A UNIFIED NUMERIC DATA TYPE IN PASCAL

Peter Kornerup
Department of Computer Science*
University of Aarhus
Denmark

Abstract

It is proposed to substitute the standard data type *real* of a high level language, with a unified data representation which can include a variety of interpretations as well as formats, in order to allow experiments with an environment containing a spectrum of non-standard arithmetics, as well as standard.

The implementation of a system is described where syntatic extensions to a language are made to support a microprogrammed virtual arithmetic unit which can treat variants such as integers, normalized, and unnormalized floating point numbers and intervals, within a unified representation.**

More specifically Pascal is chosen as the base language, because it allows the user to define new data types, and the extension then mainly consists in substituting the simple (unstructured) data type *real* with a skeletal structured type (which will be called *numeric*).

The system is intended to be implemented on a microprogrammable processor (called MATHILDA) with a 64 bit wide datapath. The language Pascal has already been partially implemented based on a stack machine specifically designed for that language, and realized by interpretation in microcode. The present compiler was constructed with the aid of a parser-generator system, which will allow the language extensions to be made with a moderate effort.

1. Introduction: Number Representation And Arithmetic Operators

The available number representations in a computer, or rather the interpretations of bit strings upon which arithmetic operations are based, are usually restricted to integers and normalized floating point numbers. The latter may typically exist in single and double precision representations. A few high level language systems exist where this selection has been extended. Normally such systems have been realized by preprocessing and/or compiling extended high level languages into the fixed host machine language. The interpretation of non-standard number representations is then performed by operators as in-line code or subroutines. Such systems have been characterized by the fact that only a single new number representation has been added (say interval arithmetic, rational arithmetic or the like), the interpretation of which could be implemented in a straightforward way in machine language code of the host machine. A few systems have been reported where non-standard operand representations have been realized in a host machine[1,2], other experimental systems have been reported which rely on an extensive software interpretation in standard machine language which made their use in practical applications prohibitive .

An ideal system would provide the user with capabilities of defining the number representations as well as the operators, as syntactic and semantic extensions to a suitable base language. But this total generality as provided in extensible languages like Algol 68, can mostly only be of an academic interest. This is first of all due to the fact that the extensions have to be expressed in the base language and then compiled to or interpreted in the host machine language. Since realization of an arithmetic, which is not readily implementable by means of number representations or operations of the host machine, requires a great deal of field interpretations and manipulations, then such realizations are bound to be hopelessly inefficient.

This paper presents a compromise to the above mentioned complete generality. It is based on a quite limited language extensibility (within a class of number representations and interpretations) in combination with a generalized arithmetic unit (realized in microcode to achieve efficiency).

. The language Pascal[3] has been chosen as a base for the language extensions, because of its facilities for type definitions, which allows the user to define new data types out of existing ones. The extension necessary then mainly consists in providing a skeleton of a new (structured) arithmetic data type.

It is the author's opinion that operators in a high level language should be polymorphic, and that implicit type conversions should take place automatically according to well defined rules. There is, and will probably forever be, a standing debate about these questions; presumably this is a matter of taste. Personally, this opinion is based on the achieved natural shorthand notation, and the fact that it is possible to control the numeric behavior of an algorithm, by means of explicit conversions, when necessary.

*Currently at University of Southwestern Louisiana.

**The work on the arithmetic unit is being supported by NATO Grant 755.

***The construction of MATHILDA has been supported by the Danish Research Council, Grant No. 1546-511.

2. Language Extensions And The Numeric Data Type

In this section proposals for extensions to the language Pascal will be given, together with a brief description of the supporting arithmetic unit called UNRAU (Unified Numeric Representation Arithmetic Unit). The emphasis will be on the language point of view, and the UNRAU will be described in more detail in another paper.[4]

2.1 The Interpretation of the Proposed Number Representation

The arithmetic unit operates on operands represented as 5-tuples (t,a,e,f,r). The (e,f,r) triple may be interpreted in four different ways, which are described in the table below:

Name of representation (qualifier)	Interpretation of (e,f,r) in the representation
fixed:	the value represented is f, where the f-field is interpreted as a sign-magnitude integer, the e and r fields are not used.
normalized:	the value represented is f * 2^e, where f and e fields are interpreted as sign-magnitude integers. The f-field is assumed to be normalized, i.e., the most significant bit of the f-magnitude fields is a 1 if the field is non-zero. The r-field is not used.
unnormalized:	the value represented is f * 2^e, where the interpretation is the same as for normalized numerics. The f-field is not normalized, and the least significant bit of f is considered as being the last "correct" bit. The r-field is not used.
centered:	the "value" represented is the interval [f-r, f+r] * 2^e, where e and f are sign-magnitude integers, and r is the magnitude of the unsigned integer value of the r-field. The f-field is not normalized.

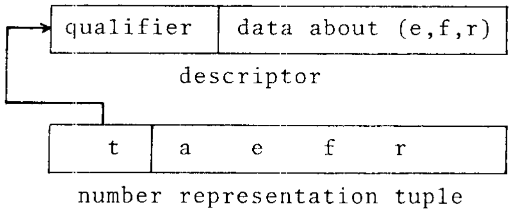
Within each of these four interpretations a non-representable (augmented) form may be introduced as the result of an arithmetic operation, or whenever a mapping into the finite representation is impossible. The a-field provides an escape bit for such augmented forms. If the a-field has the value "normal" indicating a representable number, then the interpretation of the (e,f,r) triple will be as described above. But whenever the a-field indicates that a non-representable value has been introduced, the (e,f,r) triple will be interpreted as one field "nrv" which can assume the following values:

negmax, negmin, posmin, posmax, undef

Within the four different representations these values may be interpreted in the following way:

qualifier nrv	fixed	normalized & unnormalized & centered
negmax	too large neg. value	too large neg. value(exp. overflow)
negmin	not applicable	too small neg. value(exp. underflow)
posmin	not applicable	too small pos. value(exp. underflow)
posmax	too large pos. value	too large pos. value(exp.overflow)
undef	undefined value	undefined value

The first field in the 5-tuple (t,a,e,f,r) is a tag-field which provides a pointer to a descriptor containing information about the representation:



The descriptor thus contains the qualifier specifying which of the four interpretations is to be used, and furthermore it contains data about the positions and sizes of the e,f, r-fields. This data will be referred to as the "format information".

2.2 The Numeric Data Type Definition Added To Pascal

The new data type is to be included in the language Pascal in such a way that the syntax of algorithms, and specially that of expressions, can be kept as close as possible to the usual. The main extensions will be to the type definitions and variable declarations. A new skeletal (structured) data type called numeric will substitute the standard (scalar) data type real. Furtherfore an extension is made by including a type conversion operator "↓", and finally a syntax of litterals (data constants) is added.

In the following type definitions and variable declarations are described in terms of Pascal itself. An implementation may not follow exactly this description, since part of the implementation will be realized by the compiler (static information) and part of it will be realized in the arithmetic unit (dynamic information). A proposal for the handling of descriptors with this separation will be given in section$^{3.1}$.

It is assumed that the following declarations and type definitions are provided by the system globally to form a foundation of the data type definition facility:

A class-declaration:

```
descriptor:
  class of
  record
     qual : qualifier;
     exs  : 0..N; (size of exponent field)
     frs  : 0..M; (size of fraction field)
     ras  : 0..K; (size of radius field)
  end
```

where N, M and K are implementation dependent non-negative constants, possibly N+M+K is restricted by some upper bound W (by wordsize and other fields). In the following it will be assumed that the base of the arithmetic is 2, and that specification of field sizes will be given as numbers of bits.

