COMPATIBLE NUMBER REPRESENTATIONS

Roy A. Keir
University of Utah
Salt Lake City, Utah 84112

A compatible number system for mixed fixed-point and floating-point arithmetic is described in terms of number formats and opcode sequences (for hardwired or microcoded control). This inexpensive system can be as fast as fixed-point arithmetic on integers, is faster than normalized arithmetic in floating point, gets answers identical to those of normalized arithmetic, and automatically satisfies the Algol-60 mixed-mode rules. The central concept is the avoidance of meaningless "normalization" following arithmetic operations. Adoption of this system could lead to simpler compilers.

Key Ideas

The purposes of this paper are to dispel some lingering myths about normalization and to bring to a wider audience certain interesting characteristics of some ancient work (circa 1959) which apparently will be novel to some present-day computer architects and language designers. This paper is explicitly based on the Bendix G-20 development. The Burroughs B-5000 (a contemporary of the G-20) exploited the same concepts.

Opcodes for Arithmetic

The central idea is simply to make fixed-point and floating-point number representations totally compatible (by using tags, by using descriptors, or simply by dispensing with a fixed-point format), then trim the opcode list to include only desired functions, letting the hardware automatically handle the arithmetic formats. For example, the entire list of arithmetic opcodes might consist of only the following:

Store the content of the (selected) accumulator.
Store the integer portion of the content of the accumulator.*
Clear and add
Clear and subtract
Add (rounded in an unbiased manner)
Subtract (ditto)
Multiply (ditto)
Divide (ditto)
Find the integer quotient
Find the residue
Various compare or branch opcodes, including a "branch on loss of digits" (see below).

Note the absence of FIXED-ADD, FLOATING ADD pairs, etc. Note the absence of a FLOAT command and the union of the IFIX function with the STORE opcode.

The proposed arithmetic system automatically does precisely what the algol 60 mixed-mode rules require, with little attention to types needed except during compilation of assignments (store commands). Comprehensibility of compiled code should also be simplified. These objectives were discussed long ago (CACM) in an article by A.A. Grau concerning a related proposal. In effect, all arithmetic is done in floating point, with the exponent appropriate to integers specified by tag, descriptor, or an exponent field when an integer operand is fetched from storage.

*Separate STORE opcodes for round, chop, and entier may be desirable for various applications.

Living Without Fixed-Point Arithmetic. Since we propose to do away with fixed-point arithmetic, we must respond to those needs now served by fixed-point computer instructions, namely speed and exactness. Normalized floating point arithmetic has historically been much slower (in smaller machines) than fixed-point arithmetic -- the proposed system uses a carefully-selected form of unnormalized arithmetic and can operate at fixed-point speeds. Non-trivial uses of fixed-point arithmetic require defenses or warning (preferably both) against loss of non-zero digits on either end of any operand -- the system to be described provides warning of inexact results and can easily be implemented so as to provide either retention or recovery of the "excess" digits.

Contrasts With Other Unnormalized Systems. This number system must be distinguished from several other systems which have been called "unnormalized". This is not Metropolis's method of sacrificing result digits to maintain an estimate of significance -- it gets answers which are indistinguishable in value from those of normalized arithmetic. It is not the practice of supplying unnormalized operands to an arithmetic unit designed for normalized operands and then looking to see if the answers can be useful -- the sequence shown in the Appendix for multiplication shows that real changes are required to make a coherent design. It isn't even the automatic pre-normalization of operands that is demolished in Sterbenz's fine book.[8] It is a system which avoids normalization wherever it can do so without loss of accuracy.

Some Challenges and Responses

If unnecessary normalization is avoided, it is of course necessary that add-subtract be designed to allow "scale left" during exponent equalization, and similar measures are needed in multiplication and division. Fundamental designs for some of this arithmetic are given in Appendix I so that the reader may verify the answers given below to some of the obvious challenges.

Isn't Normalization Essential to Achieving Maximum Accuracy?

No. If all you can save of an answer is 0079.3, writing it as 79.300 does not affect the accuracy one whit. It has long been known (see Sweeney 1965 for a concise demonstration) that at most one left shift is needed after differencing to keep all the answer you can ever keep, and if cancellation has occured, additional shifts can only supply a "stuffing" of zeros without improving the accuracy of the result. It's hard to accept, but those otherwise excellent articles which say things like "maximum accuracy is attained by use of normalized numbers" (Yohe 1973) are speaking of a sufficiency condition, not an always-necessary condition. Check those flowcharts in the appendix again. And check the next article you read on floating point error analysis to be sure that the author has allowed for the "stuffing" in his expression for the uncertainty in the values of input operands.

