Computing Science Technical Report No. 89

A Test of a Computer’s Floating-Point Arithmetic Unit

N. L. Schryer

February 4, 1981

A Test of a Computer's Floating-Point Arithmetic Unit

N. L. Schryer

February 4, 1981

A Test of a Computer's Floating-Point Arithmetic Unit

N. L. Schryer

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes a test of a computer's floating-point arithmetic unit. The test has two goals. The first goal deals with the needs of users of computers, and the second goal deals with manufacturers of computers. The first and major goal is to determine if the machine supports a particular mathematical model of computer arithmetic. This model was developed as an aid in the design, analysis, implementation and testing of portable, high-quality numerical software. If a computer supports the arithmetic model, then software written using the model will perform correctly and to specified accuracy on that machine. The second goal of the test is to check that the basic operations perform as the manufacturer intended. For example, if division (x / y) is implemented as a composite operation ($x \times (1/y)$), then the test should detect that fact. Also, the accuracy lost in such a division due to the extra arithmetic operations can tell the manufacturer whether it has been implemented with sufficient care.

Most computers allow the representation of far too many floating-point numbers to allow exhaustive testing of the floating-point arithmetic unit. A small and well-motivated set of floating-point numbers is presented that can be used to detect a vast number of floating-point arithmetic "problems" in existing machines. In fact, that set can be used to detect at least one instance of **every** floating-point arithmetic problem known to the author.

The test is written in portable FORTRAN and has been run on seven different vendor's hardware, with results that range from perfection to disaster.

1. Introduction

Most users of computer software have, at one time or another, asked the question:

> Does the floating-point arithmetic unit of this computer perform correctly?

All too often, the answer to this question is **no.** The difficulties range from gross hardware problems to subtle design errors.

For example, the Honeywell 6080 computer once had the property that a small number (approximately $-\ 10^{-39}$) when divided by roughly 2 gave a large result (approximately 10^{+38}). A subtle design feature on the CRAY-1 results in $1 \times x = \infty$ whenever x has the largest legal floating-point exponent in the machine. These are not isolated or rare cases; the computing world is a jungle of individualistic and sometimes too clever arithmetic units.

It is not sufficient for the user to simply look in the owner's manual for the machine and conclude, for example, that it is a base 2 machine with 56 bits in the mantissa, even though those facts may be correct. The user needs to know the *dynamic* behavior of the arithmetic unit upon the stored data. To make the point clear, assume that addition is botched and $x + y$ is only computed accurate to 24 bits, rather than 56. Then some algorithms are not going to behave correctly if they believe that floating-point operations are accurate to 56 bits. For example, if x is the current Newton iterate and y is the current Newton correction in a nonlinear equation solver, then $x + y$, the next Newton iterate, will only be good to 24 bits. Hence, asking for 30 bits in the solution of a nonlinear equation, while reasonable with a 56 bit mantissa, will result in an infinite-loop for Newton's method. It can take a user a great deal of time to discover that the problem is not in the code for Newton's method, but in the floating-point addition operator. The user needs to know the manner in which floating-point numbers are represented **and** how accurately the arithmetic unit operates upon those stored data values.

It is not easy for the user to assess the computer's dynamic floating-point behavior. A major problem is that most computers allow the representation of far too many floating-point numbers to allow exhaustive testing of the arithmetic unit. For example, the IBM 370 series, in single-precision, is a machine with 6 hexadecimal digits in the mantissa and a base-16 exponent range of $[\ -64,\ +63\ ]$. This represents more than 10^9 floating-point numbers. Checking that $x + y$ gives the correct result for all $(\ x,\ y\ )$ pairs would involve 10^{18} tests, or, at 1 test per micro-second, many thousand cpu-years.

We cannot expect to be able to **prove** the arithmetic unit of a machine correct. In general, all we can do is gain *confidence* that it is correct.

Many attempts have been made previously to test the floating-point arithmetic units of a computer [1, 5, 7]. These efforts have either been aimed at a specific class of machines or have proven to be ineffective on some machines. Further, these attempts have used ad-hoc, or "folk", algorithms to test a small, selected set of conditions with an extremely limited set of operands. None of them have used a systematic definition of "correct" floating-point arithmetic as part of the test.

The folk-algorithm approach fundamentally assumes that the machine being tested is healthy. That is, any errors in the arithmetic are assumed to be in the last digit of its floating-point representation. Under that assumption, it is sufficient to find an *i* so that the i^{th} digit in the machine behaves correctly and the $(i+1)^{st}$ digit behaves in a certain strange manner. From that value of *i*, the precision of the machine can be determined. However, when a machine is not healthy, the i^{th} digit may behave correctly and the $(i-2)^{nd}$ digit may not. Thus, the folk-algorithm approach may not correctly determine precision on a sick machine. The present test uses a much larger set of test operands and a comprehensive mathematical definition of correct floating-point arithmetic.

Section 2 of this paper presents a small and well-motivated subset of all floating-point (FP) numbers that, when used as test operands, triggers all anomalous FP behavior previously known to the author and has found further unsuspected errors. For a base-*b* machine with *t* base-*b* digits in the mantissa, there are $O(\ b^t\ )$ possible mantissas. The sample subset of section 2 involves only $O(\ t\ )$ mantissas. This reduces the amount of testing to a reasonable level. The careful choice of those $O(\ t\ )$ mantissas also allows a good deal of confidence that they will uncover any FP troubles in the machine being tested.

To use the selected mantissas of section 2 to detect problems, we need some definition of what the "correct" result of, for example, $x + y$ is. Section 3 describes a previously developed model of FP arithmetic that will be taken to define correct arithmetic.

Sections 2 and 3 completely define the test by giving the FP numbers to be used as test operands and by stating the criteria by which to judge the FP arithmetic unit's performance upon those operands. The implementation of the test was a ticklish and tedious chore, involving some 70 subprograms and more than 12,000 lines of portable FORTRAN. Section 4 describes this implementation and the tools from the UNIX® operating system used to create the FORTRAN code for the test from the *documentation* for the test cases.

Since the test is implemented in FORTRAN, it should be noted that it is *not* the arithmetic unit of the machine *alone* that is being tested. The entire compiler-operating system-hardware configuration will be tested. This is as it should be, since a correctly designed and implemented piece of hardware may be misused by the compiler. Conversely, a well designed and implemented compiler may compensate for a badly designed piece of hardware. In this paper, the term "machine" will refer to the compiler-operating system-hardware configuration.

Section 5 is a user's guide to the FP test software. Drivers for the test are provided so that the user can request either a "Yes-No" answer to such questions as "Is this a base 2 machine with 48 bits in the mantissa?", or a numerical response to such questions as "What is the base *b* of this machine and how many base-*b* digits does it correctly support?".

Section 6 describes two small main programs for driving the test in the above two modes and gives detailed run-time and memory requirements for using them.

Section 7 is a wish-list of things that should be done, in the future, to make the test more effective.

There are seven appendices that give, in alphabetical order, the results of running the test on hardware from: Amdahl, Cray, DEC, Honeywell, IBM, Interdata and Perkin-Elmer. You might enjoy turning to the appendix dealing with the vendor supplying your local computation center.

2. Sample Subset.

Since we cannot afford to test all FP numbers, we can only test a subset. The choice of that subset is crucial to the success and effectiveness of the test. The subset should be large enough to detect all known anomalous FP behavior and, if possible, increase the list of such. Yet the subset should be sufficiently coherent that people can easily grasp it and have "confidence" in its ability to trigger incorrect FP behavior, if it exists, in the machine being tested.

To motivate and make clear the subset choice, we need to have a simple, clear model for the *representation* of FP numbers. For this we use the tried and true representation of [12] and [4]. This machine model is based on the assumption that FP numbers can be represented in the signed, base *b*, *t* digit form

$$\pm b^e \left(\frac{a_1}{b} + \cdots + \frac{a_t}{b^t} \right) = \pm b^e \sum_{i=1}^{t} \frac{a_i}{b^i} , \tag{2.1}$$

where either $a_1 = \cdots = a_t = 0$ or $0 < a_1, 0 \le a_i < b, i = 1, \cdots, t$, and $e_{\min} \le e \le e_{\max}$.

For our purposes, the unknown parameters in (2.1) are the base *b*, the number of base-*b* digits *t*, and the exponent range limits $e_{\min}$ and $e_{\max}$.

This representation model is sufficiently abstract that it can be used to describe all existing FP machines. The model does not worry about whether the numbers *are* stored in the machine in the form of (2.1), it only asserts that they *can* be represented in that form. This allows analysis to proceed independent of such nitty-gritty details as whether the mantissa is stored in 1's or 2's complement, or whether the mantissa is stored in the machine as a fraction or an integer (as on CDC machines). All these questions, and many more, are submerged in the abstract representation model of (2.1). Yet, this model is delightful for a numerical-analyst to work with. That is, it is practical. Virtually all rounding-error analyses of the last 25 years have been based on it. Thus, as a representational model, (2.1) is an excellent balance between abstraction and specialization.

The immense number of FP numbers that machines support is due nearly entirely to the number of base-*b* digits *t*, not to the number of exponents. For example, on the IBM 370 series in single-precision, there are only 128 exponents, but there are millions of mantissas. This paper concentrates on reducing the number of FP mantissas to be used, and does nothing in particular to reduce the number of exponents to be used. Practical experience with the test has shown how the number of exponent samples can be reduced (see section 6), but that reduction was not built into the test.

It seems a good idea to use test operands with mantissas near the smallest (b^{-1}) and largest ($\sum_{i=1}^{t} (b-1)\, b^{-i} \equiv 1 - b^{-t}$) normalized mantissas. It also seems desirable to use operands with strings of 0's, 1's and $(b-1)$'s in their representations, so that isolated bits and bursts of bits are used. With these thoughts in mind, there are 5 "obvious" mantissa patterns

$$
\begin{array}{lll}
b^{-1} + b^{-i} & \text{for } i = 2, \ldots, t, & Type 1, \\
\sum_{j=1}^{i} b^{-j} & \text{for } i = 1, \ldots, t, & Type 2, \\
0 & Type 3, & \\
(b-1) \sum_{j=1}^{i} b^{-j} & \text{for } i = 1, \ldots, t, & Type 4, \\
(b-1)\,(b^{-1} + b^{-i}) & \text{for } i = 2, \ldots, t, & Type 5.
\end{array}
\qquad (2.2)
$$

On a base-10 machine, for $i = 5$, examples from these five mantissa classes are

$$
\begin{array}{l}
0.100010 \cdots 0 \\
0.111110 \cdots 0 \\
0 \\
0.999990 \cdots 0 \\
0.900090 \cdots 0\ .
\end{array}
$$

For $b = 2$, clearly Types 4 and 5 are redundant and are not used in the test. The mantissas of (2.2) are natural in that they "bunch" together near $1/b$ and 1, the endpoints of the range for normalized mantissas. (2.2) also *contains* the smallest (b^{-1}) and largest ($1 - b^{-t}$) mantissas. As with software, most of the errors in machine design arise in the handling of conditional statements (IF's). Re-normalization is such a conditional response to the mantissa of the computed result being less than b^{-1} or greater than $1 - b^{-t}$. Thus, the FP unit is most likely to fail near these extremities and that is where (2.2) samples the mantissa most densely.

Other choices of mantissa could, and perhaps should, be made. However, the appendices show that those of (2.2) are extremely useful.

Absolutely no claim is made that (2.2) is sufficient to detect all anomalous FP behavior. The only claim made for (2.2) is that it is a small, simple and well-motivated subset of all FP mantissas that can be used to detect a vast number of FP arithmetic "problems" in existing machines. In fact, (2.2) can be used to detect at least one instance of **every** FP arithmetic problem known to the author. However, there are specific cases of pathology not in the sample set. For example, on the Interdata 8/32, $x - 16^{-64}\ x \equiv 0$ for all machine x's; yet the test finds this only for sample x's. The author would appreciate additions to the collection described in the appendices.

A somewhat offbeat example of the utility of (2.2) in detecting errors is the now famous wiring error on the early models of the HP-45 hand calculator. That error resulted in $\log\ (\ \exp(1.01)\ )$ not even being close to 1.01. Since the HP-45 is a base 10 machine, 1.01 is one of the numbers which would be used from (2.2). log and exp are hard-wired processes and $\log(\exp(x)) \approx x$ would be a logical relation to test. In fact, $\log(\exp(x)) \approx x$ failed for **all** x's with type 1 mantissas from (2.2), not just 1.01.

The number of FP mantissas represented by (2.1) is $O(\ b^t\ )$, while (2.2) only represents $O(\ t\ )$ mantissas. This reduces the number of mantissas to be used in the test to a very reasonable level. For example, on an IBM 370 machine in single-precision, we have $b = 16$ and $t = 6$. Using (2.2) as test mantissas, rather than all of (2.1), reduces the number of mantissas from roughly 10^6 to a few dozen. This reduction, from $O(\ b^t\ )$ to $O(\ t\ )$, allows testing to be done in minutes instead of millennia.

3. Dynamic Model.

Section 2 presented a sample set of FP numbers (2.2) to be used as mantissas of operands in testing the dynamic behavior of the FP arithmetic unit of the host computer. How do we decide if $x + y$, for example, has been computed "correctly" by the machine? This section describes a previously developed model of FP arithmetic that will be used to define correct arithmetic for the purposes of the test.

Let $fl\ (\ x\ ^*\ y\ )$ be the machine computed value of $x\ ^*\ y$, where * is any of $+,\ -,\ \times$ and $/$. Then one way to assess the accuracy of FP arithmetic would be to use the representation model (2.1). Using the relation [4],

$$fl\ (\ x\ ^*\ y\ )\ =\ (\ x\ ^*\ y\ )\times(1+\delta\ ),\qquad |\delta|\le\varepsilon\equiv b^{1-t} \tag{3.1}$$

for all *x* and *y* in the sample set, we could attempt to compute $\varepsilon\ \equiv\ \underset{x,\,y}{Max}\ |\ \delta\ |$and then, indirectly, *t*. However, this approach has serious problems. First, δ is very difficult or impossible to compute accurately from (3.1). Second, (3.1) allows $1\ +\ 1\ =\ 2\ +\ \delta$, for δ≠0, which is not acceptable since most everyone agrees that $1\ +\ 1\ =\ 2$ **must** hold on any reasonable machine. Thus, (3.1) is not suitable to define "correct" FP arithmetic.

Clearly, at least some FP results are going to have to be exactly right, like $1\ +\ 1\ =\ 2$, but which ones? A model of the dynamic FP behavior of a machine, which takes this and many other things into account, is given in [13]. That model will be used to define "correct" FP arithmetic. In the interest of completeness and conciseness, an outline of the axioms and results of that paper is presented below.

