A Decimal Floating-Point Specification

Michael F. Cowlishaw,
IBM UK
P.O. Box 31, Birmingham Rd.
Warwick CV34 5JL. UK
mfc@uk.ibm.com

Eric M. Schwarz, Ronald M. Smith, Charles F. Webb
IBM Server Division
2455 South Rd., MS:P310
Poughkeepsie, NY 12601 USA
eschwarz@us.ibm.com

Abstract

Even though decimal arithmetic is pervasive in financial and commercial transactions, computers are still implementing almost all arithmetic calculations using binary arithmetic. As chip real estate becomes cheaper it is becoming likely that more computer manufacturers will provide processors with decimal arithmetic engines. Programming languages and databases are expanding the decimal data types available while there has been little change in the base hardware. As a result, each language and application is defining a different arithmetic and few have considered the efficiency of hardware implementations when setting requirements.

In this paper, we propose a decimal format which meets the requirements of existing standards for decimal arithmetic and is efficient for hardware implementation. We propose this specification in the hope that designers will consider providing decimal arithmetic in future microprocessors and that future decimal software specifications will consider hardware efficiencies.

1. Introduction

People habitually perform arithmetic in base 10. When calculations are moved to computers there is usually a loss in translating decimal fractions to a binary representation. Common decimal values such as 0.1 can be represented exactly in decimal but can only be approximated in a binary floating-point format. Traditionally, financial transactions are therefore performed in a scaled binary or decimal integer format. Both of these formats are fixed point and so, even though they avoid fraction approximation, they are severely limited in the range of numbers they can represent. Further, in many applications, such as the calculation of a monthly mortgage payment, rounding is required. In the fixed point formats, rounding must be explicitly applied in software rather than being provided by the hardware. To address these and other limitations, we propose implementing a decimal floating-point format. But what should this format be? This paper discusses the issues of defining a decimal floating-point format.

First, we consider the goals of the specification. It must be compliant with standards already in place. One standard we consider is the ANSI X3.274-1996 (Programming Language REXX) [1]. This standard contains a definition of an integrated floating-point and integer decimal arithmetic which avoids the need for two distinct data types and representations. The other relevant standard is the ANSI/IEEE 854-1987 (Radix-Independent Floating-Point Arithmetic) [2]. The IEEE 754 standard [3] is a binary standard which also meets this radix-independent standard. The radix-independent standard gives us guidelines for formats and exponent ranges that need to be considered and specifically describes requirements for base 10 floating point.

A floating-point format must also permit efficient implementation. One of the difficulties with decimal arithmetic is that it is less efficient than binary arithmetic. Representations take up more memory and execution time of an operation is inherently longer. However, if the data is already stored in memory in a decimal format and conversions between decimal and binary formats are costly, then it can be more efficient to carry out computations directly in decimal. Further, if the chosen format requires very little hardware to support then it is likely that future microprocessors would implement it in hardware, just as most implement binary floating-point arithmetic today. From a financial and business software standpoint, decimal floating-point hardware is as attractive as a graphics engine is attractive to a games developer.

Other considerations in developing a specification

0-7695-1150-3/01 $10.00 © 2001 IEEE

Language	Platform	Support	Precision	Comments
C & C++	S/390	Fixed	31	C only
	AS/400	Fixed	31	C only
	Others	Fixed	31-38	In various libraries
COBOL	All	Fixed	31	32 digit Floating proposed
C#	All	Float	28	
Java	All	Floating	infinite	Using IBM BigDecimal class; java.math is fixed point
OS/400 CL	AS/4000	Fixed	15	Scale <= 9
PL/I	S/390	Fixed	15	The FLOAT Decimal data type is actually binary
	AS/400	Fixed	15	
	Others	Fixed	31	
PSM	All	Fixed	31	Implemented via translation to C
Rexx	All	Floating	infinite	
RPG	AS/400	Fixed	30	Scale <= 9

Table 1. Software and Platforms Implementing Decimal Arithmetic

are that it should be efficient for reading and storing decimal data in databases. Decimal data are usually stored in Binary Coded Decimal (BCD) notation. Typical decimal data are fixed point with up to 20 digits of precision, including up to 10 fractional digits. There needs to be efficiency in reading the data out of a database, operating on it, scaling and rounding it, and then storing the data back into the database with a fixed point notation. These requirements favor a BCD format.