Furthermore the following three scalar type definitions are provided system-globally:

```
     type qualifier = (fixed, normalized,
               unnormalized, centered);
     type non_representable_values = (negmax,
               negmin, posmin, posmax,
               undef);
     type sign = (pos, neg);
```

A numeric data type may then be introduced in a program with a type definition:

```
 type<identifier>=numeric<list>of<qualifier>
```

where <qualifier> has to be substituted by a constant of type qualifier and <list> by a list of three integer constant which lie within the ranges specified in the global class descriptor.

The type definition:

```
     type: <identifier>=numeric n,m,k of q
```

is considered equivalent to the definition:

```
type:  identifier =
record
  tag:↑descriptor; (comment: "↑" means "point-
                            er to")
  case a:(normal, augmented) of
  normal:(e:record s: sign; val: 0..(2^{n-1}-1)
                            end;
          f:record s: sign; val: 0..(2^{m-1}-1)
                            end;
          r:record val: 0..(2^{k}-1) end);
  augmented: (nrv: non_representable_values);
end.
```

This type-definition is equivalent to a record-definition, which allows the user to have access to the fields of the numeric. Simultaneously the definition involves an allocation of an element of the class descriptor, and an initialization of the descriptor according to the parameters of the type-definition.

One difference is that the user can only read the tag field and the descriptor pointed to, but not to redefine these. This is caused by the fact that the descriptor has to be protected, and is "hidden" in the firmware.

Variables may now be declared in the standard Pascal syntax. They are system initialized at declaration time with the non-representable value "undef".

Example 1

In this example we will show how a data type may be defined, and how variables of the data type may be declared. As the data type has been chosen a standard floating point representations, which will be identified by the name real:

```
   type real = numeric 8, 32, 0 of normalized;
```

The allocation and initialization of the corresponding descriptor corresponds to the execution of the following piece of code (including the declaration of an anonymous variable to hold the pointer to the descriptor):

```
  var real_tag: ↑ descriptor;
  alloc (real_tag); (allocates a descriptor
                     and assigns to real_tag
                     a pointer to the descrip-
                     tor)
  real_tag ↑. qual:=normalized;
  real_tag ↑. exs:=8;
  real_tag ↑. frs:=32;
  real_tag ↑. ras:=0;
```

This initialization is of course performed automatically. In this description it is assumed that it takes place at run-time, but it may be performed at compile-time (see section 3.1).

Now the declaration of a variable can take place, this is expressed as:

```
  var x : real;
```

The system will then allocate space for x (on the run-time stack) and perform an initialization of x, corresponding to the execution of the following code:

```
  x.tag:=real_tag; (initializes the tag-field
                    with pointer to descrip-
                    tor)
  x.a:=augmented; (initializes the value of x
  x.nrv:=undef;    to be "undefined")
```

Since we have not yet introduced syntax and semantics of expressions and litterals, we will instead demonstrate some of the non-standard access-facilities which has become possible in a system like this.

If the user wants to change the value of x into abs(x) he may write this as:

```
  x.t.s := pos;
```

which, irrespective of its previous value, sets the sign of the fraction to be positive. Similarly the predicate:

```
  x.e.s = neg
```

will be true if the sign of the exponent is negative. Of course this has only a meaning if x has a representable value, which may be tested by the predicate:

x.a = normal.

If this predicate has the value false (⇒ x.a = augmented) then the predicate:

x.nrv in [negmax, posmax]

(in [ ] means set membership) will tell whether an overflow situation has occurred in the algorithm leading to the value of x.

Finally since the tag-field provides a pointer to the descriptor of x, it is possible to get access to information on the type of x. Thus:

x.tag ↑.qual

will give the qualifier of x.

