RC 6203 (#26651) 9/14/76
Computer Science 11 pages

Research Report

A Unified Decimal Floating-Point Architecture for the Support of High-Level Languages
(Extended Abstract)

Frederic N. Ris

IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

IBM Research Division
San Jose · Yorktown · Zurich

RC 6203 (#26651) 9/14/76
Computer Science 11 pages

A Unified Decimal Floating-Point Architecture for the Support of High-Level Languages
(Extended Abstract)

Frederic N. Ris

IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

Abstract: This paper summarizes a proposal for a decimal floating-point arithmetic interface for the support of high-level languages, consisting both of the arithmetic operations observed by application programs and facilities to produce subroutine libraries accessible from these programs. What is not included here are the detailed motivations, examinations of alternatives, and implementation considerations which will appear in the full work.

An architecture for decimal floating-point operations is presented with the following characteristics:

- Operations are storage-to-storage, with no visible hierarchy of registers and store, in common with the data models of high-level, procedural languages.
- Elementary storage units occupy 32, 64, and 128 bits, in common with the basic high-speed data paths on much general-purpose digital computing equipment.
- The architecture is suitable both for compiler-generated code and for the implementation of mathematical subroutine libraries used by high-level applications.
- The computational model underlying the architecture is close to longstanding human computational practice and thus substantially easier to comprehend than existing floating-point systems.
- Regularity and freedom from anomaly have been given highest priority; time and storage requirements have been viewed as less important than consistency and accuracy when modest increments in hardware can be used to gain performance.

Acknowledgements: This paper is an outgrowth of numerous conversations held with Prof. W. M. Kahan while he was on sabbatical in the Mathematical Sciences Department at IBM Research in 1972–3; many of his ideas are reflected here. W. C. Carter and F. G. Gustavson contributed to most of these discussions; W. G. Bouricius and B. Tuckerman also took part in some.

Many early drafts were subjected to thorough scrutiny by W. C. Carter, F. G. Gustavson, W. M. Kahan, and H. P. Schlaeppi, each of whom has suggested numerous improvements.

1. BASIC ARCHITECTURAL ASSUMPTIONS.

This decimal floating-point architecture does not arise from first principles. It is specifically intended to be sympathetic with current high-performance, broad-function, general-purpose, commercially-oriented computing systems and with the semantics of languages such as Fortran and PL/I. Because we wish the ability to integrate this design with minimal impact in existing environments, a processor has not been designed around the floating-point unit. Rather, some basic assumptions now discussed are made which have significant influence on the decimal floating-point architecture by way of constraint.

We assume the store to be composed of eight-bit bytes and addressed to the byte level. The essence of much computing is character manipulation—even in Fortran, though under the covers [2]—and character addressibility is vital in producing efficient software.

We assume a fixed-point binary format for instruction and operand addressing. In order to do address arithmetic and to keep efficiently accessed arrays of pointers, a word length for addresses of 16, 32, or 64 bits leads to enormous architectural simplifications, even if the architected addressibility comprises fewer bits than the basic word. Because 16-bit addressibility gives access only to 64K bytes, it is assumed that the fixed-point word size is 32 bits, whether or not some addressing mechanisms employ a larger number. A 32-bit data path is thus likely to be a fundamental part of either the hardware or of a logical interface presented by hardware and a modest amount of microcode.

It is assumed that instructions occupy an even number of bytes and that the first two or four bytes of the instruction specify the operation and any status or options which may be applicable. Subsequent byte pairs (the number of which is determined by the first two bytes) specify operands. The mode in which the operands are addressed is of little concern, save that in this architecture there is no register-storage hierarchy, so that addresses in the store must be developed for each operand. Addressing possibilities include, but are not limited to, 32-bit direct address, 4-bit base register plus 12-bit byte displacement, 16-bit byte displacement from an implicit base register, and 24-bit absolute address modified by the sum of two 4-bit index register specifications.

We assume that the four-bit halves of bytes continue to be important in three key respects. The first is in base or index register specification as in the addressing examples just cited. The second is as BCD encoded decimal digits used in fixed-point decimal computations in which two digits are packed in a byte. The third is as a length specification for such strings of digits.