First, we need to define some terms. The numbers defined by (2.1) are called *model numbers*, and the parameters must be chosen so they are a subset of the machine numbers. The smallest positive model number is

$$\sigma\ \equiv\ b^{e_{\min}\ -1}$$

and the largest model number is

$$\lambda\ \equiv\ b^{e_{\max}}\ (\ 1\ -\ b^{-t}\ ).$$

The maximum relative spacing of FP numbers is

$$\varepsilon\ \equiv\ b^{1-t}$$

$t\ \ge\ 2$ is required. For any real number *x*, we say that *x* is λ*-bounded* if $|\ x\ |\ \le\ \lambda$. We say that *x* is *in-range* if $x\ =\ 0$ or $\sigma\ \le\ |\ x\ |\ \le\ \lambda$, and is *out-of-range* otherwise. If $0\ <\ |\ x\ |\ <\ \sigma$, we say *x underflows*, while if $|\ x\ |\ >\ \lambda$, we say that it *overflows.* Since error analysis is closely akin to interval analysis [8], it is convenient to formulate the axioms in terms of intervals. In particular, if the endpoints of a closed interval are both model numbers, we call it a *model interval;* if they are adjacent model numbers, we call it an *atomic model interval.*

If *x* is a λ-bounded real number, we let $x\ '$ denote the smallest model interval containing *x*. Thus, if *x* is a model number, then $x\ '\ =\ x$; otherwise, $x\ '$ is the atomic model interval that contains *x*.

Since the model is presented in terms of intervals, we need definitions similar to the above for intervals. We say that a real interval *X* is *in-range* or λ*-bounded* if all its elements are. Similarly, a real interval under or over flows if one of its elements does.

If *X* is a λ-bounded real interval, let $X\ '$ be the smallest model interval containing *X*.

For given *X*, we also define an interval X^+ that is generally a little larger than X'. If neither endpoint of X' is 0 or ±λ, then X^+ is obtained from X' by adjoining an atomic model interval at each end. If an endpoint of X' is ±λ, then it is impossible to adjoin an atomic model interval at that end and we shall say that X^+ *overflows* in this case. On the other hand, if an endpoint of X' is 0, we let X^+ share that endpoint, instead of making the extension to ±σ.

There are machine numbers and model numbers. The former is a superset of the latter, although the two sets may be identical.

The *basic arithmetic operations* are addition, subtraction, multiplication, negation and division by $B\ =\ \pm b^k$ where *k* is any integer such that *B* is in-range. Since division is sometimes implemented as a composite of two or more sub-operations, each susceptible to round-off, it cannot realistically be considered *basic*.

For division, and perhaps other non-basic operations, Axioms 1a and 2a are given below as weak alternatives to Axioms 1 and 2. Any operation that conforms to Axiom 1 or 2 will be called *strongly supported*. Any operation that conforms only to Axiom 1a or 2a will be called *weakly supported*.

The parameters must be chosen so that the basic operations satisfy Axioms 1 and 2, and so that division at least satisfies the weaker Axiom 2a.

Axioms

We now present the axioms of [13] that define "correct" arithmetic. Let x and y be λ-bounded machine numbers. We first present the "strong" versions of the Axioms.

Axiom 1 (For $+$, $-$, $\times$ and possibly $/$.)

Let * be a strongly supported binary operator. Then

$$fl\ (\ x\ ^*\ y\ )\ \in\ (\ x'\ ^*\ y'\ )\ '$$

provided that the interval $x'\ ^*\ y'$ is λ-bounded.

Axiom 2 (For negation, division by $\pm b^k$, and possibly reciprocation.)

Let * be a strongly supported unary operator. Then

$$fl\ (\ ^*x\ )\ \in\ (\ ^*(\ x'\ )\ )\ '$$

provided that the interval $^*(\ x'\ )$ is λ-bounded.

The "weak" versions of the above 2 Axioms follow.

Axiom 1a (Alternative for division and perhaps other operators.)

Let * be a supported binary operator. Then

$$fl\ (\ x\ ^*\ y\ )\ \in\ (\ x'\ ^*\ y'\ )\ ^+$$

provided that the interval $(\ x'\ ^*\ y'\ )^+$ is defined.

Axiom 2a (Alternative for reciprocation; not used in the test.)

Let * be a supported unary operator. Then

$$fl\ (\ ^*x\ )\ \in\ (\ ^*(\ x'\ )\ )\ ^+$$

provided that the interval $^*(\ x'\ )^+$ is defined.

We now consider arithmetic comparison between machine numbers that are λ-bounded.

Axiom 3

In comparing λ-bounded machine numbers x and y, the computer may report any result obtainable by an exact comparison of any $\hat{x} \in x'$ and any $\hat{y} \in y'$, but it may not report any other result.

The containment assertions of Axioms 1-2 and comparison assertions of Axiom 3 are the relations to be tested. If Axioms 1-3 hold for all x and y in the sample set, we shall declare the machine to perform FP arithmetic "correctly."

Theorems 1 and 2 of [13] show that Axioms 1-2 imply relation (3.1). However, Axioms 1-2 also imply the following exactness results.

Theorem 1

Let x and y be model numbers, and let * be a strongly supported binary operator. If $x\ ^*\ y$ is also a model number, then

$$fl\ (\ x\ ^*\ y\ )\ =\ x\ ^*\ y.$$

Thus, $1\ +\ 1\ \equiv\ 2$.

Theorem 2

Let x be a model number and let * be a strongly supported unary operator. If $^*x$ is also a model number, then

$$fl\ (\ ^*x\ )\ =\ ^*x\ .$$

Thus, $fl\ (1/b) \equiv b^{-1}$.

It is these additional exactness theorems that make Axioms 1-3 especially well-suited to testing FP arithmetic, as we shall see in the next section. These exactness theorems allow **exact** evaluation of the mantissas of (2.2) and of numbers like $b^e \times m$. The same cannot be said of the δ model of (3.1)

The model of [13] also requires the following relations between ε, σ and λ.

$$\sigma < \varepsilon^2 , \qquad \text{that is, } e_{\min} \le 2\,(1 - t\,)$$

$$1 / \varepsilon^2 < \lambda , \qquad \text{that is, } e_{\max} \ge 2\,t - 1$$

$$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (3.2)$$

$$\sigma / \varepsilon < 1 / (\,\sigma\,\lambda\,) , \qquad \text{that is, } 2\,e_{\min} + e_{\max} \le 3 - t$$

$$1 / (\,\sigma\,\lambda\,) < \varepsilon\,\lambda , \qquad \text{that is, } t + 1 \le e_{\min} + 2\,e_{\max} .$$

These relations will prove useful when we consider the automatic determination of *b*, *t*, $e_{\min}$ and $e_{\max}$ in section 5.

4. Implementation

The test can now be completely specified: the elements of the sample set of section 2 are to be used as operands in the operations of Axioms 1-3 of section 3 and the test will check whether or not those axioms are valid for any given set of parameters. This section shows how to compute the inclusion intervals in the axioms *without* rounding error, provided those axioms are valid for the claimed parameters on the machine being tested. This computation is a rather tricky and tedious chore. The software implementing the test in portable FORTRAN is massive and the tools from the *UNIX* operating system facilitating the implementation are described. Finally, some portability issues are discussed.

Inclusion Interval Computation

Axioms 1-3 define "correct" FP arithmetic in terms of the computed result being an element of a *model interval.* Thus, to test the validity of those axioms, we need to be able to compute the model intervals *exactly.* Many different schemes were considered before the technique described below was evolved for computing the model intervals. For example, multiple-precision (integer) arithmetic, while being able to compute **any** $x\ {}^*\ y$, is far too slow to be useful. The method used was analytic hand-derivation of $x\ {}^*\ y$ for all sample model numbers $x,\ y$ and operations $+,\ -,\ \times$ and /.

The idea is easy to illustrate. Take, for example, the product of 2 numbers with Type 1 mantissas from (2.2). We desire a normalized exponent-mantissa representation for the result of

$$\left[b^{e_1} \ (b^{-1} + b^{-i1}) \right] \times \left[b^{e_2} \ (b^{-1} + b^{-i2}) \right] \equiv$$

$$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (4.1)$$

$$b^{e_1 + e_2} \left\{ b^{-1} (b^{-1} + b^{-i1} + b^{-i2} + b^{-(i1+i2-1)}) \right\}$$

where $i1,\ i2 \ge 2$. If we can obtain a normalized FP representation for the item in braces { ... }, say $b^e\ m$, then we can easily obtain one for $x\ {}^*\ y$, namely $b^{e_1 + e_2 + e}\ m$. The item in braces { ... } is a normalized representation unless $b = 2$ and $i1 = i2 = 2$, when it is $b^0\ (\ b^{-1} + b^{-4}\ )$. Put more succinctly, the normalized representation for the braced-part of (4.1) is given by

```
If ($b = 2$ & $i1 = 2$ & $i2 = i1$)
  { $b^0 (b^{-1} + b^{-4})$ }
                                                        (4.2)
Else
  { $b^{-1} (b^{-1} + b^{-i1} + b^{-i2} + b^{-(i1+i2-1)})$ }
```

Although this is a trivial example of deriving, by hand, the *exact* result of $x\ ^*\ y$, it does illustrate the technique.

Once the exact result of $x\ ^*\ y$ is known, the smallest model interval $[\ L,\ R\ ]$ containing it is also easily computed by hand. Consider, for example, the braced-part of (4.1) given by (4.2). If the result is exactly a model number, then $L = R$. Otherwise, it suffices to get the left-hand-side L of the model interval, since the right-hand-side R can simply be obtained by adding b^{-t} to the normalized mantissa for L. The formula for L of the model interval containing the braced-part of (4.1) is, using (4.2),

```
If ( $b = 2$ & $i1 = 2$ & $i2 = i1$ )
  {
  $b^0(b^{-1}+b^{-4})$
  If ( $t \geq 4$ ) { Exact }
  }
                                                      (4.3)
Else
  {
  $b^{-1}(b^{-1}+b^{-i1}+b^{-i2}+b^{-(i1+i2-1)})$
  If ( $i1 + i2 - 1 \leq t$ ) { Exact }
  }
```

where it is understood that only digits between 1 and *t* are to be used. That is, if $i1 + i2 - 1 > t$ in the "Else" above, then that last term is not added in. The above "code" also notes whether the result is exactly computed. Note that each of the above expressions for L involve adding together powers of the base *b*. By Theorems 1 and 2, each of the powers and sums of them are exactly computable, and, thus, so is *L*, provided the machine supports the model to *t* base-*b* digits.

Note that in using formulae like (4.3) to determine the containment intervals we are using the machines FP arithmetic to check itself. This self-referential technique is believable only because formulae like (4.3) are derived independently of any machine and hence give an independent mechanism for evaluating the containment intervals. Furthermore, as noted above, the FP model guarantees that the formulae will be evaluated **exactly** if the model is supported. If the model is not supported, then containment interval evaluation *and* mantissa (2.2) evaluation may fail. In that case we will use the wrong operands in the arithmetic operations and attempt to find their results in the wrong containment intervals. In short, if the model is supported, the test will run without a hitch, and if it is not supported, the test will encounter all kinds of trouble.

Formulae like (4.3) allow computation of a normalized exponent-mantissa form for the left-hand-side of the smallest model interval containing $x\ ^*\ y$, for all sample model numbers $x,\ y$ and operations *, provided the machine supports the model. Once the interval $[\ L,\ R\ ]$ has been computed, we then have to verify that $fl\ (\ x\ ^*\ y\ ) \in [\ L,\ R\ ]$. This is accomplished by comparisons, as in

```
  $z = fl\ (\ x\ ^*\ y\ )$

If ( $L \leq z$ & $z \leq R$ ) { OK }
Else { WOOPS! }
```

Since *L* and *R* are model numbers, when *z* is a model number, Axiom 3 guarantees that the comparisons will be done correctly. When *z* is *not* a model number, however, the model does *not* guarantee that these comparisons will be done correctly. Specifically, if *z* is just outside $[\ L,\ R\ ]$, *z* may test inside it because Axiom 3 allows equality comparison for numbers in the same model interval. This somewhat fuzzy situation is tolerable for 2 reasons. First, if *z* is more than one model interval outside of $[\ L,\ R\ ]$, the error will be detected. Second, if *z* is outside of $[\ L,\ R\ ]$, but so close to being in it that the error cannot be detected by a comparison, there will probably be another case where the comparison error will go the other way, and the error will be detected.

Thus, if the machine supports the model, then the test will run to completion without incident. Otherwise, we can expect detect a blizzard of errors, among which will be examples of each anomaly. For example, if $fl\ (\ b^{-1}\ )$ is incorrectly computed, then the inclusion interval $[\ L,\ R\ ]$ of (4.3) will not be correct, and neither will the *x* or *y* operands of the input to $x \times y$, even if the multiply itself is correct. Thus, an error in one operator (division) *may* trigger spurious error messages for other operators (multiplication) as well as itself. Usually, such things don't occur, but when they do, a careful examination of the output must be made if the user wishes to determine what *really* went wrong.

UNIX Tools Used

The above outline of the computation of the *exact* containment intervals uses the simplest example of all the $x\ {}^{*}\ y$ results to be obtained. Most such derivations are 2-3 pages in length, and the worst one required 12 pages to derive. Furthermore, there are 42 distinct cases of $x\ {}^{*}\ y$ to be considered. Clearly, it is sufficient to work with positive *x* and *y*, for example, $(+x)\ +\ (-y)\ \equiv\ (+x)\ -\ (+y)$. Since the formulas are trivial when either operand is zero, we can confine our attention to 4 types of operands. Hence there are potentially 16 cases for each of the operators. However, commutativity reduces this to 10 cases each for +, – and ×. For division, each of the 4 cases in which both operands are of Type 4 or Type 5 yields the same formula as another case in which both are of Type 2 or Type 1, so only 12 distinct cases remain. Most of these cases of $x\ {}^{*}\ y$ are fairly routine, but taken together they represent a major effort — more than 100 pages of formulae to implement.

The author attempted to implement these formulae by hand and was quickly convinced that manual transcription of the formulae into a programming language is a silly and dangerous waste of time.

It is silly because of the incredibly nasty nature of the code, with If ... Else's nested as many as 8 levels deep and very messy expressions to evaluate. It is dangerous in that it is *very* important that the program be *correct,* that is, faithfully reflect the derivations for the exact results of $x\ {}^{*}\ y$. Yet, the derivations, at least initially, were full of errors (bugs) and changes would have to be reflected in both the derivations *and* the program. When attempted by hand, these got out-of-phase in a hurry. "Bugs" in the program did not mean "errors" in the derivations, and "errors" in the derivation did not mean "bugs" in the program.