2.3 Explicit Type Conversions

To control the numeric behavior of an algorithm it might be necessary to force explicit conversions upon expressions. This is achieved by the introduction of an additional operator "↓" which, combined with a numeric data type, forces a conversion into a value of the <type> specified:

<type> ↓ <expression>

irrespective of the numeric data type of <expression> . Furthermore <expression> is allowed to be standard integers of Pascal, and exponent, fraction and radius-fields of numeric data types. The operator "↓" is assumed to have the highest priority among operators. Unlike implicit conversions in expressions (see section 2.5), the conversion which may be performed in connection with an assignment operator, is considered semantically equivalent to an explicit conversion. If, as in example 1, the variable x has been declared to be of the type real, then the assignment:

x := <expression>;

is equivalent to:

x := real ↓(<expression>);

2.4 Data Constants (litterals)

The usage of litterals in programming systems for numerical algorithms, may often require litterals to be expressed in the same base as that of the internal representation, to avoid base conversion errors. In the system presented here this is not necessary since assignments may be made (expressed as litterals in base 10) to the individual fields of a numeric variable. No conversion error will be introduced because the fields are interpreted as integers, but of course representation errors may still occur.

If no special precautions have to be taken, the above method of specifying data constants is too clumsy, and a more natural way of expressing a constant as a litteral is needed.

Assuming that we express the syntax of standard Pascal (and Algol 60) real's as:

$\pm dd.ddd_{10}\pm dd$

a similar notation is proposed for litterals of the "normalized" types.

This notation may then be extended to litterals of the "centered" type as:

$\pm dd.ddd[dd]_{10}\pm dd$

where the integer value of the number in the (square) brackets represents the radius, and the remaining part of the litteral represents the center. The position of unity in the center when interpreting the radius, is obtained by neglecting the decimal point.

In the case of "unnormalized" numbers it then seems natural to use a similar notation (to indicate an uncertainty beyond the last digit represented) in the following way:

$\pm dd.ddd[\ ]_{10}\pm dd.$

The empty brackets then only serves to indicate the unnormalized representation of the fraction, as opposed to the notation for "normalized" litterals.

Finally, since a notation for the "fixed" data type is needed which differs from that of standard integers of Pascal, the notation:

ddddd[ ]

will be adopted for "fixed" litterals.

Notice that the brackets are then consistently being used where ever the fraction is represented right-adjusted (unnormalized) in the f-field.

The proposal syntax of litterals uniquely determines the qualifier of the data constant expressed, but contains no information about the format of the internal representation. Hence the data type of the litteral is not uniquely determined by itself, and further information is needed in places where a strict type-matching is required (see section 2.7). Explicit type conversions may then be used to have the compiler represent the litteral as the specified type.

If, on the other hand, the precise data type of a litteral is of no concern the system will provide four different global predefined data types to be used by default, one for each qualifier.

Example 2

The litteral 3.141529 will be interpreted as a "normalized" constant, but its internal representation will depend on the context in which it is written in a program. Assuming that the variables x and y has been declared of the type real, as defined in Example 1, then in the code of the statements

x := 3.141529 or in x := y*real ↓ 3.141529

the litteral will be represented as being of the data type real. But in the statement:

x := y* 3.141529

form is introduced).

From normalized into unnormalized and centered:
The e and f-fields are copied, the r-field is set to zero.

From unnormalized into fixed:
The nearest integer value is chosen (if representable, otherwise an augmented form is introduced.

From unnormalized into normalized:
By standard normalization.

From unnormalized into centered:

The new fractional part will be 2 * f if possible, otherwise f; and the r-field will be set to 1.

From centered into fixed or normalized:
Gives the result of the center interpreted as an exact quantity.

From centered into unnormalized:
The f-field is shifted entier ($\log_2 r$) +1 places to the right, the e-field is adjusted accordingly, and the r-field is set to zero.

Example 4

Assume that we want to use the value of x, which has been declared real (like in Ex. 1), as the value of the center of some variable a which has been declared as type interval, this type being defined as:

type interval = numeric 8, 32, 16 of centered.

Furthermore we want to have the radius to reflect a relative accuracy in x of say 10^{-6}.

