PARANOIA . ADA :

A DIAGNOSTIC PROGRAM TO EVALUATE ADA FLOATING-POINT ARITHMETIC

May 12, 1985

Chris Hjermstad
Package_Architects, Inc.
8950 Villa La Jolla Drive
Suite 1200
La Jolla, California 92037
(619) 587-1815

F.3.4.1

INTRODUCTION

Programmers have traditionally approached floating-point arithmetic with great trepidation. Brown and Feldman in their landmark paper on model numbers call floating-point arithmetic the "*bete noire*" (black beast) of computing. Programmers are haunted by the suspicion that floating-point calculations harbor hidden errors. This resistance stems, at least partly, from the variety of inconsistent floating-point representations implemented over the years by different computer manufacturers.

In many respects, the programming language Ada* is not so much a breakthrough in technology as it is an evolutionary melding of many advancements achieved by computer science research during the 1970s. This is certainly the case with respect to Ada's treatment of floating-point arithmetic. Ada explicitly adheres to concepts of environmental inquiry initially proposed by Naur in 1967 and of model number parameterization advanced by Brown and Feldman in 1980 and formalized by Brown in 1981. Following these precepts, Ada encourages the development of safe, transportable numerical programs. This paper traces major historical efforts to establish effective standards for floating-point arithmetic. It describes previously developed programs written in languages such as FORTRAN and BASIC which partially undertake the testing of conformance to such standards. It provides results obtained from a contemporary program, Paranoia.Ada, which tests various aspects of floating-point arithmetic in the context of the Ada programming language.

SPECIFICATION OF FLOATING-POINT ARITHMETIC

The last two decades have witnessed efforts within the computer science community to establish floating-point arithmetic standards. These efforts have been primarily motivated by a desire to perform consistent arithmetic in a common transportable programming language across many different computing environments and hardware architectures. Naur, writing in 1967, introduced the concept of an "environmental inquiry" as a means of ascertaining the arithmetic characteristics of a computing environment. His ideas where incorporated into the ALGOL 68 language and are reflected in the "attribute" feature of Ada.

The International Federation for Information Processing (IFIP) Working Group 2.5 (Mathematical Software) introduced the concept of floating-point parameters as a means of determining the characteristics of a specific programming environment's floating-point arithmetic implementation. The design of FORTRAN 77 provided access to such floating-point parameters.[1]

---

*Ada is a registered trademark of the U.S. Government, AJPO (Ada Joint Program Office).

BROWN-FELDMAN CONTRIBUTIONS

More recently, Brown and Feldman, using model number theory, further specified floating-point parameterization. They defined a generalized standard representation of floating-point numbers independent of underlying machine architecture. Their landmark work resulted in precise definitions for floating-point arithmetic based on model numbers and model intervals. They established rigorous theorems concerning the dependability of computational results derived from operations that adhered to the basic model definitions.[2] They defined a standard model number representation as:

$x = b^e f$, where

b is the specified radix,

e is an integer exponent of specified range, and

f is the significand expressed as a base-b digit.

They identified seven model parameters as necessary to the specification of a floating-point arithmetic implementation. Four parameters consist of basic integer values:

BASE	b
PRECISION	p
MINIMUM EXPONENT	emin
MAXIMUM EXPONENT	emax

Three additional parameters consist of floating-point values derivable from the basic parameters:

MAXIMUM RELATIVE SPACING	Epsilon = b^{1-p}
SMALLEST POSITIVE NUMBER	Sigma = b^{emin-1}
LARGEST NUMBER	Lambda = $b^{emax}(1-b^{-p})$

CURRENT IEEE STANDARDIZATION EFFORTS

Two committees within the IEEE are working to further refine and extend the Brown-Feldman model of floating-point arithmetic. Committee P754 is developing a detailed specification to be applied to computers employing a binary representation. P854 is developing a compatible super-set specification that is both radix and word length independent. The objective of both committees is to establish additional environmental rules which will precisely define the outcome of all floating-point operations. Such rules are intended to eliminate all implementation-dependent or ambiguous circumstances with particular emphasis on consistent treatment of error conditions.[3]

As an example of this focus, both IEEE draft specifications require the implementation of at least the five following exception conditions:

- Invalid operation
- Division by zero
- Overflow
- Underflow
- Inexact result

ADA FLOATING-POINT PARAMETERIZATION

The design of floating-point arithmetic in the Ada programming language explicitly complies with the Brown-Feldman model. It requires a limited, conservative interpretation of the Brown-Feldman parameters. Ada assumes a binary representation and arbitrarily assigns values to the other parameters based on the elemental precision specification of DIGITS in a real object type definition. Although the minimal Ada model number parameter values frequently result in an artificially limited precision range, they do encourage portability, predictability and understandability.