The consequence of this is that fixed-point decimal operations naturally take place on strings composed of an odd number of digits and a sign (occupying one of the digit positions), and that the operations are effective on data types comprising 1, 3, 5, ..., 31 digits. For fixed-point operations, thirty-one digits plus sign occupying sixteen bytes is thus a very natural limitation in the assumed underlying architecture that will remain.

We assume that efficient conversion operations exist among the fixed binary formats used in addressing, the fixed decimal formats used in computation, and the character formats used in input/output.

2. ONE-LEVEL STORE.

The decimal floating-point instructions have been assumed to be storage-to-storage operations for a number of reasons. Chief among these is that the data models of high level languages do not encompass a two-level, register/main store hierarchy. When an arithmetic expression has been parsed into its constituent parts and the semantic rules of the language have assigned attributes to the resulting generated temporary data items, a transformation with source-level equivalents has been performed. In a machine in which operations must be done in registers, code generation must additionally manage these resources and insert appropriate loading and storing operations.

For floating-point data this hardly seems worth the trouble. If the registers are architectural copies of storage formats, the extra work this entails must be repaid in performance, for the overhead incurred provides no additional function. For high-performance machines we expect that frequently referenced data which are natural candidiates for registers will reside in a cache. For other machines, we do not see that the presence of registers provides execution efficiency, although as an addressing mode the presence of registers means fewer addressing bits are needed in instructions—but at the expense of a larger instruction set and the code required to do the loading and storing operations.

An architecture which wishes to retain the two-level storage hierarchy characteristic of real hardware should examine carefully the question whether the registers should mirror the store or whether additional information kept in registers can be used to provide better function; for example, additional exponent range. In this case, making a relationship between high-level language code and the resulting operations is more difficult.

Note that we assume that basic addressing and indexing is accomplished with the help of registers, both for reasons of performance and to keep instruction lengths down, and that only the floating-point data resides exclusively in storage. Substantial opportunities exist to obtain performance gains with incremental hardware expenditure to perform addressing and storage operations in parallel with other sequencing operations as well as for instruction lookahead, pipelining, or other forms of logical overlap.

If the base architecture supports array manipulation in a reasonably atomic manner then reductive operations can exist with substantially more functional power than their naive programmed counterparts. The essence of what is required is best illustrated by example. One useful reductive operation is finding the largest element in an array. However, in many contexts in which the maximum is required the position at which is occurs is also important. Thus the most primitive operation should deliver as result not the value of the maximum but the value of the index at which the maximum is to be found (or the first such in case of duplication). If the value is required, the index should have been delivered in such a way that a simple array reference has already been set up. In the case of a two-dimensional array, a pair of indices should be produced. Finally, in the case of a two-dimensional array it is valuable to be able to treat a cross section (i.e., a particular row or column) as a one-dimensional array for the same operation. Cross-sections should also be feasible with the computational reductive operations such as sum. Support of these addressing modes for arrays of higher dimension may not be cost-justified, in which case a standard addressing mechanism for arrays of arbitrary dimension are reasonably cleanly handled. If two-dimensional array references are effected by specifying an array descriptor and two actual indices, the effective address computation which is customarily done in software will then be moved beneath the architectural interface. While this reduces the possibilities of obtaining very efficient code in special cases where an optimizing compiler can save much work using strength reduction and code motion, these operations can be reasonably efficiently implemented in hardware if some pre-computation is done when the index registers are loaded, taking advantage of the ability of hardware to perform operations in parallel and to keep alternate representations of quantities which may be used in different contexts.

3. TRAPS.

It is assumed that abnormal conditions which arise during execution because of data irregularities (e.g., invalid operand format, overflow, division by zero, etc.) will give rise to traps. One trap is associated with each condition, each trap may independently have a specified transfer-point, and each trap causes a default action to arise if a transfer-point has not been specified. When a trap is taken, information must be available at the transfer-point to be interrogated by the program at that point as to where the trap occurred and other information as appropriate to the trap; for example, if an invalid source operand was encountered which operand it was. From the transfer-point it must be possible to continue execution by returning to the instruction causing the trap with the trap still enabled for that instruction, by returning to the instruction causing the trap with the default action to be taken if the instruction again fails, by returning to the instruction following the one which trapped, by branching to some other legal point, or by branching to one of the operands of the trapped instruction, in case of a conditional branch instruction. The transfer-addresses must either exist on stacks or be available for program interrogation so that dynamic control of the trap interception protocol is efficient.

