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)

- **Ai(0:9)**
- **Bi(0:9)**
- **7bit add**
- **10 bit add**
- **Ugly carry at 10 bit boundary**
- **Carry chain**
- **Critical path**
- **A+B+24**
- **A+B**
- **A+B+1**
- **A+B+25**
- **Sum based on carry select**
- **0000011000 (24)**
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)

0110(6)

3b

Ai(0:3)

Bi(0:3)

4 bit add

A+B+6

4 bit add

A+B

4 bit add

A+B+7

4 bit add

A+B+1

Critical path

Carry Chain
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

<table>
<thead>
<tr>
<th>Base 1000</th>
<th>BCD</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Non-Optimal</strong> 3 way add</td>
<td>A(0:9) + B(0:9) + 24 7 bit adder</td>
</tr>
<tr>
<td><strong>Optimal Binary Adder tree for Greater than</strong> Group carries</td>
<td>A(0:3) + B(0:3) + 6 3 bit adder</td>
</tr>
<tr>
<td>And carry select</td>
<td>6 Groups</td>
</tr>
<tr>
<td></td>
<td>16 Groups</td>
</tr>
</tbody>
</table>

Base 1000 has one more level in the non-optimal 3 way 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)

20 bits but not in base 1000

To get to base 1000
Need divide by 1000
And remainder
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

20 bit non base1000

\[ \begin{align*}
XYZ \times 123 & \\
XYZ \times 789 & \\
XYZ \times DEF & \\
XYZ \times 0 & \\
XYZ \times 456 & \\
XYZ \times ABC & \\
\end{align*} \]

Div 1000

\[ \begin{align*}
1 & \\
3 & \\
5 & \\
7 & \\
9 & \\
11 & \\
0 & \\
2 & \\
4 & \\
6 & \\
8 & \\
10 & \\
\end{align*} \]

Base 1000 addition
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](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 $\approx 0.01$ $\mu$s (0.5 % this chart)  
I/O $\approx 0.44$ $\mu$s (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

55 %
‘Bad compiler’ variant

-chunk size (10^x)-

-0.9%

-base case
-DPD data
-base-1000

-Total microseconds

-3.5
-3
-2.5
-2
-1.5
-1
-0.5
-0
Conversions when $n$ not $3 \times k$

(Need lookup in these cases anyway)

4 declets, each 0-999 or DPD

$n = 4$

3 chunks

Split declets

Some assembly required
‘Toy compiler’ variant

- Chunk size (10^x)
- Total microseconds

- Base case
- DPD data
- Base-1000

100 %
‘Toy compiler’ variant

![Bar chart showing total microseconds for different chunk sizes (10^x)]

- Base case
- DPD data
- Base-1000

1.8%
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:

<table>
<thead>
<tr>
<th>Sign</th>
<th>Pure</th>
<th>Comb. field</th>
<th>Significand</th>
</tr>
</thead>
</table>

**Decode:**

\[
\text{exponent} = (\text{pure} \times 102) + \left(\frac{\text{comb}}{10}\right)
\]

\[
d0 = \text{rem}(\text{comb}, 10)
\]

**Encode:**

\[
\text{pure} = \text{exponent} / 102
\]

\[
\text{comb} = \text{rem}(\text{exponent}, 102) \times 10 + d0
\]
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 $\rightarrow$ BCD equations (espresso)

$pqrstuvwxy \rightarrow abcd efgh ijk$ m

\[ a = (!s&v&w) \mid (t&v&w&x) \mid (v&w&!x); \]
\[ b = (p&s&x) \mid (p&!w) \mid (p&!v); \]
\[ c = (q&s&x) \mid (q&!w) \mid (q&!v); \]
\[ d = (r); \]
\[ e = (t&v&!w&x) \mid (s&v&w&x) \mid (!t&v&x); \]
\[ f = (p&t&v&w&x) \mid (s&!x) \mid (s&!v); \]
\[ g = (q&t&w) \mid (t&!x) \mid (t&!v); \]
\[ h = (u); \]
\[ i = (t&v&w&x) \mid (s&v&w&x) \mid (v&w&!x); \]
\[ j = (p&!s&!t&w) \mid (s&v&!w&x) \mid (p&w&!x) \mid (!v&w); \]
\[ k = (q&!s&!t&v&w) \mid (q&v&w&!x) \mid (t&v&w&x) \mid (!v&x); \]
\[ m = (y); \]