A further constraint on a specification is the need to allow for future expansion of the notation. An architecture or format should be robust enough to grow with future changing needs. One possible mechanism could be through the use of special values. There needs to be enough room in the format to support infinities, Not-a-Number (NaN) values, and both positive and negative zeros. In the future there might be additional values, such as subnormal numbers, considered too.

In summary, the goals of the specification are:

1. It should allow efficient hardware or software implementation of decimal floating-point
2. It should support numbers to be used for the floating-point and integer decimal arithmetic defined in ANSI X3.274-1996 (Programming Language REXX)
3. It should support numbers and values to be used for the floating-point arithmetic defined in ANSI/IEEE 854-1987 (Radix-Independent Floating-Point Arithmetic)
4. It should allow efficient use of existing data in databases
5. It should allow for future expansion.

The remainder of this paper discusses a specification for decimal floating-point data types which best meets these goals and fits within appropriate lengths. Existing hardware architectures suggest that representations be 32, 64, 80, or 128 bits in length, and these are convenient for software, too. It is proposed that only 64 and 128 bit representations be defined at this time, forming the single and double formats as described in the IEEE 854 standard. Some prior work by Johnstone and Petry [4] showed a 32 bit format but this length does not meet most software requirements. Hull [5] suggests a variable precision format but this does not map easily to hardware implementations. Note that the 854 standard requires a single precision be defined and recommends that there should be an extended precision. Double precision satisfies the requirements of single-extended precision.

Software requirements are first discussed, followed by the overall decimal format, which is similar to that described by Ris[6]. These are followed by discussion on the efficiency and requirements for each part of the format, including the representation of the integer, exponent, and special values, the ordering of parts, and the length of the exponents. Finally, complete formats will be described.

2. Software Requirements

Table 1 shows a list of several common languages which define decimal arithmetic (sometimes using libraries). Also listed is the support that is provided on a given platform. The IBM S/390 (z-Series) platform has 31 digit decimal integer arithmetic hardware and this is used by many of the languages including C and COBOL. Java and REXX both provide floating-point

decimal representation with an unlimited number of digits. For other languages, the required support is approximately 31 or 32 decimal digits. There are some instances of 38 digit support in both languages and databases. Hence, the maximum requirement in practice appears to be 38 digits, although 31 or 32 digits meets the needs of almost all real applications.

3. Decimal Arithmetic Specification

There has already been work in defining a standard decimal floating-point arithmetic specification [7, 8]. These documents define a set of operations and rules, but do not define a concrete representation of numbers. The present study discusses the choice of representation.

These other decimal arithmetic specifications [7, 8], however, detail many useful aspects such as the operations, formats, and rounding modes that need to be supported to be useful for multiple languages and applications. The core operations needed are Add, Subtract, Plus, Minus, Multiply, Divide, Power, Divide-Integer, Remainder, and Compare. Several exponential formats may need to be provided, including scientific and engineering notation, but these can be supported by a single machine representation. Rounding modes of round-half-up, round-half-even, round-ceiling, round-down, and round-floor are required by one or other of the existing standards. Rounding is applied when the input operands or a result are too long, as set by a maximum precision control.

There are no requirements forced on the machine representation from these decimal arithmetic specifications, other than certain limits and the need to distinguish integer values. However, a decimal floating-point number is assumed to consist of a sign, an integer (the specification assumes no maximum size), and an exponent which indicates a power of ten. The numerical value of a number is described by the following:

$$(-1)^{sign} \times (integer) \times 10^{exponent}$$

The next sections describe the choice of the machine (concrete) representations for each part of the format.

