proceedings

NOVEMBER 8-11, 1977

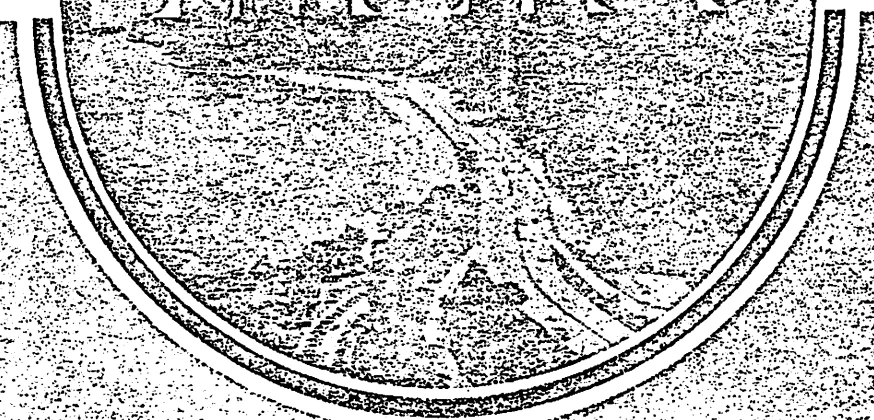
compsac 77

SHERATON-O'HARE MOTOR HOTEL / CHICAGO O'HARE AIRPORT

The IEEE Computer Society's First International

Computer Software & Applications Conference

CHICAGO



THE INTEL STANDARD FOR FLOATING POINT ARITHMETIC

John F. Palmer

IEEE COMPUTER SOCIETY

INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS

IEEE Catalog No. 77CH1291-4C

THE INTEL STANDARD FOR FLOATING-POINT ARITHMETIC

John F. Palmer

Intel Corporation
Santa Clara, CA

Microprocessors will inevitably enter the realm of serious floating-point computation. At the present there are some software packages and bit-slice boards that perform floating-point computation, but LSI technology will soon be capable of numeric calculation with respectable speeds. If we are to avoid the chaotic situation that exists in this area among mainframes and minicomputers, it is imperative that a standard be adopted. A standard, just because it is standard, will confer some benefits; but to have any hope of permanence the standard should be carefully developed, paying particular attention to those with experience in the field. We have studied the present situation and consulted with known experts and have adopted an internal standard for floating-point formats and arithmetic that we believe could be adopted for microprocessors in general.

Floating-point arithmetic is an area where microprocessors have so far found little use. Because of slow execution speeds and narrow data paths, they have not been capable of serious numerical computation. However, as the power of microprocessors increases, it is inevitable that floating-point capability will eventually be provided in silicon.

In order to avoid some of the problems that have in the past been associated with floating-point arithmetic, we have decided to adopt a standard for the formats and arithmetic algorithms. The purpose of this paper is to present this standard and briefly discuss some of its merits.

Adopting a standard would confer advantages even if the standard had some defects since at least everyone would be "programming around" the same anomalies. However, the standard we propose has the additional advantages of providing maximum accuracy with no anomalies. The arithmetic is done according to easily understood rules which make results predictable and accurate. In addition to the obvious advantage of making mathematical software easier to write, another benefit that will be explained later is the ability to

perform Interval Arithmetic[1] only about two to four times slower than ordinary floating-point arithmetic. (A well implemented Interval Arithmetic package on a typical mainframe, because of faulty arithmetic, takes from forty to three hundred times as long as floating-point computation).

We have consulted with several experts in this field in order to formulate a superior standard, and we believe that it could be adopted as a general floating-point standard for microprocessors. This would do much to avoid the chaotic situation that now exists in the mainframe and minicomputer environments.

As evidence of this chaos and motivation for the need of a standard we will present three examples taken from the present and recent past. These examples will also motivate our giving accuracy a high priority in our standard.

