'. The rounding error is less than 1 ulp.

The rules still apply if the exact result underflows the model, i.e. lies between zero and ± model-σ: the computed result must lie in the same interval. It is assumed that underflow does not halt the program. However if the exact result overflows the model, the rules cease to apply. Overflow and underflow are discussed further in subsection 3.5.

In some arithmetics the complete system of machine-numbers conforms to the rules of the model with a suitable choice of parameters; for example single precision numbers on IBM 370 machines, with B = 16, P = 6, EMIN = -64, EMAX = 63. In other arithmetics it is only a large subset of the machine-numbers which conforms to the model, because of anomalous behaviour at the limits of precision and range. For example, on CDC Cyber 170 series machines, the single precision machine-numbers can be described by the parameters B = 2, P = 48, EMIN = -974, EMAX = 1070; however, to conform to the rules for arithmetic operations and comparisons, the set of model numbers must be restricted by setting P = 47 and EMIN = -929. (The reasons are that normalisation is performed after rounding in addition/subtraction; and comparisons are performed via subtraction which gives incorrect results if the difference underflows.) In Schryer's words, penalties must be imposed to make the model fit the arithmetic.

Thus for some arithmetics we need two sets of values of the parameters B, P, EMIN and EMAX:

- a set of **machine-parameters** to define (as closely as possible) the set of representable numbers;

- a set of **model-parameters** to define the largest possible subset of the representable numbers which conforms to the Brown model.

(In fact FPV requires B to have the same value in both sets of parameters.)

If the model parameters are not equal to the machine parameters, then there exist machine-numbers which are not model-numbers. They may be either **extra-precise** numbers (if the model-P is less than the machine-P); or **out-of-range** numbers, overflowing or underflowing, (if the model values of EMIN and EMAX lie inside the machine values).

The Brown model requires that arithmetic on extra-precise numbers must be consistent with arithmetic on model-numbers according to the following rule: replace each machine-number by the smallest model-interval in which it lies, perform the

operation on these intervals (in the usual sense of interval arithmetic) and widen the resulting interval to the smallest model-interval which contains it; this model-interval must contain the computed result (see Fig. 3.2).

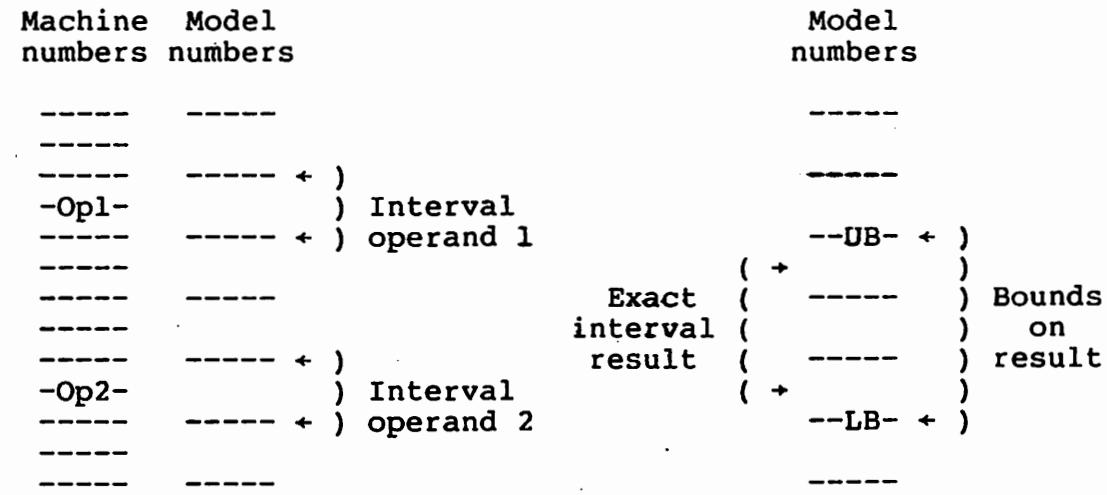


Fig. 3.2

Comparisons on extra-precise numbers x and y may yield the same result as the exact comparison of any two numbers $\hat{x}$ and $\hat{y}$ which lie in the smallest model-intervals containing x and y, but may not yield any other result. Hence if there are no model-numbers between the machine-numbers x and y, x<y, x=y, and x>y are all permissible results; if there is just one model-number between x and a larger machine-number y, x<y and x=y are permissible, but x>y is not.

FPV can test the correctness of arithmetic on extra-precise numbers according to these criteria. Since the criterion for correct comparisons is slack enough to permit anomalous combinations of results, FPV also tests whether the results of comparisons are consistent with one another (i.e. it reports an inconsistent comparison if, say, x<y and x>y both yield the result 'true').

Arithmetic on out-of-range numbers is discussed in subsection 3.5.

3.4 Brown Model Bounds: Weakly Supported Operators

Any arithmetic operator which conforms to the rules of subsection 3.3, is said by Brown to be **strongly supported.**

A less stringent set of rules is allowed by Brown for so-called **weakly supported** operators. For these, the model-interval within which the result must lie, is extended to the next model-number on either side, as illustrated in Fig. 3.3:

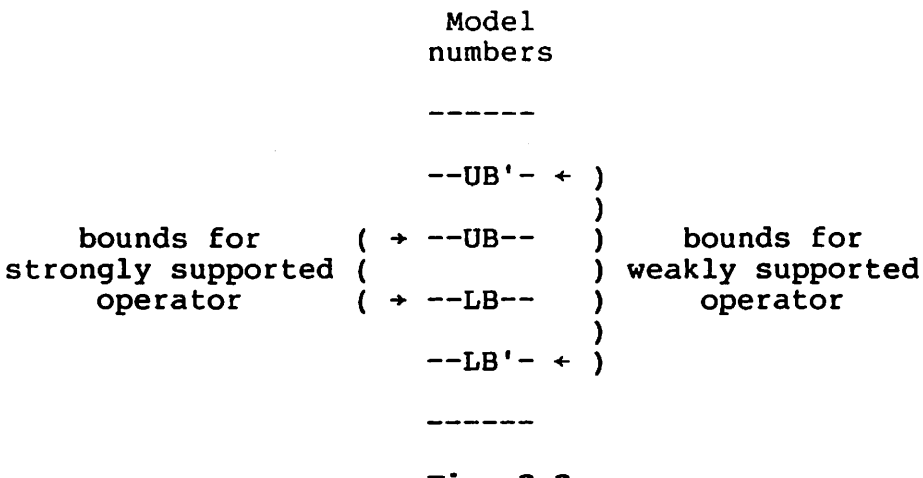


Fig. 3.3

(However if either UB or LB is zero, so is UB' or LB' respectively: the interval is not extended beyond zero.) Even if UB and LB are equal (i.e. an exact result is expected from a strongly supported operator), UB' and LB' remain defined as above (i.e. the result of a weakly supported operator is never required to be exact).

The concept of a weakly supported operator is useful, for example, when modelling an arithmetic in which division is implemented as a composite operator (reciprocation followed by multiplication), because then the result is subject to more than one rounding error. It may also be needed when double precision arithmetic is implemented in software, using single precision floating-point hardware.

The criterion for checking the comparisons is the same in either case since Brown does not define weakly supported comparisons.

FPV allows the criterion of either strong support or weak support according to the Brown model, to be applied independently to each of the basic arithmetic operators. Normally one should attempt to find reasonable values of the model parameters according to which most operators are strongly supported, but possibly one or two (most likely division or square-root) are only weakly supported; the two sets of operators would have to be tested in separate runs of FPV.

3.5 Overflow and Underflow

Implementations of floating-point arithmetic display a variety of behaviour with regard to overflow and underflow. The overflow threshold may differ between different floating-point operators, and may even differ depending on the values of the operands. When overflow occurs, usually an exception is signalled, but it may not be; and usually the program halts, but it may continue with or without some floating-point value being set as the result of the operation. When underflow occurs, it is more usual for an

exception not to be signalled, but it may be; and usually the program sets the result to zero or to some very small value and continues, but it may halt.

To cope with this variety, FPV offers three different **modes** of testing with regard to overflow, and, independently, three modes with regard to underflow. We describe the details first for the case where the arithmetic is being checked according to the rules of the Brown model, having in mind the possibility that the model-EMAX and the model-EMIN may differ from the machine-EMAX and the machine-EMIN: thus there may exist machine-numbers which are out-of-range in terms of the model-parameters. We distinguish between the machine-λ (the largest positive machine-number defined using the machine-EMAX as in subsection 3.1) and the model-λ (defined similarly, but using the model-EMAX); likewise, between the machine-σ and the model-σ. Although we describe overflow and underflow in parallel, it is **not** necessary to select the same mode for testing with regard to both overflow and underflow.

Mode 1:

This is the normal mode of testing. FPV aims to test the arithmetic within the safe bounds set by the model-parameters.

Overflow: If either of the bounds on the result of an operation would overflow the model, that operation is skipped. Thus if the model-parameters have been set correctly, overflow should not occur.

Underflow: Operations whose results are expected to underflow the model, are still performed, and the results checked against the appropriate model bounds involving zero and $\pm$ model-σ. It is assumed that underflow does not interrupt the flow of the program.

Mode 2:

For this mode FPV must be modified to trap any overflow or underflow exceptions that may be signalled (see subsection 9.2). FPV aims to test the arithmetic up to the limits at which machine overflow or machine underflow occurs, and to check that overflow or underflow exceptions are signalled correctly.

We define an **extension of the Brown model** in which bounds on the result are defined exactly as in subsections 3.3 or 3.4, but ignoring the limits imposed by the model-EMAX and the model-EMIN. Either the computed results must satisfy the bounds or an exception must be signalled, but an exception may not be signalled if both bounds are within the range of the model. In practice the bounds can only be applied if they lie within the range of the machine, and an exception must be signalled if both bounds lie outside the range of the machine.

Overflow: All operations are performed regardless of whether or not the result is expected to overflow the model.

- if both bounds overflow the range of the machine, then overflow must be signalled;
- if both bounds lie within the range of the model, then overflow must not be signalled;
- if at least one bound lies within the range of the machine, and at least one bound overflows the range of the model, then overflow may or may not be signalled;
- in every case if overflow is not signalled, the computed result must satisfy any bounds which lie within the range of the machine.

Underflow: Mode 2 for underflow is exactly analogous to Mode 2 for overflow, with 'underflow' replacing 'overflow' everywhere in the definition. This is only applicable if underflow does signal an exception.

