Mathematics and Computer Science at Odds over Real Numbers

by Thomas J. Scott, Ph.D.

Computer Science Department
Western Illinois University
Macomb, IL 61455

ABSTRACT

This paper discusses the "real number" data type as implemented by "floating point" numbers. Floating point implementations and a theorem that characterizes their truncations are presented. A teachable floating point system is presented, chosen so that most problems can be worked out with paper and pencil. Then major differences between floating point number systems and the continuous real number system are presented. Important floating point formats are next discussed. Two examples derived from actual computing practice on mainframes, minicomputers, and PCs are presented. The paper concludes with a discussion of where floating point arithmetic should be taught in standard courses in the ACM curriculum.

FLOATING POINT BASICS

Floating point numbers are used to implement the abstract data type often called "real". In typical scientific notation, a number is of the form a * 10^b, where 1 <= a < 10. In typical floating point systems that use base-2, a number is of the form a * 2^b, where 0.5 <= a < 1.0.

A floating point number, in any system, consists of three parts, as follows:

S ee..e mm...m .

1) **The algebraic sign of the number (S).**
This sign is normally the left-most bit in the representation. Because of "**two's complement**" notation for negative numbers, 0 represents a positive number and 1 a negative number.

2) **The Exponent (ee...e)** . The exponent bits are normally in "**excess notation**", which allows signed exponents. This means that the exponent of 0 is coded as 10...0, the exponent +1 is coded as 10...01, the exponent -1 is coded as 01...1, 1...1 is the largest positive exponent, and 0...0 is the smallest negative exponent.

3) **The Mantissa (mm...m)**. The mantissa is represented in "**normalized**" form, which forces the left-most, or most significant, bit to be 1. In most representations, the implied decimal point is to the left of the mantissa.

Floating point formats use a fixed number of bits. Thus if a decimal fraction, such as 1/3, is represented as a floating point number, only as many bits as will fit in the mantissa are kept.

It is well known that all irrational numbers have infinite, non-repeating decimal expansions, and that

rational numbers have either terminating or infinitely repeating decimal expansions. When translated into floating point format, infinite or infinitely repeating decimal representations all suffer truncation. The **definitive answer** to the question, **"Which fractions lose accuracy?"** is given by the Theorem 1 and its Corollary.

Theorem 1: Suppose r is a positive rational number less than 1, and that r = a/b with a and b integers and the ratio a/b is in "lowest terms", i.e., the greatest common divisor of a and b is 1. Then the representation for r terminates in base c if and only if all the prime factors for b are also prime factors of c.

Proof. The representation for r is terminating in base-c if and only if r = a/b and for some n and d_1, ... , d_n, we have $a/b = 0.d_1 \ldots d_n = (d_1*c^{-1} + \ldots + d_n*c^{-n}) = (c^{n-1}*d_1 + \ldots + d_n) / c^n$. After reducing this expression, we have $a * c^n = b * (c^{n-1}*d_1 + \ldots + d_n)$. Since a and b are in lowest terms, they share no common prime factors. Thus the prime factors for b must be a subset of the prime factors for c.

Corollary 1.1: If a/b is a fraction in lowest terms, then the **base 2 representation** for a/b is terminating if and only if b is a power of 2. Furthermore, theoretically terminating binary representations suffer floating point truncation if the number of bits in the terminating block is greater than the number of mantissa bits.

The base 2 and base 10 representations for 1/3 are non-terminating, because 3 is prime and not a factor of either 2 or 10. The representation for 1/10 terminates in base 10. But since 5 is a factor of 10, 100, 1000, etc., Corollary 1.1 shows that the base-2 representations for 1/10, 1/100, 1/1000, etc., are non-terminating. Since typical computer calculations using real numbers are first translated from base-10 to base-2, then computed with, and then translated back to base-10, it should be clear that most real number arithmetic involves some truncation error.