The first example we will consider was the delivery of a major line of computers with no guard digits in the double precision hardware. One explanation is that it seems somewhat counter intuitive that one must calculate results to greater precision than one displays. The absence of guard digits is disastrous: for example, there is no multiplicative identity. Consider the following four bit example:

$$\begin{array}{rl} x = & .1111 * 2^0 \\ & \underline{.1000 * 2^1} \\ & .01111000 * 2^1 \end{array}$$

With no guard digits the last one bit in the product will be lost before normalization. The normalized result will be

$$.1110 \times 2^0$$

and therefore

$$1.0 * X \neq X$$

Another consequence of no guard digits is that relative errors can be far larger than those in computations with at least one guard digit. For an explanation of why guard digits are imperative see Kahan and Parlett[2].

When the programming difficulties caused by this oversight became clear, an offer to retrofit all existing hardware with a guard digit was made even though this product had been on the market for about two years.

Another problem that is common in numerical computing is the lack of a format standard, even within the same company. One major manufacturer has, with very few exceptions, a different floating-point format for each computer model. One of the most popular models even has a different format for the double precision software than for the hardware. One consequence of the lack of a format standard is that converting from one machine to another or upgrading from software to hardware can cause severe difficulties. Another is the proliferation of machine dependent software.

The last example we will cite has some interesting parallels with the previously cited case of no guard digit. In this case the guard digits are there, but they are not properly used. A floating-point subtraction on this widely used machine is performed in a double length register. Since an unnormalized quantity often results, the subtraction is followed by a normalize instruction. Unfortunately, this instruction only normalizes the most significant half of the double length result. This apparently innocent feature can cause catastrophic results. Consider the following four bit example:

$$(.1000 * 2^1) - (.1111 * 2^0)$$

$$\begin{array}{ll} .1000 & 0000 \\ \underline{.0111} & \underline{1000} \\ .0000 & 1000 \end{array}$$

correct result : $.1000 \times 2^{-3}$

computed result: 0

relative error : $\left| \dfrac{.1000 \times 2^{-3} - 0}{.1000 \times 2^{-3}} \right| = 1$

It should be pointed out that since the guard bits are there, a 5 instruction sequence exists that will yield a fairly accurate subtraction operation. Therefore, it is left to users, if aware of the problem, to decide if the extra overhead is worth it. Complicating the decision is the fact that one of the main reasons for using this powerful machine is its execution speed.

We have attempted to learn from the past; and if there is anything that seems evident, it is the need for carefully developed standards. The INTEL standard for floating-point arithmetic has been adopted to apply to all general purpose products including software, systems and components. (A software Floating-point Arithmetic Library (FPAL) for the 8080 and a Math Board (SBC-310) made of series 3000 bit slices have already been produced using this standard.) In presenting the standard we will first discuss the formats and then the arithmetic.

FORMAT STANDARD

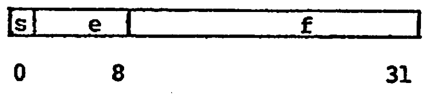
In selecting floating-point formats the primary considerations are word size and the radix of the arithmetic. The choice of word size will be discussed first. From the literature[3] it appears that a 32 bit word is reguarded as too small for scientific computation. However, scientific computation is not the only application of floating-point arithmetic, and in an environment of 8 and 16 bit processors, using a word length of 48 bits would generate 50% unnecessary overhead for a user whose application only required 32 bits. Therefore, we decided, as have most manufacturers, that two word sizes were needed: a short precision word of 32 bits and a 64 bit long precision word. This seems to be an adequate solution: providing a short (but potentially fast) precision for many non-scientific applications and a long precision for scientific and commercial applications requiring a high degree of accuracy. (It is our opinion that long precision is often used out of an exaggerated fear of roundoff errors, and in time highly accurate short precision arithmetic will be found adequate).

The selection of the radix of the arithmetic was primarily based on the observation that if one is going to use a short word of only 32 bits then one should choose the radix that provides the greatest potential accuracy.

Brent[4] showed that for a given word size and exponent range, binary arithmetic with an implicit first bit had round-off characteristics superior to radices 4, 8 or 16. In fact it was found that such a binary scheme with correctly rounded arithmetic was approximately one DECIMAL digit more accurate than truncated hexadecimal arithmetic. In addition binary arithmetic is well understood, easy to implement and potentially very fast. These considerations led us to choose a binary radix implemented with an implicit initial bit.