Mode 3:

This mode is provided for testing conformity with a rigorous implementation of the Brown model in which an exception must be signalled whenever the limits of the model are exceeded. This might be required when testing an implementation of Ada, but otherwise is unlikely to be useful. The rules are the same as for Mode 2, except that 'the range of the model' replaces 'the range of the machine' everywhere in the definition. If the model-EMAX is equal to the machine-EMAX, Mode 3 for overflow is identical to Mode 2; likewise for underflow.

Note that the rules for Mode 2 allow for a 'grey' area in which overflow may or may not be signalled, thus tolerating arithmetics in which there is no single overflow threshold for all operations. Also the rules assume that the set of representable numbers is precisely described by the machine-parameters; if this is not so (see subsection 3.1), the reports of 'invalid' results must be interpreted with care.

Now we describe the effects of the different modes when arithmetic is being tested according to a specific rounding rule.

Here again, in order to tolerate arithmetics which do not have a single overflow or underflow threshold, FPV allows for a model-EMAX and model-EMIN different from the machine-EMAX and machine-EMIN. The model-EMAX and model-EMIN define safe limits within which overflow and underflow are not expected to occur.

A variation to be accommodated is that overflow and

underflow may be detected either before or after rounding. FPV therefore defines upper and lower bounds on the result rather than an exact value in the following very special cases:

- if the exact result strictly exceeds the model-λ by less than 1 ulp, then the lower bound is set to the model-λ and the upper bound to the next larger machine-number;

- if the exact result strictly exceeds the machine-λ by less than 1 ulp, then the lower bound is set to the machine-λ and the upper bound overflows.

Analogous bounds are set if the exact result is slightly less than the model-σ or slightly less than the machine-σ; and also of course for the corresponding negative numbers.

With these preliminaries, the rules for modes 1, 2 and 3 carry over unchanged. Note that in Mode 1 for underflow, if the exact result lies between zero and $\pm$ model-σ, then the computed result is simply required to lie within the same bounds. Thus gradually underflowed results (as defined, for example, in the IEEE standard) cause no difficulty, although they cannot be checked for precise accuracy.

4. Testing Strategy

In order to test arithmetic effectively without using exorbitant amounts of computer time, it is essential to select operands which are particularly likely to reveal errors. FPV follows the selection strategy of Schryer (1981):

- first, the pattern of digits in the mantissae must conform to one of a limited number of types;
- second, a subset of mantissae with these digit patterns can be selected;
- third, independent of the selection of the mantissae, a subset of exponent values can be selected.

The rationale behind Schryer's strategy is that - especially in an implementation which is almost correct - errors are most likely to occur as **edge-effects**, at or near some discontinuity or boundary in the values of the operands or some part of them.

Details of each aspect of operand specification are described in the next four subsections, followed by a summary of the internal working of FPV and a discussion of its reliability as a validation tool.

4.1 Mantissa Patterns

Schryer chose to use as operands for testing only those numbers whose mantissae f conformed to one of five types of digit patterns. (If B = 2, these reduce to three.) In fact in his experience of using FPTST, two types proved sufficient to detect all the errors that he has discovered (or learnt of). FPV uses them also as basic mantissa patterns. They are (Z denotes (B-1)):

(1) .100...00100...000 (Schryer's type 1)

(2) .ZZZ...ZZZ00...000 (Schryer's type 4)
i^{th} digit

Pattern 1 takes the values:

$1/B$ (when $i = 1$), and $1/B + 1/B^i$ for $2 \leq i \leq P$

Pattern 2 takes the values:

$1 - 1/B^i$ for $1 \leq i \leq P$

Thus the values are clustered near the limits of the range $[1/B,1)$ of values of f, in accordance with the strategy of looking for edge-effects.

FPV derives further mantissa patterns from these by adding

or subtracting $1/B^P$, giving the types

(3) .100...00100...001

(4) .100...000ZZ...ZZZ

(5) .ZZZ...ZZZ00...001

(6) .ZZZ...ZZYZZ...ZZZ
i^th digit

(here Y = B-2). Thus each individual mantissa of patterns 1 or 2 can be extended to a cluster of 3 adjacent mantissa values. In our experience types 3, 4, 5 and 6 have occasionally revealed properties of the arithmetic that were not shown up when using types 1 and 2 alone.

On machines with B > 2, again following an idea of Schryer, FPV allows operands of similar pattern but with Z(= B-1) replaced by 1, 1 replaced by Z, and Y(= B-2) replaced by 2, i.e.

(7) .Z00...00Z00...000 (Schryer's type 5)

(8) .111...11100...000 (Schryer's type 2)

(9) .Z00...00Z00...00Z

(10) .Z00...00011...111

(11) .111...11100...00Z

(12) .111...11211...111
i^th digit

It must be admitted that there is no clear rationale for these additional patterns. Indeed while on the one hand there is no great difficulty in adding additional operand patterns to FPV, there is also no clear guidance as to which patterns to add, and as yet no proven need to do so. Of course an error which is revealed by testing operands of the above patterns, will very likely also occur with many other patterns of operand: FPV is designed to **detect** errors but not to discover the complete range of situations in which they occur (see further in Section 8).

Finally, in order to test zero operands, we define type

(0) .000.....000 (Schryer's type 3)

4.2 Selection of Mantissae

Even with the limited choice of mantissa patterns available in FPV, further selectivity is desirable especially for short initial runs. This is achieved by selecting specific

values of i (i.e. the position of the i^{th} digit). We refer to i as the 'mantissa index'. Schryer's recommendation is to concentrate on the ends of the range and also at intermediate points that might coincide with byte- or word-boundaries. Hence a useful initial set of values for i might be:

1, P/2, P

and this can, and should, be extended by adding neighbouring values, e.g.

1, 2, (P/2-1), P/2, (P/2+1), (P-1), P

(The input to FPV makes it easy to specify such clusters of values, see Section 6). The clusters can be enlarged, and new clusters added, centred for example around P/4 and 3P/4.

Note: for some individual mantissa types FPVGEN in fact ignores certain mantissa index values near the limits of the range 1 to P. This is to avoid unnecessary duplication: for such values of the index different types of pattern may yield the same mantissa. The range of values used for each mantissa type is as follows (but normally users need not bother about the details):

Type	B > 2	B = 2
1	1..P	1..P
2	1..P	3..P
3	2..P	2..P-2
4	2..P-1	2..P-2
5	1..P-1	3..P-2
6	1..P	3..P-2
7	3..P	
8	3..P	
9	2..P	
10	2..P-1	
11	2..P-1	
12	3..P	

4.3 Selection of Exponent Values

For selecting exponent values, a similar approach is recommended, namely to concentrate on the ends of the range of values and at a few critical values in between. Thus an initial set of values might be clustered around:

EMIN, 0, EMAX

to which should be added values which differ from the above by ±P or ±P/2, since special cases arise in addition and subtraction when exponents differ by P, and sometimes also when they differ by P/2 (e.g. when double precision arithmetic is implemented in software).

4.4 Selection of Sign Combinations

After the mantissa and exponent of each operand has been selected, only the signs remain to be specified. Given a pair of operands (assumed for the moment to be both positive), FPV can with little extra work test a particular binary operation on any of the sign combinations

++ , +- , -+ , --

For example, given two operands Op1 and Op2, along with a lower bound, LB, and an upper bound, UB, on their product Op1*Op2, we know that the product (-Op1)*(-Op2) should also be bounded below by LB and above by UB. Similarly, (-Op1)*Op2 and Op1*(-Op2) both should be bounded below by -UB and above by -LB.

The situation for addition and subtraction is slightly different. Given Op1 and Op2, and knowing that

$$Op1 + Op2 \in [LB,UB]$$

we also know that

$$(-Op1) + (-Op2) \in [-UB,-LB]$$

$$Op1 - (-Op2) \in [LB,UB]$$

$$(-Op1) - Op2 \in [-UB,-LB]$$

Thus LB and UB can be used to bound the results of two addition and two subtraction operations. This interaction between addition and subtraction explains why FPV does not allow them to be tested independently.

Note however that the above rules do not apply when testing the rounding rules 'round to plus-infinity' or 'round to minus-infinity'. For these rules, only one combination of signs can be tested at a time. Otherwise it is usually worthwhile to test all four combinations because little extra work is involved.

Square root is tested only on positive operands.

4.5 Summary of Internal Working of FPV

We now summarise the internal working of FPV.

Pairs of operands are selected as described in the previous subsections. For both operands, each specified mantissa value is used in combination with each specified exponent value; and each specified value of operand 1 is used in combination with each specified value of operand 2. Each operand is stored as a data structure consisting of sign, exponent and mantissa; the mantissa itself is held as an array of integers.

Each operator to be tested is performed on each pair of operands by a set of subroutines which simulate interval floating-point arithmetic with the specified base, precision and range; these subroutines work entirely with integer arithmetic and determine either the upper and lower bounds on the correct result according to the Brown model, or the exact expected result according to a specified rounding rule. (In the latter case we may simply regard the upper and lower bounds as being equal.) The bounds are held in the same form of data structure as the operands. All the above is performed by the program FPVGEN.

The actual testing of each operation is performed either by FPVGEN or by FPVTGT, but in essentially the same way by both programs. The only difference is that, if FPVTGT is being used, the operands and bounds are written to a file by FPVGEN and read back by FPVTGT; the representation of the operands and bounds on the file is such that there is no risk of conversion error.

In either FPVGEN or FPVTGT the operands are converted to floating-point numbers in the machine's internal representation; the specified operations are performed; the computed results are converted back into the same data structure as the bounds; and the results are then checked against the bounds by subroutines that simulate floating-point comparisons. Any results that violate the bounds are reported.

4.6 Reliability and Robustness of FPV

Can FPV fail to detect errors in the arithmetic? Can FPV report false 'errors'?

We emphasise that the answer to the first question is certainly 'yes' - in principle - simply because FPV, even when running its most exhaustive set of tests, only tests a small sample of all possible pairs of operands. However to test that small sample may take months or years of computer time, so selectivity is unavoidable. In practice, the grounds for confidence in the selection strategy used by FPV (following Schryer) are very strong: we are not aware of any design errors in completed implementations of floating-point arithmetic which would **not** be detected by FPV, even with a very restricted selection of mantissa and exponent values; on the other hand Schryer's program FPTST, with a more limited choice of operands than FPV, **has** detected several errors that had not been detected by other testing procedures. (FPV will of course detect all the errors reported by FPTST).

In practical runs lasting only a few hours or minutes, the effectiveness of FPV is dependent in part on the users' understanding of the testing strategy. It is the user's responsibility to specify a reasonable sample of mantissa types, mantissa index values and exponent values as suggested in subsections 4.1 to 4.3 above, and to include

all operators and sign combinations that are required to be tested.

It is also the users' responsibility to specify sufficiently rigorous criteria for checking the correctness of the arithmetic.

