Binary-Integer Decimal?

IEEE 754 – 14 July 2005

Eric Schwarz
Mike Cowlishaw
Mark Erle

Overview

- Why we didn't propose binary-integer decimal formats
  - prior experience and measurements
  - as explained in Arith15 paper, in 2001
- Costs of conversions to software formats
- Critical operations

The decimal model

- 754r: a significand '…is a string of digits…'
- 754r decimals make it possible to have languages with a single numeric type (for both integers and floating-point)
  - there is more than arithmetic to be done
- We must "design computers for the way people are rather than hope that people will adapt to computers" WMK 6/2005

BigInteger (binary) significands

- BigInteger significands are good for multiply
- For other operations they have no advantage, and often have very significant disadvantages, as shown by plentiful existing experience
  - that's why decNumber has chunks base 10^n
- BigIntegers make simple things hard …

BigInteger significand problems

- Counting digits needs full-width comparison
- Aligning an operand, shifting, or rounding all require multiplications (or a division)
- Conversions (string, BCD, Oracle ...) need multiple multiplications (or divides)

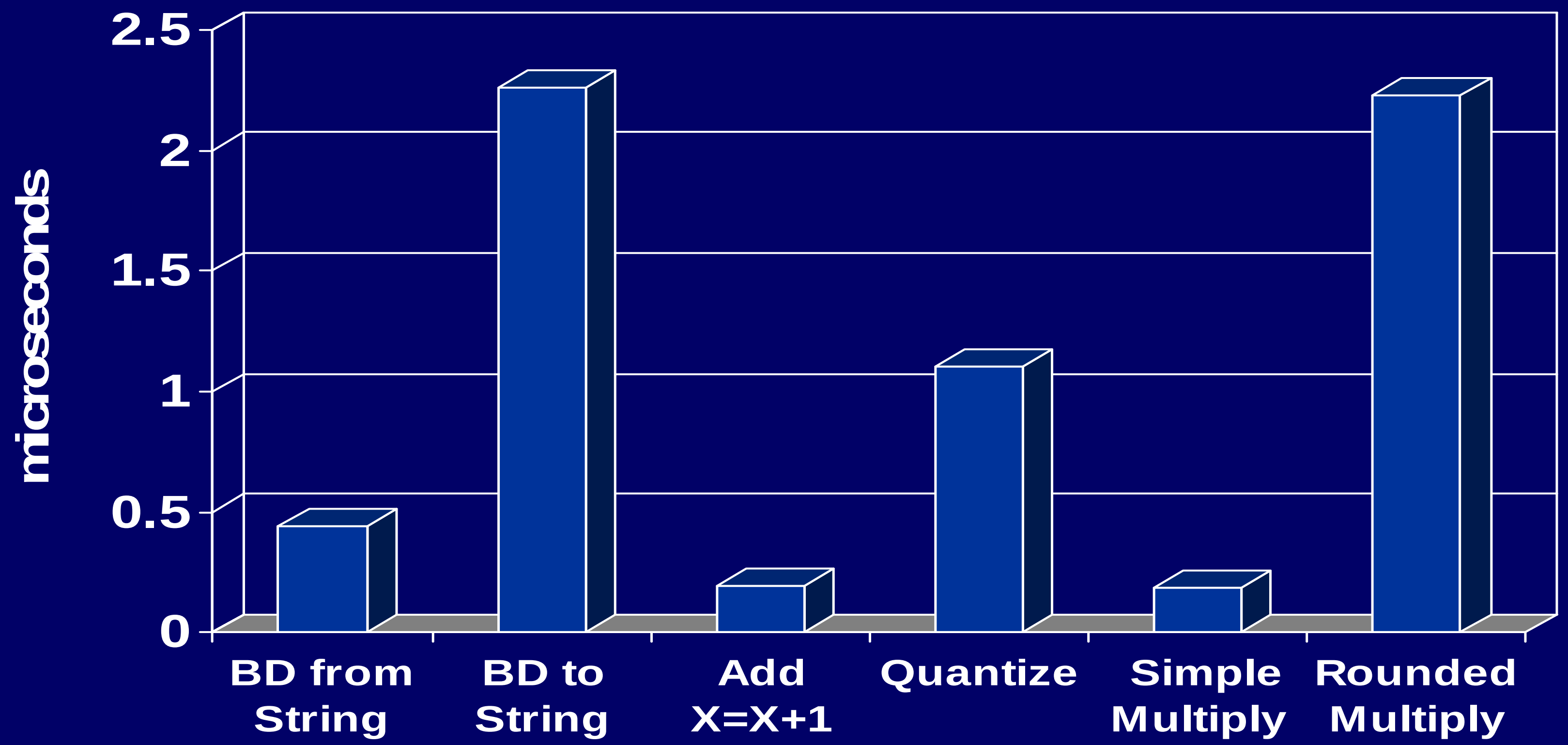
- Unexpected performance characteristics lead programmers to choose the wrong algorithms; we should help programmers, not confuse them

Java BigDecimal (1996)

- Based on BigInteger, as BigInteger is a highly-tuned class; the assumption was that this would lead to a fast BigDecimal

- Experience:
  - good performance for simple multiply
  - very poor rounding and conversions
  - continuing customer complaints

BigDecimal, using BigInteger



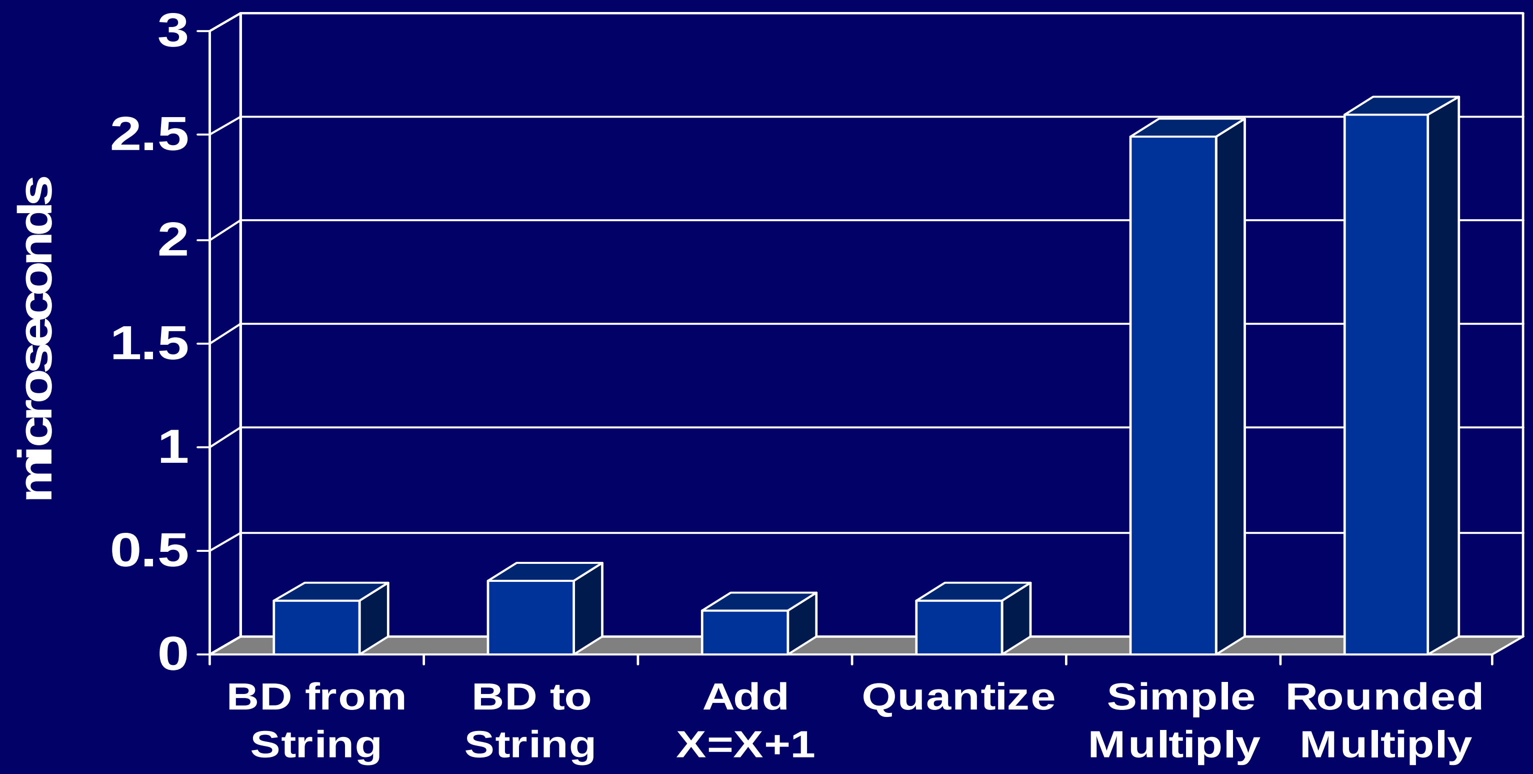
(9-digit operands, Java 1.5 BD on JVM 1.4, WinXP, P4 3GHz.)

