Decimal Formats in IEEE 754

Analysis and Benchmarks – 21 April 2005

Eric Schwarz
Mike Cowlishaw

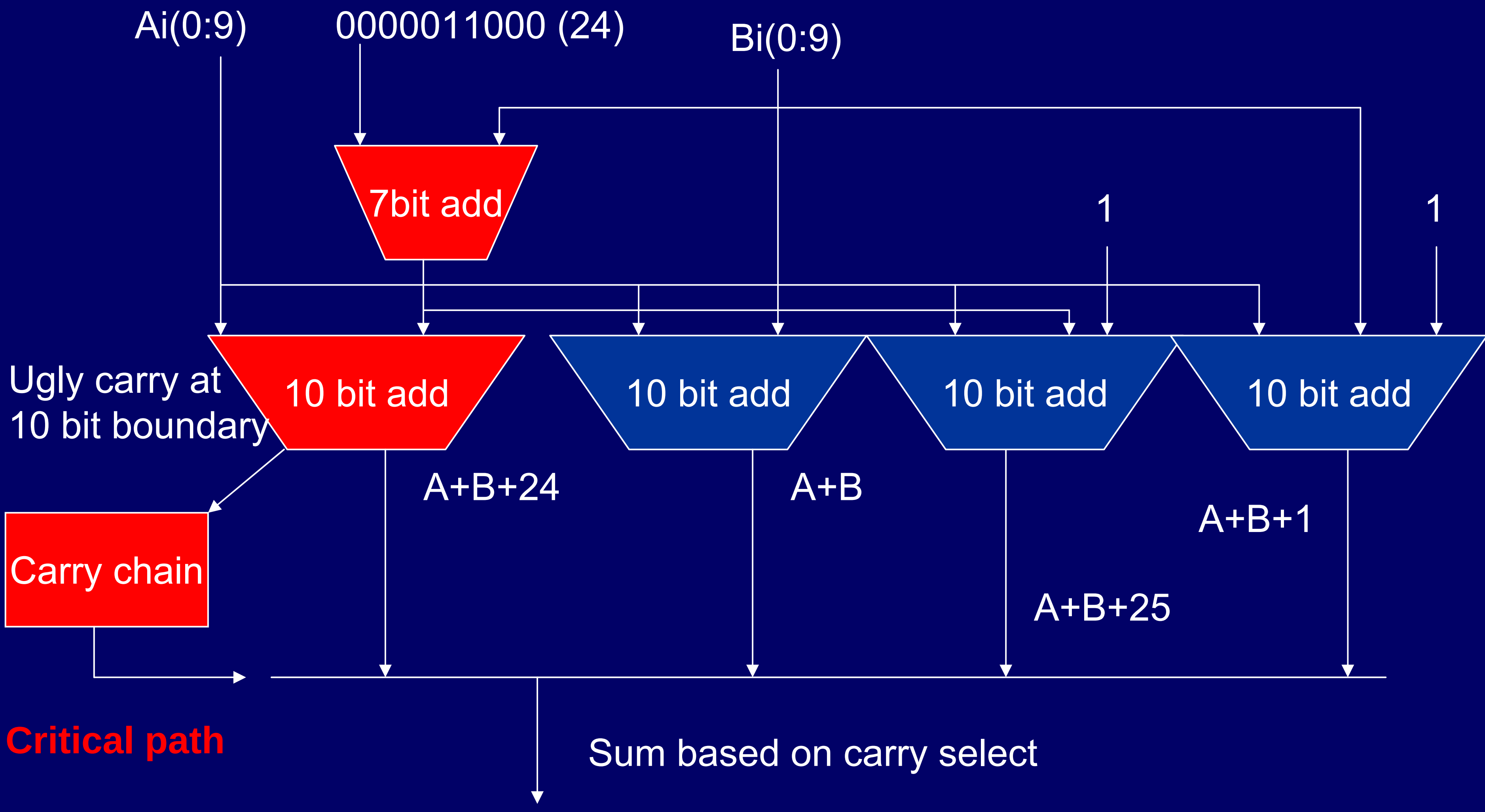
Overview

- Hardware comparison: Base-1000 and BCD
- Software Benchmarks
  - chunking
  - costs of conversions & DPD
  - cost of re-ordering combination field
- Format conversions
  - where necessary, where avoidable

Hardware Comparison of Base1000 to BCD

(Neglecting floating-point alignment and rounding, which favor BCD)

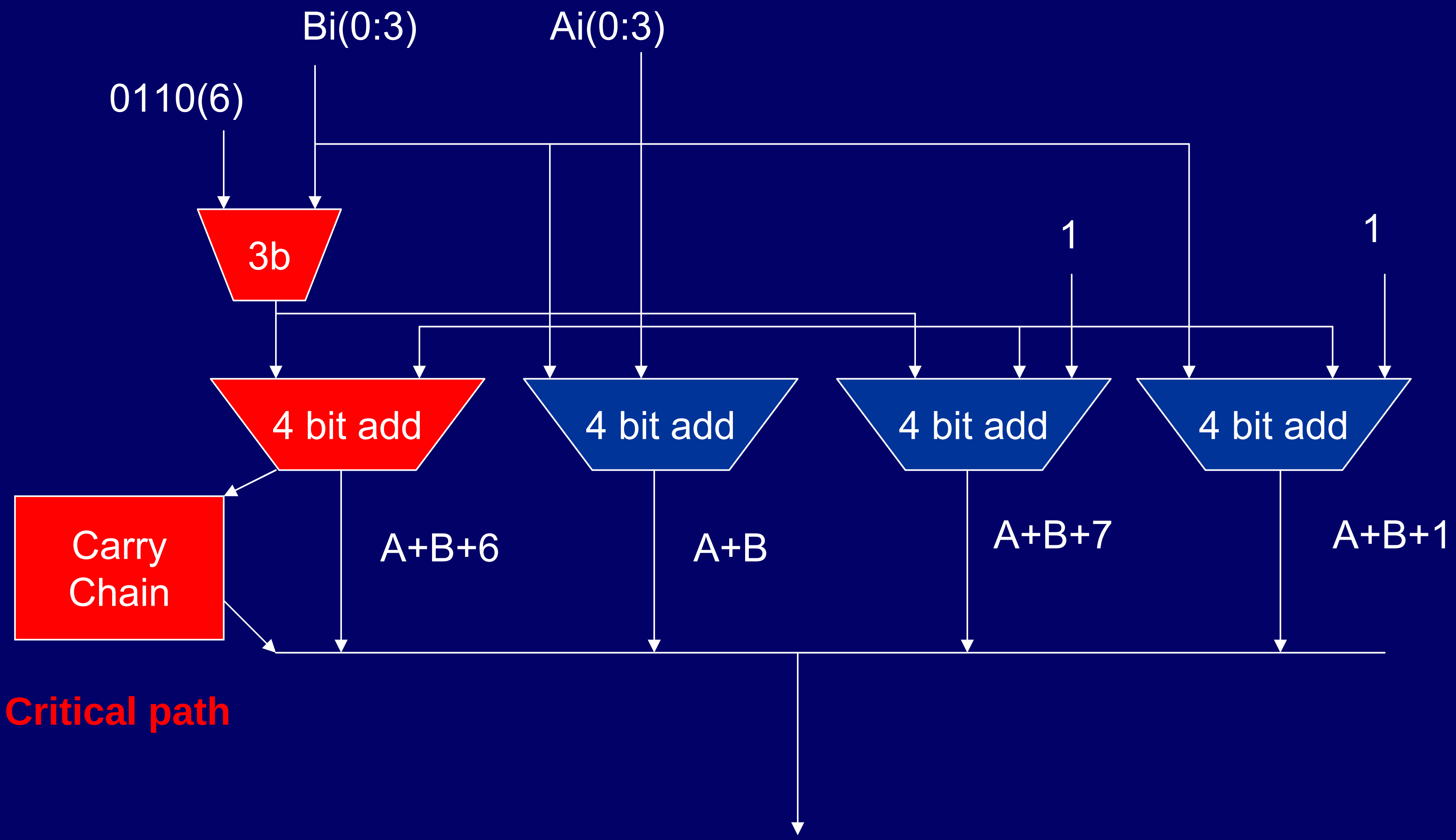
Base 1000 Add (fixpt)



Base 1000 add (fixpt)

