The INTEL® 8087 numeric data processor

by JOHN F. PALMER
Intel Corporation
Santa Clara, California

INTRODUCTION

The INTEL® 8087 is a high performance general purpose numeric data processor. It is used with the INTEL® 8086, or the INTEL® 8088, microprocessors to extend their instruction sets with over 100 instructions (not counting addressing mode). The 8087 has all of the 8086 addressing modes and through a coprocessing interface is able to execute numeric instructions concurrently with the 8086 (or 8088). The high performance overlapped execution is transparent to the user who sees the 8087 simply as an extension of the 8086 (8088). Furthermore, the 8087 is the only chip that must be added to an 8086-based system to provide numerics capability with a performance enhancement over software of more than 100. In addition to high performance, great care was taken to ensure that the 8087 could be used in any application involving numbers—including commercial calculations. This required an unprecedented level of accuracy and reliability to be built into the processor. The intent was to greatly simplify the production of high performance but reliable numeric software.

Mathematical software is easy for the uninitiated to write but notoriously hard for the expert. This paradox exists because the beginner is satisfied if his code usually works in his own machine while the expert attempts, against overwhelming obstacles, to produce programs that always work on a large number of computers. The problem is that while standard formulas of mathematics are fairly easy to translate into FORTRAN they often are subject to instabilities due to roundoff error. Consider, for example, the quadratic equation

$$Ax^2 - 2Bx + C = 0$$

whose solutions are

$$x_1 = (B + \sqrt{B^2 - AC})/A$$

$$x_2 = (B - \sqrt{B^2 - AC})/A$$

Programs using these formulas, when run on a conventional computer, will produce results that are very sensitive to roundoff damage.

Since roundoff analysis is subtle, difficult and exceedingly tedious, our intent in the 8087 design was not only to make reliable and robust software easier for the expert to build but to make it more likely that the unanalyzed code of the average programmer would run successfully. For example, the above formulas for the quadratic roots will be far less sensitive to roundoff error if evaluated on the 8087 instead of a typical computer.

Another important aspect of the 8087 is that it is an implementation of a very carefully designed standard, proposed to the IEEE and destined to be emulated by many other manufacturers. The establishment of this standard will go far to provide an environment for experts to produce ever more reliable software. Until now most experts, in an attempt to produce portable code, have written for a mythical computer whose capabilities are an intersection of the capabilities of all major computers and whose arithmetic is a collection of all the ugliness of any of them. Thus these programs, while useful for everyone, are ideal for no one. Assuming a standard environment, professional programmers will be able to concentrate on optimizing the code since portability will be automatic.

The proposed IEEE Floating-Point Standard (1, 2, 3, 4) specifies two data formats

REAL (32 bits, 8 bit exponent)
LONG REAL (64 bits, 11 bit exponent)

and a support format we call

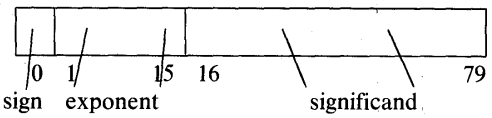
TEMPORARY REAL (80 bits, 15 bit exponent)

to indicate its intended use as a format to hold intermediate results. Along with the formats, the standard specifies three rounding rules, required operations (+, −, *, /, REM, SQRT, COMPARE) and exception conditions. The 8087 implements the full standard and many extensions. Some of the major benefits provided by the 8087 will be explained shortly but first an architectural overview will be given to serve as a framework for the more detailed later discussion.

ARCHITECTURAL OVERVIEW

The major architectural feature of the 8087 is its operand-result stack of 8 registers, each capable of storing an operand

in TEMPORARY REAL (80 bit) format as shown below:



All operands used within the 8087 are first converted to this format which provides 64 bits of precision and a range of about $10^{\pm 4900}$. In addition to the stack there is a set of registers called the ENVIRONMENT which contains the exception flags and pointers and processor control flags.