Besides word size and radix there are other considerations such as representation, precision and exponent range. On the matter of representation, there seems to be agreement that sign-magnitude is superior to 2's complement[3,5]. If the word size is fixed, the range and precision are functions of each other. Cody (1971) states that a range of $10^{\pm 75}$ is not adequate but that $10^{\pm 300}$ is. However, most people are using less range than $10^{\pm 100}$ and there seems to be much less complaint about the range than about the low precision of only 20 to 24 bits. Therefore, range should be sacrificed for precision, particularly for short words. Nevertheless, there is a large and growing group that is becoming used to ranges of at least $10^{\pm 100}$. This group includes users of pocket calculators, desk top computers and a few mainframes. Consistent with the observation that there was a need for two word lengths, it was decided to provide two exponent ranges. In the short word the range is small to provide as much precision as possible while in the long word the exponent range is very large. (This choice also has the interesting advantage that the long product of two short numbers cannot overflow or underflow.) Another balancing criterion is that the mantissa of the long word should be more than twice as long as the short mantissa to provide for almost error-free accumulation of inner products. The two formats are described in detail below.

<u>Short Format</u>



bit 0 : s = sign of the mantissa (s=1 means negative)

bits 1-8 : e = biased exponent (the bias is $2^7 -1$)

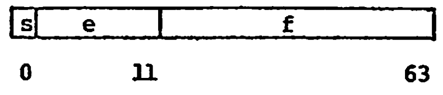
bits 9-31: f = fraction (when e $\neq$ 0 there is an assumed 1 bit at the left of the fraction; the binary point is between the assumed bit and the first explicit fraction bit)

formula : the number represented by the above floating-point word is

$$(-1)^s\, 2^{e-(2^7-1)} \quad (1. + .f)$$

provided e $\neq$ 0.

<u>Long Format</u>



bit 0 : s = sign

bits 1-11: e = biased exponent (the bias is $2^{10}-1$)

bits 12-63: f = fraction (when e $\neq$ 0 there is an assumed bit as explained above)

formula : the number represented by the long word is

$$(-1)^S 2^{e-(2^{10}-1)} \quad (1. + .f)$$

provided e $\neq$ 0.

There are some specific observations that should be made concerning these formats.

1) The bias was chosen so that the range was as balanced as possible with the constraint that all small numbers had representable reciprocals. Thus, contrary to usual practice, our range is slightly biased toward causing underflow since, for reasons discussed below, underflow is easier to deal with than overflow.

2) It has been thought that a weakness of the implicit first bit is that it precludes implementation of gradual underflow[6]. However, as explained below, a method has been discovered to accommodate both features.

3) There are several special cases.

a) The exponent field of all zeros is reserved for

1) zero - all bits are zero

2) uninitialized data - the sign bit is 1 and all others are zero.

3) denormalized numbers - all bit patterns with a zero exponent and a non zero fraction are to be interpreted as denormalized numbers. In performing the arithmetic, the leading bit is set to zero, and the implied 1 bit is instead added to the exponent. This interpretation will allow gradual underflow to be implemented.

Denormalized formula

$$(-1)^s\, 2^{1-(2^7-1)} \quad (0. + .f)$$

b) The exponent of all ones is also reserved.

1) +00 - the sign is zero and all other bits are one

2) -00 - all bits are one

3) indefinite – the sign and fraction are zero and exponent is all ones

4) others – the rest are reserved for as yet undefined uses.

The formats that have been chosen are to be an INTEL standard; it is envisioned that all INTEL floating-point products using binary arithmetic will implement at least one of these formats. This will make converting from one machine to another or from software to hardware much easier than if no standard were observed.

In software conversion there are also the considerations of result compatibility and accuracy. To accommodate conversion and to provide reliable and accurate results we have also developed a standard for floating-point arithmetic which we now describe.