Ada also allows for the specification of implementation-dependent "safe number" values. Such safe numbers permit additional latitude in the programming of numerically sophisticated procedures requiring greater exploitation of the complete underlying hardware architecture. A comparison between the Brown-Feldman parameters against Ada attributes relating to both the required model number values and implementation-dependent values shows a close mapping:

Brown-Feldman Parameter	Model Number Attribute	Implementation-Dependent Attribute
b	2 (BY DEFINITION)	T'MACHINE_RADIX
p	T'MANTISSA (FUNCTION OF T'DIGITS)	T'MACHINE_MANTISSA
emax	T'EMAX (4*T'MANTISSA)	T'SAFE_EMAX T'MACHINE_EMAX
emin	- T'EMAX (SYMMETRICAL RANGE)	T'MACHINE_EMIN
Epsilon	T'EPSILON (2.0**(1 - T'MANTISSA))	(Determined by Paranoia.Ada)
Sigma	T'SMALL (2.0**(-T'EMAX - 1))	T'SAFE_SMALL
Lambda	T'LARGE (2**T'EMAX * (1.0 - 2.0**(-T'MANTISSA)))	T'SAFE_LARGE

HISTORICAL FLOATING-POINT ARITHMETIC TESTS

A number of computer programs have been written in the last several years which evaluate the quality of floating-point arithmetic implementations. One such program is MACHAR written by Cody in 1979 and published in the classic reference, *Software Manual for the Elementary Functions.*[4] MACHAR, coded in FORTRAN 77, determines thirteen characteristics of a floating-point arithmetic implementation such as radix, precision, rounding phenomenon, underflow threshold and overflow threshold.

Another notable effort is the Arithmetic Unit Test Program developed by Schryer in 1979. Results from the execution of this program were reported in the seminal Brown and Feldman paper "Environmental Parameters and Basic Functions for Floating-Point Computation".[5] Schryer's test program was also coded in FORTRAN 77 and calculates the seven Brown-Feldman model parameters. The program was used to test Cray-1, IBM

370, DEC VAX, Honeywell 6000 and Interdata 8/32 computers in support of Brown and Feldman's research.

RECENT FLOATING-POINT ARITHMETIC TESTS

More recently, two members of the IEEE floating-point standardization committees have written programs that perform even more sophisticated evaluations of floating-point arithmetic implementations. Karpinsky's 1985 article, "Paranoia: A Floating-Point Benchmark" describes the program Paranoia written by University of California, Berkeley Professor W. M. Kahan.[6] The article includes both Pascal and BASIC source code listings of Guard, a subset version of the full Paranoia. Kahan's original Paranoia is written in BASIC for the IBM (Intel 8088/8087) Personal Computer. It has also been translated into FORTRAN, Pascal and "C" for execution on DEC VAX and Sun Microcomputer (Motorola 68000) architectures.[7]

ADA IMPLEMENTATION OF PARANOIA

In conjunction with its Ada evaluation activities, Package_Architects, Inc. has converted the original Paranoia program to Ada. This converted program is called Paranoia.Ada. Paranoia.Ada determines the floating-point characteristics of the hardware supporting an Ada implementation. It also evaluates the accuracy, precision and reliability of the basic, predefined Ada arithmetic operations. The program identifies errors in floating-point computations and provides a report summarizing the overall quality and acceptability of the floating-point computational capability.

Paranoia.Ada performs specific diagnostic tests related to the following aspects of floating-point arithmetic:

- Determination of correct mathematical operations on small integral values.
- Calculation of radix, precision and Epsilon parameters.
- Determination of normalization with respect to subtraction operations.
- Determination of guard digits on subtraction, multiplication and division operations.
- Determination of rounding phenomenon (e.g. chopped, rounded or rounded to even) on addition, subtraction, multiplication and division.
- Determination of commutative multiplication properties.
- Determination of underflow threshold values.

- Determination of rounding phenomenon on floating-point to integer conversion operations.
- Determination of overflow threshold values.
- Evaluation of integer power arithmetic.
- Evaluation of division by zero arithmetic.

Paranoia.Ada takes significant advantage of several advanced features of Ada. The program relies on the Ada exception feature to detect and respond to error conditions with less disruption to processing than occurs with conventional BASIC or Pascal mechanisms. The program has been architecturally redesigned into forty-six separately compiled units and consists of approximately twenty-five hundred semi-colon terminated Ada statements. The program is implemented as a generic and is instantiated through the specification of a DIGITS parameter or by reference to a predefined FLOAT_TYPE.