For these reasons it was decided to let the computer do the work. Fortunately, the *UNIX* operating system was available. The *UNIX* operating system supports many software tools that, when viewed as a whole, are unusually powerful aids to programming. The documentation, implementation and maintenance of the test was made much easier with these tools. The result of this effort was a program created automatically from the documentation describing the test cases. Thus, only the documentation had to be written; the program was automatically produced from it. Also, changes in the document (debugging) were automatically reflected in the program.

The way the *UNIX* tools were used is now outlined. It is much simpler to look at a derivation like (4.1)-(4.3) than at a program implementing the actions it contains. That is, (4.3) is easily **seen** to be the correct normalized form for the containment interval of the braced-part of (4.1). The same could not be said of a program written to implement (4.3). Indeed, it is easier to write and study equations than the programs implementing them. Now equations may be written in linear (typewriter) fashion using a phototypesetter language. Thus, the formulae for the test cases can be entered into the computer in a language that permits the production of formulae like (4.1)-(4.3) on paper. This phototypesetter language can also be transformed into another computer language that, when executed, evaluates *L* and notes if it is the exact result or not. There are huge advantages here: the derivation text is entered only once. The pretty listings, like (4.1)-(4.3), **and** the executable code for obtaining *L* are both created automatically from that one input source. Thus, the cerebral cortex can debug the formulae and the computer can make the program for computing *L*.

The *UNIX* tools used are briefly described below.

EQN — A mathematical equation-setting language [3].

When you want to say a_{ij}^{k} you simply type a sub ij sup k. A general rule of thumb for using EQN is that you type at it the words you would use in describing the object to a friend on the telephone. The output of EQN is TROFF.

TROFF— A phototypesetting language [11,pp.2115-2135]

This processor lays out text according to user given commands and built-in rules. For example, the subheading for this paragraph was produced by typing

```
.SH
TROFF-
A phototypesetting language
[11,pp.2115-2135]
```

where the .SH command tells TROFF to make the following input lines **bold**. TROFF also justifies the text on the page, does hyphenation, and generally produces a tidy document from more or less free-form input. This paper is an example of its output.

EFL — A FORTRAN pre-processor language [2].

This pre-processor provides a language of considerable power and elegance. It has an Algol-like syntax, portable FORTRAN [9] as output, the usual control-flow constructions (IF ... ELSE ..., WHILE, FOR, etc.), as well as data structures and a macro facility. A useful feature of EFL is the ability to take input while inside one program from another file during compilation, via the INCLUDE statement.

MAKE [6]

This *UNIX* command makes sure that if A and B are two files, and B can be derived from A by a command sequence, then that command sequence is executed if and only if the last date-of-change of A is later than that of B. Thus, MAKE is often used to keep object libraries up to date with respect to their source code.

ED — The *UNIX* text editor [11, pp.2115-2135]

A line oriented editor with sophisticated pattern matching ability. For example, the ED command

```
g/b sup [ˆ ]*/s/b sup \([ˆ ]*\)/B(\1)/g
```

(don’t worry, it’s easier to type than to read one of these!) changes all occurrences of b sup String into B(String), where String is any string of non-blank characters.

SHELL — The *UNIX* command interpreter [11, pp.1971-1990]

Each process on *UNIX* has a standard input and a standard output. These standard i/o "devices" may be files, teletypes, or even other processes. Thus, for example, the editor ED may take its editing commands from a file (script). Also, the output from one process may be input directly to another process. This connection is called a "pipe" and is denoted by a " | ". A typical use of a pipe is to create a document with the aid of EQN and TROFF, as in

```
EQN files | TROFF
```

where EQN produces TROFF input that is then shipped directly to TROFF to make the document.

Use of the *UNIX* Tools

There is a grand ideal to which software writers aspire — document what you want to do, and then **do** it. Believing this, the test cases were written before writing any code. A typist entered the text into a file, transcribing mathematical formulas into the notation of EQN. As an example, the preceding display for the result of (4.3) was entered as

```
If $(˜b˜=˜2˜&˜i1˜=˜2˜&˜i2˜=˜i1˜)$
  {
  $b sup 0 (b sup -1 + b sup -4 )$
  If ( $t˜>=˜4$ ) { Exact }
  }
                                                   (4.4)
Else
  {
  $b sup -1 ( b sup -1 + b sup -i1 + b sup -i2 + b sup -(i1+i2-1) )$
  If ( $i1 ˆ+ˆ i2 ˆ-ˆ1 ˜<=˜ t$ ) { Exact }
  }
```

The "$" is a delimiter telling EQN what to act on, and the "ˆ" and "˜" tell EQN to leave a little white space.

The problem now consisted of translating such formulae into a programming language (EFL). Also, great care must be taken that the code agree with the document describing it. This means that debugging such code (and formulae) must result in both the program and documentation being changed correctly and simultaneously.

The solution was quite simple: Use an ED script to convert the TROFF input into EFL and use MAKE to keep the whole thing up to date.

It is quite clear that the TROFF input for (4.3), (4.4), given earlier rather resembles an EFL program in structure (IF ... ELSE ...), but not in detail — indeed, it is a rare language that can make sense of $b^{-1}(1 + b^{-i})$. However, the EQN input corresponding to $b^{-1}(1+b^{-i})$ can be converted into a form EFL can recognize — B(1) × (1+B(i)) — by a rather general ED script fragment

```
g/b sup [ˆ ]*/s/b sup -\([ˆ ]*\)/B(\1)/g
g/) *(/s/) *(/)*(/g
```

and we can easily construct an array B such that $B(i) = b^{-i}$. A complete ED script may be constructed along the above lines. It is a long (6 pages) but simple script. The ED script applied to the TROFF input for (4.3), (4.4), gives the EFL program fragment

```
If (b == 2 & i1 == 2 & i2 == i1)
  {
  E = 0; M = (B(1)+B(HiLo(4)))
  If ( t >= 4 ) { EXACT = True }
  }
Else
  {
  E = –1
  M = ( B(1)+B(i1)+B(i2)+B(HiLo(i1+i2–1)))
  If ( i1+i2–1 <= t ) { EXACT = True }
  }
```

which gives the normalized FP representation $b^E \times M$ for the braced-part of (4.1), by defining an array $B(i)$ and a function HiLo(i) so that

$$B(\ HiLo(i)\ ) \equiv \begin{cases} b^{-i}, & 1 \le i \le t \\ 0, & i < 1 \text{ or } i > t. \end{cases}$$

This is accomplished by extending the array $B(i) = b^{-i}$, for $i = 1, \cdots, t$, to have $B(0) = 0 = B(t+1)$, and by defining the statement function

```
HiLo(i) = Max(0,Min(i,t+1))
```

which is used to get the left-hand endpoint of the smallest floating-point interval containing the exact result. There are 42 such EFL program fragments. They form the heart of the floating-point test. There is a standard EFL driver into which these fragments fit, via the EFL INCLUDE mechanism. The resulting 42 programs essentially form the floating-point test.

The above ED script mechanism produces the EFL code directly and automatically from the TROFF input. Thus, only the TROFF input must be altered by hand, the EFL production is automatic. Debugging was literally carried out at the TROFF (not the EFL) level.

However, one great problem still remained. The EFL depends on the TROFF input. How can one be sure that both the EFL and the document for it have been produced from the most recent version of the TROFF input? In all there are 42 such dependencies which must be checked. Here MAKE is invaluable. A file is created for MAKE, giving the dependencies and desired command sequences. Whenever the MAKE file is executed (by saying simply "make"), any TROFF input which has been altered since the last MAKE will be TROFFed, and a copy of it will be converted into EFL.

In all, there are 75 subprograms and some 12,700 lines of FORTRAN in the FP test. The automatic TROFF → EFL conversion mechanism outlined above accounts for 42 of these subprograms and 7351 lines of FORTRAN. Only 33 subprograms were hand-coded, in EFL, resulting in 5349 lines of FORTRAN. These hand-coded subprograms are simple drivers with loops running over all exponents and mantissas to be tested, a subprogram to check that $L \le z \le R$, and, finally, a subprogram (NAN) for checking if a storage location contains an item that is not-a-number. Thus, 60% of the code and 90% of the effort was handled automatically with the help of *UNIX*. Without this help, the project would still be a glimmer in the eye.

Form to be Checked

There are many possible arithmetic combinations, such as register-to-register, register-to-memory, memory-to-memory, etc. We have to decide which of these modes we will test and then stick to it, otherwise we will be comparing apples (registers) and oranges (memory). Also, unless we fix the mode to be tested, we cannot tell which mode we *have* tested. The implementation language, FORTRAN, does not allow the user to determine if a number is in a register or not. Thus, we shall force the storing of all computed results before checking to see if they are in the correct containment interval. This is accomplished by putting the results in an array Z, and the containment intervals into arrays L and R.

To see if $L(i) \le Z(i) \le R(i)$, a subroutine "CheckInterval" is called with arguments L, R, Z and N, where N is the length of the arrays. Since the arrays L, R and Z are assigned to the PORT [10] stack (with $L(1) \equiv$ Stack(iL), where iL is the pointer to L on the stack, etc.), the separate compilation requirements of FORTRAN virtually force the results Z(*i*) to be stored into the array Z before calling CheckInterval. Thus, only the stored values of L, R and Z are tested.

Portability

The FP test passes the PFORT Verifier [9] with two exceptions. First, the code uses arrays with 4 subscripts. This is illegal in the 1966 ANSI standard for FORTRAN, but it *is* legal in the new 1977 FORTRAN standard, which allows 7 subscripts. Second, character strings are packed 2 characters per integer word, while only one per word is considered portable by the Verifier. It is extremely useful to have these 2 character words (for LT, LE, EQ, NE, GE, GT). The alternative to this would be to use two words (one for "L" and another for "T", for example) and have a character-compare routine to decide what they represent ("LT", for example). However, character compare inside Integer words is not a portable concept. So the two-character-per-word usage inside the test remains. This is a minor restriction, and should not cause any trouble. Other than the above 2 exceptions, the code is portable in the *static* sense of the Verifier, that is, it will compile successfully on any machine supporting the ANSI 1966 FORTRAN standard.

When it comes to *running* the program there are 2 subtle points for the user to be aware of. One concerns vector machines and the other concerns machines with FP words reserved for non-numeric items, like instructions, ∞, etc.

All $fl(x * y)$ results are computed in loops like

```
Do i = 1, N { Result(i) = x(i) * y(i) }
```

so that the machine’s vector arithmetic is tested if it exists, and if the compiler recognizes the construct. To guarantee that the result registers have been stored (and do not remain in registers or other strange places), the test passes the vector Result to a subprogram that checks that Result(i) is in [L(i), R(i)], for i = 1,..., N. On vector machines this means that the user must check that the vector registers *are* stored in memory before a subroutine call. Also, if the user wishes to check the *scalar* arithmetic unit of a vector machine, it will be necessary to tell the compiler not to produce vectorized assembly language.

The other important portability consideration is whether the machine being tested has reserved FP words, or not-a-number (NAN) representations.

The NAN problem is most serious when testing that $z \in [L, R]$ and one of z, L or R is not-a-number, that is, a reserved word representing an instruction, $\pm\infty$, ∞ (projective, unsigned), underflow, or some other non-numeric object. This is especially bad if ∞ is both ≤ and ≥ all numbers. For example, on the CRAY-1 the result of an exponent overflow is ∞, and $\infty \le x \le \infty$ holds for *any* number *x*. Conversely, neither $\infty < x$ nor $x < \infty$ is true for any number *x*. In such a case, $L \le \infty \le R$ tests **true** for *any* numbers *L* and *R*, even though ∞ is clearly not the correct result. If such NAN's are not detected, they may give rise to incorrect model parameters. Detection of NAN's is not a provably portable concept. However, in the interest of making the test as robust as possible, a NAN test has been provided that at least works for the CRAY-1, the only machine with NAN's to which the author has access.

Infinities, infinitesimals (0_+ for an underflowed positive quantity, etc.), undefineds, and any other NAN's are expected to have strange properties, such as $\infty / 2 = \infty$ and $0_+ + 0_+ = 0_+$, that cannot be satisfied by ordinary FP numbers. Hence, the following code is likely to detect them.

```
If      ( $x \le -1$ ) { NAN = $x / 2 \le x$ }    # Is $x = -\infty$?
Else If ( $x \ge +1$ ) { NAN = $x / 2 \ge x$ }    # Is $x = +\infty$?
Else If ( $x \equiv 0$ )   { NAN = false }
Else If ( $x \le 0$ )   { NAN = $x + x \ge x$ }    # Is $x = 0_-$?
Else                     { NAN = $x + x \le x$ }    # $x = 0_+$.
```

where *NAN* is a Logical variable and the # indicates the rest of the line is a comment. This code correctly detects NAN's on the CRAY-1, but may not on other machines. Users should carefully check to see if their machine has NAN's and perhaps alter the subprogram NAN to *guarantee* it detects them, otherwise the test may not tell the truth.

Output

When FP errors are detected, the user may elect to print-out the offending *x*, * and *y* along with the inclusion interval $[L, R]$. This is most useful when done in base-*b* notation ("octal" or "hex"), so that the user can see the base-*b* digits in *x*, *y*, $fl (x \, ^* \, y)$, *L* and *R*. Errors as subtle as we are trying to detect may not show up if the results are printed in decimal (unless it's a decimal machine) because of rounding-error in the conversion from base-*b* to base-10. A program may have to be written by the user of the package for printing such "octal" results. The calling sequence is

SUBROUTINE OCTALP (MESSG, NMESSG, IA, NIA)

and it should print the message MESSG of NMESSG characters, followed (on the same line) by the Integer array IA of length NIA, in "octal" or "hex", or whatever is appropriate on the machine being tested. IA has been Equivalenced to a FP number in the program calling OCTALP. Typically, in single-precision NIA is 1 and in double-precision NIA is 2. A default, reasonably portable, version of OCTALP is distributed as part of the test software. When the base *b* is $2, 4,$ or 8 the "octal" format "O24" is used, otherwise it is assumed that the machine is base-16 and the "hex" format "Z18" is used.

If the machine being tested requires another format from those provided by the above default, and the user wants to see the numbers that are causing the trouble, OCTALP will have to be recoded. However, if the format desired is of the form ?## where ? is some single character, say "B", and ## is the character version of NIA (i.e., if NIA = 10 then ## ≡ "10"), then changing the statement

DATA ZEE / 1HZ /

to

```
DATA ZEE / 1HB /
```

in OCTALP will suffice.

See the DEC appendix for an example of the output produced by OCTALP.

5. Basic Software

There are three levels of software available for testing FP units. First, there is the bottom level FPTST that allows the user maximum flexibility in getting a "Yes-No" answer to questions like

Is this a base-2 machine with 48 bits in the mantissa?