However it is still reasonable to ask whether FPV is guaranteed to detect any error that occurs within the terms of the specified test-set, or whether it may report false 'errors', and indeed **there is one aspect of FPV which may require care.**

Ideally FPV should not use floating-point operations for any purpose other than actually to compute the results which are to be checked. However it does not seem possible to write a portable program which satisfies this requirement. In the supplied text of FPV, machine-independent code is provided for converting a floating-point value from its model-representation in an integer array to the machine's internal floating-point representation; and also for converting from the internal floating-point representation back to an integer array. This code is portable **only if** certain floating-point operations are performed exactly, specifically: multiplication by powers of the base; negation; and certain additions (which should involve no carries, shifts or rounding); certain comparisons must also be performed exactly. Many arithmetics meet these requirements, but some do not, either because of an inferior specification or because of errors in their implementation. FPV endeavours to check that the conversions in question are being performed correctly before embarking on the main tests, but the checks themselves involve some use of floating-point operations, so it is conceivable that anomalies in the arithmetic could mask errors in the conversion.

To make FPV completely reliable and robust the code for performing the conversions must be rewritten using machine-specific bit-manipulation operations or other non-standard facilities which certainly do not involve floating-point arithmetic. Advice on how to do this is given in subsection 9.1.

Of course, FPV assumes that integer arithmetic is performed correctly in FPVGEN. If there are any errors in the integer arithmetic, it is hard to conceive how errors in the floating-point arithmetic might go undetected; instead it is very likely that FPV will report spurious errors or exhibit other kinds of weird behaviour.

In the last resort, any report of an invalid result from FPV can easily be checked by hand, and the individual operation re-tested independently of FPV as suggested in Section 8.

5. Approach to Testing

5.1 All-in-One or Two-phase Testing?

In **all-in-one mode** the program FPVGEN generates and performs a set of tests, all on one machine, in a single run. In **two-phase mode** FPVGEN generates a set of tests but, instead of performing them, writes details of the operands and expected results to a file; a second, simpler program **FPVTGT** reads the file and performs the tests. Normally in two-phase mode FPVGEN is run on a comparatively powerful machine and FPVTGT is run on a different machine (the 'target' machine). The file or files of data may be transferred to the target machine by communication link or magnetic media.

You are recommended to use FPV in all-in-one mode if possible. This requires a suitable compiler (Fortran or Pascal) to be available on the machine to be tested, but, given this, there is usually little reason not to use all-in-one mode. The program FPVGEN is not particularly large (for precise figures see the Installation Note) and will fit into the memory of most modern computers. FPVGEN does take longer to perform a given set of tests than FPVTGT, but on most machines this is unlikely to be inconvenient: initial trial runs should be short anyway, and more extensive sets of tests can be set up to run unattended or as background jobs.

In two-phase mode, transferring files of data (that might range in size from 100 kilobytes to several megabytes) will often be inconvenient or at least time-consuming. However, in some circumstances two-phase mode is the only way to use FPV. Such circumstances are:

- when there is no Fortran or Pascal compiler available on the target machine;
- when there is a particular need to test the arithmetic accessible via some language other than Fortran or Pascal (e.g. Basic or Ada).

It will then be necessary to translate FPVTGT (or parts of it) into some suitable language, possibly even into machine language. The essential features of FPVTGT have been kept as simple as possible, but in any case you are encouraged to consult NAG before attempting a translation: it may be possible to provide one for you - or at least the basis of one. See also Section 10.

5.2 Trial Runs

Before you attempt to run FPV you should find out as much as possible about the floating-point arithmetic to be tested, normally from the manufacturer's or implementor's documentation. In particular you will need to determine suitable values of the machine-parameters B, P, EMIN and EMAX, described in subsection 3.1. Ideally also you should

be able to find out details of rounding, but in any case you are advised initially to test simply for conformity with the Brown model.

Also included on the tape with FPV is a short program FPVPAR which attempts to determine the machine-parameters automatically, but is not foolproof. The values which it gives, should always be checked to see if they are sensible. If there are errors in the arithmetic, the values may be quite wrong. (FPVPAR uses ideas from programs by Cody and Waite (1980) and Kahan (Karpinski, 1985).)

Having decided on values for the machine-parameters, try some initial runs with **small** samples of operands. Detailed instructions on how to drive FPV are given in Sections 6 and 7. Interactive running is recommended for initial runs in all-in-one mode.

The aims of trial runs should be

- to ensure that floating-point values are being correctly converted from one representation to another, as discussed in subsection 4.6;
- to determine the best set of values for the machine and model parameters, and the appropriate rounding rule;
- to gauge how much computer time is required to test a sample of a given size.

You are strongly advised to **keep your initial samples of operands small.** Otherwise you may be swamped by a deluge of invalid results which may be hard or at least tiresome to analyse. The following points should help:

- throughout all your initial runs use **only operand types 1 and 2:** almost all known errors and anomalies could be detected by these types alone;
- test each of your initial samples **first** on **positive operands only,** and **then,** if no invalid results have so far been reported, test all possible sign combinations: many errors and anomalies (though certainly not all) are independent of the signs of the operands;
- look **first** for invalid results which depend primarily on **mantissa values only:** exponent values can be restricted at first to 0 and 1, while mantissa index values can be clustered round 1, P/2 and P; if no invalid results are found, enlarge the samples by letting the mantissa index take all values from 1 to P; and then add the exponent values 2, P, P+1;
- **then** look for invalid results which depend primarily on **exponent values only:** mantissa index values can be restricted to 1 and 2 (using operand types 1 and 2), while exponent values should be clustered round EMIN, 0

and EMAX; if no invalid results are found, add the mantissa index values P-1 and P, and the exponent values EMIN + P and EMAX - P; then some mantissa index values clustered around P/2, and the exponent values EMIN + P/2 and EMAX - P/2;

- if invalid results are detected with one particular operator, **test that operator separately.**

Invalid results may occur either because the arithmetic does not behave in accordance with its specification, or because you have assigned unsuitable values for some of the parameters or the rounding rule. Section 8 gives further advice on analysing invalid results that are reported by FPV.

The values given for the parameters and rounding rule should be regarded as a **hypothesis** about the performance of the arithmetic which FPV tests by trying to find results which violate the hypothesis (invalid results). The larger the samples which FPV uses without finding any invalid results, the greater our confidence that the arithmetic conforms to the given parameter values.

It remains the user's responsibility to establish whether or not the arithmetic also conforms to a stronger hypothesis, such as a larger value for the model-precision, or an exact rounding rule.

5.3 Production Runs

Once you are reasonably confident that you have determined the most suitable values for the parameters and rounding rule, you should consider much more extensive tests, using: all possible operand types; a large number of mantissa index values, or even all possible values; and a larger selection of exponent values. Such tests can be regarded as 'production' runs.

We envisage two categories of production runs. In both cases it is likely to be more convenient to drive FPV with a data file.

The first category is exhaustive testing for design faults, using the largest test-set which FPV can generate within the constraints of the amount of computer-time and real-time available. To test an arithmetic with the same parameters as IEEE standard single precision format, for example, FPV can generate about 1,000,000,000 different pairs of operands (not counting different sign combinations), and for most other arithmetics many more pairs can be generated. Therefore in most circumstances some subset of operand pairs must still be selected. The least important aspect to test comprehensively is the complete range of exponent values. Another way to cut down the size of the samples without diminishing their effectiveness too much is to use smaller sample sizes for operand 2 than for operand 1.

The second category of production run involves using FPV regularly to test for intermittent faults due to temporary malfunctions in the hardware. A possible procedure is to run a fairly short set of tests once a day, and a much longer set once a week or once a month.

6. How to Run FPVGEN

6.1 Files Used by FPVGEN

FPVGEN uses one input file and up to three output files.

The input file, referred to as the **driving file** (unit 5 in Fortran, INPUT in Pascal), is used for specifying the details of the tests to be performed. It may be a data file, or it may represent input from a terminal when FPVGEN is being driven interactively.

One output file (unit 6 in Fortran, OUTPUT in Pascal) is referred to as the **standard output file:** it is used either for issuing prompts when FPVGEN is being driven interactively, or for recording details of the tests when it is being driven from a data file.

A second output file is referred to as the **report file:** it is used for reporting any invalid results (or optionally all results). In Fortran it may be the same as the standard output file, or else its name must be specified by the user. In Pascal it may be the standard output file or else its name must be ERROUT (or a file logically equivalent to ERROUT).

The third output file is only used when FPVGEN is being used in two-phase mode. It is referred to as the **test-set file** and holds the test data for subsequent input to FPVTGT. In Fortran its name must be specified by the user. In Pascal its name must be BOUNDS (or a file logically equivalent to BOUNDS).

File-names specified by the user may be up to 32 characters long (subject to the limits imposed by the host system).

6.2 Driving FPVGEN Interactively

The user is presented with a series of questions, the answers to which specify the basic parameters of the machine being tested, and the set of tests to be performed. Many of the questions have default answers supplied in square brackets [ ]: pressing the RETURN key gives the default reply. During the interactive dialogue, FPVGEN makes some checks on the data being entered, and may display warning messages, or even halt execution, if it thinks that incorrect data has been entered.

We now present the output from a sample run of FPVGEN testing a **small** set of opérands, suitable for an initial trial run as suggested in subsection 5.2, along with annotation discussing the effects of answering the questions in different ways. User replies to prompts from FPVGEN are shown in *ITALICS*. Replies to yes/no questions should be *Y* or *N*. Lower-case replies are also accepted.

```
Are you running interactively (Y / N) ?    [Y]
Y
```

If the reply is *Y*, then prompts for further input are written to the screen (or default output device). If the reply is *N*, then FPVGEN assumes that data is being read from a file and so will not issue prompts (see subsection 6.4).

```
Input a comment line   [ ]
Test
```

The input text is reproduced at the head of the output file and may be used to identify it.

```
Input name of test-file to be generated   [NONE]
NONE
```

Fortran Version:
For an all-in-one test the reply **must** be *NONE*. If some other file name is given, FPVGEN will generate a file of test data for subsequent input to FPVTGT in a two-phase test.