BigDecimal, using base-10

- Byte-per-digit implementation
  - prototype for Java 5 decimal enhancements
  - open source (1999, google: decimalj)
  - not performance-tuned
  - slow multiply (n^2 effect)

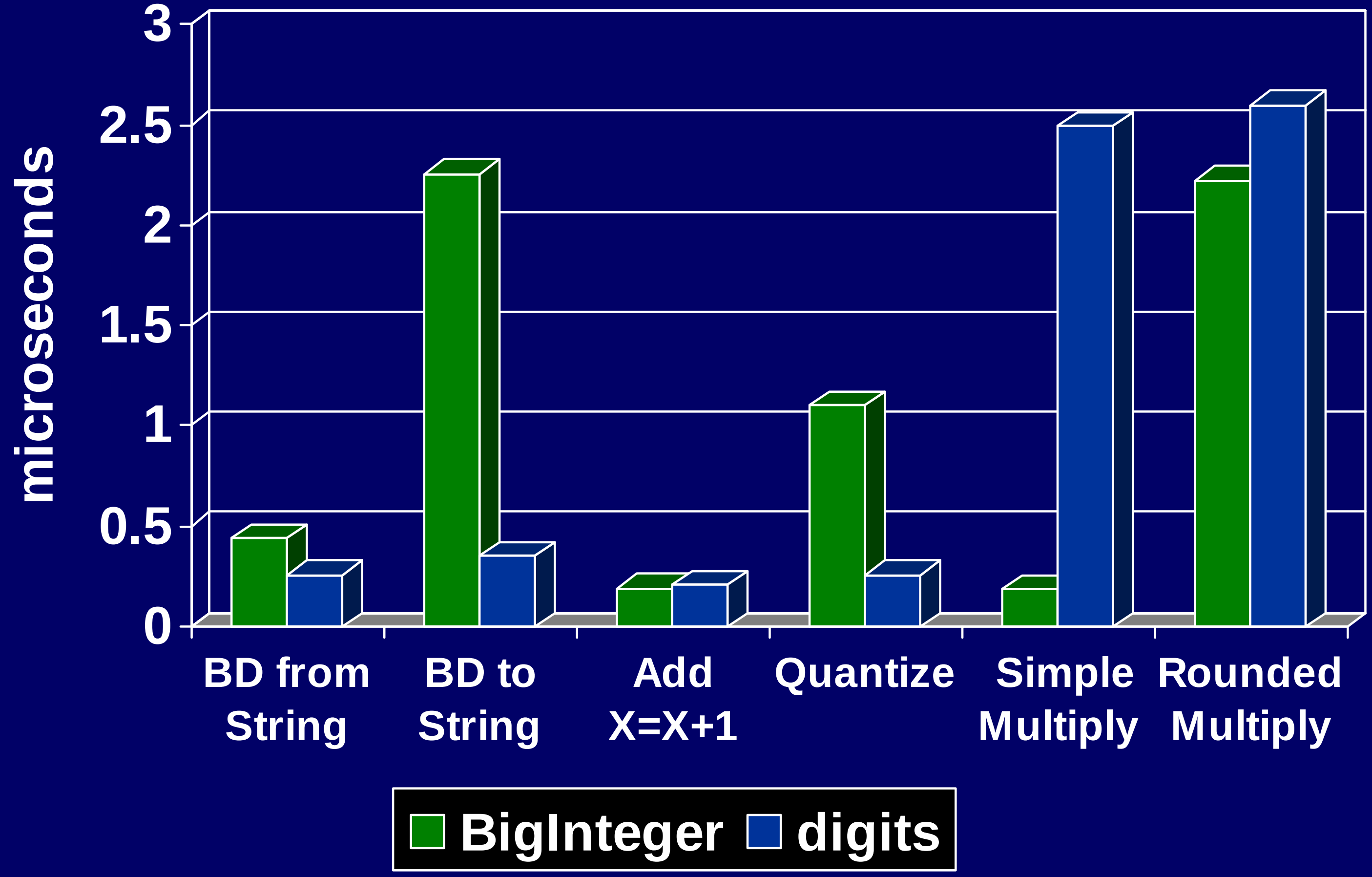
… even so, significantly faster than BigInteger-BigDecimal on SPECjbb2005

BigDecimal, using base-10

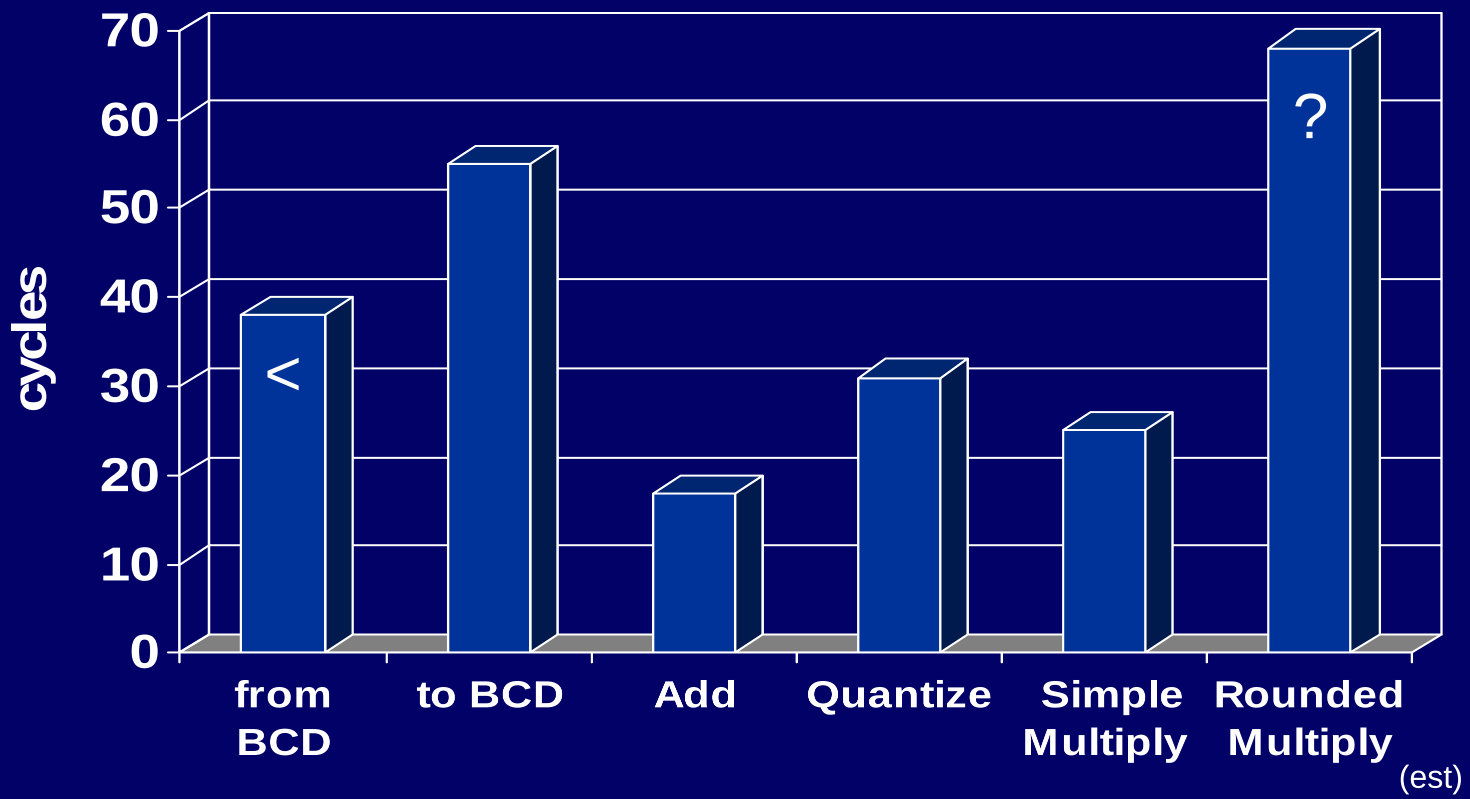


(9-digit operands, IBM decimalj BD on JVM 1.4, WinXP, P4 3GHz.)

