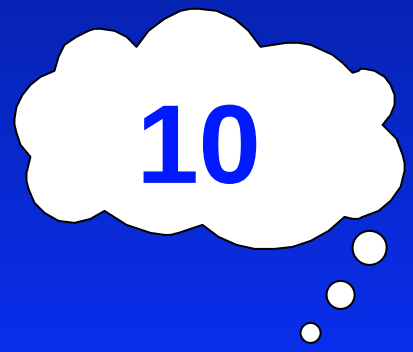
Rexx Arithmetic
- more than just numbers

Mike Cowlishaw

IBM Fellow

http://www2.hursley.ibm.com

mfc@uk.ibm.com

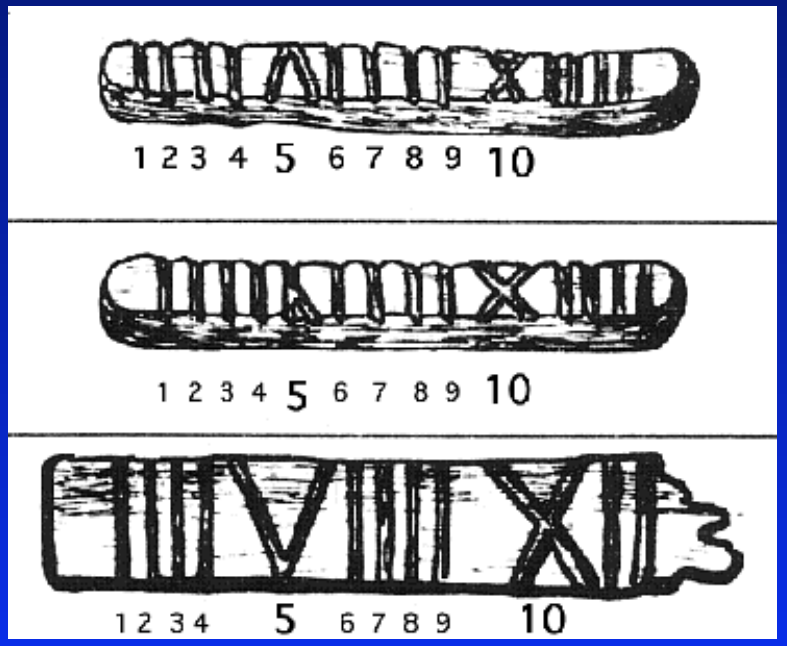


Overview

- Rexx arithmetic
- Current status
  - Databases
  - Other languages
  - Hardware
  - Standards
- Rationalizing decimal arithmetic
- Questions?

Why is decimal arithmetic important?

- Decimal arithmetic represents numbers in base ten, so uses the same number system that people have used for thousands of years



- Pervasive for financial and other commercial applications; often a legal requirement
- 55% of numeric data in commercial databases are decimal (and a further 43% are integers)

The trouble with binary

Decimal:

$$0.1 = 1/10 = 1 \times 10^{-1} = 1E\text{-}1$$

Binary:

$$= 0.0001100110011...$$

$$= 1/16 + 1/32 + 1/256 + 1/512 + 1/4096 + 1/8192 + ...$$

Repeated division by 10; what users expect:

Decimal
9
0.9
0.09
0.009
0.0009
0.00009
0.000009
9E-7
9E-8

Repeated division by 10; what they get:

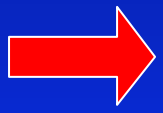
Decimal	float (binary)
9	9
0.9	0.9
0.09	0.089999996
0.009	0.0090
0.0009	9.0E-4
0.00009	9.0E-5
0.000009	9.0E-6
9E-7	9.0000003E-7
9E-8	9.0E-8

Another example

- What does the following code segment display?

```
double a=1.00D, b=0.10D;

System.out.println(a);
System.out.println(b);
System.out.println(a/b);    // divide
System.out.println(a%b);    // remainder
```



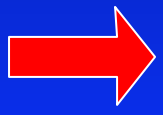
```
1.0
0.1
?
?
```

Another example

- What does the following code segment display?

```
double a=1.00D, b=0.10D;

System.out.println(a);
System.out.println(b);
System.out.println(a/b);    // divide
System.out.println(a%b);    // remainder
```



```
1.0
0.1
10.0
?
```

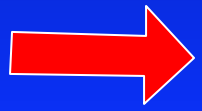
Another example

- What does the following code segment display?

```
double a=1.00D, b=0.10D;

System.out.println(a);
System.out.println(b);
System.out.println(a/b);   // divide
System.out.println(a%b);   // remainder
```

```
1.0
0.1
10.0
0.09999999999999995
```



(almost 0.1!)