Pascal Version:
In the Pascal version of FPVGEN, a different question is asked - "Generate test file BOUNDS (G) or perform all-in-one test (A) ? [A]". The user is given no option on what the test-set file will be called, and should reply with *G* or *A*.

```
Input base (B)
2
Input machine precision (machine P)
24
Input machine EMIN
-127
Input machine EMAX
127
```

These are the four basic **machine-parameters** described in subsection 3.1. For assistance in determining the correct values, refer to the provisional values given for some arithmetics in the appendix to the Installation Note; if the required values are not given there, try running the program FPVPAR mentioned in subsection 5.2. If an all-in-one test has been requested, FPVGEN tests whether floating-point operands with the specified precision and exponent range can be reliably generated; if not, the program issues an error message and halts, or an overflow or underflow exception may occur: this means either that the machine-parameters are wrong, or that the program must be modified as described in subsection 9.1.

```
Input model precision (model P)
24
Input model EMIN
-127
```

```
Input model EMAX
127
```

These are the **model-parameters** described in subsection 3.3. For an initial run they will usually be the same as the machine-parameters. If it is found that the machine does not conform to the model with those parameters, then the model-parameters may need to be adjusted in some way on subsequent runs. If a specific rounding rule is to be tested (see next question), then the model-P **must** be the same as the machine-P.

```
Input rounding rule...    [1]
 0  =  BROWN MODEL, weakly supported
 1  =  BROWN MODEL, strongly supported
 2  =  ROUND TO NEAREST (0.5 ulp rounded away from zero)
 3  =  ROUND TOWARD ZERO (i.e. truncation)
 4  =  ROUND TO NEAREST (0.5 ulp rounded to nearest even)
 5  =  ROUND TOWARD - INFINITY
 6  =  ROUND TOWARD + INFINITY
 7  =  ROUND TO NEAREST (0.5 ulp rounded either way)
1
```

If in any doubt about the rounding rule in the arithmetic being tested, specify rule 1: this tests for conformity with the Brown model as described in subsection 3.3. Rule 0 is intended to be used for operations (most likely division or square root) which are implemented as composite operations; the rule is described in subsection 3.4.

```
Test Add/Subtract  ?  [Y]
Y
Test Multiplication?  [Y]
Y
Test Division      ?  [Y]
Y
Test Square Root   ?  [N]
N
Test Unary Minus   ?  [Y]
Y
Test Absolute Value?  [Y]
Y
Test Comparisons   ?  [Y]
Y
```

These questions allow individual operations to be selected for testing. Square root is not usually one of the basic floating-point operations, so is not selected by default. The unary operations (square root, unary minus and absolute value) are tested only on operand 1, and only once for each value of operand 1.

```
How many different mantissa types do you want to test?  [2]
2
Input the 2 type numbers
1 2
```

The mantissa types and their numbers are described

in subsection 4.1. Types 1 and 2 are recommended for initial runs.

```
Input no. of basic mantissa index values for operand 1
3
Input array of basic mantissa index values
1 12 24
Input array of variance parameters
1 1 1
Operand 1 mantissa index values tested will be :-
     1
     2
    11
    12
    13
    23
    24
Are these satisfactory?   [Y]
Y
```

These questions allow the user to specify, for the first operand, the values to be used for the **mantissa index** i as defined in subsection 4.2. Rather than asking the user to specify a simple array of values of i, FPVGEN asks first for an array of 'basic' values, and then for each basic value a 'variance parameter' which defines a cluster of values centred on the basic value. If the basic value is m and the variance parameter is v, then the values in the cluster are

m-v, m-v+1, ... , m-1, m, m+1, ... , m+v-1, m+v

If the variance parameter is 0, the cluster consists of just the basic value m. The basic values must lie in the range 1 to P (i.e. the **machine** parameter P), and the variance parameters must not be negative. At most 20 basic values may be input. If any of the values between m-v and m+v fall outside the range 1 to P, they are simply ignored. Duplicate values in overlapping clusters are not allowed. FPVGEN displays the array of mantissa index values that will actually be used and asks for confirmation before proceeding. If the reply to the last question is ***N***, the set of questions will be repeated.

```
Input no. of basic mantissa index values for operand 2
3
Input array of basic mantissa index values
1 12 24
Input array of variance parameters
0 0 0
Operand 2 mantissa index values tested will be :-
     1
    12
    24
Are these satisfactory?   [Y]
```

```
Y
```

This is a similar set of questions about the mantissa index values of operand 2. The array of values for operand 2 need not be the same as that for operand 1, and it could reasonably be much smaller.

```
Input no. of basic exponent values for operand 1
2
Input array of basic exponent values
0 1
Input array of variance parameters
0 0
Operand 1 exponent values tested will be :-
     0
     1
Are these satisfactory?   [Y]
Y
```

These questions allow the user to specify the exponent values for operand 1 in the same way as the mantissa index values. The basic exponent values must lie in the range EMIN to EMAX (machine parameters). At most 20 basic values may be input.

```
Input no. of basic exponent values for operand 2
1
Input array of basic exponent values
0
Input array of variance parameters
0
Operand 2 exponent values tested will be :-
     0
Are these satisfactory?   [Y]
Y
```

This is a similar set of questions about the exponent values for operand 2. Again the array of values for operand 2 could reasonably be smaller than that for operand 1.

```
Input sign of operand 1 ( + or - )    [+]
+
Input sign of operand 2 ( + or - )    [+]
+
```

These questions ask for the signs to be given to the generated operands. Usually it will be convenient to generate operands with + signs, and to use the following set of questions to specify the testing of other combinations of signs at little extra cost. However when using rounding rule 5 or 6, or when investigating an error, it may be necessary to specify either or both of the operands to be generated with a - sign.

If FPVGEN is generating a test-set file, then no further details need be specified to FPVGEN. In this case FPVGEN skips to the final question 'PROCEED?'. The questions which have been skipped

are asked by FPVTGT, as described in subsection 7.2.

```
Which sign combinations of operands are to be tested?
  Plus o Plus ?   [Y]
Y
  Plus o Minus ?  [N]
N
  Minus o Plus ?  [N]
N
  Minus o Minus ? [N]
N
```

These questions allow the user to specify that various combinations of the signs of the operands are to be tested, at little extra cost, as described in subsection 4.4.

```
Which mode for testing overflow?   [1]
1
Which mode for testing underflow?   [1]
1
```

The replies to these questions **must** be *1* unless FPVGEN has been modified as described in subsection 9.2. The three possible modes for testing overflow and underflow are described in subsection 3.5.

```
Input name of report file   [NONE] (= standard output file)
NONE
```

Fortran Version:
FPVGEN will write details of any invalid results to the named file. If the reply is *NONE*, the details will be written to the standard output unit (i.e. the terminal when running interactively).

Pascal Version:
The question asked by the Pascal version of FPVGEN is slightly different - "Error output to standard output unit (S) or file ERROUT (E) ? [S]". The user is given no choice for the name of the report file, and should reply with *S* or *E*.

```
Output of all results, right or wrong, to report file?    [N]
N
```

If the reply is *Y*, details of **all** results, whether right or wrong, this is time-consuming and produces large volumes of output except on small test-sets: it should only be requested on short initial runs, or when investigating invalid or suspect results.

```
Model format output of numbers to report file?   [Y]
Y
Input an Edit Descriptor for output of numbers [N] (=no output)
N
Selection parameters of operands to report file?   [N]
N
```

These questions are concerned with the format of output to the report file, which is illustrated and

discussed in the next subsection. The reply to the second question, in the Fortran version, can be a character string specifying a machine-specific format (e.g. *Z20*) or an asterisk (*) specifying the standard list-directed output for floating-point values, or *N* for no output. The Pascal version instead asks the question "Machine format output of numbers to report file? [N]", to which the answer must be *Y* or *N*.

```
Stop after how many invalid results?  [20]
20
```

This reply can prevent FPVGEN producing an excessively large report file.

```
FPV will test  7 operator(s) on
        120 operand pairs.   PROCEED?   [Y]
Y
```

An estimate is supplied of the number of operand pairs to be tested. After every 500 operand pairs tested, a message is written to the screen or default output file to enable the user to judge the rate of progress of FPVGEN.

```
Execution proceeds ...
End of run ...
        600 binary operations were tested.
         48 unary operations were tested.
          0 invalid results were detected.
```

In the count of operations tested, each set of six comparisons counts as one operation. Note that some specified operations are omitted and are not included in the count: e.g. division when operand 2 is zero, square root when operand 1 is negative, or any operation whose result would overflow the model when using Mode 1 for overflow (see subsection 3.5).

When FPVGEN is generating a test-set file, it reports instead the number of records written to that file.

6.3 Format of Output to Report File

Figure 6.1 illustrates the different formats in which details of invalid results are written to the report file. Similar formats are used for valid results if output of all results has been requested.

```
                                                              Note
))))))  Invalid Result in +  ((((((                           (1)
   Opl: + e(     0) 10000000000000000000011                   (2)
              =     0.5000002                                  (3)
          [ Type= 4, Mant= 22, Exp=     0 ]                   (4)
   Op2: + e(  -127) 10000000000000000000011
              =     2.9387369E-39
          [ Type= 4, Mant= 22, Exp=  -127 ]
Up Bnd: + e(     0) 10000000000000000000100                   (5)
              =     0.5000002
Mc Res: + e(     0) 10000000000000000000010
              =     0.5000001
Lw Bnd: + e(     0) 10000000000000000000011
              =     0.5000002

))))))  Invalid .EQ. comparison  ((((((                       (6)
   Opl: + e(     0) 10000000000000000000011
              =     0.5000002
          [ Type= 4, Mant= 22, Exp=     0 ]
   Op2: + e(    -1) 10000000000000000000011
              =     0.2500001
          [ Type= 4, Mant= 22, Exp=    -1 ]
 Comparison gets  T instead of  F.
```

Fig. 6.1

Notes:

(1) The operator is defined by a code:
- `+` for addition
- `-` for subtraction
- `*` for multiplication
- `/` for division
- `S` for square root
- `U` for unary minus
- `A` for absolute value

or the name of a comparison operator.

(2) Operand 1 is here displayed in a 'model format' that corresponds to the model representation described in subsection 3.1, consisting of sign, exponent and mantissa. The exponent is given as a decimal integer in parentheses. The mantissa is displayed in base B notation, with the implied point to the left. The conventional hexadecimal notation is used to represent each digit by a single character (FPV assumes $B \leq 16$). **Note:** this format may be unreliable unless FPVGEN has been modified as described in subsection 9.1.

(3) This line, which is optional, gives operand 1 in an alternative user-specified 'machine format'. In this example the format is the standard list-directed output format for floating-point values. This may be useful to give an idea of the numerical values of the operands and result, but should **not** be relied on for accurate analysis of the results, because of the problems of base conversion. Instead it is usually preferable to specify

a non-standard machine-specific hexadecimal or octal format, so that the machine-representation of the operands can be seen.

(4) This line, which is optional, gives the 'selection parameters' of the operand, as described in Section 4, i.e. the values of the mantissa type [TYPE], the mantissa index [MANT], and exponent [EXP] which caused it to be generated.

