Decimal Library Performance

21st March 2009

Mike Cowlishaw

IBM Fellow
IBM UK Laboratories

mfc@uk.ibm.com

Version 1.12

Table of Contents

Introduction

This document describes some performance measurements of three implementations (libraries) of decimal operations. These libraries implement various subsets of the operations defined in the *Decimal Arithmetic Specification*,[1] which describes a superset of the arithmetic operations in the 2008 IEEE 754 Standard for Floating-Point Arithmetic ("754r").[2]

IEEE 754 specifies two variants of the encoding for decimal data; one with a decimal significand and the other with a binary significand. Each of the libraries measured supports one of these encodings (in various ways), and the performance measurements here use the encoding best suited to each library.

Comments on this document are welcome. Please send any comments, suggestions, and corrections to the author, Mike Cowlishaw (`mfc@uk.ibm.com`).

The libraries

The tables later in this document give measurements for operations (where available) for three decimal implementations:

decNumber module — The decNumber module is part of the IBM decNumber package;[3] it implements arbitrary-precision arithmetic with fully tailorable parameters (rounding precision, exponent range, and other factors can all be changed at run time). All decNumber operations always accept arbitrary-length operands. The decNumber module uses a general-purpose internal format (tunable at compile time) which requires conversions to and from any external format. When working with 754r encodings all parameters and results require conversions (each about 100 cycles).

decFloats modules — The decFloats modules are also part of the decNumber package; they work directly on the fixed-size encodings with decimal significand. This document gives results for the decDouble and decQuad modules (64-bit and 128-bit formats).

Intel Decimal Floating-Point Library — The Intel[4] Decimal Floating-Point Library (IDFPL) is an Intel Software Development Product.[5] The functions in the library work directly on the fixed-size encodings with binary significand (64-bit and 128-bit formats).

All three implementations are open source and written in C.

The decNumber and decFloats implementations require 32-bit binary integer types only, conform to

1 See `http://speleotrove.com/decimal/decarith.html`
2 IEEE Std 754-1985 – *IEEE Standard for Floating-Point Arithmetic*, The Institute of Electrical and Electronics Engineers, Inc., New York, 1985.
3 See `http://speleotrove.com/decimal/#decNumber`
4 "Intel" is a trade mark of the Intel Corporation.
5 See `http://www.intel.com/cd/software/products/asmo-na/eng/219861.htm`

strict aliasing and alignment rules, and are tested for use on both little-endian and big-endian architectures. They support string conversions for both ASCII/UTF8 and EBCDIC, BCD conversions, and decimal integer operations (integer divide, shift, rotate, logical and, or, xor, *etc.*).

The IDFPL implementation requires 64-bit binary integer and floating-point types, and is assumed to be little-endian and ASCII/UTF8 only (the README files do not refer to big-endian[6] or EBCDIC support). BCD conversions and decimal integer operations are not supported by the IDFPL implementation.

Description of the tables

In the tables in the later sections, timings for each operation are given in processor clock cycles. Cycle counts are generally a more useful indicator of comparative performance than "wall clock" times, but vary considerably with processor architecture.

For example, the times below are cycles measured on an Intel Pentium M processor in an IBM X41T Thinkpad[7] – on a Pentium 4 or RISC processor most of the tests would show significantly higher cycle counts. The compiler used also makes a measurable difference, so all the cycle counts were measured using the same hardware, compiler, and compiler options (detailed in the notes in the next section).

In the tables, worst-case cycle times are shown for each operation for the decFloats modules (in the column headed decDouble or decQuad), the IDFPL library (headed idfpl64 or idfpl128), and the decNumber module (headed decNum).

Worst-case timings are quoted because best-case timings are generally trivial special cases (such as NaN arguments) and "typical" instruction mixes are too application-dependent to be generally applicable.

For each operation, the name of the operation is given, along with a brief description of the worst-case form of the operation. This is the worst case for the decFloats modules (in some cases the worst case is different for the other modules).

Notes

The following notes apply to all the tables in this document.

1. All timings were made on an IBM X41T Tablet PC (Pentium M, 1.5GHz, 1.5GB RAM) under Windows XP Tablet Edition with SP2.
2. All modules were compiled using GCC version 3.4.4 with optimization settings `-O3 -march=i686` (earlier experiments have indicated that these settings are the best compromise for this hardware and version of GCC).
3. The default tuning parameters were used for decNumber and decFloats (DECUSE64=1, DECDPUN=3, *etc.*); most of these only affect decNumber.
4. The options used for compiling and measuring the IDFPL functions were DECIMAL_CALL_BY_REFERENCE=1, DECIMAL_GLOBAL_ROUNDING=0, and DECIMAL_GLOBAL_EXCEPTION_FLAGS=0; these were chosen as the other two implementations also pass parameters and context by reference.
5. Timings include call/return overhead, and for the decNumber module also include the costs of

6 In version 1.0 there are said to be references in the code to ENDIAN values, so some support may be present.
7 "Pentium" is a trade mark of the Intel Corporation. "Thinkpad" is a trade mark of Lenovo.

converting operand(s) to decNumbers and results back to the appropriate format using the decimal64 or decimal128 proxy modules.