Since this software is flexible, it is messy to use. At the next level, when the user wishes to use FPTST in a simple manner, a driver for FPTST has been written to make its usage easier. Finally, a top-level driver for FPTST has been written to answer, numerically, such questions as

How many base-*b* digits *t* does this machine support?

The next section presents main programs to drive these three levels of software.

Lower Level Software

The lowest level software is FPTST, which takes as input the values of b, t, $e_{\min}$ and $e_{\max}$, some arrays describing the sample FP mantissas from (2.2) to use, the operators $(+,\ -,\ \times,\ /,\ .LT.,\ \cdots$, etc.) to be tested, plus a few flags to control the output of the test. It decides whether the model is supported for the given parameter values.

The declaration for FPTST is

```
LOGICAL FUNCTION FPTST(B,T,EMIN,EMAX,
                       EX,NEX,IX,NIX,
                       EY,NEY,IY,NIY,
                       EXPAND,
                       CHEKPM,CHEKTM,CHEKDV,
                       CHEKNG,CHEKCM,CHEKUF,
                       TYPE,
                       EPRINT,JTEST,
                       OUTPUT)
```

The arguments to FPTST are, where → denotes an input value and ← denotes an output value,

B → The Integer base-*b* of the machine.

T → The Integer number of base-*b* digits in the mantissa.

EMIN → The Integer minimum exponent $e_{\min}$.

EMAX → The Integer maximum exponent $e_{\max}$.

EX → An Integer array of exponents for operands x in $x\ {}^{*}\ y$.

NEX → The length of the array EX.

IX → An array of i values to be used in the sample set of mantissas (2.2) for X.

NIX → The length of the array IX.

EY → An Integer array of exponents for operands y in $x\ {}^{*}\ y$.

NEY → The length of the array EY.

IY → An array of i values to be used in the sample set of mantissas (2.2) for Y.

NIY → The length of the array IY.

EXPAND → The maximum number of interval expansions to be allowed for division, that is, the number of X^{+} expansions to be made for the division inclusion interval, if it is necessary, see section 3. The model of [13] assumes EXPAND to be 1, at most.

EXPAND ← The actual number of interval expansions needed for division.

CHEKPM → A Logical flag that determines if Plus and Minus are to be tested.

CHEKTM → A Logical flag that determines if Times is to be tested.

CHEKDV → A Logical flag that determines if Division is to be tested.

CHEKNG → A Logical flag that determines if Negation is to be tested.

CHEKCM → A Logical flag that determines if Comparison is to be tested.

CHEKUF → A Logical flag that determines if UnderFlow is to be tested. That is, whether results that underflow are to be generated and tested. When CHEKUF is .TRUE., the NAN check for 0_+ and 0_- (underflow) is disabled.

TYPE → A Logical array TYPE(5,2). TYPE(i,j) tells if Type i is to be tested for x and y, with j=1 corresponding to x and j=2 to y, see (2.2)

EPRINT → A Logical flag to say if error conditions are to printed out when they arise.

JTEST → A Logical flag that determines if the test is looking for all errors or will stop if it encounters severe trouble. JTEST = .TRUE. means that the test won’t stop under any circumstances. Otherwise certain conditions may cause an abort of the run. Also, if JTEST is .TRUE., FPTST does no printing at all.

OUTPUT → The name of a subprogram that is to be called by FPTST whenever an error is detected and EPRINT is TRUE. OUTPUT must be declared EXTERNAL in the program calling FPTST. A default, reasonably portable version of OUTPUT, called FPTSO, is distributed with the test software. The calling sequence to OUTPUT is

```
SUBROUTINE OUTPUT ( SX, TX, IX, EX, X,
                    OP,
                    SY, TY, IY, EY, Y,
                    RESULT, L, R )
```

SX → An Integer giving the sign of X, 1 is + and 2 is –.

TX → An Integer giving the Type (1,...,5) of X, see (2.2).

IX → An Integer giving the value of *i* used in (2.2) for the mantissa of X.

EX → An Integer giving the base-*b* exponent of X.

X → The Real number X that caused X OP Y to fail.

OP → The operation OP that has failed. OP is an Integer with 2 characters stored in it. The characters and what they stand for are

" +" stands for addition.

" —" stands for subtraction.

" *" stands for multiplication.

" /" stands for division.

"EQ" stands for equality comparison.

"LT" stands for less-than comparison.

"GT" stands for greater-than comparison.

"LE" stands for less-than or equal comparison.

"GE" stands for greater-than or equal comparison.

"NE" stands for not-equal comparison.

"NG" stands for negation. When OP = "NG", all values of input relating to Y are undefined, since negation is only applied to X.

"RE" stands for reciprocation.

SY → An Integer giving the sign of Y, 1 is + and 2 is −.

TY → An Integer giving the Type (1,...,5) of Y.

IY → An Integer giving the value of *i* used in (2.2) for the mantissa of Y.

EY → An Integer giving the base-*b* exponent of Y.

Y → The Real number Y that caused X OP Y to fail.

RESULT → The Real computed value of *fl* (X OP Y).

L → The Real left-hand-endpoint of the containment interval for RESULT.

R → The Real right-hand-endpoint of the containment interval for RESULT.

The user can do whatever is desired in OUTPUT to print, store, analyze, etc. the numbers that are causing the trouble.

The function value of FPTST is

TRUE if errors have been detected, FALSE if no errors have been detected.

Higher Level Software

One of the problems in using FPTST is the necessity for the user to create arrays IX, IY, EX and EY telling FPTST what sample set numbers to use. Typically, the user wants to test some IX's near 1, t and maybe $t/2$, and some EX's near $e_{\min}$, $-t$, 0, $+t$ and $e_{\max}$, that is, some parameters near the "edges" of the representation. For addition and subtraction, exponents near $\pm t$ are considered "edges" because when numbers with such exponents are added to or subtracted from numbers with 0 exponents they tend to "fall off the end" and get lost. Similar things are usually wanted for Y. To make the creation of such arrays easier, a driver, FPTSD, was created for FPTST. The calling sequence for FPTSD is the same as that for FPTST except that IX, NIX, EX, and NEX, and the corresponding items for Y, are replaced by descriptions of the "edges" the user wants tested.

The calling sequence for FPTSD is

```
LOGICAL FUNCTION FPTSD(B,T,EMIN,EMAX,
                       BEX,LBEX,BEY,LBEY,
                       BIX,LBIX,BIY,LBIY,
                       NBEX,NBEY,
                       NBIX,NBIY,
                       EXPAND,
                       CHEKPM,CHEKTM,
                       CHEKDV,CHEKNG,
                       CHEKCM,CHEKUF,
                       TYPE,
                       EPRINT,JTEST,
                       OUTPUT)
```

The arguments to this subprogram are precisely as for FPTST, with the exception that IX, NIX, EX, NEX, and the corresponding items for Y, have been replaced by

BEX → An Integer array of exponents of X about which exponent samples are to be clustered, see NBEX below. A typical array would be BEX = (EMIN, −T, 0, +T, EMAX).

LBEX → The length of the array BEX. In the preceding example, LBEX = 5.

BEY → An Integer array of exponents of Y about which exponent samples are to be clustered. A typical array would be BEY = (EMIN, −T, 0, +T, EMAX).

LBEY → The length of the array BEY. In the preceding example, LBEY = 5.

BIX → An Integer array of *i*’s for the mantissa of X, see (2.2), about which mantissa samples are to be clustered, see NBIX below. A typical array would be BIX = (1, T/2, T).

LBIX → The length of the array BIX. In the preceding example, LBIX = 3.

BIY → An Integer array of *i*’s for the mantissa of Y, see (2.2), about which mantissa samples are to be clustered. A typical array would be BIY = (1, T/2, T).

LBIY → The length of the array BIY. In the preceding example, LBIY = 3.

NBEX → An array giving the number of "basic" samples to be taken for X exponents. For $i = 1, \cdots$, LBEX, if BEX(i) is in (EMIN, EMAX), exponents BEX(i)–NBEX(i)+1, ... ,BEX(i)+NBEX(i)−1 will be used. Only exponent samples in the legal exponent range [EMIN, EMAX] are taken.

NBEY → An array giving the number of "basic" samples to be taken for Y exponents.

NBIX → An array giving the number of "basic" samples to be taken for X mantissas. For $i = 1, \cdots$, LBIX, if BIX(i) is in (EMIN, EMAX), mantissas of (2.2) with parameters BIX(i)–NBIX(i)+1, ... ,BIX(i)+NBIX(i)−1 will be used. Only mantissa samples in [1, T] are taken.

NBIY → An array giving the number of "basic" samples to be taken for Y mantissas.

The function value of FPTSD is

TRUE if errors have been detected, FALSE if no errors have been detected.

When either of the exponent or mantissa sampling schemes generate overlapping sample fields, no redundancies are taken. Only distinct elements of the fields are used. For example, NBEX ≡ 1 ≡ NBEY and NBIX ≡ T ≡ NBIY would test *all* mantissa patterns for *x* and *y*, with a few exponents of *x* and *y*.

Very High Level Software

The programs FPTST and FPTSD described above decide whether a given set of model parameters are correctly supported by the host machine. It is often useful to have a program that will decide what the "correct" model parameters are from scratch. Many have set out in quest of this grail. None have found it, and perhaps nobody ever will. However, since people want such a beast, an attempt was made using the driver FPTSD. The following algorithm was created in a spirit of "let’s see what can be done." It is also a useful example of what can be done using FPTST and FPTSD.

To discover what b, t, $e_{\min}$ and $e_{\max}$ are, *automatically*, a few corners must be cut. For example, within the model of section 3, it is not possible to reliably tell the difference between a base-16 machine and a base-2 machine. Indeed, within that model, the base is not uniquely defined [13]. Let t_2 be the number of base-2 digits correctly supported and t_{16} be the number of base-16 digits correctly supported. If $t_2 = 4\ t_{16} - 3$, then the machine is quite probably base-16. However, the converse is not the case, see the Interdata appendix. In general, if a machine’s base is a power of 2, we cannot reliably tell which power it is. Thus, we will test the machine for bases 2, 3 and 10, with the notion that these are the only plausible, and only existing, choices. The test will discover whether b is a power of $2,\ 3$ or 10, but not which power. Now note that a base-2 machine cannot pass the test as a base 3 machine because $1\ /\ 3$ is not exactly representable in binary. Similar arguments show that the base choices 2, 3 and 10 are mutually exclusive, only one of them can be valid on a given machine. This makes the search for the base-*b* fairly easy and definitive. The problem is deciding if the machine supports base-*b* arithmetic according to the model of section 3.

To decide if the machine supports some kind of base-*b* arithmetic, we can treat t, $e_{\min}$ and $e_{\max}$ as independent variables. We can first find out what t is by running FPTSD with all exponent samples $\equiv 0$, so that exponent effects are eliminated. In this manner, a value of t is found which passes FPTSD. We can then find the smallest value of $e_{\min}$ which is supported with t base-*b* digits, and then find the largest value of $e_{\max}$ that is supported with t base-*b* digits.

We obtain t using bisection. If we have some given upper-bound on t, we use it. Otherwise, we can obtain one by inverse-bisection. That is, for $t = 2,\ 4,\ 8,\ 16,\ \cdots$ use FPTSD to see if t base-*b* digits are supported. This testing sequence must terminate with a value of t that fails the test. Thus, we can assume we are given an upper bound on t. We have a lower-bound on t, namely 2 or the preceding power of 2. Hence, we can use bisection to find the largest value of t that passes the test.

It is important that t be found before $e_{\min}$ and $e_{\max}$. For example, finding $e_{\min}$ using $t = 2$ may give an incorrect value for $e_{\min}$. An example of this is given in the CRAY-1 appendix, where using $t = 2$ in finding $e_{\min}$ in double-precision would give too small a value for $e_{\min}$.

With a maximal value of t now available, we can get $e_{\min}$ using a similar algorithm. First we check that t, $e_{\min}(t) \equiv 2(1-t)$ and $e_{\max}(t) \equiv 2t-1$ pass the test, where $e_{\min}(t)$ and $e_{\max}(t)$ are the largest, and smallest, respectively, values of $e_{\min}$ and $e_{\max}$ that satisfy (3.2). That is, we check the smallest permissible exponent range consistent with t. If we have some given lower-bound on $e_{\min}$, we use it. Otherwise, we keep doubling $e_{\min}$ until it finally fails the test. This gives both lower and upper bounds on $e_{\min}$ and we can use bisection to obtain the smallest value of $e_{\min}$ that passes the test.

While bisecting, we must be careful that as the value of $e_{\min}$ is lowered, the relations of (3.2) remain valid. This means that $e_{\max}$ may also have to be raised as $e_{\min}$ is lowered. We test the smallest value of $e_{\max}$ consistent with $e_{\min}$ and t in (3.2).

With the smallest value of $e_{\min}$ determined, we can play the same game and find the largest value of $e_{\max}$ that passes the test. At the end of this sequence, we have a *locally optimal result*, that is, none of t, $e_{\min}$ or $e_{\max}$ can be extended by 1 and still pass the test.

In short, the algorithm used to obtain b, t, $e_{\min}$ and $e_{\max}$ is

```
For ( $b$ = 2, 3, 10 )
    {
    If ( No upper-bound given for $t$ )
        {
        For ( $t_{bad}$ = 2, 4, 8, 16, $\cdot\cdot\cdot$ )
            {
            Test $t_{bad}$ with FPTSD, using all exponents ≡ 0.
            If ( $t_{bad}$ fails the test )  { Break }
            }
        }
    Else { Let $t_{bad}$ be the given upper-bound }

    Bisect in [2, $t_{bad}$] to get largest supported $t_{bad}$.

    Test $t$, $e_{\min}(t)$ and $e_{\max}(t)$ with FPTSD, using all exponents ≡ 0.

    If ( Fails the Test ) { Check the next base $b$ }

    If ( Lower-bound not given for $e_{\min}$ )
        {
        $e_{\min}^{bad}$ = $e_{\min}(t)$
        While ( $e_{\min}^{bad}$ passes the test )
            { $e_{\min}^{bad}$ = $2 \times e_{\min}^{bad}$ }
        }
    Else { $e_{\min}^{bad}$ =  the given lower-bound }

    Use bisection in [$e_{\min}^{bad}$, $e_{\min}(t)$] to
    get the smallest supported value of $e_{\min}$.

    If ( Upper-bound not given for $e_{\max}$ )
        {
        $e_{\max}^{bad}$ = $4 - t - 2\,e_{\min}$
        }
    Else { $e_{\max}^{bad}$ =  the given upper-bound }

    Use bisection in [$e_{\max}(t)$, $e_{\max}^{bad}$] to
    get the largest supported value of $e_{\max}$.
    }
```

The software implementing this is called FPTSB and its declaration is described below.

```
    LOGICAL FUNCTION FPTSB(B,
                           T,EMIN,EMAX,
                           CHEKUF,
                           NBEX,NBEY,NBIX,NBIY,
                           EXPAND)
```

The arguments B, EXPAND and CHEKUF to FPTSB have the same meaning as in FPTSD.