4. OTHER ARCHITECTURAL CONSIDERATIONS.

While we informally postulate the ability to mix precisions and the resulting conceptual use of generic operators, the utility of these powerful tools is substantially reduced unless subroutines can be produced to be used in the same way. This implies that programs must be able to ascertain the types and precisions of their parameters and make decisions accordingly. In particular, they must be able to obtain working storage whose details may not be known until the routine is invoked.

While this architecture can be implemented across a wide range of price and performance, it is important that it be economical to implement on a large, fast processor. To this end, it is assumed that the implementation of greatest importance will entail special-purpose hardware for key aspects of the decimal floating-point arithmetic and extensive microcode. In this environment adding additional function such as the Fortran elementary functions or the APL operator set is the most feasible. Complete implementation in hardware, while possible, may well not be cost-effective. Nor, possibly, would be a software implementation on general-purpose and non-sympathetic hardware. Thus, a floating-point simulator on a mini-computer or micro-processor along conventional lines may not be successful, if anything other than software function is considered.

5. CODING.

In the following formats, reference is made to sets of 3 coded decimal digits. By this is meant a ten bit field into which values between 000 and 999 have been encoded using the algorithm described in [1]. This encoding is extremely well suited to present digital logic technology, is fast, and is efficient of storage, making possible the packing of nine decimal digits and two signs into a thirty-two bit word, whereas ordinary BCD encoding could have accommodated at most seven digits.

6. BASIC FORMATS.

The following three formats are required to support short and long precision arithmetic adequately in high-level languages. Decoding of decimal digits occurs between storage and the floating-point unit, and coding takes place in the reverse direction. Invalid data exceptions are taken on the 24 invalid 10-bit coding combinations, on non-zero unnormalized representations whose exponent is other than the minimum, on non-standard zeroes, and on non-zero bits in formats with unused positions.

- Short precision (32 bits: 2 digit exponent, 7 digit fraction)
  - Sign in bit 0 — 0 for +, 1 for −
  - Exponent sign in bit 1 — 0 for +, 1 for −
  - Exponent and fraction digits in bits 2–31, represented as 3 sets of 3 coded decimal digits, of which the first two with the exponent sign represent a ten's complement exponent and the last seven represent the fraction; i.e.,

$$-100 \leq \textit{exponent} \leq 99$$
$$1.000\,000 \leq \textit{fraction} \leq 9.999\,999$$

  - The value 0 represented by all bits set to zero

- Long precision (64 bits: 3 digit exponent, 15 digit fraction)
  - Sign in bit 0 — 0 for +, 1 for −
  - Exponent sign in bit 1 — 0 for +, 1 for −
  - Exponent and fraction digits in bits 2–61, represented as 6 sets of 3 coded decimal digits, of which the first three with the exponent sign represent a ten's complement exponent and the last fifteen represent the fraction; i.e.,

$$-1000 \leq \textit{exponent} \leq 999$$
$$1.000\,000\,000\,000\,00 \leq \textit{fraction} \leq 9.999\,999\,999\,999\,99$$

  - Bits 62–63 unused (set to, and checked for, zero)
  - The value 0 represented by all bits set to zero

- Extended precision (128 bits: 4 digit exponent, 31 digit fraction)
  - → Sign in bit 0 — 0 for +, 1 for −
  - → Exponent sign in bit 1 — 0 for +, 1 for −
  - → Exponent and fraction digits in bits 2–121, represented as 12 sets of 3 coded decimal digits, of which the first four with the exponent sign represent a ten's complement exponent, the next thirty-one represent the fraction, and the last is set to and checked for zero throughout; i.e.,

$$\begin{aligned} -10000 &\le \textit{exponent} \le 9999 \\ 1 &\le \textit{fraction} \le 10 - 10^{-30} \end{aligned}$$

  - → Bits 122–127 unused (set to, and checked for, zero)
  - → The value 0 represented by all bits set to zero

7. EXTENDED FORMATS.

In addition, if the extended format is to be fully supported in high-level languages (especially with elementary functions and the like), then one of the following two formats is essential (although it need not be supported in the high-level language):