BigDecimal comparison



Itanium-optimized (binary significand)

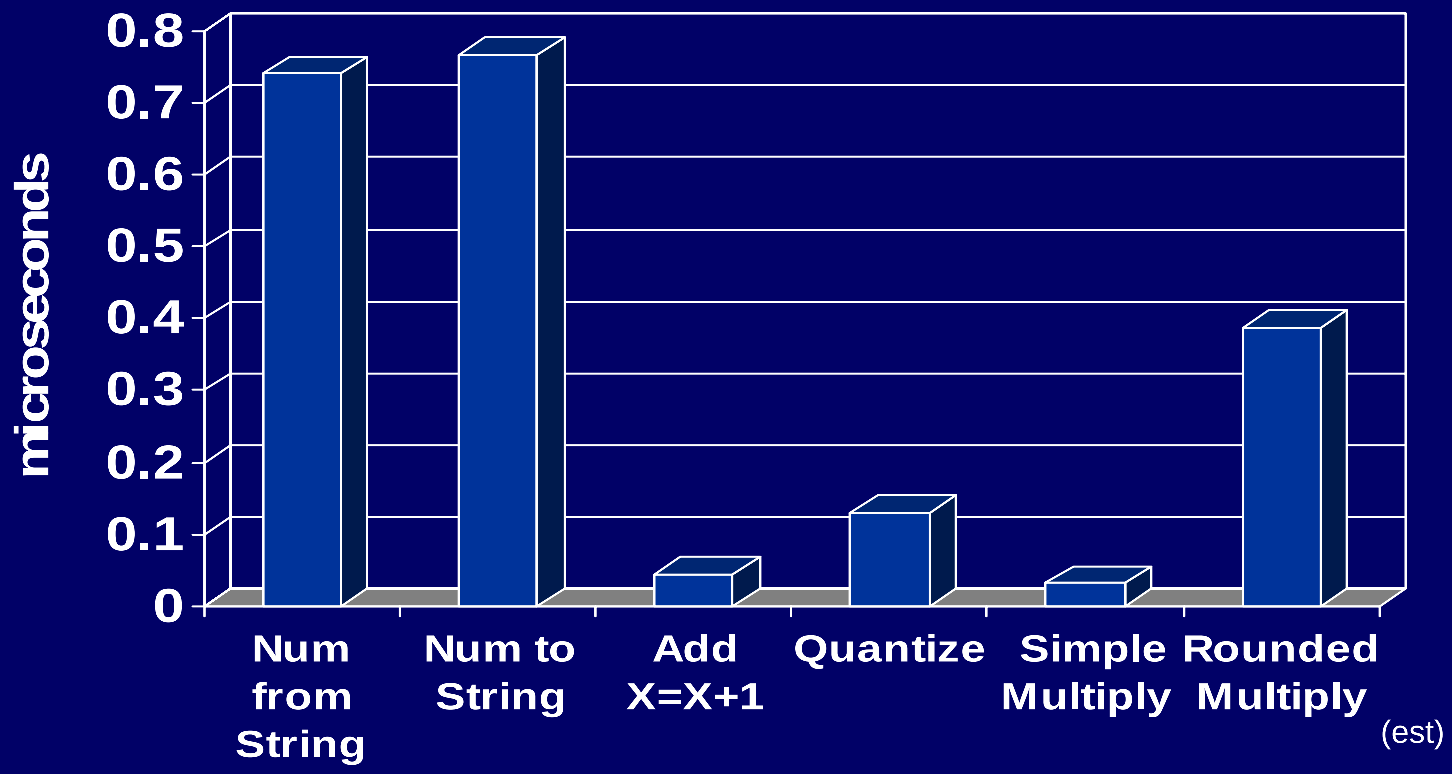


(Intel figures except rounded multiply, from presentations to 754r committee, 3/2005.)

Another BigInt implementation

- C# decimal has a binary significand
  - implemented in C
  - fixed-size, 128-bit, format
  - significand is 3-element int32 array
  - rounds at binary boundary (96 bits)
  - similar characteristics to BigInteger-based Java BigDecimal

C# decimal – using int array



(16-digit operands, .Net 1.1.4322, WinXP, P4 3GHz.)

Chunking with a 10^n base

- Good performance on decimal-specific operations and also good performance on arithmetic – tuneable by changing n

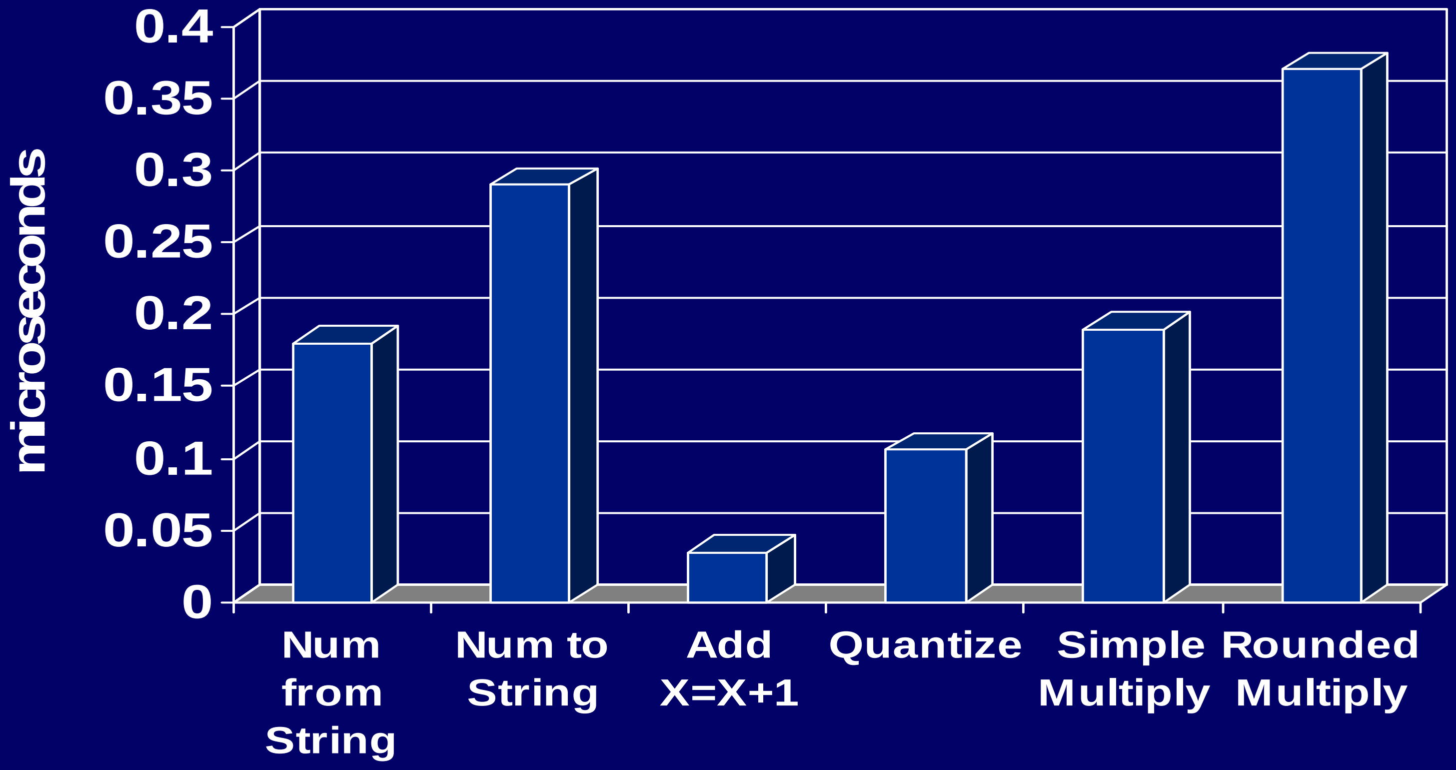
- Performance characteristics match programmer expectations (human-friendly)