T	→	If T = 0, then no upper-bound on t is known. Otherwise, use T + 1 as the upper-bound.
T	←	The supported value of t.
EMIN	→	If EMIN = 0, no lower-bound on $e_{\min}$ is known. Otherwise, use EMIN − 1 as the lower-bound.

EMIN ← The supported value of $e_{\min}$.

EMAX → If EMAX = 0, no upper-bound on $e_{\max}$ is known. Otherwise, use EMAX + 1 as the upper-bound.

EMAX ← The supported value of $e_{\max}$.

The function value of FPTSB is TRUE if base-B arithmetic passes the test, FALSE otherwise.

When testing for *t*, FPTSD is called with (NBEX,NBEY,NBIX,NBIY) ≡ (1, 1, NBIX, 1), BEX ≡ BEY ≡ (0), BIX ≡ BIY ≡ (1, T/2, T), and again with "NBIX, 1" changed to "1, NBIY". These calls to FPTSD are cheap, and effective.

When testing for $e_{\min}$ and $e_{\max}$, FPTSD is called with (NBEX,NBEY,NBIX,NBIY) ≡ (NBEX, 1, 1, 1), BEX ≡ BEY ≡ (EMIN, −T, 0, +T, EMAX), and again with the "NBEX, 1" changed to "1, NBEY". Again, these calls to FPTSD are cheap, and effective.

If the host machine can be told to continue execution after FP under and over flow, its FP accuracy may be determined by calling FPTSB with T = EMIN = EMAX = 0. This is accomplished without using *any* information about the machine's architecture.

When the values (NBEX,NBEY,NBIX,NBIY) ≡ (4, 4, 8, 8) are used in the call to FPTSB, it correctly reports the values of *b*, *t*, $e_{\min}$ and $e_{\max}$ on all machines listed in the appendix. Empirically, FPTSB reports the correct results when the search parameters NBEX, NBEY, NBIX and NBIY are chosen sufficiently large.

The Double-precision versions of FPTST, FPTSD and FPTSB are DFPTST, DFPTSD and DFPTSB, with all Real arguments made Double-precision.

6. Main Programs

This section discusses two main programs, one each for driving FPTSD and FPTSB. The parameters in each main program are described and the effect these have on the run-time, memory usage and effectiveness of the test are illustrated.

The main program for driving FPTSB will be called MAINB. In EFL [2], it is, in double precision,

```
  Procedure

# To get floating-point model parameters from scratch.

  Integer I1MACH,B,T,Emin,Emax,nbEX,nbEY,nbIX,nbIY,Expand
  Logical DFPTSB,CheckUF

  Define StackLength 20000
  Common ( CSTAK ) { Long Real Dstak(500) }
  Long Real Wstak(StackLength); Equivalence Dstak(1),Wstak(1)

  ISTKIN(StackLength,4)    # Tell the Stack how long it is.

  B    = I1MACH(10)
  T    = I1MACH(14)
  Emin = I1MACH(15)
  Emax = I1MACH(16)

  CheckUF = True    # Generate and test underflow results.

# Set the sampling sizes.

  nbEX = 1; nbEY = 1
  nbIX = 1; nbIY = 1

  Expand = 1    # Default model value.

  If (      DFPTSB( B,T,Emin,Emax,CheckUF,nbEX,nbEY,nbIX,nbIY,Expand) ) {}
  Else If ( DFPTSB( 3,T,Emin,Emax,CheckUF,nbEX,nbEY,nbIX,nbIY,Expand) ) {}
  Else If ( DFPTSB(10,T,Emin,Emax,CheckUF,nbEX,nbEY,nbIX,nbIY,Expand) ) {}
  Else
    {
    SETERR(
      "MAINB - Machine base is not a power of B, 3 or 10 - strange.",60,
               1,2)
    }

  Wrapup()

  Stop()

  End
```

The Common region CSTAK declares the PORT [10] stack, and the Define tells the stack how long it is. I1MACH is the PORT Library [10] machine constant subprogram, for example, I1MACH(2) is the local FORTRAN write unit. Wrapup prints out the stack usage at the end of the run. SETERR is the PORT Library error handler [10].

When run on the CRAY-1 in single-precision, and pretending that I1MACH uses the raw machine parameters as described in the user manual for that machine, the output of the above program is

```
1real floating-point bisection test with

   b =   2,   t =    48
   emin =   -8192,   emax =    8191

   nbex =       1,   nbix =   1
   nbey =       1,   nbiy =   1



 this test produced and tested results which underflow.
 if overflow or divide check errors occurred,
 then the basic machine constant bounds have been incorrectly set.


 the correct real floating-point model values for b =   2  are,
 assuming that x/y is implemented as a composite operation ( 1),


    t    =         47
    emin =      -8189
    emax =       8190



 used     1110 /    40000 of the stack allowed.
```

which took 18 seconds to run. The output here is in lower-case because the CRAY-1 has only that case. The above run shows the ability of the program to detect that */* is a composite operator on the CRAY-1, requiring (1) X^+ expansion in the model, and that it is a base-2 machine. The reasons for these values of t, $e_{\min}$ and $e_{\max}$ are discussed in the CRAY-1 appendix.

When run on the VAX 11/780, running the *UNIX* operating system, in single-precision the output is

```
1Real Floating-point Bisection Test With

   B =   2,   T =    24
   Emin =    -127,   Emax =     127

   nBEX =       1,   nBIX =   1
   nBEY =       1,   nBIY =   1



 This test produced and tested results which underflow.
 If overflow or divide check errors occurred,
 then the basic machine constant bounds have been incorrectly set.


 The correct Real floating-point model values for B =   2  are,


    T    =         24
    Emin =       -127
    Emax =        127



 USED      834 /    20000 OF THE STACK ALLOWED.
```

which took 2 minutes 59 seconds to run. Note that the above run correctly determines that the VAX’s static representation is supported in its dynamic operation.

The run-time and storage requirements of FPTSB are linearly proportional to NBEX, NBEY, NBIX and NBIY. Note that FPTSB required only a few minutes to do its job, it was tuned to be fast and reasonably reliable.

The results for the above examples were obtained with NBEX, NBEY, NBIX and NBIY all equal to 1. But with those values, we can not have great confidence that FPTSB was right. For example, on the Interdata 8/32 the values of NBEX, etc., must be `4, 4, 8, 8` in order to give the correct results. That machine and the VAX 11/750 are the only machines for which the default parameters in MAINB sometimes report the wrong results, see those appendices.

If the machine being tested is reasonably healthy, with errors (if any) near the ends of the mantissa and exponent ranges, then FPTSB will report the correct result. However, if the machine is really sick, with errors in the "middle" of the mantissa and exponent ranges, FPTSB may report the wrong results, unless NBEX, etc., are chosen sufficiently large.

Once FPTSB has done its job, we need to test in more thorough and expensive ways. There are many ways to drive FPTSD and the following main program, which will be called MAIN, is canonical. In EFL [2], it is, in double precision,

```
  Procedure

# Main procedure to test the PORT machine constants.

  Integer I1MACH,B,T,Emin,Emax,
          BEX(5),lBEX,BEY(5),lBEY,BIX(3),lBIX,BIY(3),lBIY,
          nbIX(3),nbIY(3),nbEX(5),nbEY(5),Expand,j
  Logical CheckPM,CheckTimes,CheckDiv,CheckNeg,CheckComp,CheckUF,
          Type(5,2),
          EPrint,JustTest,DFPTSD,valu
  External DFPTSO

  Define StackLength 20000
  Common ( CSTAK ) { Long Real Dstak(500) }
  Long Real Wstak(StackLength); Equivalence Dstak(1),Wstak(1)

  ISTKIN(StackLength,4)    # Tell the Stack how long it is.

  B = I1MACH(10); T = I1MACH(14)

  Emin = I1MACH(15); Emax = I1MACH(16)

# Set the Exponent sampling range.

  BEX(1) = Emin; BEX(2) = -T; BEX(3) = 0; BEX(4) = +T; BEX(5) = Emax
  lBEX = 5
  BEY(1) = Emin; BEY(2) = -T; BEY(3) = 0; BEY(4) = +T; BEY(5) = Emax
  lBEY = 5

# Set the mantissa sampling range.

  BIX(1) = 1; BIX(2) = (T+1)/2; BIX(3) = T; lBIX = 3
  BIY(1) = 1; BIY(2) = (T+1)/2; BIY(3) = T; lBIY = 3

# Set nbE, the number of basic exponent samples.

  Do j = 1, 5 { nbEX(j) = 3; nbEY(j) = 3 }

# Set nbI, the number of basic mantissa samples.

  Do j = 1, 3 { nbIX(j) = 4; nbIY(j) = 4 }

  Expand = 1    # Default model value.

  CheckUF = True    # Generate and test underflow results.

# Set the flags for performing the tests.

  CheckPM = True; CheckTimes = True; CheckDiv = True
  CheckNeg = True; CheckComp = True

  Do j = 1, 2
    {
    SETL(5,False,Type(1,j))
    If ( B == 2 ) { SETL(3,True,Type(1,j)) }    # Test Types 1-3.
    Else { SETL(5,True,Type(1,j)) }    # Test all Types.
    }

  EPrint = False; JustTest = False

  valu = DFPTSD(B,T,Emin,Emax,
                BEX,lBEX,BEY,lBEY,
                BIX,lBIX,BIY,lBIY,
                nbEX,nbEY,nbIX,nbIY,
                Expand,
                CheckPM,CheckTimes,CheckDiv,
                CheckNeg,CheckComp,CheckUF,
                Type,
                EPrint,JustTest,DFPTSO)

  Wrapup()

  Stop()

  End
```

When run on the CRAY-1 in double-precision the above program prints

```
1long real floating-point test with

   b =   2,   t =    94
   emin =   -8099,   emax =    8190

  bex =  -8099   -94     0    94  8190
  bey =  -8099   -94     0    94  8190
  bix =      1    47    94
  biy =      1    47    94
 nbex =      3     3     3     3     3
 nbey =      3     3     3     3     3
 nbix =      4     4     4
 nbiy =      4     4     4


 nex, ney =    21    21
 nix, niy =    15    15



 this test will produce and test results which underflow.
 if overflow or divide check errors occur,
 then the basic machine constants have been incorrectly set.


 floating-point test completed successfully,
 provided that x/y is implemented as a composite operation ( 1).


 used     6356 /    80000 of the stack allowed.
```

which took 554 seconds to run. MAIN echoes its input and prints nEX, nEY, nIX and nIY, the total number of exponent and mantissa samples used in the test.

When run on the VAX 11/780 in double-precision the above main program printed

```
 1Long Real Floating-point Test With

    B =   2,   T =    56
    Emin =    -127,   Emax =     127

   BEX =   -127   -56     0    56   127
   BEY =   -127   -56     0    56   127
   BIX =      1    28    56
   BIY =      1    28    56
  nBEX =      3     3     3     3     3
  nBEY =      3     3     3     3     3
  nBIX =      4     4     4
  nBIY =      4     4     4


  nEX, nEY =    21    21
  nIX, nIY =    15    15



  This test will produce and test results which underflow.
  If overflow or divide check errors occur,
  then the basic machine constants have been incorrectly set.


  Floating-point test completed successfully.


  USED     5444 /    40000 OF THE STACK ALLOWED.
```

which took 55 minutes 27 seconds to run.

The run-time for FPTSD is proportional to nEX × nEY × nIX × nIY. Thus, doubling all of them makes it run 16 times longer. The storage required is proportional to *Max* (nEX, nEY, nIX, nIY). These statements are true for sufficiently large values of nEX, nEY, nIX and nIY. We now show how the run-time and storage requirements for FPTSD vary with these parameters, and illustrate a battery of tests that thoroughly test FP arithmetic.

Once MAINB has been used to obtain the "correct" values for the machine being tested, and MAIN has been used to more thoroughly test those parameters, even more thorough searching is in order. A glance at the appendices shows that errors pop up all over the exponent and mantissa range. The default test in MAIN will catch most of those errors, but not the serious errors found on the Interdata 8/32 and VAX 11/750. To make sure that such disasters are not lurking about somewhere in exponent-mantissa space, we need to check more exponents and/or more mantissas. This will be more costly, and we should proceed cautiously, lest we spend time and money that needn't be spent.

There are precisely 4 knobs in MAIN to be twiddled — the arrays NBEX, NBEY, NBIX and NBIY. By varying these 4 parameters, a sequence of tests can be constructed to quickly discover problems, if they exist, and determine the correct model parameters. For the rest of this section, it will be assumed that each of the arrays NBEX, NBEY, NBIX and NBIY is a constant. If we wish to test more thoroughly, we could take NBEX = $e_{\max} - e_{\min} + 1$ = NBEY and NBIX = t = NBIY. This would test every exponent and every mantissa pattern. However, as we shall see, this would be very expensive and probably overkill as well.

We can test the exponents and mantissa patterns in four basic ways.

1) Let the elements of one of the arrays be large, and the other 3 be small.
2) Let the elements of two of the arrays be large, and the other two arrays be small.
3) Let the elements of three of the arrays be large, and the other array be small.
4) Let the elements of all four of the arrays be large.

If all the large elements are called *n*, then these four regimes have run-times proportional to $n,\ n^2,\ n^3,\ n^4$, respectively. Clearly, since there may be errors in the parameters, we should run the *n*-time regime first. That way, any errors that can be detected will be, cheaply. If that test is successful, then the n^2-time regime should be run, and so on.

In this way, a hierarchy of runs can be made, each of which contains the testing of the previous runs, and extends that testing to new test samples and combinations. If the parameters are in error, the cheapest test that can detect it will.

A Test Paradigm

It is recommended that the following paradigm be used when testing the model parameters.

Run MAINB
Run MAIN to test the output of MAINB
Run the *n*-time tests:

MAIN with (NBEX,NBEY,NBIX,NBIY) = $(E,1,1,1)$

MAIN with (NBEX,NBEY,NBIX,NBIY) = $(1,E,1,1)$

MAIN with (NBEX,NBEY,NBIX,NBIY) = $(1,1,M,1)$

MAIN with (NBEX,NBEY,NBIX,NBIY) = $(1,1,1,M)$

where $E\ =\ 100$ and $M\ =\ 100$, say. If any of these runs uncovers an error in the model parameters, they should be changed appropriately, and the paradigm should be restarted at the MAIN stage.