- "Working" precision (144 bits: 5 digit exponent, 37 digit fraction)
  - → Sign in bit 0 — 0 for +, 1 for −
  - → Exponent sign in bit 1 — 0 for +, 1 for −
  - → Exponent and fraction digits in bits 2–141, represented as 14 sets of 3 coded decimal digits, of which the first five with the exponent sign represent a ten's complement exponent and the last thirty-seven represent the fraction; i.e.,

$$\begin{aligned} -100\,000 &\le \textit{exponent} \le 99\,999 \\ 1 &\le \textit{fraction} \le 10 - 10^{-36} \end{aligned}$$

  - → Bits 142–143 unused (set to, and checked for, zero)
  - → The value 0 represented by all bits set to zero

- "Doubly-Extended" precision (256 bits: 5 digit exponent, 70 digit fraction)
  - → Sign in bit 0 — 0 for +, 1 for −
  - → Exponent sign in bit 1 — 0 for +, 1 for −
  - → Exponent and fraction digits in bits 2–251, represented as 25 sets of 3 coded decimal digits, of which the first five with the exponent sign represent a ten's complement exponent and the last seventy represent the fraction; i.e.,

$$\begin{aligned} -100\,000 &\le \textit{exponent} \le 99\,999 \\ 1 &\le \textit{fraction} \le 10 - 10^{-69} \end{aligned}$$

  - → Bits 252–255 unused (set to, and checked for, zero)
  - → The value 0 represented by all bits set to zero

8. PRINCIPLES OF ARITHMETIC.

General principles which apply to all arithmetic operations on scalar items are:

- Any source operand is considered a numerical *value* and nothing more. This value is exact. Its origin and representation are of no consequence and imply nothing of its history, future, "accuracy", "significance", or importance. In a particular departure from previous architectures, there are no "unnormalized zeroes"; the addition of zero to any other value leaves the latter value unchanged. The *representation* of a value may change when it is moved to a cell of different attributes, but the *value* will not change unless it is impossible to contain its representation in the storage cell for which it is destined.

- All arithmetic operations which do not result in a trap store the exact value of the result subject to a specified transformation which is applied only when the representation of the fraction will not fit into the space available. The storage rule which approximates values by shortening representations is called *rounding*.

- All arithmetic operations produce normalized results only, and the rounding rule applies only to normalized representations, which may require re-normalization subsequent to rounding. The unnormalized store operation may produce an unnormalized result with a minimal exponent.

- The basic (default) rounding rule in decimal floating-point arithmetic is the *unbiased round of magnitude, rounding ties to the nearest even*. This means:
  - If the digits to be discarded amount to more than half a unit in the last place retained, the magnitude of the last digit retained is increased by one.
  - If the digits to be discarded amount to less than half a unit in the last place retained, the digits retained are unchanged.
  - If the digits to be discarded amount to exactly half a unit in the last place retained, the digits retained are unchanged if the last digit retained is even, but the magnitude of the last digit is increased by one if it is odd.

- If, after rounding the fraction, the exponent of the value to be stored cannot be represented in the number of digits allocated, a special action is taken. The sign and rounded fraction as computed are stored with an exponent of zero in a designated cell of the longest floating-point format supported in the architecture. The true exponent is stored in a designated cell as a 15–digit signed decimal integer. Finally, a trap is taken to one of two previously designated points in the program, depending on whether the exponent is negative (underflow) or positive (overflow). At these points, in addition to inspecting and changing the designated cells, it must also be possible to ascertain the operation, operands, and target of the instruction which caused the trap to be taken.

- If a divisor with value zero is encountered, nothing is stored in the target nor in the cell designated for exponent underflows and overflows. Instead, a trap is taken to a point previously designated for zero divide exceptions. At this point it must be possible to ascertain the operation, operands, and target of the instruction which caused the trap to be taken.

- If any source operand contains invalid codes in the fraction digits, nothing is stored in the target nor in the cell designated for exponent underflows and overflows. Instead, a trap is taken to a point previously designated for decimal data exceptions. At this point it must be possible to ascertain the operation, operands, and target of the instruction which caused the trap to be taken. Such invalid codes may have been placed in the source for the explicit purpose of causing an interruption.

- Should zero be the result of an operation, the zero stored is represented by all bits set to zero. This departs from previous architectures in that the exponent is true zero rather than the minimum possible exponent. When zero is stored, an exception can never be taken. The value zero can result only from addition of terms of equal magnitude and opposite sign, in subtraction of equal operands, in multiplication when at least one factor is zero, and in division only when the dividend is zero and the divisor is not.