Because a number of validated Ada compilers do not provide the mathematical functions required by the Paranoia algorithms, Paranoia.Ada contains a partial mathematics library based on the Cody-Waite algorithms. The program can either use the mathematics library provided by the compiler being tested or use its own independent library for test calculations.

The program also includes a utility package called STOP_WATCH which provides timing data related to test execution. The program measures the amount of CPU time required to perform the floating-point diagnostic tests and the amount of time required to generate the resulting output report.

PARANOIA.ADA DIAGNOSTIC EVALUATIONS

Paranoia.Ada replicates the test algorithms implemented in the original BASIC language version and adheres to the evaluation criteria established by Professor Kahan. Paranoia.Ada classifies errors detected in the course of its diagnosis into four categories. Ranked according to increasing levels of severity, the error categories consist of flaws, defects, serious defects and failures. Examples of errors associated with each category are as follows:

Flaws: Comparison anomalies such as:

$X \neq -(-(X))$ or,

$X \neq Y$ but $X - Y = 0$.

Range imbalance between overflow threshold and underflow threshold.

Defects:

Comparison anomalies such as:
$Z^{**}I /= Z_1*Z_2*Z_3....*Z_I$.

Erroneously raised numeric errors.

An imbalance between the underflow threshold and Epsilon.

Multiplication and subtraction operations yield inconsistent underflow thresholds.

Serious Defects:

Absence of division by zero protection.

Absence of guard digits.

Underflow or overflow conditions not accompanied by corresponding numeric errors.

Failures:

Outright arithmetic errors such as:
$2 + 2 = 5$.

Non-normalized subtraction.

Erroneous guard digits.

Underflow to negative number.

Accuracy deterioration approaching underflow.

Paranoia.Ada maintains a record of the errors encountered in the course of its execution. In its summary report, the program generates an overall evaluation of the tested floating-point implementation. Using IEEE Standards P754 and P854 as criteria, the program rates the diagnosed arithmetic in terms of one of the following comments:

- The arithmetic diagnosed appears excellent.
- The arithmetic diagnosed seems satisfactory.
- The arithmetic diagnosed seems satisfactory though flawed.
- The arithmetic diagnosed may be acceptable despite inconvenient defects.
- The arithmetic diagnosed has unacceptable serious defects.
- A fatal failure may have spoiled this program's subsequent diagnoses.

EXECUTION OF PARANOIA.ADA AGAINST DEC ACS

Paranoia.Ada has been run extensively against the Digital Equipment Corporation (DEC) Ada Compilation System (ACS) hosted on a VAX 785 computer. The VAX architecture provides a rich and powerful floating-point arithmetic capability. The VAX supports four floating-point representations. These four representations are available through the Ada package SYSTEM pre-defined floating-point types F_FLOAT, D_FLOAT, G_FLOAT and H_FLOAT. F_FLOAT is a 32 bit representation, D_FLOAT and G_FLOAT are alternative 64 bit representations (selectable by a PRAGMA directive), and H_FLOAT is a 128 bit representation.

The DEC ACS also provides for three pre-defined floating-point types in package STANDARD. The compiler maps each of these types -- FLOAT, LONG_FLOAT, and LONG_LONG_FLOAT -- into the respective machine representation types F_FLOAT, D_FLOAT or G_FLOAT, and H_FLOAT. Paranoia.Ada has been run against all seven of these pre-defined types as well as a user-defined type of SYSTEM.MAX_DIGITS. SYSTEM.MAX_DIGITS forces the compiler to use the H_FLOAT representation. Sample output reports from D_FLOAT, G_FLOAT, H_FLOAT, and SYSTEM.MAX_DIGITS test runs are supplied as attachments.

DIAGNOSTIC ANALYSIS

Paranoia.Ada provides a consistent diagnosis of the eight tested floating-point representations. The values calculated by the Paranoia.Ada algorithms match the values reported by queries to corresponding Ada attributes. The program detects a similar set of errors on all eight representations as well. One flaw and one serious defect pertaining to underflow phenomena were discovered for each of the representations. The flaw involves an inconsistency between comparison results and arithmetic results with numerical values at or very close to the underflow threshold. The serious defect concerns the absence of a numeric error when subtraction operations on such small numbers result in underflow.