(5) The upper bound on the result is displayed first according to the model representation, and then in 'machine format'. Following this, the computed machine result and the lower bound are displayed in the same format(s). If the upper bound and the lower bound are equal, then the value is only printed once, with the prefix 'Ex Res'.

(6) For an invalid comparison, the two operands are displayed in the same formats as above, followed by a line defining the error. If subsequent comparisons on the same two operands are also wrong, the display of the operands is not repeated.

Users can specify that values are to be displayed in either model format or machine format or both (as in this example); at least one of the two must be specified.

6.4 Driving FPVGEN from a Data File

FPVGEN may also be driven by a data file, in which each record must contain one line of the user's replies as described in subsection 6.2. The only exception is that no reply must be supplied for the questions 'Are these satisfactory?' after each sample of mantissa index or exponent values has been displayed; when driving FPVGEN with a data file, the user is not given the opportunity to reject these values and supply them again. Nor is a reply required to the final question 'PROCEED?' A sample driving file is supplied with the FPV package, and it may be edited to specify the desired information in the correct format. No prompts are issued, but details of the input are written to the standard output file.

7. How to Run FPVTGT

This section may be omitted if FPV is not being used in two-phase mode.

7.1 Files used by FPVTGT

FPVTGT uses two input files and one or two output files.

The principal input file, referred to as the **test-set file**, must contain test data generated by FPVGEN. In Fortran its name must be specified by the user (defaulting to BOUNDS); In Pascal its name must be BOUNDS, or a file logically equivalent to BOUNDS. Its format is described in subsection 10.1, but the details are not important unless FPVTGT is being translated into another language. A **driving file** is used exactly as in FPVGEN to supply additional details about the tests to be performed.

The output files are a **standard output file** and a **report file**, used exactly as in FPVGEN.

7.2 Driving FPVTGT

Some of the information required by FPVTGT is independent of the contents of the test-set file, and must be supplied separately at run-time, either interactively from a terminal or from a driving file.

Below is shown a sample terminal session running FPVTGT. Again replies to prompts from FPVTGT are shown in ***ITALICS***. Most of the questions asked by FPVTGT are similar to the last few questions asked by FPVGEN when running an all-in-one test. The annotation in subsection 6.2 applies here also.

```
Are you running interactively (Y / N) ?   [Y]
Y
Input comment line   [ ]
Test
Input name of test-set file   [BOUNDS]
BOUNDS
```

(Note that the Pascal version of FPVTGT does **not** ask this question - it is always assumed that the test-set file is called BOUNDS.)

FPVTGT then reads the first 5 records from the test-set file (see the previous subsection), performs the same checks that FPVGEN performs in an all-in-one test, and displays some of the details before asking any further questions:

```
                base   =            2
  machine precision    =           24
       machine emin    =         -127
       machine emax    =          127
    model precision    =           24
```

```
          model emin    =          -127
          model emax    =           127
       rounding rule    =             1

Operators tested will be +-*/SUAC

Which sign combinations of operands are to be tested?
  Plus o Plus ?    [Y]
Y
  Plus o Minus ?   [N]
N
  Minus o Plus ?   [N]
N
  Minus o Minus ?  [N]
N
Which mode for testing overflow?  [1]
1
Which mode for testing underflow?  [1]
1
Input name of report file   [NONE] (= standard output file)
NONE
Output of all results, right or wrong, to report file?   [N]
N
Model format output of numbers to report file?   [Y]
Y
Input an Edit Descriptor for output of numbers [N] (=no output)
N
Stop after how many invalid results?  [20]
20

Execution proceeds ...
End of run ...
        600 binary operations were tested.
         48 unary operations were tested.
          0 invalid results were detected.
```

8. Interpretation of Results

This section gives some advice on what to do if FPV reports a result that is **invalid** according to the specified criteria (i.e. the parameters of the model and the rounding rule). Such results **may** be due to a definite error in the arithmetic, but in practice it is just as likely that they result from an incorrect setting of the machine or model parameters or the rounding rule. There may also be quite reasonable differences in point of view: designers and implementors of floating-point arithmetic may, indeed should, regard any deviation from the specification by even a single bit as an error; but users may be content with a clean statement that the arithmetic conforms to the Brown model with specified parameters, even if this involves slight penalties.

The advice given here is tentative and may be refined in the light of experience; please assist this process by reporting details of your experience of FPV, using the FPV Report Form.

FPV writes details of each invalid result to the report file in a choice of formats described in subsection 6.3. The term 'error in the result' is used here to denote the amount by which the machine result violates one of its bounds or deviates from the exact result. This error is conveniently measured in 'ulps' (defined in subsection 3.1).

Step 1 Is the error in the result more than just a few ulps? If so, **go to Step 9** (where examples of this kind of error are illustrated).

Step 2 You have found at least one result which is in error by just a few ulps, very likely by only one ulp, as in Figure 6.1. Are there several results that are also in error by just a few ulps? This is a vague question since the meaning of 'several' depends on the size and nature of the samples of operands. The intention is to form an initial impression as to whether the invalid results should be regarded as due to incorrect settings of the parameters and rounding rule, or as due to errors in the arithmetic. (There may in fact be no hard and fast answer to that, if the specification of the arithmetic is vague.) If at this stage you feel fairly confident that your settings of the parameters and rounding rule are correct and you have found a comparatively small proportion of invalid results, **go to Step 8.**

Step 3 You have found several results which show a slight error. If the errors occur with rounding rule 1 (try this if you have not already done so), then **go to Step 5.**

Step 4 You have found several slight errors when using a specific rounding rule, but none when using rounding rule 1. This suggests that your assumption of a particular rounding rule was incorrect. Possibly one of the other rounding rules available in FPV will prove suitable. Otherwise the rounding

rule as implemented is not one of the very limited set available in FPV; it may or may not be the designer's intention that this be so. You may choose to investigate in more detail: are the deviations from the assumed rounding rule confined to one operation, or to particular types of operand or result? Alternatively you may be quite satisfied with having shown that the arithmetic passes the tests with rounding rule 1, i.e. that it conforms to the Brown model (strongly supported). **Go to Step 11.**

Step 5 You have found several results which show a slight error when using rounding rule 1. Consider changing the values of one or more of the parameters. Do the invalid results only occur when the exponent values (in either the operands or the results) are close to EMIN or EMAX? If so, **go to Step 7.**

Step 6 You have found several results which show a slight error according to rounding rule 1 and in which the operands have exponent values in the middle of the range. Consider reducing the **model precision.** Alternatively try rounding rule 0 (Brown model weakly supported), especially if the invalid results are confined to division and square root which may be implemented as composite operations. If the invalid results disappear, then **go to Step 11.** If, however, you continue to get several invalid results after more than one or two reductions of the model precision, then either there are some gross errors in the arithmetic (which may merit investigation as suggested under Step 9), or the arithmetic behaves in some way which violates the underlying assumptions of FPV: **consult NAG. Go to Step 11.**

Step 7 You have found several results which show a slight error when using rounding rule 1, but only when the exponent values are near EMIN or EMAX. Consider modifying EMIN and/or EMAX to reduce the exponent range of the model; it may be that the arithmetic behaves poorly near the ends of the range. This is particularly likely near EMIN; for example if double precision operands are represented by a pair of single precision numbers, then when the exponent of the upper half is sufficiently close to EMIN, the exponent of the lower half is less than EMIN, i.e. the lower half underflows, causing a loss of significance. Also, if floating-point comparisons are performed via a subtraction, then for exponent values close to EMIN, the difference between the operands underflows and all comparisons report equality.

By the assumption at the beginning of this step, you will eventually be able to restrict the exponent range so that invalid results are no longer reported. If the restriction seems unreasonable, **consult NAG. Go to Step 11.**

Step 8 You have found occasional results which are slightly in error, and you believe that your settings of the parameters and the rounding rule are correct. How to proceed may depend on your point of view. You may prefer (perhaps as a user) to see if slight changes to the parameters or rounding rule

will make results valid (e.g. if occasional invalid results are reported with a specific rounding rule, do they disappear if the rounding mode is changed to 1?); in this case, proceed as if there were several such results and **go to Step 5.** Alternatively you may choose (perhaps as a designer) to regard the slight errors as serious and proceed to investigate them in the same manner as grosser errors.

Step 9 You have found a result that is grossly in error, such as the following:

```
))))))  Invalid Result in /  ((((((
   Op1: + e(     0) 10000000000000000000000000000010
   Op2: + e(     0) 10000000000000000000000010000000
Up Bnd: + e(     0) 11111111111111111111111100000101
Mc Res: + e(     0) 11111100000000011111111100000101
Lw Bnd: + e(     0) 11111111111111111111111100000100
```

Fig. 8.1

```
))))))  Invalid Result in *  ((((((
   Op1: + e(     0) 10000000000000000000000000000010
   Op2: + e(     0) 10000000000000000000000010000000
Up Bnd: + e(    -1) 10000000000000000000000010000011
Mc Res: + e(   -63) 10000000000000000000000010000010
Lw Bnd: + e(    -1) 10000000000000000000000010000010
```

Fig. 8.2

In Fig. 8.1, there is a long string of incorrect bits in the mantissa; in Fig. 8.2, although the mantissa is correct, the exponent is totally wrong.

Step 10 Having detected a gross error in the arithmetic, you will usually want to investigate under what conditions it occurs. Does the error depend only on the mantissa values of the operands, possibly also on the exponent difference (independent of the actual values of the exponents)? or does it depend only on the exponent values (independent of the mantissas)? For most errors, the answer to one of these questions is likely to be 'yes'. If the error depends on the mantissa values, which of the mantissa patterns used by FPV and which mantissa index values trigger the error? If the error depends on the exponent values, which values trigger the error? It is possible to use FPV to answer these questions, by selecting a very specific set of tests, for example: a single operation, a single mantissa type, a single exponent value and a range of mantissa index values (or alternatively a single mantissa index value and a range of exponent values). For example, to investigate the error in Fig. 8.1, try selecting: division, operand type 2, exponent value 0 and mantissa index range 1 to 53 for each operand. This might show that the error occurred when $32 \leq i(2) < i(1) \leq 53$ where i(1) and i(2) are the mantissa index values for operands 1 and 2 respectively. Such information would almost certainly be helpful to the

designer or implementor of the arithmetic in tracing the cause of the error.

At this stage, if not earlier, it is likely to be more convenient to **write special small programs** for diagnostic testing rather than use FPV which may be comparatively cumbersome for this purpose, and does not output large numbers of results in an easily digestible form. **Care must be observed when writing such diagnostic programs.** In particular, if operands of the form $1 \pm B^{-i}$ are generated by program, always check that the desired values are obtained, by printing out the numeric values in binary, octal or hexadecimal format (as the system allows) so that the precise bit pattern can be observed. Rounding errors in computing the operands may confuse the issue, and more elaborate code may be required to generate the desired values.