The proposed system is entirely compatible with (and was built with) one-digit post-operation "normalization" and unbiased (or "drift-free") rounding (see Knuth 1975, Keir 1975, Piper 1961), the best that has yet been implemented in hardware. The more generalized variants for precise numerical work (see Yohe, 1973)

can also be implemented quite easily.

When the proposed system and the normalized system (with identical roundoff rules) both get answers, those answers are identical. The proposed system gets meaningful answers in a very few cases where a strictly normalized system underflows, a phenomenon I believe has been called "gradual underflow."

Won't it be Slow to do Fixed-Point Arithmetic in a Floating-Point Unit?

No. If integers have been initially stored with a standard exponent, the proposed system does arithmetic on integers at fixed-point speed. The virtual radix point of each number stored using the STORE INTEGER opcode is adjusted to the right end of the register. The opcode sequences shown in the appendixes preserve this alignment wherever possible for integer results, so that integer operands for addition, subtraction, and STORE opcodes will have their radix points already properly aligned at the start of each operation.

Won't Floating Point Arithmetic be Slow in This System?

Curiously, the proposed system is slightly faster than conventional normalized arithmetic in terms of the number of shifts to be performed. Consider a result in which heavy cancellation has occurred: in normalized arithmetic this operand has some probability (almost .5 according to Sweeny) of having the smaller exponent in a subsequent add-subtract and therefore being subject to one or more right shifts during exponent equalization. These right shifts are therefore only undoing some of the preceding left shifts, which are thus shown to be not only unnecessary but time-wasting. In the proposed system, these shifts are postponed (in the diagrams, from states N2 & N3 to A5.3) or entirely avoided. The magnitude of the speedup is under study, but good estimates remain elusive.

One elementary result which may be new to at least some of you is that if all possible pairs of signed p-bit binary integers are added in a binary p-bit normalized accumulator the number of bits shifted (for large p) averages two and one half, of which one and one third are for exponent equalization and the remainder (1+1/6) are for post-operation normalization. In the compatible representation, no shifts at all are required for exponent equalization and only one fourth of a bit (right) shift for normalization.

Whatever speed is gained by fewer shifts is in addition to the reduction in calls on FIX and FLOAT procedures. The FLOAT operation is never needed and the FIX operation is a (perhaps microcoded) hardware sequence, usually automatically evoked within the STORE INTEGER command.

The speed effects are not all one-sided. For example, all numeric comparisons require exponent equalization, whereas in some normalized forms a fixed point comparison gets the correct answer faster. Another example is the operand with leading zeroes which happens to be the larger summand in each of several later additions and subtractions -- scaling this number all the way left once might be more efficient.

Doesn't the Proposed System Require a lot More Hardware?

Only a little more in most implementations. The extras show up as extra left-shift paths for exponent equalization, more complex predicates for microcode or sequencer and more lines of microcode or states in the sequence. At any given level of implementation, the percentage hardware system cost increase is miniscule. I argue that this cost is overwhelmed by hardware reductions resulting from drastic pruning of the opcode list and by software system cost reduction.

Isn't it Possible to Lose Digits Without Warning in Calculations That Should be Exact?

This is a hazard that must be prevented. The system described in the appendixes provides for a "digits were lost" condition bit which is automatically reset (to FALSE) on any accumulator-clearing operation and set to TRUE if any non-zero digits are subsequently shifted off the right end of the arithmetic register and not recovered by a post-operation "normalization" left shift. This condition bit can be accessed as a predicate prior to a STORE operation to give warning of fishy results (e.g., from an integer multiply, which some tricky programmer expected to be reduced modulo some fixed-point register size).

However, if an "improper fraction" can be exactly represented in the accumulator, the above condition bit will not necessarily be set by the operation which created this result. One must somehow have either this or another condition bit respond to a loss of non-zero digits during either a separate FIX opcode or during the FIX phase of a STORE INTEGER opcode. The implementation in the G-20 did not signal loss-of-digits (a regrettable omission), but under the original design criteria there probably would have been a separate NON-INTEGER condition bit activated by the STORE INTEGER command(s), and the original content of the accumulator would have been retained, both as an aid to recovery and because it makes sense in terms of the semantics which should have been associated with assignments to integer and "real" variables.

Can Mixed-Length Operands be Handled Conveniently?