- n=4 is optimal for 32-bit machines; n=3 is almost as good and maps to declets

decNumber – a C package

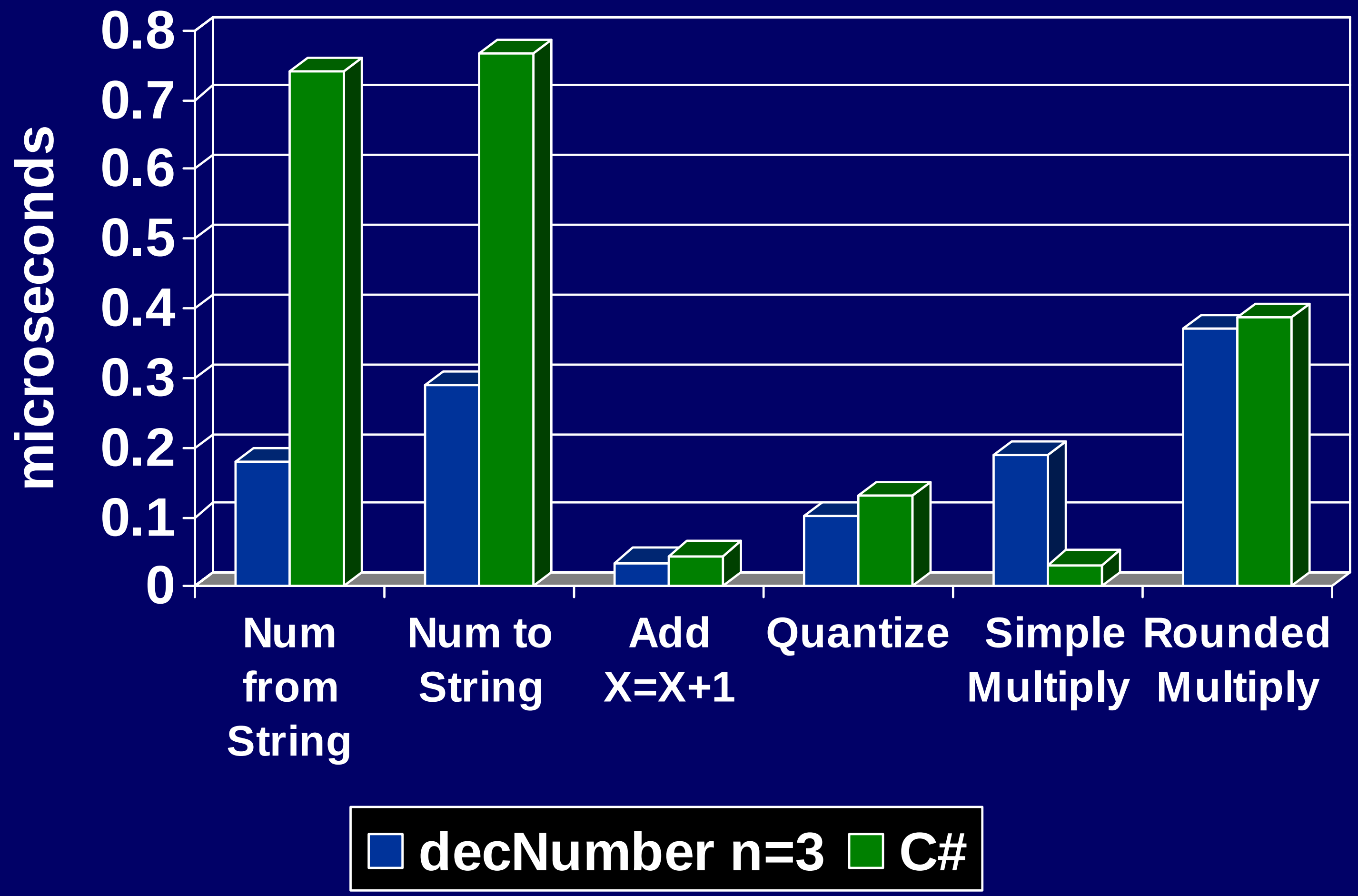
- Generic, fully dynamic (p, emax, rounding, *etc.*), precision up to 10^9 digits
- Licensed since 2001, now Open Source and commercial product (754r formats since 2/2003)
- Performance-tuned for Intel Pentium
- Chunk size selectable (1–9) at compile-time

Chunking with a 10^3 base



(16-digit operands, decNumber 3.25, WinXP, P4 3GHz.)