This can be accomplished by the following code:

a:=x; (this assigns the value of x to the center of a, and sets the radius to zero, notice that this is an explicit conversion)
a.r.val:=integer ↓(x.f. val * 10^{-6});

where integer is the standard integer type of Pascal. The assignment of the center might also be expressed as:

```
a.e.val:=x.e.val; a.e.s:=x.e.s;
a.f.val:=x.f.val; a.f.s:=x.f.s;
```

as the format of these fields conform, and this copying is by definition the conversion from "normalized" to "centered".

2.7 Parameter Passing and Type Checking

Pascal allows for parameters to procedures and functions to be passed as const or as var, which corresponds to "by value" and "by reference" respectively. Standard Pascal requires a strict formal parameter-actual parameter correspondence with respect to type, the only exception is when an integer expression is passed for a real formal const parameter.

We will allow formal parameters to be specified as numeric, meaning that expressions of any defined numeric type may be passed as the actual parameter. The specifier numeric may be used for those parameters for which no further type specification is wanted,i.e. for which the type is only restricted to be of a numeric type, but otherwise it is equivalent to the specification const (i.e. parameters are passed "by value"). But notice that any expression passed as an actual parameter is required to have a well defined type, and a normal composite expression has no such type, unless explicitly stated.

Formal parameters of numeric data types may also be specified as const or as var, where the specification as usual includes the precise data type.

Example 5

Using the data types real and interval of Ex. 1 and 4, a procedure heading might look like:

```
procedure proc (numeric a; const b : real;
   var c : interval);
```

meaning that for a any expression of type real or interval may be passed, for b any expression of type real, and for c any identifier of type interval.

Although required by the language definition, to the author's knowledge no Pascal implementation does complete type checking, as usual the run time check on actual procedure parameters is left out. This proposed system can easily be implemented with the static part of the type checking performed by the compiler, and with the run time check on parameters of numeric data types performed by the arithmetic unit, since the data itself carries type information.

The reason for allowing parameters to be passed as numeric is that a too rigid type-match requirement would restrict the user from writing general purpose subroutines. If not allowed "input" parameters would always be restricted to specific data types. Unfortunately the restriction still applies to structures (e.g. arrays) with elements of numeric types. The reason is that there is no way of specifying in Pascal the structure of a parameter, without having to specify the type of the basic elements of structure. The inclusion of this facility would hence require major syntactic changes, and has therefore been left out.

3 Implementation

The basic idea of this system is to extend a given language with a few new constructions, whose implementation only requires minor modifications to the compiler. This is possible because normally the compiler does not have to be concerned with the specific data types within the class of numeric types. The data itself carries type information, and the underlying arithmetic unit deals with the problems concerning conversions in expressions.

As usual, an implementation can take advantage of the possibilities of optimization at compile time instead of run time. The previous description of the handling of descriptors assumed that everything took place at run time (dynamically). In the following a more realistic description will be given assuming that a block oriented stack machine is available, followed by a section about an actual proposal for such an implementation.

3.1 Compile Time Handling of Descriptors

Upon a type definition of a numeric type, a descriptor is constructed using the information from the definition. The descriptor constructed can then be treated as a data-constant, so that it is accessible on run time. A tag-value then has to be associated, and inserted in the symbol table in the entry corresponding to the name of the type defined. The tag-value is constructed as an offset to a base for the procedure being compiled, in a run time descriptor stack.

Furthermore, the tag-value is used to construct the initial value, as a data-constant, to be used in initialization of variables later declared of that type. On declaration of variables, space is allocated and code for the initialization is generated. When variables are referenced, an ordinary (block number (bn), ordinal number (on)) addressing can be used in the load instructions. In storing, the compiler can supply the tag-value, so that a read of the location to get the tag will be superfluous. On exit of a procedure a de-allocation of descriptors has to be performed, just as the ordinary de-allocation of local variables.