When to use decimals?

- Binary floating point is currently essential only where performance is the overriding concern
  - for example, matrix inversions
  - for these cases, use C or Java **`float`** and **`double`** types
- In all other cases, use decimal arithmetic
  - Rexx, Object Rexx, and NetRexx all do this for you by default

Rexx arithmetic, 1979-1999

Rexx 1.0 [May 1979]

- 'Minimal' arithmetic – minimum necessary to be useful (loop counters, *etc*.)
- Integers only
- This was a 'holding' implementation, while other parts of the interpreter were designed and implemented

Rexx 1.10 [January 1980]

- Plain decimal arithmetic (no exponents)
- Up to 9 digits after the decimal point
- Precision of result is determined by the more precise of the two terms involved in an operation

8/3 → 2 8/3.00 → 2.67

- Worked well, but results could often surprise

Rexx 2.50 – the 'new' arithmetic [May-July 1981]

- Developed primarily by e-mail
- Initially controversial (because it changed the behaviour of existing programs)
- Widely discussed and researched (REX *[sic]* was in use in 43 countries by then)
- Essentially the same as the arithmetic in *"The Rexx Language"* book (1985 & 1990)

The choice of arithmetic

- The principle:

  "REX arithmetic attempts to carry out the usual operations in as 'natural' way as possible. What this really means is the rules which are followed are those which are conventionally taught in schools and colleges."

  (7 Oct. 1981)

Rexx arithmetic [1]

- Full-function decimal floating point arithmetic
- Preserves mantissa length, etc. For example, 1.20 x 2 gives 2.40 (not 2.4)
- Exact representations, as expected (0.1)
- Precision is user-selectable (**`numeric digits`**)
- Exponents from E-999999999 through E+999999999

Rexx arithmetic [2]

- Robust: all ill-defined or out-of-range results raise errors
- Integers are a seamless subset of all numbers
- Evolved over 18 years, based on user feedback and requirements (including mathematicians, experts in data processing, financial users, *etc*.)
- Numerous public-domain and commercial implementations exist

ANSI (X3-J18) refinements

- Trigger to exponential notation after 0.000001 (not dependent on DIGITS setting for numbers less than one)
- LostDigits condition (raised if input data too precise)
- Input data rounded to DIGITS (not DIGITS+1)
- Published as ANSI X3.274-1996 (see www.rexxla.org), refined through 1999

NetRexx arithmetic

- Exactly as ANSI (except that FUZZ setting is not included)
- Both **`numeric digits`** and **`numeric form`**
- Implemented in the **`netrexx.lang.Rexx`** Java class (therefore usable by all Java programs)

Decimal arithmetic everywhere?

... not a new idea ...

- "Fingers or Fists?" (Werner Buchholz, 1959)
- Unified Decimal Floating-Point (Fred Ris, 1976)
- Base 10 floating-point, (Tom Hull, 1978)
- Rexx decimal floating point (1979-1999)
- Radix-Independent Floating-Point (IEEE 854, 1987)

Current Status (Databases)

- IBM DB2: 31 digit packed decimal integers, with implied scale
- Microsoft/Sybase: 38 digit integers, held in binary
- Oracle: 38 digit bunched decimal floating point
- XML: Schema includes (broken) decimal data type

Current Status (Languages)

- C/C++: 15-31 digit fixed decimal support
- COBOL: 31 digit fixed decimal support; the new standard in 2001 will require 32-digit floating point
- Java: unlimited floating point decimal support (by IBM)
- JavaScript/JScript: floating point decimal planned
- C#, VB, *etc.* (Microsoft .Net platform): 28 digit floating point decimal
- PL/I and VisualAgeGen: as C
- Rexx family: unlimited floating point decimal support