- 4 x 10 bit add and 1 x 7 bit add (or 3:2 counter) per group
- 5 big groups in 16 digit, 11 big groups in 34 digit

Base 10 BCD add (fixpt)



BCD add (fixpt)

- 4 x 4 bit adder and 1 x 3 bit add or CSA per group
- 16 groups for 16 digits, 34 for 34 digits
- A+B+6;A+B+7 group and A+B;A+B+1 group can be built into bigger groups prior to carry select

Adder comparison

	Base 1000	BCD
Non-Optimal **3 way** add To get carry	A(0:9) + B(0:9) + 24 7 bit adder	A(0:3) + B(0:3) + 6 3 bit adder
Optimal Binary Adder tree for Greater than Group carries And carry select	6 Groups	16 Groups

Base 1000 has one more level in the non-optimal 3way add
And BCD has one more level in the optimal 2 way add
ADVANTAGE TO BCD
Actual implementations have same gate levels,
base 1000 has 4 way gates versus BCD with 2 way gates

Base 1000 Multiply (fixpt)

1. Modular group by group multiplies (1G x 1G = 3D x 3D)

2. Group by full width (1G x 6G= 3D x 16D)

3. Couple bit by full width (3-4bit x 16D)

4. Digit by full width (1D x 16D)

Base 1000 (1G x 1G = 3D x 3D)

0	1	2	3	4	5	6	7	8	9
	A			B			C		

*

0	1	2	3	4	5	6	7	8	9
	D			E			F		

20 bits but not in base 1000

0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9

To get to base 1000
Need divide by 1000
And remainder

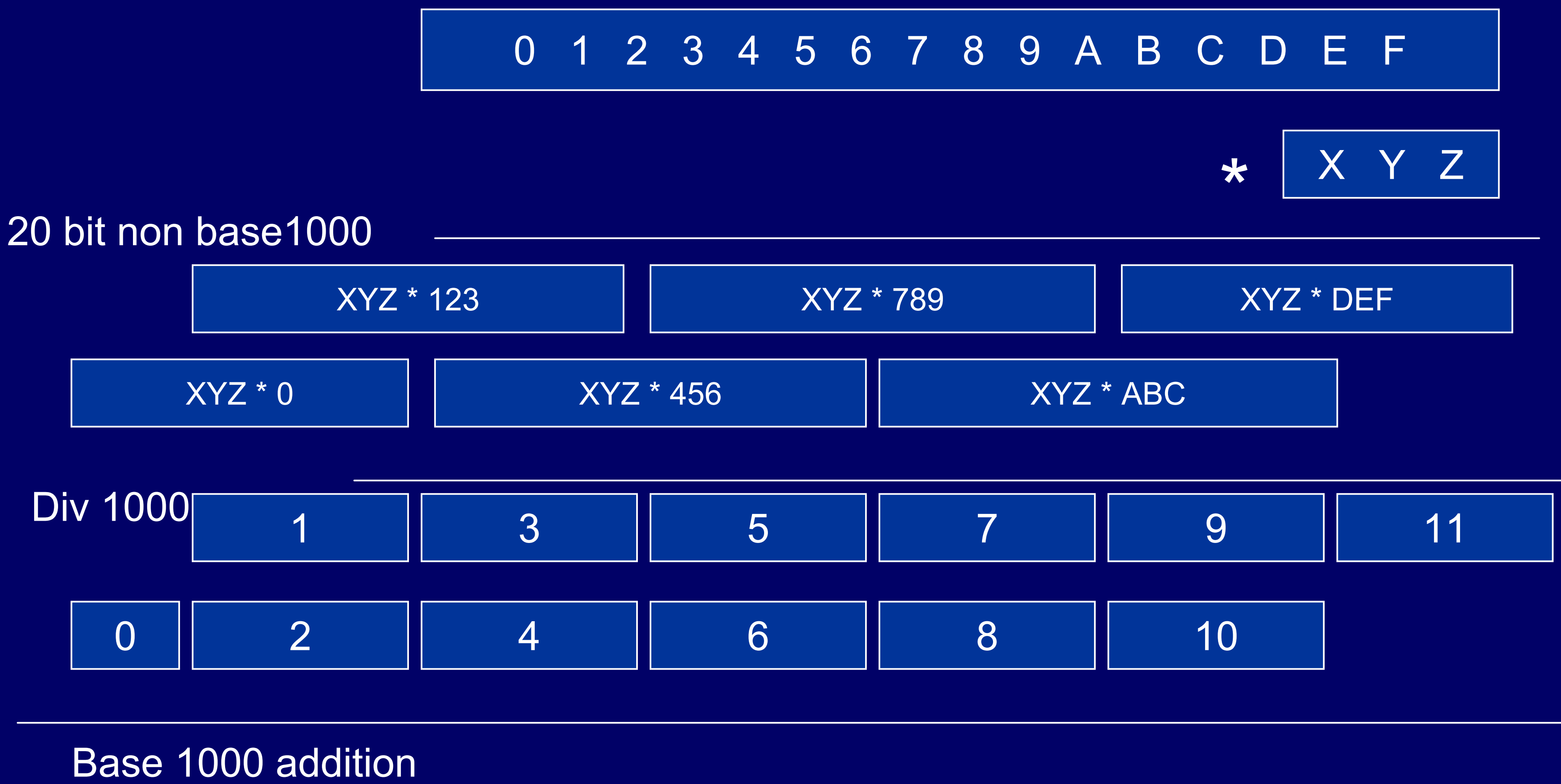
0	1	2	3	4	5	6	7	8	9
	S			T			U		

0	1	2	3	4	5	6	7	8	9
	V			W			X		

Base 1000 (1G x 1G = 3D x 3D)

- With 1 G x 1 G need to break down 6G x 6G multiplication into 36 multiplies
- Each group multiply is a
  - 10b x 10b multiply followed by at best
  - a reciprocal multiplication to get quotient,
  - and then a multiply subtract to get remainder

Base 1000 multiply 1G x 6G =3D x 16D



Base 1000 multiply 1G x 6G =3D x 16D

- This operation could be pipelined taking six group multiplies, and then 5 additions

- Each group multiply is
  - 6 parallel multiplications
  - 6 multiply by reciprocal of 1000 – get high
  - 6 multiply-subtracts for remainder – get low

Base 1000 multiply (3-4b x 16D)

1. A stored look-up table could be made for multiples between 0-7x or 0-15x
2. Then the 10 bit group could be parsed into 3, 3, 3, 1 or 4, 4, 2.
3. A partial product could be created each cycle
4. Though shifting stored partial products is possible problem, 8X and 16X
5. A partial product could be accumulated each cycle with base 1000 adder

Base 1000 multiply (1D x 16D)

- Stored multiples of 0x to 9x could be accumulated
- The 10 bit group could be divided into 3 digits, either by repetitive reciprocal multiplications of 1/10 or by other means
- Partial products for each digit could be created each cycle and shifted by 10 or 100
- Partial products could be accumulated each cycle

BCD Multiplication on Z900
(year 2000)

- Store multiples of 0-9x (1D x 16D)
- Select partial product to be accumulated each cycle
- Accumulate partial product every cycle to running sum using decimal adder

Multiplication (fixpt) Comparison

- Base 1000 cannot be done like integer or BFP multiplication, each partial product has to be converted back to base 1000
- Stored multiples very efficient but favors BCD slightly due to easy partition / shifting
- 3 Digit chunking not optimal

Hardware Conclusions

- Base 1000 and BCD comparable for addition and multiplication (fixed point), slight advantage to BCD
- For Full Floating-Point Implementation:
  - BCD has cost of conversion to/from DPD versus
    - 3 gate levels each way (partial cycle)
  - Base 1000 has costs of alignment, rounding
    - Division of constant, remainder, shifting (multiple cycles)
- There is a clear and overwhelming advantage of current exponent format over Tang's exponent format (no divisions, remainders, early diff.)