THE WIF FLOATING POINT SYSTEM

A 16-bit floating point implementation, simply called the WIF system, will be used in this paper. **The reasons for defining the WIF system and its rules are threefold:** 1) The number of bits is small enough to allow hand calculations of most items, 2) Inconsistencies between floating point numbers and a continuous real number system are easily demonstrated with WIF numbers, and 3) Analogues of the results established for WIF numbers are available in **all** other floating point implementations. The WIF system's floating point format is **S eeeee mmmmm mmmmm**. The Sign bit of 0 means a positive number and 1 a negative number. The five exponent bits, **eeeee**, are in **"excess 16"** notation. The only available WIF exponents are thus:

00000 = -16,		00001 = -15,
00010 = -14,	... ,	01111 = -1,
10000 = 0,		10001 = + 1,
10010 = +2,	... ,	11111 = +15

The 10 mantissa bits, **mmmmm mmmmm**, are always normalized with the first bit 1. The WIF system represents the number 0 as 0 00000 00000 00000 or 1 00000 00000 00000. Floating point calculation in the WIF system obeys the following rules:

Rule 1: Each floating point number is represented with sign bit, a 5-bit, excess 16, base-2 exponent, and a

normalized 10-bit binary mantissa, with the binary point assumed to be to the left of the mantissa.

Rule 2: After any calculation is done, the WIF system will normalize the answer. Intermediate normalizations are only done when necessary.

Rule 3: Any excess bits generated in the mantissa of a compound calculation are truncated. Unlike IBM and other systems, the WIF system has no "guard bits" and does not attempt to round in the least significant bit position.

Rule 4: In floating point addition, the exponents of the two addends are made equal by right shifting the mantissa of the smaller addend. This process is normally called **aligning the exponents.**

Rule 5: Subtraction is achieved by first forming the two's complement of the mantissa of the subtrahend and then adding it as described in Rule 4.

Rule 6: Multiplication and Division are performed by normal rules.

Rule 7: **Overflow** is the condition where a number to be expressed in WIF format is larger than the largest possible WIF number. The WIF system forces the processor's overflow flag to be set when an overflow condition occurs, and control is then passed back to the Operating System.

Rule 8: **Underflow** occurs when a positive number is smaller than the smallest positive WIF number, or a negative number is larger than the largest expressible WIF negative number. The WIF system uses the "old fashioned" method of setting the exponent and the mantissa to zero for both of these conditions and continuing the calculation.

Because the WIF system uses 16-bits, it only generates 2^{16} different numeric representations, each of which is called a **machine number.** Each WIF machine number, except the largest, has a unique **successor,** which is the smallest WIF machine number larger than the given WIF number. For example, the successor to the WIF machine number **0 10001 10000 00000** is **0 10001 10000 00001.** Similarly, each WIF machine number, except the smallest, has a unique **predecessor,** which is the largest of the WIF numbers smaller than the original number. The predecessor to **0 10001 10000 00000** is **0 10000 11111 11111.**

The **precision** of the WIF system is defined to be the distance between a WIF machine number and its predecessor. Because the first bit in a normalized mantissa must be 1, for each exponent, there are exactly 2^9 = 512 mantissas. Thus there are 512 WIF numbers between 2048 (2^{11} or 2K) and 4096 (4K), each having exponent 11100, and 512 WIF numbers between 0.25 and 0.5, each having exponent 01111. Thus WIF machine numbers are unequally spread out; and the WIF precision, like all other floating point systems, varies with the exponent.

Some facts about the WIF system are:

1) The largest WIF machine number, BIGWIF, is 0 11111 11111 11111 = $.11111\ 11111_2 * 2^{15}$ = $(.11111\ 11111_2 * 2^{10}) * 2^5 = (2^{10} - 1) * 2^5 = 2^{15} - 2^5 = 32768 - 32 = 32736$. The predecessor to BIGWIF is 0 11111 11111 11110, which is 32 smaller than its successor. The WIF precision at exponent 11111, which is equivalent to

15 in decimal is 2^5. An attempt to use a number larger than Abs(BIGWIF) (the absolute value of BIGWIF) results in overflow, and causes a program to abort.