Yes, indeed. The original implementation had three arithmetic store commands, STORE SINGLE-LENGTH INTEGER, STORE SINGLE-LENGTH FLOATING, and STORE DOUBLE-LENGTH FLOATING. A primitive tag informed the control logic of the length of a number being fetched as an arithmetic operand, and the second word of a double-length operand was fetched automatically. All arithmetic (yes, even index arithmetic) was done in double length floating point. Address values were automatically shifted by hardware to the standard integer location before being sent to an address register.

In retrospect, I favor a tagged representation (App. II) with four species of numbers (single-length, double-length)x(explicit-exponent, implicit-integer-exponent). The factor part of each accumulator (or stack location, of course) would be at least long enough to hold a long integer, the extra digits serving as guard digits during garden-variety floating point work. For reasons not relevant here, this plan was not possible to us at the time of the original design.

The use of the word factor instead of fraction or mantissa above was not accidental. It comes from the natural practice in compatible number representations of using an exponent value of zero when dealing with right-justified integers. Another choice may be better. The candidates and my (admittedly biased)

summary of the supporting arguments for each are:

1) (radix point at the left) $\leftrightarrow$ (fraction) $\leftrightarrow$ ($|f| < 1$). I believe the arguments for this choice all died of old age when floating point hardware came into widespread use.

2) (radix point to the right of the left-most digit) $\leftrightarrow$ ($|f| < B$). The strongest arithmetic analyst of my acquaintance argues for this ("If x is representable, 1.0/x should be representable for as many x's as possible...") (W. Kahan, private communication).

3) (radix point at the right) $\leftrightarrow$ (integer) $\leftrightarrow$ ($|f| < B^p$ for a p-digit number). Besides being "natural" for hand-coding, this choice causes the implicit exponent of an integer to be machine-independent (i.e., zero instead of p) and number-length-independent (0 and 0 instead of p and 2p). There is also a (usually negligible) cost advantage in generating and testing zeros in most hardware, but portability is the main reason that I favor this selection.

CONCLUSION

Formats and opcode definitions have been presented for a number system which, by allowing integers and floating-point numbers to be freely and safely intermixed, can considerably reduce the work-factor of a compiler in generating arithmetic code. Want to mix modes? Just do it. Want to base indexing on a real variable? Hardware can take care of it automatically. Want to write a procedure to work on any numeric operand, with (integer, real) binding deferred until activation time? Perfectly natural. Want to code a library function to accept any mix of single-length and double-length actual parameters? Automatic. Supersensitive calculation? Generate code to make heavy use of the "digits were lost" predicate.

The system is as fast as conventional fixed-point when handling integers, faster than normalized arithmetic on floating-point operands, and exactly as accurate as normalized arithmetic. The arithmetic opcode list is much shorter than that of a conventional computer geared to handle single and double-length integers and floating-point numbers.

How can you stand the clumsy old-fashioned system you're now living with when it would be so easy to microcode this modern 1959 model?

ACKNOWLEDGEMENTS:

Professor Harry Huskey provided the initial impetus to the development of the compatible number representations. Jim Buell, Bob Monroe, and Jesse Quatse all worked with me on the arithmetic of the G-20. Charles Piper provided many penetrating insights and fruitful discussions.

REFERENCES:

(1) C. A. Piper, G-20 Reference Manual, Bendix (1962).

(2) D. W. Sweeney, An Analysis of Floating-Point Addition, IBM Systems Journal, vol. 4 no. 1 (1964).

(3) J. F. Reiser and D. E. Knuth, Evading the Drift in Floating-Point Addition, Information Processing Letters, vol. 3 no. 3 (January 1975) 84-87.

(4) R. A. Keir, Should the Stable Rounding Rule Be Radix-Dependent?, accepted for publication, Information Processing Letters (1975).

(5) J. M. Yohe, Roundings in Floating-Point Arithmetic, IEEE Transactions on Computers C-22 (1972) 577-586.

(6) C. A. Piper, Roundoff (letter to the Editor), CACM vol. 4 no. 1 (March 1961) A13.

(7) R. A. Keir, Program-controllable Roundoff and the selection of a Stable Roundoff Rule, Proceedings of the Third IEEE Computer Arithmetic Symposium, 1975.

(8) P. H. Sterbenz, Floating-Point Computation, Prentice-Hall, Inc. 1974.