6. “n/s” indicates an operation that is not supported.
7. “BCD” for decNumber is Packed BCD, using the decPacked module; for decFloats it is 8-bit BCD.[8] The IDFPL implementation does not provide BCD conversions.
8. The worst case for each operation is not always obvious from the code and is implementation-dependent (for example, in the decFloats modules, an unaligned add is sometimes faster than an aligned add). It is possible that there may be unusual cases which are slower than the counts listed in the tables, for all the modules, although a wide variety of micro-benchmarks have been tried.
9. A string-to-number conversion can theoretically have an arbitrarily large worst case as the string could contain any number of leading, trailing, or embedded zeros; the timings shown in the tables measured cases where the input string’s coefficient had up to eight more digits than the precision of the destination format.
10. Since the performance measurements shown in the tables were made (in October 2007), the common case of aligned additions on relatively short numbers (6–9 digits) has been measured informally with the same compiler on similar hardware. For these, decNumber and IDFPL are close to the same speed, and decFloats requires about 65% of the cycles (and is about 2.5× as fast as the worst-case addition, for both formats).

8 The most recent decFloats modules support Packed BCD directly, however these conversions have not yet been benchmarked.

decimal64 performance

These tables indicate the performance of common 64-bit operations. Please see the Introduction for explanation.

These measurements are on decNumber/decFloats version 3.56 and IDFPL version 1.0, measured 2007.10.11 and 2007.10.19 respectively.

64-bit conversions			
Operation	**decDouble**	**idfpl64**	**decNum**
Encoding to BCD (with exponent) 16-digit finite	39	n/s	481
BCD to encoding (with exponent) 16-digit finite	46	n/s	327
Encoding to string 16-digit, with exponent	84	242	133
Exact string to encoding (unrounded) 16-digit, with exponent	229	648	196
String to encoding (rounded) 16-digit, rounded, with exponent	266	747	548
Widen to 128-bit 16-digit, with exponent	30	51	209
int32 to encoding From most negative int	39	13	199
Encoded integer to int32 To most negative int32	32	70	136
Encoding (any value) to int32 16-digit, all-nines round, to uint32	178	165	n/s

64-bit miscellaneous operations			
Operation	**decDouble**	**idfpl64**	**decNum**
Class (classify datum) Negative small subnormal	37	95	113
Copies (Abs/Negate/Sign) CopySign, copy needed	25	16	338
Count significant digits Single digit	24	n/s	122
Logical And/Or/Xor/Invert (digitwise) 16-digit	23	n/s	510
Shift/Rotate Rotate 15 digits	154	n/s	583

64-bit computations			
Operation	**decDouble**	**idfpl64**	**decNum**
Add (same-sign addition) 16-digit, unaligned, rounded	245	247	848
Subtract (different-signs addition) 16-digit, unaligned, rounded, borrow	288	251	
Compare 16-digit, unaligned, mismatch at end	126	151	442
CompareTotal 16-digit, unaligned, mismatch at end	149	142	594
Divide 16- by 16-digit (rounded)	828	556	1576
FMA (fused multiply-add) 16-digit, subtraction, rounded	785	879	1683
LogB Negative result	48	66	279
MaxNum/MinNum 16-digit, unaligned, mismatch at end	155	183	656
Multiply 16×16-digit, round needed	362	612	1305
Quantize 16-digit, round all-nines	112	196	422
ScaleB Underflow	212	221	513
To integral value 16-digit, round all-nines	135	170	709

decimal128 performance

These tables indicate the performance of common 128-bit operations. Please see the Introduction for explanation.

These measurements are on decNumber/decFloats version 3.56 and IDFPL version 1.0, measured 2007.10.11 and 2007.10.19 respectively.

128-bit conversions			
Operation	**decQuad**	**idfpl128**	**decNum**
Encoding to BCD (with exponent) 34-digit finite	53	n/s	460
BCD to encoding (with exponent) 34-digit finite	74	n/s	307
Encoding to string 34-digit, with exponent	183	629	239
Exact string to encoding (unrounded) 34-digit, with exponent	297	1331	597
String to encoding (rounded) 34-digit, rounded, with exponent	451	1680	956
Narrow to decimal64 34-digit, all nines	140	546	612
int32 to encoding From most negative int	44	18	199
Encoded integer to int32 To most negative int32	32	87	156
Encoding (any value) to int32 34-digit, all-nines round, to uint32	241	435	n/s

128-bit miscellaneous operations			
Operation	**decQuad**	**idfpl128**	**decNum**
Class (classify number) Negative small subnormal	53	355	133
Copies (Abs/Negate/Sign) CopySign, copy needed	27	33	380
Count significant digits Single digit	27	n/s	138
Logical And/Or/Xor/Invert (digitwise) 34-digit	27	n/s	622
Shift/Rotate Rotate 33 digits	222	n/s	812

128-bit computations			
Operation	**decQuad**	**idfpl128**	**decNum**
Add (same-sign addition) 34-digit, aligned	433	672	1180
Subtract (different-signs addition) 34-digit, unaligned, rounded, borrow	457	689	
Compare 34-digit, unaligned, mismatch at end	187	320	1125
CompareTotal 34-digit, unaligned, mismatch at end	238	293	778
Divide 34- by 34-digit (rounded)	2018	1961	3172
FMA (fused multiply-add) 34-digit, subtraction, rounded	1622	3903	2707
LogB Negative result	58	138	299
MaxNum/MinNum 34-digit, unaligned, mismatch at end	241	312	857
Multiply 34×34-digit, round needed	821	2444	2235
Quantize 34-digit, round all-nines	209	581	670
ScaleB Underflow	263	495	553
To integral value 34-digit, round all-nines	233	461	886

Index