Software Chunking

- Most software decimal packages either work directly with BCD or character strings, or they ‘chunk’ the significand into binary integers with maximum value $10^n - 1$

  Example, $x =$ 12345.67 with $n =$ 4
  two chunks in
  range 0–9999
  (exponent = -2)

123	4567

Software Chunking [2]

- Choice of chunk size directly affects performance; for more mathematical mixes, (multiplies) a large chunk size is best
- Generally, a chunk size of 2^k which fits in the largest hardware integer (or FP) size available is good
- *e.g.*, 2 for 8-bit machine, 4 for 16-bit, *etc.*

Software Chunking [3]

- In a 32-bit machine (assuming multiply to 64 bits is accessible), the best-compromise chunk size is 8
- However, many packages do not use this (*e.g.*, Oracle uses a variant of n=2)

decNumber – a C package

- Generic, 754r arithmetic & formats, fixed precision up to 10^9 digits
- Licensed since 2001, now Open Source in GCC (754r formats since 2/2003, 1200 downloads + commercial)
http://savannah.gnu.org/cgi-bin/viewcvs/gcc/gcc/gcc/Attic/?hideattic=1&only_with_tag=dfp-branch
- Performance-tuned for Intel Pentium
- Chunk size selectable (1–9) at compile-time

decNumber – internal form

- Internal form is an array of chunks (plus exponent, sign, and digits count)
- Modules convert from various data formats to and from the internal form
  - decimal32, 64, 128, packed BCD, *etc.*
- All calculations are done in internal form

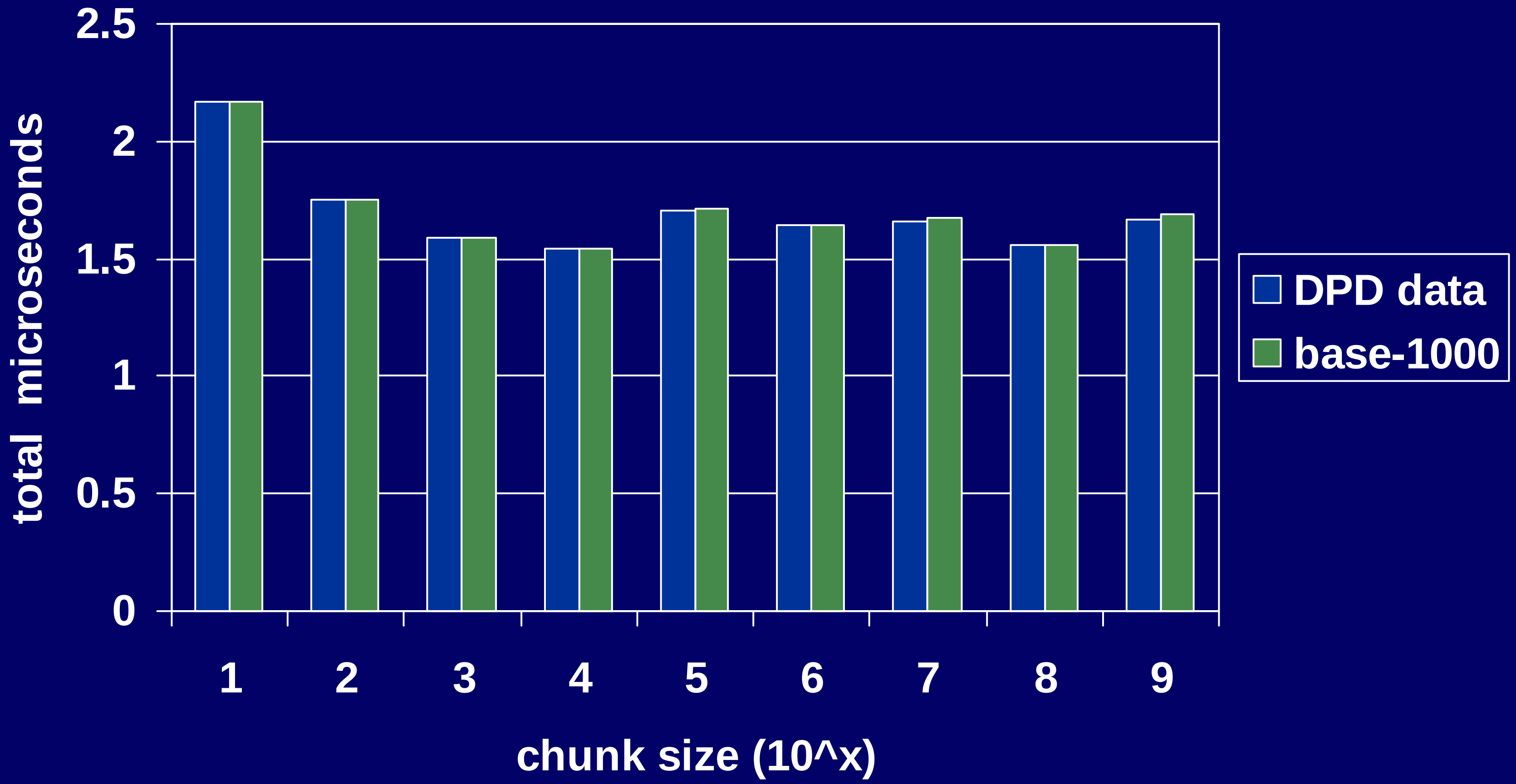
‘telco’ – a benchmark

- On the Web since 2001; well understood
- Reasonably indicative commercial mix
  - 2 conversions (1 from decimal64, 1 to string)
  - 4 aligned adds
  - 2.5 rounds
  - 2.5 multiplies (exact)
- (Mathematical mix would have far more rounding, and also unaligned adds)

(Benchmark conditions)

- Hardware: Shuttle X, 3 GHz Pentium 4, 1GB RAM, 120 GB HD
- OS: Windows XP SP 2
- Decimal package: decNumber v. 3.24
  - (also variant base-1000 decimal64, no lookup-tables)
- Compiler: GCC version 3.2 (MinGW 20020817-1)

Base benchmark



Reproducibility ≈ 0.01 μs (0.5 % this chart) I/O ≈ 0.44 μs (all charts)