4. Integer Representation

The significand of the decimal floating-point notation has its radix point to right of all digits and is considered to be an integer. The representation format of the integer has a large effect on the implementation. The integer can be chosen to be identical in format to the data in memory, which are typically in BCD format, or it could be a more compact format. If it is more compact then a transformation step is required to move data from memory into the internal format. Other effects that are important are that overflows or rounding precision boundaries are more easily spotted if the format allows easy identification of decimal digit boundaries. Similarly, carries can easily be detected and renormalization of the representation is made easier if decimal digit boundaries are preserved. On the other hand, decimal adders are slightly slower than binary adders and more complex operations are correspondingly slower, too. Therefore, the integer format must be picked carefully.

The three most interesting formats for the integer are: 1) Binary format, 2) Binary Coded Decimal format (BCD, 4 bits per digit), 3) Compressed BCD format.

Binary representation is the most compact form and allows very fast multiplications and divisions. In this format, additions and subtractions which require shifting are slow, and rounding and normalization can be difficult. Johnstone and Petry [4] chose this type of format and they show that there are complexities in decimal base scaling which requires division or multiplications by 10, as well as significand threshold detection. Also, data are typically stored in memory in BCD format and therefore would require conversion into the binary format. Binary format also requires a costly conversion in order to provide input or results as character strings. For these reasons, it is not the best choice.

The second possibility is BCD (4 bits per digit). This format eliminates the cost of the conversion of the integer data when loading an operand from memory. It is also easy to perform rounding and normalization of an intermediate result, and conversions to and from strings. It is slightly slower than the binary representation for unshifted additions and subtractions and is moderately slower for divisions and multiplications. It is also a wasteful notation since it only uses 62.5% of the representation space. This wastage can be a critical factor in determining the format since it determines how many digits can be represented within the length constraints, such as a 64 bit or 128 bit maximum.

The third, Compressed BCD, format is between a BCD representation and a binary representation. To convert between a large binary and a BCD number is time consuming and requires an iterative process of division by powers of ten. Alternative encodings, such as Chen-Ho [9] encoding, can be much faster. This encoding puts 3 decimal BCD digits into 10 binary bits which is significantly more efficient than BCD encod-

ing which would require 12 bits. It is assumed that the internal representation in the hardware would still be BCD notation and there would be a transformation step (requiring only simple boolean logic) to and from the encoded format. Compressed BCD format, therefore, has the advantages of BCD format while encoding more digits into the same space, at the cost of an extra transformation step. From the range of possibilities for compressed BCD formats the most attractive compression ratios appear to be 3 BCD digits to 10 bits and 2 BCD digits to 7 bits, so these two encodings are considered. We have devised an improved Chen-Ho encoding which allows for 7-bit encodings which are a subset of the 10-bit encodings.

Many other encodings have been used in the past, including Bi-quinary, Gray, excess-3, 2-of-5, 1-of-10, and others. These encodings do not offer compelling advantages for modern hardware.

To summarize, there is an advantage in the binary representation for pure processing power but there are significant costs in creating a rounded floating-point decimal result and other conversions. Therefore, a BCD format has a distinct advantage for ease of computation. However, 4 bits per digits is inefficient and so the compressed BCD encoding methods should be considered since they provide more decimal digits with very little extra cost.

5. Exponent Representation

There is a different set of requirements for the representation of the exponent. For the exponent, updates due to normalization or shifting the integer result in simple additions to the exponent. Also, overflows are less timing critical in the exponent calculation than for the integer. Therefore, decimal digit boundaries are less important for the exponent and a binary format is acceptable. Other concerns in representing the exponent are that, in typical binary floating-point units, determining the relative magnitude difference of the exponents is timing critical for floating-point addition. Hence, a format that allows quick comparisons of exponents is preferred.

Three representations are considered for the exponent: 1) Binary twos-complement, 2) Binary unsigned with bias, and 3) Binary Coded Decimal (BCD).

A binary twos-complement exponent without a bias is simple and allows a large range in a given amount of storage. This type of format allows a fast processing speed since binary adders are slightly faster than BCD adders. There is a slight disadvantage that a BCD representation in memory would need to be converted to a binary number but the range of exponents is usually much smaller than the range of the integer.