2)The smallest positive WIF machine number, LILWIF, is **0 00000 10000 00000**, which is $0.1_2 * 2^{-16}$, which equals 2^{-17} = 0.00000 762939 453125. Similarly, the largest negative WIF number is -LILWIF. The successor to LILWIF is **0 00000 10000 00001**. The distance between these two numbers is $.00000\ 00001_2 * 2^{-16} = 2^{-10} * 2^{-16} = 2^{-26}$, which is not representable as a normalized WIF machine number. An attempt to use a number smaller than Abs(LILWIF) results in underflow, which causes this number to be set to 0. Thus the precision at LILWIF, or exponent 00000, is theoretically 2^{-26}, but this is not representable.

3) The number 1 is represented by **0 10001 10000 00000.** The predecessor to 1, P, is **0 10000 11111 11111**, and the successor to 1, S, is **0 10001 10000 00001.** Using the WIF calculation rules for exponent alignment, we have 1-P = 2^{-9} = S-1. The WIF precision at exponent 10001 is 2^{-9}. By a similar argument, we can show that the WIF precision at exponent 10000 is 2^{-10}.

The previous examples show that the theoretical precision at a WIF number is 10 less than its normal exponent. Since WIF uses an **excess 16 exponent**, the precision is 26 less than the WIF encoded exponent. We thus have the following theorem, which completely describes the precision for the WIF system. This theorem could be generalized to apply for any other floating point system.

THEOREM 2: For any WIF floating point number, the theoretical precision is 2^x, where x = (WIF Normalized Exponent - 26). Furthermore, for a given exponent, the precision increases by one power of two at the 11111 11111 mantissa. The precision of a WIF calculation is determined by the largest exponent, not the smallest.

All floating point implementations are finite, and hence, discrete, and are never exact representations of the infinitely dense real number line.

WIF FLOATING POINT ANOMALIES

The truncations and the variable precision are responsible for other systemic shortcomings, or "**non-properties**" in the WIF system. Five of these systemic failures are collected and proven in the following theorem.

THEOREM 3: The WIF system has the following characteristics.

1) Addition of WIF numbers is not always associative, i.e., there exist WIF numbers A, B, and C such that (A + B)+C <> A+(B + C).

2) Multiplication of WIF numbers is not always associative, i.e., there exist WIF numbers A, B, and C such that (A * B) * C <> A * (B * C).

3) Multiplication does not necessarily distribute over addition, i.e., there exist WIF numbers A, B, and C such that A * (B + C) <> A * B + A * C.

4) Ordering of operations is significant, i.e., there exist WIF numbers A, B, C, D, E, and F such that A-B+C <> A+C-B, and (D * E) / F <>(D / F)*E.

5) The **cancellation property** is not always valid for WIF numbers, i.e., there exist **positive numbers** A, B, and C such that A + B = A + C and B <> C.

Proof. To establish part 1, we let A = 128 = 2^7, B = C = 0.125 = 2^{-3}. The WIF representation for A is $2^7 = 2^8 * 2^{-1}$ = **0 11000 10000 00000**, and for B is $2^{-3} = 2^{-2} * 2^{-1}$ = **0 01110 10000 00000**. To calculate A + B, we use WIF Rule 4 to adjust B's exponent. Since A's exponent (8) is 10 more than B's exponent (-2), we must right shift B's mantissa 10 times to adjust the exponents. Right shifting any WIF mantissa 10 times gives a 0 mantissa. Thus A + B = A. In fact, if we calculate the expression of n + 1 terms, A + B + ... + B, by left association, the result will be A. Thus, we have (A + B) + B = A. Associating from the right, we must now add A + (B + B). Using WIF arithmetic rules, we have B + B = **0 01111 10000 00000**. We must now add **0 11000 10000 00000** and **0 01111 10000 00000**. The smaller exponent (01111) only differs from the larger exponent (11000) by 9. Thus we add **0 11000 10000 00000** and **0 11000 00000 00001**. The result is **0 11000 10000 00001**, which equals 128.25, which is not A.