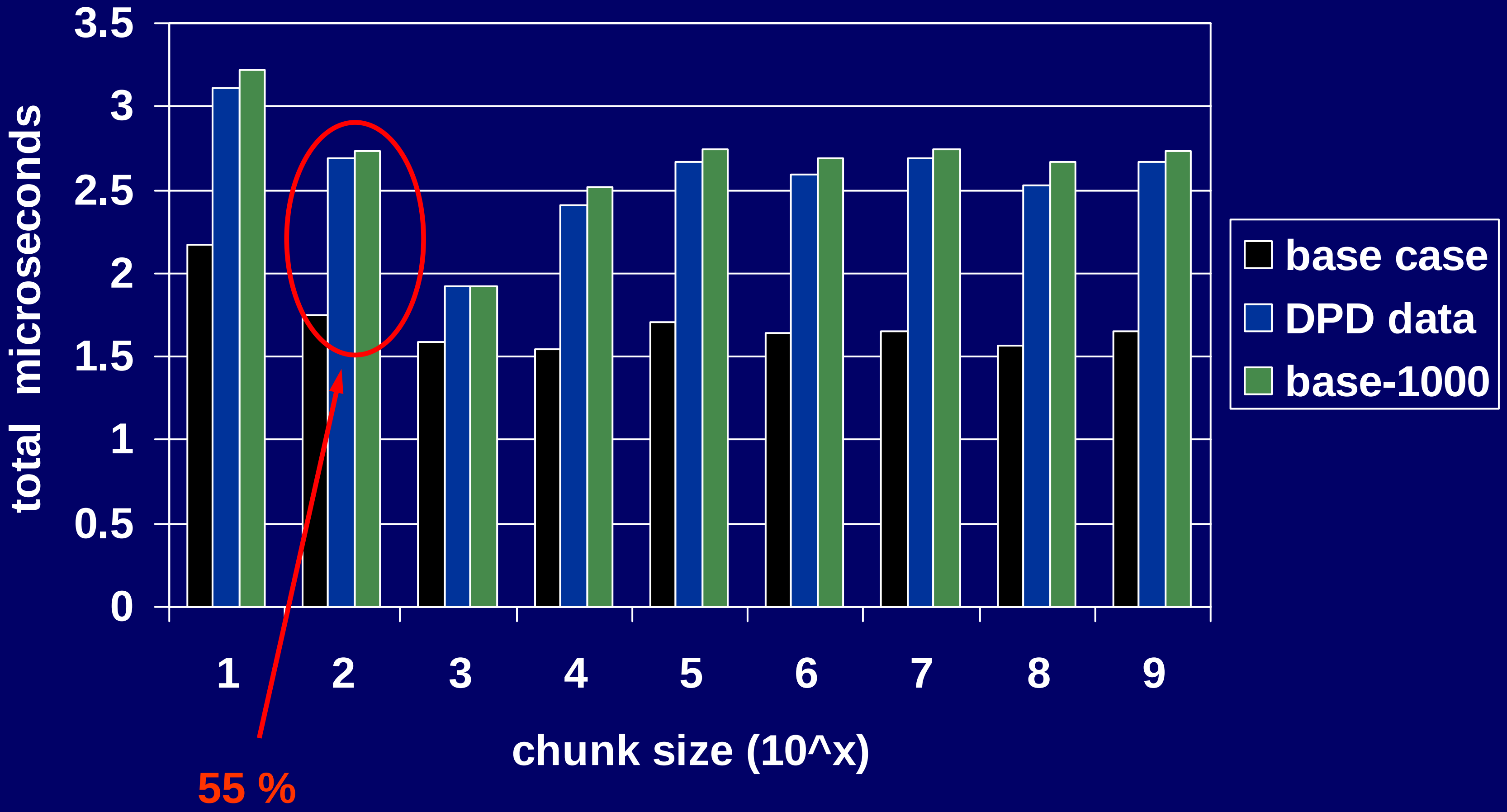
‘telco’ variants

- Benchmark only has one decimal64-to-internal form conversion per data point
  - one data point = 4,640 cycles
  - averages 1.004 DPD lookups (of 1 cycle) for each data point (data points are mostly 3 digits and hence have 4 leading zero declets)
- Constructed two variants on the benchmark to try and get a measurable DPD effect …

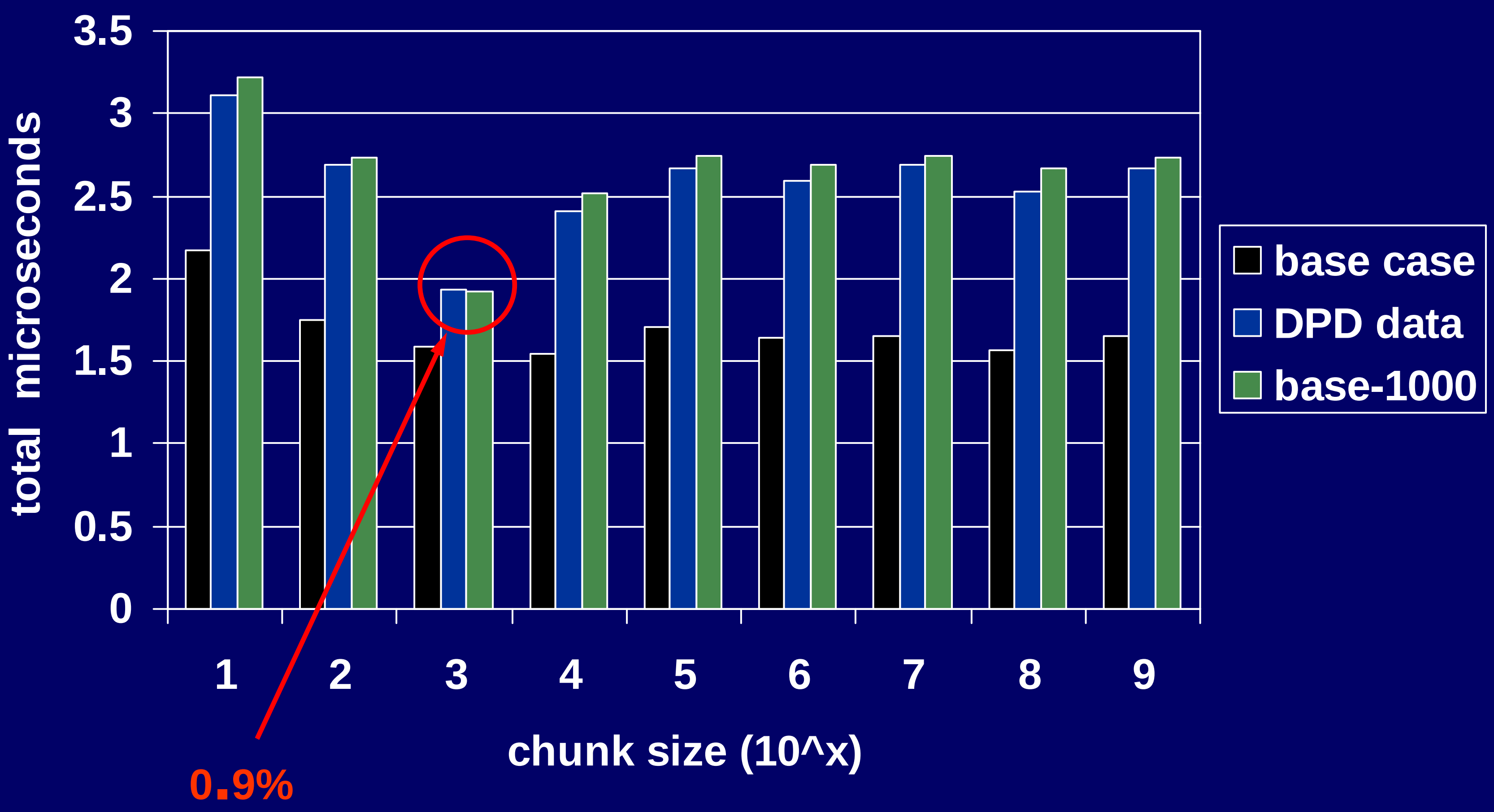
‘telco’ variants [2]

- ‘bad compiler’ – variables used outside inner loop are converted to/from decimal64
  - adds to base: 2.5 To and 3.5 From
  - now 8 conversions for every 9 operations
- ‘toy compiler’ – inner-loop variables convert, only 3 temporaries allowed
  - adds to base: 6.5 To and 6 From
  - now 14.5 conversions for every 9 operations