A binary exponent with a bias has similar advantages to binary twos-complement. This format is similar to binary floating-point exponent representation. One reasons why the IEEE 754 binary floating-point standard [3] chose this format was that it is easy to compare relative magnitudes of unsigned binary numbers. This type of format has the disadvantage that bias offsets have to be applied during arithmetic operations (e.g., Product Exponent = A exponent + B exponent - Bias). This complicates the exponent dataflow slightly.

BCD format allows the simplest conversion from memory and string formats. However, it slightly more difficult and slower for hardware to handle arithmetic computations.

We favor the binary unsigned representation with a bias since it is very similar to, and can even be the same as, the IEEE 754 binary floating-point format. Note that this exponent raises a power of 10 even though it is encoded in binary. Binary floating-point designers have perfected implementations of exponent dataflows with this type of format and therefore it should not present any new difficulties.

6. Special Values

Special values such as infinity and Not-a-Number (NaN) need to be representable in the machine format. There are several ways the format can allow this; the choice depends on the exponent format and the use of bits in the rest of the format. Two solutions are 1) using reserved exponent values, and 2) using separate bits in the format.

Using reserved exponent values is possible since there will always be invalid bit formations of the exponent. If the exponent were represented as a BCD format then invalid bit combinations could be used. Alternatively, if the exponent is in a binary format, it is always possible to reserve exponent values near the end of the range. In particular, if the exponent is limited to a decimal digit range (e.g., -99 to +99) a number of binary values outside this range will available.

Using a separate bit of the format would make it easier to detect a special number but it is inefficient to dedicate a bit of the format for this sole purpose. Therefore, we prefer representing special values by using reserved exponent values.

7. Ordering

The ordering of the sign, exponent, and integer must be determined. We prefer having the sign, then the

Precision (digits)	10	11	12	13	14	...	24	25	26	27	28	29
required Emax	26	28	31	33	36	...	61	63	66	68	71	73
preferred Emax	51	56	61	66	71	...	121	126	131	136	141	146
double Emax	415	455	495	535	575							

Table 2. Requirements for Exponent Range

exponent, and then the integer, similar to the IEEE 754 representations. In software, however, it is common to place the exponent after the sign and integer (for example, 12.3E+3). We prefer the sign and exponent on the left side to give them higher precedence than the integer, to simplify reuse of IEEE 754 circuitry, and also to keep the integer justified to one end to make conversion of memory data easier.

8. Length of Exponent

The exponent representation is assumed to be binary unsigned with a bias and it is also assumed that the two formats have total lengths of 64 bit and 128 bit. A length for the exponent field that is both reasonable and meets the requirements of the IEEE 854 standard must be determined. The IEEE 754 standard devotes 11 bits of the 64 bit format and suggests at least 15 bits for a double extended format. These lengths turn out to be good choices for decimal exponents too. An 11 bit exponent gives a range of -1024 through +1023 which for decimal exponents comfortably represents 3 digits (-999 through +999). Similarly, a 15 bit exponent yields a decimal range of 4 decimal digits (-9999 through +9999) with little waste.

The IEEE 854 standard requires that the exponent range (Emax - Emin) be greater than 5 times the maximum precision in digits, and recommends that it be greater than 10 times the precision. This gives the minimum values of Emax shown in the second and third rows of Table 2. Plausible single precisions are shown to the left of the table, plausible double precisions on the right. The bottom row in the table shows, for each of the plausible single precisions, the recommended minimum Emax for double precision. This must be greater than or equal to 8 times the Emax for single precision, plus 7.

It is apparent from the table that if the latter constraint is satisfied then the preferred Emax for double precision will also be satisfied.

IEEE 854 recommends that, for base 10 representations, the minimum exponent Emin should have the same absolute value as the maximum exponent Emax. That is, Emin = -Emax. Balancing the range in this way minimizes overflows and underflows when the inverse of a number is calculated.

Since the representation comprises an integer and exponent (instead of a fraction and an exponent), the maximum exponent in the representation must be reduced so that the effective exponent range is balanced. For example, if the integer were 13 digits and the exponent 3 digits (-999 through +999) then the range of positive numbers would be from 1E-999 through 9.999999999999E+1011, which is unbalanced.

Instead, the maximum exponent in the representation should be reduced by D-1 (where D is the number of digits in the integer).