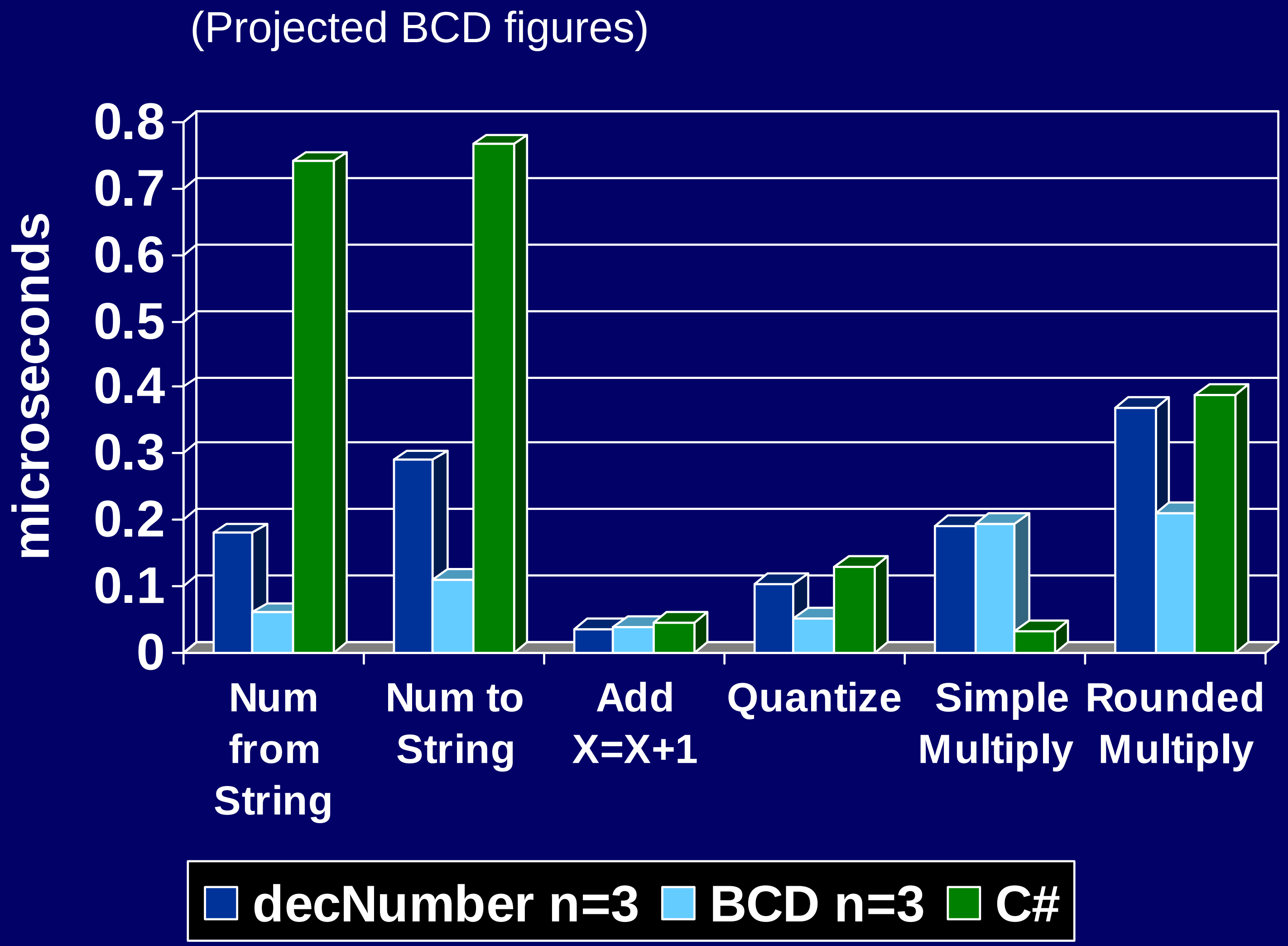
Decimal chunking vs. 96-bit integer



Chunking with a 10^n base

- Current code uses *binary* chunks; a mistake
  - slows conversions and rounding significantly
- Better would be to use n=3 or n=6 encoded as BCD (conversions and rounding then as good as n=1)
- Either is better overall than big binary integers, and is suitable for any architecture (decNumber runs on cellphones upwards)

Decimal chunking vs. big integer



Cost of conversions

Cost of conversions

- The benchmarks in BID-rationale are limited
- ‘Telco’ is a simplification of a traditional commercial mix; it is neither a modern nor a general workload
  - exact, aligned, unrounded, arithmetic (+, ×)
  - simplest (quantize) rounding (no digit counting)
  - only one conversion, of few digits

Cost of conversions

- The comparison is based on arithmetic algorithms "that are specially designed for some particular class of architecture" (BID Rationale June 17)
- "The fact that BID allows for short cuts is a crucial factor in its outstanding performance" (BID Rationale July 12)
  - 'Telco' is extraordinary in that almost all operations allow these short cuts

Cost of conversions

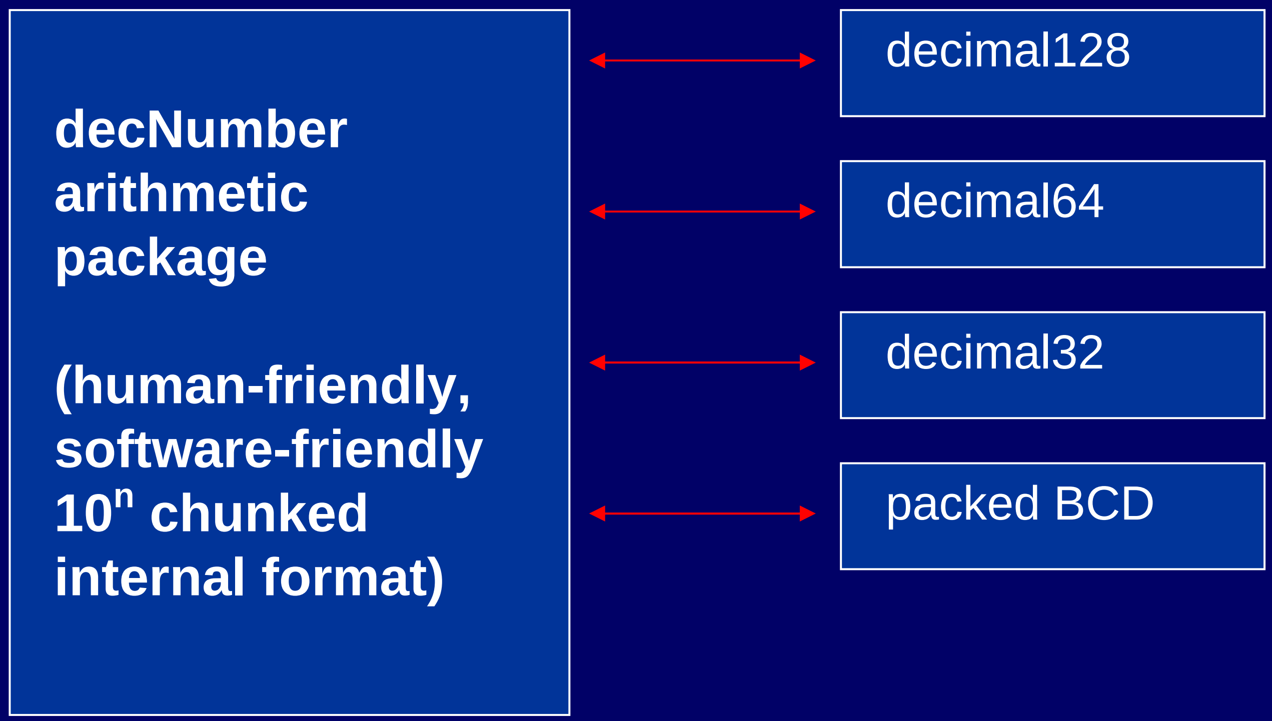
- Without the fast binary hardware support, or with more general calculations, the reported advantage disappears
- In any case, ‘Telco’ has no requirement to convert to and from a format on every operation
  - software can, of course, use whatever internal format is best for the platform

Cost of conversions

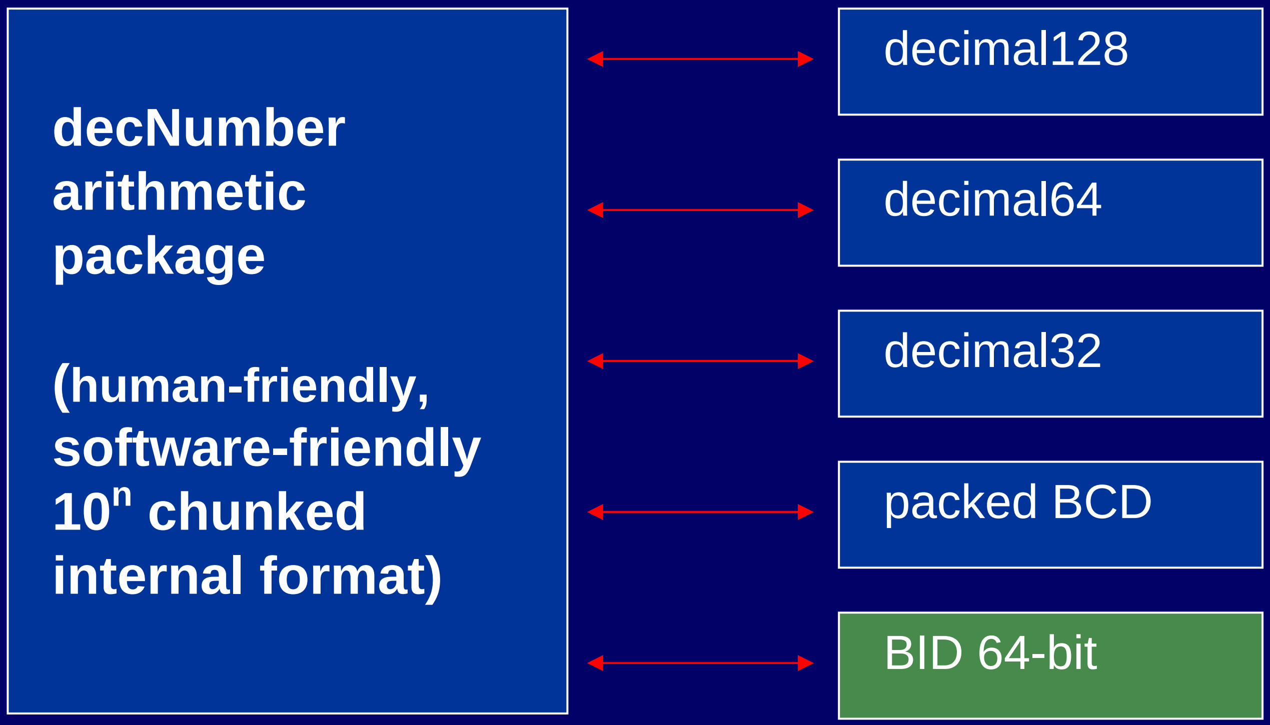
- Further, the study does not show the cost of converting BigInteger (BID) format to decimal formats
- As one example of that, we've written a 64-bit BID conversion module for decNumber, so the conversion costs can be measured separately from arithmetic
  - not the worst case, as target is binary chunks