At the simplest level the programmer may treat the 8087 registers as a pure stack. All operands are explicitly loaded into the stack and operations are performed on the top elements of the stack. The load (or push) instruction can transfer operands to the stack using any one of seven data formats:

Shorter Integer (16 bit 2's complement)
Integer (32 bit 2's complement)
Long Integer (64 bit 2's complement)
Real (32 bit)
Long Real (64 bit)
Temporary Real (80 bit)
Packed Decimal (80 bit; 18 digits and sign)

The load instruction never causes a rounding error, since TEMPORARY REAL is precise enough to hold all seven types exactly. Stack operands can be returned to memory in any one of these seven forms using the store and pop instruction which automatically converts the top of stack to the designated format, stores it in memory and then pops the stack.

The arithmetic operations which manipulate the stack pop the top two elements off the stack, perform the operation and push the result back onto the stack. The operations supported are: ADD, SUBTRACT, SUBTRACT REVERSE, MULTIPLY, DIVIDE, and DIVIDE REVERSE. There are COMPARE instructions that set two bits in the environment (indicating "greater," "equal," "less," or "unordered") and then pop both elements, pop just the top, or pop neither. The REMAINDER instruction in the 8087 is an instruction primitive. It is intended to be used in a software loop to return both the "divisor" and the partial remainder of a division. There are several other instructions that operate on the top elements of the stack:

NEGATE:	reverses the sign of the top of stack
ABSOLUTE VALUE:	sets the sign of the top of stack to positive
SQRT:	computes the square root (its operation time is as fast as divide) of the top of stack
SCALE:	treats the next-of-top as an integer and adds it to the exponent of the top of stack—a fast form of multiplying by a power of two
EXAMINE:	a four bit condition code is set to indicate the contents of the top of stack (i.e., zero, positive, invalid, empty, etc.)
DECOMPOSE:	the top of stack is decomposed into its exponent and significand and these two results are returned to the stack
TEST:	the top of stack is compared to zero
CONSTANTS:	a set of instructions that load internally stored constants onto the top of stack (i.e., π, 0, 1, etc.)
TAN:	takes the top of stack, Z as an argument, assuming that $0 \le Z \le \pi/4$, and returns two results, x and y such that $y/x = \mathrm{Tan}(Z)$.
ARCTAN:	takes the top two stack elements and returns the result Z such that $Z = \arctan(y/x)$
EXPONENTIAL:	takes the top of stack, x, assuming $0 \le x \le 1/2$, and returns $2^x - 1$
LOGARITHM:	takes the top two stack elements and returns $y * \log_2(x)$.

In addition to the stack instructions listed above there are two instruction set optimizations. The first optimization is a set of arithmetic instructions that reference memory—one of the operands comes from the top of stack, the other from memory, and the result is returned to the top of stack. The operations which may use this optimization are ADD, SUBTRACT, SUBTRACT REVERSE, MULTIPLY, DIVIDE, DIVIDE REVERSE, COMPARE and COMPARE & POP. The four types of memory operands that can be referenced by these instructions are SHORT INTEGER (16 bits), INTEGER (32 bits), REAL (32 bits) and LONG REAL (64 bits). There are also STORE (without POP) instructions that reference the same four operand types. These instructions significantly reduce the number of instructions needed to evaluate a typical expression. For example, suppose R, X and Z are REAL, S and Y are LONG REAL, I is SHORT INTEGER and J and K are INTEGER. Then the expression

$$R := (S := (X/I + Y)/((J - K) * Z))$$

is evaluated by the following code sequence:

Instruction	*Memory Reference*
LOAD REAL	X
DIVIDE SHORT INTEGER	I
ADD LONG REAL	Y
LOAD INTEGER	J
SUBTRACT INTEGER	K
MULTIPLY REAL	Z
DIVIDE REVERSE	
STORE LONG REAL	S
STORE & POP REAL	R

Without the additional memory referencing instructions the above expression would have required 14 instructions

and 3 stack elements instead of 9 instructions and 2 stack elements.