Once the above battery of tests is successfully completed, the quadratic tests should be done. That is, MAIN with (NBEX,NBEY,NBIX,NBIY) = $(E,E,1,1)$, $(E,1,M,1)$, etc. In these tests, since the run-time is proportional to the product of the large elements of the basic arrays, it would be good to keep *E* and *M* to say 10 or so, at least at first.

Similarly, the cubic and quartic tests using MAIN can be run, one after the other. The final quartic test, MAIN with (NBEX,NBEY,NBIX,NBIY) = (E,E,M,M), will run for a very long time. As a background job, this makes it a good candidate for finding transient, intermittent floating-point problems. At the same time, it is beating the exponent-mantissa space of the machine to death.

For illustrative purposes, the run-time and memory usage results of running the above paradigm for testing on the VAX 11/780 in single-precision are now given. MAINB took 2 minutes and 38 seconds to run and used 2252 Integer locations on the PORT stack. Table 1 gives the run-time and stack requirements for the various runs using MAIN.

Table 1

NBEX	NBEY	NBIX	NBIY	Integer stack used	Run-time (hr:min:sec)
1	1	1	64	3837	49
1	1	64	1	3837	43
1	64	1	1	1334	5:18
64	1	1	1	1334	5:02
8	8	1	1	1058	14:54
8	1	8	1	3949	9:42
8	1	1	8	3949	9:37
1	8	8	1	3949	9:09
1	8	1	8	3949	9:24
1	1	8	8	3984	7:13
1	1	64	64	3984	7:25
4	4	4	1	2646	15:57
4	4	1	4	2646	16:12
4	1	4	4	2682	15:40
1	4	4	4	2682	15:21
6	6	8	8	4144	9:11:41

Using the information in Table 1, we can predict that the test using (NBEX,NBEY,NBIX,NBIY) $= (t/2, t/2, t, t)$ would run for 36.8 hours. Such a run would test every mantissa pattern in (2.2) and a very large number of exponents. The full test, testing all exponents and mantissa patterns, would run for 12.3 days. For the manufacturer, this may be acceptable during development, for users this thorough a test would be a useful background job.

It is worthwhile noting that **all** floating-point errors detected so far have been found with the linear (i.e., $(1, E, 1, 1)$, etc.) and/or quadratic (i.e., $(E, 1, T, 1)$, etc.) test paradigms. These tests require only minutes to run and appear to be just as effective as the more expensive testing schemes.

It is also worth noting that **all** floating-point errors detected so far on machines with $b > 2$ have been found with Type 1 and Type 4 matissas. Thus, it appears that only two Types of mantissa need be active for any given test. This could considerably lower the cost of running the test when $b > 2$.

7. Wish-List

There are many things that could or should be done to improve the efficiency and effectiveness of the test. Some of these are easy to do and others will require substantial effort. In no particular order, they are

Register Arithmetic

Checking that register-to-register and register-to-memory operations work would be invaluable, but from a standard FORTRAN environment this appears impossible. However, with a special compiler it may be possible to force such arithmetic and thus test it. This seems worthwhile, but difficult.

Correct Rounding

The current test does not report if the machine has done correct rounding because the model of [13] does not require it. However, manufacturers may wish to test that they have done it "right". It would be a bit of work, but the formulae for computing the containment intervals could be made to specify which end-point corresponds to correct rounding. This seems fairly easy and worthwhile.

Testing Underflow Operands

The current test assumes that all operands in the test are model numbers. Many people assume that computation can continue when underflows are generated. The current test only checks that any underflow results generated are correct. It does not check that any subsequent operations with those underflowed quantities are valid. Extending the formulae for the containment intervals to include the case where the exponents for either x or y are less then $e_{\min}$ would be fairly easy and worthwhile, see the Interdata appendix.

Extra-precise Operands

The current test assumes that all patterns of the form (2.2) have $i \le t$, that is, they are all model numbers. It would be good to know that operations with extra-precise operands, $i > t$, if any, are also correct, see the DEC appendix dealing with the VAX 11/750. Axioms 1-3 would then require looking at $x' * y'$, rather than $x * y$. Since any element x of (2.2) with $i > t$ has the property that x' is a model interval with endpoints in (2.2), with $i \le t$, the current formulae can still be used. However, the logic governing their use would become more complex. This is a rather messy and tedious, but worthwhile, job.

Exponent Sampling

The cost of testing the mantissa arithmetic has been lowered tremendously in this test. However, nothing has been done to sample the exponents cleverly. Something should be done about exponent sampling. It is unclear just what, however.

Integer Arithmetic

The formulae used in the FP test can also be used to test Integer arithmetic. For example, with $e = t$, b^e times the mantissas of (2.2) are all integers. Conversion of the test to testing Integer arithmetic would be reasonably simple. However, most Integer units work correctly because users really get upset when they don't. Thus, there seems little to gain from creating such a test.

Mixed-Mode

Testing mixed-mode arithmetic, that is, Real-Integer, Integer-Double-precision, etc., would be a bit of work and would be marginally useful.

A Missing Type Combination

All $x * y$ results have been successfully hand-computed, with the exception of the case of (Type 2) / (Type 5) of (2.2), with $i_1 > i_2$. For this case, the testing of the result has been ignored because the result is unknown. Several people have tried to get a closed-form, normalized result for this case, and nobody has succeeded, yet. Completing this case would be satisfying and useful. (Anyone wishing to try is more than welcome to it.)

8. Conclusions

The appendices show the ability of the test to detect floating-point arithmetic errors due to hardware malfunction and/or design, and software bugs. The results for the Interdata 8/32 and VAX 11/750 especially show that simply testing a digit or two at the "end" of the mantissa or exponent range **is not sufficient to detect serious trouble.** Anomalous behavior may be triggered by tickling the 40th bit of the mantissa or the exponent -64. The test allows such items to be tested easily in a methodical manner.

It does appear that treating the exponent and mantissa separately is sufficient. That is, all mantissa problems uncovered so far have had the property that they can be triggered with an exponent at $e_{\min}$, $-2, -1, 0, +1, +2$ or $e_{\max}$. Conversely, all exponent problems uncovered so far have had the property that they can be triggered with a mantissa pattern with $i = 1$, $t/2$, or t. Since it appears that there is no need to test all kinds of strange mantissa and exponent patterns together, testing them separately is a reasonable way to determine the floating-point unit's integrity. Such a test requires at most a few minutes of run time, and seems extremely reliable. However, on the chance that there is evil lurking in a corner of exponent-mantissa space, a complete exponent-mantissa test should be run in the background. Such a background job, which might require a cpu-month or more, is also an excellent way to detect transient malfunction of the floating-point units.

Acknowledgments

This effort is part of the continuing development of portable mathematical software at Bell Labs. Many enlightening and spirited discussions within the numerical group continue to accompany this development. For their interest in, and comments on, this test, A. D. Hall, S. I. Feldman and P. A. Fox deserve many thanks. W. S. Brown deserves great credit for creating a model of FP arithmetic that makes it possible to test the arithmetic, for many comments on the test and a willingness to revise the model in response to real-world FP behavior. The model and the test have grown up together. The designers of the FP unit on the Honeywell 600-6000 series deserve special thanks for having done nearly everything correctly, so that the errors in the derivations and code could be detected and corrected. They also deserve thanks for having done enough wrong to make the test an interesting project. The people at Cray Research, DEC, Honeywell and Perkin-Elmer were quite responsive to the bug reports generated by the test. They quickly explained and repaired all serious problems brought to their attention. Finally, I thank Jim Cody of Argonne for bringing the problem of NAN’s on the CRAY-1 to my attention; see the CRAY-1 appendix.

Bibliography

[1] W. J. Cody and W. M. Waite, **Software Manual for the Elementary Functions,** Prentice-Hall, 1980.

[2] S.I. Feldman, "The Programming Language EFL", Bell Laboratories Computing Science Technical Report #78, 1979.

[3] B.W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics", *Comm. ACM* **18,** 151-157(1975).

[4] G. Forsythe and C. Moler, **Computer Solution of Linear Algebraic Systems,** Prentice-Hall, New York, 1967.

[5] W. M. Gentleman and S. B. Marovich, "More on Algorithms that Reveal Properties of Floating-Point Arithmetic Units", *Comm. ACM,* **17,** 276-277(1974).

[6] S. I. Feldman, "Make — A Program for Maintaining Computer Programs", Bell Laboratories Computing Science Technical Report #57, 1977.

[7] M. A. Malcolm, "Algorithms to Reveal Properties of Floating-Point Arithmetic", *Comm. ACM,* **15,** 949-951(1972).

[8] R. E. Moore, **Interval Analysis,** Prentice-Hall, 1966.

[9] B.G. Ryder, "The PFORT Verifier", *Software — Practice and Experience* **4,** 359-377(1974).

[10] P.A. Fox, A.D. Hall and N.L. Schryer, "The PORT Library Mathematical Subroutine Library", *TOMS,* **4,** 104-126(1978).

[11] "UNIX Time-Sharing System", *BSTJ* **57,** 1897-2312(1978).

[12] J.H. Wilkinson, **Rounding Errors in Algebraic Processes,** Prentice-Hall, New York, 1963.

[13] W. S. Brown, "A Simple but Realistic Model of Floating-Point Computation", Bell Laboratories Computing Science Technical Report 83, November, 1980.

Appendix Alliant

These machines have a base-2 hardware representation with an exponent range of -125 to $+128$ and $t = 24$ in single-precision, in double-precision the exponent range is -1021 to $+1024$ with $t = 53$. Thus, they implement the IEEE Floating-point standard for single and double precision.

FX8

This machine has been thoroughly tested by Prof. Wolfgang Fichtner of the ETH, Zurich, and Dave Berkeley, and the above parameters are correctly supported.

FX/2800

After an initial blaze of compiler errors, Dave Berkeley found that this machine correctly supports the IEEE parameters.

Appendix Amdahl

470/V8

This machine has a base-16 hardware representation with an exponent range of -64 to $+63$, $t = 6$ in single-precision, and $t = 14$ in double-precision.

This machine has been thoroughly tested, and the above parameters are correctly supported under the MVS operating system with the Fortran H Extended compiler.

Appendix Apollo

300

The 300 has a base-2 FP representation with an exponent range of -127 to $+127$, $t = 24$ for single-precision, and $t = 56$ for double-precision. The FP is implemented in software.

This machine has been probed by Prof. Bill Gropp of Yale and the above parameters are not yet correctly supported. The software is buggy and $t = 23$, $e_{\mathrm{min}} = -75$ and $e_{\mathrm{max}} = 50$ are currently supported. The penalty on e_{min} is due to comparisons being done by subtract and compare to zero, causing very small, close but not identical numbers to compare equal through underflow. The rest of the problems have been reported to Apollo and are being fixed.

420

The 420 has a base-2 FP hardware representation with an exponent range of -127 to $+127$, $t = 24$ for single-precision, and $t = 56$ for double-precision.

This machine has been thoroughly tested by Prof. Bill Gropp of Yale and the above parameters are correctly supported.

Appendix CCI

6/32

This machine has a base-2 hardware representation with an exponent range of -127 to $+127$, with $t = 24$ in single-precision and $t = 56$ in double-precision. This machine was tested by John Walden and found to be healthy.

The Past

Early versions of this machine had serious errors. For example, in double precision,

$$fl\,(\,1 - 2^{-55}\,) \geq 1$$

and

$$fl\,(\,(\,2^{127}\,(\,2^{-1} + 2^{-54}\,)\,) \times (\,-\sum_{i=1}^{53} 2^{-i}\,)\,) = -2^{+126} \times \sum_{1}^{25} 2^{-i}$$

which is off in the 26th bit.

In single precision, for $x = 2^{25}\,(\,2^{-1} + 2^{-15}\,)$ and $y = 0$, got

$$fl\,(\,x \times y\,) = 2^{-104}\,(\,2^{-1} + 2^{-15} + 2^{-28} + \cdots\,),$$

which is not 0.

Appendix CDC

7600

The CDC-7600 has a base-2 FP hardware representation with an exponent range of −974 to +1070, $t = 48$ in single-precision, and $t = 95$ in double-precision. The test was run under the FTN5 compiler by Kirby Fong at Lawrence Livermore Laboratories.

Single Precision

Two penalties must be assessed to obtain correct model parameters. First, $t = 47$, rather than 48, must be used. Second, $e_{\min} = -929$ must be used, rather than -974.

The reason for the first, precision, penalty is that one bit is lost when certain pairs of operands are compared. Comparisons are done by subtraction followed by a compare to zero.

The second, minimum exponent, penalty is due to FTN5 normalizing a result after subtracting two numbers. If two numbers that are different but both very small, the normalization leads to an underflow so that FTN5 thinks they are identical.

With these two penalties, the CDC-7600 passes the test in single-precision.

Double-Precision

Even though the CDC-7600 software has $t = 96$ bits in the mantissa, the manufacturer only claims 95 bit accuracy.

The parameters supported in the FP test are $t = 47$, $e_{\min} = -926$ and $e_{\max} = +1069$.

Most operations are good to 95 bits, but multiplication and division are sometimes only good to 49 bits.

In .EQ. and .NE. FTN5 forgets that the hardware needs to shift coefficients to align exponents and this can allow the 48^{th} bit to be shifted off. It is thus possible for two numbers that differ in bit 48 to appear equal.

Appendix Convex

Convex 200

This machine has a base-2 hardware representation with an exponent range of -125 to $+128$ and $t = 24$ in single-precision, in double-precision the exponent range is -1021 to $+1024$ with $t = 53$. Thus, the machine implements the IEEE Floating-point standard for single and double precision.

This machine has been thoroughly tested by John Walden and the above parameters are correctly supported.

Appendix Cray

Cray-1

The Cray-1 has a base-2 FP hardware representation with an exponent range of –8192 to +8191, $t = 48$ in single-precision, and $t = 96$ in double-precision. The test was run under the COS 1.08 operating system on hardware modified to make multiplication commutative.

Single Precision

Three penalties must be assessed to obtain correct model parameters. First, $t = 47$, rather than 48, must be used. Second, $e_{\min} = -8189$ must be used, rather than -8192. Third, the largest exponent must be $e_{\max} = 8190$, rather than 8191.

The reason for the first penalty is that multiplication uses a phantom 49^{th} bit for pre-rounding. For example, for $x = 2$ and $y = 0.1 \cdots 1_2$ we have $fl(x \times y) = 2$, and $t = 48$ would not result in exact multiplication being valid.

The minimum exponent penalty is due to x / y being a composite operation, essentially we have $x / y \equiv x \times (1/y)$. Thus, $fl(x / (2^{-8192} \times 2^{-1})) \equiv x \times (2^{+8194} \times 2^{-1}) = \infty$, the projective, un-signed infinity.