Step 11 This concludes the suggested line of investigation. But be warned that the suggestions are fairly crude. You may have encountered more than one category of invalid results which will require independent paths of investigation. Continue testing with larger and larger samples of operands: this may throw up new errors. **Consult NAG if in doubt or difficulty.**

9. Modifying FPV

In this section we give details of the various modifications to the code of FPV that may be required either to make it completely reliable and robust (as discussed in subsection 4.6) or to enable overflow or underflow exceptions to be trapped and arithmetic to be tested right up to the limits set by the overflow and underflow thresholds (see subsection 3.5).

The modifications all affect the execution phase of the tests (performed by FPVTGT or FPVGEN) and not the generation phase. The modifications to FPVGEN and FPVTGT are identical. We describe in detail the modifications to the Fortran versions; the Pascal modifications are very similar. We give, purely for illustration, examples of how the modifications might be coded for a DEC VAX-11 under VMS.

9.1 Encoding and Decoding Floating-Point Values

To convert an operand from its model-representation in an integer array to the machine's floating-point representation, FPV calls a function MCITOF. Likewise, to convert the result of a floating-point operation from the machine's floating-point representation back to an integer array, it calls a subroutine MCFTOI. The supplied versions of MCITOF and MCFTOI use floating-point operations and are liable to work incorrectly either if the arithmetic is incorrectly implemented, or if it is simply insufficiently accurate. In particular, errors may occur in the least significant digits of the mantissa, or at extremes of the exponent range, where overflow or underflow may interfere.

MCITOF and MCFTOI can be written much more reliably (and often more efficiently) using machine-specific bit-manipulation operations. Of course this requires precise knowledge of how floating-point numbers are stored in the machine, and the code will be machine-specific.

The internal format used by FPV to store floating-point numbers is defined by the integer variables BASE, MANTIS, RADIGS, RADIX and PLACES, which are passed as arguments to both MCITOF and MCFTOI.

BASE is the base B of the arithmetic;

MANTIS is the number of base-B digits in the mantissa of a floating-point number (= the machine-parameter P).

Instead of simply storing each base-B digit in a separate integer array-element, FPV may group the digits together if B is small.

RADIGS is the number of base-B digits stored in a single array-element;

RADIX = BASE**RADIGS;

PLACES is the number of array-elements required
(= MANTIS/RADIGS rounded **up** to an integer).

Each array-element holds one digit to the base RADIX. The mantissa is padded out with zeros if necessary, to make the number of base-B digits an integer multiple of RADIGS. For example, if BASE = 2, MANTIS = 23 and RADIGS = 4, then RADIX = 16 and PLACES = 6; and the mantissa

.11010001010000111110001

is stored as the successive array-elements

13, 1, 4, 3, 14, 2 .

In Fortran each floating-point number is stored in an integer array of length (PLACES+2):

elements (1) to (PLACES) hold the mantissa as just described;

element (PLACES+1) holds the sign: 0 for positive, 1 for negative;

element (PLACES+2) holds the exponent.

Zero is represented by an array with element (1) set to zero, regardless of the values of the other elements.

In Pascal each number is stored as a record with the following type declaration:

```
type BOUND = record
     POSITIVE : boolean;
     EXPONENT : integer;
  SIGNIFICAND : array [1 .. 36] of integer
end;
```

where only the first PLACES elements of SIGNIFICAND are used.

The specification of MCITOF is:

```
REAL FUNCTION MCITOF(X, BASE, RADIGS, RADIX, PLACES, MANTIS)
INTEGER BASE, RADIGS, RADIX, PLACES, MANTIS, X (PLACES + 2)
```

On entry, X contains a floating-point number in FPV internal format.

On exit, MCITOF must return the same number stored as a real variable, through the function name.

The specification of MCFTOI is:

```
SUBROUTINE MCFTOI(F, BASE, RADIGS, RADIX, PLACES, MANTIS, X, NAN)
REAL F, TX
```

```
INTEGER BASE, RADIGS, RADIX, PLACES, MANTIS, X (PLACES + 2)
LOGICAL NAN
```

On entry, F contains a real variable.

On exit, if F contains a valid representation of a floating-point number, then NAN must be set to .FALSE. and X must contain the same number stored in FPV internal format; otherwise NAN must be set to .TRUE. and no assignments need be made to X.

The arguments BASE, RADIGS, RADIX, PLACES and MANTIS are set to the values described above, on entry to both MCITOF and MCFTOI, and must not be changed by them.

Example versions of MCITOF and MCFTOI are:

```
      REAL FUNCTION MCITOF(X, BASE, RADIGS, RADIX, PLACES, MANTIS)
      INTEGER BASE, RADIGS, RADIX, PLACES, MANTIS, X (PLACES + 2)
*   converts from FPV internal format to REAL
*   example coding for DEC VAX-11/VMS Fortran
*   uses subroutine MVBITS to transfer a bit field
*   and intrinsic function ISHFTC to rotate a word
*   REAL operand must first be equivalenced to INTEGER
      INTEGER I, INPOS
      REAL TX
      INTEGER IX
      EQUIVALENCE (TX, IX)
      IX = 0
      IF (X (1) .NE. 0) THEN
*     the number is non-zero
        CALL MVBITS (X (PLACES + 1), 0, 1, IX, 31)
*     insert sign into first bit of IX
        CALL MVBITS (X (PLACES + 2) + 128, 0, 8, IX, 23)
*     bias exponent and insert after sign
        CALL MVBITS (X (1) - RADIX / BASE, 0, RADIGS - 1,
     *               IX, 24 - RADIGS)
*     remove first bit of X (1) (stored implicitly by VAX) and
*     insert after exponent
        INPOS = 24 - RADIGS * 2
        DO 10 I = 2, PLACES - 1
          CALL MVBITS (X (I), 0, RADIGS, IX, INPOS)
*       insert X (2) ... X (PLACES - 1) into IX in logical order
          INPOS = INPOS - RADIGS
  10    CONTINUE
        CALL MVBITS (X (PLACES) / BASE ** (- INPOS), 0,
     *               RADIGS + INPOS, IX, 0)
*     X (PLACES) may hold less than RADIGS bits, so remove
*     trailing zeros by division and insert into end of IX
        IX = ISHFTC (IX, 16, 32)
*     rotate IX to get bits in correct order
      END IF
      MCITOF = TX
      RETURN
      END
```

```
      SUBROUTINE MCFTOI (F, BASE, RADIGS, RADIX, PLACES, MANTIS,
     *                   X, NAN)
      REAL F, TX
      INTEGER BASE, RADIGS, RADIX, PLACES, MANTIS, X (PLACES + 2)
      LOGICAL NAN
*   converts from REAL to FPV internal format
*   example coding for DEC VAX-11/VMS Fortran
*   uses intrinsic function IBITS to extract a bit field
*   and intrinsic function ISHFTC to rotate a word
*   REAL operand must first be equivalenced to integer
      INTEGER I, IX, INPOS
      EQUIVALENCE (TX, IX)
      NAN = .FALSE.
      TX = F
      IX = ISHFTC (IX, 16, 32)
*   rotate IX to get bits in order sign, exponent, mantissa
      X (PLACES + 1) = IBITS (IX, 31, 1)
*   get the sign
      X (PLACES + 2) = IBITS (IX, 23, 8) - 128
*   extract exponent and remove bias
      IF (X (PLACES + 2) .EQ. - 128) THEN
*   if F is zero, assign X all zeros and return
        NAN = X (PLACES + 1) .EQ. 1
*     if negative sign and zero exponent, F is a floating
*     reserved operand
        DO 10 I = 1, PLACES + 2
          X (I) = 0
10      CONTINUE
        RETURN
      END IF
      X (1) = IBITS (IX, 24 - RADIGS, RADIGS - 1) + RADIX / BASE
*   extract X (1) and add back the implicit bit
      INPOS = 24 - RADIGS * 2
      DO 20 I = 2, PLACES - 1
        X (I) = IBITS (IX, INPOS, RADIGS)
*     extract X (2) ... X (PLACES - 1)
        INPOS = INPOS - RADIGS
20    CONTINUE
      X (PLACES) = IBITS (IX, 0, RADIGS + INPOS) * BASE**(- INPOS)
*   X (PLACES) may hold less than RADIGS bits so multiply to
*   fill with zeros
      RETURN
      END
```

9.2 Trapping Overflow and Underflow

Before using Modes 2 or 3 for testing overflow or underflow (see subsection 3.5), it is essential to modify FPV so that any overflow or underflow exceptions are trapped by the program. Trapping these exceptions is an advantage even when using Mode 1, in case any unexpected overflows or underflows occur. No trapping facilities are available in standard Fortran or Pascal, and the actual facilities available vary a lot from system to system (on some systems no trapping facilities may be provided).

The essential features of the modifications are:

- to establish a trap handling subprogram, which should ideally be global to the entire program;

- to provide a suitable trap-handling procedure, here called MCTRAP: if overflow or underflow is signalled, MCTRAP must set a logical variable MACOVR or MACUND to .TRUE. (in Fortran MACOVR and MACUND are in a common block /MACOV/, in Pascal they are global variables);

- to modify the subroutines SETOVM and SETUNM so that they immediately return, thereby permitting Modes 2 or 3 to be used.

The system may require MCTRAP to be either a subroutine or a function. It may expect MCTRAP to return a result to be used in place of the overflowing or underflowing value, but FPV does not require this. Overflows or underflows in the tests may occur either in the subroutine STORE or, if in a comparison, in one of the functions FPVEQ, FPVNE, FPVLT, FPVLE, FPVGT or FPVGE. FPV expects control to be returned either to the immediately following instruction in those subprograms, or to the instruction following the calls to those subprograms. Overflows and underflows may also occur in the subroutine TESTSC which tests MCITOF and MCFTOI, or in the supplied versions of MCITOF and MCFTOI themselves. Ideally the trap-handling subprogram should be established once at the start of the main program, but the system may require it to be established within each subprogram in which overflows or underflows are expected.

Note: in some systems overflow or underflow exceptions are not signalled immediately when overflowing or underflowing values are generated, but only when an attempt is made to use the value in a subsequent floating-point operation. For such systems the subroutine STORE must be modified by inserting a suitable subsequent floating-point operation (possibly, addition of zero), to ensure that the exception is signalled in that subroutine.

Although we have described trapping both overflow and underflow together, it is perfectly possible to trap just one of them. Note also that it may be necessary to use some particular compile-time option or call to a system subroutine to ensure that overflow or underflow (particularly underflow) will cause an exception.