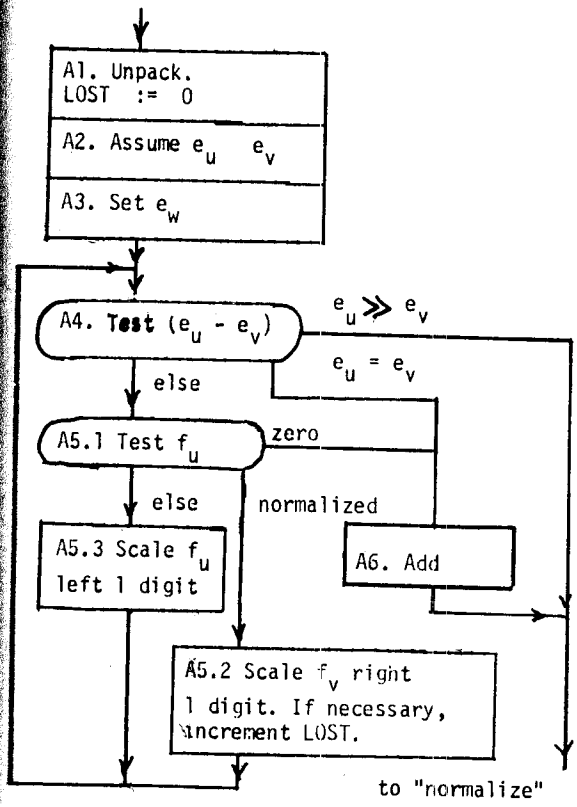
APPENDIXES:

APP. I: Fundamental algorithms for the compatible number system. Many steps shown for clarity as sequential should in practice be performed concurrently.

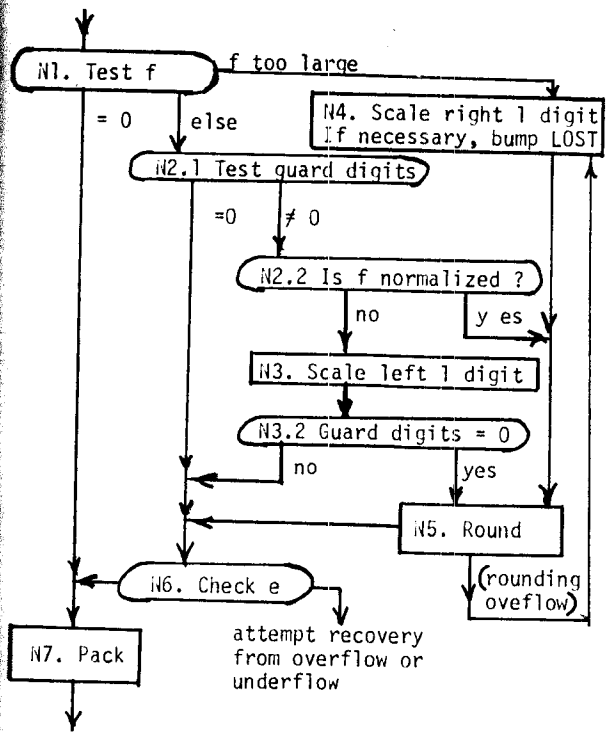
APP. II: A possible set of formats for an implementation of a compatible number system.

APPENDIX I. Some basic algorithms for a compatible number system.

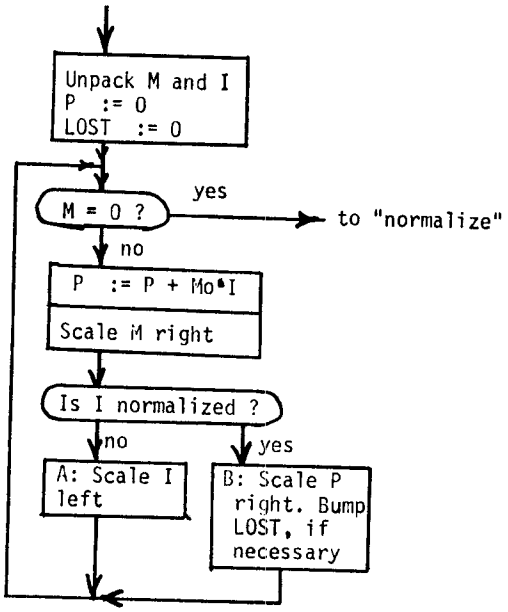
(a) Addition.