For this example the allowed range in the representation should therefore be -999 through +987, leading to a balanced range of numbers with a guaranteed maximum exponent length when converted to character form. That is, positive numbers would range from 1E-999 through 9.999999999999E+999.

Implementations of IEEE 754 use the following splits (precision is fraction bits + 1):

Binary bits			Decimal (approx)	
Total	Fraction	Exponent	precision	Emax
32	23	8	7	E+38
64	52	11	16	E+308
80	64	15	20	E+4931
128	112	15	34	E+4931

9. Proposed Format

The exponent range and format have been determined for the two formats: the exponent is binary unsigned with a bias and is 11 bits for the 64 bit format and 15 bits for the 128 bit format. This leaves 52 bits and 112 bits for the integer. The integer could be BCD or a compressed BCD format. Table 3 shows the number of digits possible for the different lengths and encoding formats. In parentheses is the number of unused bits in the encoding since 7 and 10 do not divide evenly into 52 and 112 bits. These unused bits could be left for future expansion and are probably best placed between the exponent and integer since this leaves the integer right-aligned. The BCD (4 bits per digit) format can only get a precision of 28 digits which is three

	BCD	7:2	10:3	binary
Single (52 bits)	13 digits (0)	14 digits (3)	15 digits (2)	15 digits (2)
Double (112 bits)	28 digits (0)	32 digits (0)	33 digits (2)	33 digits (2)

Table 3. Number of Decimal Digits for Different Integer Encodings

or four digits fewer than are needed by many languages. Therefore, it would appear that a compressed BCD format is needed. Both provide 32 or more digits which is acceptable – but not the 38 digit notation of the most demanding software requirements.

Either of the compression formats are reasonable and we suggest that either one could be used. Note that they are easy to implement, as detailed in the appendix. Since both are easy to implement we slightly favor the one with greater compression, the 10:3 compressed BCD format. This also leaves the same number of bits unused for both lengths.

The proposed format is summarized in Table 4.

$$(-1)^{sign} \times (integer) \times 10^{(exponent\ -\ bias)}$$

Analyzing the exponent representation further, the range of exponents should be -999 to +985 for single precision and -9999 to +9967 for double precision. This creates a range of values for single precision and double precision as shown in Table 5. The bias is chosen to be the same as IEEE 754 double and quad precision with 1023 and 16383. Special values could be assigned outside the exponent range, with infinity perhaps equal to the maximum exponent (2047 or 32767), quiet NaN at the maximum exponent minus 1, and signaling NaN at the maximum exponent minus 2. It is also suggested that the minimum exponent value (0) be reserved as an indication of an uninitialized number.

10. Decimal Floating-Point Unit Commonality

If both the proposed decimal floating-point format were implemented in a DFPU (decimal floating-point unit) and the IEEE 754 standard in a BFPU (binary floating-point unit) on a microprocessor, there is some commonality between these units. The formats are very similar so the input multiplexors could be shared which separate the data into sign, exponent, and significand. The exponent dataflow could reuse some of the adders though this is only a small area compared to the significand. And the significand probably can not use common hardware since the arithmetic is on a BCD notation versus a binary notation. Therefore, it may be best to have separate dataflows for the DFPU versus the BFPU. Though, the register file could be common. It is probable that decimal operations are clustered and the floating-point register file would be unused at this time. Load and Store instructions could be common between the two formats. This is akin to S390 architecture [10] not defining separate loads and stores for binary floating point versus hexadecimal floating point format. Thus, there are some savings possible. More commonality is possible if the significand were in a binary format, but this creates more complexities in scaling and thresholds than it is worth in area savings. As section 4 states it is also beneficial to remain in BCD notation rather than going through conversions on every load and store.

There is also a possibility of commonality of the DFPU with the fixed-point unit in microprocessors which implement decimal arithmetic. The IBM z-Series formerly the S/390 line of computers contains a 64-bit or 16-digit decimal adder on the current z900 microprocessor [11]. This decimal adder is double the size of the previous generation processors due to increased demand in decimal performance.