decNumber modules

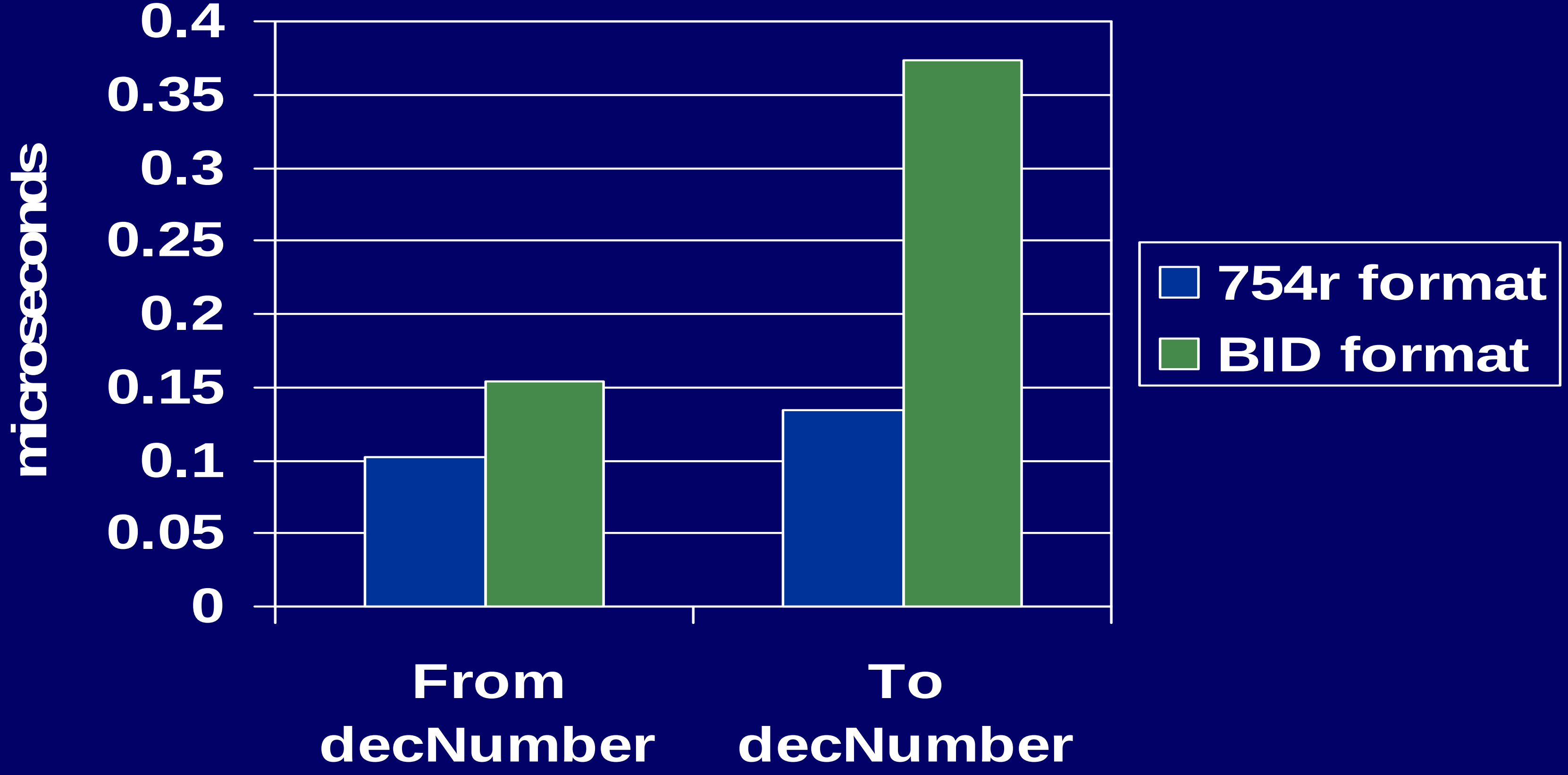
(open source)



decNumber modules



16 digit conversions (n=3)



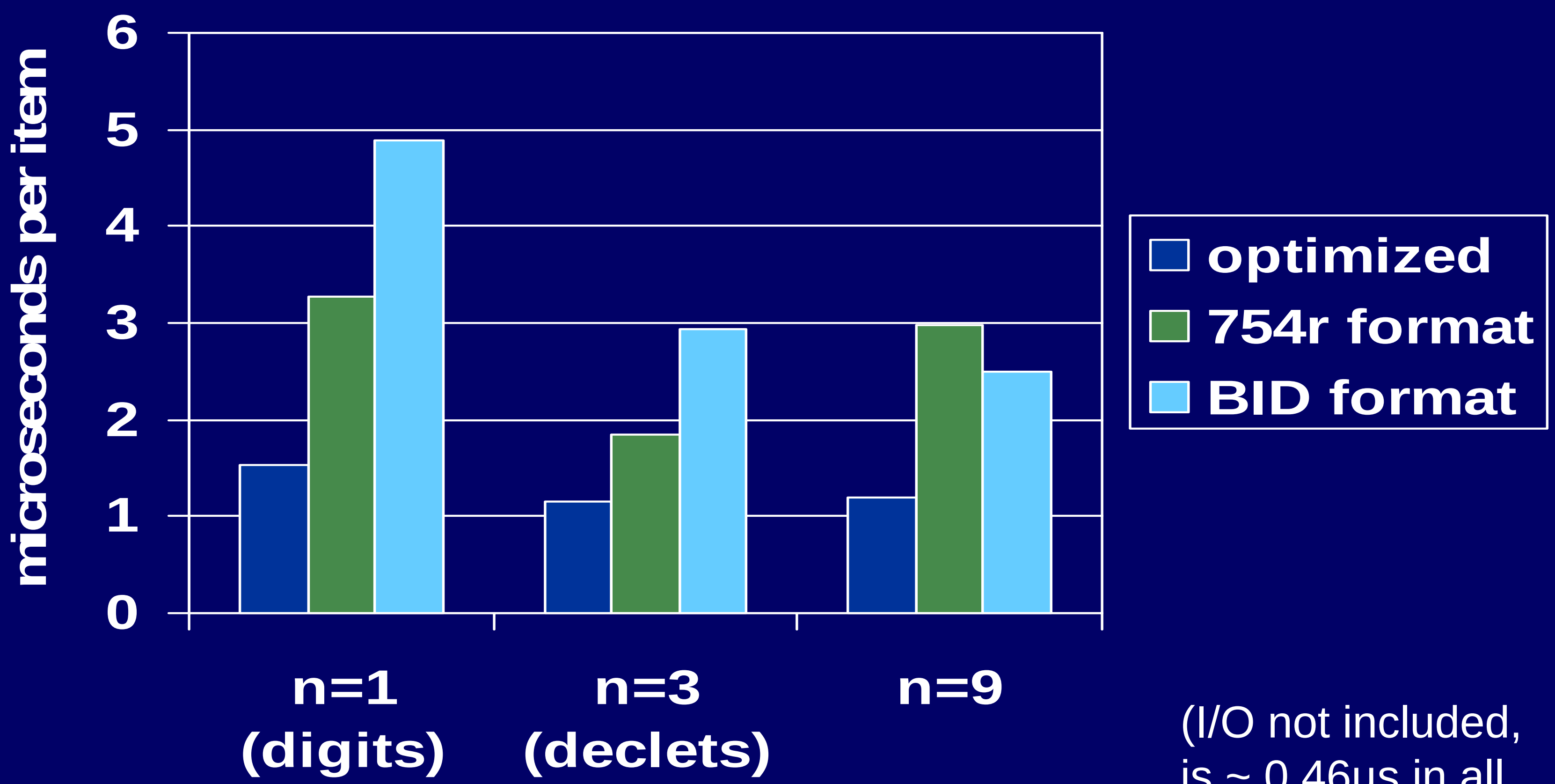
(16-digit operands, decNumber 3.25, DECDPUN=3, WinXP, P4 3GHz.)