ARITHMETIC STANDARD

The purpose of floating-point arithmetic is to provide the user a convenient number system in which he can obtain accurate results. Knuth suggests a model for floating-point arithmetic which is to produce the floating-point number nearest the true result. He also gives algorithms for implementing this general model. There are two problems with these methods: they are unnecessarily expensive, and the question of what to do when the result is exactly midway between two floating-point numbers is not treated. To resolve these difficulties, a set of rules[7] will be given that specify all cases, and then algorithms to implement these rules will be explained.

Rules

1. The set of floating-point numbers should contain 0 and 1 (additive and multiplicative identities), and if x is in the set then -x should be also.
2. If the true result of a floating-point operation is a floating-point number that number should be produced by the arithmetic, otherwise the result should be rounded according to rule 3.
3. If the true result is exactly halfway between two floating-point numbers then the arithmetic should produce the "even" one (the number whose last mantissa bit is zero). Otherwise the arithmetic should produce the floating-point number nearest the true result. (This rule assumes neither underflow nor overflow occur.)

The rules listed above (the rounding rule is called "round to even") provide maximum accuracy, and in addition remove the bias inherent in most rounding schemes. There is some experimental evidence for the desirability of unbiased rounding[8], but that is not the main reason for its implementation. There are two other very important motivations for providing such careful rounding. One is so that any user may correctly suppose that the floating-point arithmetic is yielding results as accurate as he could possibly expect: he can turn his attention from his hardware to his own problem. Another reason for rounding correctly is that if any compromise in accuracy is allowed, it becomes almost impossible to maintain a standard. Furthermore, there are algorithms to implement our rules that are very little (if at all) more expensive than any other reasonable rounding scheme. These algorithms will now be discussed.

We will present methods for doing floating-point add, subtract, multiply and divide that use "round to even" to produce the final result. To simplify the explanation we will illustrate the algorithms with 4 bit arithmetic. The existence of an accumulator will be assumed as shown

OF	B1	B2	B3	B4	G	R	ST

The bit labels denote:

1) OF – the overflow bit

2) B1-B4 – the 4 mantissa bits

3) G – the guard bit

4) R – the rounding bit

5) ST – the "sticky" bit

The sticky bit is set to one if any ones are shifted right of the rounding bit in the process of denormalization. If the sticky bit becomes set, it remains set throughout the operation. All shifting in the accumulator involves the OF, G, R and ST bits. The ST bit is not changed by left shifts, but zeros are introduced into OF by right shifts. In all of the algorithms below we will assume that the accumulator is initialized to zero and that the operands have been checked for being invalid or zero. Thus, only valid, normalized (nonzero) operands are considered. We will also assume that the appropriate exponent arithmetic for multiply, divide and shift adjustment is straightforward and need not be detailed, and that the sign of the result is also set appropriately.

Floating-point addition and subtraction will be considered together. The two cases to be considered are

1) addition of magnitudes – when numbers of the same sign are added or numbers of opposite signs are subtracted

2) subtraction of magnitudes - when numbers of the same sign are subtracted or numbers of opposite sign are added.

Addition of Magnitudes

1) Denormalization - the number with the smaller exponent is loaded into the accumulator and shifted right as many places as the difference in the exponents. (If the exponents are equal then either number is loaded and no shifting is required.)

2) Addition - the other operand is added to the accumulator.

3) Normalization - if the OF bit is set then shift the accumulator right one position.

4) Round - add 1 to the G position then if G=R=ST=0 set B4 to zero.

5) Renormalization - if OF is set then shift right.

6) Overflow - check for exponent overflow.

Subtraction of Magnitudes

1) Denormalization - load the number smaller in magnitude into the accumulator and shift right (if necessary) as before. (The result will be zero if and only if the operands are equal in which case set the result to zero and skip all subsequent steps.)

2) Subtract - Subtract the accumulator from the other operand leaving the result in the accumulator. (Thus, if ST is set it will generate a borrow.)

3) Normalization - shift the accumulator left until the B1 position is set to one.

4) Round - same as before (except no rounding is needed if more than one left shift was required in normalization.)