11. Conclusion

We have presented arguments for a machine representation of decimal floating-point numbers. The proposed formats provide for precisions of up to 33 decimal digits with exponents of up to 4 decimal digits. The two formats are compact and efficient, and occupy either 8 or 16 bytes. We were able to exceed 31 digits of precision by using a BCD compression technique which can be expanded in hardware with a couple of logic levels or in software by simple table lookup.

We propose this format specification in order to encourage microprocessor designers to implement it or a similar format. The proposed format is very similar to the IEEE 754 binary floating-point format which has been implemented in almost all microprocessors. There are still some unspecified details of the format which could be adjusted, such as the exact representation of special values. We hope to work together with other companies to specify a complete and common format.

We also detailed the reasons for the encodings of

Sign (1 bit)	Exponent (11 bit, unsigned binary with bias)	Integer (52 bits : 15 digits (Compressed BCD 10:3))
Sign (1 bit)	Exponent (15 bit, unsigned binary with bias)	Integer (112 bits : 33 digits (Compressed BCD 10:3))

Table 4. Proposed Decimal Floating-Point Formats

	Single Precision	Double Precision
Maximum Negative	-9.99999999999999E+999	-9.99999999999999999999999999999999E+9999
Minimum Negative	-1.00000000000000E-999	-1.00000000000000000000000000000000E-9999
Minimum Positive	+1.00000000000000E-999	+1.00000000000000000000000000000000E-9999
Maximum Positive	+9.99999999999999E+999	+9.99999999999999999999999999999999E+9999

Table 5. Decimal Floating-Point Format Range

each field in the format and also suggest reasonable limits for each format. This is to encourage language standards committees to consider hardware limitations when creating their specifications so they can avoid creating requirements that exceed the likely hardware capabilities.

References

[1] "Programming Language - REXX (X3.274-1996)," The American National Standard, New York, 1996.

[2] "IEEE standard for radix-independent floating-point arithmetic, ANSI/IEEE Std 854-1987," The Institute of Electrical and Electronic Engineers, Inc., New York, 1987.

[3] "IEEE standard for binary floating-point arithmetic, ANSI/IEEE Std 754-1985," The Institute of Electrical and Electronic Engineers, Inc., New York, Aug. 1985.

[4] P. Johnstone and F. Petry. "Higher radix floating point representations," In *Proc. of Ninth Symp. on Comput. Arith.*, pages 128–135, Santa Monica, CA, Sep. 1989.

[5] T. Hull, M. Cohen, and C. Hall. "Specifications for a variable-precision arithmetic coprocessor," In *Proc. of Tenth Symp. on Comput. Arith.*, pages 127–131, Grenoble, France, June 1991.

[6] F. Ris. "A unified decimal floating-point architecture for the support of high-level languages," IBM Research Report RC 6203 (#26651), Sep. 1976.

[7] M. Cowlishaw. "Standard Decimal Arithmetic Specification," http://www2.hursley.ibm.com/ decimal/ decspec.html, Aug. 2000.

[8] M. Cowlishaw. "Standard Decimal Arithmetic Extended Specification," http://www2.hursley.ibm.com/ decimal/ decext.html, Aug. 2000.

[9] T. C. Chen and I. T. Ho. "Storage-Efficient Representation of Decimal Data," *CACM*, 18(1):49–52, Jan. 1975.

[10] "Enterprise Systems Architecture/390 Principles of Operation," Order No. SA22-7201-5, available through IBM branch offices, Sept 1998.

[11] B. Curran et al. "A 1.1GHz first 64b generation z900 microcessor," In *ISSCC Digest of Technical Papers*, pages 238–239, Feb. 2001.

12. Appendix: Chen-Ho Implementation

Traditionally, n decimal digits are represented by 4n bits in computers. However, as Chen and Ho observed, two BCD digits can be compressed optimally and reversibly into 7 bits, and three digits into 10 bits, by a very simple algorithms based on the fixed-length combination of two variable field-length encodings. In over half of the cases the compressed code results from the conventional BCD code by simple removal of redundant 0 bits. A long decimal message can be subdivided into three-digit blocks, each separately compressed; the result differs from the asymptotic minimum length by only 0.34 percent. The hardware requirement is small, and the mappings can be done manually.

