Decimal Layouts for 754

A Preview

IEEE 754R - 18 April 2002

Mike Cowlishaw
IBM Fellow
mfc@uk.ibm.com

Overview

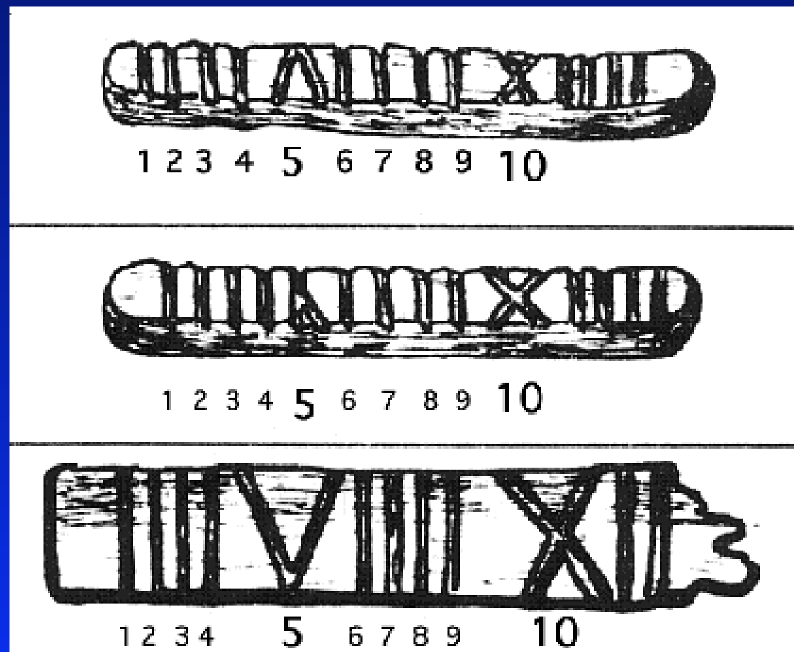
- Background
- Requirements for decimal layouts
- Design decisions
  - Overall representation
  - Coefficient, exponent, and special values
  - 'Strawman' proposal
- Questions?

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 commercial applications; often a legal requirement

- 55% of numeric data in commercial databases are decimal (and a further 43% are integers)



Programming trend is towards decimal arithmetic

- Java: BigDecimals since 1996, rounding floating-point in next release (Java 1.5)
- C#: decimal is a native (scalar) type, from .Net (ECMA 334 & 335; ISO this year)
- 'Commercial' languages (COBOL, PL/I, Visual Basic, *etc.*) all have native or library decimals
- ***But:***
  - decimal formats are all different
  - software is 100 - 1000 times slower than hardware

Why standardize layouts?

- Need to move to hardware to get performance
- Not realistic to go to hardware without agreed Standard layouts
- IEEE 854 without standard layouts is a partial standard; it cannot be implemented without invention
- Standard layouts will allow sharing of decimal data more easily (software or hardware)

Requirements for decimal layouts

- Must be able to represent numbers and values required for IEEE 854 arithmetic
- Must be able to represent decimal numbers and values used in languages and databases today (Java, C#, COBOL, Rexx, SQL, *etc.*)
- Should be efficient for hardware or software
  - Conversions to and from BCD and character strings are especially important in decimal applications

Why preserve trailing zeros?

- Currency (1 Andorran Franc = 11.29870 Algerian Dinar)
- Often indicates precision of a measurement
  - "Turn left after 14 miles" *vs.* "Turn left after 14.0 miles"
- 'Scaled' data in databases and languages
- Numbers as labels (section 3.2 *vs.* section 3.20)
- Often preserves alignment (*e.g.,* summing prices)
- Human-centric applications (follow manual processes)

Decimal Arithmetic (digression)