5) Renormalization - if the rounding caused OF to become set then shift right.

Multiplication

In describing the algorithm for floating-point multiply we will assume the facility exits to form the double length product of two numbers. (If not one can do an "add-and-shift-right" algorithm in an accumulator as described above to build up the product from low to high order. Then the unneeded low order part is lost as it is shifted out of the ST bit.)

1) Multiply - form the double length product

2) Normalization - a shift by one may be needed to normalize the product.

3) Set up G,R,ST - let the normalized double length product be

B1	B2	B3	B4	B5	B6	B7	B8

then G=B5, R=B6 and ST=(B7 v B8).

4) Round - as before.

5) Renormalization - as before.

6) Errors - check for exponent overflow on underflow.

Division

We will assume that division is implemented so that the remainder at any point is available.

1) Divide - form the first six bits of the normalized quotient:

B1	B2	B3	B4	B5	B6

2) Set up G, R, ST - set G=B5, R=B6 and ST=remainder.

3) Round - as before.

4) Renormalization - as before.

There are subtle variations on the algorithms given above that will yield the same results with possibly some speed increases. For example, if done after rounding, only one normalization is needed. However, in that case the rounding algorithm is more complicated.

One of the consequences of implementing a sticky bit is that it is easy to provide directed rounding. In directed rounding one rounds toward the right or left on a standard number line as directed. With a directed rounding capability, Interval Arithmetic may be efficiently implemented with the minimum possible growth in interval size. Interval Arithmetic, if implemented efficiently, could be a significant computational aid. Not only can round-off error be controlled but one can use it to estimate the effect of noise in data (by letting mesured data enter a computation as an interval) and to simulate the effect of variables taking on a range of values. For example, if one wishes to prove that as long as the temperature stays in a certain range, a system's performance is not degraded, then one enters the temperature variable into the simulator as an interval.

Conclusion

It is clear that we have sacrificed some execution speed to attain maximum accuracy. We know that speed is an important factor, but we are convinced that accuracy and reliability are well worth their relatively small cost in speed.

1) Accurate, reliable arithmetic makes software easier to write, debug and maintain.

2) A user can ignore the arithmetic since there are no anomalies and turn his attention to his own problem.

3) Interval Arithmetic can be efficiently supported.

These are some of the reasons for insisting on correctly rounded arithmetic at a modest cost in speed. Another is that if a standard is good it has a much better chance of enduring.

There are many issues we have not addressed such as the proper response to error conditions or what facilities are important to support elementary functions and other important computations. One of our goals is to make the development of mathematical software much easier than it has been in the past. We believe that our format and correctly rounded algorithm standards are a major step in that direction. However, there are certainly other important considerations that we are currently studying.

Acknowledgments

We are indebted to Professor W. Kahan for his lecture notes from which several examples have been taken and for many discussions which have greatly influenced our philosophy on floating-point computation.

Bibliography

1. Moore, R.E. (1966), Interval Analysis, Englewood Cliffs, N.J.: Prentice-Hall.

2. Kahan, W. and Parlett, B. (1977), "Can You Count On Your Calculator," Memorandum No. UCB/ERL M77/21, University of California, Berkely.

3. Cody, W. J. (1971), "Desirable Hardware Characteristics for Scientific Computation," SIGNUM Newsletter, 6, No. 1, 16-31.

4. Brent, R. (1973), "On the Precision Attainable with Various Floating-Point Number Systems," IEEE Trans. Computers, vol. C-22, No. 6, 601-607.

5. Knuth, D.E. (1969), "Seminumerical Algorithms," The Art of Computer Programming, vol. 2 Reading, Mass.: Addison - Wesley.

6. Sterbenz, P. (1974), Floating-Point Computation, Englewood Cliffs, N. J.: Prentice - Hall.

7. Kahan, W. (1973), "Implementation of Algorithums, Part I," Tech. Report 20, Department Comp. Sci., Univ. Cal., Berkeley.

8. Kuki, H. and Cody, W.J. (1973), "A Statistical Study of the Accuracy of floating-Point Number Systems," CACM 16, 223-230.