- Each operation which produces a rounded decimal floating-point result must be available with each of three rounding and storage options:
  - ordinary, unbiased round, as previously described, followed by storage, underflow, or overflow .
  - round to closest algebraically lower floating-point number, followed by storage of this value if within range, or the next lower representable number if not within range and possible, or overflow if all representable numbers are greater than the computed value
  - round to closest algebraically higher floating-point number, followed by storage of this value if within range, or the next higher representable number if not within range and possible, or overflow if all representable numbers are less than the computed value

  These three operations on the value x to a representation of n-digits will be respectively denoted $\boldsymbol{R}_n(x)$, $_{-}\boldsymbol{R}_n(x)$, and $_{+}\boldsymbol{R}_n(x)$.

9. EXAMPLES OF ROUNDING.

The following examples assume a seven decimal digit fraction and a two decimal digit ten's-complement exponent.

x	$\boldsymbol{R}_7(x)$	$_{-}\boldsymbol{R}_7(x)$	$_{+}\boldsymbol{R}_7(x)$
(value)	*(ordinary round)*	*(lower bound)*	*(upper bound)*
1.234567	$1.234567\cdot10^0$	$1.234567\cdot10^0$	$1.234567\cdot10^0$
1.2345678	$1.234568\cdot10^0$	$1.234567\cdot10^0$	$1.234568\cdot10^0$
−1.2345678	$-1.234568\cdot10^0$	$-1.234568\cdot10^0$	$-1.234567\cdot10^0$
9.9999985	$9.999998\cdot10^0$	$9.999998\cdot10^0$	$9.999999\cdot10^0$
9.9999995	$1.000000\cdot10^1$	$9.999999\cdot10^0$	$1.000000\cdot10^1$
$2\cdot10^{-123}$	*(underflow)*	0	$1.000000\cdot10^{-100}$
$-2\cdot10^{-123}$	*(underflow)*	$-1.000000\cdot10^{-100}$	0
$2\cdot10^{123}$	*(overflow)*	$9.999999\cdot10^{99}$	*(overflow)*
$-2\cdot10^{123}$	*(overflow)*	*(overflow)*	$-9.999999\cdot10^{99}$
0	0	0	0

Note that the directed rounding and storage rules provide guaranteed, unambiguous bounds to the original value and that the result of ordinary rounding either underflows or is identical to one of these bounds. Directed roundings never underflow.

It must be emphasized that rounding of any sort is considered an operation on a *value* which produces a storable *representation* and is applied immediately prior to the storage operation *only* if the representation of the value cannot itself be stored.

10. OPERATIONS ON SCALAR DATA.

In addition to the data formats described above and the operations to be described, the architecture contains seven special objects. Five *trap address cells* contain the addresses of locations to which branches are taken in the face of exceptions. These cells can be implemented in a number of ways. The simplest is to designate specific locations in low-order storage in which pointers to program labels are stored. (However, it is necessary to store into these locations from the problem program, so such an implementation should not be in a protected area of storage.) Another possibility is to keep registers which can be loaded and stored with problem program accessible instructions. An even more attractive prospect is a set of push down stacks, although the cost of this might not be justified. The exceptions which can arise are:

Invalid Data
Overflow
Specification
Underflow
Zero Divide

Whether these particular exceptions can be raised for other instructions outside the decimal floating-point area is an architectural question which cannot be answered here. There may be additional exceptions having to do with the specification, addressing, storage protection, etc. of the operands.

There are two special cells, registers, or pointers to user-supplied storage areas called the *Exception Fraction Cell* and the *Exception Exponent Cell*. The former is a floating-point datum of the longest architected precision; the latter is an eight byte fixed decimal datum (i.e., fifteen signed decimal digits). The same possibilities exist for the treatment of these cells as for the trap address cells. The following notation applies to the operation descriptions in this section:

T	floating decimal target of any precision
S, S1, S2	floating decimal sources of any precisions
L, L1, ...	program labels
N, N1, N2	integer sources
M	integer target
k	Index over rounding rules
rel	element from the set $\{=, \neq, <, \leq, \geq, >\}$
t	precision of T (number of digits in fraction)
F	Exception fraction cell
E	Exception exponent cell
L_I	Address in invalid data trap cell
L_O	Address in overflow trap cell
L_S	Address in specification trap cell
L_U	Address in underflow trap cell
L_Z	Address in zero divide trap cell

The following functions play a part in the operation descriptions:

$\boldsymbol{R}_n\ (x)$ denotes "round-to-closest" applied to x to give an n–digit representation.