3.2 Run Time Handling

When a procedure is entered the actual loading of descriptors onto the descriptor stack is performed. This stack is maintained as the ordinary run time stack, i.e. a display may be used to provide pointers to visible descriptor stack frames. On entry initial values of variables are loaded into the run time stack.

Part of load and store instructions will be to use the bn from the (bn, on) address, to form the (bn, tag) address for referencing the descriptor.

The type conversion instruction has to have the (bn, tag) as an argument.

3.3 A Microprogrammed Supporting Stack Machine

The system is intended to be implemented on a combination of two microprogrammable processors, RIKKE with a 16-bit wide data path[5] , and MATHILDA which has a 64-bit wide data path[6] , together with a 64-bit wide memory. A compiler for the language Pascal has been implemented by means of a parser generator system called BOBS[7] , the compiler produces code for a block structured stack machine (called the P-code machine[8]). A microprogrammed interpreter for P-code has been written, which realizes the P-code machine in RIKKE and the 64-bit wide store, except for the real arithmetic of standard Pascal. The UNRAU is then to be realized in MATHILDA, together with the run time environment for descriptors (the descriptor stack).

RIKKE is then responsible for instruction fetch and decoding of the P-code instructions. Some instructions are executed in RIKKE, while others (those concerning numeric data type operands) are executed in MATHILDA. Furthermore, RIKKE controls the wide store which contains the ordinary run time stack of the P-code machine. The UNRAU and its immediate environment (the descriptor environment) is thus isolated as a self-contained functional unit, operating as a stack machine.

4 Evaluation

A proposal for some language extensions to support a non-standard arithmetic unit has been described, which will allow an efficient implementation intended for quite large scale experimental numerical calculations. The proposal is not intended to be a final answer to "all the dreams of a numerical analyst", but only to provide an experimental system for a restricted set of non-standard arithmetics, and to test a set of language facilities for the access of such arithmetics.

The language Pascal was chosen as a base because its type definition facility provides a tool for a certain amount of extensibility, within a class of data representations. The class chosen could have been quite different, as well as other decisions about the representations.

One of the implications of the choice of Pascal has been that the representation of data types (the formats) has to be known at compile time because Pascal is a "static" language in the sense that the storage requirement of a procedure activation is fixed and also that a variable never changes its type during run time. The system proposed could have looked quite different without these implications from the language, in fact because of this staticness tag fields are unnecessary.

Another implication of Pascal concerns the handling of augmented forms. The system proposed leaves it up to the user at critical points to test whether augmented forms has been introduced. The UNRAU raises a flag at the point where such a situation occurs, but the language provides no facilities for dealing with such flags (no ON-condition facility like in PL/1). Without having to introduce such language constructs, the only possibility left are (1) to abort program execution, or (2) to continue operations on augmented forms. The latter approach was chosen, in combination with the possibility of testing for such forms.

the litteral will be represented in the default data type corresponding to the qualifier "normalized".

2.5 The UNRAU Arithmetic Unit

Although the UNRAU is described elsewhere[4] , a brief description of it is necessary to understand its interface to the language.

The UNRAU is encapsulated in an environment where descriptor are being stored and looked up before presentation. The UNRAU is assumed to evaluate expressions presented to it in reverse polish form (postfix notation), i.e., that the unit itself is a stack machine with an internal storage in the form of a stack. An advantage will be that intermediate results may be kept internally on the stack in "maximal accuracy", which means that no format specifications are needed for temporary results. If on the other hand the user wishes to force such temporary results into specific format restrictions, an explicit conversion capability will provide for this possibility.

A few comments about the execution of operations is given below to illustrate the behavior of the UNRAU.

A) A load instruction decomposes the packed number representation, and then pushes the individual fields (a,e,f,r) onto the stack, along with a modified descriptor containing only the qualifier. This representation of operands will be called the internal UNRAU representation.

B) Standard operators presented to the UNRAU manipulate the top stack elements, possibly after initial implicit type conversions (see section 2.6). The operators leave their results in the internal UNRAU representation on the stack, and do not use format information, but retains "maximal accuracy".