(Compression is much simpler)
Base-1000 $\rightarrow$ BCD equations

$$a = (p\&q\&!r\&s\&!t\&u\&!v\&!w\&!x) \mid (p\&q\&r\&!s\&!t\&!u\&!v\&!w\&x)$$
$$\mid (p\&q\&r\&!s\&!t\&!u\&v\&!w\&x) \mid (p\&q\&!r\&s\&!t\&!u\&!v\&!w\&x)$$
$$\mid (p\&q\&!r\&s\&!t\&u\&!v\&w\&!x) \mid (p\&q\&r\&!s\&!t\&u\&!v\&w\&x)$$
$$\mid (p\&q\&r\&!s\&!t\&!u\&!v\&!w\&!x) \mid (p\&q\&r\&!s\&!t\&!u\&!v\&w\&x)$$
$$\mid (p\&q\&r\&!s\&!u\&v\&!w\&!x) \mid (p\&q\&!r\&s\&!t\&u\&!v\&x)$$
$$\mid (p\&q\&r\&!t\&u\&!v\&!w\&!x) \mid (p\&q\&!r\&s\&!t\&!u\&!v\&!x)$$
$$\mid (p\&q\&r\&!s\&!t\&u\&v) \mid (p\&q\&r\&s\&u\&v\&!w)$$
$$\mid (p\&q\&r\&s\&u\&!v\&!w\&x) \mid (p\&q\&!r\&s\&!t\&v)$$
$$\mid (p\&q\&r\&s\&!t\&!u) \mid (p\&q\&r\&w) \mid (p\&q\&t);$$
Base-1000 → BCD equations

\[
b = \left( !p&q&r&s&!t&!u&!v&!w&!x \right) \lor \left( p&!q&!r&s&t&!u&!v&!w&!x \right) \lor \left( !p&q&r&!s&s&!t&!u&!v&!w&!x \right) \lor \left( p&!q&r&s&!t&!u&!v&!w&!x \right) \lor \left( p&!q&r&!s&s&!t&!u&!v&!w&!x \right) \lor \left( p&q&r&s&t&!u&!v&!w&!x \right) \lor \left( p&q&r&!s&s&t&!u&!v&!w&!x \right) \lor \left( p&q&r&!s&!s&t&!u&!v&!w&!x \right) \lor \left( p&q&r&!s&!t&!u&!v&!w&!x \right) \lor \left( p&q&r&!s&t&!u&!v&!w&!x \right) \lor \left( p&q&r&!s&t&!u&v&!w&!x \right) \lor \left( p&q&r&!s&t&u&!v&!w&!x \right) \lor \left( p&q&r&!s&t&u&v&!w&!x \right) \lor \left( p&q&r&!s&t&u&v&w&!x \right) \lor \left( p&q&r&!s&t&u&v&x \right) ;
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]
Base-1000 → BCD equations

c = ( \neg p \land \neg q \land r \land s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land \neg q \land \neg r \land s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
(p \land q \land \neg r \land \neg s \land \neg t \land u \land \neg v \land \neg w \land x ) |
Base-1000 $\rightarrow$ BCD equations

\[ d = (\neg p \wedge \neg q \wedge \neg r \wedge \neg s \wedge \neg t \wedge \neg u \wedge \neg v \wedge \neg w \wedge \neg x) \]
Base-1000 $\rightarrow$ BCD equations

\[
\begin{align*}
e &= \ldots \\
f &= \ldots \\
g &= \ldots \\
h &= \ldots \\
i &= \ldots \\
j &= \ldots \\
k &= \ldots \\
m &= (y);
\end{align*}
\]

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