The second optimization involves internal stack addressing. There is a set of arithmetic instructions: ADD, SUBTRACT, SUBTRACT REVERSE, MULTIPLY, DIVIDE and DIVIDE REVERSE, that may take one operand from the top of stack (TOP) and the other operand from any stack element addressed relation to TOP (i.e., TOP+i, i=0,...7) and the result can be written over either operand. If the result is returned to the stack element (instead of the stack top) the instruction may either leave the top unaltered or pop the stack.

Thus the new instructions are:

(TOP) op (TOP+i)→(TOP)
(TOP) op (TOP+i)→(TOP+i)
(TOP) op (TOP+i)→(TOP+i) & POP

In addition to the arithmetic instructions mentioned, LOAD, STORE, STORE & POP, and EXCHANGE instructions may also refer to stack elements relative to TOP. For example LOAD TOP+i would load the contents of the ith stack element beneath the top onto the top of stack. These instructions allow stack elements to be used to accumulate results in loops and to hold common subexpressions. For example, suppose $X(I)$ is an array of N REAL's and we want to calculate

$$R:=\sum_{i=1}^{N} X_i,\ S:=\sum_{i=1}^{N} i*X_i,\ T:=\sum_{i=1}^{N} X_i^2$$

	INSTRUCTION	*MEMORY REFERENCE*
(R)	LOAD ZERO	
(S)	LOAD ZERO	
(T)	LOAD ZERO	
	LOAD REAL	X(I)
	ADD TOP+3	
	LOAD TOP+0 (this is the DUPLICATE TOP instruction)	
LOOP on I:	MULTIPLY TOP+0 (this is the SQUARE TOP instruction)	
	ADD & POP TOP+2	
	MULTIPLY SHORT INTEGER	I
	ADD & POP TOP+2	
	STORE & POP REAL	T
	STORE & POP REAL	S
	STORE & POP REAL	R

This stack addressing capability both minimizes memory referencing and permits loop accumulations to benefit from the extended range and precision of TEMPORARY REAL thus significantly attenuating the effect of roundoff error and making intermediate overflow or underflow practically impossible. Thus the 8087 may be thought of as a "pure" stack machine with optimizations for memory and internal stack element addressing.

In addition to the computation instructions the 8087 has a set of administrative instructions for processor control and for status saving and restoring. In order to minimize context switching overhead there are single instructions, SAVE and RESTORE, that store and load respectively all 8087 volatile status. Also provided are instructions for loading and storing the 8087 status needed for software exception handling: exception flags and pointers to the offending instruction and datum. Finally, there is a 16 bit CONTROL WORD that may be loaded and stored. The contents of the control word dictate:

1. the rounding mode—there are four types of rounding.
2. the internal precision—results may be held internally in TEMPORARY REAL format but rounded to REAL (24 bit), LONG REAL (53 bit) or TEMPORARY REAL (64 bit) precision.
3. the mode of infinity arithmetic—there are two types of infinity closure, affine and projective, that will be explained later.
4. the response to exceptions—for each type of exception there is both a flag and an exception mask. According to the setting of the mask the 8087 either interrupts after setting the exception flag or it executes an on-chip microcoded exception handler and continues processing.

The usefulness of these controls and the power of the 8087 exception handling will be explained in the next section.

USER BENEFITS

Many of the 8087 features confer significant user benefits. The benefits that are provided by five of these features will be described in this section:

1. the "extended" (TEMPORARY REAL) support format
2. the rounding modes
3. the on-chip exception handling
4. the modified stack architecture
5. the high performance.