This specific circumstance is addressed by the IEEE standards. The DEC VAX implementation appears to result in an underflow to zero but without a numeric error being raised. The IEEE standards require that the underflow result be a non-normalized "tiny" number accompanied by an exception.

Paranoia.Ada uncovers a second serious defect in the D_FLOAT floating-point representation. In the VAX architecture, D_FLOAT representation is an extension of the single-precision F_FLOAT representation. (G_FLOAT is the true double-precision representation.) D_FLOAT has the same exponent range as F_FLOAT but uses an additional 32 bits of storage to allow greater precision in the significand. This allocation violates a requirement of the IEEE specification for balance between Epsilon and Sigma. In Paranoia.Ada terms, Epsilon equates to a calculated unit in the last place value and Sigma is the calculated underflow threshold value.

TIMING RESULTS

Execution and compilation timing data for each of the eight various DEC ACS floating-point representations are presented in Table 1. Execution times are also graphically depicted in Figure 1. (Since these data represent only a single sample for each type, caution is advised against drawing unjustified general conclusions.) Report generation times appear relatively consistent and provide a basis of comparison for the execution time differences. The execution times appear to increase as a function of the amount of precision provided by each type. Within the same precision, STANDARD pre-defined types seem to take longer to execute than SYSTEM pre-defined types.

Compilation times for the seven pre-defined types are also relatively constant. For these types, the DEC ACS compiles Paranoia.Ada at a rate of approximately six hundred statements per minute. The compiler generates the SYSTEM.MAX_DIGITS version of the program at a slightly slower rate.

SIGNIFICANCE OF RESULTS

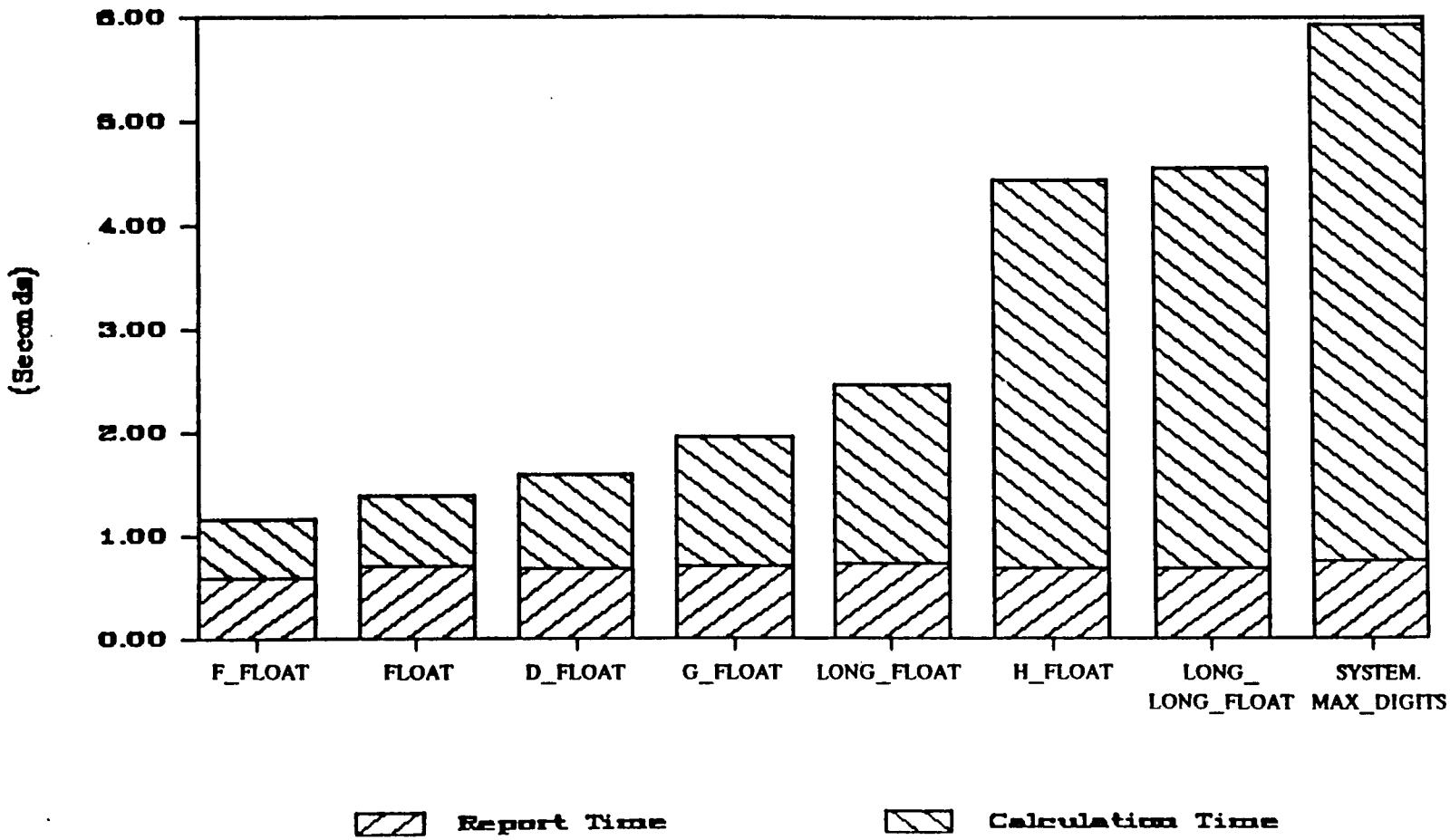
Owing to the sophistication of its diagnostic algorithms, Paranoia.Ada places heavy demands on the floating-point capabilities of an Ada compiler. The successful compilation and execution of a program as numerically complex and devious as Paranoia.Ada is a significant demonstration of a compiler's maturity, robustness and completeness. Paranoia.Ada is a practical exploration of Ada's floating-point capabilities. It tests the fidelity of an Ada implementation to the concept of model numbers, assesses the dependability of the arithmetic, and reveals Ada's suitability as an engine for further serious numerical computations. Paranoia.Ada, itself being a computationally intensive program, establishes the appropriateness of Ada as a medium for numerically demanding applications.