Division by zero, and taking the square root of a negative number are not expected to occur within FPV, and no provision need be made for trapping them.

Examples of statements to be inserted into the main program are:

```
      EXTERNAL MCTRAP
*   establish trap handler for DEC VAX-11/VMS Fortran
```

```
      CALL LIB$ESTABLISH (MCTRAP)
```

An example of a trap handler is:

```
      INTEGER FUNCTION MCTRAP (SIGARGS, MECHARGS)
      INTEGER SIGARGS(*), MECHARGS(4)
*   trap handler for floating-point overflow and underflow
*   example coding for DEC VAX-11/VMS Fortran
*   uses symbolic names defined via INCLUDE statement
      LOGICAL MACOVR, MACUND
      COMMON /MACOV/ MACOVR, MACUND
      INCLUDE '($SSDEF)'
*   determine if condition code in SIGARGS(2) matches code for
*   overflow or underflow
      I = LIB$MATCHCOND (SIGARGS(2), SS$FLTOVFF, SS$FLTUNDF,
     *                                SS$FLTOVF  , SS$FLTUND  )
      IF (I.GT.0) THEN
*      if match found
*      ensure that control will return to instruction following
*      that which caused exception
         CALL LIB$SIMTRAP (SIGARGS, MECHARGS)
         IF (I.EQ.1 .OR. I.EQ.3) THEN
*          if overflow
            MACOVR = .TRUE.
         ELSE
*          if underflow
            MACUND = .TRUE.
         ENDIF
*      control is to be returned to program
         MCTRAP = SS$CONTINUE
      ELSE
*      if not overflow or underflow
*      control is to be returned to system
         MCTRAP = SS$RESIGNAL
      ENDIF
      RETURN
      END
```

The coding of the above trap-handler is very system-specific: however the comments suggest at least some of the points which need to be taken care of.

10. Translating FPVTGT into Other Languages

If FPVTGT has to be re-written in another programming language (possibly machine language), it is not essential, and may not be desirable, to attempt an exact translation of the Fortran or Pascal version. The supplied versions of FPVTGT mimic as closely as possible those features of FPVGEN which are concerned with the execution of the tests as opposed to their generation. Some of those features (e.g. the interactive dialogue prompting the user to specify options, or the choice of formats to display invalid results) may be regarded as frills; also it may be known in advance that some of the options (e.g. modes 2 or 3 for testing overflow or underflow) will not be required.

Before undertaking a translation, consult NAG: NAG may be able to provide a translation for you - or at least the basis of one.

10.1 Format of the Test-Set File

In order to understand some of the details of FPVTGT, it is necessary to know the details of the format of the test-set file.

Figure 10.1 shows a typical example of the beginning of a test-set file. The first five records constitute a header.

```
                                                          Note
         2      24    -127     127                        (1)
        24    -127     127                                (2)
         4       6                                        (3)
         1                                                (4)
+-*/SUAC                                                  (5)
 +  -127 8 0 0 0 0 2 +  -127 8 0 0 0 0 2                  (6)
 +  -126 8 0 0 0 0 2=                                     (7)
 +     0 0 0 0 0 0 0=
U+                  =
 +     1 8 0 0 0 0 0=
 +   -63 8 0 0 0 0 0 +   -63 8 0 0 0 0 1
 -  -127 8 0 0 0 0 2=
 +  -127 8 0 0 0 0 2=
TFFTTFFTFFTTFTTTFFTFFTTF                                  (8)
 +     0 8 0 0 0 0 2 +  -127 8 0 0 0 0 2                  (9)
 +     0 8 0 0 0 0 2 +     0 8 0 0 0 0 3
 +     0 8 0 0 0 0 1 +     0 8 0 0 0 0 2
U+                  =
V+                  =
 +     011 5 0 415 4 +     011 5 0 415 5
 -     0 8 0 0 0 0 2=
 +     0 8 0 0 0 0 2=
FTFFTTFTFFTTFTTTFFFTTTFF
          .
          .
          .
```

Fig. 10.1

Notes:

(1) The machine-parameters B, P, EMIN, EMAX of the target machine.

(2) The model-parameters P, EMIN and EMAX of the target machine.

(3) The parameters PLACES and RADIGS which specify the internal format used by FPV to store floating-point numbers with the given values of B and P (see subsection 9.1).

(4) The rounding rule number, as specified to FPVGEN.

(5) The operators that will be tested by the target program, denoted by the same codes as are used in the report file (see subsection 6.3).

(6) A pair of floating-point numbers, Op1 and Op2, that will be tested with each of the specified operators.

(7) The lower and upper bounds on the result of applying the first specified operator (in this case +) to Op1 and Op2.

In the following records come the bounds on the results of each of the other specified operators. If a pair of bounds is replaced entirely by the single character 'N', this implies that the operation is a unary operation that has previously been tested, and so need not be tested again.

(8) The results of comparing Op1 with Op2 are stored as a string of 24 characters, each either 'T', 'F' or 'X' - true, false or undecidable. The first six characters define the result of performing the comparisons $=$, $\neq$, $<$, $\leq$, $\geq$, $>$ on Op1 with Op2. The second six define the results of the same comparisons of Op1 with (-Op2). Then come the results of comparing (-Op1) with Op2, and finally of (-Op1) with (-Op2).

(9) The next pair of operands to be tested, and so on.

Each floating-point number in the test-set file is represented by a character-string of which the first character is a key. The possible keys are:

' ' - the remainder of the character-string represents a number within machine-range;

'V' - the number overflows the range of the machine, with the next character giving its sign;

'U' - the number underflows the range of the machine, with the next character giving its sign;

'=' - (only used for the second number in a record) the number is equal to the first number in the record.

Numbers within machine range (with key ' ') are represented in a form which corresponds closely to the internal format used by FPV, described in subsection 9.1. The first character holds the sign (as '+' or '-'); the next six characters hold the exponent (as a decimal integer, i.e. in Fortran I6 format); the remaining characters hold the mantissa, each of the first PLACES elements of the integer array being stored as a 2-digit decimal integer (i.e. in Fortran I2 format).

Note that the bounds written to the test-set file are the same regardless of which mode is to be used in testing overflow or underflow; they are in fact the bounds defined for Mode 2. FPVTGT checks whether these bounds overflow or underflow the range of the model and modifies them if necessary.

10.2 Structure of FPVTGT

We describe here the structure of FPVTGT and comment on which features are essential, and which might be omitted from a translation. The description is based on the Fortran version of FPVTGT, but the Pascal version is very similar. Where Pascal routine names differ from the Fortran names, they are given in the call tree at the end of this section. More detailed information is given in comments in the source-text.

Main Program: This has the structure:

```
*   read heading of test-set file and initialise variables
      CALL START (OPERS, NUM, PR)
*   prompt user for specification of options
      CALL PROMPT
*   execute tests
  100 CONTINUE
*   read next pair of operands into FPV internal format
      CALL READOP (OP1, OP2, .TRUE., SKIP)
*   create their negatives in NEGOP1 and NEGOP2
*
*   convert from FPV internal format to floating-point numbers
      X1 = MCITOF(OP1, . . . )
      X2 = MCITOF(OP2, . . . )
      NEGX1 = MCITOF(NEGOP1, . . . )
      NEGX2 = MCITOF(NEGOP2, . . . )
      DO 110 I = 1, NUM
*       test I(th) operator on current pair of operands
        CALL CHECK (OPERS(I), X1, X2, NEGX1, NEGX2,
     *              OP1, OP2, NEGOP1, NEGOP2)
  110 CONTINUE
      GO TO 100
```

The program stops either when the end of the test-set file

is detected (by READOP), or when the limit on the number of errors has been reached. Here NUM is the number of operators to be tested; OPERS is a CHARACTER*1 array containing the operator codes read from the test-set file (see subsection 10.1); OP1, OP2, NEGOP1 and NEGOP2 are integer arrays holding floating-point values in FPV internal format; and X1, X2, NEGX1 and NEGX2 are the corresponding real variables.

Subroutine START: opens the test-set file, reads the first 5 records (see subsection 10.1), and initialises internal variables and arrays, most of which are held in common in Fortran or as global variables in Pascal. START calls subroutine TESTSC to test that the function MCITOF and subroutine MCFTOI (described in subsection 9.1) can reliably handle floating-point numbers with the specified precision and exponent range. This test could be performed separately rather than as part of FPVTGT.

Subroutine PROMPT: conducts the interactive dialogue described in subsection 7.2. If it is known in advance what options will be required, then PROMPT need simply set the relevant internal variables to their required values. PROMPT opens the named report file if one is specified.

Subroutine READOP: reads in a string of characters from a single record in the test-set file. If the string consists of a single 'N', then READOP sets the logical variable SKIP to TRUE and exits immediately: the string corresponds to a unary operation which has already been tested, so should be skipped. Otherwise READOP converts the string into a pair of numbers, which is assumed to be either a pair of operands or a pair of bounds, according as the logical parameter OPERND is TRUE or FALSE respectively. READOP calls subroutine DECODE (see below) to convert each number to FPV internal format. If the end of the test-set file is reached, READOP stops the program.

Subroutine CHECK: if CHECK has been called to test the comparison operators, then it reads from the test-set file the 24-character record defining the correct results, and calls subroutine TSTCOM to perform the tests; otherwise it calls subroutine READOP to read the bounds on the result and calls the subroutine TSTOPS to perform the tests.

Subroutine DECODE: converts a string of characters (which have been read from the test-set file) to a floating-point value in FPV internal format. The format of the character string is described in subsection 10.1.

If the key is '=', DECODE must simply set the flag EQBNDS to TRUE, otherwise it must set EQBNDS to FALSE.

If the key is 'V', DECODE must read the next character (the sign), and then set MCOFLG to +1 if positive or -1 if negative; otherwise it must set MCOFLG to 0.

If the key is 'U', DECODE must read the next character (the sign), and then set MCUFLG to +1 if positive or -1 if negative; otherwise it must set MCUFLG to 0.

If the key is ' ', DECODE must convert the sign, exponent and mantissa and store them in the integer array VALUE (the record VALUE in Pascal) according to the specification in subsections 9.1 and 10.1. Also, DECODE must set the flags MOOFLG and MOUFLG to +1, 0 or -1 to indicate whether or not the number overflows or underflows the model (the values have analogous meanings to those of MCOFLG and MCUFLG).

Subroutine TSTCOM: for each specified combination of signs, tests all six comparison operators on the supplied pair of operands, writing details to the report file of any invalid results. The format and content of such reports can be adapted to particular circumstances.