One of the major innovations of the 8087 is the provision of an extended support format called TEMPORARY REAL. This format provides several significant advantages. Firstly, the 8087 should be thought of as having clean REAL (single) and LONG REAL (double) precision. By this we mean that not only is the arithmetic accurate but the commonly supplied system functions are also accurate to LONG REAL precision. For example if x is LONG REAL then e^x, $\ln(x)$, $\tan(x)$, etc., will all be accurate to within less than a unit in the last place of LONG REAL precision—in fact because of the on-chip primitive functions the logarithmic and trigonometric functions will be accurate to within a few units in the last place of TEMPORARY REAL precision. The

benefits of the TEMPREAL format can also be seen by examining its use in the most demanding function in the 8087's repertoire, X^y. In calculating this function one loses in extreme cases as many fraction bits in the answer as there are bits in the exponent of y; if x and y are restricted to LONG REAL then $z=x^y$ can lose about 11 bits in these extreme cases. This is a significant error in a function that is crucial for commercial calculations involving interest rates. By using TEMPORARY REAL and the 8087 logarithmic functions we can compute x^y, where x and y are LONG REAL, accurate to about a unit in the last place of LONG REAL precision. Besides providing accurate rate of return calculations we can also ensure that integral values of the arguments yield exactly what is expected (i.e., $2^3=8$ not 8.00...01).

Another benefit of the TEMPORARY REAL format is the ability to provide accurate libraries—mathematical, statistical, commercial, etc. The user of these libraries delivers his data in REAL or LONG REAL precision and receives his results in the same format. However, the library has used TEMPORARY REAL variables to perform internal calculations, thus protecting against not only roundoff errors but intermediate overflows and underflows (most over/underflows occur on intermediate calculations since usually the input and output lie within fairly narrow ranges). Most libraries make performance claims "in the absence of over/underflow." By judiciously using TEMPORARY REAL variables, libraries will often be able to ensure that the only over/underflows that occur either do not matter or are on output where they provide the user a necessary and useful warning result.

Another advantage of this support format is that code written by programmers who are unfamiliar with analyzing their programs for roundoff errors and other problems—this includes almost all of us—will much more often work correctly.

This is particularly true because of the extended stack—it is almost impossible and certainly inconvenient to compute on the 8087 without using the TEMPORARY REAL format. Consider for example the program discussed earlier for calculating the roots of a quadratic equation:

$$R_1 := (B+\sqrt{B^2-AC})/A$$

$$R_2 := (B-\sqrt{B^2-AC})/A$$

On a typical computer with no support format these formulas from high school math are subject to severe roundoff damage. However, because of the stack of TEMPORARY REAL registers, if the expressions are evaluated on the 8087, the support format is used automatically and invisibly for the sensitive parts of the calculation and the expressions are much more accurate. The 8087 stack thus makes "certified" software easier to write and makes it more likely that uncertified software is reliable.

A second major contribution of the 8087 to numerical computation is the capability of controlling the rounding mode. As described earlier there is a field in the CONTROL WORD of the 8087 that specifies how infinitely precise results are to be rounded to fit the designated format. If the correct result is exactly representable then that result is returned regardless of the rounding mode. Otherwise the result can be specified to be any one of:

1. the nearest (if there are two then return the one with zero in the least significant bit—this avoids the usual bias)
2. the next larger
3. the next smaller (these modes are termed "directed rounding" (5))
4. the closer to zero (true truncation)

Normally one would use the "nearer" rounding to compute the most accurate and statistically unbiased estimate of the correct result. Alternatively, by using the directed roundings, one can not only compute rigorous error bounds at crucial places in a program but also implement Interval Arithmetic (6,7). Interval Arithmetic, where operands and results are intervals instead of isolated numbers, completely encloses all rounding errors. Thus when a computation yields an interval result, the user knows that the exact result is contained in that interval. Interval Arithmetic can also be used to estimate the consequences of uncertainty in data. By entering the data as intervals enclosing any possible measurement errors, the width of the resulting intervals gives an indication of the sensitivity of the computation to those errors. Another use of Interval Arithmetic is to calculate, in a simulation, the effect on a system as a variable such as TEMPERATURE passes through a range of values. Professor W. Kahan of the University of California at Berkeley has written (8):

> "No other feature would enhance safe numerical computation more that the provision of INTERVAL as a data type in FORTRAN as readily accessible as INTEGER or REAL."