$_-\boldsymbol{R}_n\ (x)$ denotes "round-to-lower-bound" applied to x to give an n–digit representation.

$_+\boldsymbol{R}_n\ (x)$ denotes "round-to-upper-bound" applied to x to give an n–digit representation.

$_k\boldsymbol{R}_n\ (x)$ denotes application of any of the three rounding rules above.

$\boldsymbol{C}_n$ denotes the chopping rule applied to x to give an n–digit representation.

If $x > 0$ then $\boldsymbol{C}_n\ (x) = {}_-\boldsymbol{R}_n\ (x)$.
If $x < 0$ then $\boldsymbol{C}_n\ (x) = {}_+\boldsymbol{R}_n\ (x)$.
$\boldsymbol{C}_n\ (0) = {}_k\boldsymbol{R}_n\ (0) = 0$.

fract (x) is the signed fraction in the normalized representation of x.

[***fract*** $(0) = 0$ and if $x \neq 0$ then $1 \leq |$***fract*** $(x)| < 10$]

exp (x) is the signed exponent in the normalized representation of x. [***exp*** $(0) = 0$]

[Therefore, if $f =$ ***fract*** (x) and $e =$ ***exp*** (x), $x = \mathrm{f} \cdot 10^e$].

$\boldsymbol{U}_n$ (F, E) is the gently underflowed, n–digit representation of $\mathrm{F} \cdot 10^{\mathrm{E}}$. [It is discussed more fully in the text.]

$\boldsymbol{N}\ (x)$ is the number of non-zero digits in the representation of x. $\boldsymbol{N}\ (0) = 0$, and for $x \neq 0$, $\boldsymbol{N}\ (x)$ is the smallest integer n for which ***fract*** $(x) \cdot 10^{n-1}$ is an integer.

The following sub-operation conventions are always applicable:

Rounding and Storage: All arithmetic operations are of the form "$\mathrm{T} \leftarrow {}_k\boldsymbol{R}_n\ (x)$". If the value of ${}_k\boldsymbol{R}_n\ (x)$ is out of the range of T then:

1) Assignment to T is not performed.
2) $\mathrm{F} \leftarrow$ ***fract*** $\{{}_k\boldsymbol{R}_n\ (x)\}$.
3) $\mathrm{E} \leftarrow$ ***exp*** $\{{}_k\boldsymbol{R}_n\ (x)\}$.
4) if $\mathrm{E} < 0$ then go to $\mathrm{L_U}$,
   else go to $\mathrm{L_O}$.

Operand Use: If a source operand is improperly formed in other than a Test operation, then:

1) No assignments are performed.
2) Go to $\mathrm{L_I}$.

Zero Divide: If a divisor of zero is encountered, then:

1) No assignment is performed.
2) Go to $\mathrm{L_Z}$.

Specification Exception: If the target of the Fraction operation is not as long as the precision of the source, or if the value of the third operand in the Round or Chop operations is negative or greater than the precision of the target, then:

1) No assignment is performed.
2) Go to $\mathrm{L_S}$.

The following operations are proposed for scalar arguments:

Operator	*Operands*	*Semantics*
Add$_k$	T,S1,S2	T $\leftarrow$ $_k\boldsymbol{R}_t$ (S1 + S2)
Sub$_k$	T,S1,S2	T $\leftarrow$ $_k\boldsymbol{R}_t$ (S1 − S2)
Compare (*rel*)	L,S1,S2	If S1 *rel* S2 then go to L
Magnitude$_k$	T,S	T $\leftarrow$ $_k\boldsymbol{R}_t$ (|S|)
Negative$_k$	T,S	T $\leftarrow$ $_k\boldsymbol{R}_t$ (−S)
Precision$_k$	T,S	T $\leftarrow$ $_k\boldsymbol{R}_t$ (S)
Round$_k$	T,S,N	T $\leftarrow$ $_k\boldsymbol{R}_N$ (S)
Chop	T,S,N	T $\leftarrow$ $\boldsymbol{C}_N$ (S)
Multiply$_k$	T,S1,S2	T $\leftarrow$ $_k\boldsymbol{R}_t$ (S1 · S2)
Scale$_k$	T,S,N	T $\leftarrow$ $_k\boldsymbol{R}_t$ (S · 10^N)
Divide$_k$	T,S1,S2	T $\leftarrow$ $_k\boldsymbol{R}_t$ (S1 / S2)
Test	L1,L2,L3,L4,S	If S is invalid go to L4 If S < 0 go to L1 If S = 0 go to L2 If S > 0 go to L3
Unnormalize	T	T $\leftarrow$ $\boldsymbol{U}_t$ (F, E)
Fraction	M,S	M $\leftarrow$ ***fract*** (S)
Exponent	M,S	M $\leftarrow$ ***exp*** (S)
Float$_k$	T,N1,N2	T $\leftarrow$ $_k\boldsymbol{R}_t$ (N1 · 10^{N2})
Non-zero-digits	M,S	M $\leftarrow$ $\boldsymbol{N}$ (S)