To establish part 2, we let A = 10K or $10*2^{10}$, B = 4K or $4*2^{10}$, and let C = 0.5. From WIF Fact 1, BIGWIF is 32K -32. The WIF product, A*B, causes a positive overflow condition, and hence cannot be computed. Thus the association, (A*B)*C, causes overflow and cannot be computed. But since (B*C) = 2K, then A*(B*C) can be computed in WIF format and equals 20K.

To establish part 3, we let A = 8, B = 128, and C = 0.125. As shown above, because of the shifting required to align C's exponent, B + C = B. Thus A * (B + C) = A * B. But A * C = 1, so A * B + A * C <> A * (B + C). To establish part 4, we let A = 10K, B = 4K, and C = 2. Then (A*B)/C causes overflow and cannot be computed, while (A/C)*B does not. Finally, to establish part 5, let A = 128, B = 0.125, and C = 0.0625. From the discussion in the proof of Part 1, we have A + B = A. Since 0 < C < B, we also have A + C = A. Thus A + B = A + C, but B <> C.

Additional examples for each of these "**non-properties**" exist, especially those that cause underflow and result in unequal calculations.

Each WIF machine number represents the "neighborhood" of real numbers between its predecessor and its successor. This neighborhood always contains infinitely many real numbers, and is the basis for many mathematical notions, such as "the average of A and B", "between any two numbers is another number", "as close as you please", and "converging to a limit". Because of the variable precision and "non-properties" of floating point implementations, implementations of many algorithms are suspect.

WIF FLOATING POINT EXERCISES

The original goal for this paper was to expose potential computational problems occurring with floating point systems by providing a vehicle where the problems could be easily computed. As a result, the following series of paper and pencil exercises, designed to illustrate floating point nuances, is suggested.

1. Find the WIF machine number that represents each of the following decimal numbers: 1025, 1.75, 0.0625, and -555.005.
2. Calculate the WIF sum of 1025 and 0.625. What is the largest WIF machine number A, such that 1025 + A = 1025. What is the largest WIF machine number B such that 0.625 + B = 0.625. What neighborhoods of real numbers does WIF represent with its machine numbers 1025 and 0.625?
3. Determine the boundary values for the 7 floating point regions of the WIF real number line, as suggested above.
4. Find other, dis-similar examples to establish each part of WIF Theorem 3.
5. Consider other methods of handling overflow. What would happen in the overflow conditions if the calculations just proceeded?
6. Consider other methods of handling underflow. What if the exponent were simply set to 00000 and the mantissa left unchanged?
7. Consider another floating point system, called PUF, that uses the same rules as WIF, but uses 5-bits for **excess - 8** base-16 exponents, and 12-bits for normalized, **base-16** mantissas. The first hex digit must be a normalized, so the first four bits of a PUF number could be 0001. Also, shifting for normalization is four bits at a time.

Answer the following PUF questions.

a. Redo Exercises 1-6 for the PUF system.
b. Formulate and prove the PUF analogue of Theorem 2.
c. Find PUF machine numbers that establish all parts of Theorem 3.
8. Formulate and prove the generalized version of Theorem 2.

VENDOR IMPLEMENTATIONS : IBM 370

IBM 370 systems implement floating point in hardware which consists of four 64-bit floating point registers, hardware supported decimal alignments, and 51 machine instructions [IBMP72]. Three different floating point forms, each using **base 16** exponents and **base 16** mantissas are provided:

Data Type	Exponent Bits	Fraction Bits	Range of Accuracy
Short	7	24	$0.54*10^{-78}$ to $0.72*10^{75}$
	Decimal	Precision	7 Digits
Long	7	56	$0.54*10^{-78}$ to $0.72*10^{75}$
	Decimal	Precision	15 Digits
Extended	7	112	$0.54*10^{-78}$ to $0.72*10^{75}$
	Decimal	Precision	28 Digits

The range of magnitudes is the same in all formats. There are 22 instructions for Short Format values, 22 for Long Format values, and 7 for Extended format values [KUDL83]. There are both normalized and unnormalized adds and subtracts for short and long format values. The Extended Format instructions contain Adds, Subtracts, Multiplies, and Rounds, but no Load, Store, Divide, or Conversion Functions.