Subroutine TSTOPS: for each specified non-comparison operator and each specified combination of signs, calls subroutine ERRTES (see below).

Subroutine ERRTES: calls subroutine STORE (see below) to apply the specified (non-comparison) operator to the supplied operands, and then checks the result against the supplied bounds, writing details to the report file of any invalid results. ERRTES could be considerably simplified if no provision were made for modes 2 or 3 for overflow or underflow; also the format and content of the report file could be adapted.

Subroutine STORE: performs a (non-comparison) operation, ensuring that the result is stored in memory, except that in Mode 1 for overflow the operation is skipped if either of the bounds on the result would overflow the range of the model; this is to avoid the possibility of an overflow exception. In the supplied version of STORE the operation is performed in a vectorisable loop, so that vector arithmetic instructions can be tested; this refinement may well not be required.

Subroutine PRTRES: is called by TSTCOM and ERRTES to write a floating-point value to the report file in either 'model-format' or 'machine-format' or both. PRTRES calls subroutine BINOUT to convert a floating-point value to the 'model-format' representation. The details of both PRTRES and BINOUT can be adapted to particular requirements; 'model-format' (and hence BINOUT) could be dispensed with entirely.

Function LWLESS: compares two floating-point numbers that are held in FPV internal format, and returns TRUE if the first is less than the second; otherwise FALSE. This is used for checking that a result obtained by the machine lies within the two bounds on the correct answer.

Function MCITOF: described in subsection 9.1.

Subroutine MCFTOI: described in subsection 9.1.

The other subprograms in FPVTGT have very simple functions which should be obvious from the comments in the source text.

The complete call-tree of the supplied version of FPVTGT is as follows (Pascal routine-names are given in brackets where they differ):

```
FPVTGT
    ASSIGN
    CHECK
        EXINFO
        READOP (READOPERAND)
            ASSIGN
            DECODE
            EXINFO
        TSTCOM (TESTCOMP)
            ASSIGN
            FPVEQ
            FPVGE
            FPVGT
            FPVLE
            FPVLT
            FPVNE
            PRTRES
                BINOUT
        TSTOPS (TESTOPS)
            ASSIGN
            ERRTES (ERRORTEST)
                CHSIGN
                LWLESS
                MCFTOI
                MCITOF
                PRTRES
                    BINOUT
                STORE
    MCITOF
    PROMPT
        NO
        SETOVM
        SETUNM
        YES
    READOP (READOPERAND)
        ASSIGN
        DECODE
        EXINFO
    START
        TESTSC
            CHFTOI
            CRETST
            MCFTOI
            MCITOF
        YES
```

FPV Installation Note

1. Software Supplied

The following files of software are supplied with the FPV package (Fortran or Pascal version)

	number of records	
	Fortran	Pascal
file 1: source-text of FPVGEN	5183	4539
file 2: source-text of FPVTGT	2154	1859
file 3: source-text of FPVPAR	214	191
file 4: sample driving file for FPVGEN	46	46
file 5: sample test-set file	653	653
file 6: sample driving file for FPVTGT	14	13
file 7: sample report file	3526	3526

In all of these files the records are at most 80 characters long.

The programs (files 1 to 3) conform either to standard Fortran 77 or to ISO standard Pascal, level 1.

2. Modifications to the programs

The programs FPVGEN and FPVTGT are intended as far as possible to be portable, and on many systems they will in fact run correctly without change. However before attempting to compile and run them, you should consider making the following modifications to them:

- in the Fortran version of FPVGEN, increase the value of the integer BIGINT (currently set to 32767) to the largest possible integer value: this will make the subroutines for simulating floating-point arithmetic more efficient;

- in FPVGEN (if running in all-in-one mode) or in FPVTGT (if running in two-phase mode) rewrite the subprograms or procedures MCITOF and MCFTOI which convert floating-point values between the program's model-representation in an integer array and the machine's internal floating-point representation. The reasons for doing this are discussed in subsection 4.6 of the User's Guide, and advice on how to do it is given in subsection 9.1. The supplied text of MCITOF and MCFTOI cannot be expected to work correctly in the face of certain errors or anomalies in the arithmetic being tested; it is also likely to be comparatively inefficient.

If you wish to test the setting of the overflow or underflow flags, you will need to make the modifications described in subsection 9.2, but we suggest that you do not do this for your first attempts to run the programs.

3. Precision conversion

The supplied Fortran version of the programs is in **single** precision, and the supplied Pascal version uses the single standard Pascal real data type. If you wish to test any other floating-point data type (whether standard or non-standard), simply change all REAL type declarations to the desired type (e.g. DOUBLE PRECISION or REAL*16). In the supplied Fortran text, the keyword REAL is always followed by at least 12 spaces, so that the keyword DOUBLE PRECISION can be substituted without increasing the line-length.

4. Compile-time or run-time options

Take care to set suitable **compile-time** or **run-time** options to specify, for example:

- whether results of arithmetic operations should be rounded directly to the precision of floating-point numbers stored in memory (rather than to the extended precision of registers) (this is important when testing implementations of the IEEE standard).

- whether vector-arithmetic instructions are to be compiled. The tests of the all operators in FPV are coded in simple vector loops which should be vectorised by any vectorising compiler, thus enabling vector-arithmetic operations to be tested, if available.

- whether overflow or underflow (particularly underflow) should signal an exception.

On a DEC VAX-11/785 the compiled code for FPVGEN occupies about 73,000 bytes, and that for FPVTGT occupies about 33,000 bytes.

5. Installation test

For a quick (but not exhaustive) check that the programs have not been corrupted and that they can be compiled without errors, the following procedure is recommended:

(1) Compile the programs FPVGEN and FPVTGT.

(2) Run FPVGEN using file 4 as the driving file: the report file produced should be the same as file 7.

(3) Edit the third line of file 4 to direct FPVGEN to generate a test-set file and run FPVGEN using the modified driving file: the test-set file produced should be the same as file 5.

(4) Run FPVTGT using file 6 as the driving file and file 5 as the test-set file: the report file produced should be the same as file 7.

Note: the test runs described in steps 2, 3 and 4 above use a trivial model of floating-point arithmetic (B = 2, P = 5, EMIN = -1, EMAX = 2). This model should be admissible on any realistic machine, provided that the base is a multiple of 2 and that arithmetic with powers of 2 is performed exactly; and the output should be identical on all such machines.

6. Contact with NAG

If you have any queries concerning FPV, please write to NAG at one of the addresses given on the inside front cover of the User's Guide.

7. Document Reference

NP1203

Appendix: Provisional Parameter Values

The following table gives (for **initial guidance** only) **provisional** values for the machine and model parameters for certain implementations of floating-point arithmetic. Values of the model parameters are given in parentheses below the corresponding machine parameters, if they differ. The values quoted apply, in general, to all the operators tested by FPV except square-root; an asterisk in the right hand column indicates that they apply to square-root as well. Individual operators in some arithmetics may satisfy tighter criteria. The values quoted do **not** take account of occasional gross errors in the arithmetic, i.e. they are the values which it is believed the arithmetic is intended to conform to if it were correctly implemented.

(s.p. = single precision, d.p. = double precision, h.p. = half precision)

Arithmetic		parameters				rounding rule	
		B	P	EMIN	EMAX		
CDC Cyber 205	h.p.	2	23 (22)	-88	134	1	*
	s.p.	2	47 (46)	-28624	28718	1	*
	d.p.	2	94 (91)	-28578 (-28534)	28718	0	
CDC 7600	s.p.	2	48 (47)	-975 (-974)	1070	1	
	d.p.	2	96 (95)	-927 (-881)	1070 (1069)	0	
Cray-1	s.p.	2	48 (47)	-8192 (-8189)	8191 (8190)	0	
	d.p.	2	96 (95)	-8192 (-8097)	8191 (8190)	0	*
DEC VAX-11	s.p.	2	24	-127	127	2	
	d.p.	2	56	-127	127	2	
	g_floating	2	53	-1023	1023	2	
	h_floating	2	113	-16383	16383	2	
IBM	s.p.	16	6	-64	63	1	
	d.p.	16	14	-64	63	1	
IEEE	s.p.	2	24	-125	128	4	*
	d.p.	2	53	-1021	1024	4	*

FPV Report Form

We invite you to use this form to report the details of any testing which you have undertaken with FPV. Your replies will help us to improve FPV, and to accumulate knowledge of the properties and peculiarities of implementations of floating-point arithmetic. Any suspected errors in the arithmetic should of course also be reported immediately to the vendor of the hardware or software concerned.

Please return the completed form to: NAG Central Office, 256 Banbury Road, Oxford OX2 7DE, England, or, in North America, to NAG Inc., 1101 31st Street, Suite 100, Downers Grove, IL 60515-1263, USA. Thank you for your co-operation.

Completed by (name): .....

(address): .....

.....

.....

(telephone number): .....

(date): .....

A. Brief description of the arithmetic which was tested (e.g. "Cray-1S Fortran single precision arithmetic" or "DECMATH.LIB with Microsoft Fortran V3.2 on IBM PC"; please complete a separate form for each arithmetic which was tested)

.....

.....

Describe (if known) how this arithmetic is implemented (i.e. hardware / microcode / subroutine library) and any other special features:

.....

.....

B. Details of the computing environment in which the testing was performed (if FPV was used in two-phase mode, give details of the environment in which FPVTGT was run)

institution .....

address .....

.....

.....

machine and model no. .....

operating system name and version no. .....

language (i.e. Fortran / Pascal / other) .....

compiler name and version no. .....

run-time library name and version no. .....

floating-point format (e.g. single / double / quad precision)

.....

was FPV used in all-in-one or two-phase mode? .....

C. Details of tests that were passed satisfactorily

machine parameters: b = ... p = ... emin = ... emax = ...

	model parameters p	emin	emax	rounding rule	mode for overflow	mode for underflow
add/subtract	....	.....	.....	....	....	....
multiply	....	.....	.....	....	....	....
divide	....	.....	.....	....	....	....
negation	....	.....	.....	....	....	....
abs. value	....	.....	.....	....	....	....
square root	....	.....	.....	....	....	....
Comparisons	....	.....	.....	....	....	....

If the model parameters satisfied by a particular operator are not equal to the machine parameters, give reasons on a separate sheet, possibly with examples of operations that fail to satisfy more stringent criteria.

D. Details of any errors detected (use separate sheet(s); try to give a very simple example, or pattern of examples, that typifies the error; state whether the error was already known, whether it has been reported to the vendor, and what response the vendor has given so far)

E. Any other comments (e.g. on the usefulness or convenience of FPV, on modifications made to FPV, on peculiarities of the arithmetic, on the vendor's documentation, and so on: use separate sheet)