- As binary floating-point, except that addition and multiplication are exact and unrounded if possible
  - 1.23 + 1.27 gives 2.50 (not 2.5)
  - 1.2 x 1.2 gives 1.44 (not 1.4); 1.2 x 1.5 gives 1.80
  - Results rounded only if they exceed precision available
  - Rounding modes as IEEE 754 plus 'round-half-up'
- Wholly compatible with IEEE 854
  - A 'redundant encoding'
- Representation in languages and databases is always two integers: coefficient and exponent

Decimal representation of 1234.50

- Traditional view: decimal integer and 'scale'

123450	2

- Floating-point view: coefficient and exponent

123450	-2

  - integer coefficient allows preservation of trailing zeros

Representation of the coefficient

- **A.1** Binary (plain base 2 binary integer)
  - maximum 38 digits in 16 bytes, 33 with exponent
  - decimal arithmetic is expensive (shifting, rounding, and conversions require multiplication or division by 10^n)
- **A.2** Binary Coded Decimal (BCD), 4 bits/digit
  - maximum 31 digits in 16 bytes, 28 with exponent
  - insufficient precision (32 digits required)
- Closely packed decimal (10 bits for 3 digits)
  - maximum 38 digits in 16 bytes, 33 with exponent
  - fast, and sufficient precision

Closely packed decimal

- Base 1000 (0x000 through 0x3e7 in 10 bits)
  - expensive to convert to and from BCD
- **A.3** Chen-Ho (Huffman encoded in 10 bits)
  - 3 gate delays to convert to or from BCD
- **<u>A.4</u>** Densely Packed Decimal (enhanced Chen-Ho)
  - not limited to multiples of 3 digits (allows arbitrary lengths)
  - can be lengthened by padding (no re-encoding needed)
  - same as BCD for numbers 0 through 79

(Note: a patent application has been filed for DPD)

Representation of the exponent

- **B.1** Binary twos-complement
  - familiar; 0 exponent is all-zero bits
- **<u>B.2</u>** Binary with bias (unsigned exponent)
  - exponent comparison and manipulation simpler
  - already used in IEEE 754; can use same widths
- **B.3** BCD (4 bits/digit)
  - significantly reduced exponent range (*e.g.,* for an Emax of 999, 4 digits (16 bits) are needed)
  - exponent widths cannot be the same as IEEE 754

Representation of NaNs and ±¥

- **C.1** Reserved values of exponent, as in IEEE 754
  - 'free', for binary-encoded exponents, especially if Emax is 10^n-1 (*e.g.,* +999)
  - test for zero changes to: ' if (c==0 && exp!=0x7ff) ' (*etc.*)
- **<u>C.2</u>** Other reserved values
  - all-zeros can be signaling NaN ("uninitialized")
  - values can be adjacent and independent of coefficient
  - specials have same value (0, 1, 2) regardless of size
- **C.3** Separate bits
  - can reduce coefficient precision by a whole decimal digit

Ordering of the fields

- **D.1** Exponent before coefficient
  - follows existing precedent (IEEE 754, *etc.*)
- **D.2** Coefficient before exponent
  - more like 'written form' (1.23E+7)

Length of the exponent

- **E.1** Follow IEEE 754 and IEEE 754R quad
  - It turns out that these lengths are excellent for decimal-related limits.
    - 11 bits gives ±999 (Emax=999)
    - 15 bits gives ±9999 (Emax=9999)
  - Following IEEE 754 also argues for a biased exponent (choice B.2), as the bias can be the same (potentially sharing circuitry)

Current 'strawman' proposal

- 64-bit (1 sign, 11 exp., 2 pad, 50 coeff.), 3+15 digits

- 128-bit (1, 15, 2, 110), 4+33 digits

Other questions...

- We've called the 64-bit and 128-bit layouts 'single' and 'double' respectively. Is this best?

- A 32-bit format would be possible, but would not be quite the same layout as IEEE 754 single:

  1 sign, 7 exp., 24 coeff., 2+7 digits (Emax = 49)

Next step?

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