If Interval Arithmetic is so useful why isn't it in widespread use? The main reason is that on a typical computer a rigorous Interval Arithmetic package can cost a factor of 100 to 300 over the ordinary floating-point arithmetic. On the 8087 this penalty is expected to be a factor of about 5. The implementation cost of providing the directed roundings was no greater than that of unbiased rounding so the value of the capability far exceeds it cost.

Another area where the 8087 makes significant contributions to safe but flexible software is exception detection and handling. Exception detection on the 8087 serves three main functions:

1. to report potentially fatal programming errors
2. to permit execution to be resumed after prearranged response to exceptional conditions
3. to allow functional extensions to the system.

Each type of exception detected by the 8087 has associated with it both a flag and a mask. (The exception masks are part of the CONTROL word and their value is set and saved by LOAD CONTROL and STORE CONTROL instructions.) When an exception occurs, the 8087 sets a flag and if the flag's mask is reset, an interrupt is generated. The interrupt procedure (exception handler) has access to the address of the instruction that caused the exception and the

address of the referenced datum (if any). If, on the other hand, the exception flag's mask is set, then the 8087 executes an on-chip microcoded exception handler that performs the second function described above: the instruction's response to the exception is "tailored" to that desired in the vast majority of cases. Execution resumes but the flag remains set until it is read and reset by software.

The exceptions that the 8087 detects and its response to them are explained below.

1. INVALID OPERATION: Stack overflow, stack underflow, indeterminate form (0/0, $\infty - \infty$, etc.) or the use of a Non-Number (NAN) as an operand. An exponent value is reserved and any bit pattern with this value in the exponent field is termed a Non-Number and causes this exception.
   a. Masked: If the exception was caused by using NAN's as operands then the NAN (the "larger" if both operands were NAN) is delivered as the result, otherwise a special NAN called INDEFINITE is returned.
   b. Unmasked: Interrupt before any processing.

This exception is used for all of the purposes described. Indeterminate forms are usually fatal errors and should be reported—either immediately or by propagating INDEFINITE to the end of the program and thus discovering both the error and how it contaminates subsequent calculations. Stack over/underflow is also usually fatal but an ambitious exception handler could use this exception to extend the 8087 stack to memory. Finally, the NAN's can be used for both run time diagnostics and functional extensions. As an example of the former, one could fill uninitialized arrays with NAN's each of whose significands contains the value of its index. Thus a reference to an uninitialized array element would not only indicate that it was uninitialized but which one it was. An example of functional extension would be to use the NAN as a pointer into a heap of values that could not be stored in the specified format. This would make it possible to implement a nearly infinite exponent range.

2. OVERFLOW: The result is too large in magnitude to fit the specified format
   a. Masked: Infinity with the sign of the correct result is returned.
   b. Unmasked: An encoding of the true result is returned and then interrupt is signalled.
3. ZERO DIVISOR: The divisor is zero while the dividend is a finite non-zero number
   a. Masked: Infinity is delivered with the sign as the XOR of the signs of the operands.
   b. Unmasked: Interrupt before processing.

Both of these exceptions, if masked, generate infinities which are special bit patterns and must be dealt with in a safe, consistent manner by the 8087 in subsequent calculations. For this reason the 8087 recognizes infinities as valid operands and deals with them in one of two modes, AFFINE or PROJECTIVE, determined by a field in the CONTROL WORD. The basic difference is that the affine treats all finite numbers as if $-\infty \leq x \leq +\infty$ while in the projective mode ∞ has no sign and cannot be compared to finite numbers. The affine mode is powerful but can give misleading results while the projective mode is always safe but not quite as useful as affine. The default is projective and this is the recommended mode unless a user has analyzed his program and is sure the affine mode is safe.