VENDOR IMPLEMENTATIONS :

DIGITAL VAX

The VAX standards use a **base 2** exponent and a normalized **base 2** mantissa, with an implied, but not stored leading 1, thus gaining one more bit of precision in the mantissa [LEVY89]. There are four forms, and,

in any form, the number 0 is represented by an exponent of 0 and a sign bit of zero. The four forms are

Data Type		Exponent Size	Fraction Size	Range of Accuracy
F	8 Decimal	23 Precision	$.29*10^{-38}$ to 7 Digits	$1.7*10^{38}$
D	8 Decimal	55 Precision	$.29*10^{-38}$ to 16 Digits	$1.7*10^{38}$
G	11 Decimal	52 Precision	$.56*10^{-308}$ to 15 Digits	$.9*10^{308}$
H	15 Decimal	112 Precision	$.84*10^{-4932}$ to 33 Digits	$.59*10^{4932}$

VAX machines provide full and complete hardware support for Arithmetic and Transfer instructions operations for all four floating point forms, as well as the famous POLY instruction in all forms [LEVY89].

IEEE FLOATING POINT STANDARD 754

By 1985, over 20 different vendor floating point implementations were in use. After 8 years work, the IEEE floating point standard 754 [IEEE85] was introduced. This standard is considered an improvement over previous formats and includes 3 forms:

a. **Short Format**, or "Single Precision", which uses 32-bits, similar to VAX form F.

b. **Long Format**, or "Double Precision", which uses 64-bits, similar to VAX form G.

c. **Extended Format**, which uses 80-bits: The Extended Format is intended primarily for use with arithmetic co-processors.

A major difference is the movement of the implied decimal point, so that the number is 1.mantissa * 2^{exp} , instead of 0.mantissa * 2^{exp}. To note the difference, the name **Significand** is used.

To more accurately handle overflow and underflow, five types of floating point numbers were defined:

Name	Sign	Exponent	Mantissa
Normalized	Any	0<Exp<MaxExp	Any
Denormalized	Any	0.....0	Nonzero
Zero	Any	0.....0	0....0
Infinity	Any	1.....1	0....0
Not a Number	Any	1.....1	Nonzero

Because of the number of existing, and often unchangeable, floating point implementations, the IEEE standard has produced much controversy. An excellent analysis of the issues is given by Flynn and Waser [FLYN82]. The strength of the standard is in the extra definitions, and the real benefit is to new hardware implementors, for whom an excellent, time-proven computational roadmap is provided.

FLOATING POINT ODDITIES

This section contains two floating point examples, each derived from actual classroom experiences. **The first example is a short FORTRAN program,** that was first run on an IBM 370. Less than 1% of students shown this program over a 10 year period correctly produced the first line of output. In its simplest form, the program is

```
      X = 78931.15
      DO 20 I = 1,3
         X = X + 0.01
         PRINT *, X
20    CONTINUE
      STOP
      END
```

Using IBM hexadecimal format, 78931.15_{10} = 13453.2_{16} = 78931.12_{10} and 0.01_{10} = 0.02_{16}. To align the exponents, 0.02_{16} must be right shifted **6 hexadecimal digits,** thus becoming 0. Thus, on IBM short precision

systems, because of shifting and truncation, each line of output will be 78931.12. This program has been translated and run on a 750 VAX using VAX FORTRAN, a PC compatible using MicroSoft GWBasic, Turbo Basic, and Turbo Pascal 5.5. The following chart gives the resulting outputs, where x = 78931:

IBM VS FORTRAN	VAX/VMS DEC FORTRAN	IBM PC Microsoft GWBASIC	IBM PC TURBO BASIC	IBM PC TURBO PASCAL 5.5
X.12	X.16	X.16	X.16	X.16
X.12	X.16	X.16	X.16	X.17
X.12	X.17	X.18	X.17	X.18

Increasing the printable precision, such as F9.3 or "##,###.###" or 9:3, had differing effects. The IBM 4341 output the decimals .125, and there was no change in VAX output. Because no floating point instructions exist on Intel 8088, GWBASIC, Turbo Basic, and Turbo Pascal implement floating point completely in software. GWBASIC output the decimals .160, .160, and .180, while Turbo Basic, which unavoidably uses double precision and then rounds back [BORL87], output the decimals .156, .164, and .172. Turbo Pascal uses a 48-bit, base-2 floating point format and produced the expected output [BORLa87].

The second floating point oddity concerns the well-known and often studied **Harmonic series** $H_n = 1 + 1/2 + 1/3 + \ldots + 1/n$. In standard courses in real analysis, using the fact that the real number system is infinitely dense and continuous, **this series is proven to diverge**. But on **any floating point implementation on any computer, the Harmonic series converges.**

For proof of the convergence of the Harmonic series, consider that as n increases, 1/(n+1) decreases, so that when it is added to H_n, the mantissa for 1/(n+1) eventually gets shifted enough so that it contains only 0's. At that point $H_{n+1} = H_n$, and the series converges.

A program, using the 32-bit single precision format, was written to find the convergence value and the CPU time, **in seconds**, required for the Harmonic series to converge. The following chart summarizes the results of running this program:

Machine Type	Language Environment	Harmonic Sum	Largest N	CPU Time
IBM 4341	VS Fortran	14.01743	1,048,577	13
VAX 750	VAX Fortran	15.40368	2,097,153	76
386-20 MhZ	Turbo Basic	15.40368	2,097,152	1,398
386-33 MhZ	Turbo Basic	15.40368	2,097,152	770
386-20 w 387	Turbo Basic	15.40368	2,097,152	138
386-20 w 387	Turbo Pascal5.5	15.40368	2,097,152	68

The results for the IBM 4341 are as expected: much faster computation because of hexadecimal mantissas and hexadecimal shifting, **but also much less accuracy.** The VAX and PC floating point formats are essentially the same, as are their accuracies. The first two PC timings were done for "386 PC Compatibles" without the Intel 80387 co-processor. These comparisons, and their huge time differences, are somewhat biased, as there is no hardware support for floating point. The last two PC timings provide dramatic evidence of the co-processor's utility in doing strenuous floating point work, and show that 386 PC's are nearly equal to mainframes and minis in raw computational power. Garner [GARN76] summarizes the trade-offs in floating point design: "Better accuracy is obtained with small base values and sophisticated round-off algorithms while computational speed is associated with larger base values and crude round-off procedures, such as truncation." Agreeing with

our results, Ginsberg [GINS77] notes that, if computational accuracy is needed, the combination of base-16, short mantissa size, and truncation (IBM Short Form) should definitely be avoided.

PLEA FOR TEACHING FLOATING POINT

This paper was written to show some of the "non-properties" that all floating point systems inherently have and to provide examples, for both pencil and paper, and in differing computing environments of real floating point anomalies. Much modern computing instruction has relegated floating point arithmetic to just another variable type. If the inherent problems posed by floating point implementations are mentioned at all, they are usually all collected under the umbrella term **round-off error.**

A discussion of floating point implementation details, similar to those included here, should be done in CS1. At the very least, Theorem 1 should be presented; even better would be to present "paper and pencil" exercises, backed up by results obtained from the ever-present hand calculators, similar to those presented in Theorem 3. Through such discussions of "non-properties" and "anomalies", various notions such as binary point alignment, truncation, and precision could be explained.