‘Bad compiler’ variant

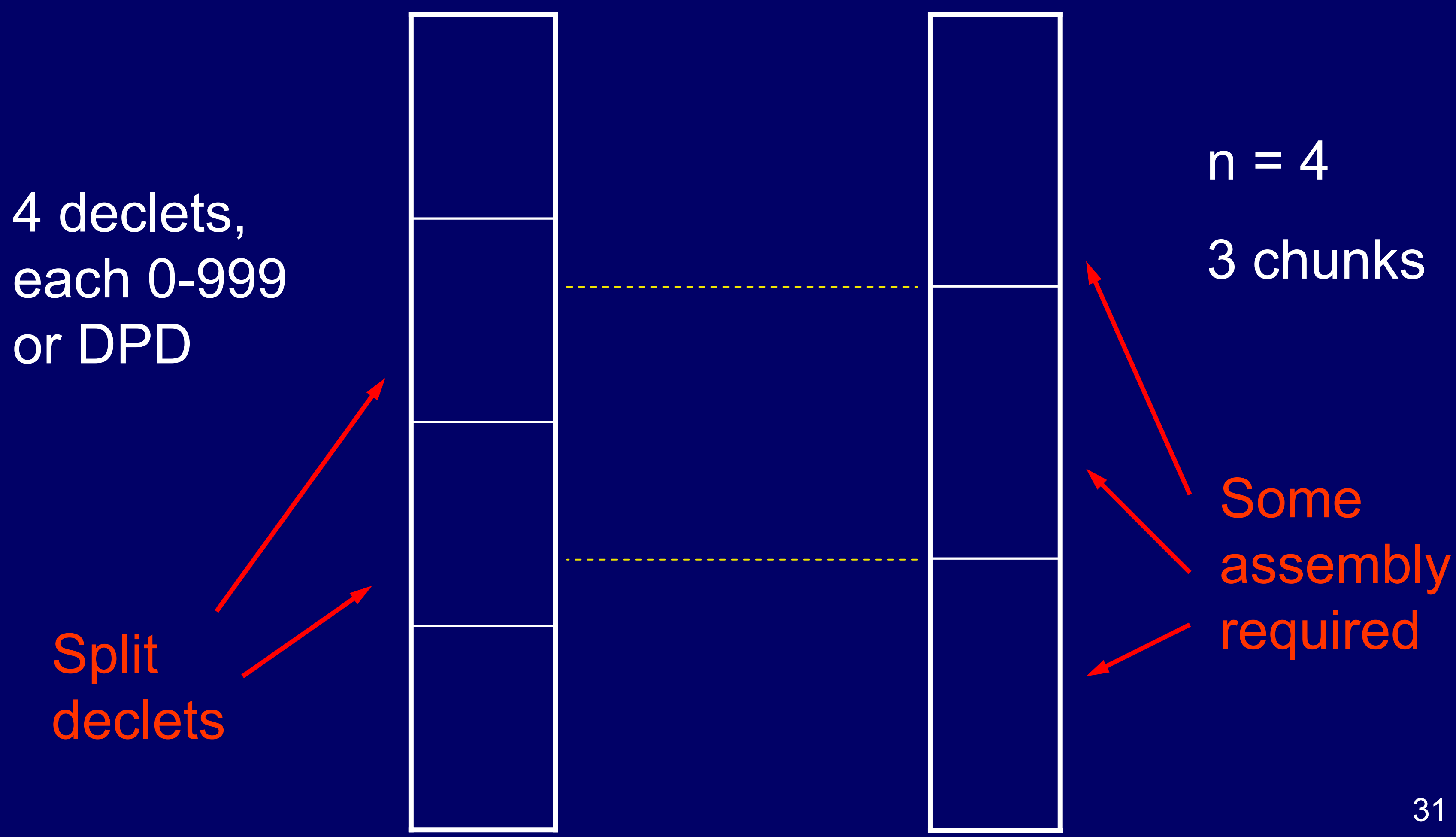


‘Bad compiler’ variant

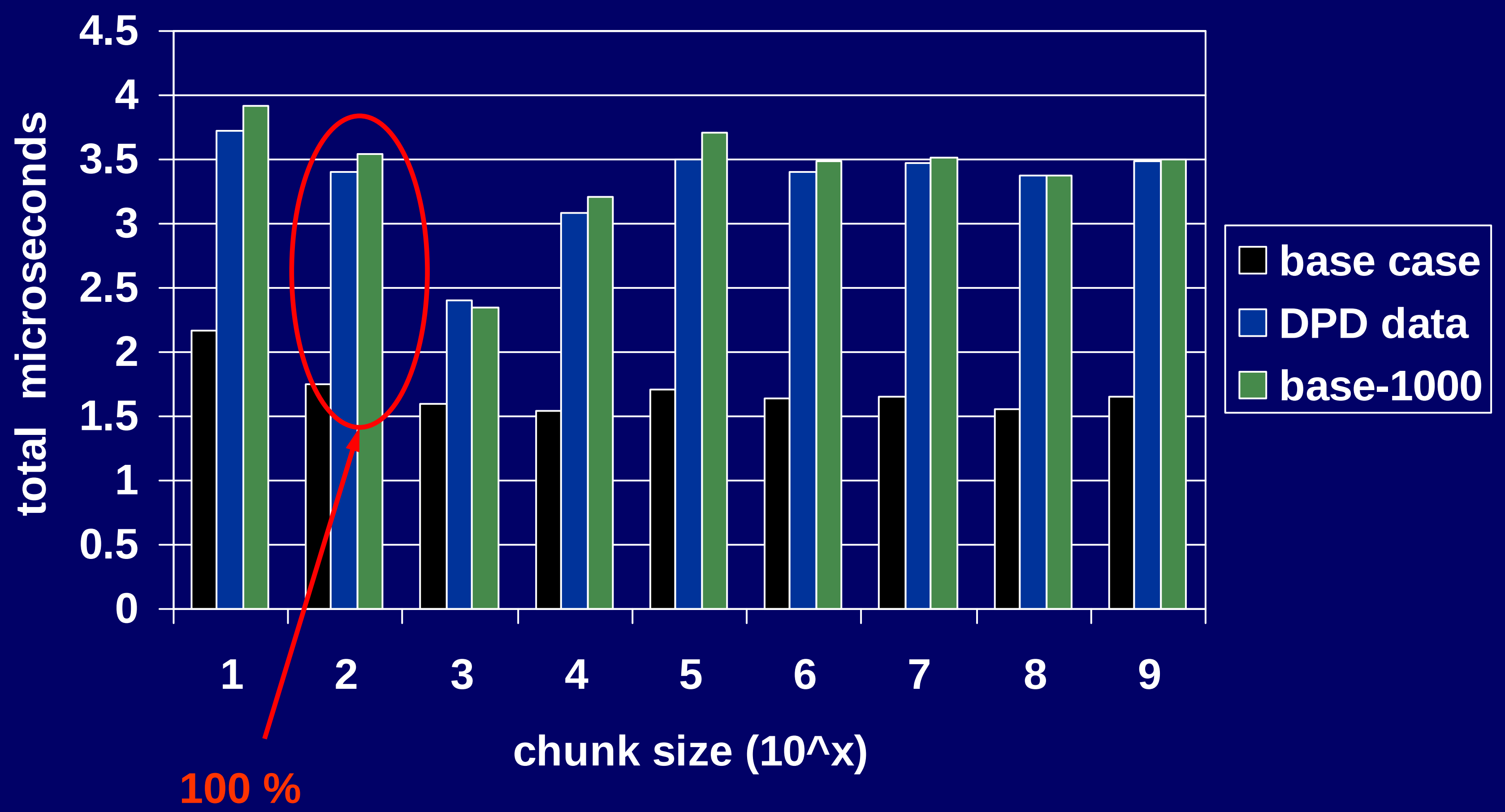


Conversions when n not 3 x k

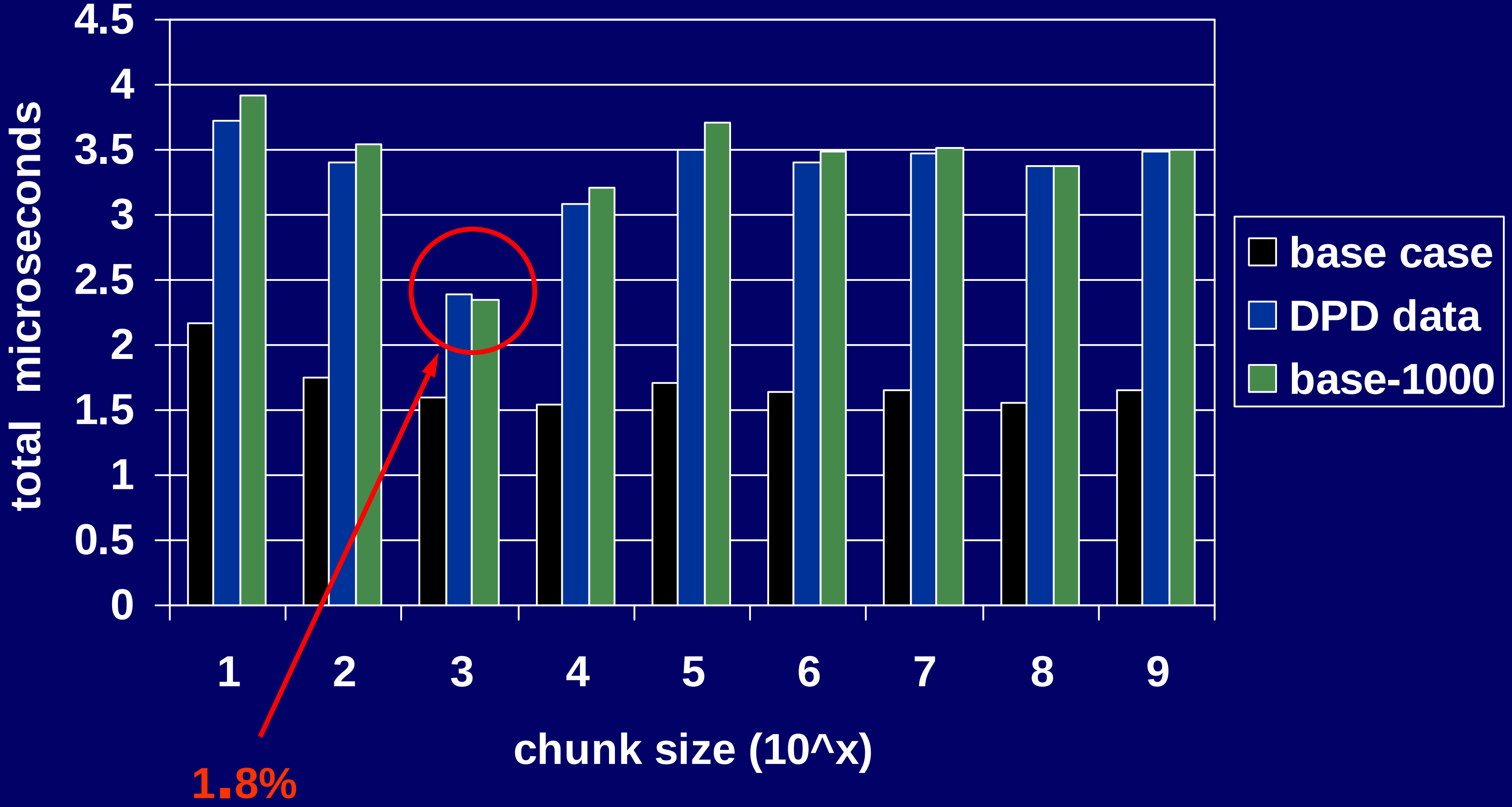
(Need lookup in these cases anyway)