The maximum exponent penalty results from the way the Cray-1 tests for overflow in multiplication. In taking a × b, if exponent(a)+exponent(b) $> e_{\max}$, the result is defined to overflow. This is often correct, but sometimes is too conservative.

With these three penalties, the Cray-1 passes the test in single-precision.

Double-Precision

Even though the Cray-1 hardware has $t = 96$ bits in the mantissa, the manufacturer only claims 95 bit accuracy.

The parameters supported in the FP test are $t = 94$, $e_{\min} = -8099$ and $e_{\max} = +8190$.

The penalty on t is due to $/$ being done in software, using multiple operations. The value of $t = 94$ passes the FP test with *one* expansion (X^+) of the containment interval for division results.

If *two* expansions are allowed, $t = 95$ passes the test. However, this extra expansion is contrary to the model, and 95 is not used. But it does show that the other arithmetic operations are supported to 95 bits. Also, since division involves three arithmetic sub-operations, it should require the extra expansion. So the designers of the double-precision division software can tell that they have introduced the minimum possible error.

The minimum exponent must be penalized by 93, to -8099, because the comparison operations (.eq., .lt., etc.) are done by subtraction and comparison to zero. Unless the 93 penalty is put on $e_{\min}$, $x - y$ may underflow causing x .eq. y to be true when $x \neq y$. This problem does not arise in single-precision because although $x - y$ may underflow, its representation, *before being stored*, has the correct sign.

The reason for the penalty on $e_{\max}$ is the same as it was in single-precision.

The Past — Single-Precision

Early in 1980, the FP test reported most of the above results, but failed to detect the penalty on $e_{\min}$. The reason for this is that comparison operations on the machine give $L \leq \infty$ and $\infty \leq R$ for **any** FP numbers L and R. This fooled the FP test into believing the result had been correctly computed. When this anomaly was discovered, the not-a-number detector of section 4 was introduced into the FP test so that such fakery cannot happen so easily any more. In fact, ∞ has some strange properties on the Cray-1. Let x be any floating-point number or ∞. Then $\infty \leq x \leq \infty$. Yet none of the following are ever true: $x = \infty$, $\infty < x$ or $x < \infty$.

This example shows that `FPTST` should always be run on the final results with `CHEKUF` = .FALSE., and it should run correctly and *produce no error messages* (underflow, overflow, divide-check, etc.).

The Past — Double-Precision

Even with the above penalties, the double-precision arithmetic failed the test. This was due to a rather obvious bug in the comparison operators .eq. and .ne. . For example, given

$x = 2^{-1} + \cdots + 2^{-48}$

and

$y = 2^{-1} + \cdots + 2^{-95}$

we had that x .eq. y was true and that x .ne. y was false. The bug in the software was reported to Cray and repaired in the next system release. This had the effect of repairing a similar bug in the *complex* .eq. and .ne. comparison operators, since the same software was used for both multiword comparisons.

"Un-Rounded" Arithmetic

The Cray-1 has the ability, in single precision, to turn "rounding" off for multiplication. The Hardware Reference Manual for that machine states that "The effect of this error ("rounding") is at most a round up of bit 2^{-48} of the result." From that statement, one might conclude that when "rounding" is off, multiplication is "chopped." That is not the case.

When FPTSB was run with "rounding" off, it reported that $t = 33$ was correctly supported. When FPTSD was run with $t = 34$ and EPRINT = .TRUE., the first example of trouble was

```
failure for operation  *
 x      =       0200037777777777740000
 y      =       0400007777777777740000
 result =       0200037777777777677777
 l      =       0200037777777777700000
 r      =       0200037777777777740000
sign(x) = +, type(x) = 2, i(x) =   34, exponent(x) =  -8189
sign(y) = +, type(y) = 2, i(y) =   34, exponent(y) =      0
```

The first line and the last two lines of the above output say that for $x = +2^{-8189}\times(2^{-1} + \cdots + 2^{-34})$ and $y = +2^{0}\times(2^{-1} + \cdots + 2^{-34})$, which are Type 2 mantissas (see (2.2)), result = $fl(x\times y)$ is not in the correct containment interval [l,r]. For clarity, the values of x, y, result l and r are printed in octal (base 8). The last 16 octal digits give the mantissa and the leading octal digits give the sign and biased exponent. Thus, when "rounding" is turned off, multiplication sometimes produces a result which is smaller than the correct chopped result, by one bit in the last place. This example also shows that "rounding" does more than simply round the result to 48 bits. Without "rounding", some multiplication results are not even in the correct model interval, let alone "chopped" to the wrong end-point.

Appendix DEC

VAX 11/780

The VAX 11/780 has a base-2 FP hardware representation with an exponent range of -127 to $+127$, $t = 24$ for single-precision, and $t = 56$ for double-precision.

This machine has been thoroughly tested and the above parameters are correctly supported when run under the *UNIX* operating systems $32V$ and $4BSD$.

Although the 11/780 has correctly designed FP arithmetic, the machines don't necessarily stay correct. Our Murray Hill computing center has had considerable fun with FP intermittents in their zoo of a dozen or so 11/780's. The high point was when our statisticians spent a week looking for a bug in their code only to discover the problem was that the FP on the 11/780 was broken. The FP accelerator (FPA) had partially fallen out of the backplane. The DEC field-engineers ran their test and diagnosis (T&D) software on the FPA and declared that all was well. We knew better, and asked them to **remove** the FPA and repeat the T&D. Lo and behold, it ran just fine! This is just more proof of the old saw "T&D is the only thing that will run when nothing else will". This example is not given to pick on DEC, which has been most responsive to reports of FP problems in their hardware, see the 11/750 example below. The problem is generic in hardware, it needs continued maintenance. IBM, Interdata and others suffer from the same problems.

As a result of the week-long bug chase that ended at the backplane, the computing center installed a tastefully designed use of `FPTST` to be run at 3am **every day on every 11/780**. Since its installation it has found that FPA's have partially fallen out of the backplane on three occasions and sent that information to the system gurus. In each case, the system was repaired before the normal working shift began. The computing center folks and their users can sleep better knowing such tests are being run daily.

The author was duly pleased to see his test running on the central computing center's 11/780s but failed to apply the same alertness to our two local 11/780s. Sure enough, the same thing happened on May 24, 1984 to one of our local 11/780s. It cost a member of the staff a day to track the problem down to the FPA. The `FPTST` now runs daily on our local 11/780s, and two of our 11/750s, as well.

VAX 11/750

The VAX 11/750 has a base-2 FP hardware representation with an exponent range of -127 to $+127$, $t = 24$ for single-precision, and $t = 56$ for double-precision.

The above parameters are correctly supported.

The Past.

The above parameters were correctly supported for all floating-point operators except addition and subtraction in double-precision, when tested under the *UNIX* operating systems $32V$ and $4BSD$.

In double-precision, addition and subtraction were accurate to only single-precision. Specifically, when $t = 56$ was tested, with `EPRINT` = `.TRUE.`, and (`NBEX`,`NBEY`,`NBIX`,`NBIY`) = $(1, 1, 1, 56)$, the following "hexadecimal" example was produced by `FPTSD`

```
Failure for operation  +
x      =        3F80      10000
y      =        BF80        200
Result =    FFFFB07F          0
L      =    FFFFB07F       8000
R      =    FFFFB07F       8000
Sign(x) = +, Type(X) = 1, I(X) =   56, Exponent(X) =     -1
Sign(y) = -, Type(Y) = 1, I(Y) =   31, Exponent(Y) =     -1
```

The first line and the last two lines of the above output say that for $x = +2^{-1} \times (2^{-1} + 2^{-56})$ and $y = -2^{-1} \times (2^{-1} + 2^{-31})$, which are Type 1 operands (see (2.2)), we have that Result = $fl(x + y)$ is not in the correct containment interval [L, R]. For clarity, the values of x, y, Result, L and R are printed in "hex" (base-16). The last two hex digits of the first word in the representation give the most significant digits in the 56 bit mantissa (the leading 1 in the mantissa is implicit), the fifth and sixth hex digits give the sign and biased exponent, the first four hex digits of the first word give more of the mantissa, the last four hex digits of the second word give, in decreasing significance, even more of the mantissa, and the first four hex digits of the second word give, in decreasing significance, the rest of the mantissa. Leading 0's are suppressed in the hex output and the length of the representation of the words varies. The output shows that addition failed in the 25th bit.

When tested for $t = 54$, the machine passed the above test correctly. However, this was quite misleading since we knew that the precision of addition was no better than 24 bits. The test allows detection of the trouble with addition, but unless we keep that fact in mind, the test may also lull us into believing that addition is more accurate (54 bits) than it really is (24 bits). This is an example where testing "extra-precise" operands would be helpful in determining precision, see section 7.

The error was reported to DEC and the bug in the microcode for $+$ and $-$ was repaired.

System 20/60

The System 20/60 has a base-2 FP hardware representation with an exponent range of -128 to $+127$, $t = 27$ for single-precision, and $t = 62$ for double-precision. The first bit of the second 36 bit word is not used in double-precision, so $t = 62$ **not** 63.

This machine has been thoroughly tested by Prof. Stan Eisenstat of Yale and the above parameters are correctly supported when run under the TOPS 20 operating system.

DEC 10, Model 1091

This machine has a base-2 FP hardware representation with an exponent range of -128 to $+127$, $t = 27$ for single-precision, and $t = 62$ for double-precision. The first bit of the second 36 bit word is not used in double-precision, so $t = 62$ **not** 63.

This machine has been thoroughly tested by Greg Astfalk of AT&T Technologies Research Center and the above parameters are correctly supported when run under the TOPS 10 operating system, version 7.01v07.

Appendix Data General

Eagle

This machine has a base-16 hardware representation with an exponent range of -64 to $+63$, $t = 6$ in single-precision, and $t = 14$ in double-precision.

This machine has been thoroughly tested, by Wolfi Fichtner of AT&T Bell Laboratories, and the above parameters are correctly supported.

Appendix Elxsi

This machine has a base-2 hardware representation with an exponent range of -125 to $+128$ and $t = 24$ in single-precision, in double-precision the exponent range is -1021 to $+1024$ with $t = 53$. Thus, the machine implements the IEEE Floating-point standard for single and double precision.

This machine has been thoroughly tested by John Walden and the above parameters are correctly supported.

The Past

In single precision, comparison of 0 with 0 was incorrect. That is, things like "x .lt. y" were ".true." when $x = y = 0$.

164

The 164 has a base-2 FP hardware representation with an exponent range of -1021 to $+1023$, with $t = 52$ for single-precision. There is no double-precision on this array processor.

This machine has been thoroughly tested by Prof. Bill Gropp of Yale and the above parameters are correctly supported.

Appendix Gould

NP 1

This machine has a base-16 hardware representation with an exponent range of -64 to $+63$, $t = 6$ in single-precision, and $t = 14$ in double-precision.

This machine has been thoroughly tested by John Walden and the above parameters are correctly supported.

The Past

An early version of this machine failed for the division of small numbers:

$$fl\ (\ (\ 16^{-64} \times 16^{-1}\ )\ /\ (\ 16^{-64} \times (\ 16^{-1} + 16^{-2}\ )\ )\ ) \approx 0.03$$

instead of $16/17 \approx 0.94$.

Appendix Honeywell

6080N

This machine has a base-2 hardware FP representation with an exponent range of −128 to +127, $t = 27$ in single-precision, and $t = 63$ in double-precision. The test was run under the SR-JS1 operating system using the unoptimized fortran compiler.

The smallest positive number $x = 2^{-128} \times 0.1_2$ does not have a representable negative because floating-point numbers are represented in 2’s complement. Thus, $e_{\min} = -127$, rather than -128, must be used.

This machine has been thoroughly tested, and with the above penalty on $e_{\min}$ the model is correctly supported.

The Past.

Before System Release J, when underflow was not tested (CHEKUF = .FALSE.), this machine passed the FP test with the minor penalty on $e_{\min}$ noted above. However, in single-precision, when underflow was tested, a gross error in the operating system underflow trap-handler was uncovered. FP exponent wrap-around could occur and FP underflow could result in huge numbers. Any operation whose mathematical result was $-\ 2^{-129}$, an unrepresentable FP number, could result in a monstrously large computed result.

The FP test detected this for both the × and / operators. Take, for example, a Type 1 mantissa, with $i = 27, x = 2^{-127}\ (\ 2^{-1} + 2^{-27}\ )$ and a Type 2 mantissa, with $i = 27, y = -2^{-1}\ (\ 2^{-1} + \cdots + 2^{-27}\ )$. Then $x \times y = -\ 2^{-128}\ (\ 2^{-1} + 2^{-29} + \cdots\ )$. To 27 bits, this result is $-\ 2^{-129}$. Yet, the computed result was $-\ 2^{+127}$, which is off by a factor of 2^{+256}.

Another example found was for any Type 1 (see (2.2)) mantissa m

$$fl\ (\ -\left\{\ 2^{-2}\ m\ \right\} / \left\{\ 2^{+127}\ m\ \right\}\ ) = -\ 2^{+127}$$

just as in the previous case of $x \times y$.

The above errors were reported to Honeywell and the trap-handler has been repaired in System Release J.

The above situation shows that **any** anomaly may imply real trouble. FPTSB reported that $e_{\min} = -127$ was correctly supported when CHEKUF = .FALSE., and the report that multiplication and division fail when CHEKUF = .TRUE. could have been written-off as "small errors in small numbers." **Any** anomaly may imply serious trouble, which further runs can define. In this case, with CHEKUF = .TRUE. and EPRINT = .TRUE. in FPTSD, the above examples of gross trouble would be printed out and the user could see that the errors were not "small."

Appendix HP

9000/210

The 9000/210 has a base-2 FP hardware representation with an exponent range of -125 to $+128$, $t = 24$ for single-precision, and $t = 56$ for double-precision.

This machine has been probed by Rick Becker. $e_{\mathrm{min}} = -102$ and $e_{\mathrm{max}} = 127$ are correctly supported. The penalty on e_{min} is due to comparisons being done by subtraction followed by compare to zero. Thus, small, close but not equal numbers may compare equal through underflow. The largest exponent is simply not handled correctly. These problems have been reported to HP and they are being fixed.

9000/720

The 9000/720 has a base-2 FP hardware representation with an exponent range of -125 to $+128$, $t = 24$ for single-precision, and $t = 56$ for double-precision. Paul Layman tested this machine and found the parameters correctly supported.

9836

The 99836 has a base-2 FP representation with an exponent range of -125 to $+128$, $t = 24$ for single-precision, and $t = 56$ for double-precision.