These explanations will help students understand the limitations of the computing system, an important foundational idea for later work in Theoretical Computer Science. Also, showing a few of the "**Not A Number**" and "**Infinity**" type anomalies explained in the IEEE proposal and excellently presented by Professor Kahan in his Turing Lecture [KAHA90] could easily be done in CS1 and/or CS2. **These floating point discussions should occupy at least one or two lectures in introductory courses,** as they are much more important to a student's long term understanding of computers and how to accurately use them than discussions of language syntax.

Floating point arithmetic could also be carefully studied in **Discrete Mathematics** courses, especially those taught concurrently with the calculus sequence. The general axiomatic properties of floating point numbers, such as numeric representation, shifting for alignment, and expected accuracy of computed results could be intensely studied. A discussion of Theorems 1, 2, and 3, and studying various "floating point anomalies" form a basis for such a study. Such a study normally gives the students "something concrete" to learn, while simultaneously giving the professor "something axiomatic" to teach. After such a study, students will normally really understand the flavor of binary computations, and come to an excellent understanding of how computers actually compute.

Finally, floating point problems are best explained in a **Numerical Methods** course. One goal of most Numerical Methods courses is to teach students how to use a computer to generate accurate results. Another goal is to understand the reasons for inaccurate results. Since round-off type errors are a fact of life for those computing with real numbers, determining bounds for the accuracy of computed results is another goal for a numerical methods course.

Thus a discussion of Theorem 1, and then a thorough discussion of Theorems 2 and 3 really meet the objectives for the Numerical Methods course. Most texts in Numerical Methods use some sort of trial floating point system, ranging from 2 decimal bits to 64 binary bits. The WIF system was designed to be "in the middle". The advantage of using a system, such as WIF, is that the results can all be verified by hand calculation. Excellent exercises, such as those given for the PUF system, can then be given to students. Finally, students can then be encouraged to study exactly how their own system handles floating point, with appropriately chosen examples that illustrate all the problems posed by Theorems 1-3. They can also be challenged "to find some anomalies" on their own. They could even be given problems that show that different formulas will be computed with different results on different machines. The resulting exposure to the details of computational problems can be a lifetime benefit to both a student of computing and a user of computing equipment.

To deprive a student of an understanding of how and why floating point implementations work is not preparing him or her for "real" world data!

BIBLIOGRAPHY

BORL87 **Turbo Pascal 4.0 Manual.** Borland International, Inc., Scotts Valley, Ca., 1987, pp 329-334.

BORL87a **Turbo Basic 1.0 Manual.** Borland International, Inc., Scotts Valley, Ca., 1987, pp 383-388.

GARN76 Garner, H.L., "**A Survey of Some Recent Contributions to Computer Arithmetic**". IEEE Transactions on Computers, Vol. C-25, No. 12, Dec. 1976, pp. 1277-1282.

GINS77 Ginsberg, M., "**Numerical Influences on the Design of Floating Point Arithmetic for Microprocessors,**" Proc. 1st Annual Rocky Mountain Symposium on Microcomputers, Aug. 1977, pp 24-72.

FLYN82 Flynn, M.J., and Waser, S. **Introduction to Arithmetic for Digital Systems Designers**, Holt Rinehart and Winston, New York, NY., 1982, pp 21-51.

IBMP72 **IBM System/370 Principles of Operation.** International Business Machines Corporation, Poughkeepsie, N.Y., 1972, pp 133-148.

IEEE85 "**Binary Floating-Point Arithmetic,**" Institute of Electrical and Electronic Engineers, IEEE Standard 754-1985, publ. no. SH10116, 1985.

KAHA90 Kahan, W., "**Turing Award Lecture**", ACM Symposium, Washington, D.C., Feb. 1990.

KUDL83 Kudlick, M.D., **Assembly Language Programming for IBM Systems 360 and 370 for OS and DOS,** Wm. C. Brown, Dubuque, Io., 1983, pp 320-352.

LEVY89 Levy, H.M., and Eckhouse, R.H., **Computer Programming and Architecture:The VAX: 2nd Ed.**", Digital Press, Maynard, Ma., 1989, pp 26-28, 182-184.