‘Toy compiler’ variant



‘Toy compiler’ variant



Cost of DPD lookup in ‘telco’

- Cost of DPD lookup, even when more conversions than operations, is < 2%
  - only chunk size = 3 shows measurable cost
  - most other sizes need lookup anyway

- Cost also depends on compiler and optimization level (MS or GCC? -O2 or Default?), but always < 2%
  - lookup sometimes optimizes to faster code

Cost of DPD: microbenchmarks

- On average, DPD lookup costs 3.5–4.3% of *conversions* (decimal64 ←→ internal)
- Cost in any application will always be less than this
- (Tested MS compiler and GCC with several levels of optimization.)

Cost of reordering fields

- Proposal to move the combination field between pure exponent bits and significand:

Sign	Pure	Comb. field	Significand

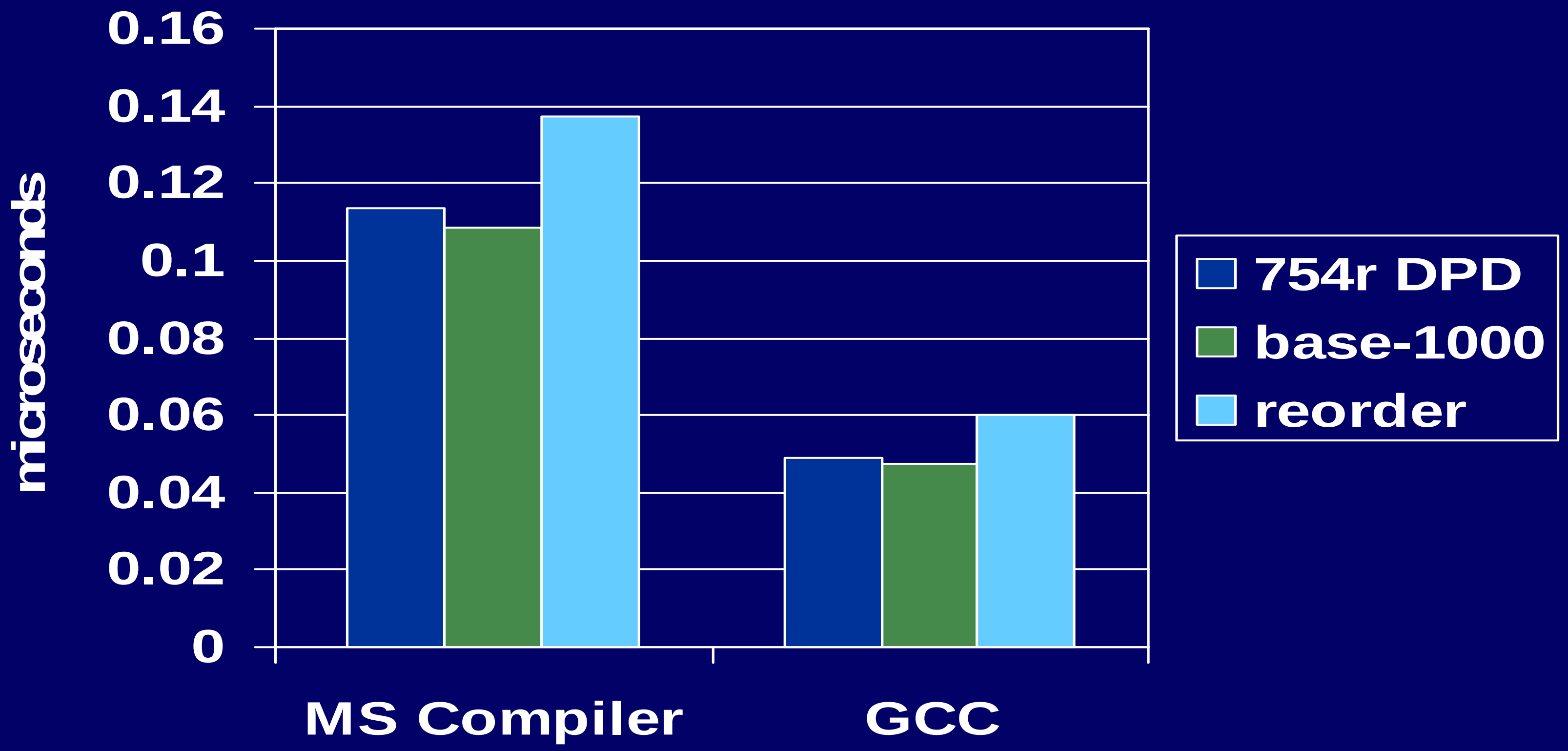
Decode: exponent = (pure x 102) + (comb/10)
d0 = rem(comb, 10)

Encode: pure = exponent / 102
comb = rem(exponent, 102) x 10 + d0

Cost of reordering fields

- Microbenchmarks show that re-ordering adds on average 25–33% to the cost of conversions
  - 2 multiplies and some adds, or one divide
- This cost is not large, but is much more than is saved by using base-1000 and chunk size 3 …

Average cost per conversion



Chunk size = 3

Necessary conversions

- Conversion of existing decimal data (BCD, character data, *etc*.), and also database internal formats
  - at least 85% of these data are base-10 or base-100
  - often can go directly to internal form for software

Avoidable conversions

- Variables within routines
  - store-back for every assignment is expensive in software (and unnecessary)
  - use of internal form is analogous to hardware
- Parameters to ‘external’ routines (I/O *etc*.)
  - less significant, but within-platform optimization would be possible

DPD is not just for performance

- The fundamental unit of decimal numbers is the *digit* – often coded as BCD or ASCII
  - allows fast and efficient shifts (alignment), rounding, and digit counting

- Encodings such as DPD and Chen-Ho provide a *direct* mapping to digits, without arithmetic (adds, carries) …

DPD → BCD equations (espresso)

```
pqrstuvwxy → abcd efgh ijkm

a = (!s&v&w) | (t&v&w&x) | (v&w&!x);
b = (p&s&x) | (p&!w) | (p&!v);
c = (q&s&x) | (q&!w) | (q&!v);
d = (r);

e = (t&v&!w&x) | (s&v&w&x) | (!t&v&x);
f = (p&t&v&w&x) | (s&!x) | (s&!v);
g = (q&t&w) | (t&!x) | (t&!v);
h = (u);

i = (t&v&w&x) | (s&v&w&x) | (v&!w&!x);
j = (p&!s&!t&w) | (s&v&!w&x) | (p&w&!x) | (!v&w);
k = (q&!s&!t&v&w) | (q&v&w&!x) | (t&v&!w&x) | (!v&x);
m = (y);
```


(Compression is much simpler)

Base-1000 → BCD equations

```
a = (p&q&!r&s&!t&u&!v&!w&!x) | (p&q&r&!s&!t&!u&!v&!w&x)
    | (p&q&r&!s&!t&!u&v&!w&x) | (p&q&!r&s&!t&!u&!v&!w&x)
    | (p&q&!r&s&!t&u&!v&w&!x) | (p&q&r&!s&!t&u&!v&!w&x) |
    (p&q&r&!s&!t&!u&!v&!w&!x) | (p&q&!r&s&!t&!u&!v&w&x) |
    (p&q&r&!s&!u&v&!w&!x) | (p&q&!r&s&!t&u&!v&x) |
    (p&q&r&!t&u&!v&!w&!x) | (p&q&!r&s&!t&!u&!v&!x) |
    (p&q&r&!s&!t&u&v) | (p&q&r&s&u&v&!w) |
    (p&q&r&s&u&!v&!w&x) | (p&q&!r&s&!t&v) |
    (p&q&r&s&!t&!u) | (p&q&r&w) | (p&q&t);
```