‘Telco’ variant (from May meeting)

- ‘Toy compiler’ variant – inner-loop variables convert, only 3 temporaries allowed
  – adds to base benchmark 14.5 conversions for every 9 operations

- Measured base benchmark (‘optimized’), and also ‘toy’ variant with conversions to/from 754r decimal64 format and BID64 format (at various chunk sizes)

‘Telco toy’ timings



(I/O not included, is ~ 0.46µs in all cases)

(decNumber 3.25, GCC, WinXP, P4 3GHz.)

Critical operations

Critical operations

- Many programming languages have only one numeric type, often decimal

- This is the preferred model for future applications programming (no need to know about binary limits, no quiet overflow)
  - binary **`int`**, **`long`**, *etc.* are not exposed
  - traditional ‘integer’ operations use decimal

Shifting

- Used when assembling numbers
  1 800 1234567

```
(1 << 10)  +  (areacode << 7) +  localcode
```

… or for extracting parts of them

```
areacode = rem( tele >> 7, 1000)
```

Bit manipulation

- Multiple flags stored in a number (*e.g.*, a state machine, 754 exception flags, *etc.*)

A B C D E

Bit manipulation

- Multiple flags stored in a number (*e.g.*, a state machine, 754 exception flags, *etc.*)

A	B	C	D	E
1	0	1	0	1

written and stored as the (decimal) number 10101

Bit manipulation

- Multiple flags stored in a number (*e.g.*, a state machine, 754 exception flags, *etc.*)

A	B	C	D	E
1	0	1	0	1

operations on 10101-style numbers:

- logical operations (and, or, xor, not)
- extract, clear, set, or test a flag

Storing and retrieving 10101

- Declets with DPD: each low order decimal digit bit is unencoded (including the one in the combination field). Not even a lookup needed: …0010000000100000001

  and, or, xor, not, test, set, *etc.*, are trivial in hardware or software

- Binary significand: …0010011101110101

Counting digits

- Needed for overflow and underflow detection, rounding, *etc.*
- With a decimal significand this is simple
  - first non-zero digit
- With BigIntegers, first non-zero is just an estimate; an almost full-width comparison is also needed (after a 34-digit multiply, this is very wide: 194 bits)

Overflow and Underflow

- n = count of significand digits
- Overflow occurs when:

  result-exponent + n > Emax + 1

- Similar calculations are needed in several other places, for subnormal and underflow detection, *etc.*

Rounding

- Quantize is relatively simple (the exponent change is easy to calculate)

- Rounding to n digits is harder
  - must count total digits first

- When digits are directly accessible, rounding is inspect-shift-add; BigIntegers need at two multiplies, and more

Rounding to 16 digits – decimal

12345678901234567 × 6543210987654321

→ 80780381836610097820445418533376

Rounding to 16 digits – decimal

$1234567890123456 \times 6543210987654321$

$\rightarrow$ 8078038183661009782044541853376

Count 16 digits

Rounding to 16 digits – decimal

1234567890123456 × 6543210987654321

→ 8078038183661009782044541853376

Count 16 digits

Inspect next digit (and zero-detect the others in 10% of cases)

Rounding to 16 digits – decimal

1234567890123456 × 6543210987654321

→ 8078038183661009782044541853376

Count 16 digits

These digits do not have to be stored once calculated, just note if all zero; this can save almost half the buffer or register width

Rounding to 16 digits – decimal

1234567890123456 × 6543210987654321

→ 8078038183661009782044541853376

Count 16 digits

These digits do not have to be stored once calculated, just note if all zero

Round up may cause a carry (all-9s case).
This is trivial to detect in decimal.

Rounding to 16 digits – binary

462D53C8ABAC0 × 173F04069C0CB1

→ 65F58C92AF4E66A42FA9AC1EC0

Count 16 digits?

Rounding to 16 digits – binary

462D53C8ABAC0 × 173F04069C0CB1

→ 65F58C92AF4E66A42FA9AC1EC0

- Must calculate all but 16 bits (almost full-width) – always
- Then (after leading-1 detect) carry out same-width comparison to find rounding point (compare against 1000000… *etc.*)

Rounding to 16 digits – binary

462D53C8ABAC0 × 173F04069C0CB1

→ 65F58C92AF4E66A42FA9AC1EC0

- Must calculate almost full-width – always
- Carry out wide comparison to find rounding point
- Divide by 10^x to shift – with accurate remainder (or equivalent operation: two multiplies plus subtract and correct)

Rounding to 16 digits – binary

462D53C8ABAC0 × 173F04069C0CB1

→ 65F58C92AF4E66A42FA9AC1EC0

- Must calculate almost full-width – always
- Carry out wide comparison to find rounding point
- Divide by 10^x to shift – with accurate remainder
- Compare remainder with $10^x/2$ (50000….)

Rounding to 16 digits – binary

462D53C8ABAC0 × 173F04069C0CB1

→ 65F58C92AF4E66A42FA9AC1EC0

- Must calculate almost full-width – always
- Carry out wide comparison to find rounding point
- Divide by 10^x to shift – with accurate remainder
- Compare remainder with $10^x/2$
- If rounding up, another 16-digit compare is needed to detect any carry (the all 9s case)

Rounded addition

- A simple, common addition such as 1.2345678901234 56 + 23.4567890123456 7 requires (with a binary significand):
  - a multiply (or two shifts and and add) to align
  - at least two compares, two multiplies, and a subtract to round

- With BCD-based addition, the shifting and inspections are simple

Summary

- Binary significands only have a useful advantage for unrounded multiplication
- They are bad for other decimal operations, conversions, and general calculations (where most results need rounding)
- They are not suitable for a general-purpose encoding

BID format-specific problems

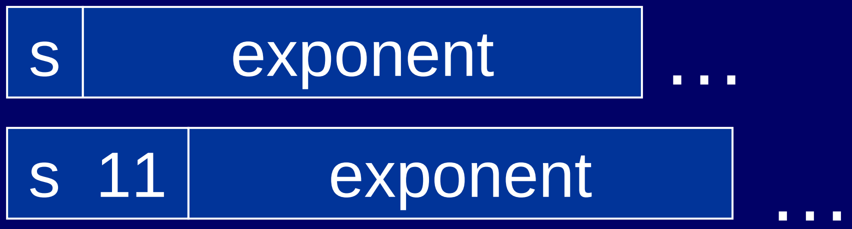
Out-of-range significands

- Binary significands are not naturally bounded to decimal digits; *e.g.,* for BID-32, the integer significand can be as large as 10485759; maximum allowed is 9999999

- Operands must be compared against Smax before use – an almost full-width comparison – then cleared if too large
  - slows either software or hardware

BID-128 complexity

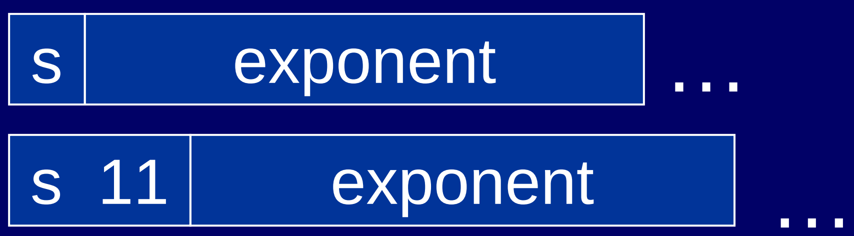
- BID-34 has values up to 9999999999999999999999999999999999 (== 0x1ED09BEAD87C0378D8E63FFFFFFFFF) yet is still defined to have two forms of exponent:



… even though valid significands fit in 113 bits

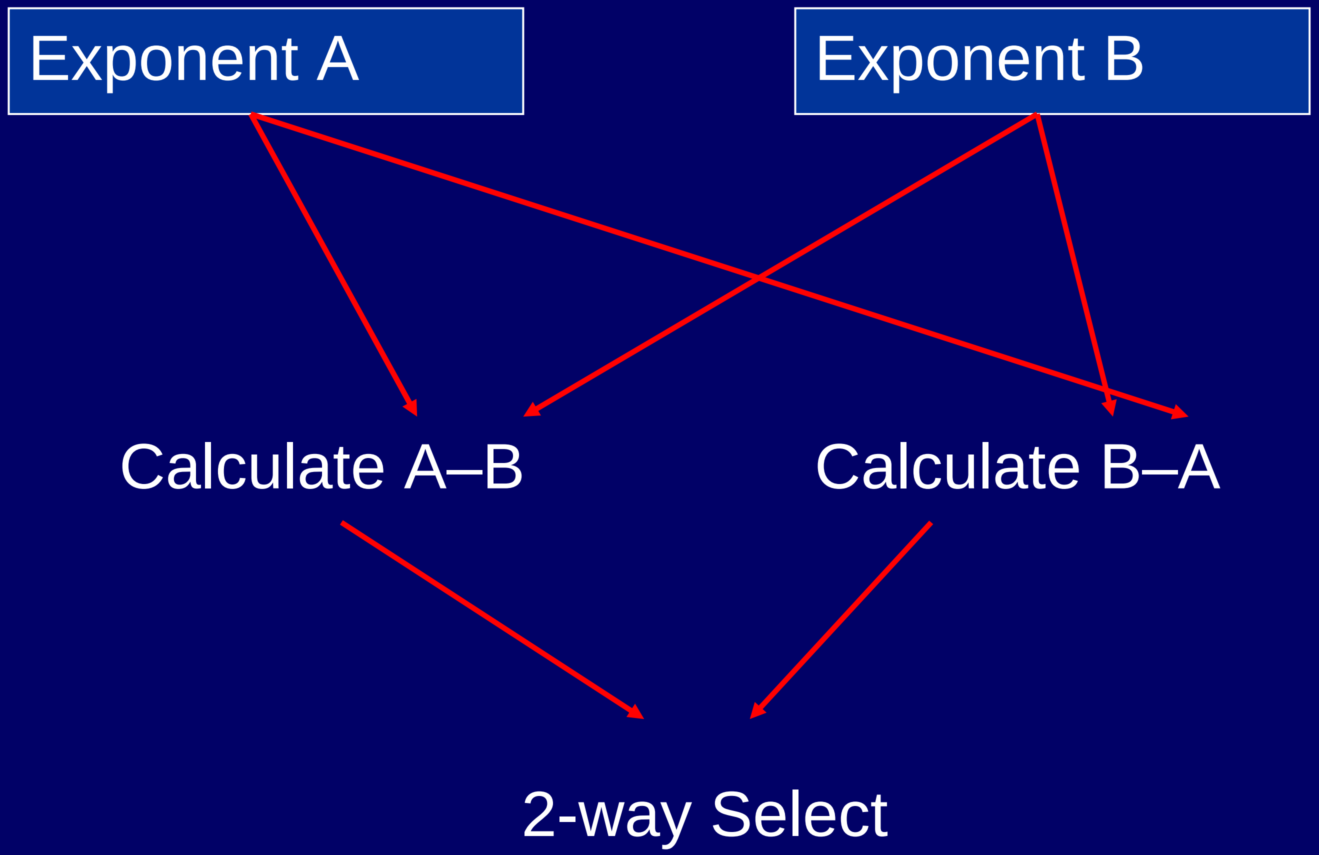
‘Moving’ exponent fields

- Depending on the most significant bit of the significand, the exponent in a format either immediately follows the sign or is shifted two bits

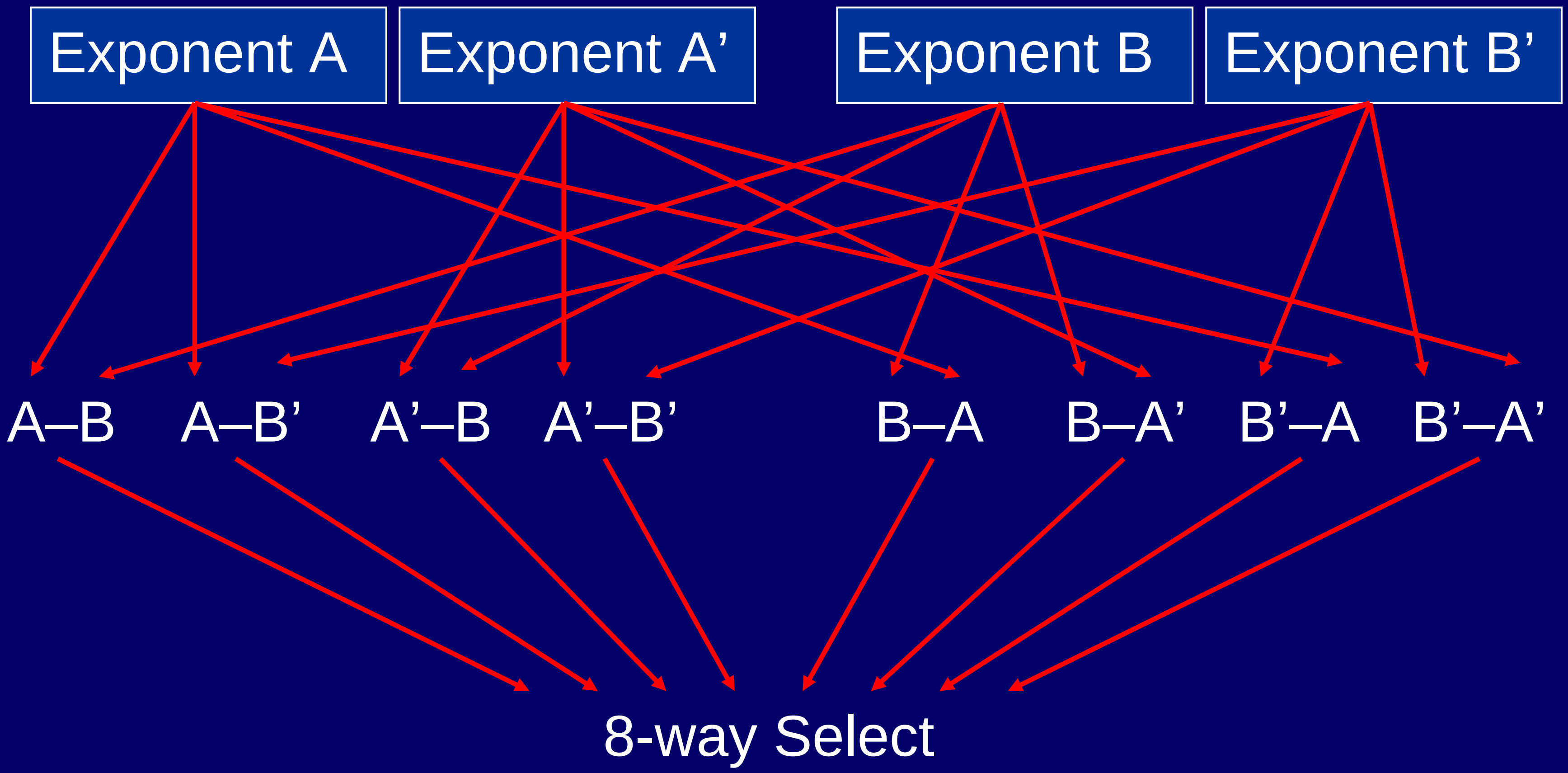


- Exponent difference calculation is on the critical path for addition

Simple exponent difference



‘Moving-exponent’ difference



(Benchmark conditions)

- Hardware: Shuttle X, 3 GHz Pentium 4, 1GB RAM, 120 GB HD
- OS: Windows XP SP 2
- Decimal package: decNumber v. 3.25
  - (also BID format decimal64)
- Compiler: GCC version 3.2 (MinGW 20020817-1)

Criteria for hardware decimals

• <1% cycle counts	no need for any improvement
• <10% cycle counts	optimized software library is fine
• 10-30% cycle counts	borderline for 10x better hardware
• >30% cycle countss	borderline for >4x hardware support

(Intel, 3/2005)

‘Telco’ results (on Itanium)

- decNumber: 83% Telco as-is
- Optimized: 77% Fast Telco + DPD
  57% Fast Telco in BID
  45% Fast Telco

(from BID paper, June 17)