Current Status (Hardware)

- z-Series (IBM S/390): decimal integer instructions (Store-to-Store) are built-in
- Most non-RISC processors (Intel x86, Motorola 68xxx, *etc.*) have decimal adjust instructions to aid decimal integer arithmetic (not accessible from C)
- In general, decimal arithmetic has to be carried out in software; 100x to 1000x slower than hardware (or worse)

Current Standards

- ANSI X3.274-1996 (Programming Language Rexx)
  - floating point arbitrary precision decimals
- IEEE 854-1987 (Radix-Independent Floating-Point Arithmetic)
  - generalization of IEEE 754, to allow for base-10
  - fixed precision

Rationalizing decimal arithmetic

- Three specifications:
  - Concrete representation, suitable for hardware or software implementation
  - Base specification: core floating point and integer operations, based on ANSI X3-274
  - Extended specification: extends the base to comply with IEEE 854
- Open specifications; available on the web

Decimal representations
For example: 1234.50

- Traditional fixed point: integer and scale

123450	2

- Floating point: mantissa and exponent

123450	-2

  - many advantages in the mantissa being an integer

Decimal digit representation

- Bi-quinary (*e.g.*, Abaci, IBM 650, 74390 TTL Ripple Counter, some 'Nixie' tubes)
- 8.4.2.1 (Binary Coded Decimal) - 4 bits, 8-bit, or 1 character per digit
- Excess-3 (4 bits/digit, values biased by 3)
- 6.3.1.1, 2.4.2.1, 2-out-of-5, 1-out-of-10, Gray
- Negadecimal (base -10), needs no sign

Multi-digit representations

- Base 100 (0-99 in a byte)
- Base 1000 (0-999 in 10 bits), Chen-Ho, and the new Densely Packed Decimal encoding
  - Probably the optimal decimal-based representation
- Base 1,000,000,000 (0-999999999 in 32 bits)
- Big Integer (mantissa is a pure binary integer); this needs fast algorithms for multiplying or dividing by powers of 10

Concrete representations

- 64-bit (3 flags, 11 exponent, 50 integer), 3+15 digits

- 128-bit (3, 15, 110), 4+33 digits

- Exponent is biased binary; integer is 3-in-10 Densely Packed Decimal

Object-oriented design concept

- Arithmetic operations depend on:
  - numbers (many instances)
  - the context in which operations are carried out (usually implied, and relatively global)
- This is mirrored by objects:
  - a class that holds (decimal) numbers
  - a small context class, which provides information such as precision and rounding mode

The IBM implementation for Java™

- BCD (byte/digit). Function upwards-compatible with the **`java.math.BigDecimal`** class
- Fully documented; implementation is open source and available from:

  `http://www2.hursley.ibm.com/decimalj`

- Working through Java Community Process to integrate into the Java Core, using the **`java.math`** (BigInteger + scale) representation

Example - Compound Interest

- $100,000 at 6.5% for 20 years...

```
/* Java program to calculate compound interest */
import com.ibm.math.*;
public class compound{
  public static void main(String arg[]){
    MathContext def=MathContext.DEFAULT;
    BigDecimal start=new BigDecimal(100000);
    BigDecimal years=new BigDecimal(20);
    BigDecimal rate =new BigDecimal("1.065"); // (6.5%)
    BigDecimal total;

    total=rate.pow(years,def).multiply(start,def);
    System.out.println("Final value ="+
                       total.toString());
  }
}
```

Other users of Rexx arithmetic
(in progress)

- DB2 database
- COBOL, PL/I, Language Environment
- ECMA/Microsoft VB, C#, and .Net (subset)
- JavaScript/JScript (ECMAScript)
- Hardware proposal at IEEE Arith15 (June 2001)

Summary

- Binary arithmetic will continue to be used
- Decimal data and arithmetic predominate
- Rexx decimal arithmetic is well understood, standardized, and quite possibly could become *the* standard arithmetic
- Native decimal datatypes are already in many languages (COBOL, PL/I, Rexx, C# ...); once hardware support is available this will accelerate

Questions?

http://www2.hursley.ibm.com/decimal