Base-1000 → BCD equations

```
b = (!p&q&r&s&!t&!u&!v&!w&!x) | (p&!q&!r&s&t&!u&!v&!w&!x) | (p&!q&r&!s&t
    &u&!v&!w&!x) | (!p&q&r&s&!t&!u&v&!w&x) | (p&!q&!r&s&t&!u&v&!w&x) | (
    p&!q&r&!s&!t&!u&v&!w&!x) | (p&!q&!r&!s&!t&u&!v&!w&!x) | (!p&q&r&!s&t
    &!u&!v&!w&x) | (p&!q&!r&s&!t&!u&v&w&!x) | (p&!q&!r&!s&t&!u&!v&w&!x) | (
    p&!q&r&s&!t&!u&!v&w&!x) | (p&!q&!r&s&t&u&!v&w&!x) | (p&q&!r&!s&!t&u&!v
    &w&!x) | (p&!q&!r&!s&t&!u&!w&x) | (!p&q&r&!s&t&!u&!v&!w&!x) | (p&q&!r
    &!s&!t&v&!w&!x) | (!p&q&r&s&!t&!u&!v&w&!x) | (p&!r&!s&!t&!u&!v&w&!x) | (
    p&q&!r&!s&!t&!u&!v&!w&!x) | (p&!q&!r&s&t&!u&!v&w&!x) | (p&!q&!r&!s&!t
    &!u&!v&!w&x) | (p&!q&!r&s&!t&!u&v&w&x) | (p&!q&!r&s&!t&u&!v&!w&x) | (
    p&!q&!r&!s&!t&!u&!v&w&x) | (p&q&!r&!s&!t&u&v&!w&x) | (p&!q&r&!s&t&u&v &!w&x)
    &!w&x) | (p&!q&r&!s&!t&u&!v&!w&x) | (p&!q&r&!s&t&u&!v&w&!x) | (p&!q&!r
    &s&t&!u&!v&w&x) | (!p&q&r&!s&t&!u&v&!w) | (!p&q&r&s&!t&!u&v&!w&!x) | (
    p&!q&s&t&!u&!v&!w&x) | (p&!q&r&!s&!t&!u&v&w&!x) | (p&!q&!r&!s&!t&u&v &!w&x)
    | (p&!q&r&s&!u&!v&!w&!x) | (p&!q&!r&s&t&!u&v&!w&!x) | (!p&q&r &s&!t&!u&!v&x)
    &s&!t&!u&!v&x) | (p&!q&!r&s&!t&!u&v&!w) | (p&!q&r&!s&t&u&v&!w&!x) | (
    p&!q&!r&!s&t&!u&!v&!w) | (p&!q&!r&s&t&u&!v&!w) | (p&!q&r&!s&!t&u&!v&!w &!x)
    | (p&!q&!r&s&!t&!u&!v&!w&x) | (p&!q&r&s&!t&!u&!w&x) | (p&q&!r&!s
    &!t&u&!v&!w) | (p&!q&r&!s&t&u&!v&x) | (p&!q&!r&!t&!u&!v&!w&!x) | (!p
    &q&r&!s&t&!u&w) | (p&!r&!s&!t&u&v&!w&!x) | (p&!q&!r&!s&!t&!v&w&!x) | (
    p&!q&r&!s&!t&!u&v&x) | (p&!q&!r&s&!t&u&!v&!x) | (p&!q&r&!s&!t&u&v&!w) | (
    p&q&!r&!s&!t&!u&v&!w&x) | (p&!q&!r&!s&!t&u&!v&x) | (p&r&s&t&u&!v&!w&x) | (
    p&!q&r&!s&t&!u&v) | (p&!q&r&s&!t&!u&v&!x) | (p&!q&!r&s&!t&!v&w) | (p
    &!q&!r&!s&t&!u&w&x) | (p&!q&r&s&!t&!u&w&x) | (p&!q&!r&s&t&u&w&x) | (
    p&q&!r&!s&!t&u&w&x) | (!p&q&r&s&!t&v&w) | (p&!q&!r&!s&!t&!u&v) | (p&!q
    &!r&s&t&v&w) | (p&!q&!r&!s&u&v&w) | (p&!q&r&t&u&v&w) | (p&!q&!r&!s&t
    &!u&v&!x) | (p&q&!r&!s&!t&v&w) | (!p&q&r&s&t&!u) | (p&!q&r&!s&!t&u&w) | (
    p&q&!r&!s&!t&!u&!v&x) | (p&r&s&t&!u&v&!w) | (p&r&s&t&u&!v&!x) | (p&!q
    &r&!s&!u&!v) | (p&!q&!r&!s&t&u) | (p&!q&r&s&t&w) | (p&!q&r&s&!t&u) | (
    p&!q&s&u&v) | (!p&q&r&u);
```

Base-1000 → BCD equations

```
c = (!p&!q&r&s&!t&u&!v&!w&!x) | (!p&q&!r&!s&t&!u&!v&!w&!x) | (p&!q&!r&s&t &!u&!v&!w&!x) |
    (p&!q&r&!s&t&u&!v&!w&!x) | (!p&q&!r&!s&t&u&!v&w&!x) | ( !p&!q&r&s&t&!u&!v&w&!x) |
    (p&!q&!r&s&t&!u&v&!w&x) | (!p&q&r&!s&!t&!u &!v&w&!x) | (p&!q&r&!s&!t&!u&v&!w&!x) |
    (!p&q&!r&!s&!t&!u&v&w&!x) | ( p&!q&r&s&!t&!u&!v&w&!x) | (p&!q&!r&s&t&u&!v&w&!x) |
    (!p&!q&r&s&!t&u&!v &w&!x) | (p&q&!r&!s&!t&!u&!v&w&!x) | (!p&q&!r&!s&t&!u&v&!w&x) |
    (!p&q!r&s&t&u&!v&!w&!x) | (!p&q&!r&!s&!t&!u&!v&!w&x) | (!p&q&!r&!s&t&!u&!v &w&!x) |
    (!p&!q&r&s&t&!v&!w&x) | (!p&q&!r&s&t&u&v&!w&x) | (p&q&!r&!s &!t&!u&!v&!w&!x) |
    (p&!q&!r&s&t&!u&!v&w&!x) | (!p&q&!r&s&t&!u&!v&w&x) | ( !q&r&s&t&!u&!v&!w&!x) |
    (!p&q&!r&!s&!t&!u&v&w&x) | (!p&q&!r&!s&t&u&v &!w&x) | (p&!q&r&!s&t&u&v&!w&x) |
    (p&!q&r&!s&!t&u&!v&!w&x) | (p&!q&r &!s&t&u&!v&w&!x) | (!p&q&!r&!s&t&!u&v&!w&!x) |
    (!p&q&!r&!s&!t&!u&!v&!w &!x) | (p&!q&!r&s&t&!u&!v&w&x) | (!p&q&!r&!s&t&u&!v&!w) |
    (!p&q&!r&!s &!t&!u&v&!w&x) | (!p&!q&r&s&!t&u&!v&x) | (!p&q&r&!s&!t&!u&v&!w&x) | (
    !p&q&!r&s&!t&u&!v&!w&x) | (!p&q&!r&!s&!t&u&!v&!w&x) | (!p&q&!r&!s&t&!u &!v&x) |
    (!p&q&r&!s&!t&!u&!v&!w) | (!p&q&!r&s&!t&u&v&!w&x) | (p&!q&s &t&!u&!v&!w&x) |
    (!p&q&!r&s&t&u&!v&w&!x) | (!p&q&!r&!s&!t&u&!v&w&x) | ( p&!q&r&!s&!t&!u&v&w&!x) |
    (!p&q&!r&s&t&!u&!v&!w&x) | (q&!r&!s&!t&!u&v &!w&!x) | (p&!q&r&s&!u&!v&!w&!x) |
    (p&!q&!r&s&t&!u&v&!w&!x) | (!p&q&!r &s&u&v&!w&!x) | (p&!q&r&!s&t&u&v&!w&!x) |
    (p&!q&!r&s&t&u&!v&!w) | (p &!q&r&!s&!t&u&!v&!w&!x) | (!p&q&!r&t&u&!v&w&x) |
    (p&!q&r&s&!t&!u&!w&x) | ( p&q&!r&!s&!t&u&!v&!w) | (!p&q&!r&s&!t&u&!v&!w&!x) |
    (!p&!q&r&s&t&!u&w &x) | (p&!q&r&!s&t&u&!v&x) | (!p&q&r&!s&!t&!u&w&x) |
    (!p&q&!r&t&u&v&!w&!x) | (p&!q&r&!s&!t&!u&v&x) | (p&!q&r&!s&!t&u&v&!w) | (p&q&!r&!s&!t
   &!u&v&!w&x) | (p&!q&r&!s&t&!u&v) | (p&!q&r&s&!t&!u&w&x) | (q&!r&!s&!t &!u&!v&w) |
    (p&!q&!r&s&t&u&w&x) | (!p&q&r&!s&!t&!u&v&!x) | (!p&q&!r&!s &!t&u&!v&!x) |
    (p&q&!r&!s&!t&u&w&x) | (!p&q&!r&s&t&!u&!v&!x) | (!p&q &!r&s&!t&u&w) | (p&!q&!r&s&t&v&w)
    | (p&!q&r&t&u&v&w) | (!p&q&!r&s&t&u &!v&x) | (p&q&!r&!s&!t&v&w) | (p&!q&r&!s&!t&u&w) |
    (q&!r&!s&!t&u&v) | ( p&q&!r&!s&!t&!u&!v&x) | (!p&q&!r&s&t&!u&v) | (!p&q&!r&t&v&w) |
    (p&!q &r&!s&!u&!v) | (!p&q&!r&s&!t&!u) | (p&!q&r&s&t&w) | (p&!q&r&s&!t&u) | (p&!q&s&u&v)
    | (!q&r&s&t&u) | (!q&r&s&v);
```