TABLE 1: Paranoia.Ada Compilation, Diagnostic Calculation and Report Generation Times

Type	F_FLOAT	FLOAT	D_FLOAT	G_FLOAT	LONG_FLOAT	H_FLOAT	LONG_ LONG_FLOAT	SYSTEM. MAX_DIGITS
Type'SIZE	32 Bits	32 Bits	64 Bits	64 Bits	64 Bits	128 Bits	128 Bits	128 Bits
Compilation Time	04:18.12	04:20.87	04:20.51	04:19.85	04:20.32	04:22.26	04:22.56	04:37.22
Diagnostic Calculation Time	00:00.57	00:00.68	00:00.93	00:01.26	00:1.72	00:03.75	00:03.86	00:04.42
Report Generation Time	00:00.60	00:00.71	00:00.68	00:00.71	00:00.74	00:00:69	00:00.70	00:00.71

Note: Times in minutes and seconds.

FIGURE 1: Paranoia.Ada Execution Time Comparison



CONCLUSION

Many essential software functions in the mission critical computer resource application domain depend on floating-point arithmetic. Numerically intensive functions associated with the Space Station project, such as ephemeris generation or the implementation of Kalman filters, are likely to employ the floating-point facilities of Ada. Paranoia.Ada appears to be a valuable program to insure that Ada environments and their underlying hardware exhibit the precision and correctness required to satisfy mission computational requirements.

As a diagnostic tool, Paranoia.Ada reveals many essential characteristics of an Ada floating-point implementation. Equipped with such knowledge, programmers need not tremble before the "black beast" of floating-point computation.

REFERENCES

1. W. J. Cody, "Floating-point Parameters, Models and Standards," in *The Relationship Between Numerical Computation and Programming Languages*, J.K. Reid, ed., North-Holland Publishing Co., Amsterdam, 1982, pp. 51-65.

2. W. S. Brown, "A Simple But Realistic Model of Floating-point Computation," *ACM Transactions on Mathematical Software*, Vol. 7, No. 4, December 1981, pp. 445-480.

3. W. J. Cody, W. Kahan, et. al. "A Proposed Radix- and Word-length-independent Standard for Floating-point Arithmetic," *IEEE Micro*, August 1984, pp. 86-100.

4. W. J. Cody and W. Waite, *Software Manual for the Elementary Functions*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1980.

5. W. S. Brown and S. I. Feldman, "Environmental Parameters and Basic Functions for Floating-point Computation", *ACM Transactions on Mathematical Software*, Vol. 6, No. 4, December 1980, pp. 510-523.

6. R. Karpinsky, "Paranoia: A Floating-Point Benchmark", *BYTE*, Vol. 10, No. 2, February 1985, pp. 223-235.

7. BASIC, FORTRAN, Pascal and "C" Paranoia source code is available from Mr. Richard Karpinsky, IEEE P854 Mailings, U-76, University of California, San Francisco, San Francisco, CA 94143.