Chen-Ho encoding is one of many possible encodings for decimal digits; it encodes three decimal digits in 10 bits with a 0.34% wastage, giving a 20% more efficient encoding than simple BCD (one digit in 4 bits). This compression allows a 33-digit decimal number with a three- or four-digit scale or exponent to be held in a 128-bit representation. We've also explored similar encodings which allow the same encoding for the least significant 7 bits of a 10 bit encoding as a 7 bit encoding.

The specific encoding preserves much of the identity of the three decimal digits, and allows simple processing; it does not require multiplications or divisions to encode or decode to or from BCD.

The following tables fully describe the encoding (compression) and decoding (expansion to BCD); a–l represent the 12 bits of three BCD digits, and p–y represent the 10 bits of the encoded digits.

Compression: (abcd)(efgh)(ijkl) becomes (p)(qrs)(tuv)(wxy)

aei	p	qrs	tuv	wxy
000	0	bcd	fgh	jkl
100	1	00d	fgh	jkl
010	1	01d	bch	jkl
001	1	10d	fgh	bcl
011	1	11d	00h	bcl
101	1	11d	01h	fgl
110	1	11d	10h	jkl
111	1	11d	11h	00l

Expansion: (p)(qrs)(tuv)(wxy) becomes (abcd)(efgh)(ijkl)

pqrtu	abcd	efgh	ijkl
0....	0qrs	0tuv	0wxy
100..	100s	0tuv	0wxy
101..	0tus	100v	0wxy
110..	0wxs	0tuv	100y
11100	0wxs	100v	100y
11101	100s	0wxv	100y
11110	100s	100v	0wxy
11111	100s	100v	100y

(a dot means don't-care)

In hardware, simple boolean operations on input bits define each output bit (for example, during compression: s=d, and p=a+e+i). Full details are in the referenced paper, and also appear as boolean expressions below. The operators are: juxtaposition=AND, '|'=OR, '$\overline{a}$'=NOT.]

Compress 12 bit BCD to 10 bit binary encoding:

$$p = a \mid e \mid i \quad (1)$$
$$q = (b\overline{e}) \mid i \mid (ae) \quad (2)$$
$$r = (c\overline{i}) \mid e \mid (ai) \quad (3)$$
$$s = d \quad (4)$$
$$t = (ae) \mid (f(\overline{a} \mid \overline{i})) \mid (be\overline{i}) \quad (5)$$
$$u = (ai) \mid (ce\overline{i}) \mid (g\overline{e}) \quad (6)$$
$$v = h \quad (7)$$
$$w = j \mid (bi) \mid (fai) \quad (8)$$
$$x = k \mid (ci) \mid (gai) \quad (9)$$
$$y = l \quad (10)$$
$$(11)$$

Expand 10 bit binary encoding to 12 bit BCD:

$$a = (p\overline{q}\overline{r}) \mid (pqr(t \mid u)) \quad (12)$$
$$b = (q\overline{p}) \mid (tp\overline{q}r) \mid (wq(\overline{r} \mid (\overline{t}\overline{u}))) \quad (13)$$
$$c = (r(\overline{p} \mid (\overline{q}u))) \mid (xpq(\overline{r} \mid (\overline{t}\overline{u}))) \quad (14)$$
$$d = s \quad (15)$$
$$e = (pr(\overline{q} \mid \overline{u} \mid t)) \quad (16)$$
$$f = (t(\overline{p} \mid \overline{r})) \mid (pqr\overline{t}uw) \quad (17)$$
$$g = (u(\overline{p} \mid \overline{r})) \mid (pqr\overline{t}ux) \quad (18)$$
$$h = v \quad (19)$$
$$i = (pq(\overline{r} \mid \overline{t} \mid u)) \quad (20)$$
$$j = (w(\overline{p} \mid \overline{q} \mid (rt))) \quad (21)$$
$$k = (x(\overline{p} \mid \overline{q} \mid (rt))) \quad (22)$$
$$l = y \quad (23)$$
$$(24)$$