4. UNDERFLOW: The result is non-zero but too small in magnitude to fit in the specified format
   a. Masked: The significand of the result is denormalized (shifted right) until the exponent is in range. This allows underflowed numbers "gradually" to become zero retaining as much information as possible and is called "gradual underflow."
   b. Unmasked: An encoding of the correct result is delivered and then an interrupt is signalled.

Underflow is usually not a fatal error and by using gradual underflow (masking the exception) one can proceed, confident that the risk of undetected fatal underflow is commensurate with the risk of fatal roundoff damage (see 4).

5. DENORMALIZED OPERAND: At least one of the operands is denormalized, it has the smallest exponent but a non-zero significand.
   a. Masked: The operation proceeds as if the operand were unnormalized.
   b. Unmasked: Interrupt without processing. This exception is used to implement, via exception handlers, an optional mode of arithmetic described in the proposed IEEE Standard for Floating-Point Arithmetic (2) in which no unnormalized results are generated.
6. INEXACT RESULT: If the true result is not exactly representable in the specified format, the result is rounded according to the rounding mode, the flag is set and
   a. Masked: Execution continues
   b. Unmasked: Interrupt is signalled.

This exception is used to implement exact arithmetic in floating-point for, among other uses, accounting calculations and preconditioning (see 4).

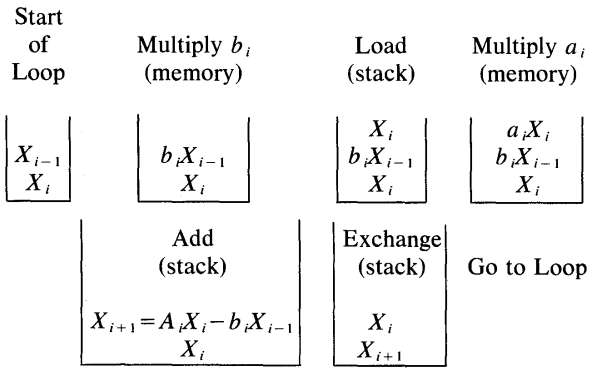
Exception handlers are difficult to write, debug and maintain and they consume valuable memory space at run time. Therefore, we have provided, on the 8087, exception handling that will be ideal for the vast majority of situations. We recommend that most users mask all exceptions except INVALID OPERATION. With the built-in exception handling and reliable infinity arithmetic it is the only exception that is likely to be fatal. User exception handling software can thus be kept to a minimum.

Another special feature of the 8087 to enhance performance and accuracy is the ability to select operands from and return results to internal stack elements. This stack element addressing mechanism which has already been described is useful for holding common subexpressions and for holding

accumulations during the execution of a loop. Another example will further illustrate its usefulness. An important calculation that is often found in the inner loop of numeric programs is the evaluation of recurrence relations. A particular example is the following three term recurrence

$$X_{i+1} = a_i X_i + b_i X_{i-1},\ i = 2,\ldots,N-1$$

Using a typical evaluation stack this computation would require, in addition to the add and two multiples, five memory references—four loads and one store. The evaluation on the 8087 stack requires only two memory references:



The "program" shown above illustrates a general principle. Almost all important numerical computations have inner loops that will benefit from the ability to access inner stack elements.

High performance was another of the important design goals of the 8087. It is difficult to compare 8087 performance with other machines since it is not feasible to obtain the same accuracy and reliability as the 8087 on even the largest mainframes. For example in executing a primitive instruction like MULTIPLY the 8087 provides:

1. A result with an extended precision and range
2. Correct unbiased rounding with optional direct roundings for error bounding
3. Reliable exception detection and safe, automatic handling
4. Forms of the instruction to minimize memory references.