The current situation for this machine is the same as for the 9000/210 above. It too is being repaired.

Appendix IBM

370/168, 3032, 3081, 4341-1

These machines have a base-16 hardware representation with an exponent range of -64 to $+63$, $t = 6$ in single-precision, and $t = 14$ in double-precision.

These machines have been thoroughly tested, and the above parameters are correctly supported.

Intermittents

Intermittent errors are common and IBM is no exception. During testing, our 3032 had precisely one intermittent fault detected. The test reported that a containment interval $[L, R]$ for one of the results had $L > R$. This is not only a logical impossibility, but it has never happened before or since on that machine. The owner of this busy machine was concerned enough to run the test 15 minutes a day for two months looking for the bug to pop up again, to no avail.

PC/XT

This machine has a base-2 hardware representation with an exponent range of -125 to $+128$ and $t = 24$ in single-precision, in double-precision the exponent range is -1021 to $+1024$ with $t = 53$. Thus, the machine implements the IEEE Floating-point standard for single and double precision.

This machine has been thoroughly tested by Leonilda Farrow of Bell Central Research, and the above parameters are correctly supported.

RS6000

This machine has a base-2 hardware representation with an exponent range of -125 to $+128$ and $t = 24$ in single-precision, in double-precision the exponent range is -1021 to $+1024$ with $t = 53$. Thus, the machine implements the IEEE Floating-point standard for single and double precision.

This machine has been thoroughly tested and the above parameters are correctly supported.

The Past

Due to compiler and micro-code errors, the early versions of these machines had some strange behavior. For example, in double precision

$$fl\left(\sum_{i=1}^{53} 2^{-i} \right) > 1.$$

Appendix ICL

2900

This machine has a base-16 hardware representation with an exponent range of -64 to $+63$, $t = 6$ in single-precision, and $t = 14$ in double-precision.

This machine has been thoroughly tested by Brian Wichmann of NPL in England. The above parameters are correctly supported with the exception that the machine suffers from terminal exponent wraparound. The smallest exponent in the machine is -64 and if any computed result should have an exponent less than that it becomes positive. For example, $fl\ (16^{-65} \ / \ 16^{13} \ ) = 16^{+50} \approx 1.6\ 10^{+60}$.

Appendix Intel

8087

This machine has a base-2 hardware representation with an exponent range of -125 to $+128$ and $t = 24$ in single-precision, in double-precision the exponent range is -1021 to $+1024$ with $t = 53$. Thus, the machine implements the IEEE Floating-point standard for single and double precision.

This machine has been thoroughly tested by Wayne Fullerton of IMSL, and the above parameters are correctly supported.

8/32

This machine has a base-16 hardware FP representation with an exponent range of -64 to $+63$, $t = 6$ in single-precision and $t = 14$ in double-precision.

This machine has been thoroughly tested and the above parameters are correctly supported under the manufacturers operating system and complier.

The Past.

This machine had a very serious design error

$$fl\,(x - 16^{-64}\,x) \equiv 0$$

for *any* number *x*. Thus, for example,

$$fl(\,1 - 16^{-64}\,) = 0\ !$$

This error was detected by `FPTSB` and reported to Interdata, which had been purchased in the meantime by Perkin-Elmer. Perkin-Elmer immediately identified the error, in the division micro-code, and re-designed their newer models as a result.

Perkin-Elmer ran the test software in both single and double precision, under their own operating system, pretending that the machine was base-2. In single-precision, the machine tested correctly with $b = 2$, $t = 21$, $e_{\min} = -256$ and $e_{\max} = +252$. This is consistent with a base-16 machine with 6 hex digits. In double-precision, the exponents were the same, and $t = 53$. This is consistent with a base-16 machine with 14 hex digits.

The repaired machine is correct to all the digits the manufacturer claims.

Local Intermittent Errors

Fortunately for the floating-point test software, our Bell Laboratories in-house machine displayed other anomalous FP behavior not found in any other 8/32's, old or new. Specifically, division lost accuracy, and the amount varied from day to day, month to month. This shows the ability of the test to detect and provide evidence of intermittent floating-point errors. As a result of this discovery, we corrected temperature control problems and expanded our program of machine repair and maintenance. We then gave the machine away.

The test was run on an in-house 8/32 that used the *UNIX* 6.0 operating system. The `FORTRAN` compiler on this system only supports double-precision. The program can say `REAL`, but the compiler would compile double-precision. So only double-precision could be checked in-house. We continued to model the 8/32 as a t_2 bit base-2 machine, even though it was base-16, because the number of bits correctly supported was not reflected by the number of hex digits t_{16} supported. That is, $t_2 \neq 4\,t_{16} - 3$. The final bad news on $/$ was that it was only good to 39 bits, that is, for $b = 2$ only $t = 39$, rather than $t = 53$, was correctly supported. What happened to the 14 missing bits? To see what the problem was, `FPTSD` was run with T = 40. `FPTSD` reported that T = 40 was correctly supported, provided that division was implemented as a composite operation. However, division on the 8/32 is **not** a composite operation. Thus, the above warning about division is serious. To see what the problem was, the above run was repeated with `EXPAND` = 0 and `EPRINT` = `.TRUE.`. This doesn't allow any Expanding of containment intervals for $/$. The printout gave the following example.

```
Failure for operation  /
x      =    40800000      40000
y      =    40200000       4000
Result =    41400000       ffff
L      =    41400000      10000
R      =    41400000      18000
Sign(x) = +, Type(X) = 1, I(X) =   38, Exponent(X) =      0
Sign(y) = +, Type(Y) = 1, I(Y) =   40, Exponent(Y) =     -2
```

The first line and last two lines of the above output shows that for $x = + 2^0(2^{-1} + 2^{-38})$ and $y = + 2^{-2} (2^{-1} + 2^{-40})$ we have that Result = $fl(x / y)$ was not in the correct containment interval [L, R]. For clarity, the values of x, y, Result, L and R are printed in "hex" (base-16). The last 14 hex digits are the mantissa and leading hex digits are the sign and biased exponent. Leading 0’s are suppressed in the hex output and the length of the second word in the representation of the FP numbers varies. The error was very small, in the last bit. However, this division was done correctly on other 8/32’s, both old and new.

When the user is told that the test has been passed with the assumption that division is a composite operator, there are two possible responses. If the user is a physicist trying to get some work done, he can be happy and go on with his work. However, if the user is a mainframe designer or maintenance engineer, who knows that division was not intended to be composite, the response must be to see if the message means digits are being dropped.

Appendix MIPS

Mips 2000

These machines have a base-2 hardware representation with an exponent range of -125 to $+128$ and $t = 24$ in single-precision, in double-precision the exponent range is -1021 to $+1024$ with $t = 53$. Thus, they implement the IEEE Floating-point standard for single and double precision.

These machines have been thoroughly tested and the above parameters are correctly supported.

Appendix Miscellaneous Machines

This appendix is a compilation of tidbits about various machines. The test has not been run on these machines but the errors discussed would have been found if the test had been run on them. The errors are too good to pass up discussing.

Ancient History

The first two examples are from "old" machines.

Atlas-1

David Wheeler of Cambridge notes that the Atlas-1 made by Ferranti in 195? had the last bit of its FP result register stuck at 1, permanently.

HP-45 Hand Calculator

A somewhat offbeat example of the utility of (2.2) in detecting errors is the now famous wiring error on the early models of the HP-45 hand calculator. That error resulted in log ($\exp(1.01)$) not even being close to 1.01. Since the HP-45 is a base 10 machine, 1.01 is one of the numbers which would be used from (2.2). log and exp are hard-wired processes and $\log(\exp(x)) \approx x$ would be a logical relation to test. In fact, $\log(\exp(x)) \approx x$ failed for **all** x's with type 1 mantissas from (2.2), not just 1.01.

The Present

Now for some more modern machines.

Apple II Plus and Apple IIe

This example is from Prof. Kendall Atkinson of the University of Iowa and was also reported in the SIGNUM Newsletter, volume 18, 1983, page 2. The base is 2 and the mantissas of these machines are 4 bytes (32 bits) long. That is, $t = 32$. Let $z = 0.5 + 2^{-25}$. We would expect that $w = 2*(z/2)$ should give $w = z$. However, we get $w = 0.5 + 2^{-26}$. Thus, the best supported value of t is 24, quite a bit different from 32.

A C language Vax 11/780 FP Simulator

Andy Koenig made a simulator for Vax 11/780 FP arithmetic in the C programming language. The intent was to provide a familiar FP to micro-computers with the only assumption being that 16 bit integer arithmetic is correctly supported. He wanted to test his simulator.

This was done by changing 12 lines of code in the FPTST software to call the C simulation routines to do the FP operations. That is, statements like "z = x + y" were changed to read "call add(x,y,z)". The test was run on a Vax 11/780, which has the identical FP representation. So the Vax was used to check the C simulator. Two minor errors were found. First, the sign bit was lost when adding 0 to a negative number. Second, underflow was checked against an exponent threshold that was one too small.

Prime 400/550.

This machine has a base-2 hardware representation with $t = 47$ and $e_{\max} = 32639$ in single-precision. The problems discussed below were reported by E. H. Stafford at Georgia Tech in "A Report on the Accuracy of Prime Computer's Floating Point Hardware and Software", Technical Report GIT-ICS-83/09, 1983.

Whenever a number x with the smallest possible exponent is multiplied by a number greater than 0.5 an overflow occurs, thus, $1 \times x$ overflows. Also, multiplication always sets the last two bits of the result to zero. Thus, 1 is not the multiplicative identity unless $t = 45$ is used.

Telefunken TR 440

– Miscellaneous Machines-2 –

This is a bizarre base-16 machine with 9.5 base-16 digits in the mantissa and a base-16 exponent range of – 64 to 63. It was brought to my attention by Thomas Haarmann of the University of Osnabruck. The problem here is how to represent 9.5 hex digits in the model used in the test. Clearly, using $b = 16$ is out of the question. However, by using $b = 2$ the arithmetic can be modeled with $t = 35$. The 3 bits lost are due to the "wobbling" precision of hex arithmetic.

Appendix Multiflow

This machine has a base-2 hardware representation with an exponent range of -125 to $+128$ and $t = 24$ in single-precision, in double-precision the exponent range is -1021 to $+1024$ with $t = 53$. Thus, the machine implements the IEEE Floating-point standard for single and double precision.

The Past

An early version of this machine was thoroughly tested by John Walden and errors were found. In double precision,

$$fl\ (\ (\ 2^{-1021} \times \sum_{i=1}^{53} 2^{-i}\ )\ /\ (\ 2^{1} \times 2^{-1}\ )\ )\ =\ 0$$

instead of just returning the first argument.

Appendix Perkin-Elmer

3320.

The Perkin-Elmer 3220 has a base-16 hardware FP representation with an exponent range from -64 to $+63$, $t = 6$ in single-precision, and $t = 14$ in double-precision.

This machine has been thoroughly tested, and the above parameters are correctly supported when run under the manufacturer’s operating system and compiler.

Appendix Pyramid

This machine has a base-2 hardware representation with an exponent range of -125 to $+128$ and $t = 24$ in single-precision, in double-precision the exponent range is -1021 to $+1024$ with $t = 53$. Thus, the machine implements the IEEE Floating-point standard for single and double precision.

This machine has been thoroughly tested and the above parameters appear to be correctly supported by the hardware, but the machine gives non-reproducable results. That is, the errors reported by the test are not repeatable from one run to the next. Furthermore, Stu Feldman of Bell Central Research has determined that a small `FORTRAN` program can get random results as well. The current conjecture is that the problem is due to internal FP registers not being saved on context switches or page faults.

The Past

This machine had a serious design error

$$fl\left(2^{-54} \sum_{i=1}^{53} 2^{-i} - 2^{-54} \right) \equiv -2^{-85} \left(2^{-1} + 2^{-22} \right)$$

The correct result is -2^{-107} and the computed result is off by about 2^{+22}.

Appendix Sequent

Balance 8000

This machine has a base-2 hardware representation with an exponent range of -125 to $+128$ and $t = 24$ in single-precision, in double-precision the exponent range is -1021 to $+1024$ with $t = 53$. Thus, the machine implements the IEEE Floating-point standard for single and double precision.

This machine has been thoroughly tested by Ken Thompson, in double precision, and the above parameters are correctly supported.

Appendix Silicon Graphics

These machines have a base-2 hardware representation with an exponent range of -125 to $+128$ and $t = 24$ in single-precision, in double-precision the exponent range is -1021 to $+1024$ with $t = 53$. Thus, they implement the IEEE Floating-point standard for single and double precision.

These machines have been thoroughly tested and the above parameters are correctly supported.

The Past

Early verions of the Iris 2400 Turbo using the Weitek chip had problems in single precision, causing things like

$$fl\,(\,(\,2^{-125} \times 2^{-1}\,) \times 1\,) \equiv 0.$$

Appendix Solbourne

This machine has a base-2 hardware representation with an exponent range of -125 to $+128$ and $t = 24$ in single-precision, in double-precision the exponent range is -1021 to $+1024$ with $t = 53$. Thus, the machine implements the IEEE Floating-point standard for single and double precision.

This machine has been thoroughly tested by John Walden and the above parameters are correctly supported.

Appendix Sun

Sun 3 and Sun 4/280

These machines have a base-2 hardware representation with an exponent range of -125 to $+128$ and $t = 24$ in single-precision, in double-precision the exponent range is -1021 to $+1024$ with $t = 53$. Thus, they implement the IEEE Floating-point standard for single and double precision.

These machines have been thoroughly tested by John Walden and the above parameters are correctly supported.

Appendix Tandy

2000

This machine has a base-2 hardware representation with an exponent range of -125 to $+128$ and $t = 24$ in single-precision, in double-precision the exponent range is -1021 to $+1024$ with $t = 53$. Thus, the machine implements the IEEE Floating-point standard for single and double precision.

This machine has been thoroughly tested by David Gay of AT& Bell Laboratories and the above parameters are correctly supported.

Appendix Univac

1100

This machine has a base-2 hardware FP representation with an exponent range of −128 to +127, $t = 27$ in single-precision, and $t = 63$ in double-precision.

The compiler used ("field-data") was so buggy that not much of the test could be run, but we got some information. The first fact we got was that 1 is not the multiplicative identity, that is, $fl\ (1 \times x) \neq x$ for some x. The last bit is lost in such operations, not a great loss. The next piece of news was much more interesting. There are numbers $x < 0$ and $y < 0$ such that $fl\ (x \times y) < 0$! The problem arises when x and y are chosen to be so small that $fl\ (x \times y)$ will underflow. The underflow trap then sets the sign of the result to be the same as one of the operands.

The test of this machine remains incomplete: the machine has been sold and we no longer have access to one.