Base-1000 → BCD equations

```
d = (!p&!q&r&!s&!t&!u&!v&!w&!x) | (p&q&r&!s&t&!u&!v&!w&!x) | (!p&q&!r&!s &t&u&!v&w&!x) |
    (!p&!q&r&!s&!t&u&!v&w&!x) | (!p&q&r&!s&!t&!u&!v&w&!x) | ( p&!q&r&s&t&!u&!v&!w&x) |
    (p&!q&!r&!s&!t&u&!v&!w&!x) | (p&!q&!r&s&!t&!u &v&w&!x) | (p&!q&!r&!s&t&!u&!v&w&!x) |
    (p&!q&r&s&!t&!u&!v&w&!x) | (p &q&!r&!s&!t&u&!v&w&!x) | (p&q&r&!s&!t&!u&v&!w&x) |
    (!p&q&!r&s&t&u&!v &!w&!x) | (p&q&r&!s&!t&u&!v&!w&x) | (!p&!q&!r&s&t&u&!v&!w&x) | (p&!q
    &!r&!s&t&!u&!w&x) | (p&q&r&!s&!u&v&!w&!x) | (p&q&!r&!s&!t&v&!w&!x) | (
    !p&!q&r&s&!t&!u&!v&!w&x) | (!p&!q&r&!s&!t&u&!v&!w&x) | (p&!r&!s&!t&!u &!v&w&!x) |
    (p&q&!r&!s&!t&!u&!v&!w&!x) | (!p&!q&r&s&!t&!u&!v&w&x) | ( p&!q&!r&!s&!t&!u&!v&!w&x) |
    (!p&!q&r&!s&!t&!u&v&!w&x) | (p&!q&!r&s&!t &!u&v&w&x) | (!p&!q&!r&s&t&!u&v&!w) |
    (!p&!q&r&!s&!t&!u&!v&w&!x) | ( !p&!q&r&!s&u&!v&!w&!x) | (p&!q&!r&s&!t&u&!v&!w&x) |
    (p&!q&!r&!s&!t&!u &!v&w&x) | (p&q&!r&!s&!t&u&v&!w&x) | (!p&q&!r&!s&t&u&v&!w&x) | (!p&!q
    &r&!s&!t&u&v&!w&x) | (!p&!q&r&!s&t&u&v&!w) | (!p&q&!r&!s&t&u&!v&!w) | (
    !p&q&r&!s&!t&!u&v&!w&x) | (!p&q&!r&s&!t&u&!v&!w&x) | (!p&!q&r&!s&t&u &!v&!w&x) |
    (!p&q&r&!s&!t&!u&!v&!w) | (!p&q&!r&s&!t&u&v&!w&x) | (p&!q &!r&!s&!t&u&v&!w&x) |
    (!p&q&!r&s&t&!u&!v&!w&x) | (p&!q&r&s&!u&!v&!w&!x) | ( !p&!q&r&!s&!t&!u&v&!w&!x) |
    (p&q&r&!t&u&!v&!w&!x) | (!p&q&!r&s&u&v&!w &!x) | (p&!q&!r&s&!t&!u&v&!w) |
    (p&!q&!r&!s&t&!u&!v&!w) | (!p&!q&r&!s &!t&!u&!v&x) | (p&q&r&!s&t&!u&!v&x) |
    (p&!q&!r&s&!t&!u&!v&!w&x) | (!p &!q&!r&s&t&u&!v&!x) | (!p&q&!r&t&u&!v&w&x) |
    (p&!q&r&s&!t&!u&!w&x) | ( p&q&!r&!s&!t&u&!v&!w) | (!p&q&!r&s&!t&u&!v&!w&!x) |
    (!p&!q&r&!s&!t&u &w&x) | (p&!q&!r&!t&!u&!v&!w&!x) | (p&q&r&t&u&v&!w&x) | (!p&!q&r&s&!t
    &!u&!v&!x) | (!p&!q&r&!s&t&u&w) | (p&!r&!s&!t&u&v&!w&!x) | (!p&!r&s&t &u&v&!w) |
    (!p&q&r&!s&!t&!u&w&x) | (!p&q&!r&t&u&v&!w&!x) | (p&!q&!r&!s &!t&!v&w&!x) |
    (!p&!q&r&!s&!u&v&w) | (q&r&s&t&u&v&!w) | (p&!q&!r&s&!t &u&!v&!x) | (p&q&r&t&!u&v&!w&x) |
    (p&q&!r&!s&!t&!u&v&!w&x) | (p&!q&!r &!s&!t&u&!v&x) | (p&r&s&t&u&!v&!w&x) |
    (p&q&r&!s&!t&u&v) | (p&!q&r&s &!t&!u&v&!x) | (!p&!q&r&!s&!t&u&v&!x) |
    (p&!q&!r&s&!t&!v&w) | (p&!q&!r &!s&t&!u&w&x) | (p&!q&r&s&!t&!u&w&x) |
    (!p&q&r&!s&!t&!u&v&!x) | (p&q &!r&!s&!t&u&w&x) | (p&q&r&s&u&v&!w) | (p&q&r&t&u&!v&!w) |
    (p&!q&!r&!s &!t&!u&v) | (!p&q&!r&s&t&!u&!v&!x) | (!p&q&!r&s&!t&u&w) | (p&!q&!r&!s
    &u&v&w) | (p&q&r&s&u&!v&!w&x) | (p&!q&r&t&u&v&w) | (p&!q&!r&!s&t&!u&v &!x) |
    (!p&q&!r&s&t&u&!v&x) | (p&q&!r&!s&!t&v&w) | (!p&q&s&t&u&w) | ( p&q&!r&!s&!t&!u&!v&x) |
    (p&r&s&t&!u&v&!w) | (!p&q&!r&s&t&!u&v) | (p&q &r&t&u&v&!x) | (p&q&r&s&t&!w&x) |
    (!p&q&!r&t&v&w) | (p&r&s&t&u&!v&!x) | ( p&r&s&t&u&v) | (!p&q&!r&s&!t&!u) |
    (!p&!q&r&!s&t&!u) | (p&q&r&s&!t&!u) | ( !p&!r&s&t&w) | (p&!q&!r&!s&t&u) |
    (p&!q&r&s&t&w) | (p&!q&r&s&!t&u) | ( p&q&r&s&t&!x) | (p&q&r&w);
```

Base-1000 → BCD equations

```
e = . . .
f = . . .
g = . . .
h = . . .
i = . . .
j = . . .
k = . . .
m = (y);
```

as bad or worse than a – d

Conclusions

- DPD with BCD has the performance advantage in hardware
- DPD is simpler and more flexible
  - and is already in use
- DPD cost is insignificant in software
- 754r should allow computation in internal form, or even more global optimizations