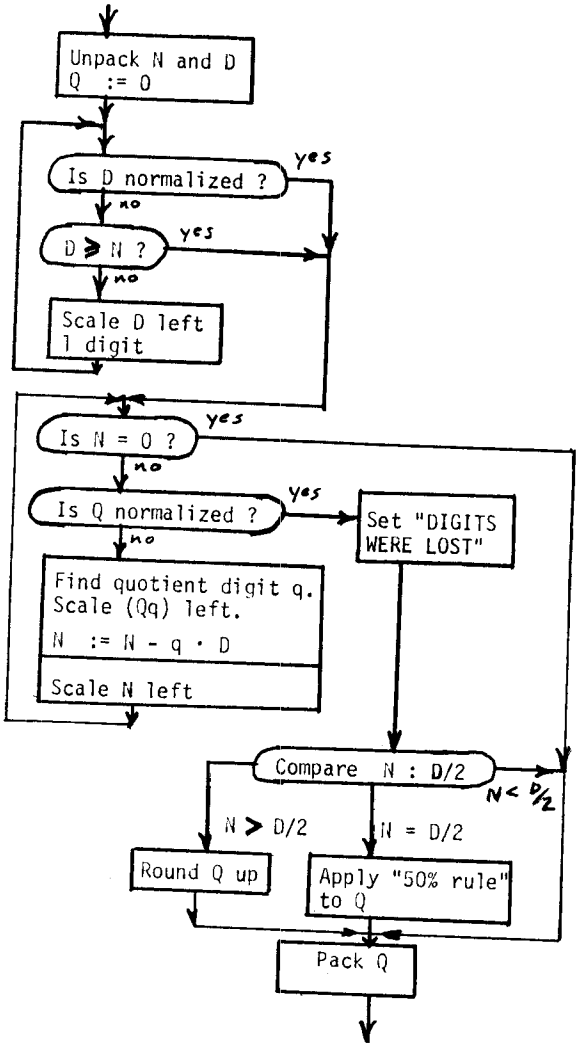
(b) "Normalization"



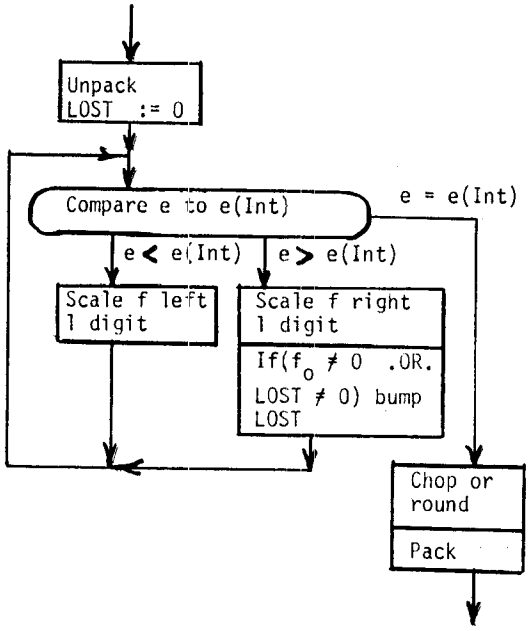
(c) Multiplication



(d) Division



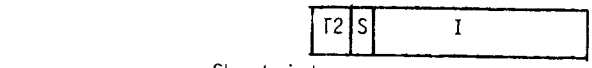
(e) Standardizing an integer



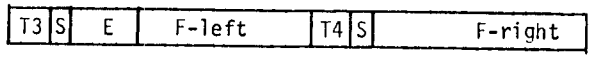
APPENDIX II. Some possible formats for a compatible number system.

(a) Using tags for length and exponent.

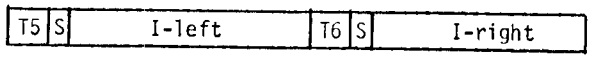
Short floating



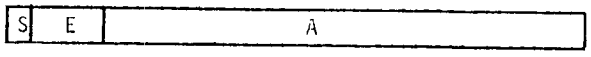
Short integer



Long floating



Long integer

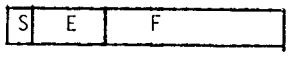


Standard form in arithmetic unit

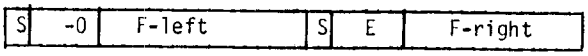
1. (Each E has k digits; each F or F-left has p digits; each F-right, I, I-left, or I-right has (k + p) digits; and A has (2k+2p) digits)

2. (T4 and T6 can be combined, or T3 and T5)

(b) Using a "forbidden" exponent as a length tag, no special form for integers.



Short number



Long number

1. (The address of the long number must of course be that of the left half)

2. (F, F-left, and F-right each has p digits)