C) Conversion operators are presented along with a complete descriptor of the data type wanted, and converts the top element of the stack into a representation of the number in the data type. The result will, although the format information is being used during the conversion, be delivered in the internal representation on the stack.

D) A store instruction (for output from unit) will push the (unpacked) top element off the stack, perform an explicit type conversion, pack it together, and then finally deliver it as output from the unit. The store instruction thus needs a complete descriptor of the output wanted.

Notice the difference in the treatment of the implicit type conversions interior to expressions, which need not be concerned with format information, since conversion is from internal to internal representation, and the conversion performed in connection with assignment. Notice also that the latter conversion is dictated by the type of the variable on the lefthand side.

Example 3

Assume that x, y, z, t, r has been declared real as in Example 1, and that: $x = y = 2^{100}$ and $z = 2^{80}$. Hence x, y and z are representable (the maximal exponent is 127), and x * y is not representable in the real data type, but x * y/z will be.

Hence:

```
 t := x * y ⇒ t will get the "value" posmax
 r := t/z   ⇒ r will get the "value" undef
and,
 r := real ↓ (x * y)/z ⇒ r will get the
                          "value" undef
```

Now assuming that the internal UNRAU representation allows exponents beyond 200, then:

r := (x * y)/z = r will get the value 2^{120}.

2.6 Rules of Conversions

Conversions in arithmetic expressions are performed automatically so that both operands of a dyatic operator have the same qualifier, which also will be that of result. The conversions will follow a simply priority scheme as expressed in the following table:

b a	fixed	norm.	unnorm.	centered
fixed	fixed	norm.	unnorm.	centered
norm.	norm.	norm.	unnorm.	centered
unnorm.	unnorm.	unnorm.	unnorm.	centered
centered	centered	centered	centered	centered

Qualifier of the Converted Operands, a and b, and the Result c, In Mixed Mode Operation, C ← a op b

Conversion of an augmented form will, irrespective of which one, always give as the result the form "undef" of the type wanted. Similarly if an augmented form enters a computation, the result will always be "undef".

The principles of conversions between internal representations of representable values are given below:

From fixed into all other representations:
The f-field is copied, e and r are set to zero. If the conversion is into normalized, the f-field is left shifted until it is normalized, and the e-field is adjusted accordingly.

From normalized into fixed:
The nearest integer value is chosen (if representable, otherwise an augmented

Acknowledgments

The author wants to express his gratitude towards the Danish Research Council which supported the development of the MATHILDA Processor. The author further expresses his gratitudes towards NATO, Division of Scientific Affairs, Bruxelles, which has supported the work on the arithmetic unit. Finally I am sincerely indebted to Bruce D. Shriver, my collaborator on the arithmetic unit, whose comments to the presentation of this paper have been invaluable.

References

1. Goldstein, M.: Significance Arithmetic on a Digital Computer, CACM6, p. 111-17.

2. Ashenhurst, R. L.: The Maniac III Arithmetic System, SJCC21, p. 195-202.

3. Wirth, N.: The Programming Language Pascal, Acta Informatica 1, 35-63.

4. Kornerup, P.; Shriver, B. D.: UNRAU- A Unified Numeric Representation Arithmetic Unit, this conference.

5. Staunstrup, J.: A Description of the RIKKE1 System, DAIMI PB25, Dept. of Computer Science, University of Aarhus, Denmark.

6. Kornerup, P.; Shriver, B. D.,: An Overview of the MATHILDA System, SIGMICRO Newsletter Jan. 75, Vol. 5, no. 4.

7. Kristensen, B. B. et al: A Pascal Environment Machine (P-code), DAIMI PB28, Dept. of Computer Science, University of Aarhus, Denmark.

8. Kristensen, B. B. et al: A Short Description of a Translator Writing System (The BOBS System), DAIMI PB47, Dept. of Computer Science, University of Aarhus, Denmark.