11. REDUCTIVE OPERATIONS.

General principles which apply to the reductive operations (sum, product, inner product, polynomial evaluation, and search for maximum, minimum, maximum magnitude, or minimum magnitude) are:

- The reductive operations behave identically to a sequence of elementary operations in which intermediate results are held at the longest precision supported and with essentially infinite exponent range so that an intermediate result never underflows or overflows. After each elementary operation prior to the last, one of the three rounding rules is applied to round the intermediate result to the length of the longest precision. After the final operation, the same rounding rule is applied to round the final result to the length of the target. The computed exponent is then tested against the range allowed for the target. If an overflow or underflow condition then exists, it is treated in the same way as for elementary operations.

- If a source operand contains invalid codes in the fraction digits, the intermediate result must be saved in the designated cells and format used for overflow and underflow and a trap taken to the point designated for decimal data exceptions. At this point it must be possible to ascertain the operation, operands, index of exceptional operand involved, and ultimate target of the instruction causing the exception.

For example, the dot (inner) product operation applied to vectors X and Y of length n should be equivalent to the following sequence, where T and U are internal temporaries with precision equal to the highest precision supported in the architecture and enough exponent range not to underflow or overflow:

```
Multiply        T,X(1),Y(1)
Multiply        U,X(2),Y(2)
Add             T,T,U
Multiply        U,X(3),Y(3)
...             .....
Multiply        U,X(n),Y(n)
Add             Target,T,U
```

Invalid decimal digit exceptions can arise in the Multiply operations, but overflow or underflow can arise *only* at the final Add operation. No other exception can arise.

12. ARRAY OPERATIONS.

The following general principles apply to array operations (if supported):

- Array operations behave identically to a sequence of elementary operations. The rules which apply to elementary operations as regards operand values, rounding rules, and exception handling apply equally to array operations.
- If a source operand to an array operation is a scalar, its value is prefetched and preserved before any storage is done.
- If an exception occurs in the processing of an element in a array operation, it is treated as if it were the corresponding exception arising in the elementary operation. At the point of exception, a Resume instruction must start afresh the elementary operation which caused the interruption. Alternatively, it must be possible to restart with the elementary operation *following* the cause of the interruption, or to terminate the array operation entirely.

For example, if X and Y are vectors of length *n* and C is a scalar, the multiplication of X by C to give Y should be equivalent to the following sequence, where A is an internal temporary with enough precision to hold C exactly and enough exponent range so that normalization of C will not lead to underflow:

```
L(0):      Precision   A,C
L(1):      Multiply    Y(1),X(1),A
L(2):      Multiply    Y(2),X(2),A
.....      ....        ..........
L(n):      Multiply    Y(n),X(n),A
L(n+1):    (Next Instruction)
```

An invalid digit exception can occur at L(0), in which case it should be possible to resume execution at either L(0) or L(*n*+1). The multiply instruction at L(*k*) may lead to invalid digit exceptions, underflow, or overflow. In each case it must be possible to resume execution at any of L(0), L(*k*), L(*k*+1), or L(*n*+1).

REFERENCES.

[1] Chen, T. C. and Ho, Irving T. "Storage-Efficient Representation of Decimal Data." *Communications of the ACM 18* (January, 1975), 49–52.

[2] Knuth, D. E. "An Empirical Study of FORTRAN Programs." *Software Practice and Experience 1*, (1971), 105–133.