No other computer—mainframe or minicomputer—integrates these features into a single architecture. But in addition to "architectural performance" a great deal of attention was given to raw instruction performance. For simplicity and execution speed the 8087 was implemented with an internal data path and ALU of 67 bits. There is a shifter that can shift left or right from 0 to 63 places in one clock cycle. This shifter was indispensable in normalization, data formatting and the transcendental functions which were evaluated using a modified CORDIC algorithm. The loops for MULTIPLY, DIVIDE and SQUARE ROOT were implemented with a hardware sequencer. MULTIPLY was optimized by checking for 40 least significant zeros and skipping them in the multiply loop—this would occur if either operand were originally SHORT REAL or if either value were an integer and less than 2^{25} in magnitude. The timing for several instructions demonstrates the 8087's performance.

Instruction	Execution Time (microseconds)
COMPARE	6
ADD (MAGNITUDE)	10
SUBTRACT (MAGNITUDE)	16
MULTIPLY	16,24*
DIVIDE	38
SQUARE ROOT	38

* The shorter time applies if either operand were originally SHORT REAL as explained earlier.

Additional performance is gained by the overlapped execution of the 8086 (8088) and the 8087. The amount is hard to estimate but is definitely material.

CONCLUSION

The architecture of the INTEL® 8087 has been described along with a review of its user benefits. The 8087 has unprecedented performance, reliability and capability—it can be used in any numerical application to provide a hundredfold increase in mathematical performance over the 8086 or 8088 alone. In contributing to and being compatible with the proposed IEEE Floating-Point Standard the 8087 has carefully balanced safety with utility.

The many features of the 8087, when combined, can make it appear complex. Like a car's automatic transmission the 8087 *is* complex, but also like an automatic transmission the user need not see the complexity to reap the benefits of Interval Arithmetic, reliable rounding, safe automatic exception handling and an integrated support format that virtually eliminates intermediate over/underflows and makes intermediate roundoff error negligible. The 8087 removes many of the pitfalls of numeric computation.

ACKNOWLEDGMENTS

I would like here to acknowledge some of the many people who contributed to the 8087. The architectural design was the joint work of Bruce Ravenal and myself, relying extensively on the advice of Professor W. Kahan of the University of California at Berkeley. Robert Koehler made significant contributions to the system aspects of the 8087 and Janis Baron designed the assembly language and implemented the 8087 Emulator—a software emulation for systems without an 8087. Rafi Nave and his engineering team in INTEL ISRAEL implemented the 8087—the largest microprocessor device yet in INTEL's history, and Dar-Sun Tsien carefully reviewed all aspects of the implementation. The management of INTEL must also be acknowledged for committing sig-

nificant resources to both implementation and promotion of a standard for reliable numeric data processing.

REFERENCES

1. Palmer, J. (1977), "The INTEL Standard for Floating-Point Arithmetic," *Proc. COMPSAC,* 107-112.
2. Coonan, J., Kahan, W., Palmer, J., Pittman, T. and Stevenson, D. (1979), "A Proposed standard for Binary Floating-Point Arithmetic," *SIGNUM Newsletter,* October.
3. Coonan, J. (1980), "Specifications for a Proposed Standard for Floating-Point Arithmetic," *Computer*, January.
4. Kahan, W. and Palmer, J. (1979), "On a Proposed Floating-Point Standard," *SIGNUM Newsletter,* October.
5. Yohe, J. (1973), "Roundings in Floating-Point Arithmetic," *IEEE Trans. Computers,* Vol. C-22, No. 6, 577-586.
6. Moore, R. E. (1966), *Interval Analysis,* Englewood Cliffs, N.J.: Prentice-Hall.
7. Kahan, W. (1968), "A More Complete Interval Arithmetic," Lecture Notes for a course at University of Michigan, June 17-21.
8. Kahan, W. (1972), "A Survey of Error Analysis," Information Processing 71, North Holland Publishing Company, 1214-1239.